WinHEC 96

CIRRUS LOGIC®

Electronic Engineering TIMES

Microsoft®

COMPUTER & COMMUNICATIONS OEM MAGAZINE

Win32® Driver Model Seminar

Steve Timm
Senior Technical Evangelist
Microsoft Corporation

Device Drivers
New challenges and opportunities

- New bus support, more devices
- Multifunction devices
- Common driver model
  - Windows NT® and future versions of Windows®
- Reduced latency
- Lower development cost

New Development
Win32 Driver Model

- Core architecture evolution for SIPC
  - Extensible for enhanced connectivity
    - New device and bus support
- Based on Windows NT I/O subsystem
  - Source/(x86) binary-compatible drivers across Windows and Windows NT
- Driver structure simplifies development
  - Reusable driver modules
    - E.g., device class x on random busy

Win32 Driver Model

Backwards compatibility

- Initial targets are new device and bus support
- The Win32 model coexists with existing class-specific driver models
  - E.g., mass-storage and networking
- Windows Virtualization Drivers can virtualize legacy hardware interfaces
  - Send class-specific commands to the appropriate Win32 class driver

Win32 Driver Model
Why WDM is important to you

- Common I/O services
- Source/(x86) binary-compatible drivers across Windows and Windows NT
- Reduced latency
- Higher-quality drivers
- Lower development cost
- Hardware innovation
- Easy, new bus support

Agenda

Win32 Driver Model seminar

- Windows NT, Win32 driver architecture
- Future developments in the Win32 Driver Model
- Win32 Driver Model
- Questions and answers

WinHEC 96

CIRRUS LOGIC®

Electronic Engineering TIMES

Microsoft®

COMPUTER & COMMUNICATIONS OEM MAGAZINE

Windows NT®, Win32®
Driver Architecture

Bob Rinne
Software Design Engineer
Windows NT Development
Microsoft Corporation

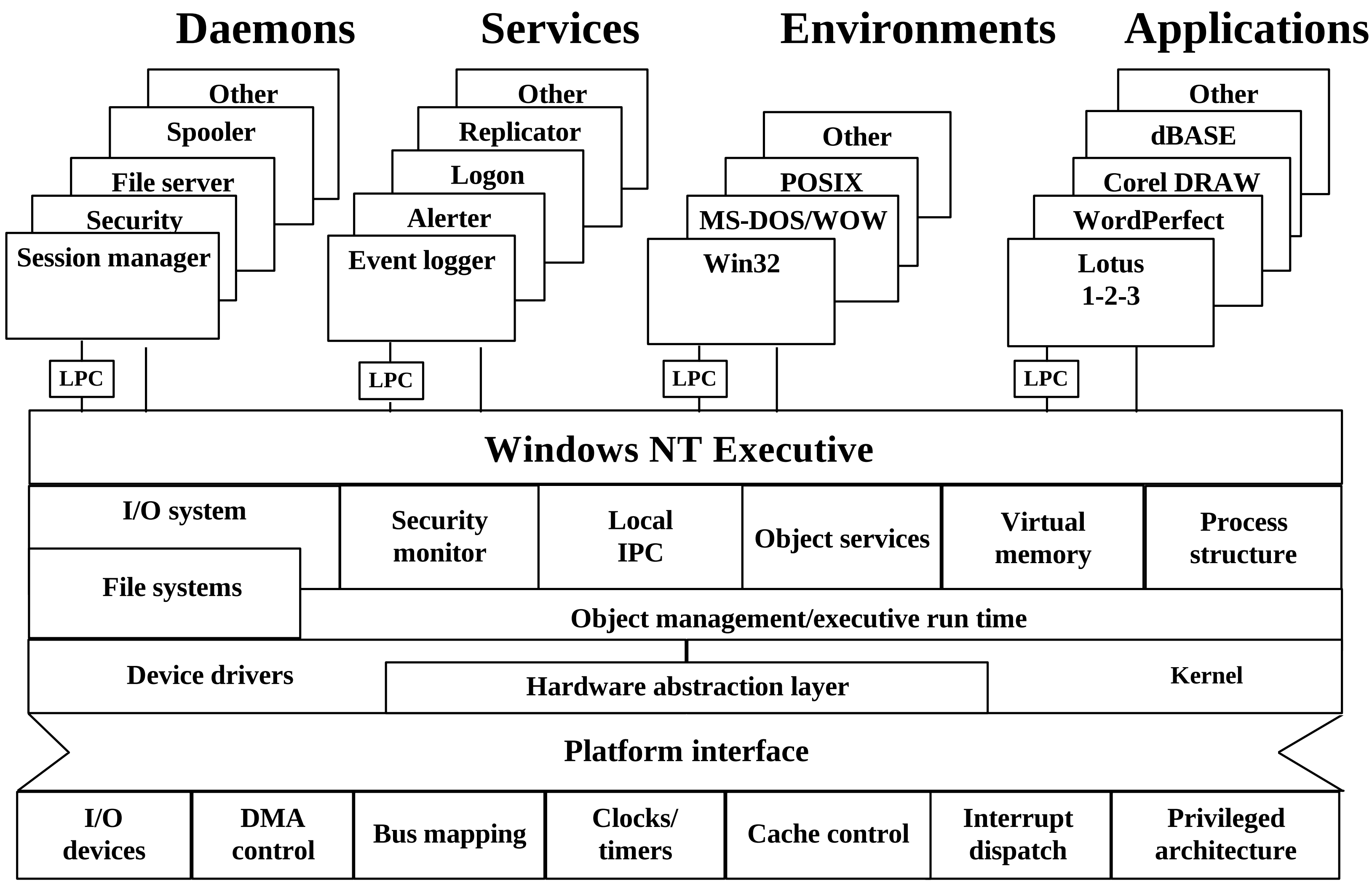
Introduction

ws NT 4.0 kernel

ws NT 4.0 device driver

Driver Model

Windows NT System Structure



Windows NT Kernel Architecture

- Small, well-contained body of code that implements:
  - Scheduling and context switching
  - MP synchronization
  - Exception and interrupt handling
  - Low-level hardware functions

Windows NT Kernel Architecture

- Nonpageable, nonpreemptable, but interruptible
- Allows for pageable system components
- Exports abstractions in the form of:
  - Dispatcher objects
  - Control objects
- Provides generic wait operations

Dispatcher Objects

- Control scheduling and synchronization
- Have “signal” state and are waitable
- Dispatch objects
  - Threads
  - Mutual exclusion
  - Event
  - Semaphore
  - Timer
  - Event pairs

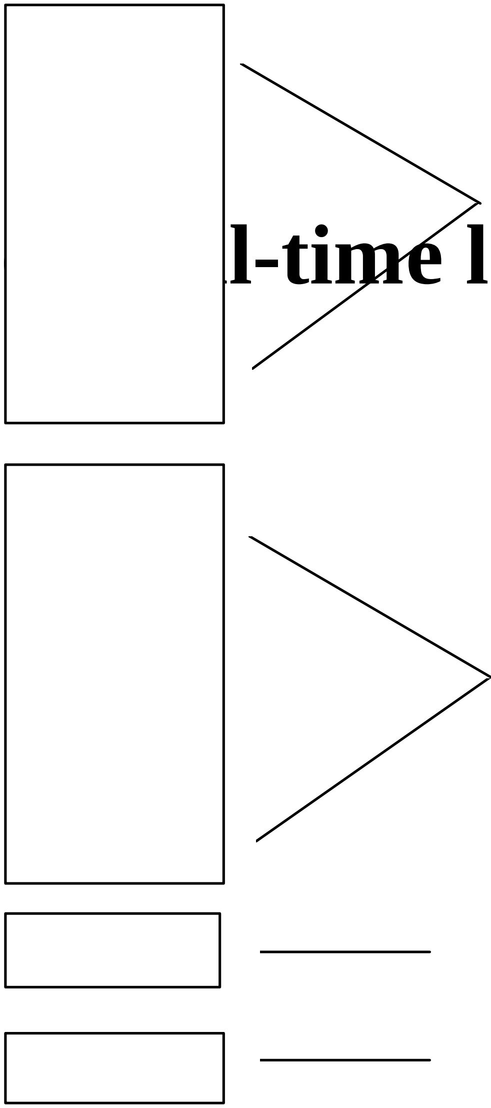
Control Objects

- Provide executive and device-driver control
- No “signal” state and not waitable
- Control objects
  - Process
  - Interrupt
  - Device queue
  - Asynchronous procedure call (APC)
  - Deferred procedure call (DPC)

Threads

- Execution agents
- Register context
- Process address space
- Priority/affinity
- Scheduling state

Priority Levels



Sixte [illegible] l-time levels

Fifteen variable levels

One system level

One idle thread level

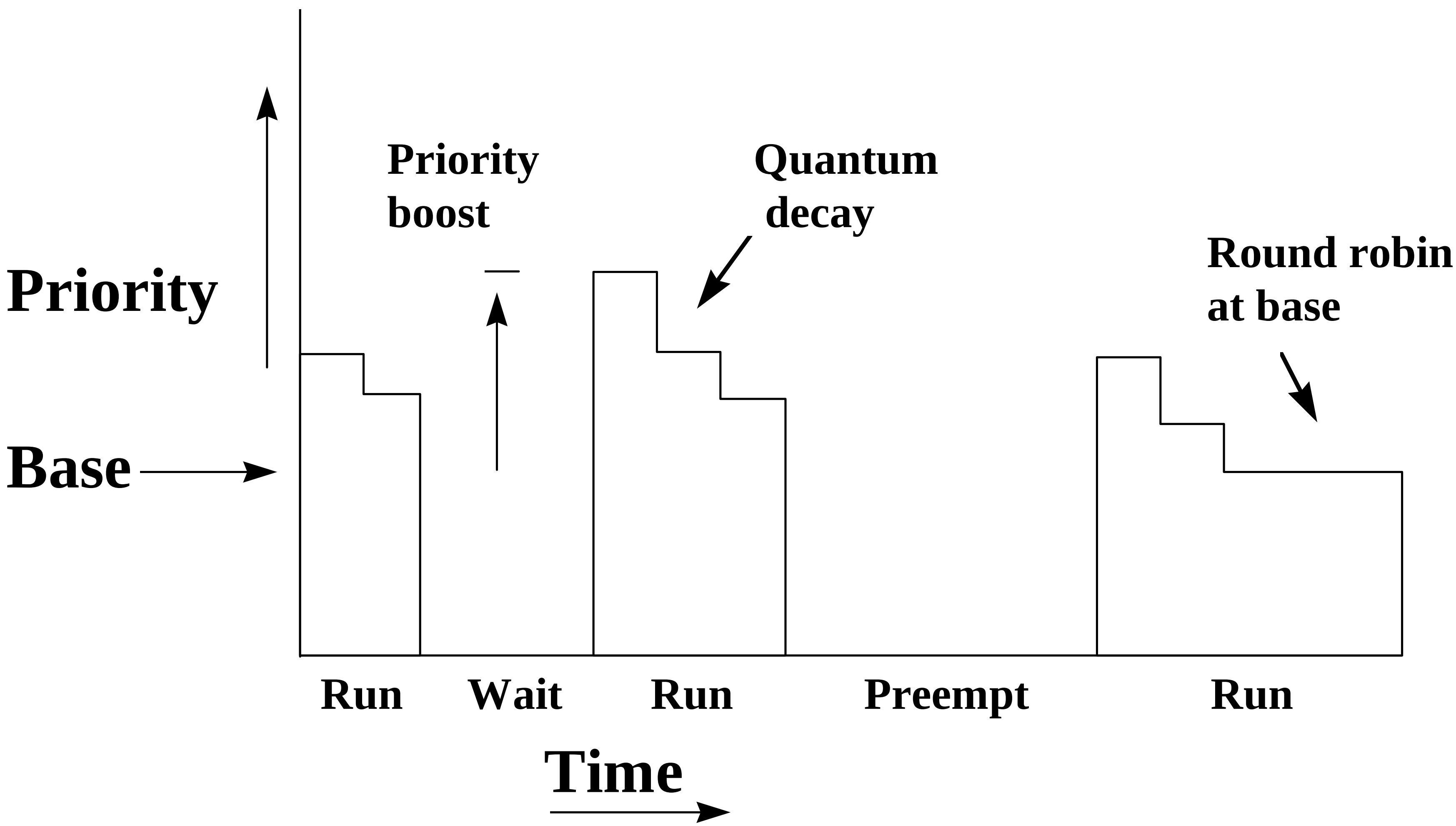
Scheduling

- Event-driven - no scheduler per se
- Preemptive priority policy
- Round robin at real-time levels
- Priority boosts/decay at variable levels
- Highest priority thread guaranteed running

Waiting

- Wait for object to attain signal state
- Wait for single object
- Wait for multiple objects - any/all
- Optional time-out
- Express client/server event pair

Thread “Life”



Priority
Base
Priority
boost
Quantum
decay
Round robin
at base
Run
Wait
Run
Preempt
Run
Time

APC Object

- Breaks into the execution of a target thread
- Interrupts user mode when alertable
- Interrupts kernel mode when enabled
- Used to post asynchronous events to a thread

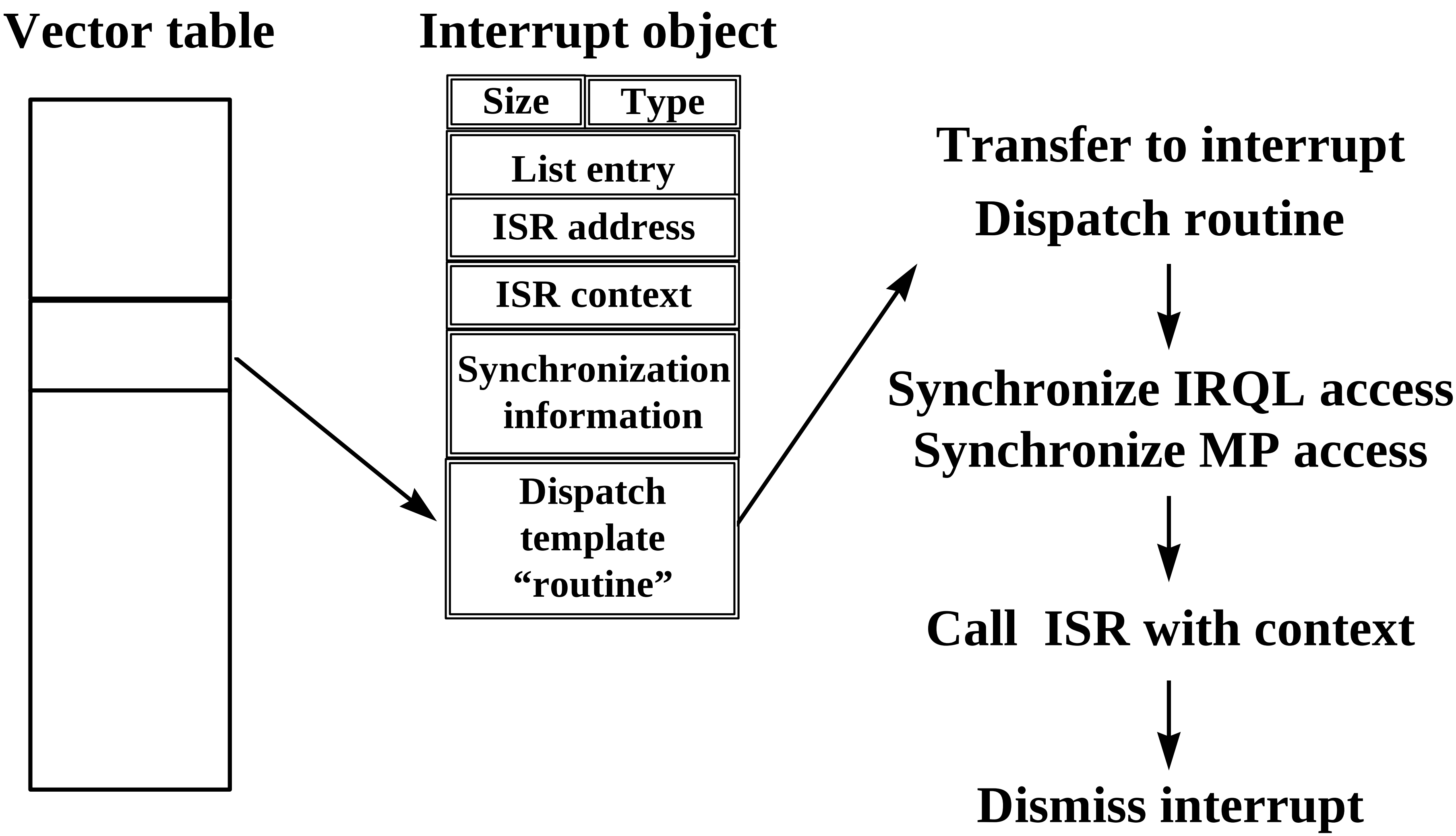
DPC Object

- Breaks into the execution of any thread
- Executes specified procedure at dispatch level
- Used to defer processing from higher interrupt level
- Used heavily for I/O driver completion
- Used for quantum-end and timer expiration

Interrupt Object

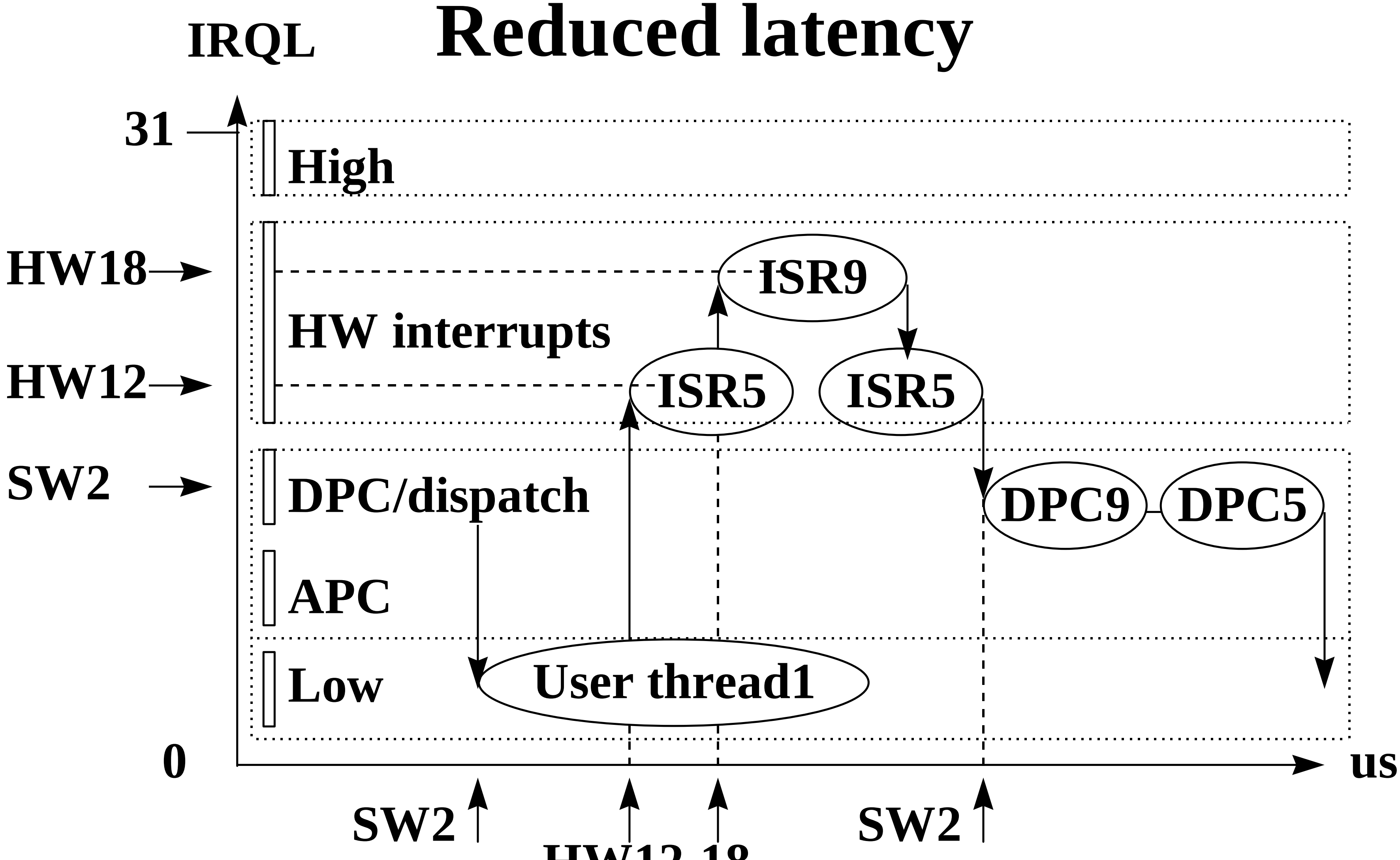
- Connects an interrupt vector to an ISR
- Allows for chained interrupts at single vector
- Automatically forces synchronization on MP system
- Used to synchronize execution between I/O driver and ISR

Interrupt Dispatch

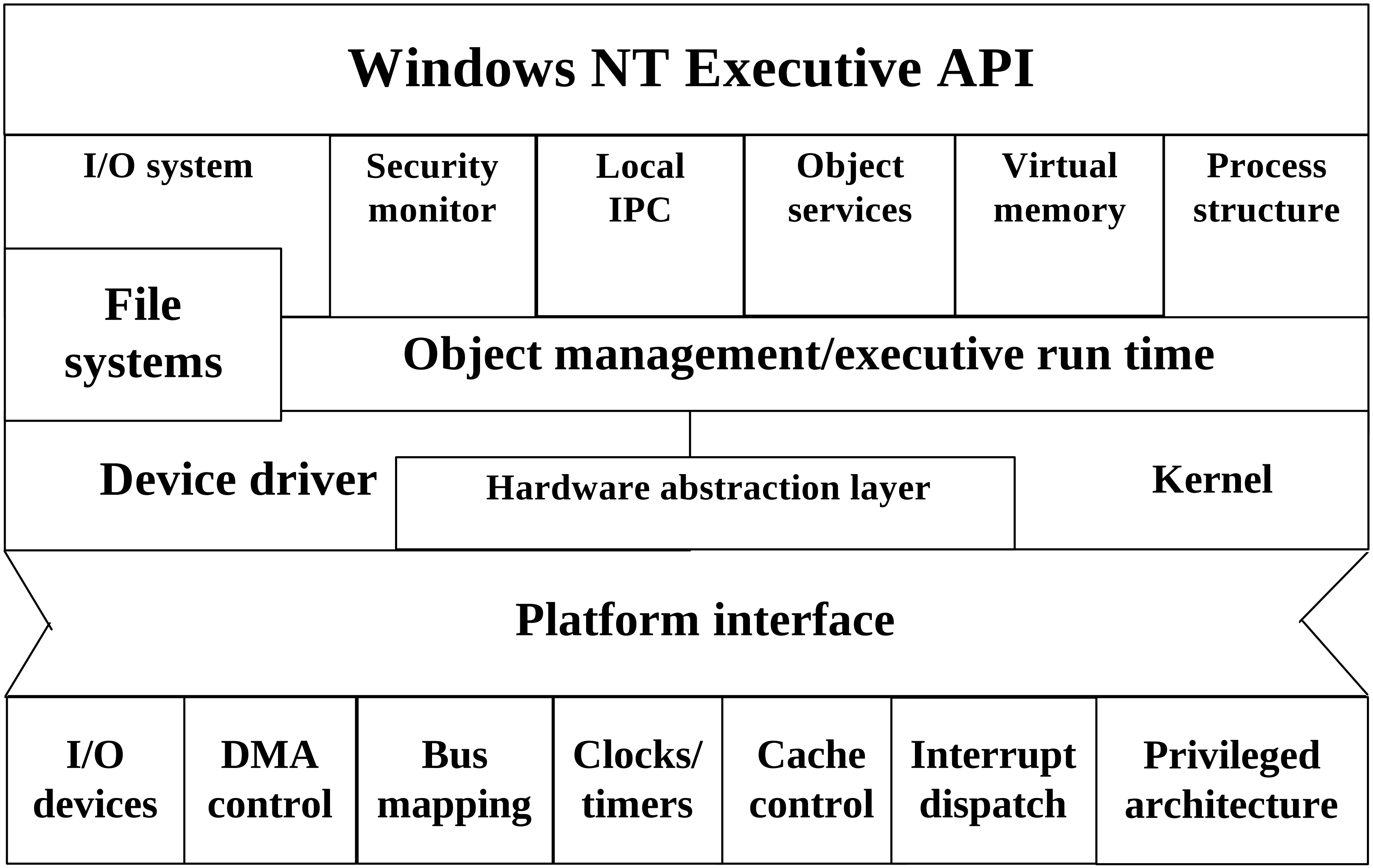


DPC Processing

Reduced latency



Windows NT Driver Architecture



Design Goals

- Easy driver development
- Portable
- Secure
- Multiuser
- Support installable file systems
- Layered drivers

Assumptions

- Driver model assumes scatter/gather hardware
- Support for many devices
- Security and robustness are becoming increasingly important
- Portability

I/O System Components

ubsystem

vice driver routines

e system driver routines

stem service routines

I/O System Components

- **Driver API**
  - **Driver invokes IoXxx routines**
  - **I/O routines operate on "objects"**
  - **Communication via I/O request packets (IRP)**

I/O Routines

- Examples
  - IoCallDriver
  - IoCompleteRequest
  - IoBuildAsynchronousFsdRequest
  - IoReadPartitionTable
  - IoStartNextPacket
- There are approximately 80 IoXxx functions

I/O System
Data structures

- Driver object
- Device object
- Controller object
- Adapter object
- Interrupt object

Data Structures

- Driver object
  - Describes driver to I/O system
  - Contains size, dispatch routine addresses, etc.
- Device object
  - Represents physical device to I/O system
  - Device-independent section
  - Device-dependent section

Data Structures

- Controller object
  - Represents controller to I/O system
  - Allows allocation and synchronization by devices
- Adapter object
  - Represents hardware mapping registers and channel
  - Allows allocation and synchronization by devices

Data Structures

- Interrupt object
  - Provided by the kernel
  - Allows drivers to associate ISR with an interrupt vector
  - Allows driver to synchronize with ISR via KeSynchronizeExecution

Driver Model Description

- Drivers loaded on boot or dynamically
- Two types of drivers
  - Device drivers (DD)
  - File system drivers (FSD)
- Device drivers
  - Limited context
  - Execute in context of calling thread, ISR, and DPC routine

Parts Of A Device Driver

- Initialization routine
- Dispatch routines
- Start I/O routine
- Interrupt service routine (ISR)
- DPC routine
- Unload routine
- Completion routines (optional)
- Error log routines (optional)

Driver Layering

- Drivers can be layered
  - Allows “intermediate” drivers between two drivers
  - Example: stripe or mirror drivers
- Accomplished through “I/O stack locations” in IRP
  - One stack location per driver layer
  - Allows reuse of IRP

Driver Layering

- Each stack location allows communication with next driver
- Stack location contents
  - Major/minor function codes
  - File object pointer
  - Device object pointer
  - Parameter flags
  - Four function parameters
  - Completion routine information

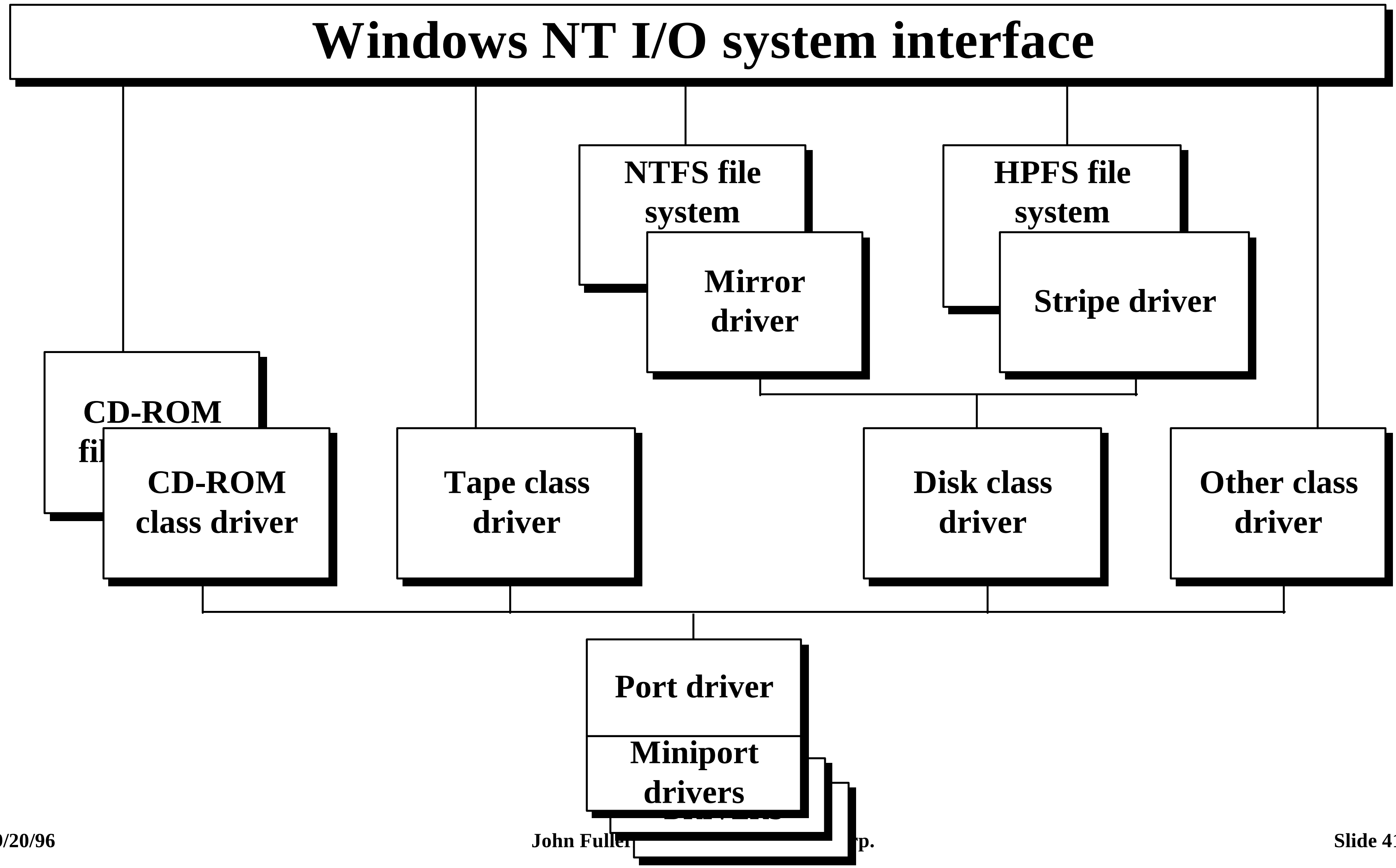
IRP Contents

Example: File system NtWrite gives the following IRP:

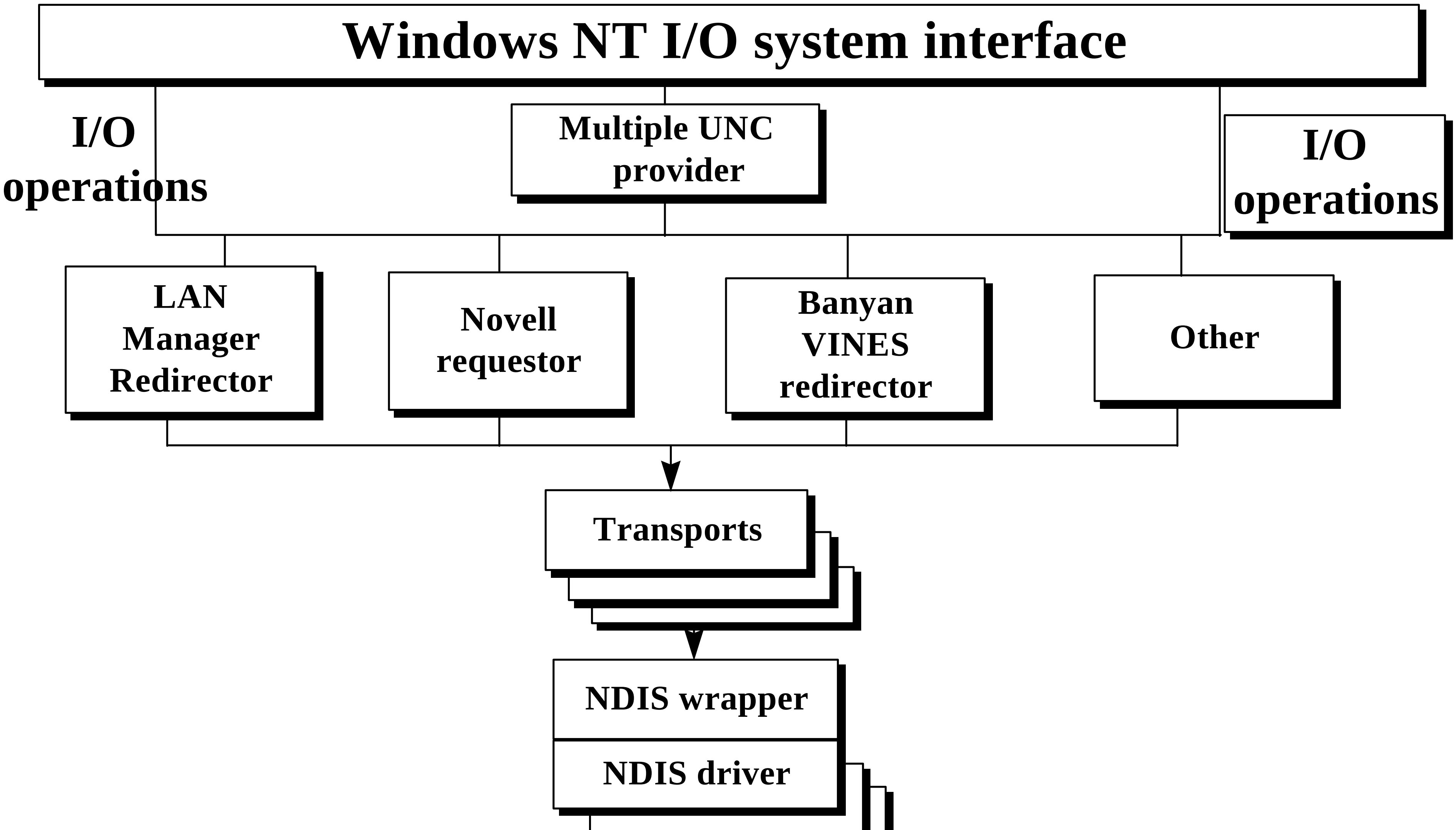
IRP body is accessible to all drivers and contains information such as buffer pointers, event objects, etc.
Stack location contains disk driver parameters
Stack location contains file system driver parameters

IRP stack locations are reserved for communications between adjacent layered drivers

Typical SCSI Layering



Network Layering



I/O System Features

- Portable - drivers written for one platform often port with few or no code changes
- Secure - data from one process is protected against access or corruption by others
- Multiuser, multithread, multiprocessor - I/O architecture allows effective use of many threads of execution, even on different processors

I/O System Features

- Layered drivers - driver functionality can be compartmentalized to make developing drivers easier
- Object-oriented - all knowledge of a driver is confined to the knowledge in objects exposed to the I/O subsystem - so replacing or modifying a driver is easier
- Fast - once the operation is in kernel mode, all other drivers are a function call away

Future Directions

- Consolidate Windows Driver Model
  - Plug and Play advances
  - Power Management advances
  - New bus support
  - New device support

Win32 Driver Model (WDM)

- Common model for driver development
  - Market-driven from USB, 1394 connectivity enhancements and peripherals
- Based on existing Windows NT driver model
  - Brought forward features for SMP and additional platform independence
- Added missing features
  - Plug and Play

Areas Of Change
Windows NT To WDM

- **Device driver setup and shutdown**
  - **Drivers will be notified of device arrival instead of having to “search” for devices**
  - **All drivers will be able to unload**
- **Additional functions**
  - **Power state control**
  - **Bus enumeration**
- **Steady-state operation is unchanged!**

Win32 Driver Model

- I/O chapter in *"Inside Windows NT"*
- Windows NT DDK
- Questions?

WinHEC 96

CIRRUS LOGIC

Electronic Engineering TIMES

Microsoft

COMPUTER & COMMUNICATIONS OEM MAGAZINE

Future Developments In The Win32® Driver Model

Lonny McMichael
Software Design Engineer
Windows NT Development
Microsoft Corporation

Agenda

- WDM goals/nongoals
- Plug and Play/Power Management Implementation overview
- Plug and Play interaction for WDM drivers
- User-mode Plug and Play components
- Summary

Goals

- Common Plug and Play and Power Management device driver interfaces for future versions of Windows NT® and Windows®
- Support Plug and Play hardware standards
- Build on existing Windows NT I/O infrastructure

Nongoals

- Windows NT will not support VxDs
  - VxDs continue to work unchanged in future versions of Windows as non-WDM drivers
- Future versions of Windows will not support non-Plug and Play (legacy) Windows NT drivers
  - Windows NT will continue to support existing drivers, but with reduced Plug and Play/Power Management functionality

Implementation Overview

- Plug and Play/Power Management bus functionality via WDM bus driver
- Control centralized in kernel-mode Plug and Play/Power Manager
  - Directs bus driver (enumeration, configuration, etc.)
  - Directs device driver (add device, start device, etc.)

WDM Bus Drivers

- Standard WDM driver that exposes a bus
- "Bus" is any device off of which other devices are enumerated (includes multifunction adapters)
- Responds to standard bus IOCTLs
- Extensible via filter drivers

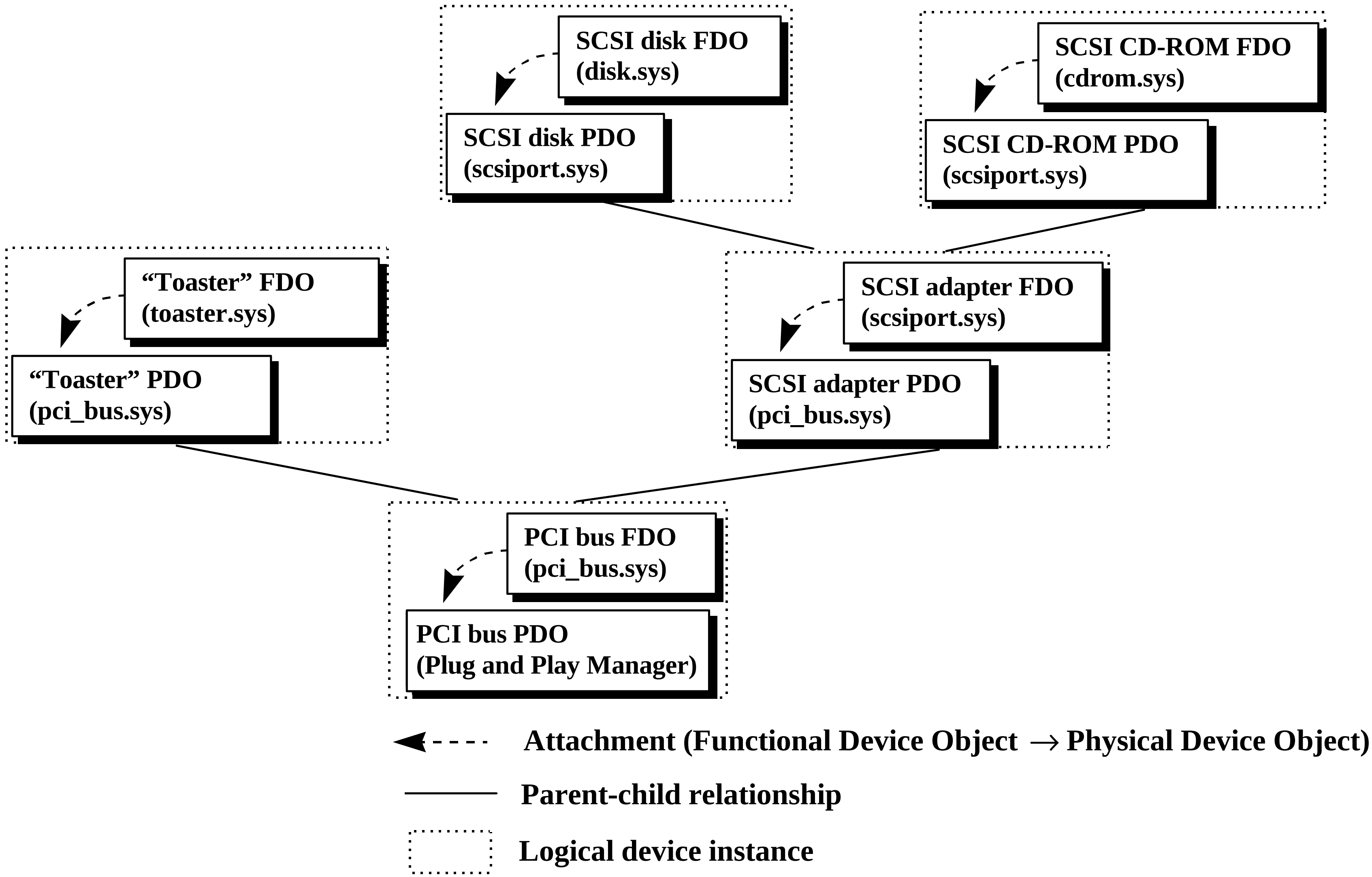
Enumeration

- **Plug and Play Manager enumerates devices off a bus à la *FindFirst*/*FindNext***
- **Bus driver returns a reference to a *Physical Device Object* (PDO) for each device on the bus**
- **PDO is conceptually the "handle" to the physical device (analogous to Windows 95 *devnode*)**

Device Driver Initialization

- **Driver init**
  - **Global initialization only - no devices**
- **Add device**
  - **Driver is given a PDO representing a new device it will control**
  - **Driver creates its own *Functional Device Object* (FDO) and attaches it to the PDO**
  - **Resource requirements may be filtered**

WDM Device Tree



Device Control

- Start device
  - Device receives assigned resources
  - Driver begins controlling the device
- Stop device
  - Driver releases resources, stops controlling device
  - Preceded by query-stop
  - Driver may be stopped, then started again, in order to assign new resources

User-Mode Plug And Play Components

- **Common APIs on both Windows NT and Windows 95:**
  - **32-bit-extended versions of user-mode Windows 95-based ConfigMgr APIs**
  - **32-bit device installer APIs, functionally equivalent to Windows 95 *setupx***
  - **Win32 APIs**

Summary

- Single set of Plug and Play interfaces for WDM drivers
- Concepts familiar to Windows 95 DVxD developers
- Addresses Windows NT goals
  - Portability, security, robustness
- Power Management discussion later this morning in session 2

WinHEC 96

CIRRUS LOGIC®

Electronic Engineering TIMES

Microsoft®

COMPUTER & COMMUNICATIONS OEM MAGAZINE

Win32® Driver Model

Forrest Foltz
Software Design Engineer
Microsoft Corporation

Objective

- To create a driver infrastructure that:
  - Is extensible, both for Microsoft and third-party providers
  - Offers not only device abstraction but OS, and bus abstraction, as well
  - Further enables code/binary sharing across the Windows® and Windows NT®-based platforms
  - Lays the groundwork for OS-common bus support...makes supporting a new bus class as straightforward as supporting a new device class
  - Enables legacy hardware-specific applications to interoperate with devices and buses of the future

Problems Today

- Monolithic drivers under Windows span many logical layers
  - VKD (keyboard driver) for example: communicates with user/kernel at upper edge, directly with 8042 at the bottom
  - Third parties must replace (possibly already enhanced) VKD to add keyboard device value

Problems Today

- Inflexible architecture
  - Multiple pointing devices are not supported
  - Support for multifunction devices is difficult at best
  - Support for existing device classes on new buses does not exist
- Performance
  - Long latencies reduce response time in interactive situations

Win32 Driver Model

- Enables device, bus, and os-independent functionality for device classes
- Leverages existing Windows NT I/O subsystem to provide reduced latency for interactive applications
- Can coexist with existing class-specific driver models such as mass-storage and networking

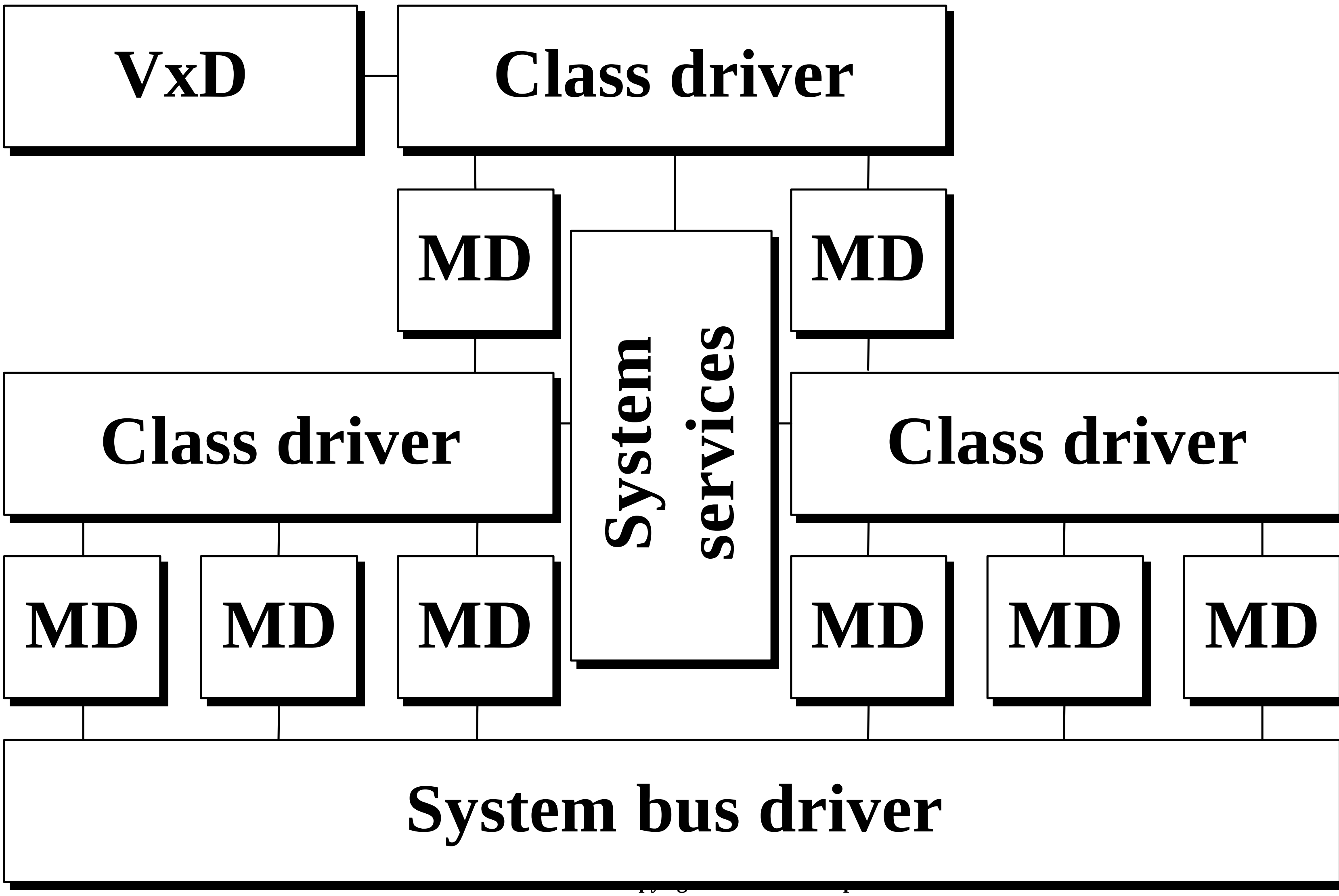
Win32 Driver Model

- Enables legacy direct-to-hardware applications (i.e., a real-mode Sound Blaster™ application) over a new sound device on an arbitrary bus
- Three main classes of drivers:
  - Minidrivers
  - Class drivers
  - Virtualization drivers

Minidrivers

- Indirectly communicate with a specific hardware device via a specific bus class driver
- Source and (x86) binary-compatible across Windows and Windows NT
- Dynamically loadable/unloadable

Minidrivers



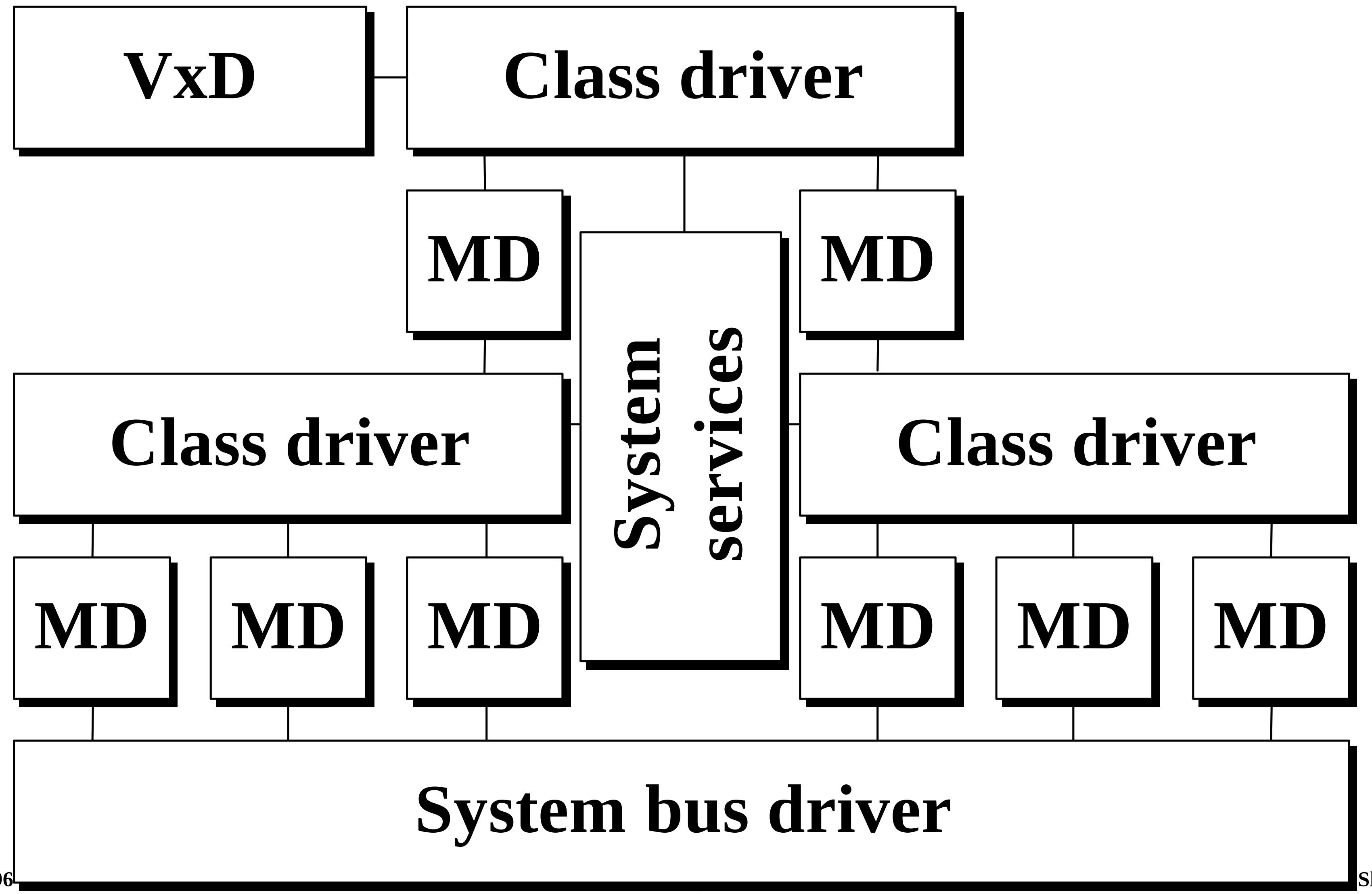
Minidrivers

- Contain only hardware-specific functionality
- Multifunction minidrivers can expose multiple class interfaces
- Class driver is sole client of a driver's interface, minidriver does not perform client/processor arbitration within an interface

Class Drivers

- Define the class interface to the rest of the OS
- Lower edge communicates with identical, class-specific interfaces, exposed by minidrivers (except in the case of the system bus driver, a.k.a., HAL)
- Source and (x86) binary-compatible across Windows and Windows NT

Class Drivers



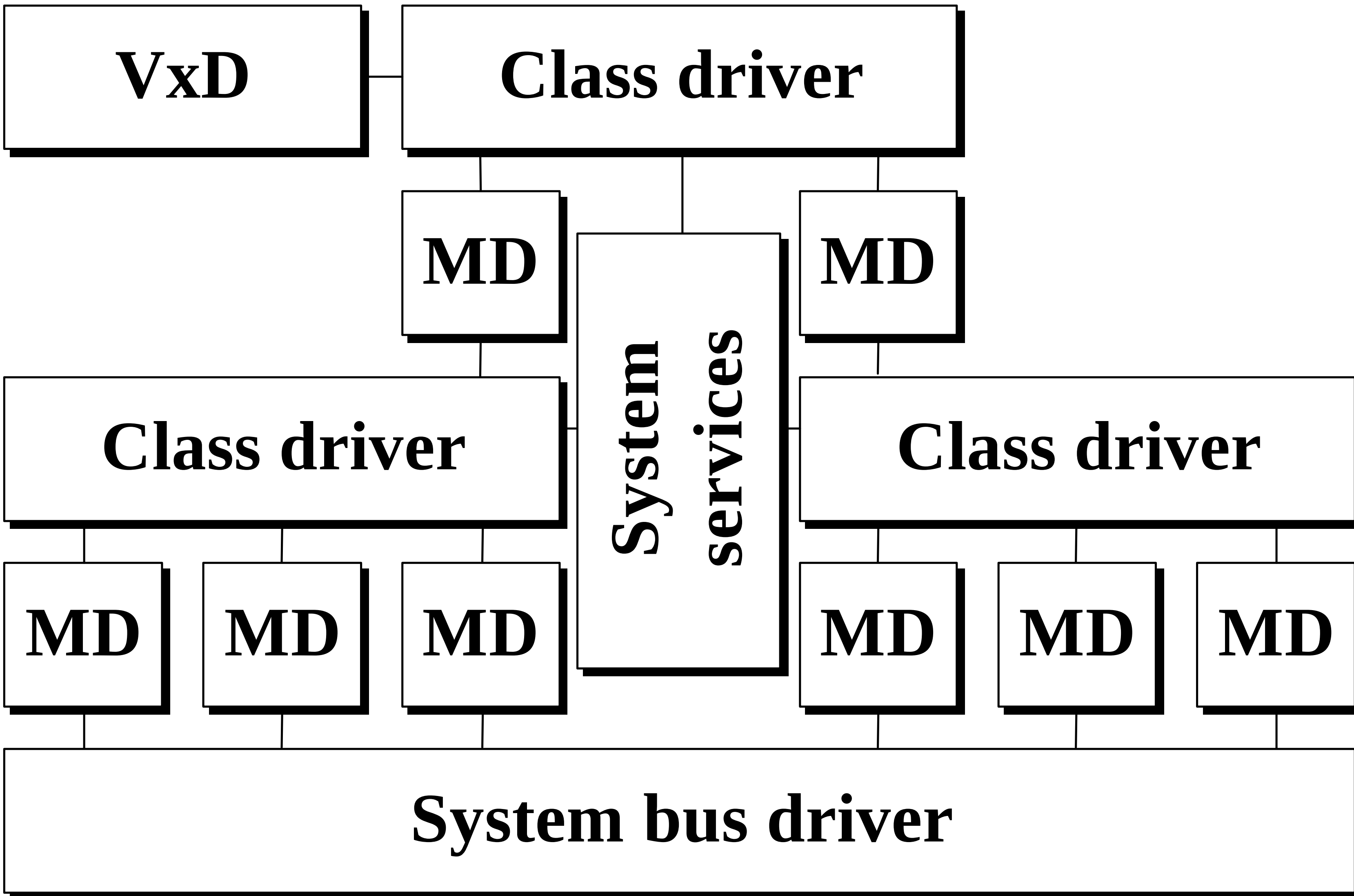
Class Drivers

- Provides class-specific functionality, not hardware or bus-specific (except, of course, where the bus type is the class)
- Dynamically loads/unloads
- Contains only class-specific functionality
- Exposes a single, class-specific interface, to multiple clients

Virtualization Drivers

- Virtualize legacy hardware interfaces, send class-specific commands to the appropriate device class driver
  - Legacy game port access converted to joystick class commands, sent to joystick device on USB bus
  - Legacy Sound Blaster access converted to sound device commands, sent to audio device on 1394 bus
- Do not drive hardware directly

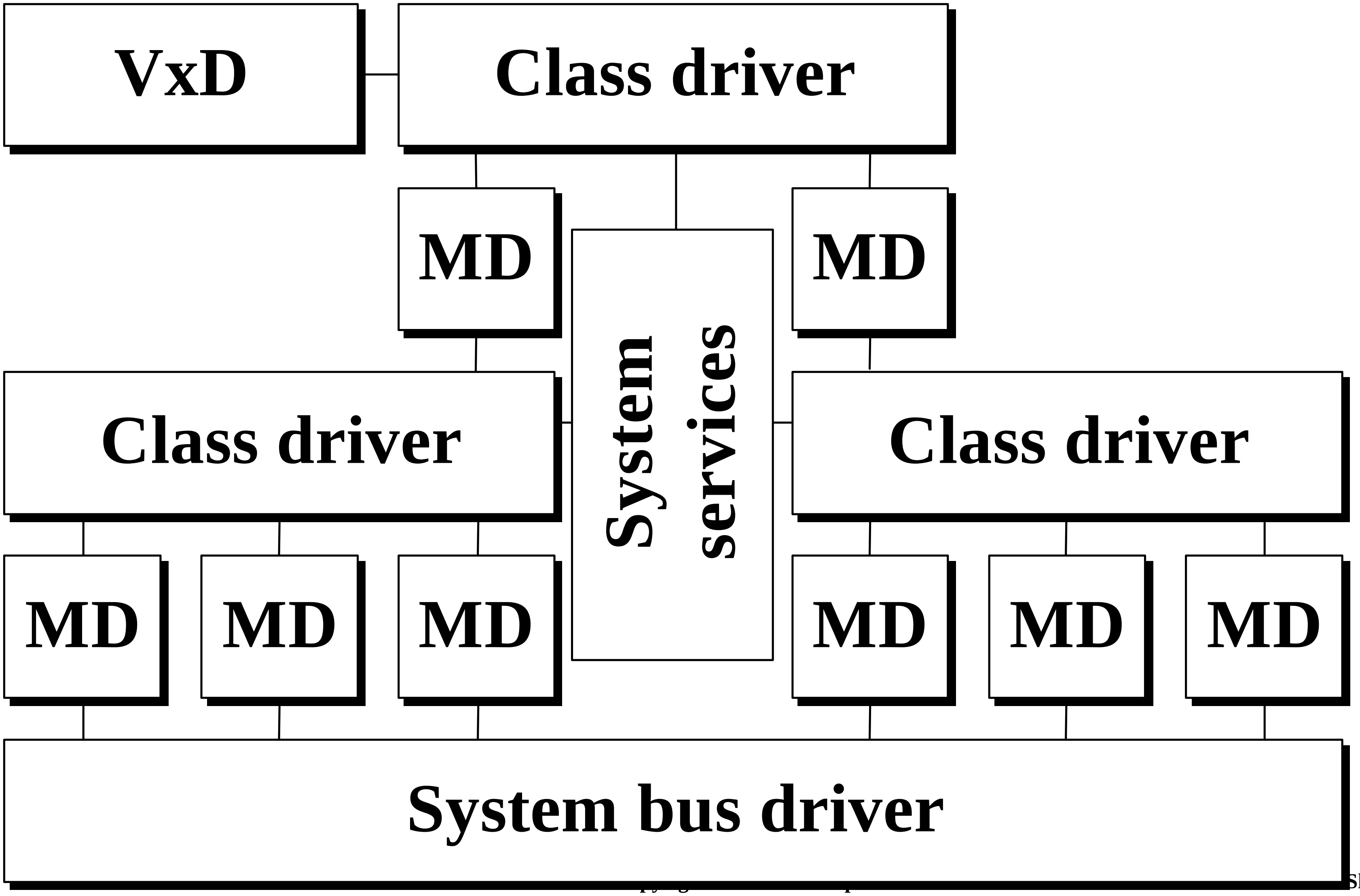
Virtualization Drivers

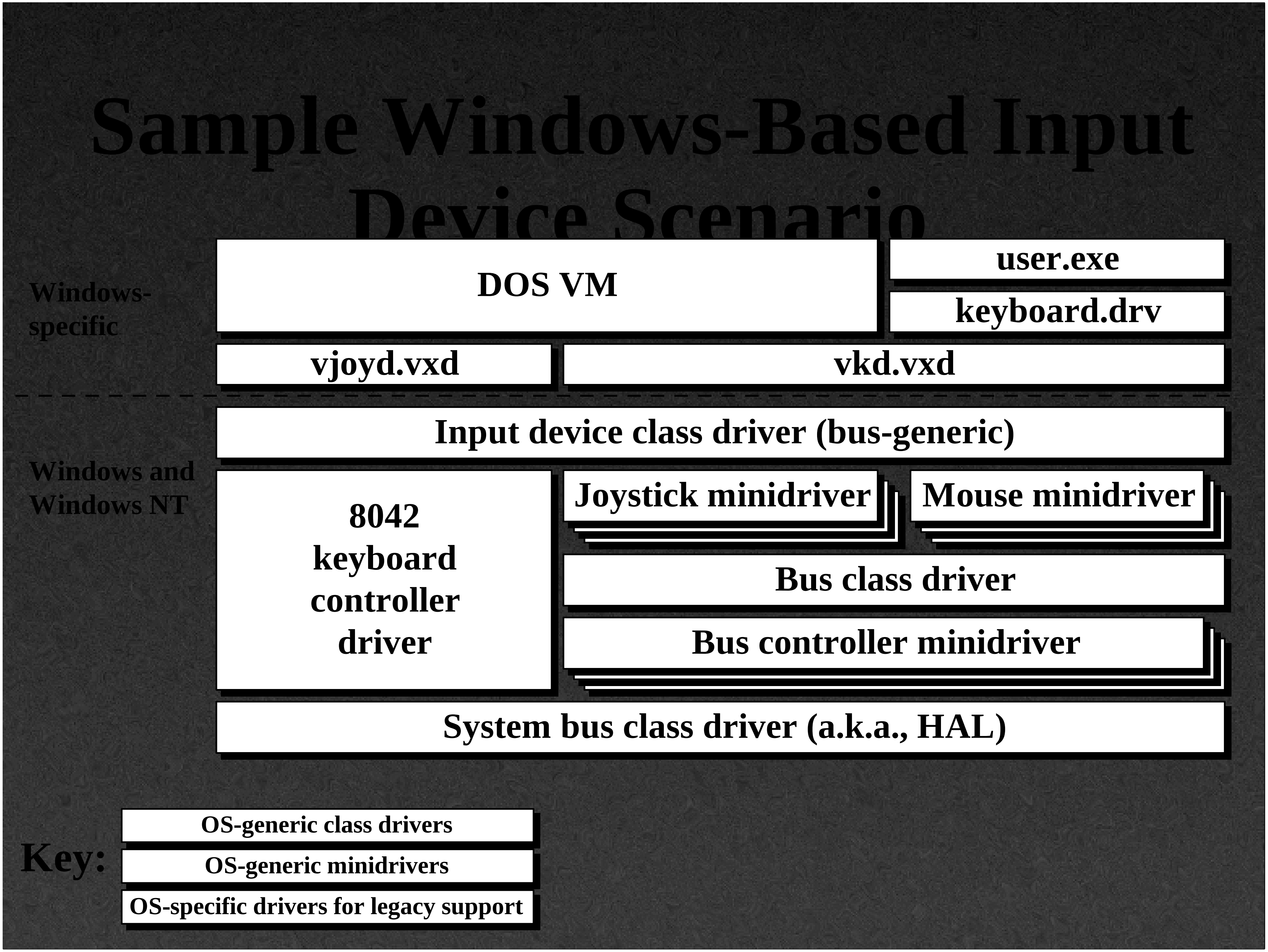


OS Services

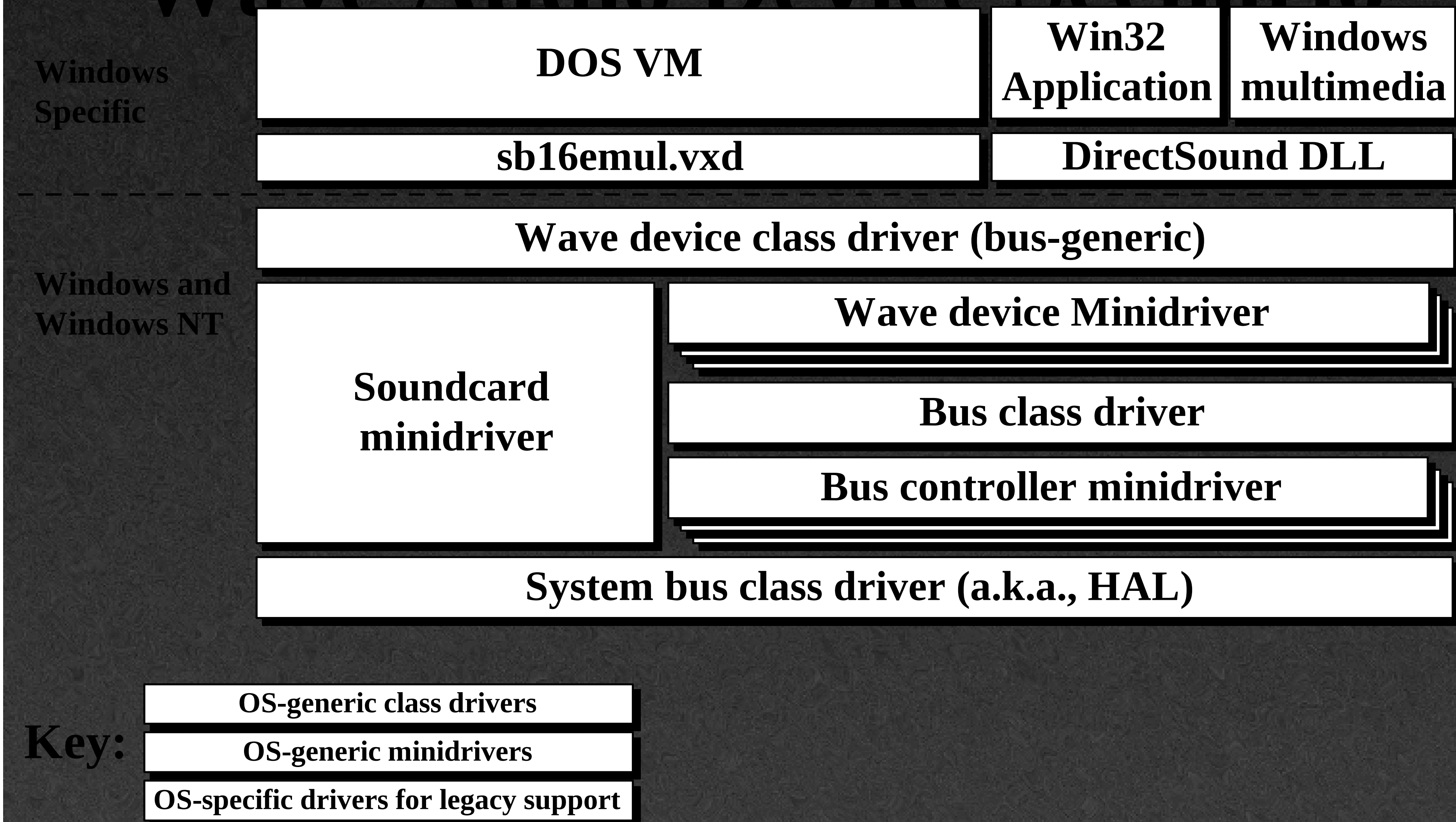
- Subset of DDIs available to Windows NT kernel-mode device drivers
- Offer abstraction of OS-specific functionality to minidrivers
- Examples:
  - Driver communication
  - Plug and Play
  - Event services

OS Services


VxD
Class driver
MD
MD
System services
Class driver
Class driver
MD
MD
MD
MD
MD
MD
System bus driver


Sample Windows-Based Input Device Scenario
Windows-specific
DOS VM
user.exe
keyboard.drv
vjoyd.vxd
vkd.vxd
Windows and Windows NT
Input device class driver (bus-generic)
8042 keyboard controller driver
Joystick minidriver
Mouse minidriver
Bus class driver
Bus controller minidriver
System bus class driver (a.k.a., HAL)
Key:
OS-generic class drivers
OS-generic minidrivers
OS-specific drivers for legacy support

Sample Windows-Based Wave Audio Device Scenario



Win32 Driver Model
Why WDM is important to you

- Common I/O services
- Source/(x86) binary-compatible drivers, across Windows and Windows NT
- Reduced latency
- Higher-quality drivers
- Lower development cost
- Hardware innovation
- Easy, new bus support

Win32 Driver Model
Actions and opportunities

- **Get familiar with the Win32 driver architecture**
  - ***"Inside Windows NT"*** **by Helen Custer**
  - **Windows NT DDK**
    - **MSDN Level 2 (msdn@microsoft.com)**
  - **Win32 Driver Development Kit**
    - **Common driver services**
    - **E-mail ihv@microsoft.com**

Win32 Driver Model
Open process

- Participate in Microsoft Developer Events (confirmation required due to limited space)
  - Input Devices: April 23
  - Win32 Driver Model: May 15
  - Others to be announced
    - Send registration information to ihv@microsoft.com

Win32 Driver Model

Open process

- Acquire the latest design specifications
  - http://www.microsoft.com/windows/thirdparty/hardware
    - Power Management
    - Plug and Play
    - 1394 design guidelines
    - Others to be announced

Win32 Driver Model
Provide feedback

- ihv@microsoft.com
  - General feedback
- rt@microsoft.com
  - Low latency requirements
- 1394@microsoft.com
  - Win32 class driver interfaces
    - 1394 command and protocol standards
- power@microsoft.com
  - Power Management - OnNow

Call To Action

- Support Power Management
- Build Plug and Play-compliant hardware
- Build USB and 1394 peripherals
  - Leverage Win32 standard class/minidriver interfaces
  - Follow design guidelines for Plug and Play busses

Win32 Driver Model

Microsoft commitment

- Windows 95 OEM Service Release 2 releases addressing market demand
- New device support
  - Consumer audio/video
  - Input
  - Storage
- New bus support
  - USB
  - 1394
- Others to follow

Win32 Driver Model Seminar Questions And Answers

WinHEC 96

CIRRUS LOGIC®

Electronic Engineering
TIMES

Microsoft®

COMPUTER & COMMUNICATIONS
OEM
MAGAZINE