

# Handling Device I/O

主講人：虞台文

1

## Content

- ⌘ Overview
- ⌘ Accessing Data Buffers
- ⌘ Hardware Abstraction Layer
- ⌘ Hardware Configuration
- ⌘ Port I/O & Memory I/O
- ⌘ Servicing Interrupts
- ⌘ DMA

2

# Handling Device I/O

## Overview

3

## The Main Task of a Driver

- ⌘ **Read/Write** data from/to a device.
- ⌘ **User-mode** applications use a set of standard API's to access the device.
  - **CreateFile**
  - **ReadFile**
  - **WriteFile**
  - **DeviceIoControl**
- ⌘ Most of the corresponding IRPs finally will be fulfilled by **kernel-mode drivers** through communicating with the hardware.

4

# Handling Device I/O

- ⌘ Hardware Configuration
  - **IRP\_MN\_STARTDEVICE**
- ⌘ Start Device I/O
  - **StartIO** Procedure
- ⌘ Handling Interrupt
  - **Interrupt Service Routine (ISR)**
- ⌘ Defer Procedure Call
  - **DPCForISR**

5

## Hardware Abstraction Layer

- ⌘ An NT-based operating system component that provides **platform-specific support** for the NT **kernel**, the **I/O Manager**, **kernel-mode debuggers**, and the lowest-level **device drivers**.
- ⌘ The HAL exports routines that **abstract** platform-specific hardware details about **caches**, I/O **buses**, **interrupts**, and so forth, and provides an **interface** between the platform's **hardware** and the **system software**.

6

## Accessing Data Buffers

- ⚡ The ultimate goal of a device driver is to **read/write** data from/to a device on a **user-mode data buffer**.
- ⚡ Methods:
  - **PIO**
    - Buffered** I/O, **Direct** I/O and **Neither**
  - **DMA**
    - Adapter** Objects

7

## Handling Device I/O

### Accessing Data Buffers

8

## Addressing a Data Buffer

- ⚡ Buffered I/O
  - Use a **system buffer** created by I/O Manager.
- ⚡ Direct I/O
  - User-Mode data buffer is **locked**
  - Access the locked pages through the **MDL**.
- ⚡ Neither Buffered Nor Direct I/O
  - The I/O Manager simply passes the **user-mode virtual address** to you.
  - You work- very **carefully**- with the user-mode address

9

## The IRP

Type	Size
MdlAddress	
Flags	
AssociatedIrp	
ThreadListAddress	
IoStatus	
RequestorMode	PendingReturned
Cancel	CancelIrql
StackCount	CurrentLocation
ApcEnvironment	AllocationFlags
UserIosb	
UserEvent	
Overlay	
CancelRoutine	
UserBuffer	
Tail	

10

## The IRP

Type	Size
MdlAddress	
Flags	
AssociatedIrp	
union { <pre>                     struct _IRP *MasterIrp;                     .                     PVOID SystemBuffer;                     } AssociatedIrp;                     </pre>	
UserBuffer	
Tail	

11

## Specifying a Buffering Mode

```

NTSTATUS AddDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject)
    
```

1. Call **IoCreateDevice** to create an **FDO** or **FIDO** for the device being added.
2. Create one or more symbolic links (device interfaces) to the device.
3. Initialize your device extension and the **Flags** member of the device object.
4. Call **IoAttachDeviceToDeviceStack** to Attach the device object to the device stack.
5. Clear the **DO\_DEVICE\_INITIALIZING** flag in the **FDO** or filter **DO**.

12

Flag	Description
DO_BUFFERED_IO	Reads/write using the <b>buffered method</b>
DO_EXCLUSIVE	Open by <b>one thread at a time</b>
DO_DIRECT_IO	Reads/write using the <b>direct method</b>
DO_DEVICE_INITIALIZING	Device object <b>isn't</b> initialized yet.
DO_POWER_PAGABLE	IRP_MJ_PNP must be handled at <b>PASSIVE_LEVEL</b>
DO_POWER_INRUSH	Requires large <b>inrush</b> of current during <b>power-on</b> .

1. Call IoCreateDevice to create an FDO or FIDO for the device being added.
2. Create one or more symbolic links (device interfaces) to the device.
3. Initialize your device extension and the Flags member of the device object.
4. Call IoAttachDeviceToDeviceStack to Attach the device object to the device stack.
5. Clear the DO\_DEVICE\_INITIALIZING flag in the FDO or filter DO.

13

## Flag Setting

- ⚡ The **Lowest-Level Driver**
  - ORing do->Flags with DO\_DIRECT\_IO or with DO\_BUFFERED\_IO.
- ⚡ The **Intermediate Drivers**
  - do->Flags **matches** the next-lower one.
- ⚡ The **Highest-Level Driver**
  - In general, do->Flags to **matches** the next-lower one.
  - However, it can **avoid setting Flags** for either buffered or direct I/O if the driver writer decides the additional work involved will pay off in **better driver performance**.

14

## Flag Setting for the Lowest-Level Driver

```
NTSTATUS AddDevice(...)
{
    PDEVICE_OBJECT fdo;
    IoCreateDevice(..., &fdo);
    .....
    fdo->Flags = DO_BUFFERED_IO;
    <or>
    fdo->Flags = DO_DIRECT_IO;
    <or>
    fdo->Flags = 0; // i.e., neither direct nor buffered.
    .....
}
```

17

## Flag Setting for Higher-Lever Drivers

```
NTSTATUS AddDevice(..., PDEVICE_OBJECT pdo)
{
    PDEVICE_OBJECT fdo;
    IoCreateDevice(..., &fdo);
    .....
    pdo->NextDo = IoAttachDeviceToDeviceStack(fdo, pdo);
    // Maintain Various I/O and power flags.
    fdo->Flags |= (pdo->NextDo->Flags
    & (DO_BUFFERED_IO | DO_DIRECT_IO));
    fdo->Flags |= (pdo->NextDo->Flags
    & (DO_POWER_INRUSH | DO_POWER_PAGABLE));
    .....
}
```

1

## The Buffered I/O

- ⚡ A driver that services an **interactive** or **slow** device, or one that usually **transfers** relatively **small amounts** of data at a time, should use the **buffered I/O** transfer method.
- ⚡ This will **improve** overall physical **memory usage**, because the Memory Manager **doesn't** need to **lock** down a full physical page for each transfer, as it does for drivers that request direct I/O.
- ⚡ Generally, **video, keyboard, mouse, serial,** and **parallel** drivers request buffered I/O.

## How Buffered I/O Works?

```
VOID uva; // <= user-mode virtual buffer address
ULONG length; // <= length of user-mode buffer
VOID sva = ExAllocatePoolWithQuota(NonPagedPoolCacheAligned,
length);

if (writing)
    RtlCopyMemory(sva, uva, length);
Irp->AssociatedIrp.SystemBuffer = sva;
PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);
if (reading)
    stack->Parameters.Read.Length = length;
else
    stack->Parameters.Write.Length = length;
<code to send and await IRP>
if (reading)
    RtlCopyMemory(uva, sva, length);
ExFreePool(sva);
```

1

## The Direct I/O

- Drivers for devices that can transfer **large amounts** of data **at a time** should use direct I/O for those transfers.
- This improves a driver's performance:
  - Reducing its **interrupt** overhead
  - Eliminating memory **allocation** and **copy**
- Generally, **mass-storage** device drivers request direct I/O for transfer requests.

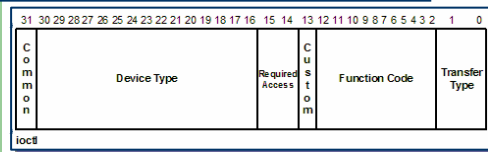
19

## The Direct I/O

- Setting up a **direct I/O transfers** varies slightly, depending on whether **DMA** or **PIO** is being used.
- Even drivers **that request direct I/O** use **buffered I/O** to satisfy certain IRPs.
  - In particular, drivers **typically** use **buffered I/O** for some **IRP\_MJ\_DEVICE\_CONTROL** requests that require data transfers, **whether or not** the driver uses **direct I/O** for **read** and **write** operations.

20

## I/O Control Code



- In particular, drivers **typically** use **buffered I/O** for some **IRP\_MJ\_DEVICE\_CONTROL** requests that require data transfers, **whether or not** the driver uses **direct I/O** for **read** and **write** operations.

21

## I/O Control Code

Flag	Description	Transfer Type
METHOD_BUFFERED	Suitable for transfers <b>small amounts of data</b> for the request.	
METHOD_IN_DIRECT	<b>Read a large amount of data</b> for the request using DMA or PIO and must transfer the data quickly.	
METHOD_OUT_DIRECT	<b>Write a large amount of data</b> to the device for the request using DMA or PIO and must transfer the data quickly.	
METHOD_NEITHER	<b>Workable only if running in the context of the thread that originates the I/O control request.</b>	

22

## How Direct I/O Works?

```

PVOID uva; // <= user-mode virtual buffer address
ULONG length; // <= length of user-mode buffer
KPROCESSOR_MODE mode; // <= either KernelMode or UserMode

PMDL mdl = IoAllocateMdl(uva, length, FALSE, TRUE, Irp);

MmProbeAndLockPages(mdl, mode,
                    reading ? IoWriteAccess : IoReadAccess);

<code to send and await IRP>

MmUnlockPages(mdl);

IoFreeMdl(mdl);
    
```

23

Don't access the structure directly.

## The Memory Descriptor List (MDL)

```

typedef struct _MDL {
    struct _MDL *Next;
    USHORT Size;
    USHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
    
```

24

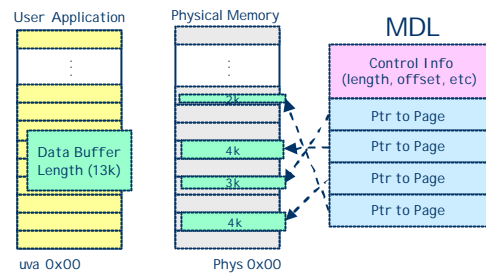
Never access the structure directly.

## The Memory Descriptor List (MDL)

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

25

## The Memory Descriptor List (MDL)



26

## Address Mappings and MDLs

⌘ MmGetPhysicalAddress	⌘ MmCreateMdl
⌘ MmGetMdlVirtualAddress	⌘ MmPrepareMdlForReuse
⌘ MmGetSystemAddressForMdlSafe	⌘ MmInitializeMdl
⌘ MmBuildMdlForNonPagedPool	⌘ MmMapIoSpace
⌘ MmGetMdlByteCount	⌘ MmUnmapIoSpace
⌘ MmGetMdlByteOffset	⌘ MmProbeAndLockPages
⌘ MmMapLockedPages	⌘ MmUnlockPages
⌘ MmUnmapLockedPages	⌘ MmSecureVirtualMemory
⌘ MmMapLockedPagesWithReservedMapping	⌘ MmUnsecureVirtualMemory
⌘ MmAllocateMappingAddress	⌘ MmProtectMdlSystemAddress
⌘ MmUnmapReservedMapping	⌘ IoAllocateMdl
⌘ MmIsAddressValid	⌘ IoBuildPartialMdl
⌘ MmSizeOfMdl	⌘ IoFreeMdl

27

## Using Direct I/O with PIO

```
NTSTATUS DispatchReadOrWrite (PDEVICE_OBJECT fdo,
                             PIRP Irp)
{
    PCHAR buffer;
    . . . . .

    buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
                                           NormalPagePriority);
    . . . . .
}
```

## Using Neither Buffered Nor Direct I/O

- ⌘ If device objects specify **neither direct nor buffered** I/O, then the I/O Manager will pass the original **user-space virtual addresses** in IRPs sent to the driver.
- ⌘ Typically only **highest-level** drivers, such as FSDs, can use this method for accessing buffers.

29

## How 'Neither' I/O Works?

```
PVOID uva; // <== user-mode virtual buffer address
ULONG length; // <== length of user-mode buffer

Irp->UserBuffer = uva;

PIO_STACK_LOCATION stack = IoGetNextIrpStackLocation(Irp);

if (reading)
    stack->Parameters.Read.Length = length;
else
    stack->Parameters.Write.Length = length;

<code to send and await IRP>
```

30

You should use a **structured exception** frame in either case to avoid a bug check.

## Manage “Neither I/O” User Buffers

- ⚡ Check the **validity** of the user buffer for read or write **before** accessing the buffer.
  - **ProbeForRead**
  - **ProbeForWrite**
- ⚡ Or, do the **similar jobs** that I/O manager does for you on **Direct I/O**.
  - **IoAllocateMdl**
  - **MmProbeAndLockPages**

31

## Using ‘Neither’ I/O

```

NTSTATUS DispatchRead(PDEVICE_OBJECT fdo, PIRP Irp)
{
    . . . . .
    PVOID buffer = Irp->UserBuffer;
    ULONG length = stack->Parameters.Read.Length;
    if (Irp->RequestorMode != KernelMode){
        _try {
            PMDL mdl = IoAllocateMdl(...);
            MmProbeAndLockPages(...);
            -or-
            ProbeForRead(...);
            <access memory at buffer>
        }
        _except(EXCEPTION_EXECUTE_HANDLER) {
            return CompleteRequest(Irp, GetExceptionCode(), 0);
        }
        . . . . .
    }
}

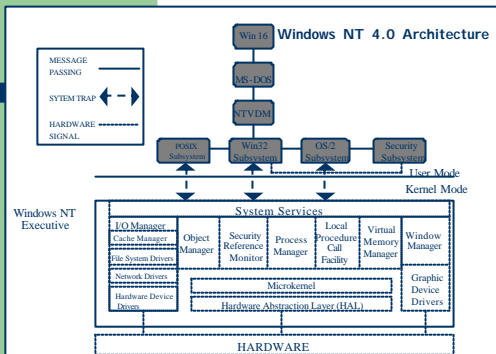
```

32

## Handling Device I/O

Hardware  
Abstraction  
Layer

33



34

## Windows NT and the HAL

- ⚡ Windows NT uses a **Hardware Abstraction Layer (HAL)** to allow NT to **communicate** correctly with the **hardware** on the system.
- ⚡ The HAL is actually a **.DLL** file that creates "hooks" so the **operating system** and the **drivers** do **not** have to **know** anything about the **processor**, the **BIOS**, or the **system board**.
- ⚡ When NT wants to send something to the processor, it uses a **driver**, **Registry** information, and an **API** built into NT, and loaded during installation.

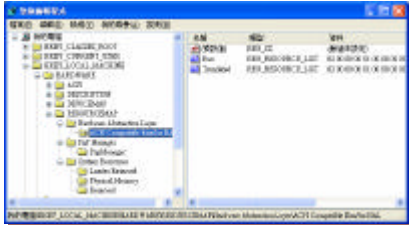
35

## Core Windows 2000 System Files

Filename	Components
Ntoskrnl.exe	Executive and kernel
Ntkrnlpa.exe	Executive and kernel with support for Physical Address Extension (PAE), which allows addressing of up to 64 GB of physical memory
Hal.dll	Hardware abstraction layer
Win32k.sys	Kernel-mode part of the Win32 subsystem
Ntdll.dll	Internal support functions and system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	CoreWin32 subsystem DLLs

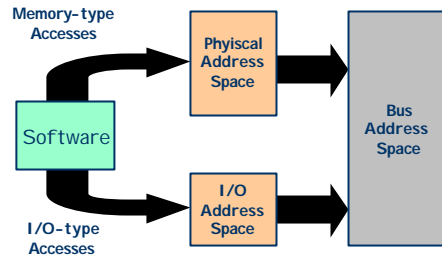
36

## The Registry for HAL



37

## Device Memory Access



38

## HAL Functions

Access Width	Functions for Port Access	Functions for Memory Access
8 bits	READ_PORT_UCHAR WRITE_PORT_UCHAR	READ_REGISTER_UCHAR WRITE_REGISTER_UCHAR
16 bits	READ_PORT_USHORT WRITE_PORT_USHORT	READ_REGISTER_USHORT WRITE_REGISTER_USHORT
32 bits	READ_PORT_ULONG WRITE_PORT_ULONG	READ_REGISTER_ULONG WRITE_REGISTER_ULONG
String of 8-bit bytes	READ_PORT_BUFFER_UCHAR WRITE_PORT_BUFFER_UCHAR	READ_REGISTER_BUFFER_UCHAR WRITE_REGISTER_BUFFER_UCHAR
String of 16-bit words	READ_PORT_BUFFER_USHORT WRITE_PORT_BUFFER_USHORT	READ_REGISTER_BUFFER_USHORT WRITE_REGISTER_BUFFER_USHORT
String of 32-bit doublewords	READ_PORT_BUFFER_ULONG WRITE_PORT_BUFFER_ULONG	READ_REGISTER_BUFFER_ULONG WRITE_REGISTER_BUFFER_ULONG

Port I/O

Memory I/O

39

## Handling Device I/O

Hardware Configuration

40

## I/O Resources

Resource Type	Overview
Port	Map port range; save base port address in device extension
Memory	Map memory range; save base address in device extension
Dma	Call <code>IoGetDmaAdapter</code> to create an adapter object
Interrupt	Call <code>IoConnectInterrupt</code> to create an interrupt object that points to your interrupt service routine (ISR)

41

Handle PnP IRP's minor code  
`IRP_MN_STARTDEVICE`

## Setup Hardware Configuration

```

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);
    .....
}
    
```

### Handle PnP IRP's minor code IRP\_MN\_STARTDEVICE

## Setup Hardware Configuration

```

NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
    .....
    // call StartDevice()
    PCM_PARTIAL_RESOURCE_LIST raw, translated;

    raw = &stack->Parameters.StartDevice
        .PartialResourceList;
    translated = &stack->Parameters
        .AllocatedResourcesTranslated->List[0].PartialResourceList;
    status = StartDevice(pdx, raw, translated);
    .....
}

```


The resources given by the PnP Manager through negotiation.

## CM\_PARTIAL\_RESOURCE\_LIST

```

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;

```



44

## CM\_PARTIAL\_RESOURCE\_LIST

Version	Revision	Count		Flags	PartialDescriptors[0];
Type	ShareDisposition	Type	u		
Type	ShareDisposition	Flags	u	PartialDescriptors[1];	
Type	ShareDisposition	Flags	u		
				⋮	

45

## CM\_PARTIAL\_RESOURCE\_LIST

Version	Revision	Type Value	u Member
Count		CmResourceTypePort	u.Port
Type	ShareDisposition	CmResourceTypeInterrupt	u.Interrupt
u		CmResourceTypeMemory	u.Memory
u		CmResourceTypeDma	u.Dma
u		CmResourceTypeDevicePrivate	u.DevicePrivate
Type	ShareDisposition	CmResourceTypeBusNumber	u.BusNumber
u		CmResourceTypeDeviceSpecific	u.DeviceSpecificData
u		u.DeviceSpecificData	u.DevicePrivate
u		CmResourceTypePcCardConfig	u.DevicePrivate
u		CmResourceTypeConfigData	Reserved for system use
u		CmResourceTypeNonArbitrated	Not used

46

## CM\_PARTIAL\_RESOURCE\_LIST


Version	Revision	Count		Flags	PartialDescriptors[0];
Type	ShareDisposition	Type	u		
Type	ShareDisposition	Flags	u	PartialDescriptors[1];	
Type	ShareDisposition	Flags	u		
		Value	Definition		
		CmResourceShareDeviceExclusive	exclusive use of the resource		
		CmResourceShareDriverExclusive	Not supported for WDM drivers		
		CmResourceShareShared	shared without restriction		

47

## CM\_PARTIAL\_RESOURCE\_LIST

Its value depends on the resource type.

Version	Revision	Count		Flags	PartialDescriptors[0];
Type	ShareDisposition	Type	u		
Type	ShareDisposition	Flags	u	PartialDescriptors[1];	
Type	ShareDisposition	Flags	u		
				⋮	



48

Handle PnP IRP's minor code  
IRP\_MN\_STARTDEVICE

## Setup Hardware Configuration

```

NTSTATUS HandleStartDevice(PDEVICE_EXTENSION pdx, PIRP Irp)
{
    // call StartDevice()
    PCM_PARTIAL_RESOURCE_LIST raw, translated;

    raw = &stack->Parameters.StartDevice
    .PartialResourceList;
    translated = &stack->Parameters
    .AllocatedResourcesTranslated->List[0].PartialResourceList;
    status = StartDevice(pdx, raw, translated);
}

```

The resources given by the PnP Manager through negotiation.

## Identify the Device Resources

```

NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    <local variable declarations>
    for (ULONG i = 0; i < nres; ++i, ++resource){

```

Enumerate each type of resource, and save their information to the device extension and local variables.

```

    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}

```

## Identify the Device Resources

```

NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    <local variable declarations>
    for (ULONG i = 0; i < nres; ++i, ++resource){
        switch (resource->Type){
            case CmResourceTypePort:
                <save port info in local variables>
                break;
            case CmResourceTypeInterrupt:
                <save interrupt info in local variables>
                break;
            case CmResourceTypeMemory:
                <save memory info in local variables>
                break;
            case CmResourceTypeDma:
                <save DMA info in local variables>
                break;
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}

```

## Handling Device I/O

Port I/O  
&  
Memory I/O

## HAL Functions

Access Width	Functions for Port Access	Functions for Memory Access
8 bits	READ_PORT_UCHAR WRITE_PORT_UCHAR	READ_REGISTER_UCHAR WRITE_REGISTER_UCHAR
16 bits	READ_PORT_USHORT WRITE_PORT_USHORT	READ_REGISTER_USHORT WRITE_REGISTER_USHORT
32 bits	READ_PORT_ULONG WRITE_PORT_ULONG	READ_REGISTER_ULONG WRITE_REGISTER_ULONG
String of 8-bit bytes	READ_PORT_BUFFER_UCHAR WRITE_PORT_BUFFER_UCHAR	READ_REGISTER_BUFFER_UCHAR WRITE_REGISTER_BUFFER_UCHAR
String of 16-bit words	READ_PORT_BUFFER_USHORT WRITE_PORT_BUFFER_USHORT	READ_REGISTER_BUFFER_USHORT WRITE_REGISTER_BUFFER_USHORT
String of 32-bit doublewords	READ_PORT_BUFFER_ULONG WRITE_PORT_BUFFER_ULONG	READ_REGISTER_BUFFER_ULONG WRITE_REGISTER_BUFFER_ULONG

Port I/O                      Memory I/O

## Resource Type CmResourceTypePort

```

typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            PHYSICAL_ADDRESS Start;
            ULONG Length;
        } Port;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR,
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;

```

## Resource Type CmResourceTypePort

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            PHYSICAL_ADDRESS Start;
            ULONG Length;
        } Port;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR,
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

Version		Revision	
Count			
Type	ShareDisposition	Flags	
u			
Type	ShareDisposition	Flags	
u			

55

## Setup Port I/O Resources

```
typedef struct _DEVICE_EXTENSION {
    . . . . .
    PCHAR portbase;
    ULONG nports;
    BOOLEAN mappedport;
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

56

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    PHYSICAL_ADDRESS portbase; // base address of range
    for (ULONG i = 0; i < nres; ++i, ++resource) {
        switch (resource->Type) {
            case CmResourceTypePort:
                <save port info in local variables>
                break;
            case CmResourceTypeInterrupt:
                <save interrupt info in local variables>
                break;
            case CmResourceTypeMemory:
                <save memory info in local variables>
                break;
            case CmResourceTypeDma:
                <save DMA info in local variables>
                break;
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

57

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    PHYSICAL_ADDRESS portbase; // base address of range
    for (ULONG i = 0; i < nres; ++i, ++resource) {
        switch (resource->Type) {
            case CmResourceTypePort:
                pdx->portbase = resource->u.Port.Start;
                pdx->nports = resource->u.Port.Length;
                pdx->mappedport =
                    (resource->Flags & CM_RESOURCE_PORT_IO) == 0;
                break;
            case CmResourceTypeDma:
                <save DMA info in local variables>
                break;
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

58

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    PHYSICAL_ADDRESS portbase; // base address of range
    for (ULONG i = 0; i < nres; ++i, ++resource) {
        switch (resource->Type) {
            case CmResourceTypePort:
                if (pdx->mappedport) {
                    pdx->portbase = (PCHAR) MmMapIoSpace(portbase,
                    pdx->nports, MmNonCached);
                    if (!pdx->portbase)
                        return STATUS_NO_MEMORY;
                }
                else
                    pdx->portbase = (PCHAR) portbase.QuadPart;
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

59

## Release Port I/O Resources

```
VOID StopDevice(...)
{
    . . . . .
    if (pdx->portbase && pdx->mappedport)
        MmUnmapIoSpace(pdx->portbase, pdx->nports);
    pdx->portbase = NULL;
    . . . . .
}
```

60

## Resource Type CmResourceTypeMemory

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            PHYSICAL_ADDRESS Start;
            ULONG Length;
        } Memory;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR,
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

61

## Setup Memory I/O Resources

```
typedef struct _DEVICE_EXTENSION{
    . . . . .
    PUCHAR membase;
    ULONG nbytes;
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

62

## Setup Memory I/O Resources

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
    PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    PHYSICAL_ADDRESS membase; // base address of range
    for (ULONG i = 0; i < nres; ++i, ++resource){
        switch (resource->Type){
            . . . . .
            case CmResourceTypeMemory:
                <save memory info in local variables>
                break;
            . . . . .
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

63

## Setup Memory I/O Resources

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
    PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    PHYSICAL_ADDRESS membase; // base address of range
    for (ULONG i = 0; i < nres; ++i, ++resource){
        switch (resource->Type){
            . . . . .
            case CmResourceTypeMemory:
                membase = resource->u.Memory.Start;
                pdx->nbytes = resource->u.Memory.Length;
                break;
            . . . . .
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

64

## Setup Memory I/O Resources

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
    PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    PHYSICAL_ADDRESS membase; // base address of range
    for (ULONG i = 0; i < nres; ++i, ++resource){
        switch (resource->Type){
            . . . . .
            case CmResourceTypeMemory:
                membase = resource->u.Memory.Start;
                pdx->nbytes = resource->u.Memory.Length;
                break;
            . . . . .
        }
    }
    pdx->membase = (PUCHAR) MmMapIoSpace(membase, pdx->nbytes,
        MmNonCached);
    if (!pdx->membase)
        return STATUS_NO_MEMORY;
    . . . . .
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

65

## Release Memory I/O Resources

```
VOID StopDevice(...)
{
    . . . . .
    if (pdx->membase)
        MmUnmapIoSpace(pdx->membase, pdx->nbytes);
    pdx->membase = NULL;
    . . . . .
}
```

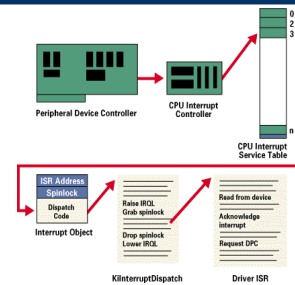
66

# Handling Device I/O

## Servicing Interrupts

67

# NT Interrupt Handling



68

# The Interrupt Objects

- Each driver of a physical device that generates interrupts must **register** an **ISR** **before starting** the device.
- When a driver registers an **ISR**, the system creates an **interrupt object**.

69

# The Interrupt Objects

- An **interrupt object** includes the following information:
  - Processor **affinity** ?
    - The set of processors on which interrupts can occur.
  - The system-assigned **interrupt vector**.
  - The **DIRQL** assigned to each system interrupt vector.
  - The address of the **InterruptService** routine.
  - Additional** information for internal use by the system.

70

# Registering an ISR

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR,
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

71

# Registering an ISR

```
NTSTATUS IoConnectInterrupt(
    OUT PK_INTERRUPT *InterruptObject,
    IN PKSERVICE_ROUTINE ServiceRoutine,
    IN PVOID ServiceContext,
    IN PKSPIN_LOCK SpinLock OPTIONAL,
    IN ULONG Vector,
    IN KIRQL Irql,
    IN KIRQL SynchronizeIrql,
    IN K_INTERRUPT_MODE InterruptMode,
    IN BOOLEAN ShareVector,
    IN K_AFFINITY ProcessorEnableMask,
    IN BOOLEAN FloatingSave
);
```

72

## Registering an ISR

```
NTSTATUS IoConnectInterrupt(
    OUT PKINTERRUPT *InterruptObject,
    IN PKSERVICE_ROUTINE ServiceRoutine,
    IN PVOID ServiceContext,
    IN PKSPIN_LOCK SpinLock OPTIONAL,
    IN ULONG TRUE the interrupt is handled by the device.
    IN KIRQL FALSE the converse.
);

BOOLEAN InterruptService(
    IN struct _KINTERRUPT *Interrupt,
    IN PVOID ServiceContext
);
```

73

## Registering an ISR

```
NTSTATUS IoConnectInterrupt(
    OUT PKINTERRUPT *InterruptObject,
    IN PKSERVICE_ROUTINE ServiceRoutine,
    IN PVOID ServiceContext,
    IN PKSPIN_LOCK SpinLock OPTIONAL,
);
```

Points to an initialized spin lock, for which the driver supplies the storage. This parameter is **required** if the ISR handles **more than one vector** or if the driver **has more than one ISR**. Otherwise, the driver need not allocate storage for an interrupt spin lock and the input pointer is **NULL**.

74

## Re

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR;
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;

IN PVOID ServiceContext,
IN PKSPIN_LOCK SpinLock OPTIONAL,
IN ULONG Vector,
IN KIRQL Irql,
IN KIRQL SynchronizeIrql,
IN KINTERRUPT_MODE InterruptMode,
IN BOOLEAN ShareVector,
IN KAFFINITY ProcessorEnableMask,
IN BOOLEAN FloatingSave
);
```

75

## Re

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR;
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;

IN PVOID ServiceContext,
IN PKSPIN_LOCK SpinLock OPTIONAL,
IN ULONG Vector,
IN KIRQL Irql,
IN KIRQL SynchronizeIrql,
IN KINTERRUPT_MODE InterruptMode,
);
```

Specifies the **DIRQL** at which the **ISR** will execute. If the ISR handles **more than one interrupt vector** or the driver has **more than one ISR**, this value must be the **highest** of the Irql values passed at **u.Interrupt.Level** in each interrupt resource. Otherwise, the **Irql** and **SynchronizeIrql** values are **identical**.

76

## Re

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR;
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;

IN PVOID ServiceContext,
IN PKSPIN_LOCK SpinLock OPTIONAL,
IN ULONG Vector,
IN KIRQL Irql,
IN KIRQL SynchronizeIrql,
IN KINTERRUPT_MODE InterruptMode,
);

(resource->Flags == CM_RESOURCE_INTERRUPT_LATCHED)
? Latched : LevelSensitive;
```

77

## Re

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
        . . . . .
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR;
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;

IN PVOID ServiceContext,
IN PKSPIN_LOCK SpinLock OPTIONAL,
IN ULONG Vector,
IN KIRQL Irql,
resource->ShareDisposition == CmResourceShareShared;

IN BOOLEAN ShareVector,
IN KAFFINITY ProcessorEnableMask,
IN BOOLEAN FloatingSave
);
```

78

## Re

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        . . . . .
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR,
*PCM_PARTIAL_RESOURCE_DESCRIPTOR;

NTSTATUS
. . . . .
IN PVOID ServiceContext,
IN PKSPIN_LOCK SpinLock OPTIONAL,
IN ULONG Vector,
IN KIRQL Irql,
IN KIRQL SynchronizeIrql,
IN KINTERRUPT_MODE InterruptMode,
IN BOOLEAN ShareVector,
IN KAFFINITY ProcessorEnableMask,
IN BOOLEAN FloatingSave
);
```

79

## Registering an ISR

```
NTSTATUS IoConnectInterrupt(
    OUT PKINTERRUPT *InterruptObject,
    IN PKSERVICE_ROUTINE ServiceRoutine,
    IN PVOID ServiceContext,
    IN PKSPIN_LOCK SpinLock OPTIONAL,
    IN ULONG Vector,
    IN KIRQL Irql,
    IN KAFFINITY ProcessorEnableMask,
    IN BOOLEAN FloatingSave
);
```

Specifies whether to save the floating-point stack when the driver's device interrupts. For X86-based platforms, this value must be set to FALSE.

80

## Setup Interrupt Resources

```
typedef struct _DEVICE_EXTENSION{
    . . . . .
    PKINTERRUPT InterruptObject;
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

81

## Setup Interrupt Resources

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
    PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    <local variable declarations>
    for (ULONG i = 0; i < nres; ++i, ++resource){
        switch (resource->Type){
            . . . . .
            case CmResourceTypeInterrupt:
                <save interrupt info in local variables>
                break;
            . . . . .
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

82

```
ULONG vector; // interrupt vector
KIRQL irql; // interrupt level
KINTERRUPT_MODE mode; // latching mode
KAFFINITY affinity; // processor affinity
BOOLEAN irqshare; // shared interrupt?
```

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
    PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR resource = translated->PartialDescriptors;
    ULONG nres = translated->Count;
    <local variable declarations>
    for (ULONG i = 0; i < nres; ++i, ++resource){
        switch (resource->Type){
            . . . . .
            case CmResourceTypeInterrupt:
                <save interrupt info in local variables>
                break;
            . . . . .
        }
    }
    <use local variables to configure driver & hardware>
    return STATUS_SUCCESS;
}
```

83

```
ULONG vector; // interrupt vector
KIRQL irql; // interrupt level
KINTERRUPT_MODE mode; // latching mode
KAFFINITY affinity; // processor affinity
BOOLEAN irqshare; // shared interrupt?
```

```
NTSTATUS StartDevice(PDEVICE_OBJECT fdo,
    PCM_PARTIAL_RESOURCE_LIST raw, PCM_PARTIAL_RESOURCE_LIST translated)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    irql = (KIRQL) resource->u.Interrupt.Level;
    vector = resource->u.Interrupt.Vector;
    affinity = resource->u.Interrupt.Affinity;
    mode = (resource->Flags == CM_RESOURCE_INTERRUPT_LATCHED)
        ? Latched : LevelSensitive;
    irqshare =
        resource->ShareDisposition == CmResourceShareShared;
    break;
}
<use local variables to configure driver & hardware>
return STATUS_SUCCESS;
}
```

84

```

ULONG vector;           // interrupt vector
KIRQL irq;             // interrupt level
KINTERRUPT_MODE mode; // latching mode
KAFFINITY affinity;   // processor affinity
BOOLEAN irqshare;     // shared interrupt?

typedef struct _DEVICE_EXTENSION{
    . . . . .
    PKINTERRUPT InterruptObject;
    . . . . .
}
NTSTATUS StartDevice(PDRIVER_OBJECT DriverObject,
    PDEVICE_EXTENSION pdx,
    PKINTERRUPT InterruptObject,
    . . . . .
    ULONG mres = translate)
{
    . . . . .
    <local variable declarations>
    for (ULONG i = 0; i < mres; ++i, ++resource){
        switch (resource->Type){
            . . . . .
            case CmResourceTypeInterrupt:
                status = IoConnectInterrupt(&pdx->InterruptObject,
                    (PKSERVICE_ROUTINE) OnInterrupt, (PVOID) pdx, NULL,
                    vector, irq, irq, mode, irqshare, affinity, FALSE);
                . . . . .
                <use local variables to configure driver & hardware>
                return STATUS_SUCCESS;
            . . . . .
        }
    }
}

```

85

Executes at **DIRQL**.

## Handling Interrupts

```

BOOLEAN OnInterrupt(PKINTERRUPT InterruptObject, PVOID Context)
{
    if (<device not interrupting>)
        return FALSE;

    <handle interrupt>

    return TRUE;
}

```

86

Executes at **DIRQL**.

## Handling Interrupts

Usually, we choose the device extension as the context while calling **IoConnectInterrupt**.

```

BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject, PVOID Context)
{
    if (<device not interrupting> )
        return FALSE;

    <handle interrupt>

    return TRUE;
}

```

Leave the ISR as soon as possible. Otherwise, schedule a DPC.

87

Executes at **DIRQL**.

## Does My Device Interrupt?

```

BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject, PVOID Context)
{
    UCHAR devstatus = READ_PORT_UCHAR(pdx->portbase + ???);
    if ((devstatus & MY_DEV_INT))
        return FALSE;

    <handle interrupt>

    return TRUE;
}

```

88

Executes at **DIRQL**.

## Synchronizing Operations with the ISR

```

BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject, PVOID Context)
{
    if (<device not interrupting> )
        return FALSE;

    <handle interrupt>

    return TRUE;
}

```

ISR may share data and hardware resources with other parts of the driver.

Skip ↻

89

Callers' **IRQL ? DIRQL** of an associated interrupt object.

## Access Resources Shared with ISR

```

BOOLEAN KeSynchronizeExecution(
    IN PKINTERRUPT Interrupt,
    IN PKSynchronizeRoutine SynchronizeRoutine,
    IN PVOID SynchronizeContext
);

```

**Synchronizes** the execution of a given routine with that of the **ISR** associated with the given interrupt object pointer.

90

Callers' **IRQL ? DIRQL** of an associated interrupt object.

A pointer to a set of interrupt objects.

Specifies a caller-supplied **SynchCriticalSection** to be synchronized with the execution of the ISR associated with the interrupt objects.

Return true is succeeded.

```

BOOLEAN KeSynchronizeExecution(
    IN PKINTERRUPT Interrupt,
    IN PKSYNCHRONIZE_ROUTINE SynchronizeRoutine,
    IN PVOID SynchronizeContext
);

BOOLEAN SynchCriticalSection(
    IN PVOID SynchronizeContext
);

```

interrupt object pointer.

91

Callers' **IRQL ? DIRQL** of an associated interrupt object.

### Access Resources Shared with ISR

```

BOOLEAN KeSynchronizeExecution(
    IN PKINTERRUPT Interrupt,
    IN PKSYNCHRONIZE_ROUTINE SynchronizeRoutine,
    IN PVOID SynchronizeContext
);

```

- ✦ The IRQL is **raised** to **DIRQL** specified on call to IoConnectInterrupt.
- ✦ Synchronized with the corresponding ISR by **acquiring** the associated **interrupt object spin lock**.
- ✦ **SynchronizeContext** will be passed to the **SynchCriticalSection** routine.
- ✦ The **SynchCriticalSection** routine runs at **DIRQL**, so it must execute very quickly.

92

### Example

```

VOID StartIo(...) -or- DpcForIsr(...)
{
    ...
    KeSynchronizeExecution(pdx->InterruptObject,
        TransferFirstSection, (PVOID) pdx);
    ...
}

BOOLEAN TransferFirstSection (PVOID context)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) context;
    <initialize device for new operation>
    return TRUE;
}

```

93

Callers' **IRQL ? DIRQL** of an associated interrupt object.

### Acquire/Release Interrupt Spin Locks

```

NTKERNELAPI KIRQL KeAcquireInterruptSpinLock (
    IN PKINTERRUPT Interrupt
);

NTKERNELAPI VOID KeReleaseInterruptSpinLock (
    IN PKINTERRUPT Interrupt,
    IN KIRQL oldIrql
);

```

Acquires the spin lock associated with an interrupt object.

Releases an interrupt spin lock acquired by **KeAcquireInterruptSpinLock**.

94

### Example

```

VOID StartIo(...) -or- DpcForIsr(...)
{
    KIRQL oldirql =
    KeAcquireInterruptSpinLock (pdx->InterruptObject);
    <initialize device for new operation>
    KeReleaseInterruptSpinLock (pdx->InterruptObject, oldirql);
}

```

95

Executes at **DIRQL**.

### Defer Procedure Calls

```

BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject, PVOID Context)
{
    if (<device not interrupting>)
        return FALSE;

    Perform necessary operations, e.g., read the hardware status register.
    Determine whether the IRQ completes the current task of the device.
    If so, in general, schedule a DPC here for further processing.

    return TRUE;
}

```

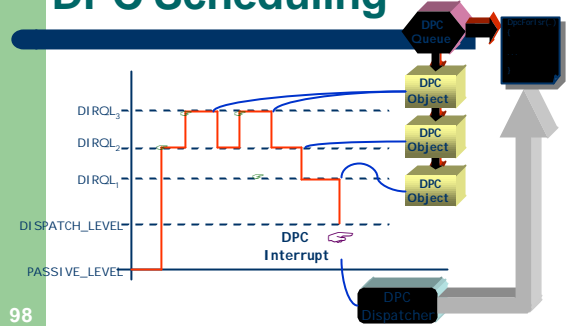
96

## Types of DPC Objects

- ≪ DpcForIsr
  - The system supplies **one** DPC object for each device object.
  - IoInitializeDpcRequest
  - IoRequestDpc
- ≪ CustomDpc
  - A driver can create additional DPC objects if **more than one** DPC is needed.
  - KeInitializeDpc
  - KeInsertQueueDpc
- ≪ Timer Dpc
  - KeInitializeTimer
  - KeSetTimer

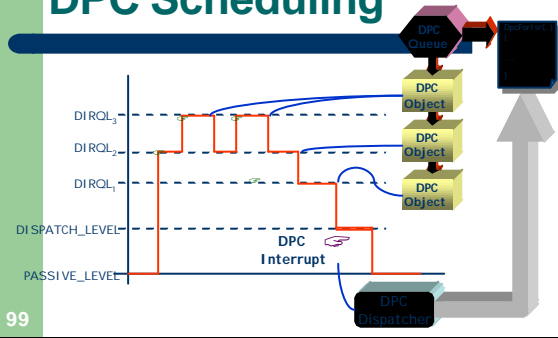
97

## DPC Scheduling



98

## DPC Scheduling



99

## Registering and Queuing a DpcForIsr

```
NTSTATUS AddDevice(...) or HandleStartDevice(...)
{
    ...
    IoInitializeDpcRequest(fdo, DpcForIsr);
    ...
}
```

```
VOID DpcForIsr(
    IN PKDPC Dpc,
    IN struct _DEVICE_OBJECT *DeviceObject,
    IN struct _IRP *Irp,
    IN PVOID Context
);
```

100

## Registering and Queuing a DpcForIsr

```
NTSTATUS AddDevice(...) or HandleStartDevice(...)
{
    ...
    IoInitializeDpcRequest(fdo, DpcForIsr);
    ...
}
```

```
BOOLEAN OnInterrupt(...)
{
    ...
    IoRequestDpc(pdx->DeviceObject, NULL, (PVOID) pdx);
    ...
}
```

101

## Registering and Queuing a CustomDpc

```
VOID KeInitializeDpc(
    IN PKDPC Dpc,
    IN PKDEFERRED_ROUTINE DeferredRoutine,
    IN PVOID DeferredContext
);
```

```
VOID (*PKDEFERRED_ROUTINE)(
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
);
```

102

## Registering and Queuing a CustomDpc

```
BOOLEAN KeInsertQueueDpc(
    IN PRKDPC Dpc,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
);
```

Call this before ending an ISR.

```
VOID (*PKDEFERRED_ROUTINE)(
    IN PRKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
);
```

10  
3

## Guideline for Writing DPCs

- ⚡ **Synchronizing** accessing share data.
- ⚡ Running at I RQL **DISPATCH\_LEVEL**
  - Support routines are restricted, e.g., **don't** access **pageable** memory.
- ⚡ Typically responsible for **starting the next I/O operation** on the device.
- ⚡ How to deal with a device that uses **DMA**?
- ⚡ How to deal with **Overlapped I/O** Operations?
  - That is, **multiple instances** of DPC.
  - E.g., **full-duplex** serial port.

10  
4

## Handling Device I/O

DMA

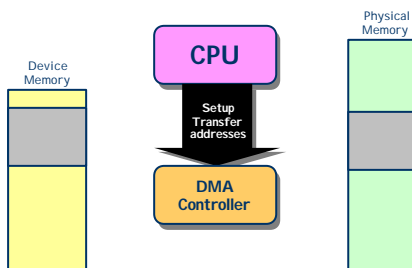
10  
5

## What is DMA?

- ⚡ **Direct memory access (DMA) devices**
  - Directly access (**read** from and **write** to) CPU memory, **without** CPU intervention.
- ⚡ **Non-DMA devices (PIO)**
  - **Cannot** directly access CPU memory.

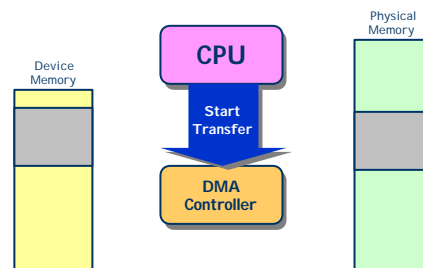
10  
6

## DMA Transfer

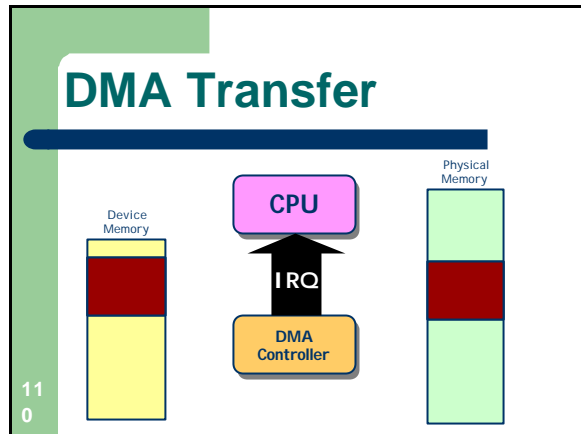
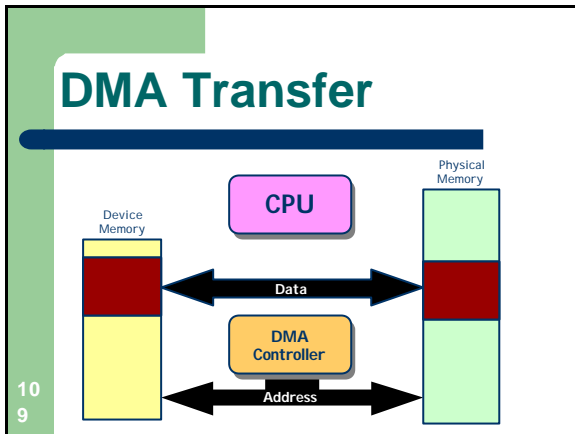


10  
7

## DMA Transfer



10  
8



## DMA Controller Used

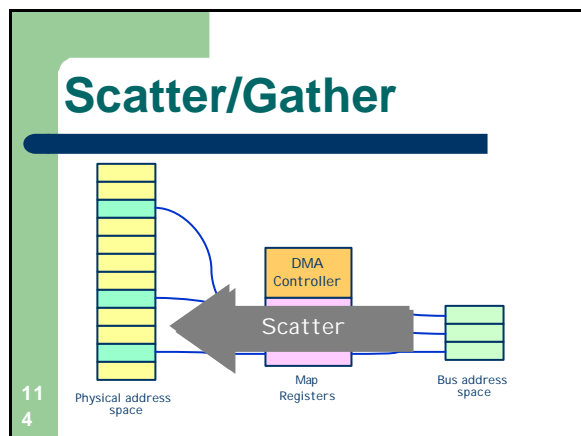
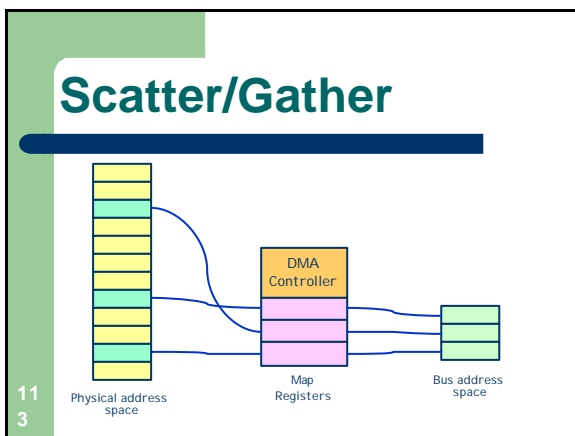
- ≠ **System (Slave) DMA**
  - Such devices are called **slave** devices and are said to "use system (or slave) DMA."
  - Associated with the **ISA** bus
  - Relative **slower** than Bus-Master DMA (even slower than PIO)
- ≠ **Bus-Master DMA**
  - Such devices **arbitrate** with the system for use of the I/O bus, and thus use bus-master DMA.

The number '111' is in the bottom-left corner.

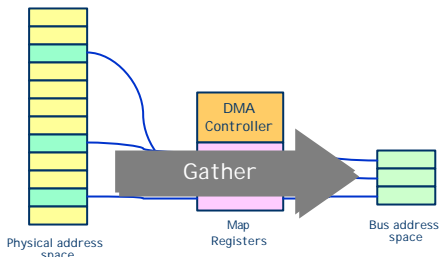
## Capacities & Limitations

- ≠ Most DMA devices require **contiguous physical memory**
  - both for **system** & **bus-master** DMAs.
- ≠ The **access range** of memory
  - **System DMA**: the first **16M**
  - **Bus-Master DMA**: the **full** range
- ≠ **Scatter/Gather** capability
  - **Scatters** incoming data to a collection of physically disjoint memory blocks.
  - **Gathers** outgoing data from a similar collection.

The number '112' is in the bottom-left corner.

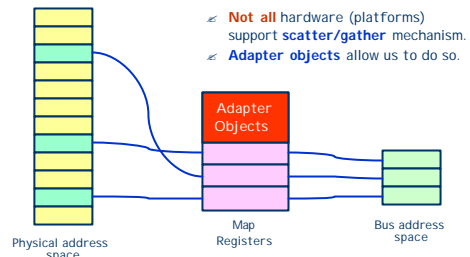


## Scatter/Gather



11  
5

## Abstract Computer Model for DMA Transfers



11  
6

- ⚡ Not all hardware (platforms) support scatter/gather mechanism.
- ⚡ Adapter objects allow us to do so.

## Abstract Computer Model for DMA Transfers



Does using scatter/gather mechanism always good if the hardware doesn't support this capability?

11  
7

- ⚡ Not all hardware (platforms) support scatter/gather mechanism.
- ⚡ Adapter objects allow us to do so.

## Transfer Strategies

- ⚡ **Packet-Based** DMA
  - Transfer a discrete amount of data by using the data **buffer** that accompanies an **IRP**.
- ⚡ **Scatter/Gather** DMA
  - Support **Packet-Based** DMA
- ⚡ **Common-Buffer** DMA
  - DMA transfer is taking place in a **common buffer**.

11  
8

## Adapter Objects

- ⚡ Any driver that uses **direct I/O** and **DMA** must **create an adapter object**.
  - Call **IoGetDmaAdapter** while processing the **IRP\_MN\_STARTDEVICE** minor code for the Pnp IRP.
- ⚡ The adapter object represents either a **DMA controller channel or port**, or a **bus-master device**.

11  
9

## Adapter Objects

- ⚡ On any Windows platform, the **set of adapter objects** usually **includes an adapter object** for:
  - Each **system DMA controller channel or port** to which a slave device is attached.
  - Each **bus-master DMA device** in the machine.

12  
0

# Get DMA Adapter

```

PDMA_ADAPTER IoGetDmaAdapter (
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PDEVICE_DESCRIPTION DeviceDescription,
    IN OUT PULONG NumberOfMapRegisters
);
    
```

12  
1

# Get DMA Adapter

Pointer to the physical device object for the device requesting the DMA adapter structure.

```

    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PDEVICE_DESCRIPTION DeviceDescription,
    IN OUT PULONG NumberOfMapRegisters
);
    
```

Pointer to, on output, the maximum number of map registers that the driver can allocate for any DMA transfer operation.

12  
2

# Get DMA Adapter

```

PDMA_ADAPTER IoGetDmaAdapter (
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PDEVICE_DESCRIPTION DeviceDescription,
    IN OUT PULONG NumberOfMapRegisters
);
    
```

Returns a pointer to a DMA\_ADAPTER structure, which contains pointers to functions that support system-defined DMA operations (to be detailed). If the structure cannot be allocated, the routine returns NULL.

12  
3

# DEVICE\_DESCRIPTION

```

typedef struct _DEVICE_DESCRIPTION {
    ULONG Version;
    BOOLEAN Master;
    BOOLEAN ScatterGather;
    BOOLEAN DemandMode;
    BOOLEAN AutoInitialize;
    BOOLEAN Dma32BitAddresses;
    BOOLEAN IgnoreCount;
    BOOLEAN Reserved1;
    BOOLEAN Dma64BitAddresses;
    ULONG BusNumber;
    ULONG DmaChannel;
    INTERFACE_TYPE InterfaceType;
    DMA_WIDTH DmaWidth;
    DMA_SPEED DmaSpeed;
    ULONG MaximumLength;
    ULONG DmaPort;
} DEVICE_DESCRIPTION, *PDEVICE_DESCRIPTION;
    
```



12  
4

Field Name	Description	Device
Version	Version number of structure- initialize to DEVICE_DESCRIPTION_VERSION	All
Master	Bus-master device	All
ScatterGather	Device supports scatter/gather list	All
DemandMode	Use system DMA controller's demand mode	Slave
AutoInitialize	Use system DMA controller's autoinitialize mode	Slave
Dma32BitAddresses	Can use 32-bit physical addresses	All
IgnoreCount	Controller doesn't maintain an accurate transfer count	Slave
Reserved1	FALSE	
Dma64BitAddresses	Can use 64-bit physical addresses	All
DoNotUse2	0	
DmaChannel	DMA channel number- initialize from Channel attribute of resource descriptor	Slave
InterfaceType	Bus type- initialize to InterfaceType Undefined	All
DmaWidth	Width of transfers- set based on your knowledge of device to Width8Bits, Width16Bits, or Width32Bits	Slave
DmaSpeed	Speed of transfers- set based on your knowledge of device to Compatible TypeA, TypeB, TypeC, or TypeF	Slave
MaximumLength	Maximum length of a single transfer- set based on your knowledge of device (round up to a multiple of PAGE_SIZE)	All
DmaPort	Microchannel-type bus port number- initialize from Port attribute of resource descriptor	Slave

12  
5

# Example

```

typedef struct _DEVICE_EXTENSION {
    . . . . .
    PDMA_ADAPTER AdapterObject;
    ULONG mMaxMapRegisters;
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
    
```

```

NTSTATUS StartDevice(...)
{
    . . . . .
    DEVICE_DESCRIPTION dd;
    RtlZeroMemory(&dd, sizeof(dd));
    dd.Version = DEVICE_DESCRIPTION_VERSION;
    dd.Master = TRUE;
    dd.InterfaceType = InterfaceTypeUndefined;
    dd.MaximumLength = MAXTRANSFER;
    dd.Dma32BitAddresses = TRUE;

    pdx->AdapterObject = IoGetDmaAdapter(pdx->Pdo, &dd,
        &pdx->mMaxMapRegisters);
    . . . . .
}
    
```

Suppose that the device is a PCI bus master but doesn't have scatter/gather capability.

12  
6

## DMA\_ADAPTER

```

PDMA_ADAPTER IoGetDmaAdapter (
    IN PDEVICE_OBJECT PhysicalDeviceObject,

typedef struct _DMA_ADAPTER {
    USHORT Version;
    USHORT Size;
    PDMA_OPERATIONS DmaOperations;
    // Private Bus Device Driver data follows,
} DMA_ADAPTER, *PDMA_ADAPTER;
    
```

12  
7

## DMA Operations

```

typedef struct _DMA_OPERATIONS {
    ULONG Size;
    PFUN_DMA_ADAPTER PutDmaAdapter;
    PALLOCATE_COMMON_BUFFER AllocateCommonBuffer;
    PFREE_COMMON_BUFFER FreeCommonBuffer;
    PALLOCATE_ADAPTER_CHANNEL AllocateAdapterChannel;
    PFLUSH_ADAPTER_BUFFERS FlushAdapterBuffers;
    PFREE_ADAPTER_CHANNEL FreeAdapterChannel;
    PFREE_MAP_REGISTERS FreeMapRegisters;
    PMAP_TRANSFER MapTransfer;
    PGET_DMA_ALIGNMENT GetDmaAlignment;
    PREAD_DMA_COUNTER ReadDmaCounter;
    PGET_SCATTER_GATHER_LIST GetScatterGatherList;
    PPUT_SCATTER_GATHER_LIST PutScatterGatherList;

    // version 2 operations
    PCALCULATE_SCATTER_GATHER_LIST_SIZE CalculateScatterGatherList;
    PBUILD_SCATTER_GATHER_LIST BuildScatterGatherList;
    PBUILD_MDL_FROM_SCATTER_GATHER_LIST BuildMdlFromScatterGatherList;
} DMA_OPERATIONS, *PDMA_OPERATIONS;
    
```

12  
8

## DMA Operations

DmaOperations Function Pointer	Description
PutDmaAdapter	Destroys adapter object
AllocateCommonBuffer	Allocates a common buffer
FreeCommonBuffer	Releases a common buffer
AllocateAdapterChannel	Reserves adapter and map registers
FlushAdapterBuffers	Flushes intermediate data buffers after transfer
FreeAdapterChannel	Releases adapter object and map registers
FreeMapRegisters	Releases map registers only
MapTransfer	Programs one stage of a transfer
GetDmaAlignment	Gets address alignment required for adapter
ReadDmaCounter	Determines residual count
GetScatterGatherList	Reserves adapter and constructs scatter/gather list
PutScatterGatherList	Releases scatter/gather list

12  
9

## Example

```

NTSTATUS StartDevice(...)

VOID StopDevice(...)
{
    .....
    if (pdx->AdapterObject)
        (*pdx->AdapterObject->DmaOperations->PutDmaAdapter)
            (pdx->AdapterObject);
    pdx->AdapterObject = NULL;
    .....
}

pdx->AdapterObject = IoGetDmaAdapter(pdx->Pdo, &dd,
    .....
    &pdx->nMaxMapRegisters);
    .....
}
    
```

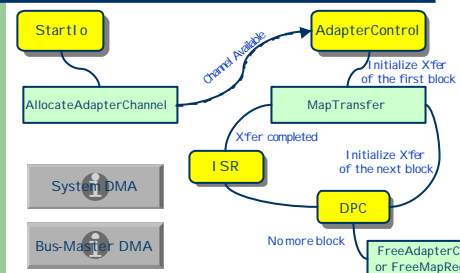
13  
0

## Packet-Based DMA

- Using the method indicates that the device will use **direct method** to transfer data
  - That is, the **DO\_DIRECT\_IO** flag is set.
- Performing DMA transfer in the **StartIo** routine, and **DpcForIsr**, if necessary.
- The DMA controller could be **system DMA** or **bus-master DMA**.

13  
1

## Using Packet-Based DMA Transfer



13  
2

## Example

```
typedef struct _DEVICE_EXTENSION {
    . . . . .
    PDMA_ADAPTER AdapterObject; // device's adapter object
    ULONG nMaxMapRegisters; // max # map registers
    ULONG nMapRegistersAllocated; // # allocated for this xfer
    PVOID regbase; // map register base for this stage
    PVOID vaddr; // virtual address for current stage
    ULONG numxfer; // # bytes transferred so far
    ULONG xfer; // # bytes to transfer during this stage
    ULONG nbytes; // # bytes remaining to transfer
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

13  
3

## StartIo

```
VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    . . . . .
    initialize the state variables for DMA Xfer in the device extension.
    . . . . .
    Prepare the DMA Xfer for the first block of data.
    (May be we need more than one Xfer)
    . . . . .
    Allocate channel to start the Xfer for the first block.
    . . . . .
}
```

13  
4

## StartIo

```
VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    . . . . .
    PMDL mdl = Irp->MdlAddress;
    pdx->numxfer = 0;
    pdx->xfer = pdx->nbytes = MmGetMdlByteCount(mdl);
    pdx->vaddr = MmGetMdlVirtualAddress(mdl);
    . . . . .
    Prepare the DMA Xfer for the first block of data.
    (May be we need more than one Xfer)
    . . . . .
    Allocate channel to start the Xfer for the first block.
    . . . . .
}
```

13  
5

## StartIo

```
VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    . . . . .
    PMDL mdl = Irp->MdlAddress;
    pdx->numxfer = 0;
    pdx->xfer = pdx->nbytes = MmGetMdlByteCount(mdl);
    pdx->vaddr = MmGetMdlVirtualAddress(mdl);
    . . . . .
    ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);
    if (nregs > pdx->nMaxMapRegisters) {
        nregs = pdx->nMaxMapRegisters;
        pdx->xfer = nregs * PAGE_SIZE - MmGetMdlByteOffset(mdl);
    }
    pdx->MapRegistersAllocated = nregs;
    . . . . .
    Allocate channel to start the Xfer for the first block.
    . . . . .
}
```

13  
6

## StartIo

```
VOID StartIo(PDEVICE_OBJECT fdo, PIRP Irp)
{
    . . . . .
    PMDL mdl = Irp->MdlAddress;
    pdx->numxfer = 0;
    pdx->xfer = pdx->nbytes = MmGetMdlByteCount(mdl);
    pdx->vaddr = MmGetMdlVirtualAddress(mdl);
    . . . . .
    ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);
    if (nregs > pdx->nMaxMapRegisters) {
        nregs = pdx->nMaxMapRegisters;
        pdx->xfer = nregs * PAGE_SIZE - MmGetMdlByteOffset(mdl);
    }
    pdx->MapRegistersAllocated = nregs;
    NTSTATUS status = (*pdx->AdapterObject->DmaOperations
        ->AllocateAdapterChannel)(pdx->AdapterObject, fdo, nregs,
        (PDRIVER_CONTROL) AdapterControl, pdx);
    if (NT_SUCCESS(status)) {
        CompleteRequest(Irp, status, 0);
        StartNextPacket(&pdx->Queue, fdo);
    }
    . . . . .
}
```

13  
7

## Allocate DMA Channel

```
typedef struct _DMA_OPERATIONS {
    ULONG Size;
    PFUNTION PutDmaAdapter;
    PALLOCATE_COMMON_BUFFER AllocateCommonBuffer;
    PFREE_COMMON_BUFFER FreeCommonBuffer;
    PALLOCATE_ADAPTER_CHANNEL AllocateAdapterChannel;
    PFLUSH_ADAPTER_BUFFERS FlushAdapterBuffers;
    PFREE_ADAPTER_CHANNEL FreeAdapterChannel;
    PFREE_MAP_REGISTERS FreeMapRegisters;
    PMAP_TRANSFER MapTransfer;
    PGET_DMA_ALIGNMENT GetDmaAlignment;
    PREAD_DMA_COUNTER ReadDmaCounter;
    PGET_SCATTER_GATHER_LIST GetScatterGatherList;
    PPUT_SCATTER_GATHER_LIST PutScatterGatherList;
    . . . . .
    // version 2 operations
    PCALCULATE_SCATTER_GATHER_LIST_SIZE CalculateScatterGatherList;
    PBUILD_SCATTER_GATHER_LIST BuildScatterGatherList;
    PBUILD_MDL_FROM_SCATTER_GATHER_LIST BuildMdlFromScatterGatherList;
} DMA_OPERATIONS, *PDMA_OPERATIONS;
```

13  
8

## Allocate DMA Channel

```
NTSTATUS AllocateAdapterChannel(
    IN PDMA_ADAPTER DmaAdapter,
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG NumberOfMapRegisters,
    IN PDRIVER_CONTROL ExecutionRoutine,
    IN PVOID Context
);
```

13  
9

## Adapter Control Routine

```
NTSTATUS AllocateAdapterChannel(
    IN PDMA_ADAPTER DmaAdapter,
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG NumberOfMapRegisters,
    IN PDRIVER_CONTROL ExecutionRoutine,
    IN PVOID Context
);
IO_ALLOCATION_ACTION AdapterControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID MapRegisterBase,
    IN PVOID Context
);
```

14  
0

## Adapter Control Routine

```
typedef enum _IO_ALLOCATION_ACTION {
    KeepObject = 1,
    DeallocateObject,
    DeallocateObjectKeepRegisters
} IO_ALLOCATION_ACTION, *PIO_ALLOCATION_ACTION;
```

```
IO_ALLOCATION_ACTION AdapterControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID MapRegisterBase,
    IN PVOID Context
);
```

14  
1

## Example

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
    PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
```

Get the necessary parameters of this IRP.

Flush the X'fer buffer due to caching.

Map transfer registers.

Use WRITE\_Xxx HAL routines to begin transferring data.

Return action required after this X'fer.

}

## Example

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
    PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->Queue);
    PMDL mdl = Irp->MdlAddress;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;
```

Flush the X'fer buffer due to caching.

Map transfer registers.

Use WRITE\_Xxx HAL routines to begin transferring data.

Return action required after this X'fer.

}

## Example

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
    PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->Queue);
    PMDL mdl = Irp->MdlAddress;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;

    KeFlushIoBuffers(mdl, isread, TRUE);
```

Map transfer registers.

Use WRITE\_Xxx HAL routines to begin transferring data.

Return action required after this X'fer.

}

## Example

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->Queue);
    PMDL mdl = Irp->MdlAddress;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;

    KeFlushIoBuffers(mdl, isread, TRUE);

    pdx->regbase = regbase;
    PHYSICAL_ADDRESS address =
        (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject, mdl, regbase, pdx->vaddr, pdx->xfer,
         !isread);

    Use WRITE_Xxx HAL routines to begin transferring data.

    Return action required after this X fer.
}
```

## Example

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->Queue);
    PMDL mdl = Irp->MdlAddress;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;

    KeFlushIoBuffers(mdl, isread, TRUE);

    pdx->regbase = regbase;
    PHYSICAL_ADDRESS address =
        (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject, mdl, regbase, pdx->vaddr, pdx->xfer,
         !isread);

    return DeallocateObjectKeepRegisters;
}
```

## Example

```
IO_ALLOCATION_ACTION AdapterControl(PDEVICE_OBJECT fdo,
PIRP junk, PVOID regbase, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->Queue);
    PMDL mdl = Irp->MdlAddress;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;

    KeFlushIoBuffers(mdl, isread, TRUE);

    pdx->regbase = regbase;
    PHYSICAL_ADDRESS address =
        (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject, mdl, regbase, pdx->vaddr, pdx->xfer,
         !isread);

    Use WRITE_Xxx HAL routines to begin transferring data.

    return DeallocateObjectKeepRegisters;
}
```

## Map & Transfer

```
typedef struct _DMA_OPERATIONS {
    ULONG Size;
    PFUN_DMA_ADAPTER PutDmaAdapter;
    PALLOCATE_COMMON_BUFFER AllocateCommonBuffer;
    PFREE_COMMON_BUFFER FreeCommonBuffer;
    PALLOCATE_ADAPTER_CHANNEL AllocateAdapterChannel;
    PFLUSH_ADAPTER_BUFFERS FlushAdapterBuffers;
    PFREE_ADAPTER_CHANNEL FreeAdapterChannel;
    PFREE_MAP_REGISTERS FreeMapRegisters;
    PMAP_TRANSFER MapTransfer;
    PGET_DMA_ALIGNMENT GetDmaAlignment;
    PREAD_DMA_COUNTER ReadDmaCounter;
    PGET_SCATTER_GATHER_LIST GetScatterGatherList;
    PPUT_SCATTER_GATHER_LIST PutScatterGatherList;

    // version 2 operations
    PCALCULATE_SCATTER_GATHER_LIST_SIZE CalculateScatterGatherList;
    PBUILD_SCATTER_GATHER_LIST BuildScatterGatherList;
    PBUILD_MDL_FROM_SCATTER_GATHER_LIST BuildMdlFromScatterGatherList;
} DMA_OPERATIONS, *PDMA_OPERATIONS;
```

14  
8

## Map & Transfer

```
PHYSICAL_ADDRESS MapTransfer(
    IN PDMA_ADAPTER DmaAdapter,
    IN PMDL Mdl,
    IN PVOID MapRegisterBase,
    IN PVOID CurrentVa,
    IN OUT PULONG Length,
    IN BOOLEAN WriteToDevice
);
```

14  
9

## The Defer Procedure Call

15  
0

```

VOID DpcForIsr (KDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PDEVICE_EXTENSION pdx)
{
    Update the state on the completion of the last DMA X'fer.

    if(pdx->nbytes && NT_SUCCESS(status)){ // more transfer required

        Prepare & start the next DMA X'fer.

    }
    else{

        Complete this IRP.

    }
}

```

```

VOID DpcForIsr (KDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_READ;
    (*pdx->AdapterObject->DmaOperations->FlushAdapterBuffers)
    (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
    pdx->nbytes -= pdx->xfer; pdx->numxfer += pdx->xfer;
    NTSTATUS status = STATUS_SUCCESS;

    if(pdx->nbytes && NT_SUCCESS(status)){ // more transfer required

        Prepare & start the next DMA X'fer.

    }
    else{

        Complete this IRP.

    }
}

```

```

VOID DpcForIsr (KDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_READ;
    (*pdx->AdapterObject->DmaOperations->FlushAdapterBuffers)
    (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
    pdx->nbytes -= pdx->xfer; pdx->numxfer += pdx->xfer;
    NTSTATUS status = STATUS_SUCCESS;

    if(pdx->nbytes && NT_SUCCESS(status)){ // more transfer required
        pdx->vaddr = (PVOID)((PUCHAR)pdx->vaddr + pdx->xfer);
        pdx->xfer = pdx->nbytes;
        ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);
        if(nregs > pdx->nMapRegistersAllocated){
            nregs = pdx->nMapRegistersAllocated;
            pdx->xfer = nregs * PAGE_SIZE;
        }
        PHYSICAL_ADDRESS address = (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
        <Use WRITE_Xxx HAL routines to begin transferring data>
    }
    else{

        Complete this IRP.

    }
}

```

```

VOID DpcForIsr (KDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_READ;
    (*pdx->AdapterObject->DmaOperations->FlushAdapterBuffers)
    (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
    pdx->nbytes -= pdx->xfer; pdx->numxfer += pdx->xfer;
    NTSTATUS status = STATUS_SUCCESS;

    if(pdx->nbytes && NT_SUCCESS(status)){ // more transfer required
        pdx->vaddr = (PVOID)((PUCHAR)pdx->vaddr + pdx->xfer);
        pdx->xfer = pdx->nbytes;
        ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);
        if(nregs > pdx->nMapRegistersAllocated){
            nregs = pdx->nMapRegistersAllocated;
            pdx->xfer = nregs * PAGE_SIZE;
        }
        PHYSICAL_ADDRESS address = (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
        <Use WRITE_Xxx HAL routines to begin transferring data>
    }
    else{
        ULONG numxfer = pdx->numxfer;
        (*pdx->AdapterObject->DmaOperations->FreeMapRegisters)
        (pdx->AdapterObject, pdx->regbase, pdx->nMapRegistersAllocated);
        StartNextPacket(&pdx->Queue, fdo);
        CompleteRequest(Irp, status, numxfer);
    }
}

```

```

VOID DpcForIsr (KDPC Dpc, PDEVICE_OBJECT fdo, PIRP junk, PDEVICE_EXTENSION pdx)
{
    PIRP Irp = GetCurrentIrp(&pdx->dqReadWrite);
    PMDL mdl = Irp->MdlAddress;
    BOOLEAN isread = IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_READ;
    (*pdx->AdapterObject->DmaOperations->FlushAdapterBuffers)
    (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
    pdx->nbytes -= pdx->xfer; pdx->numxfer += pdx->xfer;
    NTSTATUS status = STATUS_SUCCESS;

    if(pdx->nbytes && NT_SUCCESS(status)){ // more transfer required
        pdx->vaddr = (PVOID)((PUCHAR)pdx->vaddr + pdx->xfer);
        pdx->xfer = pdx->nbytes;
        ULONG nregs = ADDRESS_AND_SIZE_TO_SPAN_PAGES(pdx->vaddr, pdx->nbytes);
        if(nregs > pdx->nMapRegistersAllocated){
            nregs = pdx->nMapRegistersAllocated;
            pdx->xfer = nregs * PAGE_SIZE;
        }
        PHYSICAL_ADDRESS address = (*pdx->AdapterObject->DmaOperations->MapTransfer)
        (pdx->AdapterObject, mdl, pdx->regbase, pdx->vaddr, pdx->xfer, !isread);
        <Use WRITE_Xxx HAL routines to begin transferring data>
    }
    else{
        ULONG numxfer = pdx->numxfer;
        (*pdx->AdapterObject->DmaOperations->FreeMapRegisters)
        (pdx->AdapterObject, pdx->regbase, pdx->nMapRegistersAllocated);
        StartNextPacket(&pdx->Queue, fdo);
        CompleteRequest(Irp, status, numxfer);
    }
}

```

## FlushAdapterBuffers & FreeMapRegisters

```

typedef struct _DMA_OPERATIONS {
    ULONG Size;
    PFPUT_DMA_ADAPTER PutDmaAdapter;
    PALLOCATE_COMMON_BUFFER AllocateCommonBuffer;
    PFREE_COMMON_BUFFER FreeCommonBuffer;
    PALLOCATE_ADAPTER_CHANNEL AllocateAdapterChannel;
    PFLUSH_ADAPTER_BUFFERS FlushAdapterBuffers;
    PFREE_ADAPTER_CHANNEL FreeAdapterChannel;
    PFREE_MAP_REGISTERS FreeMapRegisters;
    PMAP_TRANSFER MapTransfer;
    PGET_DMA_ALIGNMENT GetDmaAlignment;
    PREAD_DMA_COUNTER ReadDmaCounter;
    PGET_SCATTER_GATHER_LIST GetScatterGatherList;
    PPUT_SCATTER_GATHER_LIST PutScatterGatherList;

    // version 2 operations
    PCALCULATE_SCATTER_GATHER_LIST_SIZE CalculateScatterGatherList;
    PBUILD_SCATTER_GATHER_LIST BuildScatterGatherList;
    PBUILD_MDL_FROM_SCATTER_GATHER_LIST BuildMdlFromScatterGatherList;
} DMA_OPERATIONS, *PDMA_OPERATIONS;

```

## FlushAdapterBuffers & FreeMapRegisters

```

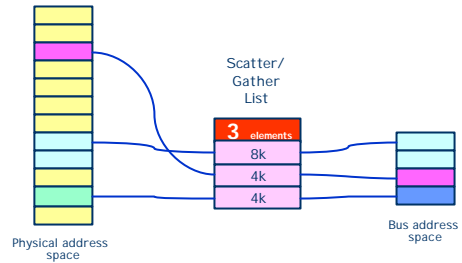
BOOLEAN FlushAdapterBuffers(
    IN PDMA_ADAPTER DmaAdapter,
    IN PMDL Mdl,
    IN PVOID MapRegisterBase,
    IN PVOID CurrentVa,
    IN ULONG Length,
    IN BOOLEAN WriteToDevice
);
    
```

```

VOID FreeMapRegisters(
    IN PDMA_ADAPTER DmaAdapter,
    PVOID MapRegisterBase,
    ULONG NumberOfMapRegisters
);
    
```

15  
7

## Using Scatter/Gather DMA



15  
8

## SCATTER\_GATHER\_LIST

```

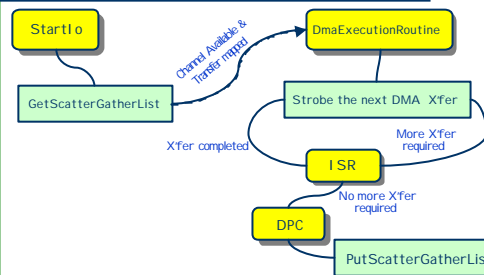
typedef struct _SCATTER_GATHER_LIST {
    ULONG NumberOfElements;
    ULONG_PTR Reserved;
    SCATTER_GATHER_ELEMENT Elements[];
} SCATTER_GATHER_LIST, *PSCATTER_GATHER_LIST;
    
```

```

typedef struct _SCATTER_GATHER_ELEMENT {
    PHYSICAL_ADDRESS Address;
    ULONG Length;
    ULONG_PTR Reserved;
} SCATTER_GATHER_ELEMENT, *PSCATTER_GATHER_ELEMENT;
    
```

15  
9

## Using Scatter/Gather DMA



16  
0

## Example

```

typedef struct _DEVICE_EXTENSION {
    . . . . .
    PDMA_ADAPTER AdapterObject; // device's adapter object
    ULONG nMaxMapRegisters; // max # map registers

    PSCATTER_GATHER_LIST sgl; // s/g list for this transfer
    ULONG isg; // index of 1st s/g element for current stage
    ULONG sgdone; // # elements processed so far
    ULONG nbytes; // total # bytes for the current xfer
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
    
```

16  
1

## Example

```

NTSTATUS StartDevice(...)
{
    . . . . .
    DEVICE_DESCRIPTION dd;
    RtlZeroMemory(&dd, sizeof(dd));
    dd.Version = DEVICE_DESCRIPTION_VERSION;
    dd.Master = TRUE;
    dd.ScatterGather = TRUE;
    dd.InterfaceType = InterfaceTypeUndefined;
    dd.MaximumLength = MAXTRANSFER;
    dd.Dma32BitAddresses = TRUE;

    pdx->AdapterObject = IoGetDmaAdapter(pdx->Pdo, &dd,
        &pdx->nMaxMapRegisters);
    . . . . .
}
    
```

Suppose that the device is a  
PCI bus master with  
scatter/gather capability

16  
2

## Example

```
VOID StartIo(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
```

Get the IRP Parameters.

Preparescatter/gather list for DMA X'fer.  
Setup **DmaExecutionRoutine** when the channel is available.

1  
3  
}

## Example

```
VOID StartIo(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);

    PMDL mdl = Irp->MdlAddress;
    ULONG nbytes = pdx->nbytes = MmGetMdlByteCount(mdl);
    PVOID vaddr = MmGetMdlVirtualAddress(mdl);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;
```

Preparescatter/gather list for DMA X'fer.  
Setup **DmaExecutionRoutine** when the channel is available.

1  
4  
}

## Example

```
VOID StartIo(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
{
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);

    PMDL mdl = Irp->MdlAddress;
    ULONG nbytes = pdx->nbytes = MmGetMdlByteCount(mdl);
    PVOID vaddr = MmGetMdlVirtualAddress(mdl);
    BOOLEAN isread = stack->MajorFunction == IRP_MJ_READ;

    NTSTATUS status =
        (*pdx->AdapterObject->DmaOperations->GetScatterGatherList)
        (pdx->AdapterObject, fdo, mdl, vaddr, nbytes,
        (PDRIVER_LIST_CONTROL) DmaExecutionRoutine, pdx, isread);

    if (INT_SUCCESS(status)){
        CompleteRequest(Irp, status, 0);
        StartNextPacket(&pdx->Queue, fdo);
    }
}
```

1  
5  
}

## Get Scatter/Gather List

```
NTSTATUS GetScatterGatherList (
    IN PDMA_ADAPTER DmaAdapter,
    IN PDEVICE_OBJECT DeviceObject,
    IN PMDL Mdl,
    IN PVOID CurrentVa,
    IN ULONG Length,
    IN PDRIVER_LIST_CONTROL ExecutionRoutine,
    IN PVOID Context,
    IN BOOLEAN WriteToDevice
);
```

16  
6

## Get Scatter/Gather List

```
NTSTATUS GetScatterGatherList (
    IN PDMA_ADAPTER DmaAdapter,
    IN PDEVICE_OBJECT DeviceObject,
    IN PMDL Mdl,
    IN PVOID CurrentVa,
    IN ULONG Length,
    IN PDRIVER_LIST_CONTROL ExecutionRoutine,
    IN PVOID Context,
    VOID AdapterListControl(
        IN struct _DEVICE_OBJECT *DeviceObject,
        IN struct _IRP *Irp,
        IN PSCATTER_GATHER_LIST ScatterGather,
        IN PVOID Context
    );
```

16  
7

Executes in an arbitrary thread context at  
IRQL **DISPATCH\_LEVEL**.

## DMA Execution Using S/G List

- When the adapter is available, the system calls the **AdapterListControl** routine.
- Its only responsibilities are
  - to save the scatter/gather list pointer (for release).
  - set up its device, and use the scatter/gather list to start DMA.

```
VOID AdapterListControl(
    IN struct _DEVICE_OBJECT *DeviceObject,
    IN struct _IRP *Irp,
    IN PSCATTER_GATHER_LIST ScatterGather,
    IN PVOID Context
);
```

16  
8

## Start DMA Using S/G List

```
VOID DmaExecutionRoutine(PDEVICE_OBJECT fdo, PIRP junk,
    PSCATTER_GATHER_LIST sglst, PDEVICE_EXTENSION pdx)
```

Save **sglist** for releasing when the job done.

Initialize the **state variables** for this X'fer.

Start the **first** stage of a possibly **multi-stage transfer**

16  
9

## Start DMA U

```
typedef struct _DEVICE_EXTENSION {
    . . . . .
    PDMA_ADAPTER AdapterObject;
    ULONG nMaxMapRegisters;
    . . . . .
    PSCATTER_GATHER_LIST sglst;
    ULONG isg;
    ULONG sgdone;
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
VOID DmaExecutionRoutine(
    PSCATTER_GATHER_LIST sglst, PDEVICE_EXTENSION pdx)
```

```
PIRP Irp = GetCurrentIrp(&pdx->Queue);
pdx->sglist = sglst; // save for deallocation in DPC routine
```

Initialize the **state variables** for this X'fer.

Start the **first** stage of a possibly **multi-stage transfer**

17  
0

## Start DMA U

```
typedef struct _DEVICE_EXTENSION {
    . . . . .
    PDMA_ADAPTER AdapterObject;
    ULONG nMaxMapRegisters;
    . . . . .
    PSCATTER_GATHER_LIST sglst;
    ULONG isg;
    ULONG sgdone;
    . . . . .
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

```
VOID DmaExecutionRoutine(
    PSCATTER_GATHER_LIST sglst, PDEVICE_EXTENSION pdx)
```

```
PIRP Irp = GetCurrentIrp(&pdx->Queue);
pdx->sglist = sglst; // save for deallocation in DPC routine

pdx->isg = 0; // index of current scatter gather element
pdx->sgdone = 0; // 0 elements processed so far
```

Start the **first** stage of a possibly **multi-stage transfer**

17  
1

## Start DMA Using S/G List

```
VOID DmaExecutionRoutine(PDEVICE_OBJECT fdo, PIRP junk,
    PSCATTER_GATHER_LIST sglst, PDEVICE_EXTENSION pdx)
```

```
PIRP Irp = GetCurrentIrp(&pdx->Queue);
pdx->sglist = sglst; // save for deallocation in DPC routine

pdx->isg = 0; // index of current scatter gather element
pdx->sgdone = 0; // 0 elements processed so far
```

```
if(!KeSynchronizeExecution(pdx->InterruptObject,
    (PKSYNCHRONIZE_ROUTINE) StartNextTransfer, pdx))
    IoRequestDpc(fdo, NULL, NULL); // nothing to do
```

17  
2

## Start DMA Using S/G List

```
BOOLEAN StartNextTransfer(PDEVICE_EXTENSION pdx)
```

Compute the **sglist elements** to be used for the next DMA X'fer.

If **none**, return **FALSE**.

Use **WRITE\_Xxx HAL** routines to begin transfer.

17  
3

## Start DMA Using S/G List

```
BOOLEAN StartNextTransfer(PDEVICE_EXTENSION pdx)
```

```
{
    PSCATTER_GATHER_LIST sglst = pdx->sglist;
    ULONG i = pdx->isg + pdx->sgdone; // next starting entry
    ULONG n = sglst->NumberOfElements - i; // # left to do
    if (!n) return FALSE; // request is now finished
    n = n > MAXSG ? MAXSG : n; // MAXSG: Hardware dependent
```

Use **WRITE\_Xxx HAL** routines to begin transfer.

17  
4

## Start DMA Using S/G List

```
BOOLEAN StartNextTransfer (PDEVICE_EXTENSION pdx)
{
    PSCATTER_GATHER_LIST sglist = pdx->sglist;
    ULONG i = pdx->isg += pdx->sgdone; // next starting entry
    ULONG n = sglist->NumberOfElements - i; // # left to do
    if (!n) return FALSE; // request is now finished
    n = n > MAXSG ? MAXSG : n; // MAXSG: Hardware dependent

    Use information from sglist->Elements[] to Elements[i+n-1]
    to setup the DMA controller and start the Xfer.

    return TRUE;
}
```

17  
5

## The ISR

```
BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject,
                    PDEVICE_EXTENSION pdx)
{
    Check whether our device raises the IRQ.
    If no, return FALSE

    if (!StartNextTransfer(pdx)){
        Our job is done.
        Schedule a DPC to complete the IRP.
    }
    return TRUE;
}
```

17  
6

## The ISR

```
BOOLEAN OnInterrupt (PKINTERRUPT InterruptObject,
                    PDEVICE_EXTENSION pdx)
{
    Check whether our device raises the IRQ.
    If no, return FALSE

    if (!StartNextTransfer(pdx)){
        PIRP Irp = GetCurrentIrp (&pdx->Queue);
        Irp->IoStatus.Status = STATUS_SUCCESS;
        Irp->IoStatus.Information = pdx->nbytes;
        IoRequestDpc (pdx->DeviceObject, NULL, NULL);
    }
    return TRUE;
}
```

17  
7