# How to Port WDM Driver to KMDF

# Agenda

- Introduction to WDF
- Why should I convert to KMDF: Case Study
- Basic object model
- DriverEntry
- PnP/Power callbacks
- Self-Managed I/O callbacks
- How to configure wait-wake & idle power management
- Interrupt handling
- Callbacks specific to FDO and PDO
- Order of callbacks with respect to PnP/Power actions

# Agenda (con’t)

- Different types of queues
- How queue states are managed by WDF
- Request Cancellation
- Handling Create, Cleanup & Close requests
- Handling I/O requests – Read/Write/Ioctl
- Timers/DPC/Work items
- Locks: WaitLock, Spinlock
- Automatic I/O synchronization
- Sending request to another driver
- Escaping to WDM

# What is WDF?

- Windows Driver Foundation consists of
  - User Mode Driver Framework (UMDF )
  - Kernel Mode Driver Framework (KMDF)
  - Tools: SDV, Driver PRE*fast*, DIG, etc.
- KMDF is built on top of WDM
- Drivers written using KMDF are compatible from Windows 2000 forward
- Drivers are written using objects

# Why Convert to WDF?

- List of things you worry about in WDM
- Tons of rules on handling PnP and power IRPs
- When to use remove locks
- IRP queuing and cancellation
- When to map and unmap HW resources
- When to enable/disable device interfaces
- When to register/deregister with  WMI
- When to connect & disconnect interrupts
- Timer DPC and device remove/unload synchronization

# Why Convert to WDF? (con’t)

- Converting S IRPs to D IRPs
- Supporting wait-wake
- Supporting selective suspend (S0 Sleep)
- Fast resume
- Asynchronous start
- Child device enumeration
- Complex rules on deleting a PDO
- Handling PnP/power IRPs in a filter driver
- Error handling
- Backward compatibility

# Case Study: PCIDRV Sample

| Stats | WDM | WDF | Comments |
|---|---|---|---|
| Line Count | 13,147 | 7,271 | Explicit registration of granular event callbacks adds to the line count |
| LOC devoted to PnP/PM | 7,991 | 1,795 | Almost 6000 lines of code are eliminated |
| Locks | 8 | 3 | This is the most important statistic. This explains the complexity. |
| State variables devoted to PnP/PM | 30 | 0 | There are fewer paths in the driver and thus less testing and complexity. |

- This sample is written for the Intel E100B NIC Card
- It's a WDM version of network driver with NDIS interfaces separated out  in an upper filter driver (ndisedge)
- Both samples are in the DDK and are functionally equivalent

# Case Study: Serial Sample

| Stats | WDM | WDF | Comments |
|---|---|---|---|
| Line Count | 24,000 | 17,000 | Explicit registration of granular event callbacks adds to the line count |
| LOC devoted to PnP/PM | 5,000 | 2,500 | |
| Locks | 10 | 0 | This is the most important statistic. This explains the complexity. |
| State variables devoted to PnP/PM | 53 | 0 | There are fewer paths in the driver and thus less testing and complexity. |

- WDF sample does not support multi-port serial (WDM sample supports it)
- WDM statistics exclude multi-port support serial code

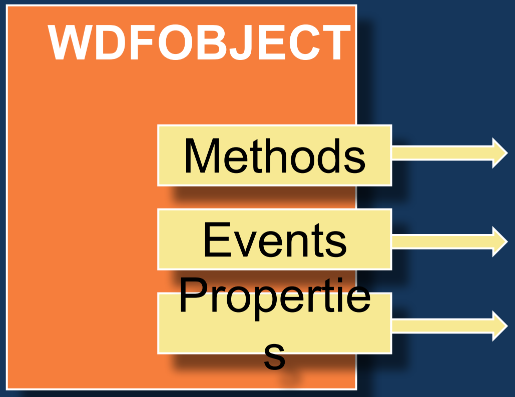
# Case Study: OSRUSBFX2 Sample

| Stats | WDM | WDF | Comments |
|---|---|---|---|
| Line Count | 16,350 | 2,300 | Explicit registration of granular event callbacks adds to the line count |
| LOC devoted to PnP/PM | 6,700 | 742 | 742 includes code to initialize the USB |
| Locks | 9 | 0 | This is the most important statistic. This explains the complexity |
| State variables devoted to PnP/PM | 21 | 0 | There are fewer paths in the driver and thus less testing and complexity |

- The WDM version of OSRUSBFx2 sample (available on osronline.com) and the WDF version provided in the DDK are functionally equivalent

# Object Model

- Objects are the basis of WDF
  - Everything in framework is represented by objects (Driver, Device, Request, etc.)
  - Objects have properties, methods, and events



WDF functions that operate on object

Calls made by WDF into the driver to notify something

Methods that get or set a single value

- Have one or more driver owned context memory areas
- Lifetime of the object is controlled by reference counts
- Organized hierarchically for controlling object life time
  - Not an inheritance based hierarchy
- Driver references objects as handles, not pointers

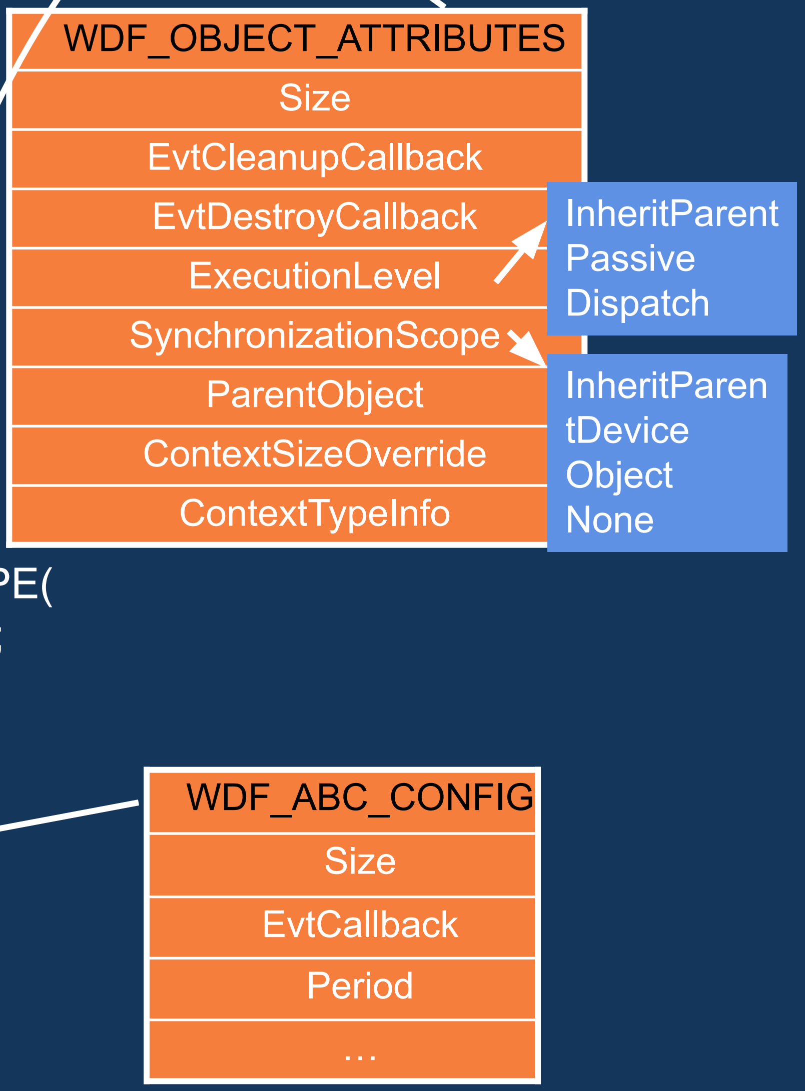
# Creating an Object (Abc)

Header File:

```
Struct _ABC_CONTEXT {
    …
} ABC_CONTEXT *PABC_CONTEXT

WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(
        ABC_CONTEXT, GetAbcContext )
```

Source File:

```
WDF_OBJECT_ATTRIBUTES_INIT(&Attributes);
WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE(
                                   &Attributes, ABC_CONTEXT );

Attributes.EvtCleanupCallback = AbcEvtCleanup;
Attributes.EvtDestroyCallback = AbcEvtDestroy;

WDF_ABC_CONFIG_INIT( &Config );
WdfAbcCreate( &Attributes,
                   &Config,  &Handle )

Context = GetAbcContext( Handle );
```



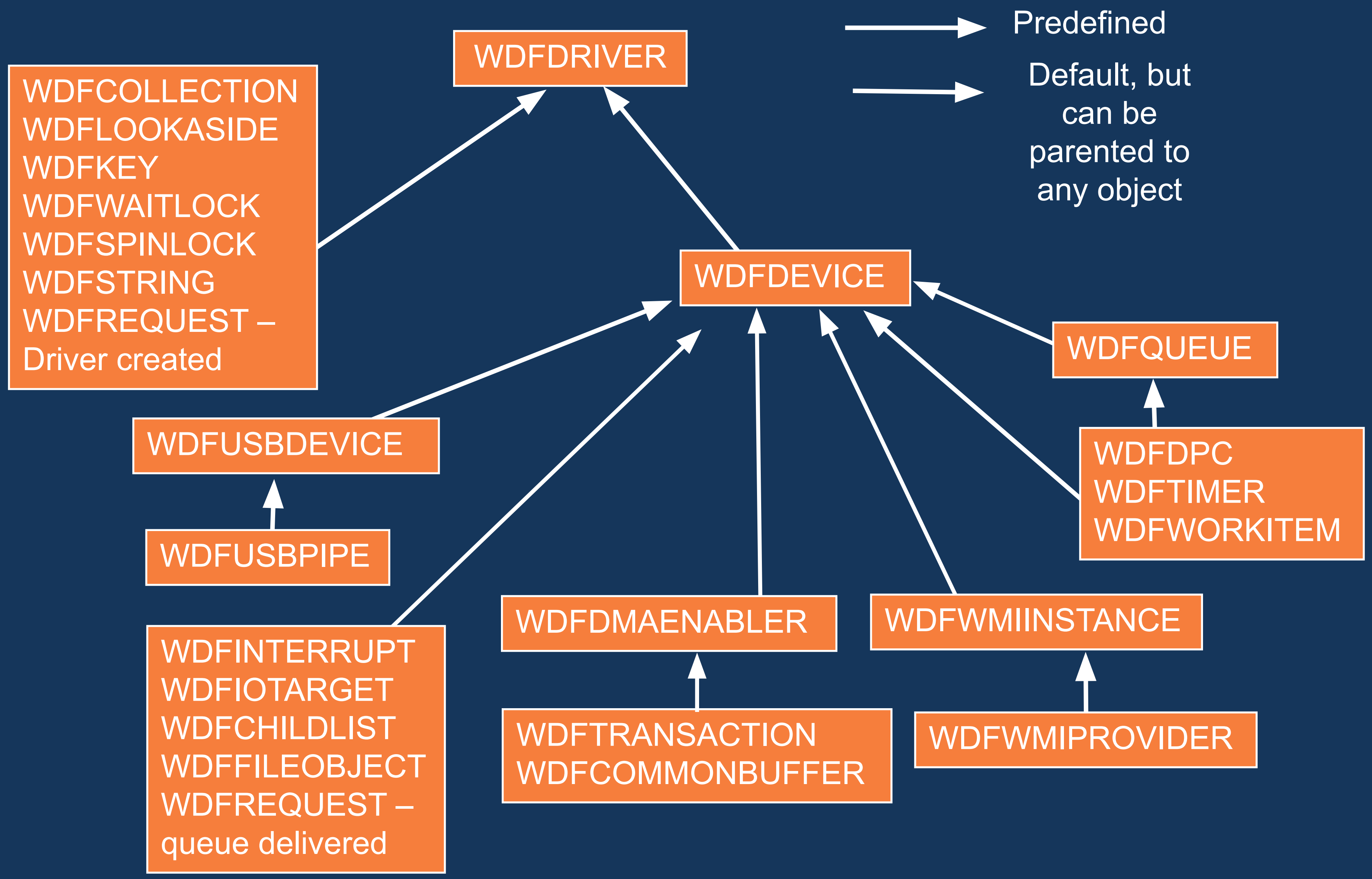
| WDF_OBJECT_ATTRIBUTES |
|---|
| Size |
| EvtCleanupCallback |
| EvtDestroyCallback |
| ExecutionLevel |
| SynchronizationScope |
| ParentObject |
| ContextSizeOverride |
| ContextTypeInfo |

InheritParent
Passive
Dispatch

InheritParentDevice
Object
None

| WDF_ABC_CONFIG |
|---|
| Size |
| EvtCallback |
| Period |
| … |

# Object Relationship



Predefined

Default, but
can be
parented to
any object

WDFDRIVER

WDFCOLLECTION
WDFLOOKASIDE
WDFKEY
WDFWAITLOCK
WDFSPINLOCK
WDFSTRING
WDFREQUEST –
Driver created

WDFDEVICE

WDFQUEUE

WDFDPC
WDFTIMER
WDFWORKITEM

WDFUSBDEVICE

WDFUSBPIPE

WDFINTERRUPT
WDFIOTARGET
WDFCHILDLIST
WDFFILEOBJECT
WDFREQUEST –
queue delivered

WDFDMAENABLER

WDFTRANSACTION
WDFCOMMONBUFFER

WDFWMIINSTANCE

WDFWMIPROVIDER

# Deleting an Object

- WdfObjectDelete() - single delete function to delete all types of objects
- Child objects will be deleted when their parent is deleted
- Some objects cannot be deleted by the driver because the lifetime is controlled by WDF
  - WDFDRIVER
  - WDFDEVICE for FDO and PDO
  - WDFFILEOBJECT
  - WDFREQUEST
  - Etc.

# Mapping – WDF Objects to WDM

| | |
|---|---|
| WDFDRIVER | Driver object |
| WDFDEVICE | Device object |
| WDFQUEUE | Cancel-safe queue/Dispatching /Serialization/Auto-locking/Synch with PnP |
| WDFREQUEST | IRP |
| WDFINTERRUPT | Interrupt |
| WDFDPC | DPC |
| WDFWORKITEM | Work item |
| WDFDMAENABLER | DMA adapter object |
| WDFIOTARGET | Sending I/O to another driver - IoCallDriver |
| WDFWAITLOCK | Event dispatcher object – passive level lock |
| WDFSPINLOCK | Spinlock |
| WDFMEMORY | Kernel pool - refcounted |
| WDFKEY | Registry access |

# Naming Pattern

- Methods:
  - Status = WdfDeviceCreate();

  Object

  Operation
- Properties:
  - Cannot fail
    - WdfInterruptGetDevice();
    - WdfInterruptSetPolicy();
  - Can fail:
    - Status = WdfRegistryAssignValue();
    - Status = WdfRegistryQueryValue();
    - Status = WdfRequestRetrieveInputBuffer();
- Callbacks:
  - PFN_WDF_INTERRUPT_ENABLE EvtInterruptEnable
- Init Macros:
  - WDF_XXX_CONFIG_INIT
  - WDF_XXX_EVENT_CALLBACKS_INIT

# DriverEntry – WDM

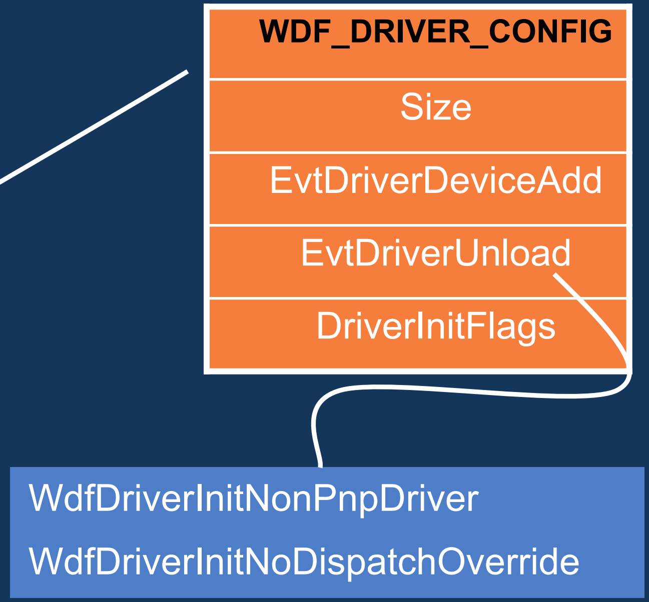
- Called when the driver is first loaded in memory
- Sets Dispatch routines and returns

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject
    IN PUNICODE_STRING RegistryPath
    )
{

    DriverObject->DriverExtension->AddDevice           = AddDevice;
    DriverObject->MajorFunction[IRP_MJ_PNP]            = DispatchPnp;
    DriverObject->MajorFunction[IRP_MJ_POWER]          = DispatchPower;
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] =
DispatchSysControl;

    ….

    return STATUS_SUCCESS;
}
```

# DriverEntry – WDF

- DriverEntry is called when the driver is first loaded in memory
- FxDriverEntry initializes the framework and calls DriverEntryv

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject
    IN PUNICODE_STRING RegistryPath
    )
{
    WDF_DRIVER_CONFIG_INIT( &config
                          ToasterEvtDeviceAdd );


    status = WdfDriverCreate(
                DriverObject
                RegistryPath
                WDF_NO_OBJECT_ATTRIBUTES
                &config
                WDF_NO_HANDLE
            );

    return STATUS_SUCCESS;
}
```



**WDF_DRIVER_CONFIG**

Size

EvtDriverDeviceAdd

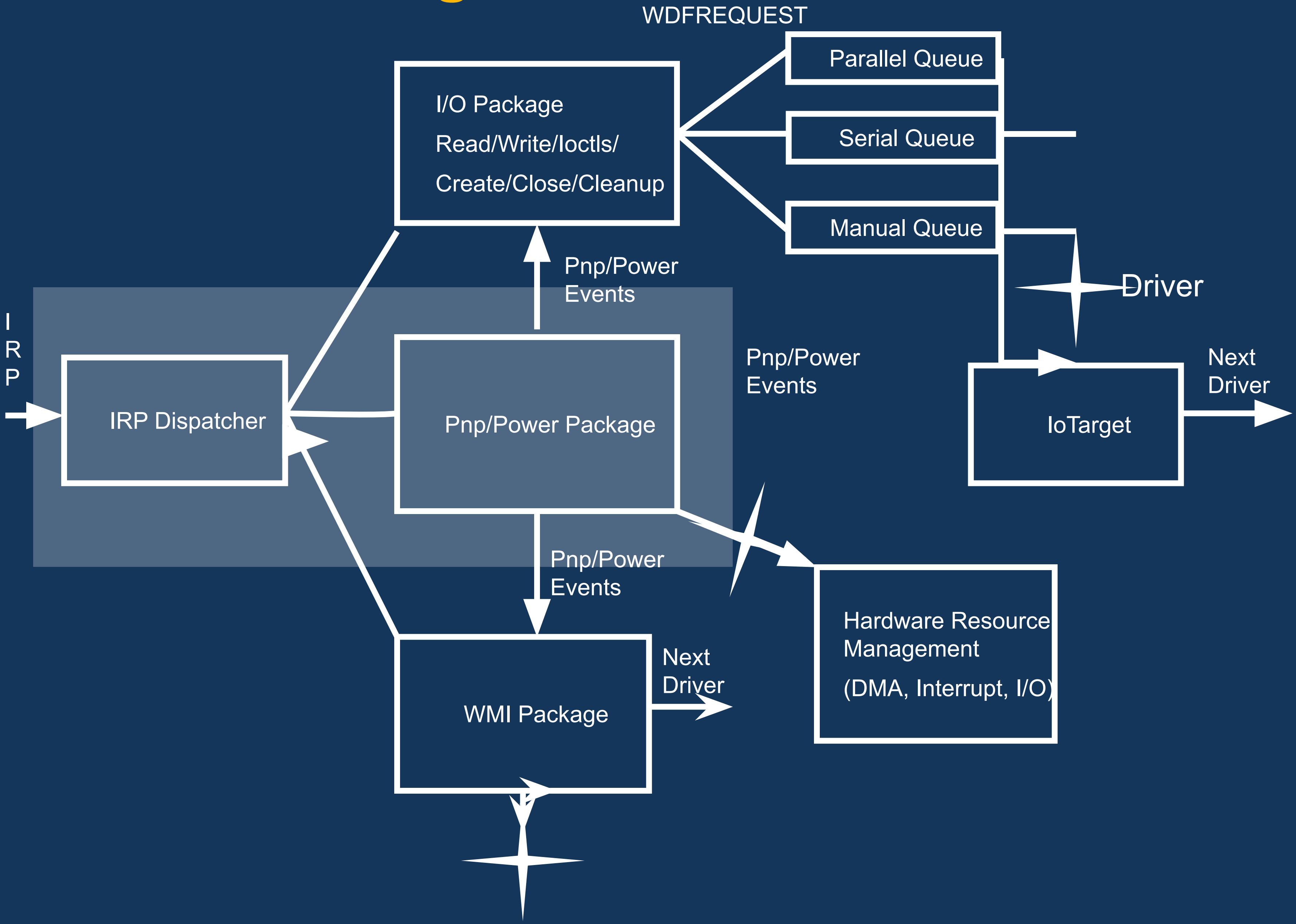
EvtDriverUnload

DriverInitFlags

WdfDriverInitNonPnpDriver

WdfDriverInitNoDispatchOverride

# PnP/Power Stage



# AddDevice – WDM

```
ToasterAddDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject
{
    status = IoCreateDevice (... &deviceObject);

    fdoData = (PFDO_DATA) deviceObject->DeviceExtension;

    fdoData->UnderlyingPDO = PhysicalDeviceObject;

    deviceObject->Flags |= (DO_POWER_PAGABLE | DO_BUFFERED_IO);

    fdoData->NextLowerDriver = IoAttachDeviceToDeviceStack ( );

    IoRegisterDeviceInterface ( &GUID_DEVINTERFACE_TOASTER);

    deviceObject->Flags &= ~DO_DEVICE_INITIALIZING;

    return status;
}
```

# PnP/Power Boilerplate – WDM

```
DispatchPnp (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
   status = IoAcquireRemoveLock (, Irp);

    switch (irpStack->MinorFunction) {
    case IRP_MN_START_DEVICE:
        status = IoForwardIrpSynchronously(, Irp);
        Irp->IoStatus.Status = status;
        IoCompleteRequest (Irp, IO_NO_INCREMENT);
        IoReleaseRemoveLock(, Irp);
        return status;
    case IRP_MN_REMOVE_DEVICE:
        IoReleaseRemoveLockAndWait(, Irp);
        IoSkipCurrentIrpStackLocation(Irp);
        status = IoCallDriver(, Irp);
        IoDetachDevice();
        IoDeleteDevice(DeviceObject);
        return status;
    case IRP_MN_QUERY_STOP_DEVICE:
        status = STATUS_SUCCESS;    break;
    case IRP_MN_CANCEL_STOP_DEVICE:
        status = STATUS_SUCCESS;   break;
    case IRP_MN_STOP_DEVICE:
        status = STATUS_SUCCESS;    break;
    case IRP_MN_QUERY_REMOVE_DEVICE:
```

```
        status = STATUS_SUCCESS;    break;
    case IRP_MN_SURPRISE_REMOVAL:
        status = STATUS_SUCCESS;      break;

    case IRP_MN_CANCEL_REMOVE_DEVICE:
        status = STATUS_SUCCESS;     break;
    default:
        status = Irp->IoStatus.Status;   break;
    }

    Irp->IoStatus.Status = status;
    status = ForwardIrp(NextLowerDriver, Irp);
    return status;
}

NTSTATUS
DispatchPower(
    IN PDEVICE_OBJECT    DeviceObject,
    IN PIRP              Irp
    )
{
    status = IoAcquireRemoveLock (, );
    PoStartNextPowerIrp(Irp);
    IoSkipCurrentIrpStackLocation(Irp);
    status = PoCallDriver(, Irp);
    IoReleaseRemoveLock(, );
    return status;
}
```

# PnP/Power – WDF

- WDF requires that you register zero or more of these callback events, depending on the device, to support pnp/power management
- Rest of this talk is about how and when to register these events, and how they map to WDM irps

| | |
|---|---|
| EvtDeviceD0Entry | EvtDeviceD0Exit |
| EvtDevicePrepareHardware | EvtDeviceReleaseHardware |
| EvtInterruptEnable | EvtInterruptDisable |
| EvtDeviceD0EntryPostInterruptsDisabled | EvtDeviceD0ExitPreInterrutpsDisabled |
| EvtDmaEnablerFill | EvtDmaEnablerFlush |
| EvtDmaEnablerEnable | EvtDmaEnablerDisable |
| EvtDmaEnablerSelfManagedIoStart | EvtDmaEnablerSelfManagedIoStop |
| EvtDeviceArmWakeFromS0 | EvtDeviceDisarmWakeFromS0 |
| EvtDeviceArmWakeFromSx | EvtDeviceDisarmWakeFromSx |
| EvtDeviceWakeFromSxTriggered | EvtDeviceWakeFromS0Triggered |
| EvtDeviceSelfManagedIoInit | EvtDeviceSelfManagedIoCleanup |
| EvtDeviceSelfManagedIoSuspend | EvtDeviceSelfManagedIoRestart |
| EvtIoStop | EvtIoResume |
| EvtDeviceQueryRemove | EvtDeviceQueryStop |
| EvtDeviceSurpriseRemoval | |

# EvtDeviceAdd – Software Driver

WdfDeviceInitSetIoType
WdfDeviceInitSetExclusive
WdfDeviceInitSetPowerNotPageable
WdfDeviceInitSetPowerPageable
WdfDeviceInitSetPowerInrush
WdfDeviceInitSetDeviceType
WdfDeviceInitAssignName
WdfDeviceInitAssignSDDLString
WdfDeviceInitSetDeviceClass
WdfDeviceInitSetCharacteristics

```
NTSTATUS
ToasterEvtDeviceAdd(
    IN WDFDRIVER Driver
    IN PWDFDEVICE_INIT DeviceInit
    )
{

    WdfDeviceInitSetIoType(DevcieInit WdfIoTypeBuffered);

    WDF_OBJECT_ATTRIBUTES_INIT(&fdoAttributes);
    WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE(&fdoAttributes FDO_DATA);

    status = WdfDeviceCreate(&DeviceInit &fdoAttributes &device);

    fdoData = ToasterFdoGetData(device);

    status = WdfDeviceCreateDeviceInterface(&GUID_DEVINTERFACE_TOASTER );

    return status;
}
```

# EvtDeviceAdd – Filter Driver

```
NTSTATUS
FilterEvtDeviceAdd(
    IN WDFDRIVER Driver
    IN PWDFDEVICE_INIT DeviceInit
    )
{
    WdfFdoInitSetFilter(DeviceInit);

    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE(&attributes  FILTER_DATA);

    status = WdfDeviceCreate(&DeviceInit &attributes &device);

    fdoData = FilterGetDeviceContext(device);

    return status;
}
```

# EvtDeviceAdd – Hardware Driver

WdfDeviceInitSetPnpPowerEventCallbacks
WdfDeviceInitSetPowerPolicyEventCallbacks
WdfDeviceInitSetPowerPolicyOwnership
WdfDeviceInitSetIgnoreQueryStopRemove
WdfDeviceInitRegisterPnpStateChangeCallback
WdfDeviceInitRegisterPowerStateChangeCallback
WdfDeviceInitRegisterPowerPolicyStateChangeCallback

```
NTSTATUS
EvtDeviceAdd(
    IN WDFDRIVER        Driver,
    IN PWDFDEVICE_INIT  DeviceInit
    )
{
    WdfDeviceInitSetIoType(DeviceInit, WdfDeviceIoDirect);


    WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpPowerCallbacks);

    pnpPowerCallbacks.EvtDevicePrepareHardware = EvtPrepareHardware;
    pnpPowerCallbacks.EvtDeviceReleaseHardware = EvtReleaseHardware;
    pnpPowerCallbacks.EvtDeviceD0Entry         = EvtDeviceD0Entry;
    pnpPowerCallbacks.EvtDeviceD0Exit          = EvtDeviceD0Exit;

    WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpPowerCallbacks);

    WDF_OBJECT_ATTRIBUTES_INIT(&fdoAttributes);
    WDF_OBJECT_ATTRIBUTES_SET_CONTEXT_TYPE(&fdoAttributes, FDO_DATA);

    fdoAttributes.EvtCleanupCallback = EvtDeviceContextCleanup;

    status = WdfDeviceCreate(&DeviceInit, &fdoAttributes, &device);

    status = NICAllocateSoftwareResources(fdoData);
 ….
    return status;
}
```

# PnP/Power Callbacks

- EvtDevicePrepareHardware
  - One time initialization, first callback where device is in D0
  - Map in memory mapped I/O, inspect hw for revision, features, etc.
- EvtDeviceReleaseHardware
  - One time deinitialization, called when the device is in Dx!
  - Unmap in memory mapped I/O, etc.
- EvtDeviceD0Entry
  - Bring the device into D0, no interrupts connected
- EvtDeviceD0Exit
  - Move the device into Dx, no interrupts connected

# Mapping – WDF Callbacks to WDM IRPs

| | |
|---|---|
| **EvtPrepareHardware** | ↑IRP_MN_START_DEVICE |
| **EvtReleaseHardware** | ↓IRP_MN_STOP_DEVICE<br>↓IRP_MN_SURPRISE_REMOVAL<br>↓IRP_MN_REMOVE_DEVICE |
| **EvtDeviceD0Entry** | ↑IRP_MN_START_DEVICE<br>↑ IRP_MN_SET_POWER – D0 Irp |
| **EvtDeviceD0Exit** | ↓ IRP_MN_SET_POWER – Dx Irp<br>↓IRP_MN_SURPRISE_REMOVAL<br>↓IRP_MN_REMOVE_DEVICE<br>↓IRP_MN_STOP_DEVICE |
| **EvtDeviceContextCleanup** | ↓IRP_MN_REMOVE_DEVICE |

- Up arrow means callback is invoked when the IRP is completed by the lower driver.
- Down arrow means callback is invoked before forwarding the IRP

# Self Managed I/O

- Drivers may want to override automatic WDF queuing behavior by using non-power managed queues
- Drivers may have I/O paths that don’t pass through WDFQUEUEs (timers, DPC, etc.)
- WDF provides a set of callbacks that correspond to state changes
  - EvtDeviceSelfManagedIoInit
  - EvtDeviceSelfManagedIoCleanup
  - EvtDeviceSelfManagedIoSuspend
  - EvtDeviceSelfManagedIoRestart
  - EvtDeviceSelfManagedIoFlush

# Self Managed I/O – Mapping

| | |
|---|---|
| **EvtDeviceSelfManagedIoInit** | START_DEVICE |
| **EvtDeviceSelfManagedIoSuspend** | SURPRISE_REMOVAL or REMOVE, Power-Dx |
| **EvtDeviceSelfManagedIoRestart** | Power – D0, START after STOP |
| **EvtDeviceSelfManagedIoFlush** | REMOVE – For PDO it's called when the PDO is  present |
| **EvtDeviceSelfManagedIoCleanup** | REMOVE -  For PDO it's called when the PDO is about to be deleted (SurpriseRemove) |

- PCIDRV sample uses Self Managed I/O callbacks to start and stop a watchdog timer DPC

# Power Policy Owner

- Default rules on power policy ownership

| Device Type | Policy Owner |
|---|---|
| FDO | Yes |
| Filter | No |
| PDO | No |
| Raw-PDO | Yes |

- Override the default by calling WdfDeviceInitSetPowerPolicyOwnership

# Enabling Wake from Sx

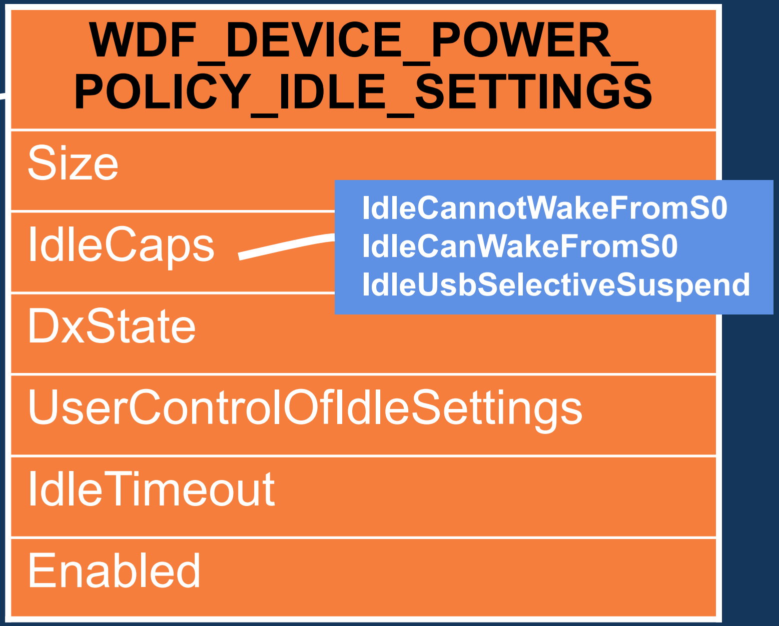
```
WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS
        wakeSettings;
WDF_DEVICE_POWER_POLICY_WAKE_SETTINGS_INIT(
                        &wakeSettings);
status = WdfDeviceAssignSxWakeSettings(Device,
                   &wakeSettings);
```

- Interaction with WMI to present the power management tab in device manager is automatically handled
- Can be called multiple times to change the settings at run-time
- Default is to allow user control

| WDF_DEVICE_POWER_ POLICY_WAKE_SETTINGS |
|---|
| Size |
| DxState |
| UserControlOfIdleSettings |
| Enabled |

# Idle-Time Power Management – S0

```
WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS
           idleSettings;

WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS_
INIT( &idleSettings,IdleCanWakeFromS0 );

idleSettings.IdleTimeout = 10000; // 10-sec

status =  WdfDeviceAssignS0IdleSettings(
              WdfDevice, &idleSettings );
```



**WDF_DEVICE_POWER_ POLICY_IDLE_SETTINGS**

- Size
- IdleCaps
- DxState
- UserControlOfIdleSettings
- IdleTimeout
- Enabled

**IdleCaps:**

- IdleCannotWakeFromS0
- IdleCanWakeFromS0
- IdleUsbSelectiveSuspend

- You can manually stop and resume the IdleTimer by calling WdfDeviceStopIdle or WdDeviceResumeIdle
- WMI interaction is handled automatically
- Can be called multiple times to change the settings

# Power Policy Event Callbacks

```
WDF_POWER_POLICY_EVENT_CALLBACKS    powerPolicyCallbacks;

WDF_POWER_POLICY_EVENT_CALLBACKS_INIT(&ppc);

ppc.EvtDeviceArmWakeFromS0 = PciDrvEvtDeviceWakeArmS0;
ppc.EvtDeviceDisarmWakeFromS0 = PciDrvEvtDeviceWakeDisarmS0;
ppc.EvtDeviceWakeFromS0Triggered = PciDrvEvtDeviceWakeTriggeredS0;

ppc.EvtDeviceArmWakeFromSx = PciDrvEvtDeviceWakeArmSx;
ppc.EvtDeviceDisarmWakeFromSx = PciDrvEvtDeviceWakeDisarmSx;
ppc.EvtDeviceWakeFromSxTriggered = PciDrvEvtDeviceWakeTriggeredSx;

WdfDeviceInitSetPowerPolicyEventCallbacks(Device,
        &powerPolicyCallbacks);
```

# Mapping – Wake Callbacks to Power IRPs

| | |
|---|---|
| Suspend or hibernate - goto Sx | WDF receives IRP_MN_QUERY_POWER Sx<br>WDF receives IRP_MN_SET_POWER Sx<br>WDF sends IRP_MN_SET_POWER Dx<br>WDF sends IRP_MN_WAIT_WAKE<br>EvtDeviceArmWakeFromSx<br>EvtDeviceD0Exit |
| Resume from Sx due to wake event | IRP_MN_WAIT_WAKE (completed by bus)<br>Receives IRP_MN_SET_POWER S0 – fast resume<br>Sends IRP_MN_SET_POWER D0<br>EvtDeviceD0 Entry<br>EvtDeviceWakeFromSxTriggered<br>EvtDeviceDisarmWakeFromSx |
| Idle-out - goto Dx in S0 | Sends IRP_MN_SET_POWER Dx<br>Sends IRP_MN_WAIT_WAKE<br>EvtDeviceArmWakeFromS0<br>EvtDeviceD0Exit |
| Resume from Dx in S0 due to wake event | IRP_MN_WAIT_WAKE (completed by bus)<br>Sends IRP_MN_SET_POWER - D0<br>EvtDeviceD0Entry<br>EvtDeviceWakeFromS0Triggered<br>EvtDeviceDisarmWakeFromS0 |

# Interrupts

```
NTSTATUS
EvtDeviceAdd( )
{
…
 WDF_INTERRUPT_CONFIG_INIT(&Config,
                          NICInterruptHandler,
                          NICDpcForIsr);

Config.EvtInterruptEnable  = NICEvtInterruptEnable;
Config.EvtInterruptDisable = NICEvtInterruptDisable;

status = WdfInterruptCreate(Device,
                     &Config,
                     WDF_NO_OBJECT_ATTRIBUTES,
                     &Interrupt);
}
```

| WDF_INTERRUPT_CONFIG |
|---|
| Size |
| SpinLock |
| ShareVector |
| FloatingSave |
| QueueDpcOnIsrSuccess |
| AutomaticSerialization |
| EvtInterruptIsr |
| EvtInterruptDpc |
| EvtInterruptEnable |
| EvtInterruptDisable |

- WdfInterruptQueueDpcForIsr – to manually queue DpcForIsr
- Register EvtDeviceD0EntryPostInterruptsEnabled and EvtDeviceD0ExitPreInterruptsDisabled to be called at PASSIVE_LEVEL

# FDO and PDO-Specific Callbacks

- Register FDO-specific events by calling WdfFdoInitSetEventCallbacks

| | |
|---|---|
| EvtDeviceFilterAddResourceRequirements | ↓IRP_MN_FILTER_RESOURCE_REQUIREMENTS |
| EvtDeviceFilterRemoveResourceRequirements | ↑IRP_MN_IRP_MN_FILTER_RESOURCE_REQUIREMENTS |
| EvtDeviceRemoveAddedResources | ↓ IRP_MN_START_DEVICE |

- Register PDO-specific events by calling WdfPdoInitSetEventCallbacks

| | |
|---|---|
| EvtDeviceResourcesQuery | ↓ IRP_MN_QUERY_RESOURCE |
| EvtDeviceResourceRequirementsQuery | ↓IRP_MN_QUERY_RESOURCE_REQUIREMENTS |
| EvtDeviceEject | ↓ IRP_MN_EJECT |
| EvtDeviceSetLock | ↓ IRP_MN_SET_LOCK |
| EvtDeviceEnableWakeAtBus | ↓ IRP_MN_WAIT_WAKE |
| EvtDeviceDisableWakeAtBus | ↑ IRP_MN_WAIT_WAKE |

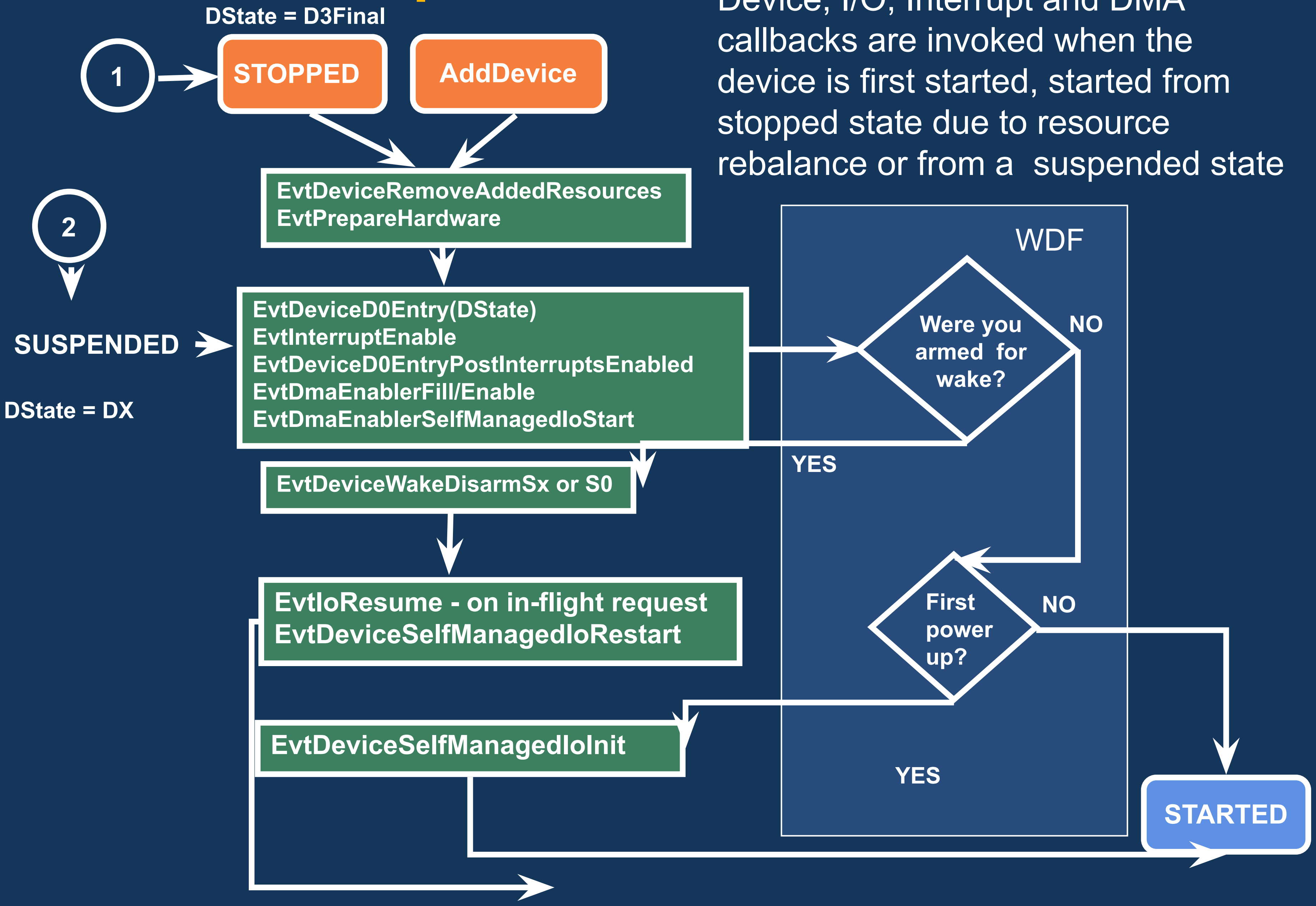
# Summary - Callback Order

- WDF treats PnP and Power as a unified model
- WDF callbacks are based around primitive operations
- Order in which the primitives are called is guaranteed
- Next two slides show the order in which these callback are invoked for start/power-up and remove/suspend
  - You can see the commonalities between pnp & power operation

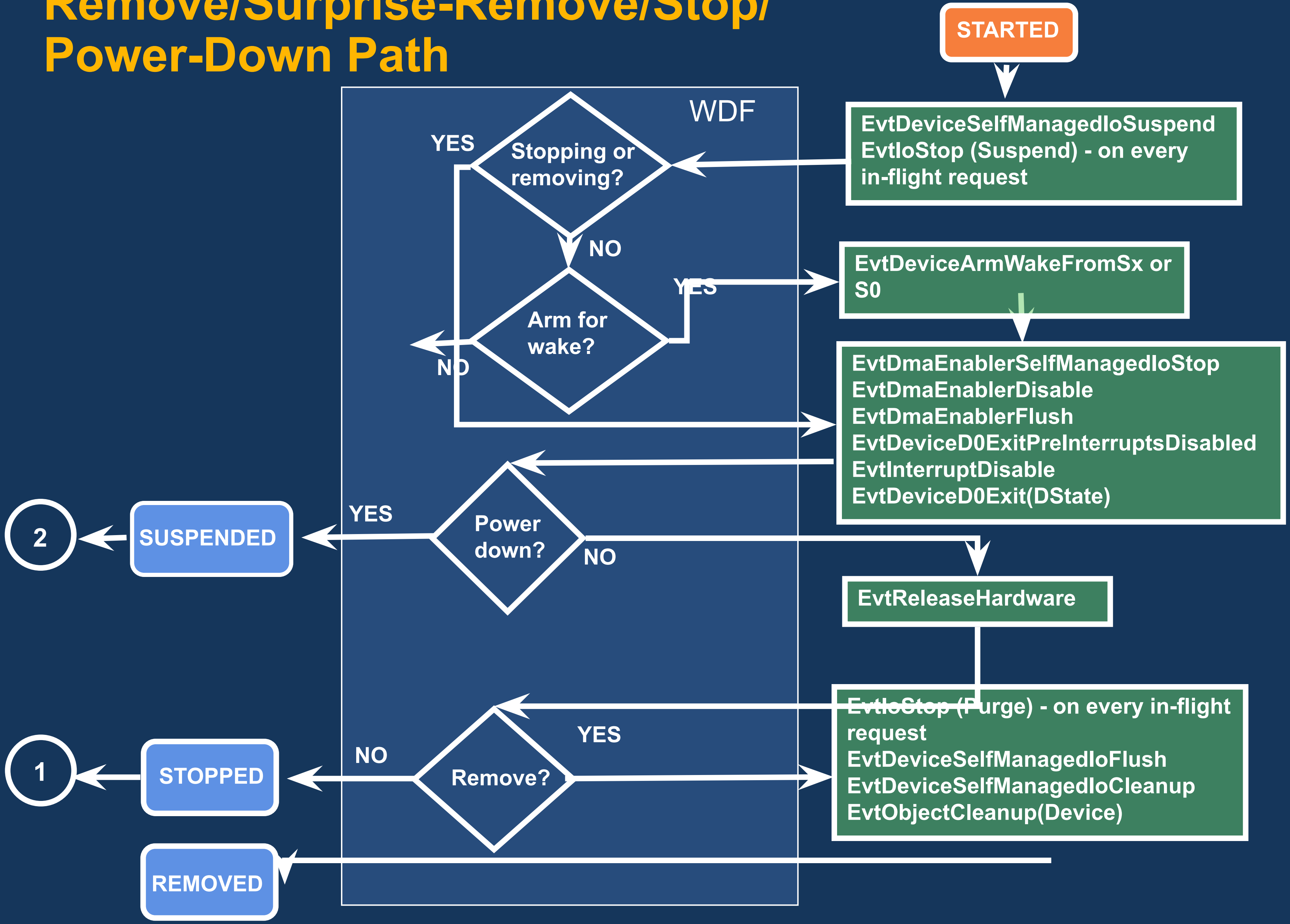
| | |
|---|---|
| EvtDeviceD0Entry | EvtInterruptEnable |
| EvtDeviceD0Exit | EvtInterruptDisable |
| EvtDevicePrepareHardware | EvtDeviceD0EntryPostInterruptsDisabled |
| EvtDeviceReleaseHardware | EvtDeviceD0ExitPreInterrutpsDisabled |
| EvtDeviceQueryRemove | EvtDeviceArmWakeFromS0 |
| EvtDeviceQueryStop | EvtDeviceDisarmWakeFromS0 |
| EvtDeviceSurpriseRemoval | EvtDeviceArmWakeFromSx |
| EvtDeviceSelfManagedIoInit | EvtDeviceDisarmWakeFromSx |
| EvtDeviceSelfManagedIoCleanup | EvtDeviceWakeFromSxTriggered |
| EvtDeviceSelfManagedIoSuspend | EvtDeviceWakeFromS0Triggered |
| EvtDeviceSelfManagedIoRestart | EvtDmaEnablerFill/Flush |
| EvtIoStop | EvtDmaEnablerEnable/Disable |
| EvtIoResume | EvtDmaEnablerSelfManagedIoStart/Stop |

# Start/Power Up Path

This flow chart shows the order Device, I/O, Interrupt and DMA callbacks are invoked when the device is first started, started from stopped state due to resource rebalance or from a  suspended state


DState = D3Final
1
STOPPED
AddDevice
EvtDeviceRemoveAddedResources
EvtPrepareHardware
2
SUSPENDED
DState = DX
EvtDeviceD0Entry(DState)
EvtInterruptEnable
EvtDeviceD0EntryPostInterruptsEnabled
EvtDmaEnablerFill/Enable
EvtDmaEnablerSelfManagedIoStart
WDF
Were you armed  for wake?
NO
YES
EvtDeviceWakeDisarmSx or S0
EvtIoResume - on in-flight request
EvtDeviceSelfManagedIoRestart
First power up?
NO
YES
EvtDeviceSelfManagedIoInit
STARTED

# Remove/Surprise-Remove/Stop/ Power-Down Path



STARTED

EvtDeviceSelfManagedIoSuspend
EvtIoStop (Suspend) - on every in-flight request

Stopping or removing?

Arm for wake?

EvtDeviceArmWakeFromSx or S0

EvtDmaEnablerSelfManagedIoStop
EvtDmaEnablerDisable
EvtDmaEnablerFlush
EvtDeviceD0ExitPreInterruptsDisabled
EvtInterruptDisable
EvtDeviceD0Exit(DState)

Power down?

SUSPENDED

2

EvtReleaseHardware

EvtIoStop (Purge) - on every in-flight request
EvtDeviceSelfManagedIoFlush
EvtDeviceSelfManagedIoCleanup
EvtObjectCleanup(Device)

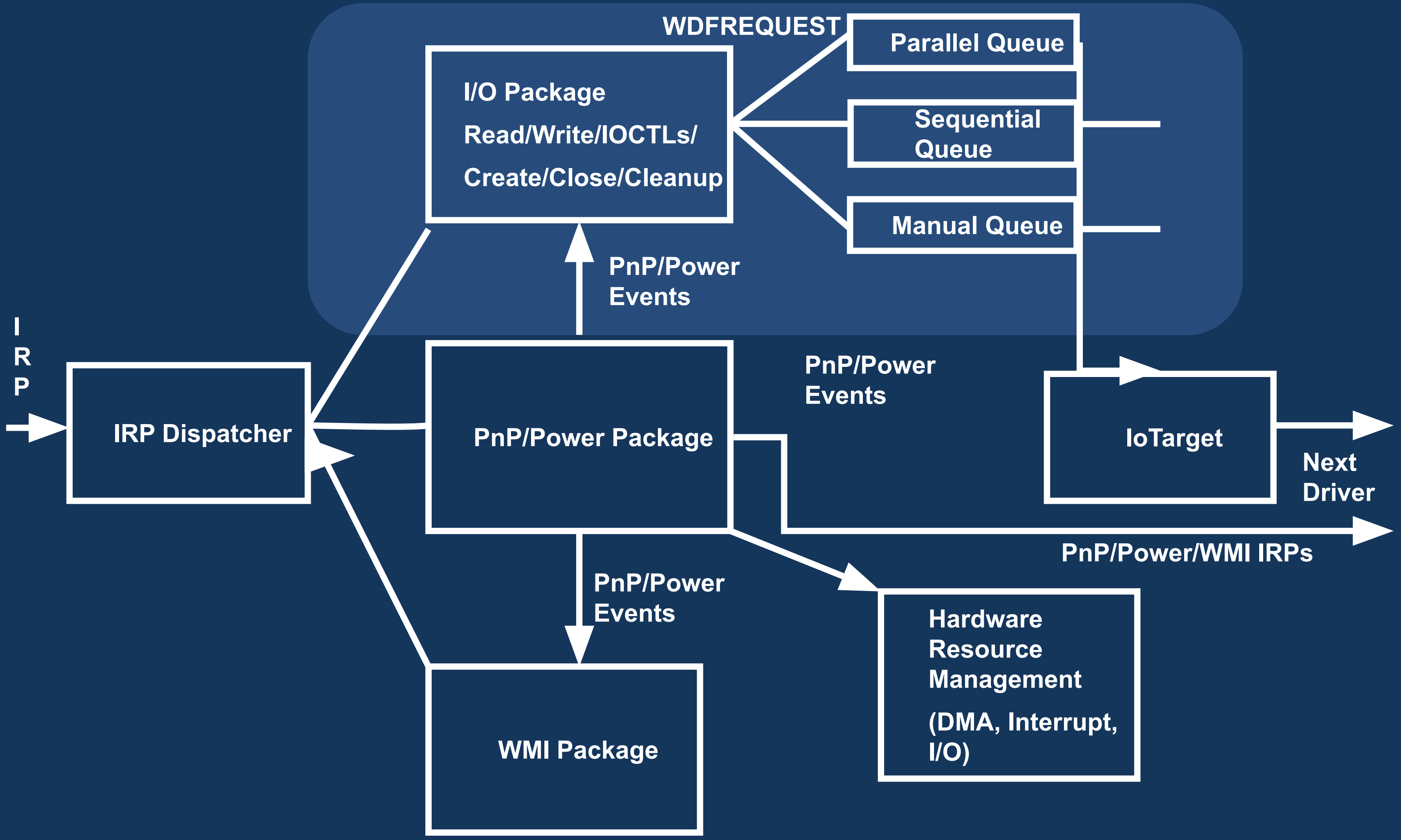
Remove?

STOPPED

1

REMOVED

# I/O Stage



WDFREQUEST

Parallel Queue

I/O Package

Read/Write/IOCTLs/

Create/Close/Cleanup

Sequential Queue

Manual Queue

PnP/Power Events

I R P

IRP Dispatcher

PnP/Power Package

PnP/Power Events

IoTarget

Next Driver

PnP/Power/WMI IRPs

PnP/Power Events

Hardware Resource Management

(DMA, Interrupt, I/O)

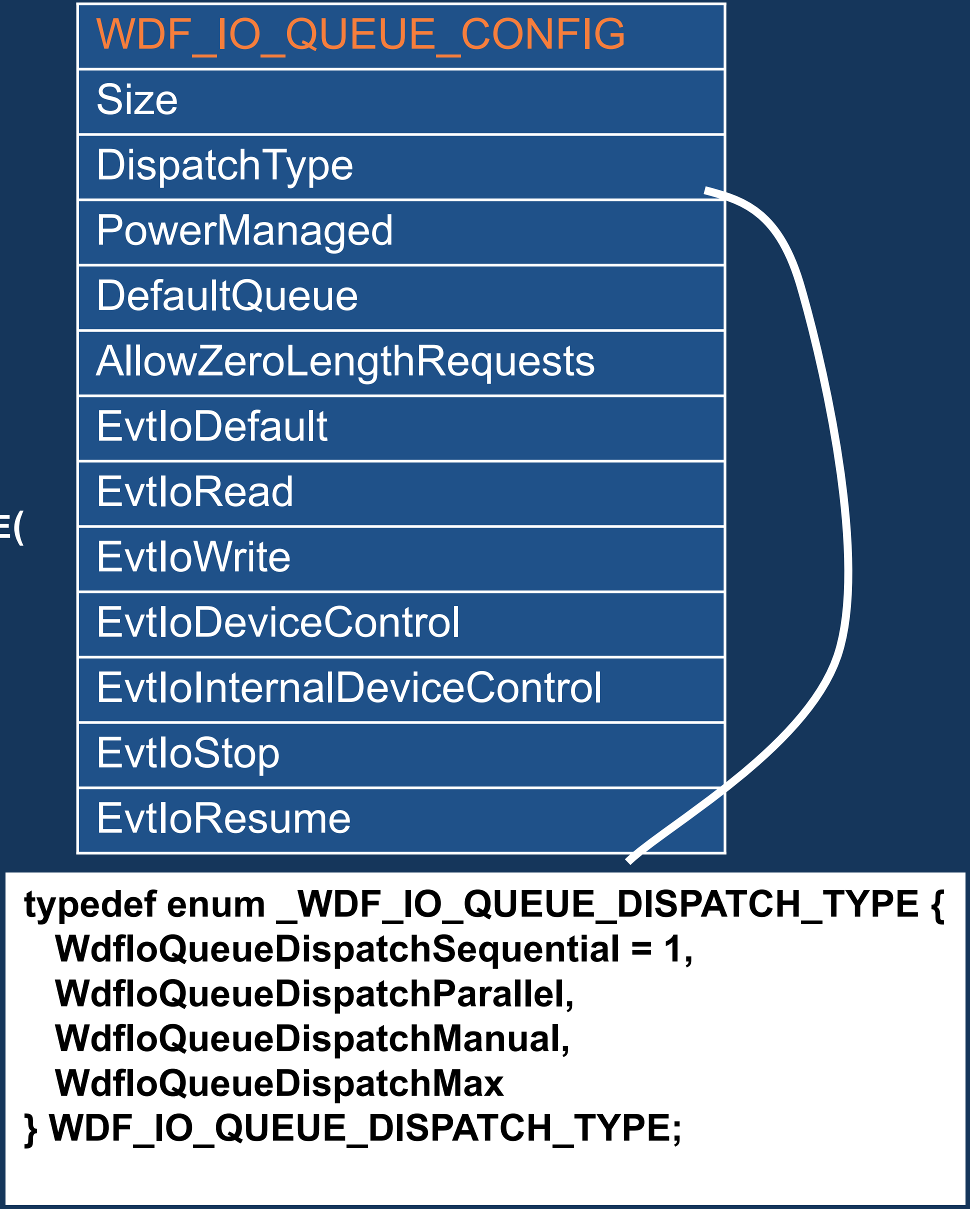
WMI Package

# Queues

- Queue object is used to present WDFREQUEST to the driver
- Only create, read, write, and IOCTL IRPs are converted to WDFREQUEST and presented by queues
- Delivery of requests is based on the queue type
  - Sequential: Requests are delivered one at a time
  - Parallel: Requests are delivered to the driver as they arrive
  - Manual: Driver retrieves requests from the WDQUEUE at its own pace
- WDF_EXECUTION_LEVEL and WDF_SYNCHRONIZATION_SCOPE can be used to control serialization and IRQL level of those callbacks
- WDFQUEUE is more than a list of pending requests!

# Creating a Queue

```
NTSTATUS
EvtDeviceAdd(
    IN WDFDRIVER        Driver,
    IN PWDFDEVICE_INIT  DeviceInit
    )
{

….

    WDF_IO_QUEUE_CONFIG_INIT_DEFUALT_QUEUE(
         &Config,
         WdfIoQueueDispatchParallel );

    Config.EvtIoStart = PciDrvEvtIoStart;
    Config.AllowZeroLengthRequests = TRUE;

    status = WdfIoQueueCreate(
                 WdfDevice,
                 &Config,
                 WDF_NO_OBJECT_ATTRIBUTES,
                 &Queue // queue handle
             );

    return status;
}
```



WDF_IO_QUEUE_CONFIG

Size

DispatchType

PowerManaged

DefaultQueue

AllowZeroLengthRequests

EvtIoDefault

EvtIoRead

EvtIoWrite

EvtIoDeviceControl

EvtIoInternalDeviceControl

EvtIoStop

EvtIoResume

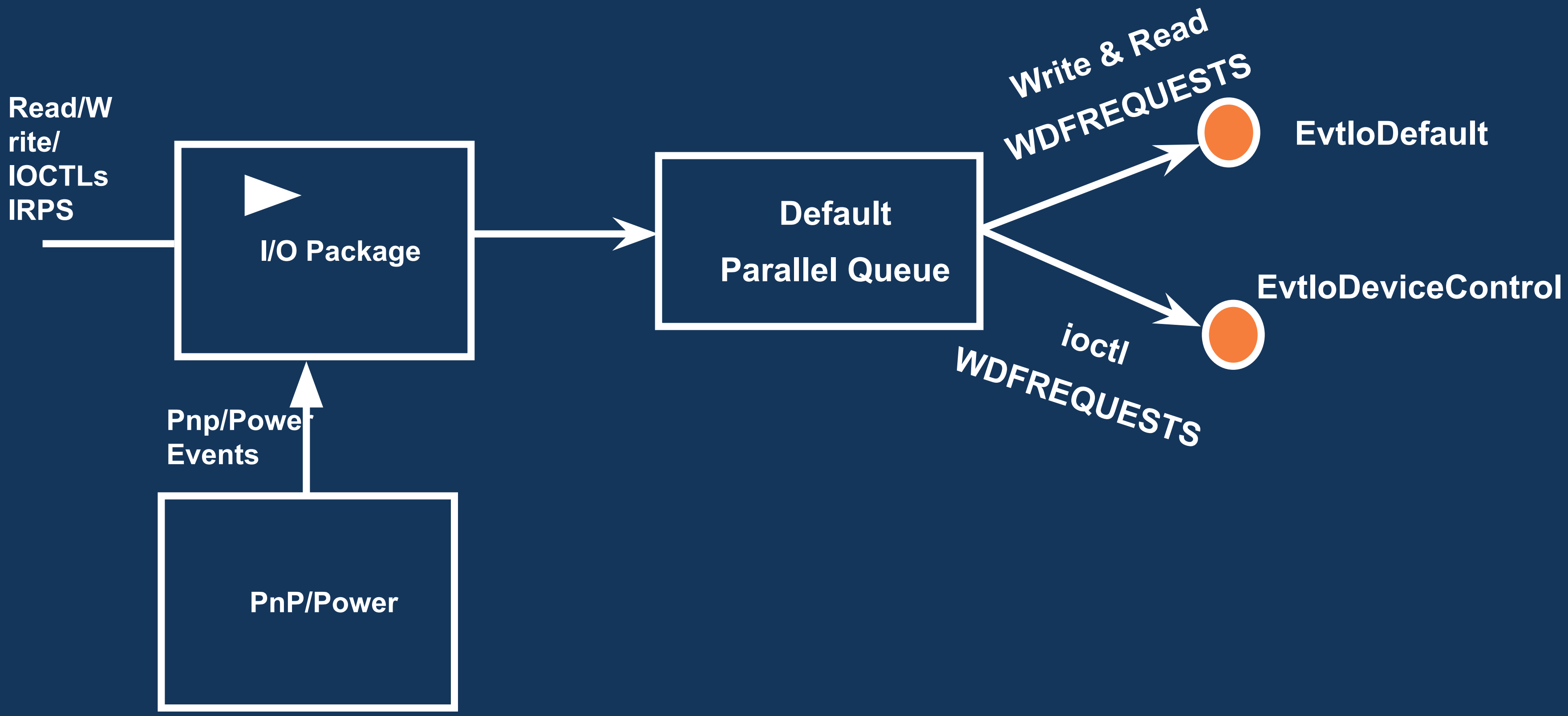
```
typedef enum _WDF_IO_QUEUE_DISPATCH_TYPE {
   WdfIoQueueDispatchSequential = 1,
   WdfIoQueueDispatchParallel,
   WdfIoQueueDispatchManual,
   WdfIoQueueDispatchMax
} WDF_IO_QUEUE_DISPATCH_TYPE;
```

# WDFQUEUE Events

- EvtIoDefault – Called for any request that does not have a specific callback registered
- EvtIoRead – Called for IRP_MJ_READ requests
- EvtIoWrite – Called for IRP_MJ_WRITE requests
- EvtIoDeviceControl – Called for IRP_MJ_DEVICE_CONTROL
- EvtIoInternalDeviceControl – Called for IRP_MJ_INTERNAL_DEVICE_CONTROL requests
- EvtIoStop – Called for all inflight requests when a power down transition occurs
- EvtIoResume - Called for all inflight requests when a power up transition occurs

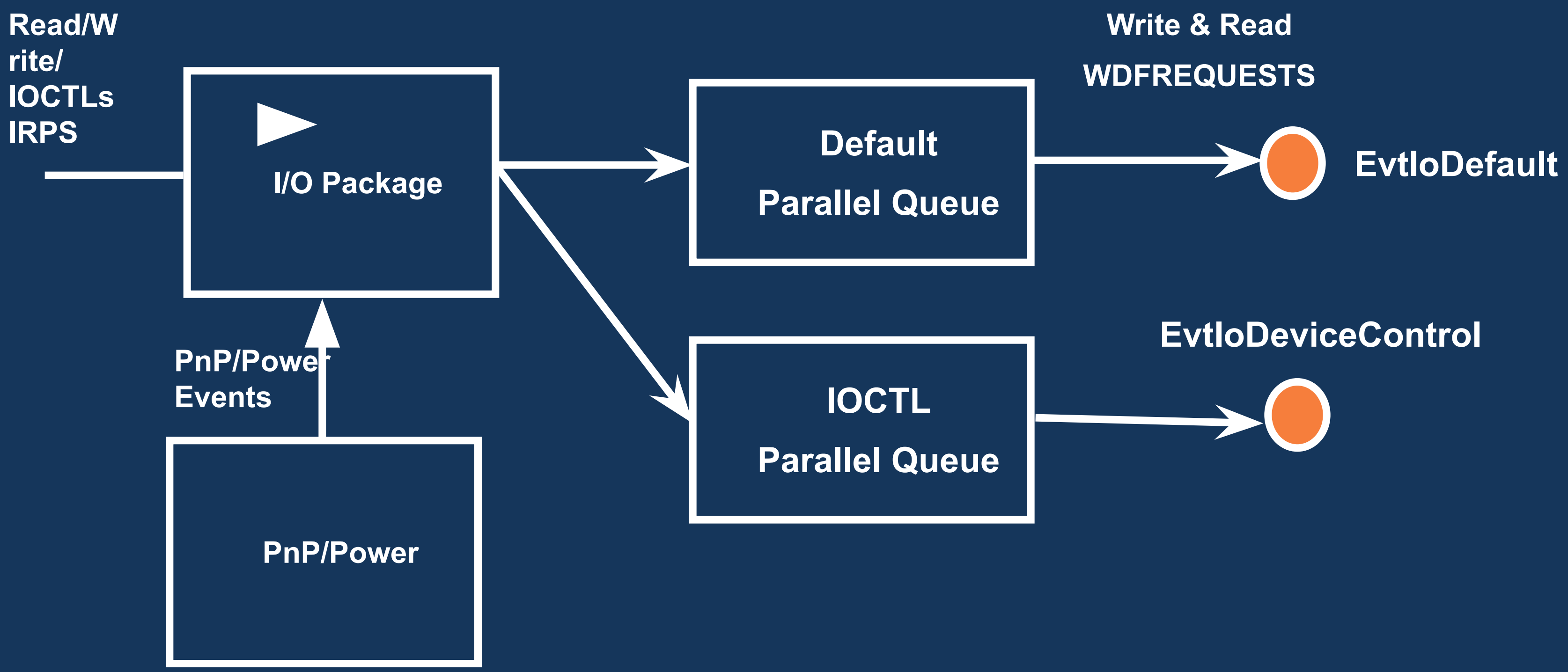
# Default Queue

- Default queue receives all requests that are not configured to go to other queues
- There can be only one default queue per device



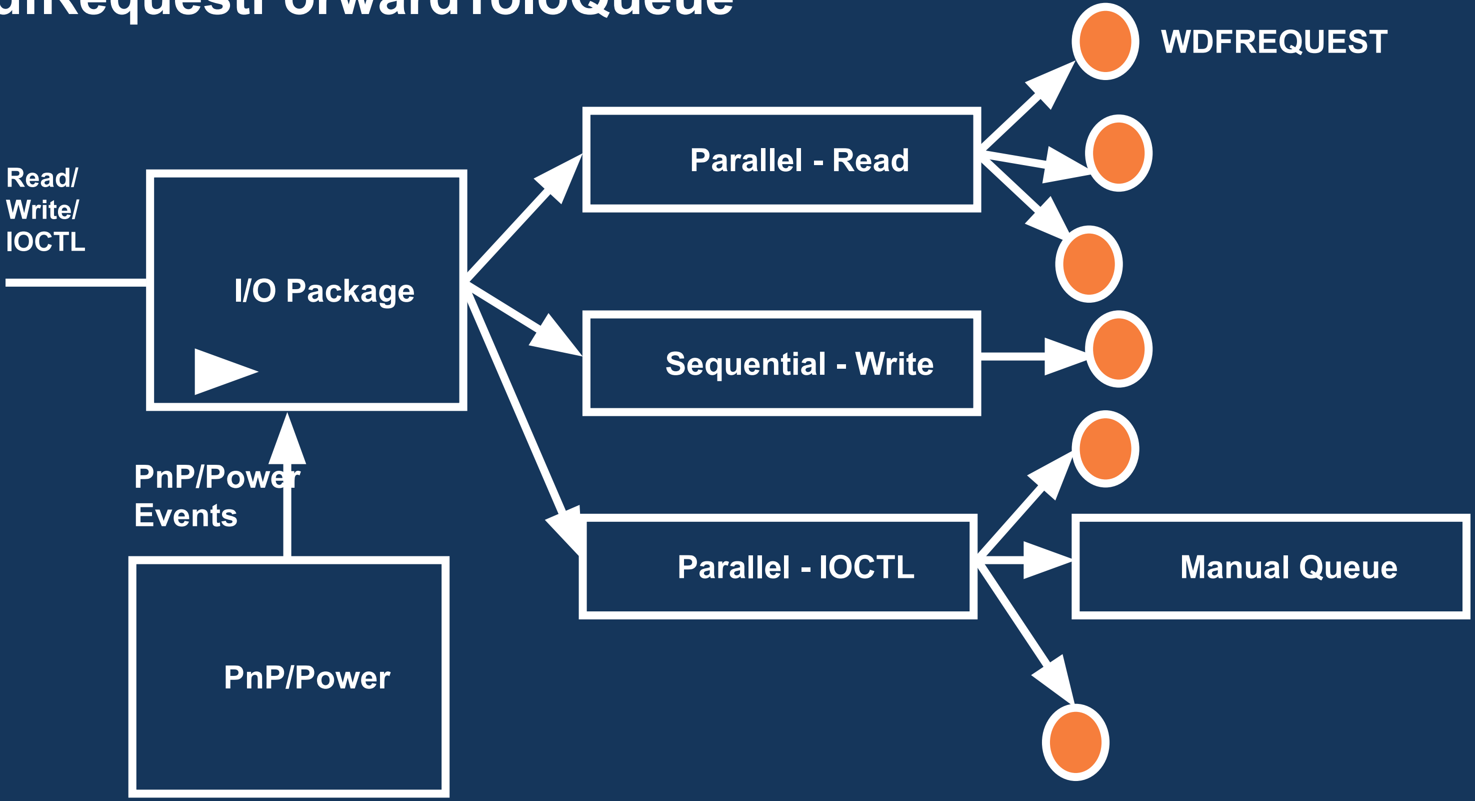
# Preconfigured Queue

- Preconfigure the queue by calling **WdfDeviceConfigureRequestDispatching** to automatically forward requests based on the I/O type



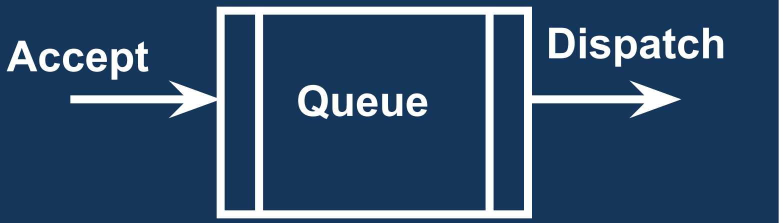
# Multiple queues

- Manually forward requests by calling **WdfRequestForwardToIoQueue**



# Queue State

- Queue state is determined by whether it’s accepting and dispatching requests to the driver



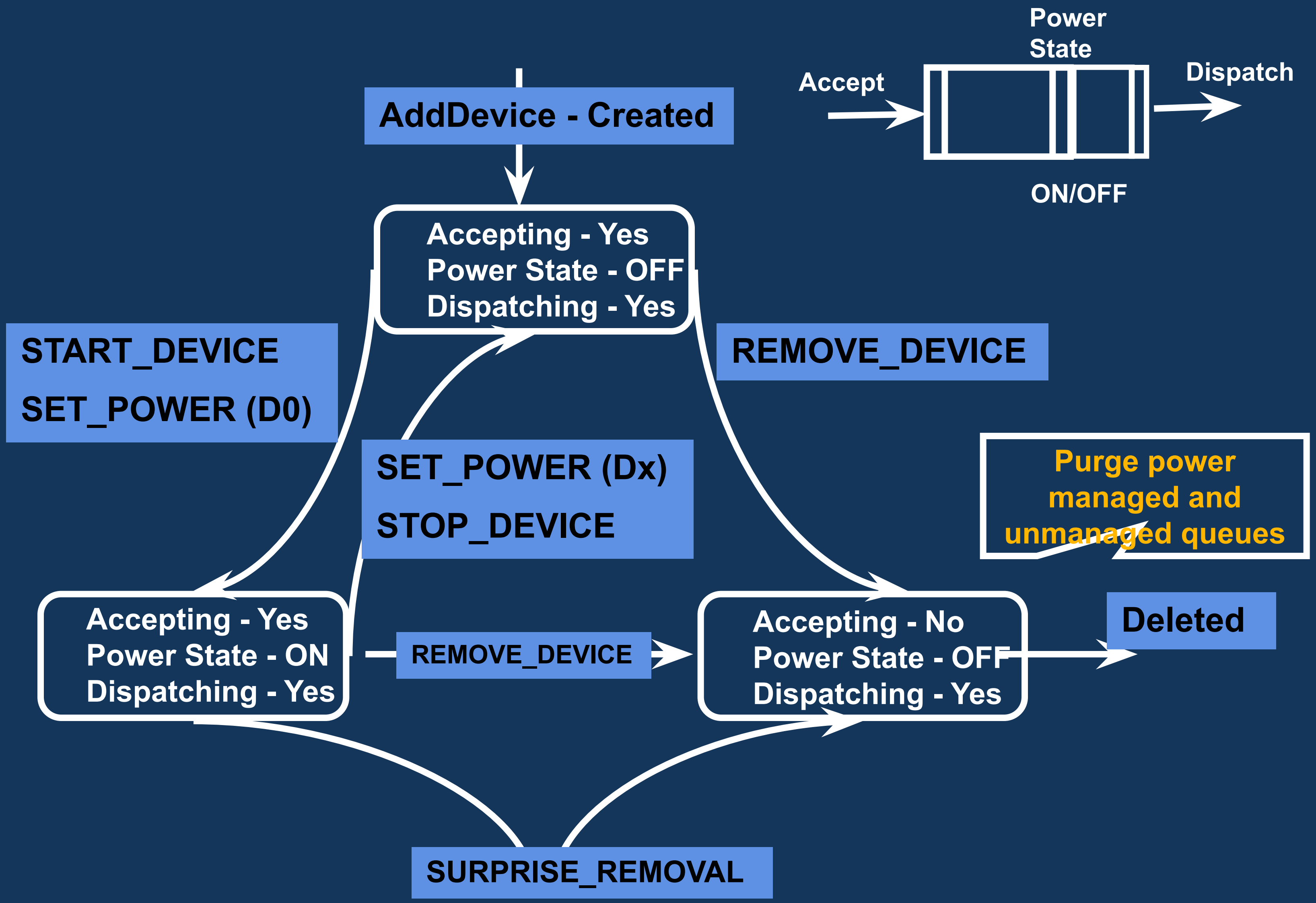
| | | |
|---|---|---|
| Started | Accepting – Y | Dispatching - Y |
| Stopped | Accepting – Y | Dispatching - N |
| Draining | Accepting – N | Dispatching - Y |
| Purging | Accepting – N | Dispatching - N |

- For non-power managed queue, driver controls the state of the queue
  - Queue can be moved to any state from any state
- For power managed queue, state change happens due to PnP/Power events

# DDIs for Changing Queue States

| | |
|---|---|
| WdfIoQueueStart | Accept and dispatch requests |
| WdfIoQueueStop | Accept and queue requests |
| WdfIoQueueStopSynchronously | Accept and queue requests, and wait for the driver-owned request to complete before returning to the caller |
| WdfIoQueueDrain | Fail new requests and dispatch queued requests |
| WdfIoQueueDrainSynchronously | Fail new requests, dispatch queued requests and wait for all the requests to complete before returning to the caller |
| WdfIoQueuePurge | Fail new requests, cancel queued requests, cancel in-flight requests (if marked cancelable) |
| WdfIoQueuePurgeSynchronously | Fail new requests, cancel queued requests, cancel in-flight requests (if they are marked cancelable), and wait for all the requests to complete before returning to the caller |

# Power Managed Queue



Power State

Accept

Dispatch

ON/OFF

AddDevice - Created

Accepting - Yes
Power State - OFF
Dispatching - Yes

START_DEVICE

SET_POWER (D0)

REMOVE_DEVICE

SET_POWER (Dx)

STOP_DEVICE

Purge power managed and unmanaged queues

Accepting - Yes
Power State - ON
Dispatching - Yes

REMOVE_DEVICE

Accepting - No
Power State - OFF
Dispatching - Yes

Deleted

SURPRISE_REMOVAL

# Create/Cleanup/Close

- Register during device initialization if you are interested in handling Create, Close and Cleanup requests
- WDF by default succeeds these requests if you don’t register a callback

```
WDF_DEVICE_FILE_OBJECT_INIT(
&fileObjConfig,
           FileIoEvtDeviceFileCreate,
           FileIoEvtDeviceClose,
           WDF_NO_EVENT_CALLBACK);

 WdfDeviceInitSetFileObjectConfig(DeviceInit,
             &fileObjConfig,
             WDF_NO_OBJECT_ATTRIBUTES);
```

| |
|---|
| Size |
| AutoForwardCleanupClose |
| EvtDeviceFileCreate |
| EvtFileClose |
| EvtFileCleanup |
| FileObjectClass |

```
EvtDeviceFileCreate(
   WDFDEVICE Device,
   WDFREQUEST Request,
   WDFFILEOBJECT FileObject
)
```

```
EvtFileCleanup(WDFFILEOBJECT
FileObject)

EvtFileClose(WDFFILEOBJECT FileObject);
```

# Create/Close/Cleanup

- Create request
  - You can pend, forward it another queue, send it to an IoTarget
  - You can configure to auto-dispatch create to a specific queue
  - EvtIoDefault callback is invoked when a create request is dispatched by a queue
- Cleanup/Close
  - WDF doesn’t provide a request for these events
  - If you send create requests down the stack, you must set the AutoForwardCleanupClose  property so that WDF can forward Cleanup and Close requests
  - For filters, if the callbacks are not registered, WDF will auto-forward Create, Close and Cleanup

# Request Cancellation

- Requests waiting in the queue to be delivered to the driver are automatically cancelable
- In-flight requests cannot be canceled unless explicitly made cancelable by calling
  - WdfRequestMarkCancelable(Request, EvtRequestCancel)
- A request should be made cancelable by the driver if:
  - The I/O is going to take long time to complete
  - The I/O operation on the hardware can be stopped in mid-operation
- A cancelable request must be unmarked (WdfRequestUnmarkCancelable) before completion unless it's completed by the cancel routine
- These rules are similar to WDM

# Read/Write/IOCTL Callbacks

```
VOID
EvtIoRead(
   IN WDFQUEUE      Queue,
   IN WDFREQUEST Request,
   IN size_t           Length
   )
```

```
VOID
EvtIoWrite(
   IN WDFQUEUE          Queue,
   IN WDFREQUEST        Request,
   IN size_t            Length
   )
```

```
VOID
EvtIoDeviceControl(
   IN WDFQUEUE         Queue,
   IN WDFREQUEST       Request,
   IN size_t           OutputBufferLength,
   IN size_t           InputBufferLength,
   IN ULONG            IoControlCode
   )
```

# Request Buffers

- Getting input buffer
  - WdfRequestRetrieveInputBuffer
  - WdfRequestRetrieveInputMemory
  - WdfRequestRetrieveInputWdmMdl
- Getting output buffer
  - WdfRequestRetrieveOutputBuffer
  - WdfRequestRetrieveOutputMemory
  - WdfRequestRetrieveOutputWdmMdl
- ‘Input’ or ‘Output’ denotes the direction of memory access
  - Input: read from memory and write to device
  - Output: read from device and write to memory

# Retrieve Buffer of Read Request

| Function | Read - Buffered | Read - Direct |
|---|---|---|
| WdfRequestRetrieveOutputBuffer | Return Irp->AssociatedIrp. SystemBuffer | Return SystemAddressForMdl( Irp->MdlAddress) |
| WdfRequestRetrieveOutputWdmMdl | Build an MDL for Irp->AssociatedIrp.SystemBuffer and return the MDL. | Return Irp->MdlAddress |
| WdfRequestRetrieveOuputMemory | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you Irp->AssociatedIrp.SystemBuffer | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you SystemAddressFor( Irp->MdlAddress). |

- Calling WdfRequestRetrieveInputXxx functions on Read request will return STATUS_INVALID_DEVICE_REQUEST error.

# Retrieve Buffer of Write Request

| Function | Read - Buffered | Read - Direct |
|---|---|---|
| WdfRequestRetrieve InputBuffer | Return Irp->AssociatedIrp. SystemBuffer | Return SystemAddressForMdl( Irp->MdlAddress) |
| WdfRequestRetrieve InputWdmMdl | Build an MDL for Irp->AssociatedIrp.SystemBuffer and return the MDL. | Return Irp->MdlAddress |
| WdfRequestRetrieve InputMemory | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you Irp->AssociatedIrp.SystemBuffer | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you SystemAddressFor( Irp->MdlAddress). |

- Calling WdfRequestRetrieveOutputXxx functions on Write request will return STATUS_INVALID_DEVICE_REQUEST error

# Retrieve Buffers of IOCTL Request

| Function | Buffered - IOCTL |
|---|---|
| WdfRequestRetrieveInputBuffer | Return Irp->AssociatedIrp. SystemBuffer |
| WdfRequestRetrieveInputWdmMdl | Build an MDL for Irp->AssociatedIrp. SystemBuffer and return the MDL |
| WdfRequestRetrieveInputMemory | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you Irp->AssociatedIrp.SystemBuffer |
| WdfRequestRetrieveOutputBuffer | Return Irp->AssociatedIrp. SystemBuffer |
| WdfRequestRetrieveOutputWdmMdl | Build an MDL for Irp->AssociatedIrp. SystemBuffer and return the MDL |
| WdfRequestRetrieveOutputMemory | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you Irp->AssociatedIrp.SystemBuffer |

# Retrieve Buffers of IOCTL Request (con’t)

| Function | Buffered - IOCTL |
|---|---|
| WdfRequestRetrieveInputBuffer | Return Irp->AssociatedIrp.SystemBuffer |
| WdfRequestRetrieveInputWdmMdl | Build an mdl for Irp->AssociatedIrp.SystemBuffer and return the MDL |
| WdfRequestRetrieveInputMemory | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give you Irp->AssociatedIrp.SystemBuffer |
| WdfRequestRetrieveOutputBuffer | Return SystemAddressForMdl(Irp->MdlAddress ) |
| WdfRequestRetrieveOutputWdmMdl | Return Irp->MdlAddress |
| WdfRequestRetrieveOutputMemory | WdfMemoryBufferGetBuffer on the returned WDFMEMORY will give SystemAddressFor(<br>Irp->MdlAddress) |

# METHOD_NEITHER Requests

- To handle this type of request, you must register EvtIoInCallerContext callback by calling WdfDeviceInitSetIoInCallerContextCallback
- Callback is invoked in the calling thread context
- Retrieve buffers using
  - WdfRequestRetrieveUnsafeUserInputBuffer
  - WdfRequestRetrieveUnsafeUserOutputBuffer
  - Lock using WdfRequestProbeAndLockUserBufferForRead/Write

| | InputBuffer | OutputBuffer |
|---|---|---|
| Read | Error | Irp->UserBuffer |
| Write | Irp->UserBuffer | Error |
| IOCTL | irpStack->Parameters.<br>DeviceIoControl.Type3InputBuffer | Irp->UserBuffer |

# Timer/DPC/Work Item

| | |
|---|---|
| WDFTIMER | KTIMER<br>(KeInitializeTimerEx) |
| WDFDPC | KDPC<br>(KeInitializeDpc) |
| WDFWORKITEM | IO_WORKITEM<br>(IoAllocateWorkItem) |

- Value add
  - Allows you to synchronize execution with the callback events of a specific queue (by parenting to WDFQUEUE) or all queues (by parenting to WDFDEVICE)
  - Ensures callbacks events are not invoked after the object is deleted – rundown protection
  - Ensures that object is not deleted until the callback has run to completion
  - Enables you to have private context

# DPC

```
NTSTATUS
EvtDeviceAdd( )
{
…
 WDF_DPC_CONFIG_INIT(&config, EvtDpc);

config.AutomaticSerialization = TRUE;

WDF_OBJECT_ATTRIBUTES_INIT(&attributes);

attributes.ParentObject = device;

status = WdfDpcCreate(&Config,
                      &attributes,
                      &hDpc);
}
```

| WDF_DPC_CONFIG |
|---|
| Size |
| EvtDpcFunc |
| DriverWdmDpc |
| AutomaticSerialization |

| WdfDpcCreate() |
|---|
| WdfDpcEnqueue() |
| WdfDpcCancel(Wait) |
| WdfDpcGetParentObject() |
| WdfDpcWdmGetDpc() |
| WdfObjectDelete() |

# Timer

```
NTSTATUS
EvtDeviceAdd( )
{
…
 WDF_TIMER_CONFIG_INIT(&config, EvtTimer);

config.AutomaticSerialization = TRUE;

WDF_OBJECT_ATTRIBUTES_INIT(&attributes);

attributes.ParentObject = device;

status = WdfTimerCreate(&Config,
                        &attributes,
                        &hTimer);
}
```

| WDF_TIMER_CONFIG |
|---|
| Size |
| EvtTimerFunc |
| Period |
| AutomaticSerialization |

| |
|---|
| WdfTimerCreate() |
| WdfTimerStart() |
| WdfTimerStop(Wait) |
| WdfTimerGetParentObject() |
| WdfObjectDelete() |

# Work Item

```
NTSTATUS
EvtDeviceAdd( )
{
…
 WDF_WORKITEM_CONFIG_INIT(&config,
                    EvtWorkItem);

config.AutomaticSerialization = TRUE;

WDF_OBJECT_ATTRIBUTES_INIT(&attributes);

attributes.ParentObject = device;

status = WdfWorkItemCreate(&Config,
                      &attributes,
                      &hWorkItem);
}
```

| WDF_TIMER_CONFIG |
|---|
| Size |
| EvtWorkItemFunc |
| AutomaticSerialization |

| |
|---|
| WdfWorkItemCreate() |
| WdfWorkItemEnqueue() |
| WdfWorkItemFlush() |
| WdfWorkItemGetParentObject() |
| WdfObjectDelete() |

# Locks

- Framework provides two kinds of locks:
  - WDFWAITLOCK
    - Synchronize access to resources at IRQL < DISPATCH_LEVEL
  - WDFSPINLOCK –
    - Synchronize access to resources at IRQL <= DISPATCH_LEVEL
- Value add
  - Has its own deadlock detection support
  - Tracks acquisition history
  - WaitLock protects against thread suspension
  - You can have private context specific to lock

Mapping

| WDF | WDM |
|---|---|
| WdfWaitLockCreate | KeInitializeEvent (SychronizationEvent) |
| WdfWaitLockAcquire (Optional - TimeOut) | KeEnterCriticalRegion<br>KeWaitForSingleObject |
| WdfWaitLockRelease | KeSetEvent<br>KeLeaveCriticalRegion |

| WDF | WDM |
|---|---|
| WdfSpinLockCreate | KeInitializeSpinLock |
| WdfSpinLockAcquire | KeAcquireSpinLock |
| WdfSpinLockRelease | KeReleaseSpinLock |

# Synchronization Scope & Execution Level

| WDF_EXECUTION_LEVEL |
|---|
| WdfExecutionLevelInheritFromParent |
| WdfExecutionLevelPassive |
| WdfExecutionLevelDispatch |

| WDF_SYNCHRONIZATION_SCOPE |
|---|
| WdfSynchronizationScopeInheritFromParent |
| WdfSynchronizationScopeDevice |
| WdfSynchronizationScopeObject |
| WdfSynchronizationScopeNone |

- WdfExecutionLevelPassive
  - Callbacks will be invoked at PASSIVE_LEVEL
  - Can be set only on device, queue and fileobject
  - Creation of timer and DPC with AutomaticSerialization won’t be allowed if this attribute is set on its parent which could be device or queue
- WdfSynchronizationScopeDevice
  - Callback events of queue, fileobject, timer, dpc, & workitem will be synchronized by a common lock
  - Choice of lock depends on the execution level (fast mutex or spinlock)
- WdfSynchronizationScopeObject
  - Can be set only on queue if you want all the callbacks of a queue to be serialized with its own lock

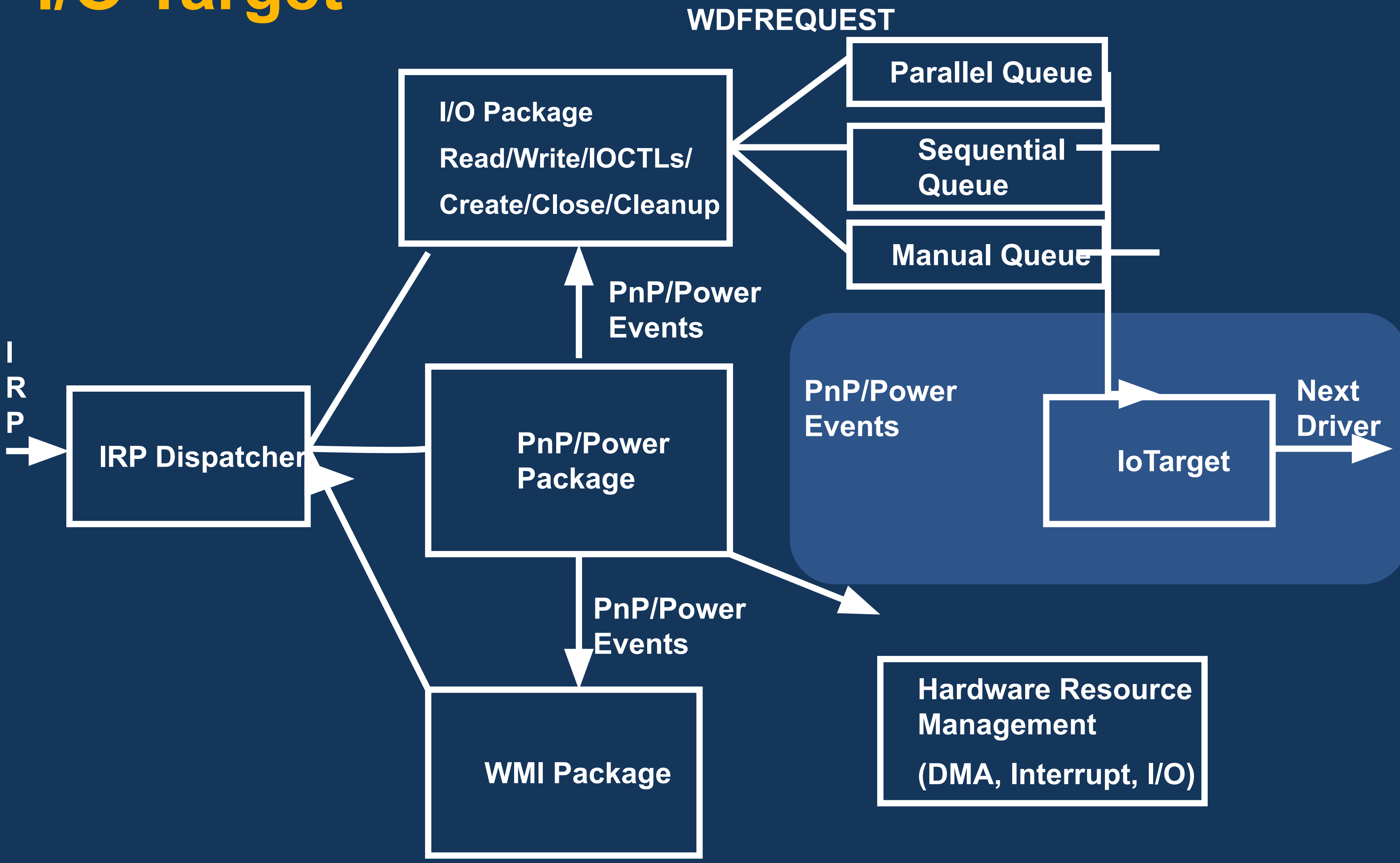
# Sample Scenario – Serial

```
DriverEntry() {
   WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
   attributes.SynchronizationScope = WdfSynchronizationScopeDevice;
   status = WdfDriverCreate(,,&attributes,,);
}
EvtDeviceAdd()
{
    WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&queueConfig, Parallel);
    queueConfig.EvtIoRead    = SerialEvtIoRead;
    queueConfig.EvtIoWrite  = SerialEvtIoWrite;

    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    attributes.SynchronizationScope = WdfSynchronizationScopeDevice;
    status = WdfIoQueueCreate(, queueConfig,&attributes, );

    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    attributes.ParentObject = Device;
    WDF_TIMER_CONFIG_INIT(&timerConfig,   SerialTimeoutXoff);
    timerConfig.AutomaticSerialization = TRUE;

    status = WdfTimerCreate(&timerConfig,  &attributes,);
}
```

# Synchronization Scope & Execution Level - Summary

| **Object** | **Locks**: what kind and who provides it? |
|---|---|
| WDFDRIVER | If you specify SynchScopeObject, framework acquires fast mutex when it calls EvtDeviceAdd. Since there is no parent for WDFDRIVER, the SynchScopeInheritFromParent is same as SynchScopeNone. If SynchScopeDevice is used, all the devices will be created with SynchScopeDevice attributes. |
| WDFDEVICE | Depending on the ExecutionLevel, this object provides a spin lock or fast mutex as the presentation lock to other objects such as WDFQUEUE WDFDPC, WDFTIMER, WDFWORKITEM, WDFFILEOBJECT.<br>PnP/Power events do not use this presentation lock. |
| WDFQUEUE | If you specify SynchScopeDevice or InheritFromParent, lock is provided by the device. If you specify SynchScopeObject, lock is provided by the queue. Depending on the execution level, the lock is either a spin lock or fast mutex. |
| WDFFILEOBJECT | SynchScopeObject is not allowed on this object.<br>If you specify ExecLevelPassive and SynchScopeDevice or InheritFromParent  then the parent device ExecLevel should also be Passive. |
| WDFTIMER/DPC/ WORKITEM | By setting AutomaticSerialization property, you can synchronize its events with the parent object’s events |

# I/O Target



WDFREQUEST
I/O Package
Read/Write/IOCTLs/
Create/Close/Cleanup
Parallel Queue
Sequential Queue
Manual Queue
I R P
IRP Dispatcher
PnP/Power Events
PnP/Power Package
PnP/Power Events
IoTarget
Next Driver
PnP/Power Events
WMI Package
Hardware Resource Management
(DMA, Interrupt, I/O)

# Sending Request - I/O Target

- What is an IoTarget?
  - A “target” device object to which you want to send requests
  - This “target” device object  can be the next attached device (default target) or can be a device object outside your device stack (remote target)
- Where would I use it?
  - Instead of  IoCallDriver() – either for forwarding request that you received from driver above or when you are rolling your own request and sending to another driver
    - IoTarget sends I/O in coordination with PnP state of the target owner and the target state itself
    - IoTarget provides synchronization of sent I/O with target state changes

# Default I/O Target

- WdfDeviceGetIoTarget returns WDFIOTARGET for the next lower device object

```
ForwardRequest( WDFDEVICE Device, WDFREQUEST Request)
{
   BOOLEAN ret;
    WDFIOTARGET ioTarget =  WdfDeviceGetIoTarget(Device);

    WdfRequestCopyCurrentStackLocationToNext ( Request );
    WdfRequestSetCompletionRoutine (Request, CompletionRoutine, NULL);

    ret = WdfRequestSend (Request, ioTarget, NULL);
    if (!ret) {
        status = WdfRequestGetStatus (Request);
        DebugPrint( ("WdfRequestSend failed: 0x%x\n", status));
        WdfRequestComplete(Request, status);
    }

}
```

# Remote I/O Target

- Remote I/O target represents a device object: either part of your driver or created by some other driver
- Replacement for IoGetDeviceObjectPointer, ZwCreateFile & IoRegisterPlugPlayNotification( EventCategoryTargetDeviceChange )

```
status = WdfIoTargetCreate(Device,
                           WDF_NO_OBJECT_ATTRIBUTES,
                           &IoTarget);


WDF_IO_TARGET_OPEN_PARAMS_INIT_EXISTING_DEVICE(
                                     &openParams,
                                      WdfTrue,
                                      DeviceObject);

status = WdfIoTargetOpen(IoTarget, &openParams);
```

**INIT_OPEN_BY_NAME**

**INIT_CREATE_BY_NAME**

# Send Your Own Request - Synchronous

- IoBuildSynchronousFsdRequest maps to:
  - WdfIoTargetSendReadSynchronously
  - WdfIoTargetSendWriteSynchronously

- IoBuildDeviceIoControlRequest maps to:
  - WdfIoTargetSendIoctlSynchronously
  - WdfIoTargetSendInternalIoctlSynchronously
  - WdfIoTargetSendInternalIoctlOthersSynchronously

# Send Your Own Request - Synchronous

- Buffers used in synchronous requests can be a PVOID, MDL or WDFMEMORY handle

INIT_MDL

INIT_HANDLE

```
WDF_MEMORY_DESCRIPTOR_INIT_BUFFER(&inputBufDesc,  &inBuf, inLen);

WDF_MEMORY_DESCRIPTOR_INIT_BUFFER(&outputBufDesc, outBuf,outLen);

status = WdfIoTargetSendIoctlSynchronously(ioTarget,
                                    NULL, // let framework allocate IRP
                                     IOCTL_ACPI_ASYNC_EVAL_METHOD,
                                    &inputBufDesc,
                                    &outputBufDesc,
                                    NULL, // Option
                                    NULL); // bytesReturned
```

- Requests may be sent with a combination of the following options
  - Timeout
  - Force Send (override I/O Target’s Dispatching state)

# Roll Your Own Request- Asynchronous

- IoBuildAsynchronousFsdRequest maps to

  WdfIoTargetFormatRequestForWrite

  WdfIoTargetFormatRequestForRead

  WdfIoTargetFormatRequestForIoctl

  WdfIoTargetFormatRequestForInternalIoctl

  followed by - WdfRequestSend
- I/O targets exclusively use reference counted memory handles for asynchronous IO
  - The driver cannot use raw pointers!

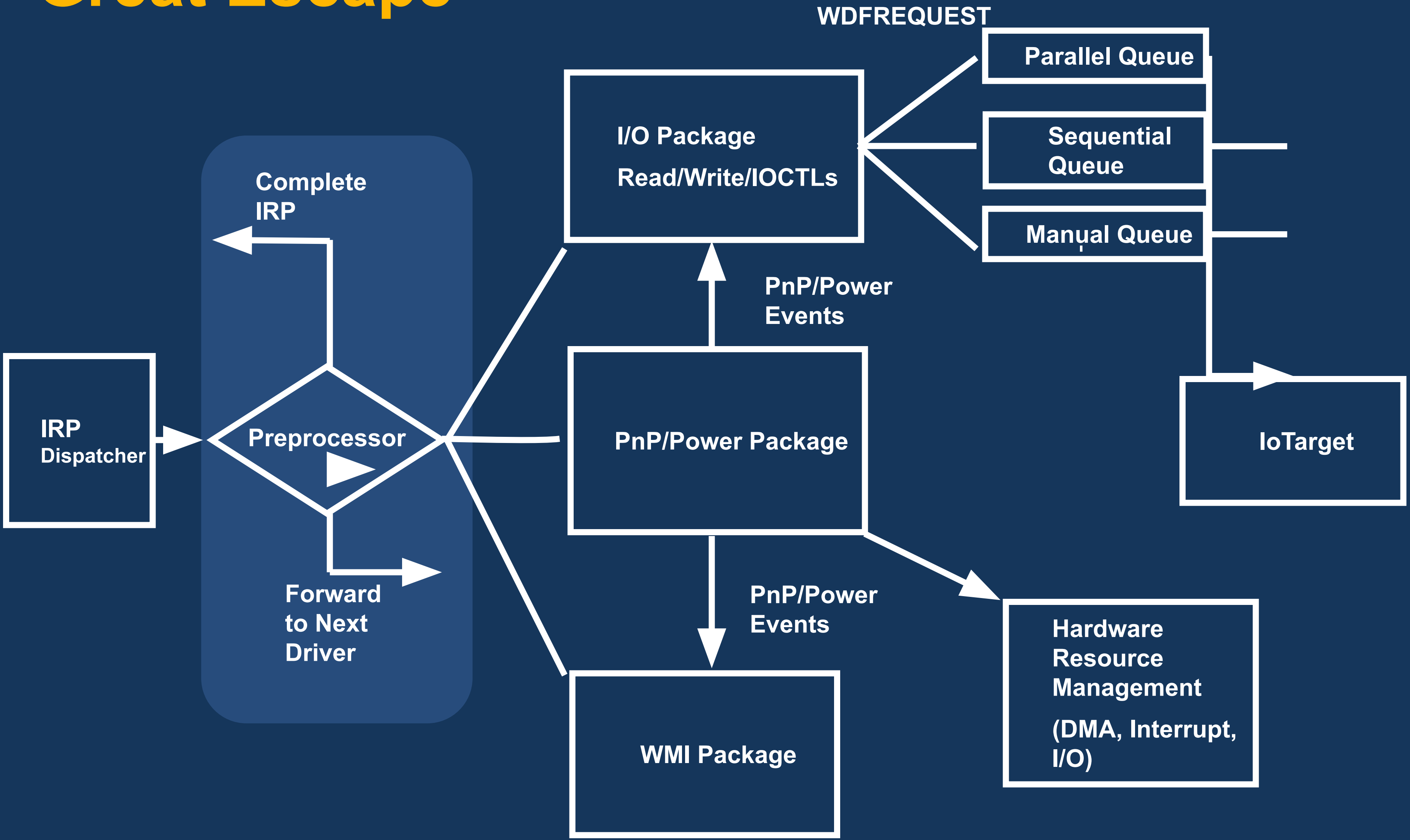
# Send Your Own Request - Asynchronous

```
status = WdfRequestCreate(WDF_NO_OBJECT_ATTRIBUTES,
           IoTarget,
           &Request);
status = WdfMemoryCreate(WDF_NO_OBJECT_ATTRIBUTES,
           NonPagedPool,
           POOL_TAG,
           sizeof(struct ABC),
           &Memory,
           (PVOID*) &buffer);
status = WdfIoTargetFormatRequestForRead(IoTarget,
           Request,
           Memory, //InputBuffer
           NULL, //  BufferOffset
           NULL); // DeviceOffset
WdfRequestSetCompletionRoutine(Request,
     ReadRequestCompletion,
     WDF_NO_CONTEXT);

if( WdfRequestSend(Request, IoTarget, NULL) == FALSE) {
     status = WdfRequestGetStatus(Request);
}
```

# Escape to WDM

- Converting to WDF is an iterative process
- Do the conversion stage by stage
  - PNP/POWER – escape to WDM for other things
  - Request handling
  - I/O Target
- WDF allows you to get all the underlying WDM objects easily
  - WdfRequestWdmGetIrp
  - WdfDeviceWdmGetAttachedDevice
  - WdfDeviceWdmGetPhysicalDevice
  - WdfDeviceWdmGetDeviceObject

# Great Escape



# Great Escape – Sample Code

```
EvtDeviceAdd()
{
     status = WdfDeviceInitAssignWdmIrpPreprocessCallback(
                  DeviceInit, PowerDispatchHandler,  IRP_MJ_POWER, NULL, 0);
}
NTSTATUS PowerDispatchHandler(WDFDEVICE Device, PIRP Irp)
{
    irpStack = IoGetCurrentIrpStackLocation(Irp);
    irpString = (irpStack->Parameters.Power.Type == SystemPowerState) ?
                                        "S-IRP" : "D-IRP";
    state = irpStack->Parameters.Power.State;
    extensionHeader = GetDeviceContext(Device);

    DebugPrint((0, "%s: %s %s %s:0x%x \n",
              (extensionHeader->IsFdo? "FDO":"PDO"), irpString,
              PowerMinorFunctionString(irpStack->MinorFunction),
                   powerStateString, Irp));
    return WdfDeviceWdmDispatchPreprocessedIrp(Device, Irp);
}
```

# Call to Action

- Work together with us to make WDF successful
- Consider WDF for any Windows driver development project
- Join WDF beta program
  - Use the special guest account (Guest ID: **Guest4WDF)** on http://beta.microsoft.com
- Provide feedback
  - Email
    - windf @ microsoft.com - Kernel Mode Driver Framework
    - umdfdbk @ microsoft.com - User Mode Driver Framework
    - drvpft @ microsoft.com - PREfast for Drivers
    - sdvfdbk @ microsoft.com - Static Driver Verifier
  - Newsgroups
    - microsoft.beta.windows.driverfoundation
    - microsoft.beta.windows.driverfoundation.announcements
- Web Resources

  http://www.microsoft.com/whdc/driver/wdf/default.mspx

  http://www.microsoft.com/whdc/DevTools/ddk/default.mspx

# Reference Slides

# Sample Scenarios – Callback Order

- Following slides show in what order all the events of device, queue, interrupt and DMA enabler object are triggered by the PnP/Power stage for the following scenarios
    - Start device
    - Disable or uninstall the device
    - Surprise-Removal
    - Resource rebalance
    - Failed query-remove or failed query-stop
    - System suspend
    - System resume
- Slides also show the PnP/Power IRP context in which these events are invoked

# Start Device

- AddDevice
  - EvtDeviceAdd
- IRP_MN_START_DEVICE
  - EvtDevicePrepareHardware
  - EvtDeviceD0Entry
  - EvtInterruptEnable
  - EvtDeviceD0EntryPostInterruptsEnabled
  - EvtDmaEnablerEnable
  - EvtDmaEnablerFill
  - EvtDmaEnablerSelfManagedIoStart
  - EvtDeviceSelfManagedIoInit

# Disable or Uninstall Device

- IRP_MN_QUERY_REMOVE_DEVICE
  - EvtDeviceQueryRemove
- IRP_MN_REMOVE_DEVICE
  - EvtDeviceSelfManagedIoSuspend
  - EvtIoStop – Suspend
  - EvtDmaEnablerSelfManagedIoStop
  - EvtDmaEnablerDisable
  - EvtDmaEnablerFlush
  - EvtInterruptDisable
  - EvtDeviceD0Exit - D3Final
  - EvtDeviceReleaseHardware
  - EvtIoStop - Purge
  - EvtDeviceSelfManagedIoFlush
  - EvtDeviceSelfManagedIoCleanup
  - EvtDeviceContextCleanup

```
_WDF_POWER_DEVICE_STATE {
   WdfPowerDeviceUnspecified = 0,
   WdfPowerDeviceD0,
   WdfPowerDeviceD1,
   WdfPowerDeviceD2,
   WdfPowerDeviceD3,
   WdfPowerDeviceD3Final,
   WdfPowerDevicePrepareForHiber,
   WdfPowerDeviceMaximum,
} WDF_POWER_DEVICE_STATE,
```

# Surprise Remove Device

- IRP_MN_SURPRISE_REMOVAL
  - EvtDeviceSurpriseRemoval
  - EvtDeviceSelfManagedIoSuspend
  - EvtIoStop – Suspend
  - EvtDmaEnablerSelfManagedIoStop
  - EvtDmaEnablerDisable
  - EvtDmaEnablerFlush
  - EvtInterruptDisable
  - EvtDeviceD0Exit - D3Final
  - EvtDeviceReleaseHardware
  - EvtIoStop - Purge
  - EvtDeviceSelfManagedIoFlush
  - EvtDeviceSelfManagedIoCleanup
- IRP_MN_REMOVE_DEVICE
  - EvtDeviceContextCleanup

# Resource Rebalance

- IRP_MN_QUERY_STOP_DEVICE
  - EvtDeviceQueryStop
- IRP_MN_STOP_DEVICE
  - EvtDeviceSelfManagedIoSuspend
  - EvtIoStop – Suspend
  - EvtDmaEnablerSelfManagedIoStop
  - EvtDmaEnablerDisable/Flush
  - EvtInterruptDisable
  - EvtDeviceD0Exit - D3Final
  - EvtDeviceReleaseHardware
- IRP_MN_START_DEVICE
  - EvtDevicePrepareHardware
  - EvtDeviceD0Entry
  - EvtInterruptEnable
  - EvtIoResume
  - EvtDmaEnablerEnable/Fill
  - EvtDmaEnablerSelfManagedIoStart
  - EvtDeviceSelfManagedIoRestart

# Failed Remove or Stop

- Failed Remove
  - IRP_MN_QUERY_REMOVE_DEVICE
    - EvtDeviceQueryRemove
  - IRP_MN_CANCEL_REMOVE_DEVICE
- Failed Stop:
  - IRP_MN_QUERY_STOP_DEVICE
    - EvtDeviceQueryStop
  - IRP_MN_CANCEL_STOP_DEVICE

# System Suspend

- IRP_MN_QUERY_POWER Sx
  - (WDF doesn't send IRP_MN_QUERY_POWER Dx)
- IRP_MN_SET_POWER Sx
- IRP_MN_SET_POWER Dx
  - EvtDeviceSelfManagedIoSuspend
  - EvtIoStop - Suspend on every in-flight request
  - WDF sends IRP_MN_WAIT_WAKE
  - EvtDeviceArmWakeFromSx
  - EvtDmaEnablerSelfManagedIoStop
  - EvtDmaEnablerDisable/Flush
  - EvtInterruptDisable
  - EvtDeviceD0Exit

# System Resume

- System sends IRP_MN_SET_POWER S0
  - WDF completes it first to allow fast resume
  - Then WDF sends IRP_MN_SET_POWER D0
- WDF cancels IRP_MN_WAIT_WAKE
- IRP_MN_SET_POWER D0
  - EvtDeviceD0Entry
  - EvtInterruptEnable
  - EvtDmaEnablerFill
  - EvtDmaEnablerEnable
  - EvtDmaEnablerSelfManagedIoStart
  - EvtIoResume
  - EvtDeviceSelfManagedIoRestart
  - EvtDeviceDisarmWakeFromSx

# Parsing HW Resources

```
NTSTATUS
PciDrvEvtDevicePrepareHardware (
    WDFDEVICE       Device,
    WDFCMRESLIST    Resources,
    WDFCMRESLIST    ResourcesTranslated    )
{
    PCM_PARTIAL_RESOURCE_DESCRIPTOR desc;

    for (i=0; i<WdfCmResourceListGetCount(ResourcesTranslated); i++)    {

        desc = WdfCmResourceListGetDescriptor(ResourcesTranslated, i);

        switch (desc->Type) {

        case CmResourceTypePort:  break;
        case CmResourceTypeMemory:  break;
        }
    }
}
```

Microsoft®

Your potential. Our passion.™

This presentation is for informational purposes only. Microsoft makes no warranties, express or implied, in this summary.