Windows Kernel Internals II
Windows Driver Foundation

University of Tokyo – July 2004

Dave Probert, Ph.D.
Advanced Operating Systems Group
Windows Core Operating Systems Division
Microsoft Corporation

Topics

WDF Overview

Toaster samples

Framework objects (incl **DEVQUEUES**)

PnP/Power

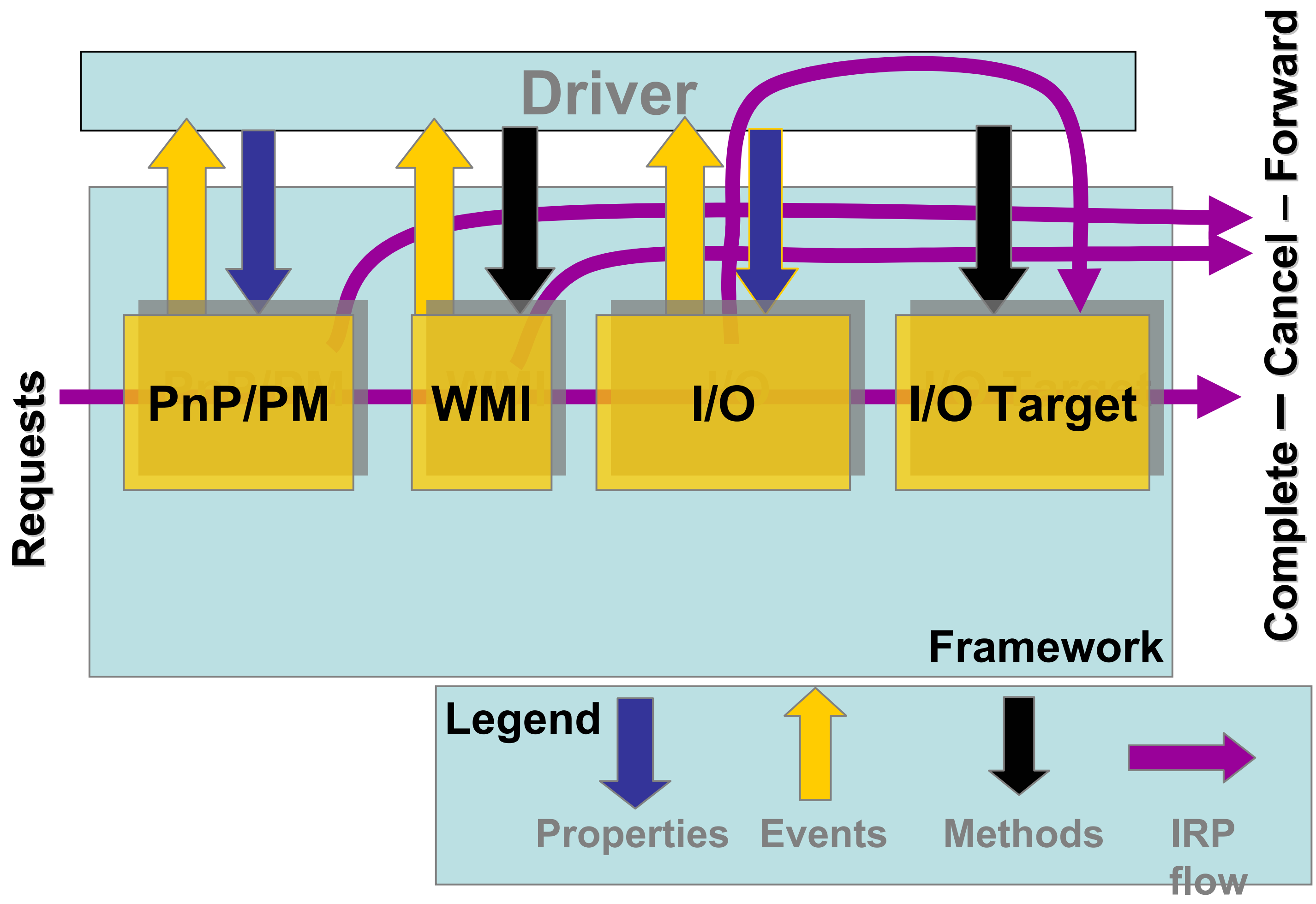
Device Interface Generation (DIG)

Windows Driver Framework

A library atop WDM:

- Simpler interfaces
- Handles most PnP and Power operations
- Simplifies MP synchronization
- Provides OO interfaces to drivers

Request Pipeline



Windows Driver Framework Architecture

WDF defines:

- Object properties
- Object methods
- Object event callbacks (into drivers)
- Object handles (used to reference objects)

Simple Framework-based Driver

A DriverEntry routine which calls:

- **WdfDriverCreate**

An *EvtDriverDeviceAdd* event callback:

- called by PnP for hardware id match

An *EvtIoStart* event callback

- called when system has queued a request

I/O Queue Event Callbacks

Corresponds to IRP major codes for:

– READ, WRITE, DEVICE_CONTROL, INTERNAL_DEVICE_CONTROL

and Cancellation

WdfFdoInitSetFilter

– Used to mark as driver as filter:

- Any operation without callback registered bypasses driver

Topics

WDF Overview

Toaster samples

Framework objects (incl **DEVQUEUES**)

PnP/Power

Device Interface Generation (DIG)

Toaster Filter Sample - 1

DriverEntry()

- Initialize driver config to control the attributes that are global to the driver.
- call WdfDriverCreate()
- call WdfCollectionCreate to create a collection object and store filter device objects.

FilterEvtDriverUnload() callback

- call WdfObjectDereference() to dereference collection

Toaster Filter Sample - 2

FilterEvtDeviceAdd() callback

- EvtDeviceAdd() is called by the framework in response to AddDevice call from the PnP manager.
- pDO = WdfFdoInitWdmGetPhysicalDevice().
- use IoGetDeviceProperty() to decide to attach.
- call WdfFdoInitSetFilter().
- call WdfDeviceCreate().
- call WdfCollectionAdd() to add to collection (while holding the lock).
- call our FilterCreateControlDevice() routine.

Toaster Filter Sample - 3

FilterCreateControlDevice() routine

- // create ctrl DO so app can talk to filter directly.
- call WdfCollectionGetCount() to determine if exists
- call WdfControlDeviceInitAllocate()
- call WdfDeviceInitSetExclusive(FALSE)
- call WdfDeviceInitUpdateName(NAME_STRING)
- call WdfDeviceCreate()
- call WdfDeviceCreateSymbolicLink()
- call WdfDeviceCreateDefaultQueue() to create device queue (callback is FilterEvtDeviceControlIoctl)
- call WdfDeviceFinishInitializing() to clear init flag in DO

Toaster Filter Sample - 4

FilterEvtDeviceContextCleanup() callback

- – acquire lock (WdfCollectionAcquireLock)
- – call WdfCollectionRemoveItem()
- – n = WdfCollectionGetCount()
- – if n==0 call FilterDeleteControlDevice() routine
- – release lock (WdfCollectionReleaseLock)

Toaster Filter Sample - 5

FilterDeleteControlDevice() callback

- call WdfObjectDereference(controlDevice)

FilterEvtDeviceControlIoctl() callback

- acquire lock (WdfCollectionAcquireLock)
- n = WdfCollectionGetCount()
- call WdfCollectionGetItem() n times
- release lock (WdfCollectionReleaseLock)
- call WdfRequestCompleteWithInformation()

Simple Toaster Function Sample

ToasterEvtDeviceAdd() –

- not sharing DO, so no collection needed
- call WdfDeviceCreateDeviceInterface()
- call WdfDeviceCreateDefaultQueue() to register IO callbacks, like ToasterEvtIoRead()

ToasterEvtIoRead() – do operation, then

- call WdfRequestCompleteWithInformation()

Toaster Bus Sample - 1

DriverEntry()

- call WdfDriverCreate() w/ Bus_EvtDeviceAdd()

Bus_EvtDeviceAdd() callback

- WdfDeviceInitSetDeviceType (FILE_DEVICE_BUS_EXTENDER)
- call WdfDeviceInitSetExclusive(TRUE)
- set callbacks: Bus_EvtDeviceListCreatePdo, Bus_EvtDeviceListIdDescription{Duplicate, Compare, Cleanup}
- WdfFdoInitSetDefaultDeviceListConfig()

Toaster Bus Sample - 2

Bus_EvtDeviceAdd() callback (cont)

- call WdfDeviceCreate()
- call WdfDeviceCreateDefaultQueue() [ioctl]
- call WdfDeviceCreateDeviceInterface() to create device interface
- call WdfFdoSetBusInformation()
- call our Bus_WmiRegistration() and Bus_DoStaticEnumeration()

Toaster Bus Sample - 3

Bus_EvtDeviceControl() callback

- call WdfIoQueueGetDevice()
- call WdfRequestRetrieveBuffer()
- switch on IOCTL

  PLUGIN_HARDWARE: Bus_PlugInDevice()

  UNPLUG_HARDWARE: Bus_UnPlugDevice()

  EJECT_HARDWARE: Bus_EjectDevice()
- call WdfRequestCompleteWithInformation()

Toaster Bus Sample - 4

Bus_PlugInDevice() [simulation]

- init device description (descr)
- list = WdfFdoGetDefaultDeviceList()
- call WdfDeviceListAddOrUpdateChild-DescriptionAsPresent (list, descr)

Bus_UnPlugDevice() [simulation]

- list = WdfFdoGetDefaultDeviceList()
- call WdfDeviceListUpdateChildDescription-AsMissing (list, serialno)

Toaster Bus Sample - 5

Bus_EjectDevice() [simulation]

- list = WdfFdoGetDefaultDeviceList()
- call WdfDeviceListRequestChildEject (list, serialno)

Bus_DoStaticEnumeration() [simulation]

- read devices from registry to simulate boot enum
- call Bus_PlugInDevice() on each ‘device’

Toaster Bus Sample - 6

Bus_EvtDeviceListIdentificationDescription-{Duplicate,Compare,Cleanup}() callbacks

– duplicate a descriptor, compare 2 descriptors (by serialno), and free memory

Bus_EvtDeviceListCreatePdo() callback

– calls our Bus_CreatePdo() routine

Toaster Bus Sample - 7

Bus_CreatePdo() routine

- WdfDeviceInitSetDeviceType(FILE_DEVICE_BUS_EXTENDER)
- WdfDeviceInitSetCharacteristics (…)
- WdfDeviceInitSetExclusive(FALSE)
- WdfPdoInit{UpdateDevice,AddHardware, AddCompatible,UpdateInstance}ID (Ids) to satisfy IRP_MN_QUERY_ID IRPs
- call WdfPdoInitAddDeviceText()
- call WdfPdoInitSetDefaultLocale()

Toaster Bus Sample - 8

- call WdfPdoInitSetEventCallbacks() for
  Bus_Pdo_EvtDeviceResourceRequirementsQuery
- call WdfDeviceCreate()
- init capabilities, call WdfPdoSetCapabilities()
- call WdfDeviceAddQueryInterface()
- call WdfDeviceFinishInitializing()

Toaster Bus Sample - 9

Bus_Pdo_EvtDeviceResourceRequirementsQuery()

- call WdfCollectionCreate(), WdfResourceIoCreate(), and WdfCollectionAdd() to collect resources
- add our collection to the ‘collection of resource collections’

Topics

WDF Overview

Toaster samples

Framework objects (incl DEVQUEUES)

PnP/Power

Device Interface Generation (DIG)

Framework Objects

WDFDRIVER: a driver

WDFDEVICE: a device

WDFILEOBJECT:

WDFMEMORY:

WDFQUEUE: queue of I/O requests

WDFREQUEST: an I/O request

WDFDPC:

WDFTIMER:

WDFWORKITEM:

WDFINTERRUPT:

Framework Object Collections

Used to represent:

- resource requirement lists
- resource lists
- set of connected child devices
- set of exported device interfaces
- any set of framework objects in driver
- collections of collections

WDFQUEUE Object

- Supports numerous operations
- Requests enqueuing and dequeuing
- Controls concurrency of requests presented to the driver
- Allows processing to pause and resume
- Requests cancellation and cancel-safe queues
- Synchronizes I/O operations with PnP/Power state transitions
- Reports outstanding I/O operations to PnP/Power stage
- Serializes event callbacks
- Defers event callbacks to comply with PASSIVE_LEVEL constraints

WDFQUEUE Request Events

WDFQUEUE objects use callbacks to notify driver of WDFREQUEST events

- EvtIoRead – IRP_MJ_READ requests
- EvtIoWrite – IRP_MJ_WRITE requests
- EvtIoDeviceControl – device control requests
- EvtIoCancel – a request is cancelled
- EvtIoStop – a power state change requested
- EvtIoStart – request w/o a specific callback

WDFQUEUE Concurrency

“in-flight” requests:

- received from queue, not yet completed

Concurrency control for “in-flight” requests

- Serial, single request model
- Parallel model
- Manual model

WDF may ask cancel/suspend “in-flight” requests

- due to IO cancel, PnP/Power events, dev removal
- driver implements EvtIoCancel/EvtIoStop callbacks

Auto cancel/suspend of queued requests

WDFQUEUE Power Management

Power management of WDFQUEUEs

- Enabled by default
- Configurable on a per WDFQUEUE basis

Advantages of power-managed queues

- Notify PnP/Power stage of arriving I/O requests so that device power can be restored
- Notify PnP/Power stage of empty queue so that device can be powered down
- Notify driver of power-state changes for in-flight requests through the *EvtIoStop* callback

WDFQUEUE Serialization and Constraints

Outstanding I/O request serialization

- I/O requests received from a WDFQUEUE are asynchronous
- Requests completed in event callback or later
- Driver configures number of concurrent I/O operations per queue

Constraints on concurrent execution of event callbacks

- Set in WDF_OBJECT_CONSTRAINTS
- Control simultaneous event callbacks (not actual I/O operations)
- Help manage shared access to WDFQUEUE context memory

Callbacks can have PASSIVE_LEVEL constraint

- WDFQUEUE automatically invokes the callback from a system work item if required

Object Context Memory

- Can be associated with any WDF object
- Similar to a device extension
- Provides storage for a drivers object-specific information
- Allocated from non-paged pool in driver-supplied size and type
- Macros assist in defining the type from a C struct
- Accessed through pointer stored/retrieved through the object handle
- Object’s can have more than one memory context, if the types differ
- Optional event callback EvtObjectDestroy deallocates context when the object handle is destroyed

Asynchronous Processing

- Objects used for asynchronous events
  - WDFDPC, WDFTIMER, WDFWORKITEM
- Associated with a WDFDEVICE or WDFQUEUE
- Automatically handle race conditions
- Asynchronous processing can serialize with an object's event callbacks
- IRQL of the object must be compatible

WDFINTERRUPT / WDFDPC

Supports

- Wire line and message signaled interrupts
- Notification of assignment of interrupt resources
- DIRQL synchronization functions
- Associated with WDFDEVICE object

EvtInterruptIsr callback

- services interrupt, stores in context memory
- after dismissing, calls **WdfInterruptQueueDpcForIsr**

I/O Targets

Target for forwarding request

- local I/O target: next driver in stack
- remote I/O target: some other driver
- I/O targets list where requests went (for cancel)
- can be general or specialized (e.g. USB)

I/O target states:

- Started, Query-stop, Stopped, Query-remove, Removed, Surprise-removed, Closed

Topics

- WDF Overview
- Toaster samples
- Framework objects (incl **DEVQUEUES**)
- **PnP/Power**
- Device Interface Generation (DIG)

WDF PnP/Power Design Goals

- **Remove as much boilerplate as possible**
- **Driver callbacks only for “interesting” events**
- **Automatically provide good default PnP behavior**
  - Rebalance, Removal, Surprise Removal
- **Automatically provide good default Power behavior**
  - Support Sleep/Hibernate, “Fast Resume”, idle-time power management
- **Provide clear error-handling paths**
  - Some software errors automatically handled
  - Some hardware errors handled by resetting device

WDF PnP/Power Design Goals

- Integrate driver primitives with PnP/Power actions
- Automatically stop presenting requests when leaving D0 (high-power)
- When leaving D0 disconnect interrupts, Stop DMA & I/O Targets
- All PnP/Power callbacks at PASSIVE
- Remove need for drivers to track state
- Callback primitives small w/ straightline code

WDF Bus Drivers Trivial to write

WDF can handle most of the details:

- Reporting children to WDM
- Coordinating scanning for children
- Maintaining the list of children

Drivers responsible for:

- Identifying children
- Generating IDs
- Generating resource requirements
- Identifying capabilities
- Notification that children have been removed

Power Managed Queues

- **Queues (optionally) aware of device power state**
- **Device hardware held in high-power state until requests completed or marked as “stopped”**
- **Requests queued and not presented to driver until machine fully resumes from a sleep state and device is in D0**
- **Not all queues are power-managed:**
  – Queues for requests that touch hardware should be power-managed
  – Device Control queues and queues in software-only drivers usually should NOT be power-managed

Power Policy Ownership

WDF provides a rich set of automatic behaviors

Device to low-power when the system goes to sleep/hibernate

Device to low-power when the device is idle

Device to high-power when there are requests to process

Automatic arming for wake while the system is running (device is idle)

Automatic arming for wake while the system is sleeping

Simplest WDF Driver

Only required PnP/PM fcn: EvtDeviceAdd

1. Set some device constraints
2. Create a WDFDEVICE object
3. Create queues for handling requests

WDF handles PnP/PM events automatically

[If EvtDeviceAdd allocates state, must provide EvtDeviceContextCleanup]

Simple PnP/PM Callback Groups

For all devices with hardware

- **EvtDeviceD0Entry** – everytime device turned-on
- **EvtDeviceD0Exit** – everytime device turned-off

For all devices which use interrupts

- **EvtInterruptEnable** – called after EvtDeviceD0Entry
- **EvtInterruptDisable** – Called before EvtDeviceD0Exit

For all devices which have memory-mapped registers

- **EvtDevicePrepareHardware** – one-time setup ops
- **EvtDeviceReleaseHardware**

For all USB devices

- **EvtDevicePrepareHardware**

Advanced Power Management

Drivers opt into advanced PM

- Devices only in D0 when there is work
- Otherwise devices in a low-power state
- Devices ->D0 by power-managed queues
- Devices ->D0 when wake signals trigger

Very little code needed. Driver provides:

- Arm/Disarm wake callbacks
- Info on idle detection and D-states for idle

PnP Child Enumeration

Properties for static data

- Bus instance ID
- Compatible IDs
- Hardware IDs, etc.

Callbacks for dynamic data and child specific actions

- Associated resources
- Eject
- Create child

Two conceptual API groupings

- “Software” child device APIs
- “Hardware” child device APIs

Software (“static”) Children

Enumerated as result of

- Request from user mode
- Registry setting
- Hard coded logic in the driver

Once enumerated, rarely reported missing

Simple API for reporting child to WDF

Hardware (“dynamic”) Children

These devices come/go frequently

True physical dependents of the parent

Enumeration driven by bus events

Redetection of child when parent ->D0

WDFDEVICELIST simplifies enumeration

- Parent reports arrival/departure of child
- Reporting asynchronous with scanning

WDFDEVICELIST APIs

Scanning

- WdfDeviceList**{Begin,End}Scan**

Updating status

- WdfDeviceList**{**
  AddOrUpdateChildDescriptionAsPresent, UpdateChildDescriptionAsMissing, RequestChildEject}

List Iteration

- WdfDeviceList**{Begin,End}Iteration**
- WdfDeviceList**GetNextDevice**

Child Device Identification

WDFDEVICELIST uniquely ids children

Two types of identification:

- identification description: how device is **found** on bus (fixed)
- address description: how device is **spoken to** on bus (dynamic)

WDFDEVICELIST Callbacks

Only the bus driver knows the following

– How big the ID description is
– If an address description is required
– How to compare two ID descriptions
– How to copy an ID
– How to cleanup an ID's buffer

Topics

- WDF Overview
- Toaster samples
- Framework objects (incl **DEVQUEUES**)
- PnP/Power
- **Device Interface Generation (DIG)**

Device Interface Generation

Replace IOCTL as programming model with something more client/server-like

Goals

- type-safety
- simplified driver code
- separate interfaces and implementation
- enable new transports (i.e. not just syscall)

What is a Device Interface

A contract between client and driver that defines:

- Operations, parameters, results, and constraints
- Access permission required, IRQ Level, etc

Interface Definition should drive the implementation

- Define the interface as "how the implementation works"

Interface Definition does not explicitly address:

- Transfer mode, transport mechanism, packet format
- Separate interface from its binding to a particular transport

DIG plan

Basic strategy

- Extract interfaces from the code
- Specify interfaces abstractly in XML
- “Regenerate” the interface code from XML

Advantages/opportunities

- DIG can generate interface code for multiple languages
- Provide help for static verification
- Use multiple IOCTL codes underneath to improve efficiency
- Generate wrappers (stubs) for clients
- Impedance-match 32-bit clients to 64-bit drivers

Discussion