Developing an OEM Adaptation Layer

8/27/2008

An OEM adaptation layer (OAL) is a layer of code that resides between the Windows Embedded CE kernel and the hardware of your target device. It facilitates communication between your operating system (OS) and your target device and includes code to handle interrupts, timers, generic I/O control codes (IOCTLs), and so on.

In This Section

OEM Adaptation Layer

Provides an overview about an OAL, the kernel libraries, the kernel independent transport layer (KITL), and the event tracking subsystem.

Production-Quality OAL

Contains information about the production-quality OAL available in Windows Embedded CE, which provides an improved level of OAL componentization through code libraries, directory structures, and consistent architecture across processor families and hardware platforms.

How to Develop an OEM Adaptation Layer

Describes the process for creating an OAL that interacts with custom device hardware.

OEM Adaptation Layer Reference

Contains links to topics that list the OAL programming elements.

OEM Adaptation Layer

8/27/2008

An OEM adaptation layer (OAL) is a layer of code that logically resides between the Windows Embedded CE kernel and the hardware of your target device. Physically, the OAL is linked with the kernel libraries to create the kernel executable file.

The OAL facilitates communication between your operating system (OS) and your target device and includes code to handle interrupts, timers, power management, bus abstraction, generic I/O control codes (IOCTLs), and so on.

Creating the OAL is one of the more complex tasks in the process of getting a Windows Embedded CE–based OS to run on a new hardware platform. In general, the easiest way to create an OAL is to copy the OAL implementation from a working OS design, and then modify it to suit the specific requirements of your hardware platform.

If you must create an OAL from the beginning, the task can be greatly simplified by approaching the development process in stages. Each stage adds a little more functionality than the previous one and provides a convenient separation point where new functionality can be fixed and validated before proceeding to the next step.

For more information on how to develop an OAL,

Kernel Image Libraries

8/27/2008

The OAL is physically a part of the kernel image, and as such, is tightly coupled to the kernel build process. The requisite build and configuration directories and files are put in place to create a kernel image from the OAL as it is progressively developed. Microsoft provides the kernel code for the microprocessor of the target device.

The following table shows the requisite libraries that make up the kernel image.

Library	Description
Nk.lib	Base code supplied by Microsoft for a specified microprocessor.
NkProf.lib	Profile version of the microprocessor-specific kernel code that Microsoft supplies with the Platform Builder integrated development environment (IDE). Optional.
KITL.lib	Kernel Independent Transport Layer (KITL) debugging services. Optional if KITL is not required.
FullLibc.lib	Microsoft C Run-Time Library for Windows Embedded CE.

Platform Builder ships OAL code that builds three different kernel libraries. Each is customized for a particular task and is described in the following table.

Kernel type	Description
Kern.exe	Kernel that does not include the debugging subsystem.
Kernkitl.exe	Kernel that includes the KITL subsystem.
Kernkitlprof.exe	Kernel that includes profiling and KITL.

For more information about debugging and profiling, see Diagnostics and Debugging for Mobile and Embedded Development and Tools for Performance Tuning.

The following table shows the libraries necessary to build these versions of the kernel.

Library	Included in Kern.exe	Included in Kernkitl.exe
Nk.lib	X	X
NkProf.lib		
FullLibc.lib	X	X
KITL.lib		X

When the IMGNODEBUGGER environment variable is not set, the kernel debugger is included as part of the kernel image and the kernel debugger runs as soon as the image boots.

The following additional files are added to the kernel image when IMGNODEBUGGER is not set:

- Kd.dll
- Hd.dll
- Osaxst0.dll
- Osaxst1.dll

Kernel Scheduler

8/27/2008

The kernel-scheduling interface defines how the kernel coordinates thread scheduling with an OAL written for a particular hardware platform. The interface consists of several global variables and subroutines. The global variables are defined in the kernel but accessed by the hardware platform OAL. Some of the subroutines are defined in the OAL and some in the kernel; they make up the procedural interface between the kernel and the OAL.

In addition to thread scheduling, the kernel and the OAL coordinate closely on several other time-related areas of functionality. These are closely related to the thread scheduling implementation.

The main areas of the scheduling interface include the following:

- Global timing variables exported by the kernel
- Management of the thread scheduling timer
- Interrupt latency timing
- High-performance counter support
- Monte Carlo profiling support

Thread Scheduling Timers

8/27/2008

The kernel's thread scheduling algorithm is closely related to power management as most devices are very sensitive to power consumption. The kernel calls OEMIdle whenever no threads are ready to run. The job of **OEMIdle** is to put the CPU into a low-power state until one of the following events occurs:

- An interrupt wakes the system.
- A thread is ready to be scheduled.

In practice, many devices spend most of their time waiting for input from the user. As a result, they will spend a great deal of time in **OEMIdle**.

When the system is not idle, threads are using the CPU to do work and **OEMIdle** is not called. Instead, a thread timer goes off periodically to update the system time and invoke the kernel's scheduler. When thread context switches occur, the timer interrupts the currently running thread, the thread context is saved, and another thread's context is restored when the timer ISR returns.

Kernel Global Variables for Scheduling

8/27/2008

The kernel exports a number of global variables that the OAL reads and modifies. The kernel exports the following items:

- CurMSec.

  DWORD counter of the number of milliseconds since system boot. The user can read this variable by calling GetTickCount.

- dwReschedTime.

  DWORD containing the millisecond count of the next time the kernel expects to run the scheduler.

- curridlehigh, curridlelow, and idleconv.

  DWORD variables that implement a 64-bit counter reflecting the number of milliseconds the system has been idle. The user can read this counter by calling GetIdleTime.

The OAL samples also define a number of global variables that are local to the OAL and not accessed by the kernel. These are not part of the kernel-scheduling interface and may not be present on all OAL samples.

Kernel States

8/27/2008

During normal operation, the kernel will make calls to required functions and optional OEM API. This kernel operational state is known as normal state.

The following states are the four other kernel operational states:

- Suspend State
- OEM Power Off State
- Resume State
- OEM Resume State

Suspend State

8/27/2008

When a device is asked to suspend, it is being asked to remain powered to the point that RAM is in a self-refresh state where an interrupt can wake the device. The suspend process can occur in three ways:

- The keyboard driver issues a VK_OFF to GWES. This eventually causes GwesPowerOffSystem to be called.
- The OEM can call **GwesPowerOffSystem** directly.
- The OEM can call **SetSystemPowerState**.

The **GwesPowerOff** function performs key operations before a device can suspend.

To suspend a device

1. Notify the Taskbar that the device is being suspended.

   Post the WM_POWERBROADCAST message with the flag PBT_APMSUSPEND. Only the registered Taskbar will get this message.

2. Abort calibration if the calibration screen is up and in one of the following states:

- Waiting at cross hairs.
- If calibration was waiting at confirmation.

3. Turn off window message queues, stopping the processing of messages.
4. Determine if the Startup UI screen needs to appear on resume.
5. Save video RAM to system RAM is necessary to preserve state on resume.
6. Call **SetSystemPowerState** with the arguments (NULL, POWER_STATE_SUSPEND, POWER_FORCE). This calls into the power manager that coordinates the rest of the suspend operation. At this point, **GwesPowerOff** is not completed until the system resumes.
7. Power manager performs the following actions:
   - Calls FileSystemPowerFunction(FSNOTIFY_POWER_OFF) to power off file system drivers.
   - Calls PowerOffSystem.
   - Calls Sleep(0) to allow the kernel scheduler to run and perform the final suspend process.
8. The kernel performs the following final steps to suspend:
   - Power off GWES process.
   - Power off Filesys.exe.
   - If this is an SHx microprocessor, call OEMFlushCache.
   - Call OEMPowerOff.

OEM Power Off State

8/27/2008

If you have created a device that does not contain GWES or Power Manager, you will have to manage the suspend processes. You must make the following calls:

- FileSystemPowerFunction(FSNOTIFY_POWER_OFF) to power off file system drivers.
- PowerOffSystem.

You should then call Sleep(0) to allow the kernel's scheduler to run and perform the final suspend process.

Resume State

8/27/2008

When a resume occurs the kernel is the first to execute. A device will only resume from a halted state if an interrupt occurs and the CPU has been programmed to wake when an interrupt occurs.

Procedures

To resume a device

1. The kernel performs the following clean-up tasks before resuming normal scheduling:
    - If this is an SHx, it calls OEMFlushCache.
    - Calls InitClock to re-initialize timer hardware to a 1 ms tick.
    - Calls Filesys.exe with a power on notification.
    - Calls GWES with a power on notification.
    - Initializes the KITL interrupt if one is in use.
2. Power manager then calls FileSystemPowerFunction with the (FSNOTIFY_POWER_ON) argument.
3. On the resume, GWES performs the following tasks:
    - Restores video memory from RAM.
    - Powers on the Window Manager.
    - Sets the screen contrast.
    - Shows the Startup UI screen, if required.
    - Posts a message to the Taskbar to tell it the device has resumed by send a WM_POWERBROADCAST message with PBT_APMRESUMESUSPEND parameter.
    - Sends the same message to the User notification subsystem.
    - Triggers applications that have requested to be triggered on resume.

OEM Resume State

8/27/2008

If you have created a device that does not contain GWES or the power manager, you will have to manage the resume process.

You must make a call to FileSystemPowerFunction (FSNOTIFY_POWER_ON) to power on the file system drivers.

Kernel Independent Transport Layer

8/27/2008

The Kernel Independent Transport Layer (KITL) is designed to provide an easy way for you to support any debugging 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 device's communication hardware. The hardware transport layer is then layered under KITL to keep KITL from needing to understand different types of communication hardware. For example, you could create both desktop and Windows Embedded 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.

Note:

KITL over Ethernet or IP4 does not support the IP Security (IPSec) protocol. When a debugging network card is shared with a device such as Vmini, the IPSec connection with the development workstation hangs the KITL connection.

KITL Functionality

- KITL can be loaded on demand on retail devices.
- KITL uses desktop timer packets to renew DHCP addresses of any devices. If KITL is enabled, and a device enters the Suspend state, debug messages are redirected to serial output before the device power is switched off. This prevents a possible halt, because when the device wakes up, any debug messages before the KITL hardware is initialized try to go through KITL hardware.
- Power handlers can be implemented in the KITL. For information about the functions to implement for the KITL Ethernet driver, see the OAL_KITL_ETH_DRIVER structure.
- IOCTL_KITL_GET_INFO can be implemented to retrieve the current KITL hardware information, such as the transport being used and the transport parameters. This help drivers fail gracefully if hardware is being used for KITL.

For more information on KITL, see the following topics:

- Adding KITL Initialization Code
- KITL Transport Communications
- Active and Passive KITL
- Interrupt and Polled Transport

For information about the KITL APIs you can use, see the following topics:

- Required KITL Functions
- Optional KITL Functions
- KITL IOCTLs
- KITLTRANSPORT

KITL Transport Communications

8/27/2008

There are times when the desktop and device KITL transports must communicate information independently of the KITL protocol. To send this information from the device side, call KitlSendRawData. **KitlSendRawData** calls TransportSend in the proper state; you should not call the transport **TransportSend** directly.

Interrupt and Polled Transport

8/27/2008

Interrupt Transport

To set up an interrupt base mechanism, you must pass a SYSINTR_* value to the kernel in the *Interrupt* parameter of the KITLTRANSPORT structure. This structure is filled when the kernel calls OEMKitlInit. When an interrupt base model is in place the kernel calls TransportEnableInt to enable or disable the KITL transport interrupt. The interrupt is enabled when KITL is started, on a system resume, and when the kernel is started.

Polled Transport

To set up a polling base mechanism, you must pass the KITL_SYSINTR_NOINTR value to the kernel in the *Interrupt* parameter of **KITLTRANSPORT**. This structure is filled in by the OEM when the kernel calls **OEMKitlInit**. When a polling based model is in place, the kernel will not call **TransportEnableInt** to enable or disable the KITL transport interrupt. Rather, KITL actively calls TransportRecv in a polling manner to retrieve data from the desktop.

Active and Passive KITL

8/27/2008

The kernel independent transport layer (KITL) contains the following two modes of operation:

- **Active**: The device boots the OAL, calls KitlInit(TRUE) to initialize KITL, and initializes all default KITL clients like the kernel debugger. The KITL service then calls OEMKitlInit and registers all default clients that are used by the system.
- **Passive**: KITL is not automatically started when the device boots. In this mode, KITL initializes itself and the default KITL clients. Call **KitlInit**(FALSE) to initialize in passive mode. **OEMKitlInit** is then called when a client registers with the KITL servers.

You must determine which mode the device should enter on boot. Platform Builder provides a flag called KTS_PASSIVE_MODE that determines which mode is used. You can set KTS_PASSIVE_MODE in Platform Builder by enabling active or passive KITL. For more information, see Enabling Active or Passive KITL.

You must pass the KTS_PASSIVE_MODE flag, if used, to the OAL before **KitlInit** is called.

On the CEPC hardware platform, passing the data through the boot loader in the BootArgs section does this. To view an example, see %_WINCEROOT%\Platform\Cepc\Kernel\Hal\Halkitl.c.

Active KITL is best suited for the development processes so that the debugger can maintain a constant connection. Passive KITL is better suited to a real-world scenario where the debugger is not constantly needed. The benefit of the passive mode is that it allows you to create a device that does not have to be tethered to the desktop tools. The device can even be mobile and if it enters a state where the desktop tools are needed, it initiates a KITL connection between the device and desktop.

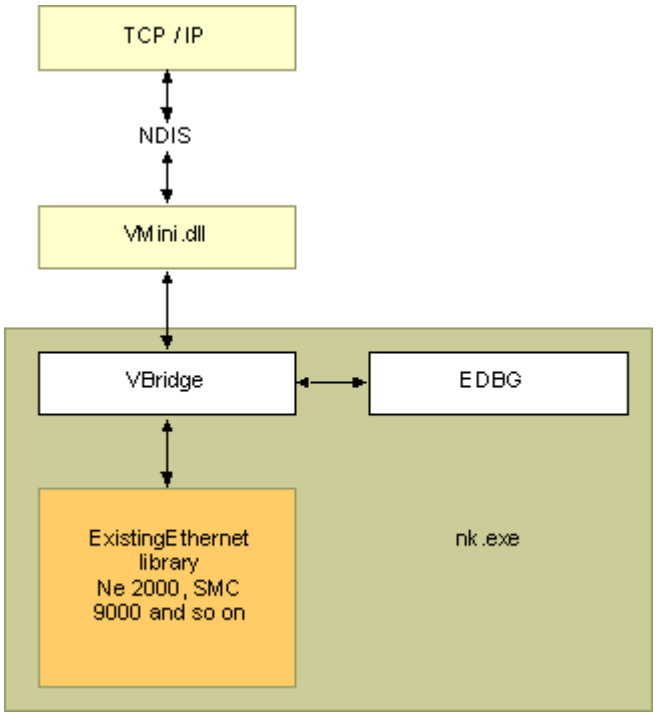
The following code example shows how this is done.

Copy Code

```
BOOL OEMKitlInit (PKITLTRANSPORT pKitl)
{
    BOOL fRet = TRUE;
```

```
    BOOT_ARGS *pBootArgs = (BOOT_ARGS *) ((ULONG)(*(PBYTE
*)BOOT_ARG_PTR_LOCATION) | 0x80000000);

    KITLOutputDebugString ("+OEMKitlInit\n");

    // start the requested transport
    switch(pBootArgs->KitlTransport & ~KTS_PASSIVE_MODE)
    {
        case KTS_SERIAL:
            fRet = InitSerial(pKitl);
            break;

        case KTS_ETHER:
        case KTS_DEFAULT:
        default:
            fRet = InitEther(pKitl);
            break;
    }

    if(!fRet)
        KITLOutputDebugString ("Unable to initialize KITL
Transports!\n");

    KITLOutputDebugString ("-OEMKitlInit\n");
    return fRet;
}

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");
```

```
    }
}
```



Including the VBridge Library in the Run-Time Image

Windows Mobile | Windows Embedded CE

8/27/2008

The VBridge library acts as a MAC layer bridging the kernel's Ethernet debug traffic and the TCP/IP and Winsock application traffic. It should be linked into Nk.exe. The VBridge library communicates with the VMini miniport driver through kernel IOCTLs. For more information about these IOCTLs, see VMINI IOCTLs.

The procedure that follows shows how to include the Vbridge library in the run-time image, but is only required if you are not using the common libraries. In addition to the steps that follow, if you are not using the common library you must also modify functions in kitleth.c that are used to send or receive packets in the KITL network.

Note:

Including Vbridge.lib in Nk.exe will cost around 30 KB of memory to be reserved for VBridge operation.

To include the VBridge library in the run-time image

1. Include VBridge.lib in the sources file in %_WINCEROOT%\Platform\<*Hardware Platform Name*>\Src\KITL.
2. Include Ethdbg.h to the source file containing the OEMIoControl function, and add the following IOCTLs.

Copy Code

```
case IOCTL_VBRIDGE_GET_TX_PACKET:
```

```
  return VBridgeUGetOneTxPacket((PUCHAR *)lpOutBuf,
nInBufSize);

case IOCTL_VBRIDGE_GET_TX_PACKET_COMPLETE:
  VBridgeUGetOneTxPacketComplete((PUCHAR)lpInBuf, nInBufSize);
  return TRUE;

case IOCTL_VBRIDGE_GET_RX_PACKET:
  return VBridgeUGetOneRxPacket((PUCHAR *)lpOutBuf,
lpBytesReturned);

case IOCTL_VBRIDGE_GET_RX_PACKET_COMPLETE:
  VBridgeUGetOneRxPacketComplete((PUCHAR)lpInBuf);
  return TRUE;

case IOCTL_VBRIDGE_GET_ETHERNET_MAC:
  VBridgeUGetEDBGMac((PBYTE)lpOutBuf);
  return TRUE;

case IOCTL_VBRIDGE_CURRENT_PACKET_FILTER:
  // Check whether the filter setting has been implemented and
then
  // inform VBridge of the new filtering.
  if (OEMEthCurrentPacketFilter((PDWORD)lpInBuf))
  {
    VBridgeUCurrentPacketFilter((PDWORD)lpInBuf);
    return TRUE;
  }
  return FALSE;

case IOCTL_VBRIDGE_802_3_MULTICAST_LIST:
  if (OEMEthMulticastList((PUCHAR)lpInBuf, nInBufSize))
    return TRUE;
  return FALSE;

case IOCTL_VBRIDGE_SHARED_ETHERNET:
  // This kernel supports a shared Ethernet port.
  return TRUE;
```

Enabling the VMini Virtual Adapter

8/27/2008

In order to include the VBridge library in the run-time image, the VMini virtual adapter needs to be enabled. For more information about Vbridge.lib, see Including the VBridge Library in the Run-Time Image.

To enable the VMini virtual adapter

1. Include VMini.dll in the run-time image.
2. The VMini virtual adapter is now included in Common.reg as long as the IMGNOKITL and BSP_NOSHAREETH environment variables are not set.

Copy Code

```
IF IMGNOKITL !
IF BSP_NOSHAREETH !
[HKEY_LOCAL_MACHINE\Comm\VMini]
;LOC_FRIENDLYVMINI
    "DisplayName"=mui_sz:"netmui.dll,#9006"
    "Group"="NDIS"
    "ImagePath"="VMini.dll"
    "Priority256"=dword:83

[HKEY_LOCAL_MACHINE\Comm\VMini\Linkage]
    "Route"=multi_sz:"VMINI1"

[HKEY_LOCAL_MACHINE\Comm\VMINI1]
;LOC_FRIENDLYVMINI
    "DisplayName"=mui_sz:"netmui.dll,#9006"
    "Group"="NDIS"
    "ImagePath"="VMini.dll"

[HKEY_LOCAL_MACHINE\Comm\VMINI1\Parms]
    "BusNumber"=dword:0
    "BusType"=dword:0
ENDIF BSP_NOSHAREETH !
ENDIF IMGNOKITL !
```

For multicast, VMini assumes that the adapter has 8 entries for the multicast address list.

Implementation of the Variable Tick Scheduler

8/27/2008

The variable tick scheduler enables OEMs to generate a timer interrupt only when required by the Windows Embedded CE scheduler. In previous versions of Windows Embedded CE, the OEM timerISR is set at every millisecond. If there is no thread that needs scheduling, the timerISR is set to sys-int-nop and goes back to sleep. This means that the system wakes up every millisecond and then goes back to sleep.

In Windows Embedded CE, the kernel calls pOEMUpdateRescheduleTime to set the time before calling OEMIdle. For example, if 100 milliseconds is passed to **pOEMUpdateRescheduleTime**, you are setting the timer tick to wait 100 milliseconds to prevent it from waking up every millisecond.

The variable tick scheduler is backward compatible. If you choose not to use this functionality, the OAL does not require any change.

If you choose to use the Windows Embedded CE variable tick scheduler, you must update a set of timer-related functions. You must also update **pOEMUpdateRescheduleTime** to point to a function that you implemented that reprograms the system timer to interrupt at the time required by the scheduler.

For more information on how to implement the variable tick scheduler, see the following procedure.

Note:

The procedure uses MIPS-specific examples to demonstrate each step and provide more information.

To implement the variable tick scheduler

1. Set up some constants.

   For example, you need to set up these SOC (system-on-chip) required constants: *OEMTimerFreq* and *OEMMinTickDistance*.

2. Implement a function to update reschedule time and to update the function pointer pOEMUpdateRescheduleTime.

   In the sample MIPS and XScale implementations, *NextINTMSec* is assigned as the millisecond value next time a timer interrupt occurs.

   The **pOEMUpdateRescheduleTime** function passes in a millisecond value, which you can evaluate to make sure it is not smaller than 1 millisecond or larger than the maximum milliseconds that the timer counter can handle.

   You can then update the corrected millisecond value to *NextINTMSec* and then update *NextINTMSec* to the timer compare register for the next interrupt.

3. Change the following timer-related functions to handle the variable tick correctly.

   a. SC_GetTickCount must reflect the real system tick, instead of only returning CurMSec.

   b. OEMIdle becomes simpler and only the following functionality needs to be implemented:
   - Call **CPUEnterIdle** routine to put the CPU in power saving mode.
   - Turn off interrupts when returning from the **CPUEnterIdle** function.
   - Calculate the idle time by updating the curridlelow and curridlehigh variables.

   c. InitClock function should be called in OEMInit.

   To initialize the clock correctly, do the following:
   - Assign the following derived constants from **OEMTimerFreq**. You can change the names of these constants depending on your design:
   MIPSCount1MS = *OEMTimerFreq*/1000; // 1 millisecond
   MIPSMaxSchedMSec = *nMaxSchedMSec* = 0x7FFFFFF/*MIPSCount1MS* // Maximum millisecond represented by 32 integer

LastINTMSec = *CurMSec*
NextINTMSec = *LastINTMSec* + *nMaxSchedMSec*;
CurTicksLowPart = *<Current OS timer counter value>*
idleconv = *MIPSCount1MS*;
- Map the following function pointers:
pQueryPerformanceCounter = OEMQueryPerformanceCounter;
pQueryPerformanceFrequency = OEMQueryPerformanceFrequency;
pOEMUpdateRescheduleTime = MIPSUpdateReschedTime;

d. PerfCountFreq has changed to **OEMTimerFreq**.

e. PerfCountSinceTick returns the difference between the current system timer counter value and the counter value at last timer interrupt.

f. **OEMQueryPerformanceCounter** updates the function pointer **pQueryPerformanceCounter**.
Like **PerfCountSinceTick**, the **OEMQueryPerformanceCounter** function also sets the current performance count value.

g. **OEMQueryPerformanceFrequency** updates the function pointer **pQueryPerformanceFrequency**.
The **OEMQueryPerformanceFrequency** function now assigns **OEMTimerFreq** as the performance counter frequency.

Remarks

Implementing the variable tick scheduler is not a simple task. To help with the implementation process, Windows Embedded CE provides a standard implementation in the system-on-chip (SOC) for all CPUs that meet the hardware requirement. These CPUs are SH4, XScale, and MIPS.

You can see a sample implementation for all MIPS, Intel XScale, and SH4 BSPs in Platform Builder. For other CPUs, you can reference the MIPS implementation if you have a count-and-compare system timer, or the SH4 implementation if you have a countdown system timer.

Common variable tick scheduler source code may be found at this location:

%_WINCEROOT%\Platform\Common\Src\Common\Timer\Vartick.

Implementation of the SPI_GETPLATFORMVERSION IOCTL

8/27/2008

SPI_GETPLATFORMVERSION is used to assign a version number to a run-time image.

The IOCTL returns a single version. For example, if you want to check for version 4.2 or later, simply do a >= comparison of the return value and {4,2}.

The version information is necessary because CAB files can now specify that a run-time image only be installed on the target device if the following conditions are true:

- The run-time image name matches the name specified in the CAB file.
- The run-time image version number is in the range specified in the CAB file.

Device Information IOCTL Unification

8/27/2008

In Windows Embedded CE 6.0, IOCTL_HAL_GET_DEVICEID and IOCTL_HAL_GET_UUID are deprecated. They have been functionally replaced by new SPI_* codes in IOCTL_HAL_GET_DEVICE_INFO. Previously, the mechanisms for retrieving device information were spread across multiple IOCTLs.

The examples below illustrate the required changes to transition existing OS components and applications to use the new unified design.

IOCTL_HAL_GET_DEVICE_ID

Deprecated IOCTL:

The deprecated IOCTL example fills the *deviceID* buffer with a Unicode string describing the platform type, i.e. " CEPC" as well as a multibyte string describing the BOOTME name of the device, i.e. " CEPC53902".

Copy Code

```
DWORD deviceID[64];
DWORD dwSize = sizeof(deviceID);
KernelIoControl(IOCTL_HAL_GET_DEVICEID, NULL,
                0, deviceID, dwSize, &dwSize));
PDEVICE_ID pDeviceId = (PDEVICE_ID)(&(deviceID[0]));
```

Unified IOCTL:

The unified IOCTL example fills the *platformName* buffer with a Unicode string describing the platform type, i.e. " CEPC" and fills the *bootmeName* buffer with a Unicode string describing the BOOTME name of the device, i.e. " CEPC53902".

Copy Code

```
WCHAR platformName[32];
DWORD platformNameSize = sizeof(platformName);
UINT32 spiValue;
WCHAR bootmeName[32];
DWORD bootmeNameSize = sizeof(bootmeName);
spiValue = SPI_GETPLATFORMNAME;
KernelIoControl(IOCTL_HAL_GET_DEVICE_INFO, spiValue,
                sizeof(spiValue), platformName,
                platformNameSize, &platformNameSize));
spiValue = SPI_GETBOOTMENAME;
KernelIoControl(IOCTL_HAL_GET_DEVICE_INFO, spiValue,
                sizeof(spiValue), bootmeName,
                bootmeNameSize, &bootmeNameSize));
```

IOCTL_HAL_GET_UUID

Deprecated IOCTL:

The deprecated IOCTL example fills the *uuid* buffer with a platform-unique GUID. If the platform does not support a unique GUID, a potentially non-unique value will be returned.

Copy Code

```
GUID uuid;
DWORD dwSize = sizeof(uuid);
KernelIoControl(IOCTL_HAL_GET_UUID, NULL,
                0, uuid, dwSize, &dwSize));
```

Unified IOCTL:

The unified IOCTL example fills the *uuid* buffer with a platform-unique GUID. If the platform does not support a unique GUID, **KernelIoControl** will return false. If existing code relies on the potentially non-unique value returned by the deprecated IOCTL, you can call IOCTL_HAL_GET_DEVICE_INFO / SPI_GETBOOTMENAME to request (potentially non-unique) data to fill the *uuid* structure. This is similar to the deprecated technique, but without creating the expectation of unique data.

Copy Code

```
GUID uuid;
DWORD dwSize = sizeof(uuid);
UINT32 spiValue = SPI_GETUUID;
KernelIoControl(IOCTL_HAL_GET_DEVICE_INFO, spiValue,
                sizeof(spiValue), &uuid,
                dwSize, &dwSize));
```

IOCTL_HAL_GET_DEVICE INFO / SPI_GETPLATFORMTYPE

Deprecated IOCTL:

The deprecated IOCTL example fills the *platformType* buffer with a Unicode string describing the platform name, i.e. " CEPC".

Copy Code

```
DWORD platformType[32];
DWORD dwSize = sizeof(platformType);
UINT32 spiValue = SPI_GETPLATFORMTYPE;
KernelIoControl(IOCTL_HAL_GET_DEVICE_INFO, spiValue,
                sizeof(spiValue), platformType,
                dwSize, &dwSize));
```

Unified IOCTL:

The unified IOCTL example fills the *platformName* buffer with a Unicode string describing the platform name, i.e. " CEPC".

Copy Code

```
WCHAR platformName[64];
DWORD dwSize = sizeof(platformName);
UINT32 spiValue = SPI_GETPLATFORMNAME;
KernelIoControl(IOCTL_HAL_GET_DEVICE_INFO, spiValue,
                sizeof(spiValue), platformName,
                dwSize, &dwSize));
```

IOCTL_HAL_GET_DEVICE INFO / SPI_GETPLATFORMVERSION

Deprecated IOCTL:

The deprecated IOCTL expected an array of versions returned. For example, by checking for a {4,2} value in the array, it could verify the version as WinCE 4.2 or later.

Unified IOCTL:

The new unified IOCTL returns a single version. For example, if you want to check for version 4.2 or later, simply do a >= comparison of the return value and {4,2}.

How to Develop an OEM Adaptation Layer

8/27/2008

An OEM adaptation layer (OAL) is a layer of code between the Windows Embedded CE kernel and the hardware of your target device.

You develop an OAL to communicate data between your operating system (OS) and your target device and include code to handle interrupts, timers, and so on. For more information about the OAL, see OEM Adaptation Layer.

Some of the startup code for your OAL might be completed if you already implemented a boot loader.

For example, you can reuse parts of the **StartUp** function and debug routines that were created during the boot loader development process.

Conversely, if you have not developed a boot loader, the following steps cover implementation of stand-alone OAL startup code.

For more information about developing a boot loader, see How to Develop a Boot Loader.

Hardware and Software Assumptions

- You developed and built your boot loader code; public headers and libraries have gone through the Sysgen phase.
- You are developing the OAL without assistance from SOC (system-on-chip) OAL code. If SOC libraries and a driver exist for your hardware, use them. The libraries and driver help shorten the time during development and test phases.

Note:

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

The set of procedures presented in following steps provides only a basic kernel with a file system and debug network support, and offers no support for drivers. To develop an OAL, follow the steps in the table below.

Note:

You need to create or modify other files outside of the OAL in order to create a tiny kernel board support package (BSP). For example, you need to modify Config.bib, which contains information about the RAM regions used by the OS.

Step	Topic
1. Create a hardware platform directory and a subdirectory for the OAL on your development workstation. The following example shows the basic naming convention for a hardware platform called MyPlatform.	Not applicable

• Hardware platform directory: %_WINCEROOT%\Platform\MyPlatform • OAL directory: %_WINCEROOT%\Platform\MyPlatform\Src\Kernel\OAL You need to include the directories in the dirs file if you want the directories to build as part of a run-time image. For more information about dirs file, see Dirs File.	
2. Implement the **Startup** function for the OAL. **Note:** In general, **StartUp** initializes the CPU core, including the SDRAM controller, memory management unit (MMU), and caches. The function performs this in preparation for running the Windows Embedded CE kernel. **Note:** Parts of this code can be shared with your hardware platform's boot loader. **Note:** Be careful not to initialize hardware twice in this situation. OAL **StartUp** should function whether or not the boot loader has run. For example, the OAL should not attempt to enter x86-protected mode if the boot loader has already done so. **Note:** You can typically obtain this function from a sample OAL using the same CPU core.	OAL StartUp Function Implementation
3. Create a sources file and a makefile to assemble and compile the file containing the **StartUp** function.	Creating the OAL Sources and Makefile Files
4. Build the **StartUp** source file. This step verifies that the build files are correct and in place. This step creates a Hal.lib binary in your hardware platform's Lib directory.	Building the OAL Source Code
5. Create the Kernel directories and create a dirs file to direct the build process. The Windows Embedded CE kernel, of which the OAL is a part, is an .exe file that is created as part of the BSP build process. There are typically three different kernel variants: basic kernel, kernel with KITL, and a profiling kernel. The focus of this OAL development process is on the kernel with KITL. You must create additional build directories and sources files if you want to create the remaining kernel images. **Note:** If you try to build the Kernkitl.exe image at this stage, you will receive a number of unresolved externals because a number of required functions have not yet been implemented.	Creating the Kernel Directory
6. Create stub versions of the following CPU-specific OAL functions: For ARM-based hardware platforms only:	Creating Stubs for OAL Functions

• OEMARMCacheMode • OEMDataAbortHandler • OEMInterruptHandler • OEMInterruptHandlerFIQ For MIPS-based hardware platforms only: • CacheErrorHandler • **CacheErrorHandler_End** For x86-based hardware platforms only: • OEMNMIHandler For SHx-based hardware platforms only: • Not applicable **Note:** Create the source file with these stub routines in the %_WINCEROOT%\Platform\MyPlatform\Src\Kernel\OAL directory. **Note:** In this step and in the following steps, you will need to create stub versions to resolve link-time dependencies, and thus allow for an OAL and a kernel image to be built. **Note:** You can then add implementation details to each of the functions in an ordered approach. The routines that follow are roughly divided into functional areas. **Note:** These areas may lend themselves to being organized into different source files. **Note:** Any new source files should be added to the sources file to ensure that they are compiled and included in the OAL library.	
7. Create stub versions of the following required OAL functions: • InitClock • OEMInit	Creating Stubs for OAL Functions

• OEMCacheRangeFlush • OEMGetExtensionDRAM	
8. Create stub versions of the following functions: • OEMDebugInit • OEMInitDebugSerial • OEMReadDebugByte • OEMWriteDebugByte • OEMWriteDebugString • **OEMParallelPortGetByte** • **OEMParallelPortSendByte** • **Note:** The **OEMParallelPort*XXX*** functions are obsolete. You do not need to implement them, but stub versions must be created to build a Kernkitl image.	Creating Stubs for OAL Functions
9. Create stub versions of the following functions: • OEMIdle • OEMPowerOff	Creating Stubs for OAL Functions
10. Create stub versions of the following interrupt functions: • OEMInterruptDisable • OEMInterruptDone • OEMInterruptEnable • SC_GetTickCount	Creating Stubs for OAL Functions
11. Create stub versions of the following real-time clock (RTC) functions: • OEMGetRealTime • OEMSetAlarmTime • OEMSetRealTime	Creating Stubs for OAL Functions
12. Create a stub versions of the following function: • OEMIoControl	Creating Stubs for OAL Functions
13. Define the following global variables required by Nk.lib. Additional variable for ARM-based and x86-based hardware	Not applicable

platforms only: • OEMAddressTable Additional variable for MIPS-based hardware platforms only: • OEMTLBSize Additional variables for SHx-based hardware platforms only: • SH4CacheLines	
14. Build the kernel executable image, Kernkitl.exe, when stub versions of all the functions and global variables are defined. **Note:** At this point, the kernel image is not yet very useful, but if you successfully create the resulting Kernkitl.exe image, this establishes the OAL framework and verifies that the build-related files are configured properly.	Building the Kernel Executable Image
15. After creating Kernkitl.exe, you can start adding functionality to the run-time image in stages. For the remainder of the OAL development process, each stage is implemented incrementally. However, because there is no support for the kernel debugger, you must perform debugging and verification at this point using any of the following forms of debugging: a hardware probe or debugger, serial prints, LED blinks, or another form of debugging.	Not applicable
16. Implement the following CPU-specific pre-**OEMInitDebugSerial** functions and global variables: For ARM-based hardware platforms only: • OEMAddressTable • OEMARMCacheMode • OEMDataAbortHandler For MIPS-based hardware platforms only: • OEMTLBSize • CacheErrorHandler • **CacheErrorHandler_End** For x86-based hardware platforms only: • OEMAddressTable • OEMNMIHandler For SHx-based hardware platforms only: • SH4CacheLines	Microprocessor-specific Issues

The code for these functions can be copied from the OAL for a hardware platform that uses the same CPU, or by linking in the SOC library for your CPU.	
17. Implement the OEMCacheRangeFlush function. The code for this function can be copied from the OAL for a hardware platform that uses the same CPU, or by linking in the SOC library for your CPU.	OEMCacheRangeFlush Function Implementation
18. Stop and ensure that the run-time image builds.	Not applicable
19. Create a %_WINCEROOT%\Platform\MyPlatform\Files directory and create the following empty files in the new directory: Platform.bib, Platform.reg, Platform.db, and Platform.dat. Copy and edit a Config.bib file from a similar hardware platform, and then specify the MEMORY and CONFIG section information for your hardware platform. Config.bib should minimally specify the NK and RAM section address information. Romimage.exe uses this address information to know how to organize the OS to fit the hardware resources available on your standard development board (SDB). **Note:** The addresses in Config.bib must be virtual, not physical. ROMOFFSET should be used to ensure that the run-time image downloads to the correct physical RAM or flash memory location. For more information, see the following topics: • MEMORY Section • CONFIG Section • Binary Image Builder File • Romimage	Not applicable
20. Create a %_WINCEROOT%\Platform\MyPlatform\Cesysgen directory and copy the makefile file from a similar hardware platform into that directory. For more information about makefile files, see Makefile File.	Not applicable
21. Ensure that all appropriate hardware platform directories have dirs files. For more information, see Dirs File.	Not applicable
22. From the command line, build an Nk.bin image by entering the following command. Copy Code `blddemo clean -q` The .bin file contains the Windows Embedded CE kernel, and depending on the build flags, it can also contain other executables and support files. For more information about the Windows Embedded CE build tool, see Build Tool.	Not applicable
23. Verify that you can download the Nk.bin image and that you can boot up to at least the **OEMInitDebugSerial** function in the OAL using any of the following debugging tools: hardware debugger or probe, LED write, or serial debug prints. Put `while(1);` in your code to stop the boot.	Not applicable

Note: After **OEMInitDebugSerial** is reached, the CPU is executing in virtual address mode. Do not use physical addresses from this point forward, for example, when writing to the debug serial port or LEDs. **Note:** If **OEMInitDebugSerial** is never reached, you may have incorrectly specified **OEMAddressTable** or are otherwise causing unmapped memory to be accessed, triggering an abort or exception. After completing this step, you have confirmed that the kernel has set up its virtual memory structures or tables and that caches, translation look-aside buffers (TLBs), and write buffers are enabled.	
24. Implement the following serial debug functions. These are the same routines used by the boot loader so you can share the code with the OAL. • OEMInitDebugSerial • OEMReadDebugByte • OEMWriteDebugByte • OEMWriteDebugString **Note:** Sharing these functions between the boot loader and OAL may require minor changes. For more information, see Sharing Code Between the Boot Loader and the OAL.	Enabling the Debug Serial Port Implementing the Serial Debug Functions
25. Rebuild Nk.bin and verify that the boot process proceeds through **OEMInit**. You should also verify that the kernel calls **OEMInitDebugSerial** before it calls **OEMInit** and that serial text is written to the initialized UART.	Not applicable
26. Implement the OEMInit function.	Implementing the OEMInit Function
27. Implement the interrupt-related code. For all hardware platforms: • OEMInterruptEnable • OEMInterruptDisable • OEMInterruptDone For ARM-based hardware platforms only: • OEMInterruptHandler • OEMInterruptHandlerFIQ For non-ARM hardware platforms:	Implementing an ISR

• An interrupt service routine (ISR) that handles the 1-millisecond (ms) system tick interrupt	
28. Implement the power management functions: • OEMIdle • OEMPowerOff • **Note:** Implementation of the **OEMPowerOff** function is highly device-dependent and often requires changes to the boot loader to resume from the suspend state. Implementing suspend and resume is beyond the scope of this how-to topic. You may leave the **OEMPowerOff** function stubbed out.	Enabling Power Management
29. Rebuild the Nk.bin image and verify that it boots successfully. Using a hardware debugger or probe, serial prints, or LEDS, verify that **OEMInit** returns and that **OEMIdle** is eventually called.	Not applicable
30. Add KITL initialization code to **OEMInit**.	Adding KITL Initialization Code
31. Add the Windows Embedded CE Target Control Shell to your run-time image. At the build shell prompt, enter the following command: Copy Code `set SYSGEN_SHELL=1`	Not applicable
32. Rebuild the Nk.bin image and verify that it boots successfully. Ensure that Platform Builder can connect over KITL to the target using the Windows Embedded CE Target Control. Verify that the **gi** [**proc** | **thrd** | **mod** | **delta** | **all**] commands work. After this step, you have the basic kernel in place. The basic kernel has interrupts enabled and uses virtual memory, caches, TLB, and write buffers, which make up most of the tiny kernel or minkern functionality.	Not applicable
33. Implement the following RTC functions: • OEMGetRealTime • OEMSetAlarmTime • OEMSetRealTime	Implementing the Real-Time Clock and System Timer
34. Customize memory usage by doing the following tasks: • Override MainMemoryEndAddress if you want to include additional contiguous memory than is described in Config.bib. • Implement OEMGetExtensionDRAM if you want to include additional noncontiguous memory.	Customizing Memory
35. Implement the OEMIoControl function.	Implementing the OEMIoControl Function

OAL StartUp Function Implementation

8/27/2008

The source file Startup.s, located in %_WINCEROOT%\Platform\<*Hardware Platform Name*>\Src\Kernel\OAL, specifies the startup function of the kernel with the assignment `LEAF_ENTRY StartUp`.

The StartUp function specified is executed either by a jump from the boot loader or through the CPU reset vector.

The **StartUp** function has the following two purposes:

- It initializes the CPU to a known state.

  For more information about initializing the CPU, see CPU Initialization.
- It calls the kernel initialization function, which takes no parameters.

  For more information about initializing the kernel, see Kernel Initialization.

For sample implementations of the **StartUp** and OAL initialization functions, see %_WINCEROOT%\Platform\<*Hardware Platform Name*>\Src\Kernel\OAL.

For information about the implementation details for the **StartUp** function during the boot loader development process, see Implementing the Boot Loader StartUp Function.

CPU Initialization

8/27/2008

This is the OEM hardware initialization code where you set up ROM and DRAM access to get the minimum CPU running.

For sample hardware initialization code, see x86 Kernel.

When the StartUp function is called, the boot loader may already have initialized some of the hardware. OAL **StartUp** must be flexible enough to handle this situation. For example, it may be safe to initialize some hardware twice, while other hardware must not be reinitialized.

During execution of the **StartUp** function, you must initialize the CPU and put the CPU into a known state before calling the kernel initialization function.

The following topics detail processor-specific **StartUp** considerations:

- ARM Kernels
- MIPSII and MIPSIV Kernel
- SHx Kernels
- x86 Kernel

To ensure that the kernel is successful in completing **StartUp**, include some debug support, based on the SDB support, to show the progress through **StartUp** such as with LEDs or serial output.

ARM Kernels

8/27/2008

Before control is transferred to the kernel, the boot loader places the CPU into an initialized state. You may also have to perform BSP/CPU specific initializations as mentioned in the BSP user manual.

The following table shows the ARM-based BSPs and source file locations included in Platform Builder for Windows Embedded CE 6.0 for which the StartUp routine is called to initialize the CPU.

BSP	Source file
Intel PXA27x Processor Development Kit (MainstoneIII)	%_WINCEROOT%\Platform\MainstoneIII\Src\OAL
Texas Instruments SDP2420 Development Board	%_WINCEROOT%\Platform\H4sample\Src\OAL
TI OMAP5912 Aruba Board	%_WINCEROOT%\Platform\Arubaboard\Src\OAL
Device Emulator	%_WINCEROOT%\Platform\Deviceemulator\Src\OAL

OEMs developing for the ARM processor using the ARM kernel are expected to perform the following tasks:

- Put the processor in supervisor mode.
- Disable the interrupt request (IRQ) and fast interrupt request (FIQ) inputs at the CPU.
- Disable the memory management unit (MMU) and both the instruction and data caches.
- Flush or invalidate the instruction and data caches and the translation look-aside buffer (TLB) and empty the write buffers.
- Determine the reason you are in the startup code, such as cold reset, watchdog reset, GPIO reset, and sleep reset.
- Configure the GPIO lines per the requirements of the board. GPIO lines must be enabled for on-board functionality like LED.
- Configure the memory controller, set refresh frequency, and enable clocks. Program data width and memory timing values and power up the banks.
- Configure the interrupt controller. Mask and clear any pending interrupts.
- Initialize the real-time clock count register to 0 and enable the real-time clock.
- Set up the power management/monitoring registers. Set conditions during sleep modes.
- Turn on all board-level clocks and on chip peripheral clocks.
- Get the physical base address of the OEMAddressTable and store in r0.
- Jump to **KernelStart**.

To enable floating-point support, see ARM Vector Floating-Point Unit Support.

OEMs developing for the XScale processor can do the following to improve performance:

XScale performance can be improved by changing the Branch Target Buffer Enable bit. To enable this, have the OAL change the bit in OEMInit. The branch target buffer enable bit is bit 11 of the ARM control register.

The BSP for the Texas Instruments SDP2420 Development Board now includes support for ARMV6 and V7 features. For more information, see ARMV6 and V7 Supported Features.

ARM Vector Floating-Point Unit Support

8/27/2008

The ARM kernels enable support for ARM floating-point units (FPUs). The standard specification for the floating-point unit requires, at a minimum, imprecise handling of floating point exceptions where an error is not detected until a second error occurs.

To fully enable support for the FPU, you must implement the functions shown in the following code example.

Copy Code

```
void OEMRestoreVFPCtrlRegs(LPDWORD lpExtra, int nMaxRegs);
void OEMSaveVFPCtrlRegs(LPDWORD lpExtra, int nMaxRegs);
BOOL OEMHandleVFPException(EXCEPTION_RECORD *er, PCONTEXT pctx);
```

For more information, see OEMRestoreVFPCtrlRegs, OEMSaveVFPCtrlRegs, and OEMHandleVFPException.

The following functions are assigned their appropriate addresses in the OAL and then during OEMInit:

- **pOEMSaveVFPCtrlRegs**
- **pOEMRestoreVFPCtrlRegs**
- **pOEMHandleVFPException**

ARMV6 and V7 Supported Features

8/27/2008

The ARMV6 and V7 architectures have enhanced cache designs which can improve the performance of the Windows Embedded CE 6.0 kernel. The ARMV6 processor is architecturally similar to the ARMV4I. The ARMV6 can run an OS image targeted for the ARMV4I processor. Both architectures supported direct querying of the processor to determine what features are supported.

Supported features include:

- Address Space Identifier (ASID) support

  The primary advantage of using the ARMV6 processor is enhanced caching capabilities through ASID support. ASID is used in a Translation Lookaside Buffer (TLB) to differentiate the same virtual address across different processes.
  ASID improves efficient use of the cache and avoids the need for costly flushing and loading of translation buffers on context switches. To accomplish this cached memory is identified using both the VM address and the ASID. Process level debugging is also enhanced since the ASID is also found in the process ID.
- Branch Target Address Cache (BTAC) flushing enabled

  Flushing the cache is an expensive operation both in terms of CPU performance and in power consumption. BTAC stores recent branch targets in an effort to improve branch prediction.

These features provide a significant saving in software overhead on context switches, avoiding the need to flush on-chip translation buffers in most cases. The result is improved application and operating system performance in battery-powered systems.

Kernel Initialization

8/27/2008

The kernel initialization function is named **KernelInitialize** for x86-based hardware platforms and **KernelStart** on all other hardware platforms.

They are the first non-OEM functions that are called after **StartUp** has completed it tasks.

The kernel initialization function calls additional OAL functions, schedules threads, but must also complete tasks that are specific to each class of CPU.

For processor specific kernel initialization considerations, see the following topics:

- ARM Kernel Initialization
- MIPS Kernel Initialization
- SHx Kernel Initialization
- x86 Kernel Initialization

KernelInitialize and **KernelStart** call **SystemInitialization**, which calls the following functions:

- OEMInit
- OEMInitDebugSerial
- OEMWriteDebugString

The following table shows the naming convention for your kernel's **StartUp** function.

Startup function	Kernel
StartUp	ARM, MIPS, x86

ARM Kernel Initialization

8/27/2008

The following sequence describes how the ARM kernel is initialized by the operating system (OS):

- Initialize first level page tables.
- Enable MMU and cache.
- Enable stacks for each mode.
- Initialize global data for the kernel.
- Perform serial debug functions.
- Call OEMInit.

- Perform memory initializations.
- Perform other initializations.

Kernel Relocation

8/27/2008

During this stage, kernel data is relocated and global data for the kernel is initialized.

If using a debug build, the kernel must be part of the run-time image, otherwise, the following message is sent and the boot is halted:

Copy Code

```
"ERROR: Kernel must be part of ROM image!"
```

Memory Initialization

8/27/2008

The kernel initializes any multiple XIP regions by reading OEMRomChain. The kernel checks RAM to determine if this is a cold boot or warm rest.

For more information, see Multiple XIP Regions

If this is a cold boot, the following actions take place:

- A debug string is sent. The following example code shows the debug string:

  Copy Code

  ```
  "Old or invalid version stamp in kernel structures - starting clean!"
  ```

- The kernel memory structures are set up to indicate a cold boot.

  Filesys.exe will use this information later to determine if the object store is a cold or warm boot.
- Kernel calls OEMGetExtensionDRAM.

  The kernel calls to determine if there is a second, noncontiguous memory section available to the kernel.
- If pNKEnumExtensionDRAM is assigned a function pointer then this function is called, and not **OEMGetExtensionDRAM** to add up to 16 memory sections.
  The kernel can only address 512 MB of memory.
- Calculate the amount of memory that will be allocated to the object store.

  You can calculate the amount of memory by using the information in Config.bib called FSRAMPERCENT.
- pOEMCalcFSPages is called to provide an opportunity to override this value based upon information gathered at boot time.
  Memory is divided between the program space and the object store and the object store can only be allocated a total of 128 MB of memory.

If this is a warm boot, the following events occur:

- The previous memory configuration is used.
- CacheSync(0).
- Zeros memory.
- Zero only the amount of memory specified by the OEM in dwOEMCleanPages.

Completion of Kernel Initialization

8/27/2008

After the kernel has completed initializing the CPU and memory, and initialized or relocated the kernel data, the following steps are performed to complete the kernel initialization process:

1. The kernel initializes the debug subsystem.
2. The kernel initializes the following OAL time functions:
   - KSystemTimeToFileTime
   - KLocalFileTimeToFileTime
   - KFileTimeToSystemTime
   - KCompareFileTime
3. The kernel creates an idle thread to zero the remaining memory. This thread is an idle thread that will run after the system fully boots and comes to a steady state.
4. The kernel creates the alarm thread that is used by the notification system to trigger time-based events.
   During execution of the alarm thread the kernel will call the following functions:
   - OEMGetRealTime
   - OEMSetAlarmTime

   The alarm thread relies on dwNKAlarmResolutionMSec to set the resolution of the real-time clock alarm. By default, this is set to 10 seconds but can be overridden by the OAL in OEMInit.
5. The kernel calls the OEMIoControl function with IOCTL_HAL_POSTINIT.
   This is the last chance for an OEM to perform work before the rest of the OS is started.

System Halt

8/27/2008

In the rare case when the system encounters a non-recoverable error and the only recourse is to halt the system, it posts the debug message "Halting system". At that point, the kernel has effectively locked the system and posted the debug message about the halting condition.

The kernel exports a function pointer that can be overridden by the OEM. If the default function pointer value is replaced by the address of an OEM function in the OAL, when the kernel goes to halt the OS, the OAL will be called. If the OEM does not override the function pointer, the default action is to halt the system.

The kernel declares the following function pointer:

Copy Code
```
extern void (*lpNKHaltSystem)(void);
```
The following code examples show how the OEM can reassign the function pointer in the OAL.

The following code example shows how to implement the **OEMHaltSystem** function.

Copy Code
```
void OEMHaltSystem (void )
{
    //Reset the device.
}
```
The following code example shows how to implement **OEMHaltSystem** in OEMInit.

Copy Code
```
void OEMInit()
{
    extern void (*lpNKHaltSystem)(void);
    lpNKHaltSystem = OEMHaltSystem;
}
```
The lpNKHaltSystem function does not have any return values or take any parameters. The system will not continue to function after the **lpNKHaltSystem** function is called.

Creating the OAL Sources and Makefile Files

8/27/2008

The Windows Embedded CE build process is driven by makefiles with build configuration information such as CDEFINES and include and library file paths, provided by the sources file. You can use a single sources file to build the OAL into a library called Hal.lib, which is then linked into the kernel image.

For more information, see Sources File and Makefile File.

To create the sources and makefile files

1. Create a sources file.

   The sources file must contain the following code. The code shows the macro variables for the sources file.

   Copy Code
   ```
   TARGETNAME=Oal
   TARGETTYPE=LIBRARY
   RELEASETYPE=PLATFORM
   ARM_SOURCES=arm\startup.s
   ```
2. Create a makefile file.

The makefile file must contain the following code:

Copy Code

```
!INCLUDE $(_MAKEENVROOT)\makefile.def
```

Building the OAL Source Code

8/27/2008

When you build OAL source code, you generate a file called Oal.lib. Oal.lib is the name specified in the **TARGETNAME** macro variable of the OAL sources file. For more information, see Creating the OAL Sources and Makefile Files.

To build the OAL source code

1. At the command prompt in the %_WINCEROOT%\Platform\MyPlatform\Src\Kernel\OAL directory, enter the following command:

   Copy Code

   ```
   build -c
   ```

   For more information about the build tool and the parameters you can use, see Build Tool.

2. Verify that %_WINCEROOT%\Platform\MyPlatform\Lib\ARMV4I\Retail\Oal.lib was created.

   If the Oal.lib binary was not created, see the Build.log file, which is written by the build subsystem to the source directory.

Creating the Kernel Directory

8/27/2008

The Kernel directory contains source code files for building and linking the kernel.

To create the Kernel directory

1. Copy an existing hardware platform's %_WINCEROOT%\Platform\<*Hardware Platform Name*>\Src\Kernel directory to your own.

   Make sure dirs files are added through the directories to support build recursion into the tree. For more information about dirs files, see Dirs File.

2. Edit the %_WINCEROOT%\Platform\<*Hardware Platform Name*>\Src\Kernel\Dirs file and comment out the Kern and Kernkitlprof entries using `!if0` and `!endif` tags.

   This will simplify your task because you are only focusing on the KITL-enabled kernel image.

3. Edit the Kernkitl sources file and comment out all library references except for Nk.lib, Hal.lib, and FullLibc.lib.

   You can add the other libraries later when you need them. For more information, see Kernel Image Libraries.

Creating Stubs for OAL Functions

8/27/2008

Stubs are routines that do not contain executable code. They act as placeholders for functions that need to be fully implemented later.

When implementing stubs, you can leave comments that describe what you eventually need to do to add functionality to your function. By scanning existing sample OAL code, you can easily obtain the function header, such as return type and arguments, for each of these functions.

You can either create the stub functions from scratch or copy an existing boot loader or OAL that is of similar CPU type and board design, and then use `#if 0` and `#endif` tags for the function contents.

To create a stub function

1. Obtain the function header definition for the function to be stubbed by referring to a sample OAL that supports similar hardware. The function header includes a return type and a series of function arguments.
2. From the function header definition, create an empty function implementation.
3. If the function header specifies a return type, add a placeholder return value to satisfy the compiler.

   For example, if the function returns a Boolean value, add either `return (TRUE);` or `return (FALSE);` to satisfy the compiler. A successful function call is usually signaled by a TRUE or non-zero value, while failure is usually signaled by a FALSE or 0 value. However, this depends on the function and the data being returned.

The following code example shows the stub for the OEMReadData function:

Copy Code

```
BOOL OEMReadData (DWORD cbData, LPBYTE pbData) { return(TRUE); }
```

Building the Kernel Executable Image

8/27/2008

The kernel, combined with the OAL, is the first service to run, and through the bring-up stages, it runs without the benefit of any sophisticated debug services. Bring the kernel up in stages and verify each stage before proceeding.

To build the kernel executable image

1. From the command prompt, go to the kernel subdirectory: %_WINCEROOT%\Platform\<*Hardware Platform Name*>\Src\Kernel.
2. At the command prompt, enter the following command:

   Copy Code

   ```
   build
   ```

   This generates %_WINCEROOT%\Platform\MyPlatform\Target\ARMV4I\Kernkitl.exe. A Build.log file is also generated in the current working directory, which summarizes the build process. If the build fails, errors are recorded in the build log file.

   For more information about the Windows Embedded CE build tool, see Build Tool.
3. Resolve any build errors by adding the necessary stub function or variable.

For more information about creating stubs, see Creating Stubs for OAL Functions.

ARM Microprocessor

8/27/2008

The following issues should be noted when using an ARM kernel:

- The ARM kernel does not restrict your use of registers.

  In addition, you can use C code as well as assembler code when you are developing your interrupt service routines (ISRs). However, be aware that the ISR should be as small and fast as possible.
- You can specify mapping of physical resources to the statically mapped (direct-mapped) virtual addresses of the kernel in 1-MB blocks.
  In an ARM-specific OAL, create an OEMAddressTable table that defines the mapping from the 4-GB physical address space to the kernel's 512-MB, statically mapped spaces. Each entry in the table consists of the following entries:
  - Virtual base address to which to map
  - Physical base address from which to map
  - Number of MB to map

  Although the order of the values in the table is arbitrary, DRAM should be placed first for optimal performance.
  Because the table is zero-terminated, the last entry must be all zeroes.
  The kernel creates two ranges of virtual addresses from this table:

  - One region, from 0x80000000 to 0x9FFFFFFF, will have caching and buffering enabled.
  - The other region, from 0xA0000000 to 0xBFFFFFFF, will have caching and buffering disabled.
- The OEMInterruptHandlerFIQ function must be part of the OAL for any ARM build to succeed, even if it is just a stub.
- Nested interrupts.

  The ARM CPU architecture supports two interrupts: one hardware interrupt request (IRQ) and one fast interrupt request.
  The OEM must choose a nesting scheme and perform the appropriate masking of the interrupt register in OEMInterruptHandler before re-enabling interrupts. To do so, call INTERRUPTS_ON once the lower priority interrupts have been masked.
  OEMInterruptHandler might call INTERRUPTS_OFF again if it requires non-reentrant code, but this call is not required because the kernel handles it when returning from **OEMInterruptHandler**.
- By default, when a process calls GetSystemInfo, the **dwProcessorType** member returned in the SYSTEM_INFO structure does not return the correct microprocessor type.
  To work around this issue, OEMs should initialize the kernel global variable *CEProcessorType* to either PROCESSOR_ARM720, PROCESSOR_ARM920, or PROCESSOR_ARM7TDMI in OEMInit. The kernel uses the value of *CEProcessorType* to initialize **dwProcessorType**. The

choice to use PROCESSOR_ARM720, PROCESSOR_ARM920, or PROCESSOR_ARM7TDMI depends upon your microprocessor. For more information, see CEProcessorType.

OEMCacheRangeFlush Function Implementation

8/27/2008

The Windows Embedded CE kernel has replaced calls to **FlushDCache**, **FlushICache** and **ClearTLB** with calls to the OEMCacheRangeFlush function.

You must implement the **OEMCacheRangeFlush** function. For sample implementations, see the %_WINCEROOT%\Platform directory.

Some routines reference global variables containing cache and TLB size information. Your hardware platform must resolve these variables. You can further optimize such implementations by hard-coding variables in a private implementation of **OEMCacheRangeFlush** in your hardware platform.

Cache Flush Routines

8/27/2008

Instruction and data caches are flushed based on the CPU architecture and the kernel.

Data Cache Flush

The data cache flush function OEMCacheRangeFlush is called to flush the data cache by determining how many cache lines are available and then flushing each line until all lines have been flushed.

Note:

The old data cache flush function, **FlushDCache**, is deprecated and should never be used.

OEMCacheRangeFlush is called in the following cases:

- CacheSync is called. This is a public API that anyone can call.
- When the kernel loads DLLs or executable files.
- Whenever the kernel debugger is active.

Instruction Cache Flush

The instruction cache flush function **OEMCacheRangeFlush** is called at various times to flush the instruction cache by determining how many cache lines are available and then flushing each line until all lines have been flushed.

Note:

The old instruction cache flush function, **FlushICache**, is deprecated and should never be used.

OEMCacheRangeFlush is called in the following cases:

- **CacheSync** is called with flags CACHE_SYNC_INSTRUCTIONS or CACHE_SYNC_DISCARD. This is a public API that anyone can call.
- When DLLs or executable files are loaded or paged in by the kernel.

- Whenever the kernel debugger is performing active.

> **Note:**
>
> On x86-based hardware platforms, there is only one cache flush routine called OEMFlushCache.

Enabling the Debug Serial Port

8/27/2008

The kernel calls OEMInitDebugSerial after it performs basic initialization operations but before calling OEMInit.

When **OEMInitDebugSerial** returns, the kernel calls OEMWriteDebugString and displays a logon message to your terminal program through the serial port.

If the debug serial port is initialized correctly, you will see terminal output program similar to the following code example.

Copy Code

```
Windows Embedded CE Kernel for MIPS Built on Mar 27 2001 at 17:58:35
```

In addition to these two functions, the following functions add support for the full range of functionality required by the kernel:

- OEMReadDebugByte
- OEMWriteDebugByte
- OEMWriteDebugString
- OEMClearDebugCommError

These functionalities are all based in the OAL and communicate directly with your SDBs serial port.

The kernel and other services that are up and running use serial debug port before KITL is initialized and the debugger is engaged.

The serial port should be set to the following settings to ensure that all SDBs have a consistent approach.

Setting	Value
Baud rate	38,400
Character size	8 bit
Number of stop bits	1
Parity bit	none

Sharing Code Between the Boot Loader and the OAL

8/27/2008

Sharing code can minimize duplication of some routines used by both the boot loader and the OAL.

To share code between the boot loader and the OAL

1. In %_WINCEROOT%\Platform\MyPlatform\Src\Kernel\OAL, create one source file called Debug.c, which contains the code.
2. To share the code with the boot loader, create another file called Boot_debug.c in %_WINCEROOT%\Platform\MyPlatform\Src\BootLoader\Eboot.

   The Boot_debug.c file should contain only the following code:

   Copy Code
   ```
   #include "\\..\\kernel\\oal\\debug.c"
   ```
   Using `#ifdef BOOTLOADER` in the source code, you can conditionally compile the code to use physical memory addresses in the boot loader, or virtual memory addresses in the OAL.

   For example, assuming your peripherals are physically located at 0x40000000 and mapped to 0xB0000000 in **OEMAddressTable**, you would need to add 0x70000000 to the register addresses when running with virtual memory enabled.

   The following code example shows how to do this.

   Copy Code
   ```
   void OEMInitDebugSerial(void)
   {
      #ifdef BOOTLOADER
          UINT32 va_periph_offset = 0x00000000;  // Boot loader
   uses physical addresses.
      #else
          UINT32 va_periph_offset = 0x70000000;  // Kernel startup
   uses virtual addresses.
      #endif

          volatile UART1reg   *s2410UART1    = (UART1reg *)
   va_periph_offset + UART1_BASE;
          volatile IOPreg     *s2410IOP      = (IOPreg *)
   va_periph_offset + IOP_BASE;

      ...
   }
   ```

Implementing the OEMInit Function

8/27/2008

The OEMInit function is responsible for initializing the board-level hardware.

During **OEMInit**, you must:

- Initialize all hardware peripherals needed to support the hardware platform.

- Initiate the debug KITL transport.
- Set kernel variables that are required by the kernel to enable or alter functionality.

For more information about KITL initialization, see Adding KITL Initialization Code.

The **OEMInit** function contains much of the functionality present in the OEMPlatformInit function in the boot loader. For more information, see Implementing the OEMPlatformInit Function.

The following list shows the suggested initialization tasks for the **OEMInit** function.

To implement the OEMInit function

1. Initialize interrupt mapping tables. These are two private OAL tables that map between physical interrupts — interrupt requests (IRQs) — and logical interrupts — SYSINTR values. The following code example shows how to initialize the interrupt mapping tables:

   Copy Code
   ```
   for (i = 0; i <= IRQ_MAXIMUM; i++) {
       IRQToSysIntr[i] = SYSINTR_UNDEFINED;
   }

   for (i = 0; i < SYSINTR_MAXIMUM; i++) {
       SysIntrToIRQ[i] = IRQ_UNDEFINED;
   }
   ```

   Note:

   The interrupt mapping tables, for example, IRQToSysIntr and SysIntrToIRQ, are private to the OAL. OEMs can give these interrupt mapping tables any name. OEMs can use the interrupt mapping tables to implement the functions OEMTranslateIrq and OEMTranslateSysIntr.

2. Create any initial static mappings required by the OAL. For example, SYSINTR_OS_TICK might map to hardware IRQ_OS_TICK, as shown in the following code example:

   Copy Code
   ```
   SysIntrToIRQ[SYSINTR_OS_TICK] = IRQ_OS_TICK;
   IRQToSysIntr[IRQ_OS_TICK] = SYSINTR_OS_TICK;
   ```
   For more information about Windows Embedded CE interrupts, see Interrupts and Defining an Interrupt Identifier.

3. Configure the system timer, real-time clock (RTC), or any other timekeeping device by implementing the InitClock function and calling it from **OEMInit**. For more information, see System Tick Timer Initialization.

4. Configure any CPU-level and board-level interrupt controllers.

5. Provide LED debug support. While optional, it is helpful when debugging the kernel.

6. If necessary for your CPU type, call the HookInterrupt function to register one or more interrupt service routines (ISRs).

   The **HookInterrupt** function associates an ISR with an interrupt request line (IRQ) value. **OEMInit** must register the ISR for the system tick. This is the base functionality required by

the kernel to schedule threads. For more information, see System Tick Timer Implementation.

Note:

HookInterrupt is not required or possible for ARM-based hardware platforms. There are only two interrupts and they are handled by OEMInterruptHandler and OEMInterruptHandlerFIQ.

The following code example shows how you can register the interval-timer ISR, `TimerISR`, for hardware interrupt 5.

Copy Code

```
void OEMInit(void)
{
  ...
  HookInterrupt(5, TimerISR); // Hook timer interrupt
  ...
}
```

7. Mask all unconfigured interrupt sources at the CPU-level and board-level. This prevents interrupts from accidentally being delivered during the kernel initialization.
8. Leave the 1-millisecond (ms) system tick unmasked, so that timekeeping and thread scheduling function properly.

The following list shows optional functionality that you can implement in the **OEMInit** function:

- Logging functions
- Registry functions
- Secure loader functions
- Save and restore coprocessor registers
- High performance counters
- System halt
- Event tracking
- Detection of idle time
- User notification alarm
- Multiple execute-in-place (XIP) regions
- Default thread quantum
- Enable debug IO control
- Erasing the object store
- Supporting CPU utilization functionality
- Object store cold boot
- ARM FPU support
- Zeroing memory

System Tick Timer Initialization

8/27/2008

The system tick timer should be initialized in OEMInit with a call to InitClock, also defined in the OAL.

The following list shows the suggested tasks for the **InitClock** function:

- Determine what timer interrupt source to use for the system tick. The timer interrupt source depends on the CPU and available external sources.
- Initiate the 1-millisecond (ms) tick interval required by the kernel by configuring the timer and unmasking the interrupt.

If the target device contains high-resolution timers, **OEMInit** should also set the pQueryPerformanceCounter and pQueryPerformanceFrequency global variables. For more information, see Supporting High-Resolution Timers.

System Tick Timer Implementation

8/27/2008

You must implement the system tick timer as part of the OAL.

Note:

On ARM-based processors, you do not need to call **HookInterrupt**. All interrupts are serviced by the OEMInterruptHandler function.

The system tick is the only interrupt required by the kernel and you must hook the interrupt with HookInterrupt and then provide a system tick interrupt service routine (ISR) to handle the tick.

When threads are active in the system the system tick timer goes off every millisecond. The system tick ISR updates CurMSec every millisecond so blocking operations and GetTickCount calls are as accurate as possible.

The tick ISR only returns SYSINTR_RESCHED to the kernel's interrupt dispatcher if a thread scheduling event is ready to occur; that is, if dwReschedTime is less than or equal to **CurMSec**.

If no scheduling event is ready, the ISR returns SYSINTR_NOP. SYSINTR_NOP indicates that the interrupted thread will be resumed immediately, without having to wait for the kernel to run its scheduling algorithm.

The system tick ISR does not schedule an interrupt service thread (IST). It returns the appropriate SYSINTR_* value, which causes the kernel to schedule an IST as needed.

To test that the system tick has been initialized and functional, you can write to a LED when a system tick occurs.

Enabling Power Management

8/27/2008

Power management functions respond to system calls for turning the system off or for idling it. These system calls may be triggered by either hardware or software events, such as throwing a power switch or an idle timer count.

The following table shows the functions you must implement to enable power management on your target device.

Function	Description
OEMPowerOff	Places the target device in a power-off state.
OEMIdle	Places the target device in an idle state. This is the lowest energy usage state possible balanced with the need to return from idle quickly.

The kernel calls **OEMIdle** whenever there are no threads ready to run. The kernel provides an **extern DWORD** *dwReschedTime* value that indicates when the next known event is to occur. Such asynchronous events as user-generated interrupt from a keyboard or mouse can occur before then and need to be handled.

The kernel calls **OEMPowerOff** when the user presses the OFF button or when the system requests the device to power off. **OEMPowerOff** is responsible for handling the state of the board-level logic through the suspend and resume process.

Note:

Individual drivers are responsible for handling the board-level logic for individual devices, such as video, audio, and USB.

You can find code samples of **OEMIdle** in %_WINCEROOT %\Platform\Common\Src\Common\Timer\Idle\Idle.c.

When the kernel calls the **OEMIdle** function, 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 if necessary, stopping the clock, and suspending execution.

In order to conserve power while continually awakening the target device, **OEMIdle** should sleep for as long as possible. This is usually until the sooner of two events: dwReschedTime, or the maximum delay supported by the hardware timer.

When an interrupt occurs, scheduled or otherwise, the device ends its idle state, the previous state is restored, and the scheduler is invoked. If no new threads are ready to run, the kernel will again call **OEMIdle**.

When the system returns from idle, **OEMIdle** must update the CurMSec variable with the real number of milliseconds that have elapsed. The sample OAL also keeps partial millisecond counts, *dwPartialCurMSec*, in case another interrupt occurs, which will cause the system to stop idling before the system timer fires.

The following code example shows how to implement **OEMIdle** with these variables.

Copy Code

```
void OEMIdle(DWORD idleParam)
{
    UINT32 baseMSec;
    UINT32 idleTimeMSec;
    INT32 idleSysTicks;
    INT32 usedCounts;
```

```
    INT32 idleCounts;
    ULARGE_INTEGER idle;

    // Get current system timer counter
    baseMSec = CurMSec;

    // Compute the remaining idle time
    idleTimeMSec = dwReschedTime - baseMSec;

    // Idle time has expired - we need to return
    if ((INT32)idleTimeMSec <= 0) return;

    // Limit the maximum idle time to what is supported.
    // Counter size is the limiting parameter.  When kernel
    // profiler or interrupt latency timing is active it is set
    // to one system tick.
    if (idleTimeMSec > g_oalTimer.maxIdleMSec) {
        idleTimeMSec = g_oalTimer.maxIdleMSec;
    }

    // We can wait only full systick
    idleSysTicks = idleTimeMSec/g_oalTimer.msecPerSysTick;

    // This is idle time in hi-res ticks
    idleCounts = idleSysTicks * g_oalTimer.countsPerSysTick;

    // Find how many hi-res ticks was already used
    usedCounts = OALTimerCountsSinceSysTick();

    // Prolong beat period to idle time -- don't do it idle time
isn't
    // longer than one system tick. Even if OALTimerExtendSysTick
function
    // should accept this value it can cause problems if kernel
profiler
    // or interrupt latency timing is active.
    if (idleSysTicks > 1) {
        OALTimerExtendSysTick(idleCounts, g_oalTimer.countsMargin,
0);
    }

    // Move SoC/CPU to idle mode
    OALCPUIdle();
```

```
    // Return system tick period back to original. Don't call when
idle
    // time was one system tick. See comment above.
    if (idleSysTicks > 1) {

        // Return system tick period back to original
        idleSysTicks = OALTimerReduceSysTick(
            g_oalTimer.countsPerSysTick, g_oalTimer.countsMargin
        );

        // Do we need offset counters?
        if (idleSysTicks > 0) {
            // Fix system tick counters
            CurMSec += idleSysTicks * g_oalTimer.msecPerSysTick;
            g_oalTimer.curCounts += idleSysTicks *
g_oalTimer.countsPerSysTick;
        }

        // Get where we are inside tick
        idleCounts = OALTimerCountsSinceSysTick();
    }

    // Update idle time
    idle.LowPart = curridlelow;
    idle.HighPart = curridlehigh;
    idle.QuadPart += idleCounts - usedCounts;
    curridlelow  = idle.LowPart;
    curridlehigh = idle.HighPart;
}
```

Use the previous code example and the system tick ISR to implement the extended idle period.

The following code example is used to determine whether the kernel should schedule a thread (SYSINTR_RESCHED returned) or do nothing (SYSINTR_NOP returned).

Copy Code

```
ULONG ulRet = SYSINTR_NOP;
if ((int)(CurMSec >= dwReschedTime))
       ulRet = SYSINTR_RESCHED;
```

By returning SYSTINTR_NOP when appropriate, a system tick ISR can prevent the kernel from rescheduling unnecessarily and therefore save power.

For more information about the variables the OAL needs to update in order to support **OEMIdle**, see CPU Utilization.

CPU Utilization

8/27/2008

You can calculate CPU utilization for the operating system (OS) by determining how much time the OS spends in the IDLE state over a period of time. The device goes idle when the kernel calls OEMIdle.

The OAL must update the following three global variables to support **OEMIdle**:

- curridlehigh
- curridlelow
- idleconv

Idleconv is the conversion to be applied to the 64-bit time value stored in *curridlehigh* and *curridlelow* to return a millisecond value.

If *curridlehigh* and *curridlelow* are based on milliseconds then *idleconv* is set to 1.

The values of *curridlehigh* and *curridlelow* are always increasing and the increase is based on the amount of time that the device will idle or on the amount of time that the device idled before being wakened. From an application, a call to GetIdleTime will return the OAL values.

Interactions between OEMIdle and the Thread Timer

8/27/2008

The OEMIdle algorithm is complicated by the fact that it must cooperate with the system tick interrupt service routine (ISR) in updating CurMSec.

When an interrupt wakes the CPU from its low power mode during **OEMIdle**, the ISR for the interrupt executes. Once it has executed, control returns to the point in **OEMIdle** at which the CPU was put to sleep.

If a timer interrupt woke the system, the tick ISR has already updated **CurMSec** by one millisecond before control returns to **OEMIdle**. If another interrupt woke the CPU, **CurMSec** has not been incremented.

It is up to **OEMIdle** to make sure that **CurMSec** is properly updated regardless of the wakeup interrupt.

The following factors affect both **OEMIdle** and the system tick ISR:

- Interrupt latency timing support code
- Monte Carlo profiling support code
- Possible support for software-only alarms

Types of System Tick Timers

8/27/2008

In general, hardware platforms provide one of the following types of timers:

- Fixed interval timers that cannot be reprogrammed. These interrupt the system at a fixed rate and cannot be updated once they are started.

CEPC uses this kind of timer.
Fixed interval timers should be avoided in mobile battery-powered systems as there is no way for the kernel to conserve power with OEMIdle.

- Programmable interval timers (PITs) automatically count down to zero, generate an interrupt, and automatically reload and start over. Most SHx-based hardware platforms use this kind of timer.
- Free-running counters with value and compare registers count up or down until the value register equals the match register, at which point they generate an interrupt. Even after the interrupt, the value register keeps counting. To generate another interrupt, the match register must be reprogrammed. XScale and DDB5476 hardware platforms use this kind of timer.

Fixed-rate timers limit the functionality of **OEMIdle** and the thread timer to conserve power. For more information, see Boilerplate Interface Routines.

CPUClearSysTimerIRQ Routine

8/27/2008

This routine is prototyped as shown in the following code example.

Copy Code

```
BOOL CPUClearSysTimerIRQ(void);
```

It takes no parameters but clears the timer interrupt and returns TRUE if the timer interrupt is pending. If the timer interrupt is not pending, it returns FALSE.

Some parts of OEMIdle execute with interrupts disabled and it is possible that the timer will try to interrupt the system during those parts of the code. Hardware platforms that implement Programmable Interval Timers (PITs) need to know whether the interrupt is pending when they update CurMSec.

CPUEnterIdle Routine

8/27/2008

CPUEnterIdle enables interrupts and puts the CPU, or the hardware platform, into a low-power state. When an interrupt occurs it will awaken the system. **CPUEnterIdle** is invoked with the same *dwIdleParam* parameter as OEMIdle.

This routine is prototyped as the following code example indicates.

Copy Code

```
void CPUEnterIdle(DWORD dwIdleParam);
```

If your hardware platform does not support low-power modes, the implementation can be used as shown in the following code example.

Copy Code

```
void CPUEnterIdle(DWORD dwIdleParam)
{
  fInterruptFlag = FALSE;
  INTERRUPTS_ON();
```

```
  while (!fInterruptFlag) {
  // Just wait here. Any interrupt will bump us out.
  }
}
```

Note:

This requires that all interrupt service routines (ISRs) set the variable *fInterruptFlag*, which must be declared as volatile.

CPUGetSysTimerCountElapsed Routine

8/27/2008

OEMIdle calls **CPUGetSysTimerCountElapsed** to determine how much time has elapsed since the last timer tick.

In some situations, **CPUGetSysTimerCountElapsed** is called with interrupts off, making it possible for a timer interrupt to be pending when it is called. This routine must compensate for the interrupt service routine (ISR), which will also update **CurMSec** when its interrupt is unmasked.

Systems that use a PIT should note the return value from **CPUClearSysTimerIRQ** as it indicates whether the PIT interrupt was serviced in time to avoid losing timer information.

This routine is prototyped as shown in the following code example.

Copy Code

```
DWORD CPUGetSysTimerCountElapsed(
   DWORD dwTimerCountdownMSec,
   volatile DWORD *pCurMSec,
   DWORD *pPartialCurMSec,
   volatile ULARGE_INTEGER *pCurTicks
);
```

The following table shows the parameters used in this routine:

Parameter	Description
dwCountdownMSec	The number of milliseconds that the system expected to sleep before the next timer interrupt.
pCurMSec	Points to a millisecond counter, often but not always the global variable CurMSec.
pPartialCurMSec	Some number of timer counts, not timer ticks, not accounted for in the current value of **pCurMSec*.
pCurTicks	Pointer to a 64-bit tick counter.

CPUGetSysTimerCountElapsed determines the number of counter increments or decrements that have elapsed and adds it to **pPartialCurMSec*. It then calculates the number of whole milliseconds the sum represents and updates **pCurMSec* with that value. Finally, it updates **pPartialCurMSec* with any counts left over from this calculation.

CPUGetSysTimerCountElapsed treats **pCurTicks* as a running number of timer counts, reflecting the number of times the counter has been incremented or decremented, not the number of interrupts it has generated or the number of milliseconds elapsed.

The following code example shows an implementation of **CPUGetSysTimerCountElapsed** for an ARM-based hardware platform, using a PIT.

Copy Code

```
DWORD CPUGetSysTimerCountElapsed(
   DWORD dwTimerCountdownMSec,
   volatile DWORD *pCurMSec,
   DWORD *pPartialCurMSec,
   volatile ULARGE_INTEGER *pCurTicks
)
{
    TimerStruct_t  *pTimer = gSysTimers[OS_TIMER].pt;
    DWORD dwTick = dwTimerCountdownMSec * OEMCount1ms;
    DWORD dwCount;

    // If timer IRQ is pending, a full rescheduled period elapsed.
    if (CPUClearSysTimerIRQ( )) {
       *pCurMSec += dwTimerCountdownMSec;
        pCurTicks->QuadPart += dwTick;
        return dwTimerCountdownMSec;
    }
    // If no timer IRQ is pending, calculate how much time has
elapsed.
    dwCount = pTimer->TimerValue;
    if (dwCount > dwTick) {
       // This is an error case. Recover gracefully.
       dwCount = dwTick;
    }
    else {
       dwCount = dwTick - dwCount;
    }
    pCurTicks->QuadPart += dwCount;
    dwCount += *pPartialCurMSec;
    *pPartialCurMSec = dwCount % OEMCount1ms;
    *pCurMSec += (dwCount /= OEMCount1ms);

    return dwCount;
}
```

The following code example shows an implementation for the DDB5476 hardware platform using a free-running timer with value and compare registers.

Copy Code

```
DWORD CPUGetSysTimerCountElapsed(
   DWORD dwTimerCountdownMSec,
   volatile DWORD *pCurMSec,
   DWORD *pPartialCurMSec,
```

```
    volatile ULARGE_INTEGER *pCurTicks
)
{
  DWORD dwTick = dwTimerCountdownMSec * OEMCount1ms;
  DWORD dwCount = R4000Compare( ) - R4000Count( );

  // Note: If dwCount is negative, the counter went past the compare
  // point. The math still works because it accounts
  // for the dwTick time plus the time past the compare point.
  dwCount = dwTick - dwCount;
  pCurTicks->QuadPart += dwCount;
  dwCount += *pPartialCurMSec;
  *pPartialCurMSec = dwCount % OEMCount1ms;
  dwCount /= OEMCount1ms;
  *pCurMSec += dwCount;

  return dwCount;
}
```

CPUGetSysTimerCountMax Routine

8/27/2008

OEMIdle calls **CPUGetSysTimerCountMax** to determine whether it can program the timer interrupt service routine (ISR) for the full interval until the next thread-scheduling event.

If the requested delay, indicated by *dwIdleMSecRequested*, is longer than the underlying hardware can support, **CPUGetSysTimerCountMax** returns the maximum delay possible.

This routine is prototyped as shown in the following code example.

Copy Code

```
DWORD CPUGetSysTimerCountMax(DWORD dwIdleMSecRequested);
```

The following code example requires that you define IDLE_MAX_MS with the maximum millisecond delay that the hardware supports.

Copy Code

```
DWORD CPUGetSysTimerCountMax(DWORD dwIdleMSecRequested)
{
  if (dwIdleMSecRequested > IDLE_MAX_MS) {
  // Our timer cannot idle more than IDLE_MAX_MS milliseconds.
  // The sleep time needs to be broken into reasonable chunks.
  return IDLE_MAX_MS;
  }
  return dwIdleMSecRequested;
}
```

See Also

CPUSetSysTimerCount Routine

8/27/2008

This routine is prototyped as shown in the following code example.

Copy Code

```
void CPUSetSysTimerCount(DWORD dwIdleMSec);
```

The *dwIdleMSec* parameter is the millisecond delay until the next timer interrupt. This delay value is calculated with the help of **CPUGetSysTimerCountMax** so it is guaranteed to be supported by the timer hardware. This routine should program the timer hardware to generate a periodic interrupt with this interval.