MCTS

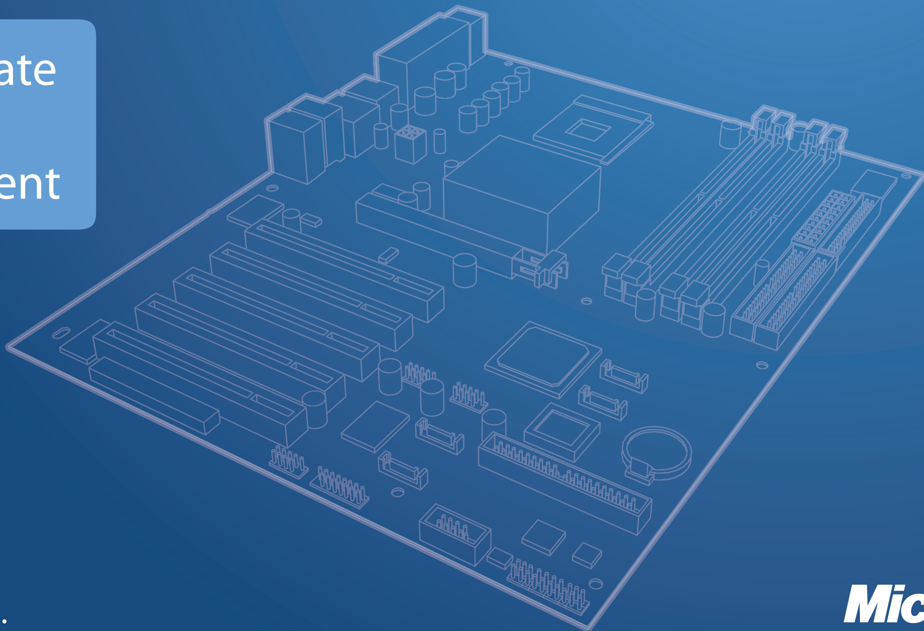
Exam 70-571

Windows® Embedded CE 6.0

Preparation Kit

Certification Exam Preparation

Up to Date with R2 Content



Not for resale.

Microsoft®

Published by

Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT. The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Information in this document, including URL and other Internet Web site references, is subject to change without notice.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation. Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Copyright © 2008 Microsoft Corporation. All rights reserved.

Microsoft, ActiveSync, IntelliSense, Internet Explorer, MSDN, Visual Studio, Win32, Windows, and Windows Mobile are trademarks of the Microsoft group of companies. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

Acquisitions Editor: Sondra Webber, Microsoft Corporation

Authors: Nicolas Besson, Adeneo Corporation
Ray Marcilla, Adeneo Corporation
Rajesh Kakde, Adeneo Corporation

Writing Lead: Warren Lubow, Adeneo Corporation

Technical Reviewer: Brigette Huang, Microsoft Corporation

Editorial Production: Biblioso Corporation

Body Part No. 098-109627

Contents at a Glance

	Foreword	xi
	Introduction	xiii
1	Customizing the Operating System Design	1
2	Building and Deploying a Run-Time Image	37
3	Performing System Programming	81
4	Debugging and Testing the System	145
5	Customizing a Board Support Package	197
6	Developing Device Drivers	241
	Glossary	309
	Index	313
	About the Authors	333

Table of Contents

Foreword ... **xi**

Introduction ... **xiii**
Intended Audience ... xiv
Features of This Book ... xiv
Hardware Requirements ... xv
Software Requirements ... xv
Notational Conventions ... xvi
Keyboard Conventions ... xvi
Notes ... xvii
About the Companion CD-ROM ... xvii
Microsoft Certified Professional Program ... xviii
Technical Support ... xviii

1 Customizing the Operating System Design ... **1**
Before You Begin ... 1
Lesson 1: Creating and Customizing the Operating System Design ... 3
Operating System Design Overview ... 3
Creating an OS Design ... 3
OS Design Customization with Catalog Components ... 5
Build Configuration Management ... 6
OS Design Property Pages ... 6
Advanced OS Design Configurations ... 10
Lesson Summary ... 12
Lesson 2: Configuring Windows Embedded CE Subprojects ... 13
Windows Embedded Subprojects Overview ... 13
Creating and Adding Subprojects to an OS Design ... 14
Configuring a Subproject ... 16
Lesson Summary ... 17
Lesson 3: Cloning Components ... 18
Public Tree Modification and Component Cloning ... 18
Cloning Public Code ... 19
Lesson Summary ... 20
Lesson 4: Managing Catalog Items ... 21
Catalog Files Overview ... 21
Creating and Modifying Catalog Entries ... 22

Catalog Component Dependencies ... 24
Lesson Summary ... 25
Lesson 5: Generating a Software Development Kit ... 26
Software Development Kit Overview ... 26
SDK Generation ... 26
Installing an SDK ... 28
Lesson Summary ... 28
Lab 1: Creating, Configuring, and Building an OS Design ... 29
Chapter Review ... 34
Key Terms ... 34
Suggested Practice ... 34

2 Building and Deploying a Run-Time Image ... 37
Before You Begin ... 38
Lesson 1: Building a Run-Time Image ... 39
Build Process Overview ... 39
Building Run-Time Images in Visual Studio ... 41
Building Run-Time Images from the Command Line ... 46
Windows Embedded CE Run-Time Image Content ... 46
Lesson Summary ... 56
Lesson 2: Editing Build Configuration Files ... 57
Dirs Files ... 57
Sources Files ... 59
Makefile Files ... 61
Lesson Summary ... 62
Lesson 3: Analyzing Build Results ... 63
Understanding Build Reports ... 63
Troubleshooting Build Issues ... 65
Lesson Summary ... 67
Lesson 4: Deploying a Run-Time Image on a Target Platform ... 68
Choosing a Deployment Method ... 68
Attaching to a Device ... 71
Lesson Summary ... 71
Lab 2: Building and Deploying a Run-Time Image ... 72
Build a Run-Time Image for an OS Design ... 72
Configure Connectivity Options ... 73
Change the Emulator Configuration ... 74
Test a Run-Time Image on the Device Emulator ... 75
Chapter Review ... 77
Key Terms ... 78
Suggested Practice ... 78

3 **Performing System Programming** **81**
Before You Begin 81
Lesson 1: Monitoring and Optimizing System Performance 82
Real-Time Performance 82
Real-Time Performance Measurement Tools 84
Lesson Summary 90
Lesson 2: Implementing System Applications 91
System Application Overview 91
Start an Application at Startup 91
Windows Embedded CE Shell 96
Windows Embedded CE Control Panel 97
Enabling Kiosk Mode 101
Lesson Summary 102
Lesson 3: Implementing Threads and Thread Synchronization 103
Processes and Threads 103
Thread Scheduling on Windows Embedded CE 103
Process Management API 104
Thread Management API 104
Thread Synchronization 110
Troubleshooting Thread Synchronization Issues 115
Lesson Summary 117
Lesson 4: Implementing Exception Handling 118
Exception Handling Overview 118
Exception Handler Syntax 120
Termination Handler Syntax 121
Dynamic Memory Allocation 121
Lesson Summary 124
Lesson 5: Implementing Power Management 125
Power Manager Overview 125
Driver Power States 127
System Power States 127
Activity Timers 128
Power Management API 130
Power State Configuration 134
Processor Idle State 135
Lesson Summary 136
Lab 3: Kiosk Mode, Threads, and Power Management 138
Chapter Review 143
Key Terms 143
Suggested Practices 144

4 **Debugging and Testing the System** ... **145**
Before You Begin ... 146
Lesson 1: Detecting Software-Related Errors ... 147
Debugging and Target Device Control ... 147
Kernel Debugger ... 149
Debug Message Service ... 149
Target Control Commands ... 158
Debugger Extension Commands (CEDebugX) ... 159
Advanced Debugger Tools ... 161
Application Verifier Tool ... 162
CeLog Event Tracking and Processing ... 163
Lesson Summary ... 166
Lesson 2: Configuring the Run-Time Image to Enable Debugging ... 168
Enabling the Kernel Debugger ... 168
KITL ... 169
Debugging a Target Device ... 171
Lesson Summary ... 174
Lesson 3: Testing a System by using the CETK ... 176
Windows Embedded CE Test Kit Overview ... 176
Using the CETK ... 178
Creating a Custom CETK Test Solution ... 182
Analyzing CETK Test Results ... 184
Lesson Summary ... 185
Lesson 4: Testing the Boot Loader ... 186
CE Boot Loader Architecture ... 186
Debugging Techniques for Boot Loaders ... 188
Lesson Summary ... 189
Lab 4: System Debugging and Testing based on KITL,
Debug Zones, and CETK Tools ... 190
Chapter Review ... 195
Key Terms ... 196
Suggested Practices ... 196

5 **Customizing a Board Support Package** ... **197**
Before You Begin ... 198
Lesson 1: Adapting and Configuring a Board Support Package ... 199
Board Support Package Overview ... 199
Adapting a Board Support Package ... 201
Cloning a Reference BSP ... 202
Implementing a Boot Loader from Existing Libraries ... 205

Adapting an OAL ... 212
Integrating New Device Drivers ... 217
Modifying Configuration Files ... 218
Lesson Summary ... 218
Lesson 2: Configuring Memory Mapping of a BSP ... 219
System Memory Mapping ... 219
Memory Mapping and the BSP ... 224
Enabling Resource Sharing between Drivers and the OAL ... 226
Lesson Summary ... 227
Lesson 3: Adding Power Management Support to an OAL ... 228
Power State Transitions ... 228
Reducing Power Consumption in Idle Mode ... 229
Powering Off and Suspending the System ... 230
Supporting the Critical Off State ... 232
Lesson Summary ... 233
Lab 5: Adapting a Board Support Package ... 234
Chapter Review ... 238
Key Terms ... 239
Suggested Practices ... 239

6 Developing Device Drivers ... 241
Before You Begin ... 242
Lesson 1: Understanding Device Driver Basics ... 243
Native and Stream Drivers ... 243
Monolithic vs. Layered Driver Architecture ... 244
Lesson Summary ... 246
Lesson 2: Implementing a Stream Interface Driver ... 247
Device Manager ... 247
Driver Naming Conventions ... 248
Stream Interface API ... 250
Device Driver Context ... 252
Building a Device Driver ... 254
Opening and Closing a Stream Driver by Using the File API ... 257
Dynamically Loading a Driver ... 258
Lesson Summary ... 259
Lesson 3: Configuring and Loading a Driver ... 261
Device Driver Load Procedure ... 261
Kernel-Mode and User-Mode Drivers ... 268
Lesson Summary ... 271

Lesson 4: Implementing an Interrupt Mechanism in a Device Driver 272
Interrupt Handling Architecture 272
Interrupt Identifiers (IRQ and SYSINTR) 276
Communication between an ISR and an IST 279
Installable ISRs 280
Lesson Summary 282
Lesson 5: Implementing Power Management for a Device Driver 283
Power Manager Device Drivers Interface 283
Lesson Summary 287
Lesson 6: Marshaling Data across Boundaries 288
Understanding Memory Access 288
Allocating Physical Memory 290
Application Caller Buffers 291
Using Pointer Parameters 292
Using Embedded Pointers 292
Handling Buffers 293
Lesson Summary 296
Lesson 7: Enhancing Driver Portability 297
Accessing Registry Settings in a Driver 297
Interrupt-Related Registry Settings 298
Memory-Related Registry Settings 299
PCI-Related Registry Settings 299
Developing Bus-Agnostic Drivers 300
Lesson Summary 301
Lab 6: Developing Device Drivers 302
Chapter Review 306
Key Terms 307
Suggested Practice 307

Glossary 309

Index 313

About the Authors 333
Nicolas Besson 333
Ray Marcilla 333
Rajesh Kakde 334

Foreword

It seems like yesterday that we released Windows CE 1.0 to the market, although 12 successful years have passed and many things have changed. New technologies have emerged, while others have vanished; and we continue to push forward with our partners to take full advantage of new hardware and software innovations. Windows Embedded CE continues to evolve, yet remains a small-footprint, real-time, embedded operating system that runs on multiple processor architectures and an amazing array of devices, including robots, portable ultrasound imaging systems, industrial controllers, remote sensor and alarm systems, point-of-sale front-ends, media streamers, game consoles, thin clients, and even devices most of us would never associate with a Microsoft operating system. Perhaps one day Windows Embedded CE will run on devices on the moon. It would not come as a surprise. Windows Embedded CE can be everywhere that computer devices help to make life easier and fun.

Right from the start, we have focused on the needs of professional embedded developers by creating a comprehensive suite of development tools and by supporting Windows programming interfaces and frameworks. We have integrated the Windows Embedded CE development tools with Visual Studio 2005 to provide developers with the freedom to customize the operating system and build the applications for the operating system. Today, Windows Embedded CE 6.0 supports x86, ARM, MIPS and SH4 processors out of the box, and includes approximately 700 selectable operating system components. CE provides the tools needed to configure, build, download, debug, and test operating system images and applications, ships with source code for the kernel, device drivers, and other features, and gives application developers the flexibility to create Win32, MFC, or ATL native code applications or managed applications based on the .NET Compact Framework. As part of the Microsoft Shared Source Initiative, we ship more than 2.5 million lines of CE source code, which gives developers the ability to view, modify, rebuild, and release the modified source. And recently we launched a "Spark your Imagination" program to give hobbyist developers access to hardware and CE software development tools at low costs.

You can find plenty of information about the CE operating system, development tools, and concepts in this preparation kit for Microsoft Certified Technology Specialist (MCTS) Exam 70-571 “Microsoft Windows Embedded CE 6.0, Developing” released in May 2008. We are very excited about Exam 70-571. It signifies another important milestone in the Windows Embedded CE success story. Now, for the first

time, embedded developers have the ability to assess and demonstrate their skills regarding the development of embedded solutions based on Windows Embedded technologies, and they can gain recognition for their knowledge and proficiency. Anybody with a passion for CE 6.0 should consider taking the exam. We hope that this book accelerates your preparation just as Windows Embedded CE 6.0 accelerates your development processes. Best wishes from all of us here at the Microsoft development team!

Mike Hall

Windows Embedded Architect
Microsoft Corporation

Introduction

Welcome to the Microsoft Windows Embedded CE 6.0 Exam Preparation Kit. The purpose of this preparation kit is to help Windows Embedded CE developers prepare for the Microsoft Certified Technology Specialist (MCTS) Windows Embedded CE 6.0 Application Development certification exam.

By using this preparation kit, you can maximize your performance on the following exam objectives:

- Customize the operating system design.
- Clone Windows Embedded CE components and manage catalog items.
- Generate a Software Development Kit (SDK).
- Build a run-time image and analyze build results.
- Deploy, monitor, and optimize a run-time image.
- Develop multi-threaded system applications.
- Implement exception handling.
- Support power management in applications, device drivers, and in the OEM adaptation layer (OAL).
- Configure a Board Support Package (BSP), including customizations to boot loader and memory mappings.
- Develop full-featured stream interface drivers.
- Implement Interrupt Service Routines (ISRs) and Interrupt Service Threads (ISTs) and marshal data between kernel-mode and user-mode components.
- Debug kernel-mode and user-mode components to eliminate software-related errors.
- Use the Windows Embedded CE Test Kit (CETK) to perform standard and user-defined tests on a development workstation and on a target device.
- Develop Tux extension components to include custom device drivers in CETK-based tests.

Intended Audience

This Exam Preparation Kit is for system developers with a basic level of knowledge about operating system design, programming system components, and debugging on the Windows Embedded CE platform.

Specifically, this Preparation Kit is designed for readers with the following skills:

- Basic knowledge of Windows and Windows Embedded CE development and development
- At least two years of experience with C/C++ programming and the Win32 Application Programming Interface (API).
- Familiarity with Microsoft Visual Studio 2005 and Platform Builder for Windows Embedded CE 6.0.
- Basic debugging skills using standard Windows debugging tools.

MORE INFO Audience profile for Exam 70-571

For information about prerequisites to pass the certification exam, see the Audience Profile section in the Preparation Guide for Exam 70-571 at *http://www.microsoft.com/learning/exams/70-571.mspx*.

Features of This Book

Each chapter opens with a list of exam objectives covered in the chapter and a "Before You Begin" section, which prepares you for completing the chapter. The chapters are then divided into lessons. Each lesson begins with a list of objectives and states an estimated lesson time. The lesson content is subdivided further according to topics and lesson objectives.

Each chapter ends with hands-on procedures and a short summary of all chapter lessons. This is followed by a brief check of key terms and suggested practices which test your knowledge of the chapter material and help you successfully master the exam objectives presented in the chapter.

▶ The hands-on examples give you an opportunity to demonstrate a particular concept or skill and test what you have learned in the chapter lessons. All hands-on examples include step-by-step procedures that are identified with a bullet symbol like the one to the left of this paragraph. To help you successfully master the presented procedures,

worksheets with detailed step-by-step instructions for each lab are also included in the companion material for this book.

To complete the hands-on procedures, you must have a development computer with Microsoft Windows XP or Microsoft Windows Vista, Visual Studio 2005 Service Pack 1, and Platform Builder for Windows Embedded CE 6.0 installed.

Hardware Requirements

The development computer must have the following minimum configuration, with all hardware on the Windows XP or Windows Vista Hardware Compatibility List:

- 1 GHz 32-bit (x86) or 64-bit (x64) processor or faster.
- 1 gigabyte (GB) of RAM.
- 40 GB hard drive with at least 20 GB of available disk space for Visual Studio 2005 and Platform Builder.
- DVD-ROM drive.
- Microsoft Mouse or compatible pointing device.
- Paging file set to twice the amount of RAM or larger.
- VGA-compatible display.

Software Requirements

The following software is required to complete the procedures in this course:

- Microsoft Windows XP SP2 or 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.

> **NOTE Trial versions of Visual Studio 2005 and Windows Embedded CE 6.0**
> Installation guidelines and evaluation versions of Visual Studio 2005 and Windows Embedded CE 6.0 are available on the Microsoft Website, at *http://www.microsoft.com/windows/embedded/products/windowsce/getting-started.mspx*.

Notational Conventions

- Characters or commands that you type appear in **bold lowercase** type.
- <Angle brackets> in syntax statements indicate placeholders for variable information.
- *Italic* is used for book titles and Web addresses.
- Names of files and folders appear in Title Caps, except when you are to type them directly. Unless otherwise indicated, you can use all lowercase letters when you type a file name in a dialog box or at a command prompt.
- File name extensions appear in all lowercase.
- Acronyms appear in all uppercase.
- Monospace type represents code samples, examples of screen text, or entries that you might type at a command prompt or in initialization files.
- Square brackets [ ] are used in syntax statements to enclose optional items. For example, [*filename*] in command syntax indicates that you can choose to type a file name with the command. Type only the information within the brackets, not the brackets themselves.
- Braces { } are used in syntax statements to enclose required items. Type only the information within the braces, not the braces themselves.

Keyboard Conventions

- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press ALT+TAB" means that you hold down ALT while you press TAB.
- A comma (,) between two or more key names means that you must press each of the keys consecutively, not together. For example, "Press ALT, F, X" means that you press and release each key in sequence. "Press ALT+W, L" means that you first press ALT and W together, and then release them and press L.
- You can choose menu commands with the keyboard. Press the ALT key to activate the menu bar, and then sequentially press the keys that correspond to the highlighted or underlined letter of the menu name and the command name. For some commands, you can also press a key combination listed in the menu.

- You can select or clear check boxes or option buttons in dialog boxes with the keyboard. Press the ALT key, and then press the key that corresponds to the underlined letter of the option name. Or you can press TAB until the option is highlighted, and then press the spacebar to select or clear the check box or option button.
- You can cancel the display of a dialog box by pressing the ESC key.

Notes

Several types of Notes appear throughout the lessons.

- Notes marked **Tip** contain explanations of possible results or alternative methods.
- Notes marked **Important** contain information that is essential to completing a task.
- Notes marked **Note** contain supplemental information.
- Notes marked **Caution** contain warnings about possible loss of data.
- Notes marked **Exam Tip** contain helpful hints about exam specifics and objectives.

About the Companion CD-ROM

The Companion CD contains a variety of informational aids that may be used throughout this book. This includes worksheets with detailed step-by-step instructions and source code used in hands-on exercises, as well as complimentary technical information and articles from the Microsoft developers.

An electronic version (eBook) of this book is included with a variety of viewing options available. The Companion CD also contains a complete set of post-press files for this official self-paced study guide to produce a printed book. The post-press files are in Portable Document Format (PDF) and have the required crop marks for professional printing and binding.

Microsoft Certified Professional Program

The Microsoft Certified Professional (MCP) program provides the best method to prove your command of current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies. Computer professionals who become Microsoft certified are recognized as experts and are sought after industry-wide. Certification brings a variety of benefits to the individual, employers, and organizations.

MORE INFO All the Microsoft certifications

For a full list of Microsoft certifications, go to *http://www.microsoft.com/learning/mcp/default.asp*.

Technical Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. If you have comments, questions, or ideas regarding Windows Embedded CE development, contact a Windows Embedded CE specialist through Microsoft Product Support Services (PSS), Microsoft Developer Network (MSDN), or the following blog sites:

- **Nicolas BESSON's Weblog** Contact the principal author of the Windows Embedded CE 6.0 Exam Preparation Kit with feedback and subject suggestions for new articles related to those subjects at *http://nicolasbesson.blogspot.com*.
- **Windows Embedded Blog** Read about Mike Halls tricks, tips, and random thoughts on Windows Embedded at *http://blogs.msdn.com/mikehall/default.aspx*.
- **Windows CE Base Team Blog** Get background information about Windows Embedded CE kernel and storage technologies and system tools directly from the Microsoft developers at *http://blogs.msdn.com/ce_base/default.aspx*.

MORE INFO Windows Embedded CE product support

For detailed information about all available Windows Embedded CE product support options, go to *http://www.microsoft.com/windows/embedded/support/products/default.mspx*.

Chapter 1

Customizing the Operating System Design

Whenever you want to deploy Windows® Embedded CE 6.0 R2 on a target device, you must use a run-time image that includes the necessary operating system (OS) components, features, drivers, and configuration settings. The run-time image is the binary representation of the OS design. You can use Microsoft® Platform Builder for Windows Embedded CE 6.0 to create or customize an OS design and generate the corresponding run-time image. For each OS design, you typically create a new development project in Microsoft® Visual Studio® 2005 and include only the necessary components for your target device and applications. This helps to reduce the footprint of the operating system and to lower hardware requirements. However, in order to generate compact and functional run-time images, you must have an intimate understanding of Platform Builder, including the user interface (UI), the catalog components, and the specifics of the build procedure. This chapter covers these aspects by explaining how to create an OS design and generate a new Windows Embedded CE run-time image.

Exam objectives in this chapter:

- Creating and customizing OS designs
- Configuring Windows Embedded CE subprojects
- Cloning components
- Managing catalog items
- Generating a Software Development Kit (SDK)

Before You Begin

To complete the lessons in this chapter, you must have the following:

- At least some basic knowledge about Windows Embedded CE software development.

- A basic understanding of the directory structure and build process of Platform Builder for Windows Embedded CE 6.0 R2.
- Familiarity creating binary Windows Embedded CE run-time images and downloading run-time images to target devices.
- Experience using an SDK to develop applications for Windows Embedded CE.
- A development computer with Microsoft Visual Studio 2005 Service Pack 1 and Platform Builder for Windows Embedded CE 6.0 installed.

Lesson 1: Creating and Customizing the Operating System Design

You can use Platform Builder in Visual Studio 2005 to create an OS design with as many or as few of the features available in Windows Embedded CE 6.0 R2 as you find necessary for your specific purpose. For example, you can create an OS design for a particular target device, such as a portable multimedia device, and another OS design for a remotely programmable wireless-enabled digital thermostat. These two target devices might rely on the same hardware, but the purposes of the devices are different and so are the corresponding OS design requirements.

After this lesson, you will be able to:

- Understand the role and specifics of an OS design.
- Create, customize, and use OS designs.

Estimated lesson time: 30 minutes.

Operating System Design Overview

The OS design defines the components and features contained in a run-time image. Essentially, it corresponds to a Visual Studio with Platform Builder for Windows Embedded CE 6.0 R2 project. The OS design can contain any or all of the following elements:

- Catalog items, including software components and drivers
- Additional software components in the form of subprojects
- Custom registry settings
- Build options, such as for localization or debugging based on Kernel Independent Transport Layer (KITL)

Additionally, every OS design contains a reference to at least one Board Support Package (BSP) with device drivers, hardware-specific utilities, and an OEM adaptation layer (OAL).

Creating an OS Design

Windows Embedded CE includes an OS Design Wizard, which, as the name suggests, provides a convenient way to create OS designs. To launch it, start Visual Studio 2005 with Platform Builder for Windows Embedded CE 6.0 R2, open the File menu, then

point to New, and then click Project to display the New Project dialog box. In this dialog box, under Project Types, select Platform Builder for CE 6.0; and under Visual Studio Installed Templates, select OS Design, enter a name for the OS design in the Name field, and then click OK to start the Windows Embedded CE 6.0 OS Design Wizard.

The OS Design Wizard enables you to select a BSP and a design template with commonly used options and preselected catalog components. Any settings that you specify within the wizard you can also modify later, so don't worry about the individual settings too much for now. Depending on the template that you select on the Design Templates page, the OS Design Wizard might display an additional Design Template Variants page with more specific options related to the selected template. For example, Windows Thin Client, Enterprise Terminal, and Windows Network Projector are all devices that use the Remote Desktop Protocol (RDP) and are therefore variants of the same Thin Client design template. Depending on the selected template and variant, the OS Design Wizard might display additional pages to include specific components in the OS design, such as ActiveSync®, WMV/MPEG-4 video codec, or IPv6.

The OS Design Template

A CE 6.0 OS design template is a subset of the catalog components required to use Windows Embedded CE for a particular purpose. It is not necessary to start from a template when creating a new OS design, although it can save a significant amount of time to do so. It is straightforward to change catalog components later by selecting them in the Catalog Items View.

Choosing an appropriate template can save you development time and effort. For example, you might have to demonstrate the features of a new development board at a trade show. In this case, it is a good idea to start with the PDA Device or Consumer Media Device design template and add the required components and common Windows applications in the OS Design Wizard, such as the .NET Compact Framework 2.0, Internet Explorer®, and WordPad. On the other hand, if you are developing a driver for a Controller Area Network (CAN) controller, it might be better to start with the Small Footprint Device design template and only add what's absolutely necessary to minimize the size of the run-time image and to keep startup times at a minimum.

The OS Design Wizard is flexible and supports custom design templates. Template files are Extensible Markup Language (XML) documents, located in the %_WINCEROOT%\Public\CEBase\Catalog folder. You can start with a copy of an existing Platform Builder Catalog XML (PBCXML) file and modify the PBCXML structures according to your specific needs. Platform Builder automatically enumer-

ates all .pbcxml files in the Catalog folder when you start Visual Studio or refresh the Catalog Items View in Visual Studio.

OS Design Customization with Catalog Components

After completing the OS Design Wizard, it is straightforward to customize the OS design. The catalog is a repository for all the components that can be added to an OS design. It is accessible directly from within the integrated development environment (IDE). Click Catalog Items View in the Solution Explorer window pane. Almost every CE feature is divided into separate user-selectable catalog components, from ActiveSync to TCP/IP. You can select these components directly in the UI. Each catalog item is a reference to all the components necessary to build and integrate a feature into the run-time image.

When you add a catalog item that depends on other catalog items, you implicitly add these items as dependencies to the OS design as well. The Catalog Items View shows these items with a green square in the check box to indicate that they are part of the OS design due to existing dependencies. In contrast, the Catalog Items View shows manually selected items and items included based on a design template with a green check mark.

In the Catalog Items View, you can show all catalog components or enable a filter to display only selected catalog items. Click the downward arrow on the Filter button in the top left corner of the Catalog Items View in Solution Explorer to apply a filter or select the option All Catalog Items in the catalog to display the complete list of catalog items.

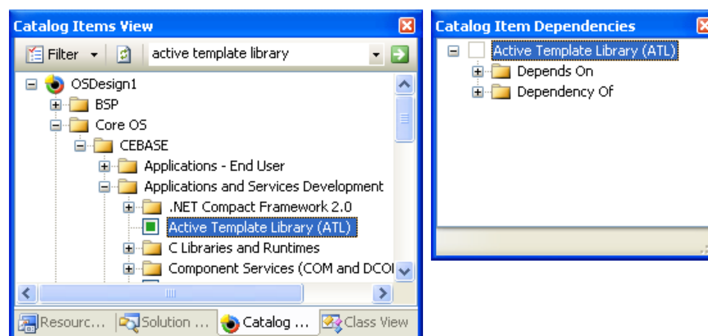


Figure 1-1 Catalog Items View with the search box and Catalog Item Dependencies window

Provided that you know the name of the catalog item or the SYSGEN variable a component sets, you might find it more convenient and faster to search for the desired

catalog item that you want to add or remove than to look for it manually. To search by item name or SYSGEN variable, type the search term into the text box at the top of the Catalog Items View and click the green arrow next to it.

To analyze the dependencies of a catalog item, you can right-click the item and select Show Dependencies to display the Catalog Item Dependencies window, as illustrated in Figure 1-1. For example, you can use this feature to see the reason for the inclusion of a specific catalog item as a dependency. In CE 6.0 R2, Platform Builder dynamically traverses the catalog to enumerate all components that depend on the selected item as well as all components that this item depends on.

Build Configuration Management

Windows Embedded CE supports multiple build configurations that you can modify separately. The two standard configurations are Release and Debug. These build configurations are automatically available when you create an OS design. In the Debug build configuration, the compiler generates debug information, maintains links to the source code in program database (.pdb) files, and, to facilitate debugging and step-by-step code execution, does not optimize the code. Windows Embedded CE run-time images compiled in Debug build configuration are generally 50 percent to 100 percent larger than images compiled by using the Release configuration. To choose a build configuration, open the Build menu in Visual Studio, click Configuration Manager, and then, in the Configuration Manager dialog box, select the desired build configuration under Active Solution Configuration. You can also select the desired build configuration by using the pull-down menu in the Standard toolbar.

OS Design Property Pages

For each build configuration, it is possible to configure a number of project properties, such as the locale, whether or not to include KITL, custom build actions, inclusion of subprojects in the binary image, and custom SYSGEN variables. To access these options, display the Property Pages dialog box by right-clicking the OS design node in Solution Explorer and selecting Properties. The OS design node is the first child object under the Solution top-level node. The caption corresponds to the project name, such as OSDesign1. If Solution Explorer is not visible, open the View menu and click Solution Explorer, and if Solution Explorer currently displays the Catalog Items View or the Class View, click the Solution Explorer tab to display the solution tree.

TIP Setting properties for multiple configurations

In the top left corner of the Property Pages dialog box, you can find a list box to select the build configuration. Among other options, you can select All Configurations or Multiple Configurations. These options are useful if you want to set properties for multiple build configurations at the same time.

Locale Options

In the Property Pages dialog box, under Configuration Properties, you can find the Locale node, which enables you to configure language settings for the Windows Embedded CE image, as illustrated in Figure 1-2. For most languages, the Locale property page covers all requirements to localize the OS design, but some languages, particularly East Asian languages such as Japanese, require additional catalog components. It is also important to note that some catalog components related to internationalization significantly increase the size of the run-time image.

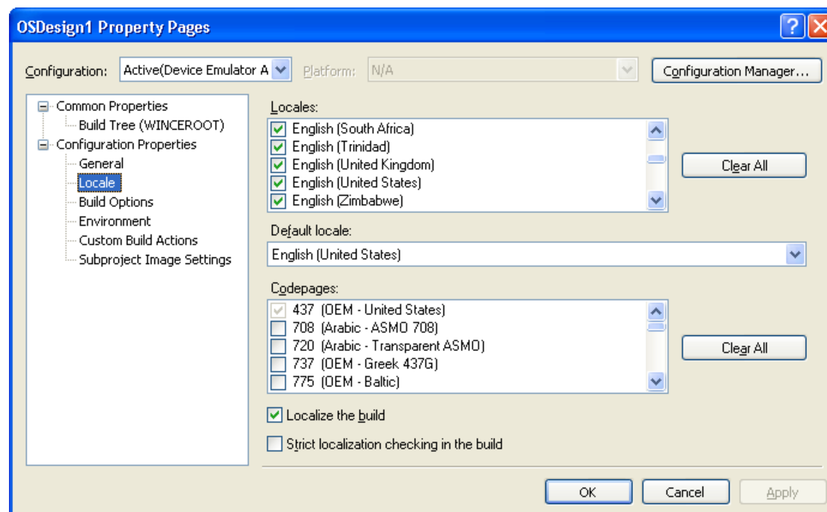


Figure 1-2 Locale property page

The Locale property page enables you to configure the following options for the run-time image:

- **Locales** Selects the languages that will be available to localize the run-time image. If a selected language has a default ANSI and OEM code page, the code page is automatically added to the OS design, as indicated by a marked corresponding code page entry in the Codepages list.

- **Default Locale** Defines the default locale for the OS design. The default language is English (United States), which uses the default code page 437 (OEM-United States).
- **Code Pages** Specifies the ANSI and OEM code pages that will be available in the OS design.
- **Localize The Build** Instructs the build process to use localized string and image resources. Platform Builder performs the localization of the OS design during the make image step of the OS design build process. Localized resource files are integrated inside the binary files for the common components, through res2exe.
- **Strict Localization Checking In The Build** Causes the build process to fail if localization resources are missing, rather than just using the resources based on the default locale.

Build Options

Directly under the Locale node in the Property Pages dialog box, you can find the Build Options node, which enables you to control event tracking, debugging, and other build options for the active OS design, as illustrated in Figure 1-3.

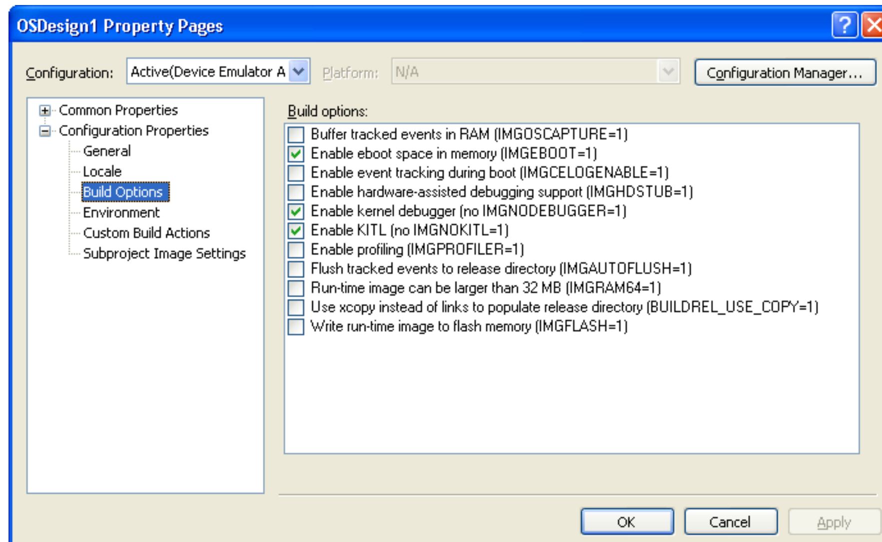


Figure 1-3 Build Options property page

The Build Options property page enables you to configure the following options for the run-time image:

- **Buffer Tracked Events In RAM** Causes Platform Builder to include OSCapture.exe in the CE image. Also enables logging of operating system events tracked by OSCapture.exe in RAM so they can be flushed to a file and viewed later.
- **Enable Eboot Space In Memory** Enables the Ethernet boot loader (EBOOT) to pass data to the Windows Embedded CE OS at start time.
- **Enable Event Tracking During Boot** Enables CE event log data collection much earlier during the start process than it would normally be collected otherwise. If you activate this option, event tracking starts before most of the kernel and file system initialization is complete.
- **Enable Hardware-Assisted Debugging Support** This is required for some third-party hardware debugging tools (JTAG probes compliant with exdi2).
- **Enable Kernel Debugger** Enables the Windows Embedded CE debugger so you can step through the code in the run-time image. Kernel debugging requires KITL to communicate with Platform Builder at run time.
- **Enable KITL** Adds KITL to the run-time image. KITL is a useful debugging feature that enables developers to use the kernel debugger, interact with the remote device's file system, registry, and other components, as well as run code. You should not include KITL in the final build of the operating system, because it introduces overhead and wastes time during the start process trying to connect to a host computer.
- **Enable Profiling** Enables the kernel profiler in the run-time image, which you can use to collect and view timing and performance data. The kernel profiler is a useful tool for optimizing the performance of Windows Embedded CE on a target device.
- **Flush Tracked Events To Release Directory** Adds CeLogFlush.exe to the run-time image, which automatically flushes log data collected by OSCapture.exe to the Celog.clg file in the release directory on the development computer.
- **Run-Time Image Can Be Larger Than 32 MB** Enables you to build a larger-than-32-MB image. However, you should not use this option if you want to build an image larger than 64 MB. In this case, you must set an environment variable for the appropriate size (such as IMGRAM128).

- **Use Xcopy Instead Of Links To Populate Release Directory** Creates actual copies of the files by using xcopy rather than copylink. Copylink might only create hard links to the files rather than copying them, and it requires the NTFS file system on the development computer.
- **Write Run-Time Image To Flash Memory** Instructs EBOOT to write the run-time image to the flash memory of the target device.

Environment Options

The Property Pages dialog box provides an Environment option to configure environment variables that will be used during the build process. You can enable most features in Windows Embedded CE 6.0 R2 by using catalog components, but for some options you need to set a SYSGEN variable so that Platform Builder compiles the necessary code and includes it in the run-time image. Setting environment variables that influence the build process can be helpful when developing a BSP. Environment variables are accessible during the Windows Embedded CE build process from the command line. You can also use environment variables to specify flexible information in the sources, binary image builder (.bib), and registry (.reg) files.

> **TIP If it works in Debug but not in Release**
>
> If you can build a run-time image in the Debug configuration, but not in the Release configuration, display the Property Pages dialog box, select All Configurations from the Configuration list box, and then select the Environment option to set the environment variables for both Debug and Release to the same values.

Advanced OS Design Configurations

This section covers several advanced topics related to OS designs. Specifically, this section explains how to support multiple platforms with the same OS design and discusses the file locations and file types that an OS design typically includes.

Associating an OS Design with Multiple Platforms

When creating a new OS design project by using the OS Design Wizard, you can select one or more BSPs on the Board Support Packages wizard page. By associating an OS design with multiple BSPs, you can generate separate run-time images with identical content for multiple platforms. This is particularly useful in projects that include multiple development teams, especially if the final target hardware is currently not available. For example, you can generate a run-time image for an emulator-

based platform so that the application development team can start before the final hardware is available. In terms of OS functionality, the application development team can use the application programming interfaces (APIs) before the final target platform is available. The APIs will be included in the final target because the two run-time images share the same set of components and configuration settings.

You can also add support for multiple platforms to an OS design after the initial creation. All you need to do is select the corresponding check boxes under BSP in the Catalog Items View of Solution Explorer. Selecting a BSP automatically adds the additional platform to the configuration for Release and Debug. You can then switch between the different platforms and build configurations by using Configuration Manager, which is available on the Build menu in Visual Studio. However, it is necessary to run the entire build process, including the time-consuming SYSGEN phase, for each platform individually.

OS Design Paths and Files

In order to use and redistribute your OS designs, you need to know exactly what files constitute an OS design and where they are located on your development computer. By default, you can find the OS designs in the %_WINCEROOT%\OSDesigns directory. Each project corresponds to a separate child directory. OS designs typically correspond to the following file and directory structure:

- ***<Solution Name>*** The parent directory that Visual Studio created for the project.
  - ***<Solution Name>*.sln** The Visual Studio solution (.sln) file to store settings specific to the OS design project. The file name is generally the same as that of your OS design.
  - ***<Solution Name>*.suo** The Visual Studio solution user options (.suo) file, which contains user-dependent information, such as the state of the Solution Explorer views. The file name is generally the same as your OS design.
  - ***<OS Design Name>*** The parent directory for the remaining files included in the OS design project.
    - ***<OS Design Name>*.pbxml** Your OS design's catalog file. This file contains references to selected catalog components and all the settings related to your OS design.
    - **Subprojects** This directory includes a separate subfolder for each subproject created as part of the OS design.
    - **SDKs** This directory includes the Software Development Kits (SDKs) created for the OS design.

- **Reldir** This is the release directory. Platform Builder copies the files into this directory during the process of creating the run-time image that can then be downloaded to a target device.
- **WinCE600** This is where files are copied after the Sysgen phase is complete, including resource files and configuration files for the current OS design.

Source Control Software Considerations

Basically, an OS design is a set of configuration files for Platform Builder to generate a Windows Embedded CE run-time image. If you are using source control software to coordinate the work of your development team, you only have to store these configuration files in your source control repository. You do not need to include any files from the CESysgen folder (used during the build process of the run-time image) or Reldir directories, because they can be reconstituted on any workstation with Platform Builder and the BSP installed. Also, omit files ending in .user or .suo because those are user-specific settings for the IDE, and omit .ncb files because these files only contain IntelliSense® data.

Lesson Summary

Platform Builder for Windows Embedded CE 6.0 R2 includes an OS Design Wizard to help you accomplish the basic OS design creation steps quickly and conveniently. You can select one or multiple BSPs to include all hardware-specific device drivers and utilities for your target platform and a design template with possible template variants to include additional catalog items. After completing the OS Design Wizard, you can further customize the OS design. You can exclude unnecessary catalog items, include additional components, and configure project properties such as the Debug and Release build options. In the Debug build configuration, Platform Builder includes debug information in the run-time image, which leads to an increase of 50 percent to 100 percent in comparison to Release builds. However, Debug builds facilitate debugging and step-by-step code execution during the development process. Because you can configure Debug and Release build options separately, you might encounter a situation in which your OS design compiles in the Debug configuration but not in the Release configuration. In this case, it can be helpful to set environment variables in both Debug and Release to the same values. In order to distribute your OS designs, you need to locate the source files, which you can find by default in the %_WINCEROOT%\OSDesigns directory. You can use source control software to coordinate the work of a development team.

Lesson 2: Configuring Windows Embedded CE Subprojects

A subproject is a Visual Studio project inserted into a parent project to include relatively independent components in an overall solution. In our case, the parent project typically corresponds to an OS design. Subprojects can take the following forms:

- An application (managed or unmanaged).
- A dynamic-link library (DLL).
- A static library.
- An empty project containing only configuration settings.

Subprojects are a good way to include a particular application, device driver, or other code module in an OS design and to maintain this code and the OS design together as one solution.

After this lesson, you will be able to:

- Create and configure subprojects.
- Build and use subprojects.

Estimated lesson time: 20 minutes.

Windows Embedded Subprojects Overview

Platform Builder for Windows Embedded CE enables you to create subprojects as part of an OS design. Because subprojects are both modular and easily redistributable, they provide a convenient way to add applications, drivers, or other files to your OS design without manually including them in the build tree as part of the BSP. You can also create subprojects for internal test applications and development tools to make it quick and easy to build these tools and run them on a test device.

Types of Subprojects

Windows Embedded CE supports the following subproject types:

- **Applications** Win32® applications with a graphical user interface (GUI), programmed in C or C++.
- **Console applications** Win32 applications without a GUI, programmed in C or C++.

- **Dynamic-link library (DLL)** Drivers or other code libraries, loaded and used at run time.
- **Static library** Code modules in the form of library (.lib) files that you can link to from other subprojects or export as part of the OS design's SDK.
- **TUX dynamic-link library** Windows Embedded CE custom test components for the Microsoft Windows CE Test Kit (CETK), as explained in more detail in Chapter 4.

Creating and Adding Subprojects to an OS Design

It is straightforward to create a new subproject or add an existing project as a subproject to an OS design. For the most part, you can use the Windows Embedded CE Subproject Wizard to accomplish this task, which you can start by right-clicking the Subprojects folder in Solution Explorer and clicking Add New Subproject or Add Existing Subproject. However, an understanding of the details, including the purpose of the various subproject types, the files and settings created by the CE Subproject Wizard, the build process, and customization possibilities for subprojects, is still helpful.

The CE Subproject Wizard creates a subfolder in the OS design folder, which contains all the required configuration files, including:

- **<*Name*>.pbpxml** An XML-based file that contains metadata information about the subproject. This file references the .bib, .reg, sources, and dirs files to build the subproject.
- **<*Name*>.bib** A binary image builder (.bib) file used during the makeimg step in the build process to dictate files to include in the binary image.
- **<*Name*>.reg** A registry file with settings to be included in the final run-time image.
- **Sources** A Windows Embedded CE sources file. This is a makefile that contains build options to control the Windows Embedded CE build process.
- **Makefile** A file used in association with the sources file in the Windows Embedded CE build process.

To make a copy of a subproject for later use, open your OSDesigns folder (%_WINCEROOT%\OSDesigns), and then open the solution folder for your OS design. The solution folder typically contains the <*OS Design Name*>.sln file and a folder named according to the OS design. Within this folder, you can find the definition file of the OS design <*OS Design Name*>.pbxml and several subdirectories. One of these sub-

directories should be your Subproject folder, as illustrated in Figure 1-4. It is a good idea to back up this folder. You can then add it to any OS design later by right-clicking the Subprojects container in Solution Explorer and selecting Add Existing Subproject.

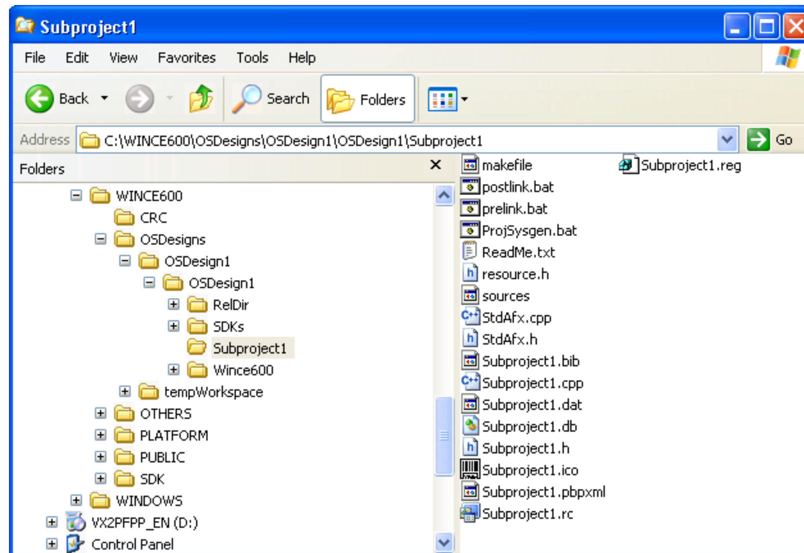


Figure 1-4 A subproject folder in an OS design project

Creating Windows Embedded CE Applications and DLLs

To add a Windows Embedded CE application or DLL to an OS design, use the CE Subproject Wizard to create the corresponding subproject. Although you can start with an empty subproject, it is generally more convenient to select a simple console or GUI application template, adding your own code afterward as necessary.

Creating Static Libraries

The CE Subproject Wizard also provides you with an option to create a static library, which you can then link to another subproject or export as part of an SDK. This is helpful for dividing up more sophisticated subprojects or providing more options to application developers who develop solutions for your hardware and firmware. If other subprojects in your OS design rely on a static library, you might have to adjust the build order of the subprojects to use the library efficiently. For example, if a Windows Embedded CE application uses your static library, you should build the library first so that the application build process uses the updated library.

Creating a Subproject to Add Files or Environment Variables to a Run-Time Image

Some subprojects do not necessarily include source code. For example, you can create an empty subproject by using the CE Subproject Wizard, modify the sources file, and set **TARGETTYPE=NOTARGET** to indicate you do not want to generate binary target files. You can then add files to the run-time image by adding corresponding references to the subproject's .bib file. You can also add registry settings to the subproject's .reg file and you can add SYSGEN variables by editing the subproject's Projsysgen.bat file. Although it is generally faster and more convenient to modify the .reg and .bib files and project properties of the OS design directly, creating a subproject for this purpose can be advantageous if you are planning to reuse customizations in multiple OS designs in the future.

Configuring a Subproject

Visual Studio provides a number of options in the project properties that you can configure to customize the build process for subprojects. To configure these settings, display the Property Pages dialog box for your OS design, as explained earlier in this chapter. You can then find the subproject properties under Subproject Image Settings. For each subproject added or created in the current OS design, you can configure the following parameters:

- **Exclude From Build** Activating this option excludes the subproject from the build process of the OS design, meaning the build engine does not process the source files that belong to the selected subproject.
- **Exclude From Image** Sometimes it can be time-consuming to deploy a run-time image when subprojects change. You have to disconnect from the target platform, rebuild the project, create a new image, reconnect to the target platform, and download the updated image every time a change is made with a subproject. To save time and effort when working on a subproject, you should exclude it from the run-time image by using the Exclude From Image option. In this case, you should also create a way to update the file on the device at run time through KITL, ActiveSync, or any other way you can transfer it to the device.
- **Always Build And Link As Debug** By using this option, you build the subproject in Debug build configuration while your current OS design build process uses the Release configuration. In this way, you can debug the subproject code by using the Kernel Debugger while the operating system is running in the Release version (this option will not automatically enable the Kernel Debugger).

NOTE Exclusion from the run-time image

When you exclude a subproject from the run-time image, you implicitly exclude the subproject's files from the Nk.bin file that is downloaded to the target device. Instead, Windows Embedded CE accesses the subproject's files on an as-needed basis directly from the Release directory over KITL (when KITL is enabled). This means that you can modify the code in a driver or application subproject without having to redeploy the run-time image. You should only need to verify that the remote device is not currently running the code, and then you can rebuild the code and run it again.

Lesson Summary

You can use Windows Embedded CE subprojects to add applications, drivers, DLLs, and static libraries to an OS design, which is useful if you want to manage a complex Windows Embedded CE development project that includes a large number of applications and components. For example, you can include a custom shell application or a device driver for a USB peripheral in the form of a subproject to an OS design, and then have different development teams implement these components. You can also use Windows Embedded CE subprojects to add registry settings, environment variables, or specific files to various OS designs, such as the run-time files for the Core Connectivity (CoreCon) interfaces or a test application. It is possible to back up subprojects individually and add them as existing subprojects to future OS designs.

Lesson 3: Cloning Components

Platform Builder for Windows Embedded CE 6.0 R2 comes with public source code that you can reuse and adapt for various purposes. You can analyze and modify the source code for most of the components included in Windows Embedded CE, from the shell to the serial driver's model device driver (MDD) layer. However, you must not modify the public source code directly. Instead, create a functional copy of the public code so that you can modify the desired components without affecting the original Windows Embedded CE 6.0 R2 code base.

After this lesson, you will be able to:

- Identify components to clone.
- Clone an existing component.

Estimated lesson time: 15 minutes.

Public Tree Modification and Component Cloning

Once you have discovered that the code you want to modify resides in the %_WINCEROOT%\Public folder, you might be tempted to modify this code in place and then build it without moving it to another folder first. However, there are a number of reasons not to modify the Public tree:

- You have to back up the Public directory and maintain separate directory versions for each of your OS design projects, such as WINCE600\PUBLIC_Company1, WINCE600\PUBLIC_Company2, and WINCE600\PUBLIC_Backup.
- Windows Embedded CE updates, patches provided by quick fix engineering (QFE), and service packs might overwrite or be incompatible with your modifications.
- Redistributing your code is difficult and error-prone.
- Worst of all, when you change code in the Public directory tree, you have to spend up to three hours building the operating system. If you already know the CE build process so well that you can rebuild just your particular code without having to rebuild the entire Public folder, you will also already know enough to clone the components.

CAUTION Public code modifications

Never modify the contents of the Public folder tree.

At a first glance, component cloning might seem like a lot of trouble, but it will save you development time and effort in the long run.

Cloning Public Code

Platform Builder supports instant cloning for some Windows Embedded CE components. To clone these components, right-click the catalog item in the Catalog Items View of Solution Explorer and select Clone Catalog Item. Platform Builder will automatically create a subproject for the component you selected in your OS design with a copy of the code. Before using any other method, such as the Sysgen Capture tool, you should check to see if the desired catalog component supports the Clone Catalog Item option. If it does, then you are two mouse-clicks from completion, as illustrated in Figure 1-5.

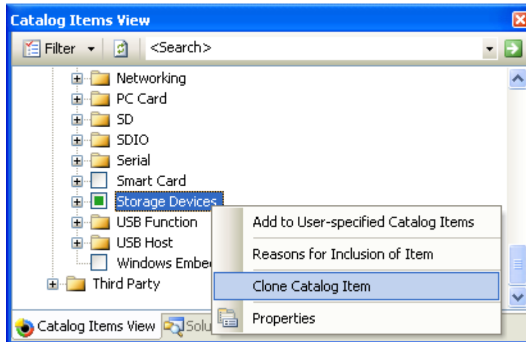


Figure 1-5 Cloning a catalog item

If you cannot automatically clone a component by using the IDE, you have to do it manually. However, when you look at the sources file for a .dll or .exe file in the Public directory tree, you see that this file is not the same as the sources file in your platform directory or in a subproject directory. This is because the build process for the Public directory tree differs from the BSP build process. All the build instructions are defined in the makefile file, which is always located in the same directory as the associated sources file. The Public directory tree must support the Sysgen phase, where the required components are linked together relatively.

Converting a component from the Public directory tree to a BSP component or a subproject requires a number of steps, which are outlined in detail in the Platform Builder for Microsoft Windows Embedded CE product documentation under "Using the Sysgen Capture Tool" at *http://msdn2.microsoft.com/en-us/library/aa924385.aspx*.

Basically, you need to perform the following steps:

1. Copy the code of the desired Public component into a new directory.
2. Edit the sources file in the new directory and add the line RELEASETYPE=PLATFORM or change the value to PLATFORM if the line already exists so that the build engine places the output from this build into the %_TARGETPLATROOT% folder.
3. Add WINCEOEM=1 to the sources file and build the component in the new directory. This might require further modifications to resolve all build errors.
4. Use the Sysgen Capture tool to create modular sources and dirs files.
5. Rename and use the files created by the Sysgen Capture tool along with a makefile to rebuild the new cloned module.

Once you apply all required modifications to the cloned component, you can modify and redistribute it as easily as any other code.

Lesson Summary

Windows Embedded CE includes a Public directory tree with the source code for most of the CE components, but you should not modify the source code in the Public directory tree directly. Instead, you should clone the items either automatically or manually. Modifying the source code in the Public directory tree causes more trouble now as well as in the future unless you already know the build system very well, in which case you already know all the good reasons why you should use the cloning method.

Lesson 4: Managing Catalog Items

One of Windows Embedded CE's most useful features is its catalog system. By using the catalog, developers can quickly and conveniently customize the Windows Embedded CE firmware to suit their needs. If you create a custom catalog item for each of your components, you can facilitate the installation and configuration of your components. This is a differentiating factor between ad-hoc and professional Windows Embedded CE solutions. For ad-hoc solutions, it might be sufficient to provide basic installation notes and a list of required SYSGEN variables, but professional software should include catalog items with proper values for SYSGEN variables and configuration settings.

After this lesson, you will be able to:

- Customize the content of the catalog.
- Add a new component entry to a BSP catalog.

Estimated lesson time: 20 minutes.

Catalog Files Overview

The Windows Embedded CE catalog uses files in Extensible Markup Language (XML) format with a .pbcxml file-name extension. The catalog includes a large number of .pbcxml files, located inside the WINCEROOT directory. Platform Builder automatically enumerates these files to generate the Catalog Items View in Solution Explorer.

Platform Builder parses the following directories to enumerate catalog items:

- **Public catalog files** %_WINCEROOT%\Public\<*any subdirectory*>\Catalog\
- **BSP catalog files** %_WINCEROOT%\Platform\<*any subdirectory*>\Catalog\
- **Third-party catalog files** %_WINCEROOT%\3rdParty\<*any subdirectory*>\Catalog\
- **Common system-on-chip (SOC) files** %_WINCEROOT%\Platform\Common\Src\soc\<*any subdirectory*>\Catalog\

NOTE 3rdParty folder

The 3rdParty folder usually contains standalone applications or source applications that can be included and distributed as part of an OS design. By enumerating the .pbcxml files in the 3rdParty folder, Platform Builder provides a way to add entries to the Catalog Items View for those components.

Creating and Modifying Catalog Entries

To add a new catalog item to the Windows Embedded CE catalog, you can create a copy of an existing catalog file (.pbcxml file) and then edit the file content by using the Catalog Editor provided with Platform Builder. You can also create a new catalog file in Platform Builder if you open the File menu in Visual Studio, point to New, and then select File. In the New File dialog box, under Platform Builder for CE 6.0 R2, select Platform Builder Catalog File, and then click Open.

NOTE Editing catalog files

Always edit catalog files by using the Catalog Editor provided with Platform Builder. There are no settings that require you to work with a text editor such as Notepad. Opening and editing catalog files manually outside of Platform Builder is unnecessarily time-consuming.

Catalog Entry Properties

Each catalog entry has a number of properties that you can modify in Platform Builder, as illustrated in Figure 1-6. The most important properties include the following:

- **Unique Id** A unique identifier string.
- **Name** The name of the catalog component as it appears in the Catalog Items View.
- **Description** An expanded description of the component, which appears when the user hovers the mouse pointer over the catalog item for several seconds.
- **Modules** A list of files that belong to this catalog component.
- **Sysgen variable** An environment variable for the catalog item. If your catalog component sets a SYSGEN variable, this is where to put it.
- **Additional Variables** A collection of additional environment variables for the catalog item. This is possibly the most important part of the catalog component in a BSP, because this field enables you to set environment variables used in

sources, .bib, and .reg files to control the build process. You can also use this field to generate dependencies on other components.

- **Platform directory** The location of the catalog item files. For a new BSP, set this property to the name of the BSP's directory.

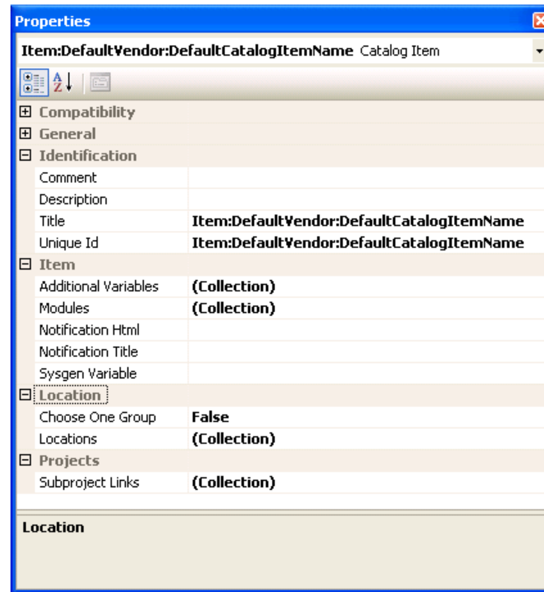


Figure 1-6 Catalog item properties

> **NOTE** Unique names
>
> Each catalog component must have a unique ID, typically composed of the vendor and the component names. When you clone a BSP by using the Clone Catalog Item feature, Platform Builder creates a unique name for the cloned component automatically; however, when editing catalog files manually, be sure to use unique identifiers.

Adding a New Catalog Item to an OS Design

To use a new catalog file or catalog item, ensure that the corresponding .pbcxml file exists in a subfolder called Catalog under a subdirectory of the 3rdParty or Platform directories, and then click the Refresh Catalog Tree button in the Catalog Items View in Visual Studio. Platform Builder dynamically regenerates the catalog by traversing the 3rdParty and Platform directories and processing any existing catalog files. With the new component listed in the Catalog Items View, you can include it in the OS design by selecting its check box, as explained earlier in Lesson 1.

Using a Catalog Item for BSP Development

Now that you have added your new catalog component and learned how to set item-specific environment variables, you can use this technique to include the component in a BSP, set C/C++ build directives, and modify system registry settings in the run-time image. When other developers using this BSP select your catalog item in an OS design project, they will implicitly use the settings you specified. To include a catalog component in a BSP, you need to edit the BSP's Platform.bib file and add a conditional statement based on your settings. You can choose to include a component if a variable is or isn't defined by using if-else statements. Note that it might be necessary to run the Rebuild Current BSP and Subprojects command, which you can find in Visual Studio on the Build menu, under Advanced Build Commands, for changes to the .bib and .reg files to take effect. Chapter 2 covers the Rebuild Current BSP and Subprojects command in more detail.

To set a C/C++ directive based on an environment variable that you specified in the catalog item's properties, you can use a conditional statement in the sources file based on the variable and add a **CDEFINES** entry. You should generally try to avoid setting C/C++ build directives based on catalog item properties, as this approach will make it difficult to distribute a binary version of your BSP in the future.

You can also change entries in the system registry by using conditional statements. You only need to edit the .reg files to include or exclude certain registry files related to the new component.

Exporting a Catalog Item from the Catalog

Some catalog items do not support direct cloning. To clone these components, you must create either a new catalog file, if you are creating a new entry under the 3rdParty folder, or a new entry in a BSP's existing catalog file. In any case, you should verify that the original values for all SYSGEN and additional environment variables are preserved. Do not forget to change the ID, because each item in the catalog must have a unique ID, as mentioned earlier in this lesson.

Catalog Component Dependencies

The catalog in Platform Builder for Windows Embedded CE 6.0 R2 supports component dependencies. To specify that a component is dependent on another component, you must set the SYSGEN or Additional Variables field for the component of the catalog item, and then include this value in the form of an additional environment variable in the dependent component. For example, if you have catalog components

in your BSP for both a display driver and a backlight driver for the display, you can set the Additional Variables field for the display driver to **BSP_DISPLAY** and the Additional Variables field for the backlight driver to **BSP_BACKLIGHT**. If you now want the display driver to be dependent on the backlight driver, you can edit the catalog entry for **BSP_DISPLAY** in the Catalog Editor and add **BSP_BACKLIGHT** to the additional environment variables. Then, whenever you include the display driver in an OS design, Platform Builder automatically includes the backlight driver as well. The Catalog Items View will show the check box of the backlight driver with a green square to indicate that this component is included as a dependency of the display driver.

Lesson Summary

Platform Builder for Windows Embedded CE 6.0 R2 comes with a file-based catalog system that you can use to contain your own catalog items by including them in separate catalog files in the Platform or 3rdParty directory within the %_WINCEROOT% directory tree. The file format of catalog files is XML and the file-name extension is .pbcxml. Platform Builder automatically enumerates the .pbcxml files when you start Visual Studio or refresh the Catalog Items View in Solution Explorer. To add a new catalog item to the Windows Embedded CE catalog, you can start with a new catalog file or create a copy of an existing catalog item and then edit the file content by using the Catalog Editor. There is no need to edit .pbcxml files by using a text editor, such as Notepad, because all settings are available directly within Platform Builder. Among other things, you can specify SYSGEN and additional environment variables for conditional C/C++ build directives, registry modifications, and dependency definitions.

Lesson 5: Generating a Software Development Kit

Developers who want to create applications for a target device require a Software Development Kit (SDK). An SDK will automatically correspond to your OS design so that the developers can only use those features that are actually available. The SDK includes features that are present in the OS design so that application developers do not accidentally create code that fails to run at run time due to an unsupported API.

After this lesson, you will be able to:

- Identify the purpose of an SDK.
- Generate an SDK.
- Localize SDK files on your hard drive.
- Use an SDK.

Estimated lesson time: 20 minutes.

Software Development Kit Overview

In order to compile and create valid applications for your OS design, developers need to include the necessary header files and link to the correct libraries in their development projects. You must ensure that the SDK for your OS design contains all required header files and libraries, including headers and libraries for any custom components you want to provide to application developers. Platform Builder for Windows Embedded CE 6.0 R2 enables you to create SDKs for your OS designs by exporting all the required header files and libraries.

SDK Generation

It is generally the task of the OS design creator to generate and distribute customized SDKs. Platform Builder provides an SDK export feature for this purpose. The SDK export feature creates the customized SDK for your OS design, along with a .msi file that includes the SDK Setup Wizard.

Configuring and Generating an SDK

To create and configure an SDK by using the SDK export feature of Platform Builder, follow these steps:

1. Configure your OS design and build it at least once in the Release configuration.
2. Display Solution Explorer, right-click SDKs, and select Add New to display the SDK Property Pages dialog box.
3. In the SDK Property Pages dialog box, configure the Install properties of the SDK and define the MSI Folder Path, MSI File Name, and Locale, as illustrated in Figure 1-7. You can also specify a number of custom settings.
4. To include additional files, select the Additional Folders node in the SDK Property Pages dialog box.
5. Click OK.

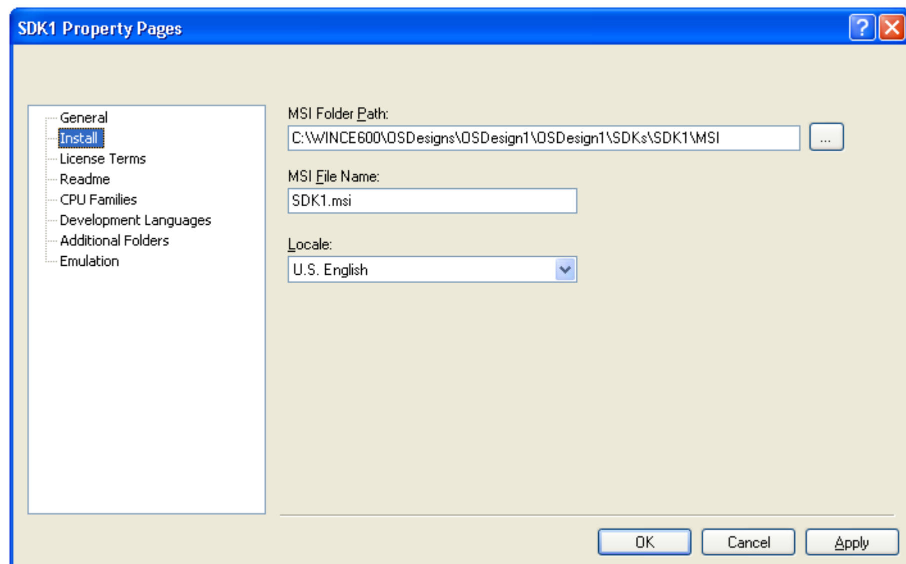


Figure 1-7 SDK Property Pages dialog box

Adding New Files to an SDK

You can add files to an SDK manually by either using the Additional Folders option in the SDK property pages or by copying them into the SDK directory for your OS design, typically in %_WINCEROOT%\OSDesigns\<*Solution Name*>\<*OS Design Name*>\WinCE600\<*Platform Name*>\SDK. It is also possible to automate that process by using .bat and sources files, so that the build engine copies the latest version of the files into the SDK each time you perform a build.

Make sure you copy the files into the following SDK subdirectories:

- **Inc** Contains the header files included in the SDK.
- **Lib\<*Processor Type*>\<*Build Type*>** Contains the libraries included in the SDK.

Installing an SDK

After completing the SDK build process, you can find the .msi file in the SDK subdirectory of your OS design folder. This is typically %_WINCEROOT%\OSDesigns \<*Solution Name*>\<*OS Design Name*>\SDKs\SDK1\MSI\<*SDK Name*>.msi. You can freely redistribute this MSI according to your licensing agreements for Platform Builder and any third-party components included.

You can install this MSI package on any computer with Visual Studio 2005 and use it to develop Windows Embedded CE applications for your target device. On a computer with the SDK installed, you can find the files under %PROGRAMFILES% \Windows Embedded CE Tools\WCE600.

Lesson Summary

Windows Embedded CE 6.0 R2 is a componentized operating system, which implies that application developers require a customized SDK that corresponds to your OS design in order to develop applications that will work on your target device. The custom SDK should not only include the required Windows Embedded CE components, but also the headers and libraries for any custom components that you included in the OS design, to avoid problems due to missing files or libraries at build and run time. Platform Builder provides an SDK export feature to generate SDKs and to create an MSI package for convenient SDK deployment on application development computers by means of an SDK Setup Wizard.

Lab 1: Creating, Configuring, and Building an OS Design

In this lab, you create an OS design, and then customize that design by adding components from the catalog. It is important to complete all the procedures in this lab, because it provides the foundation for subsequent exercises in other chapters of this Microsoft Windows Embedded CE 6.0 R2 Exam Preparation Kit.

NOTE Detailed step-by-step instructions

To help you successfully master the procedures presented in this lab, see the document "Detailed Step-by-Step Instructions for Lab 1" in the companion material for this book.

▶ Create an OS Design

1. In Visual Studio 2005 with Platform Builder for Windows Embedded CE 6.0 R2, select the File menu, New submenu, and Project menu option, and then create a new OS design project.
2. Use the default OS design name (OSDesign1).
3. Visual Studio will launch the Windows Embedded CE 6.0 OS Design Wizard.
4. Select the check box for Device Emulator: ARMV4I in the BSP list and click Next.
5. From the list of available design templates, select PDA Device. From the list of available design variants select Mobile Handheld.
6. Deselect .NET Compact Framework 2.0 and ActiveSync on the next wizard page, as illustrated in Figure 1-8. The Internet Browser and Quarter VGA Resources-Portrait Mode check boxes should remain checked.
7. On the Networking Communications wizard page, deselect TCP/IPv6 Support and Personal Area Network (PAN) to exclude Bluetooth and Infrared Data Association (IrDA) support. Leave Local Area Network (LAN) selected.
8. Click Finish to complete the Windows Embedded CE 6.0 OS Design Wizard. On completion, Visual Studio opens your OS design project. The Solution Explorer tab should be active and show your new OS design project under the Solution container.

Figure 1-8 Creating an OS design for a PDA device

NOTE Subsequent OS design changes

The OS Design Wizard creates the initial configuration for your OS design. You can make further changes to the OS design after completing the wizard.

▶ Inspect the OS Catalog

1. In Visual Studio, locate Solution Explorer and click the Catalog Items View tab.
2. Expand the individual container nodes to analyze the selected check boxes and icons in the catalog. Check boxes with a green check mark indicate items specifically added as a part of the OS design. Check boxes with a green square indicate items that are part of the OS design due to dependencies. Selection boxes that are not marked indicate items that are not included in the OS design but are available to be added.
3. Locate a catalog item with a green square in its check box.
4. Right-click this catalog item and choose Reasons For Inclusion Of Item. The Remove Dependent Catalog Item dialog box displays the catalog items that caused Platform Builder to include the selected catalog item in the OS design, as illustrated in Figure 1-9.
5. Expand the Core OS | CEBASE | Applications – End User | ActiveSync node in the catalog.
6. Right-click either of the ActiveSync system cpl items and select Display In Solution View. The view changes to the Solution Explorer tab to display the subproject containing the ActiveSync component. This is a great way to navigate through the source code that comes with Windows Embedded CE 6.0.

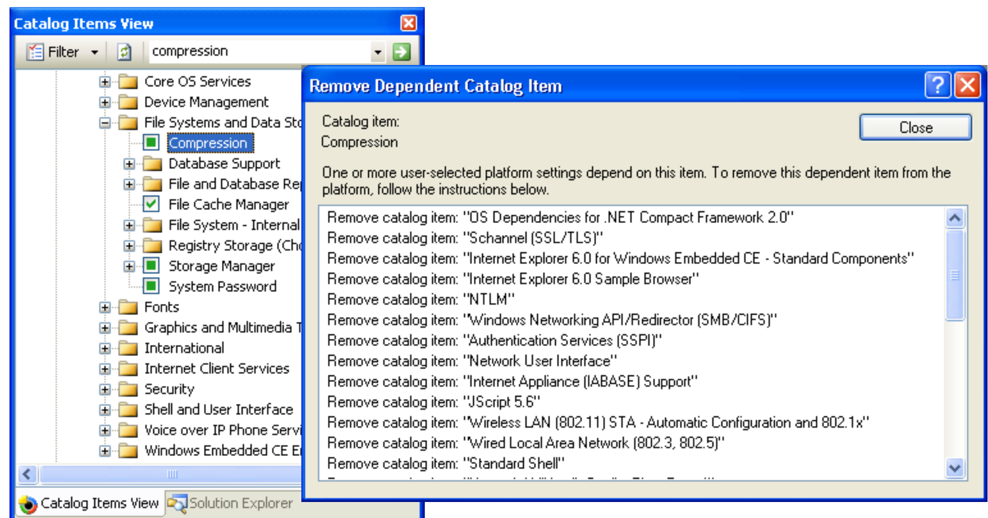


Figure 1-9 Reason for including a catalog item as a dependency

▶ Add Support for the Internet Explorer 6.0 Sample Browser Catalog Item

1. Select the Catalog Items View tab to display the OS design catalog. Verify that the filtering option is set to All Catalog Items In Catalog.
2. In the Search text box to the right of the Catalog Items View Filter button, type **Internet Explorer 6.0 Sample** and press Enter or click the green arrow.
3. Verify that the search locates the Internet Explorer 6.0 Sample Browser catalog item. Select the corresponding check box to include this catalog item in the OS design, as illustrated in Figure 1-10.

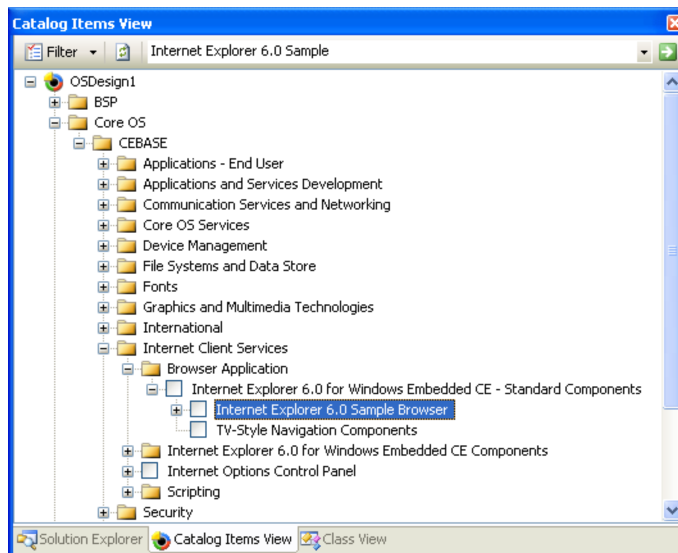


Figure 1-10 Including the Internet Explorer 6.0 Sample Browser catalog item in an OS design

▶ Add Support for Managed Code Development to Your OS Design

1. In the Search text box, type **ipconfig** and press Enter.
2. Verify that the search locates the Network Utilities (IpConfig, Ping, Route) catalog item.
3. Add the Network Utilities (IpConfig, Ping, Route) catalog item to your OS design by selecting the corresponding check box.
4. In the Search text box, type **wceload** and press Enter.
5. Verify that the search locates the CAB File Installer/Uninstaller catalog item. The search can find this catalog item because the value of its SYSGEN variable is **wceload**.

6. Add the Cab File Installer/Uninstaller catalog item to your OS design.
7. Use the search feature in a similar way to locate the OS Dependencies for .NET Compact Framework 2.0 container. Verify that the OS Dependencies for .NET Compact Framework 2.0 catalog item is included in your OS design, as illustrated in Figure 1-11.

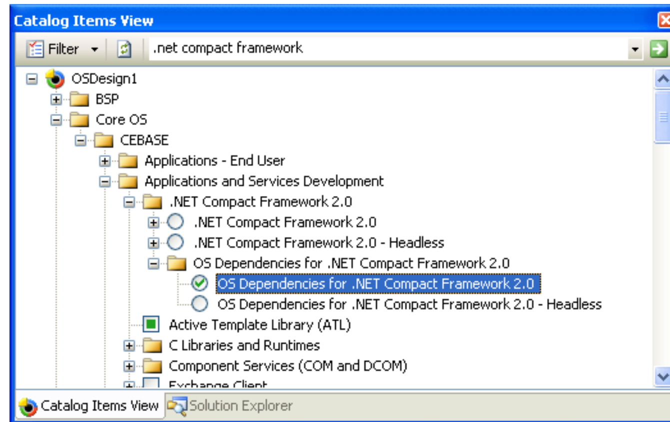


Figure 1-11 Adding the OS Dependencies for .NET Compact Framework 2.0 catalog item to an OS design

NOTE ***Headless*** **.NET Compact Framework**

There are two separate components in this category. Be sure you select the one that does not have the –Headless modifier in its description, because the headless version is intended for devices with no display.

Chapter Review

In order to deploy Microsoft Windows Embedded CE 6.0 R2 on a target device, you must create an OS design that includes the necessary operating system (OS) components, features, drivers, and configuration settings. You can then use Platform Builder to build the corresponding run-time image for deployment. The most important tasks you must accomplish to create a customized OS design that suits your requirements include:

- Creating an OS design project in Visual Studio by using the OS Design Wizard.
- Adding and removing components from the OS manually and through dependencies.
- Setting environment and SYSGEN variables through the Catalog Editor.
- Configuring regional settings for localization of the OS design.
- Cloning components from the catalog either automatically by clicking Clone Catalog Item or manually by using the Sysgen Capture tool.
- Exporting a custom SDK for your OS design to facilitate application development for your target device.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- OS design
- Component
- SYSGEN variable
- Environment Variable
- Software Development Kit

Suggested Practice

To help you successfully master the exam objectives presented in this chapter, complete the following tasks:

Create a Custom OS Design

By using the OS Design Wizard, create an OS design based on the Device Emulator: ARMV4I BSP and the Custom Device design template. Perform the following tasks after OS design creation:

- **Add the .NET Compact Framework 2.0** Add this catalog item by using the search feature in the Catalog Items View.
- **Localize your run-time image** Display the OS Design property pages and localize the OS design for the French language.

Generate and Test an SDK

Based on the OS design generated during Lab 1, perform the following tasks:

- **Build and generate the binary image** Build and generate the binary image for the OS design generated in the Release build configuration.
- **Create and install the SDK** Verify that the build process completes successfully, and then create a new SDK, build it, and install it on an application development computer.
- **Use the SDK** Use another instance of Visual Studio and create a Win32 Smart Device application. Use your custom SDK as the reference SDK for the project and build the application.

Chapter 2

Building and Deploying a Run-Time Image

The Microsoft® Windows® Embedded CE 6.0 R2 build process is very complex. This process includes several phases and relies on a variety of tools to initialize the Windows Embedded CE build environment, compile the source code, copy modules and files to a common release directory, and create the run-time image. Batch files and build tools, such as the Sysgen tool (Sysgen.bat) and the Make Binary Image tool (Makeimg.exe), automate this process. You can run these tools directly at the command prompt or start the build process in Microsoft Platform Builder for Windows Embedded CE 6.0 R2. The Platform Builder integrated development environment (IDE) relies on the same processes and tools. In either case, a thorough understanding of the build process and the steps required to deploy the resulting run-time image is essential if you want to create run-time images efficiently, troubleshoot build errors, or deploy Board Support Packages (BSPs) and subprojects as part of a run-time image on a target device.

Exam objectives in this chapter:

- Building run-time images
- Analyzing build results and build files
- Deploying a run-time image on a target device

Before You Begin

- To complete the lessons in this chapter, you must have:
- An understanding of operating system (OS) design aspects, including catalog items and the configuration of environment variables and SYSGEN variables, as explained in Chapter 1, "Customizing the Operating System Design."
- At least some basic knowledge about Windows Embedded CE software development, including source code compilation and linking.
- A development computer with Microsoft Visual Studio® 2005 Service Pack 1 and Platform Builder for Windows Embedded CE 6.0 R2 installed.

Lesson 1: Building a Run-Time Image

The Windows Embedded CE build process is the final step in the run-time image development cycle. Based on the settings defined in the OS design, Platform Builder compiles all components, including subprojects and the BSP, and then creates a run-time image that you can download to the target device. The build process entails several build phases, automated by means of batch files. You must understand these phases and the build tools if you want to configure build options correctly, create run-time images efficiently, and solve build-related issues.

After this lesson, you will be able to:

- Understand the build process.
- Analyze and fix build issues.
- Deploy a run-time image to target hardware.

Estimated lesson time: 40 minutes.

Build Process Overview

The Windows Embedded CE build process includes four main phases, as illustrated in Figure 2–1. They follow each other sequentially, but you can also carry them out independently if you know the purpose and tools used for each phase. By selectively running the build tools, you can perform individual build steps in a targeted way, which helps to save build time and ultimately increases your efficiency.

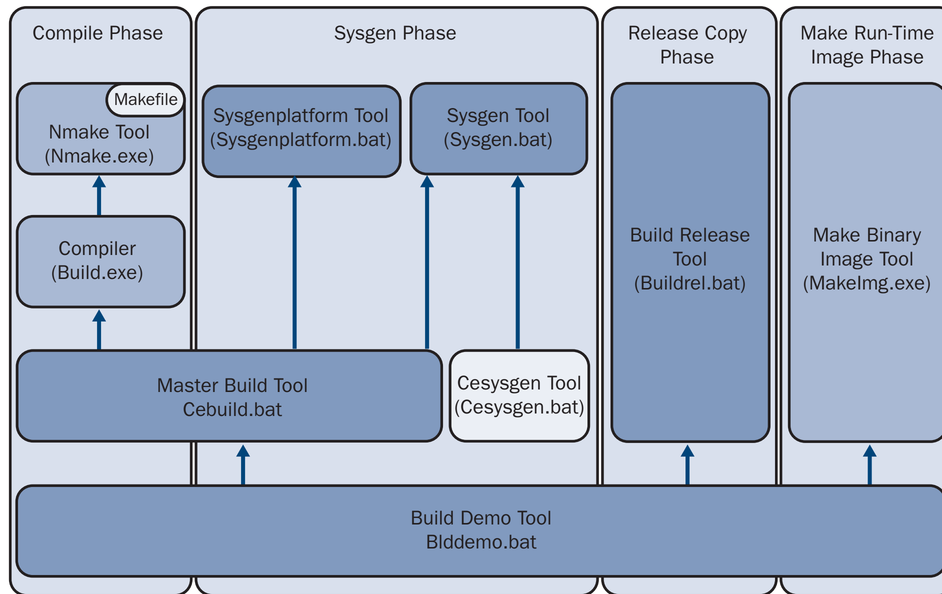


Figure 2-1 Build phases and build tools

The build process includes the following key phases:

- **Compile phase** Compiler and linker use source code and resource files to generate executable (.exe) files, static (.lib) libraries, dynamic-link library (.dll) files, and binary resource (.res) files according to the selected locales. For example, the build system compiles the source code in the Private and Public folders into .lib files during this phase. This process can take several hours to complete, but fortunately it is seldom required to rebuild these components because binaries are already provided by Microsoft. In any case, you should not modify the source code in the Private and Public folders.
- **Sysgen phase** The build system sets or clears SYSGEN variables based on the catalog items and dependency trees included in the OS design, filters the header files and creates import libraries for the Software Development Kits (SDKs) defined in the OS design, creates a set of run-time image configuration files for the OS design, and builds the BSP based on the source files in the Platform directory.
- **Build phase** The build system processes the source files of your Board Support Package and applications using the files generated during the Sysgen phase. At this time, hardware-linked drivers and the OEM adaptation layer (OAL) are

built. Although the processes during the build phase are carried out automatically during the Sysgen phase, it is important to understand that if you modify only the BSP and subprojects, then you can rebuild the BSP and subprojects without running the Sysgen tool again.

- **Release Copy phase** The build system copies all files required to create the run-time image to the OS design's release directory. This includes the .lib, .dll, and .exe files created during the Compile and Sysgen phases, as well as binary image builder (.bib) and registry (.reg) files. The build system might skip this phase if headers and libraries are up-to-date.
- **Make Run-time Image phase** The build system copies project-specific files (Project.bib, Project.dat, Project.db, and Project.reg) to the release directory and assembles all files in the release directory into a run-time image. Directives based on environment variables specified in .reg and .bib files control which catalog items the build system includes in the final run-time image. The run-time image is typically a file named Nk.bin, which you can download and run on the target device.

Building Run-Time Images in Visual Studio

During the installation of Windows Embedded CE 6.0 R2 on your development workstation, Platform Builder integrates with Visual Studio 2005 and extends the Build menu so that you can control the build process directly in the Visual Studio IDE. Figure 2–2 shows the Platform Builder commands that are available on the Build menu when you select the OS design node in Solution Explorer.

You can use the Platform Builder commands on the Build menu to perform selective build steps or a combined series of steps that span multiple build phases. For example, you can use the Copy Files To Release Directory command to ensure that the build system copies updated .bib and .reg files to the release directory even if header files and libraries have not changed. Otherwise, the build system skips the Release Copy phase and .bib file or .reg file changes are not applied to the run-time image.

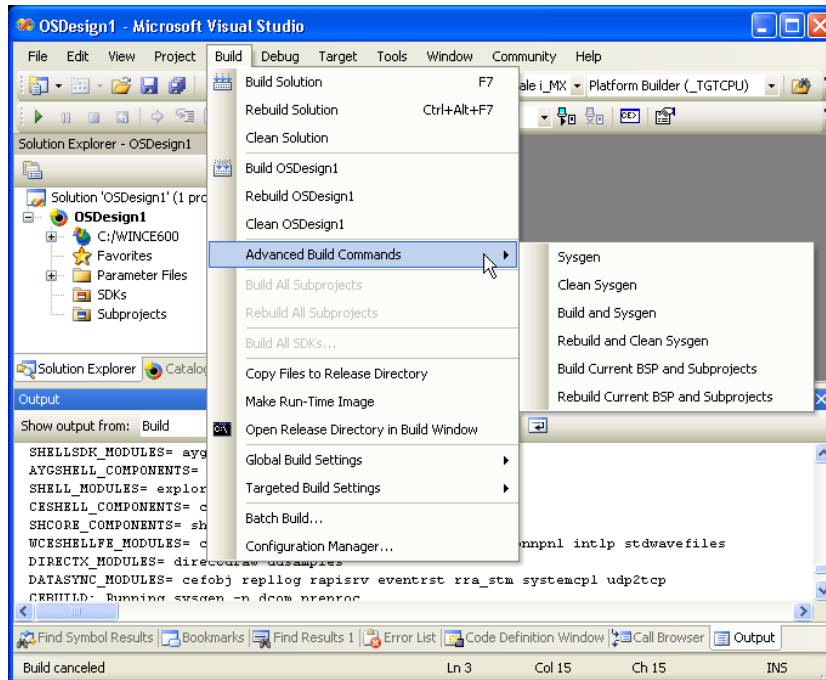


Figure 2-2 Windows Embedded CE build commands in Visual Studio 2005

Table 2–1 summarizes the purpose of the Windows Embedded CE build commands.

Table 2-1 Windows Embedded CE build and rebuild commands

Menu Option	Description
Build Solution	Equivalent to the Sysgen command on the Advanced Build Commands submenu.
Rebuild Solution	Equivalent to the Clean Sysgen command on the Advanced Build Commands submenu.
Clean Solution	Cleans the release directory by deleting all intermediate files.
Build <*OS Design Name*>	Helpful in solutions that include multiple OS designs. In solutions with a single OS design, these options correspond to the Build Solution, Rebuild Solution, and Clean Solution commands.
Rebuild <*OS Design Name*>	
Clean <*OS Design Name*>	

Table 2-1 Windows Embedded CE build and rebuild commands (Continued)

Menu Option		Description
Advanced Build Commands	Sysgen	Runs the Sysgen tool and links the .lib files in the Public and Private folders to create the files for the run-time image. The files remain in the WinCE folder of the OS design. Depending on global build settings, the build process can automatically advance to the Release Copy and then Make Run-time Image phases.
	Clean Sysgen	Cleans out intermediate files created during previous builds before running the Sysgen tool. Use this option if you added or removed files or catalog items after a previous Sysgen session to reduce the risk of build errors.
	Build And Sysgen	Compiles the entire contents of the Public and Private folders, and then links the files by using the settings in your OS design. This process takes several hours and is only necessary if you modified the contents of the Public folder. Unless you modify the Windows Embedded CE code base (not recommended), you should not use this option.
	Rebuild And Sysgen	Cleans out intermediate files created during previous builds in the Public and Private folders, and then runs the Build and Sysgen steps. You should not use this option.
	Build Current BSP And Subprojects	Builds the files in the directory for the current BSP and any subprojects in the OS design, and then runs the Sysgen tool. Note that this option will build other BSPs than the ones used in the current OS design, so make sure your BSPs are compatible with each other or remove unused BSPs.
	Rebuild Current BSP And Subprojects	Cleans out intermediate files created during previous builds, and then runs the Build Current BSP And Subprojects steps.

Table 2-1 Windows Embedded CE build and rebuild commands (Continued)

Menu Option		Description
Build All Subprojects		Compiles and links all subprojects, skipping any files that are up-to-date.
Rebuild All Subprojects		Cleans, compiles, and links all subprojects.
Build All SDKs		Builds all SDKs in the project and creates corresponding Microsoft Installer (MSI) packages. Because there is generally no reason to create debug versions of MSI packages, use this option only for a Release build configuration.
Copy Files To Release Directory		Copies the files generated for the BSP and other components during the Compile and Sysgen phases to the release directory in order to include these file in the run-time image.
Make Run-Time Image		Takes all the files in the release directory to create the run-time image. Following this step, you can download the run-time image to a target device.
Open Release Directory In Build Window		Opens a Command Prompt window, changes into the release directory, and sets all necessary environment variables to run batch files and build tools manually. Use this to perform build steps at the command prompt. The standard Command Prompt window does not initialize the development environment to run the build tools successfully.
Global Build Settings	Copy Files To Release Directory After Build	Enables or disables automatic advancement to the Release Copy phase for all commands.
	Make Run-Time Image After Build	Enables or disables automatic advancing to the Make Run-time Image phase after any build operation.

Table 2-1 **Windows Embedded CE build and rebuild commands (Continued)**

Menu Option		Description
Targeted Build Settings	Make Run-Time Image After Build	Enables or disables the Make Run-time Image phase.
Batch Build		Enables you to perform multiple builds sequentially.
Configuration Manager		Enables you to add or remove build configurations.

The Advanced Build Commands submenu provides access to several Platform Builder–specific build commands that you might find useful on a regular basis. For example, you need to run the Sysgen or Clean Sysgen command when you add or remove catalog components to or from the OS design to create the binary versions for the run-time image. Exceptions to this rule are components that do not modify SYSGEN variables, such as components in the ThirdParty folder. It is not necessary to run Sysgen or Clean Sysgen when you select or deselect these items. Following the Sysgen phase, Platform Builder continues the build process similar to running the Build Current BSP And Subprojects command.

You can select the Build Current BSP And Subprojects or the Rebuild Current BSP And Subprojects commands in Visual Studio if you want to compile and link the source code in the Platform directory and any subprojects in the OS design and put the code into the target directory under Platform\<*BSP Name*>\Target and Platform\<*BSP Name*>\Lib. This is necessary, for instance, if you modify the source code in the Platform directory. Depending on the Copy Files To Release Directory After Build and Make Run-Time Image After Building options, Platform Builder copies the files to the release directory and creates the run-time image. You can also perform these steps individually either through the menu or by running the Buildrel.exe and Makeimg.exe tools at the command prompt.

CAUTION **Clean Sysgen affects multiple build configurations**

If you run the Clean Sysgen command in one build configuration, you also have to run Sysgen for the other build configurations later. Keep in mind that the Clean Sysgen command deletes all files generated for other build configurations, as well as for the current build configuration.

Building Run-Time Images from the Command Line

The Platform Builder for CE6 R2 plug-in for Visual Studio 2005 provides convenient access to batch files and build tools, but you can also run these batch files and build tools directly at the command prompt. Each build command in Visual Studio with Platform Builder corresponds to a specific build command, as listed in Table 2–2. Remember, however, to use the Open Build Window command in Visual Studio to open a Command Prompt window for this purpose. The standard command prompt does not initialize the development environment. The build process will fail without the presence of the required environment variables.

Table 2-2 Build commands and command line equivalents

Build Command	Command Line Equivalent
Build	blddemo -q
Rebuild	blddemo clean -q
Sysgen	blddemo -q
Clean Sysgen	blddemo clean -q
Build And Sysgen*	Blddemo
Rebuild And Clean Sysgen*	blddemo clean cleanplat -c
Build Current BSP And Subprojects	blddemo -qbsp
Rebuild Current BSP And Subprojects	blddemo -c -qbsp

* Not recommended

Windows Embedded CE Run-Time Image Content

As illustrated in Figure 2–3, the run-time image includes all items and components that you want to deploy and run on a target device as part of the OS design, such as kernel components, applications, and configuration files. The most important configuration files for developers are binary image builder (.bib) files, registry (.reg), database (.db), and file system (.dat) files. These files determine the memory layout and specify how Platform Builder initializes the file system and the system registry. It is important to know how to work with these files. For example, you can modify the .reg and .bib files for a BSP directly in the OS design or create a subproject to add custom settings to the run-time image in a more componentized way. As mentioned in

Chapter 1, it is generally faster and more convenient to modify the .reg and .bib files of an OS design directly, yet subprojects facilitate the reuse of customizations across multiple OS designs.

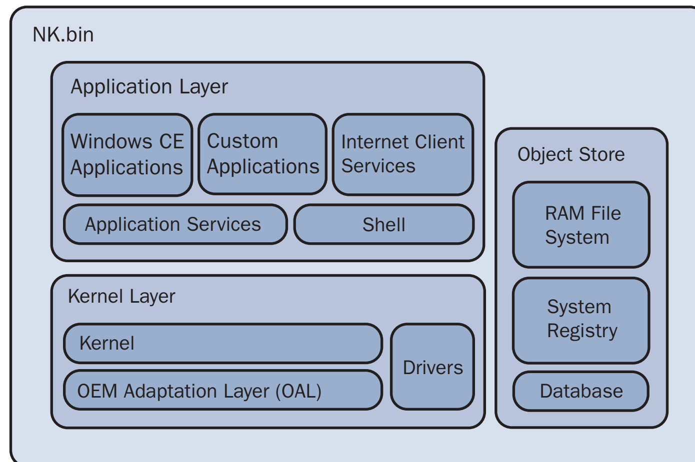


Figure 2-3 Contents of a run-time image

Binary Image Builder Files

The Windows Embedded CE build process relies on .bib files to generate the content of the run-time image and to define the final memory layout of the device. At the end of the build process, during the Make Run-time Image phase, the Make Binary Image tool (Makeimg.exe) calls the File Merge tool (Fmerge.exe) to combine all applicable .bib files, such as Config.bib and Platform.bib from the Platform\<*BSP Name*>\Files folder, Project.bib, Common.bib, and any subproject .bib files, into a file named Ce.bib in the release directory. The Make Binary Image tool then calls the ROM Image Builder tool (Romimage.exe) to process this file and determine which binaries and files to include in the run-time image.

A .bib file can include the following sections:

- **MEMORY** Defines the parameters for the memory layout. You can typically find this section in the Config.bib file for your BSP, such as C:\Wince600\Platform\DeviceEmulator\Files\Config.bib.
- **CONFIG** Defines configuration options for Romimage.exe to customize the binary run-time image. You can typically find this section in the Config.bib file for your BSP. This section is optional.

- **MODULES** Specifies a list of files that Romimage.exe marks to be loaded into RAM or executed in place (XIP). Only uncompressed object modules can execute directly from read-only memory. You can list native executable files in this section, but not managed binaries, because the Common Language Runtime (CLR) must convert the Microsoft Intermediate Language (MSIL) content into native machine code at run time.
- **FILES** References executables and other files that the operation system should load into RAM for execution. You should specify managed code modules in this section.

.BIB File MEMORY Section The MEMORY section in the Config.bib file defines reserved memory regions, assigning each region a name, address, size, and type. A good example is the MEMORY section that you can find in the Config.bib file in the Device Emulator BSP. This Device Emulator BSP is available with Platform Builder for CE 6.0 R2 out-of-the-box. You can find Config.bib in the PLATFORM\\<*BSP Name*>\FILES directory. Figure 2–4 shows this MEMORY section in Visual Studio 2005.

```
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
```

Figure 2-4 MEMORY section from a .bib file

The fields in the MEMORY section define the following parameters:

- **Name** This is the name of the MEMORY section. The name must be unique.
- **Address** This hexadecimal number represents the starting address of the memory section.
- **Size** This hexadecimal number defines the total length of the memory section in bytes.
- **Type** This field can have one of the following values:
  - ***RESERVED*** Indicates that this area is reserved. Romimage.exe skips these sections during image creation. For example, the Ce.bib file shown in Figure 2–4 includes several RESERVED sections, such as an ARGS section to provide a shared memory area for the boot loader (EBOOT) to pass data to the system after startup (ARGS) and a DISPLAY section for a display buffer. The Ce.bib file of other OS designs might include different RESERVED sections for memory areas that the kernel is not supposed to use as system memory.
  - ***RAMIMAGE*** Defines the memory area that the system can use to load the kernel image and any modules you specified in the MODULES and FILES sections of .bib files. A run-time image can only have one RAMIMAGE section and the address range must be contiguous.
  - ***RAM*** Defines a memory area for the RAM file system and for running applications. This memory section must be contiguous. If you need a noncontiguous memory section, such as for extension dynamic RAM (DRAM) present on the device, you can allocate noncontiguous memory by implementing the OEMGetExtensionDRAM function in the OAL of the BSP. Windows Embedded CE supports up to two sections of physical noncontiguous memory.

.BIB File CONFIG Section The CONFIG section defines additional parameters for the run-time image, including the following options:

- **AUTOSIZE** Automatically combines RAMIMAGE and RAM sections and allocates any unused memory in the RAMIMAGE section to RAM, or if necessary takes memory from the RAM section and provides it to the RAMIMAGE.
- **BOOTJUMP** If specified, moves the boot jump page to a specific area within the RAMIMAGE section, rather than by using the default area.
- **COMPRESSION** Automatically compresses writable memory sections in the image. The default value for this option is ON.

- **FIXUPVAR** Initializes a kernel global variable during the Make Binary Image phase.
- **FSRAMPERCENT** Sets the percentage of RAM used for the RAM file system.
- **KERNELFIXUPS** Instructs Romimage.exe to relocate memory writable by the kernel. This option is generally enabled (ON).
- **OUTPUT** Changes the directory that Romimage.exe uses as the output directory for the Nk.bin file.
- **PROFILE** Specifies whether the image includes the profiler.
- **RAM_AUTOSIZE** Expands the size of RAM to the end of the last XIP section.
- **RESETVECTOR** Relocates the jump page to a specified location. This is required for MIPS processors to boot from 9FC00000.
- **ROM_AUTOSIZE** Resizes XIP regions, taking into account the ROMSIZE_AUTOGAP setting.
- **ROMFLAGS** Configures the following options for the kernel:
  - ***Demand paging*** Fully copying a file into RAM before executing it or paging in parts of it.
  - ***Full kernel mode*** Run every OS thread in kernel mode, which leaves the system vulnerable to attack but improves performance.
  - ***Trust only ROM modules*** Marks only files in ROM as trusted.
  - ***Flush the X86 TLB on X86 systems*** Improves performance but adds a security risk.
  - ***Honor the /base linker setting*** Defines whether or not to use the /base linker setting in DLLs.
- **ROMOFFSET** Enables you to run the run-time image in a memory location that is different from the storage location. For example, you can store the run-time image in FLASH memory, and then copy and run it from RAM.
- **ROMSIZE** Specifies the size of the ROM in bytes.
- **ROMSTART** Specifies the ROM's starting address.
- **ROMWIDTH** Specifies the number of data bits and how Romimage.exe splits the run-time image. Romimage.exe can put the entire run-time image into one file, split the run-time image into two files of even and odd 16-bit words, or create four files of even and odd 8-bit bytes.
- **SRE** Determines whether Romimage.exe generates a .sre file. Motorola S-record (SRE) is a file format recognized by most ROM burners.

- **X86BOOT** Specifies whether or not to add a JUMP instruction at the x86 reset vector address.
- **XIPSCHAIN** Enables the creation of Chain.bin and Chain.lst files to set up an XIP chain, so that you can split an image into multiple files.

.BIB File MODULES and FILES Sections BSP and OS design developers must frequently edit the MODULES and FILES sections of a .bib file to add new components to a run-time image. The format for the MODULES and FILES section is practically identical, although the MODULES section supports more configuration options. The key difference is that the MODULES section lists files not compressed in memory to support XIP, while the FILES section lists files that are compressed. The operating system must decompress the data when accessing the files.

The following listing shows two small MODULES and FILES sections from a Platform.bib file. For a complete example, check out the Platform.bib file of the Device Emulator BSP.

```
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
```

You can define the following options for file references in MODULES and FILES sections:

- **Name** The name of the module or file as it appears in the memory table. This name is usually the same as the file name in the run-time image.
- **Path** The complete path to the file that Romimage.exe incorporates into the run-time image.

- **Memory** References the name of a memory area in the MEMORY section of the Config.bib file into which Romimage.exe loads the module or file. It is usually set to NK to integrate the file in the NK area defined in the MEMORY section.
- **Section Override** Enables you to specify modules in a FILES section and files in a MODULES section. Essentially, Romimage.exe ignores the section in which the entry resides, and treats the entry as a member of the specified section. This parameter is optional.
- **Type** Specifies the file type and can be a combination of flags, as shown in Table 2-3.

Table 2-3 File type definitions for MODULES and FILES sections

MODULES and FILES Sections	MODULES Section Only
■ **S** The file is a system file. ■ **H** The file is hidden. ■ **U** The file is uncompressed. (The default setting for files is compressed.) ■ **N** The module is not trusted. ■ **D** The module cannot be debugged.	■ **K** Instructs Romimage.exe to assign a fixed virtual address to the DLL's public exports and runs the module in kernel mode rather than user mode. Drivers must run in kernel mode to have direct access to the underlying hardware. ■ **R** Compress resource files. ■ **C** Compress all data in the file. If the file is already in RAM, it will be decompressed again into a new section of RAM, which results in higher RAM consumption. ■ **P** Do not check the CPU type on a per-module basis. ■ **X** Sign the module and include the signature in the ROM. ■ **M** Signals that the kernel must not page the module on demand. (See Chapter 3 for more information on the effects of demand paging.) ■ **L** Instructs Romimage.exe not to split the ROM DLL.

Conditional .bib File Processing It is important to note that .bib files support conditional statements based on environment variables and SYSGEN variables. You can set environment variables through catalog items, and then check these variables in IF statements in a .bib file to include or exclude certain modules or other files. For SYSGEN variables, use @CESYSGEN IF statements instead.

The MODULES and FILES listing in the previous section illustrates the use of @CESYSGEN IF and IF statements for processing conditions based on SYSGEN and environment variables. For example, the @CESYSGEN IF CE_MODULES_DISPLAY statement in the MODULES sections specifies that the BSP should automatically include the display driver if the OS design includes a display component. You can verify that Platform Builder adds the display component to the BSP automatically if you display the Catalog Items View in Visual Studio for an OS design that uses a display, as illustrated in Figure 2–5.

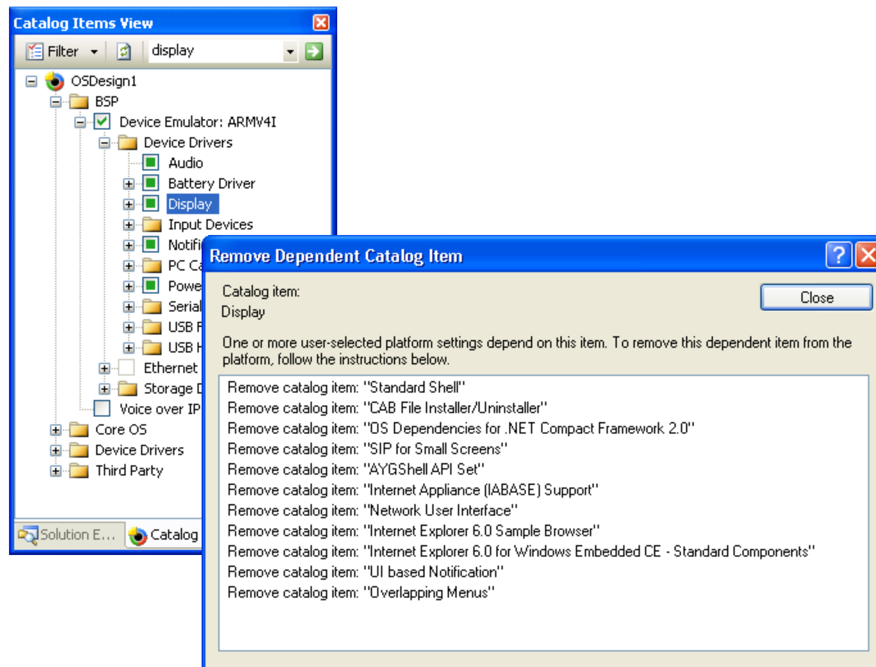


Figure 2-5 Core OS components that depend on the display item

Registry Files

Registry (.reg) files are used to initialize the system registry on the remote device. These files are almost identical to registry files of Windows desktop operating systems, except that the CE .reg files do not start with a header and version information. If you accidentally double-click a CE .reg file on your development computer and confirm that you want to add the settings to the desktop registry, a dialog box appears to inform you that the .reg file is not a valid registry script. Another difference is that CE .reg files can include conditional statements similar to .bib files, so that you can import registry settings according to the selected catalog items. The following snippet from the Platform.reg file of the Device Emulator BSP illustrates the use of preprocessing conditions.

```
; 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
```

Database Files

Windows Embedded CE relies on database (.db) files to set up the default object store. The object store is a transaction–based storage mechanism. In other words, it is a repository for databases in RAM that operating system and applications can use for persistent data storage. For example, the operating system uses the object store to

manage the stack and memory heap, to compress and decompress files, and to integrate ROM–based applications and RAM–based data. The transaction-oriented nature of the storage mechanism ensures data integrity even in the event of a sudden power loss while data is being written to the object store. When the system restarts, Windows Embedded CE either completes the pending transaction, or reverts to the last known good configuration prior to the interruption. For system files, the last known good configuration can mean that Windows Embedded CE must reload the initial settings from ROM.

File System Files

File system (.dat) files, specifically Platform.dat and Project.dat, contain settings to initialize the RAM file system. When you cold start the run-time image on a target device, Filesys.exe processes these .dat files to create the RAM file system directories, files, and links on the target device. The Platform.dat file is typically used for hardware-related entries while the Project.dat file applies to the OS design, yet you can use any existing .dat file to define file system settings because the build system eventually merges all .dat files into one file named Initobj.dat.

For example, by customizing the Project.dat file, you can define root directories in addition to the Windows directory for a run-time image. By default, items placed in the ROM image appear in the Windows directory, yet by using a .dat file, you can make files also appear outside the Windows directory. You can also copy or link to files in the ROM Windows directory. This is particularly useful if you want to place shortcuts on the desktop or add links to your applications to the Start menu. Similar to .reg and .bib files, you can use IF and IF ! (if not) conditional blocks in .dat files.

The following listing illustrates how to use a Project.dat file to create two new root directories named Program Files and My Documents, create a My Projects subdirectory under Program Files, and map the Myfile.doc file from the Windows directory into the My Documents directory.

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

Lesson Summary

A thorough understanding of the build system can help to decrease development time and therefore project costs. You must know the steps performed during each phase of the build process if you want to test source code changes quickly and without unnecessary compilation cycles. You must also know the purpose and location of the run-time image configuration files, such as .reg, .bib, .db, and .dat files, to create and maintain OS designs efficiently.

The Windows Embedded CE build system combines the various .reg, .bib, .db, and .dat files during the Make Run-time Image phase into consolidated files that the build system then uses to configure the final run-time image. It is a good idea to check these files if you want to verify that a specific setting or file made it into the final image without having to load the run-time image on the target device. You can find the various run-time image configuration files in the release directory of the OS design. If you discover that expected entries are missing, check the conditional statements and the environment variables and SYSGEN variables defined in your catalog items.

The build system creates the following run-time image configuration files during the Make Run-time Image phase:

- **Reginit.ini** Combines the Platform.reg, Project.reg, Common.reg, IE.reg, Wceapps.reg, and Wceshell.reg files.
- **Ce.bib** Combines the Config.bib, Platform.bib, Project.bib, and Subproject bib files.
- **Initdb.ini** Combines the Common.db, Platform.db, and Project.db files.
- **Initobj.dat** Combines the Common.dat, Platform.dat, and Project.dat files.

Lesson 2: Editing Build Configuration Files

In addition to run-time image configuration files, Windows Embedded CE also uses build configuration files to compile and link source code into functional binary components. Specifically, the build system relies on three types of source code configuration files: Dirs, Sources, and Makefile. These files provide the Build tool (Build.exe) and the compiler and linker (Nmake.exe) with information about the source-code directories to traverse, the source code files to compile, and what type of binary components to build. As a CE developer, you frequently must edit these files, such as when cloning public catalog items, by following the procedures discussed in Chapter 1.

After this lesson, you will be able to:

- Identify the source code configuration files used during the build process.
- Edit build configuration files to generate applications, DLLs, and static libraries.

Estimated lesson time: 25 minutes.

Dirs Files

Dirs files identify directories that contain source-code files to be included in the build process. When Build.exe finds a Dirs file in the folder in which it is run, it traverses the subdirectories referenced in the Dirs file to build the source code in these subdirectories. Among other things, this mechanism enables you to update parts of a run-time image selectively. If you make changes to the source code in Subproject1, you can rebuild this subproject selectively by running Build.exe in the Subproject1 directory. You can also exclude directories in the source code tree from the build process by removing the corresponding directory references from the Dirs file, or by using conditional statements.

Dirs files are text files with a straightforward content structure. You can use the DIRS, DIRS_CE, or OPTIONAL_DIRS keyword, and then specify the list of subdirectories on a single line, or on multiple lines if you terminate each line with a backslash to continue on the next line. Directories referenced by using the DIRS keyword are always included in the build process. If you use the DIRS_CE keyword instead, Build.exe only builds the source code if the source code is written specifically for a Windows Embedded CE run-time image. The OPTIONAL_DIRS keyword designates optional directories. Keep in mind, however, that Dirs files can contain only one DIRS directive. Build.exe processes the directories in the order they are listed, so be sure to

list prerequisites first. It is also possible to use the wildcard "*" to include all directories.

The following listing, taken from default Windows Embedded CE components, illustrates how to include source code directories in the build process by using Dirs files.

```
# 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
```

NOTE Editing Dirs files in Solution Explorer

The Solution Explorer in Visual Studio with Platform Builder for Windows Embedded CE 6.0 R2 uses Dirs files to generate a dynamic view of the Windows Embedded CE directory structure in an OS design project. However, you should not add or remove directories in Solution Explorer, because editing Dirs files in Solution Explorer can lead to a changed build order, which can result in build errors that require a second build to resolve.

Sources Files

If you check the folders and files of a standard OS design, such as C:\Wince600\OSDesigns\OSDesign1, you will find that the project includes no Dirs files by default. If you include subprojects for custom components and applications, you will find a Sources file in each subproject's root folder instead. The Sources file provides more detailed information about the source-code files, including build directives, which a Dirs file cannot provide. However, a source-code directory can only contain one Dirs file or one Sources file, not both. That means that a directory with a Sources file cannot contain subdirectories with more code. During the build process, Nmake.exe uses the Sources files to determine what file type to build (.lib, .dll, or .exe), and how to build it. Similar to Dirs files, Sources files expect you to specify declarations in a single line, unless you terminate the line with a backslash to continue the declaration on the next line.

The following listing shows the content of a Sources file in the Device Emulator BSP. By default, you can find this file in the C:\Wince600\Platform\DeviceEmulator \Src\Drivers\Pccard folder.

```
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}
```

You can define the following directives in a Sources file:

- **TARGETNAME** This is the name of the target file, without file name extension.
- **TARGETTYPE** Defines the type of file to be built, as follows:
- **DYNLINK** A dynamic-link library (.dll).
- **LIBRARY** A static-link library (.lib).
- **PROGRAM** An executable file (.exe).
- **NOTARGET** Build no file.
- **RELEASETYPE** Specifies the directory where Nmake.exe places the target file, as follows:
    - ***PLATFORM*** PLATFORM\<*BSP Name*>\<*Target*>.
    - ***OAK, SDK, DDK*** %_PROJECTROOT%\Oak\<*Target*>.
    - ***LOCAL*** The current directory.
    - ***CUSTOM*** A directory specified in TARGETPATH.
    - ***MANAGED*** %_PROJECTROOT%\Oak\<*Target*>\Managed.
- **TARGETPATH** Defines the path for RELEASETYPE=CUSTOM.
- **SOURCELIBS** Specifies libraries to be linked with the target file specified in TARGETNAME to create the final binary output. This option is typically used for creating a .lib file but not .dll or .exe files.
- **TARGETLIBS** Specifies additional libraries and object files to link to the final binary output, typically used for creating .dll or .exe files but not for .lib files.
- **INCLUDES** Lists additional directories to search for include files.
- **SOURCES** Defines the source files to be used for this particular component.
- **ADEFINES** Specifies parameters for the assembler.
- **CDEFINES** Specifies parameters for the compiler, which can be used as additional DEFINE statements for use in IFDEF statements.
- **LDEFINES** Sets linker definitions.
- **RDEFINES** Specifies DEFINE statements for the resource compiler.

- **DLLENTRY** Defines the entry point for a DLL.
- **DEFFILE** Defines the .def file which contains a DLL's exported symbols.
- **EXEENTRY** Sets the entry point of an executable file.
- **SKIPBUILD** Marks the build of the target as successful without an actual build of the target.
- **WINCETARGETFILE0** Specifies nonstandard files that should be built before building the current directory.
- **WINCETARGETFILES** This macro definition specifies nonstandard target files that Build.exe should build after Build.exe links all other targets in the current directory.
- **WINCE_OVERRIDE_CFLAGS** Defines compiler flags to override default settings.
- **WINCECPU** Specifies that the code requires a certain CPU type and should only be built for that particular CPU.

NOTE Performing specific actions before and after the build

In addition to the standard directives, Windows Embedded CE Sources files support the directives PRELINK_PASS_CMD and POSTLINK_PASS_CMD. You can use these directives to perform custom actions based on command-line tools or batch files before and after the build process, such as PRELINK_PASS_CMD=pre_action.bat and POSTLINK_PASS_CMD=post_action.bat. This is useful, for example, if you want to copy additional files to the release directory when developing a custom application.

Makefile Files

If you look closer at the contents of a subproject folder, you can also find a file named Makefile to provide default preprocessing directives, commands, macros, and other expressions to Nmake.exe. However, in Windows Embedded CE, this Makefile includes only a single line that references %_MAKEENVROOT%\Makefile.def. By default, the environment variable %_MAKEENVROOT% points to the C:\Wince600\Public\Common\Oak\Misc folder and the Makefile.def file in this location is the standard Makefile for all CE components, so you should not modify this file. Among other things, the Makefile.def file contains include statements to pull in Sources file, such as !INCLUDE $(MAKEDIR)\sources, which specify the Sources file from the subproject folder. You should edit the Sources file in the subproject folder to adjust the way Nmake.exe builds the target file.

Lesson Summary

The Windows Embedded CE 6.0 R2 development environment relies on Makefile, Sources, and Dirs files to control how Build.exe and Nmake.exe compile and link source code into functional binary components for the run-time image. You can use Dirs files to define the source code directories included in the build process or Sources files to specify compile and build directives in greater detail. The Makefile, on the other hand, requires no customization. It merely references the default Makefile.def file with general preprocessing directives, commands, macros, and other processing instructions for the build system. You must thoroughly understand the purpose of files and how they control the build process if you want to clone public catalog items or create new components efficiently.

Lesson 3: Analyzing Build Results

You are certain to encounter build errors during the software-development cycle. In fact, it is not uncommon to use compile errors as a syntax check for source code, although IntelliSense® and other coding aids available in Visual Studio 2005 help to reduce the amount of typos and other syntax errors. Syntax errors are relatively uncomplicated to fix because you can double-click the corresponding error message in the Output window and jump right to the critical line in the source code file. However, compiler errors are only one type of build errors that can occur. Other common build errors are math errors, expression evaluation errors, linker errors, and errors related to run-time image configuration files. In addition to error messages, the build system also generates status messages and warnings to help you analyze and diagnose build issues. The amount of information generated during the build process can be overwhelming. You need to know the different types and general format of build messages if you want to identify, locate, and solve build errors efficiently.

After this lesson, you will be able to:

- Locate and analyze build reports.
- Diagnose and solve build issues.

Estimated lesson time: 15 minutes.

Understanding Build Reports

When you perform a build either in the Visual Studio IDE or the command prompt, the build process outputs a significant amount of build information. The build system tracks this information in a Build.log file. Details about compilation or linker warnings and errors also can be found in the Build.wrn and Build.err files. If you started a complete build or a Sysgen operation for an OS design by using one of the corresponding commands on the Build menu in Visual Studio, the build system writes these files in the %_WINCEROOT% folder (by default, C:\Wince600). On the other hand, if you perform a build for only a particular component, such as by right-clicking a subproject folder in Solution Explorer and clicking the Build command from the context menu, the build system writes these files in that specific directory. In either case, the Build.wrn and Build.err files only exist if you encounter warnings and errors during the build process. However, you do not need to open and parse through these files in Notepad or another plain-text editor. Visual Studio 2005 with Platform Builder for CE 6.0 R2 displays this information during the build process in the

Output window. You can also examine status messages, warnings, and errors in the Error List window that you can display by clicking Error List, which is available under Other Windows on the View menu.

Figure 2–6 shows the Output window and the Error List window in undocked view. The Output window displays the Build.log content if you select Build from the Show Output From list box. The Error List window displays the contents Build.wrn and Build.err files.

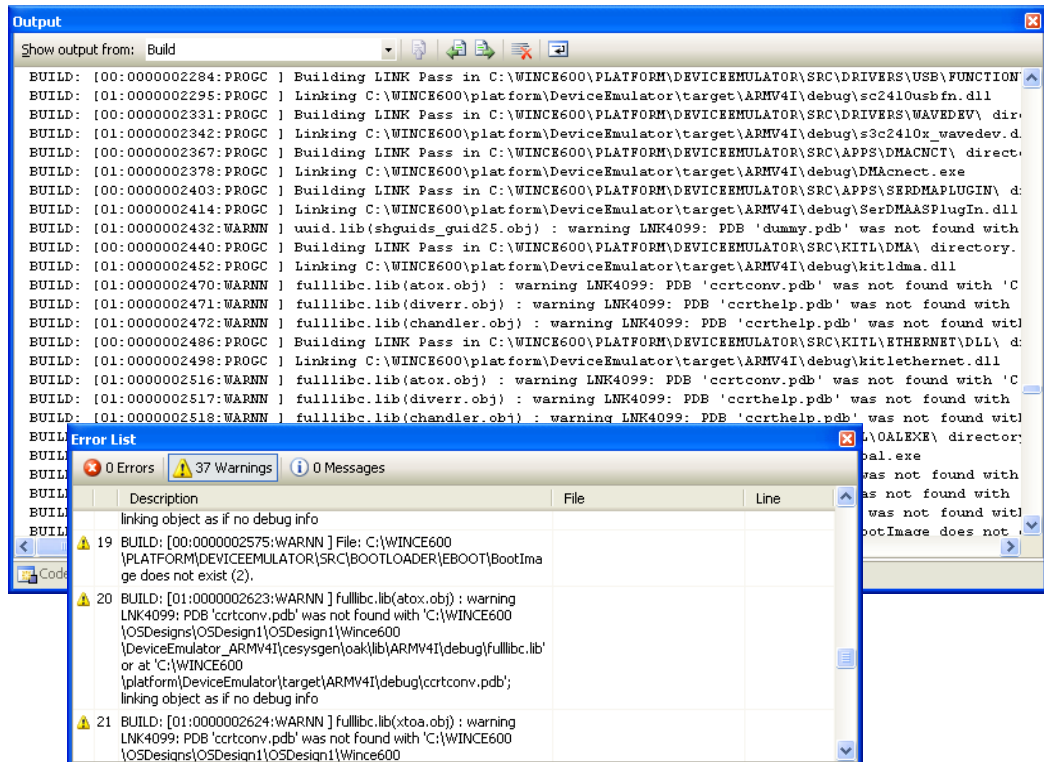


Figure 2-6 Output window and Error List window with build information in Visual Studio

Specifically, you can find the following information in the build log files:

- **Build.log** Contains information about the individual commands issued within each phase during the build process. This information is useful for analyzing both the build process in general, and build errors in particular.

- **Build.wrn** Contains information about warnings generated during the build process. If possible, try to eliminate or at least identify the reasons for the warnings. The information in Build.wrn is also included in Build.log.
- **Build.err** Contains specific information about build errors encountered during the build process. This information is also available with additional details in Build.log. This file is created only when an error occurs.

NOTE Identifying the build step

The build system keeps track of skipped and entered build phases in the Build.log file. For example, the entry *CEBUILD: Skipping directly to SYSGEN phase* indicates that the build system skipped the Compile phase for a component. You can determine where the Sysgen phase begins, how the build process transitions from SYSGEN to BUILD, and how BUILD eventually leads to MAKEIMG.

Troubleshooting Build Issues

While analyzing build log files can give you great insight into the build process in general, it is most useful when troubleshooting build errors. If an error message is related to a source code file, you can jump to the relevant line of code by double-clicking the message entry in the Error List window. However, not all build errors are related to source code. Linker errors due to missing library references, sysgen errors due to missing component files, copy errors due to exhausted disk capacities, and make run-time image errors due to incorrect settings in run-time image configuration files can also cause a build process to fail.

Errors during the Sysgen Phase

Sysgen errors are generally the result of missing files. The Build.log file might provide detailed information about the reason. Components that you recently added to or removed from an OS design can cause this type of error if the required dependencies are unavailable. To diagnose a Sysgen error, it is a good idea to verify all changes related to catalog items and their dependencies. Also note that some components require you to perform a clean Sysgen build instead of a regular Sysgen cycle. Typically, you should not use the Clean Sysgen command because performing a clean Sysgen in Release or Debug build configuration requires you to perform a regular Sysgen in the other build configuration as well. However, when adding or removing catalog items and encountering Sysgen build errors afterward, during the next regular Sysgen, you might have to perform a clean Sysgen build to solve the issue.

Errors during the Build Phase

Build errors are typically caused by compiler errors or linker errors. Compiler errors are syntax errors, missing or illegal parameters in function calls, divisions by zero and similar issues that prevent the compiler from generating valid binary code. By double-clicking a compiler error, you can jump to the critical line of code. Keep in mind, however, that compiler errors can be the results of other compiler errors. For example, an incorrect variable declaration can cause numerous compiler errors if the variable is used in many places. It is generally a good idea to start at the top of the error list, fix the code, and recompile. Even small code changes can often eliminate a very large number of errors from the list.

Linker errors are harder to troubleshoot than compiler errors. They are typically the result of missing or incompatible libraries. Incorrectly implemented APIs can also result in linker errors if the linker cannot resolve external references to exported DLL functions. Another common cause has its root in incorrectly initialized environment variables. Build files, specifically the Sources file, use environment variables instead of hard-coded directory names to point to referenced libraries. If these environment variables are not set, the linker will not be able to locate the libraries. For example, %_WINCEROOT% must point to C:\Wince600 if you installed Windows Embedded CE in the default configuration and %_FLATRELEASEDIR% must point to the current release directory. To verify the values of environment variables, open the Build menu in Visual Studio and select Open Release Directory in Build Window, and then at the command prompt use the **set** command with or without an environment variable, such as **set _winceroot**. Running the **set** command without parameters displays all environment variables, but be aware that this list is long.

Errors during the Release Copy Phase

Buildrel errors encountered during the Release Copy phase are generally a sign of inadequate hard drive space. During the Release Copy phase, the build system copies files to the release directory. It might be necessary to free up hard drive space or place the OS design folder on a different drive. Make sure that the new path to the OS design folder contains no spaces because spaces in the path or in the OS design name cause errors during the build process.

Errors during the Make Run-Time Image Phase

Errors encountered during this final phase in the build process generally result from missing files. This can happen if a component failed to build in an earlier step, but the build process nevertheless continued to proceed to the Make Run-time Image phase. Syntax errors in .reg files or .bib files can lead to this situation when the build system is unable to create the Reginit.ini file or Ce.bib file. Makeimg.exe calls the FMerge tool (FMerge.exe) during the build process to create these files, and if this fails, such as due to incorrect conditional statements, you encounter a make-image error. Another possible error is Error: Image Exceeds (*X*), which means the image is larger than the maximum possible size specified in Config.bib.

Lesson Summary

Platform Builder for Windows Embedded CE 6.0 R2 integrates with the build-logging system of Visual Studio 2005 to provide you with convenient access to status information, warnings, and error messages generated during the build process and tracked in Build.log, Build.wrn, and Build.err files. Depending on how you start the build process in Visual Studio, these files reside either in the %_WINCEROOT% folder or in a subproject directory, yet the actual location of the files is not important because you can analyze the content from these files directly in the Output window and the Error List window in Visual Studio. It is not necessary to open these files in Notepad or another text editor.

By analyzing build log files, you can gain a better understanding of the build process in general and build issues in particular. Typical build issues you might encounter occasionally are compiler errors, linker errors, Sysgen errors, build errors, and other errors generated during the Release Copy and Make Run-time Image phases. If a build error is related directly to a line in a source code file, you can double-click the message entry in the Error List window, and Visual Studio automatically opens the source-code file and jumps to the critical line. Other issues, such as buildrel errors due to inadequate hard drive space, require you to perform troubleshooting steps outside of the Visual Studio IDE.

Lesson 4: Deploying a Run-Time Image on a Target Platform

Having solved all build issues and successfully generated a run-time image, you are ready to deploy Windows Embedded CE on the target device. There are several ways to accomplish this task. The method you choose depends on the startup process you use to load Windows Embedded CE on the target device. There are several ways you can start a Windows Embedded CE 6.0 run-time image. You can start an image directly from ROM, in which case you must deploy the run-time image on the target device by using a ROM tool. You can also use a boot loader, and then either download the run-time image every time the device starts or store the image in persistent memory for reuse. Windows Embedded CE 6.0 R2 comes with generic boot-loader code that you can customize according to your specific needs. It is also straightforward to implement a third-party boot loader. Essentially, Windows Embedded CE can accommodate almost any start environment, and makes it easy to download new run-time images quickly and conveniently during the development cycle and for release to the end user.

After this lesson, you will be able to:

- Decide how to deploy a run-time image on a target device.
- Configure Platform Builder to select the correct deployment layer.

Estimated lesson time: 15 minutes.

Choosing a Deployment Method

In order to deploy a run-time image, you must establish a connection to the target device. This requires you to configure several communication parameters that determine how Platform Builder communicates with the device.

The Core Connectivity infrastructure of Windows Embedded CE supports various download methods and transport mechanisms to accommodate hardware platforms with varying communication capabilities. To define the communication parameters for your target device, open the Target menu in Visual Studio and select Connectivity Options, which displays the Target Device Connectivity Options dialog box. By default, Platform Builder provides a target device named CE Device in the Target Device list box, as illustrated in Figure 2–7, but you can also create additional devices with unique names by clicking the Add Device link.

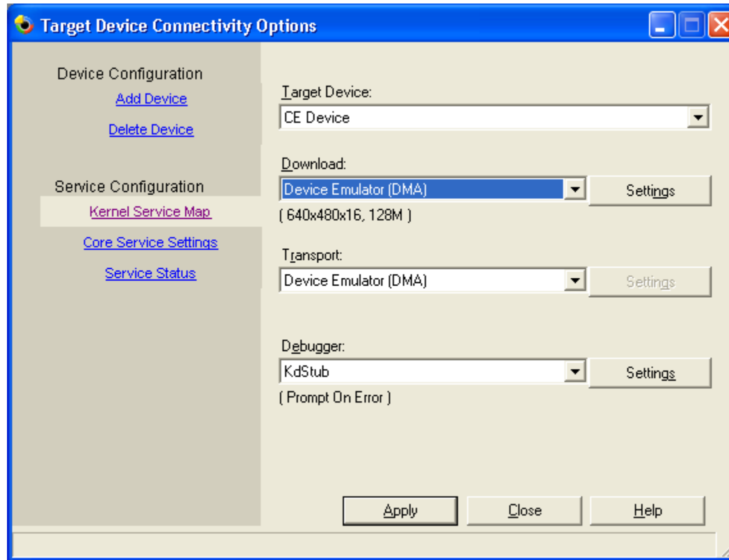


Figure 2-7 Target Device Connectivity Options window

Download Layer

The Download list box and associated Settings button enable you to configure the download service used for downloading the run-time image to your target device. The Core Connectivity infrastructure supports the following download layers for deploying a run-time image:

- **Ethernet** Downloads the run-time image over an Ethernet connection. Use the Settings button to configure the Ethernet download service. The development workstation and the target device must be on the same subnet; otherwise, you cannot connect to the target device.
- **Serial** Downloads the run-time image over an RS232 connection. Use the Settings button to configure the port, baud rate, and other serial communication parameters.
- **Device Emulator (DMA)** Downloads the run-time image to a device emulator through Direct Memory Access (DMA). Use the Settings button to configure the device emulator.
- **USB** Downloads the run-time image over a Universal Serial Bus (USB) connection. There are no settings to configure.

- **Image Update** Updates the image in the device's flash memory. There are no settings to configure.
- **None** Select this option if you do not want to download or update the run-time image.

Transport Layer

After transferring the run-time image to the remote device, you can attach to the device if you enabled Kernel Independent Transport Layer (KITL) in the OS design. In general, the selected kernel transport service should match the download service that you selected in the Download list box. The Core Connectivity infrastructure supports the following transport layer options:

- **Ethernet** Communicates with the target device over an Ethernet connection. The connection uses the same settings as the download service.
- **Serial** Communicates with the target device over an RS232 connection. The connection uses the same settings as the download service.
- **Device Emulator (DMA)** Communicates with a device emulator through DMA.
- **USB** Communicates with the target device over a USB connection.
- **None** Disables communication with the target device.

Debugger Options

If you enabled support for one or more debuggers in the OS design, the debugger names will appear as options in the Debugger list box. By default, the following debugger options are available:

- **Sample Device Emulator eXDI2 Driver** This is a sample Extensible Resource Identifier (XRI) Data Interchange (XDI) driver included in Windows Embedded CE 6.0 R2. XDI is a standard hardware-debugging interface.
- **KdStub** This is the Kernel Debugger. KdStub stands for kernel debugger stub, which instructs Platform Builder and Visual Studio to use the software debugger.
- **CE Dump File Reader** If you added the Error Report Generator catalog item to your OS design, you can use this option for postmortem debugging.
- **None** Select this option if you do not want to use a debugger.

Attaching to a Device

Having configured the device connection, you are ready to transfer the run-time image to the target device or device emulator by using the Core Connectivity infrastructure. This is accomplished in Visual Studio 2005 by using the Attach Device command that is available on the Target menu. Even if you do not plan to use KITL or the Core Connectivity infrastructure for debugging, you must attach to the device so that Platform Builder can download the run-time image.

Following the image download, the start process commences, KITL becomes active if enabled on the target device, and you can use the Kernel Debugger to follow the start process, and debug operating system components and application processes. By using KITL, you can also exploit remote tools available in Visual Studio with Platform Builder on the Target menu, such as File Viewer to interact with the device's file system, Registry Editor to access the device's registry settings, Performance Monitor to analyze resource utilization and response times, and Kernel Tracker and other remote tools to view detailed information on the running system. You can find more information about system debugging in Chapter 4, "Debugging and Testing the System."

Lesson Summary

Windows Embedded CE supports run-time image deployment over a variety of device connections to accommodate hardware platforms with varying requirements and capabilities, including Ethernet connections, serial connections, DMA, and USB connections. For example, DMA is the right choice if you want to deploy CE 6.0 R2 on a Device Emulator. You only need to configure the communication parameters and you are ready to deploy Windows Embedded CE by clicking the Attach Device command on the Target menu in Visual Studio 2005 with Platform Builder.

EXAM TIP

To pass the certification exam, you must be familiar with the varioust ways to deploy a Windows Embedded CE run-time image. In particular, make sure you know how to deploy a run-time image for a Device Emulator.

Lab 2: Building and Deploying a Run-Time Image

In this lab, you build and deploy an OS design based on the Device Emulator BSP, analyze the build information in the Visual Studio Output window to identify the start of the various build phases, and then configure a connection to a target device in order to download the run-time image. To demonstrate how to customize a target device, you modify the Device Emulator configuration to support a larger screen resolution and to enable network communication. In a final step, you download the run-time image and attach to the target device with the Kernel Debugger, so you can examine the Windows Embedded CE start process in detail. To create the initial OS design in Visual Studio, follow the procedures outlined in Lab 1, "Creating, Configuring, and Building an OS Design."

NOTE Detailed step-by-step instructions

To help you successfully master the procedures presented in this Lab, see the document "Detailed Step-by-Step Instructions for Lab 2" in the companion material for this book.

Build a Run-Time Image for an OS Design

1. After completing Lab 1, select Sysgen under Advanced Build Commands on the Build menu in Visual Studio, as illustrated in Figure 2–8. Alternatively, you can select Build Solution under the Build menu, which will perform a build starting with the Sysgen step.

TIP Sysgen operations

Sysgen operations can take up to 30 minutes to complete. To save time, do not run Sysgen every time you change the OS design. Instead, run Sysgen after adding and removing all desired components.

2. Follow the build process in the Output window. Examine the build information to identify the SYSGEN, BUILD, BUILDREL, and MAKEIMG steps. You can press Ctrl+F to display the Find And Replace dialog box, and then search for the following text to identify the start of these phases:
   a. **Starting Sysgen Phase For Project** The SYSGEN steps start.
   b. **Build Started With Parameters** The BUILD steps start.
   c. **C:\WINCE600\Build.log** The BUILDREL steps start.
   d. **BLDDEMO: Calling Makeimg—Please Wait** The MAKEIMG steps starts.

3. Open the C:\Wince600 folder in Windows Explorer. Verify that Build.* files exist.
4. Open the Build.* files in a text editor, such as Notepad, and examine the content.

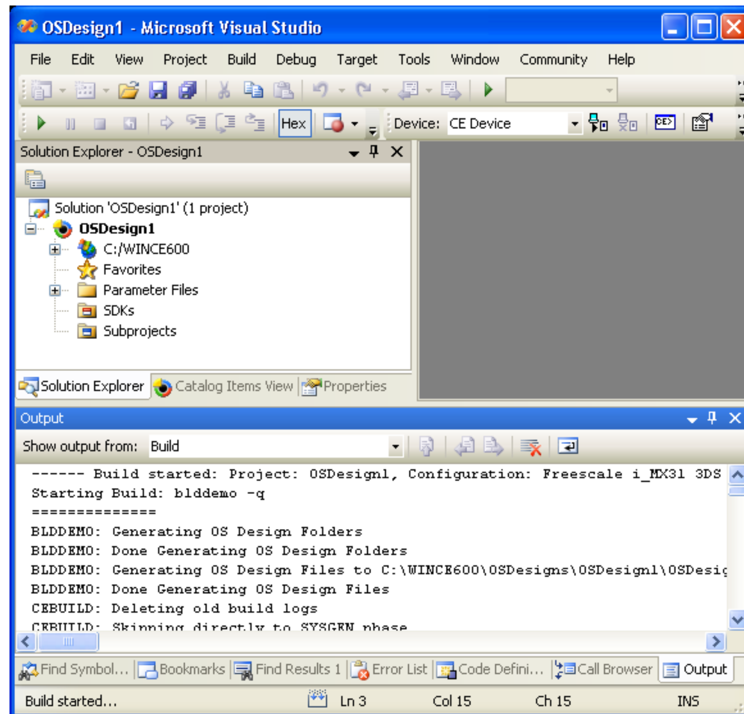


Figure 2-8 Building an OS design

Configure Connectivity Options

1. In Visual Studio, open the Target menu and select Connectivity Options to display the Target Device Connectivity Options dialog box.
2. Verify that CE Device is selected in the Target Device list box.
3. Select Device Emulator (DMA) from the Download list box.
4. Select Device Emulator (DMA) from the Transport list box.
5. Select KdStub from the Debugger list box, as illustrated in Figure 2–9.

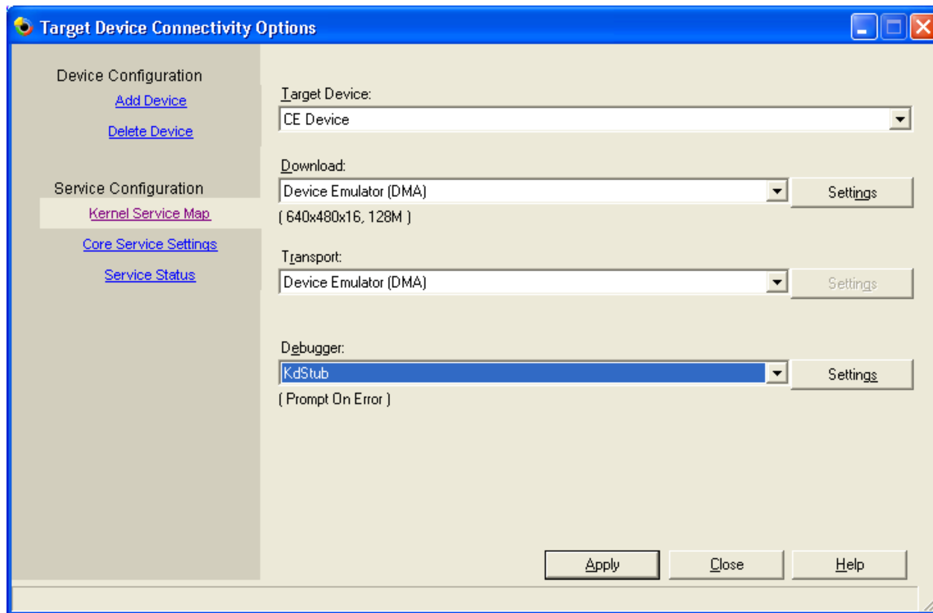


Figure 2-9 Setting Target Device Connectivity Options

Change the Emulator Configuration

1. Next to the Download list box, click the Settings button.
2. In the Emulator Properties dialog box, switch to the Display tab.
3. Change the Screen Width to **640** pixels and the Screen Height to **480** pixels.
4. Switch to the Network tab.
5. Select the Enable NE2000 PCMCIA Network Adapter And Bind To check box, then select the Connected Network Card option from the list box, as illustrated in Figure 2–10, and then click OK.
6. Click Apply to save the new device configuration.
7. Click Close to close the Target Device Connectivity Options dialog box.

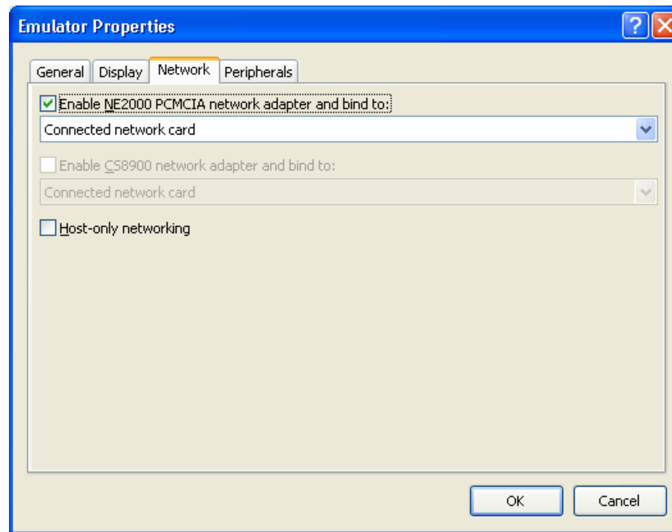


Figure 2-10 Device Emulator network options

Test a Run-Time Image on the Device Emulator

1. In Visual Studio, open the Target menu, and then click Attach Device.
2. Verify that Visual Studio downloads the run-time image to the target device. The download can take several minutes to complete.
3. Follow the debug messages during the start process in the Visual Studio Output window.
4. Wait until Windows Embedded CE has completed the start process, and then interact with the Device Emulator and test the features of your OS design, as illustrated in Figure 2–11.

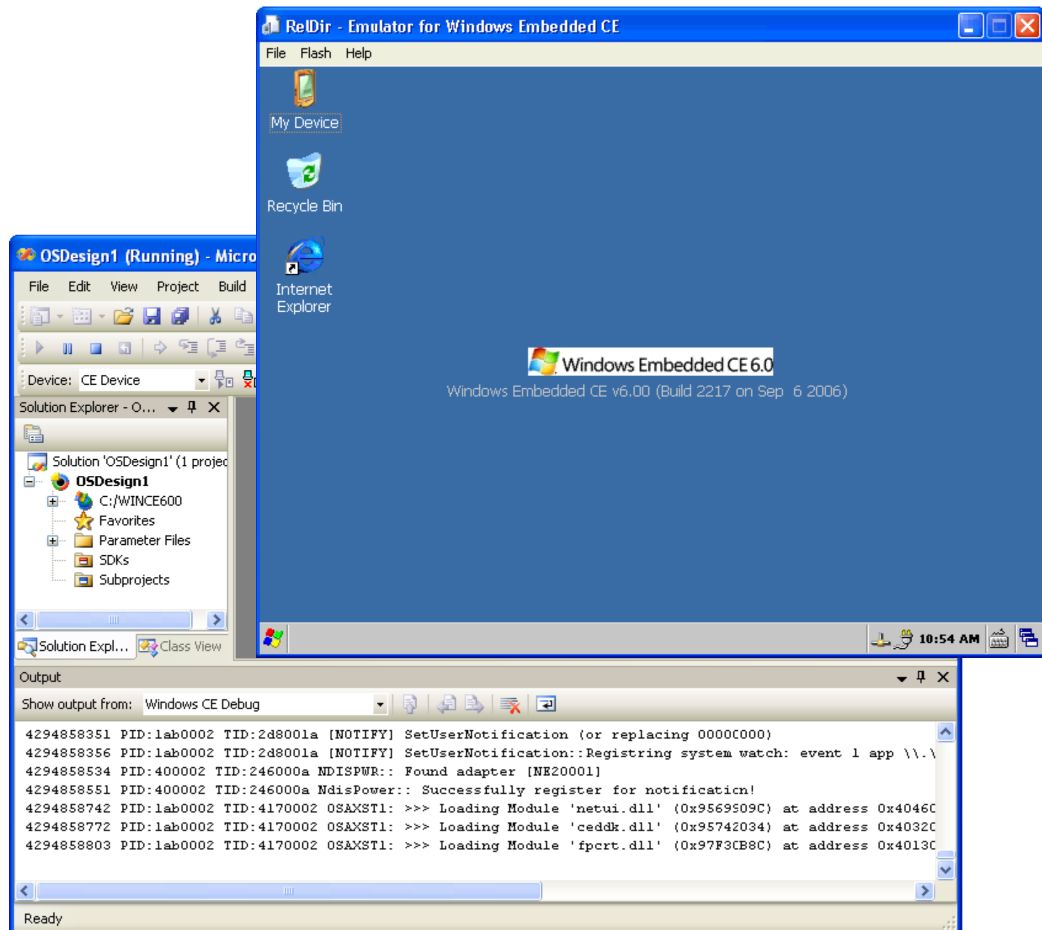


Figure 2-11 Windows Embedded CE device emulator

Chapter Review

The Windows Embedded CE build process includes several phases and relies on a variety of build and run-time image configuration files to compile the source code and create the run-time image. It includes a compile phase to generate .exe files, static libraries, DLLs, and binary resource (.res) files for the BSP and subprojects; a Sysgen phase to filter and copy source code based on SYSGEN variables from the Public folder for catalog items selected in the OS design, and create a set of run-time image configuration files; a Release Copy phase to copy the files from the BSP and subprojects required to build the run-time image into the release directory; and finally a Make Run-time Image phase to create the run-time image from the content in the release directory according to the setting specified in .bib, .reg, .db, and .dat files.

You can examine the build process if you analyze the information that Platform Builder generates and tracks in Build.log, Build.wrn, and Build.err files. The Build.log file contains detailed information about every build command issued in each build phase. Build.wrn and Build.err contain the same information, but filtered for warnings and errors encountered during the build process. You do not need to open these text files directly in Notepad. It is more convenient to work with build status information and error messages in Visual Studio. The Output window and the Error List window provide convenient access.

Build errors can occur for a variety of reasons. The most common causes are compiler and linker errors. For example, running build commands in an incorrectly initialized build environment will lead to linker errors when environment variables that identify library directories in the Sources file point to invalid locations. Other important build configuration files, such as Dirs files and Makefile.def, can also rely on SYSGEN variables and environment variables in conditional statements and in directory paths.

Having successfully generated a run-time image, you can deploy Windows Embedded CE on a target device. This requires you to configure a device connection based on the Core Connectivity infrastructure. The final deployment step is simply to click the Attach Device command that you can find on the Target menu in Visual Studio with Platform Builder for Windows Embedded CE 6.0 R2.

It is important that you are familiar with the following configuration files, which control the Windows Embedded CE build process:

- **Binary image builder (.bib) files** Configure the memory layout and determine the files included in the run-time image.

- **Registry (.reg) files** Initialize the system registry on the target device.
- **Database (.db) files** Set up the default object store.
- **File system (.dat) files** Initialize the RAM file system layout at start time.
- **Dirs files** Determine which directories to include in the build process.
- **Sources files** Define preprocessing directives, commands, macros, and other processing instructions for the compiler and linker. Take the place of Makefile files in the Visual Studio with Platform Builder IDE.
- **Makefile files** Reference the default Makefile.def file and should not be edited.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Sysgen
- Buildrel
- Flat release directory
- Connectivity options
- KITL

Suggested Practice

To help you successfully master the exam objectives presented in this chapter, complete the following tasks:

Start the Build Process from the Command Line

To increase your understanding of the Windows Embedded CE build processes, perform the following steps:

1. **Sysgen process** Run **sysgen –q** from the command line after setting an environment variable in a catalog item.
2. **Build process** Open a Command Prompt window, change into the current BSP folder (%_TARGETPLATROOT%) and run **build-c** with and without setting the WINCEREL environment variable to 1 (**set WINCEREL=1**). Check the content of the %_FLATRELEASEDIR% folder before and after the build.

Deploy Run-Time Images

Deploy a Windows Embedded CE run-time image to a Device Emulator by using different download, transport, and debugging settings for the target device in Platform Builder.

Clone a Public Catalog Component Manually

Clone a component from the %_WINCEROOT%\Public folder by copying the source files to a BSP, as explained in Chapter 1, "Customizing the Operating System Design." Next, run **sysgen_capture** to create a Sources file that defines the component dependencies. Modify the new Sources file to build the component as part of your BSP. For detailed step-by-step information to accomplish this advanced development task, read the section "Using the Sysgen Capture Tool" in the Platform Builder for Microsoft Windows Embedded CE product documentation available on the Microsoft MSDN® website at *http://msdn2.microsoft.com/en-us/library/aa924385.aspx*.

Chapter 3

Performing System Programming

System performance is critical for user productivity. It directly influences the user's perception of a device. In fact, it is not uncommon for users to judge the usefulness of a device based on the performance of the system and the look and feel of the user interface. By providing too complex of an interface, you can confuse users and open your device to potential security risks or unexpected user manipulations. By using the incorrect APIs, or incorrect applications architecture in a multithreaded environment, you may significantly impact performance. Performance optimization and system customization are real challenges for firmware providers. This chapter discusses the tools and highlights best practices to achieve optimal system response times on target devices.

Exam objectives in this chapter:

- Monitoring and optimizing system performance
- Implementing system applications
- Programming with threads and thread synchronization objects
- Implementing exception handling in drivers and applications
- Supporting power management at the system level

Before You Begin

To complete the lessons in this chapter, you must have the following:

- A thorough understanding of real-time systems design concepts, such as scheduler functionality in an operating system, interrupts, and timers.
- Basic knowledge of multithreaded programming, including synchronization objects.
- A development computer with Microsoft® Visual Studio® 2005 Service Pack 1 and Platform Builder for Microsoft Windows® Embedded CE 6.0 installed.

Lesson 1: Monitoring and Optimizing System Performance

Performance monitoring and optimization are important tasks in the development of small-footprint devices. The need for optimized system performance remains critical because of an ever-growing number of increasingly complex applications and the requirement for intuitive and therefore resource-intensive user interfaces. Performance optimization requires firmware architects and software developers to constrain resource consumption within their system components and applications so that other components and applications can use the available resources. Whether developing device drivers or user applications, optimized processing algorithms can help to save processor cycles, and efficient data structures can preserve memory. Tools exist at all system levels to identify performance issues within and between drivers, applications, and other components.

After this lesson, you will be able to:

- Identify the latency of an interrupt service routine (ISR).
- Improve the performance of a Windows Embedded CE system.
- Log and analyze system performance information.

Estimated lesson time: 20 minutes.

Real-Time Performance

Drivers, applications, and OEM adaptation layer (OAL) code impact system and real-time performance. Although Windows Embedded CE may be used in real-time and non-real-time configurations, it is important to note that using non-real-time components and applications can decrease system performance in a real-time operating system (OS) configuration. For example, you should keep in mind that demand paging, device input/output (I/O), and power management are not designed for real-time devices. Use these features carefully.

Demand Paging

Demand paging facilitates memory sharing between multiple processes on devices with limited RAM capacity. When demand paging is enabled, Windows Embedded CE discards and removes memory pages from active processes under low-memory conditions. However, to keep the code of all active processes in memory, disable

demand paging for the entire operating system or for a specific module, such as a dynamic-link library (DLL) or device driver.

You can disable demand paging by using the following methods:

- **Operating system** Edit the Config.bib file and set the ROMFLAGS option in the CONFIG section.
- **DLLs** Use the LoadDriver function instead of the LoadLibrary function to load the DLL into memory.
- **Device drivers** Add the DEVFLAGS_LOADLIBRARY flag to the Flags registry entry for the driver. This flag causes Device Manager to use the LoadLibrary function instead of the LoadDriver function to load the driver.

Windows Embedded CE allocates and uses memory as usual, but does not discard it automatically when you disable demand paging.

System Timer

The system timer is a hardware timer that generates system ticks at a frequency of one tick per millisecond. The system scheduler uses this timer to determine which threads should run at what time on the system. A thread is the smallest executable unit within a process that is allocated processor time to execute instructions in the operating system. You can stop a thread for an amount of time by using the Sleep function. The minimum value that you can pass to the Sleep function is 1 (**Sleep(1)**), which stops the thread for approximately 1 millisecond. However, the sleep time is not exactly 1 millisecond because the sleep time includes the current system timer tick plus the remainder of the previous tick. The sleep time is also linked to the priority of the thread. The thread priority determines the order in which the operating system schedules the threads to run on the processor. For those reasons, you should not use the Sleep function if you need accurate timers for real-time applications. Use dedicated timers with interrupts or multimedia timers for real-time purposes.

Power Management

Power management can affect system performance. When the processor enters the Idle power state, any interrupt generated by a peripheral or the system scheduler causes the processor to exit this state, restore the previous context, and invoke the scheduler. Power context switching is a time-consuming process. For detailed information about the power management features of Windows Embedded CE, see the section "Power Management" in the Windows Embedded CE 6.0 documentation

available on the Microsoft MSDN® website at *http://msdn2.microsoft.com/en-us/library/aa923906.aspx*.

System Memory

The kernel allocates and manages system memory for heaps, processes, critical sections, mutexes, events, and semaphores. Yet, the kernel does not completely free the system memory when releasing these kernel objects. Instead, the kernel holds on to the system memory to reuse it for the next allocation. Because it is faster to reuse allocated memory, the kernel initializes the system memory pool during the startup process and allocates further memory only if no more memory is available in the pool. System performance can decrease depending how processes use virtual memory, heap objects, and the stack.

Non-Real-Time APIs

When calling system APIs, or Graphical Windows Event System (GWES) APIs, be aware that some APIs rely on non-real-time features, such as for window drawing. Forwarding calls to non-real-time APIs may dramatically decrease system performance. Consequently, you should make sure that your APIs in real-time applications are real-time compliant. Other APIs, such as ones used for accessing a file system or hardware, can have an impact on performance because these APIs may use blocking mechanisms, such as mutexes or critical sections, to protect resources.

NOTE Non-real-time APIs

Non-real-time APIs can have a measurable impact on real-time performances and, unfortunately, the Win32® API documentation provides little detail on real-time issues. Practical experience and performance testing can help you choose the right functions.

Real-Time Performance Measurement Tools

Windows Embedded CE includes many performance monitoring and troubleshooting tools that can be used to measure the impact of Win32 APIs on system performance. These tools are helpful when identifying inefficient memory use, such as an application not releasing the system memory it allocates.

The following Windows Embedded CE tools are particularly useful to measure the real-time performance of your system components and applications:

- **ILTiming** Measures Interrupt Service Routine (ISR) and Interrupt Service Thread (IST) latencies.
- **OSBench** Measures system performance by tracking the time the kernel spends managing kernel objects.
- **Remote Performance Monitor** Measures system performance, including memory usage, network throughput, and other aspects.

Interrupt Latency Timing (ILTiming)

The ILTiming tool is particularly useful for Original Equipment Manufacturers (OEMs) who want to measure ISR and IST latencies. Specifically, ILTiming enables you to measure the time it takes to invoke an ISR after an interrupt occurred (ISR latency) and the time between when the ISR exits and the IST actually starts (IST latency). This tool uses a system hardware tick timer by default, but it is also possible to use alternative timers (high-performance counters).

NOTE Hardware timer restrictions

Not all hardware platforms provide the required timer support for the ILTiming tool.

The ILTiming tool relies on the OALTimerIntrHandler function in the OAL to implement the ISR for managing the system tick interrupt. The timer interrupt handler stores the current time and returns a SYSINTR_TIMING interrupt event, which an ILTiming application thread waits to receive. This thread is the IST. The time elapsed between the reception of the interrupt in the ISR and the reception of the SYSINTR_TIMING event in the IST is the IST latency that the ILTiming tool measures.

You can find the ILTiming tool's source code in the %_WINCEROOT%\Public \Common\Oak\Utils folder on your development computer if you have installed Microsoft Platform Builder for Windows Embedded CE 6.0 R2. The ILTiming tool supports several command-line parameters that you can use to set the IST priority and type according to the following syntax:

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

Table 3–1 describes the individual ILTiming command-line parameters in more detail.

Table 3-1 ILTiming parameters

Command-Line Parameter	Description
-i0	No idle thread. This is equivalent to using the -ni parameter.
-i1	One thread spinning without performing any actual processing.
-i2	One thread spinning, calling SetThreadPriority (THREAD_PRIORITY_IDLE).
-i3	Two threads alternating SetEvent and WaitForSingleObject with a 10-second timeout.
-i4	Two threads alternating SetEvent and WaitForSingleObject with an infinite timeout.
-i5	One thread spinning, calling either VirtualAlloc (64 KB), VirtualFree, or both. Designed to flush the cache and the translation look-aside buffer (TLB).
-p *priority*	Specifies the IST priority (zero through 255). The default setting is zero for highest priority.
-ni	Specifies no idle priority thread. The default setting is equal to the number of idle priority thread spins. This is equivalent to using the -i0 parameter.
-t *interval*	Specifies the SYSINTR_TIMING timing interval, with clock ticks in milliseconds. The default setting is five.
-n *interrupt*	Specifies the number of interrupts. Using this parameter you can specify how long the test will run. The default setting is 10.
-all	Specifies to output all data. The default setting is to output the summary only.
-o *file_name*	Specifies to output to file. The default setting is to output to the debugger message window.

NOTE Idle threads

ILTiming may create idle threads (command-line parameters: -i1, -i2, -i3, and -i4) to generate activity on the system. This enables the kernel to be in a non-preemptive kernel call that must be finished before handling the IST. It can be useful to enable idle threads in background tasks.

Operating System Benchmark (OSBench)

The OSBench tool can help you measure system performance by identifying the time that the kernel spends managing kernel objects. Based on the scheduler, OSBench collects timing measurements by means of scheduler performance-timing tests. A scheduler performance-timing test measures how much time basic kernel operations, such as thread synchronization, require.

OSBench enables you to track timing information for the following kernel operations:

- Acquiring or releasing a critical section.
- Waiting for or signaling an event.
- Creating a semaphore or mutex.
- Yielding a thread.
- Calling system APIs.

NOTE OSBench test

To identify performance issues in different system configurations, use OSBench in conjunction with a stress test suite, such as the Microsoft Windows CE Test Kit (CETK).

The OSBench tool supports several command-line parameters that you can use according to the following syntax to collect timing samples for kernel operations:

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

Table 3–2 describes the individual OSBench command-line parameters in more detail.

Check out the OSBench source code to identify the test content. You can find the source code at the following locations:

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

Test results are by default sent to the debug output, but can be redirected to a CSV file.

Table 3-2 OSBench parameters

Command-Line Parameter	Description
-all	Run all tests (default: run only those specified by -t option): TestId 0: CriticalSections. TestId 1: Event set-wakeup. TestId 2: Semaphore release-acquire. TestId 3: Mutex. TestId 4: Voluntary yield. TestId 5: PSL API call overhead. TestId 6: Interlocked API's (decrement, increment, testexchange, exchange).
-t *test_case*	ID of test to run (need separate -t for each test).
-list	List test ID's with descriptions.
-v	Verbose: show extra measurement details.
-n *number*	Number of samples per test (default =100).
-m *address*	Virtual address to write marker values to (default = <none>).
-o *file_name*	Output to comma-separated values (CSV) file (default: output only to debug).

NOTE OSBench requirements

The OSBench tool uses system timers. The OAL must therefore support the QueryPerformanceCounter and QueryPerformanceFrequency functions initialized in the OEMInit function.

Remote Performance Monitor

The Remote Performance Monitor application can track the real-time performance of the operating system as well as memory usage, network latencies, and other elements. Each system element is associated with a set of indicators that provide information on usage, queue length, and delays. Remote Performance Monitor can analyze log files generated on a target device.

As the name suggests, the Remote Performance Monitor application is a remote tool. The application monitors devices both under development and out in the field, as long as you have a way to connect to the device and deploy the application.

The Remote Performance Monitor monitors the following objects:

- Remote Access Server (RAS).
- Internet Control Message Protocol (ICMP).
- Transport Control Protocol (TCP).
- Internet Protocol (IP).
- User Datagram Protocol (UDP).
- Memory.
- Battery.
- System.
- Process.
- Thread.

This list is extended by implementing your own Remote Performance Monitor extension DLL. For sample code, look in the % COMMONPROGRAMFILES% \Microsoft Shared\Windows CE Tools\Platman\Sdk\WCE600\Samples\CEPerf folder.

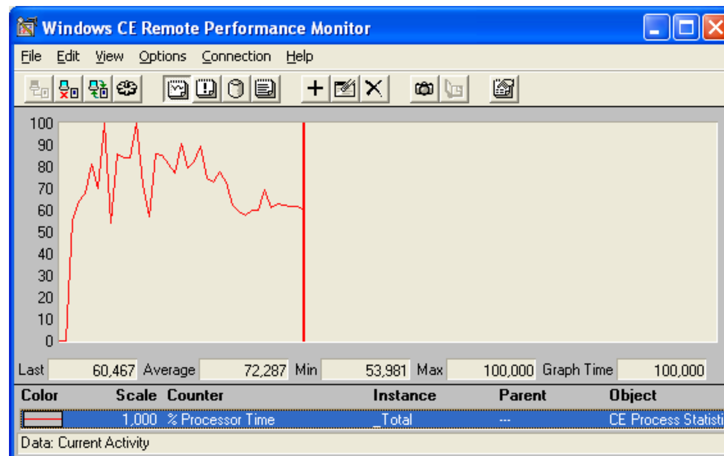


Figure 3-1 A performance chart in Remote Performance Monitor

Similar to the Performance tool on a Windows workstation, Remote Performance Monitor can create performance charts, configure alerts triggered at specified thresholds, write raw log files, and compile performance reports based on the performance objects available on the target device. Figure 3–1 shows a performance chart example.

Hardware Validation

ILTiming tool, OSBench, and Remote Performance Monitor cover most performance monitoring needs. However, some cases may require other methods of gathering system performance information. For example, if you want to obtain exact interrupt latency timings, or if your hardware platform does not provide the required timer support for the ILTiming tool, you must use hardware–based performance measuring methods based on the General Purpose Input/Output (GPIO) interface of the processor and a waveform generator.

By using a waveform generator on a GPIO, it is possible to generate interrupts that are handled through ISRs and ISTs. These ISRs and ISTs then use another GPIO to generate a waveform in response to the received interrupt. The time elapsed between the two waveforms—the input waveform from the generator and the output waveform from the ISR or IST—is the latency time of the interrupt.

Lesson Summary

Windows Embedded CE provides many tools that can be employed in a development environment to measure the system performance and validate real-time device performance. The ILTiming tool is useful for measuring interrupt latencies. The OSBench tool enables you to analyze how the kernel manages system objects. Remote Performance Monitor provides the means to gather performance and statistical data in charts, logs, as well as report on devices under development and out in the field. Remote Performance Monitor has the ability to generate alerts based on configurable performance thresholds. Beyond the capabilities of these tools, you have the option to use hardware monitoring for latency and performance-measurement purposes.

Lesson 2: Implementing System Applications

As discussed in Chapter 1 “Customizing the Operating System Design”, Windows Embedded CE acts as a componentized operating system and a development platform for a wide variety of small-footprint devices. These range from devices with restricted access for dedicated tasks, such as mission-critical industrial controllers, to open platforms offering access to the complete operating system, including all settings and applications, such as personal digital assistant (PDA). However, practically all Windows Embedded CE devices require system applications to provide an interface to the user.

After this lesson, you will be able to:

- Launch an application at startup.
- Replace the default shell.
- Customize the shell.

Estimated lesson time: 25 minutes.

System Application Overview

Developers distinguish between system applications and user applications to emphasize that these applications have different purposes. In the context of Windows Embedded CE devices, the term *system application* generally refers to an application that provides an interface between the user and the system. In contrast, a *user application* is a program that provides an interface between the user and application-specific logic and data. Like user applications, system applications can implement a graphical or command-line interface, but system applications are typically started automatically as part of the operating system.

Start an Application at Startup

You can configure applications to start automatically as part of the Windows Embedded CE initialization process. This feature can be set in several ways, depending on whether you want to run the applications before or after Windows Embedded CE loads the shell user interface (UI). One method is to manipulate several registry settings that control the application startup behavior. Another common method is to place a shortcut to the application in the Startup folder so that the standard shell can start the application.

HKEY_LOCAL_MACHINE\INIT Registry Key

The Windows Embedded CE registry includes several registry entries to start operating system components and applications at startup time, such as Device Manager and Graphical Windows Event System (GWES). These registry entries are located under the HKEY_LOCAL_MACHINE\INIT registry key, as illustrated in Figure 3-2. You can create additional entries at this location to run your own applications included in the run-time image without having to load and run these applications manually on your target device. Among other things, automatically starting an application can facilitate debugging activities during software development.

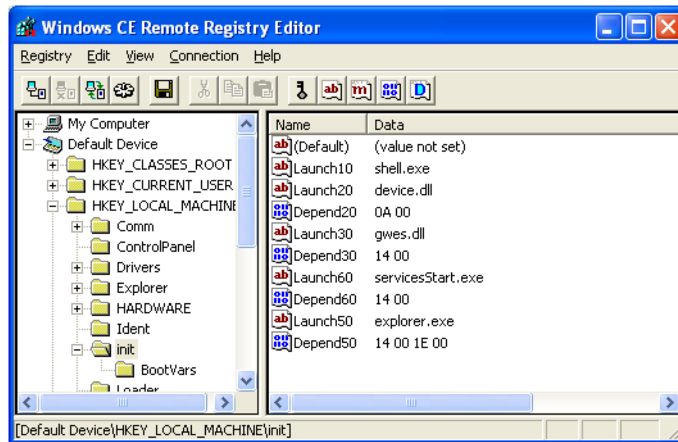


Figure 3-2 The HKEY_LOCAL_MACHINE\INIT registry key

Table 3-3 lists three examples of registry entries to start typical Windows Embedded CE components when the run-time image starts.

Table 3-3 Startup registry parameter examples

Location	HKEY_LOCAL_MACHINE\INIT		
Component	Device Manager	GWES	Explorer
Binary	Launch20= "Device.dll"	Launch30= "Gwes.dll"	Launch50= "Explorer.exe"
Dependencies	Depend20= hex:0a,00	Depend30= hex:14,00	Depend50= hex:14,00, 1e,00

Table 3-3 Startup registry parameter examples (Continued)

Location	HKEY_LOCAL_MACHINE\INIT
Description	The LaunchXX registry entry specifies the binary file of the application and the DependXX registry entry defines the dependencies between applications.

If you look at the Launch50 registry entry in Table 3–3, you can see that the Windows Embedded CE standard shell (Explorer.exe), will not run until process 0x14 (20) and process 0x1E (30) have started successfully, which happen to be Device Manager and GWES. The hexadecimal values in the DependXX entry refer to decimal launch numbers *XX*, specified in the name of the LaunchXX entries.

Implementing the SignalStarted API helps the kernel manage process dependencies between all applications registered under the HKEY_LOCAL_MACHINE\INIT registry key. The application can then use the SignalStarted function to inform the kernel that the application has started and initialization is complete, as illustrated in the following code snippet.

```
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;
    }
```

Dependency handling is straightforward. The kernel determines the launch number from the Launch registry entry, uses it as a sequence identifier, and passes it as a startup parameter in lpCmdLine to the WinMain entry point. The application performs any required initialization work and then informs the kernel that it has finished this part by calling the SignalStarted function. The call to the _wtol function in the SignalStarted code line performs a conversion of the launch number from a string to a long integer value because the SignalStarted function expects a DWORD parameter. For example, Device Manager must pass a SignalStarted value of 20 and GWES must pass a value of 30 back to the kernel for the kernel to start Explorer.exe.

The Startup Folder

If you are using the standard shell on your target device, you can drop the application or a shortcut to the application into the Windows\Startup folder of the device. Explorer.exe examines this folder and starts all found applications.

NOTE StartupProcessFolder function

Only use the Windows\Startup folder if your target device runs the Windows Embedded CE standard shell. If you are not using the standard shell, then create a custom launch application for the same purpose and initiate it at start time based on entries under the HKEY_LOCAL_MACHINE\INIT registry key. For sample code that demonstrates how to examine the Startup folder and launch the applications, find the Explorer.cpp file in the %_WINCEROOT%\Public\Shell\OAK\HPC\Explorer\Main folder. Look for a function called StartupProcessFolder and use it as a starting point for your own implementation.

The Windows Embedded CE standard shell can handle executable and shortcut files. Windows Embedded CE shortcut files differ from the shortcut files of Windows XP, but provide similar functionality. CE shortcut files are text files with an .lnk file-name extension. They contain the command-line parameters for the linked target according to the following syntax:

nn# command [optional parameters]

The placeholder *nn* stands for the number of characters followed by a pound sign (#), and the actual command, such as **27#\Windows\iexplore.exe -home** to start Internet Explorer® and open the home page. After creating and adding the desired .lnk file to the run-time image, edit the Platform.dat or Project.dat file to map the .lnk file to the Startup folder, similar to the following .dat file entry:

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

Chapter 2 covers these configuration tasks in more detail.

NOTE Startup folder restriction

The key advantage of the Startup folder is that the applications placed in this folder do not need to implement the SignalStarted API to inform the kernel that the initialization and start process completed successfully. However, this also implies that the operating system cannot manage dependencies between applications or enforce a specific startup sequence. The operating system starts all applications in the Startup folder concurrently.

Delayed Startup

Another interesting option to start applications automatically is to leverage the services host process (Services.exe). Although Windows Embedded CE does not include a full-featured Service Control Manager (SCM), it does include built-in services and also comes with a sample service called Svcstart that can be used to start applications.

Svcstart is particularly useful for applications with dependencies on system components and services that are not immediately available after the startup process finishes. For example, it might take a few seconds to obtain an Internet Protocol (IP) address from a Dynamic Host Configuration Protocol (DHCP) server for a network interface card (NIC) or initialize a file system. To accommodate these scenarios, the Svcstart service supports a Delay parameter that specifies the time to wait before starting an application. You can find the Svcstart sample code in the %_WINCEROOT%\Public\Servers\SDK\Samples\Services\Svcstart folder. Compile the sample code into Svcstart.dll, add this DLL to your run-time image, and then run the **sysgen -p servers svcstart** command to register the Svcstart service with the operating system. Load it by using Services.exe.

Table 3–4 lists the registry settings that the Svcstart service supports to start applications.

Table 3-4 Svcstart registry parameters

Location	HKEY_LOCAL_MACHINE\Software\Microsoft\Svcstart\1
Application Path	@="iexplore.exe"
Command-line Parameters	Args="-home"
Delay Time	Delay=dword:4000
Description	Starts the application with the specified command-line parameters after a delay time defined in milliseconds. See the Svcstart.cpp file for more details.

Windows Embedded CE Shell

By default, Platform Builder provides three shells to implement the interface between the target device and the user: the command processor shell, the standard shell, and a thin client shell. Each shell supports different features to interact with the target device.

Command Processor Shell

The command processor shell provides console input and output with a limited set of commands. This shell is available for both display-enabled devices and headless devices without keyboard and display screen. For display-enabled devices, include the Console Window component (Cmd.exe) so that the command processor shell can handle input and output through a command-prompt window. Headless devices, on the other hand, typically use a serial port for input and output.

Table 3–5 lists registry settings that you must configure on the target device to use a serial port in conjunction with the command processor shell.

Table 3-5 Console registry parameters

Location	HKEY_LOCAL_MACHINE\Drivers\Console	
Registry Entry	OutputTo	COMSpeed
Type	REG_DWORD	REG_DWORD
Default Value	None	19600
Description	Defines which serial port the command processor shell uses for input and output. ■ Setting this value to -1 will redirect input and output to a debug port. ■ Setting this value to zero specifies no redirection. ■ Setting this value to a number greater than zero and less than 10 will redirect input and output to a serial port.	Specifies the data transfer rate of the serial port in bits per second (bps).

Windows Embedded CE Standard Shell

The standard shell provides a graphical user interface (GUI) similar to the Windows XP desktop. The primary purpose of the standard shell is to start and run user applications on the target device. This shell includes a desktop with Start menu and taskbar that enables the user to switch between applications, from one window to another. The standard shell also includes a system notification area to display additional information, such as the status of network interfaces and the current system time.

Windows Embedded CE Standard Shell is a required catalog item if you select the Enterprise Terminal design template when creating an OS design project in Visual Studio by using the OS Design Wizard. If you want to clone and customize this shell, you can find the source code in the %_WINCEROOT\Public\Shell\OAK\HPC folder. Chapter 1 explains how to clone catalog items and add them to an OS design.

Thin Client Shell

The thin client shell, also called the Windows–based Terminal (WBT) shell in the product documentation, is a GUI shell for thin-client devices that do not run user applications locally. You can add Internet Explorer to a thin-client OS design, yet all other user applications must run on a Terminal server in the network. The thin client shell uses the Remote Desktop Protocol (RDP) to connect to the server and display the remote Windows desktop. By default, the thin client shell displays the remote desktop in full-screen mode.

Taskman

You can also implement your own shell by cloning and customizing the Windows Task Manager (TaskMan) shell application. The source code in the %_WINCEROOT% \Public\Wceshellfe\Oak\Taskman folder is a good starting point.

Windows Embedded CE Control Panel

The Control Panel is a special repository for central access to system and application configuration tools. The product documentation refers to these configuration tools as applets, to indicate the fact that they are embedded in the Control Panel. Each applet serves a specific and targeted purpose and does not depend on other applets. You can customize the content of the Control Panel by adding your own applets or by removing existing Control Panel applets included with Windows Embedded CE.

Control Panel Components

The Control Panel is a configuration system that relies on the following three key components:

- **Front-End (Control.exe)** This application displays the user interface and facilitates starting Control Panel applets.
- **Host Application (Ctlpnl.exe)** This application loads and runs the Control Panel applets.
- **Applets** These are the individual configuration tools, implemented in form of .cpl files listed with icon and name in the Control Panel user interface.

For details regarding the implementation of the Windows Embedded CE Control Panel, check out the source code in the %_WINCEROOT%\Public\Wceshellfe\Oak\Ctlpnl folder. You can clone the Control Panel code and customize it to implement your own Control Panel version

Implementing Control Panel Applets

As mentioned, a Control Panel applet is a configuration tool for a system component or user application implemented in form of a .cpl file and located in the Windows folder on the target device. Essentially, a .cpl file is a DLL that implements the CPlApplet API. A single .cpl file can contain multiple Control Panel applications, yet a single applet cannot span multiple .cpl files. Because all .cpl files implement the CPlApplet API, it is a straightforward process for Control.exe to obtain detailed info about the implemented applets at startup in order to display the set of available applets in the user interface. Control.exe only needs to enumerate all .cpl files in the Windows folder and to call the CPlApplet function in each file.

According to the DLL nature and CPlApplet API requirements, .cpl files must implement the following two public entry points:

- **BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)** Used to initialize the DLL. The system calls DllMain to load the DLL. The DLL returns true if initialization succeeded or false if initialization failed.
- **LONG CALLBACK CPlApplet(HWND hwndCPL, UINT message, LPARAM lParam1, LPARAM lParam2)** A callback function that serves as the entry point for the Control Panel to perform actions on the applet.

NOTE DLL entry points

You must export the DllMain and CPlApplet entry points so that the Control Panel application can access these functions. Non-exported functions are private to the DLL. Make sure the function definitions are in export "C" { } blocks to export the C interface.

The Control Panel calls the CPlApplet function to initialize the applet, obtain information, provide information about user actions, and to unload the applet. The applet must support several Control Panel messages, listed in Table 3–6, to implement a fully functional CPlApplet interface:

Table 3-6 Control Panel messages

Control Panel Message	Description
CPL_INIT	The Control Panel sends this message to perform global initialization of the applet. Memory initialization is a typical task performed at this step.
CPL_GETCOUNT	The Control Panel sends this message to determine the number of Control Panel applications implemented in the .cpl file.
CPL_NEWINQUIRE	The Control Panel sends this message for all the Control Panel applications specified by CPL_GETCOUNT. At this step each Control Panel application must return a NEWCPLINFO structure to specify the icon and title to display in the Control Panel user interface.
CPL_DBLCLK	The Control Panel sends this message when the user double-clicks on an icon of the applet in the Control Panel user interface.
CPL_STOP	The Control Panel sends this message once for each instance specified by CPL_GETCOUNT.
CPL_EXIT	The Control Panel sends this message once for the applet before the system releases the DLL.

NOTE NEWCPLINFO information

Store the NEWCPLINFO information for each Control Panel application that you implement in a Control Panel applet in a resource embedded in the .cpl file. This facilitates the localization of icons, names, and applet descriptions returned in response to CPL_NEWINQUIRE messages.

Building Control Panel Applets

To build a Control Panel applet and generate the corresponding .cpl file, find the source code folder of the applet subproject and add the following CPL build directive on a new line at the end of the Sources file:

```
CPL=1
```

You also must add the path to the Control Panel header file to the Include Directories entry on the C/C++ tab in the applet subproject settings in Visual Studio, as illustrated in Figure 3–3:

```
$(_PROJECTROOT)\CESysgen\Oak\Inc
```

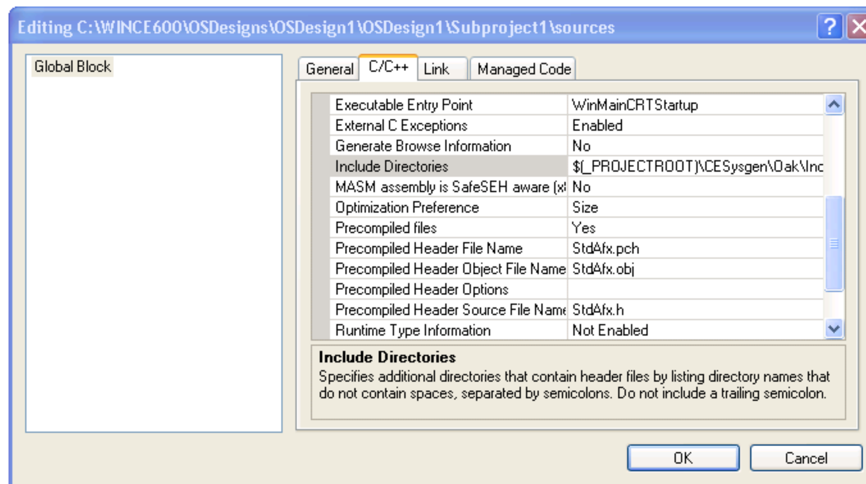


Figure 3-3 Include Directories entry for a Control Panel applet

Enabling Kiosk Mode

Many Windows Embedded CE devices, such as medical monitoring devices, automated teller machines (ATM), or industrial control systems are dedicated to a single task. The standard graphical shell is not useful for these devices. Removing the standard shell restricts access to the Control Panel configuration settings and also protects users from starting additional applications. The result is a device in kiosk mode that opens an application according to the special purpose of the target device directly with no shell access.

Kiosk applications for Windows Embedded CE are developed in native code or managed code. The only requirement is to start this application in place of the standard shell (Explorer.exe). The system then starts a black shell, meaning no shell application is running on the device. You only need to configure the registry entries under the HKEY_LOCAL_MACHINE\Init key to implement this configuration. As mentioned earlier in this chapter, the LaunchXX entry for Explorer.exe is Launch50. Replace Explorer.exe with your custom kiosk application and consider the job completed, as shown in Table 3–7. Keep in mind that your custom kiosk application must implement the SignalStarted API for the kernel to manage the application dependencies correctly.

Table 3-7 Startup registry parameter examples

Location	HKEY_LOCAL_MACHINE\INIT
Component	Custom Kiosk Application
Binary	Launch50="myKioskApp.exe"
Dependencies	Depend50=hex:14,00, 1e,00
Description	To enable kiosk mode replace the Launch50 entry for Explorer.exe in the device registry with an entry that points to a custom kiosk application.

NOTE Kiosk Mode for managed applications

To run a managed application in place of the standard shell, include the binary file in the run-time image and edit the .bib file that belongs to the managed application. Specifically, you must define binary files in a FILES section for the system to load the application inside the Common Language Runtime (CLR).

Lesson Summary

Windows Embedded CE is a componentized operating system with a broad palette of items and customizable features. One such feature enables you to configure automatic launching of applications at start time, which is particularly useful for installation and configuration tools. You can also customize the Control Panel by adding your own applets, implemented in custom .cpl files, which are DLLs that adhere to the CPlApplet API so that the Control Panel can call into the applets. For special-purpose devices, such as ATMs, ticket machines, medical monitoring devices, airport check-in terminals, or industrial control systems, you can further customize the user environment by replacing the standard shell with your kiosk application. You do not need to customize the code base or start process of the Windows Embedded CE operating system. Enabling kiosk mode is merely a task of replacing the default Launch50 registry entry with a custom Launch50 entry that points to your standard or managed code application.

Lesson 3: Implementing Threads and Thread Synchronization

Windows Embedded CE is a multithreaded operating system. The processing model differs from UNIX–based embedded operating systems because processes can include multiple threads. You need to know how to manage, schedule, and synchronize these threads within a single process and between processes in order to implement and debug multithreaded applications and drivers and to achieve optimal system performance on your target devices.

After this lesson, you will be able to:

- Create and stop a thread.
- Manage thread priorities.
- Synchronize multiple threads.
- Debug thread synchronization issues.

Estimated lesson time: 45 minutes.

Processes and Threads

A process is a single instance of an application. It has a processing context, which can include a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, and environment variables. It also has a primary thread of execution. A thread is the basic unit of execution managed by the scheduler. In a Windows process, a thread can create additional threads. There is no hard-coded maximum number of threads per process. The maximum number depends on available memory resources because every thread uses memory and the physical memory is limited on the platform. The maximum number of processes on Windows Embedded CE is limited to 32,000.

Thread Scheduling on Windows Embedded CE

Windows Embedded CE supports preemptive multitasking to run multiple threads from various processes simultaneously. Windows Embedded CE performs thread scheduling based on priority. Each thread on the system has a priority ranging from zero to 255. Priority zero is the highest priority. The scheduler maintains a priority list and selects the thread to run next according to the thread priority in a round-robin fashion. Threads of the same priority run sequentially in a random order. It is

important to note that thread scheduling relies on a time-slice algorithm. Each thread can only run for a limited amount of time. The maximum possible time slice that a thread can run is called the *quantum*. Once the quantum has elapsed, the scheduler suspends the thread and resumes the next thread in the list.

Applications can set the quantum on a thread-by-thread basis to adapt thread scheduling according to application needs. However, changing the quantum for a thread does not affect threads with a higher priority because the schedule selects threads with higher priority to run first. The scheduler even suspends lower-priority threads within their time slice if a higher-priority thread becomes available to run.

Process Management API

Windows Embedded CE includes several process management functions as part of the core Win32 API. Three important functions are listed in Table 3–8 that are useful for creating and ending processes.

Table 3-8 Process management functions

Function	Description
CreateProcess	Starts a new process.
ExitProcess	Ends a process with cleanup and unloading DLLs.
TerminateProcess	Terminates a process without cleanup or unloading DLLs.

MORE INFO Process management API

For more information about process management functions and complete API documentation, see the Core OS Reference for Windows Mobile® 6 and Windows Embedded CE 6.0, available on the Microsoft MSDN website at *http://msdn2.microsoft.com/en-us/library/aa910709.aspx*.

Thread Management API

Each process has at least one thread called the primary thread. This is the main thread of the process, which means that exiting or terminating this thread also ends the process. The primary thread can also create additional threads, such as worker threads, to perform parallel calculations or accomplish other processing tasks. These additional threads can create more threads if necessary by using the core Win32 API. Table 3–9 lists the most important functions to use in applications that work with threads on Windows Embedded CE.

Table 3-9 Thread management functions

Function	Description
CreateThread	Creates a new thread.
ExitThread	Ends a thread.
TerminateThread	Stops a specified thread without running cleanup or other code. Use this function only in extreme cases because terminating a thread can leave memory objects behind and cause memory leaks.
GetExitCodeThread	Returns the thread exit code.
CeSetThreadPriority	Sets the thread priority.
CeGetThreadPriority	Gets the current thread priority.
SuspendThread	Suspends a thread.
ResumeThread	Resumes a suspended thread.
Sleep	Suspends a thread for a specified amount of time.
SleepTillTick	Suspends a thread until the next system tick.

MORE INFO Thread management API

For more information about thread management functions and complete API documentation, see the Core OS Reference for Windows Mobile 6 and Windows Embedded CE 6.0, available on the Microsoft MSDN website at *http://msdn2.microsoft.com/en-us/library/aa910709.aspx*.

Creating, Exiting, and Terminating Threads

The CreateThread function used to create a new thread expects several parameters that control how the system creates the thread and the instructions that the thread runs. Although it is possible to set most of these parameters to null or zero, it is necessary to provide at least a pointer to an application-defined function that the thread is supposed to execute. This function typically defines the core processing instructions for the thread, although you can also call other functions from within this function. It is important to pass the core function as a static reference to CreateThread because the linker must be able to determine the core function's starting address at compile time. Passing a non-static function pointer does not work.

The following code listing is copied from the Explorer.cpp file that you can find in the %_WINCEROOT%\Public\Shell\OAK\HPC\Explorer\Main folder. It illustrates how to create a thread.

```
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();
}
```

This code specifies FontThread as the new thread's core function. It immediately closes the returned thread handle because the current thread does not need it. The new thread runs parallel to the current thread and implicitly exits upon returning from the core function. This is the preferred way to exit threads because it enables C++ function cleanup to occur. It is not necessary to explicitly call ExitThread.

However, it is possible to explicitly call the ExitThread function within a thread routine to end processing without reaching the end of the core function. ExitThread invokes the entry point of all attached DLLs with a value indicating that the current thread is detaching, and then deallocates the current thread's stack to terminate the current thread. The application process exits if the current thread happens to be the primary thread. Because ExitThread acts on the current thread, it is not necessary to specify a thread handle. However, you must pass a numeric exit code, which other threads can retrieve by using the GetExitCodeThread function. This process is useful to identify errors and reasons for the thread exiting. If ExitThread is not explicitly called, the exit code corresponds to the return value of the thread function. If GetExitCodeThread returns the value STILL_ACTIVE, the thread is still active and running.

Although you should avoid it, there can be rare situations that leave you no other way to terminate a thread except for calling the TerminateThread function. A malfunctioning thread destroying file records might require this function. Formatting a file system might need you to call TerminateThread in debugging sessions while your code is still under development. You need to pass the handle to the thread to be terminated and an exit code, which you can retrieve later by using the

GetExitCodeThread function. Calling the TerminateThread function should never be part of normal processing. It leaves the thread stack and attached DLLs behind, abandons critical sections and mutexes owned by the terminated thread, and leads to memory leaks and instability. Do not use TerminateThread as part of the process shutdown procedure. Threads within the process can exit implicitly or explicitly by using the ExitThread function.

Managing Thread Priority

Each thread has a priority value ranging from zero to 255, which determines how the system schedules the thread to run in relationship to all other threads within the process and between processes. On Windows Embedded CE, the core Win32 API includes four thread management functions that set the priority of a thread as follows.

- **Base priority levels** Use the SetThreadPriority and SetThreadPriority functions to manage the thread priority at levels compatible with early versions of Windows Embedded CE (zero through seven).
- **All priority levels** Use the CeSetThreadPriority and CeGetThreadPriority functions to manage the thread priority at all levels (zero through 255).

NOTE Base priority levels

The base priority levels zero through seven of earlier versions of Windows Embedded CE are now mapped to the eight lowest priority levels 248 through 255 of the CeSetThreadPriority function.

It is important to keep in mind that thread priorities define a relationship between threads. Assigning a high thread priority can be detrimental to the system if other important threads run with lower priority. You might achieve better application behavior by using a lower priority value. Performance testing with different priority values is a reliable manner of identifying the best priority level for a thread in an application or driver. However, testing 256 different priority values is not efficient. Choose an appropriate priority range for your threads according to the purpose of your driver or application as listed in Table 3–10.

Table 3-10 Thread priority ranges

Range	Description
zero through 96	Reserved for real-time drivers.
97 through 152	Used by default device drivers.

Table 3-10 Thread priority ranges

Range	Description
153 through 247	Reserved for real-time below drivers.
248 through 255	Maps to non-real-time priorities for applications.

Suspending and Resuming Threads

It can help system performance to delay certain conditional tasks that depend on time-consuming initialization routines or other factors. After all, it is not efficient to enter a loop and check 10,000 times if a required component is finally ready for use. A better approach is to put the worker thread to sleep for an appropriate amount of time, such as 10 milliseconds, check the state of the dependencies after that time, and go back to sleep for another 10 milliseconds or continue processing when conditions permit. Use the Sleep function from within the thread itself to suspend and resume a thread. You can also use the SuspendThread and ResumeThread functions to control a thread through another thread.

The Sleep function accepts a numeric value that specifies the sleep interval in milliseconds. It is important to remember that the actual sleep interval will likely exceed this value. The Sleep function relinquishes the remainder of the current thread's quantum and the scheduler will not give this thread another time slice until the specified interval has passed and there are no other threads with higher priority. For example, the function call `Sleep(0)` does not imply a sleep interval of zero milliseconds. Instead, `Sleep(0)` relinquishes the remainder of the current quantum to other threads. The current thread will only continue to run if the scheduler has no other threads with the same or higher priority on the thread list.

Similar to the `Sleep(0)` call, the SleepTillTick function relinquishes the remainder of the current thread's quantum and suspends the thread until the next system tick. This is useful if you want to synchronize a task on a system tick basis.

The WaitForSingleObject or WaitForMultipleObjects functions suspend a thread until another thread or a synchronization object is signaled. For example, a thread can wait for another thread to exit without having to enter a loop with repeated Sleep and GetExitCodeThread calls if the WaitForSingleObject function is enabled instead. This approach results in a better use of resources and improves code readability. It is possible to pass a timeout value in milliseconds to the WaitForSingleObject or WaitForMultipleObjects functions.

Thread Management Sample Code

The following code snippet illustrates how to create a thread in suspended mode, specify a thread function and parameters, change the thread priority, resume the thread, and wait for the thread to finish its processing and exit. In the last step, the following code snippet demonstrates how to check the error code returned from the thread function.

```
// 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);
}
```

Thread Synchronization

The real art of multithreaded programming lies in avoiding deadlocks, protecting access to resources, and ensuring thread synchronization. Windows Embedded CE provides several kernel objects to synchronize resource access for threads in drivers or applications, such as critical sections, mutexes, semaphores, events, and interlocks functions. Yet, the choice of the object depends on the task that you want to accomplish.

Critical Sections

Critical sections are objects that synchronize threads and guard access to resources within a single process. A critical section cannot be shared between processes. To access a resource protected by a critical section, a thread calls the EnterCriticalSection function. This function blocks the thread until the critical section is available.

In some situations, blocking the thread execution might not be efficient. For example, if you want to use an optional resource that might never be available, calling the EnterCriticalSection function blocks your thread and consumes kernel resources without performing any processing on the optional resource. It is more efficient in this case to use a critical section without blocking by calling the TryEnterCriticalSection function. This function attempts to grab the critical section and returns immediately if the critical section cannot be used. The thread can then continue along an alternative code path, such as to prompt the user for input or to plug in a missing device.

Having obtained the critical section object through EnterCriticalSection or TryEnterCriticalSection, the thread enjoys exclusive access to the resource. No other thread can access this resource until the current thread calls the LeaveCriticalSection function to release the critical section object. Among other things, this mechanism highlights why you should not use the TerminateThread function to terminate threads. TerminateThread does not perform cleanup. If the terminated thread owned a critical section, the protected resource becomes unusable until the user restarts the application.

Table 3–11 lists the most important functions that you can use to work with critical section objects for thread synchronization purposes.

Table 3-11 Critical Section API

Function	Description
InitializeCriticalSection	Create and initialize a critical section object.
DeleteCriticalSection	Destroy a critical section object.
EnterCriticalSection	Grab a critical section object.
TryEnterCriticalSection	Try to grab a critical section object.
LeaveCriticalSection	Release a critical section object.

Mutexes

Whereas critical sections are limited to a single process, mutexes can coordinate mutually exclusive access to resources shared between multiple processes. A mutex is a kernel object that facilitates inter-process synchronization. Call the CreateMutex function to create a mutex. The creating thread can specify a name for the mutex object at creation time, although it is possible to create an unnamed mutex. Threads in other processes can also call CreateMutex and specify the same name. However, these subsequent calls do not create new kernel objects, but instead return a handle to the existing mutex. At this point, the threads in the separate processes can use the mutex object to synchronize access to the protected shared resource.

The state of a mutex object is signaled when no thread owns it and non-signaled when one thread has ownership. A thread must use one of the wait functions, WaitForSingleObject or WaitForMultipleObjects, to request ownership. You can specify a timeout value to resume thread processing along an alternative code path if the mutex

does not become available during the wait interval. On the other hand, if the mutex becomes available and ownership is granted to the current thread, do not forget to call ReleaseMutex for each time that the mutex satisfied a wait in order to release the mutex object for other threads. This is important because a thread can call a wait function multiple times, such as in a loop, without blocking its own execution. The system does not block the owning thread to avoid a deadlock situation, but the thread must still call ReleaseMutex as many times as the wait function to release the mutex.

Table 3–12 lists the most important functions that you can use to work with mutex objects for thread synchronization purposes.

Table 3-12 Mutex API

Function	Description
CreateMutex	Create and initialize a named or unnamed mutex object. To protect resources shared between processes, you must use named mutex objects.
CloseHandle	Closes a mutex handle and deletes the reference to the mutex object. All references to the mutex must be deleted individually before the kernel deletes the mutex object.
WaitForSingleObject	Waits to be granted ownership of a single mutex object.
WaitForMultipleObjects	Waits to be granted ownership for a single or multiple mutex objects.
ReleaseMutex	Releases a mutex object.

Semaphores

Apart from kernel objects that enable you to provide mutually exclusive access to resources within a process and between processes, Windows Embedded CE also provides semaphore objects that enable concurrent access to a resource by one or multiple threads. These semaphore objects maintain a counter between zero and a maximum value to control the number of threads accessing the resource. The maximum value amount is specified in the CreateSemaphore function call.

The semaphore counter limits the number of threads that can access the synchronization object concurrently. The system will keep decrementing the counter

every time a thread completes a wait for the semaphore object until the counter reaches zero and enters the nonsignaled state. The counter cannot decrement past zero. No further thread can gain access to the resource until an owning thread releases the semaphore by calling the ReleaseSemaphore function, which increments the counter by a specified value and again switches the semaphore object back into signaled state.

Similar to mutexes, multiple processes can open handles of the same semaphore object to access resources shared between processes. The first call to the CreateSemaphore function creates the semaphore object with a specified name. You can also construct unnamed semaphores, but these objects are not available for interprocess synchronization. Subsequent calls to the CreateSemaphore function with the same semaphore name do not create new objects, but open a new handle of the same semaphore.

Table 3–13 lists the most important functions that work with semaphore objects for thread synchronization purposes.

Table 3-13 Semaphore API

Function	Description
CreateSemaphore	Creates and initializes a named or unnamed semaphore object with a counter value. Use named semaphore objects to protect resources shared between processes.
CloseHandle	Closes a semaphore handle and deletes the reference to the semaphore object. All references to the semaphore must be closed individually before the kernel deletes the semaphore object.
WaitForSingleObject	Waits to be granted ownership of a single semaphore object.
WaitForMultipleObjects	Waits to be granted ownership for a single or multiple semaphore objects.
ReleaseSemaphore	Releases a semaphore object.

Events

The Event object is another kernel object that synchronizes threads. This object enables applications to signal other threads when a task is finished or when data is available to potential readers. Each event has signaled/non-signaled state information used by the API to identify the state of the event. Two types of events, manual events and auto-reset events, are created according to the behavior expected by the event.

The creating thread specifies a name for the event object at creation time, although it is also possible to create an unnamed event. It is possible for threads in other processes to call CreateMutex and specify the same name, but these subsequent calls do not create new kernel objects.

Table 3–14 lists the most important functions for event objects for thread synchronization purposes.

Table 3-14 Event API

Function	Description
CreateEvent	Creates and initializes a named or unnamed event object.
SetEvent	Signal an event (see below).
PulseEvent	Pulse and signal the event (see below).
ResetEvent	Reset a signaled event.
WaitForSingleObject	Waits for an event to be signaled.
WaitForMultipleObjects	Waits to be signaled by a single or multiple event objects.
CloseHandle	Releases an Event object.

The behavior of the events API is different according to the type of events. When you use SetEvent on a manual event object, the event will stay signaled until ResetEvent is explicitly called. Auto-reset events only stay signaled until a single waiting thread is released. At most, one waiting thread is released when using the PulseEvent function on auto-reset events before it immediately transitions back to the non-signaled state. In the case of manual threads, all waiting threads are released and immediately transition back to a non-signaled state.

Interlocked Functions

In multithread environments, threads can be interrupted at any time and resumed later by the scheduler. Portions of code or applications resources can be protected using semaphores, events, or critical sections. In some applications, it could be too time consuming to use those kinds of system objects to protect only one line of code like this:

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

The sample source code above in C is one single instruction, but in assembly it could be more than that. In this particular example, the thread can be suspended in the middle of the operation and resumed later, but errors can potentially be encountered in the case of another thread using the same variable. The operation is not atomic. Fortunately, it is possible in Windows Embedded CE 6.0 R2 to increment, decrement, and add values in multithreading-safe, atomic operations without using synchronization objects. This is done by using interlocked functions.

Table 3–15 lists the most important interlocked functions that are used to atomically manipulate variables.

Table 3-15 Interlock API

Function	Description
InterlockedIncrement	Increment the value of a 32 bit variable.
InterlockedDecrement	Decrement the value of a 32 bit variable.
InterlockedExchangeAdd	Perform atomic addition on a value.

Troubleshooting Thread Synchronization Issues

Multithreaded programming enables you to structure your software solutions based on separate code execution units for user interface interaction and background tasks. It is an advanced development technique that requires careful implementation of thread synchronization mechanisms. Deadlocks can happen, especially when using multiple synchronization objects in loops and subroutines. For example, thread One owns mutex A and waits for mutex B before releasing A, while thread Two waits for mutex A before releasing mutex B. Neither thread can continue in this situation because each depends on a resource being released by the other. These situations are hard to locate and troubleshoot, particularly when threads from multiple processes

are accessing shared resources. The Remote Kernel Tracker tool identifies how threads are scheduled on the system and enables you to locate deadlocks.

The Remote Kernel Tracker tool enables you to monitor all processes, threads, thread interactions, and other system activities on a target device. This tool relies on the CeLog event-tracking system to log kernel and other system events in a file named Celog.clg in the %_FLATRELEASEDIR% directory. System events are classified by zone. The CeLog event-tracking system can be configured to focus on a specific zone for data logging.

If you have Kernel Independent Transport Layer (KITL) enabled on the target device, the Remote Kernel Tracker visualizes the CeLog data and analyzes the interactions between threads and processes. This is illustrated in Figure 3–4. While KITL sends the data directly to the Remote Kernel Tracker tool, it is also possible to analyze collected data offline.

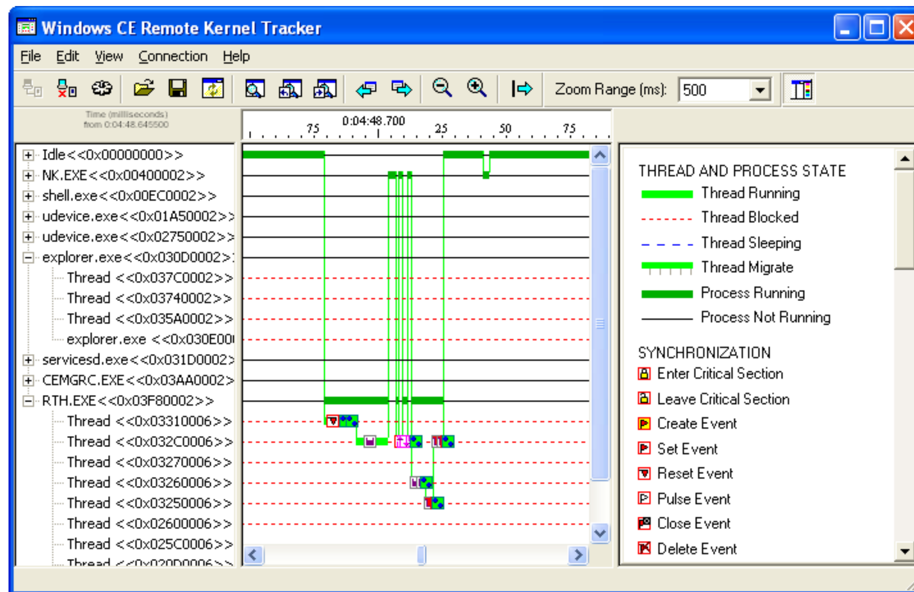


Figure 3-4 The Remote Kernel Tracker tool

MORE INFO: CeLog event tracking and filtering

For more information about CeLog event tracking and CeLog event filtering, see the section "CeLog Event Tracking Overview" in the Windows Embedded CE 6.0 documentation available on the Microsoft MSDN website at *http://msdn2.microsoft.com/en-us/library/aa935693.aspx*.

Lesson Summary

Windows Embedded CE is a multithreaded operating system that provides several process management functions to create processes and threads, assign thread priorities ranging from zero through 255, suspend threads, and resume threads. The Sleep function is useful in suspending a thread for a specified period of time, but the WaitForSingleObject or WaitForMultipleObjects functions can also be used to suspend a thread until another thread or a synchronization object is signaled. Processes and threads are ended in two ways: with and without cleanup. As a general rule, always use ExitProcess and ExitThread to give the system a chance to perform cleanup. Use TerminateProcess and TerminateThread only if you have absolutely no other choice.

When working with multiple threads, it is beneficial to implement thread synchronization in order to coordinate access to shared resources within and between processes. Windows Embedded CE provides several kernel objects for this purpose, specifically critical sections, mutexes, and semaphores. Critical sections guard access to resources within a single process. Mutexes coordinate mutually exclusive access to resources shared among multiple processes. Semaphores implement concurrent access by multiple threads to resources within a process and between processes. Events are used to notify the other threads, and interlocked functions for the manipulation of variables in a thread-safe atomic way. If you happen to encounter thread synchronization problems during the development phase, such as deadlocks, use the CeLog event-tracking system and Remote Kernel Tracker to analyze the thread interactions on the target device.

EXAM TIP

To pass the certification exam, make sure you understand how to use the various synchronization objects in Windows Embedded CE 6.0 R2.

Lesson 4: Implementing Exception Handling

Target devices running Windows Embedded CE include exceptions as part of system and application processing. The challenge is to respond to exceptions in the appropriate manner. Handling exceptions correctly ensures a stable operating system and positive user experience. For example, instead of unexpectedly terminating a video application, you might find it more useful to prompt the user to connect a Universal Serial Bus (USB) camera if the camera is currently disconnected. However, you should not use exception handling as a universal solution. Unexpected application behavior can be the result of malicious code tampering with executables, DLLs, memory structures, and data. In this case, terminating the malfunctioning component or application is the best course of action to protect the data and the system.

After this lesson, you will be able to:

- Understand the reason for exceptions.
- Catch and throw exceptions.

Estimated lesson time: 30 minutes.

Exception Handling Overview

Exceptions are events resulting from error conditions. These conditions can arise when the processor, operating system, and applications are executing instructions outside the normal flow of control in kernel mode and user mode. By catching and handling exceptions, you can increase the robustness of your applications and ensure a positive user experience. Strictly speaking, however, you are not required to implement exception handlers in your code because structured exception handling is an integral part of Windows Embedded CE.

The operating system catches all exceptions and forwards them to the application processes that caused the events. If a process does not handle its exception event, the system forwards the exception to a postmortem debugger and eventually terminates the process in an effort to protect the system from malfunctioning hardware or software. Dr. Watson is a common postmortem debugger that creates a memory dump file for Windows Embedded CE.

Exception Handling and Kernel Debugging

Exception handling is also the basis for kernel debugging. When you enable kernel debugging in an operating system design, Platform Builder includes the kernel debugging stub (KdStub) in the run-time image to enable components that raise exceptions to break into the debugger. Now you can analyze the situation, step through the code, resume processing, or terminate the application process manually. However, you need a KITL connection to a development workstation in order to interact with the target device. Without a KITL connection, the debugger ignores the exception and lets the application continue to run so that the operating system can use another exception handler as if no debugger was active. If the application does not handle the exception, then the operating system gives the kernel debugger a second chance to perform postmortem debugging. In this context, it is often called just in time (JIT) debugging. The debugger must now accept the exception and waits for a KITL connection to become available for the debug output. Windows Embedded CE waits until you establish the KITL connection and start debugging the target device. Developer documentation often uses the terms first-chance exception and second-chance exception because the kernel debugger has two chances to handle an exception in this scenario, but they are, in fact, referring to the same exception event. For more information about debugging and system testing, read Chapter 5, "Debugging and Testing the System."

Hardware and Software Exceptions

Windows Embedded CE uses the same structured exception handling (SEH) approach for all hardware and software exceptions. The central processing unit (CPU) can raise hardware exceptions in response to invalid instruction sequences, such as division by zero or an access violation caused by an attempt to access an invalid memory address. Drivers, system applications, and user applications, on the other hand, can raise software exceptions to invoke the operating system's SEH mechanisms by using the RaiseException function. For example, you can raise an exception if a required device is not accessible (such as a USB camera or a database connection), if the user specified an invalid command-line parameter, or for any other reason that requires you to run special instructions outside the normal code path. You can specify several parameters in the RaiseException function call to specify information that describes the exception. This specification can then be used in the filter expression of an exception handler.

Exception Handler Syntax

Windows Embedded CE supports frame–based structured exception handling. It is possible to enclose a sensitive sequence of code in braces ({}) and mark it with the __try keyword to indicate that any exceptions during the execution of this code should invoke an exception handler that follows in a section marked by using the __except keyword. The C/C++ compiler included in Microsoft Visual Studio supports these keywords and compiles the code blocks with additional instructions that enable the system either to restore the machine state and continue thread execution at the point at which the exception occurred, or to transfer control to an exception handler and continue thread execution in the call stack frame in which the exception handler is located.

The following code fragment illustrates how to use the __try and __except keywords for structured exception handling:

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

The __except keyword supports a filter expression, which can be a simple expression or a filter function. The filter expression can evaluate to one of the following values:

- **EXCEPTION_CONTINUE_EXECUTION** The system assumes that the exception is resolved and continues thread execution at the point at which the exception occurred. Filter functions typically return this value after handling the exception to continue processing as normal.
- **EXCEPTION_CONTINUE_SEARCH** The system continues its search for an appropriate exception handler.
- **EXCEPTION_EXECUTE_HANDLER** The system thread execution continues sequentially from the exception handler rather than from the point of the exception.

> **NOTE Exception handling support**
> Exception handling is an extension of the C language, but it is natively supported in C++.

Termination Handler Syntax

Windows Embedded CE supports termination handling. As a Microsoft extension to the C and C++ languages, it enables you to guarantee that the system always runs a certain block of code not matter how the flow of control leaves the guarded code block. This code section is called a termination handler, and is used to perform cleanup tasks even if an exception or some other error occurs in the guarded code. For example, you can use a termination handler to close thread handles that are no longer needed.

The following code fragment illustrates how to use the __try and __finally keywords for structured exception handling:

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

Termination handling supports the __leave keyword within the guarded section. This keyword ends thread execution at the current position in the guarded section and resumes thread execution at the first statement in the termination handler without unwinding the call stack.

NOTE Using __try, __except,and __finally blocks

A single __try block cannot have both an exception handler and a termination handler. If you must use both __except and __finally, use an outer try-except statement and an inner try-finally statement.

Dynamic Memory Allocation

Dynamic memory allocation is an allocation technique that relies on structured exception handling to minimize the total number of committed memory pages on the system. This is particularly useful if you must perform large memory allocations. Pre-committing an entire allocation can cause the system to run out of committable pages and result in virtual memory allocation failures.

The dynamic memory allocation technique is as follows:

1. Call VirtualAlloc with a base address of NULL to reserve a block of memory. The system reserves this memory block without committing the pages.
2. Try to access a memory page. This raises an exception because you cannot read from or write to a non-committed page. This illegal operation results in a page fault exception. Figure 3–5 and Figure 3–6 show the outcome of an unhandled page fault in an application called PageFault.exe.
3. Implement an exception handler based on a filter function. Commit a page in the filter function from the reserved region. If successful, return EXCEPTION_CONTINUE_EXECUTION to continue thread execution in the __try block at the point where the exception occurred. If the page allocation failed, return EXCEPTION_EXECUTE_HANDLER to invoke the exception handler in the __except block and release the entire region of reserved and committed pages.

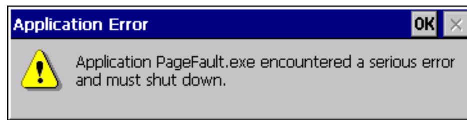


Figure 3-5 An unhandled page fault exception from a user's perspective

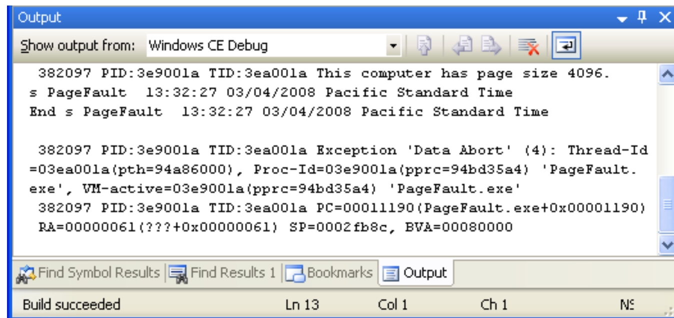


Figure 3-6 An unhandled page fault exception's debug output over KITL in Visual Studio 2005

The following code snippet illustrates the dynamic memory allocation technique based on page fault exception handling:

```
#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);
}
```

Lesson Summary

Windows Embedded CE supports exception handling and termination handling natively. Exceptions in the processor, operating system, and applications are raised in response to improper instruction sequences, attempts to access an unavailable memory address, inaccessible device resources, invalid parameters, or any other operation that requires special processing, such as dynamic memory allocations. You can use try–except statements to react to error conditions outside the normal flow of control and try–finally statements to run code no matter how the flow of control leaves the guarded __try code block.

Exception handling supports filtering expressions and filtering functions, which enable you to control how you respond to raised events. It is not advisable to catch all exceptions because unexpected application behavior can be the result of malicious code. Only handle those exceptions that need to be dealt with directly for reliable and robust application behavior. The operating system can forward any unhandled exceptions to a postmortem debugger to create a memory dump and terminate the application.

EXAM TIP

To pass the certification exam, make sure you understand how to use exception handling and termination handling in Windows Embedded CE 6.0 R2.

Lesson 5: Implementing Power Management

Power management is essential for Windows Embedded CE devices. By lowering power consumption, you extend battery life and ensure a long-lasting, positive user experience. This is the ultimate goal of power management on portable devices. Stationary devices also benefit from power management. Regardless of equipment size, you can reduce operating costs, heat dissipation, mechanical wear and tear, and noise levels if you switch the device into a low-power state after a period of inactivity. And of course, implementing effective power-management features helps to lessen the burden on our environment.

After this lesson, you will be able to:

- Enable power management on a target device.
- Implement power-management features in applications.

Estimated lesson time: 40 minutes.

Power Manager Overview

On Windows Embedded CE, Power Manager (PM.dll) is a kernel component that integrates with the Device Manager (Device.exe) to implement power management features. Essentially, Power Manager acts as a mediator between the kernel, OEM adaptation layer (OAL), and drivers for peripheral devices and applications. By separating the kernel and OAL from drivers and applications, drivers and applications can manage their own power state separately from the system state. Drivers and applications interface with Power Manager to receive notifications about power events and to perform power management functions. Power Manager has the ability to set the system power state in response to events and timers, control driver power states, and respond to OAL events that require a power state change, such as when the battery power state is critical.

Power Manager Components and Architecture

Power Manager exposes a notification interface, an application interface, and a device interface according to its tasks. The notification interface enables applications to receive information about power management events, such as when the system state or a device power state changes. In response to these events, power management–enabled applications use the application interface to interact with Power Manager to communicate their power management requirements or change the system power

state. The device interface, on the other hand, provides a mechanism to control the power level of device drivers. Power Manager can set device power states separately from the system power state. Similar to applications, device drivers may use the driver interface to communicate their power requirements back to Power Manager. The important point is that Power Manager and the Power Manager APIs centralize power management on Windows Embedded CE, as illustrated in Figure 3–7.

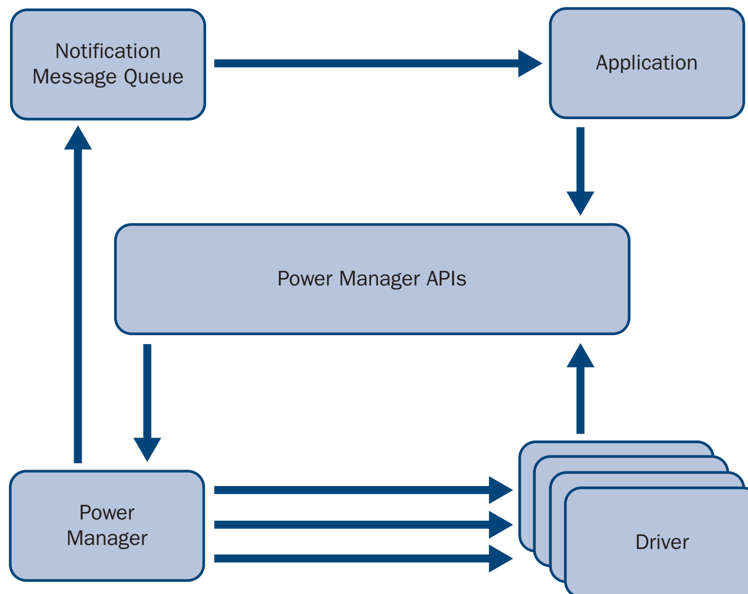


Figure 3-7 Power Manager and power management interaction

Power Manager Source Code

Windows Embedded CE comes with source code for Power Manager, which is found in the %_WINCEROOT%\Public\Common\Oak\Drivers\Pm folder on your development computer. Customizing this code provides personalized power handling mechanisms on a target device. For example, an Original Equipment Manufacturer (OEM) can implement additional logic to shut down special components before calling the PowerOffSystem function. See Chapter 1, "Customizing the Operating System Design" for techniques to clone and customize standard Windows Embedded CE components.

Driver Power States

Applications and device drivers are able to use the DevicePowerNotify function to control the power state of peripheral devices. For instance, you can call DevicePowerNotify to inform Power Manager that you want to change the power level of a backlight driver, such as BLK1:. Power Manager expects you to specify the desired power state at the one of the following five different power levels, according to the hardware device capabilities:

- **D0** Full On; the device is fully functional.
- **D1** Low On; the device is functional, but the performance is reduced.
- **D2** Standby; the device is partially powered and will wake up on requests.
- **D3** Sleep; the device is partially powered. In this state the device still has power and can raise interrupts that will wake up the CPU (device-initiated wakeup).
- **D4** Off; device has no power. The device should not consume any significant power in this state.

NOTE CE device power state levels

The device power states (D0 through D4) are guidelines to help OEMs implement power management functions on their platforms. Power Manager does not impose restrictions on device power consumption, responsiveness, or capabilities in any of the states. As a general rule, higher-numbered states should consume less power than lower numbered states and power states D0 and D1 should be for devices perceived as operational from the perspective of the user. Device drivers that manage the power level of a physical device with fewer granularities can implement a subset of the power states. D0 is the only required power state.

System Power States

In addition to sending power-state change notifications to device drivers in response to application and device driver requests, Power Manager can also transition the power state of the entire system in response to hardware-related events and software requests. Hardware events enable Power Manager to respond to low and critical battery levels and transitions from battery power to AC power. Software requests enable applications to request a change of the system power state in a call to Power Manager's SetSystemPowerState function.

The default Power Manager implementation supports the following four system power states:

- **On** The system is fully operational and on full power.

- **UserIdle** The user is passively using the device. There was no user input for a configurable period of time.
- **SystemIdle** The user is not using the device. There was no system activity for a configurable period of time.
- **Suspend** The device is powered down, but supports device-initiated wakeup.

It is important to keep in mind that system power states depend on the requirements and capabilities of the target device. OEMs can define their own or additional system power states, such as InCradle and OutOfCradle. Windows Embedded CE does not impose a limit on the number of system power states that can be defined, but all system power states eventually translate into one of the device power states, mentioned earlier in this lesson.

Figure 3–8 illustrates the relationship between the default system power states and the device power states.

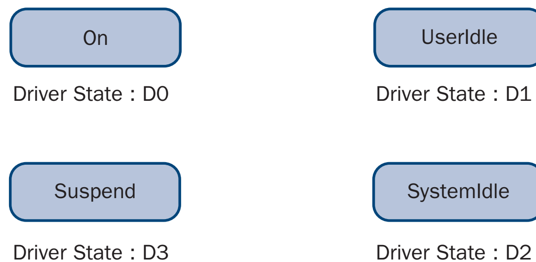


Figure 3-8 Default system power states and associated device power states

Activity Timers

System state transitions are based on activity timers and corresponding events. If a user is not using the device, a timer eventually expires and raises an inactivity event, which in turn causes Power Manager to transition the system into Suspend power state. When the user returns and interacts with the system again by providing input, an activity event occurs causing Power Manager to transition the system back into an On power state. However, this simplified model does not take into account prolonged periods of user activity without input, such as a user watching a video clip on a personal digital assistant (PDA). This simplified model also does not take into account target devices without any direct user input methods, as in the case of display panels. To support these scenarios, the default Power Manager implementation distinguishes between user activity and system activity and accordingly transitions the system power state, as illustrated in Figure 3–9.

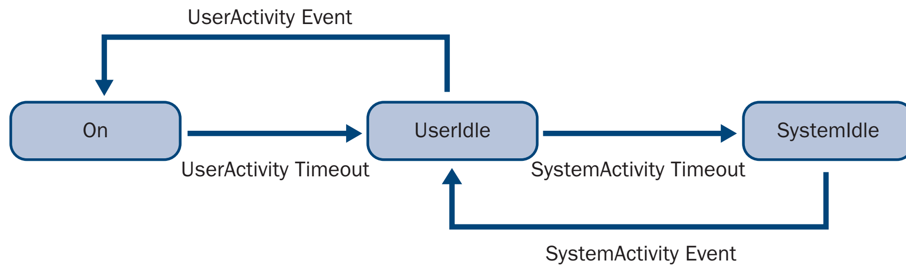


Figure 3-9 Activity timers, events, and system power state transitions

To configure system-activity and user-activity timeouts, use the Power Control Panel applet. You can also implement additional timers and set their timeouts by editing the registry directly. Windows Embedded CE does not limit the number of timers you can create. At startup, Power Manager reads the registry keys, enumerates the activity timers, and creates the associated events. Table 3–16 lists the registry settings for the SystemActivity timer. OEMs can add similar registry keys and configure these values for additional timers.

Table 3-16 Registry settings for activity timers

Location	**HKEY_LOCAL_MACHINE\System\CurrentControlSet \Control \Power\ActivityTimers\SystemActivity**	
Entry	Timeout	WakeSources
Type	REG_DWORD	REG_MULTI_SZ
Value	A (10 minutes)	0x20
Description	The Timeout registry entry defines the timer threshold in minutes.	The WakeSources registry entry is optional and defines a list of identifiers for possible wake sources. During device-initiated wakeup, Power Manager uses the IOCTL_HAL_GET_WAKE_SOURCE input and output control (IOCTL) code to determine the wake source and sets associated activity timers to active.

NOTE Activity timers

Defining activity timers causes the Power Manager to construct a set of named events for resetting the timer and for obtaining activity status. For more information, see the section "Activity Timers" in the Windows Embedded CE 6.0 Documentation available on the Microsoft MSDN website at *http://msdn2.microsoft.com/en-us/library/aa923909.aspx*.

Power Management API

As mentioned earlier in this lesson, Power Manager exposes three interfaces to enable applications and drives for power management: notification interface, driver interface, and application interface.

Notification Interface

The notification interface provides two functions that applications can use to register and deregister for power notifications through message queues, as listed in Table 3–17. It is important to note that power notifications are multicast messages, which means that Power Manager sends these notification messages only to registered processes. In this way, power management–enabled applications can seamlessly coexist on Windows Embedded CE with applications that do not implement the Power Management API.

Table 3-17 Power Management notification interface

Function	Description
RequestPowerNotifications	Registers an application process with Power Manager to receive power notifications. Power Manager then sends the following notification messages: ■ **PBT_RESUME** The system resumes from Suspend state. ■ **PBT_POWERSTATUSCHANGE** The system transitions between AC power and battery power. ■ **PBT_TRANSITION** The system changes to a new power state. ■ **PBT_POWERINFOCHANGE** The battery status changes. This message is only valid if a battery driver is loaded.

Table 3-17 Power Management notification interface (Continued)

Function	Description
StopPowerNotifications	Unregisters an application process so it no longer receives power notifications.

The following sample code illustrates how to use power notifications:

```
// 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.
    }
}
```

Device Driver Interface

In order to integrate with the Power Manager, device drivers must support a set of I/O controls (IOCTLs). Power Manager uses these to query device-specific power capabilities as well as to set and change the device's power state, as illustrated in Figure 3–10. Based on the Power Manager IOCTLs, the device driver should put the hardware device into a corresponding power configuration.

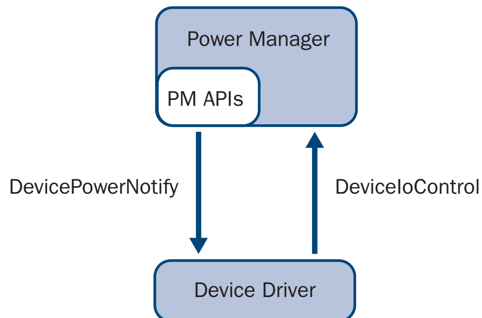


Figure 3-10 Power Manager and device driver interaction

Power Manager uses the following IOCTLs to interact with device drivers:

- **IOCTL_POWER_CAPABILITIES** Power Manager checks the power management capabilities of the device driver. The returned information should reflect the capabilities of the hardware and the driver managing the hardware device. The driver must return only supported Dx states.
- **IOCTL_POWER_SET** Power Manager forces the driver to switch to a specified Dx state. The driver must perform the power transition.
- **IOCTL_POWER_QUERY** Power Manger checks to see if the driver is able to change the state of the device.
- **IOCTL_POWER_GET** Power Manager wants to determine the current power state of the device.
- **IOCTL_REGISTER_POWER_RELATIONSHIP** Power Manager notifies a parent driver to register all child devices that it controls. Power Manager sends this IOCTL only to devices that include the POWER_CAP_PARENT flag in the Flags member of the POWER_CAPABILITIES structure.

NOTE Internal power state transitions

To ensure reliable power management, device drivers should not change their own internal power state without the involvement of Power Manager. If a driver requires a power state transition, the driver should use the DevicePowerNotify function to request the power state change. The driver can then change its internal power state when Power Manager sends a power state change request back to the driver.

Application Interface

The application interface provides functions that applications can use to manage the power state of the system and of individual devices through Power Manager. Table 3–18 summarizes these power management functions.

Table 3-18 Application interface

Function	Description
GetSystemPowerState	Retrieves the current system power state.
SetSystemPowerState	Requests a power state change. When switching to Suspend mode, the function will return after the resume because suspend is transparent to the system. After the resume, you can analyze the notification message to identify that the system resumed from suspend.
SetPowerRequirement	Requests a minimal power state for a device.
ReleasePowerRequirement	Releases a power requirement previously set with the SetPowerRequirement function and restores the original device power state.
GetDevicePower	Retrieves the current power state of a specified device.
SetDevicePower	Requests a power state change for a device.

Power State Configuration

As illustrated in Figure 3–8, Power Manager associates system power states with device power states to keep system and devices synchronized. Unless configured otherwise, Power Manager enforces the following default system-state-to-device-state mappings: On = D0, UserIdle = D1, SystemIdle = D2, and Suspend = D3. Overriding this association for individual devices and device classes can be accomplished by means of explicit registry settings.

Overriding the Power State Configuration for an Individual Device

The default Power Manager implementation maintains system-state-to-device-state mappings in the registry under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\State key. Each system power state corresponds to a separate subkey and you can create additional subkeys for OEM-specific power states.

Table 3–19 shows a sample configuration for the system power state On. This configuration causes Power Manager to switch all devices, except the backlight driver BLK1: driver, into the D0 device power state. The backlight driver BLK1: can only go to the D2 state.

Table 3-19 Default and driver-specific power state definitions for system power state On

Location	HKEY_LOCAL_MACHINE\System\CurrentControlSet\State\On		
Entry	Flags	Default	BKL1:
Type	REG_DWORD	REG_DWORD	REG_DWORD
Value	0x00010000 (POWER_STATE_ON)	0 (D0)	2 (D2)
Description	Identifies the system power state associated with this registry key. For a list of possible flags, see the Pm.h header file in the Public\Common\Sdk\Inc folder.	Sets the power state for drivers by default to the D0 state when the system power state is On.	Sets the backlight driver BLK1: to the D2 state when the system power state is On.

Overriding the Power State Configuration for Device Classes

Defining device power state for multiple system power states individually can be a tedious task. Power Manager facilitates the configuration by supporting device classes based on IClass values, which can be used to define the power management rules. The following three default class definitions are found under the HKEY_LOCAL_MACHINE \System\CurrentControlSet\Control\Power\Interfaces registry key.

- **{A3292B7-920C-486b-B0E6-92A702A99B35}** Generic power management-enabled devices.
- **{8DD679CE-8AB4-43c8-A14A-EA4963FAA715}** Power- management-enabled block devices.
- **{98C5250D-C29A-4985-AE5F-AFE5367E5006}** Power-management-enabled Network Driver Interface Specification (NDIS) miniport drivers.

Table 3–20 shows a sample configuration for the NDIS device class, which specifies that NDIS drivers only go as high as the D4 state.

Table 3-20 Sample power state definition for NDIS device class

Location	HKEY_LOCAL_MACHINE\System CurrentControlSet \Control\Power\State\On\{98C5250D-C29A-4985-AE5F-AFE5367E5006}
Entry	Default
Type	REG_DWORD
Value	4 (D4)
Description	Sets the device power state for NDIS drivers to the D4 state when the system power state is On.

Processor Idle State

In addition to power-management-enabled applications and device drivers, the kernel also contributes to power management. The kernel calls the OEMIdle function, part of the OAL, when no threads are ready to run. This action switches the processor into idle state, which includes saving the current context, placing the memory into a refresh state, and stopping the clock. The processor idle state reduces power consumption to the lowest possible level while retaining the ability to return from the idle state quickly.

It is important to keep in mind that the OEMIdle function does not involve Power Manager. The kernel calls the OEMIdle function directly and it is up to the OAL to switch the hardware into an appropriate idle or sleep state. The kernel passes a DWORD value (dwReschedTime) to OEMIdle to indicate the maximum period of idle time. When this time passes or the maximum delay supported by the hardware timer is reached, the processor switches back to non-idle mode, the previous state is restored, and the scheduler is invoked. If there still is no thread ready to run, the kernel immediately calls OEMIdle again. Driver events, as in the response to user input via keyboard or stylus, may occur at any time and cause the system to stop idling before the system timer starts.

The scheduler is, by default, based on a static timer and system ticks at a one-millisecond frequency. However, the system can optimize power consumption by using dynamic timers and by setting the system timer to the next timeout identified by using the scheduler table content. The processor will then not switch back out of idle mode with every tick. Instead, the processor switches only to non-idle mode after the timeout defined by dwReschedTime expires or an interrupt occurres.

Lesson Summary

Windows Embedded CE 6.0 R2 provides a default Power Manager implementation with a set of power-management APIs that you can use to manage the power state of the system and its devices. It also provides the OEMIdle function, which it executes when the system does not have any threads scheduled in order to provide original equipment manufacturers (OEMs) a chance to put the system into a low power idle state for a specified period of time.

Power Manager is a kernel component that exposes a notification interface, an application interface, and a device interface. It acts as a mediator between kernel and OAL on one side and device drivers and applications on the other side. Applications and device drivers can use the DevicePowerNotify function to control the power state of peripheral devices at five different power levels. Device power states may also associate with default and custom system power states to keep system and devices synchronized. Based on activity times and corresponding events, Power Manger can automatically perform system state transitions. The four default system power states are On, UserIdle, SystemIdle, and Suspend. Customizations for system-state-to-device-state mapping take place in the registry settings of individual devices and device classes.

In addition to Power Manager, the kernel supports power management by means of the OEMIdle function. Switching the processor into idle state reduces power consumption to the lowest possible level while retaining the ability to return from the idle state quickly. The processor will return to non-idle state periodically or when interrupts occur, as when it responds to user input or when a device requests access to memory for data transfer.

You can significantly reduce the power consumption of a device if you implement power management properly using Power Manager and OEMIdle, thereby increasing battery life, decreasing operating costs, and extending device lifetime.

Lab 3: Kiosk Mode, Threads, and Power Management

In this lab, you develop a kiosk application and configure a target device to run this application instead of the standard shell. You then extend this application to run multiple threads in parallel in the application process and analyze the thread execution by using the Remote Kernel Tracker tool. Subsequently, you enable this application for power management.

> **NOTE** Detailed step-by-step instructions
>
> To help you successfully master the procedures presented in this lab, see the document "Detailed Step-by-Step Instructions for Lab 3" in the companion material for this book.

▶ Create a Thread

1. Using the New Project Wizard, create a new WCE Console Application named HelloWorld. Use the Typical Hello_World Application option.
2. Before the _tmain function, implement a thread function named 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. By using the CreateThread function, start a thread:

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

4. Check the returned value of CreateThread to verify that the thread was created successfully.
5. Wait for the thread to reach the end of the thread function and exit:

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

6. Build the run-time image and download it to the target device.
7. Launch Remote Kernel Tracker and analyze how threads are managed on the system.

8. Start the HelloWorld application and follow the thread execution in the Remote Kernel Tracker window, as illustrated in Figure 3-11.

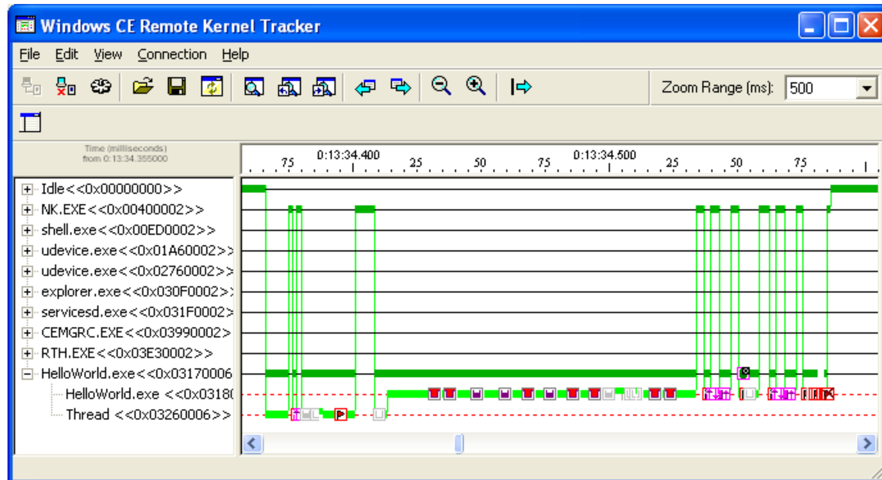


Figure 3-11 Tracking thread execution in Remote Kernel Tracker tool

▶ Enable Power Management Notification Messages

1. Continue to use the HelloWorld application in Visual Studio.
2. Generate power-management notifications in more frequent intervals by going into the subproject registry settings and setting the registry entry for the UserIdle timeout in AC power mode (ACUserIdle) to five seconds:

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

3. In the ThreadProc function, create a message queue object:

```
// 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. Request to receive notifications from Power Manager and check the received messages:

```
// 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. Build the application and rebuild the run-time image.
6. Start the run-time image.
7. You generate user activity by moving the mouse cursor. After five seconds of inactivity, Power Manager should notify the application, as illustrated in Figure 3–12.

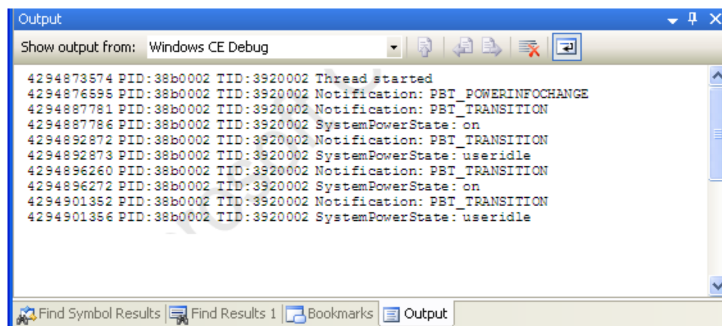


Figure 3-12 Received Power Management notifications

▶ Enable Kiosk Mode

1. Create a WCE Application named Subproject_Shell using the Subproject Wizard. Use the Typical Hello_World Application option.
2. Before the first LoadString line, add a SignalStarted instruction.

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

3. Build the application.
4. Add a registry key in the subproject .reg file to launch the application at startup. Add the following lines, which create the corresponding Launch99 and Depend99 entries:

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

5. Build and start the run-time image.
6. Verify that the Subproject_Shell application starts automatically.

7. Replace the reference to Explorer.exe in the Launch50 registry key with a reference to the Subproject_Shell application, as follows:

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

8. Build and start the run-time image.
9. Verify that the target device runs the Subproject_Shell application in place of the standard shell, as illustrated in Figure 3–13.

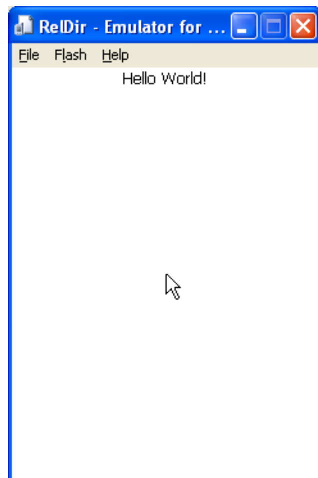


Figure 3-13 Replacing the standard shell with a Subproject_Shell application

Chapter Review

Windows Embedded CE provides a variety of tools, features, and APIs that you can use to ensure optimal system performance and power consumption on your target devices. Performance tools, such as ILTiming, OSBench, and Remote Performance Monitor, can help identify performance issues within and between drivers, applications, and OAL code, such as deadlocks or other issues related to thread synchronization. The Remote Kernel Tracker enables you to examine process and thread execution in great detail, while relying on structured exception handling, which Windows Embedded CE supports natively.

Windows Embedded CE is a componentized operating system. You can include or exclude optional components and even replace the standard shell with a custom application. Replacing the standard shell with an application configured to start automatically lays the groundwork for enabling a kiosk configuration. Windows Embedded CE runs with a black shell in a kiosk configuration, meaning that the user cannot start or switch to other applications on your device.

Regardless of the shell, you can implement power management functions in your device drivers and applications to control energy usage. The default Power Manager implementation covers the typical needs, but OEMs with special requirements add custom logic. The Power Manager source code is included with Windows Embedded CE. The power management framework is flexible and supports any number of custom system power states that can map device power states by means of registry settings.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- ILTiming
- Kiosk Mode
- Synchronization Objects
- Power Manager
- RequestDeviceNotifications

Suggested Practices

Complete the following tasks to help you successfully master the exam objectives presented in this chapter:

Use the ILTiming and OSBench Tools

Use Iltiming and OSBench on the emulator device to examine the emulated ARMV4 processor's performance.

Implement a Custom Shell

Customize the look and feel of the target device by using Task Manager, included in Windows Embedded CE with source code, to replace the shell.

Experiment with Multithreaded Applications and Critical Sections

Use critical section objects in a multithreaded application to protect access to a global variable. Complete the following tasks:

1. Create two threads in the main code of the applications and in the thread functions wait two seconds (Sleep(2000))and three seconds (Sleep(3000)) in an infinite loop. The primary thread of the application should wait until both threads exit by using the WaitForMultipleObjects function.
2. Create a global variable and access it from both threads. One thread should write to the variable and the other thread should read the variable. By accessing the variable before and after the first Sleep and displaying the values, you should be able visualize concurrent access.
3. Protect access to the variable by using a CriticalSection object shared between both threads. Grab the critical section at the beginnings of the loops and release it at the ends of the loops. Compare the results with the previous output.

Chapter 4

Debugging and Testing the System

Debugging and system testing are vital tasks during the software-development cycle, with the ultimate goal to identify and solve software-related and hardware-related defects on a target device. Debugging generally refers to the process of stepping through the code and analyzing debug messages during code execution in order to diagnose root causes of errors. It can also be an efficient tool to study the implementation of system components and applications in general. System testing, on the other hand, is a quality-assurance activity to validate the system in its final configuration in terms of typical usage scenarios, performance, reliability, security, and any other relevant aspects. The overall purpose of system testing is to discover product defects and faults, such as memory leaks, deadlocks, or hardware conflicts, whereas debugging is a means to get to the bottom of these problems and eliminate them. For many developers of small-footprint and consumer devices, locating and eliminating system defects is the hardest part of software development, with a measurable impact on productivity. This chapter covers the debugging and testing tools available in Microsoft® Visual Studio® 2005 with Platform Builder for Microsoft Windows® Embedded CE 6.0 R2 and in the Windows Embedded CE Test Kit (CETK) to help you automate and accelerate these processes so that you can release your systems faster and with fewer bugs. The better you master these tools, the more time you can spend writing code instead of fixing code.

Exam objectives in this chapter:

- Identifying requirements for debugging a run-time image
- Using debugger features to analyze code execution
- Understanding debug zones to manage the output of debug messages
- Utilizing the CETK tool to run default and user-defined tests
- Debugging the boot loader and operating system (OS)

Before You Begin

To complete the lessons in this chapter, you must have the following:

- At least some basic knowledge about Windows Embedded CE software development and debugging concepts.
- A basic understanding of the driver architectures supported in Windows Embedded CE.
- Familiarity with OS design and system configuration concepts.
- A development computer with Microsoft Visual Studio 2005 Service Pack 1 and Platform Builder for Windows Embedded CE 6.0 R2 installed.

Lesson 1: Detecting Software-Related Errors

Software-related errors range from simple typos, uninitialized variables, and infinite loops to more complex and profound issues, such as critical race conditions and other thread synchronization problems. Fortunately, the vast majority of errors are easy to fix once they are located. The most cost-effective way to find these errors is through code analysis. You can use a variety of tools on Windows Embedded CE devices to debug the operating system and step through drivers and applications. A good understanding of these debugging tools will help you accelerate your code analysis so that you can fix software errors as efficiently as possible.

After this lesson, you will be able to:

- Identify important debugging tools for Windows Embedded CE.
- Control debug messages through debug zones in drivers and applications.
- Use the target control shell to identify memory issues.

Estimated lesson time: 90 minutes.

Debugging and Target Device Control

The primary tool to debug and control a Windows Embedded CE target device is by using Platform Builder on the development workstation, as illustrated in Figure 4–1. The Platform Builder integrated development environment (IDE) includes a variety of tools for this purpose, such as a system debugger, CE target control shell (CESH), and debug message (DbgMsg) feature, that you can use to step through code after reaching a breakpoint or to display information on memory, variables, and processes. Moreover, the Platform Builder IDE includes a collection of remote tools, such as Heap Walker, Process Viewer, and Kernel Tracker, to analyze the state of the target device at run time.

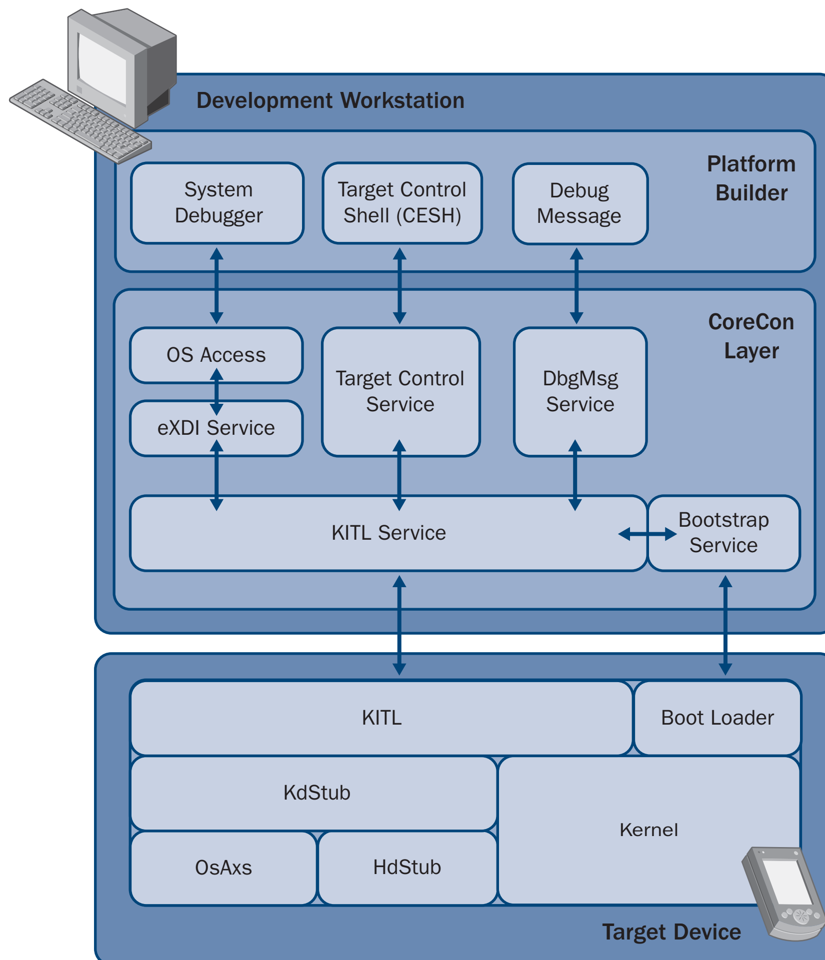


Figure 4-1 CE debugging and target control architecture

In order to communicate with the target device, Platform Builder relies on the Core Connectivity (CoreCon) infrastructure and debugging components deployed on the target device as part of the run-time image. The CoreCon infrastructure provides OS Access (OsAxS), target control, and DbgMsg services to Platform Builder on one side, and interfaces with the target device through the Kernel Independent Transport Layer (KITL) and the bootstrap service on the other side. On the target device itself, the debugging and target control architecture relies on KITL and the boot loader for communication purposes. If the run-time image includes debugging components, such as the kernel debugger stub (KdStub), hardware debugger stub (HdStub), and the OsAxS library, you can use Platform Builder to obtain kernel run-time information

and perform just-in-time (JIT) debugging. Platform Builder also supports hardware-assisted debugging through Extended Debugging Interface (eXDI), so that you can debug target device routines prior to loading the kernel.

Kernel Debugger

The Kernel Debugger is the CE software-debugging engine to debug kernel components and CE applications. On the development workstation, you work directly in Platform Builder, such as to insert or remove breakpoints in the source code and run the application, yet you must include support for KITL and debugging libraries (KdStub and OsAxS) in the run-time image so that Platform Builder can capture debugging information and control the target device. Lesson 2, "Configuring the Run-Time Image to Enable Debugging," later in this chapter provides detailed information about system configurations for kernel debugging.

The following target-side components are essential for kernel debugging:

- **KdStub** Captures exceptions and breakpoints, retrieves kernel information, and performs kernel operations.
- **OsAxS** Retrieves information about the state of the operating system, such as information about memory allocations, active processes and threads, proxies, and loaded dynamic-link libraries (DLLs).

> **NOTE Application debugging in Windows Embedded CE**
>
> By using the Kernel Debugger, you can control the entire run-time image as well as individual applications. However, KdStub is a kernel component that receives first-chance and second-chance exceptions, as explained in Chapter 3, "Performing System Programming." If you stop the Kernel Debugger during a session without stopping the target-side KdStub module first and an exception occurs, the run-time image stops responding until you reconnect the debugger, because the Kernel Debugger must handle the exception so that the target device can continue to run.

Debug Message Service

In Platform Builder, when you attach to a KITL–enabled and KdStub–enabled target device, you can examine debug information in the Output window of Microsoft Visual Studio 2005, which Platform Builder obtains from the target device by using the DbgMsg service in the CoreCon infrastructure. Debug messages provide detailed information about the running processes, signal potentially critical issues, such as invalid input, and give hints about the location of a defect in the code that you can

then study further by setting a breakpoint and stepping through the code in Kernel Debugger. One of the kernel debugger stub's features is support for dynamic management of debug messages, so you can configure the debugging verbosity without source code modifications. Among other things, you can exclude Timestamps, Process IDs, or Thread IDs, if you display the Debug Message Options window that you can reach through the Target menu in Visual Studio. You can also send the debug output to a file for analysis in a separate tool. On the target device, all debug messages are sent directly to the default output stream handled through the NKDbgPrintf function.

NOTE Debug messages with and without KITL

When both Kernel Debugger and KITL are enabled, the debug messages are displayed in the Output window of Visual Studio. If KITL is not available, the debug information is transferred from the target device to the development computer over a serial port configured and used by the OEM adaptation layer (OAL).

Macros for Debug Messages

To generate debug information, Windows Embedded CE provides several debugging macros that generally fall into two categories, debug macros and retail macros. Debug macros output information only if the code is compiled in the debug build configuration (environment variable WINCEDEBUG=debug), while retail macros generate information in both debug and retail build configurations (WINCEDEBUG=retail) unless you build the run-time image in ship configuration (WINCESHIP=1). In ship configuration, all debugging macros are disabled.

Table 4–1 summarizes the debugging macros that you can insert in your code to generate debug information.

Table 4-1 Windows Embedded CE macros to output debugging messages

Macro	Description
DEBUGMSG	Conditionally prints a printf-style debug message to the default output stream (that is, the Output window in Visual Studio or a specified file) if the run-time image is compiled in debug build configuration.

Table 4-1 Windows Embedded CE macros to output debugging messages (Continued)

Macro	Description
RETAILMSG	Conditionally prints a printf-style debug message to the default output stream (that is, the Output window in Visual Studio or a specified file) if the run-time image is compiled in debug or release build configuration, yet not in ship build configuration.
ERRORMSG	Conditionally prints additional printf-style debug information to the default output stream (that is, the Output window in Visual Studio or a specified file) if the run-time image is compiled in debug or release build configuration, yet not in ship build configuration. This error information includes the name of the source code file and the line number, which can help to quickly locate the line of code that generated the message.
ASSERTMSG	Conditionally prints a printf-style debug message to the default output stream (that is, the Output window in Visual Studio or a specified file) and then breaks into the debugger, if the run-time image is compiled in debug configuration. In fact, ASSERTMSG calls DEBUGMSG followed by DBGCHK.
DEBUGLED	Conditionally passes a WORD value to the WriteDebugLED function, if the run-time image is compiled in debug build configuration. This macro is useful on devices that provide only light-emitting diodes (LEDs) to indicate the system status and requires an implementation of the OEMWriteDebugLED function in the OAL.
RETAILLED	Conditionally passes a WORD value to the WriteDebugLED function, if the run-time image is compiled in debug or release build configuration. This macro is useful on devices that provide only LEDs to indicate the system status and requires an implementation of the OEMWriteDebugLED function in the OAL.

Debug Zones

Debug messages are particularly useful tools to analyze multi-threaded processes, especially thread synchronization and other timing issues that are difficult to detect by stepping through the code. However, the number of debug messages generated on

a target device can be overwhelming if you heavily use debugging macros in your code. To control the amount of information generated, debugging macros enable you to specify a conditional expression. For example, the following code outputs an error message if the dwCurrentIteration value is greater than the maximum possible value.

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

In the example above, ERRORMSG outputs debugging information whenever dwCurrentIteration is greater than dwMaxIteration. You can also control debugging messages by using debug zones in the conditional statement. This is particularly useful if you want to use the DEBUGMSG macro to examine code execution in a module (that is, an executable file or a DLL) at varying levels without changing and recompiling the source code each time. First, you must enable debug zones in your executable file or DLL, and register a global DBGPARAM variable with the Debug Message service to specify which zones are active. You can then specify the current default zone programmatically or through registry settings on the development workstation or the target device. It is also possible to control debug zones dynamically for a module in Platform Builder via CE Debug Zones on the Target menu or in the Target Control window.

TIP Bypassing debug zones

You can bypass debug zones in drivers and applications if you pass a Boolean variable to the DEBUGMSG and RETAILMSG macros that you can set to TRUE or FALSE when you rebuild the run-time image.

Zones Registration

To use debug zones, you must define a global DBGPARAM variable with three fields that specify the module name, the names of the debug zones you want to register, and a field for the current zone mask, as summarized in Table 4–2.

Table 4-2 DBGPARAM elements

Field	Description	Example
lpszName	Defines the name of the module with a maximum length of 32 characters.	`TEXT("ModuleName")`
rglpszZones	Defines an array of 16 names for the debug zones. Each name can be up to 32 characters long. Platform Builder displays this information to the user when selecting the active zones in the module.	`{ 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	The current zone mask used in the DEBUGZONE macro to determine the currently selected debug zone.	`MASK_INIT | MASK_ON | MASK_ERROR`

NOTE Debug zones

Windows Embedded CE supports a total of 16 named debug zones, yet not all have to be defined if the module doesn't require them. Each module uses a separate set of zone names that should clearly reflect the purpose of each implemented zone.

The Dbgapi.h header file defines the DBGPARAM structure and debugging macros. Because these macros use a predefined DBGPARAM variable named dpCurSettings, it is important that you use the same name in your source code as well, as illustrated in the following code snippet.

```
#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;
}
```

Zone Definitions

The sample code above registers six debug zones for the module that you can now use in conjunction with conditional statements in debugging macros. The following line of code shows one possible way to do this:

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

If the debug zone is currently set to MASK_ERROR, the conditional expression evaluates to TRUE and DEBUGMSG sends the information to the debug output stream. However, to improve the readability of your code, you should use the DEBUGZONE macro defined in Dbgapi.h, as illustrated in the following code snippet, to define flags for your zones. Among other things, this approach simplifies the combination of debug zones through logical AND and OR operations.

```
#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")));
```

Enabling and Disabling Debug Zones

The DBGPARAM field ulZoneMask is the key to setting the current debug zone for a module. You can accomplish this programmatically in the module by changing the ulZoneMask value of the global dpCurSettings variable directly. Another option is to change the ulZoneMask value in the debugger at a breakpoint within the Watch window. You can also control the debug zone through another application by calling the SetDbgZone function. Another option available at run time is to use the Debug Zones dialog box, illustrated in Figure 4–2, which you can display in Visual Studio with Platform Builder via the CE Debug Zones command on the Target menu.

The Name list shows the modules running on the target device that support debug zones. If the selected module is registered with the Debug Message service, you can find the list of 16 zones displayed under Debug Zones. The names correspond to the selected module's dpCurSettings definition. You can select or deselect zones to enable or disable them. By default, the zones defined in the dpCurSettings variable are enabled and checked in the Debug Zones list. For modules not registered with the Debug Message service, the Debug Zone list is deactivated and unavailable.

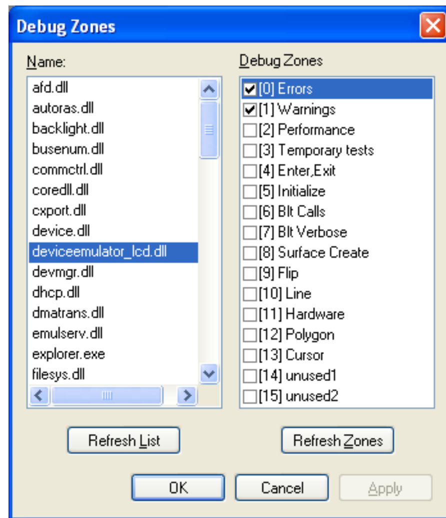


Figure 4-2 Setting debug zones in Platform Builder

Overriding Debug Zones at Startup

Windows Embedded CE enables the zones you specify in the dpCurSettings variable when you start the application or load the DLL into a process. At this point, it is not yet possible to change the debug zone unless you set a breakpoint and change the ulZoneMask value in the Watch window. However, CE supports a more convenient method through registry settings. To facilitate loading a module with different active debug zones, you can create a REG_DWORD value with a name that corresponds to the module name specified in the lpszName field of the dpCurSettings variable and set it to the combined values of the debug zones you want to activate. You can configure this value on the development workstation or the target device. It is generally preferable to configure this value on the development workstation because changing target device registry entries requires you to rebuild the run-time image, whereas a modification of the registry entries on the development workstation only requires you to restart the affected modules.

Table 4–3 illustrates the configuration for a sample module called *ModuleName*. Make sure you replace this placeholder name with the actual name of your executable file or DLL.

Table 4-3 Startup registry parameter examples

Location	Development Workstation	Target Device
Registry Key	HKEY_CURRENT_USER \Pegasus\Zones	HKEY_LOCAL_MACHINE \DebugZones
Entry Name	ModuleName	ModuleName
Type	REG_DWORD	REG_DWORD
Value	0x00000001 - 0x7FFFFFFF	0x00000001 - 0x7FFFFFFF
Comments	The Debug Message system uses the target-side value for a module only if the development workstation is unavailable or if the development-side registry does not contain a value for the module.	

NOTE Enabling all debug zones

Windows Embedded CE uses the lower 16 bits of the REG_DWORD value to define named debug zones for application debugging purposes. The remaining bits are available to define unnamed debug zones, with the exception of the highest bit, which is reserved for the kernel. Therefore, you should not set a module's debug zone value to 0xFFFFFFFF. The maximum value is 0x7FFFFFFF, which enables all named and unnamed debug zones.

MORE INFO Pegasus registry key

The name Pegasus refers to the code name of the first Windows CE version that Microsoft released for handheld personal computers and other consumer electronics in 1996.

Best Practices

When working with debug messages, keep in mind that heavy use of debug messages slows down code execution. Perhaps even more important, the system serializes the debug output operations, which can provide an unintentional thread synchronization mechanism. For example, multiple threads running unsynchronized in release builds might cause issues not noticeable in debug builds.

When working with debug messages and debug zones, consider the following best practices:

- **Use Conditional statements** Use debug macros with conditional statements based on debug zones. Do not use DEBUGMSG(TRUE). Also avoid using retail macros without conditional statements, such as RETAILMSG(TRUE), although some model device driver (MDD)/platform dependent driver (PDD) drivers must use this technique.
- **Exclude debugging code from release builds** If you only use debug zones in debug builds, include the global variable dpCurSettings and zone mask definitions in #ifdef DEBUG #endif guards and restrict the use of debug zones to debug macros (such as DEBUGMSG).
- **Use retail macros in release builds** If you also want to use debug zones in release builds, include the global variable dpCurSettings and zone mask definitions in #ifndef SHIP_BUILD #endif guards and replace the call to DEBUGREGISTER with a call to RETAILREGISTERZONES.
- **Clearly identify the module name** If possible, set the dpCurSettings.lpszName value to the module's file name.
- **Limit verbosity by default** Set the default zones for your drivers to ZONE_ERROR and ZONE_WARNING only. When bringing up a new platform, enable ZONE_INIT.
- **Restrict the error debug zone to unrecoverable issues** Use ZONE_ERROR only when your module or significant functionality fails due to incorrect configuration or other issues. Use ZONE_WARNING for recoverable issues.
- **Eliminate all errors and warnings if possible** Your module should be able to load without any ZONE_ERROR or ZONE_WARNING messages.

Target Control Commands

The Target Control service provides access to a command shell for the debugger to transfer files to the target device and debug applications. This target control shell, displayed in Figure 4-3, is accessible from within Visual Studio with Platform Builder via the Target Control option on the Target menu. However, it is important to keep in mind that the target control shell is only available if the Platform Builder instance is attached to a device through KITL.

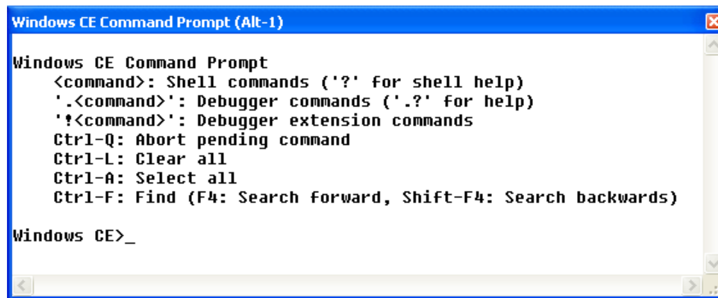


Figure 4-3 The target control shell

Among other things, the target control shell enables you to perform the following debugging actions:

- Break into the Kernel Debugger (**break** command).
- Send a memory dump to the debug output (**dd** command) or to a file (**df** command).
- Analyze memory usage for the kernel (**mi kernel** command) or the entire system (**mi full** command).
- List processes (**gi proc** command), threads (**gi thrd** command), and thread priorities (**tp** command), as well as the modules loaded on the system (**gi mod** command).
- Launch processes (**s** command) and end processes (**kp** command).
- Dump the processes heap (**hp** command).
- Enable or disable the system profiler (**prof** command).

NOTE Target control commands

For a complete list of target control commands, see the section "Target Control Debugging Commands" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN® Web site at *http://msdn2.microsoft.com/en-us/library/aa936032.aspx*.

Debugger Extension Commands (CEDebugX)

In addition to the regular debugger commands, the Target Control service provides the debugger with a debugger commands extension (CEDebugX) to increase the efficiency of kernel and application debugging. This extension provides additional features to detect memory leaks and deadlocks and diagnose the overall health of the

system. The additional commands are accessible through the target control shell and start with an exclamation point (!).

To use CEDebugX, you need to break into the Kernel Debugger by using the **break** command in the target control shell or the Break All command on the Target menu in Visual Studio. Among other things, you can then enter a **!diagnose all** command to identify the potential reason for a failure, such as heap corruption, deadlocks, or memory starvation. On a healthy system, the CEDebugX should detect no any issues, as illustrated in Figure 4–4.

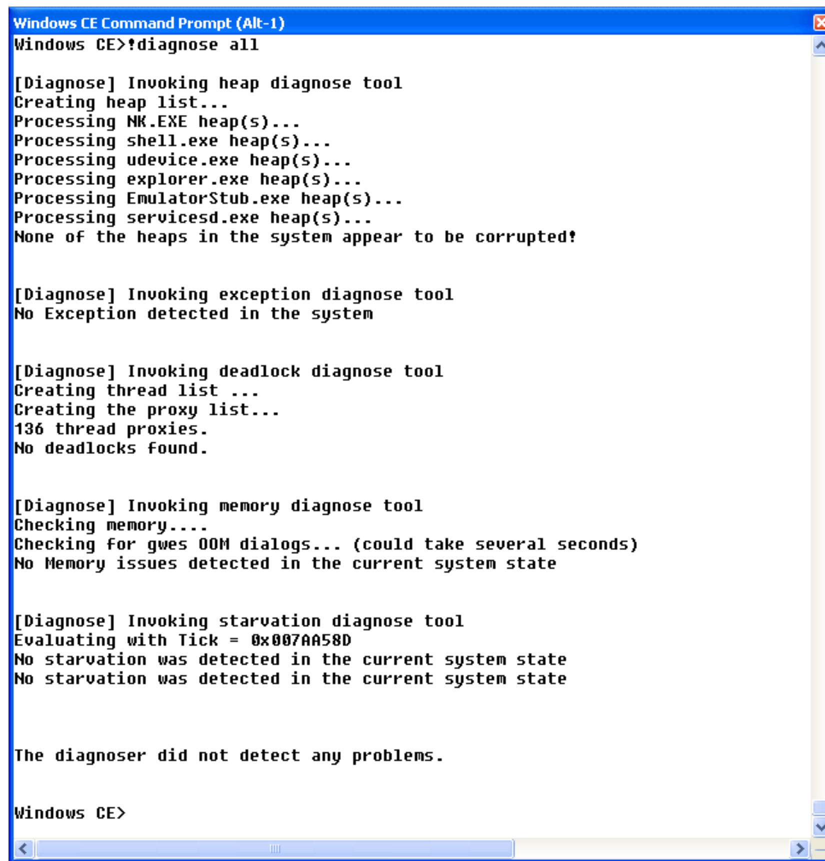


Figure 4-4 Diagnosing a run-time image with CEDebugX

The **!diagnose all** command runs the following diagnostics:

- **Heap** Diagnoses all the heap objects of all processes on the system to identify potential content corruption.
- **Exception** Diagnoses an exception that occurs on the system and is able to provide details on the exception, such as process and thread ID, and PC address at the exception time.
- **Memory** Diagnoses the system memory to identify potential memory corruptions and low memory conditions.
- **Deadlock** Diagnoses the thread states and system objects (see Chapter 3 for more details on thread synchronization). It can provide a list of system objects and thread IDs that generated the deadlock.
- **Starvation** Diagnoses threads and system objects to identify potential thread starvation. Starvation occurs when a thread is never scheduled on the system by the scheduler because the scheduler is busy with higher-priority threads.

Advanced Debugger Tools

The target control shell and CEDebugX commands enable you to perform a thorough analysis of a running system or a CE dump file (if you select the CE Dump File Reader as the debugger to perform postmortem debugging), yet you are not restricted to the command-line interface. Platform Builder includes several graphical tools with a dedicated purpose to increase your debugging efficiency. You can access these advanced debugger tools in Visual Studio via the Debug menu when you open the Windows submenu.

The Platform Builder IDE includes the following advanced debugger tools:

- **Breakpoints** Lists the breakpoints enabled on the system and provides access to the breakpoint properties.
- **Watch** Provides read and write access to local and global variables.
- **Autos** Provides access to variables similar to the Watch window, except that the debugger creates this list of variables dynamically, while the Watch window lists all manually added variables whether they are accessible or not. The Autos window is useful if you want to check the parameter values passed to a function.
- **Call Stack** Accessible only when the system is in a break state (code execution has halted on a breakpoint). This window provides a list of all processes enabled on the system and a list of hosted threads.

- **Threads** Provides a list of the threads running in the processes on the system. This information is dynamically retrieved and can be updated at any time.
- **Modules** Lists the modules loaded and unloaded on the system and provides the memory address where those modules are loaded. This feature is useful for identifying whether a driver DLL is actually loaded or not.
- **Processes** Similar to the Threads window, this window provides a list of the processes running on the system. Among other things, you can terminate processes if required.
- **Memory** Provides direct access to device memory. You can use memory addresses or variable names to locate the desired memory content.
- **Disassembly** Reveals the assembly code of the current code line executed on the system.
- **Registers** Provides access to the CPU register values when running a specific line of code.
- **Advanced Memory** Can be used to search the device memory, move portions of memory to different sections, and fill memory ranges by using content patterns.
- **List Nearest Symbols** Determines a specific memory address for the nearest symbols available in the binaries. It also provides the complete path to the file containing the symbol. This tool is useful to locate the name of a function that generated an exception.

CAUTION Memory corruption

The Memory and Advanced Memory tools can modify memory content. Using these tools incorrectly can cause system failures and damage the operating system on the target device.

Application Verifier Tool

Another useful tool to identify potential application compatibility and stability issues and necessary source code–level fixes is the Application Verifier tool, included in the CETK. This tool can attach to an application or a DLL to diagnose problems that are otherwise difficult to track on standalone devices. The Application Verifier tool does not require a device connection to a development workstation and can be launched at system startup to check and validate drivers and system applications. You can also start this tool from the CETK user interface or manually on the target device. If you

want to use the Application Verifier tool outside of the CETK, you should use the Getappverif_cetk.bat file to copy all the required files into the release directory.

NOTE Application Verifier tool documentation

For detailed information about the Application Verifier tool, including how to use shim extension DLLs to run custom test code or change the behavior of functions during application testing, see the section "Application Verifier Tool" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa934321.aspx*.

CeLog Event Tracking and Processing

Windows Embedded CE includes an extensible event-tracking system that you can include in a run-time image to diagnose performance problems. The CeLog event-tracking system logs a set of predefined kernel and coredll events related to mutexes, events, memory allocation, and other kernel objects. The extensible architecture of the CeLog event-tracking system also enables you to implement custom filters to track user-defined events. For platforms connected to a development workstation through KITL, the CeLog event-tracking system can selectively log events based on zones specified in the ZoneCE registry entry, as summarized in Table 4–4.

Table 4-4 CeLog registry parameters for event logging zones

Location	HKEY_LOCAL_MACHINE\System\CELog
Registry Entry	ZoneCE
Entry Type	REG_DWORD
Value	*<Zone IDs>*
Description	By default, all zones are logged. For a list of all possible zone ID values, see the section "CELog Zones" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa909194.aspx*.

By using the CeLog event-tracking system, you can collect data, which CeLog stores in a buffer in RAM on the target device. Performance tools, such as Remote Kernel Tracker and Readlog, can then process the collected data. It is also possible to flush the data periodically to a file by using the CELogFlush tool.

> **NOTE CELog and ship builds**
>
> You should not include the CeLog event-tracking system in final builds to avoid performance and memory penalties due to CeLog activities, and to reduce the attack surface through which a malicious user could try to compromise the system.

Remote Kernel Tracker

The Remote Kernel Tracker tool enables you to monitor system activities on a target device based on processes and threads. This tool can display information from the target device in real time through KITL, yet it is also possible to use Remote Kernel Tracker offline based on CeLog data files. You can find more information about the Remote Kernel Tracker tool in Chapter 3, “Performing System Programming.”

Figure 4–5 shows Kernel Tracker on a target device collecting information about thread activities.

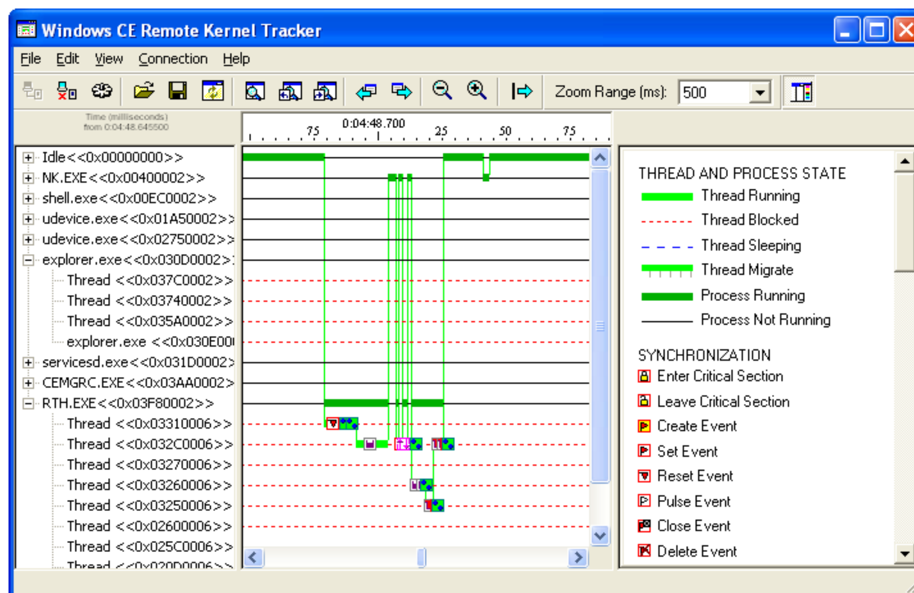


Figure 4-5 Thread information in Kernel Tracker

CeLogFlush Tool

To create CeLog data files, use the CeLogFlush tool to save the CeLog event data buffered in RAM into a .clg file. This file can be located in the RAM file system, persistent storage, or the release file system on a development workstation. To

minimize data loss due to buffer overruns, you can specify a larger RAM buffer and increase the frequency at which CeLog flushes the buffer. You can optimize the performance if you keep the file open to avoid repeated file open and close operations and store the file in the RAM file system instead of a slower persistent storage medium.

NOTE **CELogFlush configuration**

For detailed information about the CeLogFlush tool, including how to configure this tool through registry settings, see the section "CeLogFlush Registry Settings" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa935267.aspx*.

Readlog Tool

In addition to the graphical Remote Kernel Tracker application, you can process CELog data files by using the Readlog tool, located in the %_WINCEROOT%\Public\Common\Oak\Bin\i386 folder. Readlog is a command-line tool to process and display information not exposed in Remote Kernel Tracker, such as debug messages and boot events. It is often useful to analyze system activities in Remote Kernel Tracker first and then focus on an identified process or thread with the Readlog tool. The raw data that the CeLogFlush tool writes into the .clg file is ordered by zones to facilitate locating and extracting specific information. You can also filter the data and extend filtering capabilities based on extension DLLs to process custom data captured through the custom events collector.

One of the most useful Readlog scenarios is to replace thread start addresses (the functions passed to the CreateThread call) in CeLog data files with the names of the actual thread functions to facilitate system analysis in Remote Kernel Tracker. To accomplish this task, you must start Readlog with the **-fixthreads** parameter (**readlog -fixthreads**). Readlog looks up the symbol .map files in the release directory to identify the thread functions based on the start addresses and generates new logs with the corresponding references.

Figure 4–6 shows CeLog data in Remote Kernel Tracker, captured through the CeLog event-tracking system, flushed to a .clg file with the CeLogFlush tool, and prepared for a more user-friendly display of the information by using the Readlog application with the **-fixthreads** parameter.

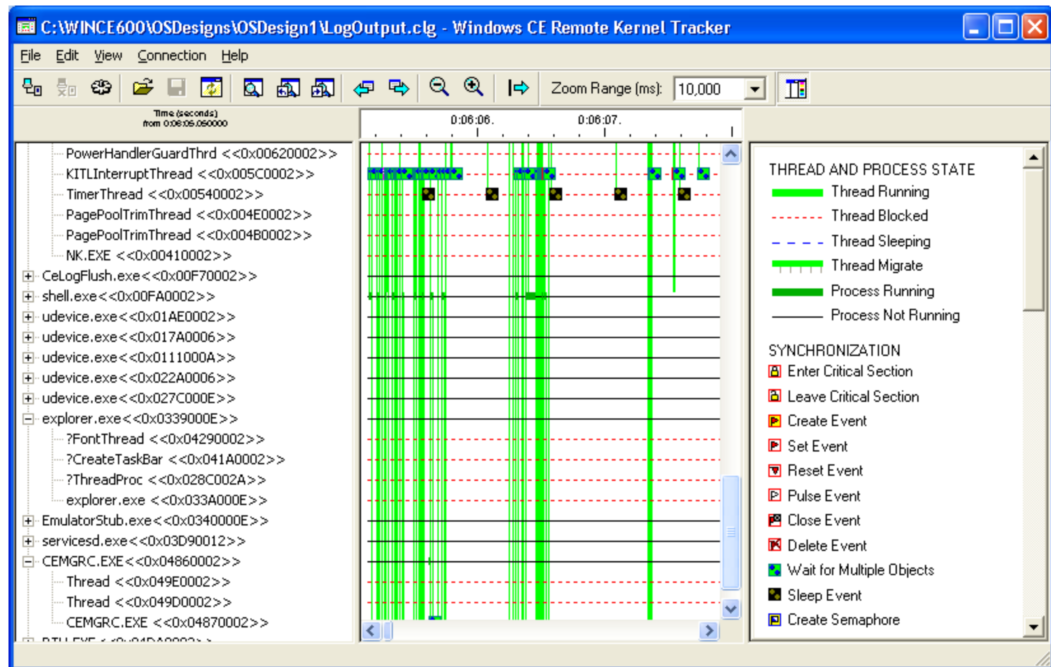


Figure 4-6 A CeLog data file prepared with **readlog -fixthreads** and opened in Remote Kernel Tracker

> **NOTE Improving reference naming matching**
>
> The CeLog event-tracking system can take advantage of the kernel profiler to look up thread function names based on start addresses when capturing CreateThread events, if you explicitly enable the kernel profiler and add profiling symbols to the run-time image by rebuilding the image with the IMGPROFILER environment variable set. However, CeLog can only look up the profiling symbols built into the run-time image. Symbols of applications developed based on a Software Development Kit (SDK) are generally unavailable to the CeLog event-tracking system.

Lesson Summary

Debugging an operating system and applications requires familiarity with both the CE system and the debugging tools included in Platform Builder and CETK. The most important debugging tools are the system debugger, debug message feature, and CE target control shell. The system debugger enables you to set breakpoints and step through kernel and application code, while the debug message feature provides the option to analyze system components and applications without interrupting code execution. A variety of debug and retail macros are available to output debugging

information from target devices with or without a display component. Because systems and applications can potentially generate a large number of debug messages, you should use debug zones to control the output of debugging information. The key advantage of debug zones is that you can change the debug information verbosity dynamically without having to rebuild the run-time image. The target control shell, on the other hand, enables you to send commands to the target device, such as a **break** command followed by a **!diagnose all** command to break into the debugger and perform a CEDebugX check on the overall system health, including memory leaks, exceptions, and deadlocks.

Apart from these core debugging tools, you can use typical CE configuration and troubleshooting tools, such as the Application Verifier tool, to identify potential application compatibility and stability issues, and Remote Kernel Tracker to analyze processes, threads, and system performance. Remote Kernel Tracker relies on the CeLog event-tracking system, which typically maintains logged data in memory on the target device; you can also flush this data to a file by using the CeLogFlush tool. If symbol files are available for the modules that you want to analyze, you can use the Readlog tool to replace the thread start addresses with the actual function names and generate new CeLog data files for more convenient offline analysis in Remote Kernel Tracker.

Lesson 2: Configuring the Run-Time Image to Enable Debugging

The debugging features of Windows Embedded CE rely on development workstation components and the target device, and require specific settings and hardware support. Without a connection between the development workstation and the target device, debug information and other requests cannot be exchanged. If this communication link breaks—for example, because you stop the debugger on the development workstation without first unloading the target-side debugging stub—the run-time image might stop responding to user input while waiting for the debugger to resume code execution after an exception occurred.

After this lesson, you will be able to:

- Enable the Kernel Debugger for a run-time image.
- Identify the KITL requirements.
- Use the Kernel Debugger in a debugging context.

Estimated lesson time: 20 minutes.

Enabling the Kernel Debugger

As discussed in Lesson 1, the development environment for Windows Embedded CE 6.0 includes a Kernel Debugger that enables developers to step through and interact with code running on a CE target device. This debugger requires you to set kernel options and a communication layer between the target device and the development computer.

OS Design Settings

To enable an OS design for debugging, you must unset the environment variables IMGNODEBUGGER and IMGNOKITL so that Platform Builder includes the KdStub library and enables the KITL communication layer in the Board Support Package (BSP) when building the run-time image. Platform Builder provides a convenient method to accomplish this task. In Visual Studio, right-click the OS design project and select Properties to display the OS Design property pages dialog box, switch to the Build Options pane, and then select the Enable Kernel Debugger and Enable KITL check boxes. Chapter 1, "Customizing the Operating System Design," discusses the OS Design property pages dialog box in more detail.

Selecting a Debugger

Having enabled KdStub and KITL for a run-time image, you can select a debugger to analyze the system on the target device in the communication parameters for your target device. To configure these parameters, display the Target Device Connectivity Options dialog box in Visual Studio by opening the Target menu and selecting Connectivity Options, as explained in Chapter 2, "Building and Deploying a Run-Time Image."

By default, no debugger is selected in the connectivity options. You have the following debugger choices:

- **KdStub** This is the software debugger for the kernel and applications to debug system components, drivers, and applications running on a target device. KdStub requires KITL to communicate with Platform Builder.
- **CE Dump File Reader** Platform Builder provides you with an option to capture dump files, which you can then open by using the CE dump-file reader. Dump files enable you to study the state of a system at a particular point in time and are useful as references.
- **Sample Device Emulator eXDI 2 Driver** KdStub cannot debug routines that the system runs prior to loading the kernel, nor can it debug interrupt service routines (ISRs), because this debugging library relies on software breakpoints. For hardware-assisted debugging, Platform Builder includes a sample eXDI driver that you can use in conjunction with a joint test action group (JTAG) probe. The JTAG probe enables you to set hardware breakpoints handled by the processor.

NOTE Hardware-assisted debugging

For detailed information about hardware-assisted debugging, see the section "Hardware-assisted Debugging" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa935824.aspx*.

KITL

As illustrated in Figure 4–1 at the beginning of this chapter, KITL is an essential communication layer between the development computer and the target device and must be enabled for Kernel Debugger support. As the name implies, KITL is completely hardware independent and works over network connections, serial cables, Universal Serial Bus (USB), or any other supported communication

mechanism, such as Direct Memory Access (DMA). The only requirement is that both sides (development computer and target device) support and use the same interface. The most common and fastest KITL interface for the device emulator is DMA, as illustrated in Figure 4-7. For target devices with a supported Ethernet chip, it is typically best to use the network interface.

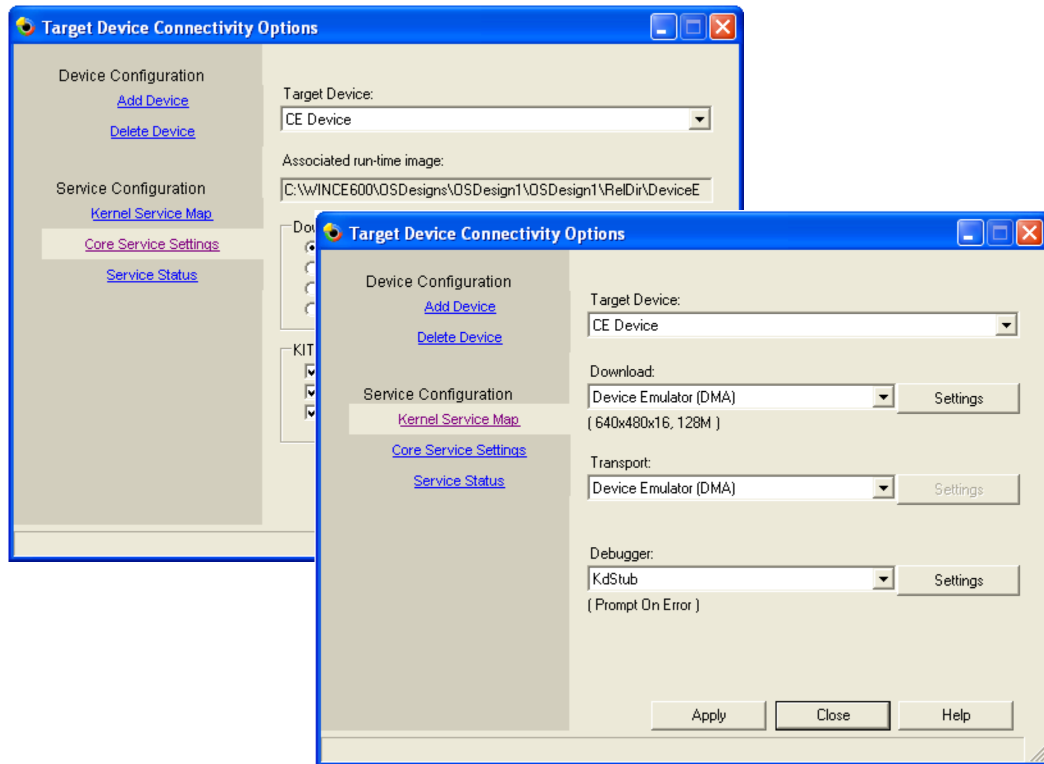


Figure 4-7 Configuring the KITL communication interface

KITL supports the following two methods of operation:

- **Active mode** By default, Platform Builder configures KITL to connect to the development computer during the start process. This setting is most useful for kernel and application debugging during the software-development cycle.
- **Passive mode** By clearing the check box Enable KITL on Device Boot, you can configure KITL for passive mode, meaning Windows Embedded CE initializes the KITL interface, but KITL does not establish a connection during the startup process. If an exception occurs, KITL makes an attempt to establish a connection

to the development computer so that you can perform JIT debugging. Passive mode is most useful when working with mobile devices that do not have a physical connection to the development computer at startup.

NOTE KITL modes and boot arguments

The Enable KITL on Device Boot setting is a boot argument (BootArgs) that Platform Builder configures for the boot loader. For more information about boot loaders and their advantages during the BSP development process, see the section "Boot Loaders" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa917791.aspx*.

Debugging a Target Device

It is important to keep in mind that development-side and target-side debugger components run independently of each other. For example, it is possible to run the Kernel Debugger in Visual Studio 2005 with Platform Builder without having an active target device. If you open the Debug menu and click Start or press the F5 key, the Kernel Debugger starts and informs you in the Output window that it is waiting for a connection to the target device. On the other hand, if you start a debugging-enabled run-time image without an active KITL connection to a debugger and an exception occurs, the run-time image appears to hang because the system halts, waiting for control requests from the debugger, as mentioned earlier in this chapter. For this reason, the debugger typically starts automatically when you attach to a debugging-enabled target device. Instead of pressing F5 to start a debugging session, use Attach Device on the Target menu.

Enabling and Managing Breakpoints

The debugging features of Platform Builder provide most of the functionality also found in other debuggers for Windows desktop applications. You can set breakpoints, step through the code line-by-line, and use the Watch window to view and change variable values and object properties, as illustrated in Figure 4–8. Keep in mind, however, that the ability to use breakpoints depends on the existence of the KdStub library in the run-time image.

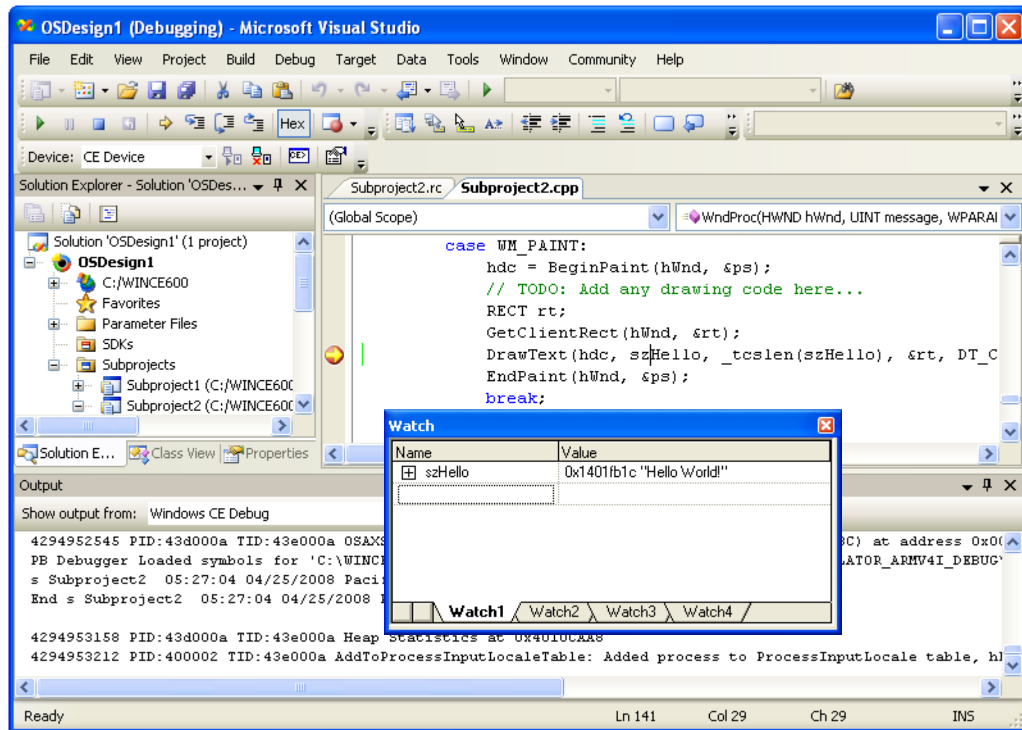


Figure 4-8 Debugging a Hello World application

To set a breakpoint, use the Toggle Breakpoint option on the Debug menu in Visual Studio. Alternatively, you can press F9 to set a breakpoint at the current line or click the left margin area of the code line. According to your selection, Platform Builder indicates the breakpoint with a red dot or a red circle, depending on whether the debugger can instantiate the breakpoint or not. The red circle indicates an un-instantiated breakpoint. Un-instantiated breakpoints occur if the Visual Studio instance is not linked to the target code, the breakpoint is set but has not yet been loaded, the debugger is not enabled, or if the debugger is running but code execution has not yet halted. If you set a breakpoint while the debugger is running, then the device must break into the debugger first before the debugger can instantiate the breakpoint.

You have the following options to manage breakpoints in Visual Studio with Platform Builder:

- **Source code, Call Stack, and Disassembly windows** You can set, remove, enable, or disable a breakpoint by either pressing F9 and selecting Toggle

Breakpoint from the Debug menu or selecting Insert/Remove Breakpoint from the context menu.

- **New Breakpoint dialog box** You can display this dialog box via the submenus available under New Breakpoint on the Debug menu. The New Breakpoint dialog box enables you to set breakpoints by location and conditions. The debugger stops at a conditional breakpoint only if the specified condition evaluates to TRUE, such as when a loop counter or other variable has a specific value.
- **Breakpoints window** You can display the Breakpoints window by clicking Breakpoints under the Windows submenu on the Debug menu, or by pressing Alt+F9. The Breakpoints window lists all set breakpoints and enables you to configure breakpoint properties. For example, instead of using the New Breakpoint dialog box, which requires you to specify location information manually, you can set the desired breakpoint directly in the source code and then display the properties of this breakpoint in the Breakpoints window to define conditional parameters.

TIP Too many breakpoints

Use breakpoints sparingly. Setting too many breakpoints and constantly selecting Resume impacts debugging efficiency and makes it hard to focus on one aspect of the system at a time. Consider disabling and re-enabling breakpoints as necessary.

Breakpoint Restrictions

When configuring the properties of a breakpoint in the New Breakpoint dialog box or the Breakpoints window, you may notice a Hardware button, which you can use to configure the breakpoint as a hardware breakpoint or software breakpoint. You cannot use software breakpoints in OAL code or interrupt handlers, because the breakpoint must not completely halt the execution of the system. Among other system processes, the KITL connection must remain active, because it is the only way to communicate with the debugger on the development workstation. KITL interfaces with the OAL and uses the kernel's interrupt-based communication mechanisms. If you set a breakpoint in an interrupt handler function, then the system will not be able to communicate any longer when the breakpoint is reached because interrupt handling is a single-threaded and non-interruptible function.

If you must debug interrupt handlers, you can use debug messages or hardware breakpoints. However, hardware breakpoints require an eXDI-compliant debugger

(such as JTAG Probe) to register the interrupt in the processor's debug register. Typically, only one hardware debugger can be enabled on a processor at a time, although JTAG can manage multiple debuggers. You cannot use the KdStub library for hardware-assisted debugging.

To configure a hardware breakpoint, follow these steps:

1. Open the Breakpoint window by opening the Debug menu and then clicking Breakpoint.
2. Select the breakpoint in the breakpoint list and then right-click it.
3. Click Breakpoint Properties to display the Breakpoint Properties dialog box and then click the Hardware button.
4. Select the Hardware radio button and then click OK twice to close all dialog boxes.

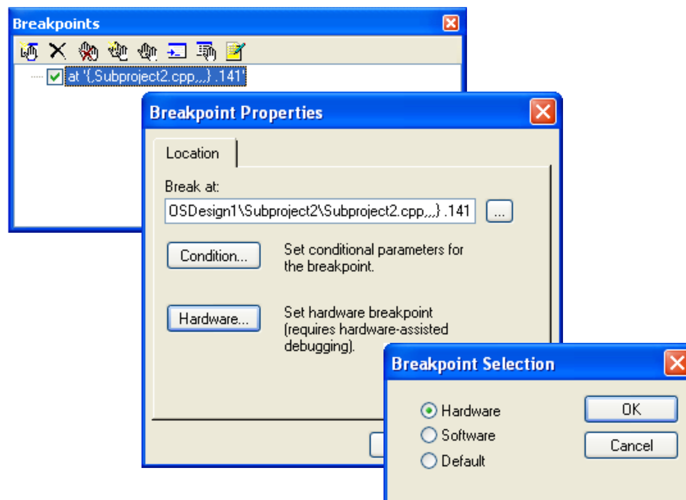


Figure 4-9 Setting a hardware breakpoint

Lesson Summary

Enabling the debugger is a straightforward configuration process in the Platform Builder IDE if you include KITL and debugger libraries in the run-time image. You can then display the Target Device Connectivity Options dialog box and select an appropriate transport and debugger. Typically, the transport is DMA or Ethernet, but you can also use a USB or serial cable to connect the development workstation to the target device.

Platform Builder provides most of the debugging features also found in other debuggers for Windows desktop applications. You can set breakpoints, step through the code line-by-line, and use the Watch window to view and change variable values and object properties. Platform Builder also supports conditional breakpoints to halt code execution according to specified criteria. The debugger of choice for software debugging is KdStub, although you can also use an eXDI driver with Platform Builder for hardware-assisted debugging based on a JTAG probe or other hardware debugger. Hardware-assisted debugging enables you to analyze system routines that run prior to loading the kernel, OAL components, and interrupt handler functions where you cannot use software breakpoints.

Lesson 3: Testing a System by using the CETK

Automated software testing is a key to improving product quality while lowering development and support costs. This is particularly important if you create a custom BSP for a target device, added new device drivers, and implemented custom OAL code. Before releasing a new series of the system to production, it is vital to perform functional testing, unit testing, stress testing, and other types of testing to validate each part of the system and ensure that the target device operates reliably under normal conditions. It is generally much more expensive to fix defects after a new product reaches the market than to create testing tools and scripts that simulate users operating the target device and fix any defects while the system is still under development. System testing should not be an afterthought. To perform system testing efficiently throughout the software development cycle, you can use the CETK.

After this lesson, you will be able to:

- Describe typical usage scenarios for CETK test tools.
- Create user-defined CETK tests.
- Run CETK tests on a target device.

Estimated lesson time: 30 minutes.

Windows Embedded CE Test Kit Overview

The CETK is a separate test application included with Platform Builder for Windows Embedded CE to validate the stability of applications and device drivers based on a series of automated tests organized in a CE test catalog. The CETK includes numerous default tests for several driver categories of peripheral devices and you can also create custom tests for your specific needs.

NOTE CETK tests

For a complete list of default tests included in the CETK, see the section "CETK Tests" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa917791.aspx*.

CETK Architecture

As illustrated in Figure 4–10, the CETK application is a client/server solution with components running on the development computer and on the target device. The

development computer runs the workstation server application (CETest.exe) while the target device runs the client-side application (Clientside.exe), test engine (Tux.exe), and test results logger (Kato.exe). Among other things, this architecture enables you to run concurrent tests on multiple different devices from the same development workstation. Workstation server and client-side applications can communicate through KITL, ActiveSync® or a Windows Sockets (Winsock) connection.

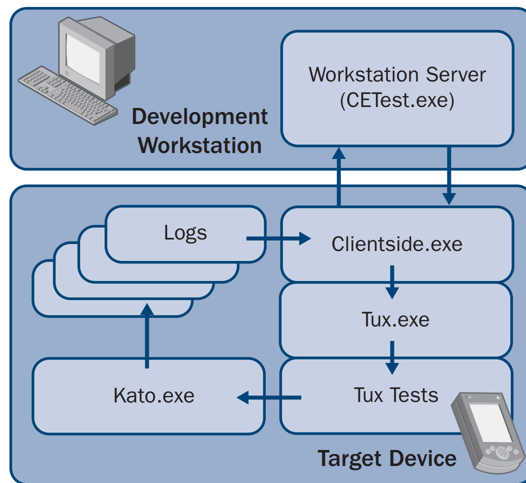


Figure 4-10 The CETK client/server architecture

The CETK application includes the following components:

- **Development workstation server** CETest.exe provides the graphical user interface (GUI) to run and manage CETK tests. This application also enables you to configure server settings and connection parameters, as well as connect to a target device. Having established a device connection, the workstation server can automatically download and start the client-side application, submits test requests, and compile test results based on captured logs in real-time for display.
- **Client-side application** Clientside.exe interfaces with the workstation server application to control the test engine and return test results to the server application. If Clientside.exe is unavailable on the target device, the workstation server cannot establish a communication stream to the target device.
- **Test engine** CETK tests are implemented in DLLs that Tux.exe loads and runs on the target device. Typically, you start the test engine remotely through

workstation server and client-side application, yet it is also possible to start Tux.exe locally, in stand-alone fashion with no workstation server requirement.

- **Test results logger** Kato.exe tracks the results of the CETK tests in log files. Tux DLLs can use this logger to provide additional information about whether a test succeeded or failed and have their output routed to multiple user-defined output devices. Because all CETK tests use the same logger and format, it is possible to use a default file parser or implement a custom log file parser for automatic result processing according to specific requirements.

NOTE CETK for managed code

A managed version of the CETK is available to validate native and managed code. For details about the managed version, see the section "Tux.Net Test Harness" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa934705.aspx*.

Using the CETK

You can run CETK tests in a variety of ways according to the connectivity options supported on the target device. You can use KITL, Microsoft ActiveSync, or a TCP/IP connection to connect to the target device, download the target-side CETK components, run the desired tests, and view the results in the graphical user interface on the development workstation. On the other hand, if your target device does not support these connectivity options, you must run the tests locally with appropriate command-line options.

Using the CETK Workstation Server Application

To work with the workstation server application, click Windows Embedded CE 6.0 Test Kit in the Windows Embedded CE 6.0 program group on your development computer, open the Connection menu and select the Start Client command. You can then configure the transport by clicking the Settings button. If the target device is switched on and connected to your development workstation, click Connect, select the desired target device, and then click OK to establish the communication channel and deploy the required binaries. The CETK application is now ready to run tests on the target device.

As illustrated in Figure 4–11, the CETK application automatically detects the device drivers available on the target and provides a convenient method to run the tests. One way is to click the device name under Start/Stop Test on the Tests menu, which causes

CETK to test all detected components. Another way is to right-click the Test Catalog node and select the Start Tests command. You can also expand the individual containers, right-click an individual device test, and click Quick Start to test only a single component. The workstation server application also provides access to Application Verifier, CPU Monitor, Resource Consume, and Windows Embedded CE Stress tool when you right-click the device node and open the Tools submenu.

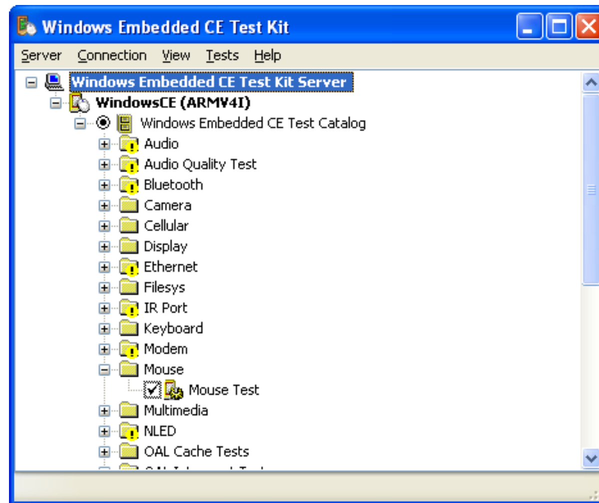


Figure 4-11 The graphical user interface of the CETK application

Create a Test Suite

Apart from running all tests at once or quick tests individually, you can create test suites that include a custom series of tests that you want to perform repeatedly throughout the software development cycle. To create a new test suite, use the Test Suite Editor, available in the workstation server application on the Tests menu. The Test Suite Editor is a graphical tool to select the tests that belong to a suite conveniently. You can export test suite definitions in the form of Test Kit Suite (.tks) files and import these files on additional development computers to ensure that all workstation server applications perform the same set of tests. These .tks files can also provide the basis for test definition archives.

Customizing Default Tests

The graphical user interface also enables you to customize the command lines that the workstation server application sends to the test engine (Tux.exe) to perform the tests. To modify the parameters of a test, right-click the test in the Test Catalog and

select the Edit Command Line option. For example, the Storage Device Block Driver Benchmark Test analyzes the performance of a storage device by reading and writing data to every sector on the device. This implies that all existing data on the storage device will be destroyed. To protect you from accidental data loss, the Storage Device Block Driver Benchmark Test is skipped by default. To run the Storage Device Block Driver Benchmark Test successfully, you must edit the command line and explicitly add a special parameter called **-zorch**.

The supported command-line parameters depend on each individual CETK test implementation. Tests might support or require a variety of configuration parameters, such as an index number to identify the device driver to test, or additional information that must be provided to run the test.

> **NOTE Command-line parameters for CETK tests**
> For a complete list of default CETK tests with links to additional information, such as command-line parameters, see the section "CETK Tests" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/ms893193.aspx*.

Running Clientside.exe Manually

If you have included the Windows Embedded CE Test Kit catalog item in your run-time image, downloaded the CETK components with the workstation server application, or exported the components from your development workstation to the target device by using the File Viewer remote tool, you can start Clientside.exe on the target device and establish a connection to a workstation server manually. If your target device does not provide the Run dialog box for this purpose, open the Target menu in the Platform Builder IDE, select Run Programs, select Clientside.exe, and then select Run.

Clientside.exe supports the following command-line parameters that you can specify to connect to a specific workstation server application, detect installed drivers, and run tests automatically:

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

It is important to note that you can define these parameters also in a Wcetk.txt file or in the HKEY_LOCAL_MACHINE/Software/Microsoft/CETT registry key on the target device so that you can start Clientside.exe without command-line parameters.

In this case, Clientside.exe searches for Wcetk.txt in the root directory, then in the Windows directory on the target device, and then in the release directory on the development workstation. If Wcetk.txt does not exist in any of these locations, it checks the CETT registry key. Table 4–5 summarizes the Clientside.exe parameters.

Table 4-5 Clientside.exe start parameters

Command Line	Wcetk.txt	CETT Registry Key	Description
/n	SERVERNAME	ServerName (REG_SZ)	Specifies the host server name. Cannot be used together with **/i** and requires Domain Name System (DNS) for name resolution.
/i	SERVERIP	ServerIP (REG_SZ)	Specifies the host IP address. Cannot be used together with **/n**.
/p	PORTNUMBER	PortNumber (REG_DWORD)	Specifies the server port number that can be configured from the workstation server interface.
/a	AUTORUN	Autorun (REG_SZ)	When set to one (1), the device automatically starts the test after the connection is established.
/s	DEFAULTSUITE	DefaultSuite (REG_SZ)	Specifies the name of the default test suite to run.

Table 4-5 Clientside.exe start parameters (Continued)

Command Line	Wcetk.txt	CETT Registry Key	Description
/x	AUTOEXIT	Autoexit (REG_SZ)	When set to one (1), the application automatically exits when the tests are completed.
/d	DRIVERDETECT	DriverDetect (REG_SZ)	When set to zero (0), the detection of devices drivers is disabled.

Running CETK Tests in Standalone Mode

Clientside.exe connects to CETest.exe on the developer workstation, yet it is also possible to run CETK tests without a connection, which is particularly useful for devices that provide no connectivity possibilities. If you include the Windows Embedded CE Test Kit catalog item in the run-time image, you can start the test engine (Tux.exe) directly, which implicitly starts the Kato logging engine (Kato.exe) to track the test results in log files. For example, to perform mouse tests (mousetest.dll) and track the results in a file called test_results.log, you can use the following command line:

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

NOTE **Tux command-line parameters**

For a complete list of Tux.exe command-line parameters, see the section "Tux Command-Line Parameters" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa934656.aspx*.

Creating a Custom CETK Test Solution

The CETK includes a large number of tests, yet the default tests cannot cover all testing requirements, especially if you added your own custom device drivers to a BSP. To provide you with an option to implement user-defined tests for your custom drivers, the CETK relies on the Tux framework. Platform Builder includes a WCE

TUX DLL template to create a skeleton Tux module with a few mouse clicks. When implementing the logic to exercise your driver, you might find it useful to check out the source code of existing test implementations. The CETK includes source code, which you can install as part of the Windows Embedded CE Shared Source in the Setup wizard for Windows Embedded CE. The default location is %_WINCEROOT%\Private\Test.

Creating a Custom Tux Module

To create a custom test library that is compliant with the Tux framework, start the Windows Embedded CE Subproject Wizard by adding a subproject to the OS design of your run-time image and select the WCE TUX DLL template. This causes the Tux wizard to create a skeleton that you can customize according to your driver requirements.

You must edit the following files in the subproject to customize the skeleton Tux module:

- **Header file Ft.h** Defines the TUX Function Table (TFT), including a function table header and function table entries. The function table entries associate test IDs with the functions that contain the test logic.
- **Source code file Test.cpp** Contains the test functions. The skeleton Tux module includes a single TestProc function that you can use as reference to add custom tests to the Tux DLL. You can replace the sample code to load and exercise your custom driver, log activities through Kato, and return an appropriate status code back to the Tux test engine when the tests are completed.

Defining a Custom Test in the CETK Test Application

The skeleton Tux module is fully functional, so you can compile the solution and build the run-time image even without code modifications. To run the new test function on a target device, you must configure a user-defined test in the CETK workstation server application. For this purpose, CETK includes a User-Defined Test Wizard, which you can start by clicking the User Defined command on the Tests menu. Figure 4–12 shows the User-Defined Test Wizard with configuration parameters to run a skeleton Tux module.

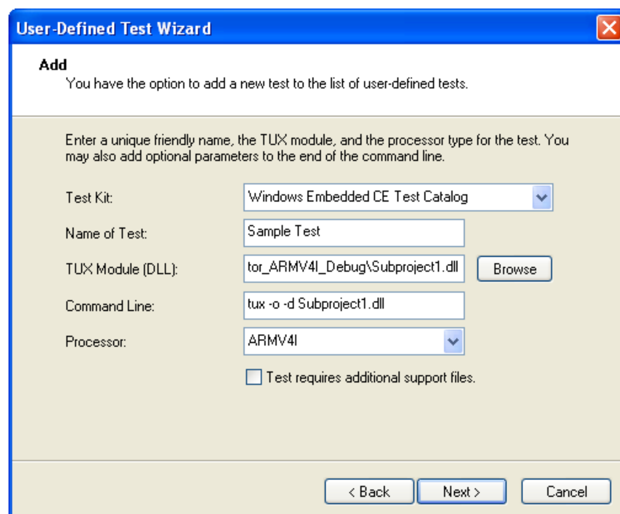


Figure 4-12 Configuring a custom test in the User-Defined Test Wizard

Debugging a Custom Test

Because Tux tests rely on code and logic implemented in Tux DLLs, it might be necessary to debug the test code. One issue worth mentioning is that you can set breakpoints in your test routines, but when code execution halts on those breakpoints, you lose the connection between the client-side application (Clientside.exe) and the workstation server application (CETest.exe). Consider using debug messages instead of breakpoints. If you must use breakpoints for thorough debugging, run Tux.exe directly on the target device in standalone mode, as mentioned earlier in this lesson. You can display the required command line in the workstation server application when you right-click the test and select Edit Command Line.

Analyzing CETK Test Results

CETK tests should use Kato to log test results, as demonstrated in the skeleton Tux module:

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

The workstation server application retrieves these logs automatically through Clientside.exe and stores them on the development workstation.You can also access these log files through other tools. For example, if you are using CETK in stand-alone fashion, you can import the log files to the development workstation by using the File Viewer remote tool.

The CETK includes a general CETK parser (Cetkpar.exe) located in the C:\Program Files\Microsoft Platform Builder\6.00\Cepb\Wcetk folder for convenient viewing of imported log files, as shown in Figure 4–13. Typically, you start this parser by right-clicking a completed test in the workstation server application and selecting View Results, yet you can also start Cetkpar.exe directly. Some tests, particularly performance tests based on PerfLog.dll, can also be parsed into comma-separated values (CSV) format and opened in a spreadsheet to summarize the performance data. The CETK includes a PerfToCsv parser tool for this purpose, and you can develop custom parsers for special analysis scenarios. Kato log files use a plain text format.

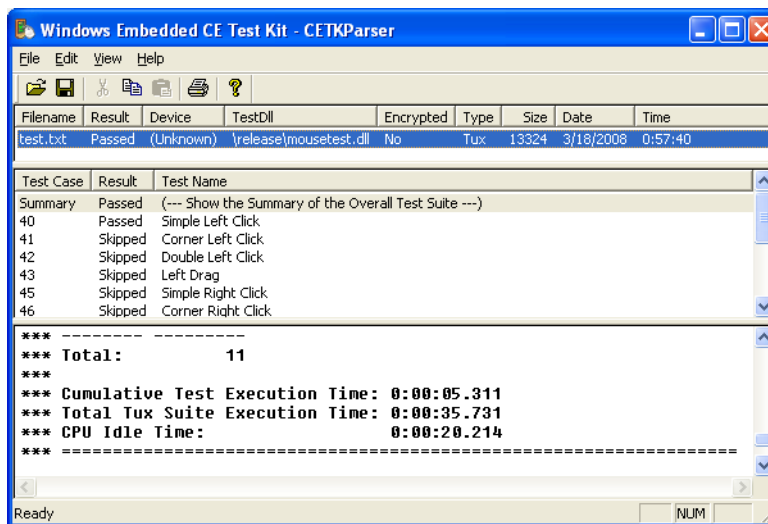


Figure 4-13 Analyzing CETK test results

Lesson Summary

The Windows Embedded CE Test Kit is an extensible tool that enables you to test drivers and applications on a target device in connected mode and in standalone mode. Running the CETK tools in standalone mode is useful if the target device does not support connectivity over KITL, ActiveSync, or TCP/IP. Most typically, developers use the CETK to test device drivers added to the BSP of a target device.

The CETK relies on the Tux test engine, which provides a common framework for all test DLLs. The Tux DLLs contain the actual testing logic and run on the target device to load and exercise the driver. Tux DLLs also interface with Kato to track test results in log files, which you can access directly in the CETK test application or process in separate tools, such as custom parsers and spreadsheets.

Lesson 4: Testing the Boot Loader

The general task of a boot loader is to load the kernel into memory and then call the OS startup routine after powering up the device. On Windows Embedded CE specifically, the boot loader is part of the BSP and in charge of initializing the core hardware platform, downloading the run-time image, and starting the kernel. Even if you do not plan to ship a boot loader in your final product and directly bootstrap the run-time image, you might find a boot loader helpful during the development cycle. Among other things, a boot loader can help to simplify run-time image deployment complexities. Downloading the run-time image over Ethernet connections, serial cable, DMA, or USB connections from a development computer is a convenience feature that can help to save development time. Based on the source code included with Platform Builder for Windows Embedded CE 6.0, you can also develop a custom boot loader to support new hardware or features. For example, you can use a boot loader to copy the run-time image from RAM into flash memory and eliminate the need for a separate flash memory programmer or Institute of Electrical and Electronic Engineers (IEEE) 1149.1-compliant test access port and boundary-scanning technology. However, debugging and testing a boot loader is a complex undertaking because you are working with code that runs before the kernel loads.

After this lesson, you will be able to:

- Describe the CE boot loader architecture.
- List common debugging techniques for boot loaders.

Estimated lesson time: 15 minutes.

CE Boot Loader Architecture

The underlying idea of a boot loader is to bootstrap a small program with pre-boot routines in linear, nonvolatile, CPU-accessible memory. Having placed the initial boot loader image on the target device at the memory address where the CPU begins to retrieve code through a built-in monitor program provided by the board manufacturer or a JTAG probe, the boot loader runs whenever you power up or reset the system. Typical boot loader tasks performed at this stage include initializing the Central Processing Unit (CPU), the memory controller, system clock, Universal Asynchronous Receiver/Transmitters (UARTs), Ethernet controllers, and possibly other hardware devices, downloading the run-time image and copying it into RAM according to the binary image builder (BIB) layout, and jumping to the StartUp

function. The last record of the run-time image contains this function's start address. The StartUp function then continues the boot process by calling the kernel initialization routines.

Although the various boot loader implementations differ in their complexity and the tasks they perform, there are common characteristics that Windows Embedded CE covers through static libraries to facilitate boot loader development, as illustrated in Figure 4–14. The resulting boot loader architecture influences how you can debug boot loader code. Chapter 5, "Customizing a Board Support Package," discusses boot loader development in more detail.

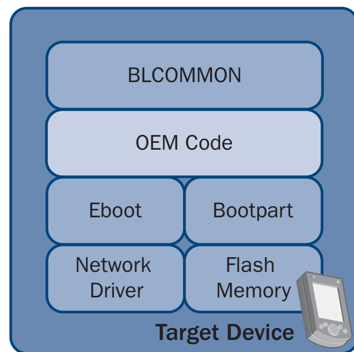


Figure 4-14 Windows Embedded CE boot loader architecture

The Windows Embedded CE boot loader architecture is based on the following code portions and libraries:

- **BLCOMMON** Implements the basic boot loader framework for copying the boot loader from flash memory to RAM for faster execution, decoding image file contents, verifying checksums, and keeping track of load progress. BLCOMMON calls well-defined OEM functions throughout the process to handle hardware-specific customizations.
- **OEM code** This is the code that OEMs must implement for their hardware platforms to support the BLCOMMON library.
- **Eboot** Provides Dynamic Host Configuration Protocol (DHCP), Trivial File Transfer Protocol (TFTP), and User Datagram Protocol (UDP) services to download run-time images over Ethernet connections.
- **Bootpart** Provides storage partitioning routines so that the boot loader can create a binary ROM image file system (BinFS) partition and a second partition

with another file system on the same storage device. Bootpart can also create a boot partition to store boot parameters.

- **Network drivers** Encapsulate the basic initialization and access primitives for a variety of common network controller devices. The interface for the libraries is generic so that both the boot loader and the OS can use the interface. The boot loader uses the interface for downloading run-time images and the OS uses the interface to implement a KITL connection to Platform Builder.

Debugging Techniques for Boot Loaders

The boot loader design typically consists of at least two distinctive parts. The first part is written in assembly language and initializes the system before jumping to a second part written in C. If you are using a BLCOMMON-based architecture as illustrated in Figure 4–14, you might not have to debug assembly code. If your device is equipped with a UART, you can use the RETAILMSG macro in C code to send data over a serial output interface to the user for display.

Depending on whether you must debug assembly or C code, the following different debugging techniques are available:

- **Assembly code** Common debugging techniques for the initial startup code rely on LEDs, such as a debugging board with seven-segment LEDs and UARTs for a serial communication interface, because it is relatively straightforward to access General Purpose Input/Output (GPIO) registers and modify the state of an input/output line.
- **C Code** Debugging is much easier at the C-code level because you can access advanced communication interfaces and debugging macros.
- **Assembly and C code** If a hardware debugger (JTAG probe) is available, you can use Platform Builder in conjunction with an eXDI driver to debug the boot loader.

EXAM TIP

To pass the certification exam, make sure you know the different techniques to debug the boot loader, kernel, device drivers, and applications.

Lesson Summary

Debugging the boot loader is a complex task that requires a good understanding of the hardware platform. If a hardware debugger is available, you can use Platform Builder in conjunction with an eXDI driver for hardware-assisted debugging. Otherwise, consider using an LED board for debugging assembly code and C-style macros to output debug messages over a serial communication interface in C code.

Lab 4: System Debugging and Testing based on KITL, Debug Zones, and CETK Tools

In this lab, you debug a console application added as a subproject to an OS design based on the Device Emulator BSP. To enable debugging, you include KdStub and KITL in the run-time image and configure corresponding target-device connectivity options. You then modify the source code of the console application to implement support for debug zones, specify initially active debug zones in the Pegasus registry key, and attach to the target device with the Kernel Debugger to examine the debug messages in the Output window of Visual Studio. Subsequently, you use the CETK to test the mouse driver included in the run-time image. To create the initial OS design in Visual Studio, follow the procedures outlined in Lab 1, "Creating, Configuring, and Building an OS Design."

NOTE **Detailed step-by-step instructions**

To help you successfully master the procedures presented in this Lab, see the document "Detailed Step-by-Step Instructions for Lab 4" in the companion material for this book.

▶ Enable KITL and Use Debug Zones

1. Open the OS design project created in Lab 1 in Visual Studio, right-click the OSDesign name and select Properties to edit the OS design properties, select Configuration Properties and then Build Options, and then select the Enable KITL check box for the run-time image.
2. In the OS Design property pages dialog box, also enable the Kernel Debugger feature, apply the changes, and then close the dialog box.
3. Verify that you are currently working in debug build configuration to build an image that contains the KITL and Kernel Debugger components activated in the previous steps.
4. Build the OS design by selecting Rebuild Current BSP and Subprojects under Advanced Build Commands on the Build menu (perform a Clean Sysgen if you encounter errors during subsequent steps).
5. Open the Target menu and click Connectivity Options to display the Target Device Connectivity Options dialog box. Configure the following settings, as shown in Table 4-6 and then click OK.

Table 4-6 Device connectivity settings

Configuration Parameter	Setting
Download	Device Emulator (DMA)
Transport	Device Emulator (DMA)
Debugger	KdStub

6. Add a subproject to the OS design and select the WCE Console Application template. Name the project TestDbgZones and select the option A Typical Hello World Application in the CE Subproject Wizard.
7. Add a new header file called DbgZone.h to the subproject and define the following zones:

```
#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. Add an include statement for the DbgZone.h header file to the TestDbgZones.c file:

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

9. Define the dpCurSettings variable for the debug zones above the _tmain function, as follows:

```
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. Register the debug zones of the module in the first line of the _tmain function:

```
DEBUGREGISTER(NULL);
```

11. Use the RETAILMSG and DEBUGMSG macros to display debug messages and associate them with debug zones, as follows:

```
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. Build the application, attach to the target device, and then start the application by using the Target Control window.

13. Note that only the first debug message is displayed in the debug Output window:

```
4294890680 PID:3c50002 TID:3c60002 Message : ZONE_INIT
```

14. Open the registry editor (Regedit.exe) on your development computer to activate the remaining debug zones, by default.

15. Open the HKEY_CURRENT_USER\Pegasus\Zones key and create a REG_DWORD value called TestDbgZone (according to the name of the module defined in the dpCurSettings variable).
16. Set the value to 0xFFFF to enable all 16 named zones, which correspond to the lower 16 bits in this 32 bit DWORD value (see Figure 4–15).
17. In Visual Studio, start the application again, and notice the following output:

```
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. Change the TestDbgZone value in the registry to enable and disable different debug zones and verify the results in the Output window of Visual Studio.

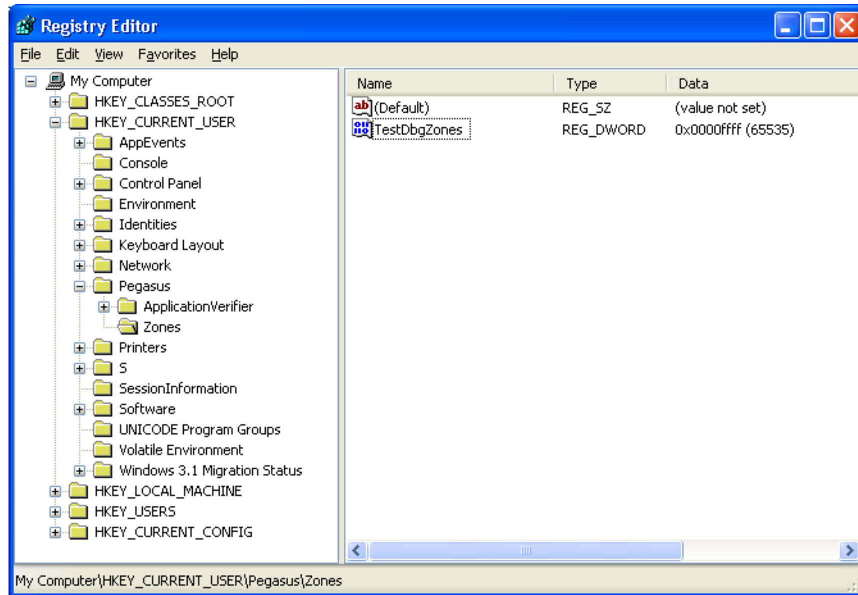


Figure 4-15 HKEY_CURRENT_USER\Pegasus\Zones: "TestDbgZone"=dword:FFFF

NOTE Enabling and disabling debug zones in Platform Builder

You cannot control the debug zones for the TestDbgZone module in Platform Builder because the application process exits before you can open and modify the active zone for this module. You can only manage debug zones for loaded modules in Platform Builder, such as for graphical applications and DLLs.

▶ Perform Mouse Driver Tests by Using the CETK

1. Open the Windows CE Test Kit application from the Start menu on your development computer (open the Windows Embedded CE 6.0 menu and click Windows Embedded CE Test Kit).
2. In the Windows Embedded CE Test Kit window, open the Connection menu and click Start Client to establish a connection to the target device.
3. Click Connect and select the device in the Connection Manager window.
4. Verify that the workstation server application connects successfully to the device, deploys the required CETK binaries, detects available device drivers, and displays a list of all components in a hierarchical tree, as shown in Figure 4–16.
5. Right-click the Windows CE Test Catalog node and click Deselect All Tests.
6. Open each node in the list and select the Mouse Test check box.
7. Open the Test menu and then clock on Start/Stop Test to perform a mouse test.
8. On the target device perform the required mouse actions.
9. Complete the test and then can access the test report by right-clicking the test entry and selecting View Results.
10. Examine the results in the CETK parser and notice successful, skipped, and failed test procedures.

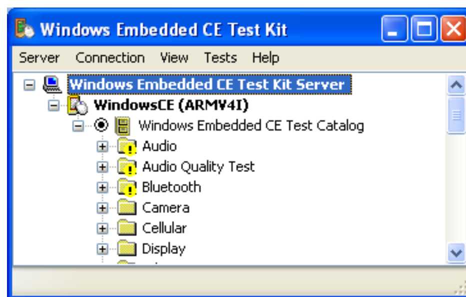


Figure 4-16 Device categories in the Windows Embedded CE Test Kit window

Chapter Review

Platform Builder for Windows Embedded CE ships with a comprehensive set of debugging and testing tools to diagnose and eliminate root causes of errors, and validate the system in its final configuration prior to its release to production. The debugging tools integrate with Visual Studio and communicate over KITL connections with the target device. Alternatively, you can create a memory dump and use the CE Dump File Reader to debug the system in offline mode, which is particularly useful for postmortem debugging. The debugging environment is also extensible by means of eXDI drivers to perform hardware-assisted debugging beyond the capabilities of the standard Kernel Debugger.

The Kernel Debugger is a hybrid debugger for kernel components and applications. Debugging starts automatically if you attach to a target device with KdStub and KITL enabled. You can use the Target Control window to start applications for debugging and perform advanced system tests based on CEDebugX commands. However, it is important to keep in mind that you cannot set breakpoints in interrupt handlers or OAL modules because at these levels, the kernel operates in single-thread mode and stops communicating with the development workstation if code execution halts. To debug interrupt handlers, use a hardware debugger or debug messages. The debug messages feature supports debug zones to control the information output without having the rebuild the run-time image. You can also use debug messages to debug the C-code portion of a boot loader, yet for the assembly code portion you must use a hardware debugger or an LED panel.

KITL is also a requirement if you want to centralize system testing based on the CETK Test application, although it is also possible to run CETK tests in standalone mode. If you are developing a custom BSP for a target device, you can use the CETK to perform automated or semi-automated component tests based on custom Tux DLLs. Platform Builder includes a WCE TUX DLL template to create a skeleton Tux module that you can extend to meet your specific testing needs. You can integrate the custom Tux DLL in the CETK test application and run tests individually or as part of a larger test suite. Because all CETK tests use the same logging engine and log file format, you can use the same parser tool to analyze the results of default and user-defined tests.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Debug Zones
- KITL
- Hardware debugger
- dpCurSettings
- DebugX
- Target Control
- Tux
- Kato

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Detect Memory Leaks

Add a subproject to the OS design for a console application that generates memory leaks by allocating memory blocks and never freeing them. Using the tools discussed in this chapter, isolate the issue and fix it.

Custom CETK Test

Add a subproject to the OS design for a WCE TUX DLL. Build the Tux DLL and register it in the Windows Embedded CE Test Kit application. Run a CETK test and verify the test results. Set breakpoints in your Tux DLL and debug the code by running a CETK test in standalone mode.

Chapter 5

Customizing a Board Support Package

Application developers do not often need to create a Board Support Package (BSP). However, Original Equipment Manufacturers (OEMs) face this requirement when porting Microsoft® Windows® Embedded CE 6.0 R2 to a new hardware platform. To help OEMs accomplish this task efficiently, Windows Embedded CE features a production-quality OEM adaptation layer (PQOAL) architecture that promotes code reuse based on a collection of OAL libraries organized by processor model and OAL function. Microsoft encourages OEM developers to clone and customize an existing BSP to meet their specific requirements and take full advantage of tested and proven production features for power management, performance optimizations, and input/output controls (IOCTL). This chapter covers the PQOAL architecture, explains how to clone BSPs, and lists the functions that OEM developers must implement in order to adapt Windows Embedded CE to new hardware architectures and models. It is advantageous to understand the various aspects of customizing a BSP even if you do not intend to develop your own. This chapter will provide an overview of the aspects of BSP customization, ranging from modifications of the startup process and implementing kernel initialization routines, to adding device drivers, power management capabilities, and support for performance optimization.

Exam objectives in this chapter:

- Understanding the BSP architecture of Windows Embedded CE
- Modifying and adapting BSPs and boot loaders for specific target devices
- Understanding memory management and layout
- Enabling power management in a BSP

Before You Begin

To complete the lessons in this chapter, you must have the following:

- At least some basic knowledge about Windows Embedded CE software development.
- A thorough understanding of hardware architectures for embedded devices.
- Basic knowledge about power management and how to implement it in drivers and applications.
- A development computer with Microsoft Visual Studio® 2005 Service Pack 1 and Platform Builder for Windows Embedded CE 6.0 R2 installed.

Lesson 1: Adapting and Configuring a Board Support Package

The BSP development process for a new hardware platform typically begins after performing functional tests of the hardware by using a ROM monitor, by cloning an appropriate reference BSP, followed by implementing a boot loader and the core OAL functions to support the kernel. The goal is to create a bootable system with the least possible amount of custom code. You can then add device drivers to the BSP to support integrated and peripheral hardware and expand the system by implementing power management and other advanced operating system (OS) features according to the capabilities of the target device.

After this lesson, you will be able to:

- Identify and locate the content of a PQOAL–based Board Support Package.
- Identify hardware-specific and common-code libraries.
- Understand how to clone a BSP.
- Adapt a boot loader, OAL, and device drivers.

Estimated lesson time: 40 minutes.

Board Support Package Overview

A BSP contains all the source code for the boot loader, OAL, and device drivers for a given platform. In addition to these components, the BSP also contains build and system configuration files, as illustrated in Figure 5–1. The configuration files are not included in the actual run-time image, yet they are part of the BSP package to specify source code files, memory layout, registry settings, and other aspects to compile and build the run-time image, as explained in Chapter 2, "Building and Deploying a Run-Time Image."

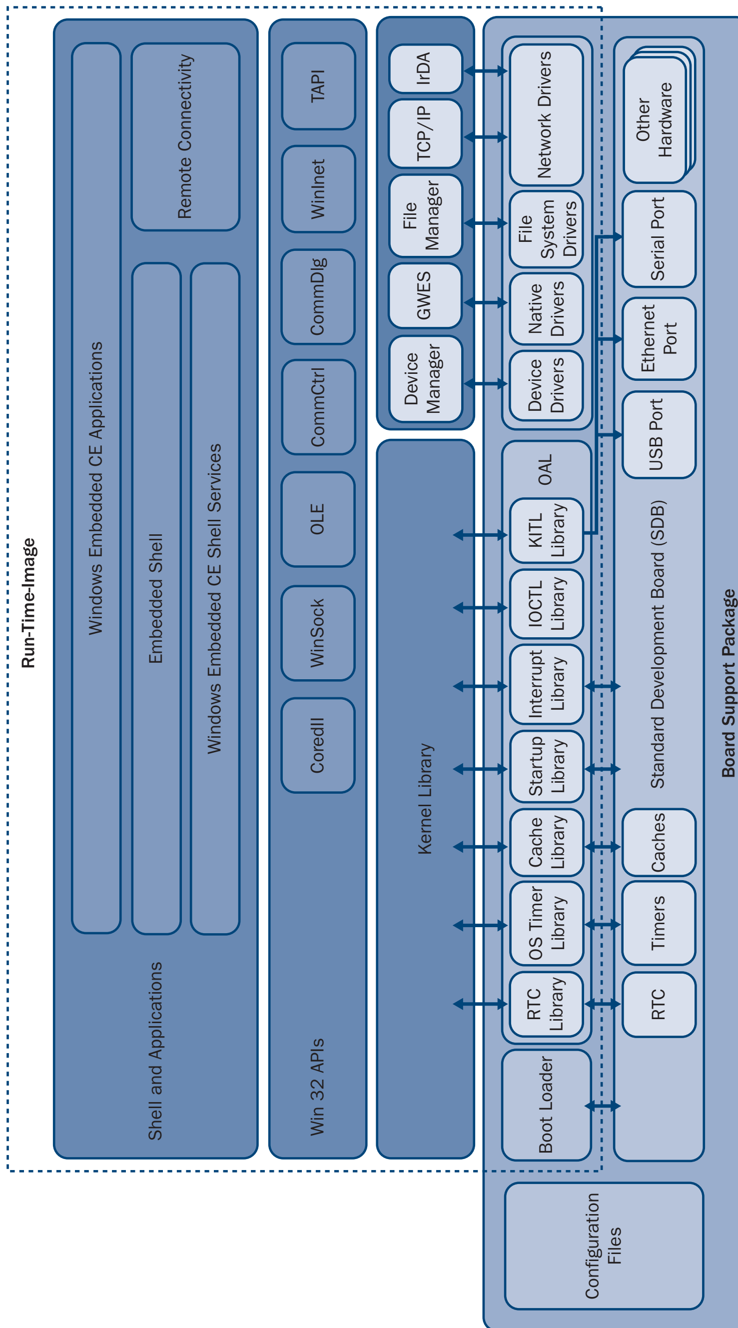


Figure 5-1 Components of a BSP in relationship to the remaining elements of Windows Embedded CE 6.0

According to Figure 5–1, BSP development includes the following main components:

- **Boot loader** Runs when powering up or resetting the device. The boot loader is responsible for initializing the hardware platform and passing execution to the operating system.
- **OEM adaptation layer (OAL)** Represents the core of the BSP and is the interface between the kernel and the hardware. Because it is linked directly to the kernel, it becomes part of the kernel in a CE run-time image. Some of the core kernel components are directly dependent on the OAL for hardware initialization, such as the interrupt handling and timer handling for the thread scheduler.
- **Device drivers** Manage the functionality of a particular peripheral and provide an interface between the device hardware and the operating system. Windows Embedded CE supports a variety of driver architectures based on the interfaces they expose, as explained in Chapter 6, "Developing Device Drivers."
- **Configuration files** Provide the necessary information to control the build process and plays a key role in the design of a platform's operating system. Typical configuration files included in BSPs are Sources files, Dirs files, Config.bib, Platform.bib, Platform.reg, Platform.db, Platform.dat, and catalog files (*.pbcxml).

Adapting a Board Support Package

It is generally a good idea to jump start the BSP development process by cloning an existing reference BSP instead of creating a BSP from scratch. Even if you must develop a BSP for an entirely new platform with an entirely new CPU, it is still recommended to clone a BSP based on a similar processor architecture. In this way, you can reduce BSP development time by reusing hardware-independent code from the existing BSP and shorten future migration cycles to new Windows Embedded versions as they become available on the market. Migrating a proprietary BSP design is generally much harder to do than migrating a PQOAL–based design because the proprietary BSP cannot benefit from those PQOAL code portions that Microsoft implicitly migrates and tests as part of the new operating system version.

Adapting a board support package includes the following sequence of steps:

1. Cloning a reference BSP.
2. Implementing a boot loader.
3. Adapting the OAL functions.

4. Modifying the run-time image configuration files.
5. Developing device drivers.

Cloning a Reference BSP

Platform Builder includes a wizard that facilitates cloning a reference BSP. This wizard copies the entire source code of the selected reference BSP to a new folder structure so that you can customize the BSP for the new hardware without affecting the reference BSP or other BSPs in the %_WINCEROOT% folder hierarchy. Figure 5-2 illustrates how to start the BSP Cloning Wizard in Microsoft Visual Studio 2005 with Platform Builder for Windows Embedded CE 6.0.

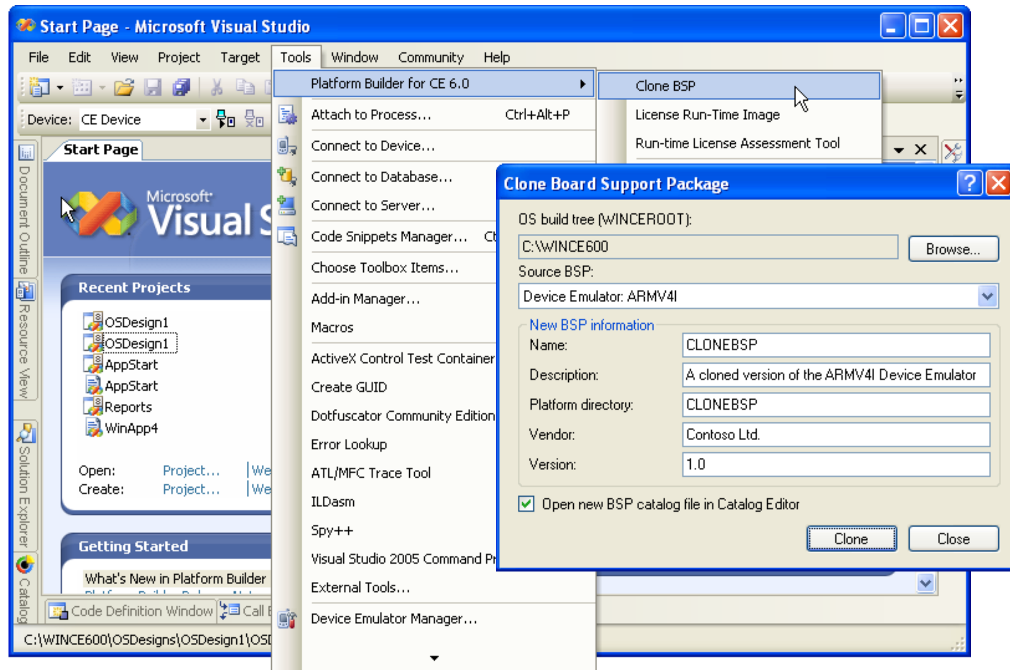


Figure 5-2 Cloning a BSP in Visual Studio 2005

> **NOTE BSP names**
>
> When cloning a BSP, you have to choose a new name for this new set of files. The name that you choose for the platform must match the name of the folder on your hard drive. As mentioned in the previous chapter, the build engine is based on a command-line script and is not compatible with spaces in folder names. Therefore, the BSP's name must not include white spaces. You can use the underscore (_) character instead.

BSP Folder Structure

To increase code reusability, PQOAL–based BSPs feature a common architecture and corresponding folder structure that is consistent across processor families. Due to this common architecture, large portions of the source code can be reused regardless of hardware-specific BSP requirements. Figure 5–3 shows the typical BSP folder structure and Table 5–1 summarizes the most important BSP folders.

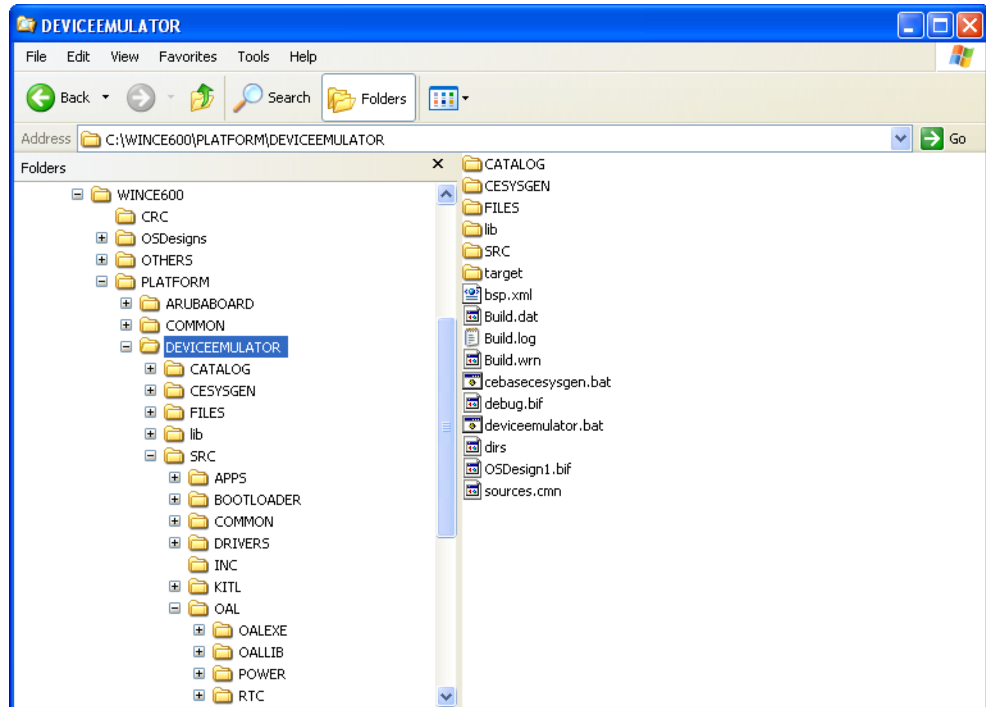


Figure 5-3 Folder structure of a typical BSP

TIP %_TARGETPLATROOT%

You can use the environment variable %_TARGETPLATROOT% in the build window to locate the path of the BSP being used in the current OS design (Open Release Directory in Build Window option on the Build menu in Visual Studio).

Table 5-1 Important BSP folders

Folder	Description
Root Folder	Contains configuration and batch files. The two most important files for developers are as follows: ■ **Sources.cmn** Contains macro definitions that are common across the entire BSP. ■ **<*BSP Name*>.bat** Sets the default BSP environment variables.
CATALOG	Contains the BSP catalog file in which all the components of the BSP are defined. This file is used in the OS design stage to add or remove BSP features. Chapter 1, "Customizing the Operating System Design," discusses how to manage catalog items.
CESYSGEN	Contains the Makefile for the Sysgen tool. Configuring a BSP does requires no changes to this directory.
FILES	Contains the build configuration files, such as .bib, .reg, .db, and .dat files.
SRC	Contains the platform-specific source code that you must adapt according to the PQOAL model, which divides the code between platform-specific and common components per CPU type.
COMMON	Exists under the Platform directory and contains most of the BSP source code. It consists of a common set of processor specific components. The BSP links to libraries in this folder, generated during the build process. These are libraries for processor–based peripherals as well as processor-specific OAL parts. If the hardware uses a CPU from the family of supported processors, then most of these libraries can be reused without modification.

Platform-Specific Source Code

The most important platform-specific source code that you must adapt as part of your BSP is for the boot loader, the OAL, and the device drivers. You can find the corresponding source code underneath the Src folder in the following subdirectories:

- **Src\Boot loader** Contains the boot loader code. However, if the boot loader relies on BLCOMMON and related libraries, then only the basic hardware-specific part of the boot loader is located in this directory. The reusable boot loader code is available in the Public folder (%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg) and linked as libraries to the BSP part. Chapter 4, "Debugging and Testing the System," introduces the static libraries that facilitate boot loader development.
- **Src\Oal** Contains the bare minimal amount of code that is specific to the hardware platform. The majority of the OAL code is located in %_WINCEROOT%\Platform\Common, divided into hardware-independent, processor-family-related, chip-set-specific and platform-specific groups. These code groups provide most of the OAL functionality and are linked to the platform-specific parts as libraries.
- **Src\Common and Src\Drivers** Contains the driver source code, organized in different categories to facilitate maintenance and portability. These categories are typically processor-specific and platform-specific. The processor-specific component is located in the Src\Common directory and requires no modifications when adapted to new hardware based on the same processor family.

Implementing a Boot Loader from Existing Libraries

Several aspects have to be considered when adapting a boot loader for a new platform, including:

- Changes in the processor architecture.
- Location of the boot loader code on the target device.
- Memory architecture of the platform.
- Tasks to perform during the boot process.
- Supported transports for downloading the run-time image.
- Additional features to be supported.

Memory Mappings

The first important adaptation task revolves around the definition of memory mappings for the boot loader. The standard BSPs included in Windows Embedded CE define the memory configuration in a .bib file, located in a boot loader subdirectory, such as %_WINCEROOT%\Platform\Arubaboard\Src\Boot loader\Eboot\Eboot.bib. The following listing shows an example of an Eboot.bib file, which you can customize to meet your specific requirements.

```
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

Among other things you can use the Eboot.bib file to reserve a memory section for the boot loader to pass information to the operating system during the startup process. This information might reflect the current state of initialized hardware, network communication capabilities if the boot loader supports Ethernet downloads, user and system flags for the operating system, such as to enable Kernel Independent

Transport Layer (KITL), and so on. To enable this communication, the boot loader and operating system must share a common region of physical memory, which is referred to as driver globals (DRV_GLB). The above Eboot.bib listing includes a DRV_GLB mapping. The data that the boot loader passes to the operating system in the DRV_GLB region must adhere to a BOOT_ARGS structure that you can define according to your specific requirements.

The following procedure illustrates how to pass Ethernet and IP configuration information from the boot loader to the operating system through a DRV_GLB region. To do this, create a header file in the %_WINCEROOT%\Platform\<*BSP Name*>\Src\Inc folder, such as Drv_glob.h, with the following content:

```
#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 Entry Point and Main Function

The StartUp entry point of the boot loader must be located in linear memory at the address where the CPU begins fetching code for execution because this routine carries out the initialization of the hardware. If the adaptation is based on a reference BSP for the same processor chipset, then most of the CPU-related and memory controller-related code can remain unchanged. On the other hand, if the CPU architecture is different, you must adapt the startup routine to perform the following tasks:

1. Put the CPU in the right mode.
2. Disable all interrupts.
3. Initialize the memory controller.
4. Setup caches, Translation Lookaside Buffers (TLBs), and Memory Management Unit (MMU).
5. Copy the boot loader from flash memory into RAM for faster execution.
6. Jump to the C code in the main function.

The StartUp routine eventually calls the main function of the boot loader, and if the boot loader is based on BLCOMMON, then this function in turn calls BootLoaderMain, which initializes the download transport by calling OEM platform functions. The advantage of using the standard libraries provided by Microsoft is that the modifications required to adapt a BSP to a new hardware platform are componentized, isolated, and minimized.

Serial Debug Output

The next step in the boot loader adaptation is the initialization of the serial debug output. This is an important part of the boot process because it enables the user to interact with the boot loader and the developer to analyze debug messages, as discussed in Chapter 4, "Debugging and Testing the System."

Table 5-2 lists the OEM platform functions required to support serial debug output in the boot loader.

Table 5-2 Serial debug output functions

Function	Description
OEMDebugInit	Initializes the UART on the platform.
OEMWriteDebugString	Writes a string to the debug UART.
OEMWriteDebugByte	Writes a byte to the debug UART, used by OEMWriteDebugString.
OEMReadDebugByte	Reads a byte from the debug UART.

Platform Initialization

Once the CPU and the debug serial output are initialized, you can turn your attention to the remaining hardware initialization tasks. The OEMPlatformInit routine performs these remaining tasks, including:

- Initializing the real-time clock.
- Setting up external memory, particularly flash memory.
- Initializing the network controller.

Downloading via Ethernet

If the hardware platform includes a network controller, then the boot loader can download the run-time image over Ethernet. Table 5–3 lists the functions that you must implement to support Ethernet–based communication.

Table 5-3 Ethernet support functions

Function	Description
OEMReadData	Reads data from the transport for downloading.
OEMEthGetFrame	Reads data from the NIC using function pointer pfnEDbgGetFrame.
OEMEthSendFrame	Writes data to the NIC using function pointer pfnEDbfSendFrame.
OEMEthGetSecs	Returns number of seconds passed relative to a fixed time.

The Ethernet support functions use callbacks into network controller-specific routines. This means that you must implement additional routines and set up appropriate function pointers in the OEMPlatformInit function if you want to support a different network controller, as demonstrated in the following sample code:

```
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;
    ...
}
```

Flash Memory Support

Having implemented network communication capabilities, you also must enable the boot loader to download run-time image onto the new hardware platform and pass control to it. Alternately, you can save the run-time image to flash memory. Table 5–4 lists the download and flash memory support functions that you must implement for this purpose if the reference BSP's boot loader does not already support these features.

Table 5-4 Functions for supporting download and flash memory

Function	Description
OEMPreDownload	Sets up the necessary download protocol supported by platform builder.
OEMIsFlashAddr	Checks if the image is for flash or RAM.
OEMMapMemAddr	Performs temporary remapping of the image to RAM.
OEMStartEraseFlash	Prepares to erase flash of enough size to fit the OS image.
OEMContinueEraseFlash	Continue erasing flash based on download progress.
OEMFinishEraseFlash	Complete the flash erasing once the download is done.
OEMWriteFlash	Write OS image to flash.

User Interaction

Boot loaders can support user interaction based on a menu that provides the user with different options to start the platform, which can be helpful during the development process and later on for maintenance and software updates. Figure 5–4 shows a standard boot loader menu. For sample source code, check out the Menu.c file located in the Src\Boot loader\Eboot directory of a reference BSP or in the %_WINCEROOT%\Platform\Common\Src\Common\Boot\Blmenu folder.

```
Microsoft Windows CE Ethernet Bootloader Common Library Version 1.1 Built Nov  3
 2005 10:24:54
Microsoft Windows CE Ethernet Bootloader 1.11 for the Marvell MainstoneIII Develop
ment Platform Built Oct 30 2005

Press [ENTER] to download now or [SPACE] to cancel.

Initiating image download in 2 seconds.

EBoot Loader Configuration:
0) IP address: 192.168.0.54
1) Subnet mask: 0.0.0.0
2) Boot delay: 3 seconds
3) DHCP: (Enabled)
4) Reset to factory default configuration
5) RNDIS MAC address: 0-0-0-0-FF-FF
6) Download new image at startup
7) Boot device order: SMSC -> USB-> PCMCIA0 -> PCMCIA1
8) Debug serial port: BTUART
D) Download image now
L) Launch existing flash resident image now
U) Boot to ULDR: (No)


Enter your selection:
```

Figure 5-4 An example of a boot loader menu

Additional Features

Beyond the core functionality, you can also add convenience features, such as download progress indication, support for downloading multiple .bin files during the same download session (multi-bin image notification), or downloading only trusted images. Additionally, you can implement support for downloading run-time images directly from Platform Builder. To accomplish this task, the boot loader must prepare a BOOTME packet with details about the target device and send it over the underlying transport. If the transport is Ethernet then this packet is broadcasted over the network. The libraries provided by Microsoft support these features, and you can customize them to suit your needs.

NOTE **OEM boot loader functions**

For detailed information about required and optional boot loader functions as well as boot loader structures, see the section "Boot Loader Reference" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN® Web site at *http://msdn2.microsoft.com/en-us/library/aa908395.aspx*.

Adapting an OAL

A significant portion of the BSP adaptation revolves around the platform-specific part of the OAL. If the new platform uses a CPU that is not currently supported, then the OAL adaptation requires you to modify most of the OAL code to support the new processor architecture. On the other hand, if the new hardware is very similar to the reference BSP's platform, you might be able to reuse most of the existing code base.

OEM Address Table

The kernel performs specialized tasks, such as initializing virtual memory, and cannot rely on a boot loader for this because the kernel must be entirely self-contained. Otherwise, the operating system would depend on the presence of a boot loader and it would not be possible to bootstrap the run-time image directly. Yet, to establish virtual-to-physical address mappings through the Memory Management Unit (MMU), the kernel must know the memory layout of the underlying hardware platform. To obtain this information, the kernel uses an important table called OEMAddressTable (or g_oalAddressTable) that defines static virtual memory regions. The OAL includes a declaration of OEMAddressTable as a read-only section and one of the first actions taken by the kernel is to read this section, set up corresponding virtual memory mapping tables, and then transition to the virtual address where the kernel can execute code. The kernel can determine the physical address of the OEMAddressTable in linear memory based on the address information available in the run-time image.

You must indicate any differences in the memory configuration of a new hardware platform by modifying the OEMAddressTable. The following sample code illustrates how to declare the OEMAddressTable section.

```
;--------------------------------------------------------------
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 Entry Point

Similar to the boot loader, the OAL contains a StartUp entry point to which the boot loader or system can jump in order to start kernel execution and initialize the system. For example, the assembly code for putting the processor in the correct state is usually the same as the code used in the boot loader. In fact, code sharing between the boot loader and the OAL is a common practice to minimize code duplication in the BSP. Yet not all code runs twice. For example, on hardware platforms that start from a boot loader, StartUp directly jumps to the KernelStart function, as the boot loader has already performed the intialization groundwork.

The KernelStart function initializes the memory-mapping tables as discussed in the previous section and loads the kernel library to run Microsoft kernel code. The Microsoft kernel code now calls the OEMInitGlobals function to pass a pointer to a static NKGLOBALS structure to the OAL and retrieve a pointer to an OEMGLOBALS structure in the form of a return value from the OAL. NKGLOBALS contains pointers to all the functions and variables used by KITL and the Microsoft kernel code. OEMGLOBALS has pointers to all the functions and variables implemented in the OAL for the BSP. By exchanging pointers to these global structures, Oal.exe and Kernel.dll have access to each other's functions and data, and can continue with architecture-generic and platform-specific startup tasks.

The architecture-generic tasks include setting up page tables and cache information, flushing TLBs, initializing architecture-specific buses and components, setting up the interlocked API code, loading KITL to support kernel communication for debugging purposes, and initializing the kernel debug output. The kernel then proceeds by calling the OEMInit function through the function pointer in the OEMGLOBALS structure to perform platform-specific initialization.

Table 5–5 lists the platform-specific funtions that Kernel.dll calls and that you might have to modify in your BSP to run Windows Embedded CE on a new hardware platform.

Table 5-5 Kernel startup support functions

Function	Description
OEMInitGlobals	Exchanges global pointers between Oal.exe and Kernel.dll.
OEMInit	Initializes the hardware interfaces for the platform.

Table 5-5 Kernel startup support functions (Continued)

Function	Description
OEMGetExtensionDRAM	Provides information about additional RAM, if available.
OEMGetRealTime	Retrieves time from RTC.
OEMSetAlarmTime	Sets the RTC alarm.
OEMSetRealTime	Set the time in the RTC.
OEMIdle	Puts CPU in idle state when no threads are running.
OEMInterruptDisable	Disables particular hardware interrupt.
OEMInterruptEnable	Enables particular hardware interrupt.
OEMInterruptDone	Signals completion of interrupt processing.
OEMInterruptHandler	Handles interrupts (is different for SHx processors).
OEMInterruptHandler	Handles FIQ (specific for ARM processors).
OEMIoControl	IO control code for OEM information.
OEMNMI	Supports a non maskable interrupt (specific to SHx processor).
OEMNMIHandler	Handler for non maskable interrupt (specific to SHx processor).
OEMPowerOff	Puts CPU in suspend state and takes care of final power down operations.

Kernel Independent Transport Layer

The OEMInit function is the main OAL routine that initializes board-specific peripherals, sets up the kernel variables, and starts KITL by passing a KITL IOCTL to the kernel. If you added and enabled KITL in the run-time image, the kernel starts KITL for debugging over different transport layers, as discussed in Chapter 4, "Debugging and Testing the System."

Table 5–6 lists the functions that the OAL must include to enable KITL support on a new platform.

Table 5-6 KITL support functions

Function	Description
OEMKitlInit	Initializes KITL.
OEMKitlGetSecs	Returns the current time in seconds.
TransportDecode	Decodes recevied frames.
TransportEnableInt	Enables or disables KITL interrupt if it is interrupt based.
TransportEncode	Encodes data accroding the transport's required frame structure.
TransportGetDevCfg	Retrieves the device's KITL transport configuration.
TransportReceive	Receives a frame from the transport.
TransportSend	Sends a frame using the transport.
KitlInit	Initializes KITL system.
KitlSendRawData	Sends raw data using the transport bypassing the protocol.
KitlSetTimerCallback	Registers a callback that is called after a specified amount of time.
KitlStopTimerCallback	Disables a timer used by the above routine.

Profile Timer Support

Located at the core of the operating system, the OAL is a perfect choice for mechanisms to measure the performance of the system and support performance optimization. As discussed in Chapter 3, "Performing System Programming," you can use the Interrupt Latency Timing (ILTiming) tool to measure the time it takes to invoke an interrupt service routine (ISR) after an interrupt occurred (ISR latency) and the time between when the ISR exits and the interrupt service thread (IST) actually starts (IST latency). However, this tool requires a system hardware tick timer or alternative high-resolution timer that is not available on all hardware platforms. If the new hardware platform supports a high-resolution hardware timer, you can support ILTiming and similar tools by implementing the functions listed in Table 5–7.

Table 5-7 Profile timer support functions

Function	Description
OEMProfileTimerEnable	Enables a profiler timer.
OEMProfileTimerDisable	Disables a profiler timer.

NOTE Thread scheduling and interrupt handling

The OAL must also support interrupt handling and the kernel scheduler . The scheduler is independent of the processor type, yet interrupt handling must be optimized for different types of processors.

Integrating New Device Drivers

Apart from the core system functions, the BSP also contains device drivers for peripherals. These peripheral devices can be components on the processor chip or external components. Even when separate from the processor, they remain an integral part of the hardware platform.

Device Driver Code Locations

Table 5–8 lists the source code locations for device drivers according to the PQOAL model. If your BSP is based on the same processor as the reference BSP, then the adaptation of device drivers mainly requires modification to the source code in the %TGTPLATROOT% folder. It is also possible to add new drivers to the BSP if the new platform includes peripherals that are not present in the reference platform. For more information about developing device drivers, see Chapter 6, "Developing Device Drivers."

Table 5-8 Source code folders for device drivers

Folder	Description
%_WINCEROOT%\Platform\%_TGTPLAT%	Contains platform dependent drivers.
%_WINCEROOT%\Platform\Common\Src\Soc	Contains drivers for processor-native peripherals.

Table 5-8 Source code folders for device drivers (Continued)

Folder	Description
%_WINCEROOT%\Public\Common\Oak\Drivers	Contains drivers for non-native peripherals that include external controllers.

Modifying Configuration Files

If you cloned your BSP from an existing BSP, all configuration files are already in place. However, it is important that you review the memory layout in the Config.bib file, as explained in detail in Lesson 2. The other configuration files require modifications only if you added new drivers or modified components in the BSP, as explained in Chapter 2, "Building and Deploying a Run-Time Image."

Lesson Summary

It is advantageous to start the BSP development process by cloning an appropriate reference BSP. Ideally, this BSP should be based on the same or similar hardware platform because this makes the most of tested and proven production features. Windows Embedded CE features a PQOAL architecture and Platform Builder tools that facilitate the cloning process. The goal is to create a bootable system with minimum customizations and then add additional features and support for peripheral devices as necessary.

The first component of a BSP that you might have to adapt is the boot loader, which is responsible for initializing the hardware platform and passing execution to the kernel. The second component is the OAL, which contains the platform-specific code that the kernel needs for hardware initialization, interrupt handling, and timer handling for the thread scheduler, KITL, and kernel debug output. The third part of the BSP you must adapt is the device drivers for peripheral devices. The fourth part of the BSP requiring adaptation is the configuration files that control the build process, determine the memory layout, and specify system configuration settings. If the BSP adaption is based on a reference BSP for the same processor architecture, then most of the CPU-related and memory controller-related BSP code can remain unchanged. You only need to address platform-specific code portions that focus on bringing up the hardware, rather than creating the necessary setup for the BSP.

Lesson 2: Configuring Memory Mapping of a BSP

Memory management in Windows Embedded CE has changed significantly from previous versions. In past versions, all processes shared the same 4 GB address space. With CE 6.0, each process has its own unique address space. The new system of managing virtual memory enables CE 6.0 to run up to 32,000 processes in contrast to the previous 32 processes limitation. This lesson covers the details of the new memory architecture and management, so that you can map virtual memory regions to correct physical memory addresses on the platform.

After this lesson, you will be able to:

- Describe how Windows Embedded CE manages virtual memory.
- Configure static memory mappings for a hardware platform.
- Map noncontiguous physical memory to virtual memory on the system.
- Share resources between OAL and device drivers.

Estimated lesson time: 15 minutes.

System Memory Mapping

Windows Embedded CE uses a paged virtual memory management system with a 32-bit virtual address space, mapped to physical memory by using the MMU. With 32 bits, the system can address a total of 4 GB of virtual memory, which CE 6.0 divides into the following two areas (see Figure 5–5):

- **Kernel space** Located in the upper 2 GB of virtual memory and shared between all application processes running on the target device.
- **User space** Located in the lower 2 GB of virtual memory and used exclusively by each individual process. Each process has its own unique address space. The kernel manages this mapping of the process address space when a process switch occurs. Processes cannot access the kernel address space directly.

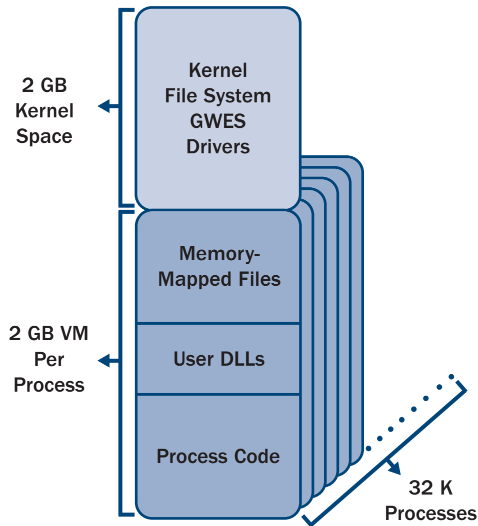


Figure 5-5 Virtual memory space in Windows Embedded CE 6.0

Kernel Address Space

Windows Embedded CE 6.0 divides the kernel address space further into several regions for specific purposes, as illustrated in Figure 5–6. The lower two regions of 512 MB each statically map physical memory into cached and non-cached virtual memory. The middle two regions for kernel execute in place (XIP) DLLs and Object Store are important for the OS design. The remaining space is for kernel modules and CPU-specific purposes.

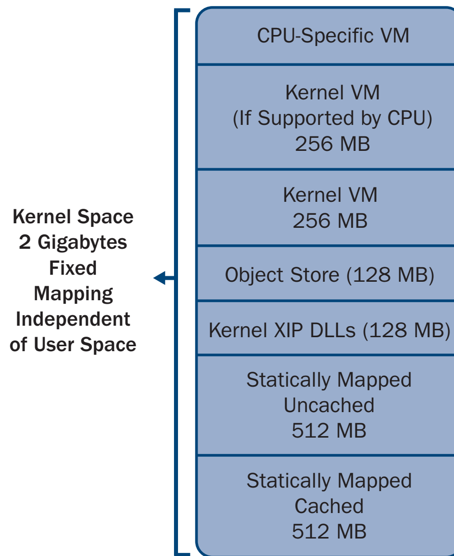


Figure 5-6 Kernel space in Windows Embedded CE 6.0

Table 5–9 summarizes the kernel virtual memory regions with start and end addresses.

Table 5-9 Kernel memory regions

Start Address	End Address	Description
0xF0000000	0xFFFFFFFF	Used for CPU specific system trap and kernel data pages.
0xE0000000	0xEFFFFFFF	Kernel virtual machine, it is CPU dependent, for example this space is not available for SHx.
0xD0000000	0xDFFFFFFF	Used for all kernel mode modules of the OS.
0xC8000000	0xCFFFFFFF	Object Store used for RAM file system, database and registry.
0xC0000000	0xC7FFFFFF	XIP DLLs.
0xA0000000	0xBFFFFFFF	Non-cached mapping of physical memory.
0x80000000	0x9FFFFFFF	Cached mapping of physical memory.

Process Address Space

The process address space ranges from 0x00000000 through 0x7FFFFFFF, and is divided into a CPU-dependent kernel data section, four main process regions, and a 1 MB buffer between user and kernel spaces. Figure 5–7 illustrates the main regions. The first process region of 1 GB is for application code and data. The next process region of 512 MB is for the DLLs and read-only data. The next two regions of 256 MB and 255 MB are for memory-mapped objects and the shared system heap. The shared system heap is read-only for the application process, but readable and writable for the kernel.

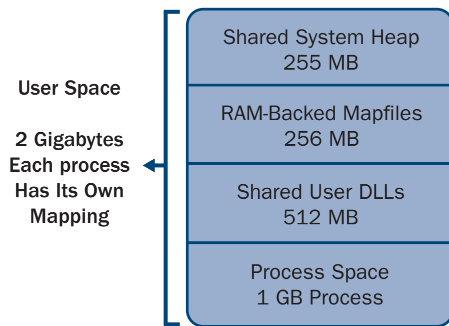


Figure 5-7 Process space in Windows Embedded CE 6.0

Table 5–10 summarizes the virtual memory regions in user space with start and end addresses.

Table 5-10 Process memory regions

Start Address	End Address	Description
0x7FF00000	0x7FFFFFFF	An unmapped buffer for protection between user and kernel spaces.
0x70000000	0x7FEFFFFF	Shared heap between the kernel and processes.
0x60000000	0x6FFFFFFF	Memory-mapped file objects that do not correspond to an actual physical file, mainly for backward compatibility with applications that used RAM-backed map files for inter-process communication.
0x40000000	0x5FFFFFFF	DLLs loaded into the process and read-only data.

Table 5-10 Process memory regions (Continued)

Start Address	End Address	Description
0x00010000	0x3FFFFFFF	Application code and data.
0x00000000	0x00010000	CPU-dependent user kernel data (read-only for user processes).

Memory Management Unit

Windows Embedded CE 6.0 requires the processor to provide a memory mapping mechanism to associate physical memory with virtual memory, up to a maximum of 512 MB of mapped physical memory. Figure 5–8 shows an example with 32 MB of flash memory and 64 MB of RAM mapped into the cached and non-cached static mapping regions of the kernel. On ARM–based and x86–based platforms, the memory mapping relies on a user-defined OEMAddressTable, whereas on the SHx–based and MIPS–based platforms, the mapping is directly defined by the CPU. The Memory Management Unit (MMU) is responsible for managing the physical-to-virtual address mappings.

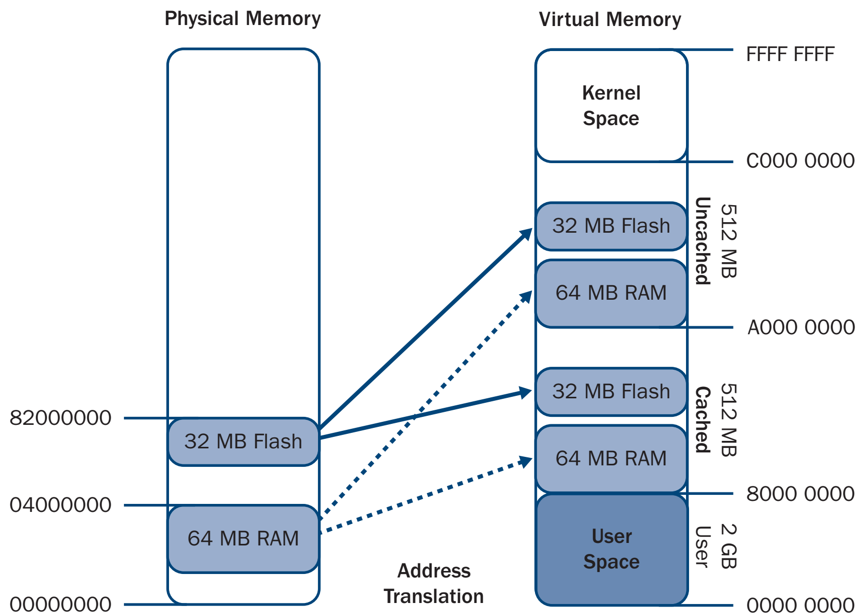


Figure 5-8 Physical-to-virtual memory mapping example

NOTE MMU initialization

The kernel initializes the MMU and creates the necessary page tables during system startup. This processor-specific part of the kernel depends on the architecture of the hardware platform. For implementation details, refer to the Windows Embedded CE private code, located in subdirectories per processor type under %_PRIVATEROOT%\Winceos\Coreos\Kernel.

Statically Mapped Virtual Addresses

The virtual memory regions depicted in Figure 5–8 are statically mapped virtual addresses to emphasize the fact that they are defined at startup time and the mapping does not change. Statically mapped virtual addresses are always available and directly accessible in kernel mode. It is noteworthy, however, that Windows Embedded CE also supports static mapping at runtime by means of CreateStaticMapping and NKCreateStaticMapping APIs. These functions return a non-cached virtual address mapped to the specified physical address.

Dynamically Mapped Virtual Addresses

The kernel can also manage the mapping of physical-to-virtual addresses dynamically, which is the technique used for all non-static mappings. A driver or DLL loaded into the kernel through LoadKernelLibrary can reserve a region of virtual memory in the kernel address space by calling VirtualAlloc and then map the virtual address to a physical address by creating a new page-table entry through a call to VirtualCopy. This is a common technique to map virtual addresses to the registers or frame buffers of peripheral devices in order to perform input/output operations. If the mapped buffer is no longer needed, the device driver or DLL can call VirtualFree to remove page-table entry and free the allocated virtual memory.

Memory Mapping and the BSP

You must customize two elements to include information about static memory mappings in a BSP:

- **Config.bib file** Provides information about how the system should use the different memory regions on the platform. For example, you can specify how much memory is available for the OS, how much can be used as free RAM and also reserve memory for specific needs.

- **OEMAddressTable** Provides information about the memory layout of the underlying platform, as discussed in Lesson 1. The memory specified in Config.bib should also be mapped in the OEMAddressTable.

Mapping Noncontiguous Physical Memory

As mentioned in Chapter 2, "Building and Deploying a Run-Time Image," you must define a single contiguous region in the RAMIMAGE memory region for the operating system in the MEMORY section of the Config.bib file. The system uses this definition to load the kernel image and any modules you specified in the MODULES and FILES sections. You cannot define multiple RAMIMAGE regions, yet the OAL can extend the RAMIMAGE region and provide additional noncontiguous memory sections at runtime.

Table 5–11 summarizes important variables and functions to extend the RAM region.

Table 5-11 Variables and functions to extend the RAM region

Variable/Function	Description
MainMemoryEndAddress	This variable indicates the end of the RAM region. The kernel sets this variable initially according to the size reserved for the operating system in the Config.bib file. The OAL OEMInit function can update this variable if additional contiguous memory is available.
OEMGetExtensionDRAM	The OAL can use this function to report the presence of an additional region of noncontiguous memory to the kernel. OEMGetExtensionDRAM returns the start address and length of the second region of memory.
pNKEnumExtensionDRAM	The OAL can use this function pointer to report the presence of more than one additional region of memory to the kernel. This mechanism supports up to 15 different noncontiguous memory regions. If you implement the pNKEnumExtensionDRAM function pointer, then OEMGetExtensionDRAM is not called during the startup process

Enabling Resource Sharing between Drivers and the OAL

Device drivers often need access to physical resources, such as memory-mapped registers or DMA buffers, yet drivers cannot directly access physical memory because the system only works with virtual addresses. For device drivers to gain access to physical memory, the physical addresses must be mapped to virtual addresses.

Dynamically Accessing Physical Memory

If a driver requires physically contiguous memory, as in the case of buffers required for DMA operations, the driver can allocate contiguous physical memory by using the AllocPhysMem function. If the allocation is successful, AllocPhysMem returns a pointer to the virtual address that corresponds to the specified physical address. Because the system allocates memory, it is important to free the memory later on when it is no longer needed by calling FreePhysMem.

On the other hand, if a driver requires non-paged access to a physical memory region defined in Config.bib, you can use the MmMapIoSpace function. MmMapIoSpace returns a non-paged virtual address that is directly mapped to the specified physical address. This function is typically used to access device registers.

Statically Reserving Physical Memory

Occasionally, it may be necessary to share a common region of physical memory between drivers or between a driver and the OAL (such as between an IST and an ISR). Similar to sharing a memory region for boot arguments between boot loader and kernel, you can reserve a shared memory region for driver communication purposes in the Config.bib file. A standard practice is to use the DRIVER_GLOBALS structure defined in Drv_glob.h, as mentioned in Lesson 1.

Communication between Drivers and the OAL

In addition to the standard set of IOCTLs required by the kernel, drivers can communicate with the OAL through custom IOCTLs implemented in OEMIoControl. Kernel-mode drivers call OEMIoControl indirectly through KernelIoControl, passing in the custom IOCTL. The kernel does no processing, other than passing the parameters straight through to OEMIoControl. However, user-mode drivers cannot directly call custom OAL IOCTLs by default. The KernelIOControl calls from user-mode drivers or processes are passed to OEMIoControl through a kernel-mode component (Oalioctl.dll), which maintains a list of user-accessible OAL IOCTL codes. The call is rejected if the requested IOCTL code is not listed in this module, but you

can customize this list by modifying the Oalioctl.cpp file that is located in the %_WINCEROOT%\Public\Common\Oak\Oalioctl folder.

Lesson Summary

A good understanding of the Windows Embedded CE 6.0 memory architecture is a must for every CE developer. Specifically for BSP developers, it is important to know how CE 6.0 maps available physical memory into the virtual memory address space. Accessing memory from OAL, kernel-mode modules, and user-mode drivers and applications requires a detailed understanding of static and dynamic mapping techniques that are available in kernel mode or user mode. For more information about the communication between kernel-mode and user-mode components, refer to Chapter 6, “Developing Device Drivers.”

Lesson 3: Adding Power Management Support to an OAL

As discussed in Chapter 3, "Performing System Programming," Windows Embedded CE 6.0 provides a comprehensive set of power management features based on a Power Manager component that OEM developers can customize to implement system power state definitions as appropriate for their hardware platforms. In relationship to the OAL, implementing power management capabilities is a twofold task. You need to enable the operating system to control the power state of the hardware components and you need to enable the hardware platform to inform the operating system about power state changes. Most embedded devices require at least basic power management support to reduce power consumption and prolong battery life.

After this lesson, you will be able to:

- Describe how to reduce processor power consumption.
- Identify the transition paths to suspend and resume the system.

Estimated lesson time: 15 minutes.

Power State Transitions

Embedded devices that are not constantly in use, such as personal digital assistants (PDAs), operate for extended periods of time in an idle state, thus providing an opportunity to preserve energy by switching from full-power mode to a reduced-power mode or suspend state. Most embedded processors available on the market today support these transitions, as illustrated in Figure 5–9.

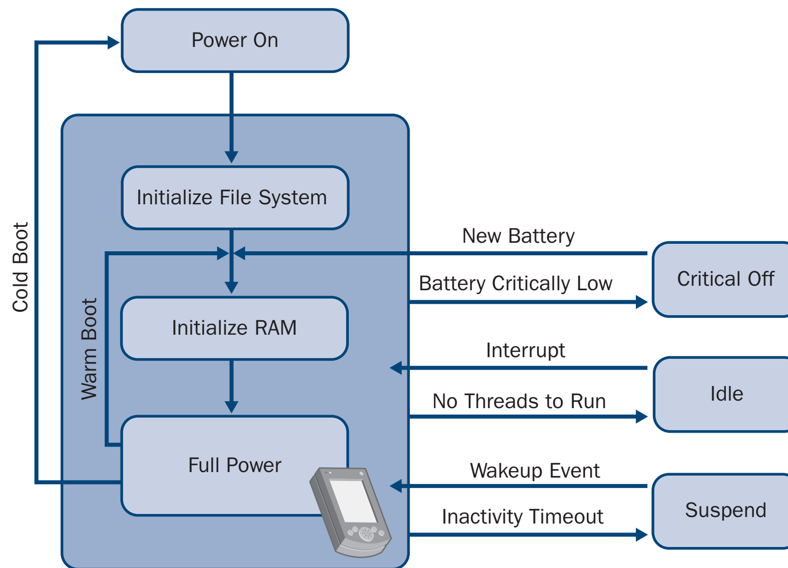


Figure 5-9 Power state transitions

Windows Embedded CE can respond to power-related events in the following ways:

- **Battery critically low** The system switches into Critical Off state in response to a nonmaskable interrupt (NMI) that a voltage comparator on the board triggers, so that the user can replace the battery and resume.
- **Idle** The system switches the CPU into reduced-power mode if the CPU has no worker threads to run and wakes up when an interrupt occurs.
- **Suspend** The system switches the device into Suspend state when the user presses the Off button or in response to an inactivity timeout and resumes in response to a wakeup event, such as the user pressing the power button again. On some embedded devices, the Suspend state corresponds to a true power-off state, in which case the system resumes with a cold boot.

Reducing Power Consumption in Idle Mode

To switch the device into reduced-power mode, Windows Embedded CE relies on the OEMIdle function, which the kernel calls when the scheduler has no threads to run. The OEMIdle function is a hardware-specific routine that depends on the capabilities of the platform. For example, if the system timer uses a fixed interval, then the OEMIdle function cannot really provide the expected power saving functionality because the system wakes up every time a timer interrupt occurs. On the other hand,

if the processor supports programmable interval timers, you can use the kernel's dwReschedTime variable to specify the amount of time spent in reduced-power mode.

On waking up from reduced-power mode, the system must update the kernel global variables used by the scheduler. This is particularly important for the CurMSec variable, which the system uses to keep track of the number of milliseconds since the last system boot. The wakeup source can be either the system timer or another interrupt. If the wakeup source is the system timer then the CurMSec variable is already updated before execution is passed back to the OEMIdle function. In other cases, the CurMSec does not contain an updated value. To learn more about the OEMIdle implementation details, refer to the Idle.c source code file, located in the %_WINCEROOT%\Platform\Common\Src\Common\Timer\Idle folder.

NOTE Kernel global variables

For detailed information about global variables that the kernel exports for scheduling, see the section "Kernel Global Variables for Scheduling" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn.microsoft.com/en-us/library/aa915099.aspx*.

Powering Off and Suspending the System

The maximum power saving state that a Windows Embedded CE device can support is the Power Off or Suspend state. The system can request the device to enter the Suspend state by calling GwesPowerOffSystem directly or SetSystemPowerState. Both functions eventually call the OEMPowerOff routine.

The OEMPowerOff routine is part of the OAL and responsible for switching the CPU into Suspend state. OEMPowerOff should also put the RAM into self-refresh mode if the processor does not automatically do so when it enters the Suspend state. You can also set up the interrupts to wake up the device. In handheld devices, this is typically the power-button interrupt, but you may use any wakeup event source that is appropriate for your target platform.

Entering the Suspend State

When entering the Suspend state, Windows Embedded CE performs the following sequence of steps:

1. GWES notifies the Taskbar about the power down event.
2. The system aborts calibration if in the calibration screen.

3. The system stops the Windows message queues. After step 3, the system enters single-thread mode, which prevents function calls that rely on blocking operations.
4. The system checks if the startup user interface (UI) must appear on resume.
5. The system saves video memory to RAM.
6. The system calls SetSystemPowerState (NULL, POWER_STATE_SUSPEND, POWER_FORCE).
7. Power Manager:
    a. Calls the FileSystemPowerFunction to power off the drivers related to the file system.
    b. Calls PowerOffSystem to inform the kernel to do the final power down.
    c. Calls Sleep(0) to invoke the scheduler.

> **NOTE FileSystemPowerFunction and PowerOffSystem**
> If the OS design does not include Power Manager or GWES, then the OEM must explicitly call FileSystemPowerFunction and PowerOffSystem.

8. Kernel:
    a. Unloads GWES process.
    b. Unloads Filesys.exe.
    c. Calls OEMPowerOff.
9. OEMPowerOff configures the interrupts and puts the CPU in Suspend state.

Waking Up from Suspend State

When a pre-configured interrupt wakes up the system, the associated ISR runs and returns to the OEMPowerOff routine. On returning from this function, the system goes through the resume sequence, which includes the following steps:

1. OEMPowerOff re-configures interrupts to original state and returns.
2. Kernel:
    a. Calls InitClock to re-initialize the system timer.
    b. Starts Filesys.exe with power on notification.
    c. Starts GWES with power on notification.
    d. Re-initializes KITL interrupt if it was in use.

3. Power Manager calls FileSystemPowerFunction with power on notification.
4. GWES:
    a. Restores video memory from RAM.
    b. Powers on Windows Manager.
    c. Sets the display contrast.
    d. Shows startup UI if required.
    e. Notifies Taskbar of resume.
    f. Notifies User Subsystem.
    g. Triggers applications as required.

NOTE Registering wakeup sources

If the OAL supports the kernel IOCTL_HAL_ENABLE_WAKE, applications can register wake up sources. For detailed information, see the section "IOCTL_HAL_ENABLE_WAKE" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa914884.aspx*.

Supporting the Critical Off State

On hardware platforms equipped with a voltage comparator that triggers NMI, you can implement support for the Critical Off state to protect the user from data loss in low-battery conditions. On x86 hardware, the kernel exports the OEMNMIHandler function to capture critical events in the system. On other systems, you might have to implement a custom IST that calls SetSystemPowerState to turn off the system gracefully with the help of Power Manager. The Critical Off state typically corresponds to the Suspend state with dynamic RAM refresh enabled.

NOTE Battery level reaches zero

When implementing Critical Off state support, make sure you trigger the NMI at a point when the system still has time to perform all power down tasks , such as powering down peripherals, putting RAM into self-refresh, perhaps setting a wakeup condition, and suspending the CPU.

Lesson Summary

Power management is an important Windows Embedded CE feature that ensures efficient power consumption on target devices. OEMs should implement power management features in the OAL to enable transitions from full-power mode to Idle and Suspend modes and Critical Off state for battery-powered devices. Implementing power management support involves re-synchronizing timer-related kernel variables, powering down peripherals, putting RAM into self-refresh mode, setting wakeup conditions, and suspending the CPU. It is not trivial to implement these low-level routines, yet Microsoft provides sufficient reference code in the sample BSPs to get a better understanding of the implementation details.

Lab 5: Adapting a Board Support Package

In this lab you clone a reference BSP in Visual Studio 2005 with Platform Builder and use it to build a run-time image. As the underlying platform, this lab uses the Device Emulator because this platform can run on the Windows Embedded CE development computer. Microsoft included the Device Emulator BSP in Platform Builder as a reference BSP.

NOTE Detailed step-by-step instructions

To help you successfully master the procedures presented in this Lab, see the document "Detailed Step-by-Step Instructions for Lab 5" in the companion material for this book.

▶ Clone a BSP

1. In Visual Studio 2005, open the Tools menu, click Platform Builder For CE 6.0, and then click Clone BSP.
2. In the Clone Board Support Package window select Device Emulator: ARMV4I as the Source BSP from the drop-down list.
3. Under New BSP Information enter the information shown in Table 5–12 (see also Figure 5–10):

Table 5-12 New BSP details

Parameter	Value
Name	DeviceEmulatorClone
Description	Clone of the Device Emulator BSP
Platform Directory	DeviceEmulatorClone
Vendor	Contoso Ltd.
Version	0.0

4. Select Open New BSP Catalog File In Catalog Editor check box and then click Clone.
5. Verify that Platform Builder clones the Device Emulator BSP successfully, and then in the corresponding Clone BSP dialog box, click OK.
6. Verify that Visual Studio automatically opens the DeviceEmulatorClone.pbcxml catalog file. Close the catalog editor without making any changes.

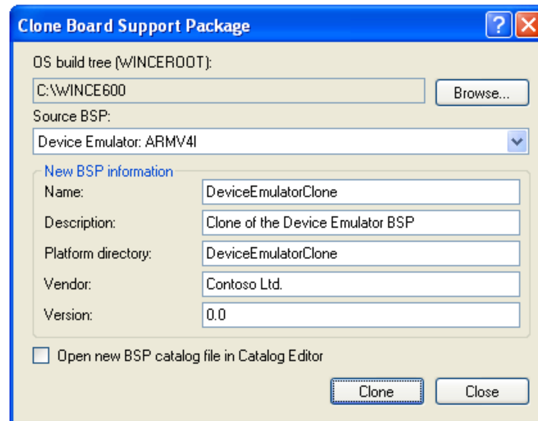


Figure 5-10 BSP cloning information

▶ Create a Run-Time Image

1. In order to validate our cloned BSP, create a new OS design based on the DeviceEmulatorClone BSP. Call the OS design DeviceEmulatorCloneTest, as illustrated in Figure 5–11 (see also Lab 1 in Chapter 1 for details on how to accomplish this step).
2. Choose Industrial Device in the Design Templates and Industrial Controller in the Design Template Variants. Accept the default options in the subsequent steps of the wizard.
3. After Platform Builder generates the DeviceEmulatorCloneTest project, verify the OS design by examining the catalog items in Catalog Items View.
4. Verify that the Debug build configuration is enabled by opening Configuration Manager on the Build menu and seeing if the Active Solution Configuration list box displays DeviceEmulatorClone ARMV4I Debug.
5. On the Build menu, click Build Solution.
6. After the build is completed, configure the Connectivity Options to use the Device Emulator.
7. Open the Target menu and click Attach Device to download the run-time image to the Device Emulator and start Windows Embedded CE. Notice the debug messages in the Output window of Visual Studio 2005. Wait until the device has started up completely.

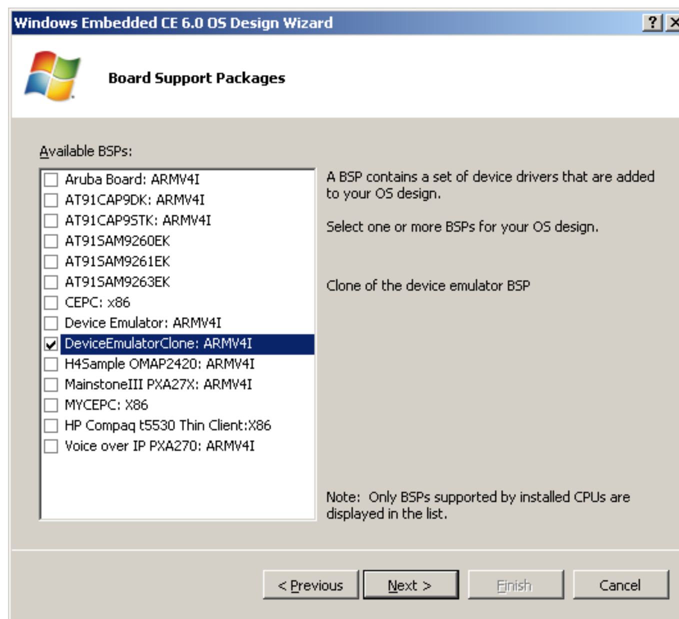


Figure 5-11 A new OS design based on the DeviceEmulatorClone BSP

NOTE BSP adaptation

Device Emulator emulates the same hardware platform for both the reference BSP and the cloned BSP. For this reason, the new run-time image runs on Device Emulator without further adaption. In practice, however, the underlying hardware is different in most cases, requiring BSP adaptations to start CE successfully.

▶ Customize the BSP

1. Detach from the target device and close Device Emulator.
2. In Visual Studio, open the init.c source code file that you can find in the %_PLATFORMROOT%\DeviceEmulatorClone\Src\Oal\Oallib folder, as illustrated in Figure 5-12.
3. Search for the OAL function OEMGetExtensionDRAM and add the following line of code to print a debug message in the Output window of Visual Studio during system startup.

```
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. Rebuild the run-time image to includes the changes, and then attach to the device again in order to download and start the new run-time image in Device Emulator. Verify that Windows Embedded CE prints the debug message in the Output window.

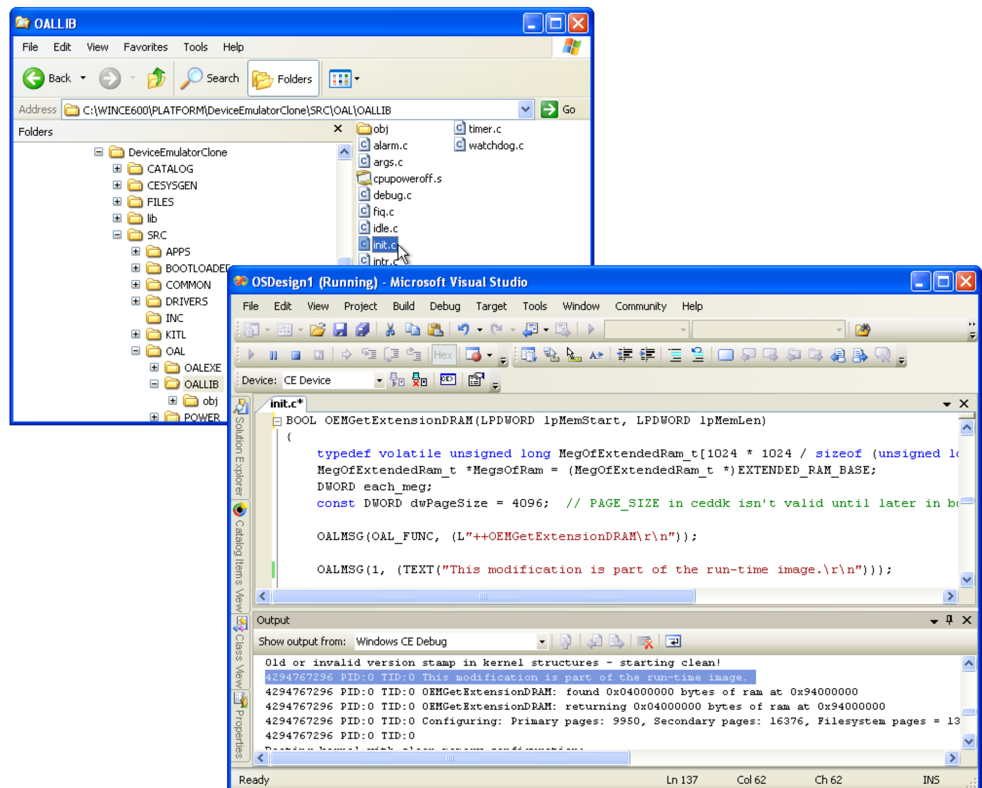


Figure 5-12 DeviceEmulatorClone BSP customization

Chapter Review

The adaptation of a BSP is one of the most complicated and critical development tasks that OEMs face when porting Windows Embedded CE 6.0 to a new hardware platform. To facilitate this undertaking, Microsoft provides reference BSPs with Platform Builder and encourages OEMs to start the development process by cloning the most suitable BSP. The PQOAL–based BSPs follow a well-organized folder and file structure to separate platform-agnostic and platform-specific code by processor type and OAL function so that OEMs can focus on platform-specific implementation details without getting side tracked by general aspects of the kernel or operating system.

OEM developers should consider the following recommendations to ensure a successful adaptation of a BSP:

- **Study the Windows Embedded CE reference BSPs** Windows Embedded CE BSPs follow a well-defined architecture with close relationships to the kernel. This makes it necessary to implement numerous APIs that the kernel requires to run the operating system. Knowing these APIs and their purpose is very important. The PQOAL–based architecture is continually evolving.
- **Clone a BSP** Avoid writing a new BSP completely from scratch. Instead, clone a BSP to jump start the adaptation process. By reusing as much code as possible from a reference BSP, you not only shorten development time, but also increase the quality of your solution and provide a solid foundation for efficient handling of future upgrades.
- **Boot loader and BLCOMMON** Use BLCOMMON and related libraries when implementing a boot loader because these libraries provide useful hardware-independent features for downloading run-time images and enabling users to interact with the target device during the startup process.
- **Memory and BSPs** Make sure you thoroughly understand how Windows Embedded CE 6.0 deals with physical and virtual memory. Configure <*Boot loader*>.bib and Config.bib files to provide accurate information about available memory to the operating system and adjust the entries in the OEMAddressTable, if necessary. Keep in mind that you cannot directly access physical memory in Windows Embedded CE. Use the correct memory-mapping APIs to map physical memory addresses to virtual memory addresses.

- **Implement power management** Implement the OEMIdle function to enable the system to switch the CPU into Idle mode. Consider implementing OEMPowerOff as well, if your platform supports power state transitions into Suspend mode in response to user actions or critical battery levels.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- PQOAL
- Boot loader
- KernelIoControl
- Driver globals

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Access the Hardware Registers of a Peripheral Device

Implement a device driver for peripheral hardware and access the hardware registers by using the MmMapIoSpace API to interact with the device. Note that it is not possible to call MmMapIoSpace from an application.

NOTE Emulator restrictions

Because Device Emulator emulates an ARM processor in software, you cannot access hardware devices. You must use a genuine hardware platform to perform this suggested practice.

Reorganize Platform Memory Mappings

By modifying the Config.bib file of the cloned Device Emulator BSB, you can increasingly reduce the available RAM on the system and study the impact in terms of available memory on the system by using the memory information APIs or Platform Builder tools.

Chapter 6

Developing Device Drivers

Device drivers are components that enable the operating system (OS) and user applications to interact with peripheral hardware that is integrated or attached to a target device, such as the Peripheral Component Interconnect (PCI) bus, keyboard, mouse, serial ports, display, network adapter, and storage devices. Rather than accessing the hardware directly, the operating system loads the corresponding device drivers, and then uses the functions and input/output (I/O) services that these drivers provide to carry out actions on the device. In this way, the Microsoft® Windows® Embedded CE 6.0 R2 architecture remains flexible, extensible, and independent of the underlying hardware details. The device drivers contain the hardware-specific code, and you can implement custom drivers in addition to the standard drivers that ship with CE to support additional peripherals. In fact, device drivers are the largest part of the Board Support Package (BSP) for an OS design. However, it is also important to keep in mind that poorly implemented drivers can ruin an otherwise reliable system. When developing device drivers, it is imperative to follow strict coding practices and test the components thoroughly in various system configurations. This chapter discusses best practices for writing device drivers with proper code structures, developing a secure and well-designed configuration user interface, ensuring reliability even after prolonged use, and supporting multiple power management features.

Exam objectives in this chapter:

- Loading and using device drivers on Windows Embedded CE
- Managing interrupts on the system
- Understanding memory access and memory handling
- Enhancing driver portability and system integration

Before You Begin

- To complete the lessons in this chapter, you must have the following:
- At least some basic knowledge about Windows Embedded CE software development, including fundamental concepts related to driver development, such as I/O control (IOCTL) and Direct Memory Access (DMA).
- An understanding of interrupt handling and how to respond to interrupts in a device driver.
- Familiarity with memory management in C and C++, as well as and knowledge of how to avoid memory leaks.
- A development computer with Microsoft Visual Studio® 2005 Service Pack 1 and Platform Builder for Windows Embedded CE 6.0 R2 installed.

Lesson 1: Understanding Device Driver Basics

On Windows Embedded CE, a device driver is a dynamic-link library (DLL) that provides a layer of abstraction between the underlying hardware, OS, and applications running on the target device. The driver exposes a set of known functions and provides the logic to initialize and communicate with the hardware. Software developers can then call the driver's functions in their applications to interact with the hardware. If a device driver adheres to a well-known application programming interface (API) such as Device Driver Interface (DDI), you can load the driver as part of the operating system, such as a display driver or a driver for a storage device. Without having to know details about the physical hardware, applications can then call standard Windows API functions, such as ReadFile or WriteFile, to use the peripheral device. You can support different types of peripherals by adding different drivers to the OS design without having to reprogram your applications.

After this lesson, you will be able to:

- Differentiate between native and stream drivers.
- Describe the advantages and disadvantages of monolithic and layered driver architectures.

Estimated lesson time: 15 minutes.

Native and Stream Drivers

A Windows Embedded CE device driver is a DLL that exposes the standard DllMain function as the entry point, so that a parent process can load the driver by calling LoadLibrary or LoadDriver. Drivers loaded by means of LoadLibrary can be paged out, but the operating system does not page out drivers loaded through LoadDriver.

While all drivers expose the DllMain entry point, Windows Embedded CE supports two different types of drivers: native drivers and stream drivers. Native CE drivers typically support input and output peripherals, such as display drivers, keyboard drivers, and touchscreen drivers. The Graphics, Windowing, and Events Subsystem (GWES) loads and manages these drivers directly. Native drivers implement specific functions according to their purpose, which GWES can determine by calling the GetProcAddress API. GetProcAddress returns a pointer to the desired function or NULL if the driver does not support the function.

Stream drivers, on the other hand, expose a well-known set of functions that enable Device Manager to load and manage these drivers. For Device Manager to interact with a stream driver, the driver must implement the Init, Deinit, Open, Close, Read, Write, Seek, and IOControl functions. In many stream drivers, the Read, Write, and Seek functions provide access to the stream content, yet not all peripherals are stream devices. If the device has special requirements beyond Read, Write, and Seek, you can use the IOControl function to implement the required functionality. The IOControl function is a universal function that can accommodate any special needs of a stream device driver. For example, you can extend a driver's functionality by passing a custom IOCTL command code and input and output buffers.

NOTE Native driver interface

Native drivers must implement different types of interfaces depending on the nature of the driver. For complete information about the supported driver types, see the section "Windows Embedded CE Drivers" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN® Web site at *http://msdn2.microsoft.com/en-us/library/aa930800.aspx*.

Monolithic vs. Layered Driver Architecture

Native and stream drivers only differ in terms of the APIs they expose. You can load both types of drivers during system startup or on demand, and both types can use a monolithic or layered design, as illustrated in Figure 6–1.

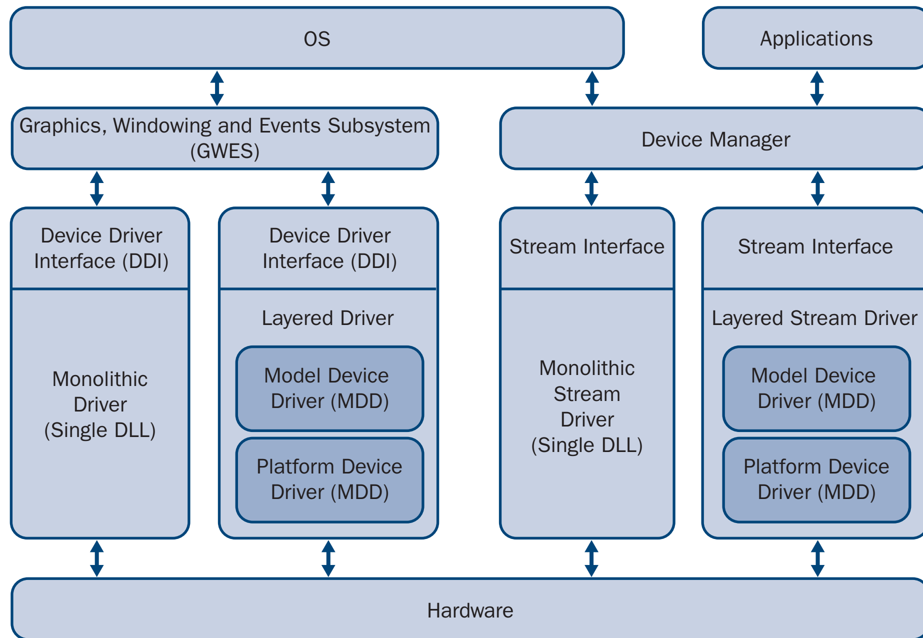


Figure 6-1 Monolithic and layered driver architectures

Monolithic Drivers

A monolithic driver relies on a single DLL to implement both the interface to the operating system and applications, and the logic to the hardware. The development costs for monolithic drivers are generally higher than for layered drivers, yet despite this disadvantage, monolithic drivers also have advantages. The primary advantage is a performance gain by avoiding additional function calls between separate layers in the driver architecture. Memory requirements are also slightly lower in comparison to layered drivers. A monolithic driver might also be the right choice for uncommon, custom hardware. If no layered driver code exists that you could reuse, and if this is a unique driver project, you might find it advantageous to implement a driver in a monolithic architecture. This is especially true if reusable monolithic source code is available.

Layered Drivers

In order to facilitate code reuse and lower development overhead and costs, Windows Embedded CE supports a layered driver architecture based on model device driver (MDD) and platform device driver (PDD). MDD and PDD provide an additional abstraction layer for driver updates and to accelerate the development of device

drivers for new hardware. The MDD layer contains the interface to the operating system and the applications. On the hardware side, MDD interfaces with the PDD layer. The PDD layer implements the actual functions to communicate with the hardware.

When porting a layered driver to new hardware, you generally do not need to modify the code in the MDD layer. It is also less complicated to duplicate an existing layered driver and add or remove functionality than to create a new driver from scratch. Many of the drivers included in Windows Embedded CE take advantage of the layered driver architecture.

NOTE MDD/PDD architecture and driver updates

The MDD/PDD architecture can help driver developers save time during the development of driver updates, such as providing quick fix engineering (QFE) fixes to customers. Restricting modifications to the PDD layer increases development efficiencies.

Lesson Summary

Windows Embedded CE supports native and stream drivers. Native drivers are the best choice for any devices that are not stream devices. For example, a display device driver must be able to process data in arbitrary patterns and is therefore a good candidate for a native driver. Other devices, such as storage hardware and serial ports, are good candidates for stream drivers because these devices handle data primarily in the form of ordered streams of bytes as if they were files. Both native and stream drivers can use either the monolithic or layered driver design. In general, it is advantageous to use the layered architecture based on MDD and PDD because it facilitates code reuse and the development of driver updates. Monolithic drivers might be the right choice if you want to avoid the additional function calls between MDD and PDD for performance reasons.

Lesson 2: Implementing a Stream Interface Driver

On Windows Embedded CE, a stream driver is a device driver that implements the stream interface API. Regardless of hardware specifics, all CE stream drivers expose the stream interface functions to the operating system so the Device Manager of Windows Embedded CE can load and manage these drivers. As the name implies, stream drivers are suitable for I/O devices that act as sources or sinks of streams of data, such as integrated hardware components and peripheral devices. It is also possible for a stream driver to access other drivers to provide applications with more convenient access to the underlying hardware. In any case, you need to know the stream interface functions and how to implement them if you want to develop a fully functional and reliable stream driver.

After this lesson, you will be able to:

- Understand the purpose of Device Manager.
- Identify stream driver requirements.
- Implement and use a stream driver.

Estimated lesson time: 40 minutes.

Device Manager

The Windows Embedded CE Device Manager is the OS component that manages the stream device drivers on the system. The OAL (Oal.exe) loads the kernel (Kernel.dll), and the kernel loads Device Manager during the boot process. Specifically, the kernel loads the Device Manager shell (Device.dll), which in turn loads the actual core Device Manager code (Devmgr.dll), which again is in charge of loading, unloading, and interfacing with stream drivers, as illustrated in Figure 6–2.

Stream drivers can be loaded as part of the operating system at boot time or on demand when the corresponding hardware is connected in a Plug and Play fashion. User applications can then use the stream drivers through file system APIs, such as ReadFile and WriteFile, or by means of DeviceIoControl calls. Stream drivers that Device Manager exposes through the file system appear to applications as regular file resources with special file names. The DeviceIoControl function, on the other hand, enables applications to perform direct input and output operations. However, applications interact with the stream drivers indirectly through Device Manager in both cases.

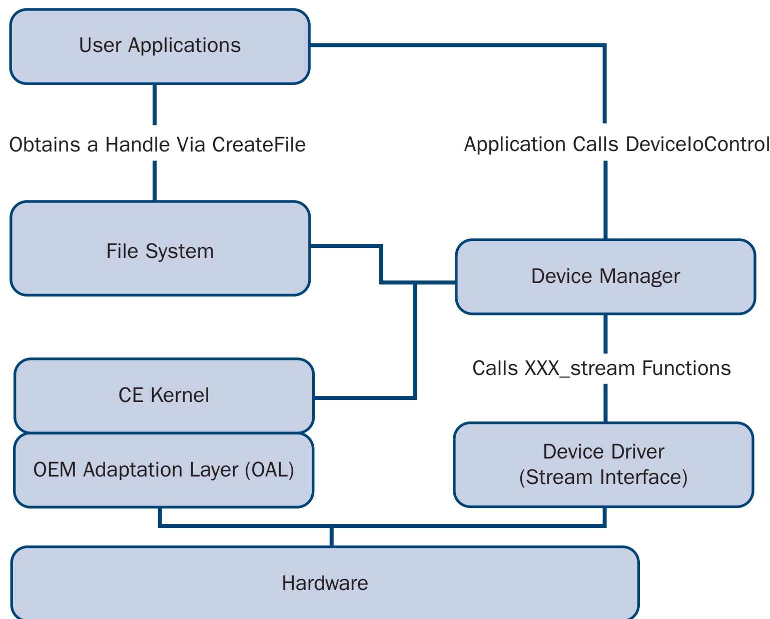


Figure 6-2 The Device Manager in Windows Embedded CE 6.0

Driver Naming Conventions

For an application to use a stream driver through the file system, the stream driver must be presented as a file resource so that the application can specify the device file in a CreateFile call to get a handle to the device. Having obtained the handle, the application can then use ReadFile or WriteFile to perform I/O operations, which Device Manager translates into corresponding calls to stream interface functions to perform the desired read and write actions. For Windows Embedded CE 6.0 to recognize stream device resources and redirect file I/O operations to the appropriate stream drive, stream drivers must follow a special naming convention that distinguishes these resources from ordinary files.

Windows Embedded CE 6.0 supports the following naming conventions for stream drivers:

- **Legacy names** The classical naming convention for stream drivers consists of three upper case letters, a digit, and a colon. The format is *XXX*[0–9]:, where *XXX* stands for the three-letter driver name and [*0–9*] is the index of the driver as specified in the driver's registry settings (see Lesson 3, "Configuring and Loading a Driver"). Because the driver index has only one digit, legacy names

only support up to ten instances of a stream driver. The first instance corresponds to index 1, the ninth instance uses index 9, and the tenth instance refers to index 0. For example, **CreateFile(L"COM1:"...)** accesses the stream driver for the first serial port by using the legacy name COM1:.

> **NOTE Legacy name limitation**
>
> The legacy naming convention does not support more than ten instances per stream driver.

- **Device names** To access a stream driver with an index of ten or higher, you can use the device name instead of the legacy name. The device name conforms to the format **\$device\XXX[index]**, where \$device\ is a namespace that indicates that this is a device name, *XXX* stands for the three-letter driver name and [*index*] is the index of the driver. The index can have multiple digits. For example, CreateFile(L"\$device\COM11"...) would access the stream driver for the eleventh serial port. Stream drivers with a legacy name can also be accessed, such as CreateFile(L"\$device\COM1"...).

> **NOTE Legacy and device name access**
>
> Although legacy names and device names differ in format and supported range of driver instances, CreateFile returns the same handle with access to the same stream driver in both cases.

- **Bus name** Stream drivers for devices on a bus, such as Personal Computer Memory Card International Association (PCMCIA) or Universal Serial Bus (USB), correspond to bus names that the relevant bus driver passes to Device Manager when enumerating the drivers available on the bus. Bus names relate to the underlying bus structure. The general format is **\$bus\BUSNAME_[bus number]_[device number]_[function number]**, where \$bus\ is a namespace that indicates that this is a bus name, *BUSNAME* refers to the name or type of the bus, and [*bus number*], [*device number*], and [*function number*] are bus-specific identifiers. For example, CreateFile(L"\$ bus\PCMCIA_0_0_0"...) accesses device 0, function 0, on PCMCIA bus 0, which might correspond to a serial port.

> **NOTE Bus name access**
>
> Bus names are primarily used to obtain handles for unloading and reloading bus drivers and for power management, but not for data read and write operations.

Stream Interface API

For Device Manager to load and manage a stream driver successfully, the stream driver must export a common interface, generally referred to as the stream interface. The stream interface consists of 12 functions to initialize and open the device, read and write data, power up or down the device, and close and de-initialize the device, as summarized in Table 6–1.

Table 6-1 Stream interface functions

Function Name	Description
XXX_Init	Device Manager calls this function to load a driver during the boot process or in answer to calls to ActivateDeviceEx in order to initialize the hardware and any memory structures used by the device.
XXX_PreDeinit	Device Manager calls this function before calling XXX_Deinit to give the driver a chance to wake sleeping threads and invalidate open handles in order to accelerate the de-initialization process. Applications do not call this function.
XXX_Deinit	Device Manager calls this function to de-initialize and de-allocate memory structures and other resources in response to a DeActivateDevice call after deactivating and unloading a driver.
XXX_Open	Device Manager calls this function when an application requests access to the device by calling CreateFile for reading, writing, or both.
XXX_PreClose	Device Manager calls this function to give the driver a chance to invalidate handles and wake sleeping threads in order to accelerate the unloading process. Applications do not call this function.
XXX_Close	Device Manager calls this function when an application closes an open instance of the driver, such as by calling the CloseHandle function. The stream driver must de-allocate all memory and resources allocated during the previous XXX_Open call.

Table 6-1 Stream interface functions (Continued)

Function Name	Description
XXX_Read	Device Manager calls this function in response to a ReadFile call to read data from the device and pass it to the caller. Even if the device does not provide any data to read, the stream device driver must implement this function to be compatible with Device Manager.
XXX_Write	Device Manager calls this function in response to a WriteFile call to pass data from the caller to the device. Similar to XXX_Read, XXX_Write is mandatory but can be empty if the underlying device does not support write operations, such as an input-only communications port.
XXX_Seek	Device Manager calls this function in response to a SetFilePointer call to move the data pointer to a particular point in the data stream for reading or writing. Similar to XXX_Read and XXX_Write, this function can be empty but must be exported to be compatible with Device Manager.
XXX_IOControl	Device Manager calls this function in response to a DeviceIoControl call to perform device-specific control tasks. For example, power management features rely on DeviceIoControl calls if the driver advertises a power management interface, such as to query power capabilities and manage the power status through the IOCTLs IOCTL_POWER_CAPABILITIES, IOCTL_POWER_QUERY, and IOCTL_POWER_SET. Applications can also use the DeviceIoControl function to read and write data in device drivers that do not use XXX_Write and XXX_Read. This is a common approach in many stream drivers.

Table 6-1 Stream interface functions (Continued)

Function Name	Description
XXX_PowerUp	Device Manager calls this function when the operating system comes out of a low power mode. Applications do not call this function. This function executes in kernel mode, can therefore not call external APIs, and must not be paged out because the operating system is running in single-threaded, non-paged mode. Microsoft recommends that drivers implement power management based on Power Manager and power management IOCTLs for suspend and resume functionality in a driver.
XXX_PowerDown	Device Manager calls this function when the operating system transitions into suspend mode. Similar to XXX_PowerUp, this function executes in kernel mode, can therefore not call external APIs, and must not be paged out. Applications do not call this function. Microsoft recommends that drivers implement power management based on Power Manager and power management IOCTLs.

NOTE XXX prefix

In the function names, the prefix *XXX* is a placeholder that refers to the three-letter driver name. You need to replace this prefix with the actual name in the driver code, such as COM_Init for a driver called COM, or SPI_Init for a Serial Peripheral Interface (SPI) driver.

Device Driver Context

Device Manager supports context management based on device context and open context parameters, which Device Manager passes as DWORD values to the stream driver with each function call. Context management is vital if a driver must allocate and deallocate instance-specific resources, such as blocks of memory. It is important to keep in mind that device drivers are DLLs, which implies that global variables and other memory structures defined or allocated in the driver are shared by all driver instances. Deallocating the wrong resources in response to an XXX_Close or XXX_Deinit call can lead to memory leaks, application failures, and general system instabilities.

Stream drivers can manage context information per device driver instance based on the following two levels:

1. **Device context** The driver initializes this context in the XXX_Init function. This context is therefore also called the Init Context. Its primary purpose is to help the driver manage resources related to hardware access. Device Manager passes this context information to the XXX_Init, XXX_Open, XXX_PowerUp, XXX_PowerDown, XXX_PreDeinit and XXX_Deinit functions.
2. **Open context** The driver initializes this second context in the XXX_Open function. Each time an application calls CreateFile for a stream driver, the stream driver creates a new open context. The open context then enables the stream driver to associate data pointers and other resources with each opened driver instance. Device Manager passes the device context to the stream driver in the XXX_Open function so that the driver can store a reference to the device context in the open context. In this way, the driver can retain access to the device context information in subsequent calls, such as XXX_Read, XXX_Write, XXX_Seek, XXX_IOControl, XXX_PreClose and XXX_Close. Device Manager only passes the open context to these functions in the form of a DWORD parameter.

The following code listing illustrates how to initialize a device context for a sample driver with the driver name SMP (such as 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;
}
```

Building a Device Driver

To create a device driver, you can add a subproject for a Windows Embedded CE DLL to your OS design, but the most common way to do it is to add the device driver's source files inside the Drivers folder of the Board Support Package (BSP). For detailed information about configuring Windows Embedded CE subprojects, see Chapter 1, "Customizing the Operating System Design."

A good starting point for a device driver is A Simple Windows Embedded CE DLL Subproject, which you can select on the Auto-Generated Subproject Files page in the Windows Embedded CE Subproject Wizard. It automatically creates a source code file with a definition for the DllMain entry point for the DLL, various parameter files, such as empty module-definition (.def) and registry (.reg) files, and preconfigures the Sources file to build the target DLL. For more detailed information about parameter files and the Sources file, see Chapter 2, "Building and Deploying a Run-Time Image."

Implementing Stream Functions

Having created the DLL subproject, you can open the source code file in Visual Studio and add the required functions to implement the stream interface and required driver functionality. The following code listing shows the definition of the stream interface functions.

```
// 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;
}
```

Exporting Stream Functions

Making the stream functions in the driver DLL accessible to external applications requires the linker to export the functions during the build process. C++ provides several options to accomplish this, yet for driver DLLs compatible with Device

Manager, you must export the functions by defining them in the .def file of the DLL subproject. The linker uses the .def file to determine which functions to export and how to do so. For a standard stream driver, you must export the stream interface functions using the prefix that you specify in the driver's source code and registry settings. Figure 6–3 shows a sample .def file for the stream interface skeleton listed in the previous section.

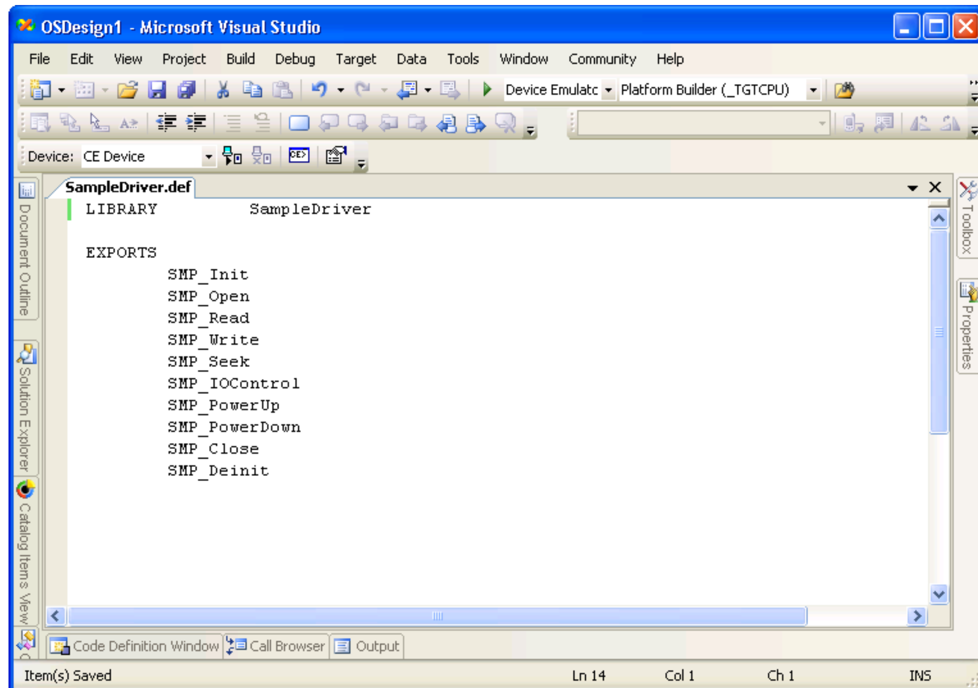


Figure 6-3 A sample .def file for a stream driver

Sources File

Prior to building the newly created stream driver, you should also check the Sources file in the root folder of the DLL subproject to ensure that it includes all necessary files in the build process. As mentioned in Chapter 2, the Sources file configures the compiler and linker to build the desired binary files. Table 6–2 summarizes the most important Sources file directives for device drivers.

Table 6-2 Important Sources file directives for device drivers

Directive	Description
WINCEOEM=1	Causes additional header files and import libraries from the %_WINCEROOT%\Public tree to be included to enable the driver to make platform-dependent function calls, such as KernelIoControl, InterruptInitialize, and InterruptDone.
TARGETTYPE=DYNLINK	Instructs the Build tool to create a DLL.
DEFFILE=<*Driver Def File Name*>.def	References the module-definition file that defines the exported DLL functions.
DLLENTRY=<*DLL Main Entry Point*>	Specifies the function that is called when processes and threads attach and detach to and from the driver DLL (Process Attach, Process Detach, Thread Attach, and Thread Detach).

Opening and Closing a Stream Driver by Using the File API

To access a stream driver, an application can use the CreateFile function and specify the desired device name. The following example illustrates how to open a driver called SMP1: for reading and writing. It is important to note, however, that Device Manager must already have loaded the driver, such as during the boot process. Lesson 3 later in this chapter provides detailed information about configuring and loading device drivers.

```
// 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);
```

Dynamically Loading a Driver

As mentioned earlier in this lesson, an application can also communicate with a stream device driver after calling the ActivateDevice or ActivateDeviceEx function. ActivateDeviceEx offers more flexibility than ActivateDevice, yet both functions cause Device Manager to load the stream driver and call the driver's XXX_Init function. In fact, ActivateDevice calls ActivateDeviceEx. Note, however, that ActivateDeviceEx does not provide access to an already loaded driver. The primary purpose of the ActivateDeviceEx function is to read a driver-specific registry key specified in the function call to determine the DLL name, device prefix, index, and other values, add the relevant values to the active device list, and then load the device driver into the Device Manager process space. The function call returns a handle that the application can later use to unload the driver in a call to the DeactivateDevice function.

ActivateDeviceEx replaces the older RegisterDevice function as a method to load a driver on demand, as illustrated in the following code sample:

```
// 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 ")));
    }
}
```

NOTE Automatic vs. dynamic loading of drivers

Calling ActivateDeviceEx to load a driver has the same result as loading the driver automatically during the boot process through parameters defined in the HKEY_LOCAL_MACHINE\Drivers\BuiltIn key. The BuiltIn registry key is covered in more detail in Lesson 3 later in this chapter.

Lesson Summary

Stream drivers are Windows Embedded CE drivers that implement the stream interface API. The stream interface enables Device Manager to load and manage these drivers, and applications can use standard file system functions to access these drivers and perform I/O operations. To present a stream driver as a file resource accessible through a CreateFile call, the name of the stream driver must follow a special naming convention that distinguishes the device resource from ordinary files. Legacy names (such as COM1:) have a limitation of ten instances per driver because they include only a single-digit instance identification. If you must support more than ten driver instances on a target device, use a device name instead (such as \$device\COM1).

Because Device Manager can load a single driver multiple times to satisfy the requests from different processes and threads, stream drivers must implement context management. Windows Embedded CE knows two context levels for device drivers, device context and open context, which the operating system passes in each

appropriate function call to the driver so that the driver can associate internal resources and allocated memory regions with each caller.

The stream interface consists of 12 functions: XXX_Init, XXX_Open, XXX_Read, XXX_Write, XXX_Seek, XXX_IOControl, XXX_PowerUp, XXX_PowerDown, XXX_PreClose, XXX_Close, XXX_PreDeinit, and XXX_Deinit. Not all functions are mandatory (such as XXX_PreClose and XXX_PreDeinit), yet any functions that the stream device driver implements must be exposed from the driver DLL to Device Manager. To export these functions, you must define them in the .def file of the DLL subproject. You should also adjust the DLL subproject's Sources file to ensure that the driver DLL can make platform-dependent function calls.

Lesson 3: Configuring and Loading a Driver

In general, you have two options to load a stream driver under Windows Embedded CE 6.0. You can instruct Device Manager to load the driver automatically during the boot sequence by configuring driver settings in the HKEY_LOCAL_MACHINE \Drivers\BuiltIn registry key, or you can load the driver dynamically through a direct call to ActivateDeviceEx. Either way, Device Manager can load the device driver with the same registry flags and settings. The key difference is that you receive a handle to the driver when using ActivateDeviceEx, which you can use later in a call to DeactivateDevice. Especially during the development stage, it might be advantageous to load a driver dynamically through ActivateDeviceEx so that you can unload the driver, install an updated version, and then reload the driver without having to restart the operating system. You can also use DeactivateDevice to unload drivers loaded automatically based on entries under the BuiltIn registry key, but you cannot reload them without calling ActivateDeviceEx directly.

After this lesson, you will be able to:

- Identify the mandatory registry settings for a device driver.
- Access the registry settings from within a driver.
- Load a driver at startup or on demand in an application.
- Load a driver in user space or kernel space.

Estimated lesson time: 25minutes.

Device Driver Load Procedure

Whether you load a device driver statically or dynamically, the ActivateDeviceEx function is always involved. A dedicated driver named the Bus Enumerator (BusEnum) calls ActivateDeviceEx for every driver registered under HKEY_LOCAL_MACHINE\Drivers\BuiltIn just as you can call ActivateDeviceEx directly, passing in an alternate registry path for the driver settings in the lpszDevKey parameter.

Device Manager uses the following procedure to load device drivers at boot time:

1. Device Manager reads the HKEY_LOCAL_MACHINE\Drivers\RootKey entry to determine the location of the device driver entries in the registry. The default value of the RootKey entry is Drivers\BuiltIn.

2. Device Manager reads the Dll registry value at the location specified in the RootKey location (HKEY_LOCAL_MACHINE\Drivers\BuiltIn) to determine the enumerator DLL to load. By default, this is the bus enumerator (BusEnum.dll). The bus enumerator is a stream driver that exports the Init and Deinit functions.
3. The bus enumerator runs at startup to scan the RootKey registry location for subkeys that refer to additional buses and devices. It can be run again later with a different RootKey to load more drivers. The bus enumerator examines the Order value in each subkey to determine the load order.
4. Starting with the lowest Order values, the bus enumerator iterates through the subkeys and calls ActivateDeviceEx passing in the current driver's registry path (that is, HKEY_LOCAL_MACHINE\Drivers\BuiltIn\<*DriverName*>).
5. ActivateDeviceEx loads the driver DLL registered in the DLL value located in the driver's subkey, and then creates a subkey for the driver under the HKEY_LOCAL_MACHINE\Drivers\Active registry key to keep track of all currently loaded drivers.

Figure 6-4 shows a typical registration under the HKEY_LOCAL_MACHINE \Drivers\BuiltIn registry key for an audio device driver.

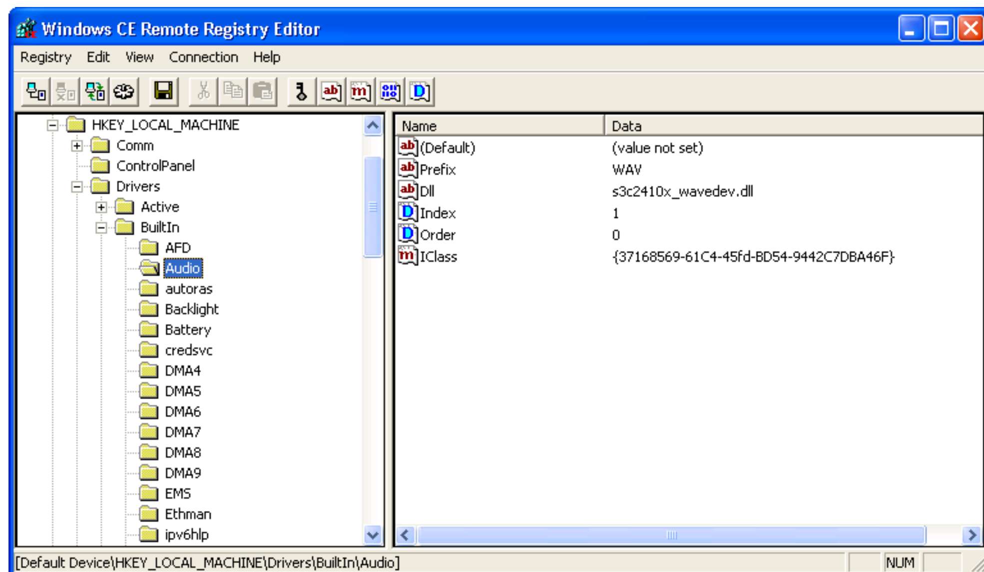


Figure 6-4 An audio device driver registration

Registry Settings to Load Device Drivers

If you use ActivateDeviceEx to load your driver dynamically, you are not required to place the driver's registry settings in a subkey under HKEY_LOCAL_MACHINE \Drivers\BuiltIn. You can use an arbitrary path, such as HKEY_LOCAL_MACHINE \SampleDriver. However, the registry values for the driver are the same in both cases. Table 6–3 lists general registry entries that you can specify for a device driver in the driver's registry subkey (see Figure 6–4 for sample values).

Table 6-3 General registry entries for device drivers

Registry Entry	Type	Description
Prefix	REG_SZ	A string value that contains the driver's three-letter name. This is the value that replaces *XXX* in the stream interface functions. Applications also use this prefix to open a context of the driver through CreateFile.
Dll	REG_SZ	This is the name of the DLL that Device Manager loads to load the driver. Note that this is the only mandatory registry entry for a driver.
Index	REG_DWORD	This is the number appended to the driver prefix to create the driver's file name. For example, if this value is 1, applications can access this driver through a call to CreateFile(L"XXX1:"...) or CreateFile(L"\$device\XXX1"...). Note that this value is optional. If you do not define it, Device Manager assigns the next available index value to the driver.

Table 6-3 General registry entries for device drivers (Continued)

Registry Entry	Type	Description
Order	REG_DWORD	This is the order in which Device Manger loads the driver. If this value is not specified, the driver will be loaded last, at the same time as other drivers with no order specified. Drivers with the same Order value start concurrently. You should only configure this value to enforce a sequential load order. For example, a Global Positioning System (GPS) driver might require a Universal Asynchronous Receiver/Transmitter (UART) driver to get access to the GPS data through a serial port. In this case, it is important to assign the UART driver a lower Order value than the GPS driver so that the UART driver starts first. This will enable the GPS driver to access the UART driver during its initialization.
IClass	REG_MULTI_SZ	This value can specify predefined device interface globally unique identifiers (GUIDs). To advertise an interface to Device Manager, such as to support the Plug and Play notification system and power management capabilities, add the corresponding interface GUIDs to the IClass value or call AdvertiseInterface in the driver.

Table 6-3 General registry entries for device drivers (Continued)

Registry Entry	Type	Description
Flags	REG_DWORD	This value can contain the following flags: ■ **DEVFLAGS_UNLOAD (0x0000 0001)** The driver unloads after a call to XXX_Init. ■ **DEVFLAGS_NOLOAD (0x0000 0004)** The driver cannot be loaded. ■ **DEVFLAGS_NAKEDENTRIES (0x0000 0008)** The entry points of the driver are Init, Open, IOControl, and so forth, without any prefixes. ■ **DEVFLAGS_BOOTPHASE_1 (0x0000 1000)** The driver is loaded during system phase 1 for systems with multiple boot phases. This prevents the driver from being loaded more than one time during the boot process. ■ **DEVFLAGS_IRQ_EXCLUSIVE (0x0000 0100)** The bus driver loads this driver only when it has exclusive access to the interrupt request (IRQ) specified by the IRQ value. ■ **DEVFLAGS_LOAD_AS_USERPROC (0x0000 0010)** Loads the driver in user mode.

Table 6-3 General registry entries for device drivers (Continued)

Registry Entry	Type	Description
UserProcGroup	REG_DWORD	Associates a driver marked with the DEVFLAGS_LOAD_AS_USERPROC (0x0000 0010) flag to load in user mode with a user-mode driver host process group. User-mode drivers that belong to the same group are loaded by Device Manager in the same host process instance. If this registry entry does not exist, Device Manager loads the user-mode driver into a new host process instance.

NOTE Flags

For details about the Flags registry value, see the section "ActivateDeviceEx" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa929596.aspx*.

Registry Keys Related to Loaded Device Drivers

Apart from configurable registry entries in driver-specific subkeys, Device Manager also maintains dynamic registry information in subkeys for loaded drivers under the HKEY_LOCAL_MACHINE\Drivers\Active key. The subkeys correspond to numerical values that the operating system assigns dynamically and increments for each driver until the system is restarted. The number does not signify a particular driver. For example, if you unload and reload a device driver, the operating system assigns the next number to the driver and does not reuse the previous subkey. Because you cannot ensure a reliable association between the subkey number and a particular device driver, you should not edit the driver entries in the HKEY_LOCAL_MACHINE\Drivers\Active key manually. However, you can create, read, and write driver-specific registry keys at load time in a driver's XXX_Init function because Device Manager passes the path to the current Drivers\Active subkey to the stream driver as the first parameter. The driver can open this registry key using OpenDeviceKey.

Table 6–4 lists typical entries that the subkeys under Drivers\Active can contain.

Table 6-4 Registry entries for device drivers under the HKEY_LOCAL_MACHINE\Drivers\Active key

Registry Entry	Type	Description
Hnd	REG_DWORD	The handle value for the loaded device driver. You can obtain this DWORD value from the registry and pass it in a call to DeactivateDevice in order to unload the driver.
BusDriver	REG_SZ	The name of the driver's bus.
BusName	REG_SZ	The name of the device's bus.
DevID		A unique device identifier from Device Manager.
FullName	REG_SZ	The name of the device if used with the $device namespace.
Name	REG_SZ	The driver's legacy device file name including index, if a prefix is specified (not present for drivers that do not specify a prefix).
Order	REG_DWORD	The same order value as in the driver's registry key.
Key	REG_SZ	The registry path to the driver's registry key.
PnpId	REG_SZ	The Plug and Play identifier string for PCMCIA drivers.
Sckt	REG_DWORD	For PCMCIA drivers, describes the current socket and function of the PC card.

NOTE Checking the Active key

By calling the RequestDeviceNotifications function with a device interface GUID of DEVCLASS_STREAM_GUID, an application can receive messages from Device Manager to identify loaded stream drivers programmatically. RequestDeviceNotifications supersedes the EnumDevices function.

Kernel-Mode and User-Mode Drivers

Drivers can either run in the kernel memory space or in user memory space. In kernel mode, drivers have full access to the hardware and kernel memory, although function calls are generally limited to kernel APIs. Windows Embedded CE 6.0 runs drivers in kernel mode, by default. On the other hand, drivers in user mode do not have direct access to kernel memory. There are some performance penalties when running in user mode, yet the advantage is that a driver failure in user mode only affects the current process, whereas the failure of a kernel-mode driver can impair the entire operating system. The system can generally recover more gracefully from the failure of a user-mode driver.

NOTE Kernel driver restrictions

Kernel drivers cannot display a user interface directly in CE 6.0 R2. To use any user interface elements, developers must create a companion DLL that will be loaded into user-mode, then call into this DLL with CeCallUserProc. For more information on CeCallUserProc, see the MSDN Web page at *http://msdn2.microsoft.com/en-us/library/aa915093.aspx*.

User-Mode Drivers and the Reflector Service

In order to communicate with the underlying hardware and perform useful tasks, user-mode drivers must be able to access system memory and privileged APIs unavailable to standard user-mode processes. To facilitate this, Windows Embedded CE 6.0 features a Reflector service, which runs in kernel mode, performs buffer marshaling, and calls privileged memory management APIs on behalf of the user-mode drivers. The Reflector service is transparent so that user-mode drivers can work almost the same way as kernel-mode drivers without modifications. An exception to this rule is a driver that uses kernel APIs that are not available in user mode. It is not possible to run these types of kernel-mode drivers in user mode.

When an application calls ActivateDeviceEx, Device Manager loads the driver either directly in kernel space or passes the request to the Reflector service, which in turn starts a user mode driver host process (Udevice.exe) through a CreateProcess call. The Flags registry entry in the driver's registry key determines whether a driver should run in user mode (DEVFLAGS_LOAD_AS_USERPROC flag). Having started the required instance of Udevice.exe and user-mode driver, the Reflector service forwards the XXX_Init call from Device Manager to the user-mode driver and returns the return code from the user-mode driver back to Device Manager, as indicated in Figure 6–5. The same proxy principle also applies to all other stream functions.

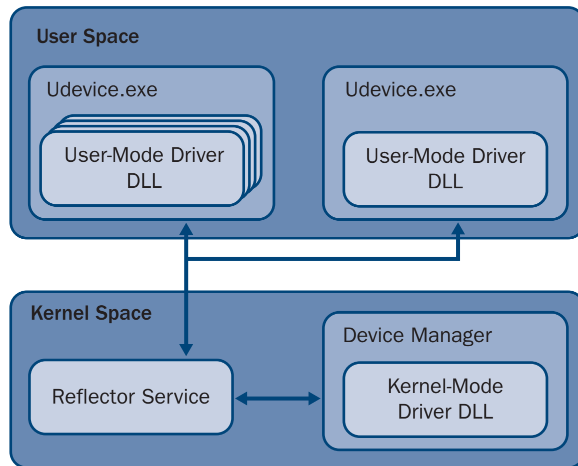


Figure 6-5 User-mode drivers, kernel-mode drivers, and the Reflector service

User-Mode Drivers Registry Settings

On Windows Embedded CE 6.0, you can run multiple user-mode drivers in a single host process or have multiple host processes enabled on the system. Drivers grouped in a single Udevice.exe instance share the same process space, which is particularly useful for drivers that depend on each other. However, drivers in the same process space can affect each other's stability. For example, if a user-mode driver causes the host process to fail, all drivers in that host process fail. The system continues to function except for the affected drivers and applications accessing these drivers, yet it is possible to recover from this situation by reloading the drivers, if the applications support it. If you isolate a critical driver in a separate user mode driver host process, you can increase the overall system stability. By using the registry entries listed in Table 6–5, you can define individual host process groups.

Table 6-5 Registry entries for user-mode driver host processes

Registry Entry	Type	Description
HKEY_LOCAL_MACHINE \ Drivers\ProcGroup_###	REG_KEY	Defines a three-digit group ID (###) for a user-mode driver host process, such as ProcGroup_003, which you can then specify in the UserProcGroup entry in a driver's registry key, such as UserProcGroup =3.

Table 6-5 Registry entries for user-mode driver host processes (Continued)

Registry Entry	Type	Description
ProcName	REG_SZ	The process that the Reflector service starts to host the user-mode driver, such as ProcName= Udevice.exe.
ProcVolPrefix	REG_SZ	Specifies the file system volume that the Reflector service mounts for the user-mode driver host process, such as ProcVolPrefix = $udevice. The specified ProcVolPrefix replaces the $device volume in driver device names.

Having defined the desired host process groups, you can associate each user-mode driver with a particular group by adding the UserProcGroup registry entry to the device driver's registry subkey (see Table 6–3 earlier in this lesson). By default, the UserProcGroup registry entry does not exist, which corresponds to a configuration in which Device Manager loads every user-mode driver into a separate host process instance.

Binary Image Builder Configuration

As explained in Chapter 2, "Building and Deploying a Run-Time Image," the Windows Embedded CE build process relies on binary image builder (.bib) files to generate the content of the run-time image and to define the final memory layout of the device. Among other things, you can specify a combination of flags for a driver's module definition. Issues can arise if .bib file settings and registry entries do not match for a device driver. For example, if you specify the K flag for a device driver module in a .bib file and also set the DEVFLAGS_LOAD_AS_USERPROC flag in the driver's registry subkey to load the driver into the user-mode driver host process, the driver fails to load because the K flag instructs Romimage.exe to load the module in kernel space above the memory address 0x80000000. To load a driver in user mode, be sure to load the module into user space below 0x80000000, such as into the NK memory region defined in the Config.bib file for the BSP.

The following .bib file entry demonstrates how to load a user-mode driver into the NK memory region:

```
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SHQ
```

The S and H flags indicate that Driver.dll is both a system file and a hidden file, located in the flat release directory. The Q flag specifies that the system can load this module concurrently into both kernel and user space. It adds two copies of the DLL to the run-time image, one with and one without the K flag, and doubles in this way ROM and RAM space requirements for the driver. Use the Q flag sparingly.

Extending the above example, the Q flag is equivalent to the following:

```
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SH
driver.dll      $(_FLATRELEASEDIR)\driver.dll    NK    SHK
```

Lesson Summary

Windows Embedded CE can load drivers into kernel space or user space. Drivers running in kernel space have access to system APIs and kernel memory and can affect the stability of the system if failures occur. However, properly implemented kernel-mode drivers exhibit better performance than user-mode drivers, due to reduced context switching between kernel and user mode. On the other hand, the advantage of user-mode drivers is that failures primarily affect the current user-mode process. User-mode drivers are also less privileged, which can be an important aspect in respect to non-trusted drivers from third-party vendors.

To integrate a driver running in user mode with Device Manager running in kernel mode, Device Manager uses a Reflector service that loads the driver in a user-mode driver host process and forwards the stream function calls and return values between the driver and Device Manager. In this way, applications can continue to use familiar file system APIs to access the driver, and the driver does not need code changes regarding the stream interface API to remain compatible with Device Manager. By default, user-mode drivers run in separate host processes, but you can also configure host process groups and associate drivers with these groups by adding a corresponding UserProcGroup registry entry to a driver's registry subkey. Driver subkeys can reside in any registry location, yet if you want to load the drivers at boot time automatically, you must place the subkeys into Device Manager's RootKey, which by default is HKEY_LOCAL_MACHINE\Drivers\BuiltIn. Drivers that have their subkeys in different locations can be loaded on demand by calling the ActivateDeviceEx function.

Lesson 4: Implementing an Interrupt Mechanism in a Device Driver

Interrupts are notifications generated either in hardware or software to inform the CPU that an event has occurred that requires immediate attention, such as timer events or keyboard events. In response to an interrupt, the CPU stops executing the current thread, jumps to a trap handler in the kernel to respond to the event, and then resumes executing the original thread after the interrupt is handled. In this way, integrated and peripheral hardware components, such as system clock, serial ports, network adapters, keyboards, mouse, touchscreen, and other devices, can get the attention of the CPU and have the kernel exception handler run appropriate code in interrupt service routines (ISRs) within the kernel or in associated device drivers. To implement interrupt processing in a device driver efficiently, you must have a detailed understanding of Windows Embedded CE 6.0 interrupt handling mechanisms, including the registration of ISRs in the kernel and the execution of interrupt service threads (ISTs) within the Device Manager process.

After this lesson, you will be able to:

- Implement an interrupt handler in the OEM adaptation layer (OAL).
- Register and handle interrupts in a device driver interrupt service thread (IST).

Estimated lesson time: 40 minutes.

Interrupt Handling Architecture

Windows Embedded CE 6.0 is a portable operating system that supports different CPU types with varying interrupt schemes by implementing a flexible interrupt handling architecture. Most importantly, the interrupt handling architecture takes advantage of interrupt-synchronization capabilities in the OAL and thread-synchronization capabilities of Windows Embedded CE to split the interrupt processing into ISRs and ISTs, as illustrated in Figure 6–6.

Windows Embedded CE 6.0 interrupt handling is based on the following concepts:

1. During the boot process, the kernel calls the OEMInit function in the OAL to register all available ISRs built into the kernel with their corresponding hardware interrupts based on their interrupt request (IRQ) values. IRQ values are numbers that identify the source of the interrupt in the processor interrupt controller registers.

2. Device drivers can dynamically install ISRs implemented in ISR DLLs by calling the LoadIntChainHandler function. LoadIntChainHandler loads the ISR DLL into kernel memory space and registers the specified ISR routine with the specified IRQ value in the kernel's interrupt dispatch table.
3. An interrupt occurs to notify the CPU that an event requires suspending the current thread of execution and transferring control to a different routine.
4. In response to the interrupt, the CPU stops executing the current thread and jumps to the kernel exception handler as the primary target of all interrupts.
5. The exception handler masks off all interrupts of an equal or lower priority and then calls the appropriate ISR registered to handle the current interrupt. Most hardware platforms use interrupt masks and interrupt priorities to implement hardware–based interrupt synchronization mechanisms.
6. The ISR performs any necessary tasks, such as masking the current interrupt so that the current hardware device cannot trigger further interrupts, which would interfere with the current processing, and then returns a SYSINTR value to the exception handler. The SYSINTR value is a logical interrupt identifier.
7. The exception handler passes the SYSINTR value to the kernel's interrupt support handler, which determines the event for the SYSINTR value, and, if found, signals that event for any waiting ISTs for the interrupt.
8. The interrupt support handler unmasks all interrupts, with the exception of the interrupt currently in processing. Keeping the current interrupt masked off explicitly prevents the current hardware device from triggering another interrupt while the IST runs.
9. The IST runs in response to the signaled event to perform and finish the interrupt handling without blocking other devices on the system.
10. The IST calls the InterruptDone function to inform the kernel's interrupt support handler that the IST has finished its processing and is ready for another interrupt event.
11. The interrupt support handler calls the OEMInterruptDone function in the OAL to complete the interrupt handling process and reenable the interrupt.

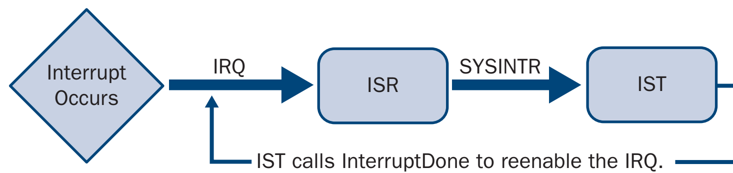


Figure 6-6 IRQs, ISRs, SYSINTRs, and ISTs

Interrupt Service Routines

In general, ISRs are small blocks of code that run in response to a hardware interrupt. Because the kernel exception handler masks off all interrupts of equal or lesser priority while this ISR runs, it is important to complete the ISR and return a SYSINTR value as quickly as possible so that the kernel can re-enable (unmask) all IRQs with minimal delay (except the currently processed interrupt). System performance can suffer significantly if too much time is spent in ISRs, leading to missed interrupts or overrun buffers on some devices. Another important aspect is that the ISR runs in kernel mode and does not have access to higher-level operating system APIs. For these reasons, ISRs usually perform no more than the most basic tasks, such as quickly copying data from hardware registers to memory buffers. On Windows Embedded CE, time-consuming interrupt processing is usually performed in an IST.

The primary task of the ISR is to determine the interrupt source, mask off or clear the interrupt at the device, and then return a SYSINTR value for the interrupt to notify the kernel about an IST to run. In the simplest case, the ISR returns SYSINTR_NOP to indicate that no further processing is necessary. Accordingly, the kernel does not signal an event for an IST to handle the interrupt. On the other hand, if the device driver uses an IST to handle the interrupt, the ISR passes the logical interrupt identifier to the kernel, the kernel determines and signals the interrupt event, and the IST typically resumes from a WaitForSingleObject call and executes the interrupt processing instructions in a loop. The latency between the ISR and the IST depends on the priority of the thread and other threads running in the system, as explained in Chapter 3, "Performing System Programming." Typically, ISTs run with a high thread priority.

Interrupt Service Threads

An IST is a regular thread that performs additional processing in response to an interrupt, after the ISR has completed. The IST function typically includes a loop and a WaitForSingleObject call to block the thread infinitely until the kernel signals the specified IST event, as illustrated in the following code snippet. However, before you

can use the IST event, you must call InterruptInitialize with the SYSINTR value and an event handle as parameters so that the CE kernel can signal this event whenever an ISR returns the SYSINTR value. Chapter 3 provides detailed information about multi-threaded programming and thread synchronization based on events and other kernel objects.

```
 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);
}
```

When the IST has completed processing an IRQ, it should call InterruptDone to inform the system that the interrupt was processed, that the IST is ready to handle the next IRQ, and that the interrupt can be reenabled by means of an OEMInterruptDone call. Table 6–6 lists the OAL functions that the system uses to interact with the interrupt controller to manage interrupts.

Table 6-6 OAL functions for interrupt management

Function	Description
OEMInterruptEnable	This function is called by the kernel in response to InterruptInitialize and enables the specified interrupt in the interrupt controller.
OEMInterruptDone	This function is called by the kernel in response to InterruptDone and should unmask the interrupt and acknowledge the interrupt in the interrupt controller.
OEMInterruptDisable	This function disables the interrupt in the interrupt controller and is called in response to the InterruptDisable function.
OEMInterruptHandler	For ARM processors only, this function identifies the interrupt SYSINTR that occurs by looking at the status of the interrupt controller.

Table 6-6 OAL functions for interrupt management (Continued)

Function	Description
HookInterrupt	For processors other than ARM, this function registers a callback function for a specified interrupt ID. This function must be called in the OEMInit function to register mandatory interrupts.
OEMInterruptHandlerFIQ	For ARM processors, used to handle interrupts for the Fast Interrupt (FIQ) line.

CAUTION WaitForMultipleObjects restriction

Do not use the WaitForMultipleObjects function to wait for an interrupt event. If you must wait for multiple interrupt events, you should create an IST for each interrupt.

Interrupt Identifiers (IRQ and SYSINTR)

Each hardware interrupt line corresponds to an IRQ value in the interrupt controller registers. Each IRQ value can be associated with only one ISR, but an ISR can map to multiple IRQs. The kernel does not need to maintain the IRQs. It just determines and signals events associated with the SYSINTR values returned from the ISR in response to the IRQ. The ability to return varying SYSINTR values from an ISR provides the basis to support multiple devices that use the same shared interrupt.

NOTE OEMInterruptHandler and HookInterrupt

Target devices that only support a single IRQ, such as ARM–based systems, use the OEMInterruptHandler function as the ISR to identify the embedded peripheral that triggered the interrupt. Original equipment manufacturers (OEMs) must implement this function as part of the OAL. On platforms that support multiple IRQs, such as Intel x86–based systems, you can associate the IRQs with individual ISRs by calling HookInterrupt.

Static Interrupt Mappings

For the ISR to determine a correct SYSINTR return value there must be a mapping between the IRQ and the SYSINTR, which can be hardcoded into the OAL. The Bsp_cfg.h file for the Device Emulator BSP demonstrates how to define a SYSINTR value in the OAL for a target device relative to the SYSINTR_FIRMWARE value. If you want to define additional identifiers in your OAL for a custom target device, keep in

mind that the kernel reserves all values below SYSINTR_FIRMWARE for future use and the maximum value should be less than SYSINTR_MAXIMUM.

To add a mapping of static SYSINTR values to IRQs on a target device, you can call the OALIntrStaticTranslate function during system initialization. For example, the Device Emulator BSP calls OALIntrStaticTranslate in the BSPIntrInit function to register a custom SYSINTR value for the built-in Open Host Controller Interface (OHCI) in the kernel's interrupt mapping arrays (g_oalSysIntr2Irq and g_oalIrq2SysIntr). However, static SYSINTR values and mappings are not a common way to associate IRQs with SYSINTRs because it is difficult and requires OAL code changes to implement custom interrupt handling. Static SYSINTR values are typically used for core hardware components of a target device where there is no explicit device driver and the ISR resides in the OAL.

Dynamic Interrupt Mappings

The good news is that you do not need to hardcode SYSINTR values into the OAL if you call KernelIoControl in your device drivers with an IO control code of IOCTL_HAL_REQUEST_SYSINTR to register IRQ/SYSINTR mappings. The call eventually ends in the OALIntrRequestSysIntr function, which dynamically allocates a new SYSINTR for the given IRQ, and then registers the IRQ and SYSINTR mappings in the kernel's interrupt mapping arrays. Locating a free SYSINTR value up to SYSINTR_MAXUMUM is more flexible than static SYSINTR assignments because this mechanism does not require any modifications to the OAL when you add new drivers to the BSP.

When calling KernelIoControl with IOCTL_HAL_REQUEST_SYSINTR, you establish a 1:1 relationship between IRQ and SYSINTR. If the IRQ-SYSINTR mapping table already has an entry for the specified IRQ, OALIntrRequestSysIntr will not create a second entry. To remove an entry from the interrupt mapping tables, such as when unloading a driver, call KernelIoControl with an IO control code of IOCTL_HAL_REQUEST_SYSINTR. IOCTL_HAL_RELEASE_SYSINTR dissociates the IRQ from the SYSINTR value.

The following code sample illustrates the use of IOCTL_HAL_REQUEST_SYSINTR and IOCTL_HAL_RELEASE_SYSINTR. It takes a custom value (dwLogintr) and passes this value to the OAL to be translated into a SYSINTR value, and then associates this SYSINTR with an IST event.

```
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);
```

Shared Interrupt Mappings

The 1:1 relationship between IRQ and SYSINTR implies that you cannot register multiple ISRs for an IRQ directly to implement interrupt sharing, but you can map multiple ISRs indirectly. The interrupt mapping tables only map an IRQ to one static ISR, yet within this ISR, you can call the NKCallIntChain function to iterate through the chain of ISRs, registered dynamically through LoadIntChainHandler. NKCallIntChain goes through the ISRs registered for the shared interrupt and returns the first SYSINTR value that is not equal to SYSINTR_CHAIN. Having determined the appropriate SYSINTR for the current interrupt source, the static ISR can pass this logical interrupt identifier to the kernel to signal the corresponding IST event. The LoadIntChainHandler function and installable ISRs are covered in more detail later in this lesson.

Communication between an ISR and an IST

Because ISR and IST run at different times and in different contexts, you must take extra care of physical and virtual memory mappings if an ISR must pass data to an IST. For example, an ISR might copy individual bytes from a peripheral device into an input buffer, returning SYSINTR_NOP until the buffer is full. The ISR returns the actual SYSINTR value only when the input buffer is ready for the IST. The kernel signals the corresponding IST event and the IST runs to copy the data into a process buffer.

One way to accomplish this data transfer is to reserve a physical memory section in a .bib file. Config.bib contains several examples for the serial and debug drivers. The ISR can then call the OALPAtoVA function to translate the physical address of the reserved memory section into a virtual address. Because the ISR runs in kernel mode, the ISR can access the reserved memory to buffer data from the peripheral device. The IST, on the other hand, calls MmMapIoSpace outside the kernel to map the physical memory to a process-specific virtual address. MmMapIoSpace uses VirtualAlloc and VirtualCopy to map the physical memory into virtual memory, yet you can also call VirtualAlloc and VirtualCopy directly if you need more control over the address mapping process.

Another option to pass data from an ISR to an IST is to allocate physical memory in SDRAM dynamically by using the AllocPhysMem function in the device driver, which is particularly useful for installable ISRs loaded into the kernel on an as-needed basis. AllocPhysMem allocates a physically contiguous memory area and returns the physical address (or fails if the allocation size is not available). The device driver can

communicate the physical address to the ISR in a call to KernelIoControl based on a user-defined IO control code. The ISR then uses the OALPAtoVA function to translate the physical address into a virtual address. The IST uses MmMapIoSpace or the VirtualAlloc and VirtualCopy functions, as already explained for statically reserved memory regions.

Installable ISRs

When adapting Windows Embedded CE to a custom target device, it is important to keep the OAL as generic as possible. Otherwise, you have to modify the code every time you add a new component to the system. To provide flexibility and adaptability, Windows Embedded CE supports installable ISRs (IISR), which a device driver can load into kernel space on demand, such as when new peripheral devices are connected in a Plug and Play fashion. Installable ISRs also provide a solution to process interrupts when multiple hardware devices share the same interrupt line. The ISR architecture relies on lean DLLs that contain the code of the installable ISR and export the entry points summarized in Table 6–7.

Table 6-7 Exported installable ISR DLL functions

Function	Description
ISRHandler	This function contains the installable interrupt handler. The return value is the SYSINTR value for the IST that you want to run in response to the IRQ registered for the installable ISR in the call to the LoadIntChainHandler function. The OAL must support chaining on at least that IRQ, which means that an unhandled interrupt can be chained to another handler (which is the installed ISR in this case) when an interrupt occurs.
CreateInstance	This function is called when an installable ISR is loaded by using the LoadIntChainHandler function. It returns an instance identifier for the ISR.
DestroyInstance	This function is called when an installable ISR is unloaded by using the FreeIntChainHandler function.
IOControl	This function supports communication from the IST to the ISR.

NOTE Generic installable ISR (GIISR)

To help you implement installable ISRs, Microsoft provides a generic installable ISR sample that covers the most typical needs for many devices. You can find the source code in the following folder: %_WINCEROOT%\Public\Common\Oak\Drivers\Giisr.

Registering an IISR

The LoadIntChainHandler function expects three parameters that you must specify to load and register an installable ISR. The first parameter (lpFilename) specifies the filename of the ISR DLL to load. The second parameter (lpszFunctionName) identifies the name of the interrupt handler function, and the third parameter (bIRQ) defines the IRQ number for which you want to register the installable ISR. In response to hardware disconnect, a device driver can also unload an installable ISR by calling the FreeIntChainHandler function.

External Dependencies and Installable ISRs

It is important to keep in mind that LoadIntChainHandler loads ISR DLLs into kernel space, which means that the installable ISR cannot call high-level operating system APIs and cannot import or implicitly link to other DLLs. If the DLL has explicit or implicit links to other DLLs, or if it uses the C run-time library, the DLL will not be able to load. The installable ISR must be completely self-sufficient.

To ensure that an installable ISR does not link to the C run-time libraries or any DLLs, you must add the following lines to the Sources file in your DLL subproject:

```
NOMIPS16CODE=1
NOLIBC=1
```

The NOLIBC=1 directive ensures that the C run-time libraries are not linked and the NOMIPS16CODE=1 option enables the compiler option **/QRimplicit-import**, which prevents implicit links to other DLLs. Note that this directive has absolutely no relationship to the Microprocessor without Interlocked Pipeline Stages (MIPS) CPU.

Lesson Summary

Windows Embedded CE relies on ISRs and ISTs to respond to interrupt requests triggered by internal and external hardware components that require the attention of the CPU outside the normal code execution path. ISRs are typically compiled directly into the kernel or implemented in device drivers loaded at boot time and registered with corresponding IRQs through HookInterrupt calls. You can also implement installable ISRs in ISR DLLs, which device drivers can load on demand and associate with an IRQ by calling LoadIntChainHandler. Installable ISRs also enable you to support interrupt sharing. For example, on a system with only a single IRQ, such as an ARM–based device, you can modify the OEMInterruptHandler function, which is a static ISR that loads further installable ISRs depending on the hardware component that triggered the interrupt.

Apart from the fact that ISR DLLs must not have any dependencies on external code, ISRs and installable ISRs have many similarities. The primary task of an interrupt handler is to determine the interrupt source, mask off or clear the interrupt at the device, and then return a SYSINTR value for the IRQ to notify the kernel about an IST to run. Windows Embedded CE maintains interrupt mapping tables that associate IRQs with SYSINTR values. You can define static SYSINTR values in the source code or use dynamic SYSINTR values that you can request from the kernel at run time. By using dynamic SYSINTR values, you can increase the portability of your solutions.

According to the SYSINTR value, the kernel can signal an IST event which enables the corresponding interrupt service thread to resume from a WaitForSingleObject call. By performing most of the work to handle the IRQ in the IST instead of the ISR, you can achieve optimal system performance because the system blocks interrupt sources with lower or equal priority only during ISR execution. The kernel unmasks all interrupts when the ISR finishes, with the exception of the interrupt currently in processing. The current interrupt source remains blocked so that a new interrupt from the same device cannot interfere with the current interrupt handling procedure. When the IST has finished its work, the IST must call InterruptDone to inform the kernel that it is ready for a new interrupt so that the kernel's interrupt support handler can reenable the IRQ in the interrupt controller.

Lesson 5: Implementing Power Management for a Device Driver

As mentioned in Chapter 3, power management is important for Windows Embedded CE devices. The operating system includes a Power Manager (PM.dll), which is a kernel component that integrates with Device Manager to enable devices to manage their own power states and applications to set power requirements for certain devices. The primary purpose of Power Manager is to optimize power consumption and to provide an API to system components, drivers, and applications for power notifications and control. Although Power Manager does not impose strict requirements on power consumption or capabilities in any particular power state, it is beneficial to add power management features to a device driver so that you can manage the state of your hardware components in a way that is consistent with the power state of the target device. For more information about Power Manager, device and system power states, and power management features supported in Windows Embedded CE 6.0, read Chapter 3, "Performing System Programming."

After this lesson, you will be able to:

- Identify the power management interface for device drivers.
- Implement power management in a device driver.

Estimated lesson time: 30 minutes.

Power Manager Device Drivers Interface

Power Manager interacts with power management–enabled drivers through the XXX_PowerUp, XXX_PowerDown, and XXX_IOControl functions. For example, the device driver itself can request a change of the device power level from Power Manager by calling the DevicePowerNotify function. In response, Power Manager calls XXX_IOControl with an IO control code of IOCTL_POWER_SET passing in the requested device power state. It might seem overcomplicated for a device driver to change the power state of its own device through Power Manager, yet this procedure ensures a consistent behavior and positive end-user experience. If an application requested a specific power level for the device, Power Manager might not call the IOCTL_POWER_SET handler in response to DevicePowerNotify. Accordingly, device drivers should not assume that successful calls to DevicePowerNotify result in a call to the IOCTL_POWER_SET handler or that any calls to IOCTL_POWER_SET are the result of a DevicePowerNotify call. Power Manager can send notifications to a device

driver in many situations, such as during a system power state transition. To receive power management notifications, device drivers must advertise that they are power management enabled, either statically through the IClass registry entry in the driver's registry subkey or dynamically by using the AdvertiseInterface function.

XXX_PowerUp and XXX_PowerDown

You can use the XXX_PowerUp and XXX_PowerDown stream interface functions to implement suspend and resume functionality. The kernel calls XXX_PowerDown right before powering down the CPU and XXX_PowerUp right after powering it up. It is important to note that the system operates in single-threaded mode during these stages with most system calls disabled. For this reason, Microsoft recommends using the XXX_IOControl function instead of XXX_PowerUp and XXX_PowerDown to implement power management features, including suspend and resume functionality.

CAUTION **Power management restrictions**

If you implement suspend and resume functionality based on the XXX_PowerUp and XXX_PowerDown functions, avoid calling system APIs, particularly thread-blocking APIs, such as WaitForSingleObject. Blocking the active thread in single-threaded mode causes an unrecoverable system lockup.

IOControl

The best way to implement power manager in a stream driver is to add support for power management I/O control codes to the driver's IOControl function. When you notify Power Manager about your driver's power management capabilities through an IClass registry entry or the AdvertiseInterface function, your driver receives corresponding notification messages.

Table 6–8 lists the IOCTLs that Power Manager can send to a device driver to perform power management–related tasks.

Table 6-8 **Power management IOCTLs**

Function	Description
IOCTL_POWER_CAPABILITIES	Requests information on what power states the driver supports. Note that the driver can still be set to other power states (D0 through D4).

Table 6-8 Power management IOCTLs (Continued)

Function	Description
IOCTL_POWER_GET	Requests the current power state of the driver.
IOCTL_POWER_SET	Sets the power state of the driver. The driver maps the received power state number to actual settings and changes the device state. The new device driver power state should be returned to Power Manager in the output buffer.
IOCTL_POWER_QUERY	Power Manger checks to see if the driver is able to change the state of the device. This function is deprecated.
IOCTL_REGISTER_POWER_RELATIONSHIP	Enables a device driver to register as the proxy for another device driver, so that Power Manager passes all power requests to this device driver.

IClass Power Management Interfaces

Power Manager supports an IClass registry entry that you can configure in the driver's registry subkey to associate your driver with one or more device class values. An IClass value is a globally unique identifier (GUID) referring to an interface, defined under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Interfaces registry key. The most important interface for driver developers is the interface for generic power management–enabled devices, associated with the GUID {A32942B7-920C-486b-B0E6-92A702A99B35}. By adding this GUID to the IClass registry entry of your device driver, you can inform Power Manager to send your driver IOCTLs for power management notifications, as illustrated in Figure 6–7.

Figure 6-7 Configuring the IClass registry entry to receive power management notifications.

MORE INFO Registry settings for power management

You can also use registry settings and device classes to configure the default power states for a device, as explained in Lesson 5, “Implementing Power Management,” of Chapter 3.

Lesson Summary

To ensure reliable power management on Windows Embedded CE, device drivers should not change their own internal power state without the involvement of Power Manager. Operating system components, drivers, and applications can call the DevicePowerNotify function to request a power state change. Accordingly, Power Manager sends a power state change request to the driver if the power state change is consistent with the current state of the system. The recommended way to add power management capabilities to a stream driver is to add support for power management IOCTLs to the XXX_IOControl function. The XXX_PowerUp and XXX_PowerDown functions only provide limited capability because Power Manager calls these functions at a time when the system operates in single-thread mode. If the device advertises a power management interface through an IClass registry entry or calls the AdvertiseInterface to announce supported IOCTL interfaces dynamically, Power Manager will send IOCTLs to the device driver in response to power-related events.

Lesson 6: Marshaling Data across Boundaries

In Windows Embedded CE 6.0, each process has its own separate virtual memory space and memory context. Accordingly, marshaling data from one process to another requires either a copying process or a mapping of physical memory sections. Windows Embedded CE 6.0 handles most of the details and provides system functions, such as OALPAtoVA and MmMapIoSpace, to map physical memory addresses to virtual memory addresses in a relatively straightforward way. However, driver developers must understand the details of data marshaling to ensure a reliable and secure system. It is imperative to validate embedded pointers and properly handle asynchronous buffer access so that a user application cannot exploit a kernel-mode driver to manipulate memory regions that the application should not be able to access. Poorly implemented kernel-mode drivers can open a back door for malicious applications to take over the entire system.

After this lesson, you will be able to:

- Allocate and use buffers in device drivers.
- Use embedded pointers in an application.
- Verify the validity of embedded pointers in a device driver.

Estimated lesson time: 30 minutes.

Understanding Memory Access

Windows Embedded CE works in a virtual memory context and hides the physical memory, as illustrated in Figure 6–8. The operating system relies on the Virtual Memory Manager (VMM) and the processor's Memory Management Unit (MMU) for the translation of the virtual addresses into physical addresses and for other memory access management tasks.

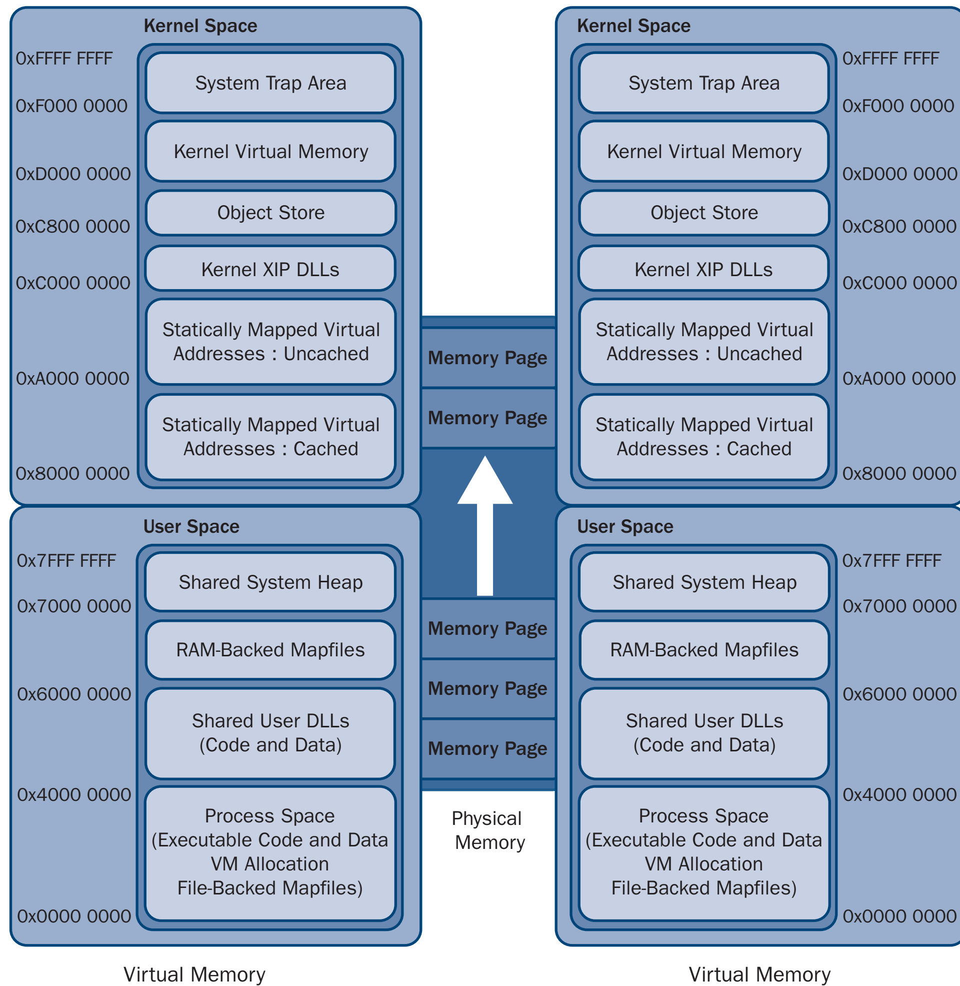


Figure 6-8 Virtual memory regions in kernel space and user space

Physical addresses are not directly addressable by the CPU except during initialization before the kernel has enabled the MMU, yet this does not imply that the physical memory is no longer accessible. In fact, every fully allocated virtual memory page must map to some actual physical page on the target device. Processes in separate virtual address spaces only require a mechanism to map the same physical memory areas into an available virtual memory region to share data. The physical address is the same across all processes running on the system. Only the virtual addresses differ. By translating the physical address per process into a valid virtual

address, processes can access the same physical memory region and share data across process boundaries.

As mentioned earlier in this chapter, kernel-mode routines, such as ISRs, can call OALPAtoVA to map a physical address (PA) into a cached or uncached virtual address (VA). Because OALPAtoVA maps the physical address to a virtual address in the kernel space, user-mode processes, such as ISTs, cannot use this function. The kernel space is inaccessible in user mode. However, threads in user-mode processes, such as ISTs, can call the MmMapIoSpace function to map a physical address to a nonpaged, cached or uncached virtual address in the user space. The MmMapIoSpace call results in the creation of a new entry in the MMU Table (TBL) if no match was found or it returns an existing mapping. By calling the MmUnmapIoSpace function, the user-mode process can release the memory again.

NOTE Physical memory access restrictions

Applications and user-mode drivers cannot access physical device memory directly. User-mode processes must call HalTranslateBusAddress to map the physical device memory range for the bus to a physical system memory address before calling MmMapIoSpace. To convert a bus address into a virtual address in a single function call, use the TransBusAddrToVirtual function, which in turn calls HalTranslateBusAddress and MmMapIoSpace.

Allocating Physical Memory

It's possible to allocate a portion of memory so you can use it in a driver or the kernel. There are two ways to do this:

- **Dynamically, by calling the AllocPhysMem function** AllocPhysMem allocates contiguous physical memory in one or more pages that you can map to virtual memory in the user space by calling MmMapIoSpace or OALPAtoVA, depending on whether the code is running in user mode or kernel mode. Because physical memory is allocated in units of memory pages, it is not possible to allocate less than a page of physical memory. The size of the memory page depends on the hardware platform. A typical page size is 64 KB.
- **Statically, by creating a RESERVED section in the Config.bib file** You can statically reserve physical memory by using the MEMORY section of a run-time image's BIB file, such as Config.bib in the BSP folder. Figure 6–9 illustrates this approach. The names of the memory regions are for informational purposes and are only used to identify the different memory areas defined on the system. The important pieces of information are the address definitions and the RESERVED

keyword. According to these settings, Windows Embedded CE excludes the reserved regions from system memory so that they can be used for DMA by peripherals and data transfers. There is no risk of access conflicts because the system does not use reserved memory areas.

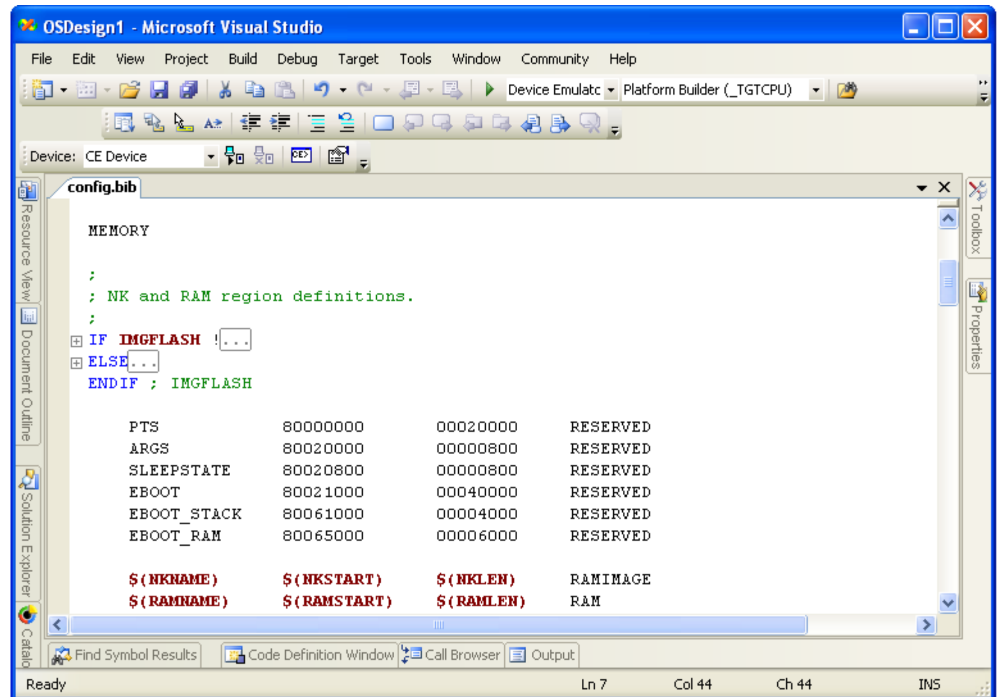


Figure 6-9 Definition of reserved memory regions in a Config.bib file

Application Caller Buffers

In Windows Embedded CE 6.0, applications and device drivers run in different process spaces. For example, Device Manager loads stream drivers into the kernel process (Nk.exe) or into the user-mode driver host process (Udevice.exe), whereas each application runs in its own individual process space. Because pointers to virtual memory addresses in one process space are invalid in other process spaces, you must map or marshal pointer parameters if separate processes are supposed to access the same buffer region in physical memory for communication and data transfer across process boundaries.

Using Pointer Parameters

A pointer parameter is a pointer that a caller can pass as a parameter to a function. The DeviceIoControl parameters lpInBuf and lpOutBuf are perfect examples. Applications can use DeviceIoControl to perform direct input and output operations. A pointer to an input buffer (lpInBuf) and a pointer to an output buffer (lpOutBuf) enable data transfer between the application and the driver. DeviceIoControl is declared in Winbase.h as follows:

```
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);
```

Pointer parameters are convenient to use in Windows Embedded CE 6.0 because the kernel automatically performs full access checks and marshaling on these parameters. In the DeviceIoControl declaration above, you can see that the buffer parameters lpInBuf and lpOutBuf are defined as in/out parameters of a specified size, while lpBytesReturned is an out-only parameter. Based on these declarations, the kernel can ensure that an application does not pass in an address to read-only memory (such as a shared heap, which is read-only to user-mode processes, but writable to the kernel) as an in/out or out-only buffer pointer or it will trigger an exception. In this way, Windows Embedded CE 6.0 ensures that an application cannot gain elevated access permissions to a memory region through a kernel-mode driver. Accordingly, on the driver's side, you do not have to perform any access checks for the pointers passed in through the XXX_IOControl stream interface function (pBufIn and pBufOut).

Using Embedded Pointers

Embedded pointers are pointers that a caller passes to a function indirectly through a memory buffer. For example, an application can store a pointer inside the input buffer passed in to DeviceIoControl through the parameter pointer lpInBuf. The kernel will automatically check and marshal the parameter pointer lpInBuf, yet the system has no way to identify the embedded pointer inside the input buffer. As far as the kernel is concerned, the memory buffer simply contains binary data. Windows Embedded CE 6.0 provides no mechanisms to specify explicitly that this block of memory contains pointers.

Because embedded pointers bypass the kernel's access checks and marshaling helpers, you must perform access checks and marshaling of embedded pointers in device drivers manually before you can use them. Otherwise, you might create vulnerabilities that malicious user-mode code can exploit to perform illegal actions and compromise the entire system. Kernel-mode drivers enjoy a high level of privileges and can access system memory that user-mode code should not be able to access.

To verify that the caller process has the required access privileges, marshal the pointer, and access the buffer, you should call the CeOpenCallerBuffer function. CeOpenCallerBuffer checks access privileges based on whether the caller is running in kernel-mode or user-mode, allocates a new virtual address for the physical memory of the caller's buffer, and optionally allocates a temporary heap buffer to create a copy of the caller's buffer. Because the mapping of the physical memory involves allocating a new virtual address range inside the driver, do not forget to call CeCloseCallerBuffer when the driver has finished its processing.

Handling Buffers

Having performed implicit (parameter pointers) or explicit (embedded pointers) access checks and pointer marshaling, the device driver is ready to access the buffer. However, access to the buffer is not exclusive. While the device driver reads data from and writes data to the buffer, the caller might also read and write data concurrently, as illustrated in Figure 6–10. Security issues can arise if a device driver stores marshaled pointers in the caller's buffer. A second thread in the application could then manipulate the pointer to access a protected memory region through the driver. For this reason, drivers should always make secure copies of the pointers and buffer size values they receive from a caller and copy embedded pointers to local variables to prevent asynchronous modification.

IMPORTANT Asynchronous buffer handling

Never use pointers in the caller's buffer after they have been marshaled, and do not use the caller's buffer to store marshaled pointers or other variables required for driver processing. For example, copy buffer size values to local variables so that callers cannot manipulate these values to cause buffer overruns. One way to prevent asynchronous modification of a buffer by the caller is to call CeOpenCallerBuffer with the ForceDuplicate parameter set to TRUE to copy the data from the caller's buffer to a temporary heap buffer.

Figure 6-10 Manipulating a marshaled pointer in a shared buffer

Synchronous Access

Synchronous memory access is synonymous with non-concurrent buffer access. The caller's thread waits until the function call returns, such as DeviceIoControl, and there are no other threads in the caller process that access the buffer while the driver performs its processing tasks. In this scenario, the device driver can use parameter pointers and embedded pointers (after a call to CeOpenCallerBuffer) without additional precautions.

The following is an example of accessing a buffer from an application synchronously. This sample source code is an exerpt from an XXX_IOControl function of a stream driver:

```
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);

    }

...

}
```

Asynchronous Access

Asynchronous buffer access assumes that multiple caller and driver threads access the buffer sequentially or concurrently. Both scenarios present challenges. In the sequential access scenario, the caller thread might exit before the driver thread has finished its processing. By calling the marshaling helper function CeAllocAsynchronousBuffer, you must re-marshal the buffer after it was marshaled by CeOpenCallerBuffer to ensure in the driver that the buffer remains available even if the caller's address space is unavailable. Do not forget to call CeFreeAsynchronousBuffer after the driver has finished its processing.

To ensure that your device driver works in kernel and user mode, use the following approach to support asynchronous buffer access:

- **Pointer parameters** Pass pointer parameters as scalar DWORD values and then call CeOpenCallerBuffer and CeAllocAsynchronousBuffer to perform access checks and marshaling. Note that you cannot call CeAllocAsynchronousBuffer on a pointer parameter in user-mode code or perform asynchronous write-back of O_PTR or IO_PTR values.
- **Embedded pointers** Pass embedded pointers to CeOpenCallerBuffer and CeAllocAsynchronousBuffer to perform access checks and marshaling.

To address the second scenario of concurrent access, you must create a secure copy of the buffer after marshaling, as mentioned earlier. Calling CeOpenCallerBuffer with the ForceDuplicate parameter set to TRUE and CeCloseCallerBuffer is one option. Another is to call CeAllocDuplicateBuffer and CeFreeDuplicateBuffer for buffers

referenced by parameter pointers. You can also copy a pointer or buffer into a stack variable or allocate heap memory by using VirtualAlloc and then use memcpy to copy the caller's buffer. Keep in mind that if you do not create a secure copy, you're leaving in a vulnerability that a malicious application could use to take control of the system.

Exception Handling

Another important aspect that should not be ignored in asynchronous buffer access scenarios revolves around the possibility that embedded pointers might not point to valid memory addresses. For example, an application can pass a pointer to a driver that refers to an unallocated or reserved memory region, or it could asynchronously free the buffer. To ensure a reliable system and prevent memory leaks, you should enclose buffer-access code in a __try frame and any cleanup code to free memory allocations in a __finally block or an exception handler. For more information about exception handling, see Chapter 3, "Performing System Programming."

Lesson Summary

Windows Embedded CE 6.0 facilitates inter-process communication between applications and device drivers through kernel features and marshaling helper functions that hide most of the complexities from driver developers. For parameter pointers, the kernel performs all checks and pointer marshaling automatically. Only embedded pointers require extra care because the kernel cannot evaluate the content of application buffers passed to a driver. Validating and marshaling an embedded pointer in a synchronous access scenario involves a straightforward call to CeOpenCallerBuffer. Asynchronous access scenarios, however, require an additional call to CeAllocAsynchronousBuffer to marshal the pointer one more time. To ensure that your driver does not introduce system vulnerabilities, make sure you handle buffers correctly, create a secure copy of the buffer content so that callers cannot manipulate the values, and do not use pointers or buffer size values in the caller's buffer after they have been marshaled. Never store marshaled pointers or other variables required for driver processing in the caller's buffer.

Lesson 7: Enhancing Driver Portability

Device drivers help to increase the flexibility and portability of the operating system. Ideally, they require no code changes to run on different target devices with varying communication requirements. There are several relatively straightforward techniques that you can use to make your drivers portable and reusable. One common approach is to maintain configuration settings in the registry instead of hardcoding the parameters into the OAL or the driver. Windows Embedded CE also supports a layered architecture based on MDD and PDD that you can leverage in your device driver design, and there are further techniques that you can use to implement drivers in a bus-agnostic way to support peripheral devices regardless of the bus type to which they are connected.

After this lesson, you will be able to:

- Describe how to use registry settings to increase the portability and reusability of a device driver.
- Implement a device driver in a bus-agnostic way.

Estimated lesson time: 15 minutes.

Accessing Registry Settings in a Driver

To increase the portability and reusability of a device driver, you can configure registry entries, which you should add to the driver's registry subkey. For example, you can define I/O-mapped memory addresses or settings for installable ISRs that the device driver loads dynamically. To access the entries in a device driver's registry key, the driver has to identify where its own settings are located. This is not necessarily the HKEY_LOCAL_MACHINE\Drivers\BuiltIn key. However, the correct path information is available in the Key value that you can find under the HKEY_LOCAL_MACHINE\Drivers\Active key in the loaded driver's subkey. Device Manager passes the path to the driver's Drivers\Active subkey to the XXX_Init function in the LPCTSTR pContext parameter. The device driver can then use this LPCTSTR value in a call to OpenDeviceKey to obtain a handle to the device's registry key. It is not necessary to read the Key values from the driver's Drivers\Active subkey directly. The handle returned by OpenDeviceKey points to the driver's registry key, which you can use like any other registry handle. Most importantly, do not forget to close the handle when it is no longer needed.

TIP XXX_Init function and driver settings

The XXX_Init function is the best place to determine all configuration settings for a driver defined in the registry. Rather than accessing the registry repeatedly in subsequent stream function calls, it is good practice to store the configuration information in the device context created and returned to Device Manager in response to the XXX_Init call.

Interrupt-Related Registry Settings

If your device driver must load an installable ISR for a device and you want to increase the portability of your code, you can register the ISR handler, IRQ, and SYSINTR values in registry keys, read these values from the registry when initializing the driver, verify that the IRQ and SYSINTR values are valid, and then install the specified ISR by using the LoadIntChainHandler function.

Table 6–9 lists the registry entries that you can configure for this purpose. By calling the DDKReg_GetIsrInfo function, you can then read these values and pass them to the LoadIntChainHandler function dynamically. For more information about interrupt handing in device drivers, see Lesson 4, “Implementing an Interrupt Mechanism in a Device Driver,” earlier in this chapter.

Table 6-9 Interrupt-related registry entries for device drivers

Registry Entry	Type	Description
IRQ	REG_DWORD	Specifies the IRQ used to request a SYSINTR for setting up an IST within the driver.
SYSINTR	REG_DWORD	Specifies a SYSINTR value to use for setting up an IST within the driver.
IsrDll	REG_SZ	The filename of the DLL containing the installable ISR.
IsrHandler	REG_SZ	Specifies the entry point for the installable ISR that the specified DLL exposes.

Memory-Related Registry Settings

Memory-related registry values enable you to configure a device through the registry. Table 6–10 lists the memory-related registry information that a driver can obtain in a DDKWINDOWINFO structure by calling DDKReg_GetWindowInfo. By using the BusTransBusAddrToVirtual function, you can map the bus addresses of memory-mapped windows to physical system addresses you can then translate into virtual addresses by using MnMapIoSpace.

Table 6-10 Memory-related registry entries for device drivers

Registry Entry	Type	Description
IoBase	REG_DWORD	A bus-relative base of a single memory-mapped window used by the device.
IoLen	REG_DWORD	Specifies the length of the memory-mapped window defined in IoBase.
MemBase	REG_MULTI_SZ	A bus-relative base of multiple memory-mapped windows used by the device.
MemLen	REG_MULTI_SZ	Specifies the length of the memory-mapped memory windows defined in MemBase.

PCI-Related Registry Settings

Another registry helper function that you can use to populate a DDKPCIINFO structure with the standard PCI device instance information is DDKReg_GetPciInfo. Table 6–11 lists the PCI-related settings you can configure in a driver's registry subkey.

Table 6-11 PCI-related registry entries for device drivers

Registry Entry	Type	Description
DeviceNumber	REG_DWORD	The PCI device number.
FunctionNumber	REG_DWORD	The PCI function number of the device, which indicates a single function device on a multifunction PCI card.
InstanceIndex	REG_DWORD	The instance number of the device.

Table 6-11 PCI-related registry entries for device drivers (Continued)

Registry Entry	Type	Description
DeviceID	REG_DWORD	The type of the device
ProgIF	REG_DWORD	A register-specific programming interface, for example, USB OHCI or UHCI.
RevisionId	REG_DWORD	The revision number of the device.
Subclass	REG_DWORD	The basic function of the device; for example, an IDE controller.
SubSystemId	REG_DWORD	The type of card or subsystem that uses the device.
SubVendorId	REG_DWORD	The vendor of the card or subsystem that uses the device.
VendorId	REG_MULTI_SZ	The manufacturer of the device.

Developing Bus-Agnostic Drivers

Similar to settings for installable ISRs, memory-mapped windows, and PCI device instance information, you can maintain any GPIO numbers or timing configurations in the registry and achieve in this way a bus-agnostic driver design. The underlying idea of a bus-agnostic driver is the support of multiple bus implementations for the same hardware chipset, such as PCI or PCMCIA, without requiring code modifications.

To implement a bus-agnostic driver, use the following approach:

1. Maintain all necessary configuration parameters in the driver's registry subkey, and use the Windows Embedded CE registry helper functions DDKReg_GetIsrInfo, DDKReg_GetWindowInfo, and DDKReg_GetPciInfo to retrieve these settings during driver initialization.
2. Call HalTranslateBusAddress to translate bus-specific addresses to system physical addresses and then call MmMapIoSpace to map the physical addresses to virtual addresses.

3. Reset the hardware, mask the interrupt, and load an installable ISR by calling the LoadIntChainHandler function with information obtained from DDKReg_GetIsrInfo.
4. Load any initialization settings for the installable ISR from the registry by using RegQueryValueEx and pass the values to the installable ISR in a call to KernelLibIoControl with a user-defined IOCTL. For example, the Generic Installable Interrupt Service Routine (GIISR) included in Windows Embedded CE uses an IOCTL_GIISR_INFO handler to initialize instance information that enables GIISR to recognize when the device's interrupt bit is set and return the corresponding SYSINTR value. You can find the source code in the C:\Wince600\Public\Common\Oak\Drivers\Giisr folder.
5. Begin the IST by calling the CreateThread function and unmask the interrupt.

Lesson Summary

To increase the portability of a device driver, you can configure the registry entries in the driver's registry subkey. Windows Embedded CE provides several registry helper functions that you can then use to retrieve these settings, such as DDKReg_GetIsrInfo, DDKReg_GetWindowInfo, and DDKReg_GetPciInfo. These helper functions query specific information for installable ISRs, memory-mapped windows, and PCI device instance information, yet you can also call RegQueryValueEx to retrieve values from other registry entries. However, to use any of these registry functions, you must first obtain a handle to the driver's registry subkey by calling OpenDeviceKey. OpenDeviceKey expects a registry path, which Device Manager passes to the driver in the XXX_Init function call. Do not forget to close the registry handle when it is no longer needed.

Lab 6: Developing Device Drivers

In this lab, you implement a stream driver that stores and retrieves a string of 128 Unicode characters in memory. A base version of this driver is available in the companion material for this book. You only need to add the code as a subproject to an OS design. You then configure .bib file and registry settings to load this driver automatically during boot time and create a WCE Console Application to test the driver's functionality. In a last step, you add power management support to the string driver.

> **NOTE** **Detailed step-by-step instructions**
>
> To help you successfully master the procedures presented in this Lab, see the document "Detailed Step-by-Step Instructions for Lab 6" in the companion material for this book.

▶ Add a Stream Interface Driver to a Run-Time Image

1. Clone the Device Emulator BSP and create an OS design based on this BSP as outlined in Lab 2, "Building and Deploying a Run-Time Image."
2. Copy the string driver source code that you can find on the companion CD in the \Labs\StringDriver\String folder into your BSP folder in the path %_WINCEROOT%\Platform\<*BSPName*>\Src\Drivers. This should result in a folder named String in your platform in the Drivers folder, and immediately inside this folder you should have the files from the driver on the companion CD, such as sources, string.c, string.def. It's possible to write a driver from scratch, but starting from a working example such as this is much quicker.
3. Add an entry to the Dirs file in the Drivers folder above your new String folder to include the string driver into the build process.

> **CAUTION** **Include In Build option**
>
> Do not use the Include In Build option in Solution Explorer to include the string driver into the build process. Solution Explorer removes important CESYSGEN directives from the Dirs file.

4. Add an entry to Platform.bib to add the built string driver, contained in **$(_FLATRELEASEDIR)**, to the run-time image. Mark the driver module as a hidden system file.
5. Add the following line to Platform.reg to include the string driver's .reg file into the run-time image's registry:

#include "$(_TARGETPLATROOT)\SRC\DRIVERS\String\string.reg"

6. Build the string driver by right clicking it in Solution Explorer and selecting Build.
7. Make a new run-time image in Debug mode.

> **NOTE Building the run-time image in Release mode**
> If you want to work with the Release version of the run-time image, you must change the DEBUGMSG statements in the driver code to RETAILMSG statements to output the driver messages.

8. Open the generated Nk.bin in the flat release directory to verify that it contains String.dll and registry entries in the HKEY_LOCAL_MACHINE\Drivers\BuiltIn\String subkey to load the driver at startup.
9. Load the generated image on the Device Emulator.
10. Open the Modules window after the image starts by pressing CTRL+ALT+U, or open the Debug menu in Visual Studio, point to Windows, and then select Modules. Verify that the system has loaded **string.dll**, as illustrated in Figure 6–11.

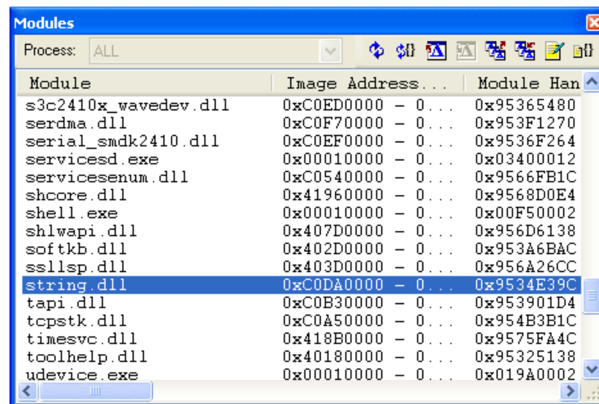


Figure 6-11 Modules window with loaded string driver

▶ Access the Driver from an Application

1. Create a new WCE Console Application subproject as part of your OS design by using the Windows Embedded CE Subproject Wizard. Select WCE Console Application and the template A Simple Windows Embedded CE Console Application.

2. Modify the subproject image settings to exclude the subproject from the image by right clicking the OS design name in the solution view and selecting Properties.
3. Include <windows.h> and <winioctl.h>
4. Add code to the application to open an instance of the driver by using CreateFile. For the second CreateFile parameter (dwDesiredAccess), pass in GENERIC_READ|GENERIC_WRITE. For the fifth parameter (dwCreationDisposition), pass in OPEN_EXISTING. If you open the driver with the $device naming convention, be sure to escape the slashes and not include the colon at the end of the filename.

```
HANDLE hDrv = CreateFile(L"\\$device\\STR1",
                         GENERIC_READ|GENERIC_WRITE,
                         0, 0, OPEN_EXISTING, 0, 0);
```

5. Copy the IOCTL header file (String_ioctl.h) from the string driver folder to the new application's folder and include it in the source code file.
6. Declare an instance of a PARMS_STRING structure defined in String_iocontrol.h, included along with the rest of the sample string driver, to enable applications to store a string in the driver using the following code:

```
PARMS_STRING stringToStore;
wcscpy_s(stringToStore.szString,
         STR_MAX_STRING_LENGTH,
         L"Hello, driver!");
```

7. Use a DeviceIoControl call with an I/O control code of IOCTL_STRING_SET to store this string in the driver.
8. Run the application by building it and selecting Run Programs from the Target menu.
9. The Debug window should show the message **Stored String "Hello, driver!" Successfully** when you run the application, as shown in Figure 6–12.

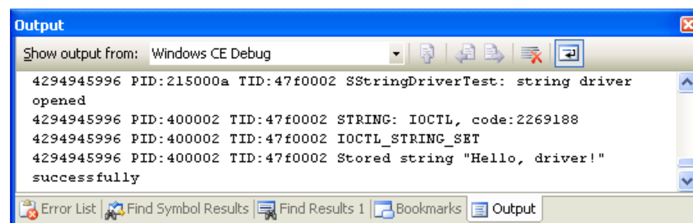


Figure 6-12 A debug message from the string driver

Adding Power Management Support

1. Turn off and detach from the Device Emulator.
2. Add the IClass for generic power management devices with the following line to the string driver's registry key in String.reg:

```
"IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

3. Add the power management code that you can find in the StringDriverPowerCode.txt file under \Labs\StringDriver\Power on the companion CD to the string driver's IOControl function to support IOCTL_POWER_GET, IOCTL_POWER_SET, and IOCTL_POWER_CAPABILITIES.
4. Add code to the string driver's device context so it stores its current power state:

```
CEDEVICE_POWER_STATE CurrentDx;
```

5. Add the header **<pm.h>** to the application, and add calls to SetDevicePower with the name of the string driver and different power states; for example:

```
SetDevicePower(L"STR1:", POWER_NAME, D2);
```

6. Run the application again, and observe the power state–related debug messages when Power Manager changes the string driver's power state, as shown in Figure 6-13.

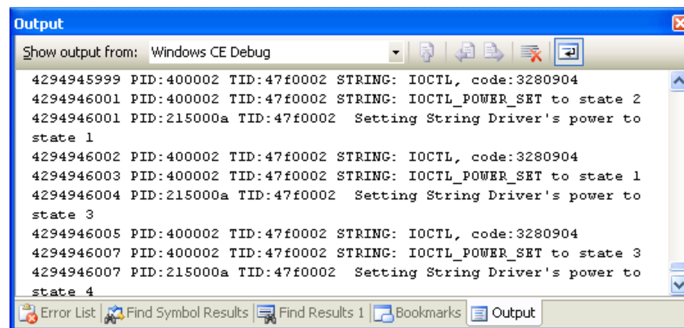


Figure 6-13 Power management–related debug messages from the string driver

Chapter Review

Windows Embedded CE 6.0 is extraordinarily modular in its design and supports ARM-, MIPS-, SH4-, and x86–based boards in a multitude of hardware configurations. The CE kernel contains the core OS code and the platform-specific code resides in the OAL and in device drivers. In fact, device drivers are the largest part of the BSP for an OS design. Rather than accessing the hardware directly, the operating system loads the corresponding device drivers, and then uses the functions and I/O services that these drivers provide.

Windows Embedded CE device drivers are DLLs that adhere to a well-known API so that the operating system can load them. Native CE drivers interface with GWES while stream drivers interface with Device Manager. Stream drivers implement the stream interface API so that their resources can be exposed as special file system resources. Applications can use the standard file system APIs to interact with these drivers. The stream interface API also includes support for IOCTL handlers, which come in handy if you want to integrate a driver with Power Manager. For example, Power Manager calls XXX_IOControl with an IOControl code of IOCTL_POWER_SET passing in the requested device power state.

Native and stream drivers can feature a monolithic or layered design. The layered design splits the device driver logic in an MDD and a PDD part, which helps to increase the reusability of the code. The layered design also facilitates driver updates. Windows Embedded CE also features a flexible interrupt-handling architecture based on ISRs and ISTs. The ISR's main task is to identify the interrupt source and notify the kernel with a SYNTINR value about the IST to run. The IST performs the majority of the processing, such as time-consuming buffer copying processes.

In general, you have two options to load a driver under Windows Embedded CE 6.0. You can add the driver's registry settings to the BuiltIn registry key to start the driver automatically during the boot process or you load the driver automatically in a call to ActivateDeviceEx. Depending on the driver's registry entries, you can run a driver in kernel mode or user mode. Windows Embedded CE 6.0 includes a user-mode driver host process and a Reflector service that enables most kernel-mode drivers to run in user mode without code modifications. Because device drivers run in different process spaces than applications on Windows Embedded CE 6.0, you must marshal the data in either a mapping of physical memory sections or copying process to facilitate communication. It is imperative to validate and marshal embedded pointers by calling CeOpenCallerBuffer and CeAllocAsynchronousBuffer and properly

handling asynchronous buffer access so that a user application cannot exploit a kernel-mode driver to take over the system.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- IRQ
- SYSINTR
- IST
- ISR
- User mode
- Marshaling
- Stream interface
- Native interface
- PDD
- MDD
- Monolithic
- Bus-agnostic

Suggested Practice

To help you successfully master the exam objectives presented in this chapter, complete the following tasks:

Enhance Power Management Features

Continue to develop the string driver's power management code.

- **Clear the string buffer** Modify the string driver to delete the contents of the string buffer when the device driver switches into the power state D3 or D4.
- **Change the power capabilities** See what happens when you return a different POWER_CAPABILITIES value to Power Manager.

More IOCTLs

Expand on the features of the string driver by adding more IOCTL handlers.

- **Reverse the stored string** Add an IOCTL to reverse the contents of the string in the buffer.
- **Concatenate a string** Add an IOCTL that concatenates a second string to the string stored without overrunning the buffer.
- **Embedded pointers** Replace the string parameter with a pointer to a string and access it with CeOpenCallerBuffer.

Installable ISR

Learn more about installable ISRs by reading the product documentation.

- **Learn more about installable ISRs** Read the section "Installable ISRs and Device Drivers" in the Windows Embedded CE 6.0 Documentation, available on the Microsoft MSDN Web site at *http://msdn2.microsoft.com/en-us/library/aa929596.aspx* to learn more about installable ISRs.
- **Find an example of an installable ISR** Find an example of an installable ISR and study its structure. A good starting point is the GIISR code that you can find in the %_WINCEROOT%\Public\Common\Oak\Drivers\Giisr folder.

Glossary

Application Programming Interface (API) An API is the function interface that an operating system or library provides to support requests from application programs.

Application Verifier (AppVerifier) AppVerifier enables developers to find subtle programming errors, such as heap corruption and incorrect handle usage, that can be difficult to identify with normal application testing

Asynchronous Access When two or more threads access the same buffer at the same time.

Binary Image Builder (.bib) A .bib file defines which modules and files are included in a run-time image.

Boot Loader Piece of code executed at the processor startup to initialize the processor and then launch an operating system.

Board Support Package (BSP) A BSP is the common name for all board hardware-specific code. It typically consists of the boot loader, the OEM adaptation layer (OAL), and board-specific device drivers.

Catalog A container of components that presents a selectable feature for an OSDesign to the user.

Debugger Extension Commands (CeDebugX) CeDebugX is an extension to the Platform Builder debugger. It presents detailed information about the state of the system at break time and attempts to diagnose crashes, hangs, and deadlocks.

Windows Embedded CE Test Kit (CETK) The CETK is a tool you can use to test device drivers that you develop for the Windows Embedded CE operating system.

Cloning During cloning, you generate an exact copy of files to keep a secure copy of them before performing modifications. Code in the PUBLIC folder should always be cloned before making modifications.

Component A CE feature that can be added to or removed from an OS Design using the catalog.

Core Connectivity (CoreCon) Windows CE supports a unified communications infrastructure called Core Connectivity that enables full-featured connectivity for downloading and debugging.

Critical Section An object with a synchronization process that is similar to a mutex object. The difference is that a critical section can only be accessed by the threads of a single process.

Data Marshaling A process done on data to check the access rights and validity of the data for a different process.

Debug Zone A flag to enable or disable debug messages related to a certain functionality or mode of a driver.

Device Driver A device driver is software that manages the operation of a device by abstracting the functionality of a physical or virtual device.

Dirs File A Dirs file is a text file that specifies the subdirectories that contain source code to be built.

Embedded Pointer A pointer embedded in a memory structure.

Environment Variable A Windows environment variable that can enable or disable features. It is generally used to configure the build system and OS design from the catalog

Event Synchronization objects used by threads and the kernel to notify other threads in the system.

Exception An exception is an abnormal situation that happens while a program is running.

Iltiming Iltiming determines interrupt service routine (ISR) and interrupt service thread (IST) latencies on a Windows Embedded CE system.

Interrupt A trigger that suspends (interrupts) the system temporarily to indicate that something has happened that requires processing. Each interrupt on a system is associated with a particular Interrupt Request (IRQ) value and this IRQ value is associated with one or more ISR.

Interrupt Service Routine (ISR) An ISR is a software routine that hardware invokes in response to an interrupt. ISRs examine an interrupt and determine how to handle it by returning a SYSINTR value, which is then associated with an IST.

Interrupt Service Thread (IST) The IST is a thread that does most of the interrupt processing. The OS wakes the IST when the OS has an interrupt to process. After each IST is associated to a SYSINTR value, the SYSINTR value can be returned from an ISR, and then the associated IST runs.

IRQ (Interrupt Request) IRQ values are associated in hardware with interrupts. Each IRQ value can be associated with one or more ISRs that the system will run to process the associated interrupt when it is triggered.

Kernel Debugger The kernel debugger integrates functionality required to configure a connection to a target device and download a run-time image to the target device. It allows the debugging of the OS, abbreviate drivers, and applications.

Kernel Independent Transport Layer (KITL) The KITL is designed to provide an easy way to support debugging services.

Kernel-Mode Driver A driver that runs in the kernel's memory space.

Kernel Tracker This tool provides a visual representation on a development workstation of OS and application events occurring on a Windows Embedded CE–based device.

Layered Driver A driver that is separated into several layers to make it easier to maintain and reuse code at a later date.

Model Device Driver (MDD) The MDD layer of a layered driver has a standardized interface to the OS and Platform Device Driver (PDD) layer and performs all hardware independent processing related to the driver.

Monolithic Driver A driver that is not separated into different layers. It can also mean any driver that does not conform to the standard Model Device Driver (MDD)/ Platform Device Driver (PDD) layer architecture of CE, even if the driver does have its own layering scheme.

Mutex A mutex object is a synchronization object whose state is set to signaled when it is not owned by any thread and nonsignaled when it is owned. A mutex can only be owned by a single thread at a time. It is used to represent a resource that should only be accessed by one thread at any given time, such as a global variable or a hardware device.

Native Driver Touchscreen, Keyboard and Display drivers are the only native drivers existing in Windows Embedded CE and are managed by GWES rather than Device Manager.

OEM adaptation layer (OAL) An OAL is a layer of code that logically resides between the Windows Embedded CE kernel and the hardware of your target device. Physically, the OAL is linked with the kernel libraries to create the kernel executable file.

Operating System Benchmark (OSBench) A tool that is used to measure the performance of the scheduler.

OS Design A Platform Builder for Windows Embedded CE6 R2 project that generates a customized binary runtime image of the Windows Embedded CE6 R2 operating system

Platform Dependent Driver (PDD) The PDD layer of a layered driver is the part that interfaces directly with hardware and performs any hardware-specific processing.

Power Manager Controls the power consumption of the system by assigning a power state between D0 (fully on) and D4 (fully off) to the system as a whole and to individual drivers. It coordinates transitions between these states based on user and system activity, as well as specified requirements.

Process A process is a program in Windows Embedded CE. Each process can have multiple threads. A process can run in either user space or in kernel space.

Production Quality OAL (PQOAL) The PQOAL is a standardized OAL structure that simplifies and shortens the process of developing an OAL. It provides you with an improved level of OAL componentization through code libraries, directory structures that support code reuse, centralized configuration files, and a consistent architecture across processor families and hardware platforms.

Quick Fix Engineering (QFE) Windows Embedded CE patches that are available from Microsoft's website. They fix bugs and provide new features.

Reflector Service The service that allows user mode drivers to access the kernel and hardware by performing requests on their behalf.

Registry The information store for Windows Embedded CE that contains configuration information for hardware and software components.

Remote Performance Monitor This application can track the real-time performance of the operating system. It can also track memory usage, network latencies, and other elements

Run-time image The binary file that will be deployed on a hardware device . It also contains the complete operating system required files for applications and drivers.

Semaphore A semaphore object is a synchronization object that protects access to a hardware or software resource, allowing only a fixed number of concurrent threads to access it. The semaphore maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, no more threads can access the resource protected by the semaphore. The state of a semaphore is set to signaled when its count is greater than zero and nonsignaled when its count is zero.

Shell The shell is the software that will interpret user interactions with the device. It

launches when the device starts up. The default shell is called AYGShell and includes a desktop, start menu, and taskbar similar to those in desktop versions of Windows.

Software Development Kit (SDK) Used to allow third-party developers to make applications for a customized Windows Embedded CE6 R2 run-time image

Sources File A Sources file is a text file that sets macro definitions for the source code in a subdirectory. Build.exe uses these macro definitions to determine how to compile and link the source code.

Stream Interface Driver A stream interface driver is any driver that exposes the stream interface functions, regardless of the type of device controlled by the driver. All drivers other than the native drivers managed by GWES export a stream interface.

Subproject A set of files that you can easily integrate, remove and reuse in an OSDesign.

Synchronization Objects A synchronization object is an object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads.

Synchronous Access When two or more separate threads are working with the same buffer. Only one thread can access the buffer at any given time and no other threads access the buffer until the current thread is finished with it.

Sysgen The Sysgen phase is the first step in the build process done to filter the public and BSP folders. It identifies the files associated with the components selected in an OSDesign. During this phase, components selected in the OS Design are linked into executables and copied into the OS Design's folder.

Sysgen Variable A directive to the sysgen phase of the CE build process where selected CE features are linked together.

SYSINTR The value that corresponds to an IRQ. It is used to signal an associated event. This value is returned by an ISR in response to an interrupt.

Target Control Shell A shell in Platform Builder for Visual Studio providing access to debugger commands. The target control shell will be available when attached to a target system through KITL.

Thread The smallest software unit that the scheduler can manage on the OS. There can be multiple threads in a single driver or application.

User Mode Drivers loaded in user mode and all applications run in user memory space. When they are in this mode, drivers and applications do not have direct access to hardware memory and have restricted access to certain APIs and the kernel.

Virtual Memory Virtual memory is a way of abstracting the physical memory in a system to appear contiguous to processes that use it. Each process in Windows Embedded CE 6.0 R2 has two gigabytes of virtual memory space available, and to access physical memory from a process this memory must be mapped into the virtual address space of the process using MmMapIoSpace or OALPAtoVA.

Index

.NET Compact Framework 2.0 4, 29
.pbcxml files 21
.tks files. *See* Test Kit Suite (.tks) files
__except keyword 120
__finally keyword 121
__try keyword 121
32 processes limitation 219
3rdParty folder 22
4 GB address space 219

A

abstraction between the underlying hardware and the operating system 243
access checks 293
ActivateDevice function 258
ActivateDeviceEx function 258
ActiveSync 4, 29, 178
activity timers 128
  registry settings for 129
address mappings
  virtual-to-physical 213
address table 213
ADEFINES directive 60
ad-hoc solutions 21
Advanced Build Commands 24, 43
  Rebuild Current BSP And Subprojects 24
advanced debugger tools 161
Advanced Memory tool 162
AdvertiseInterface function 264, 284
alerts 90
AllocPhysMem function 226, 279, 290
analyzing build results 63–67
analyzing CETK test results 184
API. *See* application programming interface (API)
applets 97–98, 100
application caller buffer 291
application debugging 149
application programming interface (API) 11
  CPlApplet API 98
  Critical Section API 111
  Event API 114
  file system API 247
  GetProcAddress API 243
  Interlock API 115
  Mutex API 112
  non-real-time 84
  Power Manager APIs 126
  Process Management API 104, 130
  SignalStarted API 93
  stream interface API 247, 250
  Thread Management API 104
  Win32 API 84
application shortcuts on the desktop 55
Application Verifier tool 162, 179
architecture-generic tasks 214
ARM–based platforms 223
assembly language 188
ASSERTMSG macro 151
associating an OS design with multiple BSPs 10
asynchronous buffer access 288, 293
ATM. *See* automated teller machines (ATM)
attaching to a target device 71
audio device driver registration 262
Autoexit parameter 182
automated software testing 176
automated teller machines (ATM) 101
automatic loading of drivers 259
automatic startup 91
Autorun parameter 181
Autos tool 161
AUTOSIZE parameter 49

B

backlight driver 25
battery life 125
  battery critically low state 229
  battery level reaches zero 232
  Power Manager (PM.dll) and 127
best practices for debug zones 157
binary image builder (.bib) files 46
  automatic startup 47
  AUTOSIZE parameter 49
  BOOTJUMP parameter 49
  COMPRESSION parameter 49
  conditional processing in 53
  CONFIG section 49
  file type definitions in 52
  FILES section 51
  FIXUPVAR parameter 50
  FSRAMPERCENT parameter 50
  H flag 271
  K flag 270
  KERNELFIXUPS parameter 50
  MEMORY section 48

MODULES section 51
NK memory region 270
noncontiguous memory and 49
OUTPUT parameter 50
PROFILE parameter 50
Q flag 271
RAM_AUTOSIZE parameter 50
RAMIMAGE parameter 49
RESETVECTOR parameter 50
ROM_AUTOSIZE parameter 50
ROMFLAGS parameter 50
ROMOFFSET parameter 50
ROMSIZE parameter 50
ROMSTART parameter 50
ROMWIDTH parameter 50
S flag 271
sections of 47
SRE parameter 50
X86BOOT parameter 51
XIPSCHAIN parameter 51
binary ROM image file system (BinFS) 187
black shell 101
BLCOMMON framework 187
Bluetooth 29
Board Support Package (BSP) 3, 168, 197–239
adapting and configuring 199–218
boot loader and 201
cloning an existing reference BSP 201
advanced debugger tools 202
Cloning Wizard 202
components of a 200
configuration files 199, 201
device drivers and 201
folder structure of a 203
hardware-independent code and 201
memory mapping of a 219–227
OEM adaptation layer (OAL) and 201
platform-specific source code 205
reducing development time 201
serial debug output functions and 209
source code folders for device drivers 217
Board Support Packages wizard page 10
boot arguments (BootArgs) 171
driver globals and 207
boot loader
architecture 186
assembly language and 188
binary ROM image file system (BinFS) and 187
BLCOMMON framework 187
Board Support Package (BSP) and 201
BootLoaderMain function 209
BOOTME packet 212
Bootpart 187
code sharing between the OAL and the 214
debugging techniques for 188
driver globals and 207
Eboot 187
Ethdbg 205
Ethernet support functions 210
flash memory support 211
general task of a 186
hardware initialization tasks 209
kernel initialization routines and 187
memory mappings for a 206
menu of a 211
network drivers and 188
run-time image download via Ethernet 210
serial debug output functions and 209
StartUp entry point 208
testing 186
typical tasks of a 186
BootArgs. *See* boot arguments (BootArgs)
BOOTJUMP parameter 49
BootLoaderMain function 209
BOOTME packet 212
Bootpart 187
bootstrap service 148
breaking into the debugger 119
breakpoints 149, 161
enabling and managing 171
hardware and 174
interrupt handlers and 173
restrictions 173
setting too many 173
Tux DLLs and 184
BSP development 24
BSP. *See* Board Support Package (BSP)
Bsp_cfg.h file 276
BSPIntrInit function 277
buffer handling 293
buffer marshaling 268
buffer overrun 274
Buffer Tracked Events In RAM option 9
build commands 42
command line equivalents for 46
build configuration files 57–62
build configuration management 6
Advanced Build Commands 24
build configuration files 57–62
build directives 59
build options 8
Clean Sysgen command and 45
configurations files 14
Environment options 10
image configuration files 56
program database (.pdb) files 6

project properties 6
source control software 12
Strict Localization Checking In The Build option 8
subproject image settings 16
Build menu 41
build options 3
active OS design and 8
Buffer Tracked Events In RAM 9
Enable Eboot Space In Memory 9
Enable Event Tracking During Boot 9
Enable Hardware-Assisted Debugging Support 9
Enable Kernel Debugger 9, 168
Enable KITL 9, 168
Enable Profiling 9
Flush Tracked Events To Release Directory 9
Run-Time Image Can Be Larger Than 32 MB 9
Use Xcopy Instead Of Links To Populate Release Directory 10
Write Run-Time Image To Flash Memory 10
build phase 40
errors during the 66
build process 37, 39
advanced build commands 43
analyzing build results 63–67
batch files and 39
build log files 64
build phase 40
compile phase 40
Copy Files To Release Directory command 41
custom actions based on command-line tools 61
directives based on environment variables and 41
errors during the 63
make run-time image phase 41
phases during the 39
Platform Builder and 37
release copy phase 41
skipping the release copy phase 41
Software Development Kit (SDK) and 40
standard command prompt and 46
Sysgen phase 40
Visual Studio 41
build reports 63
Build tool (Build.exe) 57
Build.err file 63, 65
Build.log file 63–64
Build.wrn file 63, 65
Buildrel errors 66
BuiltIn registry key 261
bus drivers 249
Bus Enumerator (BusEnum) 261
bus name access 249
bus-agnostic drivers 300
BusEnum. *See* Bus Enumerator (BusEnum)
BusTransBusAddrToVirtual function 299

C

C interface 99
Call Stack tool 161
CAN. *See* Controller Area Network (CAN)
Catalog Editor 22
Error Report Generator 70
catalog entry properties 22
catalog files 21
Catalog Item Dependencies window 6
catalog items 3
.pbcxml files 21
3rdParty folder 22
adding or removing in an OS design 45
backlight driver 25
BSP development and 24
Clone Catalog Item option 19
cloning of 18–20
conditional processing based on 53
converting from the Public directory tree to a BSP component 20
creating and modifying 22
dependencies of 5, 24
East Asian languages 7
exporting of 24
ID of 23
Internet Explorer 6.0 Sample Browser catalog item 32
managing 21–25
properties of 22
Windows Embedded CE Standard Shell 97
Catalog Items View 4, 31
Clone Catalog Item option 19
display item dependencies in 53
filter items in 5
search for catalog items 6
Solution Explorer and 5
catalog system 21
CDEFINES directive 60
CDEFINES entry 24
CE 6.0 OS design template. *See* design templates
CE Dump File Reader 70, 161, 169
CE Stress tool 179
CE target control shell (CESH) 147
Ce.bib file 47, 56
CeAllocAsynchronousBuffer function 295
CeAllocDuplicateBuffer function 295
CeCallUserProc function 268
CeCloseCallerBuffer function 293, 295
CEDebugX. *See* debugger extension commands (CEDebugX)
CeFreeAsynchronousBuffer function 295

CeFreeDuplicateBuffer function 295
CeLog event-tracking system 116, 163
  reference naming matching and 166
  Remote Kernel Tracker tool and 164
  ship builds and 164
CeLogFlush tool 164
central processing unit (CPU) 119
CeOpenCallerBuffer function 293, 295
CESH. *See* CE target control shell (CESH)
CESysgen folder 12
CETest.exe. *See* workstation server application (CETest.exe)
CETK parser (Cetkpar.exe) 185
CETK. *See* Windows CE Test Kit (CETK)
Chain.bin file 51
Chain.lst file 51
classical naming convention for stream drivers 248
Clean Sysgen command 43
client-side application (Clientside.exe) 177, 180
  standalone mode 182
  start parameters 181
cloning components 18–20
  Board Support Package (BSP) and 201
    advanced debugger tools 202
  Clone Catalog Item option 19, 23
  public tree modification 18
Cloning Wizard 202
CLR. *See* Common Language Runtime (CLR)
code pages 8
code reuse 197
code sharing between the boot loader and the OAL 214
command processor shell 96
comma-separated values (CSV) 185
Common Language Runtime (CLR) 101
common release directory 37
Common.bib file 47
compile phase 40
compiler and linker (Nmake.exe) 57
compiler errors 63
component cloning 18
componentized operating system 91
components of a Board Support Package (BSP) 200
COMPRESSION parameter 49
conditional file processing 53
conditional statements and debugging 158
CONFIG section 49, 83
Config.bib file 83, 224, 270, 290
configuration files for Platform Builder 12, 199
Configuration Manager 6, 11
connectivity options 68
Console registry parameters 96
Consumer Media Device design template 4
context management 252
  device context 253
  open context 253
Control Panel 97
  components 98
  CPlApplet API 98
  messages 99
  NEWCPLINFO information 100
  Power applet 129
  Sources file and 100
Controller Area Network (CAN) 4
controlling the build process 41
Copy Files To Release Directory command 41
copylink 10
Core Connectivity (CoreCon) 17
  download layer for 69
  infrastructure for 68
  target control architecture and 148
  transport mechanisms and 70
core debugging tools 167
CoreCon. *See* Core Connectivity (CoreCon)
CPlApplet API 98
CPU Monitor 179
CPU. *See* central processing unit (CPU)
CPU-accessible memory 186
CPU-dependent user kernel data 223
CreateFile function 257
CreateInstance function 280
CreateMutex function 111
CreateProcess function 268
CreateSemaphore function 112
CreateStaticMapping function 224
CreateThread function 105
creating threads 105
Critical Off state 229, 232
Critical Section API 111
critical sections 84, 110
CSV. *See* comma-separated values (CSV)
custom build actions based on command-line tools 61
custom CETK tests 182
custom design templates 4

D

data integrity 55
database (.db) files 46, 54
DbgMsg feature. *See* debug message (DbgMsg) feature
DBGPARAM variable 152
DDI. *See* Device Driver Interface (DDI)
DDKPCIINFO structure 299
DDKReg_GetPciInfo function 299
DDKReg_GetWindowInfo function 299
DDKWINDOWINFO structure 299
DeactivateDevice function 258

deadlocks 115, 145, 159
Debug build configuration 6, 10
debug message (DbgMsg) feature 147
debug message options 150
debug message service 149, 155
debug zones 151
  best practices for 157
  bypassing of 152
  DBGPARAM variable 152
  definition of 154
  dialog box for 155
  dpCurSettings variable 156
  enabling and disabling of 155, 157
  overriding at startup 156
  registration of 152
  registry settings for 156
  SetDbgZone function 155
  Tux DLLs and 184
  Watch window and 155
debugger extension commands (CEDebugX) 159
debugger options 70
debugging 6, 145–196
  assembly language and 188
  Board Support Package (BSP) and 168
  boot loaders and 188
  breakpoints for 149
  CE Dump File Reader 161
  conditional statements and 158
  debug zones 151
  enabling 168–175
  essential components for 149
  excluding debugging code from release builds 158
  hardware-assisted 169
  hardware-debugging interface 70
  interrupt handlers and 173
  kernel debugger 70
  macros for debug messages 150
  postmortem debugger 70, 118
  retail macros for 150
  serial debug output functions 209
  target control commands 159
  Tux DLLs 184
  verbosity of 150, 158
DEBUGLED macro 151
DEBUGMSG macro 150
default locale 8
DefaultSuite parameter 181
defects on a target device 145
DEFFILE directive 61
delayed startup 95
  Svcstart sample for 95
demand paging 50, 82
demonstrate the features of a new development board 4
dependency handling 93
DependXX entry 93
deploying a run-time image 68–71
design
  advanced configurations 10
  build options 3
  catalog items 3
  environment variables 10
  file and directory structure 11
  internationalization 7
  language settings 7
  multiple platforms support 10
  operating system (OS) 1–35
  OS design overview 3
  redistribute 11
  subprojects 3
  template variants 4
  templates 4
design template variants 4
design templates 4
  Consumer Media Device 4
  custom 4
  Device Emulator: ARMV4I 29
  Enterprise Terminal 97
  PBCXML structures 4
  PDA Device 4, 29
  Small Footprint Device 4
  Thin Client 4
DestroyInstance function 280
development cycle 145
DEVFLAGS_LOADLIBRARY flag 83
device classes 135
device context 253
device driver 13
  application caller buffers and 291
  Board Support Package (BSP) and 201
  building a 254
  bus-agnostic 300
  context management 252
  developing a 241–308
  device register access 226
  DllMain function for a 243
  IClass value for a 264
  interface GUIDs 264
  interrupt handlers in a 272–282
  IOControl function in a 244
  kernel mode restrictions 268
  layered driver architecture 244
  legacy name for a 248
  load procedure for a 261
  loading and unloading 247, 258
  monolithic driver architecture 244
  naming conventions for 248

native drivers 243
paging and 243
portability of a 297
power management for a 283–287
power states 127
Reflector service and 268
registry entries for a 263
resource sharing between the OAL and a 226
shared memory region for communication 226
source code folders for a 217
Sources file directives for a 257
stream drivers 243
Device Driver Interface (DDI) 243
Device Emulator (DMA) 69
Device Emulator: ARMV4I 29
Device Manager 83
loading device drivers at boot time 261
overview of 247
Power Manager (PM.dll) 125
registry settings 93
shell of 247
stream driver interaction 244
device names 249
device register access 226
DeviceIoControl function 247, 292
DevicePowerNotify function 127, 133, 283
DHCP. *See* Dynamic Host Configuration Protocol (DHCP)
diagnose the overall health of the system 160
directives based on environment variables 41
directives for Sources files 61
Dirs files 57
DIRS keyword 57
DIRS_CE keyword 57
Disassembly tool 162
display item dependencies 53
DLL. *See* dynamic-link libraries (DLLs)
DLLENTRY directive 61
DllMain function 243
download methods 68, 186
download progress indication 212
dpCurSettings variable 156
Dr. Watson 118
driver globals (DRV_GLB) 207
driver power states 127
driver source code 205
DRIVER_GLOBALS structure 226
DriverDetect parameter 182
Drivers\BuiltIn registry key 261
DRV_GLB. *See* driver globals (DRV_GLB)
Dynamic Host Configuration Protocol (DHCP) 95, 187
dynamic management of debug messages 150
dynamic memory allocation 121
dynamically loading a driver 258
dynamically mapped virtual addresses 224
dynamic-link libraries (DLLs) 15
C interface and 99
device drivers and 243
DYNLINK directive 60

E

East Asian languages 7
Eboot boot loader 187
Eboot.bib file 206
elements
.NET Compact Framework 2.0 4
catalog items 3
Internet Explorer 4
OS design 3
WordPad 4
embedded operating systems
UNIX–based 103
embedded pointers 288, 292
Enable Eboot Space In Memory option 9
Enable Event Tracking During Boot option 9
Enable Hardware-Assisted Debugging Support option 9
Enable Kernel Debugger option 9
Enable Profiling option 9
enabling all debug zones 157
EnterCriticalSection function 110
Enterprise Terminal design template 4, 97
EnumDevices function 267
environment options 10
environment variables 10, 103
_TARGETPLATROOT 203
conditional statements based on 53
IMGNODEBUGGER 168
IMGNOKITL 168
WINCEDEBUG 150
Error List window 64
Error Report Generator catalog item 70
ERRORMSG macro 151
errors during the build process 63
Ethdbg boot loader 205
Ethernet download service 69
Ethernet support functions 210
Event API 114
event logging zones 163
event objects 114
event tracking 9
CeLog system and 116
exception handling 118–124
first chance 119
hardware 119
just in time (JIT) debugging 119

kernel debugger and 119
memory access and 296
minimizing the total number of committed memory pages through 121
postmortem debugger 118
raise exceptions 119
RaiseException function 119
second chance 119
syntax 120
termination handler 121
unhandled page faults 122
excluding debugging code from release builds 158
exclusion from a run-time image 17
eXDI. *See* Extended Debugging Interface (eXDI)
execute in place (XIP) 220
EXEENTRY directive 61
exiting threads 105
ExitThread function 106
export directive 99
exporting a catalog item from the catalog 24
exporting stream functions 255
Extended Debugging Interface (eXDI) 149
Extensible Data Interchange (XDI) 70
Extensible Markup Language (XML) 4
Extensible Resource Identifier (XRI) 70

F

Fast Interrupt (FIQ) line 276
file and directory structure of OS designs 11
file I/O operations 248
file system (.dat) files 46, 55
file system APIs 247
file type definitions for MODULES and FILES sections 52
files
.bib files 46
.dat files 46, 55
.db files 46, 54
.pbcxml files 21
.reg files 46, 54
.tks files 179
Bsp_cfg.h 276
Build.err 63
Build.log 63
Build.wrn 63
Ce.bib 47, 56
Chain.bin 51
Chain.lst 51
Common.bib 47
Config.bib 83, 224, 290
Device.dll 247
Devmgr.dll 247
Dirs files 57
Eboot.bib 206
Initdb.ini 56
Initobj.dat 55–56
Makefile file 61
Nk.bin file 41
Oalioctl.dll 226
Platform.bib 24, 51
Platform.dat 55
Platform.reg 54
Project.bib 47
Project.dat 55
Reginit.ini 56
shortcut files 94
Sources file 24, 59
Sysgen.bat 37
Udevice.exe 268
FILES section 51
Filesys.exe 55, 231
FileSystemPowerFunction 231–232
FIQ. *See* Fast Interrupt (FIQ) line
FIXUPVAR parameter 50
Flags registry value 266
flash memory support 211
Flush the X86 TLB on X86 systems 50
Flush Tracked Events To Release Directory option 9
FMerge tool (FMerge.exe) 67
FMerge.exe. *See* FMerge tool (FMerge.exe)
folder structure of a Board Support Package (BSP) 203
footprint of the operating system 1
ForceDuplicate parameter 293
frame buffers of peripheral devices 224
FreeIntChainHandler function 280–281
FreePhysMem function 226
FSRAMPERCENT parameter 50
full kernel mode 50

G

General Purpose Input/Output (GPIO) 90
general registry entries for device drivers 263
generic installable ISR (GIISR) 281
Getappverif_cetk.bat file 163
GetExitCodeThread function 106
GetProcAddress API 243
GIISR. *See* generic installable ISR (GIISR)
globally unique identifier (GUID) 264
GPIO. *See* General Purpose Input/Output (GPIO)
graphical user interface (GUI) 97
Graphics, Windowing, and Events Subsystem (GWES) 84, 92, 243
GUI. *See* graphical user interface (GUI)

GUID. *See* globally unique identifier (GUID)
GWES. *See*Graphics, Windowing, and Events Subsystem (GWES)
GwesPowerOffSystem function 230

H

H flag 271
HalTranslateBusAddress function 290
handles to system objects 103
hardware breakpoints 174
hardware conflicts 145
hardware debugger stub (HdStub) 148
hardware exceptions 119
hardware initialization tasks 209
hardware timer 83, 85
  OEMIdle function and 136
hardware validation 90
hardware-assisted debugging 169
hardware-debugging interface 70
hardware-independent code 201
HdStub. *See* hardware debugger stub (HdStub)
Heap Walker 147
heaps 84
heat dissipation 125
high-performance counters 85
honor the /base linker setting 50
HookInterrupt function 276
host process groups 270

I

I/O controls (IOCTLs) 132
I/O operations 248
IClass definitions 135
IClass value 264, 284–285
IDE. *See* integrated development environment (IDE)
Idle event 229
Idle mode 229
Idle power state 83
idle threads 87
IEEE. *See* Institute of Electrical and Electronic Engineers (IEEE)
IISR. *See* installable ISR (IISR)
ILTiming. *See* Interrupt Latency Timing (ILTiming) tool
image configuration files 56
IMGNODEBUGGER environment variable 168
IMGNOKITL environment variable 168
inactivity timeout 229
INCLUDES directive 60
InCradle 128
increase code re-usability 203
industrial control systems 101
infinite loops 147
INIT registry key 92
Initdb.ini file 56
initializing a device context 253
initializing virtual memory 213
Initobj.dat file 55–56
input/output operations 224
installable ISR (IISR) 280
  architecture 280
  DLL functions 280
  external dependencies and 281
  Plug and Play 280
  registering an 281
instance-specific resources 252
Institute of Electrical and Electronic Engineers (IEEE) 186
integrated development environment (IDE) 5
IntelliSense 63
interface GUIDs 264
Interlock API 115, 214
internal test applications 13
internationalization 7
  code pages 8
  default locale 8
  locales 7
Internet Explorer 4
  Sample Browser catalog item 32
  thin client shell and 97
inter-process communication 222
interrupt handlers
  architecture of 272
  breakpoints and 173
  communication between an ISR and an IST 279
  device drivers and 272–282
  WaitForMultipleObjects function in 276
interrupt latency timing 85, 90, 216
Interrupt Latency Timing (ILTiming) tool 85, 216
  parameters for the 86
interrupt mappings
  dynamic 277
  kernel arrays for 277
  shared 279
  static 276
Interrupt Service Routine (ISR) 216, 273–274
Interrupt Service Thread (IST) 216, 273–274
InterruptDone function 273
InterruptInitialize function 275
interrupts 272
  synchronization capabilities in the OAL 272
IOControl function 244, 280, 284
IOCTLs. *See* I/O controls (IOCTLs)
IP address configuration 95
IPv6 4

ISR latency 85, 216
ISR. *See* Interrupt Service Routine (ISR)
ISRHandler function 280
IST latency 85, 216
IST. *See* Interrupt Service Thread (IST)

J

JIT debugging. *See* just in time (JIT) debugging
joint test action group (JTAG) probe 169, 186
JTAG probe. *See* joint test action group (JTAG) probe
just in time (JIT) debugging 119, 149

K

K flag 270
Kato logging engine 182
Kato.exe. *See* test results logger (Kato.exe)
KdStub 70, 119, 148, 169
kernel access checks 293
kernel address space 220
kernel debugger 9, 149, 168–169
  application debugging and 149
  breaking into the 119
  exception handling and 119
  KdStub 70, 119, 148
  obtaining run-time information 148
kernel driver restrictions 268
Kernel Independent Transport Layer (KITL) 3
  boot arguments for 171
  communication interface 170
  enabling 9, 169
  methods of operation 170
  Remote Kernel Tracker tool 116
  support functions 216
  target control architecture and 148
  transport mechanisms 70
kernel initialization routines 187
kernel interrupt mapping arrays 277
kernel memory regions 221
kernel objects 84
  critical sections 110
  events 110
  interlocks 110
  mutexes 110
  semaphores 110
  thread synchronization and 110
kernel process (Nk.exe) 291
kernel profiler 9
kernel space 219
kernel startup support functions 214
Kernel Tracker 147
KERNELFIXUPS parameter 50
KernelIoControl function 226, 277
kernel-mode drivers 268
KernelStart function 214
keyboard events 272
kiosk mode 101
  managed applications and 101
  sample code 138
KITL. *See* Kernel Independent Transport Layer (KITL)

L

LAN. *See* Local Area Network (LAN)
language settings 7
last known good configuration 55
latencies 85
  ISRs and ISTs 85
LaunchXX entry 93
layered driver 245
  architecture 244
LDEFINES directive 60
legacy names 248
LIBRARY directive 60
linker warnings and errors 63
List Nearest Symbols tool 162
LoadDriver function 83, 243
LoadIntChainHandler function 273, 279–281
LoadKernelLibrary function 224
LoadLibrary function 83, 243
Local Area Network (LAN) 29
locale 6
Localize The Build option 8
localize the OS design 7
lower power consumption 125

M

macros for debug messages 150
  ASSERTMSG 151
  DBGPARAM variable 152
  debug zones and 151
  DEBUGLED 151
  DEBUGMSG 150
  ERRORMSG 151
  RETAILLED 151
  RETAILMSG 151
main thread of a process 104
MainMemoryEndAddress function 225
Make Binary Image tool (Makeimg.exe) 37, 47
make run-time image phase 41
  errors during the 67
Makefile file 61

Makeimg.exe. *See* Make Binary Image tool (Makeimg.exe)
managed applications
  kiosk mode and 101
  Windows Embedded CE Test Kit (CETK) and 178
managed code development 32
mapping tables 213
marshaling data across boundaries 288–296
marshaling helpers 293
MDD. *See* model device driver (MDD)
mechanical wear and tear 125
medical monitoring devices 101
memcpy 296
memory access 288
  asynchronous 293, 295
  exception handling and 296
  synchronous 294
memory layout 46
  kernel regions 221
  memory mapping of a BSP 219–227
  process regions 222
  reserved regions from system memory 291
memory leaks 145, 159
memory management
  ARM–based platforms 223
  critical sections and 84
  demand paging 82
  DEVFLAGS_LOADLIBRARY flag 83
  dynamic allocation 121
  dynamically mapped virtual addresses 224
  heaps 84
  LoadDriver function and 83
  LoadLibrary function 83
  MIPS–based platforms 223
  mutexes and 84
  noncontiguous physical memory and 225
  pre-committing memory pages 121
  processes and 84
  reuse of system memory 84
  ROMFLAGS option 83
  sharing of memory 82
  SHx–based platforms 223
  statically mapped virtual addresses 224
  system memory pool 84
  unhandled page faults 122
  virtual address space 103
  x86–based platforms 223
Memory Management Unit (MMU) 213, 223, 288
memory mappings 206
memory regions 221–222
MEMORY section 48
Memory tool 162
memory-mapped files 222
menu of a boot loader 211
Microprocessor without Interlocked Pipeline Stages (MIPS) 281
Microsoft kernel code 214
Microsoft Platform Builder for Windows Embedded CE 6.0 1, 37
  advanced debugger tools 161
  analyzing build results 63–67
  BSP Cloning Wizard 202
  Catalog Editor 22
  configuration files for 12
  Debug Message Options 150
  Debug Zones dialog box 155
  Heap Walker 147
  Kernel Tracker 147
  OS Design Wizard 3
  Process Viewer 147
  Software Development Kit (SDK) 26
  Subproject Wizard 14, 254
  Target Control option 158
  Target Device Connectivity Options dialog box 68, 169
Microsoft Visual Studio 2005 3
  Build menu 41
  building run-time images in 41–45
  Catalog Items View 4
  Configuration Manager 6
  connectivity options in 68
  debug information in the Output window of 149
  debugging a target device 171
  Error List window 64
  IntelliSense 63
  Open Build Window command 46
  Output window 64
  Solution Explorer 5
  Watch window 155
minimizing the total number of committed memory pages 121
MIPS. *See* Microprocessor without Interlocked Pipeline Stages (MIPS)
MIPS–based platforms 223
MmMapIoSpace function 226, 279, 290
MMU. *See* Memory Management Unit (MMU)
MmUnmapIoSpace function 290
model device driver (MDD) 18, 245
MODULES section 51
Modules tool 162
monolithic driver 245
  architecture 244
mouse tests 182
multi-bin image notification 212
multiple platforms support 10
multitasking 103
multithreaded operating system 103
multithreaded programming 115

mutexes 84, 111
CreateMutex function 111
deadlocks 115
Mutex API 112
ReleaseMutex function 112
TerminateThread function and 106
My Documents directory 55

N

naming conventions for drivers 248
native drivers 243
new system of managing virtual memory 219
NEWCPLINFO information 100
NK memory region 270
Nk.bin file 41
NKCallIntChain function 279
NKCreateStaticMapping function 224
NKDbgPrintf function 150
NKGLOBALS structure 214
Nmake.exe. *See* compiler and linker (Nmake.exe)
noise levels 125
NOLIBC=1 directive 281
non-cached virtual addresses 224
non-concurrent buffer access 294
noncontiguous memory 49
physical 225
non-real-time APIs 84
non-real-time components 82
NOTARGET directive 60

O

OAL. *See* OEM adaptation layer (OAL)
OALIntrRequestSysIntr function 277
OALIntrStaticTranslate function 277
Oalioctl.dll 226
OALPAtoVA function 279, 290
OALTimerIntrHandler function 85
object store 54
OEM adaptation layer (OAL) 3, 201
architecture-generic tasks 214
code sharing between the boot loader and the 214
interrupt management functions in the 275
interrupt-synchronization capabilities in the 272
IOCTL codes 226
OEMInit function 215
power management support and 228
Power Manager (PM.dll) 125
power state transitions and 228
Profile timer support functions 217
resource sharing between drivers and the 226
StartUp entry point of the 214
OEM address table 213
OEM. *See* Original Equipment Manufacturers (OEM)
OEMAddressTable table 213, 225
OEMEthGetFrame function 210
OEMEthGetSecs function 210
OEMEthSendFrame function 210
OEMGetExtensionDRAM function 225
OEMGLOBALS structure 214
OEMIdle function 135, 229
OEMInit function 215, 272
OEMInitGlobals function 214
OEMInterruptDisable function 275
OEMInterruptDone function 273, 275
OEMInterruptEnable function 275
OEMInterruptHandler function 275–276
OEMInterruptHandlerFIQ function 276
OEMIoControl function 226
OEMNMIHandler function 232
OEMPlatformInit routine 209
OEMPowerOff routine 230
OEMReadData function 210
OEMWriteDebugLED function 151
OHCI. *See* Open Host Controller Interface (OHCI)
Open Build Window command 46
open context 253
Open Host Controller Interface (OHCI) 277
OpenDeviceKey function 266
operating costs 125
operating system (OS)
advanced configurations 10
black shell 101
build options 3
command processor shell 96
componentized 91
creating and customizing 3–12
customization 5
dependency handling 93
design 1–35
Device Manager 83
elements 3
environment variables 10
footprint 1
internationalization 7
kernel objects 84
kiosk mode 101
language settings 7
managed code development 32
multithreaded 103
optimizing the performance 9
power management 83
processing model 103
real-time performance of the 88

redistribute and OS design 11
run-time image 1
shells 96–97
source code 18
standard shell 97
system applications 91
thin client shell 97
UNIX–based 103
Windows Task Manager (TaskMan) 97
Windows-based Terminal (WBT) shell 97
Operating System Benchmark (OSBench). *See* OSBench
OPTIONAL_DIRS keyword 57
Original Equipment Manufacturers (OEM) 197
OS Access (OsAxS) 148
OS Design Wizard 3, 5, 12, 29
Board Support Packages wizard page 10
multiple platforms support 10–11
standard shell and 97
OS design. *See* operating system (OS) design
OsAxS. *See* OS Access (OsAxS)
OSBench tool 85, 87
parameters for the 88
source code of 87
OutOfCradle 128
OUTPUT parameter 50
Output window 64
overriding debug zones at startup 156

P

page faults
unhandled 122
PAN. *See* Personal Area Network (PAN)
PBCXML. *See* Platform Builder Catalog XML (PBCXML)
PCI. *See* Peripheral Component Interconnect (PCI)
PCMCIA. *See* Personal Computer Memory Card International Association (PCMCIA)
PDA Device design template 4, 29
PDA. *See* personal digital assistant (PDA)
PDD. *See* platform device driver (PDD)
Pegasus registry key 157
performance monitoring 82–90
alerts for 90
charts for 90
interrupt latency timing 85, 90
reports for 90
waveform generators and 90
performance optimization 9, 82–90
PerfToCsv parser tool 185
Peripheral Component Interconnect (PCI) 241
persistent data storage 54
Personal Area Network (PAN) 29
Personal Computer Memory Card International Association (PCMCIA) 249
personal digital assistant (PDA) 228
physical memory access restrictions 290
physical memory allocation 290
Platform Builder Catalog XML (PBCXML) 4
Platform Builder. *See* Microsoft Platform Builder for Windows Embedded CE 6.0
Platform Builder–specific build commands 45
platform device driver (PDD) 245
Platform.bib file 24, 51
Platform.dat file 55
Platform.reg file 54
platform-specific source code 205
Plug and Play 247, 280
pNKEnumExtensionDRAM function 225
pointer marshaling 293
pointer parameters 292
portability of a device driver 297
PortNumber parameter 181
POSTLINK_PASS_CMD directive 61
postmortem debugger 70, 118
Power Control Panel applet 129
power management 83, 125–137
activity timers and 128
application interface 125, 133
benefits 125
context switching 83
device drivers and 283–287
device interface 126, 132
driver power states 127
I/O controls (IOCTLs) for 132, 284
Idle event 229
Idle power state 83
notification interface 125, 130, 285
OEM adaptation layer (OAL) 125, 228
optimizing power consumption by using dynamic timers 136
processor idle state 135
restrictions of 284
sample code 138
single-threaded mode and 284
system power states 127
wakeup sources and 232
Power Manager (PM.dll) 125
APIs 126
application interface 125, 133
architecture 125
battery life and 127
components 125
device drivers and 283
device interface 126, 132
notification interface 125, 130

power states
- activity timers and 128
- configuration of 134
- Critical Off state 229, 232
- device classes and 135
- driver 127
- InCradle 128
- internal transitions of 133
- OutOfCradle 128
- overriding the configuration for an individual device 134
- power-off state 229
- processor idle 135
- registry entries for 134
- Suspend state 229–230
- system 127
- transitions of 128, 228
- waking up from Suspend state 231

power-off state 229
PowerOffSystem function 126, 231
PQOAL. *See* Production Quality OEM adaptation layer (PQOAL)
pre-boot routines 186
pre-committing memory pages 121
preemptive multitasking 103
PRELINK_PASS_CMD directive 61
preprocessing conditions 54
primary thread of execution 103
priority list for threads 103
process address space 222
process identifier 103
Process Management API 104
Process Viewer 147
processes and threads 103
Processes tool 162
processing model 103
processor idle state 135
Production Quality OEM adaptation layer (PQOAL) 197
- advanced debugger tools 201

professional Windows Embedded CE solutions 21
PROFILE parameter 50
Profile timer support functions 217
program database (.pdb) files 6
PROGRAM directive 60
Program Files directory 55
Project.bib file 47
Project.dat file 55
Projsysgen.bat file 16
public source code 18
- modifications 19

public tree modification 18

Q

Q flag 271
QRimplicit-import 281
quality assurance 145
quantum 104
QueryPerformanceCounter function 88
QueryPerformanceFrequency function 88

R

race conditions 147
RaiseException function 119
raising exceptions 119
RAM file system 50, 55
RAM_AUTOSIZE parameter 50
RAM-backed map files 222
RAMIMAGE parameter 49, 225
RDEFINES directive 60
RDP. *See* Remote Desktop Protocol (RDP)
Readlog tool 165
real-time performance 82, 88
- measurement tools 84

real-time systems design 81
rebuild commands 42
Rebuild Current BSP And Subprojects 24
redistributing an OS design 11
reducing BSP development time 201
reference naming matching 166
Reflector service 268
Reginit.ini file 56, 266
RegisterDevice function 258
Registers tool 162
registry (.reg) files 46, 54
registry settings 92
- CELog registry parameters 163
- CELogFlush tool 165
- Clientside.exe start parameters 181
- command processor shell 96
- Console key 96
- CurrentControlSet\State key 134
- debug zones and 156
- DependXX entry 93
- device classes and 135
- device drivers and 263
- event logging zones and 163
- Flags registry value 266
- HKEY_LOCAL_MACHINE\Drivers\Active 262, 297
- HKEY_LOCAL_MACHINE\Drivers\BuiltIn 261, 297
- HKEY_LOCAL_MACHINE\INIT 92
- HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Interfaces 285
- interrupt-related 298

LaunchXX entry 93
memory-related 299
PCI-related 299
Pegasus registry key 157
power state definitions 134
startup parameters 92
subprojects and 16
Svcstart sample service 95
user mode driver host process (Udevice.exe) and 269
UserProcGroup registry entry 270
Reldir directory 12
Release build configuration 6, 10
release copy phase 41
errors during the 66
skipping the 41
release directory 41
ReleaseMutex function 112
ReleaseSemaphore function 113
RELEASETYPE directive 60
Remote Desktop Protocol (RDP) 4, 97
Remote Kernel Tracker tool 116, 164
Remote Performance Monitor 85, 88
extension DLLs 89
monitored objects 89
RequestDeviceNotifications function 267
RESERVED keyword 291
RESETVECTOR parameter 50
Resource Consume tool 179
resource sharing between drivers and the OAL 226
restrictions of power management 284
ResumeThread function 108
resuming threads 108
retail macros for debugging 150
RETAILLED macro 151
RETAILMSG macro 151
reuse of code 197
reuse of system memory 84
robustness of applications 118
ROM Image Builder tool (Romimage.exe) 47
ROM image file system 187
ROM Windows directory 55
ROM_AUTOSIZE parameter 50
ROM–based applications 55
ROMFLAGS option 83
ROMFLAGS parameter 50
Romimage.exe. *See* ROM Image Builder tool (Romimage.exe)
ROMOFFSET parameter 50
ROMSIZE parameter 50
ROMSTART parameter 50
ROMWIDTH parameter 50
RS232 connection 69
run-time image 1
adding custom settings to a 46
building and deploying 37–79
building and deploying from the command line 46
configuration files for the 56
content of a 46
deploying 68–71
download methods 186, 210
excluding a subproject from a 17
Run-Time Image Can Be Larger Than 32 MB option 9

S

S flag 271
sample code
CreateFile function 257
dynamic memory allocation technique 123
dynamically loading a driver 258
implementing stream functions 254
initializing a device context 253
Interrupt Service Thread (IST) 275
IOCTL_HAL_REQUEST_SYSINTR and IOCTL_HAL_RELEASE_SYSINTR 277
kiosk mode 138
non-concurrent buffer access 294
OEMAddressTable table 213
OEMPlatformInit function 210
power management 138
power notifications 131
thread execution 138
thread management 109
Sample Device Emulator eXDI2 Driver 70, 169
scheduling
quantum 104
thread scheduler 136
time-slice algorithm 104
SCM. *See* Service Control Manager (SCM)
SDK. *See* Software Development Kit (SDK)
search for catalog items 6
sections of .bib files 47
CONFIG section 49
MEMORY 48
security context 103
SEH. *See* structured exception handling (SEH)
semaphores 112–113
CreateSemaphore function 112
nonsignaled state 113
ReleaseSemaphore function 113
sequential access scenario 295
serial communication parameters 69
serial debug output functions 209
Serial Peripheral Interface (SPI) 252
ServerIP parameter 181
ServerName parameter 181

Service Control Manager (SCM) 95
services host process (Services.exe) 95
Services.exe. *See* services host process (Services.exe)
SetDbgZone function 155
SetSystemPowerState function 230
setting a breakpoint 172
shared memory region for driver communication 226
shells 96–97
  black shell 101
  command processor shell 96
  standard shell 97
  thin client shell 97
  Windows Task Manager (TaskMan) 97
  Windows-based Terminal (WBT) 97
ship builds 164
shortcut files 94
shortcuts on the desktop 55
SHx–based platforms 223
SignalStarted API 93, 101
Simple Windows Embedded CE DLL Subproject template 254
single-threaded mode 284
skeleton Tux module 183
SKIPBUILD directive 61
skipping the release copy phase 41
Sleep function 83, 108
SleepTillTick function 108
Small Footprint Device design template 4
small-footprint devices 82
software development cycle 145
Software Development Kit (SDK) 26–28
  adding new files 27
  build process and 40
  configuring and generating 26
  generating and testing 35
  installation of 28
software exceptions 119
software-related errors 147–167
Solution Explorer 5, 41
  Catalog Item Dependencies window 6
  Catalog Items View 5
  Dirs files and 59
  Property Pages dialog box 6
  Subproject Wizard 14
source code 18
  Control Panel 98
  driver globals 207
  drivers 205
  Eboot.bib file 206
  folders for device drivers 217
  Power Manager (PM.dll) 126
  thread management sample code 109
  Windows Task Manager (TaskMan) 97
source control software 12
SOURCELIBS directive 60
SOURCES directive 60
Sources file 24, 59
  ADEFINES directive 60
  CDEFINES directive 60
  CDEFINES entry 24
  Control Panel and 100
  DEFFILE directive 61
  DLLENTRY directive 61
  DYNLINK directive 60
  EXEENTRY directive 61
  INCLUDES directive 60
  LDEFINES directive 60
  LIBRARY directive 60
  NOTARGET directive 60
  POSTLINK_PASS_CMD directive 61
  PRELINK_PASS_CMD directive 61
  PROGRAM directive 60
  RDEFINES directive 60
  RELEASETYPE directive 60
  SKIPBUILD directive 61
  SOURCELIBS directive 60
  SOURCES directive 60
  TARGETLIBS directive 60
  TARGETNAME directive 60
  TARGETPATH directive 60
  TARGETTYPE directive 60
  WINCE_OVERRIDE_CFLAGS directive 61
  WINCECPU directive 61
  WINCETARGETFILE0 directive 61
  WINCETARGETFILES directive 61
Sources file directives for a device driver 257
SPI. *See* Serial Peripheral Interface (SPI)
SRE parameter 50
standalone mode 182
standard command prompt 46
standard directives for Sources files 61
standard shell 97
  removing the 101
Start menu 55
startup configuration 91
  delayed startup and 95
StartUp entry point of a boot loader 208
Startup folder 94
  restrictions 94
StartUp function 187
startup registry parameters 92
startup time
  reducing 4
StartupProcessFolder function 94
starvation 161
static libraries 15

static mapping regions of the kernel 223
statically mapped virtual addresses 224
Storage Device Block Driver Benchmark Test 180
storage partitioning routines 187
stream driver. *See also* stream interface driver
stream drivers 243
 context management and 252
 CreateFile function and 257
 device names for 249
 exporting stream functions 255
 instance-specific resources 252
 kernel mode restrictions for 268
 legacy names for 248
 load procedure for 261
 loading and unloading 247, 258
 naming conventions for 248
 Plug and Play 247
 Sources file directives for 257
 XXX prefix and 252
stream interface API 250
 exporting stream functions 255
stream interface driver 247
Strict Localization Checking In The Build option 8
structured exception handling (SEH) 119
 __except keyword 120
 __finally keyword 121
 __try keyword 120
 frame–based 120
subprojects 3
 configurations files 14
 configuring 13, 16–17
 creating and adding 14
 Dirs files and 57
 dynamic-link libraries (DLLs) and 15
 excluding from a run-time image 17
 image settings 16
 Projsysgen.bat file 16
 registry settings and 16
 reusing customizations and 47
 static libraries and 15
 Subproject Wizard 14
 SYSGEN variables and 16
 TARGETTYPE=NOTARGET 16
 types of 13
 without source code 16
Suspend state 229–230
suspending threads 108
SuspendThread function 108
Svcstart sample service 95
 registry parameters for 95
symbols 162
synchronization
 deadlocks 115
 thread 103
 unintentional 157
synchronous memory access 294
syntax check for source code 63
Sysgen capture tool 19
Sysgen phase 40
 errors during the 65
SYSGEN variables 10
 conditional statements based on 53
 subprojects and 16
Sysgen.bat 37
SYSINTR value 273, 276
SYSINTR_NOP value 274
SYSINTR_TIMING interrupt event 85
system applications 91
system memory mapping 219
system memory pool 84
system performance
 monitoring 82–90
 optimization 82–90
 real-time operating systems and 82
system power states 127
system programming 81–144
system scheduler 83
system testing 145, 176
system timer 83

T

target control architecture 148
target control commands 159
Target Control service 158
target control shell. *See* CE target control shell (CESH)
target device
 attaching to 71
 debugger options 70
 defining communication parameters for a 68
 initialize the file system and the system registry 46
 loading Windows Embedded CE on a 68
Target Device Connectivity Options dialog box 68, 169
target device control 147
TARGETLIBS directive 60
TARGETNAME directive 60
TARGETPATH directive 60
TARGETTYPE directive 60
TARGETTYPE=NOTARGET 16
TCP/IPv6 Support 29
template variants 4
Terminal server 97
TerminateThread function 106
terminating threads 105
termination handler 121

test access port and boundary-scanning technology 186
test engine (Tux.exe) 177
  command-line parameters 182
Test Kit Suite (.tks) files 179
test results logger (Kato.exe) 177
test suite 179
testing a system 145–196
  automated 176
TFTP. *See* Trivial File Transfer Protocol (TFTP)
Thin Client design template 4
thin client shell 97
thread 83
  creating 105
  deadlocks 115
  definition 103
  exiting 105
  idle 87
  main thread of a process 104
  management functions 105
  maximum number of 103
  primary of execution 103
  priority 103, 107
  priority levels 107
  quantum 104
  resuming 108
  sample code 138
  scheduler 136
  scheduling 103
  starvation 161
  suspending 108
  synchronization 103, 110
  terminating 105
  time-slice algorithm 104
  troubleshooting synchronization 115
  unintentional synchronization 157
  worker threads 104
Thread Management API 104
thread priority 83
thread synchronization
  interrupt handling and 272
  unintentional 157
Threads tool 162
tick timer 85
timer events 272
timers
  hardware timer 83
  OALTimerIntrHandler function 85
  power management and 128
  SleepTillTick function 108
  SYSINTR_TIMING interrupt event 85
  system timer 83
time-slice algorithm 104
TLB. *See* Transition Lookaside Buffer (TLB)
tools
  advanced debugger tools 161
  Advanced Memory tool 162
  Application Verifier tool 162, 179
  Autos tool 161
  Breakpoints 161
  Build.exe 57
  Call Stack tool 161
  CE Stress tool 179
  CELogFlush tool 164
  CETest.exe 177
  Cetkpar.exe 185
  Clientside.exe 177, 180
  Control Panel 97
  CPU Monitor 179
  debugging and testing 145
  Disassembly tool 162
  Dr. Watson 118
  Filesys.exe 55
  FMerge (FMerge.exe) 67
  Heap Walker 147
  ILTiming 85, 216
  Kato.exe 177
  Kernel Tracker 147
  List Nearest Symbols tool 162
  Make Binary Image (Makeimg.exe) 37, 47
  Memory tool 162
  Modules tool 162
  Nmake.exe 57
  OSBench 85
  PerfToCsv parser 185
  Power Control Panel applet 129
  Process Viewer 147
  Processes tool 162
  Readlog tool 165
  real-time performance measurement 84
  Registers tool 162
  Remote Kernel Tracker 116, 164
  Remote Performance Monitor 85, 88
  Resource Consume tool 179
  ROM Image Builder (Romimage.exe) 47
  Sysgen capture tool 19
  Sysgen.bat 37
  Threads tool 162
  Tux.exe 177
  Watch window 161
  Windows Task Manager (TaskMan) 97
transaction–based storage mechanism 54
TransBusAddrToVirtual function 290
Transition Lookaside Buffer (TLB) 214
transport mechanisms 68, 70
trap handler 272
Trivial File Transfer Protocol (TFTP) 187

troubleshooting
  build issues 65
  thread synchronization 115
Trust only ROM modules 50
trusted images 212
TryEnterCriticalSection function 110
TUX DLL template 183
Tux.exe. *See* test engine (Tux.exe)

U

UART. *See* Universal Asynchronous Receiver/Transmitter (UART)
Udevice.exe. *See* user mode driver host process (Udevice.exe)
UDP. *See* User Datagram Protocol (UDP)
uninitialized variables 147
unintentional thread synchronization 157
Universal Asynchronous Receiver/Transmitter (UART) 186
Universal Serial Bus (USB) 69
UNIX–based embedded operating systems 103
unrecoverable system lockup 284
USB. *See* Universal Serial Bus (USB)
Use Xcopy Instead Of Links To Populate Release Directory option 10
user applications 91
  Terminal server and 97
User Datagram Protocol (UDP) 187
user mode driver host process (Udevice.exe) 268
  application caller buffers and 291
  registry entries for 269
user space 219
User-Defined Test Wizard 183
user-mode drivers 268
UserProcGroup registry entry 270

V

validate a system in its final configuration 145
video memory 232
virtual address space 103
  dynamically mapped addresses 224
  frame buffers of peripheral devices and 224
  input/output operations and 224
  kernel space 219
  non-cached 224
  noncontiguous physical memory and 225
  statically mapped addresses 224
  user space 219
virtual memory
  initializing 213
  mapping tables 213
  new system of 219
Virtual Memory Manager (VMM) 288
VirtualAlloc function 122, 224, 279, 296
VirtualCopy function 224, 279
VirtualFree function 224
virtual-to-physical address mappings 213
Visual Studio 2005. *See* Microsoft Visual Studio 2005
VMM. *See* Virtual Memory Manager (VMM)
vulnerabilities 293

W

WaitForMultipleObjects function 108, 276
WaitForSingleObject function 108, 274
wakeup sources 232
waking up from Suspend state 231
Watch window 155, 161
waveform generator 90
WCE TUX DLL template 183
Win32 API 84
WINCE_OVERRIDE_CFLAGS directive 61
WINCECPU directive 61
WINCEDEBUG environment variable 150
WINCETARGETFILE0 directive 61
WINCETARGETFILES directive 61
window drawing 84
Windows directory 55
Windows Embedded CE custom test components for the Microsoft Windows CE Test Kit (CETK) 14
Windows Embedded CE shells 96–97
Windows Embedded CE Standard Shell 97
Windows Embedded CE Subproject Wizard 14, 254
Windows Embedded CE Test Kit (CETK) 176–185
  analyzing test results 184
  Application Verifier tool 162
  architecture of 176
  CETK parser (Cetkpar.exe) 185
  client-side application (Clientside.exe) 177
  command-line parameters for 180
  custom tests based on 179–180, 182
  managed code and 178
  overview of 176
  PerfToCsv parser tool 185
  skeleton Tux module 183
  standalone mode 182
  test engine (Tux.exe) 177
  Test Kit Suite (.tks) files 179
  test results logger (Kato.exe) 177
  Test Suite Editor 179
  User-Defined Test Wizard 183
  workstation server application (CETest.exe) 178
  zorch parameter 180
Windows Manager 232

Windows Network Projector 4
Windows Sockets (Winsock) 177
Windows Task Manager (TaskMan) 97
Windows Thin Client 4
Windows-based Terminal (WBT) shell 97
Winsock. *See* Windows Sockets (Winsock)
WMV/MPEG-4 Video Codec 4
WordPad 4
worker threads 104
workstation server application (CETest.exe) 177–178
Write Run-Time Image To Flash Memory option 10
WriteDebugLED function 151

X

x86–based platforms 223
X86BOOT parameter 51
XDI. *See* Extensible Data Interchange (XDI)
XIP chain 51
XIP. *See* execute in place (XIP)
XIPSCHAIN parameter 51
XML. *See* Extensible Markup Language (XML)
XRI. *See* Extensible Resource Identifier (XRI)
XXX prefix 252
XXX_Init function 266
XXX_IOControl function 283, 292
XXX_PowerDown function 283
XXX_PowerUp function 283

Z

zone definitions 154
zones registration 152
zorch parameter 180
297

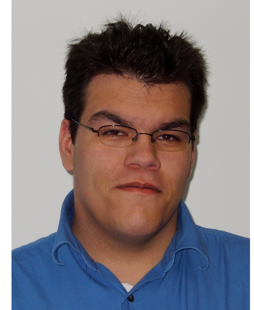
About the Authors

Nicolas Besson

Nicolas Besson has over seven years of in-depth technical experience with Windows Embedded CE technologies. He currently specializes in Software Development and Project Management at Adeneo, a key Microsoft Gold Embedded Partner with worldwide presence that focuses on Windows Embedded CE technologies. Nicolas has been a Microsoft eMVP for the last two years. He shares his knowledge by providing training for companies and people around the world. You can read more about his passion for Windows Embedded CE technologies on his blog at http://nicolasbesson.blogspot.com.

Ray Marcilla

Ray Marcilla is an Embedded Software Developer at Adeneo's American branch in Bellevue, Washington. Ray has significant experience in application development with native and managed code, as well as CE driver development. He also participates in CE-related technical presentations and workshops. Ray has worked on a number of interesting projects for ARM and x86 development platforms. In his free time, he likes to study foreign languages; he's currently fluent in Japanese, and can speak some Korean and French.



Rajesh Kakde

Rajesh has been associated with Windows Embedded CE since 2001. He has worked in various parts of the world in different segments of the industry including Consumer Electronics & Industrial Real-time Devices. He has extensive experience in BSP and driver development, as well as application development.

Currently, he is part of the Adeneo Corp. team as a Senior Windows Embedded Consultant and provides technical expertise on BSPs and drivers, as well as management for various OEM projects. He also delivers Windows Embedded CE workshops and trainings with Adeneo, which allow him to exchange ideas and share his passion for this technology.