# Windows kernel mode driver development

# OBJECTIVE

- REAL TIME IMPLEMENTATION OF IO PORTS

# CONTENTS

# Hardware Privilege Levels in Windows

1. The user application runs in a special mode of the hardware known generically as ***user mode***.
2. The application require the use of any of these prohibited operations, it must make a request of the operating system kernel.
3. Hardware I/O instructions cannot be executed.
4. A hardware-provided *trap* mechanism is used to make these requests.
5. Operating system code runs in a mode of the hardware known as ***kernel mode***.
6. Kernel-mode code can perform any valid CPU instruction, notably including I/O operations.

## User-mode drivers

- Run in the non-privileged processor mode.
- Cannot gain access to system data except by calling the Win32 API
- Win32 API calls system services
- Multiple layers between hardware and this type of driver
- Overall – Less freedom in all respects
- Easier to debug and develop

## Kernel-mode drivers

- Part of the operating system's executive
- Direct access to system services and data
- Only one layer (HAL) between hardware and this type of driver
- HAL is hardware-dependent
- HAL facilitates hardware-independence and portability

# HAL

- The Hardware Abstraction Layer (HAL) isolates processor and platform dependencies from the OS and device driver code. In general, when device driver code is ported to a new platform, only a recompile is necessary

# How Kernel-Mode Code Executes

- **Trap or Exception Context**

  When a user-mode thread makes a direct request of the I/O Manager, the I/O Manager executes within the context of the requester. In turn, the I/O Manager may call a dispatch routine within a device driver. Dispatch routines of a driver execute within this exception context

- **Interrupt Context**

  When the hardware (or software) generates an acknowledged interrupt, whatever code is executing within the system is stopped dead in its tracks. The executing context is saved and control is promptly handed over to a service routine appropriate for the kind of interrupt that occurred

- **Kernel-Mode Thread Context**

  The final possibility is that a piece of code runs in the context of a separate kernel thread. Drivers spawn separate threads to deal with devices that require polling or to deal with specialized timeout conditions. They execute when scheduled by the kernel's scheduler, in accordance with the assigned thread priority.

# CPU Priority Levels

1. Since different CPU architectures have different ways of handling hardware interrupt priorities, Windows 2000 presents an idealized, abstract scheme to deal with all platforms. The actual implementation of the abstraction utilizes HAL routines that are platform-specific.
2. The basis for this abstract priority scheme is the *interrupt request level* (IRQL). The IRQL is a number that defines a simple priority.
3. Code executing at a given IRQL cannot be interrupted by code at a lower or equal IRQL

# Interrupt Processing Sequence

- When an interrupt reaches the CPU, the processor compares the IRQL value of the requested interrupt with the CPU's current IRQL value.

  If the IRQL of the request is less than or equal to the current IRQL, the request is temporarily ignored. The request remains pending until a later time when the IRQL level drops to a lower value.

- On the other hand, if the IRQL of the request is higher than the CPU's current IRQL, the processor performs the following tasks:
  - Suspends instruction execution.
  - Saves just enough state information on the stack to resume the interrupted code at a later time.
  - Raises the IRQL value of the CPU to match the IRQL of the request, thus preventing lower priority interrupts from occurring.
  - Transfers control to the appropriate interrupt service routine for the requested interrupt

# Deferred Procedure Calls (DPCs)

- While a piece of kernel-mode code is running at an elevated IRQL, nothing executes (on the same CPU) at that or any lower IRQL. Of course, Time-critical event handling could be deferred and cause more disastrous results.

- To avoid these problems, kernel-mode code must be designed to execute as much code as possible at the lowest possible IRQL,this strategy is the *Deferred Procedure Call* (DPC).

# Operation of a DPC

1. The DPC architecture allows a task to be *triggered*, but not *executed*, from a high-level IRQL. This *deferral of execution* is critical when servicing hardware interrupts in a driver because there is no reason to block lower-level IRQL code from executing if a given task can be deferred.
2. When some piece of code running at a high (e.g., hardware) IRQL wants to schedule some of its work at a lower IRQL, it adds a DPC object to the end of the system's DPC dispatching queue and requests a DPC software interrupt. Since the current IRQL is above DISPATCH_LEVEL, the interrupt won't be acknowledged immediately, but instead remains pending.

- Eventually, the processor's IRQL falls below DISPATCH_LEVEL and the previously pended interrupt is serviced by the DPC dispatcher.
- The DPC dispatcher dequeues each DPC object from the system queue and calls the function whose pointer is stored in the object. This function is called while the CPU is at DISPATCH_LEVEL.
- When the DPC queue is empty, the DPC dispatcher dismisses the DISPATCH_LEVEL software interrupt.

# Structure of a Kernel-Mode Driver

A ***kernel-mode*** driver looks very different from conventional applications.

A driver is just a collection of routines that are called by operating system software (usually the I/O Manager). Flow charts don't provide much benefit when diagramming control paths of a device driver. The routines of the driver sit passively until they are invoked by routines of the I/O Manager.

Depending on the driver, the I/O Manager might call a driver routine in any of the following situations:

- **When a driver is loaded**
- **When a driver is unloaded or the system is shutting down**

# DriverEntry ROUTINE

- The **I/O Manager** calls this routine when a driver is first loaded, as early as boot time, but drivers may be dynamically loaded at any time.
- The DriverEntry routine performs all first-time initialization tasks, such as announcing the addresses of all other driver routines.
- It locates hardware that it will control, allocates or confirms hardware resource usage (ports, interrupts, DMA), and provides a device name visible to the rest of the system for each hardware device discovered.
- For WDM drivers participating in Plug and Play, this hardware allocation step is deferred to a later time and routine, the AddDevice function

# REINITIALIZE ROUTINE

- Some drivers may not be able to complete their initialization during DriverEntry. This could occur if the driver was dependent on another driver that had yet to load, or if the driver needed more support from the operating system than had booted at the time DriverEntry was initially called. These kinds of drivers can ask that their initialization be deferred by supplying a Reinitialize routine during DriverEntry

- **SHUTDOWN ROUTINE**

  Surprisingly, the Unload routine is not invoked during a system shutdown. the I/O Manager invokes a driver's Shutdown routine to provide an opportunity to place hardware into a quiescent state.

- **BUGCHECK CALLBACK ROUTINE**

  If a driver needs to gain control in the event of a system crash, it provides a Bugcheck routine. This routine, when properly registered, is called by the kernel during an orderly "crash" process.

# Data Transfer Routines

- Device operations involve a number of different driver routines, depending on the nature and complexity of the device.

- **START I/O ROUTINE**

  The I/O Manager calls the driver's Start I/O routine each time a device should begin the start of a data transfer. The request is generated by the I/O Manager each time an outstanding I/O request completes and another request is waiting in the queue.

  In other words, the I/O Manager (by default) queues and serializes all I/O requests,

- The start I/O routine, supplied by the driver, allocates resources needed to process the requested I/O and sets the device in motion.

# INTERRUPT SERVICE ROUTINE (ISR)

- The Kernel's interrupt dispatcher calls a driver's Interrupt Service routine each time the device generates an interrupt.
- The ISR is responsible for complete servicing of the hardware interrupt.
- By the time the ISR returns, the interrupt should be dismissed from a hardware perspective.
- the most minimal of servicing should be performed within the ISR itself.
- If additional, time-consuming activities are required as a result of servicing the interrupt, a DPC should be scheduled within the ISR. The remaining work of the ISR can then be completed at an IRQL below DIRQL

# DPC ROUTINES

- A driver can supply zero or more DPC routines that perform or complete routine device operations. This might include the release of system resources (such as a DMA page descriptor), reporting error conditions, marking I/O requests as complete, and starting the next device operation, as necessary.
- If only one DPC is required for completion of interrupt servicing, the I/O Manager supports a simplified mechanism called a DpcForIsr routine. However, some drivers may wish to provide many DPCs for varied purposes. Perhaps two different DPCs, one that completes a write operation and one that completes a read operation, is convenient. DPCs can be scheduled simply to perform work at an IRQL of DISPATCH_LEVEL.
- A driver can have any number of custom DPC routines

# UNLOAD ROUTINE

- The I/O Manager calls the Unload routine of a driver when a driver is unloaded dynamically. It must reverse every action that DriverEntry performs, leaving behind no allocated resource. This includes removing the system-wide device name supplied during initialization for all controlled devices.

# SynchCritSection ROUTINES

- Interrupt service occurs at a device-specific DIRQL while remaining driver code operates at DISPATCH_LEVEL or below.
- If the lower IRQL sections of code ever touch resources used by the ISR, that operation must execute inside of a SynchCritSection routine.
- Resources in this category include all device control registers and any other context or state information shared with the Interrupt Service routine.
- SynchCritSection routines operate somewhat differently than other synchronization techniques.
- Once the callback occurs, the IRQL level is raised to the device's DIRQL level. Thus, lower IRQL sections of code temporarily operate at device DIRQL, preventing interruption by the ISR. When the SynchCritSection completes, the IRQL is restored to its original value

# Other Driver Routines

A driver may contain any of the following additional functions.

**Timer routines.**

Drivers that need to keep track of time passage cando so using either an I/O Timer or a CustomTimerDpc routine.

**I/O completion routines.**

A higher-level driver within layers of drivers may be notified when a request sent to a lower-level driver completes. The higher-level driver may register an I/O Completion routine for this purpose.

- **Cancel I/O routines.**

  Drivers must consider the possibility that a device request may be canceled by the requester. This could happen during a long device operation (or because an unanticipated device error leaves the driver in a waiting state). The driver may provide a Cancel I/O routine that is called by the I/O Manager when the requester "gives up."

# I/O Request Packets (IRPs)

- Almost all I/O under Windows 2000 is packet-driven. Each separate I/O transaction is described by a work order that tells the driver what to do and tracks the progress of the request through the I/O subsystem. These work orders take the form of a data structure called an *I/O Request Packet* (IRP).
- With each user-mode request for I/O, the I/O Manager allocates an IRP from nonpaged system memory. Based on the file handle and I/O function requested by the user, the I/O Manager passes the IRP to the appropriate driver dispatch routine

- The Start I/O routine uses the contents of the IRP to begin a device operation.
- When the operation is complete, the driver's DpcForIsr routine stores a final status code in the IRP and returns it to the I/O Manager.
- The I/O Manager uses the information in the IRP to complete the request and send the user the final status

# Driver Objects

- DriverEntry is the only driver routine with an exported name. When the I/O Manager needs to locate other driver functions, it uses the Driver object associated with a specific device. This object is basically a catalog that contains pointers to various driver functions. The life of a driver object is explained below.
- The I/O Manager creates a driver object whenever it loads a driver. If the driver fails during initialization, the I/O Manager deletes the object.
- During initialization, the DriverEntry routine loads pointers to other driver functions into the driver object

- When an IRP is sent to a specific device, the I/O Manager uses the associated driver object to find the right Dispatch routine.
- If a request involves an actual device operation, the I/O Manager uses the driver object to locate the driver's Start I/O routine.
- If the driver is unloaded, the I/O Manager uses the driver object to find an Unload routine. When the Unload routine returns, the I/O Manager deletes the Driver object.

# Device Objects and Device Extensions

- Both the I/O Manager and a driver need to know what's going on with an I/O device at all times. *Device objects* make this possible by keeping information about the device's characteristics and state. There is one device object for each virtual, logical, and physical device on the system. The life cycle of a device object is

1. The DriverEntry routine creates a device object for each of its devices. For WDM drivers, the Device object is created by the AddDevice Plug and Play routine.

2. The I/O Manager uses a back-pointer in the device object to locate the corresponding driver object. There it can find driver routines to operate on I/O requests. It also maintains a queue of current and pending IRPs attached to the device object.
3. Various driver routines use the device object to locate the corresponding device extension. As an I/O request is processed, the driver uses the extension to store any device-specific state information.
4. The driver's Unload routine deletes the device object when the driver is unloaded. The act of deleting the device object also deletes the associated device extension. For WDM drivers, RemoveDevice performs the task of deleting the Device object.

# Device Extensions

- Connected to the device object is another important data structure, the *device extension.* The use of global or static variables violates the requirement that a driver must be fully reentrant. By keeping device state in the device extension, a single copy of driver code can manage multiple devices.

# Interrupt Objects

- Interrupt objects simply give the kernel's interrupt dispatcher a way to find the right service routine when an interrupt occurs.
- The DriverEntry or AddDevice routine creates an interrupt object for each interrupt vector supported by the device or the controller.
- When an interrupt occurs, the kernel's interrupt dispatcher uses the Interrupt object to locate the Interrupt Service routine.
- The Unload or RemoveDevice routine deletes the interrupt object after disabling interrupts from the device.
- A driver does not interact with interrupt objects other than to create and delete them. A pointer to the interrupt object is typically stored in the device extension or controller extension

# Device and Driver Layering

- In the Windows Driver Model, each hardware device has at least two device drivers.

- It’s responsible for initiating I/O operations, for handling the interrupts that occur when those operations finish, and for providing a way for the end user to exercise any control over the device that might be appropriate .

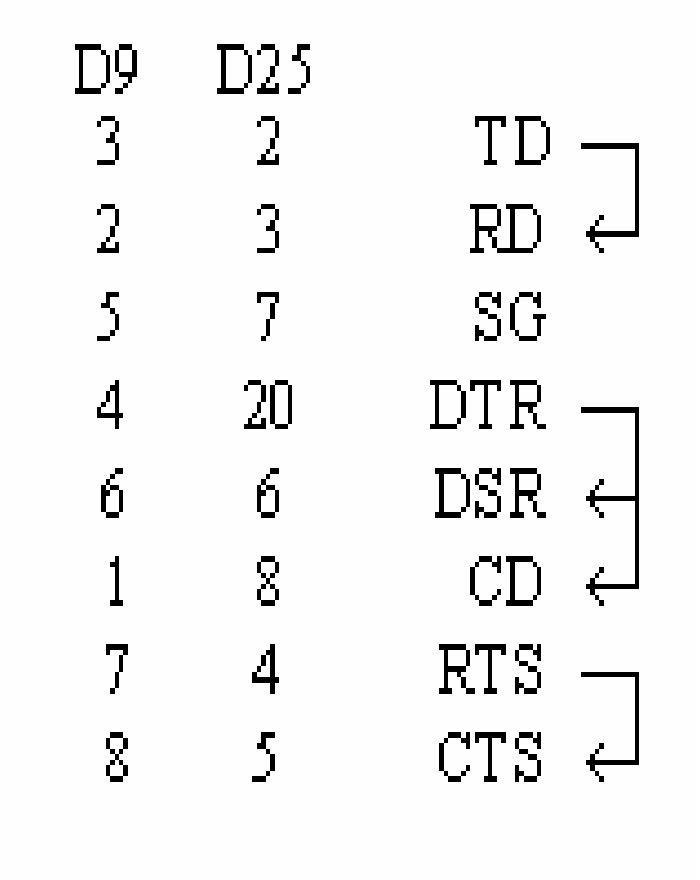
# SERIAL PORT

- **DCE (Data Communications Equipment) and DTE (Data Terminal Equipment.)**
- Data Communications Equipment are devices such as modem, TA adapter, plotter etc while Data Terminal Equipment is a Computer or Terminal.
- It is a D-Type 9 pin connector


**D-sub 9 male and female connectors**

# PIN CONNECTIONS

| D-Type-25 Pin No. | D-Type-9 Pin No. | Abbreviation | Full Name |
|---|---|---|---|
| Pin 2 | Pin 3 | TD | Transmit Data |
| Pin 3 | Pin 2 | RD | Receive Data |
| Pin 4 | Pin 7 | RTS | Request To Send |
| Pin 5 | Pin 8 | CTS | Clear To Send |
| Pin 6 | Pin 6 | DSR | Data Set Ready |
| Pin 7 | Pin 5 | SG | Signal Ground |
| Pin 8 | Pin 1 | CD | Carrier Detect |
| Pin 20 | Pin 4 | DTR | Data Terminal Ready |
| Pin 22 | Pin 9 | RI | Ring Indicator |



# PARALLEL PORT

- PC parallel port is a 25 pin D-shaped connector (Male)

| Pin | Function |
|---|---|
| 2 | D0 |
| 3 | D1 |
| 4 | D2 |
| 5 | D3 |
| 6 | D4 |
| 7 | D5 |
| 8 | D6 |
| 9 | D7 |

**The data output pins (pins 2-9)**

- **Pins 18,19,20,21,22,23,24 and 25 are all ground pins.**
- **loop-back connection for the parallel port any output pin (pin 2-9) is connected with the IRQ pin (pin 10 = ACK).**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | DO | Data Bit 0 | pC, | Data | 0 | N | 2 |
| 3 | D1 | Data Bit 1 | pC2 | Data | 1 | N | 3 |
| 4 | D2 | Data Bit 2 | pC2 | Data | 2 | N | 4 |
| 5 | D3 | Data Bit 3 | pC, | Data | 3 | N | 5 |
| 6 | D4 | Data Bit 4 | pC2 | Data | 4 | N | 6 |
| 7 | D5 | Data Bit 5 | pC | Data | 5 | N | 7 |
| 8 | D6 | Data Bit 6 | pC2 | Data | 6 | N | 8 |
| 9 | D7 | Data Bit 7 | pC2 | Data | 7 | N | 9 |
| 10 | nAck | Acknowledge (may trigger interrupt) | Printer | Status | 6 | N | 10 |
| 11 | Busy | Printer busy | Printer | Status | 7 | Y | 11 |
| 12 | PaperEnd | Paper end, empty (out of paper) | Printer | Status | 5 | N | 12 |
| 13 | Select | Printer selected (on line) | Printer | Status | 4 | N | 13 |
| 14 | nAutoLF | Generate automatic line feeds after carriage returns | pC1 | Control | 1 | Y | 14 |
| 15 | nError (nFault) | Error | Printer | Status | 3 | N | 32 |
| 16 | nInit | Initialize printer (Reset) | pC1 | Control | 2 | N | 31 |
| 17 | nSelectIn | Select printer (Place on line) | pC1 | Control | 3 | Y | 36 |
| 18 | Gnd | Ground return for nStrobe, DO | | | | | 19,20 |
| 19 | God | Ground return for D1, D2 | | | | | 21,22 |
| 20 | God | Ground return for D3, D4 | | | | | 23,24 |
| 21 | God | Ground return for D5, D6 | | | | | 25,26 |
| 22 | Gnd | Ground return for D7, nAck | | | | | 27,28 |
| 23 | Gnd | Ground return for nSelectIn | | | | | 33 |
| 24 | Gnd | Ground return for Busy | | | | | 29 |
| 25 | God | Ground return for nInit | | | | | 30 |
| | Chassis | Chassis ground | | | | | 17 |
| | NC | No connection | | | | | 15,18,34 |
| | NC | Signal ground | | | | | 16 |
| | NC | +5V | Printer | | | | 35 |

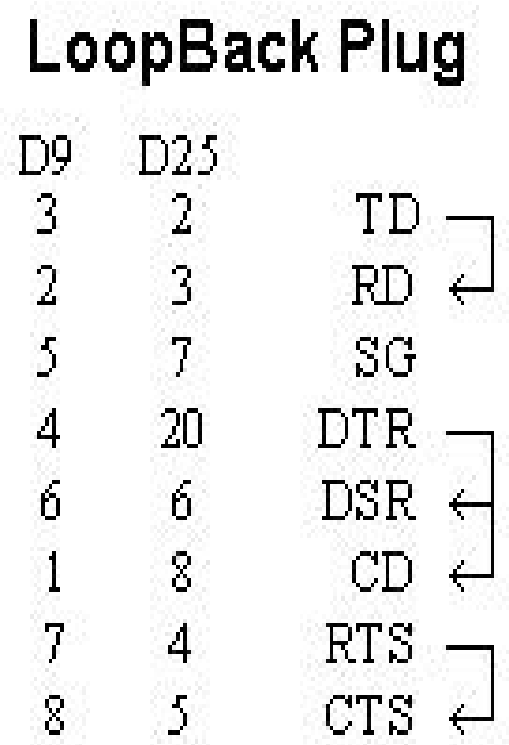
| Data Register (Base Address) | | | | | |
|---|---|---|---|---|---|
| Bit | Pin: D-sub | Signal Name | Source | Inverted at connector? | Pin: Centron-ics |
| 0 | 2 | Data bit 0 | PC | no | 2 |
| 1 | 3 | Data bit 1 | PC | no | 3 |
| 2 | 4 | Data bit 2 | PC | no | 4 |
| 3 | 5 | Data bit 3 | PC | no | 5 |
| 4 | 6 | Data bit 4 | PC | no | 6 |
| 5 | 7 | Data bit 5 | PC | no | 7 |
| 6 | 8 | Data bit 6 | PC | no | 8 |
| 7 | 9 | Data bit 7 | PC | no | 9 |
| Some Data ports are bidirectional. (See Control register, bit 5 below.) | | | | | |

| Status Register (Base Address +1) | | | | | |
|---|---|---|---|---|---|
| Bit | Pin: D-sub | Signal Name | Source | Inverted at connector? | Pin: Centron-ics |
| 3 | 15 | nError (nFault) | Peripheral | no | 32 |
| 4 | 13 | Select | Peripheral | no | 13 |
| 5 | 12 | PaperEnd | Peripheral | no | 12 |
| 6 | 10 | nAck | Peripheral | no | 10 |
| 7 | 11 | Busy | Peripheral | yes | 11 |
| Additional bits not available at the connector:<br>0: may indicate timeout (1=timeout).<br>1, 2: unused. | | | | | |

| Control Register (Base Address +2) | | | | | |
|---|---|---|---|---|---|
| Bit | Pin: D-sub | Signal Name | Source | Inverted at connector? | Pin: Centron-ics |
| 0 | 1 | nStrobe | PC[1] | yes | 1 |
| 1 | 14 | nAutoLF | PC[1] | yes | 14 |
| 2 | 16 | nInit | PC[1] | no | 31 |
| 3 | 17 | nSelectIn | PC[1] | yes | 36 |
| [1]When high, PC can read external input (SPP only).<br>Additional bits not available at the connector:<br>4: Interrupt enable. 1=IRQs pass from *nAck* to system's interrupt controller. 0=IRQs do not pass to interrupt controller.<br>5: Direction control for bidirectional Data ports. 0=outputs enabled. 1 =outputs disabled; Data port can read external logic voltages.<br>6,7: unused | | | | | |

# Null-modem loop-back connection

- **Parallel Port**
  - **Short one pin from 2-9 to 10 (ACK).**
- **Serial Port**
  - **Short DTR, DSR, and CD**
  - **Short RTS and CTS**
  - **Short TXD to RXD**



# TOOLS USED

- WINDOWS DDK
- VC++ 6.0
- Windows Debugger
- Driver Control

# SWOT Analysis

| | |
|---|---|
| **Strengths** | Working hours not limited |
| **Weaknesses** | Development on Windows platform |
| **Opportunities** | Developing kernel mode drivers in a Windows environment for I/O ports |
| **Threats** | Development of drivers for USB port within the given time constraint<br><br>Possibility of crashing; control with Windows Xp |

# CONCLUSION

- We successfully developed Kernel Mode Drivers for Serial and Parallel port. The kernel driver for USB is under development.

- **THANK YOU**