Plug & Play

主講人：虞台文

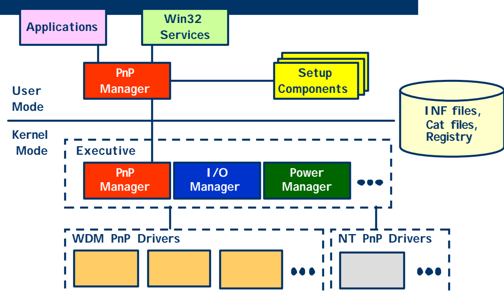
Plug and Play (PnP)

- A combination of **hardware** and **software** support that enables a computer system to **recognize** and **adapt** to **hardware configuration** changes with little or **no intervention by a user**.
- **Hardware**
  - Hardware industry define **standards** for **easy identification** of add-in boards and basic system components.
- **Software**
  - **System Software** : supports for PnP
  - **Drivers** : interact with system software

Content

- Overview
- Device States
- Device Queue for PnP Drivers
- DisptachPnP for Functions Driver
- Starting a Device
- Stopping a Device
- Removing a Device

PnP Software Components



Plug & Play

Overview

System Software Support for PnP

- Automatic and dynamic **recognition** of installed **hardware**.
- Hardware **resource allocation** (and reallocation).
- **Loading** of appropriate **drivers**.
- A **programming interface for drivers** to interact with the PnP system.
- Mechanisms for **drivers** and **applications** to **learn of changes** in the hardware environment and take appropriate actions

The PnP Drivers

- Must contain a **DispatchPnP** routine.
  - Handle **IRP_MJ_PNP** requests and associated **minor** function codes.
- Must *not* **search** for **hardware**.
  - The **PnP Manager** determines the presence of hardware devices.
  - When detected, notifies the driver by calling its **AddDevice** routine.
  - Hardware can be detected when the system is **booted**, or any time that a user **adds** a device to, or **removes** one from, a running system.
- Must *not* **allocate** hardware **resources**
  - A **PnP driver** must provide the PnP Manager with lists of **resources** that a device can **potentially use**.
  - The **PnP Manager** is responsible for **assigning resources**. It notices the driver by sending an **IRP_MN_START_DEVICE** request.
  - The driver must be **capable** on **various configurations** of hardware resources.

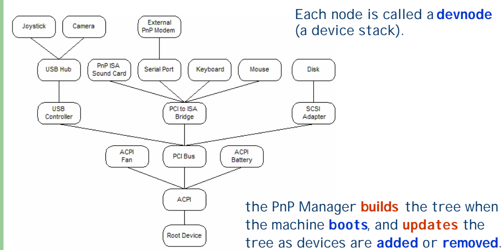
Hardware Resources

- Hardware resources are the **assignable**, **addressable** bus paths that allow **peripheral devices** and **system processors** to communicate.
- Typically include
  - I/O **port** addresses (or memory registers)
  - **interrupt** vectors, and
  - blocks of **bus-relative memory** addresses (e.g., DMA).

The PnP Manager maintains a **device tree** that **keeps track of the devices in the system**

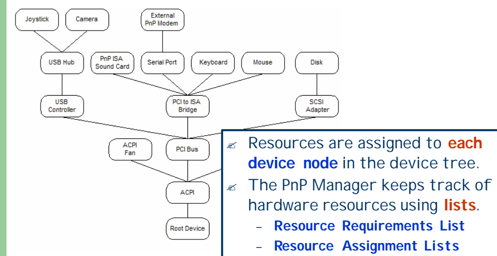
The Device Tree

Each node is called a **devnode** (a device stack).



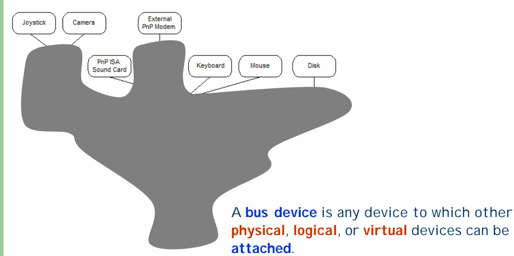
the PnP Manager **builds** the tree when the machine **boots**, and **updates** the tree as devices are **added** or **removed**.

Resource-Assignment by PnP Manager



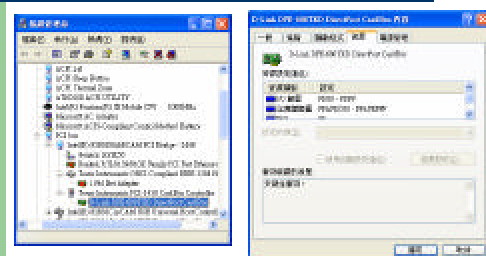
- Resources are assigned to **each device node** in the device tree.
- The PnP Manager keeps track of hardware resources using **lists**.
  - **Resource Requirements List**
  - **Resource Assignment Lists**

The Bus Devices



A **bus device** is any device to which other **physical**, **logical**, or **virtual** devices can be **attached**.

Resource-Assignment by PnP Manager



Resource Management Lists

- Resource Requirements List
  - Specifies all of the **ranges** of **hardware resources** in which the device ***can*** **operate**.
  - The PnP Manager **chooses** resources from this list when **assigning** them to the device.
  - IO_RESOURCE_REQUIREMENTS_LIST
- Resource Assignment List
  - Specifies the **resources assigned** to each device instance.
  - CM_RESOURCE_LIST

Plug and Play Minor IRPs

- IRP_MN_QUERY_RESOURCES
- IRP_MN_QUERY_RESOURCE_REQUIREMENTS
- IRP_MN_QUERY_DEVICE_TEXT
- IRP_MN_READ_CONFIG
- IRP_MN_WRITE_CONFIG
- IRP_MN_EJECT
- IRP_MN_SET_LOCK
- IRP_MN_QUERY_ID
- IRP_MN_QUERY_BUS_INFORMATION

Bus drivers handle these minor functions

IRP_MJ_PNP

PnP Requests

Plug and Play requests play **two roles**:

- **Configure**/**deconfigure** hardware
  - When and how to do so.
- Guide the driver through a series of **state transitions**
  - Start
  - Pause
  - Remove/Surprising Remove

Plug & Play

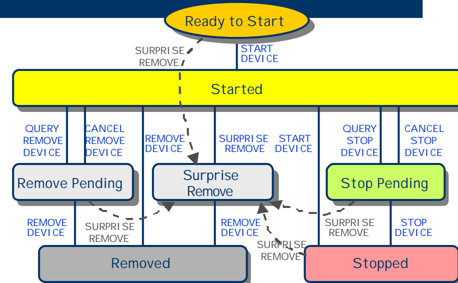
Device States

Plug and Play Minor IRPs

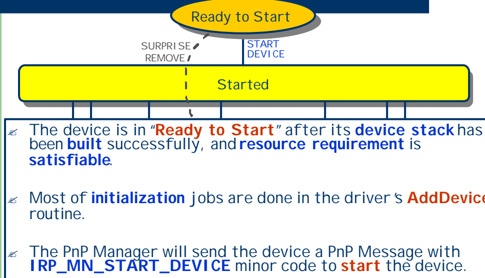
- IRP_MN_START_DEVICE
- IRP_MN_QUERY_REMOVE_DEVICE
- IRP_MN_REMOVE_DEVICE
- IRP_MN_CANCEL_REMOVE_DEVICE
- IRP_MN_STOP_DEVICE
- IRP_MN_QUERY_STOP_DEVICE
- IRP_MN_CANCEL_STOP_DEVICE
- IRP_MN_QUERY_DEVICE_RELATIONS
- IRP_MN_QUERY_INTERFACE
- IRP_MN_QUERY_CAPABILITIES
- IRP_MN_FILTER_RESOURCE_REQUIREMENTS
- IRP_MN_QUERY_PNP_DEVICE_STATE
- IRP_MN_DEVICE_USAGE_NOTIFICATION
- IRP_MN_SURPRISE_REMOVAL

Function/Filter drivers handle these minor functions

State Diagram for a Device



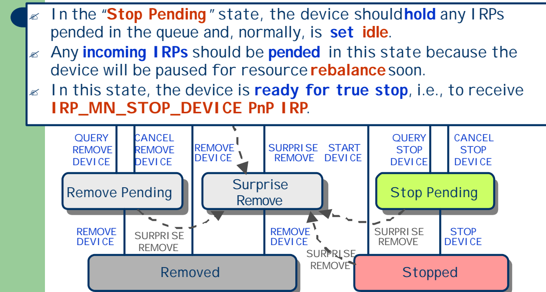
The “Ready to Start” State



- The device is in “**Ready to Start**” after its **device stack** has been **built** successfully, and **resource requirement** is **satisfiable**.
- Most of **initialization** jobs are done in the driver’s **AddDevice** routine.
- The PnP Manager will send the device a PnP Message with **IRP_MN_START_DEVICE** minor code to **start** the device.

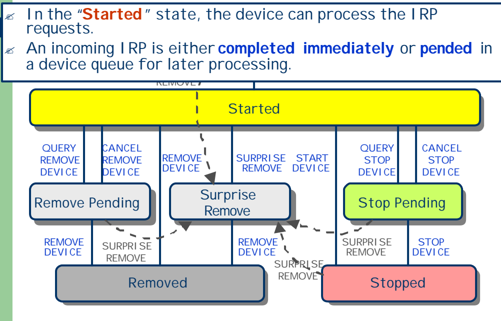
The “Stop Pending” State

- In the “**Stop Pending**” state, the device should **hold** any IRPs pended in the queue and, normally, is **set idle**.
- Any **incoming IRPs** should be **pended** in this state because the device will be paused for resource **rebalance** soon.
- In this state, the device is **ready for true stop**, i.e., to receive **IRP_MN_STOP_DEVICE PnP IRP**.



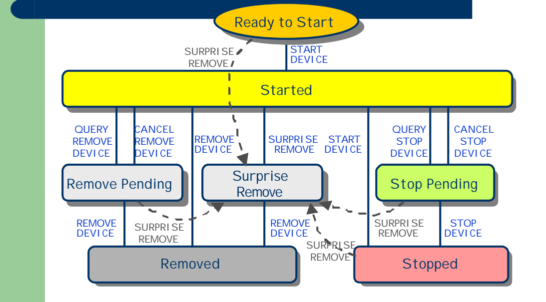
The “Started” State

- In the “**Started**” state, the device can process the IRP requests.
- An incoming IRP is either **completed immediately** or **pended** in a device queue for later processing.



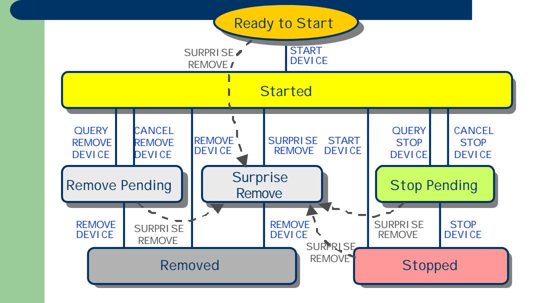
What actions should be taken to enter the “Stop Pending” state?

The “Stop Pending” State



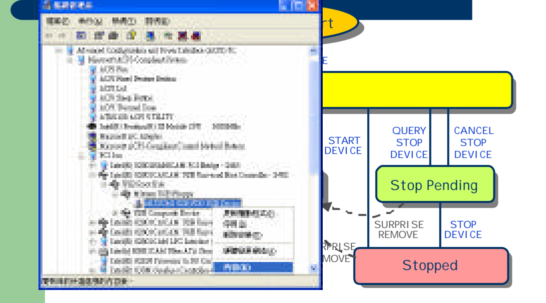
What actions should be taken to enter the “Started” state?

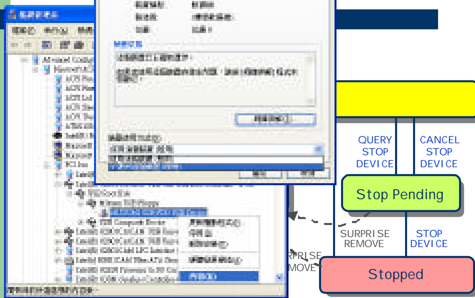
The “Started” State

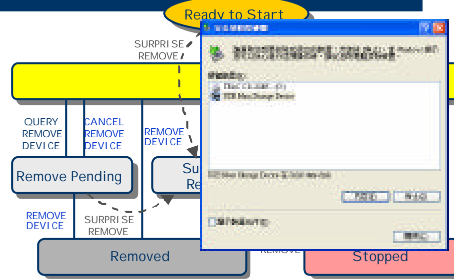


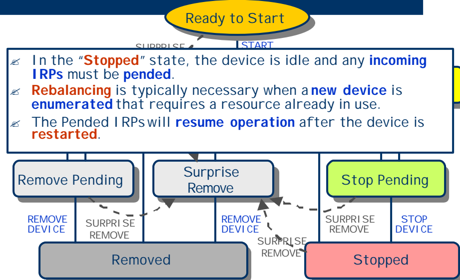
What actions should be taken to enter the “Stop Pending” state?

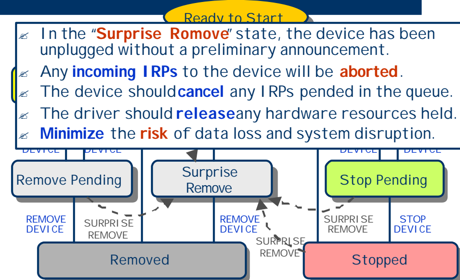
The “Stop Pending” State

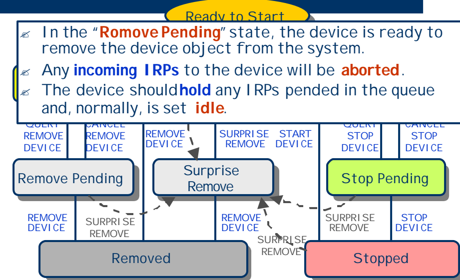


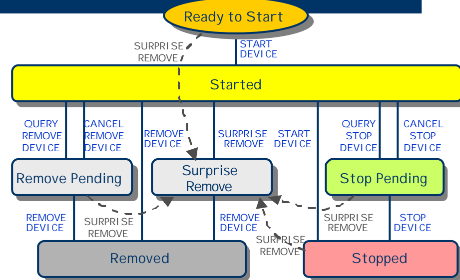

What actions should be taken to enter the "Stop Pending" state?
The ...g" State
QUERY STOP DEVICE
CANCEL STOP DEVICE
Stop Pending
SURPRISE REMOVE
STOP DEVICE
Stopped


The "Remove Pending" State
Ready to Start
SURPRISE REMOVE
QUERY REMOVE DEVICE
CANCEL REMOVE DEVICE
REMOVE DEVICE
Remove Pending
REMOVE DEVICE
SURPRISE REMOVE
Removed
Stopped


The "Stopped" State
Ready to Start
In the "Stopped" state, the device is idle and any incoming IRPs must be pended.
Rebalancing is typically necessary when a new device is enumerated that requires a resource already in use.
The Pended IRPs will resume operation after the device is restarted.
Remove Pending
Surprise Remove
Stop Pending
REMOVE DEVICE
SURPRISE REMOVE
REMOVE DEVICE
SURPRISE REMOVE
STOP DEVICE
SURPRISE REMOVE
Removed
Stopped


The "Surprise Remove" State
Ready to Start
In the "Surprise Romove" state, the device has been unplugged without a preliminary announcement.
Any incoming IRPs to the device will be aborted.
The device should cancel any IRPs pended in the queue.
The driver should release any hardware resources held.
Minimize the risk of data loss and system disruption.
Remove Pending
Surprise Remove
Stop Pending
REMOVE DEVICE
SURPRISE REMOVE
REMOVE DEVICE
SURPRISE REMOVE
STOP DEVICE
SURPRISE REMOVE
Removed
Stopped

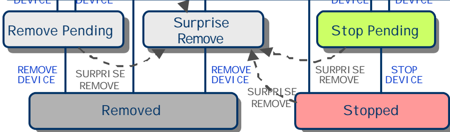

The "Remove Pending" State
Ready to Start
In the "Remove Pending" state, the device is ready to remove the device object from the system.
Any incoming IRPs to the device will be aborted.
The device should hold any IRPs pended in the queue and, normally, is set idle.
REMOVE DEVICE
REMOVE DEVICE
REMOVE DEVICE
SURPRISE REMOVE
START DEVICE
STOP DEVICE
STOP DEVICE
Remove Pending
Surprise Remove
Stop Pending
REMOVE DEVICE
SURPRISE REMOVE
REMOVE DEVICE
SURPRISE REMOVE
STOP DEVICE
SURPRISE REMOVE
Removed
Stopped


Almost any state can change to the "Surprise Remove" state due to unplugging.
The "Surprise Remove" State
Ready to Start
SURPRISE REMOVE
START DEVICE
Started
QUERY REMOVE DEVICE
CANCEL REMOVE DEVICE
REMOVE DEVICE
SURPRISE REMOVE
START DEVICE
QUERY STOP DEVICE
CANCEL STOP DEVICE
Remove Pending
Surprise Remove
Stop Pending
REMOVE DEVICE
SURPRISE REMOVE
REMOVE DEVICE
SURPRISE REMOVE
STOP DEVICE
SURPRISE REMOVE
Removed
Stopped

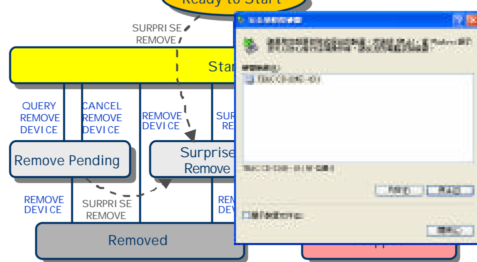
Almost any state can change to the "**Surprise Remove**" state due to **unplugging**.

The "Removed" State

- In the "**Romoved**" state, the device stack of the device is destroyed.
- The device can be **safely unplugged** if it is still connected.



The "Removed" State

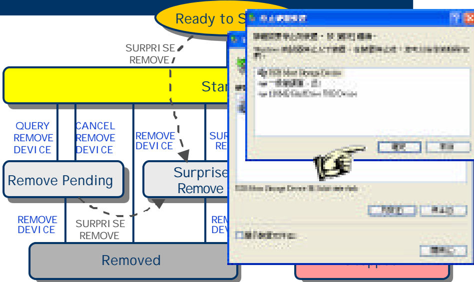


The "Removed" State

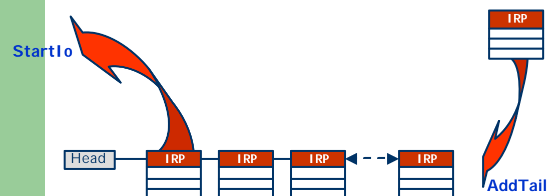
Plug & Play

Device Queue for PnP Drivers

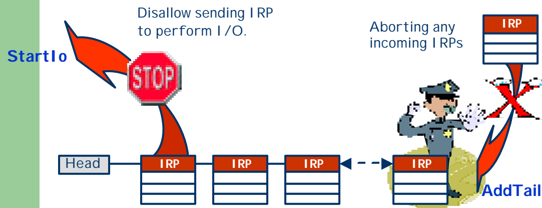
The "Removed" State



Device Queue to Support PnP



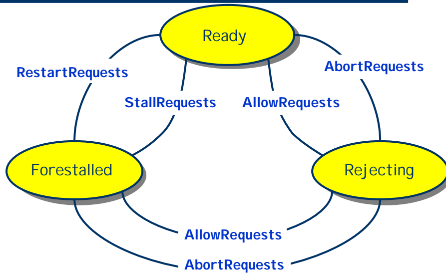
Special States of IRP Queues



DEVQUEUE Operations

Support Function	Description
AbortRequests	Aborts current and future requests
AllowRequests	Undoes effect of previous AbortRequests
AreRequestsBeingAborted	Are we currently aborting new requests?
CancelRequest	Generic cancel routine
CheckBusyAndStall	Checks for idle device and stalls requests.
CleanupRequests	Service IRP_MJ_CLEANUP
GetCurrentIrp	Get current IRP processed by associated StartIo routine
InitializeQueue	Initializes DEVQUEUE object
RestartRequests	Restarts a stalled queue
StallRequests	Stalls the queue
StartNextPacket	Dequeues and starts the next request
StartPacket	Starts or queues a new request
WaitForCurrentIrp	Waits for current IRP to finish

States of Device Queues



Initialize DEVQUEUE

```
VOID InitializeQueue(PDEVQUEUE pdq, PDRIVER_STARTIO StartIo)
{
  InitializeListHead(&pdq->head);
  KeInitializeSpinLock(&pdq->lock);
  pdq->StartIo = StartIo;
  pdq->stallcount = 1;
  pdq->CurrentIrp = NULL;
  KeInitializeEvent(&pdq->evStop, NotificationEvent, FALSE);
  pdq->abortstatus = (NTSTATUS) 0;
  pdq->notify = NULL;
  pdq->notifycontext = 0;
}
```

DEVQUEUE

```
typedef struct _DEVQUEUE {
      LIST_ENTRY head;
      KSPIN_LOCK lock;
      PDRIVER_STARTIO StartIo;
      LONG stallcount;        // >=1: stalled
      PIRP CurrentIrp;
      KEVENT evStop;          // wait IRP completion
      PQNOTIFYFUNC notify;    // no wait version
      PVOID notifycontext;
      NTSTATUS abortstatus; // non-zero: aborted
} DEVQUEUE, *PDEVQUEUE;
```

Plug & Play

DisptachPnP for Function Drivers

DispatchPnP

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT do, PIRP Irp)
{
}
```

- All PnP IRPs have the **major** function code **IRP_MJ_PNP** and a **minor** function code.
- For each IRP and each kind of driver, a driver is either **required** to handle the IRP, can **optionally** handle the IRP, or **must not** handle the IRP.

must pass the IRP to the **next** driver

Default PnP Handler

The PnP Manager initializes **Irp->IoStatus.Status** to **STATUS_NOT_SUPPORTED**.

```
NTSTATUS DefaultPnpHandler(PDEVICE_EXTENSION do, PIRP Irp)
{
  IoSkipCurrentIrpStackLocation(Irp);

  return IoCallDriver(pdx->LowerDeviceObject, Irp);
}
```

Layout I

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT do, PIRP Irp)
{
  static NTSTATUS (*fcntab[])(PDEVICE_OBJECT, PIRP) = {
    HandleStartDevice, // IRP_MN_START_DEVICE
    HandleQueryRemove, // IRP_MN_QUERY_REMOVE_DEVICE
    <etc.>,
  };
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG fcn = stack->MinorFunction;
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) do->DeviceExtension;
  if (fcn >= arraysize(fcntab))
    return DefaultPnpHandler(pdx, Irp);
  return (*fcntab[fcn])(pdx, Irp);
}
```

The PnP IRP Handled

```
NTSTATUS HandleXxx(PDEVICE_EXTENSION pdx, PIRP Irp)
{
}
```

- Depends on the **minor code**, the PnP driver invoke the next driver in two ways:
  - **Passing** PnP IRPs **Down** the Device Stack.
  - **Postponing** PnP IRP Processing Until Lower Drivers Finish.

Layout II

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT do, PIRP Irp)
{
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  PDEVICE_EXTENSION pdx =
    (PDEVICE_EXTENSION) do->DeviceExtension;

  switch(stack->MinorFunction){
  case IRP_MN_START_DEVICE:
       return HandleStartDevice(pdx, Irp);
  case IRP_MN_QUERY_REMOVE_DEVICE:
       return HandleQueryRemove(pdx, Irp);
  . . . . . . . . . . . . . . . . . . . . .
  default:
       return DefaultPnpHandler(pdx, Irp);
  }
}
```

Passing PnP IRPs Down the Device Stack

```
NTSTATUS HandleXxx(PDEVICE_EXTENSION pdx, PIRP Irp)
{
}
```

- Perform the **appropriate actions**.
- Set **Irp->IoStatus.Status** to an appropriate status, such as **STATUS_SUCCESS**. Set **Irp->IoStatus.Information**, if appropriate for the IRP.
- Set up the next stack location with **IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext**. Call the latter routine if you set an **IoCompletion** routine.
- Invoke the next lower driver by calling **IoCallDriver**.
- Do not complete the IRP. (Do **not** call **IoCompleteRequest**.) The parent **bus driver** will complete the IRP.

Postponing PnP IRP Processing

```
NTSTATUS HandleXxx(PDEVICE_EXTENSION pdx, PIRP Irp)
{
```

- Set Irp->IoStatus.Status to STATUS_SUCCESS.
- Create a event.
- Call IoCopyCurrentIrpStackLocationToNext to set the parameter for the next lower driver.
- Set Completion routine
- Invoke the next lower driver and wait until its job done, i.e. the event is signaled.
- Check Irp->IoStatus.Status
- If succeeded do the necessary IRP operation.
- Complete the IRP
- Return the appropriate status.

```
}
```

Help Functions

```
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status,
                         ULONG_PTR info)
{
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status)
{
     Irp->IoStatus.Status = status;
     IoCompleteRequest(Irp, IO_NO_INCREMENT);
     return status;
}
```

Help Functions

```
NTSTATUS ForwardAndWait(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  KEVENT event;
  KeInitialize(&event, NotificationRoutine, FALSE);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)
    ForwardAndWaitCompletionRoutine, &event , TRUE, TRUE, TRUE);
  NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
  if (status == STATUS_PENDING){
    KeWaitForSingleObject(&event, Executive, KernelMode,
      FALSE, NULL);
    status = Irp->IoStatus.Status;
  }
  return status;
}
```

Device-State Enumerator

```
enum DEVSTATE {
    STOPPED,          // device stopped
    WORKING,          // started and working
    PENDINGSTOP,      // stop pending
    PENDINGREMOVE,    // remove pending
    SURPRISEREMOVED,// removed by surprise
    REMOVED,          // removed
};
```

Help Functions

```
NTSTATUS ForwardAndWait(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  KEVENT event;
  KeInitialize(&event, NotificationRoutine, FALSE);
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, (PIO_COMPLETION_ROUTINE)
    ForwardAndWaitCompletionRoutine, &event , TRUE, TRUE, TRUE);
  NTSTATUS status = IoCallDriver(pdx->LowerDeviceObject, Irp);
```

```
NTSTATUS ForwardAndWaitCompletionRoutine(PDEVICE_OBJECT fdo,
  PIRP Irp, PKEVENT pev)
{
  if (Irp->PendingReturned)
    KeSetEvent(pev, IO_NO_INCREMENT, FALSE);
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

The Device Extension

```
typedef struct _DEVICE_EXTENSION {
  . . . . . . . . . . . . . .
  DEVQUEUE Queue;
  DEVSTATE State;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Plug & Play

Starting a Device

Starting a Device

```
NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
```

Let the **lower level** drivers do **first**.
If a lower driver **failed** the IRP, do any necessary **cleanup** the **complete** it.

Start the device

Start IRPs in the IRP-holding **queue**.
Enable interfaces for the device.
Change State to **WORKING**.

Complete this IRP.

```
}
```

Starting a Device

- The PnP Manager sends an **IRP_MN_START_DEVICE** request to drivers either to start a newly **enumerated** device or to restart an existing device that was **stopped** for **resource rebalancing**.
- **Function** and **filter drivers** must set an **IoCompletion** routine, pass the IRP down the device stack, and **postpone** their start operations until all lower drivers finish the IRP.
- The parent **bus** driver, the bottom driver in the device stack, must be the **first** driver to perform its start operations on a device.

Starting a Device

```
NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status);
```

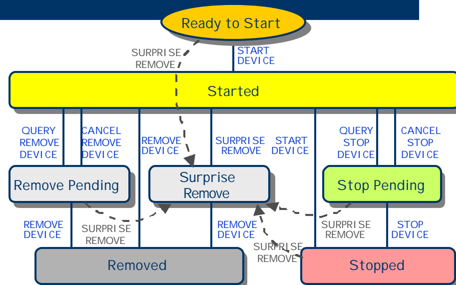
Start the device

Start IRPs in the IRP-holding **queue**.
Enable interfaces for the device.
Change State to **WORKING**.

Complete this IRP.

```
}
```

Starting a Device



Starting a Device

```
NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status);

  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  status = StartDevice(pdx, <additional args>);
```

Start IRPs in the IRP-holding **queue**.
Enable interfaces for the device.
Change State to **WORKING**.

Complete this IRP.

```
}
```

Starting a Device

```
NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status);

  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  status = StartDevice(pdx, <additional args>);
  if (NT_SUCCESS(status)){
    pdx->State = WORKING;
    RestartRequests(&pdx->Queue, pdx->DeviceObject);
    EnableAllInterfaces(pdx, True);
  }
}
```

Complete this IRP.

Parameters of IRP_MN_START_DEVICE

```
typedef struct _IO_STACK_LOCATION {
    . . . . . . . . . . . . .
    union {
        . . . . . . . . . . . . .
        //
        // Parameters for IRP_MN_START_DEVICE
        //
        struct {
            PCM_RESOURCE_LIST AllocatedResources;
            PCM_RESOURCE_LIST AllocatedResourcesTranslated;
        } StartDevice;
        . . . . . . . . . . . . .
    } Parameters;
    . . . . . . . . . . . . .
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

Starting a Device

```
NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status);

  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  status = StartDevice(pdx, <additional args>);
  if (NT_SUCCESS(status)){
    pdx->State = WORKING;
    RestartRequests(&pdx->Queue, pdx->DeviceObject);
    EnableAllInterfaces(pdx, True);
  }

  return CompleteRequest(Irp, status);
}
```

PCM_RESOURCE_LIST

```
typedef struct _CM_RESOURCE_LIST {
  ULONG  Count;
  CM_FULL_RESOURCE_DESCRIPTOR  List[1];
} CM_RESOURCE_LIST, *PCM_RESOURCE_LIST;
```

```
typedef struct _CM_FULL_RESOURCE_DESCRIPTOR {
  INTERFACE_TYPE  InterfaceType;
  ULONG  BusNumber;
  CM_PARTIAL_RESOURCE_LIST  PartialResourceList;
} CM_FULL_RESOURCE_DESCRIPTOR, *PCM_FULL_RESOURCE_DESCRIPTOR;
```

```
typedef struct _CM_PARTIAL_RESOURCE_LIST {
  USHORT  Version;
  USHORT  Revision;
  ULONG  Count;
  CM_PARTIAL_RESOURCE_DESCRIPTOR PartialDescriptors[1];
} CM_PARTIAL_RESOURCE_LIST, *PCM_PARTIAL_RESOURCE_LIST;
```

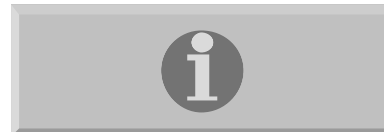
- The exact steps to start a device **vary from device to device**.
- Such as **mapping I/O space**, **initializing hardware registers**, setting the device in the **D0 power state**, and **connecting the interrupt** with **IoConnectInterrupt**.
- If restarting a device after an **IRP_MN_STOP_DEVICE** request, the driver might have device **state** to **restore**.
- The device must be **powered on** before any drivers can access it.
- If the device should be enabled for wake-up, its **power policy owner** (usually the function driver) should send a **wait/wake IRP** after it powers up the device and **before** it completes the **IRP_MN_START_DEVICE** request.

```
NTST
{
  Ir
  NT
  if (!NT_SUCCESS(status))
    return CompleteRequest(Irp, status);

  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  status = StartDevice(pdx, <additional args>);
  if (NT_SUCCESS(status)){
    pdx->State = WORKING;
    RestartRequests(&pdx->Queue, pdx->DeviceObject);
    EnableAllInterfaces(pdx, True);
  }

  return CompleteRequest(Irp, status);
}
```

CM_PARTIAL_RESOURCE_DESCRIPTOR

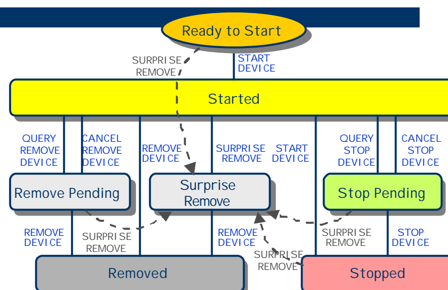


You only need to write **StartDevice** routine.

StartDevice

```
NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  . . . . . . . . . . . . . . . . . . . . . . . .
  // call StartDevice()
  PCM_PARTIAL_RESOURCE_LIST raw, translated;
  raw = &stack->Parameters.StartDevice
     .AllocatedResources->List[0].PartialResourceList;
  translated = &stack->Parameters.StartDevice
     .AllocatedResourcesTranslated->List[0].PartialResourceList;
  status = StartDevice(pdx, raw, translated);
  . . . . . . . . . . . . . . . . . . . . . . . .
}
```

Stopping a Device to Rebalance Resources

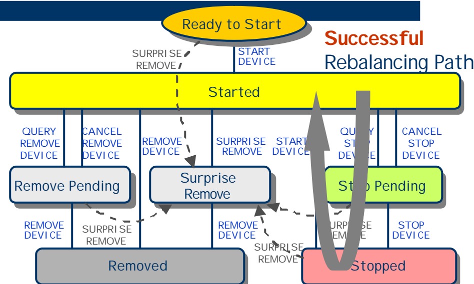


Plug & Play

Stopping a Device

Stopping a Device to Rebalance Resources

Successful Rebalancing Path



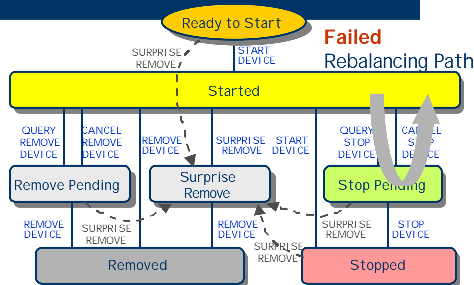
Rebalancing is typically necessary when a **new** device is **enumerated** that requires a **resource** already **in use**.

When Stopping a Device?

- The **PnP Manager** directs drivers to **stop** a device in the following situations:
  - To **rebalance** the hardware resources being used.
  - To **disable** the device in response to a Device Manager request (Windows 98/Me only). **Windows 2000** and **later** systems send **remove** IRPs in this situation.
  - After a **failed IRP_MN_START_DEVICE** request (Windows 98/Me only)

Stopping a Device to Rebalance Resources

Failed Rebalancing Path



Handle IRP_MN_QUERY_STOP_DEVICE

- The PnP Manager issues an **IRP_MN_QUERY_STOP_DEVICE** to ask **whether** the drivers for a device **can stop** the device and **release** its hardware resources.
- If **all** the **drivers** in the **device stack** return **STATUS_SUCCESS**, the drivers have put the device into a state (**stop-pending**) from which the device can be quickly stopped.
- The PnP Manager **queries as many device stacks as necessary** to rebalance the required resources.

Handle IRP_MN_QUERY_STOP_DEVICE

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != WORKING)
    return DefaultPnpHandler(pdx, Irp);
  if (!OkayToStop(pdx))
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
```

Do the necessary operations to change the device into **PENDINGSTOP** state.

Invoke the **lower driver** for **further determination**.

```
}
```

Handle IRP_MN_QUERY_STOP_DEVICE

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
```

Whether the driver **WORKING** (**STOPPED**)?
If *not* **WORKING**, let the **lower driver** conclude it.

Can the the driver be **STOPPED**?
If *no*, **complete** the IRP **unsuccessfully**.

Do the necessary operations to change the device into **PENDINGSTOP** state.

Invoke the **lower driver** for **further determination**.

```
}
```

Handle IRP_MN_QUERY_STOP_DEVICE

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != WORKING)
    return DefaultPnpHandler(pdx, Irp);
  if (!OkayToStop(pdx))
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
  StallRequests(&pdx->Queue);
  WaitForCurrentIrp(&pdx->Queue);
  pdx->State = PENDINGSTOP;
```

Invoke the **lower driver** for **further determination**.

```
}
```

Handle IRP_MN_QUERY_STOP_DEVICE

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != WORKING)
    return DefaultPnpHandler(pdx, Irp);
```

Can the the driver be **STOPPED**?
If *no*, **complete** the IRP **unsuccessfully**.

Do the necessary operations to change the device into **PENDINGSTOP** state.

Invoke the **lower driver** for **further determination**.

```
}
```

Handle IRP_MN_QUERY_STOP_DEVICE

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != WORKING)
    return DefaultPnpHandler(pdx, Irp);
  if (!OkayToStop(pdx))
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
  StallRequests(&pdx->Queue);
  WaitForCurrentIrp(&pdx->Queue);
  pdx->State = PENDINGSTOP;
  return DefaultPnpHandler(pdx, Irp);
}
```

Can a Device be Stopped?

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != WORKING)
    return DefaultPnpHandler(pdx, Irp);
  if (!OkayToStop(pdx))
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
  StallRequests(&pdx->Queue);
  WaitForCurrentIrp(&pdx->Queue);
  pdx->State = PENDINGSTOP;
  return DefaultPnpHandler(pdx, Irp);
}
```

The Response of Query Stop

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  return DefaultPnpHandler(pdx, Irp);
}
```

If this IRP is **completed successfully**, i.e., no driver in the stack denies the request, the PnP Manager will send **IRP_MN_STOP_DEVICE** PnP request in sequel.

Otherwise, **IRP_MN_CANCEL_STOP_DEVICE** PnP request will be sent.

Can a Device be Stopped?

- A driver ***must* fail** a query-stop IRP if any of the following are true:
  - A driver has been notified (through **IRP_MN_DEVICE_USAGE_NOTIFICATION**) that the device is in the path of a **paging**, **hibernation**, or **crash dump** file.
  - The device's hardware resources **cannot** be **released**.

Does the Driver Always Need to Complete the Current IRP at the Query-Stop Stage?

```
NTSTATUS HandleQueryStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != WORKING)
    return DefaultPnpHandler(pdx, Irp);
  if (!OkayToStop(pdx))
    return CompleteRequest(Irp, STATUS_UNSUCCESSFUL, 0);
  StallRequests(&pdx->Queue);
  WaitForCurrentIrp(&pdx->Queue);
  pdx->State = PENDINGSTOP;
  return DefaultPnpHandler(pdx, Irp);
}
```

Can a Device be Stopped?

- A driver ***might* fail** a query-stop IRP if the following is true:
  - The driver **must not drop I/O** requests and does **not have** a mechanism for **queuing IRPs**.
  - The **exception** to this rule is a device that is **allowed** to **drop I/O**. The drivers for such a device can succeed query-stop and stop requests without queuing IRPs.

Handle IRP_MN_STOP_DEVICE

```
NTSTATUS HandleStopDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
```

Ensure that the device is **paused**.

Release the hardware **resources**.

Change state to **STOPPED**

Pass the IRP **down** to the next driver.

```
}
```

Handle IRP_MN_STOP_DEVICE

```
NTSTATUS HandleStopDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS
  if (pdx->State != PENDINGSTOP){
    <complicated stuff>
  }
}
```

Release the hardware **resources**.

Change state to **STOPPED**

Pass the IRP **down** to the next driver.

Handle IRP_MN_STOP_DEVICE

```
NTSTATUS HandleStopDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS
  if (pdx->State != PENDINGSTOP){
    <complicated stuff>
  }

  StopDevice(pdx);

  pdx->State = STOPPED;
  return DefaultPnpHandler(pdx, Irp);
}
```

Handle IRP_MN_STOP_DEVICE

```
NTSTATUS HandleStopDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS
  if (pdx->State != PENDINGSTOP){
    <complicated stuff>
  }

  StopDevice(pdx);
}
```

Change state to **STOPPED**

Pass the IRP **down** to the next driver.

Stopping a Device

- The exact operations depend on the device and the driver, such as
  - **disconnecting** an **interrupt** with **IoDisconnectInterrupt**
  - **freeing** physical **address ranges** with **MmUnmapIoSpace**
  - **freeing I/O ports**.

```
  StopDevice(pdx);

  pdx->State = STOPPED;
  return DefaultPnpHandler(pdx, Irp);
}
```

Handle IRP_MN_STOP_DEVICE

```
NTSTATUS HandleStopDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS
  if (pdx->State != PENDINGSTOP){
    <complicated stuff>
  }

  StopDevice(pdx);

  pdx->State = STOPPED;
}
```

Pass the IRP **down** to the next driver.

Handle IRP_MN_CANCEL_STOP_DEVICE

```
NTSTATUS HandleCancelStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
}
```

If the driver is **not PENDINGSTOP**, let **lower driver** handle it.

Postpone restarting the device.

Restart the **IRP Queue**.

Complete the IRP

Handle IRP_MN_CANCEL_STOP_DEVICE

```
NTSTATUS HandleCancelStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != PENDINGSTOP)
    return DefaultPnpHandler(pdx, Irp);
```

Postpone restarting the device.

Restart the **IRP Queue**.

Complete the IRP

```
}
```

Handle IRP_MN_CANCEL_STOP_DEVICE

```
NTSTATUS HandleCancelStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != PENDINGSTOP)
    return DefaultPnpHandler(pdx, Irp);
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  pdx->State = WORKING;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  RestartRequests(&pdx->Queue, stack->DeviceObject);
  return CompleteRequest(Irp, status);
}
```

Handle IRP_MN_CANCEL_STOP_DEVICE

```
NTSTATUS HandleCancelStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != PENDINGSTOP)
    return DefaultPnpHandler(pdx, Irp);
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  pdx->State = WORKING;
```

Restart the **IRP Queue**.

Complete the IRP

```
}
```

Plug & Play

Removing a Device

Handle IRP_MN_CANCEL_STOP_DEVICE

```
NTSTATUS HandleCancelStop(PDEVICE_EXTENSION pdx, PIRP Irp)
{
  Irp->IoStatus.Status = STATUS_SUCCESS;
  if (pdx->State != PENDINGSTOP)
    return DefaultPnpHandler(pdx, Irp);
  NTSTATUS status = ForwardAndWait(pdx, Irp);
  pdx->State = WORKING;
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  RestartRequests(&pdx->Queue, stack->DeviceObject);
```

Complete the IRP

```
}
```

When Removing a Device?

- The PnP Manager directs drivers to remove their device objects for a device when
  - the device **has been**, or is **being**, **physically removed** from the machine.
  - a user requests to **update** the **drivers** for a device.
  - the device is **disabled** or **fails to start** (Windows 2000 and later).