6. Operating Systems

6.3 Windows Ce.NET Device Driver Architecture

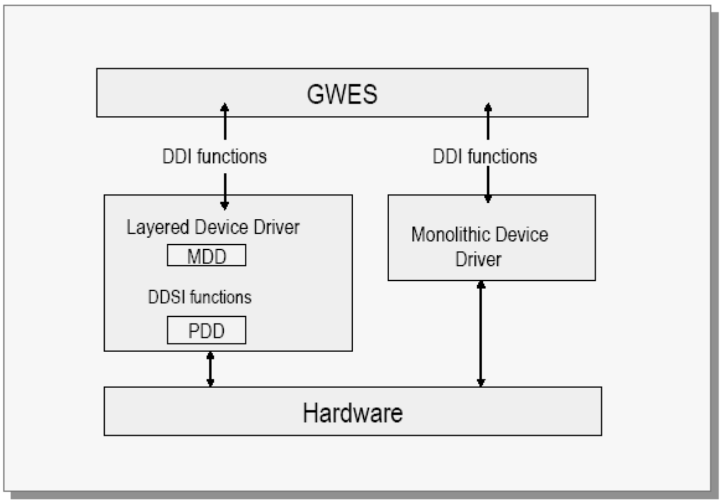
Overview

- Built-In Vs. Installable Drivers
- Device Manager
- ActivateDeviceEx
- Registry Enumerator
- Services
- Bus Drivers
- DMA
- Resource Manager
- Interrupt Model
- Device Driver Power Management
- CETK

Built-In Vs. Installable Drivers

- **Built-in Drivers**
  - Also referred to as native device drivers
  - Loaded in the GWES process space at system boot
  - Generally for devices that are hardwired or must be loaded at system boot up
  - Uses a custom interface
- **Installable Drivers**
  - Also referred to as streams device drivers
  - Dynamically loaded by the Device Manager either at boot or on insertion notification
  - Exist as standalone DLLs
  - Uses the streams interface driver architecture
- **Hybrid Drivers**
  - Expose both a custom-purpose interface and a stream interface

Monolithic Vs. Layered Device Drivers



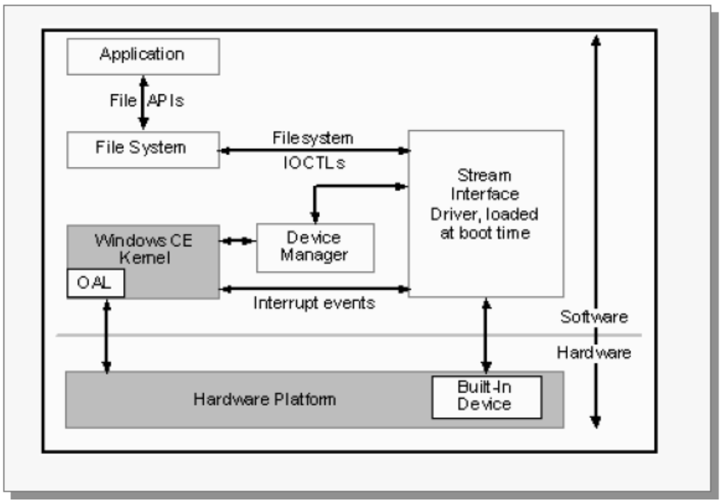
Native Device Drivers

- **Used for built-in devices**
- **Custom interfaces but a standard set of functionality**
- **Statically linked to an executable, while other are DLLs**
- **Sample native device exist for:**
  - Display, Battery, Keyboard, Touch, LED

Streams Driver

- **What is a Stream Driver?**
  - Common interface and functions to all Streams drivers
  - Ideal for I/O devices that are a data source or data sink
  - Interface functions similar to file system APIs—such as ReadFile, IOControl
  - Streams drivers are used to access, from the application level, the physical peripheral device as if it was a file.

Streams Drivers Architecture



Implementing Streams Driver

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

Power Button .def File

```
LIBRARY         PWRBUTTON

EXPORTS
        PWR_Init
        PWR_Deinit
        PWR_Open
        PWR_Close
        PWR_Read
        PWR_Write
        PWR_Seek
        PWR_IOControl
        PWR_PowerDown
        PWR_PowerUp
        PWR_PowerHandler
        PWR_DllEntry
```

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PWRBUTTON]
   "Prefix"="PWR"
   "Dll"="PwrButton.Dll"
   "Order"=dword:2
   "Ioctl"=dword:4
```


Streams Entry Points: Open and Close

- **XXX_Open**

  -Opens a device for reading and/or writing.

  -An application indirectly invokes this function when it calls **CreateFile** to open special device file names.

  -When this function is called, your device should allocate the resources that it needs for each open context and prepare for operation

- **XXX_Close**

  -In response to **CloseHandle**, the operating system invokes this function.

Streams Entry Points: Init and Deinit

- ***XXX*_Init**
  - Called when Device Manager loads the driver
  - Initializes resources that are to be used
  - Memory mapping
- ***XXX*_Deinit**
  - Called when Device Manager unloads the driver
  - Frees allocated resources, stops the IST

Streams Entry Points: Read, Write and Seek

- ***XXX*_Read**
  - Invoked when application calls **ReadFile** function
- ***XXX*_Write**
  - Invoked when application calls **WriteFile** function
- ***XXX*_Seek**
  - Allows moving the current I/O pointer

Streams Entry Points: IOControl

- *XXX*_IOControl
  - Allows performing custom operations that do not necessarily apply to files
  - I/O control code identifies the operation
  - I/O control code is device-specific

Streams Entry Points: PowerUp and PowerDown

- XXX_PowerDown
  - Restores power to a device
- XXX_PowerUp
  - Suspends power to the device
  - Useful only with devices that can be shut off under software control

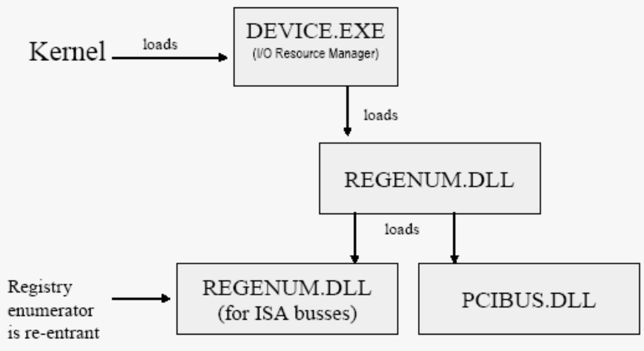
Device Manager

- Device Manager implemented as Device.exe
- User-level process that runs continously
- Not part of the Kernel but launch via Kernel
  - HKEY_LOCAL_MACHINE\Init]
    "Launch20"="Device.exe"
- Separate application that interacts with the kernel, the registry and stream interface driver DLLs'
- Provides ActivateDeviceEx and DeactivateDeviceEx APIs'

Embedded Operating Systems

Device Manager (cont.)

- Device Driver Loading Process



Embedded Operating Systems

ActivateDeviceEx

- **What is ActivateDeviceEx?**
  - A function used by Device.exe to load a device driver
  - A function used by the Registry Enumerator on each subkey it finds (to load driver)
  - ActivateDeviceEx uses the *Dll*, *Prefix*, *Index* and *Flags* fields of registry.
  - Use ActivateDeviceEx to load drivers. You can useActivateDevice, but it simply calls ActivateDeviceEx.

```
ActivateDeviceEx("\\HKEY_LOCAL_MACHINE\\Drivers\\BuiltIn\\PM", …)
```

Embedded Operating Systems

17

Registry Enumerator

- **What is a Registry Enumerator?**
  - Loaded by Device Manager (Device.exe)
  - Finds new devices by reading registry entries
  - Re-entrant
  - Implemented as REGENUM.DLL
- **Code located at WINCE400\public\common\oak\DRIVERS\REGENUM**

Embedded Operating Systems

18

Registry Enumerator (cont.)

- **How does the Registry Enumerator work?**
  - Device.exe loads Registry Enumerator checking HKLM\Drivers\RootKey
  - Init function is called with the HKLM\Drivers\RootKey key
  - Registry Enumerator examines key below HKLM\Drivers\RootKey based on “Order” value
  - Registry Enumerator traverses subkeys of HKLM\Drivers\RootKey and initializes a driver for each entry.

Embedded Operating Systems

Registry Enumerator (cont.)

- **Registry Enumerator Example (Simplified)**

```
[HKLM\Drivers]
    "RootKey"="Drivers"
    "Dll"="RegEnum.dll“

[HKLM\Drivers\Debug]
    "Dll"="RegEnum.dll"
    "Order"=dword:0
    "Flags"=dword:1

[HKLM\Drivers\Debug\EDBG]
        "Flags"=dword:4

[HKLM\Drivers\Virtual]
    "Dll"="RegEnum.dll"
    "Order"=dword:1
    "Flags"=dword:1
```

```
[HKLM\Drivers\Virtual\NDIS]
        "Dll"="NDIS.dll"
        "Order"=dword:1
        "Prefix"="NDS“

[HKLM\Drivers\PCI]
    "Dll"="PCIbus.dll"
    "Order"=dword:4
    "Flags"=dword:1
```

Embedded Operating Systems

Services

- Purpose of a Service
- Services.exe Vs. Device.exe
- Activating / Controlling a Service
- Registering a Service Programmatically
- Stopping a Running Service
- Services.exe at System Startup
- Services API's

Purpose of a Service

- Supplements existing device.exe
- Hosts services that do not require direct access to the system
- Isolates those services from the system services
- Enhances device stability in a service failure and decreases the likelihood of a system crash
- Provides a super service

Service.exe Vs. Device.exe

- **Device.exe loads device drivers that manage devices**
- **Sevice.exe loads device drivers that manages software services**
- **Services.exe is like Device.exe that hosts multiple services.**
- **To use both Device.exe and Services.exe, 2 of the 32 available Windows CE process slots will be used**

Activating / Controlling a Service

- **Activating a service:**
  - Use built-in registry key
  - Use **ActivateService** function
- **Controlling a running service:**
  - Open a handle using **CreateFile**
  - Send an I/O control or **ReadFile**, **WriteFile** and **SetFilePointer** functions

    -- OR
  - Use **GetServiceHandle** function

Services Example

```
ActivateService(L"TELNETD", 0);

HANDLE hService =
CreateFile(L"TEL0:",0,0,NULL,OPEN_EXISTING,0,NULL);
if(hService != INVALID_HANDLE_VALUE)   {
   DWORD dwState; //state values are defined in service.h
   DeviceIoControl(hService, IOCTL_SERVICE_STATUS, NULL, 0,
   &dwState, sizeof(DWORD), NULL, NULL);
   CloseHandle(hService);
}
```


Registering a Service Programmatically

- **Use RegisterService function**
- **RegisterService is analogous to the RegisterDevice function used to start device drivers running under Device.exe**

```
HANDLE hService = RegisterService("TEL",0,“
                                        telnetd.dll",0);
```

Stopping a Running Service

- Use DeregisterService function
- DeregisterService identifies and labels the service as invalid
- DeregisterService disallows any call attempts to CreateFile on a given service handle

```
HANDLE hService = GetServiceHandle(  L"TEL0:", NULL,
                                     NULL);
if (0 != hService)
    DeregisterService(hService);
```


Service.exe at System Startup

- Enumerates through registry subkeys of HKLM\Services

```
[HKLM\Services\TELNETD]
    "Dll"="TELNETD.DLL"
    "Order"=dword:8
    "Keep"=dword:1
    "Prefix"="TEL"
    "Index"=dword:0
    "Context"=dword:1
    "DisplayName"="Telnet Server"
    "Description"="Services incoming telnet requests"
```

Services APIs'

- **Services.exe implements the following functions**
  - XXX_Close
  - XXX_Deinit
  - XXX_Init
  - XXX_IOControl
  - XXX_Read
  - XXX_Seek
  - XXX_Write

Embedded Operating Systems

29

Bus Drivers

- **What is a Bus Driver?**
  - Load drivers for the devices onto their respective buses
- **Examples are:**
  - PCI (PCIBus.dll)
  - PCMCIA (PCMCIA.DLL)
  - IEEE 1394
  - USB

Embedded Operating Systems

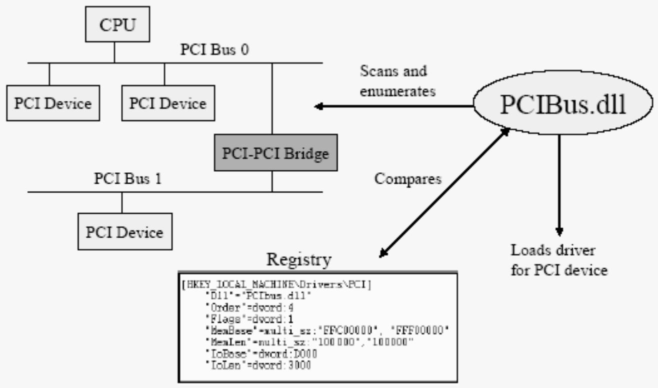
30

PCI Bus Drivers

- PCI Bus Driver enumerates the PCI bus and loads device drivers for any of the devices it finds
- PCI Bus Driver implemented as PCIBus.dll
- Sources available at \WINCE400\public\common\oak\DRIVERS\PCIBUS
- PCIBus.dll is loaded by the registry enumerator
- PCIBus.dll is usually loaded last. So that all of the fixed resources are allocated before the flexible resources of the PCI devices are configured

PCI Bus: Enumerate and Load Device Drivers

- How does PCI Bus enumerate and load device drivers?



Resource Manager

- **What is Resource Manager?**
  - Manages all I/O resources by telling whether resource is available to device driver
  - Uses registry setup to pre-allocate resources
  - Used by bus drivers to request IRQ and I/O space resources when assigning resources to device driver
  - Initial state of Resource Manager is defined in registry
  - Define your own resources using ResourceCreateList, ResourceRelease and ResourceRequest APIs.

Embedded Operating Systems 33

Resource Manager (cont.)

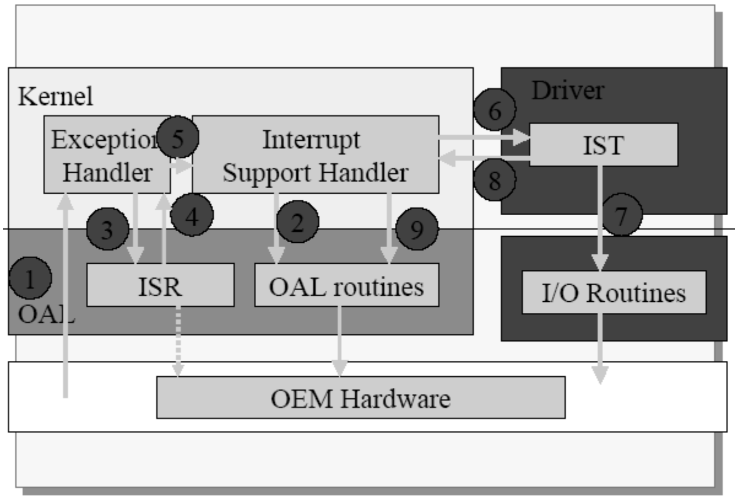
- **Initial State of Resource Manager**

```
[HKEY_LOCAL_MACHINE\Drivers\Resources\IRQ]
   "Identifier"=dword:1
   "Minimum"=dword:1
   "Space"=dword:F
   "Ranges"="1,3-7,9-0xF"
   "Shared"="1,3-7,9-0xF"

[HKEY_LOCAL_MACHINE\Drivers\Resources\IO]
   "Identifier"=dword:2
   "Minimum"=dword:0
   "Space"=dword:10000
   "Ranges"="0-0xFFFF"
```

Embedded Operating Systems 34

Interrupt Model

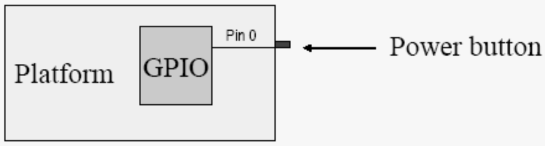


Interrupt Service Thread Priorities

- **Nested Interrupts**
  - Are supported in Windows CE 3.0 and later versions in conjunction with the Real-Time Priority System
  - Interrupts of a higher priority may preempt ISRs of a lower priority
  - Kernel saves and restores the ISR's state when high priority interrupt occurs and completes respectively
  - Level of interrupt nesting is limited solely by what the Windows CE-based platform's hardware can support
- **Interrupt Latencies**
  - Have no upper limit
  - Mostly the latency for servicing interrupts in Windows CE is less than the Windows-based desktop platforms

Implementing Interrupts: Simple Example

- **First step of implementing interrupts is to find out how the interrupt is physically connected to device**
- **Example**



  - Power button is connected to general purpose I/O pin 0
- **Some interrupts are level triggered and some are edge triggered**

Single IRQ Model: OEMInterruptHandler

- **Implement the interrupt service routine**
- **Example**

```
OEMInterruptHandler(unsigned int ra) {
    . . .
    } else if(POWER_OFF_INT_STAT) {
        // mask & clear the interupt
        POWER_OFF_INT_MASK (0);
        POWER_OFF_INT_CLR (1);
        v_pDrvGlob->misc.offButton = 1;
        iSysIntr=  SYSINTR_POWER;
    . . .
    return iSysIntr;
}
```

GPIO pin 0

Return sysintr

- Hardware interrupt happens
- **OEMInterruptHandler** gets called
- If interrupt is via GPIO pin 0, then mask the interrupt and return SYSINTR_POWER

OEMInterruptEnable

- **Performs any hardware operations necessary to allow a device to generate the specified interrupt including**
  - Setting a hardware priority for the device, setting a hardware interrupt enable port, and clearing any pending interrupt conditions from the device

```
BOOL OEMInterruptEnable ( DWORD idInt, .. .) {
    . . .
    BOOL bRet = TRUE;
    switch (idInt) {
    case SYSINTR_POWER:
        POWER_OFF_INT_CLR (1);
        POWER_OFF_INT_MASK (1);
        break;
    . . .
    return bRet;
}
```


OEMInterruptDisable

- **When a device driver is being unloaded and calls the InterruptDisable kernel routine, the kernel in turn calls OEMInterruptDisable**
- **System cannot be preempted when this function is called**
- **OEMInterruptDisable function disables the specified hardware interrupt identified in idInt**
- **Example**

```
BOOL OEMInterruptDisable ( DWORD idInt ) {
    . . .
    switch (idInt) {
    case SYSINTR_POWER:
        POWER_OFF_INT_MASK (0);
        break;
    . . .
    return bRet;
}
```

OEMInterruptDone

- Kernel calls the OEMInterrupt function when a device driver calls InterruptDone
- System cannot be preempted when this function is called
- OEMInterruptDone should re-enable the interrupt if the interrupt was previously masked
- Example

```
BOOL OEMInterruptDone ( DWORD idInt ) {
    . . .
    switch (idInt) {
    case SYSINTR_POWER:
    // the power button is a toggle so
                          let both rising...
        POWER_OFF_RISING_EDGE;
        POWER_OFF_INT_MASK (1);
        break;
    . . .
}
```


Interrupt Service Thread

- Is user-mode thread of device drivers for built-in devices
- Does the actual processing of the interrupt
- Creates an event object associated with the logical interrupt by calling CreateEvent function
- IST remains idle most of the time, awakened when the kernel signals the event object
- IST usually runs at above-normal priority, boost priority with CeSetThreadPriority function

Interrupt Service Thread (cont.)

- **InterruptInitialize**
  - Call InterruptInitialize to link the Event with the Interrupt ID of the ISR
- **WaitForSingleObject**
  - Can be used to wait for an event to be signaled
  - This call is usually inside a loop so that when interrupt is processed, the IST gets back to this call waiting for the next interrupt to be handled
- **InterruptDone**
  - After the interrupt data is processed, the IST must call the **InterruptDone** function to instruct the kernel to enable the hardware interrupt related to this thread

Typical IST Start

```
struct ISTData     // Declare the Strucure to pass to the IST
{
   HANDLE hThread; // IST Handle
   DWORD sysIntr;  // Logical ID
   HANDLE hEvent;  // handle to the event to wait for interrupt
   volatile BOOL abort;     // flag to test to exit the IST
};

ISTData g_KeypadISTData;
// Create event to link to IST
g_KeypadISTData.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
// Translate IRQ to an logical ID (x86 CEPC)
g_KeypadISTData.sysIntr =Mapirq2Sysintr(5);
// start the thread
g_KeypadISTData.hThread = CreateThread(NULL,0,&KeypadIST,
                            &g_KeypadISTData, 0, NULL);
```

Typical IST Start (cont.)

```
//Change the threads priority
CeSetThreadPriority(g_KeypadISTData.hThread,0);

//disconnect any previous event from logical ID
InterruptDisable(g_KeypadISTData.sysIntr);

// Connect Logical ID with Event
InterruptInitialize(g_KeypadISTData.sysIntr, g_KeypadISTData.
hEvent,NULL,0);
```

- **Set the IST Thread Priority**
- **Disconnect any previous events from the associated ISR**
- **Connect to the associated ISR**

Typical IST Start (cont.)

```
DWORD KeypadIST(void *dat)
{
   ISTData* pData= (ISTData*)dat;
   // loop until told to stop
   While(!pData->abort)
   {   // wait for the interrupt event...
       WaitForSingleObject(pData->hEvent, INFINITE)
       if(pData->abort)
         break;
           // Handle the interrupt...
       // Let OS know the interrupt processing is done
       InterruptDone(pData->sysIntr);
   }
   Return 0;
}
```

- **Code a loop that runs until manually aborted**
- **Once in the loop, immediately block until the triggering event is returned from the kernel**
- **Do the actual processing of the interrupt.**
- **Signal InterruptDone in the Kernel with the Interrupt ID**

Typical IST Stop

```
// set abort flag to true to let thread know
// that it should exit
g_KeypadISTData.abort =TRUE;

//disconnect event from logical ID
//this internally sets g_KeypadISTData.sysIntr which in turn
//sets g_KeypadISTData.hEvent through the kernel
InterruptDisable(g_KeypadISTData.sysIntr);

//wait for thread to exit
WaitForSingleObject(g_KeypadISTData.hEvent,INFINITE);

CloseHandle(g_KeypadISTData.hEvent);
CloseHandle(g_KeypadISTData.hThread);
```

- Set a flag that will cancel the IST loop
- Call InterruptDisable to disconnect the triggering event from the logical ID
- Close the Thread Add Reference for the code

Installable ISRs

- Installed by driver
- Handles interrupts for that device
- Need if interrupts are shared
- Installed ISR can be generic routine to check if device is the one requesting service
- Giisr.dll is the generic installable ISR

Implementing Installable ISRs

- Set up the registry for your installable ISR
- Required registry settings are IsrDll and IsrHandler
  - IsrDll is the interrupt service routine DLL name
  - IsrHandler is the ISR function name

```
[HKLM\Drivers\BuiltIn\PCI\Template\WaveDev]
   "Prefix"="WAV"
   "Dll"="es1371.dll"
   "Order"=dword:0
   "Class"=dword:04
   "SubClass"=dword:01
   "ProgIF"=dword:00
   "VendorID"=multi_sz:"1274","1274"
   "DeviceID"=multi_sz:"1371","5880"
   "IsrDll"="giisr.dll"
   "IsrHandler"="ISRHandler"
```


Implementing Installable ISRs

```
GIISR_INFO Info;
m_hIsrHandler = LoadIntChainHandler(isri.szIsrDll,
                    isri.szIsrHandler, (BYTE) isri.dwIrq);
TransBusAddrToStatic(PCIBus, 0, PortAddress,
                          m_dwPciLength, &inIoSpace,
                                 &dwPhysAddr);
Info.SysIntr = m_IntrAudio;
Info.CheckPort = TRUE;
. . .
Info.PortAddr = (DWORD)dwPhysAddr + ES1371_dSTATUS_OFF;
Info.Mask = ES1371_INTSTAT_PENDING;

if (!KernelLibIoControl(m_hIsrHandler, IOCTL_GIISR_INFO,
         &Info, sizeof(Info), NULL, 0, NULL))
    return FALSE;
```