Module 5: OEM Adaptation Layer
Instructor Notes

Presentation: 150 Minutes
Lab: 120 Minutes

In this module, the students learn about the functionality of the OEM Adaptation Layer (OAL).

After completing this module, students will be able to:

- Describe the relationship between the OEM Adaptation Layer (OAL) and kernel.
- Describe the operating system boot sequence.
- Describe the various aspects of developing an OAL.
- Identify the required OAL functions.
- Identify the optional OAL functions.
- Identify how to use GetTickCount and Sleep functions to test OAL.
- Describe how the Windows CE Kernel is built: with and without profiling.
- Implement the OAL registry functions.
- Implement basic Power Management.
- Implement custom certification to modify the OAL by using Signfile.exe to create new signatures and correctly implement your application.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

**Required Materials**
To teach this module, you need the following materials:

- Microsoft® PowerPoint® file 2535A_05.ppt

**Preparation Tasks**
To prepare for this module:

- Read all of the materials for this module.
- Complete the lab.
Module Strategy

Use the following strategy to present this module:

- **OAL Architecture**
  In this section, provide an overview of the OAL. Explain OAL architecture.

- **Operating System Boot Sequence**
  After learning about the OAL architecture, in this section, show the order in which functions of the OAL are called during boot. This will show the students all the functionality exposed by the OAL to the kernel and hardware.

- **Developing OAL**
  This section is used to show the students the work required to build an OAL. All the functions exposed by an OAL are represented here. This section is structured to follow the boot sequence.

- **Required OAL Functions**
  When implementing an OAL, there is a set of required functions. Introduce the required OAL functions in this section.

- **Optional OAL Functions**
  Some OAL functions are not required but may be necessary for your implementation. Introduce the optional OAL functions here.

- **Debugging an OAL**
  In this section, explain how to use `GetTickCount` and `Sleep` functions to test your OAL.

- **Building the Windows CE Kernel**
  In this section, explain how to build the CE kernel with and without profiling.

- **Implementing OAL Registry Functions**
  For many students, the concept of persistent registry on a Windows CE .NET device is of special interest. This section shows the OAL support used to implement persistent registry.

- **Power Management**
  A significant portion of Windows CE devices are battery powered. Hence, they require advanced power management to make efficient use of their battery life. There are controlling functions in the OAL used to suspend the CPU and initiate driver sleep.

- **Implementing Certification Model**
  Windows CE .NET has a class of devices that are used in the PC replacement market. In these public access devices, users can potentially install malicious applications. Use this section to show students how to build systems capable of blocking this type of software.
Overview

After completing this module, you will be able to:

- Describe the relationship between the OEM Adaptation Layer (OAL) and kernel.
- Describe the operating system boot sequence.
- Describe the various aspects involved in developing an OAL.
- Identify the required OAL functions.
- Identify the optional OAL functions.
- Identify how to use `GetTickCount` and `Sleep` functions to test OAL.
- Describe how the Windows CE Kernel is built: with and without profiling.
- Implement the OAL registry functions.
- Implement basic Power Management.
- Implement custom certification to modify the OAL by using Signfile.exe to create new signatures and correctly implement your application.
The illustration demonstrated on the slide shows how OAL (OEM-supplied components) interact with the kernel (Microsoft-supplied components). The OAL acts as an interface between the kernel and the hardware platform in the following ways:

- When the kernel needs to modify the real-time clock (RTC), it calls one of the OAL RTC functions. This OAL function modifies the RTC.
- When the Windows CE operating system goes into a low-power mode, the kernel calls one of the OAL power management functions. This OAL function accesses the hardware to handle the low-power mode.
- When a hardware device raises an interrupt, the kernel handles it and calls one of the OAL interrupt-related functions.
- When the kernel needs to access the debug ports (Ethernet, parallel, or serial), it calls one of the OAL debug functions. This OAL function accesses the debug port.
Before implementing the initialization functions, you need to understand the Windows CE boot sequence.

1. **Boot Vector jumps to StartUp**
   The first instruction executed by the CPU (boot vector) is to jump to hardware initialization code — the **StartUp** function. This function must be implemented in the OAL. At this stage, there is no Windows CE operating system running.

2. **StartUp jumps to KernelStart**
   Once completed, **StartUp** jumps to **KernelStart (KernelInitialize for x86)**, a function that the CPU-specific kernel exports. The kernel performs its initializations through **KernelStart**, and then calls the **OEMInitDebugSerial** function.

   **Note** On x86 platforms, the **KernelStart** function is called **KernelInitialize**.

3. **Kernel calls OEMInitDebugSerial**
   The **OEMInitDebugSerial** function is implemented in the OAL by the OEM. It initializes the debug serial port on the target platform. On completion, this function returns to the kernel, which then calls the **OEMInit** function.

4. **Kernel calls OEMInit**
   The **OEMInit** function is implemented in the OAL by the OEM. It initializes any additional hardware, sets up interrupts, installs timers, and sets kernel global variables.

   **Note** The example above shows **OEMInit** calling several functions. These are only examples. These functions will vary depending on your platform.

5. **Kernel calls OEMGetExtensionDRAM**
After calling OEMInit, the kernel calls the OEMGetExtensionDRAM function to determine if a non-contiguous bank of DRAM is available on the target platform. This non-contiguous bank of DRAM is not defined in the CONFIG.BIB file.

Now the kernel is ready to run. It then loads modules, such as Filesys, GWES, and device (depending on your configuration).
Developing OAL

**Topic Objective**
To describe the various aspects in developing an OAL.

**Lead-in**
In this section, you will learn about the various aspects in developing an OAL.

There are different aspects involved in developing OAL:

- Required OAL Functions
- Optional OAL Functions
- Debugging an OAL
- Building the Windows CE Kernel
Required OAL Functions

In this section, you will learn about the functions that are essential in OAL:

- **Startup**
- **Debug Serial**
- **OEMInit**
- **OEMInit : An Example**
- **System Timer**
- **System Timer : SC_GetTickCount**
- **Interrupt Processing**
- **Kernel Input/Output**
- **Example: Custom Kernel IOCTL**
- **KITL**
StartUp

Topic Objective
To describe the implementation of StartUp.

Lead-in
In this section, you will learn how to implement StartUp.

The StartUp function is the first OEM function to implement. It is also the first function called when the target device boots. The purpose of the StartUp function is:

- To initialize the CPU hardware to a known state.
- To call the kernel initialization function.

The kernel initialization function is named KernelInitialize for x86-based platforms and KernelStart on all other platforms. The kernel initialization is the first non-OEM function that is called after StartUp has completed its tasks.

The StartUp function will contain kernel initialization code that is specific to each class of CPU. The details on how to properly initialize the target hardware are found in their respective hardware platform manual. The platform building online docs also contains useful kernel initialization information for each class of CPU (MIPS, ARM, x86, SHx).

On x86 and ARM processors, you must create an OEMAddressTable and pass it to the kernel. Each entry in the table specifies a physical location in memory, the size of the memory, and the static virtual memory address to which to map it. The static virtual address is specified in the cached memory range and the kernel can then create the uncached address that points to the same physical address. On MIPS and SHx, this mapping is defined by the CPU and kernel.

Here is an example of how to setup the kernel for an ARM based CPU:

For code details, see
WINCE400\PLATFORM\SA11X0BD\KERNEL\HAL\ARM\Fwsarm.s

1. Put the processor in a supervisor mode.
ldr r0, =({Mode_SVC :OR: NoIntsMask}) msr cpsr_c, r0

2. Disable the interrupt request (IRQ) and FIQ inputs at the CPU.
3. Disable the MMU and both the instruction and data caches.
   \texttt{mcr p15,0,r0,c1,c0,0 \text{; Turn Off MMU, I&D Caches, WB}}

4. Flush/invalidate the instruction and data caches and the Translation Lookaside Buffer (TLB) and drain the write buffers.
   \texttt{mcr p15, 0, r0, c8, c7, 0 \text{; flush (invalidate) I/D tlb's}}
   \texttt{mcr p15, 0, r0, c7, c7, 0 \text{; flush (invalidate) I/D caches}}

5. Determine the reason for being in the startup code: such as cold reset, watchdog reset, GPIO reset, or sleep reset. Reasons for reset are generally:
   - Hard Reset
   - Wake from Sleep
   - GPIO Reset
   - Watchdog Reset
   - Eboot Handoff

6. Configure the GPIO lines per the requirements of the board. GPIO lines have to be enabled for on-board features like LED.
7. Configure the memory controller, which will set the refresh frequency and enable clocks — program data width and memory timing values and power up the banks
8. Configure the interrupt controller, which will mask and clear any pending interrupts.
9. Initialize the RTC count register to 0 and enable the RTC.
10. Set up the power management/monitoring registers, which sets conditions during sleep modes.
11. Turn on all board-level clocks and on-chip peripheral clocks.
12. Get the physical base address of \texttt{OEMAddressTable} and store in r0.
   \texttt{mov r0, pc}
   \texttt{add r0, pc, #OEMAddressTable-(.+8) \text{; (r0) = OEMAddressTable phys addr}}

13. Jump to \texttt{KernelStart}.
    \texttt{bl KernelStart}
    ;
    ; KernelStart should never return:
    ;
Debug Serial

**Topic Objective**
To describe how to implement debug serial.

**Lead-in**
In this section, you will learn how to implement the debug serial code.

- **OEMInitDebugSerial()**
  - Configures Speed, Parity, Stop bit length

- **OEMReadDebugByte()**
  - Retrieves a byte from the debug monitor port

- **OEMWriteDebugByte()**
  - Outputs a byte to the debug monitor port

- **OEMWriteDebugString()**
  - Writes a string to the debug monitor port

Debug serial is one of the first things to get working in the OAL. This gives the developer a method of debugging the OAL. If your platform only has one debug serial port, that port will not be available for retail use.

**Note**
Examples below are from WINCE400\PLATFORM\CEPC\KERNEL\HAL\X86\DEBUG.C

**OEMInitDebugSerial**
You implement this function of your serial debug port to configure: Speed, Parity, Stop bit length
Sample Code

void OEMInitDebugSerial(void)
{
    WRITE_PORT_UCHAR(IoPortBase+comLineControl, 0x80);   // Access Baud Divisor
    WRITE_PORT_UCHAR(IoPortBase+comDivisorLow, 0x03);    // 38400
    WRITE_PORT_UCHAR(IoPortBase+comDivisorHigh, 0x00);
    WRITE_PORT_UCHAR(IoPortBase+comFIFOControl, 0x01);   // Enable FIFO if present
    WRITE_PORT_UCHAR(IoPortBase+comLineControl, 0x03);   // 8 bit, no parity
    WRITE_PORT_UCHAR(IoPortBase+comIntEnable, 0x00);     // No interrupts, polled
    WRITE_PORT_UCHAR(IoPortBase+comModemControl, 0x03);  // Assert DTR, RTS
}

OEMReadDebugByte

This function retrieves a byte from the debug monitor port. The function returns
the data byte if available or it returns OEM_DEBUG_COM_ERROR if an error
is detected; or it returns OEM_DEBUG_READ_NODATA if no data is
available at the port.

Sample Code

int OEMReadDebugByte(void)
{
    unsigned char   ucStatus;
    unsigned char   ucChar;
    if ( IoPortBase ) {
        ucStatus = READ_PORT_UCHAR(IoPortBase+comLineStatus);
        if ( ucStatus & LS_RX_DATA_READY ) {
            ucChar = READ_PORT_UCHAR(IoPortBase+comRxBuffer);
            if ( ucStatus & LS_RX_ERRORS ) {
                return (OEM_DEBUG_COM_ERROR);
            } else {
                return (ucChar);
            }
        }
    }
    return (OEM_DEBUG_READ_NODATA);
}

OEMWriteDebugByte

This function outputs a byte to the debug monitor port.
Sample Code

```c
void OEMWriteDebugByte(BYTE ucChar)
{
    if ( IoPortBase ) {
        while ( !(READ_PORT_UCHAR(IoPortBase+comLineStatus) & LS_THR_EMPTY) ) {
            ;
        }
        WRITE_PORT_UCHAR(IoPortBase+comTxBuffer, ucChar);
    }
}
```

**OEMWriteDebugString**

This function writes a string to the debug monitor port.

Sample Code

```c
void OEMWriteDebugString(unsigned short *str)
{
    while ( *str ) {
        while ( !(READ_PORT_UCHAR(IoPortBase+comLineStatus) & LS_THR_EMPTY) ) {
            ;
        }
        WRITE_PORT_UCHAR(IoPortBase+comTxBuffer, (UCHAR)*str++);
    }
}
```
OEMInit

**Topic Objective**
To describe how to implement OEMInit.

**Lead-in**
In this section, you will learn how to implement OEMInit.

- **Required task is to set up hardware and register interrupt for the system tick**
- **ISRs and HookInterrupt**
- **Optional tasks to enhance system capabilities include:**
  - Setting up performance counters
  - Enabling debug IO control
  - System Halt
  - Default thread quantum
  - Zeroing memory
- **Setting Up the Interrupt Map**
  - Define a mapping of Interrupt IDs
  - Interrupt IDs are returned by the ISRs to the kernel and are used to link an incoming IRQ with a software IST

The kernel initialization function calls **OEMInit**. The minimum task of **OEMInit** is to set up remaining hardware not initialized in StartUp and to register the interrupt for the system tick. Then, once that is done you can add additional code to initialize both optional function pointers and optional variables to enhance the system capabilities. Also, within the OAL you set up your Interrupt Service Routines. From **OEMInit**, you then set the interrupt request (IRQ) to interrupt service routine (ISR) mappings using a function called **HookInterrupt**.

**Note** For more information about ISRs and **HookInterrupt**, see Module 6, “Device Driver Architecture,” in Course 2535A, *Developing a Board Support Package for Microsoft Windows CE .NET*.

**Optional Tasks to Enhance System Capabilities**
- Logging functions.
- Registry functions.
- Secure loader functions.
- Save and restore co-processor registers.
- Setting up performance counters.
- System halt.
- Event tracking.
- Detection of idle time.
- Stack scavenging.
- User notification alarm.
- Multiple XIP regions.
- Default thread quantum.
• Enable debug I/O control.
• Erasing the object store.
• Supporting CPU utilization functionality.
• Object store cold boot.
• ARM FPU support.
• Zeroing memory.

Setting Up the Interrupt Map
In the OEMInit function, you define a mapping of Interrupt IDs. Interrupt IDs’ are returned by the ISRs to the kernel and are used to link an incoming IRQ with a software IST. An interrupt identifier is a unique value used by the kernel to identify a target device that raises an interrupt that requires processing. The kernel then uses the interrupt identifier to indicate whether all handling is complete, or whether to launch an IST that handles further processing by the device driver. Platform Builder provides a set of predefined interrupt identifiers, or you can create your own. Windows CE defines a set of interrupt identifiers in the Nkintr.h file. These values can be identified by the define SYSINTR_XXX.
OEMInit: An Example

Topic Objective
To describe how to implement OEMInit function.

Lead-in
In this section, you will learn how to implement OEMInit function.

```c
Void OEMInit() {
    SetUpInterruptMap();
    PCIInitBusInfo();
    InitDebugEther();
    OEMParallelPortInit();
    InitPICs();
    InitClock();
    if (MainMemoryEndAddress == CEPC_EXTRA_RAM_START)
    {
        MainMemoryEndAddress += IsDRAM(MainMemoryEndAddress,
            CEPC_EXTRA_RAM_SIZE);
    }
    pKDIoControl = OEMKDIoControl;
}
```

The sample code illustrated on the slide is a truncated example taken from `\WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\cfwpc.c`. This is the OEMInit function for the CEPC platform.

The first function called is `SetUpInterruptMap`. This function initializes and sets up the global structures that keep track of the System Interrupt to IRQ mapping and IRQ to System Interrupt mapping.

**Note** A define SETUP_INTERRUPT_MAP macro has been utilized to make the code easier to implement and read.

Next, `PCIInitBusInfo` is called. This function registers the KITL/download device in the registry and enumerates the PCI busses.

**Note** To see the details of the code, see `\WINCE400\PLATFORM\CEPC\KERNEL\HAL\PCI.C`

Next, `InitDebugEther` is called. This function initializes debug Ethernet hardware and configures kernel services over Ethernet (debug messages, CESH, kernel debugger). Also, `KitInit` function must be called before any debug services are redirected to KITL. The `KitInit` function initializes the KITL subsystem.

Next, `OEMParallelPortInit` is called. This function initializes the parallel port. This is done so that in case you want to download binary files from the development workstation to the target device by using the parallel port.

Next, `InitPICs` is called. This function initializes the programmable interrupt controller. In addition, it also sets up the interrupt handler via a call to `HookInterrupt`. The `HookInterrupt` function registers an ISR with the kernel specifying a particular hardware interrupt indicated by its IRQ line value. In a multi interrupt line CPU, this functionality would be replaced by a simple table of `HookInterrupt` calls.

Delivery Tip
Instructor should review all the sample code files referred in this section. The samples are somewhat commented.

Delivery Tip
Emphasize the development of KITL.

Delivery Tip
Point out the difference between single IRQ and multiple IRQ CPUs.
Next, **InitClock** is called. This function is located in timer.c. First, this function sets the global variable idleconv to 1. This is a translation constant in one-millisecond units and is used to support the **GetIdleTime** function. Next, the system interrupt timer is set up. In this example, timer 0 is set up to generate an interrupt every 1ms (SYSTEM TICK). Timer 0 is also set up for kernel profiling use. Then, lastly the real time clock is initialized.

Next, extra DRAM is detected by a call to the **IsDRAM** function. The **IsDRAM** function basically walks through memory performing a write-read test until an invalid boundary is found. Once the extra DRAM is detected the MainMemoryEndAddress variable is adjusted to extend the size of the contiguous RAM section specified in the Config.bib file.

The **IsDRAM** function is present in WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\cfwpc.c

**OEMKDIoControl** is called by the OEM to support requests from the kernel debugger. You must then implement the input output controls (IOCTLs) that go along with this function.
void SetUpInterruptMap() {
    memset(g_Sysintr2IntrMap, 0xFF, sizeof(g_Sysintr2IntrMap));
    memset(g_Intr2SysintrMap, 0xFF, sizeof(g_Intr2SysintrMap));
    #define SETUP_INTERRUPT_MAP(SysIntr, Irq) \
        { \ 
        DEBUGCHK(g_Sysintr2IntrMap[SysIntr] == -1); \ 
        DEBUGCHK(g_Intr2SysintrMap[Irq] == -1); \ 
        g_Sysintr2IntrMap[SysIntr] = Irq; \ 
        g_Intr2SysintrMap[Irq] = SysIntr; \ 
    }

    // IRQ0 is timer0, the scheduler tick
    SETUP_INTERRUPT_MAP(SYSINTR_KEYBOARD, 1);
    // IRQ2 is the cascade interrupt for the second PIC
    // Serial is particularly confusing, so here's a summary
    // Debugger - 3F8, no interrupts used
    // COM1 - 2F8, interrupt 3
    // COM2 - 3E8, interrupt 4
    // COM3 - 2E8, interrupt 5
    // This would all be much easier if we moved the debugger
    // to COM4, and left CEPC's COM1-3 at the same IOB/IRQ as
    // the PC's COM1-3. The only reason we leave it is legacy.
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+3,  3);
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+4, 4);
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+5, 5);
    // IRQ6 is normally the floppy controller.
    // IRQ7 is LPT1, but ppsh doesn't use interrupts.
    // IRQ8 is the real time clock.
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+9,  9);
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+10, 10);
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+11, 11);
    SETUP_INTERRUPT_MAP(SYSINTR_TOUCH,       12);
    // IRQ13 is normally the coprocessor
    // IRQ14 is normally the hard disk controller.
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+14, 14);
    SETUP_INTERRUPT_MAP(SYSINTR_FIRMWARE+15, 15);
    #undef SETUP_INTERRUPT_MAP
}

Next, PCIInitBusInfo is called. This function registers the KITL/download device in the registry and enumerates the PCI busses. This function is too long to list here. To see the details of the code, see \WINCE400\PLATFORM\CEPC\KERNEL\HAL\PCI.C.

Next, InitDebugEther is called. This function initializes debug Ethernet hardware and configure kernel services over Ethernet (debug messages, CESH, kernel debugger). Also KitlInit function must be called before any debug services are redirected to KITL. The KitlInit function initializes the KITL subsystem. The InitDebugEther function is shown below:
void InitDebugEther (void)
{
    BOOT_ARGS *pBootArgs = (BOOT_ARGS *) ((ULONG)(*(PBYTE *(BOOT_ARG_PTR_LOCATION)) | 0x80000000));

    if ((pBootArgs->KitlTransport & ~KTS_PASSIVE_MODE) == KTS_NONE)
        return;

    // Initialize KITL transport
    if (KitlInit(!(pBootArgs->KitlTransport & KTS_PASSIVE_MODE))) {
        KITLOutputDebugString ("KITL Initialized\n");
        // no longer need to start kernel services
        // since KITL config message told us what to start and
        // kitl will start it accordingly
    } else {
        KITLOutputDebugString ("KITL Initialization Failed, No
debugging support available\n");
    }
}

Next, **OEMParallelPortInit** is called. This function initializes the parallel port. This is done so that in case you want to download binary files from the development workstation to the target device by using the parallel port. **OEMParallelPortInit** function is shown below:

```c
int OEMParallelPortInit(void)
{
    WRITE_PORT_UCHAR(
        IoPortBase + PAR_PORT_CTRL, PAR_CTRL_AUTOFEED | 
        PAR_CTRL_STROBE);
    return (TRUE);
}
```

Next, **InitPICs** is called. This function initializes the programmable interrupt controller. In addition, it also sets up the interrupt handler via a call to **HookInterrupt**. The **HookInterrupt** function registers an ISR with the kernel specifying a particular hardware interrupt indicated by its IRQ line value. The **InitPICs** function is shown below:
void InitPICs(void) {
    int i;
    __asm {
    ; First do PIC 1
    mov    al, 011h        ; Init command 1, cascade & 4th init byte
    out    020h, al
    jmp    short $+2
    mov    al, 040h        ; Init command 2, vector
    interrupts to 64
    out    021h, al
    jmp    short $+2
    mov    al, 004h        ; Init command 3, slave on IRQ 2
    out    021h, al
    jmp    short $+2
    mov    al, 001h        ; Init command 4, normal EOI
    out    021h, al
    jmp    short $+2
    mov    al, 00Bh        ; Select In Service Register for reads
    out    020h, al
    mov    al, 0FFh        ; Start with all interrupts disabled
    out    021h, al
    ; Now do PIC 2
    ;
    mov    al, 011h        ; Init command 1, cascade & 4th init byte
    out    0A0h, al
    jmp    short $+2
    mov    al, 048h        ; Init command 2, vector
    interrupts to 40
    out    0A1h, al
    jmp    short $+2
    mov    al, 002h        ; Init command 3, slave on IRQ 2
    out    0A1h, al
    jmp    short $+2
    mov    al, 001h        ; Init command 4, normal EOI
    out    0A1h, al
    mov    al, 00Bh        ; Select In Service Register for reads
    out    0A0h, al
}
mov     al, 0FFh        ; Start with all interrupts
disabled
out     0A1h, al
}
/* Setup the PeRP interrupt handler and enable the PeRP
interrupt in the BasePSR */
for (i = 64; i < 80; i++)
    HookInterrupt(i, (void *)PeRPISR);
    //
    // Enable interrupts from cascaded PIC
    //
    PICEnableInterrupt(INTR_PIC2, TRUE);
}

Next, InitClock is called. This function sets the global variable idleconv to one.
This is a translation constant in one-millisecond units and is used to support
GetIdleTime function. Next, the system interrupt timer is set up. In this
example, timer 0 is set up to generate an interrupt every 1ms (SYSTEM TICK).
Timer 0 is also set up for kernel profiling use. Then, lastly the real time clock is
initialized. The InitClock function is given below:
void
InitClock(void)
{
    BYTE cData;
    //
    // Set up translation constant for GetIdleTime() (1 ms
    // units).
    // Note: Since curridlehigh, curridlelow is counting in ms,
    // and GetIdleTime() 
    // reports in ms, the conversion ratio is one. If
    // curridlehigh, curridlelow
    // were using other units (like ticks), then the conversion
    // would be calculated
    // from the clock frequency.
    //
    idleconv = 1;
    //
    // Setup Timer0 to fire every TICK_RATE mS and generate
    // interrupt
    //
    __asm {

    // configure counter for correct mode
    mov    al, 00110100b          ; counter 0, 16-bit,
    mode 2, binary
    out    043h, al
    jmp    short $+2

    // load the timer with correct count value
    mov    ax, TIMER_COUNT       ; Divisor for SYSTEM
    TICK
    out    040h, al
    jmp    short $+2
    mov    al,ah
    out    040h, al

    } PICEnableInterrupt(INTR_TIMER0, TRUE);

dwReschedPeriod = TIMER_COUNT;
    //
    // Set up Timer2 to use its full range for kcall profiling
    //
    __asm {

    // configure counter for correct mode
    mov    al, 10110100b        ; counter 2, 16-bit,
    mode 2, binary
    out    043h, al
    jmp    short $+2

    // Start counter at highest value, it's a countdown.
    // It's confusing, but 0 is largest initial value. Read
    // the manual.
    xor    eax, eax          ; 0x00
    out    042h, al
    jmp    short $+2
    out    042h, al

    // Enable bit 0 of Port B to start counter2 counting
    in     al, 61h          ;Read Current value of Port B
or al, 00000001b ;Set bit 0 to enable gate
out 61h, al ;Store new value to Port B
}

do {
    cData = CMOS_Read( RTC_STATUS_A);
} while ( cData & RTC_SRA_UIP );
    cData = CMOS_Read( RTC_STATUS_B );
    CMOS_Write( RTC_STATUS_B, (BYTE)(cData|RTC_SR8_24HR) );
    cData = CMOS_Read( (BYTE)(RTC_STATUS_B) );
    RETAILMSG(1, (TEXT("RTC - Status Reg B - 0x%2.2X\r\n"),
    cData));
    PICEnableInterrupt(INTR_RTC, TRUE);
}

Next, extra DRAM is detected by a call to the IsDRAM function. The IsDRAM function basically walks through memory performing a write read test until a invalid boundary is found. Once the extram DRAM is detected the MainMemoryEndAddress variable is adjusted to extend the size of the contiguous RAM section specified in the Config.bib file. The IsDRAM function is located at WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\cfwpc.c

Finally, the function pointer pKDIoControl is set up to point to the function OEMKDIoControl that is implemented by the OEM to support requests from the kernel debugger.
System Timer

**Topic Objective**
To describe the implementation of the system timer.

**Lead-in**
In this section, you will learn how to implement the system timer.

- Make sure the system timer interrupt is registered with the ISR
- Program the system timer to generate an interrupt every 1ms.
- In the system timer ISR, update the global system tick counter CurMSec and CurTicks. If doing interrupt timing return SYSINTR_TIMING, else return SYSINTR_RESCHED.
- DiffMSec is no longer supported in Windows CE .NET

The system timer generates system ticks at a fixed rate of one tick per millisecond, which is the rate at which a timer interrupt is generated and serviced by the operating system. Implementing the system timer functions is one of first task to complete in the OAL.

**Example: Implementing the System Timer on an ARM-Based Platform (SA11X0BD)**

For the ARM based platform, the interrupts are handled by OEMInterruptHandler function. The setting up of this interrupt is already done in kernel. All you need to do is implement OEMInterruptHandler. For x86 platform, you will need to manually register the timer interrupt with it’s ISR in OEMInit to ensure the system timer is updated appropriately.

The following defines specifics for this platform. The goal is to set up the system timer to generate an interrupt every 1MS. That is done in OEMInit via the local function CPUSetSysTimerCount (see below). This programs the timer to fire off every RESCHED_PERIOD which is 1ms.

```c
// Platform specific to SA11X0BD
#define OEM_CLOCK_FREQ 3686400
// Reschedule period in ms
#define RESCHED_PERIOD 1
// Timer count value for reschudeler period (3686 for 1 ms system tick)
#define RESCHED_INCREMENT ((RESCHED_PERIOD * OEM_CLOCK_FREQ) / 1000)
```

The following is the OEMInit function. This is where the timer is programmed.
Void OEMInit()
{
    /* initialize and start scheduler timer */
    dwReschedIncrement = RESCHED_INCREMENT;
    OEMCount1ms = RESCHED_INCREMENT;
    OEMClockFreq = OEM_CLOCK_FREQ / TICK_PRESCALER;
    CPUSetSysTimerCount(RESCHED_PERIOD); // 1ms
}

This is the **CPUSetSysTimerCount** function:

Void CPUSetSysTimerCount(
    DWORD dwCountdownMSec
)
{
    DWORD dwCount, dwMatch;
    volatile struct ostreg *const v_pOSTReg = (struct ostreg *)OST_BASE_VIRTUAL;

    dwCurReschedIncr = dwCountdownMSec * OEMCount1ms;
    dwCount = v_pOSTReg->oscr;
    dwMatch = dwCount + dwCurReschedIncr;
    v_pOSTReg->osmr[0] = dwMatch; // Match register
}

Next, in the interrupt handler (shown below), the first thing is to check if the interrupt is coming from the system timer by reading the interrupt pending register. If it is from the system timer then clear the interrupt and increment the timer global variables, CurMSec and CurTicks.

**Note**  In Windows CE .NET, DiffMSec is no longer used.

The next thing to check is if you are doing interrupt timing. If true, then process it accordingly. If not, then return SYSTINTR_RESCHED if it is time to reschedule.

The following is a snippet of the interrupt handler for the SA11X0BD platform:
int OEMInterruptHandler(unsigned int ra) {
...

    //Read the interrupt pending register in the SA1100
    my_sa1100_intreg = v_pICReg->icip;
    //Service the timer interrupt. Note that the interrupt
    interval is hardwired at 1 mSec. (3686.4)
    if (my_sa1100_intreg.osmr0) {
        v_pOSTReg->osmr[0] += dwReschedIncrement;

        if ( ((long)(v_pOSTReg->osmr[0] - v_pOSTReg->oscr)) <
            NO_RESCHED_WINDOW ) {
            v_pOSTReg->osmr[0] = v_pOSTReg->oscr +
            dwReschedIncrement;
        }

        //Clear the interrupt
        TIMER_M0_INT_CLR(1);

        //Increment the global variable
        #if (CE_MAJOR_VER == 0x0003)
            DiffMSec += TIMER_PERIOD;
        #endif
        CurMSec += TIMER_PERIOD;
        CurTicks.QuadPart += dwReschedIncrement;

        if (fIntrTime) {
            // We're doing interrupt timing. Every other tick is a
            // RESCHED.
            //
            dwIntrTimeCountdown--;
            if (dwIntrTimeCountdown == 0) {
                dwIntrTimeCountdown = dwIntrTimeCountdownRef;
                wNumInterrupts = 0;
                dwIsrTime2 = PerfCountSinceTick();
                return (SYSINTR_TIMING);
            } else {
                #if (CE_MAJOR_VER == 0x0003)
                    if (ticksleft || (dwSleepMin && (dwSleepMin <=
                        DiffMSec)) || (dwPreempt && (dwPreempt <= DiffMSec)) )
                        #else
                            if ((int) (CurMSec -
                                dwReschedTime) >= 0)
                        #endif
                    return SYSINTR_RESCHED;
                } else { // Not interrupt timing
                    #if (CE_MAJOR_VER == 0x0003)
                        if (ticksleft || (dwSleepMin && (DiffMSec >=
                            dwSleepMin)) || (dwPreempt && (DiffMSec >= dwPreempt)) )
                            #else
                                if ((int) (CurMSec - dwReschedTime) >= 0)
                            #endif
                        return SYSINTR_RESCHED;
                    return SYSINTR_NOP;
                }
The OAL system-timer function, **SC_GetTickCount** returns the current number of milliseconds since boot. Windows CE calls **SC_GetTickCount** in your OAL when the Win32 function **GetTickCount** is called. You must register the system-timer interrupts in **OEMInit** to ensure the system timer is updated appropriately.

For x86 platform, **CPUGetSysTimerCountElapsed** function returns a 0 because the system tick is fixed. So for x86 you would use the following **CPUGetSysTimerCountElapsed** function:

```c
DWORD CPUGetSysTimerCountElapsed(  
    DWORD dwTimerCountdownMSec,  
    volatile DWORD *pCurMSec,  
    volatile DWORD *pDiffMSec,  
    DWORD *pPartialCurMSec,  
    DWORD *pPartialDiffMSec,  
    volatile ULARGE_INTEGER *pCurTicks  
)  
{  
    // Using fixed tick, so do nothing and return 0 time passed  
    return 0;  
}
```

For the ARM platform (SA110XBD), the timers are programmable so you would use the following **CPUGetSysTimerCountElapsed** function.
DWORD
CPUGetSysTimerCountElapsed(
    DWORD dwTimerCountdownMSec,
    volatile DWORD *pCurMSec,
    DWORD *pPartialCurMSec,
    volatile U_LARGE_INTEGER *pCurTicks
)
{
    volatile struct ostreg *const v_pOSTReg = (struct ostreg *)OST_BASE_VIRTUAL;

    DWORD dwTick = dwTimerCountdownMSec * OEMCount1ms;
    DWORD dwCount = (DWORD)v_pOSTReg->osmr[0] - (DWORD)v_pOSTReg->oscr;

    // Note: if dwCount is negative, the counter went past the
    // match point. The math // still works since it accounts for the dwTick time plus
    // the time past the match.
    dwCount = dwTick - dwCount;

    pCurTicks->QuadPart += dwCount;

    dwCount += *pPartialCurMSec;
    *pPartialCurMSec = dwCount % OEMCount1ms;
    dwCount /= OEMCount1ms;
    *pCurMSec += dwCount;

    return dwCount;
}
Interrupt Processing

**Topic Objective**
To describe the interrupt processing routines.

**Lead-in**
In this section, you will learn how to implement the interrupt processing routines.

Interrupt processing functions allow the kernel to begin, service, and complete interrupt processing.

When a device driver calls the InterruptInitialize, InterruptDisable, and InterruptDone functions to enable or disable interrupts, or to signal the kernel that processing is complete for an interrupt, then the kernel translates these calls into the OEMInterruptEnable, OEMInterruptDisable, and OEMInterruptDone functions, respectively.

**Note** The interrupt processing functions call OEMTranslateSysIntr. That function translates the system interrupt number to the actual hardware interrupt number.

**Note** The sample code below came from \WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\cfwpc.c

---

### OEMInterruptEnable

This function performs any hardware operations necessary to allow a device to generate the specified interrupt. This can include setting a hardware priority for the device, setting a hardware interrupt enable port, and clearing any pending interrupt conditions from the device. These actions can be split between the device driver and the code in OEMInterruptEnable. In general, actions that involve shared state among multiple devices should be managed by the OAL functions, and actions that involve private state should be managed by the device driver.

When a device driver calls the InterruptInitialize kernel routine, the kernel in turn calls OEMInterruptEnable. The system cannot be preempted when this function is called.
The **OEMInterruptEnable** function should also return success for SYSINTR_TIMING without actually doing anything. This is to support interrupt latency timing.

The **OEMInterruptEnable** function should also return success for SYSINTR_VMINI. This supports the ability to access a remote network while maintaining the capability to debug your target device.

### Sample Code

```c
BOOL OEMInterruptEnable (  
    DWORD idInt,       // @parm Interrupt ID to be enabled. See  
    // <l Interrupt ID's.Interrupt ID's>
    // for a list of possible values.
    LPVOID pvData,     // @parm ptr to data passed in in the <f InterruptInitialize> call
    DWORD cbData       // @parm Size of data pointed to be <p pvData>
)
{
    DWORD Interrupt;
    if (idInt == SYSINTR_TIMING) {
        return (TRUE);
    }
    if (idInt == SYSINTR_VMINI) {
        DEBUGMSG (1, (TEXT("Accepting VMini interrupt enable request.\r\n")));  
        return (TRUE);
    }
    if ( (Interrupt = OEMTranslateSysIntr(idInt)) != -1 ) {
        PICEnableInterrupt((UCHAR)Interrupt, TRUE);  // Enable the ISA interrupt
        return (TRUE);
    }
    return (FALSE);
}
```

### OEMInterruptDisable

This function disables the specified hardware interrupt. When a device driver is being unloaded and calls the **InterruptDisable** kernel routine, the kernel in turn calls **OEMInterruptDisable**. The system cannot be preempted when this function is called.
Sample Code

```c
void OEMInterruptDisable(
    DWORD idInt    // @parm Interrupt ID
to be disabled. See <t Interrupt ID's>
    // for the list of possible values.
)
{
    DWORD Interrupt;
    if ( (Interrupt = OEMTranslateSysIntr(idInt)) != -1 ) {
        PICEnableInterrupt((UCHAR)Interrupt, FALSE);
    }
}
```

**OEMInterruptDone**

This function signals completion of interrupt processing. The kernel calls this function when a device driver calls **InterruptDone**. The system cannot be preempted when this function is called. **OEMInterruptDone** should re-enable the interrupt if the interrupt was previously masked.

Sample Code

```c
void OEMInterruptDone(
    DWORD idInt    // @parm Interrupt ID.
See <t Interrupt ID's>
    // for the list of possible values.
)
{
    DWORD   Interrupt;
    if ( (Interrupt = OEMTranslateSysIntr(idInt)) != 0xFF ) {
        PICEnableInterrupt((UCHAR)Interrupt, TRUE);
    }
}
```
Interrupt Processing (continued)

**Topic Objective**  
To describe the interrupt processing routines.

**Lead-in**  
In this section, you will learn how to implement the interrupt processing routines.

- **OEMGetInterrupt()**  
  - Used by the PCI bus enumerator to request an IRQ for a PCI device

- **OEMRequestSysIntr()**  
  - Used in the OEMIoControl routine to implement IOCTL_HAL_TRANSLATE_IRQ and IOCTL_HAL_REQUEST_SYSINTR

- **OEMTranslateIrq()**  
  - Used by the main ISR to translate a non-shareable IRQ into a SYSINTR, or the last shared IRQ that is not covered by an installable ISR

- **OEMTranslateSysIntr()**  
  - Maps a SYSINTR to its corresponding IRQ

The code listing in this section uses the following variables and defines to map the SYSINTR to the IRQ.

```c
ULONG Logic2SysIntr[LOGINTR_MAX + 1];
ULONG SysIntr2Logic[SYSINTR_MAXIMUM];

#define SETUP_INTERRUPT_MAP(SysIntr, Irq)  
{  
  SysIntr2Logic[SysIntr] = Irq;  
  Logic2SysIntr[Irq] = SysIntr;  
}
```

**OEMGetInterrupt**  
This function is used by the PCI bus enumerator to request an IRQ for a PCI device.
Sample Code

BOOL OEMGetInterrupt(
    PDEVICE_LOCATION pDevLoc,
    PDWORD pIrq
)
{
    BOOL RetVal = TRUE;
    int Bus = pDevLoc->BusNumber;
    int Device = (pDevLoc->LogicalLoc >> 8) & 0xFF;

    *pIrq = 0xFF;

    // Currently supporting only PCIbus
    if (pDevLoc->IfcType != PCIBus) return FALSE;

    // Make sure interrupt Pin is in range
    if ((pDevLoc->Pin < 1) || (pDevLoc->Pin > 4)) return FALSE;

    if (Bus == 0) {
        // Bus 0 has devices 9 through 12 mapped
        if (((Device < 9) || (Device > 12)) {
            return FALSE;
        }

        *pIrq = IrqTable[Device - 9][pDevLoc->Pin - 1];
    return TRUE;
    } else if (Bus == 3) {
        // Only for PCI bus extender plugged into slot 3
        Device = (Device + 1) % 4;

        *pIrq = IrqTable[Device][pDevLoc->Pin - 1];
    } else {
        // All other buses behind PCI-PCI bridge conform to PCI convention
        Device = Device % 4;

        *pIrq = IrqTable[Device][pDevLoc->Pin - 1];
    }
    return RetVal;
}

OEMRequestSysIntr

This function is used in the OEMIoControl routine to implement IOCTL_HAL_TRANSLATE_IRQ and IOCTL_HAL_REQUEST_SYSINTR.
Sample Code

```c
DWORD OEMRequestSysIntr(
    DWORD Irq
)
{
    DWORD SysIntr;

    if (Irq > LOGINTR_MAX) {
        return -1;
    }

    if ((Irq != IRQ_PCIINT0) && (Irq != IRQ_PCIINT1) && (Irq != IRQ_PCIINT2) && (Irq != IRQ_PCIINT3)) {
        // Irq is not shareable, return current SYSINTR mapping if there is one
        if (Logic2SysIntr[Irq] != -1) {
            return Logic2SysIntr[Irq];
        }
    }

    // Look for next free SYSINTR
    for (SysIntr = SYSINTR_FIRMWARE; SysIntr < SYSINTR_MAXIMUM; SysIntr++) {
        if (SysIntr2Logic[SysIntr] == -1) break;
    }

    // No free SYSINTRs left
    if (SysIntr >= SYSINTR_MAXIMUM) {
        return -1;
    }

    // Make SYSINTR -> Irg association
    SysIntr2Logic[SysIntr] = Irq;

    // Make Irg -> SYSINTR association, only if not multiply-mapped
    if (Logic2SysIntr[Irq] == -1) {
        Logic2SysIntr[Irq] = SysIntr;
    }

    return SysIntr;
}
```

**OEMTranslateIrq**

This function is used by the main ISR to translate a non-shareable IRQ into a SYSINTR, or the last shared IRQ that is not covered by an installable ISR. This is a direct one-to-one mapping of IRQ to SYSINTR.
Sample Code
DWORD OEMTranslateIrq(
    DWORD Irq
)
{
    if (Irq > LOGINTR_MAX) {
        return -1;
    } else {
        return Logic2SysIntr[Irq];
    }
}

OEMTranslateSysIntr
This function maps a SYSINTR to its corresponding IRQ.

Sample Code
DWORD OEMTranslateSysIntr(
    DWORD SysIntr
)
{
    if (SysIntr >= SYSINTR_MAXIMUM) {
        return -1;
    } else {
        return SysIntr2Logic[SysIntr];
    }
}

Note  The example code listings above are from
\WINCE400\PLATFORM\ARMINTEGRATOR\KERNEL\HAL\cfw_platform .c
Another important function for which you must write code in your OAL is the \textbf{OEMIoControl} function.

\textbf{OEMIoControl}

This function is called by the Kernel when a device driver or application program calls the \textbf{KernelIoControl} function. The \textbf{OEMIoControl} function allows the OEM device driver or application to communicate with kernel mode HAL code.

The prototype of this function is as follows:

\begin{verbatim}
BOOL OEMIoControl (DWORD dwIOControlCode,
                   LPVOID lpInBuf,
                   DWORD nInBufSize,
                   LPVOID lpOutBuf,
                   DWORD nOutBufSize,
                   LPDWORD lpBytesReturned);
\end{verbatim}

- \texttt{dwIOControlCode = IOCTL_HAL_GET_DEVICE_INFO}
  
  Provides system information
  
  The kernel calls this function when an application calls the \textbf{SystemParametersInfo} function to determine the platform type or OEM information. In this case, \textbf{OEMIoControl} returns a string with platform or OEM information.

- \texttt{dwIOControlCode = IOCTL_HAL_REBOOT}
  
  Carries out a warm reset
  
  You implement this control code when you want an application to perform a warm reset. Your application can then call the \textbf{KernelIoControl} function with this code, and the kernel converts this call into an \textbf{OEMIoControl} call.

- \texttt{dwIOControlCode = IOCTL_HAL_INIT_RTC}
Initializes RTC

When the system performs a cold boot, it calls the **OEMIoControl** function to initialize RTC. You implement the **IoControlCode** function if you want to support RTC initialization.

---

**Note** For a more complete list, see WINCE400\PLATFORM\CEPC\KERNEL\HAL\OEMIOCTL.C

---

**Extend the Ethernet Debugging Interface**

Applications cannot access debug services (ETHDBG.LIB) directly, unlike the debug module. However, you can allow applications to use Ethernet debugging indirectly by implementing the following control codes relative to Ethernet Debugging:

- IOCTL_EDBG_REGISTER_CLIENT
- IOCTL_EDBG_DEREGISTER_CLIENT
- IOCTL_EDBG_SEND
- IOCTL_EDBG_RECV
- IOCTL_EDBG_SET_DEBUG
Example: Custom Kernel IOCTL

You can create your own control code to implement kernel mode functionality and allow applications to use it. With your new I/O control code implemented, you can use it in an application by calling `KernelIoControl`. The kernel automatically converts it into an `OEMIoControl` call.

The example displayed on the slide adds an I/O control code `IOCTL_MY_CONTROL1`.

To add an I/O control code:

1. Create the control code by defining it. You can add that control code in `pkfuncs.h` or create a new *.h file and add it there.
2. Modify `OEMIoControl` function. Add the control code and your specific task. For example, you can add code to turn on an LED or Toggle a GPIO line.
3. Use `KernelIoControl` function to exercise the I/O control code. The `KernelIoControl` function is the mechanism in which an application or device driver can use to perform this I/O control.

Note  See WINCE400\PUBLIC\COMMON\OAK\INC\pkfuncs.h for list of currently used I/O control codes.

When assigning the function numbers of the CTL_CODE, remember that function codes 0-2047 are reserved for Microsoft Corporation, and 2048-4095 are reserved for OEMs/IHVs. The function code can be no larger then 4095.

Here is a test application exercising the `KernelIoControl` function. The test application performs the I/O control using the `IOCTL_MY_CONTROL1` code. After that it takes the contents of Param3 and stuffs it into the global variable `gVariable`. 
Void Test() {
    UNIT Param1;
    ULONG Param3;
    unsigned long dwReturned = 0;

    if (!KernelIoControl(IOCTL_MY_CONTROL1,
        (LPVOID)&Param1, // Input buffer
        sizeof(DWORD), // Input buffer size
        (LPVOID)&Param3, // Output buffer
        sizeof(DWORD), // Output buffer size
        &dwReturned)) // Bytes returned
    {
        gVariable = Param3;
        DEBUGMSG(1, (TEXT("ERROR – IOCTL_MY_CONTROL1\r\n")));
    }
}
Prior to Windows CE .NET, you could only debug your platform with a provided pre-defined transport like serial, parallel, or Ethernet. Windows CE .NET introduces Kernel Independent Transport Layer (KITL) that is designed to provide an easy way for you to support any debug service.

KITL separates the protocol of the communication service from the layer that communicates directly with the communication hardware. This reduces your involvement in creating a hardware transport layer that understands how to pass data to the communication hardware of the device. The hardware transport layer is then layered under KITL so that KITL does not require to understand different types of communication hardware.

**Example**

You create both desktop and Windows CE device side transport mechanisms. On the desktop, the transport is a separate DLL that exports certain API functions that KITL relies on and is also registered in the system so KITL knows that it is a functional transport. On the device, the transport is built into the OAL and therefore, the kernel. On the CE device, KITL relies on the transport to support a set of API level calls that are needed to support the debug services.

To include the KITL support in the operating system image:

1. Include VBridge.lib in
   `%WINCEROOT%\KERNEL\BUILDEXE\..\..\..\SOURCES`.

2. Add support for VMini.dll to Ethdbg.h and include it in the source file containing the `OEMIoControl` function. Add the following IOCTLs:
### Module 5: OEM Adaptation Layer

IOCTL_VBRIDGE_GET_TX_PACKET:
IOCTL_VBRIDGE_GET_TX_PACKET_COMPLETE:
IOCTL_VBRIDGE_GET_RX_PACKET:
IOCTL_VBRIDGE_GET_RX_PACKET_COMPLETE:
IOCTL_VBRIDGE_GET_ETHERNET_MAC:
IOCTL_VBRIDGE_CURRENT_PACKET_FILTER:
IOCTL_VBRIDGE_802_3_MULTICAST_LIST:
IOCTL_VBRIDGE_WILD_CARD:
IOCTL_VBRIDGE_CURRENT_PACKET_FILTER:
IOCTL_VBRIDGE_802_3_MULTICAST_LIST:
IOCTL_VBRIDGE_SHARED_ETHERNET:

3. Call the VBridgeInit function in the OEMEthInit function.

4. Call the VBridgeKSetLocalMacAddress function in the OEMEthInit function to inform VBridge of the media access control address of the underlying Ethernet hardware.

5. Add the following ProcessVMiniSend function.

6. Modify the OEMEthISR function. Replace pfnXXX with the platform's appropriate function call to its Ethernet library.

7. Update the OEMEthGetFrame function.

8. Update the OEMEthSendFrame function.

9. Optional support for FILTERING and MULTICASTING can be implemented for the platform by using the IOCTL_VBRIDGE_CURRENT_PACKET_FILTER and IOCTL_VBRIDGE_802_3_MULTICAST_LIST functions.

**Note** Refer to %WINCEROOT%\PLATFORM\CEPC\Halether.c for sample implementations of these functions. Without these functions, VMINI will still work but will not provide multicast support for Winsock applications.

10. Update OEMInterruptEnable to set idInt — SYSINTR_VMINI. SYSINTR_VMINI is defined as SYSINTR_DEVICES + 7. Verify that your platform does not use the SYSINTR_DEVICES + 7 slot.

To enable the VMini virtual adapter:

1. Include VMini.dll in the operating system image.

2. Update the registry to support VMini.dll.
Optional OAL Functions

In this section, you will learn about implementing the optional functions used by the OAL:

- Real-time Clock (RTC) and Timer
- Parallel Port
- Ethernet Port Debug
- OEMGetExtensionDRAM()
Real-time Clock and Timer

- **OEMGetRealTime()**
  - Called by the kernel to get the time from the real-time clock

- **OEMSetRealTime()**
  - Sets the real time clock

- **OEMQueryPerformanceCounter()**
  - Retrieves the current value of the high-resolution performance counter, if one exists

- **OEMQueryPerformanceFrequency()**
  - Retrieves the frequency of the high-resolution performance counter, if one exists

- **OEMSetAlarmTime()**
  - Sets the alarm time

The Real-time clock functions manage the time-of-day information.

**OEMGetRealTime()**

This function is called by the kernel to get the time from the real-time clock. This function must be reentrant and, thus, must protect the hardware from multiple accesses.

Calling `GetSystemTime` in an application in turn lets the kernel call `OEMGetRealTime`.

The **OEMGetRealTime** function is shown below:

```
BOOL OEMGetRealTime(LPSYSTEMTIME lpst)
{
    BOOL RetVal;
    if (!fRTCInit) {
        InitializeCriticalSection(&RTC_critsect);
        fRTCInit = TRUE;
    }
    EnterCriticalSection(&RTC_critsect);
    RetVal = Bare_GetRealTime(lpst);
    LeaveCriticalSection(&RTC_critsect);
    return RetVal;
}
```

**OEMSetRealTime()**

This function sets the real time clock. This function is called by the kernel to set the real-time clock. This function must be reentrant and must protect the
hardware from multiple accesses. If your hardware provides two digits to hold
the value for the year, you must add additional code to avoid the year 2000
problem.

Calling **SetSystemTime** in an application in turn lets the kernel call
**OEMSetRealTime**.

The **OEMSetRealTime** function is shown below:

```c
BOOL OEMSetRealTime(LPSYSTEMTIME lpst)
{
    BOOL RetVal;
    if (!fRTCInit) {
        InitializeCriticalSection(&RTC_critsect);
        fRTCInit = TRUE;
    }
    EnterCriticalSection(&RTC_critsect);
    RetVal = Bare_SetRealTime(lpst);
    LeaveCriticalSection(&RTC_critsect);
    return RetVal;
}
```

**OEMQueryPerformanceCounter()**

The **OEMQueryPerformanceCounter** function retrieves the current value of
the high-resolution performance counter, if one exists.

Calling **QueryPerformanceCounter** in an application allows the kernel to call
the **OEMQueryPerformanceCounter**.

The **OEMQueryPerformanceCounter** function is shown below:

```c
BOOL
OEMQueryPerformanceCounter(
    LARGE_INTEGER *lpliPerformanceCount
)
{
    ULARGE_INTEGER liBase;
    DWORD dwCurCount;
    // Make sure CurTicks is the same before and after read of
counter to account for
    // possible rollover
    do {
        liBase = CurTicks;
        dwCurCount = PerfCountSinceTick( );
    } while  (liBase.LowPart != CurTicks.LowPart);
    lpliPerformanceCount->QuadPart = liBase.QuadPart +
    dwCurCount;
    return TRUE;
}
```

**OEMQueryPerformanceFrequency()**

The **OEMQueryPerformanceFrequency** function retrieves the frequency of
the high-resolution performance counter, if one exists.
Calling `QueryPerformanceFrequency` in an application allows the kernel to call the `OEMQueryPerformanceFrequency`.

The `OEMQueryPerformanceFrequency` function is shown below:

```c
BOOL OEMQueryPerformanceFrequency(
    LARGE_INTEGER *lpliPerformanceFreq
)
{
    lpliPerformanceFreq->HighPart = 0;
    lpliPerformanceFreq->LowPart  = PerfCountFreq();
    return TRUE;
}
```

Some real time clock hardware contains an alarm feature. The alarm feature works as follows:

- First, you set a register (the alarm register) with an alarm time value.
- Then, the RTC compares this alarm time value with the current time. If the current time is equal to the alarm set time, the RTC raises an ALARM interrupt. The ISR for the alarm then processes this and return a SYSINTR_ALARM notifying the kernel that the alarm has been set.

### OEMSetAlarmTime()

This function is called to set the alarm time. Do not forget to modify the alarm ISR to return SYSINTR_RTC_ALARM.

```c
BOOL OEMSetAlarmTime(LPSYSTEMTIME lpst)
{
    BOOL RetVal;
    if (!fRTCInit) {
        InitializeCriticalSection(&RTC_critsect);
        fRTCInit = TRUE;
    }
    EnterCriticalSection(&RTC_critsect);
    RetVal = Bare_SetAlarmTime(lpst);
    LeaveCriticalSection(&RTC_critsect);
    return RetVal;
}
```

**Note** An example of an alarm ISR can be found in the `PeRPISR()` function at `\WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\fwpc.c`.

The `OEMSetAlarmTime` function is shown below:

```c
BOOL OEMSetAlarmTime(LPSYSTEMTIME lpst)
{
    BOOL RetVal;
    if (!fRTCInit) {
        InitializeCriticalSection(&RTC_critsect);
        fRTCInit = TRUE;
    }
    EnterCriticalSection(&RTC_critsect);
    RetVal = Bare_SetAlarmTime(lpst);
    LeaveCriticalSection(&RTC_critsect);
    return RetVal;
}
```

**Note** The sample code above came from `\WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\rtc.c`.  


Parallel Port I/O Code

**Topic Objective**
To describe the implementation of the parallel port I/O code.

**Lead-in**
In this section, you will learn how to implement the parallel port I/O code.

- **OEMParallelPortSendByte()**
  - Sends a byte of data from the target device to the development workstation over the parallel port

- **OEMParallelPortGetByte()**
  - Receives a byte of data on the target device from the development workstation over the parallel port interface

- **OEMParallelPortGetStatus()**
  - Checks the status of the parallel port and returns a byte if the port is ready

You can download binary files from the development workstation to the target device by using the parallel port. To enable this, you need to implement the parallel port interface code for the target device.

The sample code given below lists the functions needed to be implemented to adapt the parallel port device interface on the target device.

**Note** The sample code below came from \WINCE400\PLATFORM\CEPC\KERNEL\HAL\MDPPFS.C/

**OEMParallelPortSendByte()**
This function sends a byte of data from the target device to the development workstation over the parallel port.

The **OEMParallelPortSendByte** function is shown below.
VOID OEMParallelPortSendByte(BYTE chData)
{
    if ( NoPPFS )
        return;
    LOG_ENTRY(LOG_ENTRY_WRITE);
    if ( !WaitForStatus(PAR_STAT_NBUSY, 0) ) {
        return;
    }
    LOG_ENTRY((USHORT)(LOG_ENTRY_DATA | chData));
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_DATA, chData);
    LOG_ENTRY(LOG_ENTRY_CONTROL | PAR_CTRL_AUTOFEED);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_AUTOFEED);
    if ( !WaitForStatus(PAR_STAT_NBUSY, PAR_STAT_NBUSY) ) {
        return;
    }
    LOG_ENTRY(LOG_ENTRY_CONTROL | PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    LOG_ENTRY(LOG_ENTRY_EXIT | 0);
    bLastOpWasWrite = TRUE;
}

OEMParallelPortGetByte()

This function receives a byte of data on the target device from the development workstation over the parallel port interface.

The OEMParallelPortGetByte function is shown below.
int OEMParallelPortGetByte(void)
{
    BYTE value;
    if ( NoPPFS ) {
        return (-1);
    }
Retry:
    LOG_ENTRY(LOG_ENTRY_READ);
    LOG_ENTRY(LOG_ENTRY_CONTROL | PAR_CTRL_READ | PAR_CTRL_STROBE);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_READ | PAR_CTRL_STROBE);
    if ( !WaitForStatus(PAR_STAT_NACK, PAR_STAT_NACK) ) {
        return (-1);
    }
    value = READ_PORT_UCHAR(IoPortBase + PAR_PORT_DATA);
    LOG_ENTRY((USHORT)(LOG_ENTRY_DATA | value));
    LOG_ENTRY(LOG_ENTRY_CONTROL | PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    WRITE_PORT_UCHAR(IoPortBase + PAR_PORT_CTRL, PAR_CTRL_READ | PAR_CTRL_AUTOFEED | PAR_CTRL_STROBE);
    if ( !WaitForStatus(PAR_STAT_NACK, 0) ) {
        return (-1);
    }
    LOG_ENTRY(LOG_ENTRY_EXIT | 0);
    if ( bLastOpWasWrite && value == 0x1A ) {
        // Periodically after the first character we receive after a write is the last byte sent of the previous write.
        // For now we will ignore it
        LOG_ENTRY(LOG_ENTRY_EVENT | LOG_EVENT_SKIP_RECEIVE);
        bLastOpWasWrite = FALSE;
        goto Retry;
    }
    bLastOpWasWrite = FALSE;
    return (value);
}

OEMParallelPortGetStatus()
This function checks the status of the parallel port and returns a byte if the port is ready.

The OEMParallelPortGetStatus function is shown below:
int OEMParallelPortGetStatus(void)
{
    DWORD dwStatus, dwData;

    dwStatus = PAR_AUTOEN | PAR_SELECT | PAR_NFAULT | PAR_INTR_MASK;
    *(PVDWORD)(PAR_CONTROL_REG) = dwStatus;

    dwStatus = *(PVDWORD)(PAR_CONTROL_REG);
    dwData = *(PVDWORD)(PAR_DATA_REG);

    if(!(dwStatus & PAR_INTR))
        return -1;
    *(PVDWORD)(PAR_CONTROL_REG) = (PAR_BUSY | PAR_NFAULT | PAR_AUTOEN | PAR_SELECT);
    return (dwData & 0xff);
}
The Ethernet port debug functions support the interfaces between the platform and the debug shell tool CESH.EXE. These functions offer an alternative to the parallel port debug function, and can also support interfaces between the platform and the kernel debugger, or handle debug messages (OutputDebugString).

To implement Ethernet port debugging, you must implement the following functions:

- **OEMEthInit**
  Initializes the Ethernet debug port

- **OEMEthEnableInts**
  Enables interrupt on the Ethernet adapter

- **OEMEthDisableInts**
  Disables interrupt on the Ethernet adapter

- **OEMEthISR**
  Routine that handles interrupts for the Ethernet adapter

- **OEMEthGetFrame**
  Receives data from the Ethernet debug port

- **OEMEthSendFrame**
  Sends data over the Ethernet debug port

- **OEMEthQueryClientInfo**
  Retrieves platform-specific information

- **OEMEthGetSecs**
  Returns the number of seconds elapsed since a fixed time. This function is used for handling timeouts.
Example of these functions can be found in `WINCE400\PLATFORM\CEPC\KERNEL\HAL\HALETHER.C` and also in `Public\Common\Oak\Drivers\Ethdbg\Edbsamp`.

Be sure to link Ethdg.lib into the Nk.exe kernel. Adding Ethdb.lib enables the debugging capabilities on your target device. You must also supply appropriate drivers for your hardware. Platform Builder provides sample drivers for NE2000-compatible and SMC9000-compatible adapters in the `Ne2kdbg.lib` and `Sme9000.lib` files, respectively. Alternatively, you can write a custom driver for the Ethernet hardware on your target device.
OEMGetExtensionDRAM()

**BOOL OEMGetExtensionDRAM(LPDWORD lpMemStart, LPDWORD lpMemLen)**

```cpp
{  
    return FALSE; // no extension DRAM
}
```

- **Example 1:** No extension DRAM available
- **Example 2:** Extended 4MB DRAM available at 0x81800000

```cpp
BOO L OEMGetExtensionDRAM(LPDWORD lpMemStart, LPDWORD lpMemLen)
{
    *lpMemStart = 0x81800000;
    *lpMemLen = 0x00400000; // 4MB
    return TRUE;
}
```

- **pNKEnumExtensionDRAM AND OEMEnumExtensionDRAM**

Refer to the slide:

- The first example is for systems with no extended DRAM.
- The second example is used in a system that contains additional DRAM available starting at 0x81800000 with a size of 4MB.

The **OEMGetExtensionDRAM** returns information about extension DRAM. This is used by the kernel to add additional memory to the RAM pool. This additional memory is not contiguous with the main section of RAM that is defined in the config.bib file. The sum of the two pools (one from the config.bib and the other from this function) specifies where all the general purpose RAM is in the system. The RAM is used for both Objectstore and program execution.

**Note**  The sample code shown in the slide is from \WINCE400\PUBLIC\COMMON\OAK\CSP\I486\OAL\cfwpcc.c

If there is more than one noncontiguous RAM section, you need to implement the **pNKEnumExtensionDRAM** function pointer. The **pNKEnumExtensionDRAM** variable is capable of reporting up to 15 different noncontiguous RAM regions and like **OEMGetExtensionDRAM**, it is called during the cold boot phase. If **pNKEnumExtensionDRAM** is set, **OEMGetExtensionDRAM** is not called during the boot process.

**Note**  When adding multiple sections that there is a performance impact for every memory section that is added to the kernel. The impact is due to the kernel tracking and searching for free physical memory in different memory sections

When using **pNKEnumExtensionDRAM**, you must also implement **OEMEnumExtensionDRAM**.
Debugging an OAL

Windows CE loads Filesys.exe before any other applications. To test your OAL, create a test application that replaces Filesys.exe and prints in a loop by incrementing seconds using `Sleep` and `GetTickCount`.

To test the OAL:

1. Call the `GetTickCount` function.
2. Print the value returned by `GetTickCount`.
3. Call the `Sleep` function to suspend the thread. Monitor the duration of the sleep period.
4. Call `GetTickCount` again.
5. Print the value returned by `GetTickCount`.
6. Compare the two values returned by `GetTickCount`.

The second call to `GetTickCount` should return the amount of time that has elapsed since the previous call to this function. Therefore, the value returned by the second call to `GetTickCount` should be non-zero and should not match the value returned from the first call to this function.

**Note**  
`GetTickCount` returns time in the form of milliseconds.

If the value returned by the second call to `GetTickCount` matches the amount of time that your system was in a suspended state, then you have successfully created a base OAL. You are now ready to enhance the functionality of your OAL.
The example shown in this slide is the sources file from \WINCE400\PLATFORM\CEPC\KERNEL\BUILDEXE\KERNKITL. The Sources file determines what libraries are used to build the CE kernel. This particular example is the sources file to build KERNKITL.EXE.

To build the Windows CE kernel, Platform Builder uses the following set of CPU-specific libraries:

- **nk.lib**
  Base code supplied by Microsoft for a specified microprocessor

- **kitl.lib**
  Base code supplied by Microsoft for a specified microprocessor

- **kitleth.lib**
  Common OAL Ethernet debugging interface that you implement. This contains the Ethernet port I/O code.

**Note** See \WINCE400\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\KITLETH

- **smc9000.lib**
  SMC9000 sample ethernet debugging adapter driver.

**Note** See \WINCE400\public\common\oak\DRIVERS\ETHDBG\SMC9000

- **ne2kdbg.lib**
  NE2000 sample ethernet debugging adapter driver.

**Note** See \WINCE400\public\common\oak\DRIVERS\ETHDBG\NE2000
Module 5: OEM Adaptation Layer

- rne_mdd.lib
  Remote NDIS support for network devices.

  **Note** See \WINCE400\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\RNE_MDD

- net2890lib.lib
  NET2890 USB interface controller driver. See \WINCE400\PUBLIC\COMMON\OAK\DRIVERS\RNDISFN\NET2890.

- eboot.lib
  Routines to communicate with eshell program on the desktop to get our configuration information (which services are configured over Ethernet, and what the host addresses are). See \WINCE400\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\EBOOT.

- PCireg.lib
  Populate initial registry with already configured devices. See \WINCE400\PUBLIC\COMMON\OAK\DRIVERS\HELPER\PCIREG.

  - i486oal.lib
    x86 Common OAL code. Built from \WINCE400\public\common\oak\CSP\I486.

  - loadauth.lib
    loadauth.lib is a sample verification library. These contain functions to verify that the file contains the proper signature before the kernel loads the file.

- fulllibc.lib
  Microsoft® C Run-Time Library for Windows CE.

- schedlog.lib
  schedlog.lib provides functions to assist OEMs using the LogThread and LogProcess APIs.

- profile.lib
  Kernel-call instrumented kernel profiling and Monte Carlo profiling; a component of Hal.lib.

- vbridge.lib
  This library functions as a media access control layer bridge that connects the TCP/IP and Winsock application traffic to the kernel’s Ethernet debug traffic.

OEM provided/modified:

- hal.lib
  OEM component that abstracts the kernel from the hardware. See \WINCE400\PLATFORM\CEPC\KERNEL\HAL\SOURCES for examples

- Profiler.lib
  OEM component with profiling support. See \WINCE400\PLATFORM\CEPC\KERNEL\PROFILER for an example.
**Note**  NK.EXE is actually one of the following file depending on the type of kernel you are build. In COMMON.BIB, the NK.EXE can be one of the following files:

nk.exe ← kernkitlprof.exe
nk.exe ← kern.exe
nk.exe ← kernkitl.exe
Building the Windows CE Kernel (continued)

The example shown in this slide is the sources file from \WINCE400\PLATFORM\CEPC\KERNEL\BUILDEXE\KERNKITLPROF. The Sources file determines what libraries are used to build the CE kernel. This particular example is the sources file to build KERNKITLPROF.EXE.

Profiler.lib is the OEM component with profiling support.

Note See \WINCE400\PLATFORM\CEPC\KERNEL\PROFILER for an example.
Implementing OAL Registry Functions

- **WriteRegistryToOEM()**
  - Enables writing the registry file to persistent storage defined by the OEM
  - Operating system exposes a global pointer named `pWriteRegistryToOEM` that is available in the OAL

- **ReadRegistryFromOEM()**
  - Reads into RAM a registry file from persistent storage defined by the OEM
  - Operating system exposes a global variable named `pReadRegistryFromOEM` that is available in the OAL

- **RegFlushKey()**
  - Writes all the attributes of the specified open key into the registry

Windows CE supports a true persistent registry and provides functions that you can use to save and load the registry from a specified location. To initialize a RAM-based registry, the OEM must expose the `ReadRegistryFromOEM` and `WriteRegistryToOEM` functions in the OAL.

**WriteRegistryToOEM()**

This function enables writing the registry file to persistent storage defined by the OEM. The operating system exposes a global pointer named `pWriteRegistryToOEM` that is available in the OAL.

To support persistent storage of the registry, you must implement the function `WriteRegistryToOEM` and assign the address of the function to `pWriteRegistryToOEM`.

When the operating system is instructed to flush the registry, the operating system checks for a valid `pWriteRegistryToOEM`. If `pWriteRegistryToOEM` is valid, the operating system continues to call this function to send bytes to the OEM implemented function until it returns `FALSE` (an error occurred) or the operating system completes the flush.

The registry is notified that it needs to be flushed when `RegFlushKey` is called by an application.

Only a fully saved registry can be restored; attempting to restore a partial registry produces unpredictable results. If you try to save a partial registry, `cbData` is never set to 0. Then, when `WriteRegistryToOEM` is called, an error should be returned because the registry file is incomplete.

**Sample Code**

```c
BOOL WriteRegistryToOEM ( DWORD dwFlags, LPBYTE lpData, DWORD cbData );
```

The explanation of the parameters used in the sample code is as follows:
- **dwFlags**
  Read options specified by the system; currently, the only flag is `REG_WRITE_BYTES_START`, which indicates the start of the new registry file.

- **lpData**
  Pointer to a buffer allocated by the OS and filled with registry bytes. Your `WriteRegistryToOEM` function retrieves the bytes and saves them in persistent storage.

- **cbData**
  The number of bytes in the buffer `lpData` passed in by the operating system. When set to 0, then end-of-file has been reached.

**ReadRegistryFromOEM()**

This function reads into RAM a registry file from persistent storage defined by the OEM. The operating system exposes a global variable named `pReadRegistryFromOEM` that is available in the OAL. To support restoring the registry from persistent storage, implement the function `ReadRegistryFromOEM` and assign the address of the function to `pReadRegistryFromOEM`. During registry initialization, the operating system checks for a valid `pReadRegistryFromOEM`. If it is valid, the operating system continues to call `pReadRegistryFromOEM` to retrieve bytes until `pReadRegistryFromOEM` returns 0 (end of file) or -1. If there are severe problems or conditions under which the initialization cannot continue, you may implement this function such that it does not return. For example, you can inform the end user of the problem and possible solutions. Because the operating system is running a single thread at this stage of initialization, no other operating system activity occurs unless this function returns.

**Sample Code**

```c
DWORD ReadRegistryFromOEM ( DWORD dwFlags, LPBYTE lpData, DWORD cbData );
```

The explanation of the parameters used in the sample code is as follows:

- **dwFlags**
  Read options specified by the OS; `REG_READ_BYTES_START` indicates reading must start from the beginning of the registry file.

- **lpData**
  Pointer to a buffer allocated by the OS. You must load the buffer with registry bytes up to a maximum of `lpcbData`.

- **cbData**
  Size in bytes of the buffer to which `lpData` points; passed in by the OS.

**RegFlushKey()**

This function writes all the attributes of the specified open key into the registry. The `RegFlushKey` function has two different behaviors, varying with the type of registry in use on the device. The two registry types available are the object store-based registry and the hive-based registry; the registry type is determined by the OEM.
LONG RegFlushKey( HKEY hKey );

The explanation of the parameters used in the sample code is as follows:

- **hKey**
  Handle to a currently open key or one of the following predefined reserved handle values:
  - HKEY_CLASSES_ROOT
  - HKEY_CURRENT_USER
  - HKEY_LOCAL_MACHINE
  - HKEY_USERS
In this section, you will learn about:

- How the Windows CE .NET Power Manager controls battery performance.
- How a typical system will transition between custom defined power states.
- How GWES handles control of On-to-Suspend power state.
- The OAL power management functions that are implemented on the battery operated system.
Power Manager

- Provides power management for each device
- Improves overall system power efficiency
- Implemented via Pm.dll
- Define your own System Power States
  - RunAC, RunDC, Suspend
- Device Power States (statically pre-defined)
  - Full on (D0), Low on (D1), Standby (D2), Sleep (D3), Off (D4)
- Uses IOCTLs’ to manipulate the power states of the system

The Power Manager improves overall system power efficiency, provides power management for each device, and co-exists with applications and drivers that do not support the Power Manager.

- You can use power management to reduce the power consumption of a target device and to maintain and preserve the file system in RAM during the reset, on, idle, and suspend power states.
- The Power Manager (Pm.dll) does not limit the platform power requirements or design.
- The Power Manager is built into the operating system image by default, but can be removed by setting the BSP_NOPM environment variable. This removes Pm.dll from the operating system image.
- You define your own system power states.
- RunAC, RunDC, Suspend, and so on.
  - These are not pre-defined, and are not necessarily linearly ordered. You define state names as registry keys in the system configuration. There is no limit on how many system power states you can define.
- You are also responsible for creating explicit mappings between the predefined Device Power States and the System Power States.
- Device Power state definitions are statically pre-defined. 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.

A physical device does not have to support all of the device power states. The only device power state that all devices must support is the full on state, D0. A driver that is issued a request to enter a power state not supported by its device enters the next available power state supported. If a device cannot wake up the system then it should power off rather than staying in standby.

When a device driver is loaded it should put the device into full on, D0. Before a driver is unloaded, if possible, it should put the device into off, D4.
<table>
<thead>
<tr>
<th>Device Power State</th>
<th>Registry Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full on</td>
<td>D0</td>
<td>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.</td>
</tr>
<tr>
<td>Low on</td>
<td>D1</td>
<td>State in which the device is fully functional at a lower power and/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.</td>
</tr>
<tr>
<td>Standby</td>
<td>D2</td>
<td>State in which the device is partially powered with automatic wakeup on request. A device in state D2 is effectively standing by.</td>
</tr>
<tr>
<td>Sleep</td>
<td>D3</td>
<td>State in which the device is partially powered with device-initiated wakeup if available. A device in state D3 is sleeping but capable of raising the System Power State 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.</td>
</tr>
<tr>
<td>Off</td>
<td>D4</td>
<td>State in which the device is unpowered. 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; a device on such a bus can still support D4.</td>
</tr>
</tbody>
</table>

- There exists a set of kernel IOCTLs’ to go along with power management system. The Power Manager uses these IOCTLs’ to manipulate the power states of the system:
  - IOCTL_HAL_ENABLE_WAKE
  - IOCTL_HAL_DISABLE_WAKE
  - IOCTL_POWER_SET
  - IOCTL_POWER_GET
- IOCTL_POWER_QUERY
- IOCTL_POWER_CAPABILITIES
- IOCTL_POWER_SEQUENCE
- IOCTL_REGISTER_POWER_RELATIONSHIP
You explicitly define system power state names as registry keys in the system configuration.

Refer to the example displayed on the slide. The system power state to device power state mappings are enumerated as values under each power state name key in the registry. The following table describes the values and their description:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Defines your system power state name.</td>
</tr>
<tr>
<td>Flags</td>
<td>Defines whether the state pertains to AC, DC, AC and DC, behavioral settings, and so on.</td>
</tr>
<tr>
<td>(Default)</td>
<td>Default device power setting for all devices while at system power state Name</td>
</tr>
<tr>
<td>Dx</td>
<td>Device power state. Set to D0, D1, D2, D3, or D4. This is the state the device runs in while at system power state Name.</td>
</tr>
<tr>
<td>DeviceName</td>
<td>Optional. Specifies devices with specific device power state requirements other than the (Default).</td>
</tr>
</tbody>
</table>

For example, COM1: dword:3

If the system configuration does not specify any system power state keys, then the default state for all devices is full on (D0).

The example below is an example of mapping system power state and the device power state. In this case, RunAC is mapped to D0, RunDC is mapped to D1, and Suspend is mapped to D3.
; Example SystemState-to-DeviceState mappings
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\RunAC]
   ""=dword:0
   "Flags"=dword:80010001
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\RunDC]
   ""=dword:1
   "Flags"=dword:80010002
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Suspend]
   ""=dword:3
   "Flags"=dword:80200003

The following registry example shows how to define your own power states. In this example, the registry forces the Power Manager to put all devices in D0 when in system power state Custom1, except for COM2, which enters D1. The Power Manager puts all devices in D4 when in system power state Custom2, except for COM2, which enters D3.

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Custom1]
   Flags:xxxx
   (Default):D0
   COM2:D1

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Custom2]
   Flags:xxxx
   (Default):D4
   COM2:D3
Transition Between States: An Example

The Windows CE operating system undergoes the following transitions between its states:

- No power to On
  When you power your device, it clears the file system and working RAM and goes into the On state.

- Cold Boot
  A cold boot is a complete reset of your platform. Both the file system and RAM are cleared. This happens when you remove all batteries from your system.

- Warm Boot
  A warm boot clears the working RAM. This happens when you call a warm boot through a program, or when the user pushes the Reset button.

- On to Idle
  When the kernel does not have any thread to run, it changes the power state to Idle. The CPU waits for an interrupt.

- Idle to On
  There is a transition from the Idle state to the On state when an interrupt occurs. Typically, the timer interrupt causes this transition.

- On to Suspend
  There is transition from the On state to the Suspend state if one of the following events occurs:
  - Time out of the inactivity timer
  - The Off button has been pressed

Note  You can add other events such as battery cover opened or battery low interrupt.
Suspend to On

There is a transition between the Suspend state and the On state if one of the following events occurs:

- The On button has been pressed.
- An alarm event occurs.
- Any wake-up event occurs.

On to Critical Off

There is a transition between these two modes when a critical low-battery event occurs. When you add new batteries, the system performs a warm boot.
GWES Control of On-to-Suspend Power State

You modify registry values to allow GWES to control the On-to-Suspend power state.

The registry values are described as follows:

<table>
<thead>
<tr>
<th>Registry value name</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BattPowerOff</td>
<td>300 (5 minutes)</td>
<td>On battery power, the number of seconds without user input activity before the system suspends.</td>
</tr>
<tr>
<td>ExtPowerOff</td>
<td>0 (disabled)</td>
<td>On external power, the number of seconds without user input activity before the system suspends.</td>
</tr>
<tr>
<td>WakeupPowerOff</td>
<td>60 (1 minute)</td>
<td>Following a non-user input event wakeup such as an alarm, the number of seconds without user input activity before the system suspends.</td>
</tr>
<tr>
<td>ScreenPowerOff</td>
<td>0 (disabled)</td>
<td>Number of seconds without user input activity before GWES posts an APM_POWERBROADCAST, PBT_APMUSERIDLE message to the registered taskbar window.</td>
</tr>
</tbody>
</table>
When the system changes its power state, it calls OAL functions to turn the system off or to idle it. These system calls may be triggered by either hardware or software events, such as throwing a power switch or an idle timer count.

There are two functions in the OAL that can be called for this purpose:

- OEMIdle
- OEMPowerOff

The kernel calls OEMIdle when there are no threads to run. The OEM device is requested to go into a sleep or idle state. This consists of saving the current state, placing the memory into a refresh state, stopping the clock and suspending execution. Then, when an interrupt, including a scheduled interrupt occurs, the devices ends it’s idle state, the previous state is restored, and the scheduler is invoked. If no new threads are ready to run, OEMIdle is called again.

In Windows CE .NET, the following variables are no longer required in the OAL by the kernel.

- dwSleepMin
- DwPartialDiffMSec
- Ticksleft
- DiffMSec

A new variable is added to the OAL to determine if the kernel should schedule a thread to run. This affects both OEMIdle and the handling of the system tick. The new variable is called dwReschedTime. The sample code uses a compile switch (#if CE_MAJOR_VER) to build either versions of OEMIdle.
OEMIdle
This function is called when the system into the idle state. This is the lowest energy state possible balanced with the need to return from the idle quickly. The OEMIdle function is a completely CPU dependent function.

Note A sample of OEMIdle can be found at:
\Public\common\oak\cspi486\oal\timer.c

OEMPoweOff
This function is invoked when the OFF button is pressed or GWES times out on its power-off timer. It is responsible for any final power-off state and putting the CPU into suspend mode.

Sample Code
void OEMPoweOff(void) {
    UCHAR ucOldMasterMask, ucOldSlaveMask;
    INTERRUPTS_OFF();
    //
    // Disable all the system interrupts, except those requested to wake
    //
    ucOldMasterMask = GetPICInterruptMask(TRUE);
    ucOldSlaveMask = GetPICInterruptMask(FALSE);
    SetPICInterruptMask(TRUE, (UCHAR)~(g_MasterMask | (1<<INTR_PIC2)));
    SetPICInterruptMask(FALSE, (UCHAR)~g_SlaveMask);

    __asm {
        sti
        hlt
        cli
    }
    //
    // We're awake! The wake-up ISR has already run.
    //

    // Restore old interrupt masks
    SetPICInterruptMask(TRUE, ucOldMasterMask);
    SetPICInterruptMask(FALSE, ucOldSlaveMask);
    INTERRUPTS_ON();
}

Device Drivers Power Management Functions
All device drivers need to implement power management functions that are called when the system changes its power state. These functions depend on the device driver type.

A stream interface driver must implement the xxx_PowerUp and xxx_PowerDown functions, where xxx is a three-letter prefix that identifies the device driver.
Typically, you implement xxx_PowerDown to save a context, and turn the device off. In xxx_PowerUp, you read the stored context and turn the device on.

Similarly, other driver types have their own power management functions.

**Note**  Refer to the Platform Builder documentation to handle power management in your drivers.

### SetInterruptEvent

A device driver may call this function only when it is in power-off or power-on processing — situations which occur when the device driver's XXX_PowerOn and XXX_PowerOff power handler functions are called by other operating system components. A driver should only call SetInterruptEvent from these functions because the power handler functions are called in a special, nonpre-emptible kernel mode in which most function calls are not permitted. In other contexts, simulated interrupts can be generated with the SetEvent function.

The kernel calls OEMPowerOff when the user presses the OFF button or when GWES enters a time-out period predefined by the user or OEM.
Some Windows CE platforms are open to installation of new software. In this type of device, you may wish to implement a security system on your platform which protects your platform from malicious or poorly written applications. The certification system in Windows CE keeps applications from burdening system resources.
OAL Support for Certification

- Modify OAL to support application certification using Loadauth.lib routines
  - Add OEMCertifyModuleInit and OEMCertifyModule.
  - Initialize the function pointers in OEMInit.
  - Implement OEMLoadInit and OEMLoadModule.
  - Create and export a hard-coded public key. 
    - The structure you implement for this is called g_bSignPublicKeyBlob[].
  - Incorporate the public key into the operating system image.

To implement the signature verification, you can either write the code that is necessary or use the sample verification library, Loadauth.lib.

Loadauth.lib is included in Platform Builder in the processor-specific directory under `Public\Common\Oak\Lib`.

To use Loadauth.lib:

1. Add OEMCertifyModuleInit and OEMCertifyModule to the OAL.
2. Initialize the function pointers in OEMInit. Do this by setting pEMLoadInit = OEMLoadInit and pOEMLoadModule = OEMLoadModule.
3. Implement OEMLoadInit to call CertifyModuleInit which is in LoadAuth.lib.
4. Implement OEMLoadModule. OEMLoadModule returns the following values:
   a. OEM_CERTIFY_TRUST
      - Signifies the application is trusted by the OEM to perform any operation.
   b. OEM_CERTIFY_RUN
      - Signifies the application is trusted by the OEM to run, but is restricted from making certain function calls.
   c. OEM_CERTIFY_FALSE
      - Signifies the application is not trusted by the OEM and therefore is not allowed to run.
5. Create and export a hard-coded public key.
6. The key must have a PUBLICKEYBLOB format. You can use the Signfile.exe tool to complete this step. The structure you implement for this is called g_bSignPublicKeyBlob[].
7. Incorporate the public key into the operating system image.

```c
//
// Initialize the signature verification public key.
//
InitPubKey(g_bSignPublicKeyBlob,sizeof(g_bSignPublicKeyBlob));
```

**Note** For examples, see KERNEL\HAL\KEY1024.C

The Kernel calls **OEMLoadModule** and **OEMLoadInit** before loading a file to perform it verification tasks.

Loadauth.lib library contains the following four functions called by **OEMInit**, **OEMLoadInit** and **OEMLoadModule**:

- **InitPubKey**
  This function initializes a public key to be used for signature verification.

- **CertifyModuleInit**
  This function initiates the process of verifying a signature on a module for the purpose of certifying the module.

- **CertifyModule**
  This function, which is called from OEMCertifyModule, streams the bytes of a module for certification.

- **CertifyModuleFinal**
  This function returns the final certification status of a file and any data embedded in the signature.

**Note** In addition to restricting access to some functions, untrusted modules will also be restricted from writing to certain parts of the registry. The following keys and all their subkeys and values underneath can only be modified by trusted applications. The remainder of the registry can be written to by all applications.

- HKEY_LOCAL_MACHINE\Comm
- HKEY_LOCAL_MACHINE\Drivers
- HKEY_LOCAL_MACHINE\HARDWARE
- HKEY_LOCAL_MACHINE\SYSTEM
- HKEY_LOCAL_MACHINE\init
- HKEY_LOCAL_MACHINE\WDMDrivers
Signfile.exe

**Topic Objective**
Describes how to create a signature.

**Lead-in**
In this section, you will learn how to create a signature.

- **Creating a Signature**

To create a digital signature from a file, run the file through a hash function, and then sign the resulting hash with a private key.

An easy way to create a digital signature from a file is to use Signfile.exe, which is included in Platform Builder. Signfile.exe is a tool for signing an executable file with a private key supplied by a CPU support package (CSP).

Signfile.exe uses the Secure Hashing Algorithm (SHA) to compute the cryptographic hash. SHA generates a 20-byte hash from an arbitrarily sized byte string.

Signfile.exe pads the hash as specified by Public-Key Cryptography Standards #1 (PKCS1) and encrypts it by using the RSA public key algorithm. The key modulus length can be from 512 through 1,024 bits—64 and 128 bytes, respectively. The resulting signature is the same size as the modulus. For example, the signature for a 1,024 bit key is 128 bytes. Signfile.exe then uses the **ImageAddCertificate** and **ImageGetDigestStream** Windows NT functions to embed the signature in a portable executable (PE) file

This tool signs an executable with a supplied private key. You can use the following command line parameters with this tool.

`signfile [ parameters ] Parameters`

The explanation of the command line parameters is as follows:

- **-f** PEFile
  Specifies the file to be signed
- **-a**
  Appends signature to PE File
- **-k** KeyName
  Uses private key from named CryptoAPI key container
- **-p** Cfile
Module 5: OEM Adaptation Layer

Outputs the public key to a file as a C structure

- **-s AttribString**
  Specifies an optional attribute string to be included in signature
  For example, you could add a string to indicate the trust level of the application.

- **-p SignFile**
  Outputs the signature to a file

The following command line example shows how to sign Xyz.dll using the private key in key container TESTKEY1024.

```
Signfile -fXyz.dll -a -kTESTKEY1024
```

Signfile.exe appends the WIN_CERTIFICATE structure to the end of the file and updates the file header accordingly.

---

**Note**  For sample Signfile.exe code, see Public\Common\Oak\Tools\Signfile

---

The trusted module scheme implemented by Windows CE extends to DLLs as well. Since a trusted application might load an untrusted DLL, or a trusted DLL might be loaded by an untrusted application, you must follow a set of simple rules to provide a fail-safe mode. If an untrusted application loads a trusted DLL, the trust level of the DLL is reduced from trust to run. If a trusted application tries to load an untrusted DLL, the load fails. Modules can determine their current trusted state by calling a new function, **CeGetCurrentTrust**. This would be handy for a trusted DLL. If its trust level was reduced to run due to the module that loaded it, the DLL might want to check its trust level in DllMain and refuse to load under that trust level.
Implementing Custom Certification

To implement your own custom certification, you will need to implement the functions **InitPubKey**, **CertifyModuleInit**, **CertifyModule** and **CertifyModuleFinal** are all functions from loadauth.lib. You will need to mimic these functions.

- **InitPubKey**
  This function initializes a public key to be used for signature verification.

- **CertifyModuleInit**
  This function initiates the process of verifying a signature on a module for the purpose of certifying the module.

- **CertifyModule**
  This function, which is called from **OEMCertifyModule**, streams the bytes of a module for certification.

- **CertifyModuleFinal**
  This function returns the final certification status of a file and any data embedded in the signature.
Lab 5: Exploring the OAL

**Topic Objective**
To introduce the lab.

**Lead-in**
In this lab, you will identify different parts of the OAL.

---

After completing this lab, you will be able to:

- Identify the different parts of the OEM Adaptation Layer (OAL).
Review

1. Which kernel function associates a hardware interrupt with an ISR?
   HookInterrupt

2. List the Windows CE boot sequence.
   a. The CPU boot vector jumps to Startup
   b. Startup calls KernelStart
   c. The kernel calls OEMInitDebugSerial
   d. The kernel calls OEMInit
   e. The kernel calls OEMGetExtensionDRAM

3. What are the five power states that Windows CE manages?

4. What are the two OAL functions that implement power management?
   OEMIdle and OEMPowerOff

5. How is the Windows CE kernel built?
   The Windows CE kernel (NK.EXE) is built by linking Microsoft-provided libraries, such as NK.LIB, FULLLIBC.LIB with your OAL (HAL.LIB).
6. Which OAL function does the kernel call when you use the \texttt{OutputDebugString} API?
   \textbf{OEMWriteDebugString}

7. Can you call \texttt{OEMGetRealTime} in an ISR?
   \textbf{No. In an ISR, you do not have access to any stack to call API or kernel functions.}