

Building Solutions Using Windows Embedded CE 6.0 R2



Welcome to the “Building Solutions Using Windows Embedded CE 6.0 R2” training course.

Course Outline

- **Course Introduction**
- **Module 1:** Operating System Overview
- **Module 2:** Tools for Platform Development
- **Module 3:** Operating System Internals
- **Module 4:** Operating System Components
- **Module 5:** The Build System
- **Module 6:** The Board Support Package
- **Module 7:** Device Driver Concepts
- **Module 8:** Customizing the OS Design
- **Module 9:** Application Development
- **Module 10:** Testing & Verification



Course Introduction

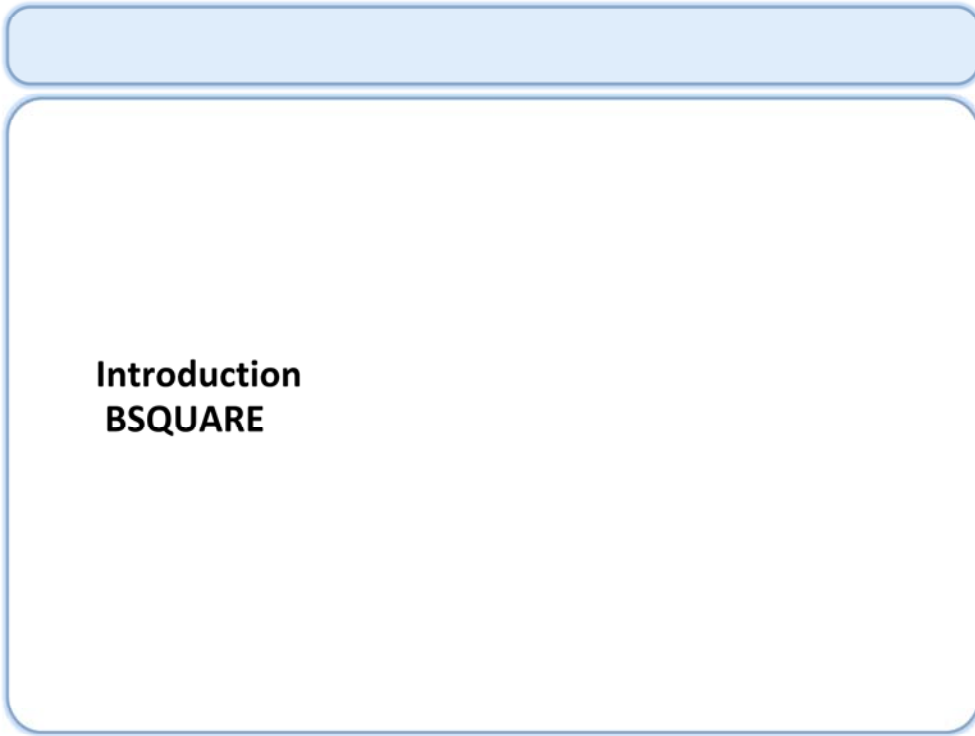
- **Welcome**
- **Microsoft's Embedded OS Offerings**
- **Who & What is This Course Targeted At?**
- **Review**



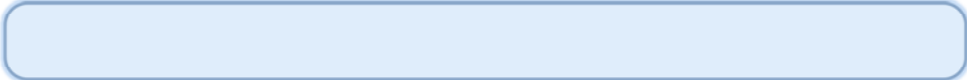
Course Introduction

- **Welcome**
- Microsoft's Embedded OS Offerings
- Who & What At?
- **Review**
 - Facilities
 - Building hours
 - Class hours
 - Restrooms
 - Parking
 - Web access
 - Meals/refreshments
 - Smoking
 - Recycling





Introduction
BSQUARE



What We Do

**BSQUARE is a smart device solutions provider
for leading OEMs, ODMs, silicon vendors and
peripheral vendors**



Our Background

- **Founded in 1994 to provide compiler & tool chain engineering services for Microsoft WinCE Team**
- **Went public (BSQR) in 1999 and continued to evolve products and services as new opportunities emerged**
- **Delivered hundreds of successful SI engagements**



- Window Embedded CE
- Windows Embedded Standard
- Targeted Platforms
- Microsoft Auto 4
- Standard (Smartphone)
- Professional (PPC Phone Edition)
- Classic (Pocket PC)
- Portable Media Center

How We Do It

Consulting & Engineering Services

- Leading systems integrator for Windows CE, Windows XPe, Windows Mobile for Smartphone and PocketPC, Windows Embedded for Point of Service (WEPOS)

Third Party Products

- Largest Value-Added Provider for Microsoft Windows Embedded & Classic OS toolkits and licenses

BSQUARE Products

- SDIO Hx Solutions
- Device Validation TestSuite™ for OMAP™ 3 BSP
- Flash UI Extender for OMAP35x
- OMAP35x Wireless DevKit
- SchemaBSP Development Tools
- TQ Countdown

Consulting & Engineering Services

- **Product Planning, Requirements, Architecture and Design Consulting**
- **Platform Software Engineering**
- **Application and Middleware Engineering**
- **Hardware Engineering**
- **QA, Testing and Pre-Certification Quality Assurance**
- **Training – classroom, custom and onsite**
- **Technical Support**

Highly customized, critical-path consulting and engineering services

Third Party Products

- **Purchasing and Manufacturing**
 - Just-in-time programs
 - Virtual warehouse
 - Shipping worldwide
 - Microsoft liaison
- **Marketing**
 - Joint marketing with Microsoft
- **Engineering**
 - Toolkits
- **Support**

BSQUARE Products

- **SDIO Hx**
 - Multi-slot, High-Performance SDIO Software Stack
- **Device Validation TestSuite™ for OMAP™ 3 BSP**
 - Comprehensive set of QA tests for an OMAP™ 3 & WinCE 6.0 R2 design
- **Flash UI Extender**
 - Middleware software package that sits between the Flash Player and platform and provides a seamless connection between Flashlite's Actionscript and machine/OS subsystem.
- **OMAP35x Wireless DevKit**
 - Complete 3G Wireless DevKit for the OMAP 35x Reference Platform
- **SchemaBSP Development Tool**
 - Software development tool for creating board support packages
- **TQ Countdown**
 - Provides the test and application development, controls, and management for test automation.

SDIO Hx –

- Only SDIO software stack offering multiple-slot support with a single host controller chip decreasing device size and cost.
- Supports 802.11g Wi-Fi throughput rates to 15 Mbps on select applications processors. Ideal for GPS, streaming media, VoIP, and other applications requiring high data throughput rates.
- Accelerates time to market leveraging BSQUARE's industry-standard SDIO software stack including support for SD, SDIO and MMC specifications.

Welcome (continued)

- **Introductions**

- Name
 - Company affiliation
 - Title/function
 - Job responsibility
 - Programming experience
 - Previous Windows Embedded CE experience
 - Expectations for the course
-

Welcome (continued)

- **Course Materials**
 - Student workbook
 - Lab manual
 - Electronic version of materials
 - Web-based course evaluation

Welcome (continued)

- **Hardware Setup**
 - Development workstation
 - Visual Studio 2005
 - Windows Embedded CE 6.0 R2
 - Reference Board

Course Introduction

- Welcome
- **Microsoft's Embedded OS Offerings**
- Who & What is This Course Targeted At?
- Review

- .NET Micro Framework
- Windows Embedded CE & SKUs
- Windows Mobile & SKUs
- Microsoft Auto
- Windows Embedded Standard
- Windows Embedded for Point of Service
- Previous versions



Windows Embedded CE SKUs include Core, Professional, Windows Embedded CE 6.0 for Handheld GPS, and Windows Embedded CE 6.0 for Set Top Box.

Windows Mobile and Windows Automotive are based on Windows Embedded CE.

Windows XP Embedded, Windows Embedded for Point of Service, Windows Vista Business for Embedded Systems, and Windows Vista Ultimate for Embedded Systems are based on desktop operating systems.

Generally the Microsoft offerings are viewed in terms of more or less functionality. This must be kept in perspective of the needs of the device running the OS.

(e.g. If the needs dictate an ARM processor, running real-time, with less than 16MB of memory, then you can see that Windows Embedded CE is really the OS with the features required.)

Previous versions include the many versions of Windows Embedded CE and the predecessor to XP Embedded; NT Embedded.

<http://www.microsoft.com/embedded>

Microsoft's Embedded OS Offerings (continued)

- **.NET Micro Framework**
 - Leverages .NET technology and the Visual Studio toolset targeting less expensive and more power efficient 32-bit processors
 - Supports managed code, C# development, including device drivers
 - SDK includes extensible emulator
 - Supports a number of ARM7 and ARM9 processors
 - Targeted at devices such as sensor nodes, auxiliary displays, health monitors, remote controls, and robotics

.NET Micro Framework landing page:
<http://msdn2.microsoft.com/en-us/embedded/bb267253.aspx>

The .NET Micro Framework can have a footprint as little as 250-500KB with support for managed code.

In addition to the bullets listed here a couple other things that could be noted include the fact that the .NET Micro Framework does not require an MMU, it is not designed to be real-time, and that native code is only supported through interop.

Microsoft's Embedded OS Offerings (continued)

- **Windows Embedded CE Based**
 - Hard real-time support
 - Support for X86, MIPS, ARM, SH4 (requires MMU)
 - Supports managed and native code application development
 - Targeted at devices such as VoIP phones, set top boxes, PDAs, GPS, and industrial controllers
 - Windows Mobile
 - Microsoft Auto 4

Windows Embedded CE landing page:
<http://msdn2.microsoft.com/en-us/embedded/aa731407.aspx>

Microsoft's Embedded OS Offerings (continued)

- **Embedded Version of Desktop Operating Systems**
 - X86 Support; typically greater than 40MB footprint; >10K components
 - Typically not as power consumption sensitive
 - PC class performance with embedded features; built with same bits as desktop OS
 - Targeted at devices such as thin clients, ATMs, and kiosks
 - Includes:
 - Windows Embedded Standard
 - Windows Embedded for Point of Service

Windows Embedded for Point of Service provides Plug-n-Play functionality for many retail device peripherals. It is built on top of XP Embedded and provides a quicker time to market for devices used in point of service applications.

Windows Embedded Standard landing page:
<http://msdn2.microsoft.com/en-us/embedded/aa731409.aspx>

Windows Embedded for Point of Service:
<http://msdn2.microsoft.com/en-us/embedded/aa714298.aspx>

Course Introduction

- Welcome
- Microsoft's Embedded OS Offerings
- **Who & What is This Course Targeted At?**
- Review

- **Who?**

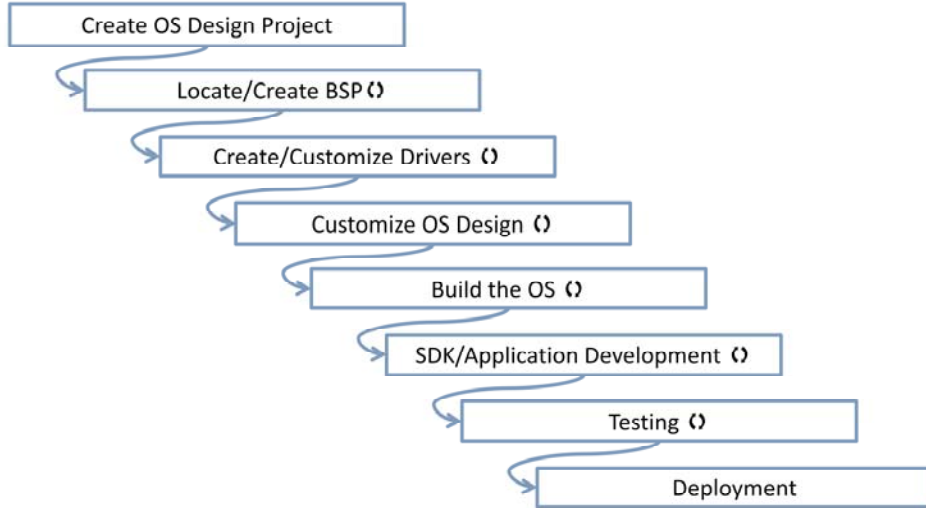
- Those that will be involved in the creation and customization of the OS image for a device.
 - BSP Developer
 - OS Builder
 - Device Driver Developer
 - QA and Test



Who & What is This Course Targeted At? (continued)

- **What?**

- The process of creating a product OS image



Who & What is This Course Targeted At? (continued)

- **Helpful Prerequisites**

- Programming experience in C or C++
- Some knowledge of Windows operating systems internals
- Win32 API programming experience
- Device driver development experience
- Embedded operating systems experience

Course Introduction

- Welcome
- Microsoft's Embedded OS Offerings
- Who & What is This Course Targeted At?
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Operating System Overview

Course Outline

- Course Introduction
- **Module 1: Operating System Overview**
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification



Operating System Overview

- **Characteristics of Windows Embedded CE**
- **History of Windows Embedded CE**
- **What's New in CE 6.0 R2?**
- **Review**



Operating System Overview

- **Characteristics of Windows Embedded CE**
 - **History of Windows Embedded CE**
 - **What's New in Windows Embedded CE 6.0**
 - **Review**
- **32 Bit, Scalable, Componentized, Real-time**
 - **Hundreds of Features**
 - **Variety of Processor Architectures Supported**
 - **Familiar & Rich Tool Chain**
 - **Source Access**
 - **Wide Support and Ecosystem**
 - **Value**



Windows Embedded CE is designed for small footprint embedded devices, and is designed to allow customers to get to the market quickly.

Characteristics of Windows Embedded CE (continued)

- **32 Bit, Scalable, Componentized, Real-time**
 - Preemptive multitasking
 - Virtual protected memory model
 - Multiple processes (PE Format EXEs)
 - Multiple threads
 - 256 priorities
 - Based on Win32

Real-time performance:

<http://msdn2.microsoft.com/en-us/library/aa908633.aspx>

Since the executables use PE format, tools that operate on PE files, such as Dumpbin and Depends will work on Windows Embedded CE executable files.

Characteristics of Windows Embedded CE (continued)

- **Hundreds of Features**
 - <http://www.microsoft.com/windows/embedded/eval/wince/components.mspx>
 - End user applications
 - Applications and services
 - Communication services and networking
 - Core OS services
 - Device management
 - File systems and data store
 - Fonts
 - Graphics and multimedia technologies
 - Media
 - International
 - Internet client services
 - Security
 - Shell and user interface
 - Voice over IP phone services
 - Windows Embedded CE error reporting

Characteristics of Windows Embedded CE (continued)

- **Variety of Processor Architectures Supported**
 - ARM, x86, MIPS, SH4
 - MMU required
 - Each with sample BSPs
 - Kernel source part of source code program

Supported processors:

<http://msdn2.microsoft.com/en-us/embedded/aa714536.aspx>

With the support for ARM, SH4, MIPS, and x86 architectures, the total number of processors supported is quite high; some with direct support out of the box, others with support from chip manufacturers or 3rd parties and still some that would require significant development.

Characteristics of Windows Embedded CE (continued)

- **Familiar & Rich Tool Chain**
 - Visual Studio, Remote Tools, Emulator, Test tools
 - Windows Embedded CE is a plug-in for Visual Studio
 - Emulator
 - Remote tools for performance tuning, debugging, and information management
 - Extensible test framework

Visual Studio landing page:
<http://msdn2.microsoft.com/en-us/vstudio/default.aspx>

Characteristics of Windows Embedded CE (continued)

- **Source Access**
 - Shared source program
 - “In the Box”
 - Premium shared source program
 - Available to those qualified through secure website
 - Community source projects
 - USB Webcam, Wi-Fi Driver, ...

Access to Windows Embedded CE source code helps developers debug, test, and make changes to an OS image. It also allows you to modify the operating system software to create differentiated features while maintaining control over your intellectual property.

Info on Shared Source:

<http://msdn2.microsoft.com/en-us/embedded/aa714518.aspx>

Community Source Projects include:

- Open SSH
- MPEG-2 Demux
- Wi-Fi Driver
- Phidgets
- Bluetooth Wrapper
- Layered Services Providers
- Gumstix BSP
- USB Webcam

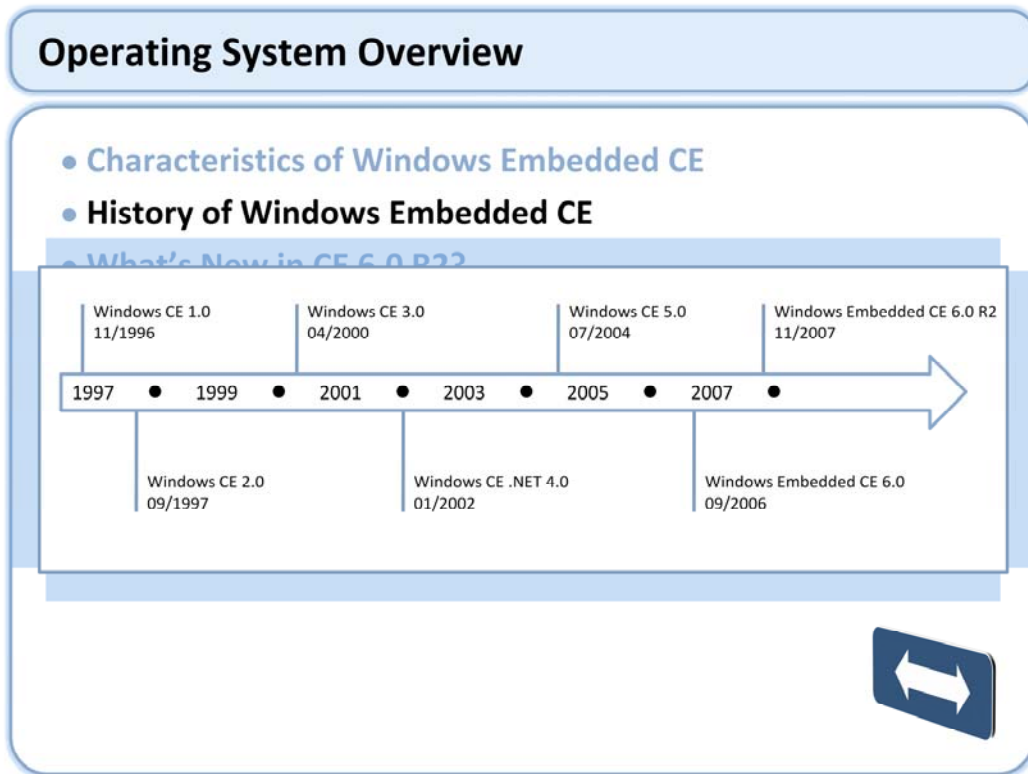
Links for these projects can be found through the shared source info link above.

Characteristics of Windows Embedded CE (continued)

- **Wide Support and Ecosystem**
 - MSDN
 - Blogs
 - Other resources

Characteristics of Windows Embedded CE (continued)

- **Value**
 - Features vs. pricing
 - Entry costs
 - Maintenance costs



Here is a link to a detailed visual timeline:

http://upload.wikimedia.org/wikipedia/commons/c/cb/Windows_CE_Timeline.png

What does CE stand for:

<http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q166915>

Windows Embedded CE started as a command line build only product and in mid 2.x releases was provided a GUI build environment all its own. For version 3.0 there was a major kernel rewrite from the ground up to improve real-time performance. Version 4.x saw the integration of the compact framework as well as many build environment improvements. Version 6.0 was the first version that saw the integration of the build environment into Visual Studio as well as major changes to the memory architecture to provide applications with increased memory space.

Operating System Features

- Characteristics of Windows Embedded CE
- History of Windows Embedded CE
- What's New in CE 6.0 R2?
- Remote Desktop Protocol (RDP) 6.0
- Updates for Internet Explorer 6
- New Sample Board Support Packages
- Web Services on Devices API (WSDAPI)
- USB CCID Smart Card Reader Class Driver
- Windows Media Player OCX 7.0
- Voice over IP (VoIP) Phone Services
- File System Updates
- Pluggable Font Support
- Check out the Release Notes!



Support for Remote Desktop Protocol (RDP) 6.0. RDP 6.0 includes support for Secure Sockets Layer/Transport Layer Security (SSL/TLS), Network Level Authentication, Server Authentication, and 32-bit color graphics. Support for Microsoft Web Services on Devices (WSDAPI), which is an unmanaged code implementation of the Devices Profile for Web Services (DPWS) protocol standard.

Support for Video over IP telephony calls.

Additional Voice over IP (VoIP) functionality, including a VoIP boot loader application and resources for QVGA landscape mode and QVGA portrait mode user interfaces.

Support for the Pocket Outlook Object Model (POOM) and ActiveSync in the VoIP Home Screen and VoIP Contacts applications.

New sample board support packages (BSPs).

Support for Auto Proxy Configuration Support in Internet Explorer 6 for Windows Embedded CE.

New driver that supports USB CCID Smart Card readers.

Support for Windows Media Player OLE Control Extension (OCX) 7.

New componentized flash driver and new partition driver for the management of flash memory.

Improved Secure Digital (SD) bus driver that supports SDHC specification 2.00 functionality, for example Secure Digital High-Capacity (SDHC) cards.

Sample Serial ATA driver, extended from the ATAPI driver, which supports the Promise PDC40518 SATA card.

Support for pluggable third-party font drivers.

Support for Extended File Allocation Table (ExFAT) and FAT32 on the x86 BIOS Loader, which provides access beyond 2 gigabytes (GB) of hard disk space.

Operating System Features

- Characteristics of Windows Embedded CE
- History of Windows Embedded CE
- What's New in CE 6.0 R2?
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Tools for Platform Development

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- **Module 2: Tools for Platform Development**
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification




Tools for Platform Development

- **Visual Studio 2005 & CE 6.0 R2 Installation**
- **Windows Embedded CE Terminology**
- **A Look at the IDE**
- **Lab 2.1 – Our First OS**
- **Introduction to the Build Process**
- **Lab 2.2 – Develop & Test an Application Subproject**
- **Testing and Debugging the OS Design**
- **Lab 2.3 – Using the Remote Tools**
- **Windows Embedded CE Directory Structure**
- **Review**



Tools for Platform Development

- **Visual Studio 2005 & CE 6.0 R2 Installation**
 - Windows Embedded CE Terminology
 - A L
 - Lab
 - Intr
 - Lab
 - Tes
 - Lab
 - Win
 - Rev
- **Installation Order**
 - Visual Studio 2005
 - Windows Embedded CE 6.0
 - Visual Studio 2005 SP1
 - Visual Studio 2005 SP1 Update for Vista (Vista only)
 - Windows Embedded CE 6.0 SP1
 - Windows Embedded CE 6.0 R2
 - CE 6.0 R2 contains QFEs through end of Aug 2007
 - Can I install Windows CE 5.0 on the same machine?
 - Windows XP vs. Windows Vista
- 

Starting with the initial release of Windows Embedded CE 6.0, the build tools are run from within the Visual Studio 2005 environment.

Installation should be completed in the order above. Note that R2 contains all updates through August 2007; any updates after the August 2007 updates will need to be installed following this installation.

<http://blogs.msdn.com/dcook/archive/2007/05/13/does-pb-5-0-work-side-by-side-with-pb-6-0.aspx>

Tools for Platform Development

- Visual Studio 2005 & CE 6.0 R2 Installation

- Windows Embedded CE Terminology

- A Look at the IDE

Term	Definition
Catalog	A container of individually selectable units of Windows Embedded CE functionality.
Catalog item	Any item that you can select from the Catalog.
OS Design	A selection of Catalog items that defines the characteristics of an OS. You can begin an OS Design with or without a design template. An OS Design corresponds to a set of Sysgen variables.
Component	The smallest unit of functionality that you can add to an OS Design.

- Review



Link to terminology in Windows Embedded CE:

<http://msdn2.microsoft.com/en-us/library/aa924102.aspx>

Windows Embedded CE Terminology (continued)

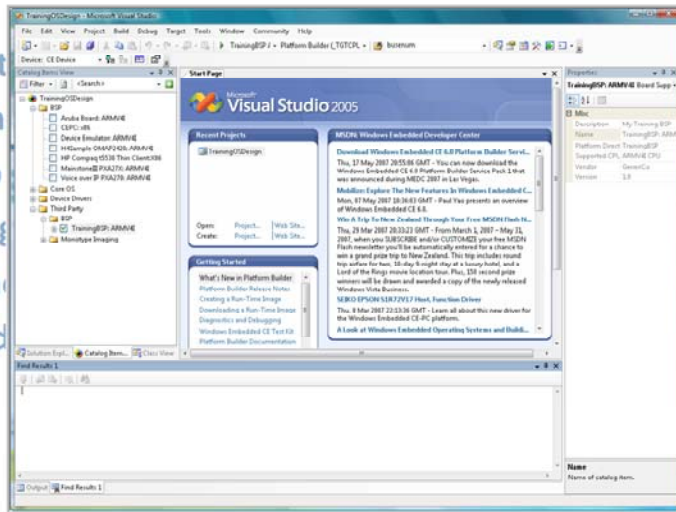
Term	Definition
Design template	A pre-defined selection of operating system (OS) components that Microsoft provides for a category of target devices. A design template is a starting point. When saved or modified, the design template becomes an OS Design.
Run-time image	Software to deploy on a target device, or the same software running on a target device. A run-time image contains the OS and associated software.
Board support package (BSP)	Software that is specific to a hardware board. This software typically includes the boot loader, OEM adaptation layer (OAL), and board-specific device drivers.
Configuration	A selection of Catalog items and a selection of build options.
Hardware platform	A hardware architecture for running a Windows Embedded CE OS and associated software.
Module	An EXE or a DLL that is a part of a Windows Embedded CE OS.
Subproject	A tracking mechanism for a collection of files that you can use to build functionality into a Windows Embedded CE OS.
Target device	An instance of a hardware architecture or an instance of a combined hardware and software architecture.
Project	A container for all files related to an OS Design.

Link to terminology in Windows Embedded CE:

<http://msdn2.microsoft.com/en-us/library/aa924102.aspx>

Tools for Platform Development

- Visual Studio 2005 & CE 6.0 R2 Installation
- Windows Embedded CE Terminology
- A Look at the IDE
- Lab 2.1 – Our First Project
- Introduction to the Platform Builder
- Lab 2.2 – Developing a Project
- Testing and Debugging
- Lab 2.3 – Using the Platform Builder
- Windows Embedded CE
- Review



Tools for Platform Development

- [Visual Studio 2005 & CE 6.0 R2 Installation Overview](#)
 - [Windows Embedded CE Terminology](#)
 - [A Look at the IDE](#)
 - **Lab 2.1 – Our First OS**
 - [Introduction to Windows Embedded CE](#)
 - [Lab 2.1 – Our First OS](#)
 - [Testing the OS](#)
 - [Lab 2.2 – Our Second OS](#)
 - [Windows Embedded CE](#)
 - [Review](#)
- **Lab Goals**
 1. Clone a BSP
 2. Create an OS Design using Visual Studio
 3. Identify the catalog features included in the design
 4. Extend the standard design by adding catalog items
 5. Build configuration for the run-time image and build a run-time image
 - [Video](#)



Tools for Platform Development

- Visual Studio 2005 & CE 6.0 R2 Installation

- Windows Embedded CE Terminology

- A Look at

- Lab 2.1 –

- **Introduce**

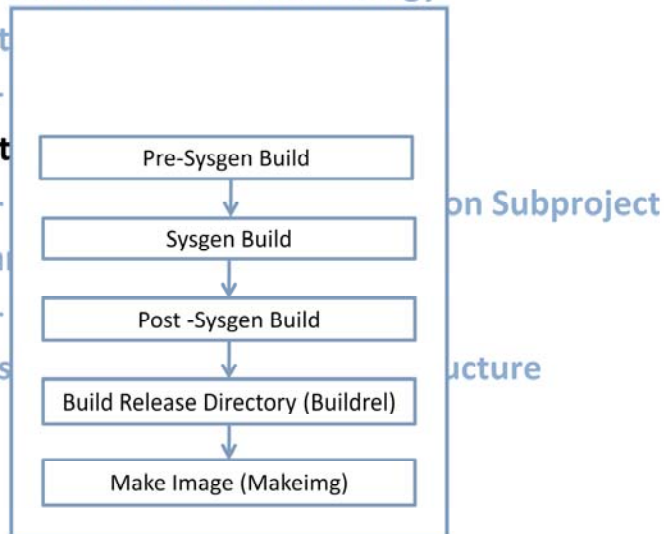
- Lab 2.2 –

- Testing a

- Lab 2.3 –

- Windows

- Review



Pre-Sysgen Build

This is the feature build phase and it compiles the source code that is provided by Microsoft in the various Public subdirectories (_DEPTREES). Microsoft provides the binaries for these components as well, so there is no need to compile the source code unless it has been changed. You should not modify the source code in these trees, so there should be no reason to run this step. This step is only exposed in the Build menu as part of the Advanced Build Commands.

System Generation and Post Sysgen Build

Filters modules and components based on OS Design settings. Then build the BSP.

Build Release – BUILDREL

Copies files into the Flat Release Directory

Make Image

Generates the OS Run-Time Image from the files in the Release Directory

This build process will be discussed in length throughout the course.

Introduction to the Build Process (continued)

- **Two Methods of Building (with many variations for expert building)**
 - IDE
 - Building in its simplest form is accomplished by selecting a menu item
 - Command Line
 - Typically considered advanced
 - Used by the IDE behind the scenes

Tools for Platform Development

- Visual Studio 2005 & CE 6.0 R2 Installation
- Windows Embedded CE Terminology
- A Look at the IDE
- Lab 2.1 – Our First OS
- Introduction to the Build Process
- **Lab 2.2 – Develop & Test an Application Subproject**
- ~~Testing and Debugging the OS Design~~
- Lab
 - Lab Goals
 1. Create an Application Subproject
 2. Deploy the Application
 3. Debug the Application
 - [Video](#)
- Win
- Rev



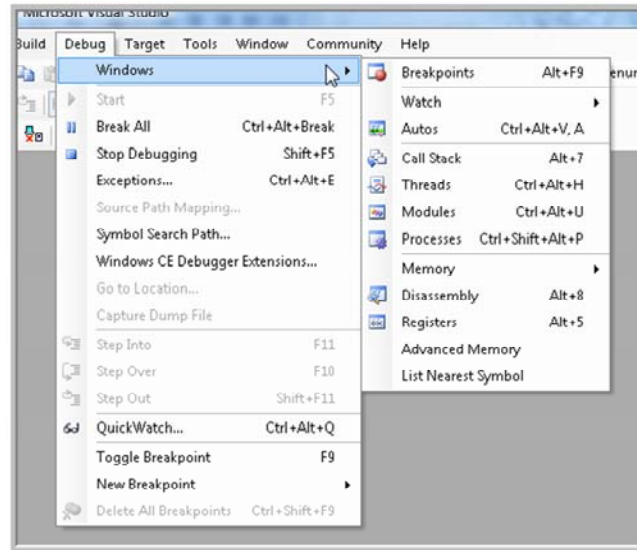
Tools for Platform Development

- Visual Studio 2005 & CE 6.0 R2 Installation
- Windows Embedded CE Terminology
- A Look at the IDE
- Lab 2.1 – Our First OS
- Introduction to the Build Process
- Lab 2.2 – Develop & Test an Application Subproject
- **Testing and Debugging the OS Design**
- Lab 2.3 – Using Remote Tools
- Windows Embedded CE Remote Tools
 - Debug Windows in Visual Studio
 - Remote Tools
- Remote Tools
 - Launching a Program
 - CETK (We'll look at this in depth later)



Testing and Debugging the OS Design (continued)

- Debug Windows in VS

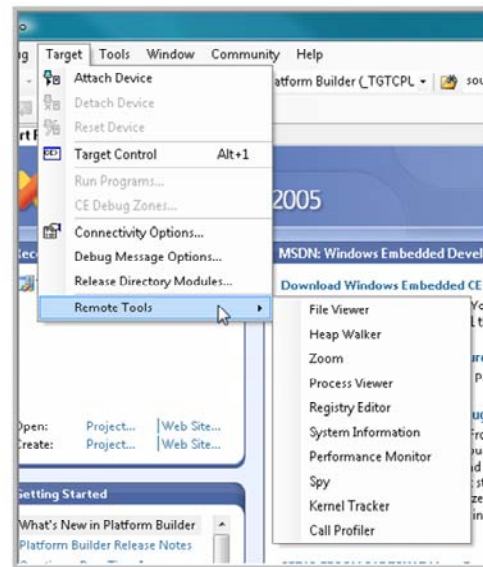


The Debug Menu provides access to many windows for debugging purposes, including those shown here.

Testing and Debugging the OS Design (continued)

- **Remote Tools**

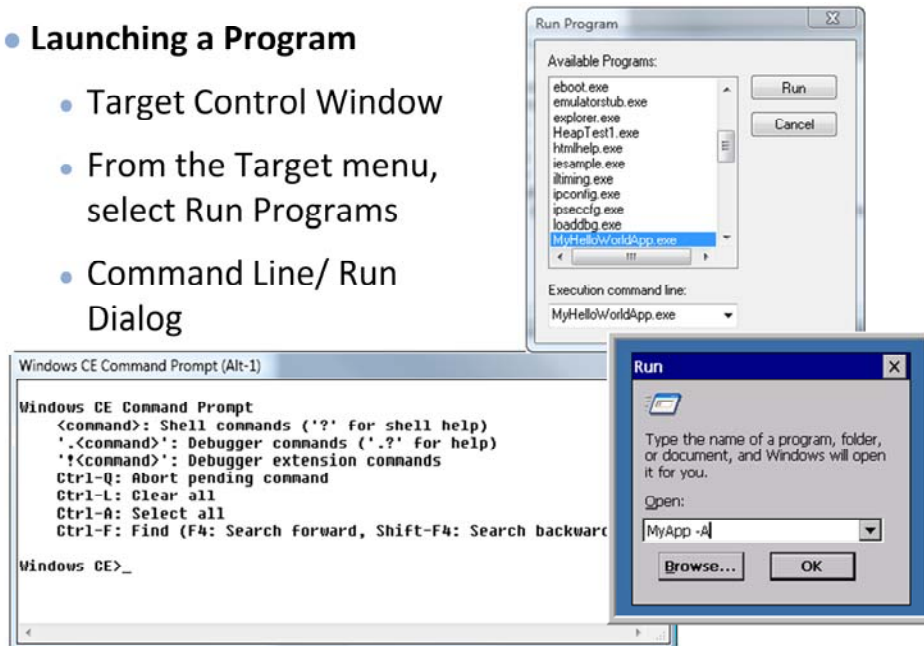
- File Viewer
- Heap Walker
- Zoom
- Registry Editor
- System Information
- Performance Monitor
- Spy
- Kernel Tracker
- Call Profiler



Testing and Debugging the OS Design (continued)

- **Launching a Program**

- Target Control Window
- From the Target menu, select Run Programs
- Command Line/ Run Dialog



In addition to the run dialog on the device the command line shell may be used if it is built in.

Tools for Platform Development

- Visual Studio 2005 R2 CE C++ D3 Installation
- Windows Embedded CE 5.0
- A Look at the Windows Embedded CE 5.0
- Lab 2.1 – Using the Remote Tools
- Introduction to the Remote Tools
- Lab 2.2 – Using the Remote Tools
- [Video](#)
- Testing and Debugging the OS Design
- **Lab 2.3 – Using the Remote Tools**
- Windows Embedded CE Directory Structure
- Review



Tools for Platform Development

- Visual Studio
- Windows Embedded CE
- Lab 2.1
- Introduction
- Lab 2.2
- Testing and Deployment
- Lab 2.3
- Environment Variables
 - %_WINCEROOT% (Often C:\WINCE600)
 - %_WINCEROOT%\OSDesigns
 - %_WINCEROOT%\Platform
 - %_WINCEROOT%\Public
 - %_WINCEROOT%\SDK
 - %_WINCEROOT%\Other
 - %_WINCEROOT%\Private (optional)
 - Visual Studio Directories
- **Windows Embedded CE Directory Structure**
- Review



Note that environment variables are used extensively for paths.

During the installation process a number of folders are “installed”. There are also folders that are created dynamically as needed.

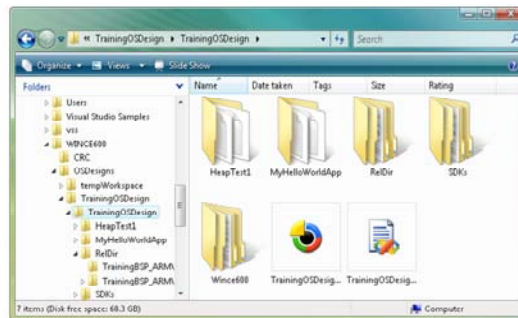
Windows Embedded CE Directory Structure (continued)

- **Environment Variables**

- Explore the Environment Variables using “set”
 - _DEPTREES
 - _FLATRELEASEDIR
 - PBWORKSPACE, PBWORKSPACEROOT
 - _PLATFORMDRIVE, _PLATFORMROOT
 - _PROJECTROOT
 - _PUBLICDRIVE, _PUBLICROOT, _SDKDRIVE, _SDKROOT
 - _TARGETPLATROOT, _TGTCPU, _TGTPLAT, _TGTPROJ
 - Many more...

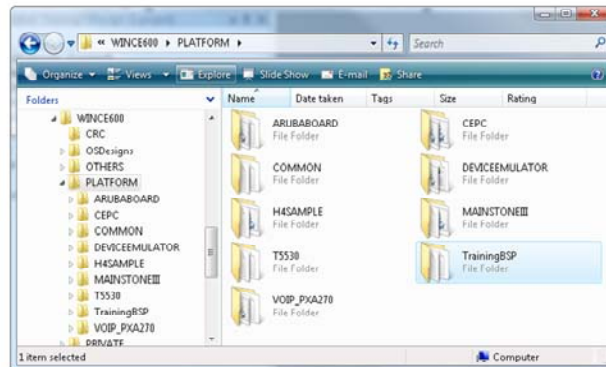
Windows Embedded CE Directory Structure (continued)

- **%_WINCEROOT%\OSDesigns**
 - Created dynamically as the default location for new OS Designs
 - Each OS Design is contained in a sub folder
 - Each OS Design contains a “reldir”



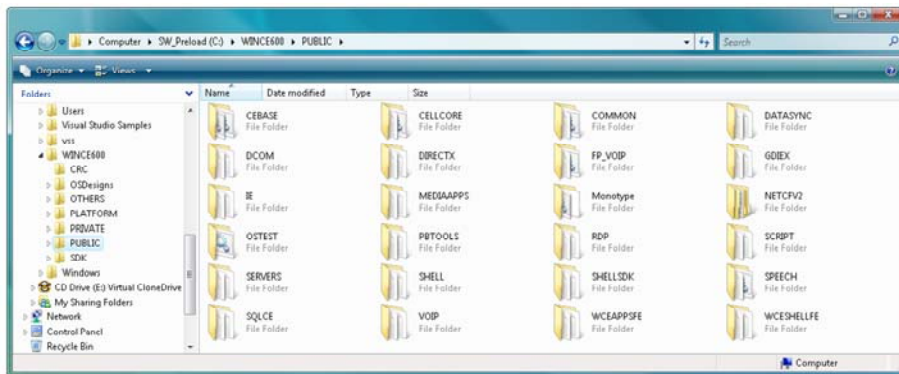
Windows Embedded CE Directory Structure (continued)

- **%_WINCEROOT%\Platform**
 - Contains the installed BSP common code
 - Contains the installed BSPs
 - Contains the newly created or cloned BSPs



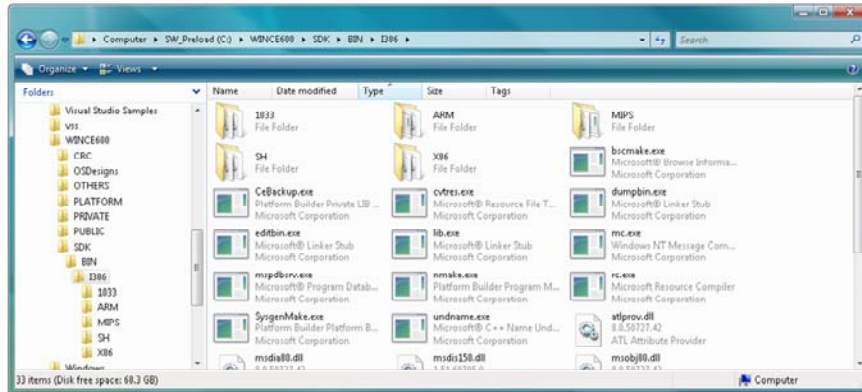
Windows Embedded CE Directory Structure (continued)

- **%_WINCEROOT%\Public**
 - Platform independent drivers and code (may be chip specific)
 - Component and configuration files not specific to a hardware platform or a specific OS Design



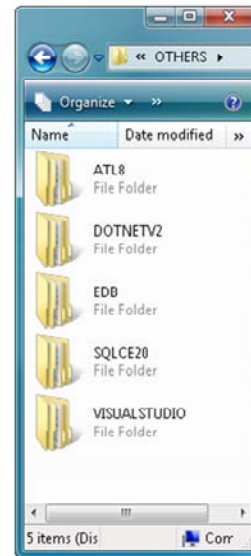
Windows Embedded CE Directory Structure (continued)

- `%_WINCEROOT%\SDK`
 - Compilers for each processor family
 - Linkers for each processor family
 - Other Tools



Windows Embedded CE Directory Structure (continued)

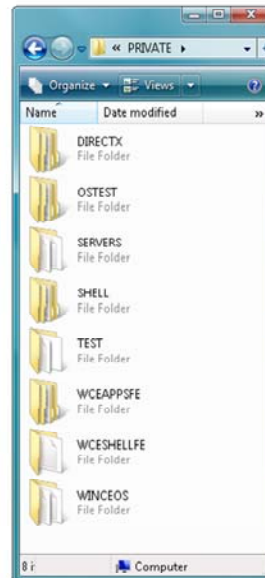
- **%_WINCEROOT%\Others**
 - ATL
 - Source & Libs
 - DOTNETV2
 - Compact Framework binaries
 - EDB
 - Binaries
 - SQLCE20
 - Binaries
 - VISUALSTUDIO
 - Authentication Utility



Windows Embedded CE Directory Structure (continued)

- **%_WINCEROOT%\Private (optional installation)**

- This is where the shared source is installed
 - DIRECTX
 - OSTEST
 - SERVERS
 - SHELL
 - TEST
 - WCEAPPSFE
 - WCESHLLFE
 - WINCEOS
- Can modify and ship



Windows Embedded CE Directory Structure (continued)

- **Visual Studio Directories**

- Visual Studio 2005 Code Top Level Projects Dir
 - C:\Documents and Settings\\My Documents\Visual Studio 2005\Projects
- Visual Studio PB Plug-in Dir:
 - C:\Program Files\Microsoft Platform Builder\6.00
- Visual Studio SmartDevice Dir:
 - C:\Program Files\Microsoft Visual Studio 8\SmartDevices
- Common Shared Files:
 - C:\Program Files\Common Files\Microsoft Shared

Tools for Platform Development

- Visual Studio 2005 & CE 6.0 R2 Installation
- Windows Embedded CE Terminology
- A Look at the IDE
- Lab 2.1 – Our First OS
- Introduction to the Build Process
- Lab 2.2 – Develop & Test an Application Subproject
- Testing and Debugging the OS Design
- Lab 2.3 – Using the Remote Tools
- Windows Embedded CE Directory Structure
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Operating System Internals

Course Outline

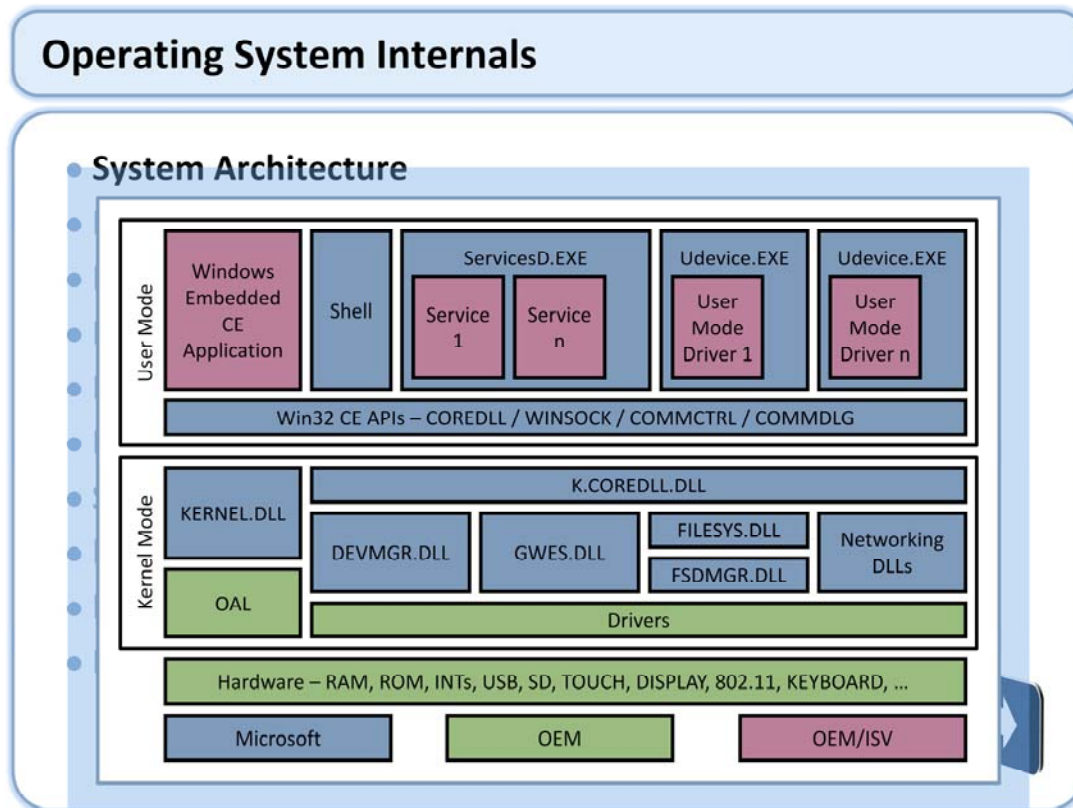
- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- **Module 3: Operating System Internals**
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification



Operating System Internals

- System Architecture
- Memory Model
- Labs 3.1, 3.2, 3.3 – Using Memory Tools
- Processes and Threads
- Lab 3.4 – Exploring Threads with Kernel Tracker
- Synchronization Objects
- Labs 3.5, 3.6 – A Look at Synchronization
- Interrupt Model
- Review





<http://msdn2.microsoft.com/en-us/library/aa924061.aspx>

This diagram shows the overall system architecture. Fundamentally an OS that support a single process in Kernel Mode with essentially an unlimited number of processes, 32K, in user mode. Process can have an unlimited number of threads, which are the actual units of execution. Drivers can be either user mode or kernel mode.

The blocks are color coded as a reference to who typically provides the major functionality within each block.

The OAL is the abstraction layer for the kernel for your specific hardware. It contains, among other things, the interface to the system timer hardware. The drivers next to the OAL here, are the kernel mode drivers in the system. These drivers will provide interface to specific pieces of hardware or hardware functionality in the system. Since these drivers are kernel mode drivers that do not need any special handling to gain access to kernel space or kernel functionality.

- kernel.dll supplies core kernel functionality
- devmgr.dll supplies the management services for drivers
- gwes.dll provides the graphics, windows, and event subsystem
- filesys.dll supplies the file system interface.
- fsdmgr.dll provides the management of file system drivers, networking DLLs,
- Kcoredll.dll provides the primary WIN32 API interface for components running in kernel mode.

In user mode, there are a number of API modules (DLLs) that provide the interfaces necessary to interface to the kernel and/or provide the basic WIN32 functionality for components running in user mode. Additionally, there are a number of processes running in user mode that supply certain functionality for the system. Udevice.exe is a hosting process for user mode drivers to provide the support needed to run drivers from user mode. There may be, and often are multiple copies of Udevice.exe running in the system. ServicesD.exe provides the hosting process for services. Next we have the shell and other applications which can vary dramatically from device to device.

Operating System Internals

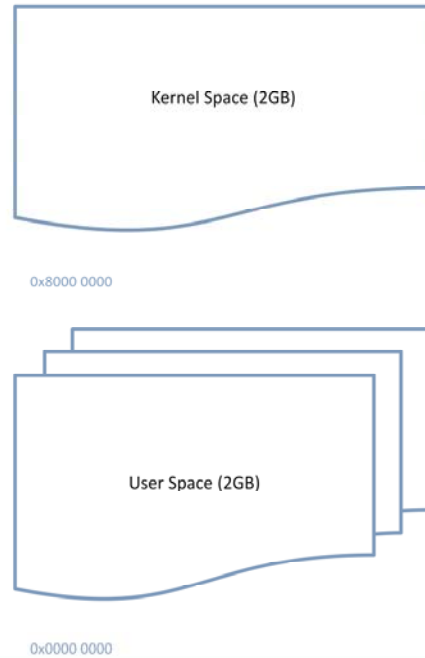
- System Architecture
- **Memory Model**
- Lab 2.1 – Using Memory Tools
 - Virtual Memory Model
 - User Virtual Memory
 - Kernel Virtual Memory
 - Memory System Tools
- Lab 3.3 – A Look at Synchronization
- Interrupt Model
- Review



Memory Model (continued)

- **Virtual Memory Model**

- Virtual Memory is divided between kernel space and user space



Virtual Memory:

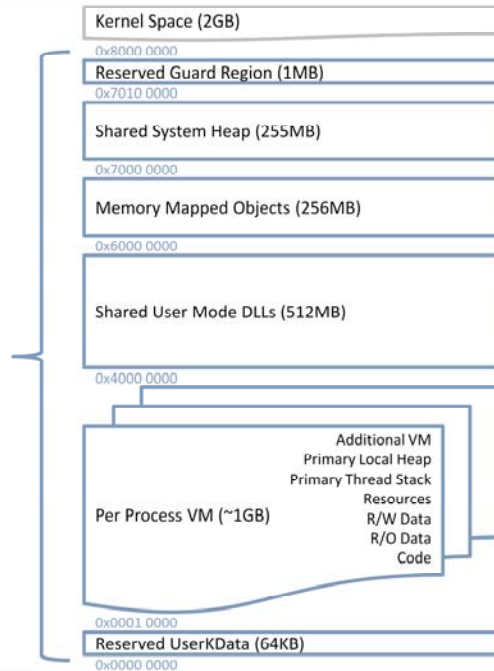
<http://msdn2.microsoft.com/en-us/library/bb202734.aspx>

The Windows Embedded CE virtual memory model divides the memory into both kernel space and user space. Kernel space is the upper 2G while user space is the lower 2G. User space is per process space, where as kernel space is single instance.

Memory Model (continued)

- **User Virtual Memory**

- User Space is all the space below 0x80000000
- Up to 32K processes
- Each with 2G address space of which 1G is independent of other processes
- Per process VM is used by code, heaps and stacks, data



Virtual Memory:

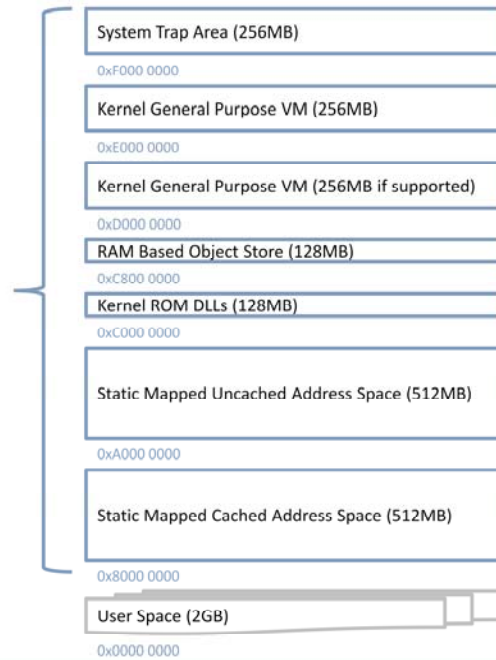
<http://msdn2.microsoft.com/en-us/library/bb202734.aspx>

Now we are looking at the area below the 2G boundary. This is considered User space. Even though we consider the complete lower 2G to be per process space there are portions that are actually shared between processes. At the 1G boundary we load all user mode dlls. These dlls are single instanced from a code point of view. There is up to 512MB of space available for the loading of these dlls. At the 1.5G boundary we have a 256MB area for memory mapped files. Above that, at the upper 256MB, of the lower 2G space, we have the shared heap area.

Memory Model (continued)

- **Kernel Virtual Memory**

- Starts at 0x80000000
- Cached and uncached space is mapped to the same physical memory address space



Virtual Memory:

<http://msdn2.microsoft.com/en-us/library/bb202734.aspx>

This diagram shows the layout of the kernel memory space with 6 major areas.

The static mapped address space provide a 512MB mapping for both cached and uncached access to up to 512MB of virtual memory space. Above that we have 128MB of kernel ROM dll space, check out addresses of modules in this space as you are performing the various labs. Above that, starting at address C8000000 we have the 128MB RAM based object store. As we will see a little later this area holds the registry, a file system, and a database. Finally, above that we have another 512MB that is used as general purpose memory for the kernel. Depending on the processor architecture the system will take advantage of 256MB or 512MB. At the top of the 512MB is the system trap area for relocated interrupt vectors.

Memory Model (continued)

- **Memory System Tools**
 - Process Viewer
 - Heap Walker
 - Kernel Tracker
 - System Information
 - Target Control Commands
 - mi
 - gi mod
 - Toolhelp.dll – heap functions

There are many tools in Windows Embedded CE that provides some aspect of memory related functionality. Toolhelp exposes:

CloseToolhelp32Snapshot - Closes a handle to a snapshot.

CreateToolhelp32Snapshot - Takes a snapshot of the processes, heaps, modules, and threads used by the processes.

Heap32First - Retrieves information about the first block of a heap allocated by a process.

Heap32ListFirst - Retrieves information about the first heap allocated by a specified process.

Heap32ListNext - Retrieves information about the next heap allocated by a process.

Heap32Next - Retrieves information about the next block of a heap allocated by a process.

Module32First - Retrieves information about the first module associated with a process.

Module32Next - Retrieves information about the next module associated with a process or thread.

Process32First - Retrieves information about the first process encountered in a system snapshot.

Process32Next - Retrieves information about the next process recorded in a system snapshot.

Thread32First - Retrieves information about the first thread of a process encountered in a system snapshot.

Thread32Next - Retrieves information about the next thread of a process encountered in the system memory snapshot.

Toolhelp32ReadProcessMemory - Copies memory allocated to another process into an application-supplied buffer.

Memory Model (continued)

- Process Viewer

The screenshot shows the Windows CE Remote Process Viewer interface. It displays three tables: Processes, Threads, and Modules.

Process	PID	Base Priority	# Threads	Base Addr	Access Key	W
NK.EXE	00400002	3	63	80070000	00000000	C
shell.exe	00F50002	3	1	00010000	00000000	
udevice.exe	01880002	3	7	00010000	00000000	E
udevice.exe	01F90002	3	1	00010000	00000000	
udevice.exe	01150006	3	1	00010000	00000000	
udevice.exe	03630002	3	1	00010000	00000000	
explorer.exe	037D0006	3	4	00010000	00000000	T
EmulatorStub...	038D0006	3	1	00010000	00000000	
servicesd.exe	03980006	3	4	00010000	00000000	

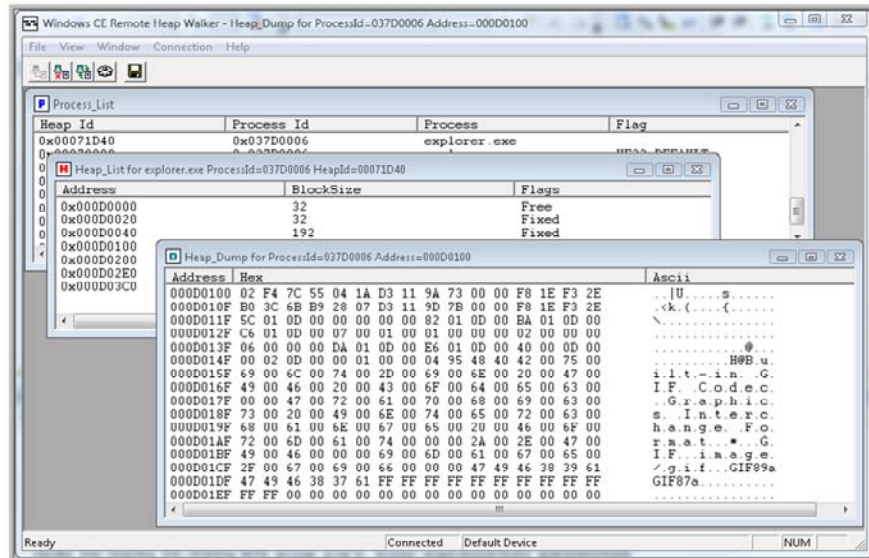
Thread ID	Current PID	Thread Priority	Access Key
03DD0012	00400002	251	00000000
03BC000E	00400002	251	00000000
037C000E	00400002	249	00000000
03750006	00400002	249	00000000
036B0006	00400002	249	00000000
035D0006	00400002	109	00000000
033B0006	00400002	251	00000000

Module	Module ID	Proc Count	Global Count	Base Addr	Base Size	hModule
ce1kitl.dll	94E91724	2	2	40E50000	24576	94E91724
ce1stsub.dll	94E8EA20	1	1	40E60000	20480	94E8EA20
toolhelp.dll	97F30BA8	2	2	400F0000	24576	97F30BA8
netui.dll	94E1C180	2	2	402D0000	249856	94E1C180
comctrl.dll	94DC66D0	3	3	40110000	401408	94DC66D0
fpct.dll	94DBC6F0	3	3	400B0000	73728	94DBC6F0
ceddk.dll	94E1C2D0	2	2	401F0000	40960	94E1C2D0
ws2.dll	94CA7E40	4	4	40240000	53248	94CA7E40

The process viewer is shown here. While its primary use is more process and thread related; it does provide some memory related information. You will notice the base address for processes as well as the base address for modules within the system. Note that all Windows Embedded CE processes have a load address of 0x10000. That is with one exception; nk.exe which is the only process that runs in kernel space.

Memory Model (continued)

- **Heap Walker**



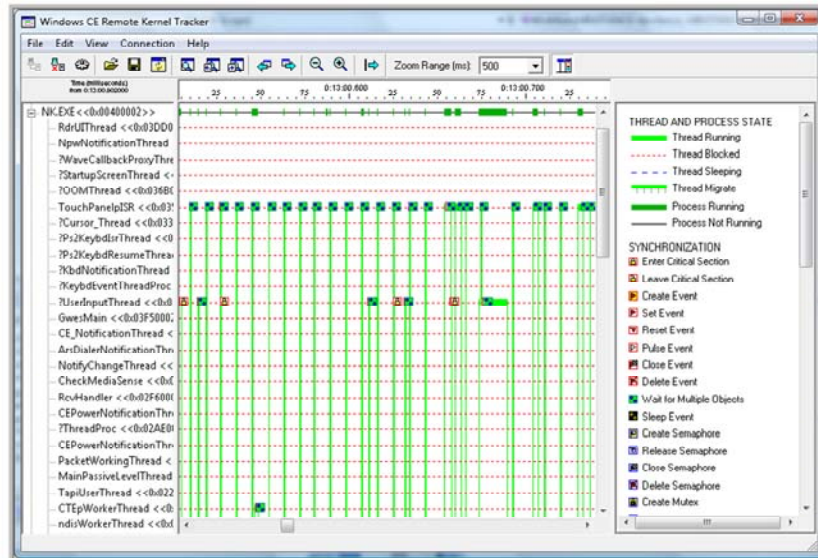
Heaps:

<http://msdn2.microsoft.com/en-us/library/bb202725.aspx>

The heap walker allows us to, as the name says, walk the heap for the active processes within the system. There are three primary windows within heap walker. First the process list window, then the heap list window a selected process and finally a dump windows for details of an item selected from the heap list.

Memory Model (continued)

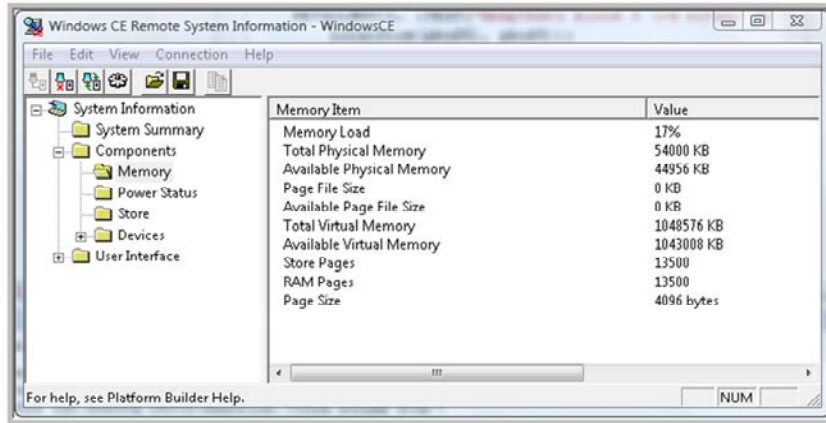
- Kernel Tracker



Kernel Tracker allows viewing of many events in the system. While not its primary use it does allow viewing heap related events as well.

Memory Model (continued)

- System Information



The screenshot shows the 'Windows CE Remote System Information' application window. The left pane displays a tree view with 'System Information' expanded, showing sub-items like 'System Summary', 'Components', 'Memory', 'Power Status', 'Store', 'Devices', and 'User Interface'. The right pane displays a table of memory-related system information.

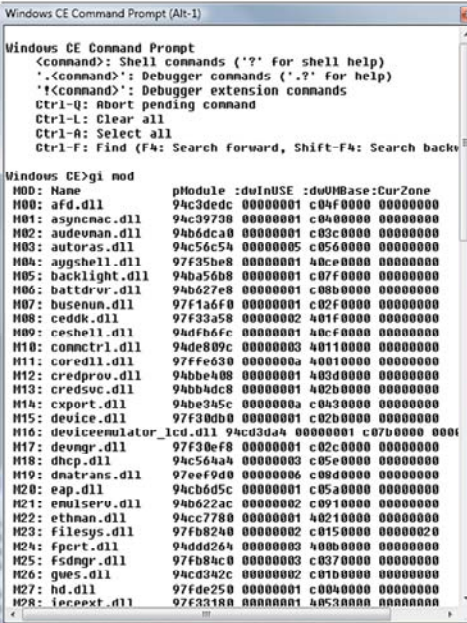
Memory Item	Value
Memory Load	17%
Total Physical Memory	54000 KB
Available Physical Memory	44956 KB
Page File Size	0 KB
Available Page File Size	0 KB
Total Virtual Memory	1048576 KB
Available Virtual Memory	1043008 KB
Store Pages	13500
RAM Pages	13500
Page Size	4096 bytes

For help, see Platform Builder Help.

Memory Model (continued)

- Target Control Commands

- gi mod
 - List modules loaded and base address



```

Windows CE Command Prompt (Alt-1)
Windows CE Command Prompt
<command>: Shell commands ('?' for shell help)
'<command>': Debugger commands ('.? for help)
!<command>: Debugger extension commands
Ctrl-U: Abort pending command
Ctrl-L: Clear all
Ctrl-A: Select all
Ctrl-F: Find (F4: Search forward, Shift-F4: Search back)

Windows CE>gi mod
M00: Name                pModule :dwInUSE :dw0HBase:CurZone
M00: afd.dll             94c3dedc 00000001 c04f0000 00000000
M01: asyncmac.dll       94c39738 00000001 c04a0000 00000000
M02: audeman.dll        94b6dca0 00000001 c03c0000 00000000
M03: autoras.dll        94c56c54 00000005 c0560000 00000000
M04: aygshell.dll       97f35be8 00000001 40ce0000 00000000
M05: backlight.dll     94ba56b8 00000001 c07f0000 00000000
M06: battdrv.dll        94b627e8 00000001 c08b0000 00000000
M07: busenum.dll        97f1a6f0 00000001 c02f0000 00000000
M08: ceddk.dll          97f33a58 00000002 401f0000 00000000
M09: ceshell.dll        94dfb6fc 00000001 40cf0000 00000000
M10: connect.dll        94de809c 00000003 40110000 00000000
M11: core.dll           97ffe630 0000000a 40010000 00000000
M12: credprov.dll       94bbe408 00000001 403d0000 00000000
M13: credsuc.dll        94bb4dc8 00000001 402b0000 00000000
M14: export.dll         94be345c 0000000a c0430000 00000000
M15: device.dll         97f30db0 00000001 c02b0000 00000000
M16: deviceemulator_lcd.dll 94cd3da4 00000001 c07b0000 0000
M17: devmgr.dll         97f30ef8 00000001 c02c0000 00000000
M18: dhcp.dll           94c564a4 00000003 c05e0000 00000000
M19: dnatrans.dll       97eeF9d0 00000006 c08d0000 00000000
M20: eap.dll            94cb6d5c 00000001 c05a0000 00000000
M21: emulserv.dll       94b622ac 00000002 c0910000 00000000
M22: ethman.dll         94cc7780 00000001 40210000 00000000
M23: filesys.dll        97fb8240 00000002 c0150000 00000020
M24: fpert.dll          94ddd264 00000003 400b0000 00000000
M25: fsdngr.dll         97fb84c0 00000003 c0370000 00000000
M26: gues.dll           94cd342c 00000002 c01b0000 00000000
M27: hd.dll             97fde250 00000001 c0040000 00000000
M28: ieecept.dll        97f33180 00000001 40530000 00000000
  
```


Info on gi command:

<http://msdn2.microsoft.com/en-us/library/aa935268.aspx>

This tool provides another way to access the module's base addresses.

Memory Model (continued)

• Target Control Commands

- mi kernel
 - Lists kernel memory details
- mi full
 - List complete memory details
- Legend 



```

Windows CE Command Prompt (Alt-1)
Windows CE>mi full

Windows CE Kernel Memory Usage Tool 0.2
Page size=4096, 13500 total pages, 11488 free pages, 11462 b
74 pages used by kernel, 0 pages held by kernel, 2012 pages
Inx size Used Max Extra Entries Name
0: 576 49536 49536 0 86( 86) Thrd
1: 64 2880 3200 320 45( 50) MapView
2: 36 43812 43848 36 1217(1218) API/GStk/Prxy/HData/
3: 156 196872 197028 156 1262(1263) Crit/Evt/Sen/Hut/Hoc
4: 288 3456 3744 288 12( 13) Process
5: 524 0 0 0 0( 0) Name
6: 1024 0 0 0 0( 0) HlprStk
7: 16 3456 3504 48 216(219) cleanEvt/StubEvt/ModL
Total Used = 300012 Total Extra = 848 Waste = 416
PRDC: Name hProcess: CurARKV :du0HBase:CurZone
P00: NK.EXE 00400002 00000000 80070000 00000000
P01: shell.exe 00F50002 00000000 00010000 00000000
P02: udevice.exe 019e0002 00000000 00010000 00000000
P03: udevice.exe 01f60002 00000000 00010000 00000000
P04: udevice.exe 01150006 00000000 00010000 00000000
P05: udevice.exe 036d0002 00000000 00010000 00000000
P06: udevice.exe 03960006 00000000 00010000 00000000
P07: explorer.exe 039a0006 00000000 00010000 00000000
P08: EmulatorStub.exe 03b70006 00000000 00010000 00000000
P09: servicesd.exe 03bf0006 00000000 00010000 00000000

Memory usage for Shared Heap:
7 0000000: -----
7 0010000: -----
7 0020000: WWWW-----

Memory usage for Process 'NK.EXE' pid 400002
c 0000000: -----
c 0010000: -CCCCCCCCWWWWWWW
c 0020000: WWWWWWWWWWWWWW
c 0030000: Wc-----
c 0040000: -CWC-----
c 0050000: -CCCCCCCCWwWc--
c 0060000: -----
c 0070000: -CWC-----
  
```

Info on mi command:

<http://msdn2.microsoft.com/en-us/library/aa935268.aspx>

Memory Model (continued)

● mi Command Legend

Legend	
	A blank space indicates a virtual page that is not currently allocated. Does not require a physical page.
-	Reserved but not in use. Indicates a virtual page that is currently allocated but not mapped to any physical memory. Does not require a physical page.
C	Code pages in ROM. Does not require a physical page.
c	Code pages in RAM. Requires a physical page.
S	Indicates a virtual page that holds a stack. Requires a physical page.
P	Indicates a virtual page that is used to map a range of hardware addresses; that is, peripheral memory pages used to map target device memory by using VirtualAlloc. Does not require a physical page. Peripheral memory may include frame buffer memory.
W	Indicates a virtual page that holds read-write data. Requires a physical page. Read-write pages include global variables as well as dynamically allocated memory.
O	Indicates a virtual page that is used by the object store. Requires a physical page. Should only appear in the Filesys process.
?	Contents unknown.
r	Read-only data pages in RAM. Requires a physical page. Read-only data primarily comes from data items that are declared as a const type in the source code.
R	Read-only data pages in ROM. Does not require a physical page. Read-only data primarily comes from data items that are declared as a const type in the source code.

Info on mi command:

<http://msdn2.microsoft.com/en-us/library/aa935268.aspx>

Operating System Internals

- System Architecture
- Memory Model
- Labs 3.1, 3.2, 3.3 – Using Memory Tools
- Processes and Threads
 - Lab Goals
 1. Understand the use of the Remote Process Viewer for viewing information about processes, threads, and modules
 2. Become familiar with heaps in Windows Embedded CE
 3. Understand the use of the Threads and Modules debug windows
 4. Become familiar with the Remote Heap Walker
 - [Video](#)



Operating System Internals

- System Architecture
- Memory Model
- Lab 3.1 – Using Memory Tools
- **Processes and Threads**
- Lab 3.2 – Exploring Threads with Kernel Tracker
 - Processes
 - Threads
 - Scheduler
- Interrupt Model
- Review



Processes and Threads (continued)

- **Processes**

- A process is a single instance of an application that provide the context within which one or more threads execute
- 32K processes
- Each process has 1G virtual space all its own

Processes:

<http://msdn2.microsoft.com/en-us/library/aa908952.aspx>

Processes and Threads (continued)

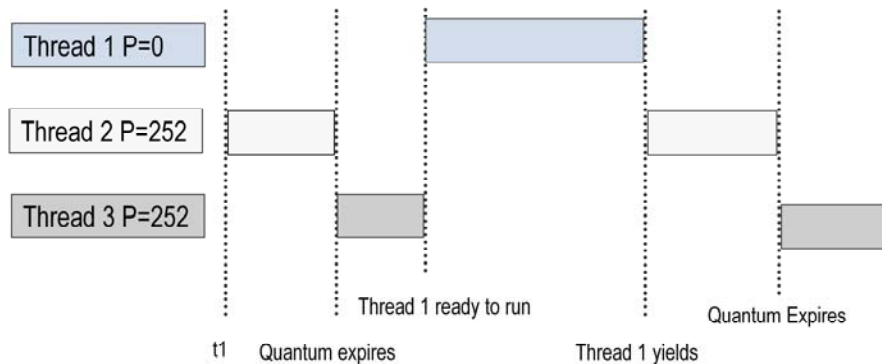
- **Threads & Scheduling**

- Each process has a primary thread and can have additional threads
- Each thread has its own context (stack, priority, quantum, ...)
- The kernel scheduler provides for efficient context switching between threads and is priority based providing allowing for a predictable sequence of execution
- 256 priority levels (0 through 255; 0 is highest priority)
- Round robin scheduling for threads of the same priority
- This is preemptive multitasking
- No detection of thread starvation; highest priority that is ready to run, will always run
- Threads run until:
 - Quantum expires (and equal priority thread is ready to run)
 - Interrupted by a higher priority thread
 - Blocks by resource contention (mutex, critical section, other)
 - Sleeps
 - Terminates

Threads:

<http://msdn2.microsoft.com/en-us/library/aa915094.aspx>

Scheduler – Quantum Time Slicing



- **Same priority threads run in round-robin fashion**
- **Switch threads when blocked**
- **Switch threads after quantum has elapsed**
- **Programmable quantum from 1 to N ms, default 100 ms**

Windows Embedded CE 6.0 uses a priority-based time-slice algorithm to schedule the execution of threads.

Threads with the same priority run in a round-robin fashion: when a thread stops running, all other threads of the same priority run before the original thread can continue. Threads at a lower priority run only after all threads with a higher priority finish or are blocked. If one thread is running and a thread of a higher priority is unblocked, the lower-priority thread is immediately suspended and the higher priority thread is scheduled.

A thread gets to run for set length of time, called a quantum
 Typically 100 milliseconds
 A quantum of 0 means the quantum never runs out
 The thread can run until blocked or interrupted
 OEMs can specify a different quantum.

A Thread runs until—
 Its quantum runs out
 It is interrupted by a higher priority thread
 Its blocked by a resource contention, such as access to a critical section or a mutex

After a thread has used up its quantum, and if any thread with the same priority is ready to run, the current thread is suspended and another thread is scheduled to run. The only exception to this is if a thread is a "run to completion" thread, which means that the thread quantum is equal to 0. Threads with a quantum set to 0 will never expire and will never be preempted by threads of the same priority. Threads that have a quantum set to 0 cannot be preempted except by a higher priority thread or an interrupt service routine (ISR).

Processes and Threads (continued)

• Typical Priority Map

Range	Use
0 - 96	Real-time drivers
97 - 152	Non real-time default drivers
153 - 247	Additional non real-time drivers
248 - 255	Applications and other non real-time priorities

Priorities:

<http://msdn2.microsoft.com/en-us/library/bb202761.aspx>

While this table shows the overall strategy for priorities in the system, taking the time to walk through the priorities using a tool such as process viewer, looking at the threads of each process, can provide a better understanding. While having a understanding of the priority of threads within a given process is important; often times it is just as important to understand the priorities relative to other threads within the system.

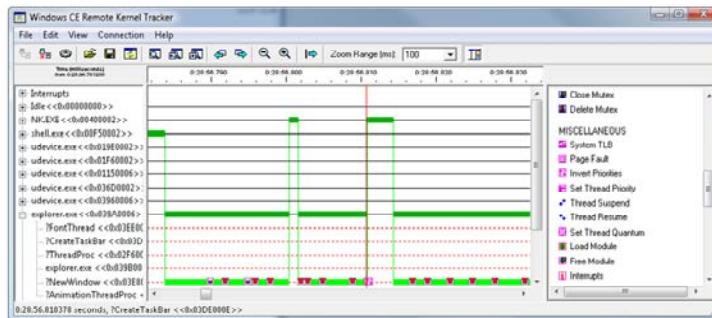
Thread Priority Map (Example)

Priority	Component
0-19	Open - Real Time Above Drivers
20	Graphics Vertical Retrace
99	Power management Resume Thread
100-108	USB OHCI UHCI, Serial
109-129	IRSIR1, NDIS, Touch
130	KITL
131	VMini
132	CxPort
145	PS2 Keyboard
148	IRComm
150	TAPI
248	Power Management
249	WaveDev, Mouse, PnP, Power
250	WaveAPI
251	Normal
252-255	Open - Applications

Processes and Threads (continued)

- **Priority Inversion**

- Corrects for - Lower priority thread, blocked by a medium priority thread, while holding a resource needed by a high priority.
- Windows Embedded CE supports single level of priority inversion to maintain hard real-time functionality.
- Avoid the need for priority inversion.



Priority inversion:

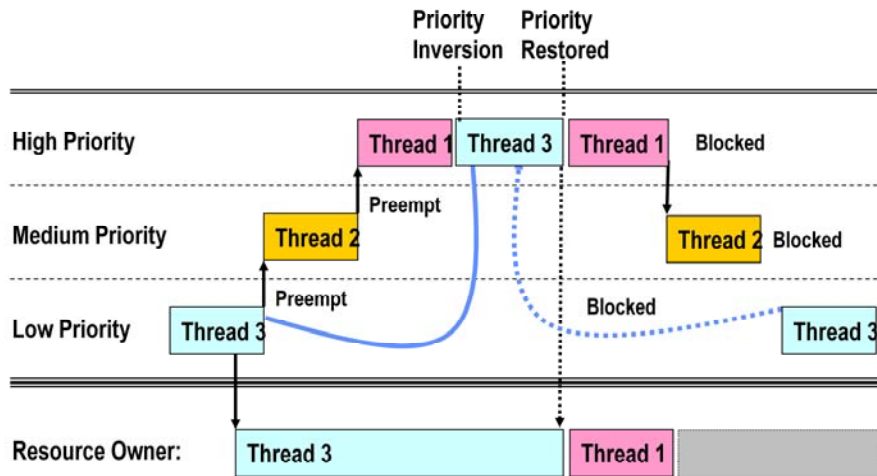
<http://msdn2.microsoft.com/en-us/library/aa915356.aspx>

One way to avoid the need for priority inversion is to understand the priority of the module that is being developed in reference to the other threads in the system that it will be interacting with.

Priority Inversion

- **Avoid priority inversion by keeping all threads waiting for same resource at the same priority**

Example: Thread 1 blocked waiting for resource owned by Thread 3, causing Priority Inversion



There are two primary architectural choices for an OS to handle Priority inversion in a system: Single Level and Fully Nested. In the Fully Nested Mode the OS will walk through all threads blocked and keep boosting each one until the high priority thread can run. This prevents an entire class of deadlocks. Unfortunately it also means an $O(n)$ operation with pre-emption turned off while the scheduler figures out how to get everything unblocked to keep things going. This is a major problem for real-time systems that need deterministic response times. In order to support hard real-time systems Windows CE V3.0 and later switched to using a Single Level handling of priority inversion. That is the OS will boost only one thread to release a block. It is therefore the responsibility of the developer to structure code such that deadlocks are avoided.

Processes and Threads (continued)

• Process & Thread Tools

- Toolhelp.dll – module, process, thread functions
- Processes Window
- Threads Window
- Target Control Windows
 - gi proc
 - gi thrd
 - gi delta
 - gi all
- Kernel Tracker
- Remote Process Viewer

```

Windows CE Command Prompt (Alt-1)
Windows CE>gi proc
PROC: Name                               hProcess: CurARY :dw0HBase:CurZone
P00: HK.EXE                               00400002 00000000 80070000 00000000
P01: Shell.exe                             00f50002 00000000 00010000 00000000
P02: udevice.exe                           019e0002 00000000 00010000 00000000
P03: udevice.exe                           01f60002 00000000 00010000 00000000
P04: udevice.exe                           01150006 00000000 00010000 00000000
P05: udevice.exe                           036d0002 00000000 00010000 00000000
P06: udevice.exe                           03960006 00000000 00010000 00000000
P07: explorer.exe                          039a0006 00000000 00010000 00000000
P08: EmulatorStub.exe                      03b70006 00000000 00010000 00000000
P09: servicesd.exe                         03bf0006 00000000 00010000 00000000
Windows CE>gi thrd
PROC: Name                               hProcess: CurARY :dw0HBase:CurZone
THRD: State :hCurThrd:hCurProc: CurARY :Cp :Bp :Kernel Time
P00: HK.EXE                               00400002 00000000 80070000 00000000
T Blockd 03500012 00400002 00000000 251 251 00:00:00.001
T Blockd 02450012 00400002 00000000 249 249 00:00:00.001
T Blockd 03860006 00400002 00000000 249 249 00:00:00.001
T Blockd 037f0006 00400002 00000000 249 249 00:00:00.005
T Blockd 03760006 00400002 00000000 109 109 00:00:00.004
T Blockd 03610006 00400002 00000000 251 251 00:00:00.025
T Blockd 034d0006 00400002 00000000 248 248 00:00:00.004
T Blockd 03420006 00400002 00000000 249 249 00:00:00.001
T Blockd 033f0006 00400002 00000000 251 251 00:00:00.001
T Blockd 032a0006 00400002 00000000 249 249 00:00:00.004
T Sl/Blk 03300006 00400002 00000000 249 249 00:00:00.085
T Sl/Blk 021b0006 00400002 00000000 251 251 00:00:00.285
T Blockd 03f60002 00400002 00000000 251 251 00:00:00.001
T Blockd 03c70002 00400002 00000000 251 251 00:00:00.001
T Blockd 03690002 00400002 00000000 251 251 00:00:00.375
T Blockd 035c0002 00400002 00000000 132 132 00:00:00.324
T Blockd 03230002 00400002 00000000 251 251 00:00:00.005
T Blockd 03150002 00400002 00000000 251 251 00:00:00.001
T Sl/Blk 02ee0002 00400002 00000000 120 120 00:00:00.452
T Blockd 02e00002 00400002 00000000 251 251 00:00:00.092
T Blockd 02c60002 00400002 00000000 251 251 00:00:00.001
T Blockd 02430002 00400002 00000000 251 251 00:00:00.001
T Blockd 02370002 00400002 00000000 251 251 00:00:00.001
  
```


Processes and Threads (continued)

● Process & Thread Related APIs

API	Description
CeGetCallerTrust	This function retrieves the assigned trust level of a process.
CeGetThreadPriority	This function gets the priority for a real-time thread.
CeGetThreadQuantum	This function gets the time quantum for the specified thread.
CeSetThreadPriority	This function sets the priority for a real-time thread on a thread by thread basis.
CeSetThreadQuantum	This function sets the time quantum for the specified thread.
CeZeroPointer	This function converts a pointer that is mapped to a process into an unmapped pointer.
CreateProcess	This function is used to run a new program
CreateThread	This function creates a thread to execute within the address space of the calling process.
ExitProcess	This function ends a process and all of its threads.
ExitThread	This function ends a thread.
FlushInstructionCache	This function flushes the instruction cache for the specified process.
GetCommandLine	This function returns a pointer to the command-line string for the current process.

Processes and Threads (continued)

● Process & Thread Related APIs (continued)

API	Description
GetCurrentProcess	This function returns a pseudo handle for the current process.
GetCurrentProcessId	This function returns the process identifier of the calling process.
GetCurrentThread	This function returns a pseudohandle for the current thread.
GetCurrentThreadId	This function returns the thread identifier, which is used as a handle of the calling thread.
GetExitCodeProcess	This function retrieves the termination status of the specified process.
GetExitCodeThread	This function retrieves the termination status of the specified thread.
GetProcessVersion	Retrieves major and minor version of the system on which the specified process expects to run.
GetThreadPriority	This function returns the priority value for the specified thread.
GetThreadTimes	This function obtains timing information about a specified thread.
OpenProcess	This function returns a handle to an existing process object.
ReadProcessMemory	This function reads memory in a specified process.
ResumeThread	This function decrements a thread's suspend count.

Processes and Threads (continued)

● Process & Thread Related APIs (continued)

API	Description
SetThreadContext	This function sets the context in the specified thread.
SetThreadPriority	This function sets the priority value for the specified thread.
Sleep	This function suspends the execution of the current thread for a specified interval.
SuspendThread	This function suspends the specified thread.
TerminateProcess	This function terminates the specified process and all of its threads.
TerminateThread	This function stops the specified thread.
ThreadProc	This function is an application-defined function that serves as the starting address for a thread.
TlsAlloc	This function allocates a thread local storage (TLS) index.
TlsFree	This function releases a thread local storage (TLS) index, making it available for reuse.
TlsGetValue	Retrieves the value in the calling thread's thread local storage (TLS) slot for a specified TLS index.
TlsSetValue	Stores a value in the calling thread's thread local storage (TLS) slot for a specified TLS index.
WriteProcessMemory	This function writes memory in a specified process.

Operating System Internals

- System Architecture
- Memory Model
- Labs 3.1, 3.2, 3.3 – Using Memory Tools
- Processes and Threads
- **Lab 3.4 – Exploring Threads with Kernel Tracker**

- Sy
 - La
 - In
 - Re
- Lab Goals
 1. Learn which build options are necessary to work with the Remote Kernel Tracker
 2. Become familiar with the Remote Kernel Tracker menu
 3. Recognize execution patterns in Kernel Tracker
 - [Video](#)



Operating System Internals

- System Architecture
- Memory Model
- Lab 3.1 – Using Memory Tools
- Processes and Threads
- Lab 3.2 – Exploring Threads with Kernel Tracker
- **Synchronization Objects**
- Lab 3.3 – A Look at Synchronization
- Int
- Re
 - Synchronization Mechanisms
 - Critical Sections
 - Mutexes
 - Semaphores
 - Events
 - Interlocked Functions
 - Point to Point Message Queues



Very often when using multiple threads in a system there is a need for some type of synchronization. Here we will explore the synchronization objects available within Windows Embedded CE.

Synchronization Objects (continued)

- **Critical Sections**

- Protects a section of code with exclusive access
- Limited to single process
- Other threads blocked until ownership is released
- More efficient than mutex
- Can be used in the OAL

API	Description
DeleteCriticalSection	This function releases all resources used by a critical section object that is not owned.
EnterCriticalSection	This function waits for ownership of the specified critical section object.
InitializeCriticalSection	This function initializes a critical section object.
LeaveCriticalSection	This function releases ownership of the specified critical section object.
TryEnterCriticalSection	This function attempts to enter a critical section without blocking.

Critical Sections:

<http://msdn2.microsoft.com/en-us/library/aa910712.aspx>

One example of critical section use would be a driver allows multiple threads access to a hardware register that requires a sequence of writes.

Synchronization Objects (continued)

• Mutexes

- Allows a single thread to have ownership
- Named mutexes permit inter-process synchronization
- Signaled when not owned

API	Description
ReleaseMutex	This function releases ownership of the specified mutex object.
CreateMutex	This function creates a named or unnamed mutex object.
WaitForMultipleObjects	This function returns when either any one of the specified objects is in the signaled state, or the time-out interval elapses.
WaitForSingleObject	This function returns when the specified object is in the signaled state or when the time-out interval elapses.

Mutexes:

<http://msdn2.microsoft.com/en-us/library/bb202813.aspx>

MsgWaitForMultipleObjects(Ex) should be used in place of the other wait functions for threads that need to process windows messages as well.

DuplicateHandle() allows for creating a handle that can be used for the same event in another process space. All types of handles can be duplicated with Windows Embedded CE 6.0.

Synchronization Objects (continued)

- **Semaphores**

- Limits the number of threads using a protected resource
- Named semaphores allow inter-process use
- Signaled when count > 0

API	Description
ReleaseSemaphore	This function increases the count of the specified semaphore object by a specified amount.
CreateSemaphore	This function creates a named or unnamed semaphore object.
WaitForMultipleObjects	This function returns when either any one of the specified objects is in the signaled state, or the time-out interval elapses.
WaitForSingleObject	This function returns when the specified object is in the signaled state or when the time-out interval elapses.

Semaphores:

<http://msdn2.microsoft.com/en-us/library/aa909242.aspx>

Synchronization Objects (continued)

- **Events**

- Object that is either signaled or reset
- Two types
 - Manual Reset Events
 - Auto Reset Events
- Named events allow easy inter-process use
- Use Wait functions to wait on event

API	Description
CreateEvent	This function creates a named or an unnamed event object.
OpenEvent	This function opens an existing named event object.
PulseEvent	This function provides a single operation that sets to signaled the state of the specified event object and then resets it to nonsignaled after releasing the appropriate number of waiting threads.
ResetEvent	This function sets the state of the specified event object to nonsignaled.
SetEvent	This function sets the state of the specified event object to signaled.

Events:

<http://msdn2.microsoft.com/en-us/library/aa915075.aspx>

Synchronization Objects (continued)

• Comparison of Auto and Manual Reset Events

Manual Reset Events	Auto Reset Events
Signaled with SetEvent	Signaled with SetEvent
1. Kernel releases all waiting threads	1. Kernel releases single waiting thread
2. Kernel releases all subsequently waiting threads	2. All remaining and subsequently waiting threads are blocked
3. Event must explicitly be set to non-signaled with ResetEvent	3. Kernel automatically transitions event to non-signaled state - remains signaled until single thread is released
Signaled with PulseEvent	Signaled with PulseEvent
1. Kernel releases all waiting threads	1. Kernel releases at most one waiting thread
2. Kernel automatically transitions event to non-signaled	2. Kernel automatically transitions event to non-signaled - even if no thread has been released

Events:

<http://msdn2.microsoft.com/en-us/library/aa915075.aspx>

This table provides a quick reference to the options for events. As we said there are two types of events, auto and manual. There are also two methods for triggering events, PulseEvent and SetEvent.

Synchronization Objects (continued)

- **Interlocked Functions**
 - Support atomic read/write/modify operations on data
 - Does not require overhead of other synchronization objects
 - Must be 32 bit aligned

Interlocked functions:

<http://msdn2.microsoft.com/en-us/library/aa911383.aspx>

Synchronization Objects (continued)

• Interlocked Function APIs

API	Description
InterlockedCompareExchangePointer	This function performs an atomic comparison of the specified parameter values and exchanges the values based on the outcome of the comparison.
InterlockedDecrement	This function both decrements (decreases by one) the value of the specified 32-bit variable and checks the resulting value.
InterlockedExchange	This function atomically exchanges a pair of 32-bit values.
InterlockedExchangeAdd	This function performs an atomic addition of an increment value to an Addend variable.
InterlockedExchangePointer	This function atomically exchanges a pair of values.
InterlockedIncrement	This function both increments (increases by one) the value of the specified 32-bit variable and checks the resulting value.
InterlockedTestExchange	This function is an interlocked function that performs a conditional setting of a variable.
InterlockedCompareExchange	This function performs an atomic comparison of the specified values and exchanges the values based on the outcome of the comparison.

Interlocked functions:

<http://msdn2.microsoft.com/en-us/library/aa911383.aspx>

Synchronization Objects (continued)

- **Point to Point Message Queues**

- Writable queue handles are signaled whenever the queue is not full
- Readable queue handles are signaled if the queue is not empty
- Supports high priority and alert messages
- Use Wait functions to wait on a queue

API	Description
CreateMsgQueue	Creates or opens a user-defined message queue.
OpenMsgQueue	Opens a handle to an existing message queue.
CloseMsgQueue	Closes an open message queue.
ReadMsgQueue	Reads a single message from a message queue.
WriteMsgQueue	Writes a single message from a message queue.
GetMsgQueueInfo	Returns information about a message queue.

Operating System Internals

- System Architecture
- Memory Model
- Lab
 - Lab Goals
 1. Understand the read/modify/write vulnerability
 2. Be able to implement an atomic read/modify/write sequence using a critical section
 - [Video](#)
- Labs 3.5, 3.6 – A Look at Synchronization
- Interrupt Model
- Review



Operating System Internals

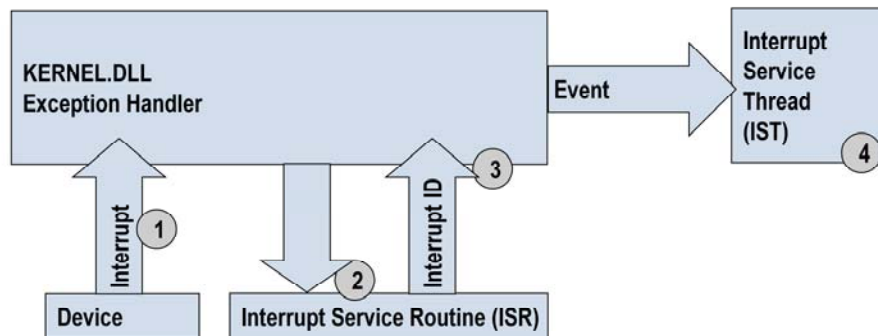
- System Architecture
- Memory Model
- Lab 3.1 – Using Memory Tools
- **Process Model**
 - Basic Model
 - Hardware triggers Kernel Interrupt Handler which calls ISR
 - ISR returns ID
 - Kernel schedules IST associated with ID
- Lab 3.2 – A LOOK at Synchronization
- **Interrupt Model**
- Review



Interrupts landing page:

<http://msdn2.microsoft.com/en-us/library/bb201995.aspx>

Interrupt Model



- **Interrupt Processing**

- Device raises registered hardware interrupt
- Kernel gets exception, calls associated Interrupt Service Routine (ISR)
- Interrupt Service Routine (ISR) quickly deals with pending interrupt
- Interrupt Service Thread (IST) in driver is signaled to process interrupt

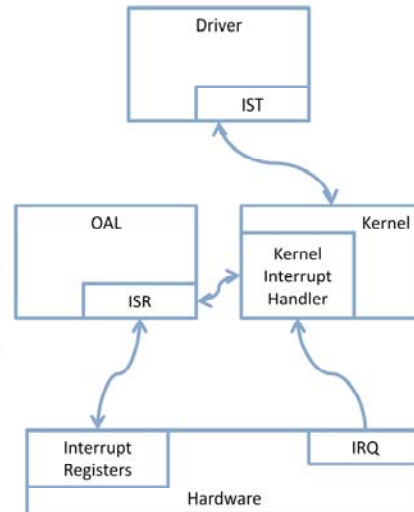
Real-time applications use interrupts to respond to external events in a timely manner. To do this, Windows Embedded CE 6.0 breaks interrupt processing into two steps: an interrupt service routine (ISR) and an interrupt service thread (IST). The ISR runs immediately to identify and mask the interrupt, and perform any high priority tasks. The corresponding IST is a normal system thread (although typically of high priority) and can perform the bulk of the handling that is not time critical. This two stage model allows the operating system to maximize the amount of time the system is able to respond to other high priority interrupts.

The kernel is able to handle a total of 64 interrupts from external sources, some of which are predefined (e.g. system timer interrupt, real time clock etc). Devices that have more than 64 interrupt sources that need to be exposed (rare) must implement a mechanism to share interrupt identifiers. Typically this is done by multiplexing related interrupts together in the ISR, and demultiplexing them in the IST.

Interrupt Model (continued)

● Interrupt Sequence

1. OAL hooks ISR to interrupt source using HookInterrupt API
2. Driver spawns Interrupt Service Thread (IST)
3. Driver creates an event to be used to signal thread when interrupt occurs
4. IST calls InterruptInitialize with SYSINTR ID and event handle
5. Kernel associates SYSINTR ID with event
6. Kernel calls OAL's OEMInterruptEnable function to enable the interrupt
7. IST calls WaitForSingleObject on interrupt event
8. Hardware device raises IRQ
9. Kernel exception handler is triggered and disables all interrupts at an equal and lower priority
10. Hooked ISR is called by kernel exception handler
11. ISR identifies interrupt and returns SYSINTR ID to kernel exception handler
12. Kernel sets interrupt event associated with the SYSINTR ID
13. IST waiting for interrupt event is scheduled based on priority by the kernel scheduler
14. IST processes interrupt
15. IST calls InterruptDone
16. Kernel calls OAL's OEMInterruptDone function to re-enable the interrupt



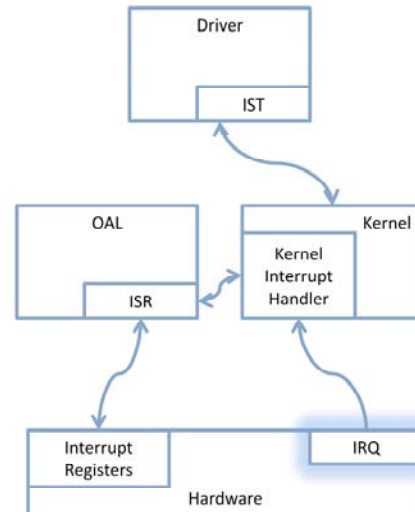
Real-time applications use interrupts to respond to external events in a timely manner. To do this, Windows Embedded CE 6.0 breaks interrupt processing into two steps: an interrupt service routine (ISR) and an interrupt service thread (IST). The ISR runs immediately to identify and mask the interrupt, and perform any high priority tasks. The corresponding IST is a normal system thread (although typically of high priority) and can perform the bulk of the handling that is not time critical. This two stage model allows the operating system to minimize the amount of time the system is able to respond to other high priority interrupts.

Interrupt processing:

<http://msdn2.microsoft.com/en-us/library/aa930251.aspx>

Interrupt Model (continued)

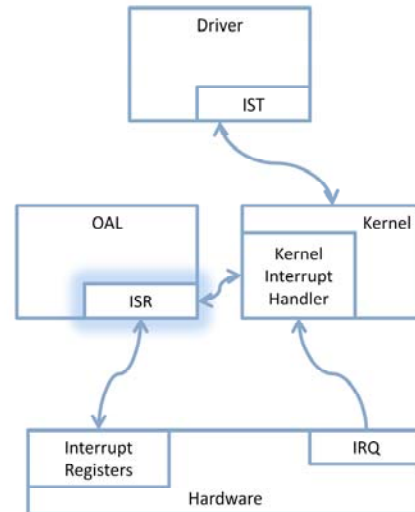
- **Interrupt Request (IRQ)**
 - Hardware identifier indicating interrupt source
 - Interpretation is specific to BSP
 - Could map to multiple SYSINTRs (shared interrupts)
- **SYSINTR ID**
 - Software identifier indicating interrupt source
 - Might be hard coded into driver (non portable)
 - Mapped to a single IRQ by the OAL
 - Associated with Event object by InterruptInitialize
 - Returned by ISR to kernel to trigger event



Interrupt Model (continued)

• Interrupt Service Routine (ISR)

- Function(s) in the OAL
- Registered to an IRQ
- Called by Kernel Interrupt Handler
- Identifies interrupt source and returns Interrupt ID to kernel
- Minimal processing; typically identify and mask interrupt
- Written to run quickly without dependencies



An interrupt service routine (ISR) is code that handles interrupt requests (IRQs) on your target device. The ISR is responsible for identifying an interrupt source, masking it, and returning a unique identifier to the Windows Embedded CE 6.0 kernel. The ISR can optionally perform other tasks that are time critical, but should be limited to those tasks that are absolutely necessary. Time spent in the ISR is time that other IRQs of lower priority are not able to be serviced.

CPU architectures that have more than one hardware IRQ require the developer to register ISR routines for each IRQ source. This is done at system initialization before interrupts are enabled. An IRQ can have only one ISR, but the same ISR can be registered for more than one IRQ. CPU architectures that have only a single IRQ source (ARM) do not need to register an ISR with the kernel.

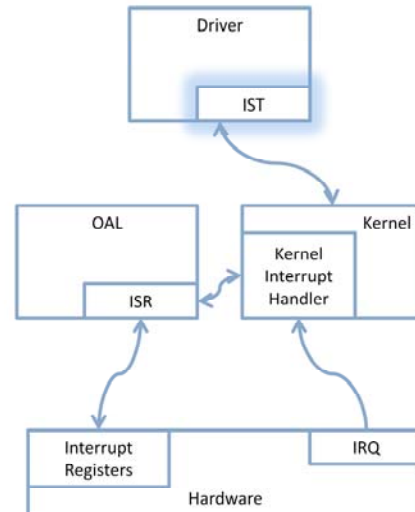
ISR landing page:

<http://msdn2.microsoft.com/en-us/library/aa930802.aspx>

Interrupt Model (continued)

- **Interrupt Service Thread (IST)**

- Created by Device Driver using CreateThread API
- Creates Event and associates with SYSINTR ID using InterruptInitialize API
- Typically IST loop blocking on event using WaitForSingleObject
- Usually runs at higher priority set by CeSetThreadPriority
- Event signaled by kernel when ISR returns corresponding Interrupt ID
- Performs bulk of processing necessary to handle interrupt
- Calls InterruptDone to unmask interrupt



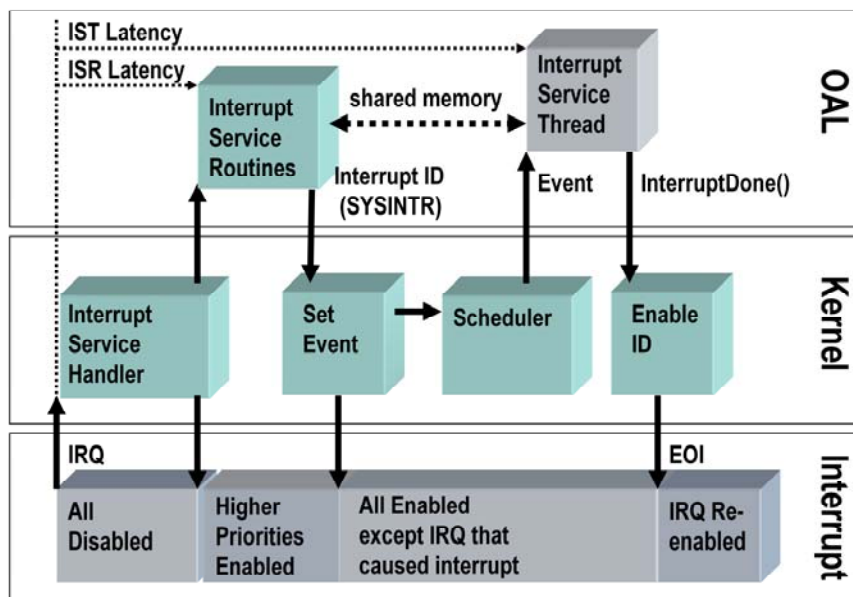
The interrupt service thread (IST) is a device driver thread that does most of the interrupt processing. The device driver associates a synchronization event object with the desired interrupt identifier and registers them with the kernel during initialization. The IST thread in the driver then waits for the event object to be signaled by the kernel, indicating that the corresponding interrupt identifier was returned by an ISR. The IST thread performs whatever processing is necessary to complete the interrupt processing, then notifies the kernel and waits for the event to be signaled again.

The IST is just a thread running inside of a driver. It typically runs at above normal priority based on the needs of the device and other system requirements. The IST is different from other threads that might be running inside the driver only in that it is handling a particular synchronization event that is registered in the kernel with an interrupt identifier.

IST processing:

<http://msdn2.microsoft.com/en-us/library/aa930165.aspx>

Interrupt Processing



Interrupt processing works as follows:

- 1) The hardware generates an interrupt request (IRQ)
- 2) The Interrupt Service Handler is the target for all interrupts and exceptions. It operates with interrupts off. The support handler sets up the stack etc.. For the "C" callable ISRs and determines the appropriate ISR to call (on ARM processors there is only one ISR)
- 3) The ISR examines the hardware to determine if it is a valid interrupt and returns the logical ID for the interrupt (SYSINTR_xxx) or SYSINTR_NOP. If shared interrupts are allowed for this IRQ then the ISR MUST call NKCallIntChain() which will return the appropriate logical ID or SYSINTR_CHAIN if no device claims the interrupt. The ISR typically disables the interrupt for this IRQ at the interrupt controller to prevent further interrupts until processing is completed.
- 4) The interrupt support handler looks up the SYSINTR in an internal table to see if there is an event associated with that ID. If there is it sets that event so the scheduler may schedule the IST waiting on it when the IST is the highest priority runnable thread.
- 5) The interrupt support handler re-enables interrupts for all interrupts
- 6) The system continues; once the IST associated with the IRQ is the highest priority runnable thread the scheduler switches to it. To process the Interrupt
- 7) The IST exits it's WaitForSingleObject() call on the interrupt event and processes the interrupt. It should minimally clear or disable the interrupt at the device then call InterruptDone() before further processing.
- 8) InterruptDone() re-enables the IRQ at the interrupt controller for other interrupts to occur. This is why it should be called as soon as possible since other devices sharing the interrupt are blocked.
- 9) The IST continues processing and clears and re-enables the interrupt at the device and goes back to wait for another interrupt.

Interrupt Model (continued)

- **Shared Interrupt Support**
 - Several devices share the same interrupt line
 - Multiple ISRs chained to handle the shared interrupt
 - Each ISR, in turn, determines if it owns the interrupt, hence implying priority
 - If owner - ISR returns the appropriate SYSINTR value or SYSINTR_NOP if no further processing is necessary
 - If not owner - The ISR return SYSINTR_CHAIN to cause NKCallIntChain to call the next ISR in the chain

Shared interrupts:

<http://msdn2.microsoft.com/en-us/library/aa929742.aspx>

The ISR routine hooked to an interrupt in [OEMInit](#) must call [NKCallIntChain](#), a kernel function, to examine a list of installed ISRs for the interrupt that has been signaled.

If the first ISR determines that its associated device has asserted the interrupt:

It performs any necessary work, and then returns the SYSINTR mapped to the interrupt.

- or -

If the ISR decides that no further processing by the IST is necessary, it returns SYSINTR_NOP.

If the ISR determines that its associated device has not asserted the interrupt, it returns SYSINTR_CHAIN, which causes **NKCallIntChain** to call the next ISR in the chain.

Interrupt Model (continued)

- **Installable ISRs**
 - Allows a driver to install an ISR at run time
 - Supports driver installation after image creation
 - OAL agnostic to device(s) using IRQ
 - OALs ISR must support interrupt chaining on that IRQ by calling NKCallIntChain
 - Loaded into kernel with LoadIntChainHandler
 - Typically used for shared interrupts
 - Supports custom functionality in ISR
 - Example: High speed serial port driver
 - All code must be contained in the DLL and cannot have dependent DLLs
 - Cannot link implicitly to other DLLs
 - NOMIPS16CODE=1
 - Must disable C run time library
 - NOLIBC=1

Windows Embedded CE also supports the concept of installable ISRs. Installable ISRs have the advantage of being loaded dynamically at run time. This allows device drivers to install a custom ISR themselves without depending on some other entity to build that ISR in to the kernel. Installable ISRs still require that there be a static ISR built into the kernel registered for the IRQ. The static ISR is responsible for notifying the kernel to walk the list of installable ISRs, and returning any interrupt identifier that comes out. However the static ISR may not have any knowledge of the devices that are hooked to the IRQ. This gives the OEM the opportunity to support any future device that might be installed to use the IRQ.

Installable ISRs are often used to support interrupt sharing, where a single IRQ is used to support multiple external devices. One example of interrupt sharing and installed IRQs is the PCI bus. The PCI bus is an expansion bus that has an interrupt specification allowing multiple devices to use the same IRQ. In addition, virtually any kind of hardware could be implemented on the PCI bus, making it impossible to write a static ISR that is aware of all the possible devices. PCI drivers use installable ISRs to overcome this limitation. The driver notifies the kernel to use a particular function in association with a particular IRQ, and the interrupt identifier that should be associated with it. The static ISR handler in the OAL calls into the installed ISRs, and the first one to recognize the device as their own returns the associated interrupt identifier. The static ISR returns that to the kernel, which notifies the IST just like any other driver.

Shared interrupts:

<http://msdn2.microsoft.com/en-us/library/aa929742.aspx>

Interrupt Model (continued)

- **Microsoft supplied generic installable ISR (GIISR)**
 - Suitable for many devices
 - Reads register to determine interrupt status
 - Configurable
 - Register/Port address
 - Register/Port size
 - Memory vs. IO
 - Mask

Interrupt Model (continued)

● Interrupt Related APIs

API	Description
HookInterrupt	Registers an ISR with the kernel for a specific IRQ line value.
InterruptDisable	This function disables a hardware interrupt as specified by its interrupt identifier.
InterruptDone	This function signals to the kernel that interrupt processing has been completed.
InterruptInitialize	Initializes an interrupt with the kernel allowing a driver to register an event and enable the interrupt.
InterruptMask	This function masks hardware interrupts.
INTERRUPTS_ENABLE	This function enables and disables all interrupts based on the argument and returns the current state.
INTERRUPTS_OFF	This function disables all interrupts.
INTERRUPTS_ON	This function enables all interrupts.
ISRHandler	This function prototype is used by an OEM/IHV to create and export an installable interrupt handler.
KernelLibIoControl	This function is called from a driver to communicate with an interrupt handler.
SetInterruptEvent	This function allows a device driver to cause an artificial interrupt event.
UnhookInterrupt	This function deregisters an ISR with a specific hardware interrupt.
LoadIntChainHandler	Called by a driver to install an ISR for an interrupt chain into the kernel.
NKCallIntChain	Called by the OAL to support shared and installable interrupts.

Operating System Internals

- System Architecture
- Memory Model
- Lab 3.1 – Using Memory Tools
- Processes and Threads
- Lab 3.2 – Exploring Threads with Kernel Tracker
- Synchronization Objects
- Lab 3.3 – A Look at Synchronization
- Interrupt Model
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Operating System Components

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- **Module 4: Operating System Components**
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification
- Course Review



Operating System Components

- **The File Systems**
- **The Registry**
- **Lab 4.1 – Using the Remote Registry Editor**
- **Power Management**
- **Lab 4.2 – Experimenting with Power Management**
- **Internationalization**
- **Review**



Operating System Components

• The File Systems

- A number of file system drivers “out of the box”
 - Several types of FAT
 - Utility APIs
 - RAM (Object Store)
 - CD/UDFS
 - BinFS
 - RELFSD
- Support for 3rd party file systems
- Support for databases including SQL-CE
- Support for file system filters
- Robust security features
- Single hierarchical namespace rooted at “\” with RAM FS or optional FS mounted at the root
- No drive letters; no current directory
- Managed by FSDMGR.DLL



UDFS - The Universal Disc File System (UDFS) and the Compact Disc File System (CDFS) are used to read compact discs (CDs), digital video discs (DVDs), and CD-ROMs. For navigating and audio/video playback, CDFS uses the ATAPI block driver, and UDFS uses the USB block driver or the ATAPI block driver.

BinFS - The binary ROM Image File System is a file system that reads the binary image (.bin) file format generated by Romimage.exe. The .bin file format organizes data into specific sections. Each section contains a section header that specifies the starting address, length, and checksum values for that section. Romimage.exe writes data organized by logical sections, such as an application's text or .data region, to the .bin file.

You can create your own specialized file system. For example, you can use installable file systems to take advantage of special functionality provided by a new type of storage hardware or to restrict what you can do with the files on standard PC Card storage hardware.

<http://msdn2.microsoft.com/en-us/library/aa914412.aspx>

Extended File Allocation Table (exFAT) is a new file system that better adapts to the growing needs of mobile personal storage. The exFAT file system not only handles large files such as those used for media storage, it enables seamless interoperability between desktop PCs and devices such as portable media devices so that files can be easily copied between desktop and device. In addition, exFAT can be adopted with minimal effort; exFAT encapsulates standard FAT and TFAT functionality.

The exFAT system offers the following advantages:

Enables the file system to handle growing capacities in media, increasing capacity to 32GB and larger

Handles more than 1000 files in a single directory

Speeds up storage allocation processes

Removes the previous file size limit of 4GB

Supports interoperability with future desktop operating systems

Provides an extensible format, including OEM-definable parameters to customize the file system for specific device characteristics

In addition, you can choose to add support for TFAT to your exFAT implementation to ensure transaction-safe operations. As of Windows Embedded CE 6.0 and later, TFAT can only be supported in an exFAT environment.

The File Systems (continued)

• FAT Utility APIs

API	Description
DefragVolume	Defragments files on a volume and removes any free space in fragmentation. It calls scan disk first to verify volume is free from errors.
DefragVolumeUI	Defragments a volume according to the options specified. It contains a dialog box user interface that can be displayed standalone or invoked from another place such as a control panel window.
FormatVolume	Formats a volume according to the options specified.
FormatVolumeUI	Formats a volume according to the options specified. It contains a dialog box user interface that can be displayed standalone or invoked from another place such as a control panel window.
ScanVolume	Scans a volume for errors in the FAT and directories, and for lost clusters according to the options specified.
ScanVolumeUI	Scans a volume according to the options specified. It contains a dialog box user interface (UI) that can be displayed alone or invoked from another place, such as a control panel window.

<http://msdn2.microsoft.com/en-us/library/aa911938.aspx>

This table shows a list of FAT related utility APIs.

The File Systems (continued)

- **RAM Object Store**
 - The File System is used to access RAM based files
 - The Registry is similar to the desktop registry
 - The Database can be used to store small property set records, a contact list for example

The object store in Windows Embedded CE provides persistent storage for applications and their related data even when the main power supply is lost, provided there is a backup power supply. One or more memory storage chips, which typically are nonvolatile RAM chips, compose the physical object store. Although file systems, databases, and the system registry share a single memory heap, they do not necessarily reside physically in the object store. They can reside in ROM, on separately installed systems, or on an external device, such as a flash memory device. Data is created and retrieved according to the storage type, independent of the actual storage device.

The operating system uses the object store to perform the following tasks:

- Manage the stack and memory heap
- Compress and expand files as necessary
- Seamlessly integrate ROM-based applications and RAM-based data
- Both the File System and the Registry will be discussed in more detail.

The built in flat database has some limited use and is sometimes called the Property database as it stores flat property-set records. The Windows Embedded CE database (CEDB) model is that of a small, flat structure and is designed for small, efficient storage. As such, the CEDB APIs do not correspond to the Win32 database APIs. Data operations are processed within the object store or a database volume, which protects against data loss. If a Windows Embedded CE-based device loses power during a data transaction, Windows Embedded CE reverts all partial database operations to the last known good state. A file system that stores a database volume still has the ability to corrupt the volume.

The default database for Windows Embedded CE is CEDB.

Windows Embedded CE also includes support for the embedded database (EDB), which enhances the functionality of CEDB and includes support for:

- Transactions.
- Access by multiple users.
- Multiple sort orders, key properties, and databases.
- Enhanced performance, especially with larger databases.

Object Store:

<http://msdn2.microsoft.com/en-us/library/aa910544.aspx>

The File Systems (continued)

- **RAM (Object Store)**
 - Transaction based
 - Maximum size 256MB
 - Max file size 32MB
 - About 4 million objects

Object Store:

<http://msdn2.microsoft.com/en-us/library/aa910544.aspx>

The File Systems (continued)

- **Support for file system filters**
 - Like a shim file system driver
 - Examples
 - Virus scan
 - Encryption
 - Security Access
 - Compression
 - Statistics
 - Performance Caching

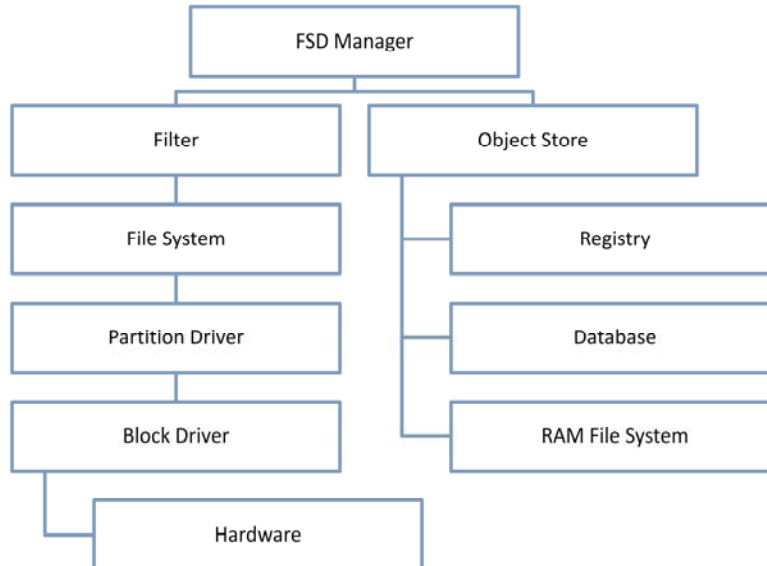
A file system filter is a dynamic-link library (DLL) that exports file system entry points. These entry points map to the standard file system functions, such as CreateFile and CreateDirectory. Because file system filters sit on top of the file system and intercept file system calls, you can use this mechanism to encrypt, compress, or virus scan any file service manager (fsdmgr) loaded file system. Multiple filters can exist on any fsdmgr loaded file system performing any combination of file manipulation before the file system sees the call.

File system filters:

<http://msdn2.microsoft.com/en-us/library/aa918566.aspx>

The File Systems (continued)

• File System Architecture



FSD Manager controls most everything related to file access, no matter where the files are stored.

“Registered File Systems”

The Object Store (RAM) File System

The ROM File System.

Release-Directory File System (RELFSD)

“Installable File Systems” - Allow the developer to extend the OS.

The internal file system in your target device controls access to ROM. The file system can also provide file storage in the object store, which is in RAM. Two internal file system options are available: the RAM and ROM file system and the ROM-only file system. These have different properties and you will want to select the correct one for your target device. Both internal file systems provide the ability to mount additional external file systems, such as file allocation table (FAT).

The RAM and ROM file system provides file storage in the object store, as well as access to the ROM. The object store is the root of the file system, and all data under the root is stored in the object store, with the exception of external file systems, which are mounted as directories under the root. Data in ROM is accessible through the Windows directory. The RAM and ROM file system is most useful in target devices that continuously power RAM because the object store is lost when RAM is not refreshed.

The ROM-only file system does not allow applications to place files in the object store. Data in ROM is accessible through the Windows directory, and external file systems are again mounted as directories under the root. Additionally, with the ROM-only file system, you have the option of choosing an external file system to be placed at the root of the file system. If you mount a file system as the root, all data below the root directory is stored in that file system, with the exception of other external file systems.

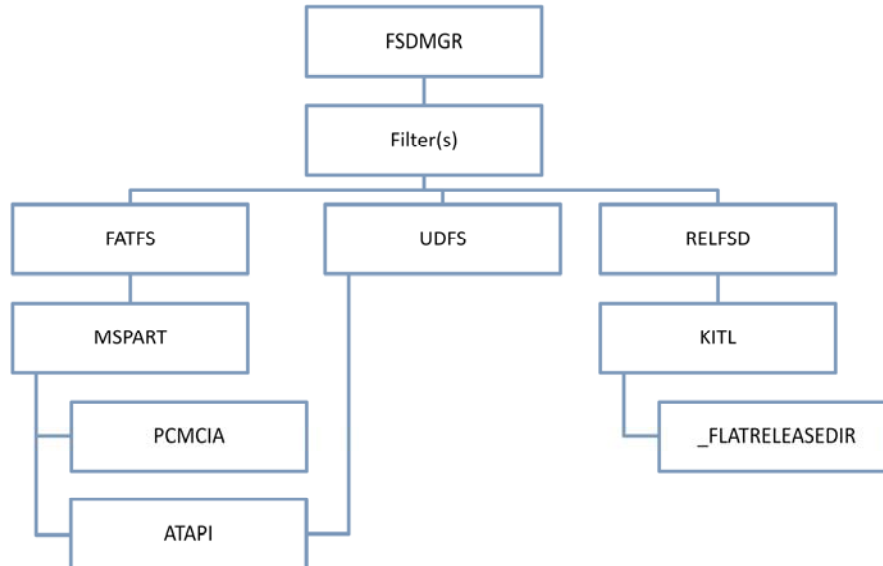
Select ROM+RAM file system if you want to use the Object Store RAM file system.

Select ROM-only file system if you don't want to use the Object Store RAM file system even if you don't want to mount is as the root of the file system.

The Object Store file system driver is implemented in FILESYS.DLL.

The File Systems (continued)

• Multiple File Systems & Filters



It is possible to have multiple file systems as well as multiple filters on a single system.

RELFS – Release-Directory File System (RELFS) This is the Platform Builder release directory file system driver for development use.

The Release Directory File System Driver (Relfsd) was created for development environments. Relfsd mounts the release directory on the development workstation (set by `_FLATRELEASEDIR`) to `\release` on the device, so that any I/O operations to the `\release` directory are routed to the `release` directory on the development workstation. In development environments, if the `LoadLibrary` function cannot find an executable module on the device; it searches for the module in the directories specified by a registry setting. By default, the directory is set to `\release` for development images.

KITL – (Kernel Independent Transport Layer) Used to communicate to between the development workstation and the target CE device.

Operating System Components

- The File Systems
- The Registry
 - Overview
 - Types
 - Hive
 - RAM
 - Registry Tools
 - Registry APIs
- Lab 4.1 - Using the Remote Registry Editor
- Power Management
- Lab 4.2 - Power Management
- Introduction to Power Management
- Review



The Registry (continued)

- **Overview**

- Similar to desktop Windows Registry
- Hierarchical trees
 - Base of each is root using well known HKEY value
 - HKEY_CLASSES_ROOT - Stores file type matching and OLE configuration data
 - HKEY_CURRENT_USER - Stores user-specific data for the user who is currently logged in (alias for user under HKEY_USERS)
 - HKEY_LOCAL_MACHINE - Stores machine-specific data for device drivers and applications
 - HKEY_USERS - Stores data for all users including a default user
 - Branches called "keys"
 - "Keys" contain subkeys or entries
 - Entries are stored as name/value pairs

The registry is a system database that stores configuration information for applications, drivers, and the operating system (OS). The registry is most commonly use for storing state information across invocations. For example, an application might have windows that a user can move and resize. Before exiting, the application could store its windows information in the registry. Then when the application starts again, it could retrieve the information and position its windows accordingly.

The basic piece of data that is stored in the registry is called a value. A value can be a variety of types, including string or binary. Each value has a name and an associated piece of data. For example, a device that is running the Windows Embedded CE Handheld PC, Professional Edition, software uses the value name Wrap to Window in the HKEY_LOCAL_MACHINE\Software\Microsoft\Pocket Word\Settings key to store an integer piece of data.

<http://msdn2.microsoft.com/en-us/library/aa912217.aspx>

The Registry (continued)

- **Types**

- **Hive based**

- Default type
- Support multiple users
- Non volatile
- Requires flush to hive

- **RAM based**

- Very simple to configure
- No support for multiple users
- Natively volatile (can be made non volatile)

There are two registry options you can use to select the registry for your target device:

Hive-based registry

RAM-based registry

In Windows Embedded CE 6.0, the registry is hive-based by default.

Hive-based

The Hive-Based Registry stores all registry data in files, also called hives, which can be located on any file system. This allows OEMs to easily persist the registry across cold boots without powering RAM.

The hive-based registry also provides separate user hives so registry configurations can be customized differently for each user. A multi-user system will contain several user hives. A user's hive can be mounted on logon and unmounted on logoff.

RAM-based

The RAM-Based Registry stores all registry data in the object store, which is in RAM. Therefore, registry data persists on warm boots but not on cold boots.

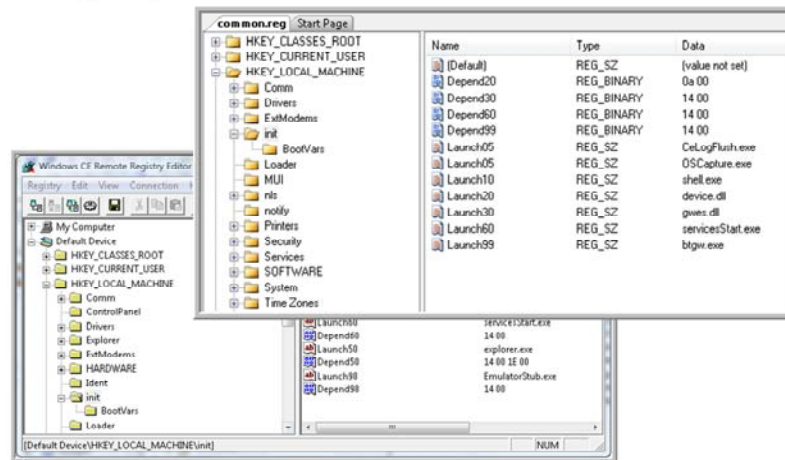
This key stores information about how the registry is configured in of all places, the registry.

HKEY_LOCAL_MACHINE\init\BootVars

<http://msdn2.microsoft.com/en-us/library/aa910532.aspx>

The Registry (continued)

- **Registry Tools**
 - Registry File Editor
 - Remote Registry Editor



<http://msdn2.microsoft.com/en-us/library/aa936059.aspx>

The Registry (continued)

● Registry Related APIs

API	Description
ReadGenericData	This function is used to read OS password data.
ReadRegData	Reads a registry file into RAM from persistent storage as defined by the OEM.
RegCloseKey	This function releases the handle of the specified key.
RegCopyFile	This function saves a copy of the current Windows Embedded CE RAM-based registry to a specified file.
RegCreateKeyEx	This function creates the specified key. If the key already exists in the registry, the function opens it.
RegDeleteKey	This function deletes a named subkey from the specified registry key.
RegDeleteValue	This function removes a named value from the specified registry key.
RegEnumKeyEx	This function enumerates subkeys of the specified open registry key.
RegEnumValue	This function enumerates the values for the specified open registry key.
RegFlushKey	This function writes all the attributes of the specified open registry key into the registry.
RegistryOperation	This function performs a series of registry operations.

Registry related functions:

<http://msdn2.microsoft.com/en-us/library/aa914389.aspx>

The Registry (continued)

• Registry Related APIs (continued)

API	Description
RegOpenKeyEx	This function opens the specified key.
RegQueryInfoKey	This function retrieves information about a specified registry key.
RegQueryValueEx	Retrieves the type and data for a specified value name associated with an open registry key.
RegReplaceKey	Replaces the file backing a registry key and all its subkeys with another file, so that when the system is next started, the key and subkeys will have the values stored in the new file.
RegRestoreFile	Places the operating system in a state in which the registry can be replaced by the supplied file on a warm boot.
RegSaveKey	Saves the specified key and all of its subkeys and values to a new file. If the specified key is not a predefined ROOT, it backs up to the ROOT of the hKey and saves there.
RegSetValueEx	This function stores data in the value field of an open registry key.
WriteGenericData	This function is used to write OS password data.
WriteRegData	Called by the OS to transfer registry data to persistent storage as defined by the OEM.

For Windows Embedded CE development, memory usage is an important consideration and the registry is no exception. The following guidelines are based on the fact that it takes less memory to store a value than to store a key:

Keep your key depth as shallow as possible. Eliminate unnecessary subkeys.

When possible, replace subkeys with values. For example, a subkey Colors might be replaced with a value named Colors.

Store as much information in one value as possible. For example, a date value could be formatted to include the time rather than using two values.

The Registry (continued)

• Registry Notification APIs

API	Description
CeFindCloseRegChange	This function stops change notification handle monitoring.
CeFindFirstRegChange	This function creates a change notification handle and sets up initial change notification filter conditions. A wait on a notification handle succeeds when a change matching the filter conditions occurs in the specified registry key, or subkeys.
CeFindNextRegChange	This function requests that the operating system signal a change notification handle the next time it detects an appropriate change.

Registry notification functions:

<http://msdn2.microsoft.com/en-us/library/aa917049.aspx>

This table lists the registry notification functions available in Windows Embedded CE. Details on each API is available in the help system and online. Drivers and applications could use these API to monitor registry changes that would affect their functionality dynamically without coding an separate eventing mechanism.

Operating System Components

- The File Systems
- The Registry
- **Lab 4.1 – Using the Remote Registry Editor**
- Power Management
- Lab
 - Lab Goals
 1. Use the remote registry editor to explore and change the device registry
 - [Video](#)
- Inte
- Rev



Operating System Components

- The File Systems
- The Registry
- Lab 4.1 – Using the Remote Registry Editor
- **Power Management**

- Lab 4.2 – Familiarization with Power Management

- Overview of Power Manager
- Power Manager Implementation
- Power Management Diagram
- System Power States
- Device Power States
- Activity Timers
- Applications & Power Management
- Idle Power Management
- System Suspend & Resume



Power Management (continued)

- **Overview of Power Manager**
 - OS Component responsible for meeting the power needs of system
 - Application interface
 - Allows communication of power needs
 - Allows notification of power events
 - Device interface
 - Allows driver to request a change to their power state
 - Request may or may not be granted
 - Allows PM to set device power state
 - Device is responsible to perform

The Power Manager manages device power, improves overall operating system (OS) power efficiency and flexibility, provides power management for each device, and coexists with applications and drivers that do not support the Power Manager. The power manager takes input from a number of sources to determine the power state of each device under its control. The Power Manager provides both an application and a device interface.

Power Management (continued)

- **Overview of Power Manager**
 - Power Manager determines when to change system power states
 - Decision logic implemented in PDD
 - Default implementation based on activity timers
 - UserActivity timer
 - SystemActivity timer
 - Timer settings exposed to user with control panel applet

The sample Power Manager implementation defines On, UserIdle, SystemIdle and Suspend as the four system power states. When the user is actively using the system, the power state is set to On. If the user stops using the system, the power state is set to UserIdle. After a longer period of user inactivity it changes to the SystemIdle state. As long as device drivers are active, the system remains in this state. If device drivers become inactive the system changes to the Suspend state.

The UserIdle state is intended for use when the user is using the device but not actively interacting with it. For example, the user may be only looking at the display and not interacting with the system. The SystemIdle state is intended for use when the user is not directly using the system but processes are still active. For example, during a file transfer the user may consider the device to be idle, even if the system processes are actually still proceeding.

The sample Power Manager implementation makes various decisions on user and system activity based upon the UserActivity and SystemActivity timers. The time-outs for transitioning between these system power states is different when the system is on AC power from when it is only on battery power.

The sample run-time images provided with Windows Embedded CE are all AC powered. You may choose to implement a separate set of power states for use when the system is on battery power, in a cradle, and so on. You can implement these customizations by copying the sample Power Manager PDD to the platform directory and modifying it appropriately.

User Idle Timer

If runs out, system goes to System Idle (turns screen off)

Reset with User Activity (keyboard, touch)

System Idle Timer

If runs out, system suspends

Reset with User Activity (keyboard, touch) and SystemIdleTimerReset()

Power Management (continued)

- **Overview of Power Manager**
 - PM only controls drivers that are Power Manager aware
 - Drivers not required to support Power Manager
 - Devices must advertise that they support the Power Manager interface for PM control
 - Iclass registry entry with Power Manager GUID
 - AdvertiseInterface with Power Manager GUID

Power manager support is optional in a driver. Drivers must implement a set of IOCTLs defined by the Power Manager, and advertise support for the Power Manager in order to be managed.

Power Management (continued)

- **Power Manager Implementation**
 - Default implementation provide “out of the box”
 - WINCE600\PUBLIC\COMMON\OAK\DRIVERS\PM
 - MDD/PDD model
 - PDD defines the supported system power states
 - PDD contains logic that determines when/how to transition between system power states
 - OEMs can modify PDD to meet unique device requirements

Microsoft provides a default power manager implementation that is suitable for many devices. The Power Manager is implemented as an MDD/PDD driver in the PUBLIC tree. The PDD contains the logic that determines what system power states are supported and how to transition between them. You may clone the PDD and modify it to implement your own system power states if your needs are different.

Power Management (continued)

• Device State Code Sample

```

pmdevsample.c
(Unknown Scope)
case IOCTL_POWER_SET:
    if(pOutBuf != NULL
        OutBufLen == sizeof(CEDEVICE_POWER_STATE)
        pduBytesTransferred != NULL) {
        // allow a set to any state, but if requested to go to D3, go to D4 instead.
        // Since our device doesn't touch any hardware, we just set its CurrentDx
        // member to update the power setting. A real device driver would probably
        // need to touch hardware.
        LOCK(pds);
        __try {
            CEDEVICE_POWER_STATE NewDx = *(PCEDEVICE_POWER_STATE) pOutBuf;
            if (VALID_Dx(NewDx)) {
                // our sample device doesn't support D3 so turn it off instead
                if (NewDx == D3) {
                    NewDx = D4;
                }
                *(PCEDEVICE_POWER_STATE) pOutBuf = NewDx;
                pduBytesTransferred = sizeof(CEDEVICE_POWER_STATE);
                pds->CurrentDx = NewDx;
                pds->fBoostRequested = FALSE;
                pds->fReductionRequested = FALSE;
                dxErr = ERROR_SUCCESS;
            }
            DEBUGMSG(ZONE_IOCTL, (_T("%s: IOCTL_POWER_SET %u %s: passing back %u\r\n"), pszFname,
                NewDx, dxErr == ERROR_SUCCESS ? _T("succeeded") : _T("Failed"), pds->CurrentDx));
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            DEBUGMSG(ZONE_IOCTL, (_T("%s: exception in ioctl\r\n")));
        }
        UNLOCK(pds);
    }
    break;

```

A device that implements all five power states manages its power dynamically by stepping down from D0 to D1, or D1 to D2, if it has been inactive for some period of time. The device does this in stages because D2's power consumption is lower but it is much less responsive also. If the device detects activity, and it is not in D0, it will attempt to go to D0.

The driver code only keeps track of whether or not it has requested a state transition, not whether the transition has occurred. This is important, because the device might request D0 while at D2, but the Power Manager might only set it to D1 because of the current power state. On the next device activity the device would request D0 again and the Power Manager might simply leave it at D1. Keeping track of the fact that it requested a state transition would prevent further unnecessary Power Manager API calls while the device is active. The same logic applies to power state reductions due to inactivity time-outs.

Power Management (continued)

- **Applications can request notification of Power events**
 - RequestPowerNotifications/StopPowerNotifications
 - Notifications generated by Power Manager
 - Uses Point to Point Message Queues
 - Applications register for desired event type
 - System power state transitions
 - Change between AC/DC power sources
 - Resume occurred
 - Battery power status field has changed
 - Notification only, no opportunity to modify, delay or block event

Applications (or drivers) can request notification of various power management related events using the RequestPowerNotifications API. These notifications are broadcast by the Power Manager to all interested parties using point to point message queues. Applications indicate the type of event they are interested in when they register for the notifications; they do not need to handle event types they are not interested in. The available events include

System power state transitions

Change between AC and DC power source

Notification that the system has resumed from a suspend state (applications generally have no knowledge when suspend is about to occur, they can only find out after the fact).

A battery power status field has changed

These are notifications only; there is no mechanism for the application to modify, delay or block the event in question. Sometimes applications want to be notified when a suspend is about to occur so they can perform an action. The notification mechanism does not provide that capability; the best solution is to move the desired functionality into a driver since drivers do participate in the system power state change.

Power Management (continued)

- **Applications can request a system power state change**
 - SetSystemPowerState will instruct the Power Manager to change power states
 - Specify by name
 - Specify by “hint” (bitmask)
 - System power states are OEM defined, so applications may not know state names
 - Power manager chooses most appropriate state based on hint
 - Hint bitmasks defined by Microsoft for standard state types
 - Power Manager may restrict applications from entering certain system power states

In some situations applications may want to change the system power state. Applications are not assumed to know which power states are available on a given Windows Embedded CE-based device, nor are they expected to know the characteristics of the system power states that are available. Rather than calling SetSystemPowerState with an explicit state name, applications can invoke it with a bitmask describing the characteristics of the power state into which they want to transition.

The Power Manager will translate this bitmask into a specific power state. For example, an application might request a system power status change with the POWER_STATE_SUSPEND bit set. The Power Manager would then transition into Suspend or SuspendCradle, depending on whether or not the system is in a cradle at the time of the request. If the device were removed from the cradle while in the SuspendCradle state, the Power Manager would transition the system into a suspend state.

The Power Manager may restrict applications from entering certain system power states. For example, if the Power Manager is actively controlling system power states based on external inputs, it may not allow you to explicitly enter an ACRun power state when the unit is running on battery power. The default Power Manager implementation permits only applications to suspend the system.

Power Management (continued)

- **Applications can set power requirement for a device**
 - Application calls SetPowerRequirement to request a specific device maintain a minimum power state
 - Allows device to be at a higher power state than system power state would otherwise allow
 - Application should call ReleasePowerRequirement as soon as possible to allow normal power management to continue

In some situations applications may want to influence the Power Manager's administration of system power states. For example, a pager application might want to keep "COM3:" at D3 or higher, even in a suspend state, so that an incoming page will wake the system. Or a streaming audio application may want to keep the network card and audio system at full power, even when the system is on battery power and has been idle for a while. The Power Manager provides the SetPowerRequirement API to support applications that have special power management needs.

The SetPowerRequirement API allows applications to ask that the Power Manager set a lower bound on device power states. If a power requirement is in effect, the Power Manager will not allow devices to set their own power state below that specified by the requirement. When the Power Manager changes system power states, it will normally keep device requirements in force even if they maintain a device's power state at a level higher than allowed by the system power state.

Device power requirements are normally set aside when the OS suspends. During an OS suspend state, the CPU is stopped and interrupts are not serviced. If an application is using a device that might be able to operate during a suspend state, it can set the POWER_FORCE flag when it calls SetPowerRequirement. It is the responsibility of a device driver to disable itself if the OS suspends.

The Power Manager may set aside device power requirements under other circumstances as well. For example, an OEM may choose to interpret the OS power state POWER_STATE_CRITICAL flag as indicating that the battery level of the OS is critically low and all devices should be turned off.

Power Management (continued)

API	Description
DevicePowerNotify	Requests that the Power Manager change the power state of a device.
GetDevicePower	Returns the current power state for a device.
GetSystemPowerState	This function returns the current system power state currently in effect.
PowerPolicyNotify	This function notifies the power manager of the events that are necessary in order to implement a power policy created by an OEM.
RegisterPowerRelationship	Establishes dynamic parent and child or bus and client driver relationship.
ReleasePowerRelationship	Releases the HANDLE returned from RegisterPowerRelationship .
ReleasePowerRequirement	This function requests that the Power Manager release a power requirement previously set with SetPowerRequirement .
RequestPowerNotifications	Allows applications and drivers to register for power notification events.
SetDevicePower	This function sets the device power state for a device.
SetPowerRequirement	Notifies power manager that an application has a specific device power requirement.
SetSystemPowerState	This function sets the system power state to the requested value.
StopPowerNotifications	Allows applications and drivers to stop receiving power notification events.

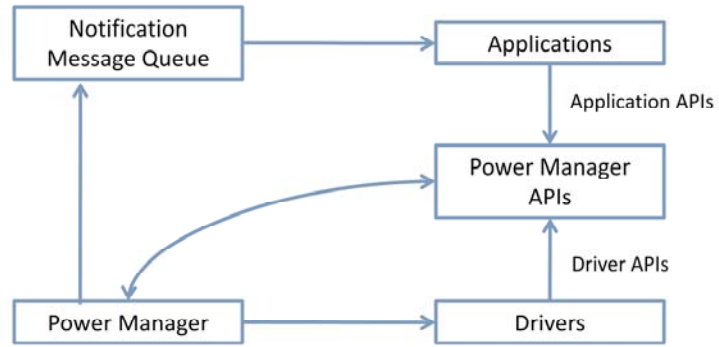
Power Management (continued)

- **Idle Power Management**
 - Kernel knows when no threads are running
 - Calls OEMIdle to put the CPU in lower power states
 - Must be supported by CPU
 - Transparent to rest of system and user
 - Independent of Power Manager
 - Small change here = big change in Standby time
 - Majority of time spent is in OEMIdle

The kernel also contributes to power management outside of the Power Manager. The kernel knows when there are no threads scheduled to run, and calls the OEMIdle function in the OAL. OEMIdle has the option to place the CPU into a lower power idle mode that can be quickly exited, assuming the CPU supports an idle mode. This is the lowest energy usage state possible balanced with the need to return from idle quickly. The idle mode is completely transparent to the user and the rest of the system.

Power Management (continued)

- **Power Management Architecture**



Power management architecture:
<http://msdn2.microsoft.com/en-us/library/aa929261.aspx>

Power Management (continued)

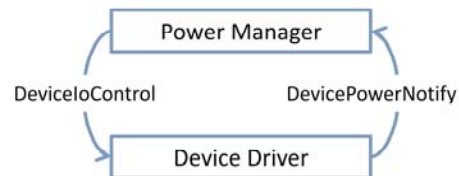
- **Device Power State Changes**
 - Drivers may not change power state unless instructed to by Power Manager
 - Drivers should manage power within a power state on their own
 - Drivers can request that the Power Manager change their state
 - Power Manager may change to requested state, another state, or no change at all
 - Drivers must not assume that their request will be honored

Drivers that are under Power Manager control can only change power states when instructed to do so by the Power Manager. Drivers can and should still adjust power within a state on their own as long as this doesn't require a power state change.

Drivers can request that the Power Manager place them into a different state. This typically occurs when the driver knows that it is able to go to a lower power state based on its current activity even though the system power state calls for a higher power device state. The Power Manager may honor that request, or it may not.

Power Management (continued)

- **Drivers communicate power needs with DevicePowerNotify**
- **Power Manager sets device state with DeviceIoControl**
 - Drivers must expose stream interface



The Power Manager uses two mechanisms to communicate with power-managed drivers. The Power Manager calls down to a device driver to determine the device's capabilities and update its device power state. Devices may call up to the Power Manager to request device power state changes. Down calls are implemented as IOCTLs. Devices call up to the Power Manager with the DevicePowerNotify API.

Because the Power Manager uses DeviceIoControl to communicate with power-managed devices, such devices must expose a stream interface. In some situations, a power management proxy may expose the interface. Network Driver Interface Specification (NDIS) exposes a stream interface that enables proxy management of NDIS miniport drivers using the RegisterPowerRelationship API. The Power Manager provides a mechanism for communicating directly with non-stream drivers. This method consists of an abstraction layer for opening a handle to a device, sending a request, and so on. A majority of devices support the stream interface, but this is not true in every instance. For example, the driver located in `Public\Common\Oak\Drivers\Pm\Mdd\Pmdisplay.cpp` implements a communication interface based on the ExtEscape function.

Opening standard device names of the format COM1:, and so on, allows access to drivers that expose a stream interface. However, the Power Manager does not require that power-manageable devices use this naming format; a device name can be any unique string. So, for example, an NDIS miniport might be named VMINI1.


Power Management (continued)

- **System Power States**
 - Named power states for entire system
 - Defined by the OEM
 - Act as global setting for device power states
 - Set maximum device power states
 - Default implementation provided by Microsoft
 - On User actively using device
 - UserIdle User passively using the device
 - SystemIdle User not using the device
 - Suspend Device powered down

The Power Manager manages device power states within the context of system power states that are defined by the OEM. System power states are described in the registry and any number can be defined. System power states impose a global upper bound on device power states.

<http://msdn2.microsoft.com/en-us/library/aa930499.aspx>

Power Management (continued)

- **Device Power States**
 - Power level of a given device
 - Device in this context means individual hardware peripheral
 - Fixed number of predefined states
 - D0 Full on Full functionality
 - D1 Low on Full functionality, reduced performance
 - D2 Standby Partial power, auto wakeup on request
 - D3 Sleep Partial power, can wake up
 - D4 Off No power
 - Devices may implement a subset of states
 - Device determines how a particular state is implemented
 - Device power states map to system power states 

The Power Manager expects all managed devices to support one or more device power states. There are a limited number of device power states, and the device may inform the Power Manager of their power consumption characteristics. Device power states generally trade off performance with power consumption.

Full on - D0 (only required state)

State in which the device is on and running. It is receiving full power from the system and is delivering full functionality to the user.

Low on - D1

State in which the device is fully functional at a lower power or performance state than D0. D1 is applicable when the device is being used, but where peak performance is unnecessary and power is at a premium.

Standby - D2

State in which the device is partially powered with automatic wakeup on request. A device in state D2 is effectively standing by.

Sleep - D3

State in which the device is partially powered with device-initiated wakeup if available. A device in state D3 is sleeping but capable of responding to an interrupt and bringing the CPU out of idle on its own. It consumes only enough power to be able to do so; which must be less than or equal to the amount of power used in state D2.

Off - D4

State in which the device has no power. A device in state D4 should not be consuming any significant power. Some peripheral busses require static terminations that intrinsically use non-zero power when a device is physically connected to the bus.

Device Power state definitions are statically predefined. The Power Manager passes a device state to a driver and the driver is responsible for mapping the state to its device capabilities and then performing the applicable state transition on its physical device.

Device power states:

<http://msdn2.microsoft.com/en-us/library/aa932261.aspx>

Power Management (continued)

- **Device powers states map to system power states**
 - Individual devices can override the default mapping via registry
 - OEM can override default mapping via registry

System Power State	Device Power State
On	D0
User Idle	D1
System Idle	D2
Suspend	D3

The following registry settings show a sample system power state to device power state mapping.

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\On]
  "Default"=dword:0      ; D0
  "Flags"=dword:10000   ; POWER_STATE_ON
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\UserIdle]
  "Default"=dword:1     ; D1
  "Flags"=dword:0
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\SystemIdle]
  "Default"=dword:2     ; D2
  "Flags"=dword:0
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Suspend]
  "Default"=dword:3     ; D3
  "Flags"=dword:200000 ; POWER_STATE_SUSPEND
; @CESYSGEN IF CE_MODULES_NDIS
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Suspend\{98C5250D-C29A-4985-
AE5F-AFE5367E5006}]
  "Default"=dword:4     ; D4
; @CESYSGEN ENDIF CE_MODULES_NDIS
```

When the system enters the Suspend state using this sample configuration, all possible wake sources are enabled with the exception of NDIS miniports. If a device does not support D3, it should automatically enter D4 instead.

Power states:

<http://msdn2.microsoft.com/en-us/library/aa929251.aspx>

Power Management (continued)

- **Activity Timers**

- **Named timers implemented by Power Manager**
 - Configurable in registry
 - Any number of timers can be implemented
 - General purpose mechanism that can be used by any component
 - Implemented with named events
 - Used by default Power Manager implementation
 - PM implements two timers for its own use
 - Used to determine user and system activity
 - Some OS components reset timers to indicate activity
GWES, Networking



The Power Manager also implements a general purpose timer control. The timers are configurable in the registry, any number of timers can be implemented. The timers are based on named events; the name is taken from the registry configuration. Any OS component can make use of this mechanism.

The default Power Manager implementation configures two activity timers for its own use. These timers are used to determine user activity and system activity. OS components that have knowledge of user and system activity reset these timers. The Power Manager uses this information in determining when to transition between system power states.

Activity timers:

<http://msdn2.microsoft.com/en-us/library/aa923909.aspx>

Power Management (continued)

- **Power Manager implements suspend/resume sequence**
 - Initiated in response to SetSystemPowerState()
 - Drivers participate in suspend/resume process
 - Includes drivers that are not managed by Power Manager
 - Applications can initiate suspend sequence if permitted by Power Manager implementation
 - Applications have no opportunity to block or otherwise participate in the suspend sequence
 - Suspend Path 
 - Resume Path 

The Power Manager implements the suspend sequence in response to a SetSystemPowerState call. The power manager notifies drivers to move to their low power suspend state as configured in the registry. Drivers that are not power manager aware will also be notified of the suspend sequence, but at a later point in the process.

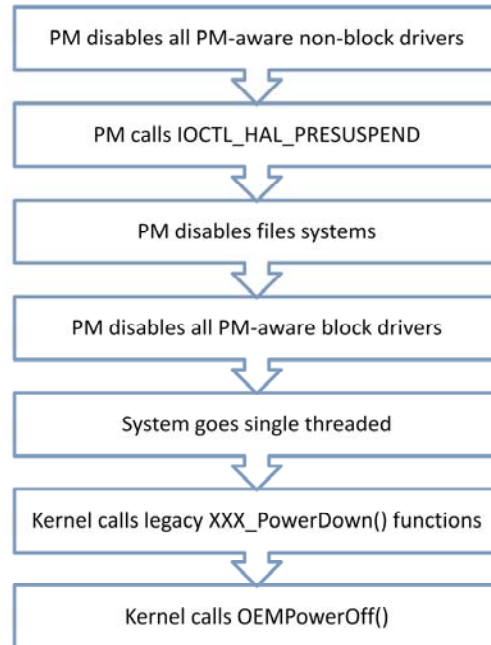
Applications can initiate a suspend sequence by calling SetSystemPower state, as long as the Power Manager implementation permits it. However, applications can not otherwise participate in the suspend process. Any functionality that needs to participate in the suspend process should be in a driver.

<http://msdn2.microsoft.com/en-us/library/aa929293.aspx>

Power Management (continued)

- **Suspend Path**

- Device goes to lowest possible power state
- Off from a user perspective, but maintains volatile memory



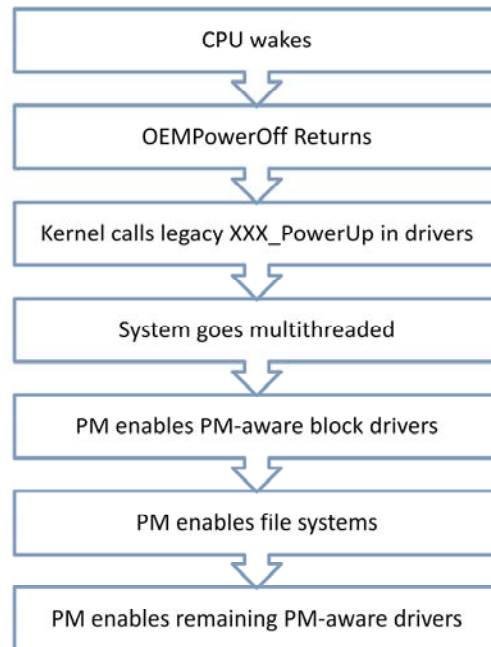
The suspend state is the lowest possible power state short of removing power from the device. This state is only possible if the CPU supports a suitable low power mode. Memory contents are maintained during suspend, allowing the device to return to exactly the state it left. The act of returning from the low power mode back to the original state is known as “resume”. Applications that are running on the device have no knowledge that this state change has occurred, unless they have requested the appropriate notifications from the Power Manager.

<http://msdn2.microsoft.com/en-us/library/aa929293.aspx>

Power Management (continued)

- **Resume Path**

- Provides “Instant on”
- Returns to state that existed prior to entering suspend
- Can be transparent to applications



The suspend state is the lowest possible power state short of removing power from the device. This state is only possible if the CPU supports a suitable low power mode. Memory contents are maintained during suspend, allowing the device to return to exactly the state it left. The act of returning from the low power mode back to the original state is known as “resume”. Applications that are running on the device have no knowledge that this state change has occurred, unless they have requested the appropriate notifications from the Power Manager.

<http://msdn2.microsoft.com/en-us/library/aa929293.aspx>

Operating System Components

- The File Systems
- The Registry
- Lab 4.1 – Using the Remote Registry Editor
- Power Management
- **Lab 4.2 – Experimenting with Power Management**

- **Lab Goals**
 1. Introduce the Windows Embedded CE 6.0 power management architecture
 2. Utilize portions of the CE 6.0 Power Management architecture
 3. Become familiar with several Power Management APIs
 4. Allow a test application to receive notification about system power events and to put power requirements into place
- [Video](#)

Operating System Components

- The File Systems
- The Registry
- Lab 4.1 – Using the Remote Registry Editor
- Power Management
- Lab 4.2 – Experimenting with Power Management
- **Internationalization**
- Review
 - Localization vs. Internationalization
 - Platform Localization Options
 - Locale Support
 - Internationalization Components



Internationalization is made up of a collection of functionality that provides general locale services and locale-specific support for certain key capabilities. Internationalization spans a range of language-specific functionality starting with code pages, keyboards and fonts.

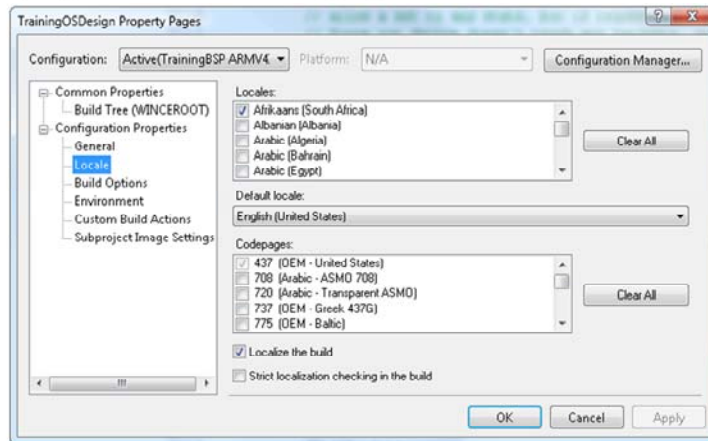
Windows Embedded CE provides general locale services for numerous code pages, and linguistic and cultural conventions through Unicode and national language support (NLS). Unicode is a universal character encoding system, while NLS carries information on date, time, calendar, number, and currency formats. NLS also provides sorting and character type information for all the locales supported by the operating system (OS). The Multilingual User Interface (MUI), functionality makes it possible for users to switch the language and locale of the user interface (UI).

In addition to general and language-specific functionality, internationalization includes support for a handwriting recognition engine that is extremely useful when working with East Asian languages. This functionality supports several East Asian language input methods (IM) and Input Method Editors (IME) that are uniquely designed for a specific language. East Asian languages require IMEs in order to input characters from a keyboard or stylus tablet.

<http://msdn2.microsoft.com/en-us/library/aa911922.aspx>

Internationalization (continued)

- **Platform Localization Options**
 - Changing the language of your operating system
 - Adding locales and selecting codepages is done from the project properties locale setting page.



Creating software that is great in many different locals in the world is an ever increasing challenge. Each market is unique, understanding those market needs and creating great software for specific area can be described as localization.

Making software that accommodates differences in language, culture, and hardware is called internationalization. The goal of internationalization is to present users with a consistent look, feel, and functionality across different language editions of a product. Users expect localized software to support the same basic functionality that the original-language edition of the product does, and they expect it to have the same level of quality. They also expect different language editions to interact smoothly with one another. Windows Embedded CE provides support for numerous character codes, as well as linguistic and cultural conventions through Unicode and national language support (NLS). Unicode is a universal character encoding system, while NLS carries information on date, time, calendar, number, and currency formats. NLS also provides sorting and character-type information for all the locales supported by the operating system (OS). In addition to character and locale codes, The international support in Windows Embedded CE includes a handwriting recognition engine that is extremely useful when working with East Asian languages and the Multilingual User Interface (MUI), functionality that makes it possible for users to switch the language and locale of the user interface (UI).

Windows Embedded CE also supports a range of language-specific technologies. These technologies include several East Asian language Input Methods (IM) and Input Method Editors (IME) that are uniquely designed for specific languages. East Asian languages require IMEs in order to input characters from a keyboard or stylus tablet. Windows Embedded CE also provides support for Complex Scripts, as well as the locales that use Complex Scripts. Windows Embedded CE includes the Unicode Script Processor to handle and process Complex Scripts.

Internationalization (continued)

- **Locale Support**

- Fonts
- Keyboard layouts and drivers
- Input Method Editors (IMEs)
- Input Methods (IMs)

Arabic	English (U.S.)
English (Worldwide)	French
German	Hebrew
Indic	Japanese
Korean	Simplified Chinese
Traditional Chinese	Thai


Windows Embedded CE provides support for a number of different locales. OEMs can also extend and customize the internationalization. The locale-specific support in Windows Embedded CE includes fonts, keyboards and keyboard drivers, Input Method Editors (IMEs), and Input Methods (IMs).

<http://msdn2.microsoft.com/en-us/library/aa913326.aspx>

Here we see the locale support “out of the box”. The exact support is based on the locale being support, typically including fonts, keyboard drivers, IMEs, and IMs. This support is extensible by OEMs.

Internationalization (continued)

• Components

- National Language Support (NLS)
- Multilingual User Interface (MUI) 
- Unicode Script Processor for Complex Scripts
- Input Method Manager (IMM)
- Handwriting Recognizer Engine (HWX)
- Keyboards and Fonts for Many Languages
- Additional language specific components

National Language Support (NLS)

`SYSGEN_CORELOC`

Adds NLS support. NLS supports the different locale-specific needs of users around the world.

Multilingual User Interface (MUI)

`SYSGEN_MULTUI`

Adds support for MUI. MUI enables you to create one run-time image for Smartphone with multiple languages, and thus allow the end-user to switch the user interface language.

Unicode Script Processor for Complex Scripts

`SYSGEN_UNISCRIBE`

Supports scripts that require special processing to show and edit because the characters are not laid out in a linear progression from left to right. Windows Embedded CE provides the correct text handling and layout for these scripts, as well as for mirroring capabilities.

Input Method Manager

`SYSGEN_IMM`

Adds Input Method Manager (IMM). IMM manages the communication between an Input Method Editor (IME) and an application.

Handwriting Recognizer Engine (HWX)

`SYSGEN_HWX`

Provides a handwriting recognition engine that supports user-drawn ideographs and characters.

<http://msdn2.microsoft.com/en-us/library/aa913456.aspx>

Internationalization (continued)

- **Multilingual User Interface**
 - Allows users to change the language of the user interface (UI)
 - Single core binary that includes the system default language
 - One resource dynamic-link library (DLL) for each additional target language

The Multilingual User Interface (MUI) allows users to change the language of the user interface (UI). To make this possible, the MUI uses a single core binary that includes the system default language, together with one resource dynamic-link library (DLL) for each additional target language. The target device boots with the system default language and then a new user-selected language goes into effect after a soft reset. This switch requires recreating windows, menus, and dialog boxes with the newly loaded resources. In addition, to be considered successful, the language switch must display these elements with the correct fonts and with the correct locale-specific information.

<http://msdn2.microsoft.com/en-us/library/aa913592.aspx>

Operating System Components

- The File Systems
- The Registry
- Lab 4.1 – Using the Remote Registry Editor
- Power Management
- Lab 4.2 – Experimenting with Power Management
- Internationalization
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

The Build System

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- **Module 5: The Build System**
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification



The Build System

- **Directory Structure of the Build Tree**
- **The Build Process**
- **The Build Tool**
- **Lab 5.1 – Static and Dynamic Libraries**
- **The Command Line**
- **Lab 5.2 – Building With the Command Line**
- **Troubleshooting a Build**
- **Lab 5.3 – Troubleshooting Link Errors**
- **Review**



The Build System

- **Directory Structure of the Build Tree**

- The Build Process

- The Build System

- Lab 5.1

- The Build System

- Lab 5.2

- Troubleshooting

- **Lab 5.3 – Troubleshooting Link Errors**

- Review

- Review

- %_WINCEROOT% (Often C:\WINCE600)
 - %_WINCEROOT%\OSDesigns
 - %_WINCEROOT%\Platform
 - %_WINCEROOT%\Public
 - %_WINCEROOT%\SDK
 - %_WINCEROOT%\Other
 - %_WINCEROOT%\Private (optional)



The Build System

- Directory Structure of the Build Tree
 - The Build Process
 - The Build Tool
 - Lab
 - The
 - Lab
 - Trou
 - Lab
 - Revi
- The output of the build process is a single run-time image consisting of many components
 - Often referred to as NK.BIN
 - NK.NBO
 - NK.SRE
 - The complete process is often described as 4 or 5 phases
 - PreSysgen
 - Sysgen
 - Post Sysgen Build
 - BuildRel
 - MakeIMG

Build system tools:

<http://msdn2.microsoft.com/en-us/library/aa908689.aspx>

The Build Process (continued)

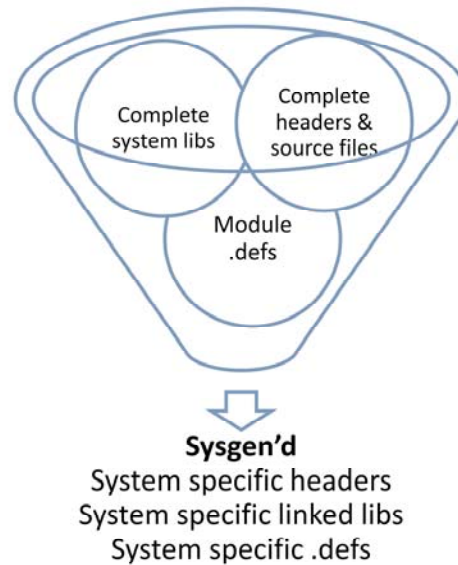
- **PreSysgen Phase**

- Typically not used
- Rebuilds installed libs
- Takes significant time
- Only needed when modifying public files
- Considered an advanced build phase

The Build Process

- **Sysgen Phase**

- Think of it as a filtering process
- Generates system specific header files, libs, .defs
- Generated from master set installed at tools installation
- Also used for configuration files
- Uses @CESYSGEN comment tags so no impact on other tools
- Sysgen'd files are per OSDesign:
 - %_PROJECTROOT%\cesysgen\
 - SDK
 - OAK
 - DDK



<http://msdn2.microsoft.com/en-us/library/aa909638.aspx>

During the OS Design generation, or Sysgen phase, the build system sets or clears Sysgen variables.

The inclusion of Sysgen variables is typically based on the selection of Catalog items that you include in your OS Design. The relationship between Sysgen variables and Catalog items can vary. A Catalog item might correspond to more than one Sysgen variable.

The build system determines which variables to set or clear by processing the Cesysgen.bat files associated with the OS Design, which are in %_WINCEROOT%\OSDesigns\\Oak\Misc.

The master Cesysgen.bat file manages a set of subordinate batch files that correspond to the dependency trees included in the OS Design. The build system uses these variables to link the corresponding static libraries into modules.

The build system also filters the header files, creating headers that only contain prototypes for the functions exported by your OS Design. Import libraries for the modules are also created during this phase.

The filtered header files and import libraries are included in the software development kit (SDK), which others can use to create applications that run on the new OS.


Also, OS Design configuration files are filtered to create a new set of configuration files specific to your OS Design, which are used in the Make Image phase.

At the end of the Sysgen phase, the board support package (BSP) is built.

Localization tasks performed during the Sysgen phase include selection of input method editors (IMEs) and fonts for Asian languages, based on the selected locales.

Environment variables and actions are executed on a per project basis, which means that environment variables you set in one project do not affect variables defined by another project.

The Build Process (continued)

- **Post Sysgen Build**
 - Build BSP
 - Uses Build.exe tool
 - Compiles and links source code in `$(_WINCEROOT)\PLATFORM\COMMON`
 - Compiles and links source code in BSP
 - Bootloader, drivers, OAL etc
 - Compiles and links source code in subprojects
 - Uses filtered OS components (Output from Sysgen)
 - Libraries, header files
 - User Projects
 - Optional Sysgen Step 

Uses Build.exe tool

Compiles and links source code in `$(_WINCEROOT)\PLATFORM\COMMON`

Compiles and links source code in BSP

Bootloader, drivers, OAL etc

Compiles and links source code in subprojects

Uses filtered OS components

Libraries, header files

The Build Process (continued)

- **Optional sysgen step**
 - Optional sysgen step can occur during this phase
 - Sysgen step applies only to local build target, not entire OS
 - Requires CESYSGEN folder with makefile
 - Typical use is for filtering configuration files
 - Remove components that are not supported by OS Design
 - Allows same BSP to be used unmodified in widely varying OS Designs
 - Used by reference BSPs provided by Microsoft

The post-sysgen build can also include a local sysgen step. The sysgen in this phase applies only to the local build target, typically the BSP. This allows the BSP to make use of the capabilities provided by the sysgen tool. The reference BSPs provided by Microsoft use the filtering capability provided by the sysgen process to selectively remove components that are not supported by the OS Design. This allows the BSP to build successfully without modification even though the OS support for a component does not exist.

BSPs can participate in this sysgen step by including the cesysgen folder in their root, containing an appropriate makefile. Most BSPs that use this capability have a simple makefile that just includes `$(_WINCEROOT)\public\common\cesysgen\CeSysgenPlatform.mak`. This makefile is configured to filter the platform configuration files.

The Build Process (continued)

- **BuildRel Phase (AKA Release Copy Phase)**
 - Copies output files to %_FLATRELEASEDIR% in preparation for Makelmg phase
 - Copies OS binaries and configuration files from the OS Design directory to the release directory
 - Copies project binaries and configuration files from the OS Design directory to the release directory
 - Copies BSP binaries and configuration files from the BSP directory to the release directory
 - This process can use hard links (default)
 - Use care when editing
 - Can override with BUILDREL_USE_COPY
 - %WINCEREL% can be used to force copy during individual component builds
 - Files in the %_FLATRELEASEDIR% are accessible through RELDIR file system driver when connected with KITL

The build process copies the OS components created during the System Generation phase and the BSP components created during the Build phase to a single directory known as the Flat Release Directory during the Build Release Directory phase (known as buildrel after the batch file that implements it).

The Buildrel tool copies files from a number of sources to a common directory in preparation for the Make Run-Time Image phase. These files include the OS components that were built during the System Generation phase as well as the BSP build phase. The contents of the following directories are copied by the buildrel tool:

```
%_PROJECTROOT%\Cesysgen\Oak\Files
%_PROJECTROOT%\Oak\Files
%_PROJECTROOT%\Cesysgen\Oak\Target\%_TGTCPU%\%WINCEDEBUG%
%_PROJECTROOT%\Oak\Target\%_TGTCPU%\%WINCEDEBUG%

%_PLATFORMROOT%\%_TGTPLAT%\Target\%_TGTCPU%\%WINCEDEBUG%
%_PLATFORMROOT%\%_TGTPLAT%\Files
%_PLATFORMROOT%\%_TGTPLAT%\cesysgen\Files
```

The WINCEREL environment variable, if set, causes the build tools to automatically binaries to the release directory when they are built. This is a performance enhancement that allows quick turnaround when recompiling BSP components. Note that you must still have performed the buildrel phase at least once in order to copy all of the OS binaries as well as configuration files to the flat release directory. You must also run the buildrel tool if you change configuration files.

The buildrel tool uses hard links instead of a normal copy to get significantly increased performance. For this reason, you should not edit files directly in the release directory. This is a shortcut that is sometimes used, but it can cause unintentional corruption of the original source file if your editor does not cause the link to be broken. These kinds of problems can be very difficult to track down. You can set the BUILDREL_USE_COPY environment variable to ensure that buildrel uses xcopy instead of hard links if you suspect problems with hard links.

The Build Process (continued)

- **MAKEIMG Phase**

- Creates the NK.BIN
 - Merges configuration files:
 - BIBs -> CE.BIB
 - REGs -> REGINIT.INI
 - DATs -> INITOBJ.DAT
 - DBs -> INITDB.INI
 - Compresses reginit.ini into binary registry file = default.fdf
 - Replaces resources in EXEs and DLLs for localization
 - Uses CE.BIB to combine modules & files into image file (NK.BIN)

In the making an image phase, the Makeimg tool performs the following steps:

1. Makeimg merges the configuration files used in the build process:
 - a. Makeimg merges all .BIB files into a unique file (CE.BIB) by using the Fmerge tool. CE.BIB identifies all the files to be combined in the image.
 - b. Makeimg merges all .reg files into a unique file (reginit.ini) by using the Fmerge tool. Reginit.ini represents all registry entries for the image.
 - c. Makeimg merges all .dat files into a unique file (INITOBJ.DAT) by using the Fmerge tool. INITOBJ.DAT provides a description of the directory and a file location for the image.
 - d. Makeimg merges all .db files into a unique file (initdb.ini) that defines default databases in the image.
2. Compresses the reginit.ini file into a binary file (DEFAULT.FDF).
3. Replaces resources in modules to adapt the image to a specific language.
The language must be specified by the LOCALE environment variable.
4. Finally, makeimg completes the Windows Embedded CE image with files and binaries specified in CE.BIB.

To make an image using the Platform Builder IDE:

- On the Build menu, click Make Image.

<http://msdn2.microsoft.com/en-us/library/aa909387.aspx>

The Build Process (continued)

- **Configuration Files**
 - BIB – Binary Image Builder Files
 - REG – Registry Files
 - DAT – Directory Specification Files
 - DB – Database Content Files

The Build Process (continued)

- **BIB – Binary Image Builder Files**

- MEMORY section (config.bib)
 - Specifies Memory Configuration

```
MEMORY
;
; NK and RAM region definitions.
;
IF INOFLASH !
#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       05ff0000 // 96mb less 4k

#define  RAMNAME     RAM
#define  RAMSTART    80070000
#define  RAMLEN      03E7F000
ENDIF ; INOFLASH

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

EFSBUF   83EEF000    00011000    RESERVED
DISPLAY  83F00000    00100000    RESERVED
```

config.bib is part of the BSP and is located in the FILES folder of the platform.

The Build Process (continued)

- **BIB – Binary Image Builder Files**

- CONFIG section (config.bib)
 - Specifies Miscellaneous Settings

```
CONFIG
; @CESYSGEN IF 'NK_NKNOCOMP'
;   COMPRESSION=ON
; @CESYSGEN ENDIF 'NK_NKNOCOMP'
; @CESYSGEN IF 'NK_NKNOCOMP'
;   COMPRESSION=OFF
; @CESYSGEN ENDIF 'NK_NKNOCOMP'

KERNELFIXUPS=ON

; Multi-Region
;

IF IMGFLASH !
;   AUTOSIZE=ON ; AUTOSIZE is used to enable
ENDIF
DLLADDR_AUTOSIZE=ON

;AUTOSIZE_ROMGAP=10000
;AUTOSIZE_DLLADDRGAP=0
;AUTOSIZE_DLLDATAADDRGAP=0
;AUTOSIZE_DLLCODEADDRGAP=0

IF IMGPROFILER
;   PROFILE=ON
ELSE
;   PROFILE=OFF
ENDIF

IF IMGFLASH
ROMSTART=88000000
ROMSIZE= 06000000
ROMWIDTH=32
ENDIF ; IMGFLASH
```

The Build Process (continued)

- **BIB – Binary Image Builder Files**
 - MODULES section (many BIBs)
 - XIP Modules

```
MODULES
: Name Path Memory Type
: -----
: @CESYSGEN IF CE_MODULES_NK
IF IMGNOKITL
nk.exe $( _FLATRELEASEDIR)\oal.exe NK SHZ
ENDIF IMGNOKITL
IF IMGNOKITL !
IF IMGNOKITLDLL
nk.exe $( _FLATRELEASEDIR)\oalkitl.exe NK SHZ
ENDIF IMGNOKITLDLL
IF IMGNOKITLDLL !
nk.exe $( _FLATRELEASEDIR)\oal.exe NK SHZ
kitl.dll $( _FLATRELEASEDIR)\kitl.dll NK SHZ
ENDIF IMGNOKITLDLL !
ENDIF IMGNOKITL !
```

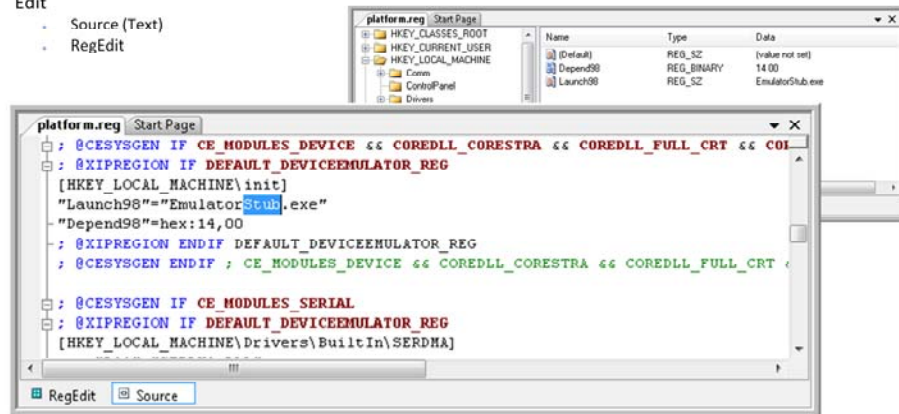
The Build Process (continued)

- **BIB – Binary Image Builder Files**
 - FILES section (many BIBs)
 - Non XIP Modules
 - Data Files

```
FILES
  ceconfig.h      $(_FLATRELEASEDIR)\ceconfig.h      NK
[ ]: @CESYSGEN IF COREDLL_CORELOC
  L
  wince.nls      $(_FLATRELEASEDIR)\wince.nls      NK SHU
  : @CESYSGEN ENDIF
[ ]: @CESYSGEN IF FILESYS_FSMAIN
  L
  initobj.dat    $(_FLATRELEASEDIR)\initobj.dat    NK SH
  : @CESYSGEN ENDIF
[ ]: @CESYSGEN IF FILESYS_FSREG
  L
  default.fdf    $(_FLATRELEASEDIR)\default.fdf    NK SH
  : @CESYSGEN ENDIF
[ ]: @CESYSGEN IF FILESYS_FSREGHIVE
  L
  hoot.hv       $(_FLATRELEASEDIR)\hoot.hv       NK SH
```

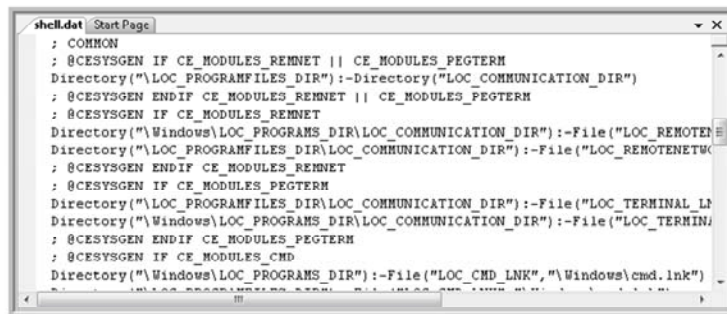

The Build Process (continued)

- REG – Registry File
 - COMMON.REG, IE.REG, WCEAPPS.REG...
 - Define registry settings for default modules
 - PLATFORM.REG
 - Platform dependant registry settings
 - PROJECT.REG
 - Project specific registry settings
 - Subproject can have their own registry settings
 - Edit
 - Source (Text)
 - RegEdit



The Build Process (continued)

- **DAT – Define Directory Structure File**
 - COMMON.DAT, IE.DAT, WCEAPPS.DAT...
 - Define settings for default modules
 - PLATFORM.DAT
 - Platform dependant settings
 - PROJECT.DAT
 - Project specific settings
 - Subproject can have their own DAT file



```
shell.dat Start Page
; COMMON
; @CESYSGEN IF CE_MODULES_REMNET || CE_MODULES_PEGTERM
Directory("%LOC_PROGRAMFILES_DIR"):-Directory("LOC_COMMUNICATION_DIR")
; @CESYSGEN ENDF CE_MODULES_REMNET || CE_MODULES_PEGTERM
; @CESYSGEN IF CE_MODULES_REMNET
Directory("%Windows\LOC_PROGRAMS_DIR\LOC_COMMUNICATION_DIR"):-File("LOC_REMNET")
Directory("%LOC_PROGRAMFILES_DIR\LOC_COMMUNICATION_DIR"):-File("LOC_REHOTENETW")
; @CESYSGEN ENDF CE_MODULES_REMNET
; @CESYSGEN IF CE_MODULES_PEGTERM
Directory("%LOC_PROGRAMFILES_DIR\LOC_COMMUNICATION_DIR"):-File("LOC_TERMINAL_LP")
Directory("%Windows\LOC_PROGRAMS_DIR\LOC_COMMUNICATION_DIR"):-File("LOC_TERMINAL")
; @CESYSGEN ENDF CE_MODULES_PEGTERM
; @CESYSGEN IF CE_MODULES_CHD
Directory("%Windows\LOC_PROGRAMS_DIR"):-File("LOC_CHD_LNK", "%Windows\cmd.lnk")
```

The Configuration Files: .DAT Files

- **Define directory structures**
 - All files are located in \Windows (ROM)
 - Filesystem parses information provided by .dat files to create and populate additional RAM directory structure
 - Results in multiple copies of files
- **Example:**

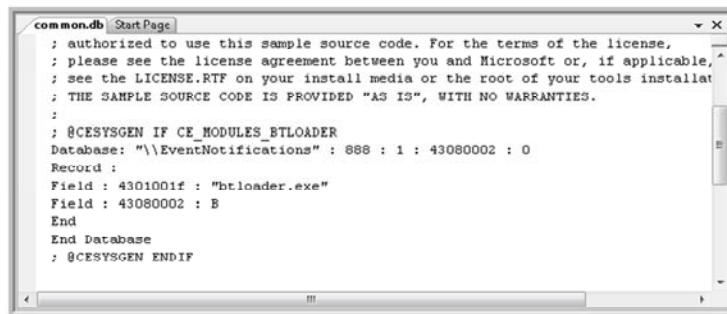
```
Root:-Directory("Program Files")
Directory("\Program Files"):-Directory("My Projects")

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

The .DAT files define folder structure of your image. In the example shown in the slide, two directories are defined, which are Program Files and My Documents. These directories are located in the root directory, which does not have any letter in Windows CE. Program Files has one subdirectory My Project. The My Documents directory contains a file MyFile.doc. Note that it is only an allocation; MyFile.doc must appear in a .BIB file to be inside the image.

The Build Process (continued)

- **DB – Define Database File**
 - COMMON.DB, IE.DB, WCEAPPS.DB...
 - Define settings for default modules
 - PLATFORM.DB
 - Platform dependent settings
 - PROJECT.DB
 - Project specific settings
 - Subproject can have their own DB file



```
common.db Start Page
; authorized to use this sample source code. For the terms of the license,
; please see the license agreement between you and Microsoft or, if applicable,
; see the LICENSE.RTF on your install media or the root of your tools installat
; THE SAMPLE SOURCE CODE IS PROVIDED "AS IS", WITH NO WARRANTIES.
;
; @CESYSGEN IF CE_MODULES_BTLOADER
Database: "\\EventNotifications" : 888 : 1 : 43080002 : 0
Record :
Field : 4301001f : "btloader.exe"
Field : 43080002 : B
End
End Database
; @CESYSGEN ENDIF
```

The Build Process (continued)

- **General Guidelines for Build Steps**

	Change in BSP Code	Change Device Memory Layout	Add New Catalog Item	Change Code in Subproject
Sysgen	No	No	Yes	No
Post-Sysgen Build	Yes	Maybe	Yes	Yes
BuildRel	Maybe	Yes	Yes	Maybe
MAKEIMG	Yes	Yes	Yes	Yes

You do not need to perform the entire build process every time you make a change in your development process. The steps that are necessary depend on the kind of change you made; the guidelines provided above are not hard rules. You must have a thorough understanding of the build process in order to make the most efficient use of your development time, which comes only with experience.

The Build Process (continued)

• Common Build Types

Debug Build	Usually includes debug information
Release Build	Build that does not include debug messages that are used during the development process; may include debugger support
Ship Build	Final build that will be shipped to customers that contains no debug code; some errors are suppressed that are displayed during the development process

During the development process, you can use the Platform Builder integrated development environment (IDE) to select one of two default build configurations for your OS Design. These configurations are called Debug and Release configurations, and offer different options.

The term debug is used to refer to a configuration which, when built, results in a run-time image that includes many debug messages and typically does not use compiler optimizations

The term release is used to refer to a configuration which, when built, results in a run-time image that usually includes compiler optimizations and few debug messages. This type of build could be used for a release candidate (RC).

The term ship refers to an optimized build that contains no debug messages. This would be used for the final RTM code ship.

Debug - Building a Debug configuration produces a very large run-time image, which has full debugging enabled. From the Build menu, select Configuration Manager, and then in the Active solution configuration field, select Debug. This sets the environment variable WINCEDEBUG=debug.

Release - Building a Release, or Retail, configuration produces a smaller run-time image, which has limited debugging enabled. Release configurations support the RETAILMSG macro, and can be configured for debugging. From the Project menu, select OS Design Properties, and then select the Build Options page. Then verify that both the Enable KITL and Enable Kernel Debugger check boxes are selected. This sets the environment variable WINCEDEBUG=retail.

Ship - Building a Ship configuration produces the final run-time image that will be provided to the customer, and which has no debugging enabled. This is the last step and may involve a social event to celebrate the successful completion of the project and RTM (Release To Manufacturing).

Major steps in creating a ship build:

Clear the Enable KITL check box, which clears the Enable CE Target Control Support check box if it is also selected. This also sets WINCEDEBUG=retail.

Verify that the Enable Kernel Debugger check box is cleared.

Select the Enable Ship Build check box. This sets WINCESHIP=1

The Build Process (continued)

- **Speeding up the Build Process**
 - Development Workstation
 - Fast Hard Drive
 - No Virus Scanning on %_WINCEROOT%
 - No File Indexing on %_WINCEROOT%
 - Defrag
 - Development Process
 - Understand the Steps
 - Do not Pre-Sysgen
 - Do not run Sysgen unnecessarily
 - Use targeted builds

The Build System

- Directory Structure of the Build Tree
- The Build Process
- **The Build Tool**
- Lab 5.1 – Static and Dynamic Libraries
 - BUILD.EXE
 - DIRS
 - SOURCES
 - SOURCES.CMN
 - MAKEFILE.DEF
- Review



The Build Tool

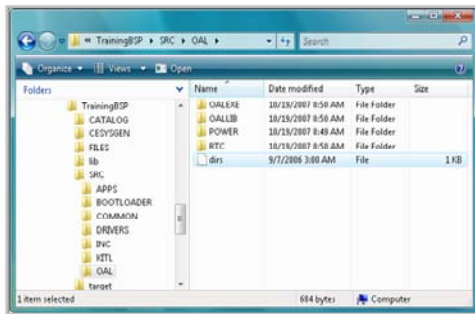
- **Build.exe**
 - Central build engine of Windows Embedded CE
 - Determines what to build
 - Controlled by DIRS and SOURCES files
 - Responsible for calling MAKE (Nmake.exe) to do the actual build
 - NMAKE uses the SOURCES file (via a makefile) to determine how to build
 - Provides automatic dependency checking for source files and include files

Build tool:

<http://msdn2.microsoft.com/en-us/library/aa909690.aspx>

The Build Tool (continued)

- **DIRS Files**
 - Text file list of “project subdirectories”
 - Supports wildcard “*”
 - Build.exe reads the list to traverse “project subdirectories”



```
dirs - Notepad
File Edit Format View Help
if 0
Copyright (c) Microsoft Corporation. All rights reserved.
endif
if 0
Use of this sample source code is subject to the license agreement under which you licensed this software. If you did not accept the terms of the license agreement, you are not authorized to use this sample source code. For more information, please see the license agreement between you and Microsoft. See the LICENSE.RTF on your install media or on the Microsoft website. THE SAMPLE SOURCE CODE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
endif

DIRS=power \
rtc \
oallib \
oalexe
```

The Build Tool (continued)

- **SOURCES Files**

- Text file providing a “macro” interface to a much more complex makefile
- MAKEFILE.DEF (%_MAKEENVROOT%) specified by local makefile
- Build.exe calls NMAKE to build projects
- Visual Studio exposes SOURCES through a GUI (Right click on subproject and select Properties)

Macro	Definition
SOURCES	List of source files to build
TARGETNAME	Name of output target without extension
TARGETTYPE	PROGRAM, DYMLINK, or LIBRARY
TARGETLIBS	Specifies additional .lib files and .obj files that should be linked into the target executable
RELEASETYPE	Indicates location for output file: LOCAL, OAK, PLATFORM...
POSTLINK_PASS_CMD	Specifies command to run after link
PRELINK_PASS_CMD	Specifies command to run before linking the final output
WINCECOD	Specifies whether assembler (.cod) files are generated during compile time
WINCEMAP	whether mapfile (.map) files are generated during compile time

Many More; see:
<http://msdn2.microsoft.com/en-us/library/aa908707.aspx>

The SOURCES file contains the component specific directives needed by build.exe and nmake. The SOURCES file is included by the shared system wide makefile called makefile.def. This allows the component makefile to remain relatively simple.. The shared system makefile is included by a simple one line makefile in the component subdirectory.

The Build Tool (continued)

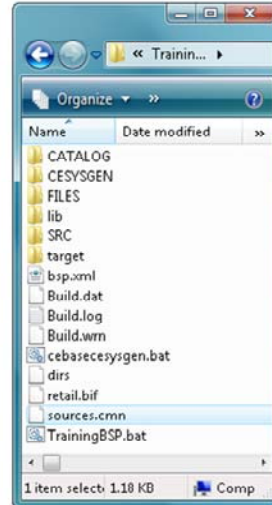
- **SOURCES Files (continued)**

- SOURCELIBS vs. TARGETLIBS

- The reason for both is a componentization feature of Windows Embedded CE. Primarily allowing stubs to be conditionally linked
 - The linker always uses the first version of a function that it finds
 - EXE - TARGETLIBS listed first to linker
 - DLL - SOURCELIBS are listed first to linker
 - LIB - Only SOURCELIBS are listed to linker

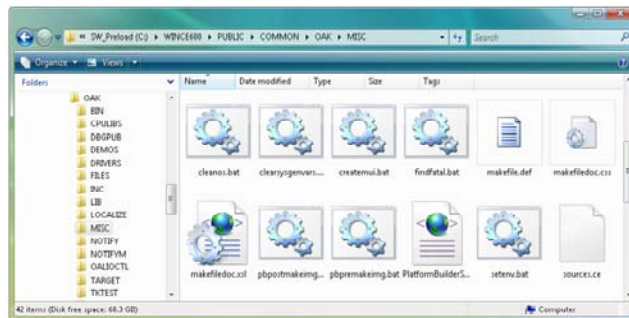
The Build Tool (continued)

- **SOURCES.CMN**
 - Common SOURCES settings applied to all SOURCES projects in a sub folder (listed in a DIRS file)
 - Placed at root of DIRS tree
 - Build.exe will walk back up directories until no parent DIRS is found and look for SOURCES.CMN in the folder with the top level DIRS file
 - Commonly used in BSPs and PUBLIC folders
 - `_COMMONPUBROOT`
 - `__PROJROOT`
 - `_ISVINCPATH`
 - `_OEMINCPATH`



The Build Tool (continued)

- **MAKEFILE.DEF**
 - Default MAKE rules for most aspects of the build
 - Provides standard rules for various types of targets
 - Reduces the amount of MAKEFILE "code" needed for most projects to the common macros in the SOURCES files
 - SOURCES file provides macro interface
 - Located in %_MAKEENVROOT%
 - %_WINCEROOT%\public\common\OAK\misc



The Build System

- Directory Structure of the Build Tree
- The Build Process
- The Build Tool
- **Lab 5.1 – Static and Dynamic Libraries**
- The Command Line
- Lab
 - Lab Goals
 1. Create simple static lib
 2. Link the static lib with a dynamic lib
 3. Link the dynamic lib with an exe
 - [Video](#)
- Review



The Build System

- Directory Structure of the Build Tree
- The Build Process
- The Build Tool
- Lab 5.1 – Static and Dynamic Libraries
- **The Command Line**
- Lab 5.2 – Building With the Command Line
- Troubleshooting
- Lab 5.3 – Building With the Command Line
- Review
 - WINCE.BAT
 - ENVIRONMENT VARIABLES
 - Setting Up Command Line Builds
 - BLDDEMO.BAT
 - Mapping IDE Commands to Command Line
 - Automated Builds



The Command Line (continued)

• WINCE.BAT

- Sets up the build environment
 - Should not normally be called directly
 - Uses these input parameters:
 - Target CPU architecture (%_TGTCPU%)
 - Target OS Design name (%_TGTPROJ%)
 - BSP name (%_TGTPLAT%)
 - Calls other batch files for additional configuration
 - %_PROJECTROOT%\%_TGTPROJ%.bat
 - %_TARGETPLATROOT%\%_TGTPLAT%.bat
 - %_WINCEROOT%\developr\%USERNAME%\setenv.bat
 - Used by the IDE as well

Wince.bat prepares the development workstation build environment by using three input parameters to determine the build environment, the location of the source files used during the build process, and the files created during the build process.

When Wince.bat is executed, it uses the following three input parameters to set the environment variables specific to the Windows Embedded CE project. These variables are used throughout the build process to build the appropriate targets.

```
%_TGTCPU%
%_TGTPROJ%
%_TGTPLAT%
```

_%_WINCEROOT% must be set before running Wince.bat. Otherwise, Wince.bat reports an error and exits. Wince.bat continues to set a series of environment variables. In addition, Wince.bat calls several batch files, which can also contain environment variables.

In addition, Wince.bat calls several other batch files which can also contribute variables to the build environment. These include

_%_TGTPLAT%.bat - Sets OS Design-dependent environment variables related to the OS Design. This must be in the %_WINCEROOT%\Platform\%_TGTPLAT% directory.

_%_TGTPROJ%.bat - Sets project-dependent environment variables. Each configuration and demonstration project folder in the Public directory contains a batch file named after the corresponding project. This must be in the %_PROJECTROOT% directory.

Setenv.bat - Sets private environment variables for the build window. This must be in the %_WINCEROOT%\Developr\%USERNAME% directory.

Wince.bat is also called by the IDE to setup the same build environment for IDE builds. There are several environment variables that the IDE sets before calling wince.bat configuring it to use the directory structure under OS Designs. If these environment variables are not properly defined, the OS Design folders will be defaulted to locations in the PUBLIC tree. In general, you should not be calling wince.bat directly so this should not be a concern. If you need to call it for some reason, examine the batch file to see how it sets up the build environment.

The Command Line (continued)

• ENVIRONMENT VARIABLES

- Variables that define OS components in OS Design
 - SYSGEN_PM Power manager
 - SYSGEN_IESAMPLE IE browser application
- Variables that provide additional control
 - BSP_xxx
 - Include specific components in BSP
 - BSP_NOxxx
 - Exclude specific components in BSP
 - IMGxxx
 - PRJ_xxx

BSP environment variables define the level of optional support available with a board support package (BSP).

There are two categories of BSP environment variables:

BSP variables are used to choose a default driver implementation for each class of device. For example, if your target device uses an RTL8139 NIC, set `BSP_NIC_RTL8139 = 1`.

BSP_NO variables are used to define options not supported by a BSP or target device. For example `BSP_NOAUDIO` - would exclude support for audio.

IMG environment variables remove modules from the image being built. These variables leave the associated registry entries in your OS Design intact. If you set or unset an IMG environment variable in your OS Design, you do not have to perform a full rebuild of your run-time image. Instead, you can simply run the Make Image tool to create the new run-time image. These variables are available for your convenience and are not for use in a shipped product. Example: `IMGNOKITL` - Selects a kernel that is not KITL-enabled.

PRJ environment variables enable project-specific functionality in your OS Design.

Example: `PRJ_BOOTDEVICE_ATAPI` - Enables ATAPI as the boot device

The Command Line (continued)

● Setting Up Command Line Builds

- Open a command shell build window within the IDE
 - Build | Open Build Release Directory... menu option
- Open a command shell build window outside the IDE
 - Use PBXmlUtils.exe
 - Can create build window batch file
 - Can open build window directly using shell extension
 - OS project must first be created from within IDE
 - Requires .pbxml file defining the OS Design

```
pbxmlutils /getbuildevn /workspace "%_WINCEROOT%\PBWorkspaces\<OS Design name>\<OS Design name>.pbxml"  
/config "<BSP>: <Target device>" > SetEnv.bat
```

There are several ways to get a command line build environment that can be used for your OS Design. You can open a build window from within the IDE using the Build menu. This option is best for day to day development, but doesn't work in automated build environments.

Platform Builder also provides an executable that will set up an appropriate build environment for you based on your OS project file (pbpxml). This option requires that you have already created the OS Design using the Platform Builder plug in for Visual Studio, and you have a pbpxml project definition. You can use the pbxmlutils.exe utility to create a build window shortcut, or to create a batch file that will setup the build environment for you.

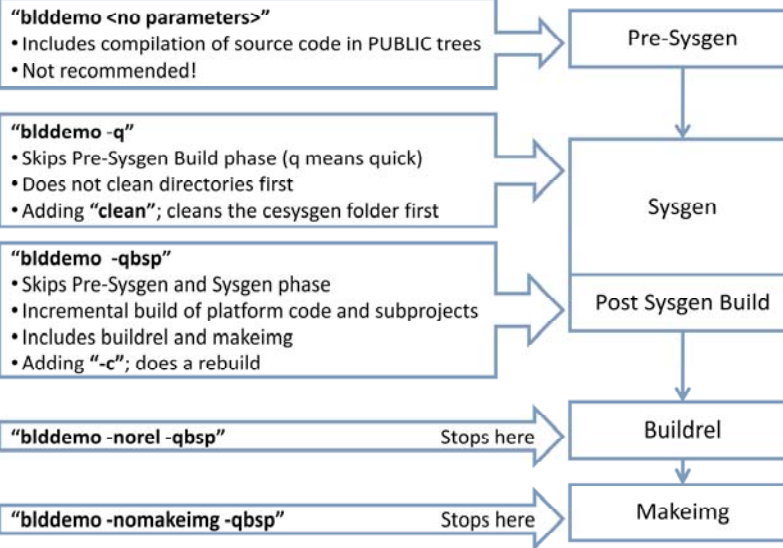
You could create the build environment yourself by calling wince.bat with the appropriate parameters, but this is not recommended.

Pbxmlutils:

<http://msdn2.microsoft.com/en-us/library/ms924887.aspx>

The Command Line (continued)

• Blddemo.bat



BldDemo.bat is the primary interface to the unified build system in Windows Embedded CE. This tool is used by the IDE to completely build the OS run-time image. It calls various other internal batch files to complete each of the build phases. Blddemo can be used in a command line environment; the unified build architecture means that builds performed from the IDE use exactly the same tools and processes as those performed from the command line.

The Command Line (continued)

• Mapping IDE Commands to Command Line

1. blddemo

2. blddemo -q

3. blddemo clean -q

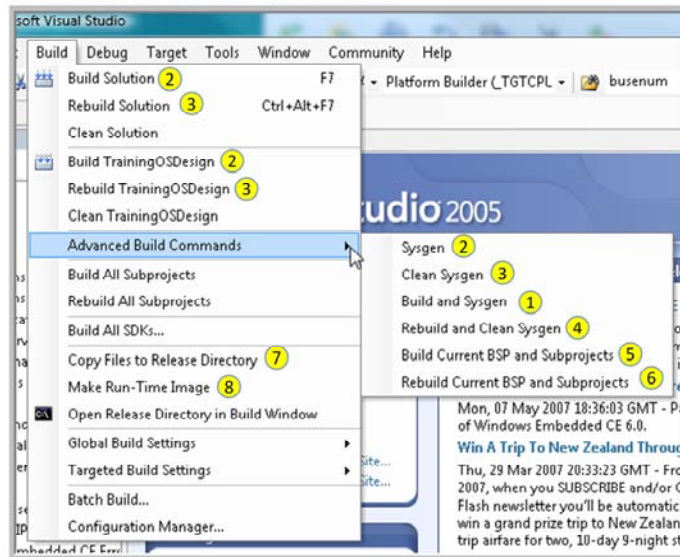
4. blddemo clean cleanplat -c

5. blddemo -qbsp

6. blddemo -qbsp -c

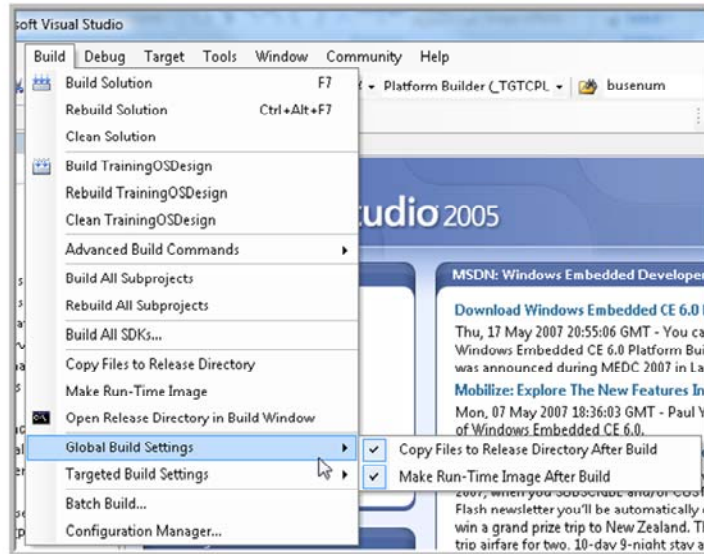
7. buildrel

8. makeimg



The Command Line (continued)

- Mapping IDE Commands to Command Line



Global settings are used for OS build selections and Targeted build settings are used for BSP, component, and other targeted build selections.

The make image phase will not run if connected to the target as the nk.bin will be in use.

The Command Line (continued)

- Mapping IDE Commands to Command Line
- Build & Rebuild apply to subprojects as well

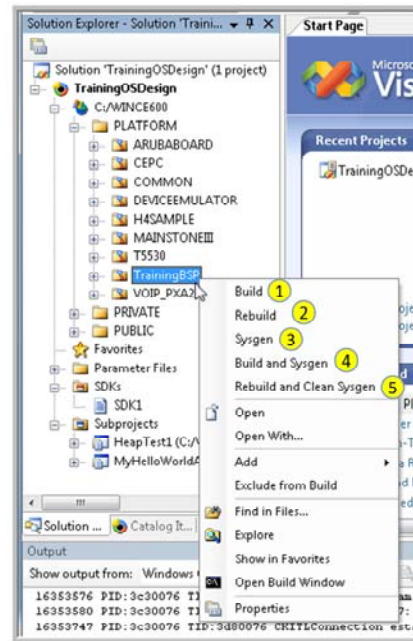
1. set wincere1=1&&build

2. set wincere1=1&&build -c

3. SysgenPlatform %_TARGETPLATROOT%
preproc&&SysgenPlatform %_TARGETPLATROOT% postproc

4. blddemo -qbsp

5. blddemo -qbsp -c



The Command Line (continued)

- **Files to Avoid Checking in to Version Control**
 - These files in the BSP tree should not be checked into your version control system
 - BSP tree is located in the %_TARGETPLATROOT% folder, i.e. %_WINCEROOT%\Platform\MyBSP
 - Opening with Platform Builder creates <OSDesign>.bif
 - Building the platform creates Build.dat, Build.log, and potentially Build.wrn and Build.err in the BSP tree root.
 - Compiling stage creates .\obj*.* in compiled folders.
 - Linking creates files in .\lib\... and .\target\... folders.
 - Platform 'sysgen' step creates .\cesysgen\files*.*

The Command Line (continued)

- **Project and OS Design Files to Avoid**
 - Output files created by sysgen
 - %_PROJECTROOT%\CeSysgen
 - %_PROJECTROOT%\SysgenSettings.out
 - %_PROJECTROOT%_CEBASE_FEATURES.txt
 - BUILDREL output
 - %_FLATRELEASEDIR%
 - SDK output
 - SDKs\<sdkname>\MSI
 - SDKs\<sdkname>\obj
 - Solution files
 - <OSDesignName>.ncb, .suo, .pbuild.user

In general, do not check in files that are outputs of the build process.

The Command Line (continued)

- **Automated Builds**
 - Save log files & console output
 - Compress & save %_FLATRELEASEDIR%
 - Can be convenient for error regressions & other testing
 - Presence of build.err or absence of nk.bin is failure

The Build System

- Directory Structure of the Build Tree
 - The Build Process
 - The Build Tool
 - Lab 5.1 – Static and Dynamic Libraries
 - The Command Line
 - **Lab 5.2 – Building With the Command Line**
 - Troubleshooting Build
 - Lab
 - Rev
- Lab Goals
 1. Learn how some of the build commands available in the Visual Studio IDE map to command line actions
 2. Compare IDE and command line build mechanisms
 - [Video](#)



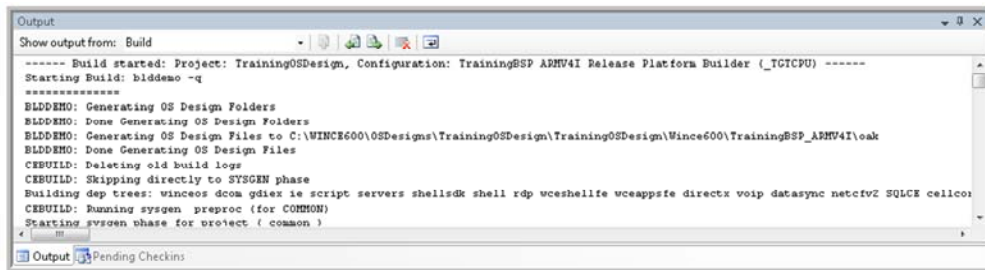
The Build System

- Directory Structure of the Build Tree
- The Build Process
- The Build Tool
- Lab 5.1 – Static and Dynamic Libraries
- The Command Line
- Lab 5.2 – Building With the Command Line
- **Troubleshooting a Build**
- Lab 5.2 – Troubleshooting Link Errors
- Review
 - Build Output Window
 - Build Log Files
 - Errors



Troubleshooting a Build (continued)

- **Build Output Window**
 - Starting the build

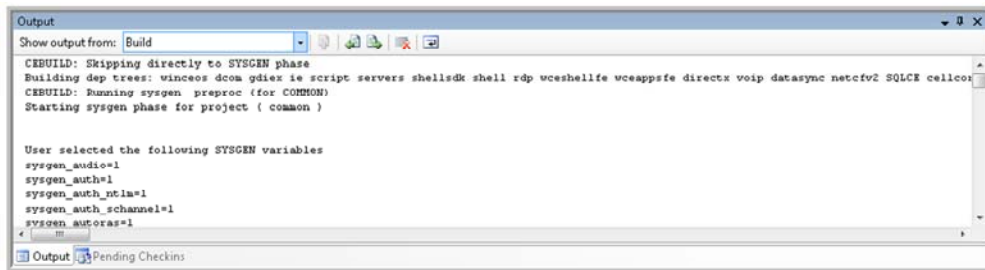


```
Output
Show output from: Build
----- Build started: Project: TrainingOSDesign, Configuration: TrainingBSP_ARMV4I Release Platform Builder (_TGTCPU) -----
Starting Build: blddemo -q
*****
BLDDemo: Generating OS Design Folders
BLDDemo: Done Generating OS Design Folders
BLDDemo: Generating OS Design Files to C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\WinCE600\TrainingBSP_ARMV4I\oak
BLDDemo: Done Generating OS Design Files
CEBUILD: Deleting old build logs
CEBUILD: Skipping directly to SYSGEN phase
Building dep trees: winceos dcom gdiex ie script servers shellsdk shell rdp wceshellfe wceappsfte directx voip datasync netctv2 SQLCE cellco:
CEBUILD: Running sysgen preproc (for COMMON)
Starting sysgen phase for protect ( common )
x [ 0% ]
Output Pending Checkins
```

The build output windows provides information for various phases of the build. Here we see the start of a build. Note that this build is running the command “blddemo -q”.

Troubleshooting a Build (continued)

- **Build Output Window**
 - Starting the sysgen phase



```
Output
Show output from: Build
CEBUILD: Skipping directly to SYSGEN phase
Building dep trees: winceos dcom gdiex ie script servers shellsdk shell rdp ucshshellfe wceappsfe directx voip datasync netctv2 SQLCE cellco
CEBUILD: Running sysgen preproc (for COMMON)
Starting sysgen phase for project ( common )

User selected the following SYSGEN variables
sysgen_audio=1
sysgen_auth=1
sysgen_auth_rt1a=1
sysgen_auth_schannel=1
sysgen_autoras=1
Output Pending Checkins
```

During the OS Design generation, or Sysgen phase, the build system sets or clears Sysgen variables.

The inclusion of Sysgen variables is typically based on the selection of Catalog items that you include in your OS Design. The relationship between Sysgen variables and Catalog items can vary. A Catalog item might correspond to more than one Sysgen variable.

The build system determines which variables to set or clear by processing the Cesygen.bat files associated with the OS Design, which are in %_WINCEROOT%\OSDesigns\

The master Cesygen.bat file manages a set of subordinate batch files that correspond to the dependency trees included in the OS Design. The build system uses these variables to link the corresponding static libraries into modules.

The build system also filters the header files, creating headers that only contain prototypes for the functions exported by your OS Design. Import libraries for the modules are also created during this phase.

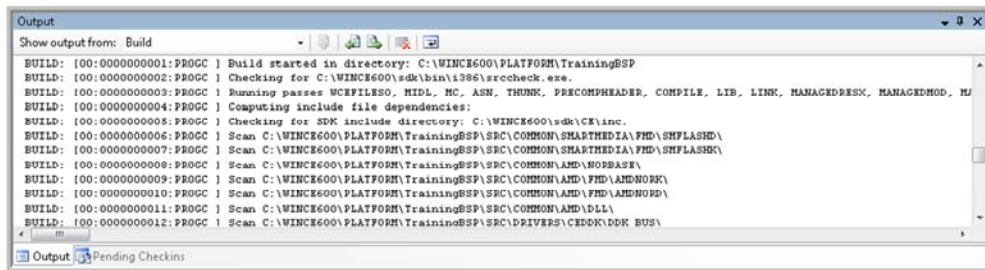
The filtered header files and import libraries are included in the software development kit (SDK), which others can use to create applications that run on the new OS.

Also, OS Design configuration files are filtered to create a new set of configuration files specific to your OS Design, which are used in the Make Image phase.

At the end of the Sysgen phase, the board support package (BSP) is built.

Troubleshooting a Build (continued)

- **Build Output Window**
 - Starting the build of a BSP

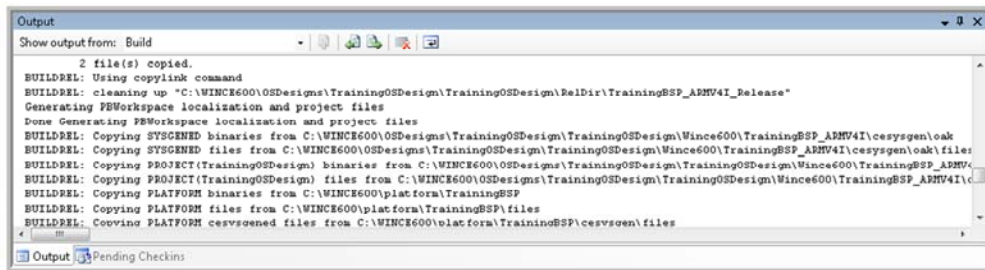


```
Output
Show output from: Build
BUILD: [00:0000000001:PROGC ] Build started in directory: C:\WINCE600\PLATFORM\TrainingBSP
BUILD: [00:0000000002:PROGC ] Checking for C:\WINCE600\sd\bin\i386\srccheck.exe.
BUILD: [00:0000000003:PROGC ] Running passes: WCFILES0, HDL, RC, ASN, THUNK, PRECOMPHEADER, COMPIL, LIB, LINK, MANAGEDRESX, MANAGEDMOD, HJ
BUILD: [00:0000000004:PROGC ] Computing include file dependencies:
BUILD: [00:0000000005:PROGC ] Checking for SDK include directory: C:\WINCE600\sd\CE\inc.
BUILD: [00:0000000006:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\COMMON\SMARTMEDIA\FMD\SMFLASHD\
BUILD: [00:0000000007:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\COMMON\SMARTMEDIA\FMD\SMFLASHD\
BUILD: [00:0000000008:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\COMMON\AMD\ROEBASE\
BUILD: [00:0000000009:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\COMMON\AMD\FMD\AMDMPK\
BUILD: [00:0000000010:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\COMMON\AMD\FMD\AMDMPK\
BUILD: [00:0000000011:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\COMMON\AMD\DLL\
BUILD: [00:0000000012:PROGC ] Scan C:\WINCE600\PLATFORM\TrainingBSP\SRC\DRIVERS\CEDOR\DDR_EUS\
Output Pending Checks
```

At the end of the sygen phase a build of the BSP starts.

Troubleshooting a Build (continued)

- **Build Output Window**
 - Starting the buildrel phase



```
Output
Show output from: Build
2 file(s) copied.
BUILDREL: Using copylink command
BUILDREL: cleaning up "C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\RelDir\TrainingBSP_ARMV4I_Release"
Generating PEWorkspace localization and project files
Done Generating PEWorkspace localization and project files
BUILDREL: Copying SYSCNED binaries from C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\Wince600\TrainingBSP_ARMV4I\cesysgen\oak
BUILDREL: Copying SYSCNED files from C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\Wince600\TrainingBSP_ARMV4I\cesysgen\oak\files
BUILDREL: Copying PROJECT(TrainingOSDesign) binaries from C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\Wince600\TrainingBSP_ARMV4I\c
BUILDREL: Copying PROJECT(TrainingOSDesign) files from C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\Wince600\TrainingBSP_ARMV4I\c
BUILDREL: Copying PLATFORM binaries from C:\WINCE600\platform\TrainingBSP
BUILDREL: Copying PLATFORM files from C:\WINCE600\platform\TrainingBSP\files
BUILDREL: Copying PLATFORM cesvsocned files from C:\WINCE600\platform\TrainingBSP\cesvsocned\files
```

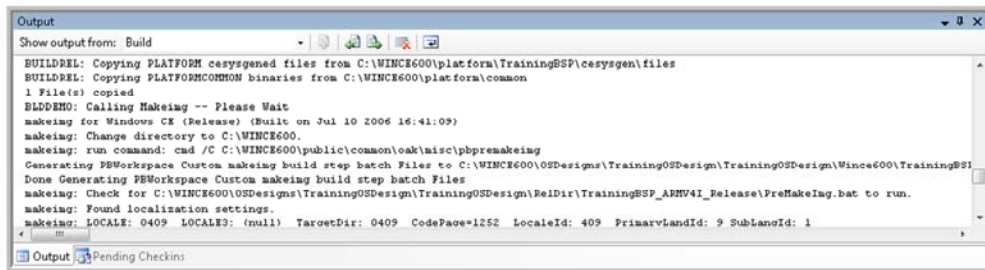
During the Release Copy phase, the build system copies all files that you need to make a run-time image to the release directory. The modules and files created during the Sysgen phase are copied to this directory first, followed by the files created in the Compile phase.

This part of the build output window is showing the building of a release directory (buildrel) phase involves copying files from the first two phases into a single directory. The net result is a collection of all the files that are to be included in the operating system image in a single place.

During the Release Copy phase, binary image builder (.bib) and registry (.reg) files are propagated to the Release directory. However, if your headers and libraries are up-to-date, this phase might not be executed. If you make changes in these files, verify that Copy Files to Release Directory After Build and Make Run-Time Image After Build are selected; then from the Build menu in Platform Builder, run Build and Sysgen to ensure propagation of the changed files.

Troubleshooting a Build (continued)

- **Build Output Window**
 - Starting the makeimg phase



```
Output
Show output from: Build
BUILDREL: Copying PLATFORM cesygened files from C:\WINCE600\platform\TrainingBSP\cesygen\files
BUILDREL: Copying PLATFORMCOMMON binaries from C:\WINCE600\platform\common
1 File(s) copied
BLDDENO: Calling Makeimg -- Please Wait
makeimg for Windows CE (Release) (Built on Jul 10 2006 16:41:09)
makeimg: Change directory to C:\WINCE600.
makeimg: run command: cmd /C C:\WINCE600\public\common\oak\misc\pbpremakeimg
Generating PEWorkspace Custom makeimg build step batch files to C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\Wince600\TrainingBSP
Done Generating PEWorkspace Custom makeimg build step batch files
makeimg: Check for C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\RelDir\TrainingBSP_ARMV4I_Release\PreMakeImg.bat to run.
makeimg: Found localization settings.
makeimg: LOCALE: 0409  LOCALES: (null)  TargetDir: 0409  CodePage=1252  LocaleId: 409  PrimaryVlanId: 9  SubLanId: 1
Output Pending Checkins
```

During the Make Run-Time Image phase, the files in the release directory are combined to create a run-time image, typically named Nk.bin.

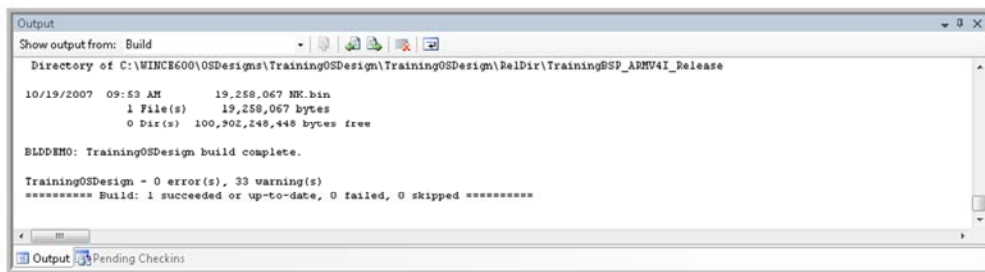
At the beginning of the phase, the project-specific files, which include Project.bib, Project.dat, Project.db, and Project.reg, are copied to the release directory.

Localization tasks performed during this phase include the attachment of resources to executables and string substitutions for configuration files, based on the selected locales.

This part of the build output window is showing the make image (Makeimg) phase involves taking all the files from the build release directory and merging the files into a single file. This single file is what you download to the reference platform hardware. This file, when packaged for downloading to a bootloader, is named NK.BIN by default.

Troubleshooting a Build (continued)

- **Build Output Window**
 - Successful build



The screenshot shows a Windows-style 'Output' window with a title bar containing 'Output' and standard window controls. The main content area displays the following text:

```
Show output from: Build
Directory of C:\WINCE600\OSDesigns\TrainingOSDesign\TrainingOSDesign\RelDir\TrainingBSP_ARMV4I_Release

10/19/2007 09:53 AM      19,258,067 MK.bin
                1 File(s)    19,258,067 bytes
                0 Dir(s)   100,902,248 bytes free

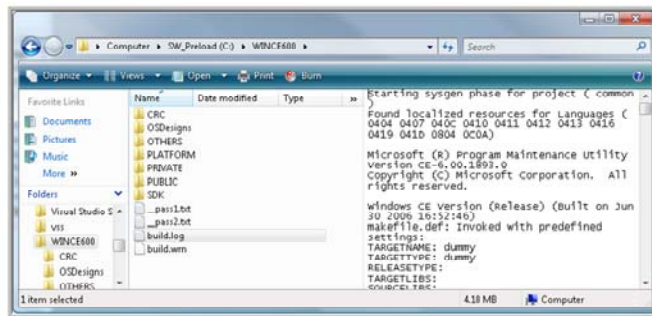
BLDDemo: TrainingOSDesign build complete.

TrainingOSDesign - 0 error(s), 33 warning(s)
***** Build: 1 succeeded or up-to-date, 0 failed, 0 skipped *****
```

At the bottom of the window, there is a status bar with a tab labeled 'Output' and a sub-label 'Pending Checksins'.

Troubleshooting a Build (continued)

- **Log Files**
 - build.log
 - build.wrn
 - build.err



The Build logs are central to troubleshooting build errors. During the build process, Build.exe generates several log files in the root directory where it was called.

Build.log - Contains a log of commands invoked by Nmake.exe.

Build.wrn - Contains a list of warnings generated during the build process.

Build.err - Contains a list of errors generated during the build process.

Note that Build.err won't be created if there aren't fatal errors, so in the screen shot above it is happy news that the file isn't present. Detecting if this file is present or not is one tool in automating build processes.

Seeing what worked and where you stopped can help you narrow in on the phase that the build error is occurring. Once you know what phase you are in you can apply the particular tips for those areas. What you would like to see at the end of the build is zero errors.

Build messages typically use the following format.

Copy CodeBUILD: [NN:SSSSSSSS:TTTTT] Message

NN - In multi-threaded builds, this specifies the ID of the thread in which the error occurred.

SS - Sequence number of the build message. This number increases with each build message.

TT - Type of message.

Message - Description

ERRORI - Error detected by Build.exe.

ERRORR - Error detected by an external program, such as Link.exe or Cl.exe.

WARNS - Serious warning detected by Build.exe.

WARNN - 'Normal' warning detected by Build.exe or an external program.

PROG - Progress.

PROGC - Console progress.

INFO - Informational message.

Troubleshooting a Build (continued)

- **Errors During the Sysgen**
 - Caused by missing files, missing configuration of the operating system features, and applications built during the Sysgen phase
- **Errors During the Module Build Phases**
 - Compilation errors or unresolved link errors
 - Syntax errors
- **Errors During the Buildrel Phase**
 - File copy errors
- **Errors During the Makeimg Phase**
 - Romimage.exe failed in CE.BIB
 - Romimage.exe failed in reginit.ini
 - Warning: Image exceeds ...

Errors During the Sysgen Phase

Errors in the Sysgen Phase are usually caused by missing files. Errors can also be caused due to missing configurations of the operating system features or applications built during the Sysgen phase. Examine build.log to determine specific problems. Determine if you have modified any components under the \Public tree.

Errors During the Build Phase

Errors in the build phase are usually compilation syntax error or unresolved link errors. Examine build.err for specific problems.

Errors During the Building a Release Directory (buildrel) Phase

In this phase, you get file copy errors. This can be due to out of disk space, locked files or read only files. Check your hard drive for enough space, check if you have a text editor open on a file in FLATRELEASEDIR and think if you've manually copied something in which could be read only. Check the build.log for more information about the errors.

Errors During the Making an Image (makeimg) Phase

Errors in this phase can be caused due to missing files in the Flat Release directory. This could be the possible result of previous errors or Bib file errors. Check the build.log or build.err to determine the specific problems.

The common Makeimg errors are:

* Romimage.exe failed in CE.BIB

This occurs due to missing files in the Flat Release directory, which have entries in one of the BIB files.

* Romimage.exe failed in reginit.ini

This error is caused due to syntax errors in the CE Registry.

* Error: Image exceeds

This error is caused by building an image that is larger than the amount of NK space you specified in config.BIB.

The Build System

- Directory Structure of the Build Tree
- The Build Process
- The Build Tool
- Lab 5.1
 - Lab Goals
 1. Identify linker errors
 2. Learn how to determine the correct link lib
 3. Resolve link errors
 - [Video](#)
- Troubleshooting a Build
- Lab 5.3 – Troubleshooting Link Errors
- Review



The Build System

- Directory Structure of the Build Tree
- The Build Process
- The Build Tool
- Lab 5.1 – Static and Dynamic Libraries
- The Command Line
- Lab 5.2 – Building With the Command Line
- Troubleshooting a Build
- Lab 5.3 – Troubleshooting Link Errors
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

The Board Support Package

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- **Module 6: The Board Support Package**
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification
- Course Review



The Board Support Package


- **BSP Overview**
- **Platform Common Code**
- **BSP Components**
- **Lab 6.1 – Registry Initialization**
- **Creating a New BSP**
- **Lab 6.2 – Adding a New IOCTL to an OAL**
- **Review**



The Board Support Package

- **BSP Overview**
- Platform Common Code
- BSP Co
- Lab 6.1
- Creatin
- Lab 6.2
- Review

- **What is a BSP?**
 - Subdirectory that contains hardware specific code
 - Bootloader
 - OAL
 - Drivers
 - Configuration files
 - 1 to 1 typical mapping between hardware platform and BSP
 - Required to build OS



A board support package (BSP) is the common name for all board hardware-specific code. It typically consists of a boot loader, OEM adaptation layer (OAL), configuration files, and board-specific device drivers

The BSP creation process involves the following tasks:

Developing a boot loader

Developing an OAL

Creating device drivers

Modifying run-time image configuration files

If you do not have a BSP, you can create a new one or clone an existing BSP that is designed for similar hardware.

If you have an existing BSP for a previous version of Windows Embedded CE, you can migrate or update it to be fully compatible with the features in Windows Embedded CE 6.0.

BSP Overview

- **Where Are BSPs?**
 - OEMs/device makers provide BSPs for their devices
 - Microsoft provides sample BSPs for each of the supported processor architectures
 - Sample BSPs are for a particular hardware reference design
 - Sample BSPs are in the Platform folder
 - Build your own or clone

The Board Support Package

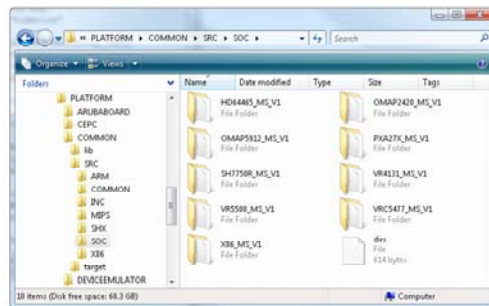
- **BSP Overview**
 - **Platform Common Code**
 - **BSP G**
 - **Lab 6**
 - **Creat**
 - **Lab 6**
 - **Revie**
- Source code for routines common to multiple BSPs
 - Promotes reuse
 - Promotes uniformity
 - Promotes modular approach
 - Exposed to BSPs as libraries
 - `_%_PLATCOMMONLIB%\<libs>`
 - Not mandatory to use common code
 - No hardware platform specific dependencies



The SOC directory is new for Windows Embedded CE 6.0. The contents of the Windows CE 5.0 CSP directory has been restructured and migrated to the SOC directory.

Platform Common Code

- **%_WINCEROOT%\Platform\Common**
 - Abort Handlers, Boot, Cache, Flash, Interrupt, IOCTLS, KITL, Log, Memory Map Support, PCI, Persistent Registry, Power, RTC, Timer, More ...
 - Support based on architecture
 - ARM, MIPS, X86, SHX
 - Support based on SOC



The Board Support Package

- **BSP Overview**
- **Platform Common Code**
- **BSP Components**
- **Lab 6.1 – Registry Initialization**
- **Creating a BSP**
 - Boot loader
 - OAL
- **Lab 6.2 – Drivers**
 - Drivers (We will discuss this in the next module)
 - Files
- **Review**



BSP Components (continued)

- **Bootloader**
 - The Bootloader is the first software component that runs on a target hardware device
 - Purpose
 - Initialize hardware
 - Perform any device specific action that needs to occur prior to booting the operating system
 - Load the operating system image into memory
 - Jump to the operating system entry point
 - Typically implemented as a separate component
 - Will have its own .bib file

The bootloader is the first software that runs on a Windows Embedded CE based device. Its primary purpose is to initialize the hardware, load the operating system into memory and jump to the operating system entry point. The bootloader can optionally perform other tasks as required by the device OEM. The bootloader is almost always implemented as a separate component (i.e. built separately from the OS). This allows the bootloader and OS to be updated independent of each other. It is possible, but not recommended, for the bootloader functionality to be incorporated directly into OAL startup code.

BSP Components (continued)

- **Boot loader (continued)**
 - Windows Embedded CE does not impose any restrictions or requirements on loader code except to call OS entry point with MMU off and hardware properly initialized
 - OEMs have unique needs for custom boot loader functionality
 - Support custom user interface for low level device management
 - Diagnostics
 - Load operating system from various resources
 - BLCOMMON library provides one possible implementation framework
 - Do not assume that a boot loader will be implemented following any particular standard

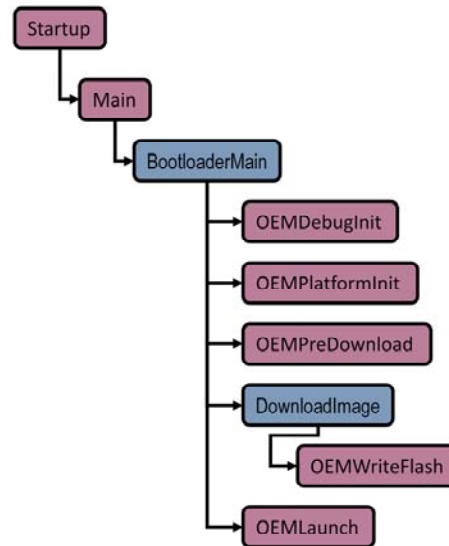
CE 6.0 does not impose any particular requirements on the bootloader except that it eventually cause the OS entry point in the kernel to be called with the MMU off and hardware properly initialized. This lack of hard requirements means that device OEMs have virtually unlimited flexibility in how they implement their bootloader. OEMs often include extended functionality such as a primitive user interface, hardware diagnostics, flash memory management functions etc. Devices that store the operating system image in flash memory often use the bootloader to update the OS image. Device OEMs can implement all of this in any manner they like. Do not assume that functionality present in one bootloader is present in another device from a different manufacturer.

BSP Components (continued)

• Boot loader (continued)

- Boot loader Framework (BLCOMMON)
 - Components located in WINCE600\PLATFORM\COMMON\SRC\COMMON\BOOT
 - Base framework simple
 - Initialize hardware
 - Call BootloaderMain()
 - Implement required callbacks
- Use other Common libraries as needed
- Supports integration with Platform Builder tools
 - Download images across debug connection
- Extensible

• BLCommon Basic Flow



The Common code includes a framework (often called BLCOMMON) that can be used to implement a Windows Embedded CE bootloader. The framework primarily supports integration with the Platform Builder tools allowing OS images to be downloaded using the IDE. The framework consists of a common bootloader function in BLCOMMON that calls back into the BSP at a number of defined points. The BSP callbacks allow the OEM to implement custom functionality specific to the device. The BSP developer can also make use of other Common libraries as appropriate (caches, PCI etc).

This slide demonstrates the basic implementation of the BLCommon bootloader. The functions in bold are some of the functions that the BSP must implement for a BLCOMMON based bootloader. The OEM has great latitude to implement platform specific functionality in the various callbacks. Note that this slide does not represent the complete functionality of the BLCommon framework.

Startup()

Low level function that initializes hardware and calls C entry point.

Main()

High level entry point. Calls BLCOMMON framework entry point, **BootloaderMain()**

BootloaderMain()

Implements bootloader control loop, calls back into BSP at defined points.

OEMDebugInit()

Initializes debug transport, usually debug serial port.

OEMPlatformInit()

High level platform initialization. Often customized to implement device specific functionality e.g. bootloader menu

OEMPreDownload()

Called prior to image download. Determine whether download should occur.

OEMWriteFlash()

Platform specific flash algorithm. Called if downloaded image needs to be written to flash.

OEMLaunch()

Launch operating system image

BSP Components (continued)

- **OAL**
 - The OAL (OEM Adaptation Layer) is the platform specific interface between the kernel and the device hardware
 - Purpose
 - Handles interrupts, timers, power management, etc.
 - Implements the operating system entry point
 - Essentially is the kernel “process” containing kernel mode DLLs
 - OAL and Kernel are not a single integrated component
 - OAL is exe; Kernel is dll
 - OAL implements a certain set of required functions and optional functions
 - Many functions - see help
 - Can use libraries from Platform\Common
 - Or even copy source and modify
 - Some libs require specific callbacks, structures, and/or variables
 - Supports single level of trust model if desired
 - Untrusted application will not run

The OEM Abstraction Layer (OAL) is the layer between the operating system kernel and the device hardware. The OAL contains the hardware specific implementations necessary to handle interrupts, timers, power management etc. The OAL “abstracts” the custom device hardware to well known interfaces exposed to the kernel. This allows the common Windows Embedded CE kernel to run on unique hardware devices.

The OAL has a set of required functions that must be implemented in order to interface with the operating system kernel. These functions can be implemented directly in the BSP, or the BSP can leverage the Common libraries. The Common libraries provide implementations for many of the required OAL functions. These implementations are included in the OAL simply by linking to the correct Common library when building OAL.EXE.

An OAL architecture that is based on the Common libraries is referred to as a Production Quality OAL or PQOAL. This does not mean that an OAL that does not leverage these libraries is not of production quality, it simply doesn't implement the PQOAL architecture. The reference BSPs provided by Microsoft use the PQOAL architecture.

Many of the implementations in the Common code have dependencies that must be resolved in the BSP. These include both functions and data structures. These BSP callbacks are the mechanism that allows the Common code to be utilized with a customized hardware implementation. These callbacks are typically easier to implement than the actual function, and make the platform specific customizations more obvious. This reduces the complexity in porting the BSP to a different hardware platform.

The Common code has a rich set of functionality that should be leveraged when creating a new BSP. However the degree of Common code use will vary depending on the BSP. It's important to note that the Common code is not one single component, but a rich set of functionality that can be leveraged as needed. There is no need to change a working BSP implementation for the sole purpose of leveraging Common code functions.

BSP Components (continued)

- **OAL Structure**
 - **OALLIB**
 - Contains BSP specific OAL functions
 - Builds as library that will eventually link with others to create OAL
 - **OALEXE**
 - Contains sources file with build instructions for OAL.EXE
 - Links OAL.LIB along other Microsoft supplied libraries to create OAL.EXE
 - OAL.EXE is renamed to NK.EXE during build process

The OAL is built in two stages. The BSP specific source code is built into a library called OAL.LIB. This library is then linked with a number of other libraries to create the final OAL.EXE component.

Note that in Windows Embedded CE6.0 only a single version of the OAL is built. There is no longer a need to build separate versions to support profiling and the Kernel Independent Transport (KITL). This is a result of the separation of the kernel from the OAL.

The OAL.EXE component is renamed to NK.EXE during the makeimg phase of the build process. NK.EXE is the traditional name of the kernel in the operating system image. Previous versions of the operating system built multiple versions of the kernel in the BSP to support combinations of kernel profiling and KITL. One of those versions was selected based on configuration options and given the common NK.EXE name.

BSP Components

• OAL APIs

Programming element	Description
InitClock	Required. This function initializes the CPU clock and set the system tick interval frequency.
OEMARMCacheMode	This function sets the cache mode used to build the ARM CPU page tables.
OEMCacheRangeFlush	This function flushes or invalidates a certain range of the cache or translation look-aside buffer (TLB). It is used by the kernel.
OEMClearDebugCommError	This function clears and initializes the serial communications port.
OEMDataAbortHandler	This function handles base-register updates and is specific to ARM processors, excluding StrongARM and XScale. It is called from the kernel when a data abort occurs.
OEMFlushCache	This function responds to a CacheSync function request. It is used by the kernel.
OEMGetExtensionDRAM	This function gets information about extension dynamic RAM (DRAM), if present on the device.
OEMGetRealTime	This function retrieves the time from the real-time clock. It is called by the kernel.

Refer to the OEM Adaptation Layer Reference in MSDN for complete OAL programming element descriptions.
<http://msdn2.microsoft.com/en-us/library/aa913478.aspx>

BSP Components

• OAL APIs (continued)

Programming element	Description
OEMIdle	This function places the CPU in the idle state when there are no threads ready to run. It is called by the kernel.
OEMInit	This function initializes all hardware interfaces for the target device. It is implemented by the OEM.
OEMInterruptDone	This function signals completion of interrupt processing.
OEMInterruptEnable	This function performs any hardware operations necessary to enable the specified hardware interrupt.
OEMInterruptHandler	This function handles interrupts. It is called by the kernel when an interrupt occurs.
OEMIoControl	This function provides a generic I/O control code (IOCTL) for OEM-supplied information.
OEMNMI	This function supports a nonmaskable interrupt. It is implemented by the OEM.
OEMNMIHandler	This function captures a nonmaskable interrupt (NMI). It is implemented by an OEM.

BSP Components

- **OAL APIs (continued)**

Programming element	Description
OEMPowerOff	This function places the CPU into a suspend state and is responsible for any final power-down state operations. It is invoked when the user presses the OFF button or when the Graphics, Windowing, and Events Subsystem (GWES) power-off timer signals that the CPU has timed out.
OEMSetAlarmTime	This function sets the real-time clock alarm. It is called by the kernel.
OEMSetRealTime	This function sets the real-time clock. It is called by the kernel.
SC_GetTickCount	This function retrieves the number of milliseconds that have elapsed since Windows Embedded CE was started. It is called from the OAL.
OEMInitDebugSerial	This function initializes the debug serial port on the target device.
OEMReadDebugByte	This function retrieves a byte from the debug monitor port.
OEMWriteDebugByte	This function outputs a byte to the debug monitor port.
OEMWriteDebugString	This function writes a string to the debug monitor port.

BSP Components

• OAL IOCTL Codes

Programming element	Description
IOCTL_HAL_INIT_RTC	This IOCTL resets the real-time clock by calling the OEMSetRealTime function.
IOCTL_HAL_POSTINIT	This IOCTL provides the OEM with a last chance to perform an action before other processes are started. It is called by the kernel and implemented in the OAL.
IOCTL_HAL_GET_DEVICE_INFO	This IOCTL returns device information using the system parameters information (SPI) codes supported by the SystemParametersInfo function.
IOCTL_HAL_DISABLE_WAKE	This IOCTL disables an interrupt source from waking the system.
IOCTL_HAL_ENABLE_WAKE	This IOCTL enables an interrupt source to wake the system.
IOCTL_HAL_GET_WAKE_SOURCE	The IOCTL returns the identifier of the wake event source that awakened the system from its most recent suspend state.
IOCTL_HAL PRESUSPEND	This IOCTL provides the OAL the time needed to prepare for a suspend operation.
IOCTL_HAL_REBOOT	This IOCTL supports a warm boot of the target device.
IOCTL_HAL_RELEASE_SYSINTR	This IOCTL releases a previously-requested SYSINTR.
IOCTL_HAL_REQUEST_IRQ	This IOCTL requests a hardware-to-IRQ mapping based on device location.
IOCTL_HAL_REQUEST_SYSINTR	This IOCTL requests an IRQ-to-SYSINTR mapping.

BSP Components

- **Files**

- Catalog File
- Platform Batch File
- Config.bib
- Platform.bib
- Platform.reg
 - Entries in platform.reg are processed last allowing them to override other settings
- Platform.dat
- Platform.db

The Board Support Package

- [BSP Overview](#)
- [Platform Common Code](#)
- [BSP Components](#)
- **Lab 6.1 – Registry Initialization**

- [Create a BSP](#)
- [Lab 6.1](#)
- [Release BSP](#)

- **Lab Goals**

1. Understand which files go into creating the initial registry
2. Understand which files have precedence in determining the initial registry

- [Video](#)



The Board Support Package

- **BSP Overview**
- Platform Common Code
- **BSP Components**
- Lab 6.1 – Registry Initialization
- **Creating a New BSP**
- Lab 6.2 – Adding a New Board
- **Review**

- Clone an Existing BSP
- BSPs included with Windows Embedded CE
- BSPs from chip manufacturers
- BSPs from other 3rd parties
- Modify as necessary
 - Boot loader
 - OAL
 - Drivers
 - Configuration files
- Get KITL working early!
- Test

Microsoft provides a BSP certification process for those interested. For more information on this program check out:

<http://msdn2.microsoft.com/en-us/embedded/bb397378.aspx>

The first step of creating a BSP is frequently to decide what BSP to use as a starting point. Once that decision is made the BSP chosen should be cloned to create the starting point for the new BSP. The level of effort could be as simple as basic modification or as complex as removing most of the functionality leaving a basic skeleton as the starting point.

The Board Support Package

- [BSP Overview](#)
- [Platform Common Code](#)
- [BSP Components](#)
- [Lab 6.1 – Registry Initialization](#)
- [Creating a New BSP](#)
- **Lab 6.2 – Adding a New IOCTL to an OAL**
- [Review](#)

- **Lab Goals**

1. Understand architecture of OAL IOCTL library in the Common code
2. Understand how to add a new IOCTL to the OAL based on the Common code

- [Video](#)



The Board Support Package

- BSP Overview
- Platform Common Code
- BSP Components
- Lab 6.1 – Registry Initialization
- Creating a New BSP
- Lab 6.2 – Adding a New IOCTL to an OAL
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Device Driver Concepts

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- **Module 7: Device Driver Concepts**
- Module 8: Customizing the OS Design
- Module 9: Application Development
- Module 10: Testing & Verification
- Course Review



Device Driver Concepts

- **An Overview of Device Drivers**
- **User Mode Driver Framework**
- **Handling Caller Buffers**
- **Loading a Stream Driver**
- **Lab 7.1 – Integrating a Device Driver**
- **Debugging Device Drivers**
- **Lab 7.2 – Debugging a Device Driver**
- **Review**



Device Driver Concepts

- **An Overview of Device Drivers**
 - User Mode Driver Framework
 - Handling I/O Requests
 - Loading and Unloading Drivers
 - Lab 7.1 – Integrating a Device Driver
 - Debugging Device Drivers
 - Lab 7.2 – Debugging a Device Driver
 - Review
- Device Drivers are software components that abstract the functionality of a physical or virtual device



Many device driver concepts are documented on MSDN:
<http://msdn2.microsoft.com/en-us/library/aa917820.aspx>

An Overview of Device Drivers

- **Drivers included in Windows Embedded CE**
 - %_WINCEROOT%\Public\Common\OAK\Drivers
 - All %_TARGETPLATROOT%\Drivers folders

Audio Drivers	Battery Drivers	Block Drivers	Bluetooth HCI Transport Driver
Direct3D Mobile Display Drivers	DirectDraw Display Drivers	Display Drivers	DVD-Video Renderer
Flash Media Drivers	HID Drivers	IEEE 1394 Drivers	Keyboard Drivers
Network Drivers	Notification LED Drivers	Parallel Port Drivers	PC Card Drivers
PCI Bus Driver	Printer Drivers	Root Bus Driver	Secure Digital Card Drivers
Serial Port Drivers	Smart Card Drivers	Stream Interface Drivers	Timer Driver
Touch Screen Drivers	USB Function Drivers	USB Host Drivers	USB Serial Host Driver

A device driver is software that abstracts physical or virtual devices from the operating system. This allows an application developer to call Win32 APIs to perform an operation on the underlying device without needing to understand the low level details of the device. A driver also allows similar devices to be exposed to applications with a common interface, even though different devices might have slightly different implementations.

For example, an application developer can transmit data across a serial port with calls `CreateFile()` on `COMx` (where `x` denotes the serial port number to be opened, for example `COM1` for serial port 1), `WriteFile()` to write some bytes of data to the serial port, and `CloseHandle()`. Other standard APIs exist to do other operations on the port including configuring its characteristics. The same sequence of APIs works for any device that is exposing itself as a serial port no matter what the underlying device actually is. It is even possible that there is no physical hardware associated with the port.

Device drivers:

<http://msdn2.microsoft.com/en-us/library/aa447514.aspx>

An Overview of Device Drivers (continued)

- **Drivers are classified in a number of different ways**
 - Layered (MDD/PDD) vs. Monolithic
 - How the driver is architected
 - Native vs. Stream
 - Who loads the driver
 - User vs. Kernel
 - Where the driver is loaded
 - Built In vs. Dynamic/Installable
 - When the driver is loaded
 - Driver Family
 - WAV/Miniport/Touch/Display/Serial etc

There are many different ways to categorize device drivers depending on the context. Drivers are differentiated based on who loads them, where they are loaded, when they are loaded, how they are architected, as well as the family of devices they control.

An Overview of Device Drivers (continued)

- **Layered (MDD/PDD) vs. Monolithic**
 - MDD/PDD or Layered Drivers
 - Model Device Driver + Platform Device Driver = Driver
 - Sample implementations provided by Microsoft
 - MDD implements functionality common to all similar devices
 - MDD implements defined architecture for specific drivers
 - MDD can often be used unmodified
 - MDD calls PDD functions for hardware specific functionality
 - PDD must be customized to match target hardware
 - Interface to PDD is sometimes called DDSI
 - Provides for code reuse, ease of implementation
 - Can be slightly less efficient due to layering

Microsoft provided layered drivers are often referred to as MDD/PDD drivers. This terminology refers to the components that make up the driver. An MDD/PDD driver model is provided by Microsoft for a number of sample or reference drivers included in Platform Builder. The MDD/PDD model is intended to ease the porting effort required to implement a driver for new hardware.

The MDD/PDD architecture contains two layers. The MDD (Model Device Driver) is a library of routines that implements functionality and features considered to be common to all drivers in a particular device class. The MDD code often can be used unmodified as long as the feature set that it supports is sufficient to meet the end user requirements.

The PDD (Platform Device Driver) layer implements that hardware specific requirements of a particular driver. The PDD layer is called by the MDD layer.

Each family of drivers that is implemented using an MDD/PDD architecture has its own set of APIs that make up the MDD/PDD for that driver family, for example there is a serial MDD/PDD model, an audio MDD/PDD model etc. The upper level interface exposed by the MDDs (in the form of specific IOCTLS) is generally expected by higher level OS components.

An Overview of Device Drivers (continued)

- **Layered (MDD/PDD) vs. Monolithic**
 - Monolithic Drivers
 - Does not leverage high level shared library of interface routines
 - Written to implement requirements as efficiently as possible
 - Can have increased or otherwise unique feature set (not dependent on MDD implementation)
 - Less code reuse, more complex development
 - Can be more efficient due to streamlined functions

Monolithic drivers do not implement the layered MDD/PDD approach. Monolithic drivers are typically written when an MDD/PDD driver model does not exist, when unique requirements can't be handled by the existing MDD implementation, and when performance needs become more important than ease of implementation. Monolithic drivers are also written when the device hardware is capable of implementing many of the features that are otherwise implemented in the MDD. This is another area where a monolithic driver can provide better performance.

Regardless of whether you implement a monolithic driver or a layered driver, you can base your implementation on the source code for any of the sample drivers.

An Overview of Device Drivers (continued)

- **Layered (MDD/PDD) vs. Monolithic**
 - Hybrid Drivers
 - MDDs are not immutable. Clone and modify as necessary
 - Modified MDDs are often used to implement missing feature requirements
 - Maintains ease of development benefits by starting with existing high level driver implementation

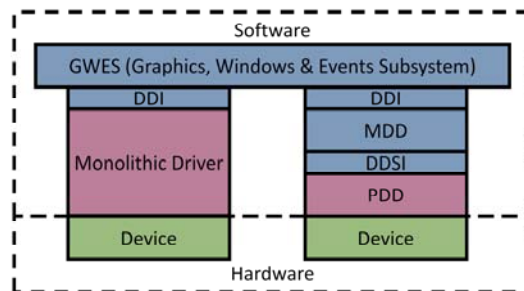
Need something a little different than the MDD/PDD sample driver offers? Consider cloning the MDD and modifying it to suit unique requirements. This allows the developer to leverage the existing code and still implement unique features of a particular device. Note that you must still expose the same minimum level of functionality at the upper edge to maintain compatibility with OS components that utilize a particular driver interface.

An Overview of Device Drivers (continued)

- **Native vs. Stream**

- **Native Drivers**

- Fixed set of native device classes defined by CE
 - Keyboard
 - Display
 - Touch
- Drivers loaded directly by GWES, not managed by device
- Each has its own model



Another way to differentiate drivers is by the way in which they are loaded and managed. There are two general driver models in this categorization: Native and Stream drivers.

Native drivers are special purpose drivers that are not managed by the Device manager component and do not have to export an API that is compatible with generic drivers. Rather, they are loaded directly by their host component (gwes.dll). The term "native" refers to the interface between GWES and the hardware. Native drivers have an interface that is unique to the hardware that they control; one native driver class will have a completely different interface than another native driver class. This native interface provides for optimal performance between the host server component (gwes) and the driver because the API is designed directly for the hardware. Notice that native drivers are all user interface related, a key responsibility of the GWE.

The native device driver model was especially important in previous versions of the operating system because GWE and Device ran in separate user processes. The native driver loading model allowed GWE to have fast access to the drivers without having to cross a process boundary.

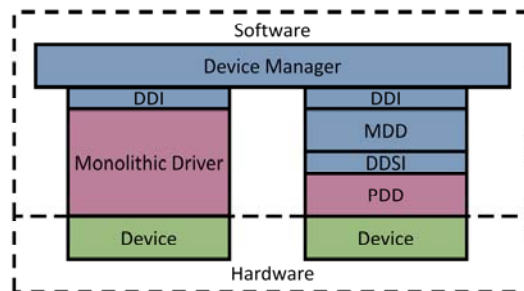
This illustration demonstrates both layered and monolithic architecture models for native drivers. Notice that GWES is the entity at the top of the slide, native drivers are loaded directly by GWES.

An Overview of Device Drivers (continued)

- **Native vs. Stream**

- **Stream Drivers**

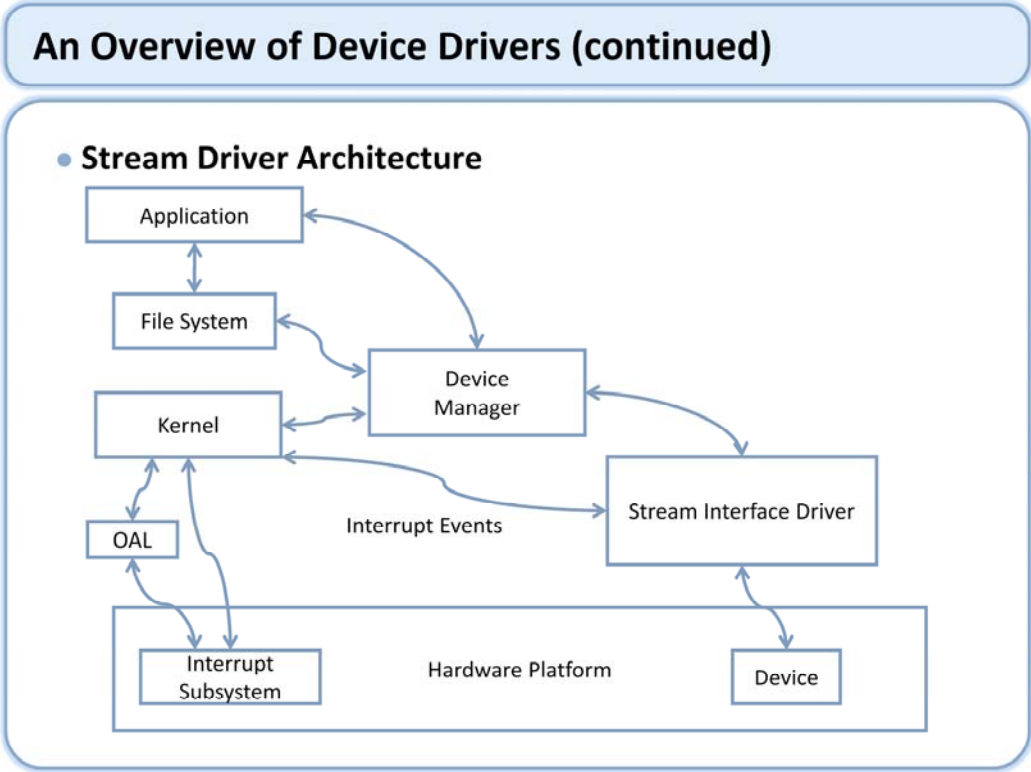
- Loaded and managed by Device Manager
 - Support any type of physical or virtual device
 - Device Manager imposes minimum interface requirements (driver entry points and behavior)
 - Stream model applies to most drivers
 - Exposed to applications through file system APIs
 - Can be loaded at boot or dynamically



Stream drivers are loaded and managed by Device manager. The device manager requires each driver under its control to implement a common interface defined by Device manager. Stream drivers can be written to control any possible type of physical or virtual device, as long as it conforms to the stream driver interface required by the device manager. This makes the device manager agnostic to the types of devices that it can support. Stream drivers can be exposed as a special file system device. Stream drivers can be named and opened with a CreateFile() call. The resulting handle can then be used to access the driver with regular file system APIs. The file system recognizes the device name as belonging to the device manager, and forwards the access on to the Device manager for proper handling.

Stream drivers have a flexible loading model. They can be loaded automatically at boot, or they can be load dynamically, on demand.

This slide demonstrates layered and monolithic stream drivers. Notice they are loaded under Device Manager – all drivers managed by Device manager are stream drivers and implement the required stream driver interface.



An Overview of Device Drivers (continued)

- **Stream Driver File Names**
 - Drivers can be accessed as special file system device
 - Three different namespaces provide different features
 - Share common 3 letter prefix
 - Legacy
 - Example: “COM1:”
 - Device
 - Example: “\Device\COM1”
 - Bus
 - Example: “\Bus\PCMCIA_0_0_0”
 - Named interface not required

Stream drivers can have names, and be accessed through file system handles. There are three different ways to access a particular named driver, providing different capabilities and access rights to the caller. The legacy device name is the original mechanism and is still supported by CE. The \Device and \Bus namespaces were added to provide additional capability and eliminate some limitations of the legacy namespace. Any named driver can be accessed using any of the three naming conventions, subject to some limitations.

Note that it is not necessary for all device drivers to have a named interface. A driver may provide a standalone function that does not require it to export an interface to applications or other drivers, or it may provide some other custom interface that does not utilize the file system interface.

An Overview of Device Drivers (continued)

- **Stream Driver File Names**
 - Legacy Namespace
 - Original naming convention
 - Used by most legacy drivers
 - Consists of Prefix followed by Index (0-9)
 - Limited to 10 instances of a particular prefix (COM0 - COM9)
 - Prefix included in driver entry points
 - Unless special registry flag is set

The legacy naming convention is a 3 letter prefix with a numeric suffix varying from 0-9, for example "COM1". This is the most common way of accessing a driver. It provides a normal file handle to the device that can be used for normal IO operations. This namespace is limited to 10 devices with a single prefix. An implementation that has more than 10 devices must either change the prefix or use the \Device namespace

Note that the Prefix (specified in the registry) must also be appended to the required stream driver entry points unless a special registry flag is set.

An Overview of Device Drivers (continued)

- **Stream Driver File Names**
 - Device Namespace
 - Similar to Legacy namespace
 - Consists of Prefix followed by number
 - Provides support for more than 10 devices
 - “\Device\COM27”

The device namespace is similar to the legacy namespace. Its primary advantage is its ability to support more than 10 devices.

An Overview of Device Drivers

- **Stream Driver File Names**
 - Bus Namespace
 - Typically used with client drivers loaded by a bus driver
 - Provides mechanism to differentiate between IO and bus control operations
 - Provides mechanism for more sophisticated driver loading/unloading/power management features

The \Device namespace provides a mechanism for more sophisticated driver level control operations. This mechanism is typically used in conjunction with bus/client drivers, where the bus driver performs various operations on behalf of the client drivers. A handle returned using the \Device namespace can have different access permissions and capabilities than a normal handle. Microsoft provides bus drivers for the most commonly used bus implementations.

An Overview of Device Drivers (continued)

• Stream Drivers - The 12 Entry Points

Function	Called When	Typical Driver Operation
XXX_Init	The driver is initialized (loaded)	Performs all initialization global to all instances; memory mapping, interrupt initialization, ...
XXX_PreDeinit	Called prior to XXX_Deinit	Marks handle as invalid, if needed, to eliminate potential race condition with multiple threads
XXX_Deinit	The driver is de-initialized (unloaded)	Cleans up from XXX_Init
XXX_Open	Application calls CreateFile	Allocates handle for use by other IO functions, allocates resources for open context, & prepares for operation
XXX_PreClose	Called prior to XXX_Close	Wakes sleeping threads, if needed, to eliminate potential race condition with multiple threads
XXX_Close	Application calls CloseHandle	Cleans up from open context
XXX_IOControl	Application calls DeviceIoControl	Performs custom driver operations using IOCTL code specific to device; this is the workhorse of most drivers
XXX_Read	Application calls ReadFile	Performs read; often not used in favor of XXX_IOControl
XXX_Write	Application calls WriteFile	Performs write; often not used in favor of XXX_IOControl
XXX_Seek	Application calls SetFilePointer	Performs seek; often not used in favor of XXX_IOControl
XXX_PowerUp	OS resumes from suspend state	Performs operations necessary to exit low power mode
XXX_PowerDown	OS going to suspend state	Performs operations necessary to go into low power mode

Note that XXX is replaced by the Prefix for the driver, e.g. COM. If the DEVFLAGS_NAKEDENTRIES flag is set in the registry for this driver then the Prefix is eliminated (e.g. use Init instead of COM_Init).

XXX_Init() and XXX_Deinit() are required for all stream drivers.

XXX_Open() and XXX_Close() are required for all stream drivers that support a named interface. If you provide a prefix entry in the registry section that loads this driver then your driver supports a named interface and requires these functions.

The XXX_PreClose and XXX_PreDeinit are optional functions provided to solve potential race conditions due to threads blocked in a driver. These functions provide a mechanism to separately invalidate handles and wake sleeping threads before XXX_Close and XXX_Deinit are called.

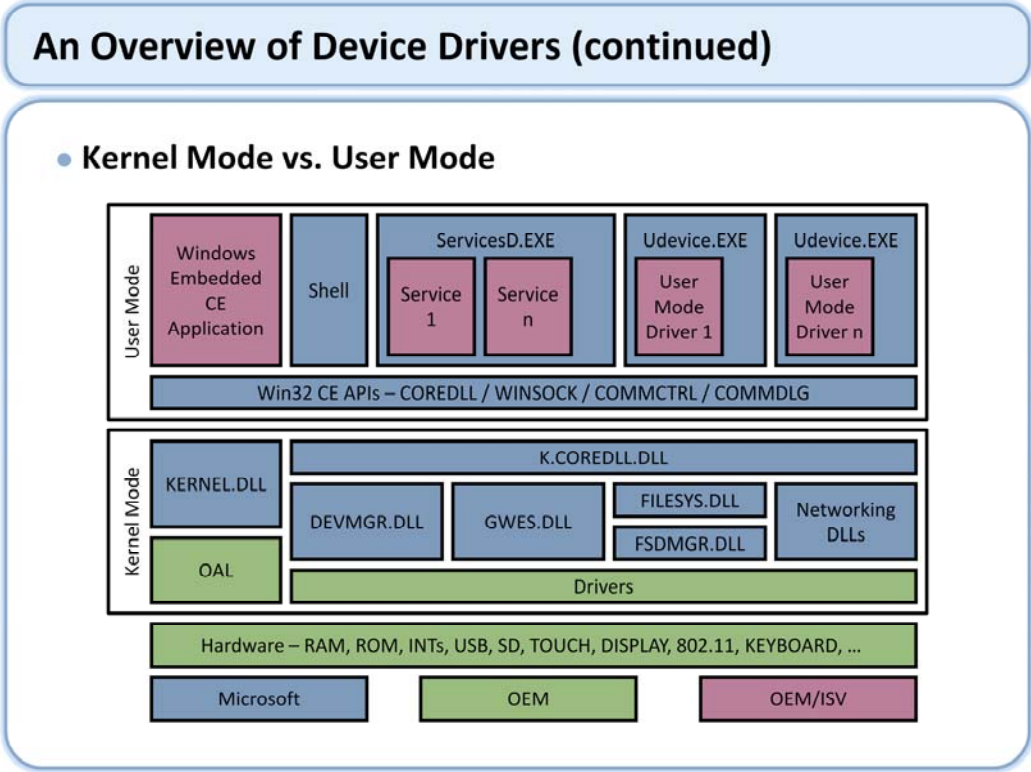
XXX_PowerDown and XXX_PowerUp are optional functions called by the Device manager when the operating system is entering and exiting its low power Suspend state. The OS is running in a limited mode when these functions execute and the types of operations and system API calls available to these functions are limited. Drivers that implement these functions should be careful to limit their scope to the minimum necessary to change the power state of the device. These functions constitute the legacy driver power management infrastructure. The Power Manager component provides more sophisticated (and complex) power management leveraging the IOControl interface.

The XXX_IOControl function is the primary interface used by most drivers. Applications and other drivers use the file handle returned by CreateFile in a call to DeviceIoControl along with a control code and driver specific data structure to communicate with the driver. This interface provide great flexibility in performing driver operations. Most drivers utilize this interface exclusively, and do not implement the Read/Write/Seek functions. If they are not implemented, the corresponding Win32 call using the device handle will fail. At least one of these functions or the IOControl function must be implemented for named devices.

An Overview of Device Drivers (continued)

- **How do you implement a Stream Driver?**
 - Select a device file name prefix
 - Implement the required entry points
 - Create the *.DEF file
 - Create the registry values necessary to load your driver

It's very easy to implement a basic stream driver that can be accessed from an application. While there are a number of infrastructure elements supported by the Device manager that could be implemented in a driver such as power management, bus/client driver relationships etc, these advanced features are not required. The only thing necessary to get the device manager to load a basic driver is to properly implement and expose the required entry points. Then create the necessary registry keys that the Device manager needs to activate your device and you have a functional driver.



Drivers can run in either kernel mode or user mode. Kernel mode drivers run in the context of the kernel. User mode drivers run in one or more user mode processes. Both user mode and kernel mode drivers are managed by the Device manager (devmgr.dll).

An Overview of Device Drivers (continued)

- **Kernel Mode Drivers**
 - Default driver model
 - Run in kernel memory space
 - Link to kernel version of coredll, k.coredll.dll
 - Automatic, no need to change build rules
 - Highest performance
 - Fast access to kernel APIs
 - Direct access to user buffers
 - Must be robust
 - Driver crash could corrupt kernel

The Device manager loads all drivers into kernel space as kernel mode drivers unless the `DEVFLAGS_LOAD_AS_USERPROC` flag is set in the registry. Kernel mode drivers provide the best performance since they can call kernel APIs directly using the kernel version of coredll called `k.coredll.dll`. Kernel drivers can synchronously access user buffers very quickly because user memory is directly available.

Kernel mode drivers must be robust because they have unlimited access to memory. A fault in a kernel mode driver could corrupt kernel memory causing a system crash.

An Overview of Device Drivers (continued)

- **User Mode Drivers**
 - Also managed by Device manager
 - Hosted by udevice.exe
 - Close compatibility with kernel mode drivers
 - UM Drivers lose Kernel privileges
 - No access to kernel structures or memory
 - Cannot call certain kernel only APIs
 - Restricted access to other kernel APIs
 - Increases system stability
 - Examples
 - Expansion buses like USB and SDIO

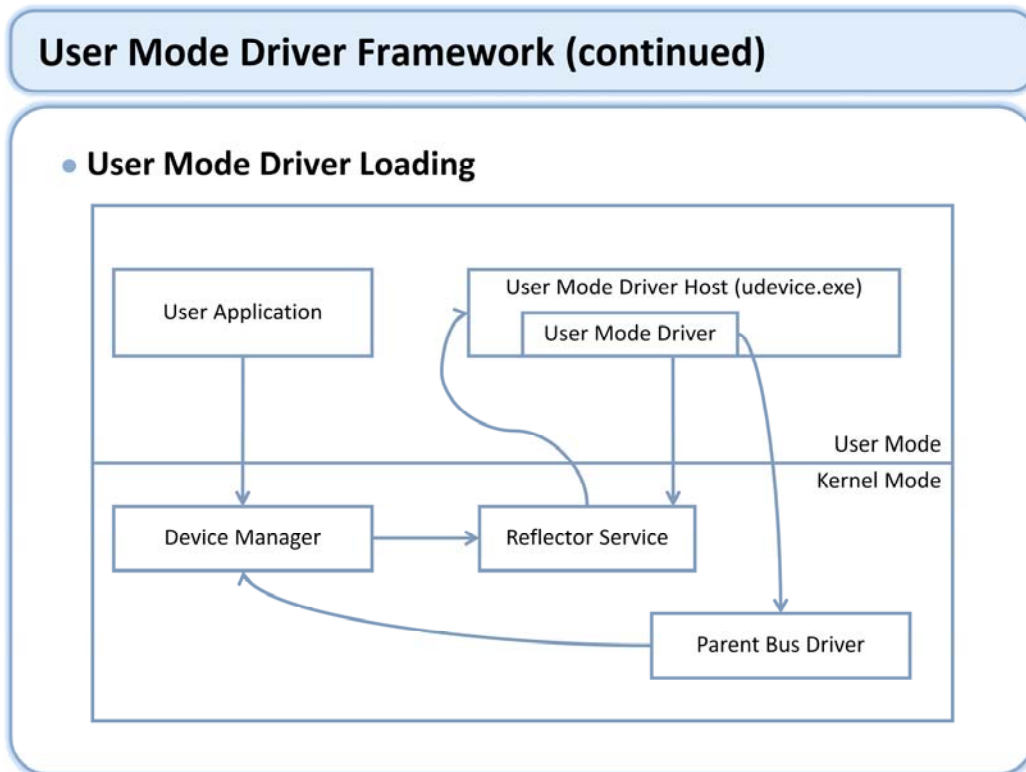
Drivers are loaded into User mode when the `DEVFLAGS_LOAD_AS_USERMODE` flag is set in the registry. This causes the driver to be loaded into a user mode driver host process called `udevice.exe`. This isolates the driver from the rest of the system increasing overall stability at the expense of performance.

User mode drivers run in the context of a user mode process and are therefore restricted from calling kernel only APIs (with some exceptions). In addition user mode drivers do not have access to kernel memory and therefore have restrictions on the kinds of pointers and asynchronous memory accesses they can perform. As a result drivers that must be able to run in either user mode or kernel mode must be written to comply with user mode driver restrictions.

Device Driver Concepts

- An Overview of Device Drivers
 - User Mode Driver Framework
 - Handling Caller Buffers
 - Loading
 - Lab 7.1
 - Debugging
 - Lab 7.2
 - Review
- Improved stability
 - User-Mode Drivers are isolated from other drivers
 - Kernel is isolated from user-mode drivers
 - Increased security
 - Kernel protected from compromised drivers
 - Lower privileges restrain a compromised driver
 - Recoverability
 - System can recover after a driver crash
 - Driver can be restarted without rebooting





The User Mode Driver Framework is divided into two physical components. The first component is the User Mode Driver Reflector, which resides inside the device manager. The second component is the User Mode Driver Host, which is a user mode application that is launched and managed by the User Mode Driver Reflector.

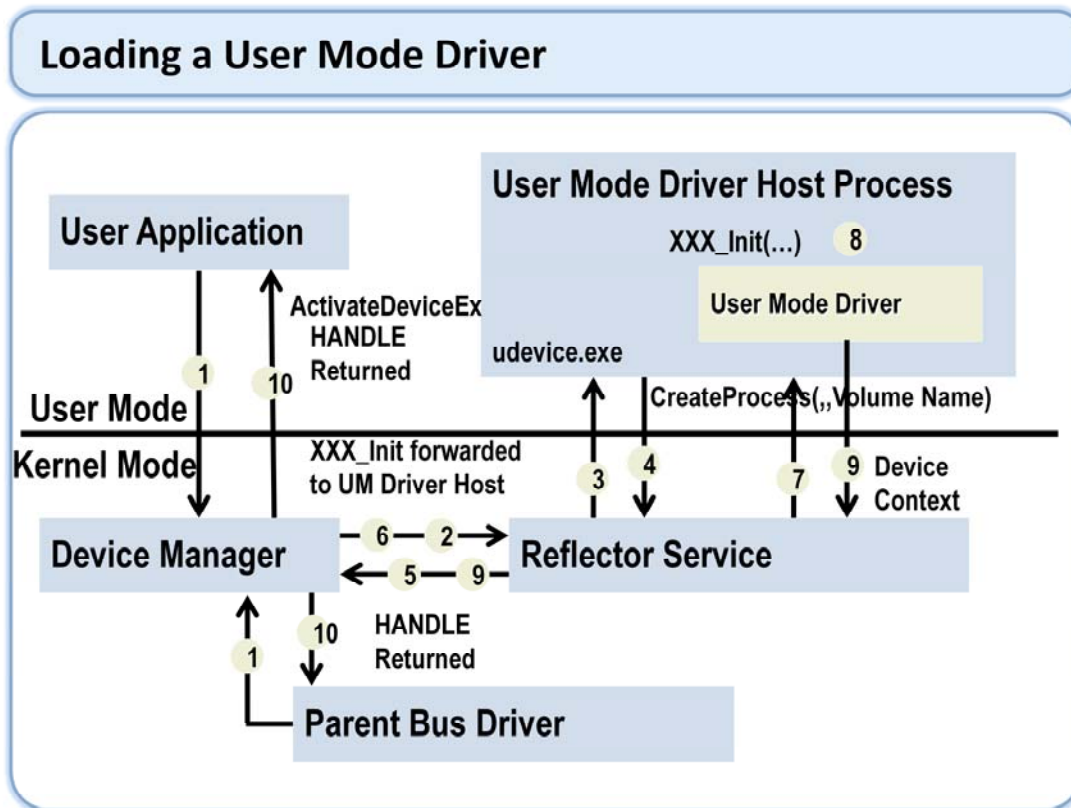
When a driver is flagged as a User Mode Driver in the registry, the device manager will access the User Mode Driver Reflector. The User Mode Driver Reflector launches the appropriate User Mode Driver Host process and forwards I/O requests to it. The User Mode Driver Host process in turn forwards I/O requests to the User Mode Driver.

The process of using a User Mode Driver begins with a user application or with a parent bus driver. User applications and parent bus drivers can load User Mode Drivers by calling `ActivateDevice` or `ActivateDeviceEx`. User applications and parent bus drivers can unload User Mode Drivers by calling `DeactivateDevice`. Once loaded, any user application or driver can access the User Mode Driver by using the device handle.

If the driver is a User Mode Driver, the call to `ActivateDevice` or to `ActivateDeviceEx` will result in the device manager calling the User Mode Driver Reflector function as well. A driver is recognized as a User Mode Driver when the `FLAGS` value is set to the setting `DEVFLAGS_LOAD_AS_USERPROC (0x10)` in the registry key of the device.

The User Mode Driver Reflector, which is aware of the original process and the destination process, uses `CeFsloControl` to forward the device manager's request to the User Mode Driver Host. The User Mode Driver Host then parses the request to either load, unload, or call the parent bus driver's entry.

The User Mode Driver has restricted access rights to the hardware, which prevents it from being able to accommodate requests such as mapping physical memory or calling interrupt functions. To accomplish these types of requests, the User Mode Driver calls the User Mode Driver Reflector, and enables the User Mode Driver Reflector to handle the request. The User Mode Driver Reflector then checks the user request against the registry settings to determine whether it should perform the requested action. Unlike with interrupts and the mapping of physical addresses, the User Mode Driver can, via the device handle and without the aid of the User Mode Driver Reflector, access the parent bus driver regardless of where the parent bus driver is located.



1. User App calls `ActivateDevice(Ex)` to Load UM Driver
(App can Call `DeactivateDevice` to unload the loaded UM Driver.)
Device Manager checks Registry Keys to find out if the driver is to be loaded in UM
2. Device Manager then creates a Reflector Service Object
Reflector Service then launches the UM Driver Host process with the Volume Name passed as an argument.
The UM Driver Host creates and mounts the specified Volume and registers a set of File System Volume APIs.
The call returns back to the reflector
The call returns back to the Device Manager
Device Manager then calls `XXX_Init`
Reflector forwards `XXX_Init` to the UM Host Process
UM Host Process parses request and loads the required driver and calls into `XXX_Init` entry of the loaded driver.
Loaded Driver returns back the Device Context to Device Manager
Device Manager creates a Handle associated with the returned Device Context and returns it to the User App.

Now the UM Driver is loaded and the User App can access it using normal Handle based APIs; `CreateFile`, `WriteFile`, `ReadFile`, `DeviceIoControl` are all supported.

User Mode Driver Framework (continued)

- **Reflector Service**
 - Nucleus of User Mode Driver Framework
 - One Reflector Object for each UM Driver
 - Launches and manages UMD Host process
 - Forwards device request to UM Driver hosted by UM Driver Host
 - Maps first level pointers from caller to UM Driver process space
 - Serves User Mode Drivers on kernel-privilege actions
 - Makes UMDs act as KMDs from the User Application's perspective

The reflector service resides in the device manager and is the bridge between a calling application and the user mode driver process. The reflector masks the difference between a user mode and kernel mode driver to other user mode processes. A calling application or another driver does not know whether a particular driver is running in kernel mode or user mode, thanks to the reflector service.

The reflector service also provides kernel support services to the user mode driver. The reflector (which runs in the kernel) performs certain kernel mode only APIs on behalf of the user mode driver. This allows a user mode driver to call certain kernel mode only APIs such as `InterruptInitialize()` and `VirtualCopy()` that would otherwise be unavailable. The reflector validates these calls before making them, limiting their use to parameters that are configured in the registry.

User Mode Driver Framework (continued)

- User Mode Driver Host process is launched and managed by registry settings

```
reginit.ini
; HIVE ROOT SECTION
; Set Device RootKey and registry enumerator
; @CESYSGEN IF CE_MODULES_DEVICE
[HKEY_LOCAL_MACHINE\Drivers]
  "RootKey"="Drivers\BuiltIn"
  "ProcName"="udevice.exe"
  "ProcVolPrefix"="$udevice"

[HKEY_LOCAL_MACHINE\Drivers\ProcGroup_0002]
  "ProcName"="servicesd.exe"
  "ProcVolPrefix"="$servicesd"
  "ProcTimeout"=dword:20000

[HKEY_LOCAL_MACHINE\Drivers\ProcGroup_0003]
  "ProcName"="udevice.exe"
  "ProcVolPrefix"="$udevice"

; @CESYSGEN IF CE_MODULES_ETHMAN
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Ethman]
  "Prefix"="ETH"
  "Dll"="ethman.dll"
  "Index"=dword:1
  ; WZCSVC must be started before ethman
  "Order"=dword:2A
  ; Flags==12 is DEVFLAGS_LOADLIBRARY and DEVFLAGS_LOAD_AS_USERPROC
  "Flags"=dword:12
  "UserProcGroup"=dword:3 ; // default to group 3
; @CESYSGEN ENDF
```

This set of registry entries configures a driver to load into a specific user mode driver host process. Multiple drivers can be loaded into the same user mode driver host by specifying the same UserProcGroup. If the UserProcGroup is not specified, the user mode driver will load into its own independent process.

User Mode Driver Framework (continued)

- **BIB File Issues**
 - Driver in MODULES section of BIB file must be fixed up properly to run in either User or Kernel mode
 - Use K flag to fix up to kernel space
 - Remove K flag to fix up to user space
 - Driver in FILES section can be loaded into user or kernel space

Drivers must be properly fixed up to run in the correct memory location just like other dlls. Drivers that are listed in the MODULES section will be fixed up in the image itself and must be properly specified in the bib file. If the K flag is present the driver will be fixed up to run in kernel space. If it is absent the driver will be fixed up to run in user space. The Q flag that allows dlls to be fixed up to both user and kernel space is irrelevant for drivers.

Drivers can be included in the image via the FILES section of the BIB file. Drivers added in this fashion will be fixed up by the loader at run time, can therefore be loaded into either kernel or user space.

User Mode Driver Framework (continued)

- **Restrictions**
 - Do not use embedded pointers
 - Reflector maps pointer parameters, but not embedded pointers
 - Access from kernel could result in embedded kernel pointer that can't be dereferenced
 - Keep all data in a single flat buffer passed with pointer parameter
 - No helper APIs yet
 - Do not support asynchronous memory access
 - Pointer parameters are not properly marshaled after call returns

User mode drivers should not use embedded pointers. The user mode driver reflector will perform the marshalling necessary for pointer parameters in function calls to be dereferenced. The reflector is not able to marshal pointers embedded in a data structure so the user mode driver must do the marshalling. If the driver is called by a kernel mode component the embedded pointer would point to a kernel address which the user mode driver can't access. The portable solution to this problem is to ensure that all data is passed into the driver in a single flat structure that does not contain pointers.

User mode drivers should not be designed to implement asynchronous memory accesses to client buffers. The reflector in the kernel will marshal the pointer parameter during the synchronous call but the buffer can't be marshaled for asynchronous access. User mode drivers can marshal embedded pointers for asynchronous use themselves, but only if they point to user address space (see above for embedded pointers to kernel space).

User Mode Driver Framework (continued)

- **Restrictions**
 - Careful use of Kernel APIs
 - Some APIs completely unavailable
 - Some APIs supported by reflector, but restricted by registry configuration
 - Some drivers must run in kernel mode
 - Display
 - Networking

User mode drivers generally can't access kernel mode only APIs. There are exceptions for APIs that are required for all device drivers, and these APIs are supported by the reflector in the User Mode driver framework. Drivers that must run in user mode need to ensure they don't call off limits APIs, and ensure the proper registry entries are in place to allow the reflector to validate other calls. There are some drivers that must run in kernel mode. These include display and networking drivers, among others. These components have frameworks that can't be used with the user mode driver restrictions. This restriction limits the number of drivers that can be run in user mode.

User Mode Driver Framework (continued)

- **Kernel Mode Driver Restrictions**
 - Can not display user interface directly
 - Requires support of UI proxy device driver
 - Use CeCallUserProc as interface
 - UI proxy device driver is loaded on first call

<http://msdn2.microsoft.com/en-us/library/aa915093.aspx>

The following list shows the process for displaying a UI from a kernel-mode device driver:

1. A user-mode application makes a call into a kernel-mode driver.
2. While the kernel-mode driver is running, the kernel-mode driver requires input from the user.
3. The kernel-mode driver calls CeCallUserProc to load the UI proxy device driver.
4. The kernel-mode function translates the call into an I/O control code call and forwards the call to the UI proxy device driver, which is hosted in udevice.exe.
5. The UI proxy device driver does the following:
 - Loads the UI proxy device driver that was passed to the CeCallUserProc function.
 - Calls the entry point for the UI proxy device driver.
 - Passes the UI proxy device driver data to the entry point.
 - Returns the UI proxy device driver data back to the caller.
6. The UI proxy device driver data is then marshaled back to the kernel, and the call returns to the kernel-mode device driver.

Device Driver Concepts

- An Overview of Device Drivers
- User Mode Driver Framework
- **Handling Caller Buffers**
- Loading a Stream Driver
- Lab 7.1
 - Pointer Parameter
 - Embedded Pointer
- Debugging
 - Access Checking
- Lab 7.2
 - Marshalling
- Review
 - Secure Copy



Handling Caller Buffers (continued)

- **Terminology**

- **Pointer Parameter**
 - Pointer passed to an API as a parameter
 - "pMyStruct" is a pointer parameter
- **Embedded Pointer**
 - Pointer passed to API within a data structure or buffer
 - "pEmbedded" is an embedded pointer
- **Access Checking**
 - Verify that caller process has privilege to access buffer
- **Marshalling**
 - Prepare pointer that a driver uses to access caller's buffer
- **Secure Copy**
 - Copy buffer to prevent asynchronous modification

```
struct MyStruct {
    BYTE *pEmbedded;
    DWORD dwSize;
};
...
DeviceIoControl (hFile,
    pMyStruct,
    sizeof(MyStruct), ...);
```

These terms are discussed in more detail on MSDN:

<http://msdn2.microsoft.com/en-us/library/aa931071.aspx>

The Windows Embedded CE driver model has changed for Windows Embedded CE 6.0. In Windows CE 5.0 and earlier, drivers ran in the Device.exe process. In Windows Embedded CE 6.0, drivers run in the NK.exe process. Due to the updated driver model, Windows CE 5.0 and earlier compatible drivers should be modified in order to work properly with Windows Embedded CE 6.0. Stability and security are also extremely important for drivers, as an instable driver in Windows Embedded CE 6.0 can potentially cause the OS to fail.

In Windows Embedded CE 6.0, the kernel process receives the top 2 GB of virtual memory space, while the remaining 2 GB is allocated for all other processes. The virtual memory for each process is not available at all times. Instead, only the kernel process along with any current processes are accessible.

Handling Caller Buffers (continued)

- **Access Checking**
 - Parameter pointers automatically access checked
 - Embedded pointers need to be access checked using `CeOpenCallerBuffer`
 - After use `CeCloseCallerBuffer` needs to be called to free resources and write back to buffer if duplication was used
 - This API also provides marshalling
 - This API can be forced to duplicate buffer using `ForceDuplicate` flag

Windows CE 5.0 and earlier, `MapCallerPtr` was used to validate a region of memory pointed to by a pointer parameter. `MapCallerPtr` was used in the I/O controls (IOCTLs) for a driver to validate a pointer parameter passed by a calling process. Device drivers in Windows CE 5.0 ran with a relatively high privilege, and had adequate access to memory. `MapCallerPtr` was used to verify both pointer parameters as well as embedded pointers.

In Windows CE 5.0 and earlier, `MapPtrToProcess` was used by a device driver to gain access to the data in the address space of an application.

In Windows Embedded CE 6.0, the kernel performs a full access check on buffer pointer parameters. This takes the responsibility for pointer parameter validation away from a device driver. However, a driver still must verify that the caller has access to memory addressed by embedded pointers.

It is possible for a malicious application to pass an embedded pointer to a kernel address space and have a driver read or write to the buffer, potentially modifying the kernel. A driver must use the `CeOpenCallerBuffer` and `CeCloseCallerBuffer` functions to verify that the caller has access to the memory that is pointed to by embedded pointers.

`CeOpenCallerBuffer` can be called with the `ForceDuplicate` parameter set to `TRUE`. This allocates a temporary heap buffer in the current process. If you choose to copy an input buffer for security purposes, and use `CeOpenCallerBuffer` for access checking, you can set `ForceDuplicate` to `TRUE`. This allows you to perform both the input buffer copy and the access check with one function call.

*/

Handling Caller Buffers (continued)

- **Marshalling Methods**
 - Direct
 - Caller buffer directly accessible, no action necessary
 - Applies only to kernel mode driver called synchronously
 - Copy
 - Copy caller input buffer to driver working buffer
 - Operate on driver working buffer
 - Copy back to caller
 - Alias
 - Create new buffer in driver mapped directly to caller buffer
 - Accesses to driver buffer are reflected in caller buffer
- **Kernel can determine most efficient marshalling method**

In Windows CE 5.0 and earlier, the `MapCallerPtr` function also performed marshalling for pointers. A driver called `MapCallerPtr` on parameters as well as embedded pointers both for validation and marshalling.

In Windows Embedded CE 6.0, marshalling is dependent on if a pointer is used synchronously or asynchronously. If a pointer parameter or embedded pointer is used synchronously, the address space of the calling process is accessible for the duration of a call into the driver. This eliminates any requirements for marshalling. The pointer of the calling process can be used unchanged by the driver, which can then access the memory of the caller directly. This method of marshalling is referred to as direct access.

However, if a pointer is used asynchronously, it is critical that the caller buffer is accessible when the caller's address space is unavailable. This means that direct access is not possible for any kind of asynchronous work after the call has returned. Windows Embedded CE 6.0 includes the `CeAllocAsynchronousBuffer` and `CeFreeAsynchronousBuffer` functions for drivers to marshal pointer parameters and embedded pointers when asynchronous access is required. For example, when a thread such as the IST requires access to the buffer of the caller, the marshalling helper functions can choose between the duplication and aliasing marshalling mechanisms. This is dependant on the size of the buffer involved. If the buffer is small enough, the kernel duplicates the buffer. However, this can affect performance, and if the buffer is too large to duplicate, the kernel will alias the buffer instead.

Handling Caller Buffers (continued)

- Synchronous access
 - Defined as access on SAME THREAD synchronously
 - Kernel automatically handles pointer parameters
 - Use CeOpenCallerBuffer to access check and marshal embedded pointers
- Asynchronous access
 - Defined as any asynchronous access or any access on another thread
 - Initial handling same as synchronous access
 - Use CeAllocAsynchronousBuffer to marshal pointer for asynchronous use

Handling Caller Buffers (continued)

- **Handling Caller Buffers**
 - **Secure Copy**
 - Copy input parameter buffer to driver local buffer
 - Validate and use local input parameters
 - Prevents caller from maliciously or inadvertently modifying input parameters
 - Performance impact
 - **Secure Copy Methods**
 - Manual copy
 - `CeOpenCallerBuffer`
 - Use with embedded pointers
 - `ForceDuplicate` flag
 - `CeAllocDuplicateBuffer`
 - Use with pointers

Performing a secure copy of input parameters is one of the best practices for developing a device driver for Windows Embedded CE. It is not necessarily safe for a driver to access the buffer of a caller. It is possible that the caller may be malicious, or even written poorly. The solution to these data integrity problems is to perform a secure copy. If a driver must access the buffer of a caller asynchronously, it must call the `CeAllocAsynchronousBuffer` and `CeFreeAsynchronousBuffer` functions. This eliminates the need to perform an additional parameter copy. If a driver is accessing parameters synchronously, you should use the `CeAllocDuplicateBuffer` and `CeFreeDuplicateBuffer` secure copy helper functions to copy the buffer of the caller.

If you are handling embedded pointers and calling the `CeOpenCallerBuffer` function for access checking, set the `ForceDuplicate` parameter to `TRUE` to obtain a local copy of the buffer of the caller. This allows you to avoid an additional function call to `CeAllocDuplicateBuffer`. The local buffer is then freed upon calling `CeCloseCallerBuffer`.

Handling Caller Buffers (continued)

- **APIs**
 - **CeOpenCallerBuffer**
 - Use on embedded pointers
 - Returns checked, marshaled pointer
 - Can force buffer duplication (Secure Copy)
 - **CeCloseCallerBuffer**
 - Frees resources allocated by CeOpenCallerBuffer
 - Write back to caller buffer if necessary

Handling Caller Buffers (continued)

- **APIs**
 - **CeAllocAsynchronousBuffer**
 - Allocate driver buffer for asynchronous use
 - Source buffer must be marshaled
 - Must be called synchronously
 - **CeFreeAsynchronousBuffer**
 - Frees resources allocated by CeAllocAsynchronousBuffer
 - Write back to caller buffer if necessary

Handling Caller Buffers (continued)

- **APIs**
 - **CeAllocDuplicateBuffer**
 - Secure copy input buffer
 - Use with pointers only
 - Use with synchronous access only
 - **CeFreeDuplicateBuffer**
 - Frees resources allocated by CeAllocDuplicateBuffer
 - Write back to caller buffer if necessary

Handling Caller Buffers (continued)

- **APIs Review**

Function	Description
CeOpenCallerBuffer	Checks access and marshals a buffer pointer from the source process so that it can be accessed by the current process.
CeCloseCallerBuffer	Frees any resources that were allocated by CeOpenCallerBuffer.
CeAllocAsynchronousBuffer	Re-marshals a buffer that was already marshaled by CeOpenCallerBuffer so that the server can use it asynchronously after the API call has returned.
CeFlushAsynchronousBuffer	Flushes any changed data between the source and the destination buffer allocated by CeAllocAsynchronousBuffer.
CeFreeAsynchronousBuffer	Frees any resources that were allocated by CeAllocAsynchronousBuffer.
CeAllocDuplicateBuffer	Abstracts the work required to make secure copies of API parameters.
CeFreeDuplicateBuffer	Frees a duplicate buffer that was allocated by CeAllocDuplicateBuffer.

Device Driver Concepts

- An Overview of Device Drivers
 - User Mode Driver Framework
 - Handling Caller Buffers
 - **Loading a Stream Driver**
 - Lab 7.1 – Integrating a Device Driver
 - Debugging Device Drivers
 - Lab 7.2 – Driver Loading
 - Review
- Driver loading controlled by registry keys
 - When the driver loads
 - How the driver loads
 - Parameters passed to the driver
 - Drivers loaded with call to `ActivateDeviceEx`
 - Takes handle to registry key as parameter
 - Different loading requirements supported
 - Can be loaded automatically at boot (Built In)
 - Can be loaded dynamically (Plug and Play)



Stream drivers are loaded via a call to `ActivateDeviceEx()`. This function takes a handle to a registry key as a parameter. The registry key contains all the information necessary to configure and load the driver. CE 6.0 supports both automatic driver loading at boot and dynamic driver loading on demand. This provides support for Plug and Play buses, and allows drivers to be unloaded as needed.

Loading a Stream Driver (continued)

- **Driver Registry Settings**
 - **Dll [Required]**
 - Specifies the name of the driver DLL
 - **Prefix [Optional]**
 - Specifies the file name for the device driver
 - **Order [Optional]**
 - Specifies order to load driver
 - **Index [Optional]**
 - Specifies the device index (x in COMx:)
 - **IClass [Optional]**
 - Specifies GUID(s) for device class(es) for use by PnP notification system
 - **Flags [Optional]**
 - Specifies load flags for the driver

There are many different registry settings that control how a driver loads. Most are optional. Some registry settings are used by the Device manager and can be optionally used by all drivers. Custom registry entries can also be added that are referenced directly by the driver itself after it loads.

DLL is a string value that specifies the dll name to load. This value is required.

Prefix is a three character string that makes up the name of the driver. This value must be present in order to have a file handle based interface to the driver. The Prefix value is the same value that must be used in the stream driver entry points unless the DEVFLAGS_NAKEDENTRIES flag is set.

Order is a dword value that provides a mechanism supporting load order dependencies. Drivers will be loaded in the order specified by the Order key. If this key does not exist the driver will be loaded at the end. Order should not be used unless there is a load order dependency to resolve.

Index is a dword value that makes up the numeric portion of the device name. This value is optional; the Device manager will pick the next sequential value if it does not exist.

IClass provides a mechanism to advertise capabilities to various system components.

Flags provides a mechanism to control the way the driver is loaded.

Loading a Stream Driver (continued)

• Driver Registry Settings - Flag Parameter

Flag	Value	Description
DEVFLAGS_UNLOAD	0x00000001	Driver unloads after a call to the XXX_Init entry point
DEVFLAGS_NOLOAD	0x00000004	Driver is not loaded
DEVFLAGS_NAKEDENTRIES	0x00000008	Driver entry points do not have an XXX Prefix
DEVFLAGS_BOOTPHASE_1	0x00001000	Driver is loaded during system phase one.
DEVFLAGS_IRQ_EXCLUSIVE	0x00000100	Bus Driver loads the driver only when it has exclusive access to the IRQ
DEVFLAGS_LOAD_AS_USERPROC	0x00000010	Loads a driver into user mode

There are a number of loading options supported by the Device manager in the ActivateDeviceEx() call. These options are governed by the Flags value in the driver loading key. The default option if no Flag parameter exists is none. Some of the more common flags include:

DEVFLAGS_BOOTPHASE_1

This flag indicates that the driver should be loaded during the first boot phase only in hive registry systems that have multiple boot phases. This flag is used in drivers that exist in the both the boot hive and the system hive, and prevent the driver from being loaded twice.

DEVFLAGS_NAKEDENTRIES

This flag indicates that the driver does not implement the three letter prefix in its stream driver exports. This allows the same driver to be used with different named prefixes, no need to recompile the driver.

DEVFLAGS_LOAD_AS_USERPROC

This flag causes Device manager to load the driver into user space. The default is to load all drivers as kernel mode dlls unless this flag is set. This is new functionality in CE6.

Loading a Stream Driver (continued)

- **Driver Registry Settings – Bus Information**
 - **InterfaceType**
 - Specifies the Interface (Bus) type used for a device
 - See `INTERFACE_TYPE` enumeration in `CEDDK.H` for a complete list of interface types defined by Microsoft. (You can define new ones if you need to)
 - **BusNumber**
 - Bus Instance number
 - Uniquely identifies the specific bus in case there is more than one of the specified type in the system

InterfaceType and BusNumber are not used by Device manager in the `ActivateDeviceEx` call. Rather they are used by the driver itself to determine how the corresponding device interfaces in the system.

Loading a Stream Driver (continued)

- **Driver Registry Settings – Memory and I/O Windows**
 - Memory and I/O Window information may be populated by a Plug and Play Bus Driver or configured by the system OEM
 - **IoBase**
 - Specifies the bus relative base of an I/O mapped window used by the device.
 - Multiple windows specified using a binary array of 32bit values, one for each window needed
 - **IoLen**
 - Specifies the length of each I/O window needed by the device
 - **MemBase**
 - Specifies the bus relative base of a memory mapped window used by the device.
 - Multiple windows specified using a binary array of 32bit values, one for each window needed
 - **MemLen**
 - Specifies the length of each memory mapped window needed by the device

These parameters indicate the memory and IO requirements of a particular device. The values are relative to the bus where the device resides. These values may be configured by the OEM for built in devices, or they might be populated by a bus driver in a Plug and Play bus such as PCI.

The driver can easily retrieve this information along with bus information from the registry using the `DDKReg_GetWindowInfo()` helper function. These bus relative addresses can then be mapped into virtual addresses usable by the driver with the `BusTransBusAddrToVirtual()` helper function.

Loading a Stream Driver (continued)

- **Driver Registry Settings – Interrupts**

- **IRQ**
 - Specifies the physical IRQ used by the device
- **SYSINTR**
 - Specifies a Logical interrupt Identifier
 - Typically allocated by a bus driver for shared IRQs, the Device driver reads this value and if present uses the one provided by the bus driver. Otherwise the driver should use the IRQ to request a new logical ID.
- **IsrDll**
 - Specifies a DLL to containing an installable interrupt handler
- **IsrHandler**
 - Specifies the ISR handler in the ISR DLL

These optional registry entries provide interrupt related configuration data for a particular driver. Typically a driver would use the Sysintr value directly if it exists in the registry. Otherwise it will request a new sysintr using the Irq value, if the Irq value exists. Drivers that use installable interrupt handlers use the IsrDll and IsrHandler entries to specify the installable interrupt handler.

These values can easily be retrieved by the driver using the `DDKReg_GetIsrInfo()` helper function.

These registry entries are not required, but they are a good practice that allows the driver to be more portable.

Loading a Stream Driver (continued)

- **Bus Drivers**
 - What is a Bus Driver?
 - Load drivers for the devices on their respective buses
 - Provides Device Manager with enough information to create bus-relative names and enable device handles
 - Provides bus level services for the bus
 - Examples are:
 - USB
 - PCI (PCIBus.dll)
 - PCMCIA (Pcmcia.dll)
 - Bus Enumerator (BusEnum.dll)

A bus driver is any software that loads drivers. Bus drivers have one or more of these responsibilities:
Managing physical busses, such as PC Card, USB, or PCI.

Loading drivers on a physical bus that the bus driver does not directly manage. An example is the Bus Enumerator, which is a bus driver that loads built-in drivers and PCI bus drivers.

Calling `ActivateDeviceEx` directly to load a device driver. The loaded device driver might manage hardware indirectly through another device driver.

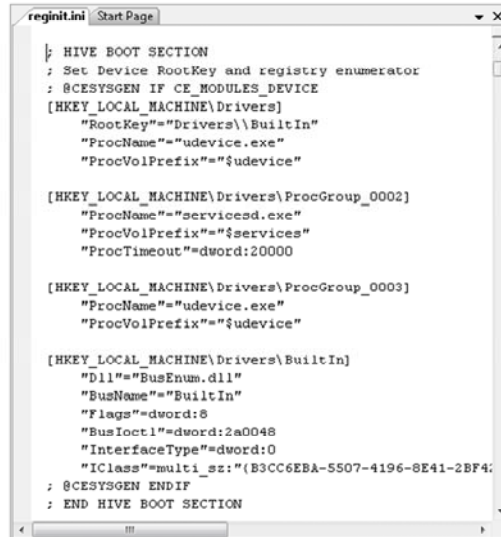
Loading a Stream Driver (continued)

- **BusEnum.dll**
 - Used by Device Manager to load drivers
 - Starts top level driver loading process
 - “Bus” consists of registry subkeys
 - Loads drivers listed in subkeys
 - Loads other bus drivers and unmanaged drivers

The bus enumerator is a bus driver loaded with a call to `ActivateDeviceEx()`. The bus enumerator calls `ActivateDeviceEx()` for each registry subkey directly below the key that activated the bus driver, and does not traverse deeper into the registry. The bus enumerator is the initial driver loaded directly by Device manager, and eventually causes all other drivers to be loaded.

Loading a Stream Driver (continued)

- Device Manager Process for Loading Drivers at Boot

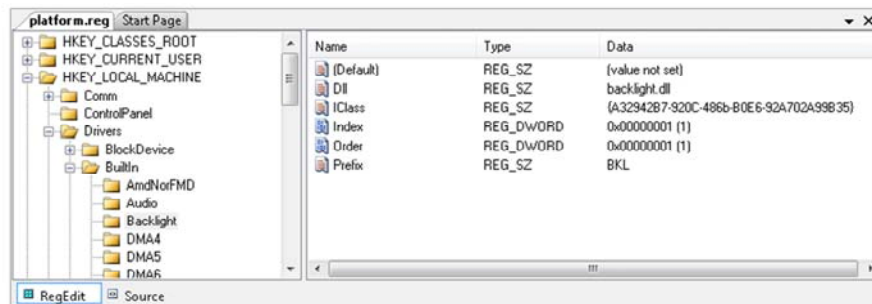


```
reginit.ini Start Page
|
|; HIVE BOOT SECTION
|; Set Device RootKey and registry enumerator
|; @CESYSGEN IF CE_MODULES_DEVICE
|[HKEY_LOCAL_MACHINE\Drivers]
|  "RootKey"="Drivers\BuiltIn"
|  "ProcName"="udevice.exe"
|  "ProcVolPrefix"="$device"
|
| [HKEY_LOCAL_MACHINE\Drivers\ProcGroup_0002]
|  "ProcName"="serviced.exe"
|  "ProcVolPrefix"="$services"
|  "ProcTimeout"=dword:20000
|
| [HKEY_LOCAL_MACHINE\Drivers\ProcGroup_0003]
|  "ProcName"="udevice.exe"
|  "ProcVolPrefix"="$device"
|
| [HKEY_LOCAL_MACHINE\Drivers\BuiltIn]
|  "Dll"="BusEnum.dll"
|  "BusName"="BuiltIn"
|  "Flags"=dword:8
|  "BusIoctl"=dword:2a0048
|  "InterfaceType"=dword:0
|  "IClass"=multi_sz:"{B3CC6EBA-5507-4196-8E41-2BF4:
|; @CESYSGEN ENDIF
|; END HIVE BOOT SECTION
```

The driver loading process starts with the Device manager reading the registry key at HKLM\Drivers to obtain the root path for driver loading. It then calls ActivateDeviceEx() on that key. The driver located at that key is typically the Bus Enumerator driver. Its responsibility is to serve as a bus driver for subkeys directly below it. In the typical example shown above this means that everything directly under the HKLM\Drivers\BuiltIn key will be automatically loaded at boot by BusEnum.dll. These drivers are typically referred to as "Built in" drivers. Built in drivers often include other bus drivers such as PCI, USB etc that each manage their own buses

Loading a Stream Driver (continued)

- Sample Registry Entry



Here is an example of a “built in” driver. This driver will be automatically loaded at boot by the Bus Enumerator driver.

Note that the registry configuration for many drivers will not be as simple as this example. Drivers will often have other required configuration parameters especially if they reside on a bus. Therefore it is important to check the documentation for the driver to determine the required configuration parameters.

Device Driver Concepts

- An Overview of Device Drivers
- User Mode Driver Framework
- Handling Caller Buffers
- Loading a Stream Driver
- **Lab 7.1 – Integrating a Device Driver**

- De
- Lab
- Rev

- **Lab Goals**

1. Be able to integrate new drivers into an existing BSP
2. Be able to use Run-Time Image Viewer to verify the contents of an OS run-time image


- [Video](#)



Device Driver Concepts

- An Overview
- User Mode D
- Handling Call
- Loading a Str
- Lab 7.1 – Inte
- Debugging De
- Lab 7.2 – Deb
- Review

- Implement KITL
 - Essential for driver development
- Do a Debug Build
 - Disables optimizations
 - Enables DEBUGMSG macro
- Use the Kernel Debugger
 - Can be included in Debug or Release build
 - Kernel debugger is a component
- Use Capabilities
 - Breakpoints, call stack, watch variables



There are several basic debugging techniques that can assist in debugging device drivers. The most important step is to ensure that KITL is available on your device. The KITL transport is required in order to use the kernel debugger and target control utility, and it can be used as the communications transport for all the other remote tools.

Do a debug build of the operating system. A debug build does two things: disables compiler optimizations, and sets the DEBUG define. Compiler optimizations make the kernel debugger less useful because the source code no longer matches the compiled output. This makes breakpoints behave erratically, many variables can't be resolved etc. Nothing wrong with the debugger, the compiler has just optimized the code into a form that is different from what appears in source. The DEBUG define is used by a number of debugging macros, namely DEBUGMSG. A debug build will produce many more messages because the DEBUGMSG macro is utilized.

Use the kernel debugger to set breakpoints, view the call stack, watch local variables etc. Note that the kernel debugger is a component that can be included in the image (IMGNODEBUGGER not set). A debug build does not necessarily include the kernel debugger (although it does by default).

Use the DEBUGMSG macro in your driver to output appropriate diagnostic messages. Use the debug zones capability to filter the output. Understand that debug messages take time, and can impact the functionality of a bandwidth sensitive driver (USB, audio).

Debugging Device Drivers (continued)

- **Use DEBUGMSG**
 - Implement appropriate debug zones
- **Understand build requirements**
 - Use targeted builds
 - Driver change only requires driver to be compiled
 - Make Runtime Image if driver in image (makeimg)
 - No need to Sysgen after each change
- **Load driver from Release Directory**
 - Improves turn around time
 - Target->Release Directory Modules
 - Skip lengthy Makeimg step

There are a number of techniques you can use to reduce the turn around time in the debug cycle. Use targeted builds effectively; there is no need to rebuild the entire BSP because you made a change to driver source code in the BSP. Even worse, there is no need to rerun the sysgen cycle on the OS Design. If you are modifying code in the Public tree, you would need to build and sysgen the Public tree – but you should never modify code in the Public tree. A new operating system image can be made with the driver change by rebuilding the driver, ensuring it is in the flat release directory, and making the run time image again.

A further improvement can be made by loading the driver directly from the flat release directory instead of the operating system image. This eliminates the need to rebuild the operating system image (make run time image) after a driver change. This can be accomplished by adding the driver to the list of modules in the Target->Release Directory Modules dialog box.

Debugging Device Drivers (continued)

- **Combine Release builds with kernel debugger**
 - Use debug version of driver
 - Build in debug window
 - Copy to Release FRD
 - Use Release build of operating system
 - Include KITL, kernel debugger components
 - More realistic performance
 - Much smaller image

Another technique that can be used when debugging drivers is to use a debug version of the driver with a Release build of the operating system. This results in a much smaller operating system image with performance characteristics that are more representative of the real world. The operating system boots much faster, resulting in better turn around times. The driver has been built with the debug settings, meaning optimizations are turned off for better a better debugger experience and debug messages are enabled.

This method requires both debug and Release builds to be performed. Copy the driver binary and supporting files (.pdb, .map. etc) to the flat release directory replacing the Release version. The driver can be built directly into the image, or loaded from the flat release directory.

Debugging Device Drivers (continued)

- **Debugger Issues**
 - If you cannot set breakpoints
 - Ensure that .dll, .pdb, and .map files are in Flat Release Directory
 - DLL name in image must match name in Flat Release Directory
 - Module sometimes renamed in bib file (rare)
 - Run-time image must be in “break” state to instantiate breakpoint
 - Underlying memory is ROM (XIP from ROM)
 - Can’t set breakpoints in ROM unless OEMIsRom implemented
 - Load module from flat release directory instead

Debugging Device Drivers (continued)

- **Debugger Issues**
 - Compiler optimizations in Release build affect debugger
 - Changes underlying code sequence
 - Single stepping does not follow source code as expected
 - Eliminates many local variables
 - Variables are not resolved because they do not exist
 - Use debug version of module with kernel debugger
 - Switch to disassembly view if necessary

Release builds are difficult to debug effectively with the kernel debugger. The debugger operates on the actual code output from the compiler, not the original source code. The compiler optimizations that are present in a release build cause the compiled output to differ significantly from the original source code. For example, the underlying code sequence is rearranged for better performance, and local variables are often optimized away. This is not apparent when viewing the source code in the IDE and causes confusion when using the debugger. The debugger is unable to resolve many local variables (because they don't actually exist), appears to step through source code erratically and breaks at unexpected times (because the compiler output doesn't match what is visible in the source code window).

This can be resolved by using a debug build for the module being debugged. Compiler optimizations are disabled in a debug build, so the assembled output matches the high level language source code. Another technique is to switch to the disassembly view, which is what the debugger is actually using. This is much more difficult to use, but is sometimes necessary to see what is really going on.

Debugging Device Drivers (continued)

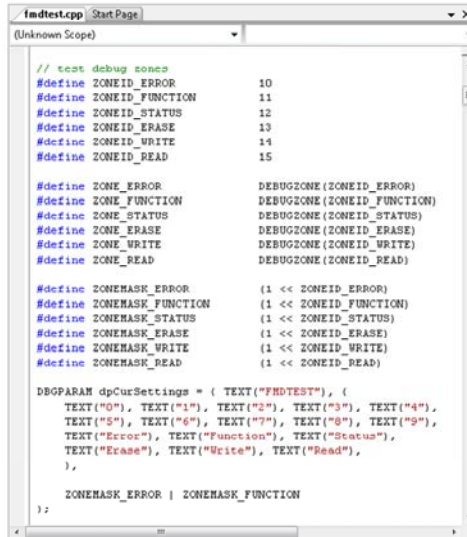
- **Debug Message Service**
 - Provide debug information without halting the system
 - Debug message output controlled by zones
 - Debug zones
 - 16 debug zones per module
 - Debug zones can be controlled at run time
 - Uses KITL transport
 - Messages appear in Platform Builder debug window
 - Messages will appear on debug serial port if KITL not available

Debugging Device Drivers (continued)

- **Debug Message Service**
 - Use predefined macros
 - DEBUGMSG
 - Message exists in debug builds only
 - RETAILMSG
 - Message exists in debug and release builds
 - Message does not exist in ship builds

Debugging Device Drivers (continued)

- Debug Message Service - Defining Debug Zones



```
fmctest.cpp Start Page
(Unknown Scope)

// test debug zones
#define ZONEID_ERROR          10
#define ZONEID_FUNCTION      11
#define ZONEID_STATUS        12
#define ZONEID_ERASE         13
#define ZONEID_WRITE         14
#define ZONEID_READ          15

#define ZONE_ERROR           DEBUGZONE(ZONEID_ERROR)
#define ZONE_FUNCTION       DEBUGZONE(ZONEID_FUNCTION)
#define ZONE_STATUS         DEBUGZONE(ZONEID_STATUS)
#define ZONE_ERASE          DEBUGZONE(ZONEID_ERASE)
#define ZONE_WRITE          DEBUGZONE(ZONEID_WRITE)
#define ZONE_READ           DEBUGZONE(ZONEID_READ)

#define ZONEMASK_ERROR      (1 << ZONEID_ERROR)
#define ZONEMASK_FUNCTION  (1 << ZONEID_FUNCTION)
#define ZONEMASK_STATUS    (1 << ZONEID_STATUS)
#define ZONEMASK_ERASE     (1 << ZONEID_ERASE)
#define ZONEMASK_WRITE     (1 << ZONEID_WRITE)
#define ZONEMASK_READ      (1 << ZONEID_READ)

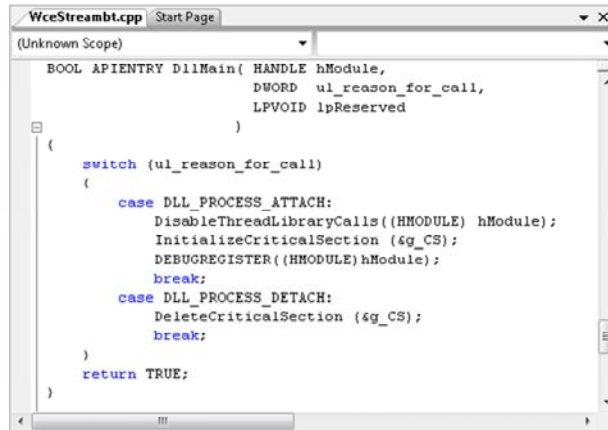
DBGPARAM dpCurSettings = { TEXT("FMCTEST"), {
    TEXT("0"), TEXT("1"), TEXT("2"), TEXT("3"), TEXT("4"),
    TEXT("5"), TEXT("6"), TEXT("7"), TEXT("8"), TEXT("9"),
    TEXT("Error"), TEXT("Function"), TEXT("Status"),
    TEXT("Erase"), TEXT("Write"), TEXT("Read"),
},
    ZONEMASK_ERROR | ZONEMASK_FUNCTION
};
```

Debugging Device Drivers (continued)

- **Debug Message Service**
 - Controlling Debug Zones
 - Default set in DBGPARAM structure
 - Override default in image
 - [HKLM\DebugZones]
 - Override default on development PC
 - [HKCU\Pegasus\Zones]
 - Control from IDE at run time
 - Target | CE Debug Zones ...
 - Target Control Utility

Debugging Device Drivers (continued)

- Debug Message Service - Register with debug subsystem



```
WccStreambt.cpp Start Page X
(Unknown Scope)
BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
            DisableThreadLibraryCalls((HMODULE) hModule);
            InitializeCriticalSection (&g_CS);
            DEBUGREGISTER((HMODULE) hModule);
            break;
        case DLL_PROCESS_DETACH:
            DeleteCriticalSection (&g_CS);
            break;
    }
    return TRUE;
}
```

DEBUGREGISTER allows debug zones to be controlled dynamically.

Debugging Device Drivers (continued)

Macro	Description
DEBUGREGISTER , RETAILREGISTERZONES (hMod)	Calls RegisterDbgZones to register zones for the current module (a DLL or process). This macro only registers zones on Debug builds; it does nothing on Retail and ship builds. This macro assumes that a global variable <code>dpCurSettings</code> has already been defined, where <code>dpCurSettings</code> must be a DBGPARAM structure.
DEBUGZONE (n)	Associates a mask bit with a zone.
DEBUGMSG (cond, printf_exp)	Conditionally outputs a formatted debugging message to NKDbgPrintFW . This macro is only present in Debug builds; it does nothing on Retail and ship builds.
RETAILMSG (cond,printf_exp)	Conditionally outputs a formatted debugging message to NKDbgPrintFW . This macro is only present in Debug and Retail builds; it does nothing on ship builds.
ERRORMSG (cond,printf_exp)	Conditionally outputs a formatted error message to NKDbgPrintFW , adding the file name and line number where the error occurred. This macro is only present in Debug and Retail builds; it does nothing on ship builds.
DEBUGCHK (exp)	Asserts an expression and produces a DebugBreak if the expression is FALSE. This assertion is only present in Debug builds; it does nothing on Retail and Ship builds. This macro assumes that a global variable <code>dpCurSettings</code> has already been defined, where <code>dpCurSettings</code> must be a DBGPARAM structure.
ASSERT (exp)	Asserts an expression and produces a DebugBreak if the expression is FALSE. This assertion does not assume <code>dpCurSettings</code> is present. The assertion is only present in Debug builds; it does nothing on Retail and Ship builds.
ASSERTMSG (msg, cond)	Asserts an expression, and if the expression is FALSE, produces an error message to NKDbgPrintFW and a DebugBreak . This macro is only present in Debug builds; it does nothing on Retail and Ship builds.
DEBUGLED (cond, parms)	Conditionally outputs a pattern to WriteDebugLED . This macro is only present in Debug builds; it does nothing on Retail and Ship builds.
RETAILED (cond, parms)	Conditionally outputs a pattern to WriteDebugLED . This macro is only present in Debug and Retail builds; it does nothing on Ship builds.

Device Driver Concepts

- An Overview of Device Drivers

- User

- Hand

- Load

- Lab 7

- Debugging Device Drivers

- **Lab 7.2 – Debugging a Device Driver**

- Review

- Lab Goals

1. Understand driver interaction with application
2. Use kernel debugger to investigate call stack
3. Understand the integration and use of Debug Zones

- [Video](#)



Device Driver Concepts

- What is a Device Driver?
- Driver Types
- An Overview of Device Drivers
- Stream Driver Architecture
- User Mode Driver Framework
- Loading a Stream Driver
- Lab 7.1 – Integrating a Device Driver
- Debugging Device Drivers
- Lab 7.2 – Debugging a Device Driver
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Customizing the OS Design

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- **Module 8: Customizing the OS Design**
- Module 9: Application Development
- Module 10: Testing & Verification
- Course Review



Customizing the OS Design

- **The Catalog**
- **Lab 8.1 – Adding an Item to the Catalog**
- **The Shell Options**
- **Lab 8.2 – Replacing the Standard Shell with IESHELL**
- **SDKs**
- **Lab 8.3 – Exporting an SDK**
- **Review**



Customizing the OS Design

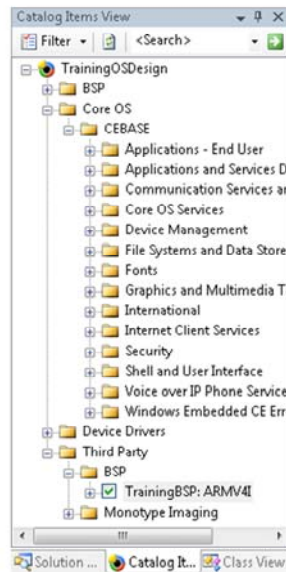
- **The Catalog**
- Lab 8.1 – Adding an Item to the Catalog
- The S
- Lab 8
- SDKs
 - Extensible database of components used to create as OS Design
 - Database is comprised of “pbxml” files located in catalog branches of the build tree
- Lab 8.3 – Exporting an SDK
- Review



The Catalog (continued)

• The Catalog Items View

- Provides
 - Hierarchical view
 - Filtering
 - Searching based on name or sysgen name
 - Viewing dependencies
 - Adding to design
 - Removing from design
 - Viewing properties
 - Viewing reasons for inclusion of system added items
- Browsing 😊



The items that the Catalog contains range from BSPs, core operating system (OS) functionality, transport layers, to device drivers. You can also display additional items, which you or a third party create, in the Catalog by importing information about the items in Catalog item (.pbcxml) files.

The catalog contains sub folders that contain items. Each item is either a module or component of a module that you can select to include in the run-time image.

You can import and export XML Catalog files to add third party items.

The Catalog Items View shows all the Catalog items that you can add to the OS Design, including BSPs, core operating system (OS) functionality, transport layers, and device drivers. You can also use filters to display only the Catalog items that are included in your OS Design.

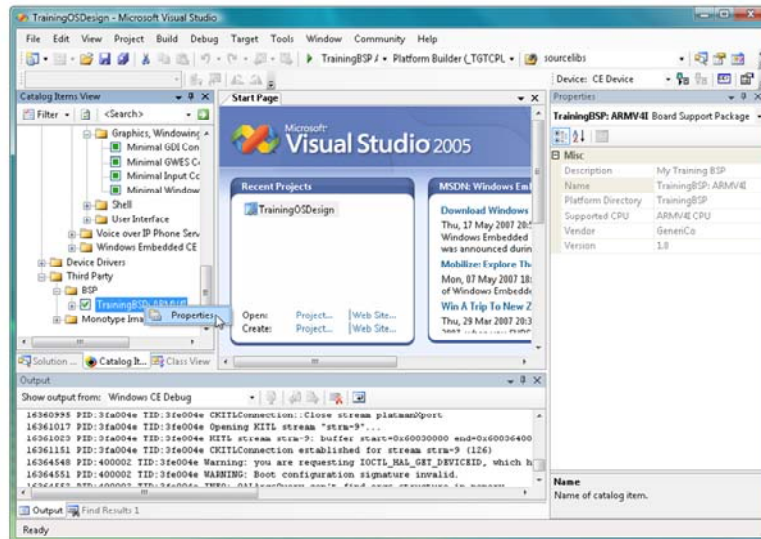
To determine which Catalog items are included in an OS Design, choose Filter, and then User-selected Catalog Items and Dependencies. View the Catalog items in your OS Design by expanding the nodes in the Catalog item tree.

Check Boxes are user added items, in this case the CE test Kit - CETK
 Green squares show system that the items was added to support an other item.
 We will see how you can view these dependencies.

If the Catalog Items View is not visible select Catalog Items View from the Other Windows Menu Item under the View Menu.

The Catalog (continued)

- Displaying the properties of a catalog items

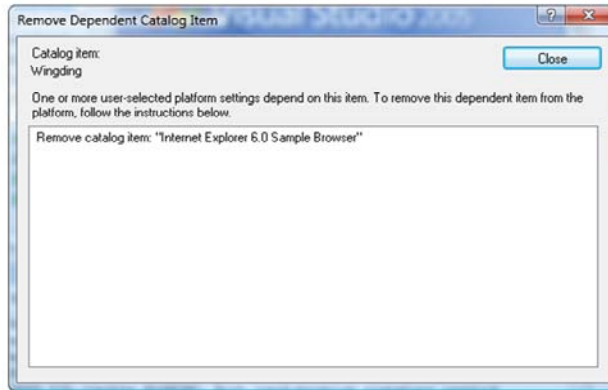


A Catalog item is specifically included in the OS Design by a design template or by the OS developer. A Catalog item might also be added during the build cycle if it is a dependency of another item. The Cesium.bat file controls additional batch files, which contain dependency rules that the build system compares against the items in your OS Design. The IDE tools automatically include additional Catalog items that are required to support the Catalog items initially included in the OS Design. Each time you add or remove a Catalog item from your OS Design, or perform any other action that requires a Sysgen of the OS Design, the display of Catalog items is refreshed. The Build tab in the Output window also displays a list of Catalog items added due to dependencies. Each time this process is run, the system starts with only required Catalog item functionality and no dependent items.

The properties of catalog items will vary based on the context of the item selected.

The Catalog (continued)

- Viewing Reasons for Inclusion



Catalog View - Catalog Dependencies

You can visually determine dependencies, by right clicking on a item in the catalog view and selecting Reasons for Inclusion of Item.

Note: This functionality is for system included items only.

Build Process - Catalog Dependencies

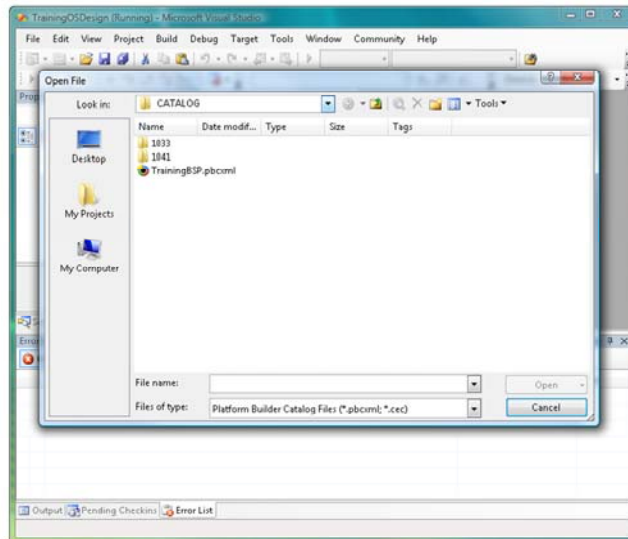
During the build, Platform Builder runs Cesiumgen.bat to discover which Catalog items to bring in by dependency. Through examining Cesiumgen.bat, which specifies Catalog item dependencies, you can discover which Catalog items depend on a selected Catalog item to function properly. For more information, examine the Cesiumgen Batch File.

In the following example of a Catalog item dependency, if your OS Design has SYSGEN_MODEM set, it brings in SYSGEN_PPP.

```
if "%SYSGEN_MODEM%"=="1" set SYSGEN_PPP=1
```

The Catalog (continued)

- **Opening a Catalog Item File for Viewing /Editing**

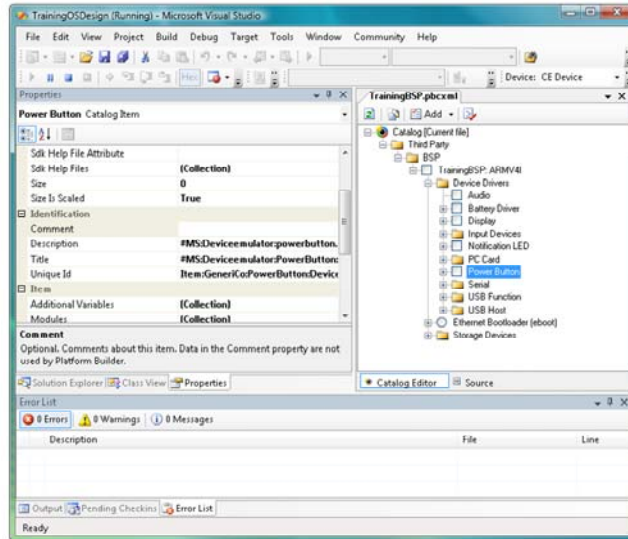


You can use the Catalog Editor to view and edit Windows Embedded CE 6.0 Catalog item (.pbcxml) files, which are XML-based files that contain metadata about the associated Catalog item.

In Windows Explorer, when you choose to open a .pbcxml file, Windows Embedded CE 6.0 opens the Catalog Editor. The visual interface is much like that of other Visual Studio editors and components.

The Catalog (continued)

• Viewing/Editing through Graphical Interface



You can use the Catalog Editor to view and edit Windows Embedded CE 6.0 Catalog item (.pbxml) files, which are XML-based files that contain metadata about the associated Catalog item.

In Windows Explorer, when you choose to open a .pbxml file, Windows Embedded CE 6.0 opens the Catalog Editor. The visual interface is much like that of other Visual Studio editors and components.

Compatibility - To modify the supported CPUs for the Catalog item, enter the CPUs that you want to support in the Supported CPU field. By default, this field is empty. Specifying a CPU means that your Catalog item is supported by only the BSPs that support the same CPU.

General - If you want to include Help for your Catalog item, enter the link for the Help associated with the Catalog item in the Help Link field. The two size items are related to providing a size estimate in bytes for your catalog.

Identification - Comment and Description are optional and non essential but can be handy.

To modify the friendly name displayed in the Catalog, enter the Catalog item name that you want to use in the Title field.

To modify the unique ID associated with the Catalog item in the Catalog, enter the ID string that you want to use. Note: It is recommended that you use an ID string that begins with the Catalog record type and the vendor name to avoid Catalog collisions. For Catalog items, this uses the format Item:DefaultVendor:DefaultItemName.

Additional Variables - You can specify additional variables for the sysgen of this item using this item.

Modules - Specify the name of the item that will implement functionality such as BarcodeScanner1.DLL.

Notifications can be used to provide information to the user when the item is added to the catalog.

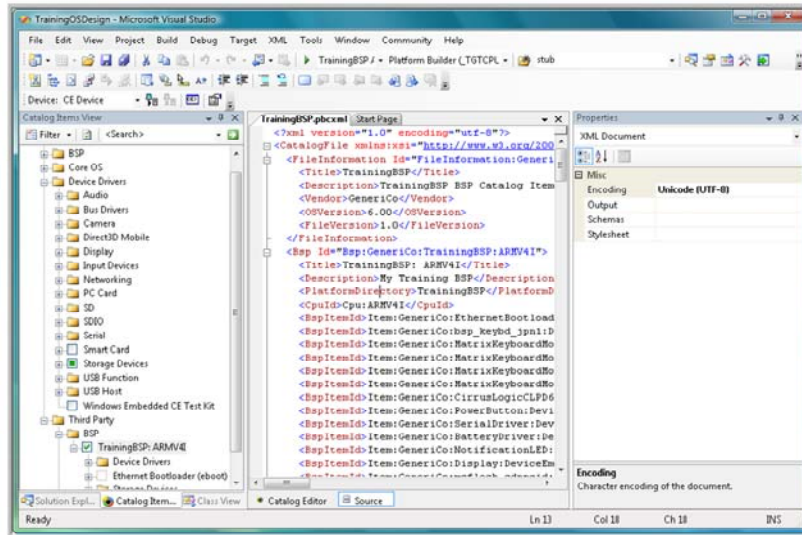
Source Code Link: In the Properties pane of the Catalog Editor, you can view and modify the default name and path properties of the source code link. To edit the name that is displayed, enter a friendly name for the source code path in the Title field. To edit the path for the source code link, enter the directory path or browse to the source code for the selected Catalog item in the Path field.

Location - Related to where the catalog item will appear in the catalog view.

Projects - One or more projects can be included with the catalog item

The Catalog (continued)

- Viewing/Editing in the XML View



You can use the Catalog Editor to view and edit Windows Embedded CE 6.0 Catalog item (.pbxml) files, which are XML-based files that contain metadata about the associated Catalog item.

In Windows Explorer, when you choose to open a .pbxml file, Windows Embedded CE 6.0 opens the Catalog Editor. The visual interface is much like that of other Visual Studio editors and components.

The Catalog (continued)

- **Select File -> New**
 - **Select Platform Builder Catalog File**
 - **Edit file as necessary**
 - **Save to one of the catalog locations**
 - %_WINCEROOT%\public\ - Microsoft catalog items
 - %_WINCEROOT%\platform\ - This is generally for BSPs
- %_WINCEROOT%\3rdParty\- 3rd Party catalog items that are not BSPs
- %_WINCEROOT%\platform\common\src\soc\- Microsoft BSP common catalog items

Typically the public and platform common paths are reserved for Microsoft.

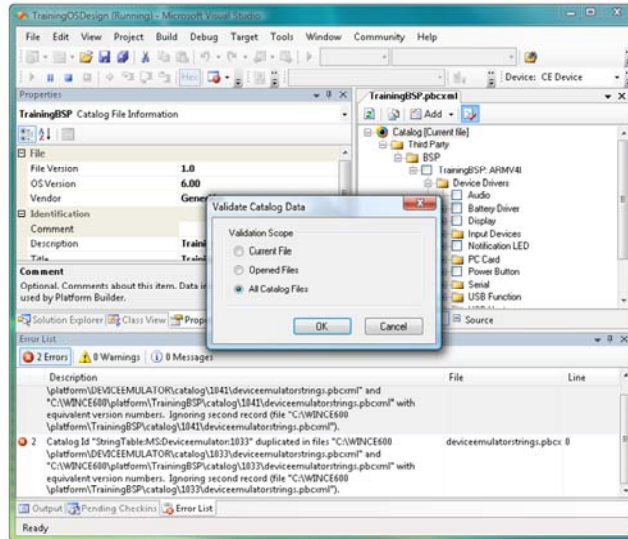
The Catalog (continued)

- **Exporting a Catalog Item from the Catalog**
 - In Windows Explorer, navigate to the location where your Catalog item (.pbcxml) file is located.
 - Copy the .pbcxml file and all related source code files into a new folder, and then zip the file for export.

Using independent XML files allows the catalog file to be managed by source control tools such as Microsoft Visual Source Safe.

The Catalog (continued)

- Validating a Catalog File



You can use the “Validate” item in the catalog editor to look for any issues. Use the tabbed dialog to look for any warnings as well as resolve any errors before moving on.

Customizing the OS Design

- The Catalog
- **Lab 8.1 – Adding an Item to the Catalog**
- The Shell Options
- Lab 8.2
 - Lab Goals
 - Understand how the Catalog works
 - Be able to add items to the catalog
- SDKs
- Lab 8.3
 - [Video](#)
- Review



Customizing the OS Design

- The Catalog
- Lab 8.1– Adding an Item to the Catalog
- **The Shell Options**
- Lab 8.2 • Installing a Custom Shell
- SDKS • Shell Shortcuts
- Lab 8.3 • Startup Folder
- Review • Typical Welcome Application

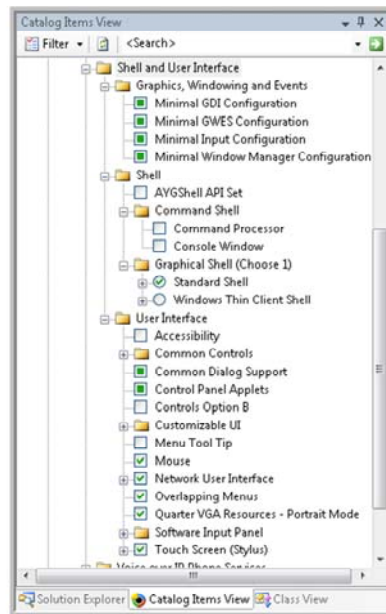


The Shell Options (continued)

- **A shell comprises:**
 - A set of user interface components
 - Underlying support routines
 - Could be a stand alone application
- **The shell architecture in Windows Embedded CE**
 - Allows you to implement a wide variety of shells.
 - Allows you to select only those components that you need to develop a custom shell, based on the hardware requirements of your device

The Shell Options (continued)

- Shell access from the catalog view



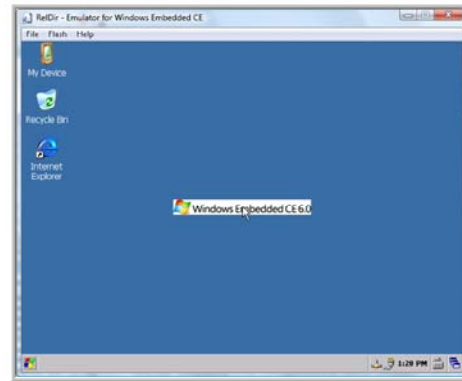
Command Processor Shell - Includes an application for a command-line-driven shell that provides console input and output and a limited number of commands. This functionality is also available on headless OS Designs.

The Standard Shell - Provides a shell that is similar to the shell on the Windows-based desktop operating systems. The source code for this shell is available for customization. %_WINCEROOT%\Public\Shell\OAK\HPC

Thin Client Shell - The Windows Thin Client design template provides the starting point for remote-desktop terminals through support for Microsoft RDP or other terminal software. Formerly known as Windows-based Terminal (WBT), the Windows Thin Client is a minimal version of Windows Embedded CE that includes the Catalog items necessary to support a Remote Desktop device — including a constrained shell and Microsoft RDP.

The Shell Options (continued)

- **Standard Explorer shell**
 - Provides start menu, task bar, desktop, wallpaper, etc...
- **Provides familiar look and feel**



Windows Embedded CE allows you to implement a wide variety of shells from simple command line interfaces to fully customized graphical user interfaces adapted for your target device. A Windows Embedded CE shell consists of modules and components that each provide a specific area of shell functionality.

The Shell Options (continued)

- **Command Shell**
 - Resembles the command.com in and cmd.exe in previous Windows versions
 - Is useful for headless devices with no display
 - Only interface is the command line, does not have GUI available
 - Is limited to standard C library I/O functions
 - Uses registry settings to direct to serial port

For many target devices, including those without a display, Windows Embedded CE includes a Command Processor shell that is similar to Command.com in Microsoft® Windows® 95 and Cmd.exe in Microsoft Windows NT®. It is a command-line-driven shell that provides a limited number of commands. To implement the Command Processor in an OS Design, include the Cmd and Console components in the Cesysgen.bat file. To use the Command Processor shell as a command-line interface for target devices with no displays, configure the Command Processor to operate over a serial port.

The following example shows how to set the registry values to allow the Command Processor to operate over a serial port.

```
[HKEY_LOCAL_MACHINE\Drivers\Console]
OutputTo = REG_DWORD:1          // Redirects CMD to COM1
COMSpeed = REG_DWORD:19200     // Speed of serial connection
```

The Shell Options (continued)

- **Sample Taskman Shell**
 - Is a starting point for developing a custom shell
 - Includes
 - Creates a full screen desktop window
 - Provides a Task Manager window
 - Provides a Run button
 - `%_WINCEROOT%\Public\Wceshellfe\Oak\Taskman`

TaskMan is a full-screen desktop window and a zero-height taskbar window that are registered with the Graphics, Windowing, and Events Subsystem (GWES) so that certain windows can be hidden behind the desktop. Shortcut keys, such as ALT+TAB, CTRL+ESC, and CTRL+ALT+BACKSPACE, are sent to the taskbar window.

A Task Manager window that lists all of the running top-level windows and enables a user to switch to or stop an application. The shortcut keys ALT+TAB, CTRL+ESC, and CTRL+ALT+BACKSPACE invoke the Task Manager window.

A Run button that enables a user to launch a file is also part of TaskMan.

The Shell Options (continued)

- **Application as shell**
 - Browser Control
 - Any application
 - Managed
 - Native
 - Not required to display
- **Examples of a Custom Shell**
 - Medical Monitoring Device
 - ATM
 - Industrial control system

Often you will not want your device to look like a standard Windows interface with a toolbar at the bottom of the screen and a bunch of icons on the desktop. A custom shell will allow you to make the GUI look like anything you want.

The Shell Options (continued)

- **Kernel loads applications listed in registry**
 - HKEY_LOCAL_MACHINE\Init
 - LaunchXX – Lists Application to load at boot time
 - DependXX – Lists Application dependencies

```
[HKEY_LOCAL_MACHINE\Init]
"Launch10"="shell.exe"
"Launch20"="device.dll"
"Depend20"=hex:0a,00
"Launch30"="gws.dll"
"Depend30"=hex:14,00
"Launch50"="explorer.exe"
"Depend50"=hex:14,00,1e,00
"Launch60"="MyShell.EXE"
"Depend60"=hex:14,00,1e,00
```

The Shell Options (continued)

- **Launched by Kernel at boot based on XX in LaunchXX**
- **Command line to App is an integer “token” as a string**
 - Cannot have other parameters
- **To allow dependent Apps to start, startup applications must call SignalStarted() with the token converted to an integer**

```
int WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
LPWSTR lpCmdLine, int nCmdShow)
{
    SignalStarted(_wtol(lpCmdLine));
    //...
}
```

Whenever a process is loaded by these launch keys it is given a command line parameter that is some token that is ultimately a string that is converted to an integer and used in a call to a function called SignalStarted. So in this example it takes the command line converts it from a Unicode string into an integer and calls SignalStarted.

SignalStarted

This function must be called by all applications that the kernel starts at startup through the HKEY_LOCAL_MACHINE\Init registry key.

The system passes the application its sequence identifier character string on the command line of the WinMain entry point.

Note The command line cannot be used to pass any information other than the sequence identifier. If an application must have information passed to it during boot, it can read the information from the registry or from a configuration file. When the application has finished initialization, it converts the string to a DWORD and passes it to SignalStarted. If SignalStarted is not called by such an application, other applications that are dependent on its launch will never run. If SignalStarted is called but the application does not run at startup, system operation will not be affected.

The Shell Options (continued)

- **Shell Shortcuts**

- Small file (ASCII text) that references a file in a different location
- Syntax is
 - nn#"<path>"
 - Where nn is the number of characters following the '#' in ASCII
- No need to copy entire EXE from ROM while allowing placement of the shortcut into a user defined folder structure

```
22#" \Windows\Welcome.exe"
```

The Shell Options (continued)

- **Startup Folder**

- At boot Standard shell looks at \Windows\Startup and runs programs (or shortcuts) located there
- Useful for launching applications at boot time
- Startup folder must be created in a custom .DAT file; does not exist by default

```
Directory("\Windows") :-Directory("StartUp")
Directory("\Windows\StartUp") :-
File("Welcome.lnk", "\Windows\Welcome.lnk")
```

The Shell Options (continued)

- Welcome Application Sample Code

```
#include "stdafx.h"
#include <shlwapi.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int
nCmdShow)
{
    HKEY hKey;
    DWORD dwType, dw=0;
    TCHAR startupPath[MAX_PATH];
    // Check for Touch Calibration Data
    if(ERROR_SUCCESS == RegOpenKeyEx(HKEY_LOCAL_MACHINE,
_T("HARDWARE\\DEVICEMAP\\TOUCH"), 0, NULL, &hKey))
    {
        if(ERROR_SUCCESS != RegQueryValueEx(hKey, TEXT("CalibrationData"), NULL, &dwType,
NULL, &dw))
        {
            RETAILMSG(1, (TEXT("No calibration data, attempting to calibrate.\r\n")));
            TouchCalibrate();
        }
        RegCloseKey(hKey);
    }
    // This will delete the link so it won't run again on warm boot.
    // This particular app is harmless to run again as it won't
    // calibrate the touch panel a second time. However, a full
    // "Welcome" app would be a bit annoying on every boot cycle.
    //
    if(SHGetSpecialFolderPath(NULL, startupPath, CSIDL_STARTUP, FALSE))
        DeleteFile(PathCombine(startupPath, startupPath, _T("TouchCal.lnk")));
    return 0;
}
```

This code shows one approach to running a "TouchCalibrate" program only once.

Customizing the OS Design

- The Catalog
- Lab 8.1 – Adding an Item to the Catalog
- The Shell Options
- **Lab 8.2 – Replacing the Standard Shell with IESHELL**
- SDKs
- Lab 8.3
 - Lab Goals
 1. Understand the fundamentals of implementing a custom shell
 - [Video](#)
- Review



Customizing the OS Design

- The Catalog
- Lab 8.1 – Adding an Item to the Catalog
- The Shell Options
- Lab 8.2 – Replacing the Standard Shell with IESHELL
- SDKs
- Lab 8.3 – Exporting an SDK
- Review
 - The files needed to build an application from Visual Studio for a device
 - SDK headers
 - Libraries
 - Help documentation
 - Other files

An SDK is a set of headers, libraries, connectivity files, run-time files, OS Design extensions, and Help documentation that developers use to write applications for a specific OS Design. The contents of an SDK allow developers to create and debug an application on the run-time image built from your OS Design.

You can use Platform Builder to develop an SDK based on your custom OS Design for installation on another development workstation. You should rebuild the runtime image and then build the SDK when you make changes to your OS Design.

During the SDK development process, Platform Builder tracks the core OS modules that belong to an OS Design, eliminating the need for you to describe the modules and components containing the technologies that the associated SDK should support. Instead, Platform Builder includes the headers and libraries associated with the modules and components in your OS Design in your SDK.

If you include a technology in your SDK that your OS Design does not support, a run-time error occurs when someone attempts to access that technology in the IDE.

You can only use Windows Embedded CE 6.0 SDKs with Microsoft Visual Studio® 2005 SP1 or greater to create, debug, and run custom applications.

SDKs (continued)

- **Prerequisite**
 - Build OSDesign
- **Creating**
 - From the Project menu select Add New SDK
- **Building**
 - From the Build menu select Build All SDKs
- **Installing**
 - The SDK MSI file can be moved to another computer and run.

Platform Builder can be used to build an SDK specific to an OS Design. By providing an SDK that is specific to your OS Design, you can accurately present to the application developer all of the capabilities and limitations of your OS Design. Other SDKs might not support all functionality supported by your OS Design, or they might support functionality that is not included in your OS Design.

Product name - Name exposed to OS during installations and uninstalls.

Company name - Name of your company.

Company Web site - Web site for your company.

Product version - Versioning to allow the installer to compare differing installations.

MSI Folder Path - Fully qualified path name for the location of the .msi file your users will run to install the SDK.

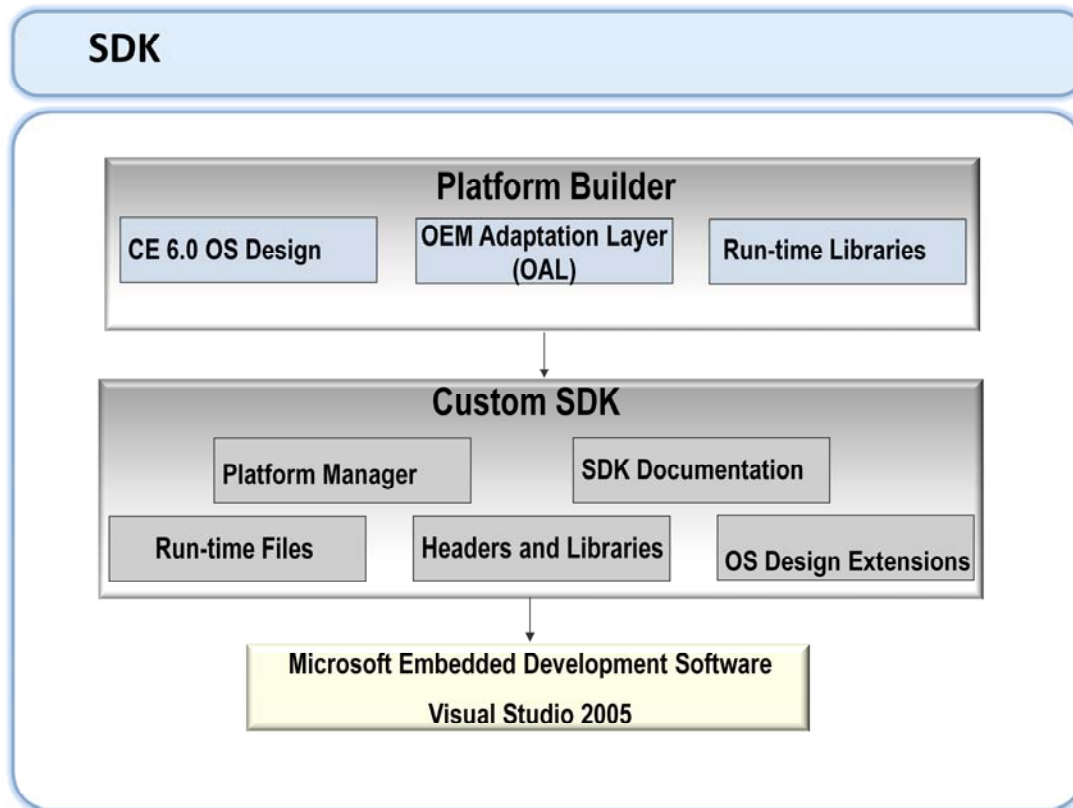
MSI File Name - Name of the installer program.

Locale - The locale you want to use for the user interface language.

After you create an SDK for an OS Design, you can add files to the SDK by choosing Additional Folders on the SDK Properties Page.

When you use Platform Builder to configure and build an SDK, the result is a MSI package. This package contains information required to install or uninstall an SDK, using the Windows Installer. The Installer automates the SDK installation process. During this process, it creates an entry for the SDK in the Add or Remove Programs dialog box under Control Panel on your development workstation. This allows a developer who installed your SDK to later remove it in a straightforward manner.

The MSI file can be moved and has a straight forward, click next install. You can over ride the install directory if so desired.



An SDK is a set of headers, libraries, connectivity files, run-time files, OS design extensions, and Help documentation that developers use to write applications for a specific OS design. The contents of an SDK allow developers to create and debug an application on the run-time image built from your OS design.

You can use Platform Builder to develop an SDK based on your custom OS design for installation on another development workstation. You should rebuild the runtime image and then build the SDK when you make changes to your OS design.

During the SDK development process, Platform Builder tracks the core OS modules that belong to an OS design, eliminating the need for you to describe the modules and components containing the technologies that the associated SDK should support. Instead, Platform Builder includes the headers and libraries associated with the modules and components in your OS design in your SDK.

If you include a technology in your SDK that your OS design does not support, a run-time error occurs when someone attempts to access that technology in the IDE.

You can only use Windows Embedded CE 6.0 SDKs with Microsoft Visual Studio® 2005 SP1 or greater to create, debug, and run custom applications.

SDK Development Steps

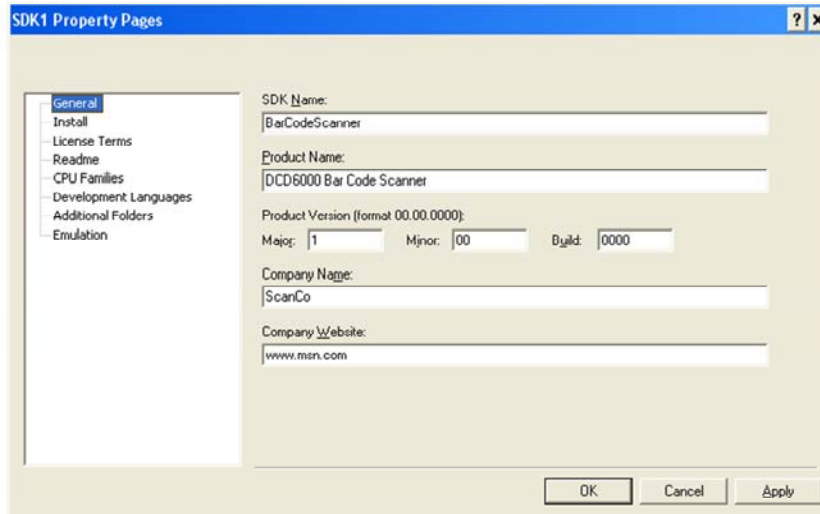
- **Build OS Design – create a run-time image**
- **Configure the SDK – use the SDK Wizard to configure basic settings**
- **Build the SDK - created SDK is a Microsoft Windows Installer (MSI) package**
- **Install the SDK on Visual Studio 2005**

As a developer of a Microsoft® Windows® CE–based OS design, you can provide an application developer with the information necessary to develop an application specifically for your OS design. You can use Microsoft Platform Builder to generate a software development kit (SDK) for your OS design.

An SDK supports the functionality that you include in your OS design. By providing an SDK that is specific to your OS design, you can accurately present to the application developer all of the capabilities and limitations of your OS design. Other SDKs might not support all functionality supported by your OS design, or they might support functionality that is not included in your OS design.

Configuring the SDK Options - General

Configurable SDK properties



The screenshot shows a dialog box titled "SDK1 Property Pages" with a "General" tab selected. The dialog contains the following fields:

- SDK Name: BarCodeScanner
- Product Name: DCD6000 Bar Code Scanner
- Product Version (format 00.00.0000): Major: 1, Minor: 00, Build: 0000
- Company Name: ScanCo
- Company Website: www.msn.com

Buttons at the bottom: OK, Cancel, Apply.

Product name - Name exposed to OS during installations and uninstalls.

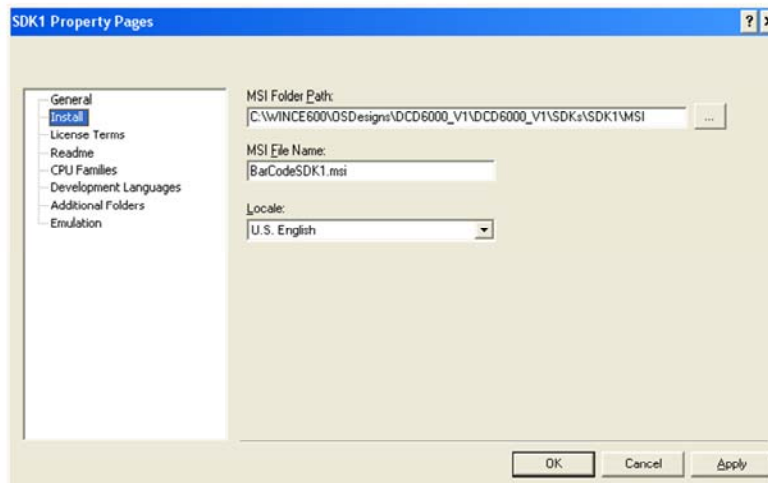
Company name - Name of your company.

Company Web site - Web site for your company.

Product version - Versioning to allow the installer to compare differing installations, using the format 00.00.0000.

Configuring the SDK Options - Install

Configurable MSI install properties



This category of the SDK Tool Property Pages Dialog Box enables you to configure installation properties for the SDK.

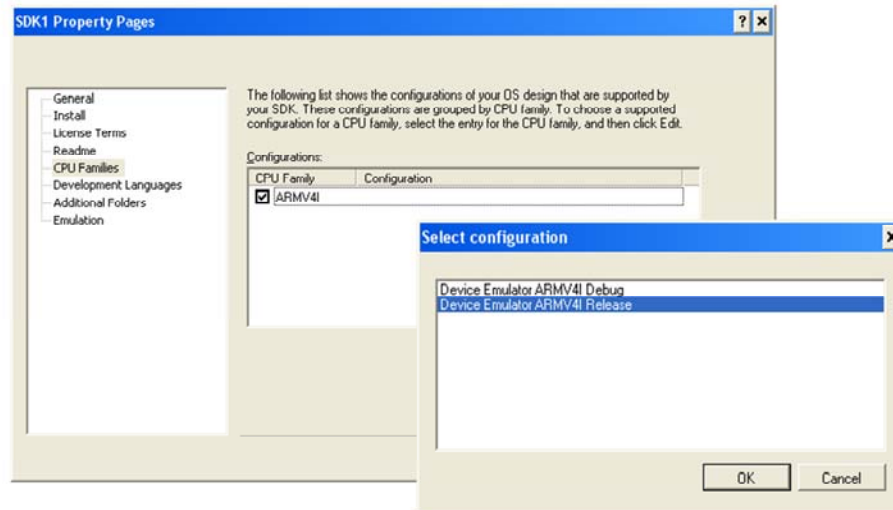
MSI Folder Path - Fully qualified path name for the location of the .msi file your users will run to install the SDK.

MSI File Name - Name of the installer program

Locale - The locale you want to use for the user interface language.

Configuring the SDK Options – CPU Families

- **Configuring CPU family supported by SDK**

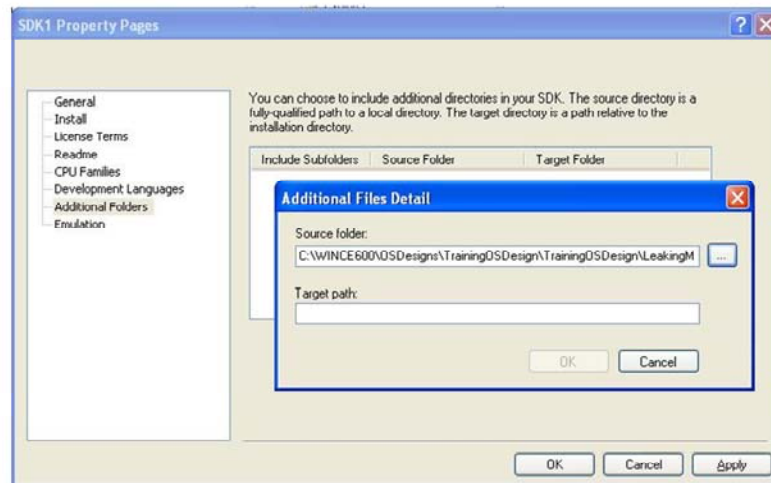


CPU Family

This category of the SDK Tool Property Pages Dialog Box enables you to select the CPU family OS configurations that you want your SDK to support installation properties for the SDK. The CPU families listed on this page are derived from the CPU families selected during Platform Builder setup. Check the box next to the CPU family for each OS configuration you want your SDK to support.

Configuring the SDK – Adding User-defined Files

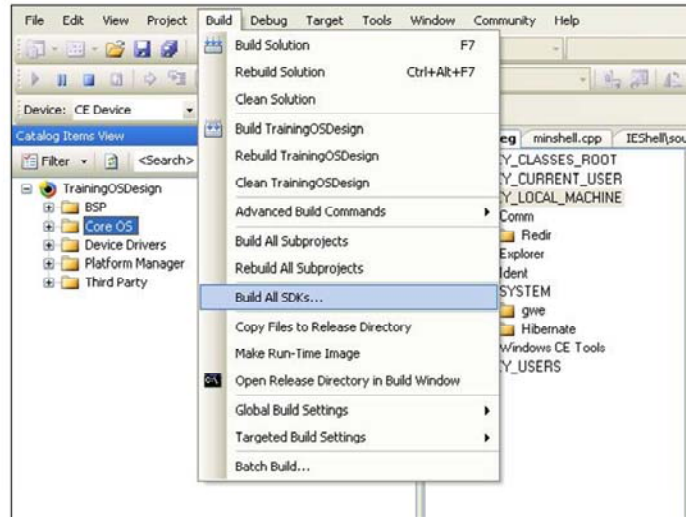
Add additional folders from the SDK's Properties Pages



After you create an SDK for an OS Design, you can add files to the SDK by choosing Additional Folders on the SDK Properties Page.

Building an SDK

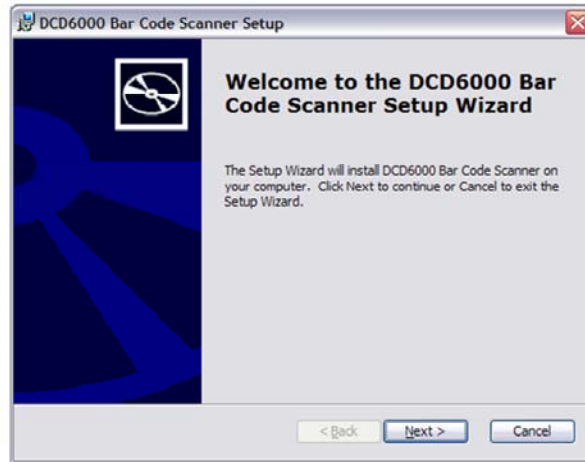
From the Build menu, choose Build All SDKs.



When you use Platform Builder to configure and build an SDK, the result is a Microsoft® Windows® Installer (MSI) package. This package contains information required to install or uninstall an SDK, using the Windows Installer. The Installer automates the SDK installation process. During this process, it creates an entry for the SDK in the Add or Remove Programs dialog box under Control Panel on your development workstation. This allows a developer who installed your SDK to later remove it in a straightforward manner.

Installing the SDK

- **Installing the SDK on other development machines (with VS 2005 SP1)**



The MSI file can be moved and has a straight forward, click next install. You can over ride the install directory if so desired.

Customizing the OS Design

- The Catalog
- Lab 8.1 – Adding an Item to the Catalog
- The Shell Options
- Lab 8.2 – Replacing the Standard Shell with IESHELL
- SDKs
- **Lab 8.3 – Exporting an SDK**
- **Review**

- Lab Goals

1. Be able to create an SDK for native code development in Visual Studio 2005

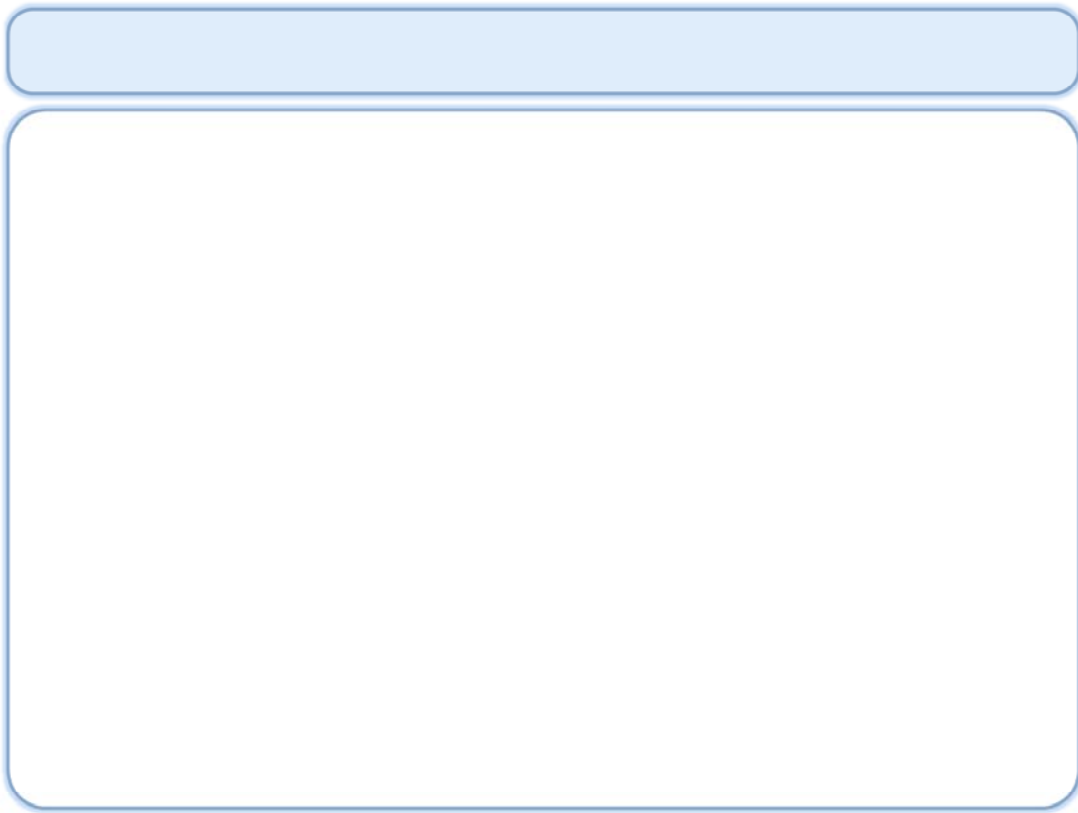
- [Video](#)



Customizing the OS Design

- The Catalog
- Lab 8.1 – Adding an Item to the Catalog
- The Shell Options
- Lab 8.2 – Replacing the Standard Shell with IESHELL
- SDKs
- Lab 8.3 – Exporting an SDK
- **Review**





Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Application Development

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- **Module 9: Application Development**
- Module 10: Testing & Verification
- Course Review



Application Development

- **Application Development Options**
- **Native Code Development**
- **Managed Code Development**
- **Lab 9.1 – Developing & Integrating a Managed App**
- **Review**

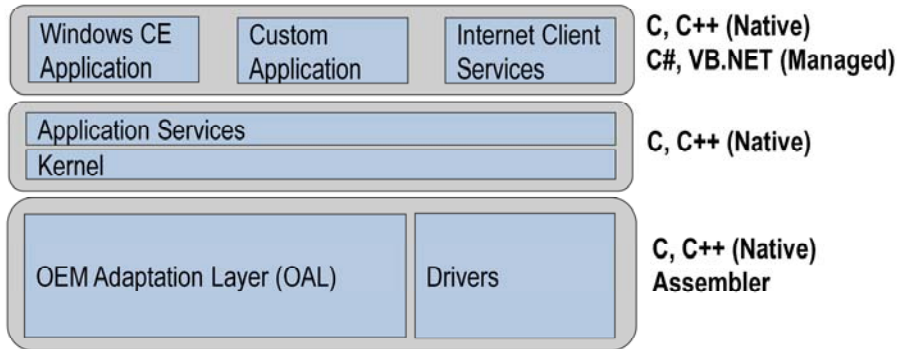


Application Development

- **Application Development Options**
- Native
- Managed
- Lab 9
- Review
- **Tools**
 - Visual Studio 2005 SP1 (w/ Windows Embedded CE add-on) - Subproject of OS Design
 - Visual Studio 2005 SP1 - Smart Device Project
- **Languages**
 - C/C++, C#, VB and Assembler
- **APIs/Frameworks**
 - WIN32 (Native)
 - MFC, ATL, WTL, STL (Native)
 - .NET Compact Framework (Managed)



Language Options



Application Development Options (continued)

- **Language/API/Framework Options**

	OAL	Drivers	Services	Applications	Shell	Internet Client Services
C# (Managed)				.NET CF	.NET CF	.NET CF
VB (Managed)				.NET CF	.NET CF	.NET CF
C/C++ (Native)	WIN32	WIN32	WIN32	WIN32, MFC, ATL, WTL, STL	WIN32, MFC, ATL, WTL, STL	WIN32, MFC, ATL, WTL, STL
Assembler (Native)	WIN32					

Application Development

- Application Development Options

- **Native Code Development**

- Managed Code Development

- Lab 9

- Review

- **Native Applications**

- Must be rebuilt for each new CPU or Platform
- Requires SDK
- Developer manages system resources
- WIN32 runs without extra support files
- Can access all operating system services and APIs
- Must be rebuilt to run on desktop systems
- Supports COM, ActiveX programming



Native Code Development (continued)


- **Native Code Frameworks**

- Microsoft Foundation Class Library (MFC)
 - Object-oriented application framework
- Active Template Library (ATL)
 - Supports implementation of COM and ActiveX components
- Windows Template Library (WTL)
 - A C++ library for developing Windows applications and UI components.
- Standard Template Library (STL)
 - A C++ library of container classes, algorithms, and iterators
- Frameworks ship with VS2005 not CE 6.0
 - Frameworks must be manually integrated into OS Design

Application Development

- Application Development Options
- Native Code Development
- **Managed Code Development**
- Lab 9
- Review

- **Managed Applications:**
 - Built once for all devices
 - Runtime engine manages system resources
 - Requires runtime support files (.NET CF)
 - Applications access the services exposed by the Compact Framework
 - May run directly on desktop without rebuilding



The choice on CE between Native and Managed is parallel to the choice you would make on a desktop.

If the managed application uses device specific functionality not available in the desktop Framework (Such as phone call management etc...) then it will not run on the desktop as there are no corresponding libraries for that.

Managed applications access the services exposed by the Compact Framework, however, they can "escape" the framework using Platform Invoke (P/Invoke) to call native APIs.

Be aware that the Compact Framework has background threads, such as garbage collection, that can impact the (RTOS) Real Time Operating System behavior. Be sure and assess the impact of that behavior on your applications with the specific device architecture that you are working with.

Managed Code Development (continued)

- **Managed Application Development**
 - Languages all compile to Intermediate Language format
 - C# application development
 - Visual Basic .NET application development
 - .NET Compact Framework (CF)
 - Device-side runtime support package for .NET applications
 - Common Language Runtime (CLR)
 - Execution engine to manage .NET applications
 - Just-In-Time compiler for intermediate language format
 - .NET Class Library
 - Form-related classes, Data and XML classes, and GDI support
 - Subset of desktop .NET Framework

The .NET Compact Framework is a subset of the full size desktop framework. The .NET Compact Framework is a hardware-independent program execution environment for secure downloadable applications optimized for resource-constrained computing target devices. It offers a choice of languages, initially Microsoft Visual Basic and Microsoft Visual C#, and eliminates some of the common problems faced with language interoperability.

Application Development

- Application Development Options
- Native Code Development
- Managed Code Development
- **Lab 9.1 – Developing & Integrating a Managed App**
- Rev

- Lab Goals

1. Develop and debug managed applications in a separate Visual Studio 2005 instance
2. Integrate a managed application into a BSP

- [Video](#)



Application Development

- Application Development Options
- Native Code Development
- Managed Code Development
- Lab 9.1 – Developing & Integrating a Managed App
- **Review**



Windows Embedded Training

**Building Solutions with
Windows Embedded CE 6.0 R2**

Testing & Verification

Course Outline

- Course Introduction
- Module 1: Operating System Overview
- Module 2: Tools for Platform Development
- Module 3: Operating System Internals
- Module 4: Operating System Components
- Module 5: The Build System
- Module 6: The Board Support Package
- Module 7: Device Driver Concepts
- Module 8: Customizing the OS Design
- Module 9: Application Development
- **Module 10: Testing & Verification**



Testing & Verification

- **Windows Embedded CE Test Kit**
- **Other Test Utilities**
- **Lab 10.1 - Using the CETK**
- **Review**



Testing & Verification

- **Windows Embedded CE Test Kit**
- Other **Test Utilities**
- **Lab 10**
- **Review**
 - Collection of tools, extensible tests and a test automation harness for testing CE based devices



Windows Embedded CE Test Kit (continued)

- **Overview**
 - Can be extended with custom test DLLs
 - Only part of the solution
 - Custom Apps not covered
 - Custom IOCTLs and drivers
 - System InterOp not covered
 - Microsoft provided automated test harness
 - Client/Server Architecture
 - Automated test loading via “Tux”
 - Actual tests implemented as DLLs loaded by TUX
 - Common logging Engine “Kato”
 - DLL exposes C and C++ API for logging to the server
 - Common peripheral test binaries

The Windows Embedded CE 6.0 Test Kit (CETK) is a tool that you can use to test device drivers that you develop for the Windows Embedded CE operating system (OS). The CETK incorporates a collection of tests into a graphical user interface (GUI) harness. The test tools in the CETK support the CPUs and hardware platforms that Windows Embedded CE supports.

These tools can also be called via the command line for the automated execution of custom test suites. CETK is used internally at Microsoft to test platforms and drivers

CETK can be setup to be a fast, automated way of running tests and verifying the stability and reliability of a device and drivers through out a project.

CETK is included with Platform Builder, nothing else to download or purchase and is also available as standalone download.

The CETK server UI is launched from the start menu under Programs\Windows Embedded CE Platform Builder after the in-product tests are added to the catalog of your project.

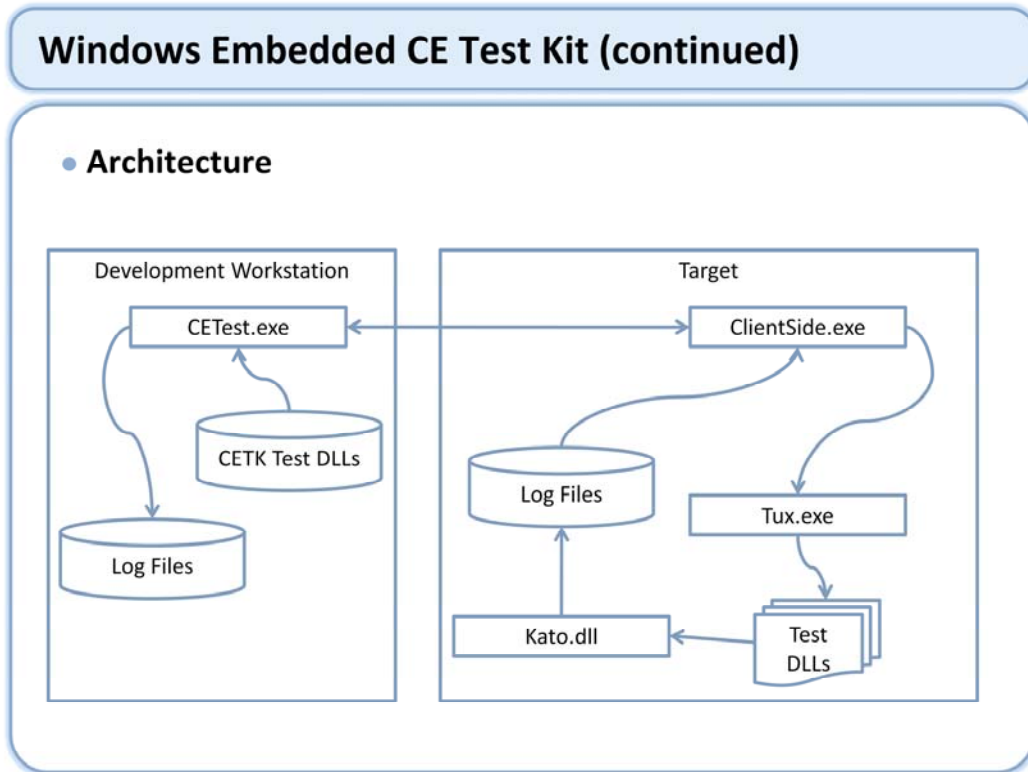
The device/host connection can be through KITL, ActiveSync, or Sockets. If a connection can't be established via the previously mentioned options, then you could copy appropriate test binaries and run manually via command line or shortcuts. CETK ships with a set of tests that can be used to test third-party drivers for Windows Embedded CE.

CETK can be used for:
Testing Device Drivers
Application testing
Stress Testing
Performance Testing

Windows Embedded CE Test Kit (continued)

- **Overview (continued)**
 - Expandable
 - Custom tests can be added with the User-Defined Test Wizard
 - Can define own test suite
 - The results of each test are displayed in an easily readable format via cetkpar.exe
 - Multiple devices can be managed from a single server
 - Only one ActiveSync connection is allowed
 - KITL and sockets allow multiple device connections
 - Automatic peripheral detection on a device

The CETK client can connect to any machine running the CETK server. Any device can be a CETK client by downloading clientside.exe to the device.



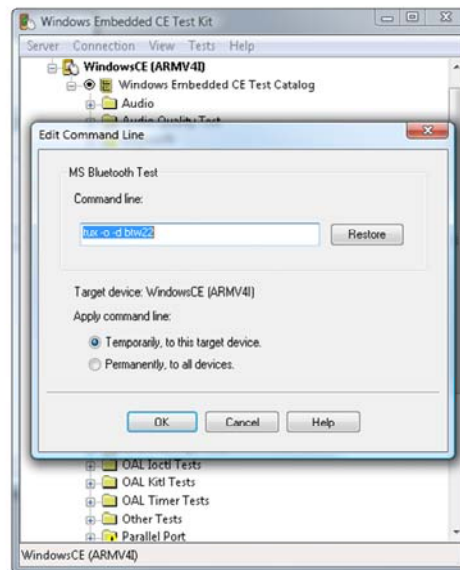
ClientSide.exe is used to communicate with the desktop CETK server. It launches TUX.EXE with the command line options specified from the desktop Server UI. It is possible to load TUX.EXE in a stand-alone fashion without the use of the remote server. Microsoft provides a number of TUX test DLLs for testing common drivers and aspects of the system. You can also create your own TUX test DLLs for custom drivers.

Windows Embedded CE Test Kit (continued)

- **CETest.exe**
 - Host PC GUI component
 - Allows connecting to device
 - Allows selecting test to execute
 - Allows changing parameters
 - Sends testing package to device
 - Provides some status on test progress
 - C:\Program Files\Microsoft Platform Builder\6.00\CEPB\wcetk

Windows Embedded CE Test Kit (continued)

- **CETest.exe**
 - Context menu allows
 - Editing the command line for the test
 - Access to results
 - Access to test information in the help system



The help system provides more information about each test and the corresponding command line parameters. The exclamation indicates that a test MAY not be available.

Windows Embedded CE Test Kit (continued)

- **Clientside.exe**
 - Device side component that receives test packages from CETEST and invokes TUX
 - Invocation Options
 - Manual
 - If DNS (name resolution) available:
Clientside.exe /n=<HOST NAME> /p=5555
 - If DNS is not available:
Clientside.exe /i=<IP Address> /p=5555
 - Populate device registry
 - HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\CETT
 - WCETK.txt
 - SERVERNAME=MyWorkstation
 - PORTNUMBER=5555
 - AUTORUN=0
 - DEFAULTSUITE=My Suite

Device Side Software

The device side of CETK is called Clientside.exe. Clientside.exe is a small application that is responsible for calling to the host computer to establish a connection, device detection, launching tests, and sending back the results to the host.

Windows Embedded CE Test Kit (continued)

- **TUX Execution Engine**
 - TUX.EXE – process that hosts TUX test DLLs
 - Launched by Clientside.exe
 - Can run Standalone on the device as well
 - Advantages of standalone process
 - Clientside process separation
 - Avoids UI interaction
 - Allows standalone execution

Windows Embedded CE Test Kit (continued)

• Tux Command Line Parameters

-b	Breaks after each Tux DLL loads.
-e	Disables exception handling.
-s filename	Specifies the Tux suite file to load and execute.
-d test_dll	Specifies the Tux DLL to load and execute.
-c parameters	Command line to pass to the Tux DLL.
-r seed	Specifies the integer starting random seed.
-x test_case	Specifies which test cases to run. You can specify a single test case or a range of test cases, as shown in the following command line. e.g. tux -x10,12,15-20
-l	Lists the test cases in the Tux DLL specified by the -d parameter.
-lv	With greater verbosity, lists the test cases in the Tux DLL specified by the -d parameter.
-t address	Specifies the name of the computer running the CETK server. Use -t with no arguments to specify a local server.
-n	Runs tests in kernel mode.
-h	Displays the list of command line parameters for Tux.

Windows Embedded CE Test Kit (continued)

• Tux Command Line Parameters (continued)

The following Tux parameters are enabled when Kato.dll is present.

-k address	Specifies the name of the computer running the CETK server. Use -k with no arguments to specify a local server.
-m	Logs all Kato output as XML.
-o	Logs all Kato output to the debugger.
-f filename	Logs all Kato output to output file filename.
-a	Appends data to the output file. Use this parameter with the -f parameter.

The following Tux parameter is enabled when Toolhelp.dll is present.

-z timeout	Cancels an existing run of the Tux DLL that is specified by the -d parameter.
------------	---

Windows Embedded CE Test Kit (continued)

- **KATO**
 - Logging library that tux tests can use for reporting test results
 - Linking to KATO.dll and creating a KATO object allows for routing output to multiple destinations

Windows Embedded CE Test Kit (continued)

- **Test Source Code**
 - Platform Builder ships most of the source files to help identify and fix problems and build custom tests for your platform
 - The source files are only installed when you select the private source tree and agree to the shared source licensing agreement
 - Most tests can be rebuilt with no modifications
 - Located at %_WINCEROOT%\Private\Tests

You can install source code for CETK tests by installing Windows Embedded CE Shared Source from the Setup wizard for Windows Embedded CE. Test source code will be placed in the Private tree under the Tests directory.

Windows Embedded CE Test Kit (continued)

- **TUX Skeleton**

- A pre-defined skeleton to help develop new custom tests that leverages from existing CETK infrastructure
- Located at:
 - C:\Program Files\Microsoft Platform Builder\6.00\cepb\wctk\tux\tuxskel
- Test Cases are handled by test procedures
- Function Table associates test case ID with test procedure

Windows Embedded CE Test Kit (continued)

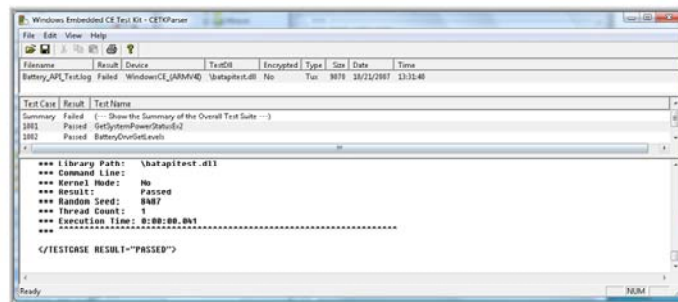
- **Log Files**

- **Stored at:**

- ..\Program Files\Microsoft Platform Builder\6.00\CEPB\WCETK\results

- **Sorted by:**

- <device name>\<date>\<time>\<test name>.log



The screenshot shows the 'Windows Embedded CE Test Kit - CETKParser' window. It displays a table of test results and a detailed log for a specific test case.

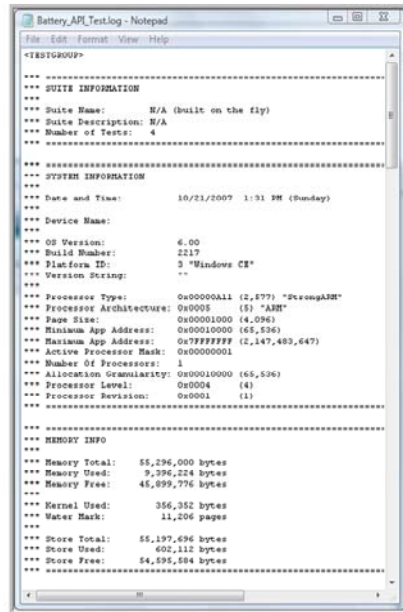
Filename	Result	Device	TestID	Encrypted	Type	Size	Date	Time
BatteryAPI_Test.log	Failed	WindowsCE_ARMV4Q	Ubatapitest.dll	No	Yes	3078	10/21/2007	13:32:48

Test Case	Result	Test Name
Summary	Failed	[-- Show the Summary of the Overall Test Suite --]
1001	Passed	GetSystemPowerStatus
1002	Passed	BatteryOpenGetLevels


```
*** Library Path: Ubatapitest.dll
*** Command Line:
*** Kernel Mode: No
*** Result: Passed
*** Random Seed: 8987
*** Thread Count: 1
*** Execution Time: 0:00:00.001
***
</TESTCASE RESULT="PASSED">
```

Windows Embedded CE Test Kit (continued)

- Log Header



```
Battery_API_Test.log - Notepad
File Edit Format View Help
-----
<<TESTGROUP>
-----
*** SUITE INFORMATION
***
*** Suite Name: N/A (built on the fly)
*** Suite Description: N/A
*** Number of Tests: 4
***
-----
*** SYSTEM INFORMATION
***
*** Date and Time: 10/21/2007 1:31 PM (Sunday)
***
*** Device Name:
***
*** OS Version: 6.00
*** Build Number: 2217
*** Platform ID: 3 "Windows CE"
*** Version String: ""
***
*** Processor Type: 0x00000011 (2,877) "StrongARM"
*** Processor Architecture: 0x0005 (5) "ARM"
*** Page Size: 0x00010000 (4,096)
*** Minimum App Address: 0x00010000 (65,536)
*** Maximum App Address: 0x7FFFFFFF (2,147,483,647)
*** Active Processor Mask: 0x00000001
*** Number Of Processors: 1
*** Allocation Granularity: 0x00010000 (65,536)
*** Processor Level: 0x0004 (4)
*** Processor Revision: 0x0001 (1)
***
-----
*** MEMORY INFO
***
***
*** Memory Total: 55,296,000 bytes
*** Memory Used: 3,396,224 bytes
*** Memory Free: 45,899,776 bytes
***
*** Kernel Used: 356,352 bytes
*** Water Mark: 11,206 pages
***
*** Store Total: 55,197,696 bytes
*** Store Used: 602,112 bytes
*** Store Free: 54,595,584 bytes
***
-----
```

Windows Embedded CE Test Kit (continued)

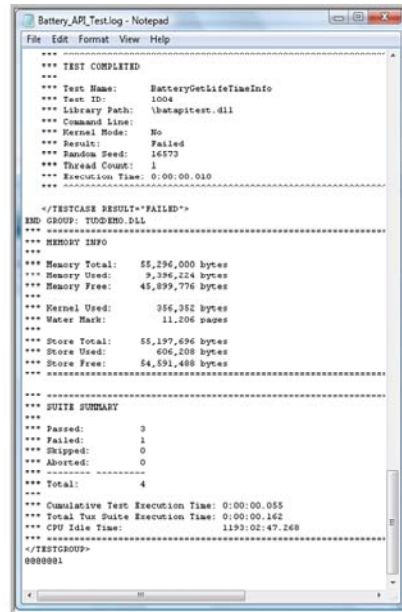
- Log Test Case Results



```
Battery_API_TestLog - Notepad
File Edit Format View Help
BEGIN GROUP: FSDTEST.DLL
<TESTCASE ID=1001>
***
*** TEST STARTING
***
*** Test Name: GetSystemPowerStatusEx2
*** Test ID: 1001
*** Library Path: \batapitest.dll
*** Command Line:
*** Kernel Mode: No
*** Random Seed: 8487
*** Thread Count: 0
***
*****
BEGIN TEST: "GetSystemPowerStatusEx2", Threads=0, Seed=8487
Try calling GetSystemPowerStatus with correct parameters
Calling GetSystemPowerStatusEx2(c, s, TRUE)
Calling GetSystemPowerStatusEx2(c, s, TRUE) succeed!
Line status is AC (0x1)
Main battery flag Off, 009, 4294967295 (0xffffffff) secou
Backup battery flag Off, 009, 4294967295 (0xffffffff) secou
4 mV, 0 mA, avg 0 mA, avg interval 0, consumed 0 mAh, 200 C,
Calling GetSystemPowerStatusEx2(c, s, FALSE)
calling GetSystemPowerStatusEx2(c, s, FALSE) succeed!
Line status is AC (0x1)
Main battery flag Off, 009, 4294967295 (0xffffffff) secou
Backup battery flag Off, 009, 4294967295 (0xffffffff) secou
4 mV, 0 mA, avg 0 mA, avg interval 0, consumed 0 mAh, 200 C,
Calling GetSystemPowerStatusEx2(c, right size + 1, TRUE)
Calling GetSystemPowerStatusEx2(c, right size + 1, TRUE) suc
Calling GetSystemPowerStatusEx2(c, right size - 1, s)
GetLastError is 87.
Calling GetSystemPowerStatusEx2(c, 0, s)
GetLastError is 87.
Calling GetSystemPowerStatusEx2(NULL, s, s)
GetLastError is 87.
END TEST: "GetSystemPowerStatusEx2", PASSED, Time=0.041
***
*** TEST COMPLETED
***
*** Test Name: GetSystemPowerStatusEx2
*** Test ID: 1001
*** Library Path: \batapitest.dll
*** Command Line:
*** Kernel Mode: No
*** Result: Passed
*** Random Seed: 8487
*** Thread Count: 1
*** Execution Time: 0:00:00.041
***
*****
```

Windows Embedded CE Test Kit (continued)

- Log Footer



```
Battery_API_Test.log - Notepad
File Edit Format View Help
*** TEST COMPLETED ***
*** Test Name: BatteryGetLifetimeInfo ***
*** Test ID: 1004 ***
*** Library Path: \batapitest.dll ***
*** Command Line: ***
*** Kernel Mode: No ***
*** Result: Failed ***
*** Pseudo Seed: 16573 ***
*** Thread Count: 1 ***
*** Execution Time: 0:00:00.010 ***
-----
~/TESTCASE RESULT="FAILED">
END GROUP: TID0280.DLL
*** MEMORY INFO ***
*** Memory Total: 55,296,000 bytes ***
*** Memory Used: 9,396,224 bytes ***
*** Memory Free: 45,899,776 bytes ***
*** Kernel Used: 356,362 bytes ***
*** Water Mark: 11,206 pages ***
*** Store Total: 55,197,696 bytes ***
*** Store Used: 606,208 bytes ***
*** Store Free: 54,591,488 bytes ***
-----
*** SUITE SUMMARY ***
*** Passed: 3 ***
*** Failed: 1 ***
*** Skipped: 0 ***
*** Aborted: 0 ***
*** Total: 4 ***
*** Cumulative Test Execution Time: 0:00:00.055 ***
*** Total Test Suite Execution Time: 0:00:00.162 ***
*** CPU Idle Time: 1193:02:47.268 ***
-----
~/TESTGROUP>
0000001
```

Windows Embedded CE Test Kit (continued)

- **Result Codes**

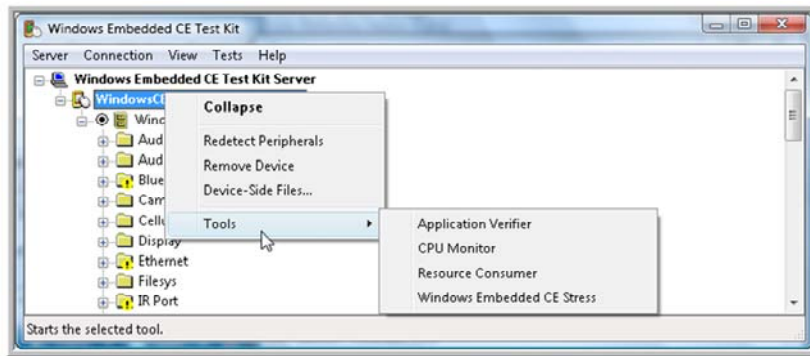
- **PASS**
 - The test case ran to completion. All behavior observed by the test case is valid.
- **FAIL**
 - The behavior of the tested functionality is not valid.
- **SKIP**
 - The test case did not run.
- **ABORT**
 - The test case did not run to completion because an error occurred. The test case could not verify the tested functionality.

Windows Embedded CE Test Kit (continued)

- **Where To Get More Information On CETK**
 - Each test is documented online MSDN and Platform Builder help system
 - The documentation includes detailed describes of each test case and any optional command line parameters that may be used
 - Windows Embedded Test Blog
 - <http://blogs.msdn.com/testembedded/default.aspx>

Windows Embedded CE Test Kit (continued)

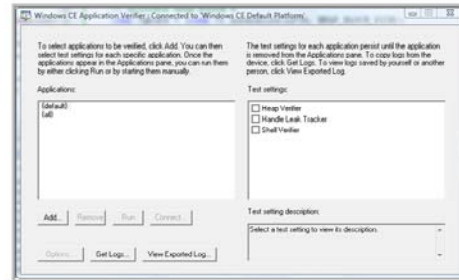
- Access to other tools



Windows Embedded CE Test Kit (continued)

- **Application Verifier Tool**

- Detects memory & resource leaks in applications
- Uses Shims
 - Inserts into code path between calling function and the intended target function
- Microsoft provides 3 shims
 - Heap Verifier – finds memory leaks and heap corruption
 - Handle Leak Tracker – finds handle leaks ex. Registry, event, semaphores, or critical sections
 - Shell Verifier – finds GDI & User defined object leaks
- Can also be used on Device Drivers

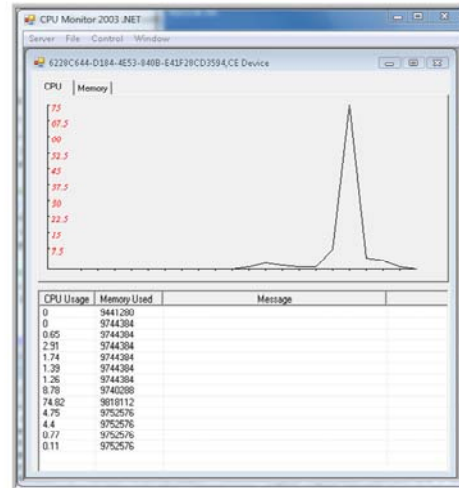


When using Application Verifier, all the functionality and paths on the application should be exercised to uncover any leaks.

Windows Embedded CE Test Kit (continued)

• CPU Monitor

- As an alternative to Perfmon, CPU Monitor tool shows CPU and memory usage
- CPU Monitor tool can run by itself or from the CETK window
- Can save stored data in XML



The CPU Monitor tool shows you the CPU and memory usage of a Windows Embedded CE–based device in the CPU Monitor for Windows Embedded CE window. The target device sends data through a network connection to the development workstation. The development workstation logs and displays the data in a graph and a list in the CPU Monitor for Windows Embedded CE window.

You can run the CPU Monitor tool by itself or you can run the tool from the Windows Embedded CE 6.0 Test Kit (CETK) window. Prior to running the CPU Monitor tool, you must install the Microsoft .NET Framework on the development workstation. You can download the .NET Framework from Microsoft Windows Update or from MSDN, the Microsoft developer program website.

To run the CPU Monitor tool from the CETK window

Add support for the CETK to the Windows Embedded CE operating system (OS) for the target device.

Connect the target device to the CETK.

Click on Tools, Windows Embedded CE Test Kit.

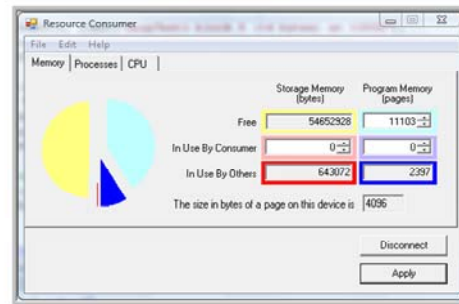
In the CETK window, right-click on the target device from which you want to view CPU and memory usage information, choose Tools, and then choose CPU Monitor.

The CPU Monitor tool receives data on port 8000 of the development workstation. You cannot change the port number over which the CPU Monitor tool receives data at this point in time.

You can save the data collected by the CPU Monitor tool in an XML file. The XML file includes the XML schema and data in XML format. You can use the DataTable object from Microsoft Visual Studio to read the XML file.

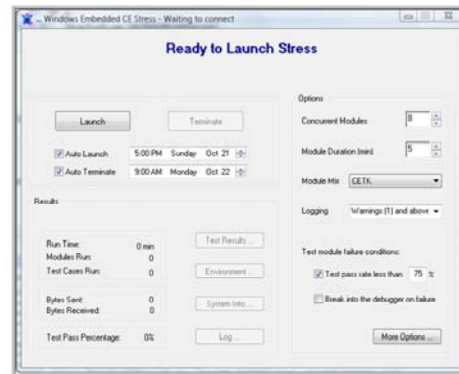
Windows Embedded CE Test Kit (continued)

- **Resource Consumer**
 - Used to load the device to specific resource conditions and then run user and/or stress scenarios.



Windows Embedded CE Test Kit (continued)

- **Windows Embedded CE Stress Tool**
 - Flexible system for all types of stress testing scenarios
 - Based on a set of modules that exercise individual components, features, or applications
 - The Stress harness controls the environment in which the modules operate
 - Stress harness monitors the stress run and collects and reports data
 - Can create custom Stress modules with Stress harness



The Modular Stress harness controls the environment in which the modules operate, you can control:

- Module "mix"
- Number of concurrent modules
- Module lifespan

Modular Stress harness monitors the stress run and collects and reports data:

- Individual module test case results
- System memory state
- Hang detection

Best Practice – "Testing the Test"

As a recommended best practice you may want to run a generic platform with out any customization to assure your self that the modular stress test runs with out issues. If a stress test fails on it's own in your environment then that needs to be fixed as a starting point. After it will run as long as you need it to you can then add in your customized DLLS and apps. If a failure occurs then, it is likely due to you code not the functionality of the test in your environment.

Testing & Verification

- Windows Embedded CE Test Kit
- Other Test Utilities
- Lab 10-1 Using the CFTK
- Review
 - Scripting Host Tool
 - Cescrypt.exe, is a ActiveX scripting host. With the tool, you can run a script independent of a Web browser on a the target device
 - Print Screen Tool
 - Prt_scrn.exe takes a screen shot and saves it as a bitmap file on the target device



Cescrypt.exe:

<http://msdn2.microsoft.com/en-us/library/aa934625.aspx>

Cescrypt.exe supports Microsoft Jscript and Microsoft VBScript programming language.

Prt_scrn.exe:

<http://msdn2.microsoft.com/en-us/library/aa934067.aspx>

Testing & Verification

- [Windows Embedded CE Test Kit](#)
- [Other Test Utilities](#)
- **Lab 10.1 - Using the CETK**
- [Review](#)

- **Lab Goals**

1. Run automated tests using the Windows Embedded CE Test Kit (CETK)
2. Modify the default behavior of the standard tests

- [Video](#)



Testing & Verification

- Windows Embedded CE Test Kit
- Other Test Utilities
- Lab 10.1 - Using the CETK
- **Review**



Windows Embedded Training

Building Solutions with Windows Embedded CE 6.0 R2

Appendix

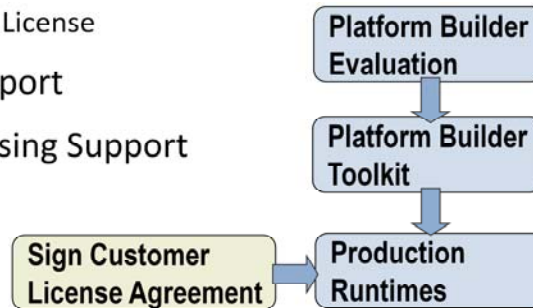
Appendix Introduction

- **Licensing Overview**
- **CE 6.0 License Types**
- **Licensing Tool**
- **License Agreements**
- **Purchasing Licenses**
- **Developer Resources**

Licensing Overview

- **Microsoft Embedded Authorized Distributor**

- Platform Builder Toolkit
- Certificate of Authenticity (COA)
 - Core License
 - Professional License
- Technical Support
- General Licensing Support



The licensing process begins with getting a free Platform Builder Toolkit Evaluation from an Embedded Authorized Distributor, this is followed by a purchase of the full version of the Platform Builder Toolkit for \$995. This toolkit allows the OEM the ability to develop, test, and debug the image using the hardware emulator that is included in the toolkit. The toolkit also includes one product identification number (PID) for testing on hardware, but cannot be commercially shipped. In order to purchase runtime licenses the OEM must sign an agreement with Microsoft called the Microsoft OEM Customer License Agreement (CLA). Once in production a runtime license must be purchased for each individual device. Platform Builder Evaluation is for 180 days. The OEM must purchase full version in order to create commercial image.

No need to sign customer license agreement or purchase licensing until product is to be shipped.

CE 6.0 License Types

- **Core License**
 - Hard real-time operating system kernel
 - File system
 - Networking and communications technologies
 - Multimedia capabilities
 - Digital rights management
 - Application development platform

The Core license is ideal for low-cost devices, such as gateways, entry-level voice over IP (VoIP) phones, industrial automation equipment, and consumer electronic devices such as CD players, digital cameras, and networked DVD players.

The number of licenses purchased, will determine the price.

CE 6.0 License Types

- **Professional License**

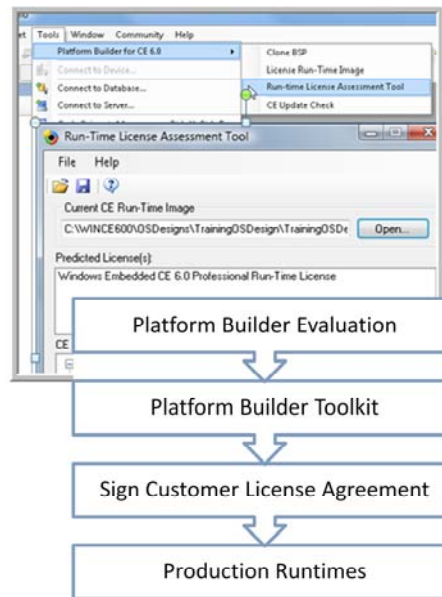
The Professional run-time license provides all the Core features plus the following:

- Remote Desktop Connection
- Windows Messenger
- WordPad
- Handwriting Recognizer
- Internet Explorer
- Streaming Media Playback
- Windows Media Player

Considering the cost variance between Core and Professional many OEMs evaluate carefully the cost of developing software features versus licensing additional features from Microsoft.

Licensing

- **Run-Time License Assessment Tool**
 - Used to predict license from a image file
- **Contact a Microsoft Embedded Authorized Distributor**
 - Microsoft OEM Customer License Agreement (CLA)
 - Required when ready to begin commercial shipment
 - Agreement between Microsoft and the OEM
 - Distributed by Microsoft Embedded Authorized Distributors
 - Not required for the purchase of Toolkit
 - One year agreement; automatically renews for a second year



The licensing process begins with getting a free Platform Builder Toolkit Evaluation from an Embedded Authorized Distributor, this is followed by a purchase of the full version of the Platform Builder Toolkit for \$995. This toolkit allows the OEM the ability to develop, test, and debug the image using the hardware emulator that is included in the toolkit. The toolkit also includes one product identification number (PID) for testing on hardware, but cannot be commercially shipped. In order to purchase runtime licenses the OEM must sign an agreement with Microsoft called the Microsoft OEM Customer License Agreement (CLA). Once in production a runtime license must be purchased for each individual device. Platform Builder Evaluation is for 180 days. The OEM must purchase full version in order to create commercial image.

No need to sign customer license agreement or purchase licensing until product is to be shipped.

License Agreement

- **Microsoft OEM Customer License Agreement (CLA)**
 - Required when OEM is ready to begin commercial shipment
 - Agreement between Microsoft and the OEM
 - Agreement Distributed by Microsoft Embedded Authorized Distributor
 - Not required for the purchase of Toolkit
 - Lasts for one year with and automatically renews for a second year

When the OEM is ready to purchase runtime licenses, the OEM should contact a Microsoft Embedded Authorized Distributor in their area and request the Microsoft OEM Customer License Agreement (CLA) and Additional Licensing Provisions (ALP). The OEM needs to sign and return the CLA to the Authorized Distributor as well as reviewing and complying with the ALP.

Developer Resources

- **Start Here** - <http://msdn2.microsoft.com/en-us/embedded/aa731145.aspx>

- Help System
- MSDN
- Code Included with Platform Builder
- Shared Source
- Community Source
- Newsgroups
- Additional Training for 3rd Parties, Colleges & Universities, Conferences
- Blogs, Chats, Webcasts
- Developer Interest and Special Interest Groups
- Online articles and news providers



Local Help is a great starting point when the troubled waters are encountered. Some users have found that targeted use of the search engine can decrease their time to problem resolution. As is normally the case with Visual Studio you can set where you search and can include online resources such as MSDN in your searches.

Platform Builder includes source code samples in the OS Design and OS directories that you can use for a variety of purposes. Sample code is provided for several types of applications as listed on the slide.

Note: In many cases, sample code is only a starting point for development. That is, some sample code is complete and ready to build, debug, and test in your OS Design; however, some samples are supplied as reference only and are incomplete. Sample code has not been tested and is not intended for production use.

A sample design template for network devices that connect to the Internet with a dial-up or broadband connection, called Gateway. Sample source code, including HTML files, for the gateway design is in the %_WINCEROOT%\Public\Servers\Oak\Gateway directory. For more information, see Developing a Gateway.

A sample Internet telephony design, called voice over IP (VoIP). Sample source code for the VoIP design is in the %_WINCEROOT%\Public\DirectX\Oak\VOIP directory. For more information, see Developing an IP Phone.

Sample code files for several drivers are in %_WINCEROOT%\Public\Common\Oak\Drivers\. The sample code in this directory is intended to be copied to your target configuration for further development.

Sample code for a variety of SOC (system-on-chip) drivers is in %_WINCEROOT%\Platform\Common\Src\SOC. Additional driver samples can be found in the %_WINCEROOT%\Platform directory.

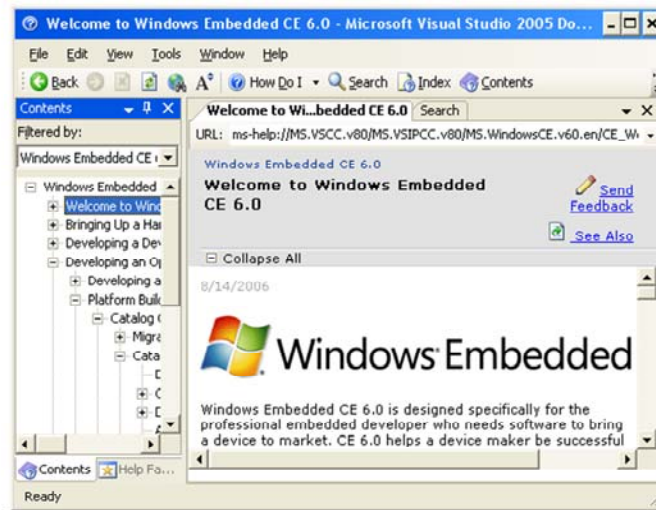
Sample code for a variety of applications is also available. For example, sample source for Bluetooth is available in the %_WINCEROOT%\Public\Common\Oak\Drivers\Bluetooth\Sample directory.

Here is the link to search for previous CE webcasts: <http://www.microsoft.com/events/AdvSearch.msp>
For some topics you might consider looking at V5 webcasts, the content for some topics, such as test can be fairly useful. Depending on the topic, some Windows Mobile content can be worth looking at as well.

Chats can be a place to toss a question out:

<http://www.microsoft.com/communities/chats/default.msp>

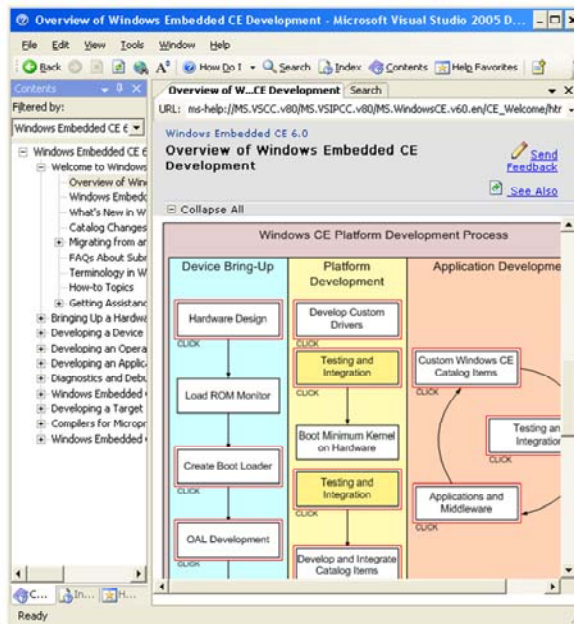
Developer Resources – Local Help



Local Help is a great starting point when the troubled waters are encountered. Some users have found that targeted use of the search engine can decrease their time to problem resolution. As is normally the case with Visual Studio you can set where you search and can include online resources such as MSDN in your searches.

Help System Layout – Resources for Developers

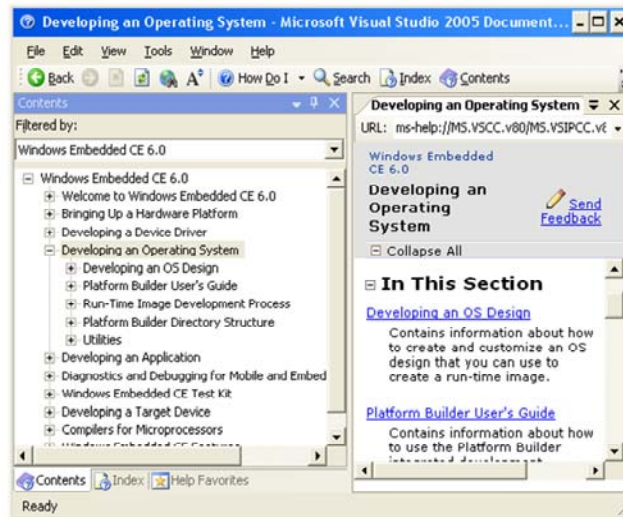
- One useful entry point into the help systems is the “Windows CE Platform Development Process” map. You can click on all of the items in the diagram.



Another place to start in help is the Windows Embedded CE OS architecture diagram.

Help System Layout – Resources for Developers

- This course covered material that is located in the “Developing an Operating System” section.



For those topics not covered in this course, the help systems is a great place to start. For example, there are additional useful utilities for such things as creating installation CAB files that are covered in the Utilities section of the help.

Developer Resources – Source Code

- **Platform Builder includes source code in each of the following directories:**
 - %_WINCEROOT%\Others
 - %_WINCEROOT%\Public
 - %_WINCEROOT%\Private
 - %_WINCEROOT%\Platform
- **100% Kernel**
- **FileSystem and Storage Manager**
- **SOAP and uPNP protocol implementations**

Platform Builder includes source code samples in the OS design and OS directories that you can use for a variety of purposes. Sample code is provided for several types of applications as listed on the slide.

Note: In many cases, sample code is only a starting point for development. That is, some sample code is complete and ready to build, debug, and test in your OS design; however, some samples are supplied as reference only and are incomplete. Sample code has not been tested and is not intended for production use.

A sample design template for network devices that connect to the Internet with a dial-up or broadband connection, called Gateway. Sample source code, including HTML files, for the gateway design is in the %_WINCEROOT%\Public\Servers\Oak\Gateway directory. For more information, see *Developing a Gateway*.

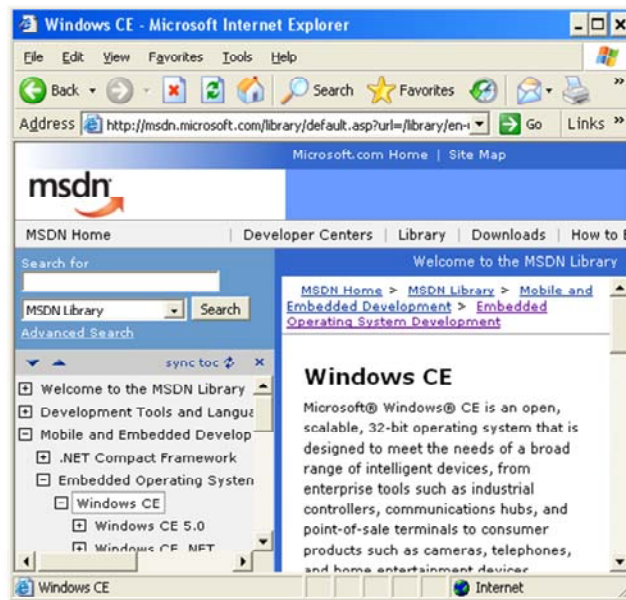
A sample Internet telephony design, called voice over IP (VoIP). Sample source code for the VoIP design is in the %_WINCEROOT%\Public\DirectX\Oak\VOIP directory. For more information, see *Developing an IP Phone*.

Sample code files for several drivers are in %_WINCEROOT%\Public\Common\Oak\Drivers\. The sample code in this directory is intended to be copied to your target configuration for further development.

Sample code for a variety of SOC (system-on-chip) drivers is in %_WINCEROOT%\Platform\Common\Src\SOC. Additional driver samples can be found in the %_WINCEROOT%\Platform directory.

Sample code for a variety of applications is also available. For example, sample source for Bluetooth is available in the %_WINCEROOT%\Public\Common\Oak\Drivers\Bluetooth\Sample directory.

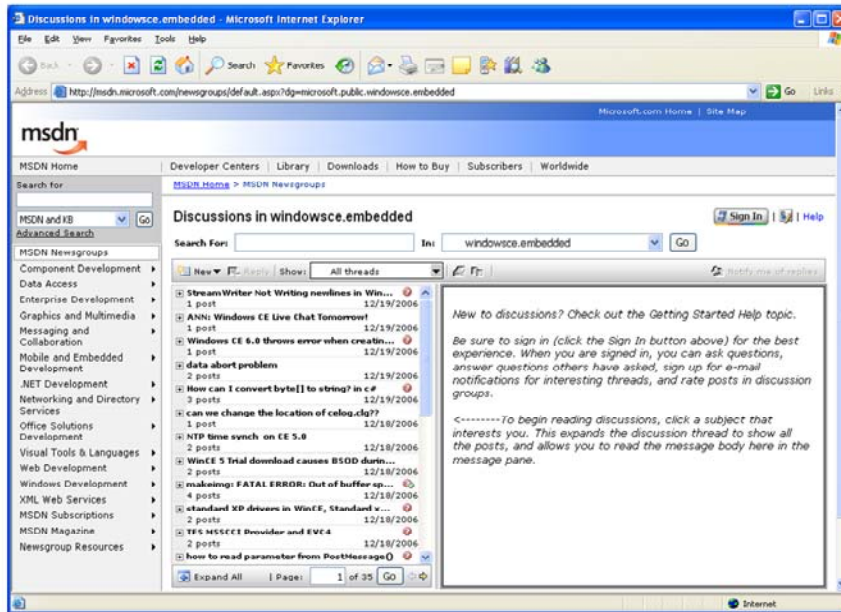
Developer Resources - MSDN



You can integrate MSDN into your help system or access it through an internet browser.

Using the advanced search options can be one way to locate information on MSDN.

Developer Resources – MSDN Newsgroups

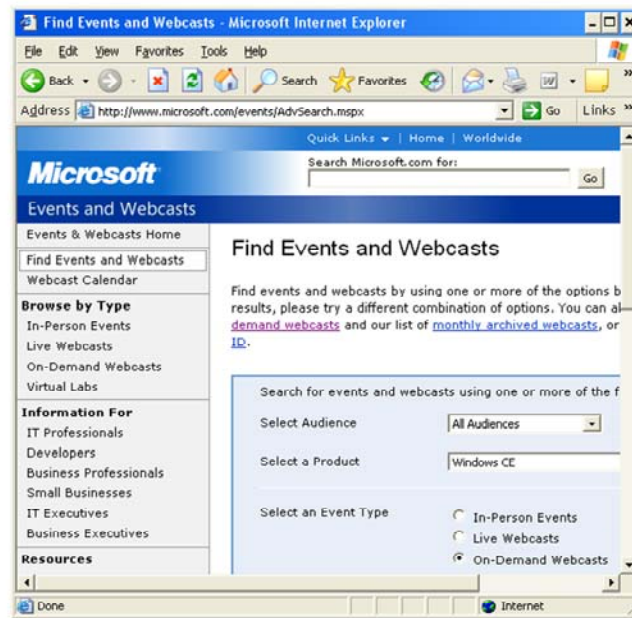


<http://msdn.microsoft.com/newsgroups/default.aspx?dg=microsoft.public.windowsce.embedded>

Developer Resources - Training

- **Formats Include: Classroom, Online, Webcasts**
- **Colleges and University**
- **Third party private partners provide classroom training on topics such as:**
 - Board Support Packages
 - Device Driver Development
 - Testing Best Practices
 - C++
 - .NET, Web Services, XML and other general application development topics

Developer Resources – Webcasts and Chats



Here is the link to search for previous CE webcasts: <http://www.microsoft.com/events/AdvSearch.aspx>
For some topics you might consider looking at V5 webcasts, the content for some topics, such as test can be fairly useful. Depending on the topic, some Windows Mobile content can be worth looking at as well.

Chats can be a place to toss a question out:

<http://www.microsoft.com/communities/chats/default.aspx>

WE-DIG



The screenshot shows the WE-DIG website interface. At the top left is the WE-DIG logo. The main header reads "Windows Embedded Developers Interest Group" with the tagline "We Dig .NET... Do you?". Below the header is a navigation bar with links for "We-Dig.org Home", "Forums", "About We-Dig.org", and "Register".

On the left side, there is a "Welcome" section with a login form containing fields for "Username/Email" and "Password", and a "Login" button. Below the login form is a "HelperWIZ" section with a "I want to..." dropdown menu containing "Register with us..." and "Login to We-Dig.org". At the bottom of this section are "Sponsor Links" for "MSDN", "Intel.org", "Intrixys", "OpenNETCF.org", and "Entrak.com".

The central part of the page features a large graphic with a yellow background and abstract colorful shapes. Overlaid on this graphic is the following XML code:

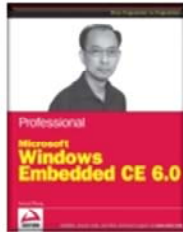
```
<CAN_YOU_DIG_IT?>  
<WE-DIG>  
<URL>http://:www.we-dig.org</URL>  
</WE-DIG>  
<CAN_YOU_DIG_IT?>
```

- www.we-dig.org

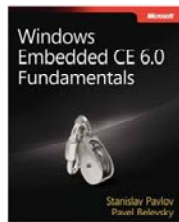
Developer Resources - Magazines

- **Windows for Devices**
<http://www.windowsfordevices.com>
- **Real Time Computing - RTC**
<http://www.rtcmagazine.com/>
- **Pocket PC Mag**
<http://www.pocketpcmag.com/>
- **Microsoft Mobiles**
<http://msmobiles.com/>

Developer Resources - Books



Professional Microsoft Windows Embedded CE 6.0
Samuel Phung

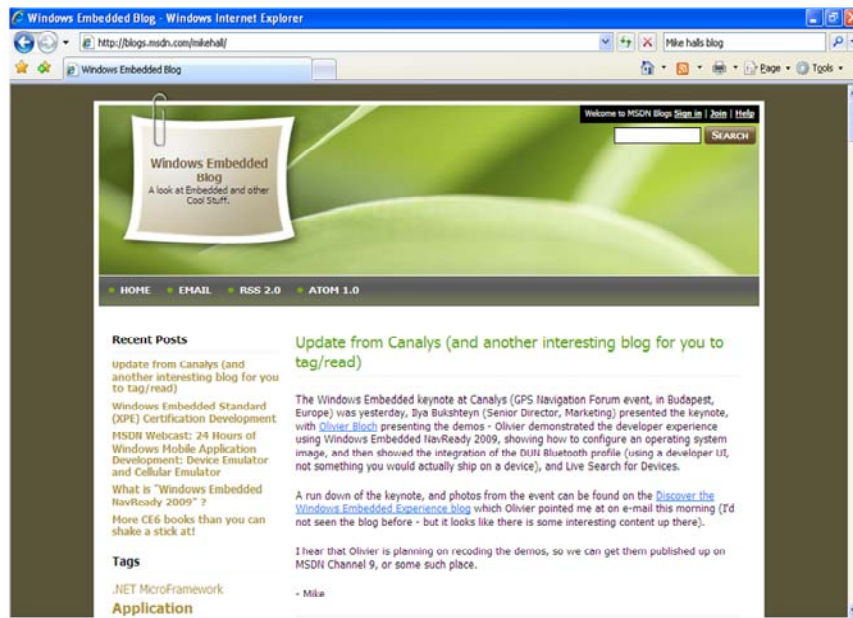


Windows Embedded CE 6.0 Fundamentals
Stanislav Pavlov and Pavel Belevsky



Programming Windows CE 6.0 Developer Reference
Doug Boling

Developer Resources - Blogs



Developer Resources - Blogs

- **CE Embedded Team**

http://blogs.msdn.com/ce_base/

- **Mike Hall**

<http://blogs.msdn.com/mikehall/>

- **Test Team**

<http://blogs.msdn.com/testembedded/>

- **Emulator**

<http://blogs.msdn.com/emulator/>

Developer Resources – Industry Events

- **Tech Ed**

www.microsoft.com/events/teched2006/worldwide.aspx

- **ESC – Microsoft Embedded Systems Conference**

- **WinHec**

www.microsoft.com/whdc/winhec/default.aspx

Building Solutions with Windows Embedded CE 6.0 R2

- Thanks for Attending!

3					8	7		
					9			4
1				6				2
		5		2	3		8	
	7				4		9	
	6				1	5		
8				5				6
7			1					
		9	8					5

Please take time to complete your evaluation and receive your class certificate.