MSDN文档

1. 对象

1.1 驱动对象

```
typedef struct _DRIVER_OBJECT {
    // 结构的类型和大小。
    CSHORT Type;
    CSHORT Size;
    // 设备对象，这里实际上是一个设备对象的链表的开始。因为 DeviceObject
    // 中有相关链表信息。读下一小节“设备对象”会得到更多的信息。
    PDEVICE_OBJECT DeviceObject;
    ……
    // 驱动的名字
    UNICODE_STRING DriverName;
    ……
    // 快速 IO分发函数
    PFAST_IO_DISPATCH FastIoDispatch;
    ……
    // 驱动的卸载函数
    PDRIVER_UNLOAD DriverUnload;
    // 普通分发函数
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
    } DRIVER_OBJECT;
```

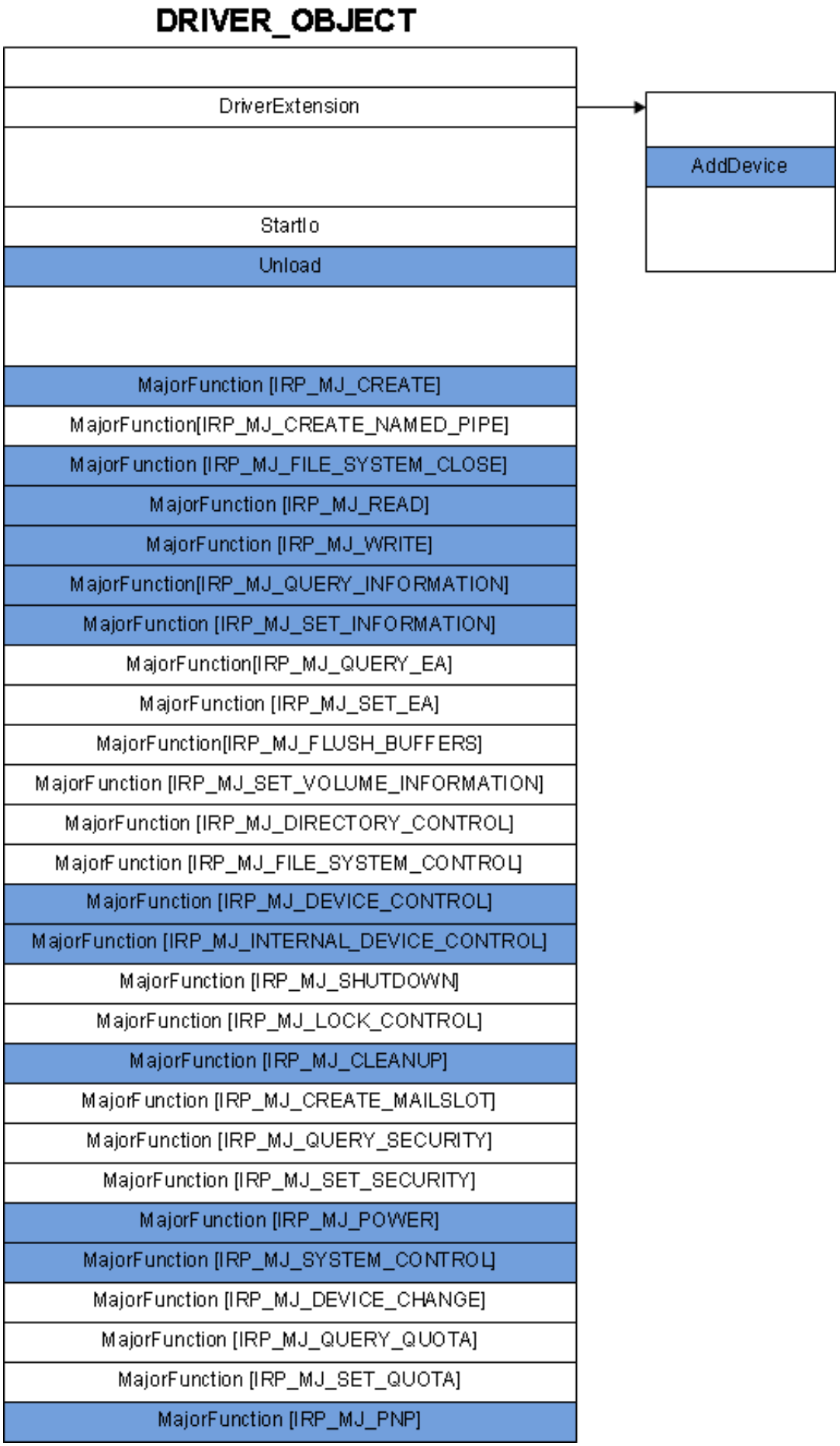
(以下操作为必要部分,但不一定是开发人员需要具体执行,在驱动程序对模型中，这部分工作交给"general".sys完成,参见<5. 微型驱动程序>)

每个内核模式驱动程序都必须实现名为 DriverEntry 的函数，该函数在加载驱动程序之后会立即得到调用。在DriverEntry中为DRIVER_OBJECT设置各种回调(`MajorFunction`)来处理IRP请求。如果没有设置的回调， I/O 管理器提供的默认值。以下示例说明了如何使用 !drvobj 调试器扩展来检查用于 parport 驱动程序的函数指针：

```
0: kd> !drvobj parport 2
Driver object (fffffa80048d9e70) is for:
 \Driver\Parport
DriverEntry:   fffff880065ea070 parport!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff880065e131c parport!PptUnload
AddDevice:     fffff880065d2008 parport!P5AddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                      fffff880065d49d0    parport!PptDispatchCreateOpen
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff80001b6ecd4    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       fffff880065d4a78    parport!PptDispatchClose
```

```
13.  [03] IRP_MJ_READ                    fffff880065d4bac    parport!PptDispatchRea
     d
14.  [04] IRP_MJ_WRITE                   fffff880065d4bac    parport!PptDispatchRea
     d
15.  [05] IRP_MJ_QUERY_INFORMATION       fffff880065d4c40    parport!PptDispatchQue
     ryInformation
16.  [06] IRP_MJ_SET_INFORMATION         fffff880065d4ce4    parport!PptDispatchSet
     Information
17.  [07] IRP_MJ_QUERY_EA                fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
18.  [08] IRP_MJ_SET_EA                  fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
19.  [09] IRP_MJ_FLUSH_BUFFERS           fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
20.  [0a] IRP_MJ_QUERY_VOLUME_INFORMATION fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
21.  [0b] IRP_MJ_SET_VOLUME_INFORMATION  fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
22.  [0c] IRP_MJ_DIRECTORY_CONTROL       fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
23.  [0d] IRP_MJ_FILE_SYSTEM_CONTROL     fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
24.  [0e] IRP_MJ_DEVICE_CONTROL          fffff880065d4be8    parport!PptDispatchDev
     iceControl
25.  [0f] IRP_MJ_INTERNAL_DEVICE_CONTROL fffff880065d4c24    parport!PptDispatchInt
     ernalDeviceControl
26.  [10] IRP_MJ_SHUTDOWN                fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
27.  [11] IRP_MJ_LOCK_CONTROL            fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
28.  [12] IRP_MJ_CLEANUP                 fffff880065d4af4    parport!PptDispatchCle
     anup
29.  [13] IRP_MJ_CREATE_MAILSLOT         fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
30.  [14] IRP_MJ_QUERY_SECURITY          fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
31.  [15] IRP_MJ_SET_SECURITY            fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
32.  [16] IRP_MJ_POWER                   fffff880065d491c    parport!PptDispatchPow
     er
33.  [17] IRP_MJ_SYSTEM_CONTROL          fffff880065d4d4c    parport!PptDispatchSys
     temControl
34.  [18] IRP_MJ_DEVICE_CHANGE           fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
35.  [19] IRP_MJ_QUERY_QUOTA             fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
36.  [1a] IRP_MJ_SET_QUOTA               fffff80001b6ecd4    nt!IopInvalidDeviceReq
     uest
37.  [1b] IRP_MJ_PNP                     fffff880065d4840    parport!PptDispatchPnp
```



在调试器输出中，你可以看到 parport.sys 实现 GsDriverEntry，驱动程序的入口点。GsDriverEntry（在构建驱动程序时自动生成）执行一些初始化，然后调用 DriverEntry（由驱动程序开发者实现）。

还可以看到 parport 驱动程序（位于其 DriverEntry 函数中）设置了如上这些回调，剩余元

素包含指向默认分配函数 nt!IopInvalidDeviceRequest 的指针。

AddDevice 函数很独特，其函数指针未存储在 DRIVER_OBJECT 结构中。而是存储在 DRIVER_OBJECT 结构扩展的 AddDevice 成员中。

值得注意,parport未提供用于 StartIo 的函数指针。

1.2 设备对象

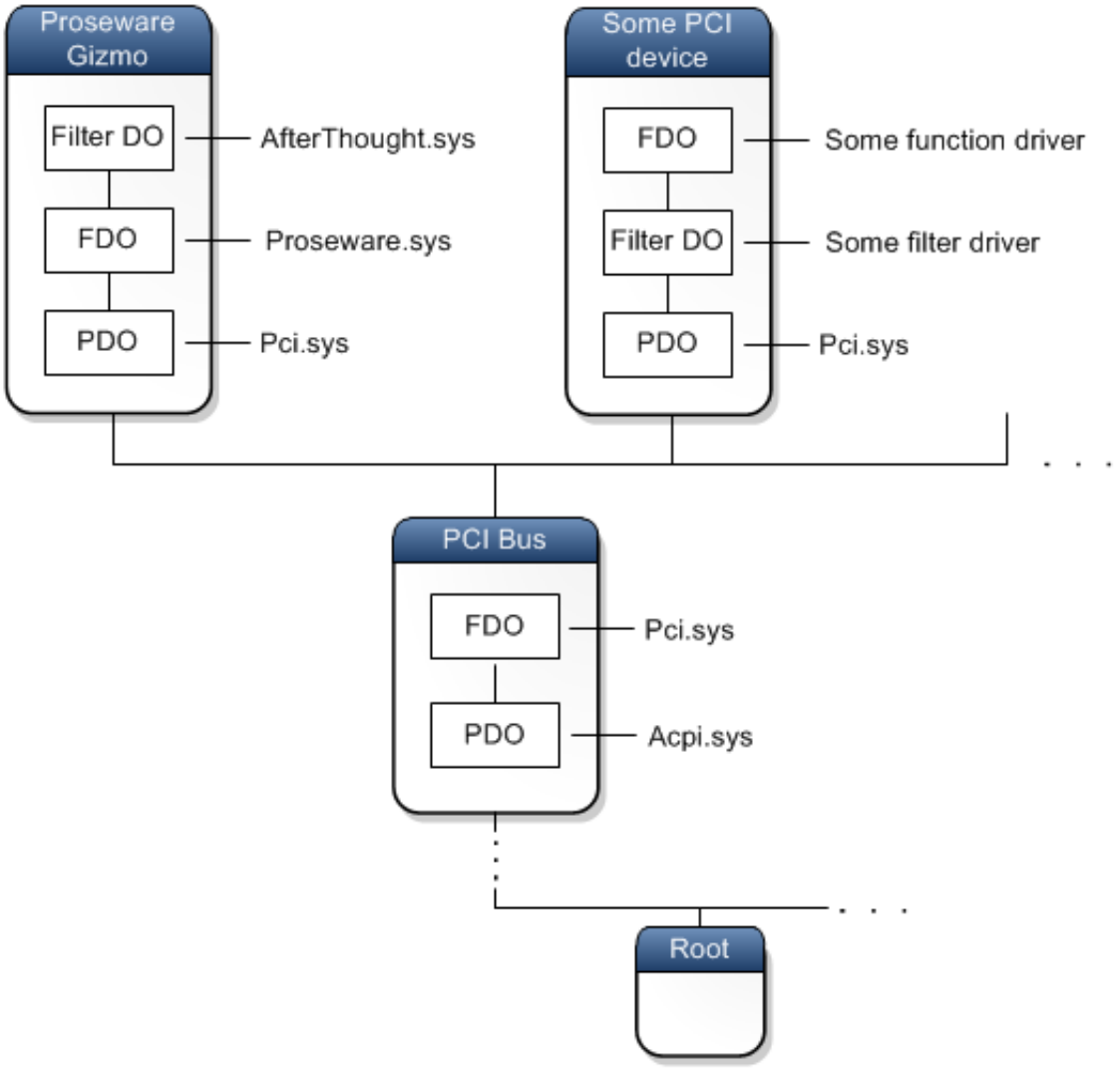
表达上有待商榷:在Windows 窗口应用程序中，窗口是唯一可以接收消息的实体。在内核的世界里，设备对象（DEVICE_OBJECT）是唯一可以接受请求（IRP）的实体。

```
typedef struct DECLSPEC_ALIGN(MEMORY_ALLOCATION_ALIGNMENT) _DEVICE_OBJECT {
    // 和驱动对象一样
    CSHORT Type;
    USHORT Size;
    // 引用计数
    ULONG ReferenceCount;
    // 这个设备所属的驱动对象
    struct _DRIVER_OBJECT *DriverObject;
    // 下一个设备对象。在一个驱动对象中有n 个设备，这些设备用这个指针连接
    // 起来作为一个单向的链表。
    struct _DEVICE_OBJECT *NextDevice;
    // 设备类型
    DEVICE_TYPE DeviceType;
    // IRP栈大小
    HAR StackSize;
    ……
    }DEVICE_OBJECT;
```

2. 设备栈

设备栈最正式的定义:为（设备对象、驱动程序）对的有序列表。但是，在某些上下文中，可以将设备堆栈视为设备对象的有序列表 或者 视为驱动程序的有序列表可能会有用。

在设备堆栈中创建的第一个设备对象位于底部，创建并附加到设备堆栈的最后一个设备对象位于顶部。



2.1 总线驱动程序

在上图中，你可以看到驱动程序 Pci.sys 扮演两个角色:

第一，Pci.sys 与 PCI 总线设备节点中的 FDO 关联。事实上，Pci.sys 已在 PCI 总线设备节点中创建 FDO。因此，Pci.sys 为 PCI 总线的功能驱动程序。

第二，Pci.sys 与 PCI 总线节点的每个子节点中的 PDO 关联。为设备节点创建 PDO 的驱动程序称为该节点的“总线驱动程序”。

如果你的参考点为 PCI 总线，则 Pci.sys 为功能驱动程序。但如果你的参考点为 Proseware Gizmo 设备，则 Pci.sys 为总线驱动程序。此双重角色为 PnP 设备树中的典型。作为总线的功能驱动程序的驱动程序也是总线子设备的总线驱动程序。

2.2 构建设备栈(设备节点)

在启动过程中，PnP 管理器请求每个总线的驱动程序枚举连接到该总线的子设备。例如，PnP 管理器请求 PCI 总线驱动程序 (Pci.sys) 枚举连接到该 PCI 总线的设备。为了响应此请求，Pci.sys 会为连接到 PCI 总线的每个设备创建一个设备对象(PDO)。

PnP 管理器将设备节点与每个新创建的 PDO 关联，并查询注册表以确定哪些驱动程序需要成为该节点设备堆栈的一部分。

2.3 遍历设备栈

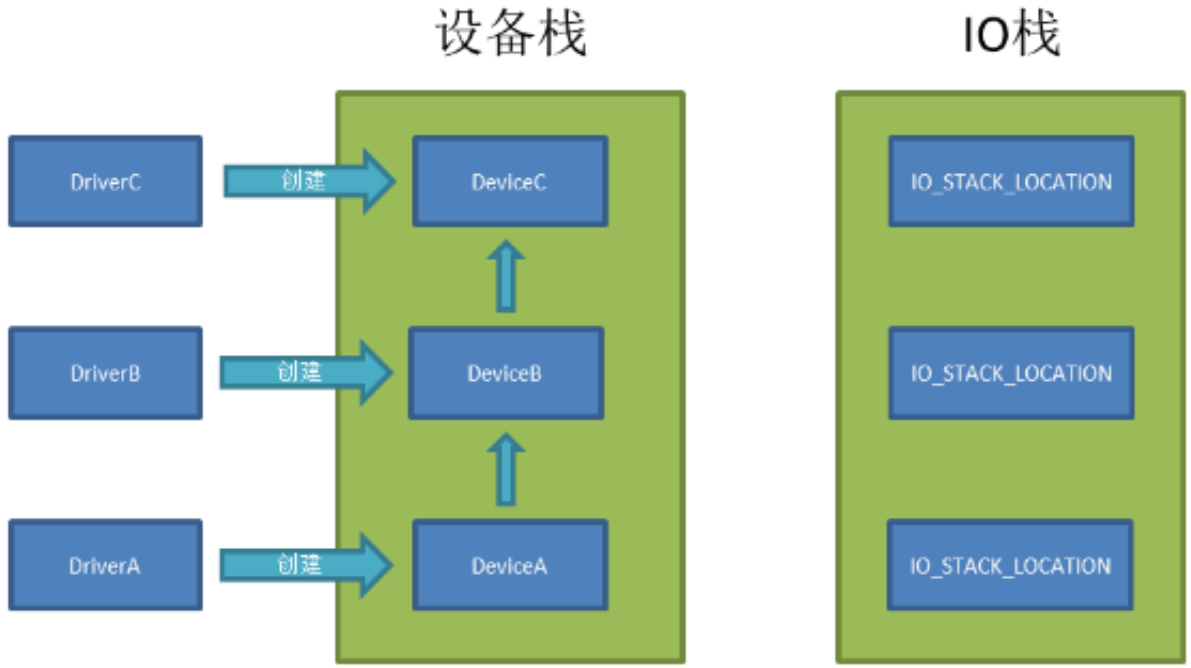
从下往上：DeviceObject::AttachedDevice就保存了当前对象上面的对象。

从上往下：需要设备自己记录 ，当我们调用IoAttachDeviceToDeviceStack的时候会返回下层设备对象，我们可以把这个对象保存到设备扩展里面，这样遍历的时候就可以从设备扩展里面得到下层设备对象。

3. IRP

每个设备对象都对应着一个IO_STACK_LOCATION。IRP内部有个指针指向正在使用的IO_STACK_LOCATION， 可以使用IoGetCurrentIrpStackLocation来获得当前I/O堆栈信息。

比如用户模式的caller打开了DeviceA，并且向这个设备发生一个IRP的过程是：DeviceC -> DeviceB -> DeviceA -> PDO



3.1 设备栈(设备节点)间的发送

IoCallDriver

3.2 设备栈内的传递

IoSkipCurrentIrpStackLocation/IoCopyCurrentIrpStackLocationToNext

3.3 IRP的处理

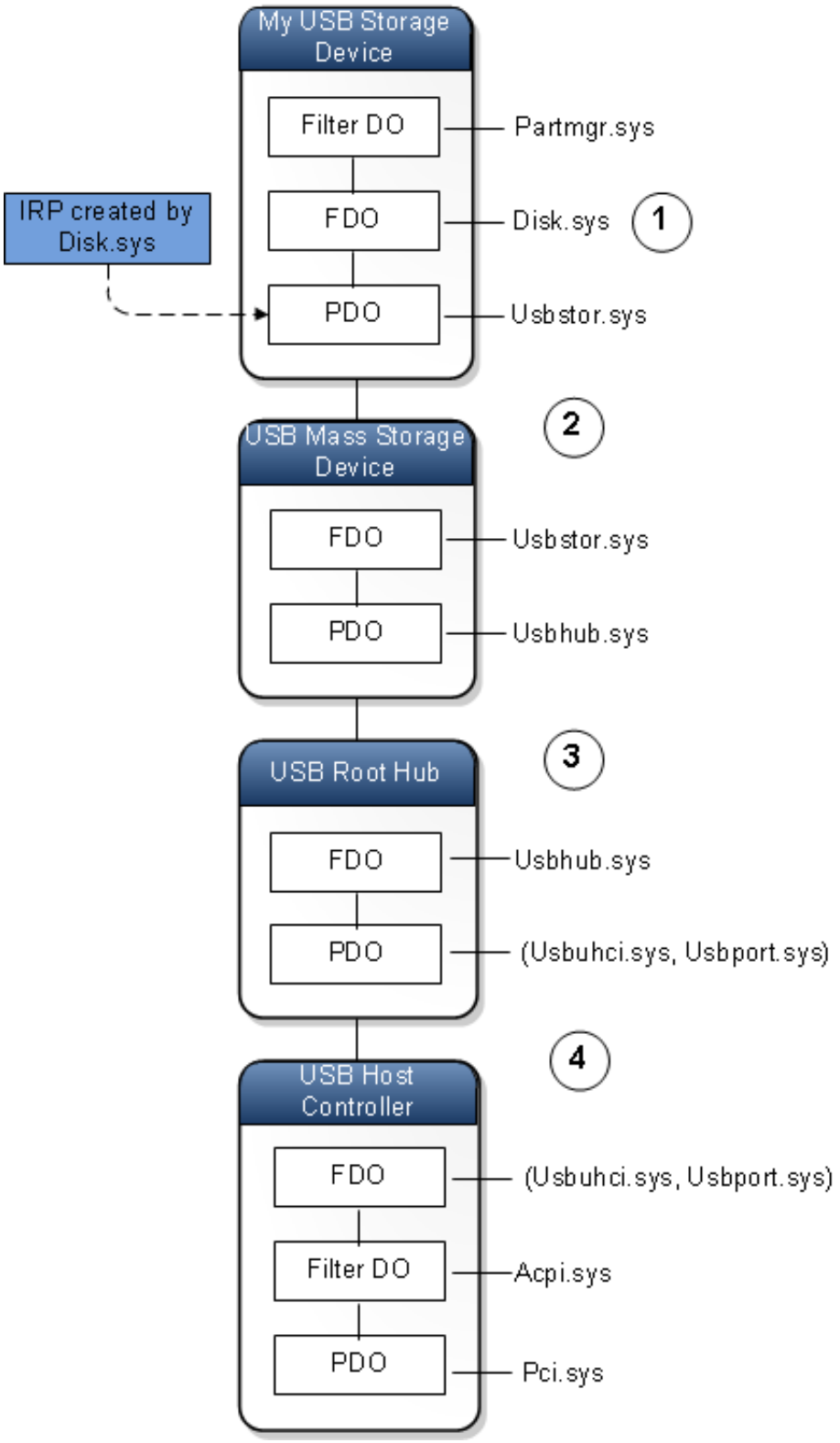
- 直接处理该irp，调用IoCompleteRequest；
- 调用StartIo，让系统将IRP串行化处理；
- 将irp传给下层驱动，让下层驱动来处理该irp，有两种方法:

- 调用IoSkipCurrentIrpStackLocation和IoCallDriver。通常如果当前设备对象不需要处理irp，我们可以这么调用。
- 调用IoCopyCurrentIrpStackLocationToNext和IoCallDriver。通常如果当前设备也需要处理irp，那么我们可以等处理完后，将irp拷贝到下层驱动，然后再调用IoCallDriver。

4. 驱动程序堆栈

若要将读、写或控制请求发送至某个设备，I/O 管理器会查找设备的设备节点，然后将 IRP 发送至该节点的设备堆栈。有时，处理 I/O 请求的过程中会涉及到多个设备堆栈。无论涉及了多少个设备堆栈，参与 I/O 请求的驱动程序整体序列称为请求的“驱动程序堆栈”。

4.1 由多个设备堆栈处理的 I/O 请求

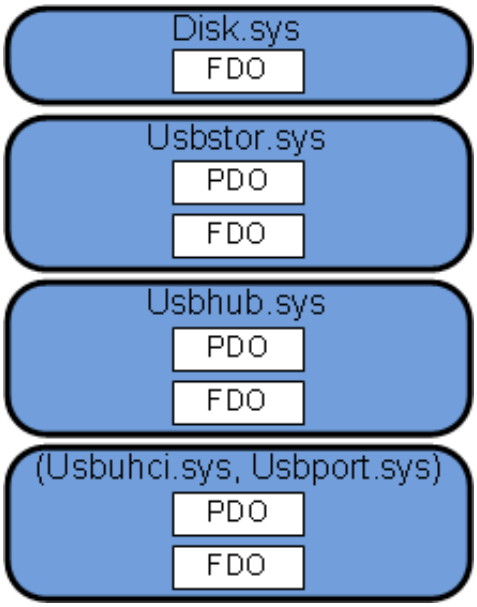


1. IRP 由 Disk.sys 创建，Disk.sys 是“我的 USB 存储设备”节点的设备堆栈中的函数驱动程序。Disk.sys 将 IRP 沿着设备堆栈向下传递到 Usbstor.sys。
2. 注意，Usbstor.sys 为“我的 USB 存储设备”节点的 PDO 驱动程序，为“USB 大容量存储设备”节点的 FDO 驱动程序。IRP 由驱动程序 Usbstor.sys 所有，并且该驱动程序可以访问 PDO 和 FDO。
3. 当 Usbstor.sys 完成处理 IRP 后，它会将 IRP 传递到 Usbhub.sys。Usbhub.sys 为“USB 大容量存储设备”节点的 PDO 驱动程序，为“USB 根集线器”节点的 FDO 驱动程序。IRP 由驱动程序 Usbhub.sys 所有，并且该驱动程序可以访问 PDO 和 FDO。

4. 当 Usbhub.sys 完成处理 IRP 后，它会将 IRP 传递到（Usbuhci.sys、Usbport.sys）对。
5. Usbuhci.sys 为微型端口驱动程序，Usbport.sys 为端口驱动程序。（微型端口、端口）对扮演单个驱动程序的角色。在此情形下，微型端口驱动程序和端口驱动程序都由 Microsoft 编写。（Usbuhci.sys、Usbport.sys）对为“USB 根集线器”节点的 PDO 驱动程序，（Usbuhci.sys、Usbport.sys）对也为“USB 主控制器”节点的 FDO 驱动程序。（Usbuhci.sys、Usbport.sys）对执行与主控制器硬件的实际通信，该硬件反过来与物理 USB 存储设备通信。

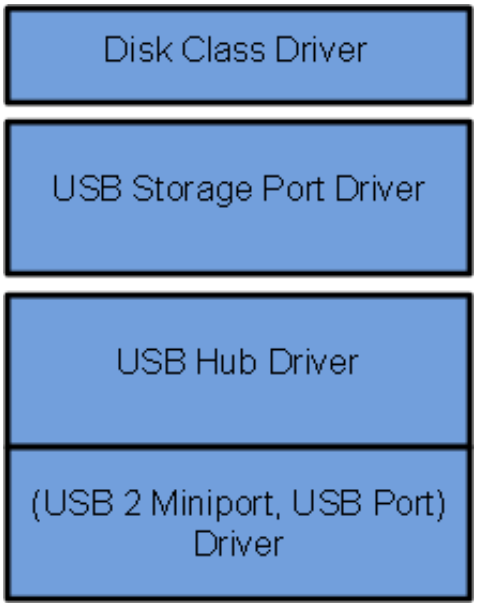
4.2 I/O 请求的驱动程序堆栈

参与 I/O 请求的驱动程序顺序称为 driver stack for the I/O request



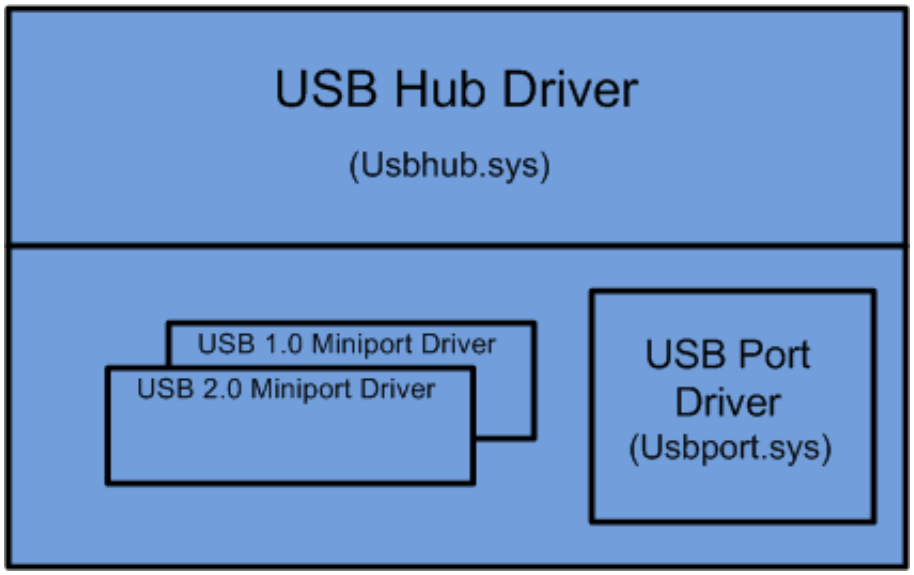
4.3 技术驱动程序堆栈

显示特定技术或操作系统的特定组件或一部分的所有驱动程序的框图称为 技术驱动程序堆栈(technology driver stack)。首先将上图简化为：



在该图中，驱动程序堆栈分为三个部分。我们可以将每个部分都视为属于特定技术或属于操作系统的特定组件或一部分。例如，我们可以说，驱动程序堆栈顶部的第一个部分属于卷管理器，第二个部分属于操作系统的存储组件，第三个部分属于操作系统的核心 USB 部分。

考虑第三个部分中的驱动程序。这些驱动程序为一组较大的核心 USB 驱动程序的子集，Microsoft 提供这些驱动程序用于处理各种 USB 请求和 USB 硬件。下图显示了整个 USB 核心框图的外观。



5. 微型驱动程序

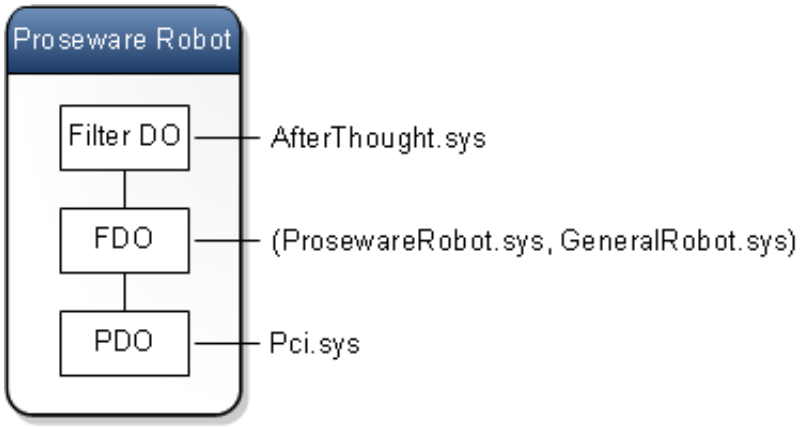
为了简化驱动程序开发,常使用（微型驱动程序/一般驱动程序）/（specific.sys、general.sys）的驱动程序对模型，这个程序对形成单个驱动程序。一般驱动程序处理整个设备集合共同的常规任务，微型驱动程序处理特定于单个设备的任务。微型驱动程序包括：微型端口驱动程序、微型类驱动程序和微型驱动程序。往往Microsoft 提供一般驱动程序，而独立的硬件供应商通常提供特定驱动程序。

在（specific.sys、general.sys）对中，Windows 会加载 specific.sys 并调用其 DriverEntry 函数。specific.sys 的 DriverEntry 函数会收到指向 DRIVER_OBJECT 结构的指针。 specific完成工作后将DRIVER_OBJECT传递给general来完成初始化：填充MajorFunction/Unload/StartIo/AddDevice； 以下代码示例说明了在（ProsewareRobot.sys、GeneralRobot.sys）对中如何调用初始化函数：

```
1.  PVOID g_ProsewareRobottCallbacks[3] = {DeviceControlCallback, PnpCallback, PowerCa
    llback};
2.
3.  // DriverEntry function in ProsewareRobot.sys
4.  NTSTATUS DriverEntry (DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
5.  {
6.     // Call the initialization function implemented by GeneralRobot.sys.
7.     return GeneralRobotInit(DriverObject, RegistryPath, g_ProsewareRobottCallbacks)
    ;
8.  }
```

当 I/O 管理器将 IRP 发送至驱动程序对时，IRP 首先转至由 GeneralRobot.sys 实现的调度函数。如果 GeneralRobot.sys 可以自行处理 IRP，则无须涉及到特定驱动程序 ProsewareRobot.sys。如果 GeneralRobot.sys 可以处理部分但不是全部的 IRP 处理，则它会从由 ProsewareRobot.sys 实现的回调函数之一获取帮助。GeneralRobot.sys 接收指向 GeneralRobotInit 调用中 ProsewareRobot 回调的指针。

在 DriverEntry 返回的某个点，会构造用于 Proseware Robot 设备节点的设备堆栈。设备堆栈可能如下所示。



注意，驱动程序对仅占用设备堆栈中的一层并且仅与一个设备对象关联：FDO。当 GeneralRobot.sys 处理 IRP 时，它可能会调用 ProsewareRobot.sys 获取帮助，但这不同于沿着设备堆栈向下传递请求。驱动程序对形成单个的 WDM 驱动程序，该驱动程序位于设备堆栈中的一层。

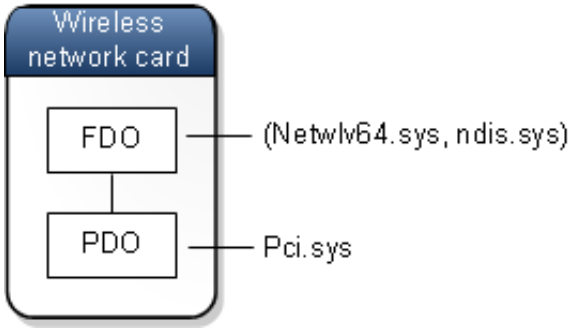
5.1 驱动程序对示例

假设笔记本电脑中有无线网卡，并且通过在“设备管理器”中查找，你确定 netwlv64.sys 为网卡驱动程序。 你可以使用 !drvobj 调试器扩展来检查用于 netwlv64.sys 的函数指针。

```
1: kd> !drvobj netwlv64 2
Driver object (fffffa8002e5f420) is for:
 \Driver\netwlv64
DriverEntry:   fffff8800482f064 netwlv64!GsDriverEntry
DriverStartIo: 00000000
DriverUnload:  fffff8800195c5f4 ndis!ndisMUnloadEx
AddDevice:     fffff88001940d30 ndis!ndisPnPAddDevice
Dispatch routines:
[00] IRP_MJ_CREATE                      fffff880018b5530 ndis!ndisCreateIrpHandler
[01] IRP_MJ_CREATE_NAMED_PIPE           fffff88001936f00 ndis!ndisDummyIrpHandler
[02] IRP_MJ_CLOSE                       fffff880018b5870 ndis!ndisCloseIrpHandler
[03] IRP_MJ_READ                        fffff88001936f00 ndis!ndisDummyIrpHandler
[04] IRP_MJ_WRITE                       fffff88001936f00 ndis!ndisDummyIrpHandler
[05] IRP_MJ_QUERY_INFORMATION           fffff88001936f00 ndis!ndisDummyIrpHandler
[06] IRP_MJ_SET_INFORMATION             fffff88001936f00 ndis!ndisDummyIrpHandler
[07] IRP_MJ_QUERY_EA                    fffff88001936f00 ndis!ndisDummyIrpHandler
[08] IRP_MJ_SET_EA                      fffff88001936f00 ndis!ndisDummyIrpHandler
[09] IRP_MJ_FLUSH_BUFFERS               fffff88001936f00 ndis!ndisDummyIrpHandler
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    fffff88001936f00 ndis!ndisDummyIrpHandler
```

```
[0b] IRP_MJ_SET_VOLUME_INFORMATION      fffff88001936f00 ndis!ndisDummyIrpHandler
[0c] IRP_MJ_DIRECTORY_CONTROL           fffff88001936f00 ndis!ndisDummyIrpHandler
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         fffff88001936f00 ndis!ndisDummyIrpHandler
[0e] IRP_MJ_DEVICE_CONTROL              fffff8800193696c ndis!ndisDeviceControlIrp
Handler
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     fffff880018f9114 ndis!ndisDeviceInternalIr
pDispatch
[10] IRP_MJ_SHUTDOWN                    fffff88001936f00 ndis!ndisDummyIrpHandler
[11] IRP_MJ_LOCK_CONTROL                fffff88001936f00 ndis!ndisDummyIrpHandler
[12] IRP_MJ_CLEANUP                     fffff88001936f00 ndis!ndisDummyIrpHandler
[13] IRP_MJ_CREATE_MAILSLOT             fffff88001936f00 ndis!ndisDummyIrpHandler
[14] IRP_MJ_QUERY_SECURITY              fffff88001936f00 ndis!ndisDummyIrpHandler
[15] IRP_MJ_SET_SECURITY                fffff88001936f00 ndis!ndisDummyIrpHandler
[16] IRP_MJ_POWER                       fffff880018c35e8 ndis!ndisPowerDispatch
[17] IRP_MJ_SYSTEM_CONTROL              fffff880019392c8 ndis!ndisWMIDispatch
[18] IRP_MJ_DEVICE_CHANGE               fffff88001936f00 ndis!ndisDummyIrpHandler
[19] IRP_MJ_QUERY_QUOTA                 fffff88001936f00 ndis!ndisDummyIrpHandler
[1a] IRP_MJ_SET_QUOTA                   fffff88001936f00 ndis!ndisDummyIrpHandler
[1b] IRP_MJ_PNP                         fffff8800193e518 ndis!ndisPnPDispatch
```

在调试器输出中，你可以看到 netwlv64.sys 实现 GsDriverEntry，驱动程序的入口点。GsDriverEntry（在构建驱动程序时自动生成）执行一些初始化，然后调用 DriverEntry（由驱动程序开发者编写）。
在本示例中，netwlv64.sys 实现 DriverEntry，但 ndis.sys 实现 AddDevice、Unload 以及多个调度函数。Netwlv64.sys 称为 NDIS 微型端口驱动程序，ndis.sys 称为 NDIS 库。两个模块共同形成（NDIS 微型端口、NDIS 库）对。



5.2 可用驱动程序对

以下是可用的一些（特定、通用）对：

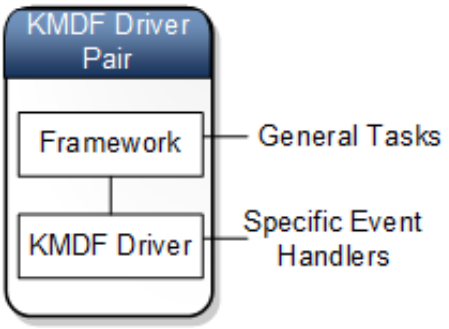
- （屏幕微型端口驱动程序、屏幕端口驱动程序）
- （音频微型端口驱动程序、音频端口驱动程序）
- （存储微型端口驱动程序、存储端口驱动程序）
- （电池微型类驱动程序、电池类驱动程序）
- （HID 微型驱动程序、HID 类驱动程序）
- （转换器微型类驱动程序、转换器端口驱动程序）
- （NDIS 微型端口驱动程序、NDIS 库）

注意：类表中“类驱动程序”不同于独立类驱动程序，也不同于类筛选器驱动程序。

6. KMDF框架

6.1 二重模型

KMDF下，一个驱动程序也遵循驱动程序对模型,如下图：



当某人将 I/O 请求数据包 (IRP) 发送到（KMDF 驱动程序、框架）对时，IRP 会转到框架。如果框架可以自行处理 IRP，则不会涉及到 KMDF 驱动程序。如果框架自身无法处理 IRP，则它会通过调用 KMDF 驱动程序实现的事件处理程序来获取帮助。以下是可以由 KMDF 驱动程序实现的事件处理程序的一些示例。

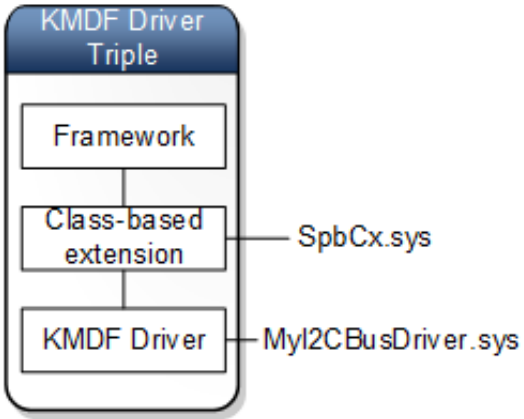
- EvtDevicePrepareHardware
- EvtIoRead
- EvtIoDeviceControl
- EvtInterruptIsr
- EvtInterruptDpc
- EvtDevicePnpStateChange

6.2 三重模型

使用基于类的 KMDF 扩展的驱动程序可以进一步降低必须由 KMDF 驱动程序执行处理的量。三重驱动程序（KMDF 驱动程序、设备类 KMDF 扩展、框架）中的三个驱动程序合并形成单个 WDM 驱动程序。

- 框架，处理大部分驱动程序共同的任务
- 基于类的框架扩展，处理特定于特殊类的设备的任务
- KMDF 驱动程序，处理特定于特殊设备的任务

举例说明：



三重驱动程序中的三个驱动程序（MyI2CBusDriver.sys、SpbCx.sys、

Wdf01000.sys）合并形成单个 WDM 驱动程序，该驱动程序用作 I2C 总线控制器的函数驱动程序。Wdf01000.sys（框架）拥有此驱动程序的计划表，因此当某人将 IRP 发送至三重驱动程序时，它会转至 wdf01000.sys。如果 wdf01000.sys 自身可以处理 IRP，则不会涉及到 SpbCx.sys 和 MyI2CBusDriver.sys。无法 wdf01000.sys 自身无法处理 IRP，则它可以通过调用 SbpCx.sys 中的事件处理程序来获取帮助。

MyI2CBusDriver.sys 实现的事件处理程序：

- EvtSpbControllerLock
- EvtSpbIoRead
- EvtSpbIoSequence

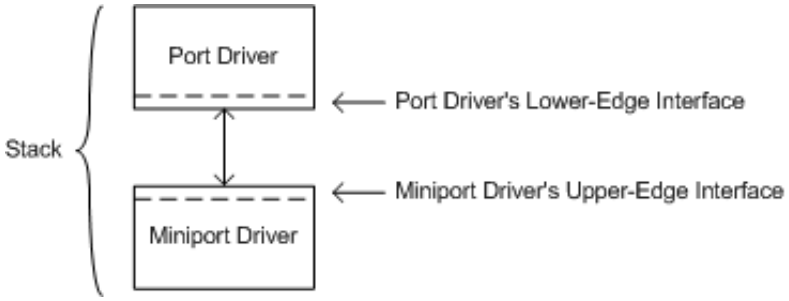
SpbCx.sys 实现的事件处理程序:

- EvtIoRead

7. 驱动程序的上沿和下沿

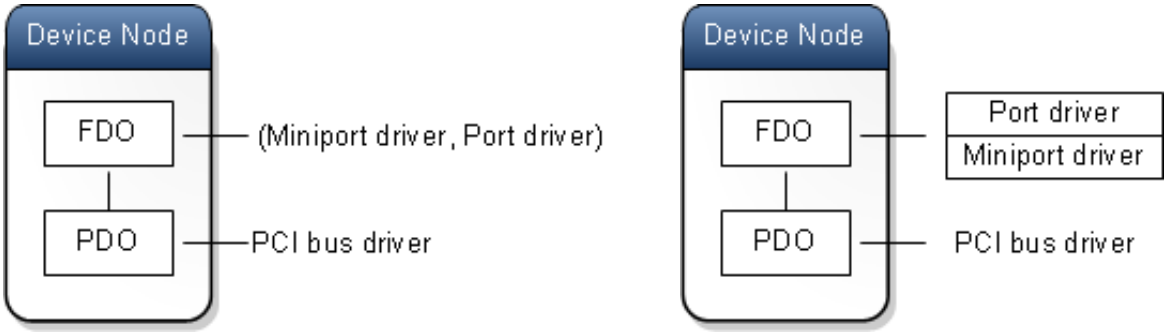
参与 I/O 请求的驱动程序顺序称为请求的“驱动程序堆栈”。驱动程序可以调用堆栈中下层驱动程序的上沿。驱动程序也可以调用堆栈中上层驱动程序的下沿。

当驱动程序实现一组高层驱动程序可以调用的函数时，该组函数称为驱动程序的上沿，或者驱动程序的上沿接口。反之为下沿。



上图认为端口驱动程序位于驱动程序堆栈中迷你端口驱动程序的上面有时是有用的。因为 I/O 请求首先由端口驱动程序处理，然后再由迷你端口驱动程序处理。然而，请记住（迷你端口，端口）驱动程序对通常位于设备堆栈中的单层. **请注意，驱动程序堆栈不同于设备堆栈。在驱动程序堆栈中垂直显示的两个驱动程序可能形成位于设备堆栈中单层上的驱动程序对。某些驱动程序不是 PnP 设备树的一部分。**

所以根据理解的角度不同可以有两种画法:

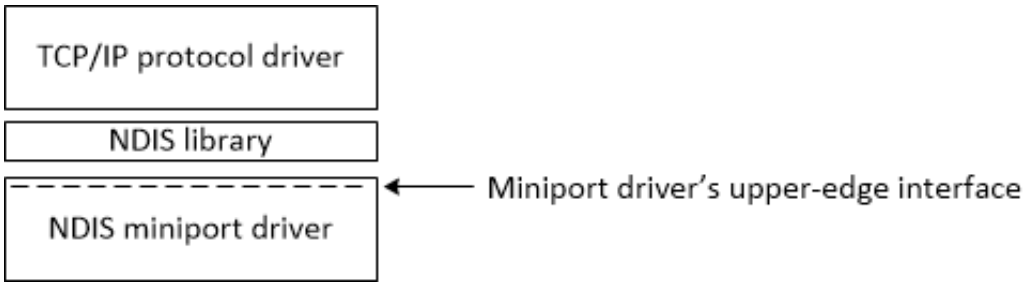


关键点在于，当端口驱动程序调用迷你端口驱动程序的上沿接口时，这不同于将 I/O 请求向下传递到设备堆栈。在驱动程序堆栈（不是设备堆栈）中，可以选择在迷你端口驱动程序的上面绘制端口驱动程序，但这并非意味着该端口驱动程序位于设备堆栈中迷你驱动程序的上面。

NDIS 示例

有时，驱动程序直接调用低层驱动程序的上沿。例如，假设 TCP/IP 协议驱动程序位于驱动程序堆栈的 NDIS 迷你端口驱动程序的上面。该迷你驱动程序将实现一组 MiniportXxx 函数，它们形成迷你端口驱动程序的上沿。我们称之为 TCP/IP 协议驱动程序绑定到 NDIS 迷你端口驱动程序的上沿。但 TCP/IP 驱动程序不直接调用 MiniportXxx 函数。而是调用 NDIS 库中的函数，然后这些函数再调用 MiniportXxx 函数。

驱动程序栈试图:



设备栈试图：