第一卷 驱动程序编写者指南

第1章　驱动程序开发环境

为 Microsoft® Windows® 2000 开发驱动程序至少需要两台机器：一台用于开发，一台用于调试。

如果驱动程序依赖于基本设备，包括高端工作站和服务器，驱动程序也可在一台多处理器计算机上调试和测试。

本章讨论如下主题：

- 1.1 自由构建（Free Build）和检查构建（Checked Build）
- 1.2 调试环境

1.1　自由构建和检查构建

微软的 Windows2000 驱动程序的测试和调试需要 Windows® 2000 的自由构建和检查构建。

- 自由构建（或零星构建）是操作系统的终端用户版本，系统和驱动程序以最优化方式构建，不可用调试断言，且调试信息已从二进制码中去除。自由系统和驱动程序更小更快，对内存的需求更小。
- 检查构建用于操作系统和内核模式驱动程序的测试和调试，检查构建包括意外错误检查、参数检查和在自由构建中不可用的调试信息。一个检查的系统或驱动程序能帮助区分和记录驱动程序的某些问题，如内存泄漏或不当的设备配置，这些问题将导致不可预测的后果。

  尽管检查构建提供额外的保护，但它比自由构建消耗更多的内存和硬盘空间，且由于下列原因，系统和驱动程序运行得更慢。
- 可执行程序含有符号调试信息。
- 由于参数检查和调试输出，执行了附加的代码（诊断信息）。

新的驱动程序开发通常包括以下步骤：

1. 编写驱动程序代码，应包括条件编译标记的调试检查。
2. 测试和调试基于操作系统的检查构建的驱动程序的检查构建。
3. 测试和调试基于自由构建的驱动程序的自由构建。
4. 基于自由构建的驱动程序的调整。
5. 使用了检查构建和自由构建的驱动程序和操作系统附加的测试和调试。
6. 使用了自由构建的最终的测试和检查。

在驱动程序开发的早期，需要 Windows2000 检查构建来调试驱动程序，检查构建的附加调试代码保护了驱动程序可能导致的许多错误（比如复发的自旋锁）。

执行调整、最终测试和检查驱动程序应该基于自由构建而完成，自由构建的速度越快越有可能发现竞争条件和其他同步性的问题。

由于自由构建与 Windows2000 的零星版本相同，最终测试和检查也应该基于自由构建而完成。

驱动程序代码通常包括预处理器符号，该符号允许被编译成自由和检查构建。

DBG 标记是一个保留的符号，可用它来决定编译时 Windows2000 的什么构建在运行，如果

Windows2000 检查构建运行，设置 DBG 为 1，如果自由构建运行，不定义 DBG（或设置为 0，如果或者包括头文件 wdm.h ，或者包括 ntddk.h）。

驱动程序也应该在至少一个多处理器平台和至少一个单处理器平台上测试和调试；且这两个平台应该运行 Windows2000 的当前版本。

1.2 调试环境

内核模式调试需要一个目标机和一个主机，目标机用来运行驱动程序或另一内核模式的应用程序，主机运行调试程序。

图 1.1 显示使用调试驱动程序的典型 Windows2000 设置。内核调试不需要自由或检查构建的特别结合，从一自由或检查系统调试一自由系统或检查系统是可行的。然而，总体上说，主机没有理由运行较慢的检查构建。

图 1.1 典型的 Windows2000 调试设置

参看在线 DDK 的 Debugging Drivers 和微软的调试程序使用文献来获得更多细节。

第2章　检查驱动程序

在 Windows2000 上，Driver Verifier 是新的可用工具，它执行几项测试和调试内核模式驱动程序的任务。

2.1 Driver Verifier

Driver Verifier 是一可帮助监视一个或多个内核模式驱动程序以证实他们没有非法函数调用或引起系统讹误的工具，Driver Verifier 在目标驱动程序上执行广泛的测试和检查任务。例如，如果驱动程序以非正当的 IRQL 使用了内存，不正当地调用或释放自旋锁和内存分配，或者释放内存池时没有首先删除任何定时器，Driver Verifier 将发布合适的错误检查。

当未装载驱动程序时，Driver Verifier 检查确信驱动程序已经正确地清理了队列、线程和其他项目。

此外，Driver Verifier 能够执行以下任何情况：

- 从一个特别内存池分配驱动程序内存请求，该过程测试驱动程序是否访问它的内存分配之外的内存，或者在释放它的分配之后访问内存。
- 通过使内存分页代码无效而给驱动程序极端的内存压力，这个过程揭示了访问分页内存的企图，分页的内存发生在当错误的 IRQL 或当保留一自旋锁时。
- 池分配的随机失效或其他的内存请求，该过程测试了驱动程序处理低内存状况的能力。
- 未装载的驱动程序检查所有的内存分配，以确信驱动程序没有漏掉内存。
- 从一特别池分配驱动程序的 IRP，监视驱动程序 I/O 以处理其他不一致的行为。

这些能力可以分别激活或禁止，此外，Driver Verifier 可同时用于任何数目的驱动程序。

下面的部分解释了驱动程序的工作方面：

- 2.1.1 Driver Verifier 的能力

详细描述了 Driver Verifier 各个作用,它应用于除过图形驱动程序之外的所有内核模式驱动程序。

- 2.1.2 图形驱动程序的 Driver Verifier 的能力

描述了 Driver Verifier 对内核模式图形驱动程序的作用，这里内核模式图形驱动程序使用了图形驱动程序接口（GDI）。

- 2.1.3 激活和监视 Driver Verifier

解释了怎样启动 Driver Verifier，怎样选择所要的功能，怎样选择所要检查的驱动程序，也解释了如何利用 Driver Verifier Manager 来监视正被检查的驱动程序的行为。

注意：Driver Verifier 能够检查任何数目的驱动程序，然而，当特别内存池和 I/O 检查选项同时运用于一驱动程序时，其效果将更为有效。

2.1.1 Driver Verifier 的能力

Driver Verifier 有两种能力：一种是一直起作用，另一种只有被选择时才起作用。

以下是对 Driver Verifier 所有能力的描述：

- 2.1.1.1 自动检查

这些是一直起作用的功能，比如 IRQL 和内存例程的检查，检查定时器检查释放的内存池，检查驱动程序的正确卸载。

- 2.1.1.2 特别内存池

这个功能使用了一个特别池来测试内存的上溢和下溢，及在内存释放之后的访问。

- 2.1.1.3 强迫 IRQL 检查

这个功能给驱动程序极端的内存压力来揭示内存分页的故障。

- 2.1.1.4 低资源模拟

这个功能注入随机分配错误和其他被拒决的请求来测试驱动程序在低内存状况下的响应。

- 2.1.1.5 内存池跟踪

这个功能检查当驱动程序被卸载时所有内存分配已经释放。

- 2.1.1.6 I/O 检查

这个功能监视驱动程序 I/O 对非法或不一致的行为的处理。

2.1.1.1 自动检查

不论 Driver Verifier 检查一个或多个驱动程序，它将执行以下功能，这些功能不受任何 Driver Verifier 选项的允许和禁止之影响。

IRQL 和内存例程的检查

Driver Verifier 为下面被禁止的功能而监视所选择的驱动程序。

- 通过调用 KeLowerIrql 提高 IRQL
- 通过调用 KeRaiseIrql 降低 IRQL
- 零内存分配请求
- 在 APC_LEVER 之上的 IRQL 分配和释放分页池
- 在 DIPATCH_LEVER 之上的 IRQL 分配和释放非分页池
- 尝试释放一个没有从前面的分配里返回的地址。
- 尝试释放一个已被释放的地址。
- 在 APC_LEVER 之上的在 IRQL 获取和释放一个快速的互斥体
- 在 IRQL 而非在 DIPATCH_LEVER 之上获取和释放一个自旋锁
- 双倍释放一个自旋锁
- 指定一非法或随机（未初始化的）参数给任一 API。

如果 Driver Verifier 没有运行，在所有状况下，这些故障不大可能会引起立即的系统崩溃，如果以上任何故障发生，则 Driver Verifier 监视驱动程序的行为并发布错误检测 0xC4。（参看微软调试程序文档的使用来获得错误检查参数。）

检查被释放的内存池定时器

Driver Verifier 检查所有被检查驱动程序所释放的内存池，如果任何定时器保留在该池里，发布错误检测 0xC7。（遗忘的定时器能导致最终系统崩溃，这是最难考虑到的。）

检查驱动程序的卸载

当一个正被检查的驱动程序卸载后，Driver Verifier 执行几个检查来确信驱动程序已被清空。特别地，Driver Verifier 寻找下面部分：

- 未删除的定时器
- 未定的 DPC
- 未删除的辅助列表
- 未删除的工作线程
- 未删除的队列
- 其他类似的资源

诸如以上的问题能潜在地引起系统错误检查在驱动程序卸载后被发布，引起这些错误检测的原因难于判断。当 Driver Verifier 运行时，这种故障将导致错误检测 0xC7 在驱动程序卸载之后立即发布。（参看微软的调试程序文件的使用来获得错误检测参数的列表。）

图形驱动程序

当检查一个驱动程序时，这些自动检查没有执行。

2.1.1.2 特别内存池（Special Memory Pool）

内存讹误是一个普通的驱动程序问题,驱动程序错误能在错误出现的长时间后导致崩溃。这些错误中最普通要数访问已被释放的内存，并分配 n 字节然后是 n+1 字节。

为发现内存讹误，Driver Verifier 能够从一特别池分配驱动程序内存并监视该池防止不正确的访问。

两个特别池定位是可行的，Verify End 定位更易于发现访问上溢，Verify Start 分配更易发现下溢。（注意：主要的内存讹误是由于上溢，而非下溢。）

当特别内存池运行并且选择了 Verify End 时，驱动程序请求的每一个内存分配被放置到不同的内存页码上。允许分配适合分页的最高可能地址被返回，这样内存被定位于页末。分页的前面部分用特殊的模式写入，前面的页码和紧邻的页码被标记为不可访问的。

如果驱动程序在分配结束之后试图访问内存，Driver Verifier 将立即发现并发布错误检测 0xCD，如果驱动程序在缓冲区的开始之前写内存，这将（大概）更改这种模式。当缓冲区被释放，Driver Verifier 将发现更改并报告错误检测 0xC1。

如果驱动程序在缓冲区释放以后读出或写入，Driver Verifier 将报告错误检测 0xCC。

当选择 Verify Start 时，内存缓冲区定位于分页的开始，在这种设置下，下溢立即引起错误检测，而上溢只有当内存被释放时才引起错误检测。这个选项的其他方面与 Verify End 选项是相同的。

由于驱动程序上溢错误比下溢错误普遍的多，所以 Verify End 是默认的定位。为改变这种设置，使用全局标记应用程序（Global Flags Utility）。

一个单独的内存分配推翻这种设置，并通过调用 ExAllocateWithTagPriority 的 Priority 参数来设置成 XxxSpecialPoolOverrun 或 XxxSpecialPoolUnderrun 来选择它的定位。（这个例程不能够激活或去活特别池，或者请求一个特别的内存分配，否则的话，这个内存分配将从普通池得到，只有此种定位才能够被这个例程控制。）

池标记

Driver Verifier 将给已选择要检查的驱动程序指定特别池，另外一种使用特别池的方法是指定它到被一个专用 *pool tag* 标记的内存池。

Global Flags Utility 能够用来将特别池给具有一给定标记的池。

同时通过 Driver Verifier 和 Global Flags Utility 来请求特别池是允许的，如果这么做，微软的 Windows2000 将试图利用特别池给指定标记的所有的池和来自指定驱动程序所有的池分配请求。

特别池效率

每个来自特别池的分配使用一个不可分页的内存页码和两个具有虚拟地址空间的页码，如果此池被耗尽，内存通过标准的方式分配，直到特别池再次变得可用为止。这样，如果特别内存池（Special Memory Pool）正在使用，不推荐同时检查多驱动程序。

发出了大量小内存请求的单个驱动程序也能够耗尽此池，出现这种情况，给驱动程序内存分配指定池标记且一次使特别池给一个池标记是可以选择的。

特别池的大小随着系统物理内存的增大而增加，理想地，其容量至少 1 GB。在 x86 机器上，当虚拟空间（还有物理空间）被消耗时，引导而没有/3 GB 开关也是可选的。增加分页文件达最小/最大数量（通过二，三当中的一个因子）也是一个好主意。

如果特别内存池可用，但是不到 95%的所有池分配已经从特别池指定，在驱动程序测试管理器（Driver Verifier Manager）的 *Driver Status screen* 上将出现一个警告。发生这种情况，你应该检查一个更短的驱动程序列表，通过池标记检查单个池，或者给你的系统增加更多的物理内存。

为确信所有的驱动程序分配已被测试，推荐加强驱动程序较长时间周期。

监视特别池

Driver Verifier Manager 的 *Global Counters screen* 能被用来监视特别池的使用。

如果是 *Allocations Succeeded* ，特别池计数器等同于 *Allocations Succeeded* 计数器，于是特别池足够覆盖所有的内存分配。如果 *Allocations Succeeded*：特别池少于 *Allocations Succeeded* ，则特别池至少已经被耗尽过一次。

由于特别池没运用于这些计数器，所以计数器不跟踪大小为一页或更多的分配。

内核调试程序扩展 !verifier 也能够运用于监视特别池使用，它展示了与 Driver Verifier Manager 相似的信息。欲获取关于调试程序扩展的信息，请参看微软调试程序使用文件。

图形驱动程序

为获取这种选项怎样作用显示于驱动程序和内核模式打印驱动程序，参看图形驱动程序的特别内存池。

2.1.1.3 强迫 IRQL 检查（Forcing IRQL Checking）

尽管内核模式驱动程序被禁止在高 IRQL 或保持一种自旋锁时访问分页内存，但如果分页实际上没被修剪，这种动作将不会被注意到。

当 *Forcing IRQL Checking* 可用时，Driver Verifier 将给所选择的驱动程序施加极端的内存压力。不论何时 IRQL 被抬伸到 DISPATCH_LEVEL 或更高，或当一自旋锁被请求时，所有的驱动程序的可分页代码和数据（和系统可分页的池、代码、数据）被标记为修剪过。如果驱动程序试图访问任何一个这种内存，Driver Verifier 发布一个错误检测。

既然别的驱动程序的 IRQL 抬伸不会引起这个动作，这个内存压力将不会直接作用于未被选择检查的驱动程序。然而，当一个正在检查的驱动程序抬伸 IRQL 时，Driver Verifier 修剪分页，该分页能在未被检查的驱动程序所使用。当这个选项运行时，未被检查的驱动程序的这种错误可能偶然被捕获。

图形驱动程序

Forcing IRQL Checking 选项不用于图形驱动程序，如被选中，将不起作用。

2.1.1.4 低资源模拟（Low Resources Simulation）

当 *Low Resources Simulation* 可用时，Driver Verifier 将引起驱动程序内存分配的一个随机选择失效，这个过程测试驱动程序对低内存和其他低资源状况下正常反应的能力。

为精确模拟一低内存条件，这些分配故障直到系统启动后的 7 分钟被注入，因此，在该过程中暴露的任何驱动程序错误将以合法的运行问题对待，而非不切实际的情况。

标记为 MUST_SUCCEED 的分配请求不服从于这一动作， MUST_SUCCEED 池的每页最大值被禁止。

Driver Verifier 能同时检查所选择的驱动程序或所有的驱动程序。

图形驱动程序

参看图形驱动程序的 *Low Resources Simulation* 来获得该选项如何作用于显示驱动程序和内核模式驱动程序的细节。

2.1.1.5 内存池跟踪（Memory Pool Tracking）

Memory Pool Tracking 监视驱动程序所做的内存分配，当驱动程序未装载时，Driver Verifier 确保驱动程序所决定的任何分配都已经释放。

不能释放的内存分配（也叫内存泄露）是引起低操作系统执行的通常原因，这些能引起系统池破碎，最终导致系统崩溃。

当这一选项运行时，如一驱动程序没有释放其所有的分配就卸载，Driver Verifier 将发布错误检测 0xC4（参数 1 等于 0x60）。

如果 Driver Verifier 发布错误检测的参数 1 等于 0x51，0x52，0x53，0x54 或 0x59，则该驱动程序已经写入分配之外的内存里，在这种状况下，你应该能够让特别内存池来定位错源。

参看微软调试程序文件的使用来获得所有错误检测参数 0xC4 的列表。

监视内存池跟踪

Driver Verifier Manager 的 *Pool Tracking screen* 能够用来监视有分页和无分页的池分配。

内核调试程序扩展 ! verifer2 能被用于驱动程序卸载之后未决的内存分配，或当驱动程序运行时跟踪当前的内存分配。这个扩展也表明了池标记，池大小和每一个分配的分配器地址。为更多的调试程序信息，请参看微软的调试程序使用文件。

图形驱动程序

Memory Pool Tracking 选项不适用于图形驱动程序，如被选中，不起作用。

2.1.1.6 I/O 检查

Driver Verifier 有两个 I/O 检查构建，1 级 I/O 检查从一特别池分配驱动程序的 IRP 和监视驱动程序的 I/O 对各种不当的动作的处理，2 级 I/O 检查执行所有 1 级的动作，和许多更细更广驱动程序 I/O 的使用。

2 级 I/O 检查是一更有力的检测驱动程序 I/O 的使用的方法，然而，这种高级的详细审查占用了更多的内存，且它也能降低操作系统的执行级别。

1 级 I/O 检查

当 1 级 I/O 检查可用时，通过 **IoAllocateIrp** 获得的所有 IRP 从一特别池分配，且它们的使用受到跟踪。

此外，Driver Verifier 检查非法的 I/O 调用，包括：

- 尝试释放一个非 IO_TYPE_IRP 类型的 IRP
- 传递非法设备对象给 **IoCallDriver**
- 传递一 IRP 给含有非法的状态或仍保留一已取消的例程集的 **IoCompleteRequest**
- 通过驱动程序调度例程的调用来改变 IRQL
- 尝试释放保留关联的一个线程的 IRP
- 传递一个设备对象给已经有初始化过的定时器的 **IoInitializeTimer**
- 传递一个非法的缓冲区给 **IoBuildAsynchronousFsdRequest** 或 **IoBuildDeviceIoControlRequest**
- 当一个 I/O 状态块分配给一极松散的堆栈时，传递一个 I/O 状态块给一个 IRP
- 当一个事件对象分配给一极松散的堆栈时，传递一个事件对象给一个 IRP

由于特别 IRP 池有大小限制，只有当一次使用于一个驱动程序时，I/O 检查才最有效。

1 级 I/O 检查失效将引起错误检测 0xC9 发布，错误检测的第一个参数表明有什么违背发生。欲得一个全部的错误检测 0xC9 参数的列表，参看微软的调试程序文件。

2 级 I/O 检查

2 级 I/O 检查错误以不同的方式显示：在一个蓝屏上、在一个崩溃的转储文件里和在一个内核调试程序里。

在一个蓝屏上，信息 **IO SYSTEM VERIFICATION ERROR** 和信息串 **WDM DRIVER ERROR** *XXX* 记下了这些错误，这里 *XXX* 是一个 I/O 的错误代码。

在一个崩溃的转储文件里，信息 **BugCheck 0xC9（DRIVER_VERIFIER_IOMANAGER_VIOLATION）**与 I/O 错误代码一道，记下了这些错误。在此情况下，I/O 错误代码以错误检测 0xC9 的第一个参数出现。

一个内核调试程序里（KD 或 WinDbg）这些错误以信息 **WDM DRIVER ERROR** 和一描述的文本信息串出现，当内核调试程序运行时，忽略 2 级错误和恢复系统操作是可能的。（不可能出现别的错误检测。）

在以上的每一情况下，额外信息（例如驱动程序名和各种可用的指针）也被显示，欲知 2 级 I/O 检查错误信息的全面描述，参看微软的调试程序文件中关于错误检测 0xC9 的部分。

图形驱动程序

I/O 检查选项不适用于图形驱动程序，如被选中，不起作用。

2.1.2 Driver Verifier 对图形驱动程序的能力

微软的 Windows2000 内核模式图形驱动程序不直接分配内存池，相反，这些驱动程序使用 GDI（图形驱动程序接口）服务例程，该例程由 *win32k.sys* 提供。

由于这个差异，Driver Verifier 对待图形驱动程序与内核模式驱动程序不同。

下面的部分描述了 Driver Verifier 在显示驱动程序和内核模式打印驱动程序上的作用。

- 2.1.2.1 图形驱动程序的特别内存池

该动作使用一特别池来检测内存的上溢和下溢，还有在其释放后访问内存。

- 2.1.2.2 图形驱动程序的低资源模拟

该动作注入随机的分配故障和其他被拒绝的请求来检测驱动程序在一个低内存状况下的反

应。

图形驱动程序的不可用选项

当在检查一个图形驱动程序时，Driver Verifier 经常执行的自动检查（IRQL 和内存例程的检查，检查定时器释放的内存池，检查驱动程序卸载）并不执行。

强迫 IRQL 检测、内存池跟踪和 I/O 检查选项不被用于图形驱动程序，如被选择，没有作用。

注意：Driver Verifier 能被设置检查 *win32k.sys* 自身。然而，这对同时检查的所有图形驱动程序都有影响。为获得更多关于图形驱动程序的具体信息，Driver Verifier 应该仅当驱动程序在调查状态下才被检查。

2.1.2.1 图形驱动程序的特别内存池

内存讹误是一个常见的驱动程序问题，驱动程序错误能导致在它们建立起来很长时间后崩溃。这些错误当中最常见的要数访问已释放的内存，并分配 n 字节然后是 n+1 字节。

当特别内存池功能应用到图形驱动程序时，**EngAllocMem** 例程分配的内存将被移出特别内存池，Driver Verifier 将监视该池以发现不正确的使用。

两种特别内存池定位是可行的，Verify End 定位能更好的发现访问上溢，Verify Start 定位能更好的发现访问下溢。（注意：最主要的内存讹误是由于上溢，而非下溢。）

当特别内存池运行且选择 Verify End 时，驱动程序所请求的每一内存分配被放到分别的各分页上，在每页上允许分配的最高可能地址被返回，以便内存分配到页末。每页前面的部分以特别形式写出。前页与下一页被标记为不可访问。

如果驱动程序在分配之后尝试访问内存，Driver Verifier 将立即发现并发布错误检测 0xCD。如果驱动程序先于缓冲区开始写入内存，这很有可能改变形式。当缓冲区被释放时，Driver Verifier 将发现并报告错误检测 0xC1。

如果驱动程序在释放缓冲区之后读或写，Driver Verifier 将报告错误检测 0xCC。

当 Verify Start 被选择，内存缓冲区被定位到页的开端，在这种设置下，下溢引起立即的错误检测，而上溢当内存释放时才引起内存检测。这种选项在其他方面与 Verify End 选项相同。Verify End 是缺省定位，是由于驱动程序上溢错误比下溢错误要普遍的多。为改变这种设置，请使用全局标记应用程序。

池标记

Driver Verifier 将给已选择检查的驱动程序分配特别内存池，使用特别内存池的另一个方法是分配它给一个具有特别标记的内存池。

可利用全局标记应用程序致力于将特别池给具有给定标记的池。

同时通过 Driver Verifier 和全局标记应用程序来请求特别池是允许的，如果这么做，Windows2000 将尝试为所有具有指定标记的池和所有指定的驱动程序的池分配请求使用特别池。

特别池效率

特别池的每一个分配使用不可分页内存的一页和有虚拟地址空间的两页。如果池耗尽，内存以标准方法分配，直到特别池再次变得可用为止。这样，如果特别内存池在用，多驱动程序同时被检查则不受推荐。

有大量小内存请求的一个单一驱动程序也会耗尽此池，出现此情况，给驱动程序内存指定池标记和致力于一次给特别池一个池标记将是更可取的。

特别池的大小随系统里的物理内存的大小增长而增长，理想的池大小至少 1GB。在 x86 机器

上，当消耗虚拟空间（对物理空间亦同）时，引导而没有/3GB 的开关也是优先选取的。增加页面文件的最大/最小量（通过两或三当中的一个因子）也是一个好主意。

如果特别内存池可用，但是不到 95%的所有池分配已经从特别池指定，在 Driver Verifier 管理器的 Driver Status screen 上将有一个警告。发生这种情况时，你应该检查一个更短的驱动程序列表，通过池标记检查单个池，或者给你的系统增加更多的物理内存。

确信所有的驱动程序分配已被检测，推荐加强驱动程序较长时间周期

非图形驱动程序

要获得这个选项怎样作用其他内核模式驱动程序的细节，参看特别内存池。

2.1.2.2 图形驱动程序的低资源模拟

当低资源模拟可用时，Driver Verifier 将引起随机的驱动程序内存分配的选择失效，这个过程检查驱动程序对低内存和其他低资源状况下正常反应的能力。

下面的 GDI 回调函数易受制于随机失效：

- **EngAllocMem**
- **EngAllocUserMem**
- **EngCreateBitmap**
- **EngCreateDeviceSurface**
- **EngCreateDeviceBitmap**
- **EngCreatePalette**
- **EngCreateClip**
- **EngCreatePath**
- **EngCreateWnd**
- **EngCreateDriverObject**
- **BRUSHOBJ_pvAllocRbrush**
- **CLIPOBJ_ppoGetPath**

为精确模拟一个低内存条件，这些分配故障直到系统启动 7 分钟后才被注入，因此，任何在该过程中暴露的驱动程序错误将被当做合法的运行时间问题，而非不切实际的现象。

标记为 MUST_SUCCEED 的分配请求不服从于这一过程， MUST_SUCCEED 池每页的最大值被禁止。

这些内存失效也许会引起着色的错误，其表现在不正确的图象和其他输出错误，所以，这个 Driver Verifier 选项应该用于对一图形驱动程序的健全检测，而当检测正确执行时也应该运行。

Low Resources Simulation 选项也许会在外壳图标的高速缓存里引起讹误，发生此情况，图标的高速缓存必须手工从硬盘删除。

非图形驱动程序

参看 *Low Resources Simulation* 来了解该选项如何作用其他内核模式驱动程序的细节。

2.1.3 激活和监视 Driver Verifier

Driver Verifier 由 Driver Verifier 的图形接口或 verifier.exe 的命令行激活，特别内存池选项也可通过 Global Flags Utility 激活成标记的内存池。

Driver Verifier 状态的大多数改变（激活、去活、改变选项、或者改变正被检查的驱动程序列表）在重新启动时仍起作用，无需改变或重新编译驱动程序。Driver Verifier 在 Microsoft® Windows® 2000 的自由和检查构建上都能执行。

然而，一些选项在没有重新启动介入时仍能被激活或去活。细节参见 Driver Verifier 管理器的易变设置窗口或命令 **verifier /volatile /flags *VALUE***。

监视驱动程序行为

Driver Verifier 管理器窗口中的全局计数器和池跟踪，可用来监视 Driver Verifier 的动作和已装载的驱动程序，这些窗口不监视图形驱动程序的检查。

内核调试程序扩展！**verifier** 也能用于监视和报告关于许多相关的 Driver Verifier 行为的统计数字。当检查图形驱动程序时，应该以！**gdikdx.verifier** 扩展代替。关于调试程序扩展的细节，参看微软调试程序的使用文件。

2.1.3.1 检查器（Verifier）命令行

应用程序 verifier.exe 可用于从命令窗口激活 Driver Verifier。

应用程序放在 Windows2000 的 system32 目录下，在 Windows2000 DDK 的 tools 目录下。

命令行选项包括：

verifier /flags VALUE [/iolevel 2] /all

使用指定的选项检查所有已安装的驱动程序，*VALUE* 是下面位的一个综合。

位	作用
0x01	特别内存池
0x02	强迫 IRQL 检查
0x04	低资源模拟
0x08	内存池跟踪
0x10	I/O 检查

位也许会自由地结合，*VALUE* 必须作为一个十进制数输入，十六进制数不被支持。缺省值是 11（1+2+8），如果包括/ **iolevel 2**，I/O 检查被设置为 *Level 2*。（缺省值是 Level 1。）

这些设置在下一次引导后生效。

verifier /flags VALUE [/iolevel 2] /driver NAME [NAME2 NAME3….]

检查所列的驱动程序，被空格分开，多个驱动程序通过罗列它们的名字被确定，但是，诸如 n*.sys 等通配符的值不被支持。象上面的一样，*VALUE* 和/ **iolevel** 相同。

这些设置在下一次引导后生效。

verifier /reset

它清除了所有的 Driver Verifier 设置,在下一次引导后，不检查驱动程序。

verifier /volatile /flags VALUE

不同标准的 Driver Verifier 激活，易变设置立即起作用，*VALUE* 是下面位的一个结合。

位	作用
0x01	特别内存池
0x02	强迫 IRQL 检查

位也许会自由地结合，*VALUE* 必须作为一十进制数输入（而非十六进制数）。

仅仅已列的选项能够有多种设置，在一个多种设置方式里，被检查的驱动程序列表不能改变，且所有的多项设置在重新引导时将丢失。

verifier /query

产生 Driver Verifier 所显示在窗口的当前数据的一个总括，数据包括内存分配的一个枚举、IRQL 的增加、自旋锁和其他 Driver Verifier 选项的相关数据。

verifier /log LOG_FILE_NAME [/interval SECONDS]

产生记录文件来保持内存、IRQL 和自旋锁信息。这个文件有以 *SECONDS* 指定的频率写上去的当前数据，缺省间隔是 30 秒。

verifier /?

显示帮助信息。

2.1.3.2 Driver Verifier 管理器

Driver Verifier 管理器是一个图形接口，其允许你激活 Driver Verifier 并监视它的动作状态。

Driver Verifier 管理器有五个不同的标记页，**Driver Status**、**Global Counters** 和 **PoolTracking** 标记用于显示信息和监视正被检查的驱动程序，**Settings** 标记用来激活和配置 Driver Verifier 的动作；对此窗口的改变将在下次引导后起作用，**Volatile Settings** 标记用于立即改变 Driver Verifier 的动作（在重新引导之前）。

启动 Driver Verifier 管理器

Driver Verifier 的快捷方式能启动 Driver Verifier 管理器，该快捷方式安装在 Windows2000 DDK 里。

在命令窗口运行应用程序 *verifier.exe* 也能启动 Driver Verifier 管理器。该文件放在 Windows2000 的*%windir%\system32* 目录下，在 Windows2000 DDK 的 *tools* 目录里也有。在没有命令行选项时运行 *verifier.exe* 来激活 Driver Verifier 管理器。

驱动程序状态窗口

图：

Driver Verifier 管理器驱动程序状态窗口

Driver Status 窗口显示哪一些驱动程序正被装载、哪一些驱动程序被检查和哪一些 Driver Verifier 选项可用。

全局标志部分表明哪一些 Driver Verifier 选项当前正在使用。

被检查的驱动程序部分列出所有驱动程序，Driver Verifier 已被指示检查这些程序，可能的状态值是：

- **Loaded**

  这意味着驱动程序当前已经装载并正被检查。

- **Unloaded**

  这意味着驱动程序自最后一次引导以来被装载和检查过至少一次，但目前没被装载。

- **Never Loaded**

  这意味着指示 Driver Verifier 来检查这个驱动程序，但驱动程序自最后一次引导以来未被装载，这可能由讹误的驱动程序图形文件所致。

通过点击标题栏上的 Driver 或 Status 来分类这个列表。

重设频率部分控制多久刷新状态栏一次，高、中和低指令 Driver Verifier 以不同的速度刷新该栏，而手工不能自动更新此栏。按钮 **Refresh Now** 可使状态栏立即被更新。

如果特别内存池可用，但是不到 95%的所有池分配已经从特别池指定，窗口上将出现一个警告信息。参看特别内存池或图形驱动程序特别内存池获取详细资料。

全局计数器窗口

图：

Driver Verifier 管理器的全局计数器窗口

全局计数器（Global Counters）窗口显示统计数字，这些数字可帮助监视 Driver Verifier 的行为。

分配计数器对监视与特别内存池相关的统计数字特别有用，这些计数器仅记录标准内核模式驱动程序所使用的池，而不包括图形驱动程序在内，这里 Driver Verifier 正在检查标准内核模式驱动程序。当然，如果相应的功能不可用，这些计数器当中的一些将被设置为零，举例说，如果低资源模拟没运行，注入计数器的故障将为零。

重设频率部分控制这些计数器多久被更新一次，高、中和低指令 Driver Verifier 以不同的速度刷新计数器，而手工不能自动更新。按钮 **Refresh Now** 可使计数器立即更新。

池跟踪窗口

图：

Driver Verifier 管理器的池跟踪窗口

池跟踪（Pool Tracking）窗口显示了关于分页和未分页的池分配的信息（包括当前值和峰值）。

此窗口对监视与内存池跟踪有关的统计数字非常有用。

单个计数器部分每次显示一个驱动程序的统计数字，此窗口顶部的下拉框允许你从当前正被检查的所有驱动程序中选择。

在全局计数器部分，不能跟踪的分配计数器显示了当前正被检查的所有驱动程序的未跟踪分配数目。由于特别池不能用于这些计数器，而计数器不跟踪大小为一页或更大的分配。

重设频率部分控制这些计数器多久被更新一次，高、中和低指令 Driver Verifier 以不同的速度刷新计数器，而手工不能自动更新。按钮 **Refresh Now** 可使计数器立即被更新。

设置窗口

图：

Driver Verifier 管理器的设置窗口

设置（Settings）窗口是 Driver Verifier 和它的选项的主要控制面板。

当按下 **Apply** 按钮时，从 **Settings** 窗口传来的所有改变被写到注册表里。如果你退出 Driver Verifier 管理器而没有按 **Apply**，将问你是否想应用或丢弃你的改变。

当再次引导之后，Driver Verifier 将遵照窗口上的改变，直到你重新引导，将不会有 **Settings** 窗口的改变对 Driver Verifier 的功能起作用。

窗口上当前系统可用的驱动程序部分列出了在最近的一次引导里系统所装载的驱动程序。

Verification Status 栏给出了每个驱动程序的当前状态，可能的检查状态值有：

- **Verify Enabled**

  这意味着驱动程序正被检查，在重新引导后也将继续被检查。

- **Verify Disabled**

  这意味着驱动程序当前没被检查，在重新引导后也不被检查。

- **Verify Enabled（Reboot Needed）**

  这意味着驱动程序当前没被检查，但是用户已请求检查该驱动程序，在重新引导后它将被检查。

- **Verify Disabled（Reboot Needed）**

  这意味着驱动程序正被检查，但是用户已请求结束检查，重新引导后，该驱动程序不再被检查。

你可以从这个列表中选择一个或多个驱动程序，点击 **Verify** 或 **Don't Verify** 按钮，你也可以在驱动程序名右击，并从弹出的菜单控制检查。

在下一次重新引导后检查这些附加的驱动程序窗口，允许你进入当前系统并未装载的驱动程序名。多个名字必须以空格分开。

如果选择检查所有驱动程序，在重新引导后，Driver Verifier 将检查所有的驱动程序。当选择这个选项按钮时，驱动程序列表和 **Verify** 和 **Don't Verify** 按钮变成灰色。

检查类型部分允许你选择在重新引导后哪一个 Driver Verifier 选项可用。参看 Driver Verifier 能力和图形 Driver Verifier 能力中关于这些选项的描述。

Preferred Settings 按钮是打开大部分常用选项的快方式（特别内存池、强迫 IRQL 检测、内存池跟踪和 2 级 I/O 检查）它也选择了检查所有驱动程序按钮。

Reset All 按钮将使所有的驱动程序检查不可用。

易变设置窗口

图：

Driver Verifier 管理器的易变设置

易变设置（Volatile Settings）窗口允许你对 Driver Verifier 做立即的改变（而不是在重新引导之后）。

三种 Driver Verifier 选项——特别内存池、强迫 IRQL 检测和低资源模拟能从该窗口打开或关闭。当按下 **Apply** 按钮时，这些选项当前正被检查的每一个驱动程序激活或去活。

Volatile Settings 窗口不能用来改变被检查的驱动程序列表，也不能改变 Memory Pool Tracking 或 I/O 检查状态，所有多项设置在重新引导后将丢失。

该窗口上的检查驱动程序和更新频率部分对应驱动程序状态窗口是同一的。

2.1.3.3 全局标记应用程序（Global Flags Utility）

通过调整运行系统的内核设置，改变图像文件的设置，全局标记应用程序 *gflags.exe* 提供了一种在系统注册表里设置特定键的简单方法。你能通过使用一个 GUI 或命令行接口设置这些键。应用程序放在 Windows2000 DDK 的 *bin* 目录下面。

全局标记应用程序也能用来配置 Driver Verifier 的特别内存池，或用特别内存池进行单个内存分配。

图：
全局标记应用程序窗口

为改变特别内存池设置，必须选择目标部分里的系统注册表选项按钮。

内核特别池标记对话框允许设置一定的特别池选项。

如果一池标记被输入，特别池将应用于具有指定的池标记的所有池分配。如果同时通过 Driver Verifier 使特别内存池可用，有指定标记的所有池和从正被检查的驱动程序所分配的所有池将从特别池分配（易受池的可用性影响）。

Verify Start 选项按钮引起特别池的定位集中在下溢检测，Verify End 选项则集中在上溢检测。这些按钮控制特别池分配的定位，这些特别池分配由池标记或 Driver Verifier 完成。

关于特别内存池的使用的细节，参看特别内存池和图形驱动程序的特别内存池。

第二卷　即插即用、电源管理和设置设计指南

第一部分　即插即用和电源管理的要求

第1章　即插即用和电源管理的介绍

本章描述了与编写支持即插即用和电源管理的驱动程序相关的主要概念，这些信息如下组织：

阅读第 2 章对驱动程序怎样支持 PnP 和电源管理有一总体印象。阅读第 2 部分获取关于 PnP 的详细信息，阅读第 3 部分获取关于电源管理的详细信息，阅读第 4 部分获取关于驱动程序和设备安装的详细信息。

1.1 什么是即插即用？

即插即用（PnP）是一个硬件和软件支持的组合，这些软件支持能让系统识别或适应硬件配置的小改变而没用用户的干预。用户可从计算机系统添加或删除设备而不必做笨拙而且混淆的手工配置，也不需要复杂的计算机硬件知识。例如，用户可以连接一便携机并使用连接站键盘、鼠标和监视器而不需要手工改变配置。

PnP 需要从硬件设备、系统软件和驱动程序获得支持。硬件工业的最初思想是给容易标识的插件板和基本的系统组件定义标准（比如 PnP ISA 定义和 PC Card 标准）。这本 DDK 文档编制集中在系统软件对 PnP 的支持和驱动程序怎么使用这个支持来实现 PnP。

系统软件对 PnP 的支持和 PnP 驱动程序一起提供以下功能：

- 对已安装硬件的自动和动态识别

在最初的系统安装过程中系统软件识别硬件，识别发生在系统引导之间的 PnP 硬件改变，并对诸如连接或断开的运行时间硬件事件和设备添加或删除作出响应。

- 硬件资源分配（和再分配）

PnP 管理器决定由每一设备（例如，输入/输出[I/O]端口、中断请求[IRQ]、直接内存访问[DMA]通道和内存定位）所请求的硬件资源，并适当分配硬件资源。当需要时，PnP 管理器重新配置资源分配。例如当新设备被添加到必须正在使用的资源系统中。

PnP 设备驱动程序不分配资源，相反，当一个设备被枚举时，设备所请求的资源是被标识过的。在资源分配过程中，PnP 管理器找到每个驱动程序的请求。对一个早期的设备而言，资源不是动态配置的，所以，PnP 管理器首先给早期的驱动程序分配资源。

- 适当的驱动程序的加载

PnP 管理器决定哪一些驱动程序对支持每个设备是必要的，并加载这些驱动程序。

- 驱动程序与 PnP 系统相互作用的编程接口

接口包括 **Io***Xxx* 例程、PnP IRP、必要的驱动程序入口点和注册表信息。

- 驱动程序和应用程序了解硬件环境变化并采取适当的动作机制

PnP 能让驱动程序和用户模式代码登记，并通知一定的硬件事件。

PnP 驱动程序是 PnP 支持的重要部分，作为一个合格的 PnP 驱动程序必须提供必要的 PnP 入口点、处理必要的 PnP IRP 和遵从 PnP 向导行。欲知更多信息，参见第 2 章。

1.1.1 PnP 组件

图 1.1 显示了一组同时工作来支持 PnP 的组件

图 1.1 PnP 软件组件

PnP 管理器有两个部分：内核模式 PnP 管理器和用户模式 PnP 管理器。内核模式 PnP 管理器与 OS 组件和驱动程序相互作用来配置、管理和维护设备。用户模式 PnP 管理器与用户安装组件相互作用来配置和安装设备，例如类安装器。用户模式 PnP 管理器也与一些应用程序相互作用，举例来说，登记一个设备改变通知的应用程序，并当一个设备事件发生时通知应用程序。

PnP 驱动程序支持物理的、逻辑的和虚拟的机器设备，WDM 驱动程序遵从微软的驱动程序模式并支持微软的 Windows2000 和 Windows98 系统上的设备。WDM PnP 驱动程序通过严格定义的 API 和 IRP 与 PnP 管理器和其他内核组件通信。参看《Windows2000 驱动程序开发参考》卷一获得这些 API 和 IRP 的详尽描述。安装了 Windows2000 的机器也许有一些 PnP 驱动程序不支持 WDM。

所有 PnP 驱动程序应该支持 PnP 和电源管理，如果一单个 PnP 的驱动程序不支持 PnP 和电源管理，它作为一个系统整体约束 PnP 驱动程序和电源管理。

参考第 4 部分关于设备和驱动程序安装的信息，包括 INF 文件、.cat 文件和注册表。

1.1.2 PnP 的支持级别

设备支持 PnP 的范围取决于设备硬件和设备驱动程序里的 PnP 支持(看表 1.1)。

表 1.1　对驱动程序/设备综合的 PnP 能力级

	PnP 驱动程序	非 PnP 驱动程序
PnP 设备	完全 PnP	无 PnP
非 PnP 设备	可能的部分 PnP	无 PnP

任何支持 PnP 的设备在其驱动程序里都应该有 PnP 支持。

如果一个非 PnP 设备被一个 PnP 驱动程序驱动的话，它能够有一些 PnP 能力。举例来说，一个 ISA 声卡和一个 EISA 网卡可手工安装，然后 PnP 驱动程序能象一个 PnP 设备一样对待该卡。

如果一驱动程序不支持 PnP，它的设备表现为非 PnP 设备而不管任何硬件 PnP 支持。一个非 PnP 驱动程序能够约束整个系统的 PnP 和电源管理能力。

在 OS 支持 PnP 之前编写的早期驱动程序继续如从前一样工作，但没有任何 PnP 能力。新的驱动程序应该包括 PnP 支持。

1.2 什么是电源管理？

电源管理是系统范围的、集成的方法来使用和保存电源，包括电源管理的软硬件支持的计算机系统提供以下功能：

- 最小的启动和关机延迟

系统能在低电力状态时休眠，在这种状态下无须完全的重新引导系统就能恢复操作。从用户的角度来看，唤醒一休眠的计算机大致可象开关电视机一样。

- 更强大的电源效率和电池生命期

仅仅当设备传输功能给用户时电源才启用于设备，如果一个设备未使用，按照命令，电源可被断开，而后再加电。

- 更安静的操作

断开未被使用的设备电源可降低噪音，此外，硬件和软件能够管理电流装载和热装载，其结果是计算机休眠时几乎听不到声音。

工业界 OnNow Initiative 定义了电源管理的硬件和软件需求，细节参看电源管理的工业萌芽。

在 Windows2000、Windows98 和其他支持电源管理的操作系统里，计算机和其他外部设备被维持在完成手头任务所需的最低可行的电平上，驱动程序与操作系统协作来管理它们的设备电源。如果所有的驱动程序支持电源管理，操作系统能够在系统范围基础上管理电力消耗，这样，电力保存、电力切断和快速恢复，及需要时再次唤醒都可被管理。

这些集成的电源管理方法---包括在操作系统、系统硬件、设备驱动程序和设备硬件里，产生如下的结果：

- 更智能的电源管理决定。在每一电平，被最优通知的组件指导电力使用。
- 更高的可靠性。更好的电源管理决定减少了不适当的时间关机和数据丢失的可能。
- 平台的独立性。操作系统在不同的硬件平台之间使用一种可控制的、单独的方法来管理电源，允许电力在不同的设备上最大保存。
- 更好的设备集成。符合工业界规范的设备驱动程序确保最大的电力保存而不管其硬件平台。

综合而言，这些优点导致保存更多电能和更有效地利用电能，这在以前是不可能的。

1.2.1 电源管理的最初含义

OnNow Initiative 描述了电源管理所需的硬件和软件支持。

高级配置和电源管理接口规范的 OnNow Initiative 部分，描述了一个硬件级接口，该接口能让操作系统以一种一致的、平台独立的方式实现电源管理。

设备类电源管理参考规范对每个普通的设备类是可行的，如声频或通信设备。每一个规范描述了一个设备类的电源管理要求。驱动程序的编写者应该参考这些规范，也可通过微软的网站来获得设备相关的细节。

1.2.2 电源管理的支持级别

支持 PnP 的驱动程序必须支持电源管理，支持电源管理的驱动程序也必须支持 PnP，电源管理和 PnP 是被集成而互不独立的。

电源管理有两个工作层次：整个系统和单个设备。

电源管理器，是操作系统内核的一部分，管理着整个系统的电源级别。如果系统中所有的驱动程序都支持电源管理，电源管理器能在系统范围的基础上管理电源消耗，不只利用全开或全关的状态，也有多种中间的系统休眠状态。

在操作系统支持支持电源管理之前编写的早期驱动程序能如从前一样继续工作。然而，包括早期驱动程序的系统不能够进入任何中间的系统休眠状态；它们只能象过去一样在全开或全关的状态下操作。

驱动程序处理设备电源管理。一个支持电源管理的驱动程序能够在需要时打开它的设备，不用时关闭设备。如果设备有硬件能力允许这么做，设备能够进入中间的电源状态。当前系统中早期的驱动程序没有影响更新的驱动程序管理它们设备电源的能力。

1.2.3 全系统范围的电源管理的总览

电源管理需要得到系统、设备硬件和系统软件、驱动程序的支持。如在“电源管理的最初含义”部分描述的一样，必要的硬件支持包括在工业规范里。这一部分包括了软件支持，特别地，驱动程序该做什么来遵照操作系统的要求，并正常的管理它们的设备电源。

图 1.2　给出了全系统范围的电源管理总览

分别通过 API 和控制面板，应用程序和用户已经分别地输入了电源管理决定。用户可通过控制面板设置一定的电源选项，包括设备专用的 GUID 控制选项，这个选项是如果响应的驱动程序支持它时，用户界面将会提供。通过 WMI，控制面板通知电源管理器和驱动程序 GUID 控制里的改变。

电源管理器管理全系统的电源策略，这些策略包括管理系统电源使用的规则。（细节参见第 3 章第 3 部分的系统电源管理。）使用从控制面板和得到的信息，电源管理器能够决定什么时候应用程序在使用，或者可能需要使用不同的设备，这样它能够适当调整系统的电源策略。

电源管理器也提供一个驱动程序接口，该接口主要包括 **Po***Xxx* 例程、电源管理 IRP 和必要的驱动程序入口点。

当电源管理器请求系统电源状态的变化时，驱动程序通过放置它们设备在适当的设备电源状态来响应。此外，驱动程序能发现是否设备闲置，并设置未用的设备处于休眠状态。总线专用的机制报告设备电源能力、设置并报告设备状态及改变设备电源。怎样和何时确切地改变设备电源取决于设备类型和设备硬件的能力。

尽管 ACPI 硬件意识到了最大地电力储存，该硬件不需要为在驱动程序里生效的电源管理而遵从 ACPI 规范。

1.2.4 电源状态

电源状态指示了系统或单个设备的电平消耗，也指示了枚举活动的范围。电源管理器设定整个系统的电源状态，而设备驱动程序设置它们单个设备的电源状态。

ACPI 规范定义了系统和单个设备的不连续电源状态。系统电源状态被引用为 *Sx*，这里 *x* 是 0 到 5 之间的一个状态数，设备状态被引用为 *Dx*，这里 *x* 是 0 到 3 之间的一个状态数。状态数与电力消耗呈反相关：越高的状态数使用越少的电力，状态 S0 和 D0 是最高的电力，也是最具功能的，是最完全的状态，状态 S5 和 D3 是最低的电力状态并有最长的唤醒时间。

严格定义的电源状态允许不同制造商的许多设备能一致地、可预测地一起工作。举例来说，当电源管理器设置系统为 S3 状态时，其能依靠支持电源管理器的驱动程序设置它们设备相应的设备状态，并能以一种可测的形式返回工作状态。

1.1 设备树

OS 维持一个能保留系统里的设备踪迹的设备树。图 1.3 给出了系统配置的设备树的一个例子。

设备树保留着存在于系统中的设备信息，当机器引导时，通过使用从驱动程序和其他组件获得的信息，OS 构建此树，并当设备被添加或删除时更新此树。设备树是分级的，总线上的设备代表着总线适配器或控制器的“子级”。你可参看设备管理器的使用来了解设备树的设备分级（选择“View”，然后选择“Devices by connection”）。

设备树的每一个节点是一个设备节点（devnode），一个 devnode 包括设备驱动程序的设备对象加上由 OS 所保留的内部信息。参看 1.4.3 部分来获取更多的信息。

图 1.3PnP 设备树例子

设备树的分级反映了机器里设备附着的结构，当 OS 管理设备时它使用该分级。例如，如果一用户请求从图 1.3 所示机器里拔掉 USB 控制器，PnP 管理器从设备树上决定该行为也将导致另外的三个设备被拔掉（USB 集线器、操纵杆和镜头）。当 PnP 管理器就 USB 控制器询问驱动程序删除控制器是否安全时，PnP 管理器同样地就控制器（USB 集线器、操纵杆和镜头）的任何子代而查询驱动程序。

除过在设备树里给出的分级关系外，机器设备之间有其他的关系，这里包括删除和抽出的关系。为了更多的信息，参见《Windows2000 驱动程序开发参考》的卷一参考页里的 IRP_MN_QUERY_DEVICE_RELATIONS。

1.3 驱动程序层次和设备对象

I/O 系统提供一个分层的驱动程序体系结构，这一部分从支持 PnP 和电源管理的驱动程序的角度讨论了体系结构，讨论包括各种驱动程序、驱动程序层次和设备对象。

1.3.1 驱动程序种类

从 PnP 的角度，有三种驱动程序：

- 总线驱动程序---驱动 I/O 总线并提供设备间独立的每个插槽功能
- 功能驱动程序---驱动一个单个的设备
- 过滤器驱动程序---过滤一个设备、设备类或总线的 I/O 请求

在本书中，总线是物理的、逻辑的或虚拟的设备所能附着的任何设备；总线包括传统的总线，如 SCSI 和 PCI，也有并口、串口和 i8042 端口。

对一个驱动程序编程人员来说，知道不同的 PnP 驱动程序种类和他们正在编写哪一类驱动程序是重要的。例如，是否一个驱动程序处理每个 PnP IRP 和怎样处理这个 IRP 取决于正在编写哪一类的驱动程序（一个总线驱动程序、功能驱动程序还是过滤器驱动程序）。

图 1.4 给出了一个设备的总线驱动程序、功能驱动程序和过滤器驱动程序之间的关系。

每个设备典型地有一个父 I/O 总线的总线驱动程序管理，一个设备的功能驱动程序，零个或多个设备的过滤器驱动程序。需要许多过滤器驱动程序的驱动程序设计不能产生最佳的执行。

图 1.4 里的驱动程序如下：

1. 总线驱动程序服务于一个总线控制器、适配器和电桥。总线驱动程序是必须的驱动程序；在一个机器上，每个类型的总线有一个驱动程序。微软为大多数通用总线提供了总线驱动程序，IHV 和 OEM 则提供了其他的总线驱动程序。
2. 总线过滤器驱动程序典型地给一总线增加值，它由微软或系统 OEM 提供，对一个总线来说，可以有任何数目的总线过滤器驱动程序。
3. 低层过滤器驱动程序典型地修改了设备硬件的行为，它们是可选择的且典型地由 IHV 提供，对一个设备来说，可以有任何数目的低层过滤器驱动程序。

图 1.4 可能的驱动程序层次

4. 功能驱动程序是主要的设备驱动程序，一个功能驱动程序典型地由且必须由设备商来编写（除非设备使用原始模式）。
5. 顶层过滤器驱动程序典型地提供设备的增值特征，它们是可选择的且典型地由 IHV 提供。

下面各部分描述了驱动程序的总体类型（总线驱动程序、功能驱动程序和过滤器驱动程序）。

1.3.1.1 总线驱动程序

总线驱动程序服务一个总线控制器，适配器，或者电桥（参看图 1.4），微软为大多数通用总线提供总线驱动程序，例如 PCI，PnpISA，SCSI 和 USB。其他的总线驱动程序由 IHV 和 OEM 提供。总线驱动程序是必须的驱动程序；在一个机器里，每一类总线有一个总线驱动程序。如果机

器里有不止一个同类的总线，则一个总线驱动程序能服务不止一个总线。

总线驱动程序的主要任务是：

- 枚举其总线上的设备
- 响应 PnP 和电源管理 IRP
- 总线的多路访问（对某些总线）
- 总体上管理其总线上的设备

在枚举过程中，一个总线驱动程序识别它的总线上的设备并为它们产生设备对象。总线驱动程序用来标识相连接的设备方法取决于特别的总线。

总线驱动程序代表其总线上的设备来执行一定的操作，包括访问设备寄存器来物理地改变设备的电源状态。例如，当设备休眠时，总线驱动程序设置设备寄存器来给设备适当的电源状态。

但要注意，总线驱动程序不能够处理其总线上的设备的读和写请求，一个设备的读和写请求由设备功能驱动程序处理（看 1.4.1.2 部分）。仅仅当设备以原始模式使用时，父总线驱动程序处理设备的读和写。

总线驱动程序为控制器，适配器，或者电桥起着功能驱动程序的作用，并因此为控制器，适配器，或者电桥管理设备电源策略。

总线驱动程序能够作为一个驱动程序/小驱动程序对来执行，即以 SCSI 端口/微端口对驱动一个 SCSI HBA（主机总线适配器）的方法。在这样的驱动程序对里，小驱动程序与次一级驱动程序（它是一个 DLL）相链接。

1.3.1.2 功能驱动程序

功能驱动程序是设备的主要驱动程序（参见图 1.4）。一个功能驱动程序典型地由设备厂商且必须由设备厂商来编写（除非设备使用原始模式）。PnP 管理器为一个设备至少装载一个功能驱动程序，功能驱动程序能服务一个或多个设备。

功能驱动程序为它的设备提供操作接口，典型地，功能驱动程序处理设备的读和写并管理设备电源策略。

一个设备的功能驱动程序能够被作为一个驱动程序/小驱动程序对来执行，例如一个端口/微端口或一个类/微类对。在这样的驱动程序对里，小驱动程序与次一级驱动程序（其是一个 DLL）相链接。

如果以原始模式驱动一个设备，它没有功能驱动程序和顶层或低层过滤器驱动程序。所有的原始模型 I/O 通过总线驱动程序和可选择的总线过滤器驱动程序来实现。

1.3.1.3 过滤器驱动程序

过滤器驱动程序是可选择的驱动程序，该驱动程序给设备增加值或修改设备的行为。过滤器驱动程序能服务于一个或多个设备。

总线过滤器驱动程序

总线过滤器驱动程序典型地给总线添加数值，它由微软或系统 OEM 提供（看图 1.4）。总线过滤器驱动程序是可选择的，对一个总线来说，可以有任何数目的总线过滤器驱动程序。

举例来说，总线过滤器驱动程序能够实现标准总线硬件的特性增强。

对 ACPI BIOS 所描述的设备，电源管理在每个这样的设备的总线驱动程序之上插入微软提供的一个 ACPI 过滤程序（总线过滤器驱动程序）。ACPI 过滤程序执行设备电源策略，并打开和关

闭设备电源，ACPI 过滤程序对其他驱动程序是透明的且在非 ACPI 机器上是没有的。

低层过滤器驱动程序

低层过滤器驱动程序典型地修改设备硬件的行为（看图 1.4），它们典型地由 IHV 提供且是可选择的，对一个设备来说，可以有任何数目的低层滤驱动程序。

低层设备过滤器驱动程序监视和/或修改一个特定设备的 I/O 请求，典型地，这些过滤程序重新定义了硬件行为来匹配期望的规范。

低层类过滤器驱动程序监视和/或修改一个设备类的 I/O 请求，例如，通过执行一非线性鼠标运动数据转换，鼠标设备的低层类过滤器驱动程序能够提供加速。

顶层过滤器驱动程序

顶层过滤器驱动程序典型地为一设备提供增值的特征（看图 1.4），这种驱动程序通常由 IHV 提供且是可选择的。一个设备可有任意数目的顶层过滤器驱动程序。

顶层设备过滤器驱动程序为一个特别的设备添加数值。例如，一个键盘的顶层设备过滤器驱动程序能够加强额外的安全检查。

顶层类过滤器驱动程序为一个特别类的所有设备添加数值。

1.3.2 驱动程序层次---一个例子

这一部分描述了 USB 硬件可能的一套 PnP 驱动程序来说明 PnP 驱动程序的层次。

图 1.5 给出了一个 USB 操纵杆的 PnP 硬件配置的例子，在图 1.5 里，USB 操纵杆插进一个 USB 集线器上的一端口。该例中的 USB 集线器常驻在 USB 主控制器面板上，且插到 USB 主控制器面板上的单个端口里，USB 主控制器插到一个 PCI 总线上。从 PnP 的角度来看，USB 集线器、USB 主控制器和 PCI 总线都是总线设备，因为它们每一个都提供端口。操纵杆不是总线设备。

图 1.5 PnP 硬件的 USB 操纵杆的例子

图 1.6 给出了一套驱动程序的例子，这些驱动程序可能为图 1.5 中的 USB 操纵杆硬件的原因而装载。

开始于图 1.6 的底部，堆栈例子里的驱动程序包括：

- 驱动 PCI 总线的一个 PCI 驱动程序，这是一个 PnP 总线驱动程序，PCI 总线驱动程序由微软的系统所带。
- USB 主控制器的总线驱动程序作为一个类/微类驱动程序对来执行。微软系统带有 USB 主控制器类和微类驱动程序。
- USB 集线器总线驱动程序驱动 USB 集线器，微软系统提供 USB 集线器驱动程序。
- 操纵杆设备的三个驱动程序，其中之一是一个类/微类对。

图 1.6 PnP 驱动程序层次的例子--- USB 操纵杆

功能驱动程序，即 USB 操纵杆设备的主要驱动程序，是一个 HID 类驱动程序/HID USB 微类驱动程序对。（HID 代表“human interface device”）HID USB 微类驱动程序支持 HID 设备的 USB 专用的语法，其依赖 HID 类驱动程序 DLL 对 HID 驱动程序的总体支持。

功能驱动程序能专用于特定的设备，或在 HID 状态下，一个功能驱动程序能服务一组设备。

在这个例子当中，HID 类/HID USB 微类驱动程序对服务于一 USB 总线系统中的任何 HID-compliant 设备。一个 HID 类驱动程序/HID 1394 微类驱动程序对将服务于 1394 总线系统里的任何 HID-compliant 设备。

功能驱动程序由设备厂商或微软编写。在这个例子当中，功能驱动程序（HID 类/HID USB 微类驱动程序对）由微软编写。

在该例中，操纵杆设备有两个过滤器驱动程序：一个顶层类过滤程序添加一个宏按钮特征和一个低层设备过滤程序使操纵杆能枚举鼠标设备。

需要过滤操纵杆 I/O 编写者编写顶层过滤程序，操纵杆厂商则编写低层过滤器驱动程序。

- 内核模式、用户模式 HID 客户和应用程序不是驱动程序，但为了全面一并给出。

1.3.3 设备对象种类

一个驱动程序为它所控制的每个设备产生设备对象，设备对象代表驱动程序的设备。从 PnP 的角度看有三种设备对象：

- 物理设备对象（PDO）---代表一个总线驱动程序的总线上的设备
- 功能设备对象（FDO）---代表一个功能驱动程序的设备
- 过滤程序设备对象（Filter DO）---代表一个过滤器驱动程序的设备

这三种设备对象都是 DEVICE_OBJECT 类型，但是使用方式不同并有不同的设备扩展。

通过产生一设备对象（**IoCreateDevice**）并将其附着到设备堆栈（**IoAttachDeviceToDevice_Stack**），驱动程序将其本身添加到处理设备的 I/O 驱动程序堆栈，**IoAttachDeviceToDeviceStack** 决定设备堆栈当前的顶层和附着新的设备对象到设备堆栈的顶层。

图 1.7 给出了设备对象的可能种类，该设备对象可附着于设备堆栈里，表示处理一个设备的 I/O 请求的驱动程序。

这一部分描述了每一类的设备对象并注意到何时产生该类。参看第 2 章获得关于在必要的 PnP 驱动程序例程里产生设备对象的细节信息，要获得 PnP 设备枚举的更多信息，参见第 2 部分。

开始于图 1.7 的底部：

- 总线驱动程序为总线上它所枚举的每个设备产生 PDO。

当总线驱动程序枚举其设备时，它为每个子设备产生 PDO。总线驱动程序枚举一设备为 PnP 管理器的 **BusRelations** 响应一个 IRP_MN_QUERY_DEVICE_RELATIONS 请求。如果自从最近一次总线驱动程序响应 **BusRelations** 的查询关系请求以来（或者这是机器被引导以来第一次查询关系）设备已经添加到总线上，则总线驱动程序为每个子设备产生一个 PDO。

PDO 表示了总线驱动程序的设备，其他内核模式系统组件也和它一样，如电源管理器、PnP 管理器和 I/O 管理器。

一个设备其他的驱动程序附着于 PDO 顶端的设备对象，但是 PDO 一直在设备堆栈的底端。

- 可选择的总线过滤器驱动程序为它们过滤的每个设备产生过滤程序 DO。

当 PnP 管理器在 **BusRelations** 列表里发现一个新设备时，它决定是否有该设备的任何总线过滤器驱动程序。如果是这样的话，对每个这样的驱动程序 PnP 管理器确保它们被装载（如果需要调用 DriverEntry）并调用驱动程序 AddDevice 例程。如果总线过滤器驱动程序为这个设备过滤操作，过滤器驱动程序产生一个设备对象并附着它到 AddDevice 例程里的设备堆栈上。如果不止一个总线过滤器驱动程序存在，且与这个设备相关，每个这样的过滤器驱动程序产生并附着于它自己的设备对象。

■ 可选择的，低层过滤器驱动程序为它们过滤的每个设备产生过滤程序 DO。

如果一可选择的低层过滤器驱动程序由于这个设备的原因而存在，PnP 管理器确信在总线驱动程序和任何总线过滤器驱动程序之后装载了这样的驱动程序。PnP 管理器调用过滤器驱动程序的 AddDevice 例程，在它的 AddDevice 例程里，低层的过滤器驱动程序为设备产生一个过滤程序 DO 且附着它到设备堆栈里。如果不止一个低层过滤器驱动程序存在，每个这样的驱动程序将产生并附着它自己的过滤程序 DO。

■ 功能驱动程序为设备产生一个 FDO。

PnP 管理器确信已安装了设备的功能驱动程序并调用功能驱动程序的 AddDevice 例程，功能驱动程序产生一个 FDO 并附着它到设备堆栈里。

■ 可选择的，顶层过滤器驱动程序为它们过滤的每个设备产生过滤程序 DO。

如果任何可选择的，顶层过滤器驱动程序为设备而存在，PnP 管理器确信在功能驱动程序调用它们的 AddDevice 例程之后被安装，每个这样的过滤器驱动程序附着它的设备对象到设备堆栈。

总之，设备堆栈包括每一驱动程序的设备对象，该驱动程序参与了特定设备的 I/O 处理。父总线驱动程序有一个 PDO，功能驱动程序有一个 FDO，每一个可选择的过滤器驱动程序有一个过滤程序 DO。

注意到所有的设备---总线适配器/控制器设备和非总线设备---在它们的设备堆栈里有一个 PDO 和 FDO。总线适配器/控制器的 PDO 由父总线的总线驱动程序产生。例如，如果一个 SCSI 适配器插入一个 PCI 总线，PCI 总线驱动程序为 SCSI 适配器产生一个 PDO。

如果一个设备正以原始模式使用，则没有功能驱动程序或过滤器驱动程序（没有 FDO 或过滤程序 DO）。此时还有父总线驱动程序的一个 PDO 和零个或更多总线过滤程序 DO。

要获取哪一个驱动程序例程负责产生和附着设备对象的信息，参看第 2 章。

设备堆栈和一些额外信息构成了一个 devnode 设备。在一个设备的 devnode 里，PnP 管理器保留诸如是否设备已经启动和哪一个驱动程序，如有，登记通知设备上的改变。内核调试程序！**devnode** 命令显示了关于一个 devnode 的信息。

1.3.4 设备对象---一个例子

这一部分描述了由 USB 硬件的驱动程序产生可能的驱动程序的设备对象来说明 PnP 设备对象以及它们怎样分层。

图 1.8 给出了在 1.4.2 部分里描述的驱动程序例子所产生的设备对象。

图 1.8　PnP 设备对象分层的例子---USB 操纵杆

开始于图 1.8 的底部，在设备堆栈例子里的设备对象包括：

1. PCI 总线的一个 PDO 和一个 FDO

根总线驱动程序枚举内部系统总线（根总线）并为它所发现的每个设备产生一个 PDO，这些 PDO 的其中之一支持 PCI 总线。（根总线的 PDO 和 FDO 没有在图中列出。）

PnP 管理器标识 PCI 驱动程序为 PCI 总线的功能驱动程序，安装驱动程序（如果还没安装的话），并传递 PDO 给 PCI 驱动程序。在它的 AddDevice 例程里，PCI 驱动程序产生 PCI 总线（**IoCreateDevice**）的一个 FDO 并增加 FDO 到 PCI 总线的设备堆栈（**IoAttachDeviceToDeviceStack**）。象 PCI 总线的功能驱动程序一样，PCI 驱动程序产生并附着这

个 FDO 作为它的部分任务。

在该例中，没有 PCI 总线的过滤器驱动程序。

2. USB 主控制器的一个 PDO 和一个 FDO

PnP 管理器指示 PCI 驱动程序来启动它的设备（IRP_MN_START_DEVICE），然后查询它的子代（IRP_MN_QUERY_DEVICE_RELATIONS **BusRelations**）PCI 驱动程序。作为响应，PCI 驱动程序枚举其总线上的设备。在这个例子中，PCI 驱动程序发现了一个 USB 主机控制器并产生那个设备的一个 PDO。图中的宽箭头表示了 USB 主机控制器是 PCI 总线的一个“子代”。PCI 驱动程序生成它子设备的 PDO，象支持 PCI 总线的总线驱动程序一样，作为它的部分任务。

PnP 管理器标识 USB 主机控制器微类/类驱动程序对作为 USB 主机控制器的功能驱动程序并装载该驱动程序对，PnP 管理器在适当的时间调用该驱动程序对生成并附着 USB 主机控制器一个 FDO。

在该例中，没有过滤器驱动程序支持 USB 主机控制器。

3. USB 集线器的一个 PDO 和一个 FDO

USB 主机控制器枚举它的总线、在单个端口上定位 USB 集线器，并为集线器生成一个 PDO。USB 集线器驱动程序生成并且为集线器附加一个 FDO。

在该例中，没有 USB 集线器的过滤器驱动程序。

4. 操纵杆设备的一个 PDO、一个 FDO 和两个过滤程序 DO

USB 集线器驱动程序枚举它的总线、定位一个 HID 设备（操纵杆）和生成一个操纵杆 PDO。在该例中，低层过滤器驱动程序已经在注册表中为操纵杆设备而设置，这样，PnP 管理器装载过滤器驱动程序。过滤器驱动程序决定相关设备并给设备堆栈生成和附加一个过滤程序 DO。

PnP 管理器决定了操纵杆设备的功能驱动程序是 HID 类/微类驱动程序对并装载这些驱动程序。驱动程序对包含一个链接到类驱动程序 DLL 的一个微类驱动程序；它们一起作为设备的一个功能驱动程序来执行。该类/微类驱动程序对生成一个设备对象、FDO，并把它附加到设备堆栈中。

顶层过滤器驱动程序生成并附加一过滤程序 DO 到设备堆栈，其方式与低层过滤程序类似。

注意到由父总线驱动程序生成的 FDO 一直处于一个特定设备的设备堆栈的底部，当驱动程序处理 PnP 或电源 IRP 时，它们必须向下通过设备堆栈传递每一个 IRP 给 PDO 和其关的总线驱动程序。

图 1.9 给出了与图 1.8 相同的设备堆栈，但强调哪一个设备对象由哪一个驱动程序生成和管理。

图 1.9 从驱动程序角度看到的设备对象层次的例子

一个总线驱动程序跨越多个设备堆栈，一个总线驱动程序为它的总线适配器/控制器生成 FDO，并为它的每个子设备生成一个 PDO。

第2章　PnP 和电源管理必须的驱动程序支持

本章描述了例程和 IRP 处理，这些在 Windows2000 和 WDM 驱动程序里对支持即插即用（PnP）和电源管理是必须的。它覆盖了以下的主题：

一些驱动程序通过系统提供的端口或类驱动程序与 PnP 和电源系统的细节是隔离的,这样的驱动程序无需实现所有的这些机制。举个例子，一个 SCSI 端口驱动程序屏蔽了一个 SCSI 微端口驱动程序的许多电源管理和 PnP 系统的细节，这样一个 SCSI 微端口驱动程序不需要直接处理电源和 PnP IRP。对这些驱动程序，参看驱动程序专用文件来获得必要的 PnP 支持细节。

参看第一章对 PnP 和电源管理的概念和术语的介绍。

2.1 必须的 PnP 支持的总览

一个支持 PnP 和电源管理的驱动程序必须具有以下功能：

- 一个用来安装驱动程序的 INF 文件
- 一个.cat(catalog)文件，该文件有驱动程序包的 WHQL 数字签名
- 一个用于初始化驱动程序的 **DriverEntry** 例程
- 一个用于初始化设备的 AddDevice 例程
- 一个 DispatchPnP 例程，该例程处理 PnP 操作的 IRP，如启动、停止和删除设备
- 一个 DispatchPower 例程处理电源操作的 IRP
- 一个 Unload 例程删除任何由 **DriverEntry** 设定的驱动程序专用资源

这些例程在下面的部分中讨论。参见第 4 部分获得关于 INF 文件、.cat 文件和驱动程序安装的信息。

2.2　PnP 和电源管理 DriverEntry 例程

一个 **DriverEntry** 例程初始化一个驱动程序，所有的驱动程序必须有一个 **DriverEntry** 例程。当装载驱动程序时，PnP 管理器为每个驱动程序调用一次 **DriverEntry**。在驱动程序初始化之后，PnP 管理器能够调用驱动程序的 AddDevice 例程来初始化由该驱动程序控制的设备。

DriverEntry 例程定义如下：

```
NTSTATUS
```

```
(*PDRIVER_INITIALIZE)  (
    IN  PDRIVER_OBJECT  DriverObject,
    IN  PUNICODE_STRING  RegistryPath
    );
```

只要驱动程序标识了链接程序的初始化例程名，你可以给初始化例程命名 **DriverEntry** 之外的某个名字。链接程序必须有初始化例程的名字来链接驱动程序的传输地址到 OS 装载程序里。一个名为 **DriverEntry** 的驱动程序初始化例程会自动生成该链接。

DriverEntry 例程在一系统线程 IRQL PASSIVE_LEVEL 的环境里被调用。

一个 **DriverEntry** 例程能使用注册表来得到它所需要的初始化驱动程序的一些信息，**DriverEntry** 例程必须在注册表里为另外的驱动程序和/或要用的被保护的子系统设置信息。为获得关于使用注册表的更多信息，请看第 4 部分。

一个 **DriverEntry** 例程是可分页的且应该在 INIT 段以便它将被丢弃。

一个 **DriverEntry** 例程必须采取以下步骤：

1. 初始化驱动程序调度表

一驱动程序必须为驱动程序 AddDevice 例程指定入口点、调度（Dispatch）例程、StartIo 例程（如有）、Unload 例程和其他任何驱动程序入口点。例如，一驱动程序应该用象下面的描述（Xxx 是一个标识驱动程序的前缀）的一样，为它的 AddDevice、DispatchPnp 和 DispatchPower 例程设置入口点：

```
    :
DriverObject->DriverExtension->AddDevice = XxxAddDevice;
DriverObject->MajorFunction[IRP_MJ_PNP]->AddDevice = XxxDispatchPnp;
DriverObject-> MajorFunction[IRP_MJ_POWER] = XxxDispatchPower;
    :
```

2. 初始化任何其他的全局驱动程序变量和数据结构。

由于一个 **DriverEntry** 例程运行在一个系统线程 IRQL PASSIVE_LEVEL 的环境里，只要驱动程序没有控制存有系统分页文件的设备，任何由 **ExAllocatePool** 分配且专在初始化过程中使用的内存能够从分页的池得到。这样的内存分配必须在 **ExFreePool** 在 **DriverEntry** 返回控制之前释放。

3. [可选择的] 在注册表里使用了 **Zw***Xxx* 和 **Rtl***Xxx* 例程来读或者设置独立于设备的值。

4. 保存输入到 **DriverEntry** 里的 RegistryPath 参数。

这个参数指向一个被枚举的 Unicode 信息串，此 Unicode 信息串指定了驱动程序注册表键的一个路径：**\Registry\Machine\System\CurrentControlSet\Services***Drivername*。该例程应该保存一个此信息串的拷贝，而不是指针本身。在 **DriverEntry** 例程返回控制以后，指针不再有效。

5. 返回状态。

如果 **DriverEntry** 例程返回的信息不是 STATUS_SUCCESS，则驱动程序不须保留装载的状态。然而，在初始化失效和返回控制之前，一个 **DriverEntry** 例程必须做下面的事情：

- 释放由它设置的任何系统资源。
- 记录错误。在内核模式驱动程序设计指南节里看 *Error Logging and NTSTATUS Values* 来得到更多的信息。

2.3 PnP 和电源管理 AddDevice 例程

在驱动程序的 AddDevice 例程里，驱动程序创建一个设备对象作为它的 I/O 设备的目标，并附着设备对象到设备堆栈，设备堆栈包括每一个相关设备驱动程序的一个设备对象。

PnP 管理器调用由驱动程序控制的每个设备的一个驱动程序 AddDevice 例程。在设备被首次枚举时，AddDevice 例程在系统初始化时被调用。当系统运行时，任何时候一个新设备被枚举，AddDevice 例程被调用。

一个驱动程序的 AddDevice 例程应该被命名为 *Xxx*AddDevice，这里 *Xxx* 是一个标识特定的驱动程序的前缀。在 **DriverEntry** 过程中，一个驱动程序在 **DriverObject_>DriverExtension->AddDevice** 里存储了它的 AddDevice 例程地址：。

在一个系统线程 IRQL PASSIVE_LEVEL 环境里调用一个 AddDevice 例程。

AddDevice 例程由 PnP 管理器定义如下：

```
NTSTATUS
(*PDRIVER_ADD_DEVICE)  (
    IN  PDRIVER_OBJECT  DriverObject,
    IN  PDEVICE_OBJECT  PhysicalDeviceObject
    );
```

DriverObject 指向代表驱动程序的驱动程序对象，PhysicalDeviceObject 指向被添加的 PnP 设备的 PDO。

在功能或过滤器驱动程序里，一个 AddDevice 例程应该有以下步骤：

1. 调用 **IoCreateDevice** 产生一个被添加的设备的设备对象（一个 FDO）。
   不要指定 FDO 的一个设备名，命名一个 FDO 绕过了 PnP 管理器的安全管理。如果一个用户模式组件需要一个设备的符号链接，须注册一个设备接口（看下一步）。如果一个内核模式组件需要一个早期的设备名，驱动程序必须命名 FDO，但并不推荐命名。
   在设备特性参数里包括 FILE_DEVICE_SECURE_OPEN，这个特性指示 I/O 管理器来执行安全检查防止设备对象的所有打开请求，包括相对打开和尾部文件名打开。
2. [可选择的]生成一个或多个与设备的符号链接。
   调用 **IoRegisterDeviceInterface** 来注册设备功能并生成一个符号链接，通过该链接应用程序或系统组件能够打开设备。当驱动程序处理 IRP_MN_STRAT_DEVICE 请求时，驱动程序应该激活接口（看第 2 部分）。
3. 在设备扩展里存储设备 PDO 的指针。
   PnP 管理器给提供一个 PDO 指针来作为 AddDevice 的 PhysicalDeviceObject 参数，驱动程序使用 PDO 指针调用诸如 **IoGetDeviceProperty** 的例程。
4. 在设备扩展里定义标志来跟踪设备一定的 PnP 状态，比如暂停、删除和突然的删除。
   例如，定义一个标记说明当设备处于一暂停状态时，到来的 IRP 应该被保留。如果驱动程序还没有一个列队 IRP 的机制，将产生一个队列来保留 IRP。为更多信息，请看第 2 部分。
   也在设备扩展里分配一个 IO_REMOVE_LOCK 结构并调用 **IoInitializeRemoveLock** 来初始化该结构。
5. 如果需要，设置电源管理的 DO_POWER_INRUSH 或 DO_POWER_PAGABLE 标记，可分页

的驱动程序必须设置 DO_POWER_PAGABLE 标记。当设备对象产生设备 PDO 时，设备对象标记典型地由总线驱动程序设置。然而，当高层的驱动程序产生 PDO 时，高层的驱动程序有时需要改变它们 AddDevice 例程里的这些标记值。看第 1 章第 3 部分的电源管理设置设备对象标记来获取细节。

6. 产生和/或者初始化任何其他的软件资源，驱动程序使用这些软件资源来管理这个设备，比如事件、自旋锁或者其他的对象。（硬件源，比如 I/O 端口，在稍后配置来响应一个 IRP_MN_START_DEVICE 请求。）

   由于一个 AddDevice 例程在一个系统线程 IRQL PASSIVE_LEVEL 环境里运行，所以只要驱动程序没有控制保留有系统分页文件的设备，任何由 **ExAllocatePool** 分配、且专门在初始化过程中使用的内存从分页池得到。这样的内存分配必须由 **ExFreePool**AddDevice 在返回控制之前释放。

7. 附着设备对象给设备堆栈（**IoAttavhDeviceToDeviceStack**）。

   在 TargetDevice 参数里给设备的 PDO 指定一个指针。

   存储由 **IoAttavhDeviceToDeviceStack** 返回的指针，当向下传递到设备堆栈时，这个指针是 **IoCallDriver** 和 **PoCallDriver** 的一个必要参数，该指针指向紧邻的低层的支持驱动程序的设备对象。

8. 用一个如下的状态，在 FDO 或者过滤程序 DO 里清除 DO_DEVICE_INITIALATION 标记：FunctionalDeviceObject->Flags &=~DO_DEVICE_INITIALIZING;

9. 准备处理设备（比如 IRP_MN_QUERY_RESOURCE_REQUIREMENTS 和 IRP_MN_STRAT_DEVICE）的 PnP IRP。

直到驱动程序接收到一个 IRP_MN_STRAT_DEVICE 时，它保留由 PnP 管理器分配给设备的硬件资源列表，才可以启动控制设备。

一个 PnP 总线驱动程序有一个 AddDevice 例程，但是，当总线驱动程序用作它的控制器和适配器的功能驱动程序时，该 PnP 总线驱动程序才被调用。例如，PnP 管理器调用 USB 集线器总线驱动程序的 AddDevice 例程来添加集线器设备，集线器驱动程序的 AddDevice 例程不会为一个子集线器调用（一个插入集线器的设备）。

2.3.1 编写 AddDevice 例程的指南

当编写一个 AddDevice 例程时，考虑下面的设计指南：

- 如果一个过滤器驱动程序决定它的 AddDevice 例程调用一个它无须服务的设备时，过滤器驱动程序必须返回 STATUS_SUCCESS 来允许剩余的设堆栈为设备而被装入。过滤器驱动程序没有生成一个设备对象，也不附着它到设备堆栈；过滤器驱动程序只是返回成功并允许其余的驱动程序添加到堆栈。
- 通常在一个设备对象的设备扩展里，对于它所使用的任何内核定义的对象和执行程序的自旋锁，一个驱动程序也必须提供特定对象的指针存储，这个对象从 I/O 管理器或其他的系统组件得到。

  你也许决定分配另外的系统空间内存来满足驱动程序的需要，比如，长期缓冲区或一个辅助列表。如果是这样的话，一个 AddDevice 例程能够调用下面的例程：

  - 分页或无分页系统空间内存的 **ExAllocatePool**
  - **ExInitializePagedLooksideList** 或 **ExInitializeNPagedLooksideList** 来初始化一个分页或

无分页的辅助列表

- 如果驱动程序有一个专为设备服务的线程或在任何内核定义的调度程序对象上等待时，它的 AddDevice 例程必须初始化调度程序对象。该调度程序对象的线程或驱动程序调用适当的 **KeInitialize***Xxx* 支持例程，该支持例程有一个事件指针、信号灯、互斥体和/或定时器等驱动程序提供存储的对象。

  由于它在一系统线程的环境里执行，一个 AddDevice 例程本身能够在一个调度程序对象上等待的一个非零间隔，只要调度程序对象在等待开始之前已经被初始化。

  参见内核模式驱动程序设计指南第 3 章的内核调度程序对象来获得更多信息。
- 如果驱动程序使用任何执行程序的自旋锁或为一个中断自旋锁提供存储，则 AddDevice 例程必须在传递它到任何别的支持例程之前，用每个这样的自旋锁调用 **KeInitializeSpinLock**。看内核模式驱动程序设计指南第 16 章的自旋锁使用来获得更多信息。
- 当调用 **IoCreateDevice** 时，加强文件打开安全。

  在调用 **IoCreateDevice** 时指定 FILE_DEVICE_SECURE_OPEN 特性，Windows2000 和 Windows NT 4.0 SP5 支持这个特性。它指令 I/O 管理器为所有打开的请求执行设备对象的安全检查。如果 FILE_DEVICE_SECURE_OPEN 特性没有在设备的类安装器 INF 或设备的 INF 上设置，并且驱动程序不执行它们自己打开请求的安全检查，在调用 **IoCreateDevice** 时厂商应该指定这个标记。

  在调用 **IoCreateDevice** 时，如果一驱动程序设置了 FILE_DEVICE_SECURE_OPEN 标记，I/O 管理器应用设备对象的安全描述符到任何相对的打开或尾部文件名打开。例如，如果 FILE_DEVICE_SECURE_OPEN 为\Device\foo 设置，并且如果\Device\foo 仅能够由管理员打开，那么，\Device\foo\abc 也能够由管理员打开。然而，I/O 管理器阻止一正常用户打开\Device\foo 和\Device\foo\abc。

  如果一个设备的驱动程序设置了 FILE_DEVICE_SECURE_OPEN。PnP 管理器传送它给设备的所有设备对象。

2.4 DispatchPnP 例程

PnP 管理器使用 IRP 来指导驱动程序启动、停止和删除设备并查询驱动程序的设备，所有的 PnP IRP 有主要的功能代码 IRP_MJ_PNP。

每个 PnP 驱动程序对处理特定的 IRP 是必要的，并能够选择地处理其余的 IRP。参见《Windows2000 驱动程序开发参考》的卷一，来获得关于每种驱动程序（功能驱动程序、过滤器驱动程序和总线驱动程序）哪一个 IRP 是必要的，哪一个 IRP 是可选择的。

驱动程序应该在一 *Xxx*DispatchPnp 例程里处理 PnP IRP，这里 *Xxx* 是一个用来标识驱动程序的前缀。在驱动程序初始化它的 **DriverEntry** 例程过程中，驱动程序在 **DriverObject->MajorFunction[IRP_MJ_PNP]**里设置它的 DispatchPnp 例程的地址。通过 I/O 管理器，PnP 管理器调用一个驱动程序的 DispatchPnp 例程。

支持一个设备的所有驱动程序必须有机会处理设备的 PnP IRP，除过在极个别情况下。（即一个功能或过滤器驱动程序允许 IRP 失效。）阅读第 2 部分获得关于 PnP IRP 的信息，包括何时 PnP IRP 被送出和它们的处理规则。

2.5 DispatchPower 例程

电源管理器使用 IRP 指示驱动程序来改变电源状态，来等待并响应系统唤醒事件，和查询驱动程序的设备。所有的电源 IRP 有主功能代码 IRP_MJ_POWER。

驱动程序应该在一个 *Xxx*DispatchPower 例程里处理电源 IRP，这里 *Xxx* 是一个用来标识驱动程序的前缀。在 **DriverObject->MajorFunction**[IRP_MJ_POWER]里，**DriverEntry** 例程设置驱动程序的 DispatchPower 例程的地址。当一个电源 IRP 被送出时，PnP 管理器调用驱动程序的 DispatchPower 例程。

DispatchPower 例程应该处理仅有主功能代码 IRP_MJ_POWER 的 IRP，该例程执行下面的任务：

- 如果可能，处理 IRP
- 使用 **PoCallDriver**，传递 IRP 给设备堆栈里下一个的低层驱动程序
- 如果是一个总线驱动程序，则执行设备上的电源操作请求和实现 IRP

一个设备的所有驱动程序必须有机会处理设备的电源 IRP，除过在极个别情况下。（即一个功能或过滤器驱动程序被允许使 IRP 失效。）大多数功能和过滤器驱动程序要么执行一些处理，要么为每个电源 IRP 设置一个 IoCompletion 例程，然后，向下传递 IRP 到下一个的低层驱动程序而不实现它。最后 IRP 到达总线驱动程序，如果需要，它物理地改变设备的电源状态并完成 IRP。

在完成 IRP 之后，当 IRP 向下传递到设备堆栈时，I/O 管理器调用任何由驱动程序设置的 IoCompletion 例程。是否一个驱动程序需要设置一个完成例程取决于 IRP 类型和单个的驱动程序需求。

加电一个设备的电源 IRP 必须首先由设备堆栈（基础总线驱动程序）里的最低层驱动程序处理，然后连续地被上面的堆栈里的每个驱动程序处理。断电一设备的电源 IRP 必须首先被顶部设备堆栈里的第一个驱动程序处理，然后连续地被下面的堆栈里的每个驱动程序处理。

可删除的设备的特别处理

在它们的 DispatchPower 例程里，可删除的设备的驱动程序应该检查是否设备仍旧存在。如果设备已被删除，驱动程序不应该向下传递 IRP 到下一个低层驱动程序，相反，驱动程序应该作下面的工作：

- 调用 **PoStartNextPowerIrp** 来开始处理下一个电源 IRP
- 设置 **Irp->IoStatus.Status** 给 STATUS_DELETE_PENDING
- 调用 **IoCompleteRequest**，指定 IO_NO_INCREMENT，完成 IRP
- 返回 STATUS_DELETE_PENDING

2.6 PnP 和电源管理 Unload 例程

一个 PnP 驱动程序必须有一个 Unload 例程来删除任何驱动程序专用的资源，如内存、线程和事件等，这些由 **DriverEntry** 例程产生。如果没有驱动程序专用的资源可以删除，驱动程序必须仍就有一个 Unload 例程，但是它能够简单地返回。

在所有的设备已被删除之后，可在任何时候调用驱动程序的 Unload 例程，PnP 管理器在系统线程 IRQL PASSIVE_LEVEL 的环境里调用一个驱动程序的 Unload 例程。

第二部分　即插即用

第1章　理解 PnP

本指南的第一部分包含一个即插即用（PnP）和电源管理的总览，并介绍了这两种技术中通用的许多术语和概念。本章包含另外的 PnP 背景、指南和操作信息来帮助驱动程序编写者理解 PnP 子系统。你必须遵照在此指南描述的和相随的参考手册中的 PnP 规则；本章给你理解这些规则的参考框架。

在阅读本章之前，你应该明白第一部分中描述的术语和概念（如功能驱动程序、总线驱动程序、FDO、PDO、过滤程序 DO、设备堆栈、设备树、devnode 和设备父/子关系）。

本章包括下面的部分：

1.1 PnP 驱动程序设计指南

如第一部分所描述的，PnP 提供：

- 自动和动态识别已安装的硬件
- 硬件资源的分配（和再分配）
- 适当的驱动程序安装
- 和 PnP 系统相互作用的驱动程序的接口
- 驱动程序和应用程序了解硬件环境的改变的机制

为了支持 PnP，驱动程序必须遵循下面的指南：

1. 必须遵循以设备为中心的 PnP 驱动程序模型。

   PnP 管理器作为一个整体管理系统上的设备，并调用驱动程序服务于这些单个的设备。例如，当驱动程序被装载时，一个 PnP 驱动程序不应该查找将要服务的潜在的设备。相反，当 PnP 管理器定位一个将要服务的驱动程序设备时，调用它的 AddDevice 例程。

   PnP 管理器为驱动程序将要服务的每个设备调用驱动程序的 AddDevice 例程，该驱动程序继续服务设备，直到 PnP 管理器指令它停止服务这个设备并删除它用于设备的数据结构。

   这个以设备为中心的模型能够让 PnP 管理器支持 PnP 机器上设备的动态到达和出发。

2. 必须遵循该文件里所描述的 PnP 规则。

   PnP 硬件必须遵照 PnP 硬件标准，PnP 驱动程序必须遵守 PnP 软件规则。例如，该文件描述了 PnP 驱动程序需要的例程和处理 PnP IRP 的规则。

   通过系统提供的端口或类驱动程序，一些驱动程序与 PnP 细节和电源系统的细节分开，这样

的驱动程序不需要实现所有的这些机制。例如，从电源和 PnP 系统的许多细节，一个 SCSI 端口驱动程序隔离一个 SCSI 微端口驱动程序，这样，一个 SCSI 微端口驱动程序无需直接处理电源和 PnP IRP。对这样的驱动程序，看驱动程序专用的文件来获得必要的 PnP 支持的细节。

3. 无需直接声明硬件资源。

   作为代替，一个 PnP 驱动程序应该给 PnP 管理器报告一个设备的资源需求，硬件设备资源包括 IRQ、I/O 端口、DMA 通道和设备内存区域，PnP 管理器比较设备的资源需求与系统上其他设备需求。PnP 管理器声明设备的资源并通知这些源给设备的驱动程序，它在 IRP_MN_START_DEVICE IRP 里为设备传达信息。

4. 必须是一个模块以便 PnP 管理器能在需要时调用它的例程。

   一个驱动程序例程，如它的 **DriverEntry** 添加例程和调度例程必须编写以便它们能够在需要时被 PnP 管理器调用，你不能够假设何时驱动程序例程在怎样的环境里被调用，除过被文档列出的。

5. 应该是灵活的。

   举例来说，如果一个设备能够在多于一套硬件资源上操作，驱动程序应该报告 PnP 管理器所有可能的硬件资源配置，以便 PnP 管理器在机器上配置设备时能够有最大的灵活性。

1.2 PnP 和设备树

如第一部分中所描述的，PnP 管理器维护一个叫设备树的信息集来记录机器上的设备踪迹。在系统引导过程中 PnP 管理器构建此树；当设备被添加或删除时，PnP 管理器更新此树。对机器上每个物理的、逻辑的和虚拟的设备来说，设备树包含一个设备节点（devnode）。

设备树是许多 PnP 管理器的操作系统的中心，你应该知道关于设备树的下面的内容：

- 设备树是分层次的。

  一个总线上的设备代表总线适配器或控制器的子代。一个总线设备是其他物理的、逻辑的或虚拟的设备能够附着的任何设备，设备树有它所需要的表示机器上所有设备的足够多的分级。当 PnP 管理器管理系统设备时，它使用设备树上层次的信息。例如，当一用户请求删除一个设备时，PnP 管理器标识所有的设备子代，由于它们必须一同被删除。

- 设备树由 PnP 管理器与 PnP 驱动程序共同构建。

  PnP 管理器构建并维护设备树，但是，它从驱动程序收集关于设备的信息。例如，PnP 管理器为它的子设备列表询问总线驱动程序，它的子设备使用了一个 P_MN_QUERY_DEVICE_RELATIONS IRP。

- 按照工业标准枚举设备。

  当 PnP 管理器询问它总线上设备一个总线驱动程序时，总线驱动程序依照它的总线协议决定它的子代列表。例如，ACPI 驱动程序寻找 ACPI 命名空间，PCI 驱动程序查询 PCI config 空间，USB 集线器驱动程序遵守 USB 总线协议。

- 除过一个总线设备在任何它的子设备之前被配置，驱动程序编写者不能够作任何关于设备树以何种次序被构建的假设。

  例如，驱动程序编写者不应该假设总线上的一个设备在其上的另一个设备之前配置。

- 设备树是动态的。

  当设备被添加到和删除机器时，PnP 管理器（与驱动程序一同）维护了系统上设备的一个当

前视图。

1.3 PnP 设备状态

在一个 PnP 系统上，当一个设备被配置、启动、可能的停止来重新平衡资源和可能的删除时，设备在多种 PnP 状态之间转换。本部分提供了 PnP 设备状态的高级总览，PnP 设备状态提供了驱动程序必须许多的 PnP 支持的一个路标，本文件的其余部分详细描述了每个状态的转换。

图 1.1 给出了设备的 PnP 状态和设备怎样从一状态转换到另一状态。

开始于图 1.1 的左上端，一个 PnP 设备物理地存在于系统中，由于或者用户刚好插入设备，或者设备在引导时存在。系统软件还不知道设备。

为了开始设备的软件配置，PnP 管理器和父总线驱动程序枚举设备。PnP 管理器，可能带有用户模式组件的帮助，标识设备的驱动程序，包括功能驱动程序和任何可选择的过滤器驱动程序。如果还没有装载驱动程序，PnP 管理器调用每个驱动程序的 **DriverEntry** 例程。看第 1 部分了解关于它的 **DriverEntry** 例程里初始化一个 PnP 驱动程序的信息 。看 1.4 部分获得更多关于报告和枚举一个 PnP 设备的信息。

一旦一个驱动程序被初始化，它必须准备初始化它的设备，PnP 管理器调用驱动程序所控制的每个设备的一个驱动程序的 AddDevice 例程。参看第 1 部分关于一个驱动程序的 AddDevice 例程的信息。

当一个驱动程序从 PnP 管理器接受到一 IRP_MN_START_DEVICE 时，驱动程序启动了设备并准备处理设备的 I/O 请求。看 3.1 部分获得处理一个 IRP_MN_START_DEVICE 请求的信息。

图 1.1 从 PnP 的角度所看到的设备状态

如果 PnP 管理器必须重新配置一个现用设备的硬件资源时，它给设备的驱动程序发出一个停止 IRP 的集合，指令它们临时停止设备。在它重新配置了硬件资源之后，PnP 管理器通过发出一个 IRP_MN_START_DEVICE 请求指令驱动程序重新启动设备，一个引导配置设备的驱动程序能够在设备启动之前接受到 IRP_MN_QUERY_STOP_DEVICE 和 IRP_MN_STOP_DEVICE 请求。（这一步在图 1.1 中没有给出。）看 3.2 部分获得处理停止 IRP 的信息。

当一个 PnP 设备从系统中物理地删除，或者已被删除时，PnP 管理器给设备驱动程序发出各种删除 IRP 指令，命令它们删除设备的软件表示（设备对象等等）。看 3.3 部分获得处理删除 IRP 的信息。

在一个驱动程序所有的设备已被删除之后，PnP 管理器卸载该驱动程序。（卸载步骤在图 1.1 中没有给出）看第一部分获得 Unload 例程的信息。

1.4 动态地添加一个新的 PnP 设备

这部分描述了当系统配置一个用户已添加到一运行的机器上的 PnP 设备时，所发生的一系列事件。这个讨论突出了在枚举和配置一新设备时，PnP 管理器、总线驱动程序、功能和过滤器驱动程序的任务。

这个讨论的大部分也与配置一个 PnP 设备有关，在机器引导时该 PnP 设备已经存在。特别地，

在它们的 INF 里，标记为 SERVICE_DEMAND_START 的设备的驱动程序以一种基本相同的方式配置，而不管该设备是否被动态添加或者在引导时已经存在。

图 1.2 给出了配置设备第一步，从用户给机器插入硬件时开始。

图 1.2 枚举和报告一个 PnP 设备

下面的注解对应于图 1.2 中的被圈起来的数字：

1. 用户插入一个 PnP 设备到 PnP 总线上的一个空槽。
   在这个例子中，用户插入一个 PnP USB 操纵杆到一个 USB 主控制器上的集线器，由于 USB 集线器有子设备与其联接，所以 USB 集线器是一个 PnP 总线设备。
2. 总线设备的功能驱动程序决定一新设备在其总线上。
   驱动程序怎么决定这种对总线结构的依赖呢？对一些总线，总线功能驱动程序接受到一个新设备的热插拔通知，如果总线不支持热插拔通知，用户必须在对话框或小应用程序里采取适当的动作以便枚举总线。
   在该例中，USB 总线支持热插拔通知，这样，支持 USB 总线的功能驱动程序被通知其子设备已经改变。
3. 总线设备的功能驱动程序通知 PnP 管理器它的一套子设备已被改变。
   功能驱动程序通过调用具有一个 **BusRelations** 类型的 **IoInvalidateDeviceRelations** 来通知 PnP 管理器。
4. PnP 管理器查询总线驱动程序当前总线上的设备列表。
   PnP 管理器发送一个 IRP_MN_QUERY_DEVICE_RELATIONS IRP 到总线的设备堆栈，**Parameters.QueryDeviceRelations.Type** 是 **BusRelations**，表明 PnP 管理器正请求存在于总线上设备的当前列表。
   PnP 管理器发送 IRP 到总线的设备堆栈里的顶端驱动程序，按照 PnP IRP 的规则， 堆栈里的每个驱动程序处理 IRP，如果合适，则向下传递 IRP 到紧邻的驱动程序。
5. 总线设备的功能驱动程序处理 IRP。
   参见《Windows2000 驱动程序开发参考》卷一参考页 IRP_MN_QUERY_DEVICE_RELATIONS 来获得关于处理 IRP 的详细信息。
   在此例中，USB 集线器驱动程序处理集线器 FDO 的这个 IRP，集线器驱动程序为操纵杆设备生成一个 PDO，并在它随 IRP 返回的子设备列表里包括一个操纵杆 PDO 的引用指针。

   当 USB 集线器的父总线驱动程序（USB 主控制器类/微类驱动程序对）完成 IRP，IRP 经由任何被集线器驱动程序注册的完成例程返回设备堆栈。
   注意到通过请求 PnP 管理器查询它的子设备的列表，总线功能驱动程序报告其子列表的一个变化，结果的 IRP_MN_QUERY_DEVICE_RELATIONS IRP 被总线设备的所有驱动程序所发现。典型地，总线功能驱动程序是处理 IRP 和报告子设备的唯一驱动程序。在一些设备堆栈里，一个总线过滤器驱动程序存在并参与构建 **BusRelations** 列表。一个例子是 ACPI，其作为一个 ACPI 设备的总线过滤器驱动程序来附着。在一些设备堆栈里，非总线过滤器驱动程序处理 IRP_MN_QUERY_DEVICE_RELATIONS IRP，但这不是典型的处理。
   在这一点上，PnP 管理器有总线上设备的当前列表，然后，PnP 管理器判断是否设备是新近到达的或已经被删除。在本例中，有一个新设备，图 1.3 给出了 PnP 管理器为新设备生成一个 devnode 并开始配置设备。

图 1.3 为一个新的 PnP 设备生成一个 devnode

下面的注释对应于图 1.3 中的被圈起来的数字：

6. PnP 管理器为总线上任何新的子设备生成 devnode。

   PnP 管理器比较在 IRP_MN_QUERY_DEVICE_RELATIONS IRP 里返回的 **BusRelations** 列表与当前记录在 PnP 设备树上的子设备列表，该总线的 PnP 管理器为每个新设备生成一个 devnode 并为任何已被删除的设备初始化删除处理。

   在本例中，有一个新设备（操纵杆），于是 PnP 管理器为操纵杆生成一个 devnode。在这一点上，为操纵杆配置的唯一的驱动程序是父 USB 集线器总线驱动程序，它生成一个操纵杆 PDO。任何可选择的总线过滤器驱动程序也应该存在于设备堆栈里，但为了简单，本例忽略了总线过滤器驱动程序。

   如第 1 部分图中所介绍的，图 1.3 中两个 devnode 之间的宽箭头表示了操纵杆 devnode 是 USB 集线器的 devnode 的子代。

7. PnP 管理器收集新设备的信息并开始配置设备。

   PnP 管理器给设备堆栈发送一系列的 IRP 来收集关于设备的信息，在这一点上，设备堆栈包括由设备父总线驱动程序生成的唯一的 PDO 和任何可选择的总线过滤器驱动程序的过滤程序 DO。于是，总线驱动程序和总线过滤器驱动程序是响应这些 IRP 唯一的驱动程序。此例中，操纵杆设备堆栈里唯一的驱动程序是父总线驱动程序，即 USB 集线器驱动程序。

   PnP 管理器通过给设备堆栈发送 IRP 来收集有关新设备的信息，这些 IRP 包括下面的内容：

   - IRP_MN_QUERY_ID，下面的每个设备 ID 分别有一个 IRP：
     BusQueryDeviceID
     BusQueryInstanceID
     BusQueryHardwareIDs
     BusQueryCompatibleIDs
   - IRP_MN_QUERY_CAPABILITIES
   - IRP_MN_QUERY_DEVICE_TEXT，下面的每个条目分别有一个 IRP：
     DeviceTextDescription
     DeviceTextLocationInformation
   - IRP_MN_QUERY_BUS_INFORMATION
   - IRP_MN_QUERY_RESOURCES
   - IRP_MN_QUERY_RESOURCE_REQUIREMENTS

   为获得处理这些 IRP 的信息，参看《Windows2000 驱动程序开发参考》的卷一参考页中的 IRP。

   在处理一个新的 PnP 设备过程中，PnP 管理器发送上面列出的 IRP，但不需要按照所列的次序，于是，你不应该作 IRP 以怎样的次序被发送的假设。而且，你不应该假设 PnP 管理器仅仅发送上面列出的 IRP。

   PnP 管理器检查注册表来决定是否设备以前已被安装到机器上。PnP 管理器为在 **Enum** 分支下的设备检查一个<*enumerator*>\<*deviceID*>的子键。在本例中，设备是新的并必须“从头开始”配置。

8. PnP 管理器在注册表里存储关于设备的信息。

   注册表 **Enum** 分支为操作系统组件的使用而维护，并且，其布局易于改变。驱动程序编写者

必须使用 API 系统来释放驱动程序相关的信息，不要直接从一个驱动程序访问 **Enum** 分支。

下面的 **Enum** 信息仅为调试的目的而列出。

■ PnP 管理器为键下的设备生成一个子键来支持设备的计数器。

PnP 管理器生成一个名叫 **HKLM\System\CurrentControlSet\Enum**<*enumerator*>\<*deviceID*>的子键，如果它事先并不存在，它生成<*enumerator*>子键。

一个计数器是一个组件，该组件发现基于 PnP 硬件标准的 PnP 设备。在微软的 Windows2000 上，一个计数器的任务由 PnP 总线驱动程序与 PnP 管理器共同执行。一设备典型地被它的父总线驱动程序枚举，如 PCI 或 PCMCIA。一些设备由总线过滤器驱动程序枚举，如 ACPI。

■ PnP 管理器为该设备的这个实例生成一个子键。

如果 **Capabilities.UniqueID** 在 IRP_MN_QUERY_CAPABILITIES 里返回 TRUE，设备的单个 ID 在系统中是独一无二的。如果没有，PnP 管理器修改 ID 以便它在整个系统里是独一无二的。

PnP 管理器生成一个名叫 **HKLM\System\CurrentControlSet\Enum**<*enumerator*>\<*deviceID*>\<*instanceID*>的子键。

■ PnP 管理器写关于设备的信息到设备实例的子键。

如果提供 PnP 管理器给设备，它存储包括下面的信息：

DeviceDesc---从 IRP_MN_QUERY_DEVICE_TEXT 而来

Location--- 从 IRP_MN_QUERY_DEVICE_TEXT 而来

Capabilities--- 从 IRP_MN_QUERY_CAPABILITIES 而来的标记

UINumber--- 从 IRP_MN_QUERY_CAPABILITIES 而来

HardwareID--- 从 IRP_MN_QUERY_ID 而来

CompatibleIDs--- 从 IRP_MN_QUERY_ID 而来

LogConf\BootConfig--- 从 IRP_MN_QUERY_RESOURCES 而来

LogConf\BasicConfigVector--- 从 IRP_MN_QUERY_RESOURCE_REQUIREMENTS 而来

不要在一个驱动程序里直接访问这些注册表键！它们被列出仅出于调试的目的。

在内核模式里，你用 **IoGetDeviceProperty** 例程或 IRP_MN_QUERY_CAPABILITIES IRP 访问这个信息。在用户模式里，你通过 **SetupDi***Xxx* 函数（设备安装程序函数）或 **CM***Xxx* 函数（配置管理器函数）访问这个信息。

在这一点上，PnP 管理器准备为设备定位功能驱动程序和过滤器驱动程序，如果有的话。（看图 1.4。）

图 1.4 寻找功能和过滤器驱动程序

下面的注释对应于图 1.4 中被圈起来的数字：

9. 内核模式 PnP 管理器与用户模式 PnP 管理器协同一致，也与用户模式安装组件协同一致来发现设备的功能和过滤器驱动程序（如果有的话）。

   内核模式 PnP 管理器排队用户模式 PnP 管理器的一个事件，并标识一个需要安装的设备。一旦一个有特权的用户登陆，用户模式组件处理发现的驱动程序。参看第 4 部分第 1 章获得安装组件和在安装设备时它们的角色信息。

10. 用户模式安装组件指示内核模式 PnP 管理器来装载功能和过滤器驱动程序。

    用户模式组件回调内核模式来使驱动程序装载，并调用驱动程序 AddDevice 例程。

    图 1.5 给出了 PnP 管理器装载驱动程序（如果合适）、调用它们的 AddDevice 例程和指示驱动程序启动设备。

图 1.5 调用 AddDevice 例程并启动新设备。

下面的注释对应于图 1.5 中的被圈起来的数字：

11. 低层过滤器驱动程序

在功能驱动程序附着到设备堆栈之前，PnP 管理器处理任何任何低层过滤器驱动程序。对每个低层过滤器驱动程序而言，如果驱动程序还没装载，PnP 管理器调用驱动程序的 **DriverEntry** 例程。然后，PnP 管理器调用驱动程序的 AddDevice 例程。在它的 AddDevice 例程里，过滤器驱动程序生成一个过滤程序 DO 并附着它到设备堆栈（**IoAttachDeviceToDeviceStack**），一旦它附着它的设备对象到设备堆栈，此驱动程序能够作为此设备的驱动程序。

在 USB 操纵杆例子中，有设备的一个低层过滤器驱动程序。

12. 功能驱动程序

在任何低层的过滤器驱动程序被附着后，PnP 管理器处理功能驱动程序。如果驱动程序还没装载，PnP 管理器调用功能驱动程序的 **DriverEntry** 例程并调用功能驱动程序的 AddDevice 例程。功能驱动程序生成一个 FDO 并附着它到设备堆栈。

在本例中，USB 操纵杆的功能驱动程序实际上是一对驱动程序：HID 类驱动程序和 HID 微类驱动程序。这两个驱动程序作为功能驱动程序一起工作，该驱动程序对仅生成一个 FDO 并附着它到设备堆栈。

13. 顶层过滤器驱动程序

在功能驱动程序被附着后，PnP 管理器处理任何顶层过滤器驱动程序。

在本例中，有设备的一个顶层过滤器驱动程序。

14. 分配资源和启动设备

PnP 管理器给设备分配资源，如果需要，发布一个 IRP 来启动设备。

- 分配资源

在配置进程的早期，PnP 管理器为从设备的父总线驱动程序而来的设备收集硬件资源需求。在全套的驱动程序为设备而装载后，PnP 管理器发送一个 IRP_MN_FILTER_RESOURCE_REQUIREMENTS IRP 到设备堆栈。在堆栈里的所有驱动程序有机会处理这个 IRP 并修改设备的资源需求列表，如果需要的话。

如果设备需要，基于设备的需求和当前可用的资源，PnP 管理器分配资源到设备。

PnP 管理器也许需要重新安排现存的设备的资源分配来满足新设备的需要，这个资源的再分配被叫作“重新平衡”。 在一个重新平衡过程中，现存的设备的驱动程序收到一系列停止和启动 IRP，但是，重新平衡对用户必须是透明的。

在这个 USB 操纵杆的例子中，USB 设备不需要硬件资源，于是，PnP 管理器设置资源列表为 NULL。

- 启动设备（IRP_MN_START_DEVICE）

一旦 PnP 管理器分配资源给设备，它发送一个 IRP_MN_START_DEVICE IRP 到设备堆栈来指令驱动程序启动设备。为了获得关于启动设备的详细信息，参见本指南的第 3 章，并参见《Windows2000 驱动程序开发参考》卷一参考页的 IRP_MN_START_DEVICE。

在设备启动之后，PnP 管理器发送三个另外的 IRP 到设备的驱动程序：

- IRP_MN_QUERY_CAPABILITIES

在启动 IRP 成功完成之后，PnP 管理器发送另外的 IRP_MN_QUERY_CAPABILITIES IRP 到

设备堆栈。设备的所有的驱动程序都有处理 IRP 的选项，在当所有的驱动程序被附着且设备启动之后，由于功能和过滤器驱动程序应该需要访问设备来收集能力信息，所以在这个时候，PnP 管理器发送这个 IRP。

- IRP_MN_QUERY_PNP_DEVICE_STATE

举个例子，这个 IRP 给一个驱动程序机会，来报告设备不应该在用户界面里显示，例如设备管理器和热插拔小程序。这对存在于系统但在当前配置中不能使用的设备而言是非常有用的，例如一个便携机的游戏端口，当便携机计算机未连接时，该便携计算机是不可用的。

- **BusRelations** 的 IRP_MN_QUERY_DEVICE_RELATIONS

PnP 管理器发送这个 IRP 来决定是否该设备有任何子设备，如果有的话，PnP 管理器配置每个子设备。

1.5 硬件资源

硬件资源是可分配的、有地址的总线路径，该路径允许外设和系统处理器相互通信。硬件资源典型地包括 I/O 端口地址、中断向量和总线相关的内存地址块。在系统能够与一个设备实例通信之前，PnP 管理器必须给设备实例分配硬件资源，该设备实例基于哪些资源可用和那些设备实例能够使用的信息。资源被分配给设备树上的（假设代表的设备需要资源且这些资源可用）每一个设备节点，PnP 管理器使用列表来跟踪和每个设备节点相关联的硬件资源，它使用两种类型的列表：

- 资源需求列表

设备被典型地设计以便在资源分配的范围内操作。举例来说，一个设备可能仅需要一个中断向量，但是，它能够使用向量范围内的任何一个。对每个设备实例来说，PnP 管理器维护一个资源需求列表，这个资源需求列表指定了设备可在其内操作的硬件资源的所有范围。列表的名字起源于以下的事实：即当分配它们给设备时，PnP 管理器必须从这个列表选择资源。

内核模式代码使用结构 IO_RESOURCE_REQUIREMENTS_LIST（或者当输入到系统例程时，或者响应 IRP 时）指定了资源需求列表。当输入到 PnP 配置管理器功能时，通过使用 PnP 配置管理器结构，用户模式代码指定了资源需求列表。

- 资源列表

当 PnP 管理器给一设备分配资源时，它通过生成一个已分配给每个设备实例的列表，来跟踪这些分配。这些列表可被称作资源分配列表，但是，那个名字典型地被缩写为资源列表。当设备被从系统添加或删除时，PnP 管理器能够改变资源列表项并且资源在随后被再分配。（资源也能够被一个 PnP BIOS 所分配，并且，安装软件---使用 INF 文件或者用户输入---能够迫使 PnP 管理器给一个设备指定特定的资源。）

通过使用结构 CM_RESOURCE_LIST（或者当输入到系统例程时，或者响应 IRP 时），内核模式代码指定了资源列表。当输入到 PnP 配置管理器功能时，用户模式代码指定了使用 PnP 配置管理器结构的资源需求列表。

PnP 管理器在注册表里存储资源需求列表和资源列表，使用 REGEDT32 可察看它们。驱动程序能够通过即插即用例程和即插即用的 IRP 而不直接地访问这些列表，用户模式应用程序能够使用 PnP 配置管理器功能。（由于存储格式在将来的版本中易于改变，驱动程序和应用程序不应该直接地访问这些使用注册表功能的列表。）

1.5.1 逻辑配置

资源需求列表和资源列表被用来说明逻辑配置，每一个逻辑配置标识了或者可接受的资源范围或者特定设备实例的一套特定资源，且包含或者单个的资源需求列表或者单个的资源列表。此外，每个支持设备实例的逻辑配置属于一个逻辑配置类型，配置类型在下表里列出，几个有相同或不同类型的逻辑配置可能被指定到每个设备实例。

1.5.1.1 支持资源需求列表的逻辑配置类型

基本配置

一个资源需求列表标识了由即插即用的设备提供的资源范围，当一个驱动程序接受到 IRP_MN_QUERY_RESOURCE_REQUIREMENTS IRP 时，驱动程序应该返回这个列表。（非 PnP 设备的基本配置能够在一个 INF 文件里被描述，这时，设备安装软件读 INF 文件并调用 PnP 配置管理器功能来生成一个需求列表。）

过滤配置

一个已提供给驱动程序设备堆栈的资源需求列表可能地被修改，且随后被驱动程序堆栈返回，来响应 IRP_MN_FILTER_RESOURCE_REQUIREMENTS IRP。PnP 管理器使用结果的过滤程序配置作为分配资源的基础。

重载配置

一个资源需求列表重载了基本的配置，典型地，如果设备的 INF 文件包含一个带有扩展的 **LogConfigOverride** 的 *DDInstall* 节，则一个设备安装程序生成一个重载配置。（在《Windows 2000 驱动程序开发参考》的卷一里参看 *INF LogConfig Directive*。）如果它的设备物理地从系统中删除，一个重载配置不被删除。

1.5.1.2 资源列表的逻辑配置类型

引导配置

当系统被引导时，一个资源列表标识了分配给设备实例的资源。（对 PnP 设备，这是由 BIOS 提供的配置；对非 PnP 设备，这些资源应该被卡上的跳线所选择。）当一个驱动程序接受到 IRP_MN_QUERY_RESOURCES IRP 时，它应该返回这个资源列表。（如果 BIOS 不能够决定一个设备使用的所有资源时，一个引导配置能够部分的空缺。）如果一个设备被删除或重新启动，PnP 管理器能够修改这个列表。对非 PnP 设备，这个配置类型能够代替一个强迫配置，在这种情况下，它比等价的强迫配置有较低的配置优先级，对每个设备实例来说，只有一个引导配置能够存在。

强迫配置

一个资源列表标识了一个设备实例必须使用的资源，一个强迫配置阻止 PnP 管理器指定另外的资源给设备实例。一个设备安装程序可能生成一个强迫配置，这种配置基于或者包含在一个 INF 里或者从用户得到的信息。如果强迫配置的设备物理地从系统中删除，强迫配置不被删除。对每个设备实例来说，只有一个强迫配置存在。

分配的配置

一个资源列表标识了设备实例当前在用的资源。仅有一个分配的配置能够为每一个设备实例而存在。

设备驱动程序负责决定一个 PnP 兼容的设备的基本配置、过滤配置和引导配置，并返回信息来响应由 PnP 管理器发送的 IRP。（为了更多的信息，请看第 2 章。）驱动程序安装软件能够生成重载配置、强迫配置和对非 PnP 设备的引导配置。PnP 管理器维护每个设备实例的分配配置。

当每个配置生成时，一个优先权指定给每个配置，如果 PnP 管理器发现一个设备实例已被指定了几个同类的逻辑配置，它尽量使用具有最高优先级的一个。如果那个配置导致资源冲突，它尝试具有次优先级的配置。（欲知配置优先权的列表，参看《Windows2000 驱动程序开发参考》的卷一中的 **Cm_Add_Empty_Log_Conf**。）

1.6 在驱动程序里使用 GUID

驱动程序和其他系统组件使用 GUID（globally unique identifiers）来标识不同的条目，系统组件为诸如设备安装类、PnP 事件、WMI 事件和静止图象事件等条目定义 GUID。你能够定义你自己一些条目的 GUID，如设备接口类、定制 PnP 事件和定制 WMI 事件。驱动程序和应用程序包含头文件，这些头文件定义了它们使用的 GUID。

本节描述了下面的内容：

1.1.1 定义和导出新的 GUID

1.1.2 在驱动程序代码里包含 GUID

参看平台 SDK 来获得在用户模式应用程序使用 GUID 的信息。

1.6.1 定义和导出新的 GUID

为驱动程序导出到其他系统组件、驱动程序或者应用程序的条目定义一个新的 GUID，举个例子，你为其上的一个设备定义了定制 PnP 事件的一个新的 GUID。为了定义和导出一个新的 GUID，你必须做下面的事情：

1. 为 GUID 选择一个符号名。

选择一个表示了 GUID 目的的名字。举个例子，操作系统使用这样的名字如 GUID_BUS_TYPE_PCI 和 PARPORT_WMI_ALLOCATE_FREE_COUNTS_GUID。

2. 使用 *uuidgen.exe* 或 *guidgen.exe* 为 GUID 生成一个值。

这些应用程序生成一个独一无二的、代表一个 128 位值的格式化的信息串，在 *uuidgen.exe* 里的-s 开关输出了格式化为一个 C 结构的 GUID。

3. 在适当的头文件里定义 GUID。

使用 DEFINE_GUID 宏将 GUID 符号名与它的值相联。（参看例 1。）

例 1：在一个仅有 GUID 的头文件里定义一个 GUID

```
:
DEFINE_GUID( GUID_BUS_TYPE_PCMCIA,  0x9343630L,  0xaf9f,  0x11d0,
      0x92,  0x9f,  0x00,  0xc0,  0x4f,  0xc3,  0x40,  0xb1  );
DEFINE_GUID( GUID_BUS_TYPE_PCI,  0xc8ebdfb0L,  0xb510,  0x11d0,
      0x80,  0xE9,  0x00,  0x00,  0xf8,  0x1e,  0x1b,  0x30  );
```

:

如果 GUID 在一个不只有 GUID 定义的头文件里被定义，你必须采取一个额外的步骤确保 GUID 在包含头文件的驱动程序里被实例化。DEFINE_GUID 陈述必须发生在任何#ifdef 语句之外，该语句阻止了多次的引用。否则，如果头文件包括在预编译的头里，GUID 在使用头文件的驱动程序里将不被实例化。在一个混合的头文件里参看例 2 关于 GUID 的定义。

例 2：在一个混合的头文件里定义 GUID

```
#ifndef   _NTDDSER_          //这个 ex.从一系列 .h 的驱动程序文件里得到
#define   _NTDDSER_
:
//在这里放置其他的头文件定义
:
#endif    //    _NTDDSER_
#ifdef   DEFINE_GUID         // 不要中断驱动程序的编译，这些驱动程序包含
                             //这个头文件但不需要 GUID
//
//在多次引用保护之外放置 GUID 定义

DEFINE_GUID ( GUID_CLASS_COMPORT,   0x86e0d1e0L,   0x8089,   0x11d0,
        0x9c,    0xe4,   0x08,   0x00,   0x3e,   0x30,   0x1f,   0x73 );

DEFINE_GUID ( GUID_SERENUM_BUS_ENUMERATOR,   0x4D36E978,   0xE325,
        0x11CE,   0xBF,   0xC1,   0x08,   0x00,   0x2B,   0xE1,   0x03,   0x18 );

:
#endif    // DEFINE_GUID
```

由于 DEFINE_GUID 定义 GUID 为一个 EXTERN_C 变量，在阻止多次调用的语句之外放置一个 GUID 定义，这个过程没有在一个驱动程序里引起多次 GUID 实例。只要类型匹配，EXTERN 变量的多次声名是被允许的。

1.6.2 在驱动程序代码里包含 GUID

为了在内核模式驱动程序里使用 GUID，必须做两件事情：

1. 包含头文件 initguid.h,它重新定义了 DEFINE_GUID 宏。

宏 initguid.h 重新定义了 DEFINE_GUID 宏来实例化 GUID（不要简单地声明一个 EXTERN 引用）。你应该在驱动程序源文件里包含这个头文件，在那里 GUID 将被实例化。（用户模式应用程序在包括含有 GUID 定义的头文件之前包含<objbase.h>）。

2. 包含定义了 GUID 的头文件。

在包含 initguid.h 的语句之后，包含有 GUID 定义的头文件。一个驱动程序可能包含不止一个的含有 GUID 定义的头文件，包括系统提供的头文件和第三方的头文件。

下面的代码给出了包含 GUID 语句的序列：

```
:
//包含系统头文件如 wdm.h

#include <initguid.h>

//包含系统和驱动程序专用的头文件，这些头文件包含 GUID 定义。
:
```

你将上面的语句放置到一个驱动程序的模块里：即典型的主模块，当上面的语句存在时，驱动程序使用它的符号名来引用一个 GUID。

第2章　处理即插即用 IRP 的规则

这一章描述了驱动程序必须作什么来处理即插即用的（PnP）IRP。

参见《Windows2000 驱动程序开发参考》的卷一中关于每个 PnP IRP 的参考内容，参见第 3 章关于处理启动、停止和删除 IRP 的扩展信息。

在第 1 部分中阅读 PnP 概念、PnP 术语和一个 PnP 驱动程序全面的需求信息，参见本部分的第 1 章中关于 PnP 设备状态的描述。

2.1 PnP IRP 需求

PnP 管理器使用 IRP 来指示驱动程序启动、停止和删除设备并查询驱动程序的设备，所有的 PnP IRP 有主功能码 IRP_MJ_PNP。当 PnP 管理器发送一个 IRP 时，它初始化 **Irp->IoStatus.Status** 为 STATUS_NOT_SUPPORTED。

PnP 驱动程序必须遵循下面的这些规则：

1. 一个驱动程序必须有一个处理 PnP IRP 的调度例程。

驱动程序在 **DriverEntry** 例程的 **DriverObject->MajorFunction[IRP_MJ_PNP]**里设置例程地址，这个调度例程是一个独立的仅处理 PnP IRP 的 DispatchPnP 例程，或者驱动程序能够在单个的调度例程里处理几个 IRP_MJ_*Xxx* 代码。

2. 除非一个功能或过滤器驱动程序处理 IRP 和遇到一个错误（例如资源不足），它必须向下传递 PnP IRP 到设备堆栈里的下一个驱动程序。

除非其中的一个驱动程序遇到一个错误，一个设备的所有驱动程序必须有机会处理设备的 PnP IRP。PnP 管理器发送 IRP 给设备堆栈里顶端的驱动程序，功能和过滤器驱动程序向下传递 IRP 给下一个驱动程序，而父驱动程序完成 IRP。参见 2.2 节来获得更多信息。

如果驱动程序尝试处理 IRP 并且遇到一个错误（如资源不足）时，它可能使 IRP 失效。如果一个驱动程序收到带有它所不能处理的代码的一个 IRP 时，该驱动程序不能够使 IRP 失效，它必须向下传递这样一个 IRP 给下一个驱动程序而无需修改此 IRP 的状态。

3. 一个驱动程序必须处理特定的 PnP IRP 和可选地处理其他的 PnP IRP。

每一个 PnP 驱动程序必须处理特定的 IRP，比如 IRP_MN_REMOVE_DEVICE，且每个驱动程序能够可选地处理其他的 IRP。参见《Windows2000 驱动程序开发参考》的卷一获得有关信息，即关于对每一类型的驱动程序（功能驱动程序、过滤器驱动程序和总线驱动程序），哪一个 IRP 是必要的，哪一个 IRP 是可选的。

一个驱动程序可能使带有一个适当的错误状态的 PnP IRP 失效，但是一个驱动程序不应该为这样一个 IRP 返回 STATUS_NOT_SUPPORTED。

4. 如果一个驱动程序成功地处理了一个 PnP IRP，该驱动程序设置 IRP 状态为成功，它不依靠堆栈里另外的驱动程序来设置状态。

一个驱动程序设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS，以便通知 PnP 管理器驱动程序已经成功地处理了 IRP。对某些 IRP，一非总线驱动程序也许能够依靠它的父总线驱动程序设置状态为成功。然而，这是一个冒险的实践，为一致性和健壮性起见，对驱动程序已经成功处理

的每个 PnP IRP，一个驱动程序必须设置 IRP 状态为成功。

5. 如果一个驱动程序使 IRP 失效，驱动程序以一个错误状态完成 IRP，并且不向下传递 IRP 到下一个驱动程序。

为了使一个象 IRP_MN_QUERY_STOP 的 IRP 失效，一驱动程序设置 **Irp->IoStatus.Status** 为 STATUS_UNSUCCESSFUL。其余的 IRP 的额外错误状态值包括 STATUS_INSUFFICIENT_RESOURCES 和 STATUS_INVALID_DEVICE_STATE。

驱动程序没有为它们处理的 IRP 设置 STATUS_NOT_SUPPORTED，这是由 PnP 管理器设置的初始状态。如果一个 IRP 在此状态下完成，这意味着没有驱动程序在堆栈里已经处理了 IRP；所有的驱动程序只是传递 IRP 给下一个驱动程序。

6. 一个驱动程序必须在它的调度例程里（在 IRP 向设备堆栈下面的方向），在一个 IoCompletion 例程里（在 IRP 向设备堆栈后上面的方向），或者两个都有，来处理 PnP IRP，如 IRP 的参考页所提及的。

在带有 IRP_MN_REMOVE_DEVICE 的情况下，一些 PnP IRP 必须首先被设备堆栈顶层的驱动程序所处理，然后被每一个下一个较底层的驱动程序所处理。在带有 IRP_MN_START_DEVICE 的情况下，一些 IRP 必须首先由父总线驱动程序处理，在带有 IRP_MN_QUERY_CAPABILITIES 的情况下，其余的 IRP 仍旧按设备堆栈次序和反向次序被处理。参看《Windows2000 驱动程序开发参考》的卷一来获得运用每个 PnP IRP 的规则，看 2.3 节获得处理 PnP IRP 的信息，这个 PnP IRP 必须首先由父总线驱动程序处理。

7. 一个驱动程序必须按 IRP 设备堆栈次序添加信息给一个 IRP，并按 IRP 反向次序修改或删除信息。

当响应一个 PnP 查询 IRP 的信息返回时，一个驱动程序必须遵守这个规范以便能够通过设备的分层的驱动程序使信息被有续地传送。

8. 除过在明确被说明的地方，一个驱动程序不应该依靠以任何特定的次序发送的 PnP IRP。

9. 当一个驱动程序发送一个 PnP IRP 时，它必须发送该 IRP 到设备堆栈里的顶端驱动程序。

大多数 PnP IRP 由 PnP 管理器发送，但是一些能够由驱动程序发送（例如，IRP_MN_QUERY_INTERFACE）。一个驱动程序必须发送一个 PnP IRP 给设备堆栈顶端的驱动程序，调用 **IoGetAttachedDeviceReference** 来得到设备堆栈顶端的驱动程序的设备对象的一个指针。

你应该用操作系统的检查构建测试你的驱动程序，Windows2000 的检查构建核实一个驱动程序是否遵照上面列出的许多 PnP 规则。

2.2 沿设备堆栈传递 PnP IRP

除非设备堆栈里的一个驱动程序使 IRP 失效，一个设备的所有驱动程序必须有机会响应一个 PnP IRP。（看图 2.1。）

没有设备的单个驱动程序能够假设它是唯一可响应一个 PnP IRP 的驱动程序。例如，一个功能驱动程序响应一个 IRP_MN_QUERY_CAPABILITIES 请求并完成 IRP 而没有传递它给下一个低层驱动程序。没有能力被低层驱动程序所支持，比如，一个唯一的实例 ID 或由父总线驱动程序支持的电源管理能力被报告。

当父总线驱动程序调用 **IoCompleteRequest**，I/O 管理器调用任何由功能驱动程序或过滤器驱动程序注册的 IoCompletion 例程时，一个 PnP IRP 沿设备堆栈返回。

图 2.1 向下传递 PnP IRP 给设备堆栈

当一个功能或过滤器驱动程序接收到一个 PnP IRP 时，它必须做下面的工作：

■ 如果驱动程序执行动作来响应 IRP：

1. 执行适当的动作。
2. 设置适当的 **Irp->IoStatus.Status** 状态，比如为 STATUS_SUCCESS，如果对 IRP 适当的话，设置 **Irp->IoStatus.Information**。
3. 用 **IoSkipCurrentStackLocation** 或 **IoCopyCurrentStackToText** 确定下一个堆栈位置。如果你设置了一个 IoCompletion 例程，调用后面的例程。
4. 如有必要，设置一个 IoCompletion 例程。
5. 不要完成 IRP（不要调用 **IoCompleteRequest**），父总线驱动程序将完成该 IRP。

■ 如果驱动程序没有执行这个 IRP 动作，它简单地准备传递 IRP 给下一个驱动程序：

1. 调用 **IoSkipCurrentStackLocation** 从 IRP 删除它的堆栈位置。
2. 不在 **Irp->IoStatus** 里设置任何域。
3. 不设置一个 IoCompletion 例程。
4. 不要完成 IRP（不要调用 **IoCompleteRequest**），父总线驱动程序将完成该 IRP。

如果一个功能或过滤器驱动程序在以前没有使 IRP 失效，它传递 IRP 给下一个具有 **IoCallDriver** 的低层驱动程序。一个驱动程序有下一个低层驱动程序的一个指针；该指针由高层的驱动程序的添加例程里的 **IoAttachDeviceToDeviceStack** 调用返回。

在执行任何任务来响应 IRP 后，父总线驱动程序完成 IRP。在总线驱动程序调用 **IoCompleteRequest** 之后，I/O 管理器调用由设备的功能或过滤器驱动程序所注册的任何 IoCompletion 例程。

2.3 延迟 PnP IRP 的处理直到低层的驱动程序完成

一些 PnP 和电源 IRP 必须首先被一设备的父总线驱动程序处理，然后被设备堆栈里每个紧邻的高层驱动程序处理。例如，父总线驱动程序必须是执行它的设备的启动操作（IRP_MN_START_DEVICE）的第一个驱动程序，紧接着被下一个高层的驱动程序处理。对这样的一个 IRP，功能和过滤器驱动程序必须设置一个 I/O 完成例程，传递 IRP 给下一个低层驱动程序，并延迟处理 IRP 的任何行动直到低层的驱动程序已经完成处理 IRP。

一个 IoCompletion 例程能够在 IRQL DISPATCH_LEVEL 上被调用，但是一个功能或过滤器驱动程序可能需要在 IRQL PASSIVE_LEVEL 上处理 IRP。为了从一个 IoCompletion 例程返回从属的 IRQL，驱动程序能够使用内核事件。驱动程序注册了一个设置了一个内核模式事件的 IoCompletion 例程，然后，驱动程序在它的 DispatchPnP 例程里等待事件。当事件被设置时，低层的驱动程序已经完成了 IRP，且驱动程序能够处理该 IRP。

注意一个驱动程序不应利用这个技巧来等待低层的驱动程序完成一个电源 IRP（IRP_MJ_POWER）。在 DispatchPower 例程里等待一个事件可能引起死锁，该 DispatchPower 例程在 IoCompletion 例程里设置。参见第 3 部分第 1 章的传递电源 IRP 来获得更多的信息。

图 2.2 和图 2.3 给出了一个驱动程序怎样等待低层的驱动程序来完成一个 PnP IRP，该例给出了功能和总线驱动程序必须做什么及它们怎样与 PnP 管理器和 I/O 管理器相互作用。

图 2.2 延迟 PnP IRP 的处理---第 1 部分

下面的注释对应图 2.2 中的被圈起来的数字：

1. PnP 管理器调用 I/O 管理器来发送一个 IRP 给设备堆栈里顶端的驱动程序。
2. I/O 管理器调用顶端驱动程序的 DispatchPnP 例程。在本例中，设备堆栈里仅有两个驱动程序（功能驱动程序和父总线驱动程序），且功能驱动程序是顶端驱动程序。
3. 功能驱动程序声明并初始化一个内核模式事件，确定下一个低层驱动程序的堆栈位置，并为这个 IRP 设置一个 IoCompletion 例程。

功能驱动程序能够利用 **IoCopyCurrentStackLocationToNext** 来确定堆栈位置。

在调用 **IoSetCompletionRoutine** 时，功能驱动程序设置 *InvokeOnSuccess*、*InvokeOnError* 和 *InvokeOnCancel* 为 TRUE 并且传递内核模式事件作为环境参数的部分。

4. 在执行任何处理 IRP 操作之前，功能驱动程序向下传递 IRP 到具有 **IoCallDriver** 的设备堆栈。
5. I/O 管理器通过调用驱动程序的 DispatchPnP 例程发送一个 IRP 给设备堆栈里的下一个低层驱动程序。
6. 本例中的下一个低层驱动程序是设备堆栈里的最低层的驱动程序，即父总线驱动程序，总线驱动程序执行它的操作来启动设备。总线驱动程序设置 **Irp->IoStatus.Status**，如果相关于这个 IRP 则设置 **Irp->IoStatus.Information**，并通过调用 **IoCompleteRequest** 完成 IRP。

如果总线驱动程序调用其余的驱动程序例程或发送 I/O 给设备来启动它，总线驱动程序在它的 DispatchPnP 例程没有完成 PnP IRP，相反，它必须用 **IoMarkIrpPending** 标记 IRP 待定，并从它的 DispatchPnP 例程返回 STATUS_PENDING。驱动程序以后从另一个驱动程序例程里调用 **IoCompleteRequest**，可能是一个 DPC 例程。

图 2.3 给出了例子的第 2 部分，在这个例子中，设备堆栈里的高层的驱动程序恢复它们被延迟的 IRP 处理。

下面的注释对应图 2.3 中的被圈起来的数字：

7. 当总线驱动程序调用 **IoCompleteRequest** 时，I/O 管理器检查更高驱动程序的堆栈位置并调用它发现的任何 IoCompletion 例程。例子中，I/O 管理器定位并为下一个高层的驱动程序，即功能驱动程序，调用 IoCompletion 例程。
8. 功能驱动程序的 IoCompletion 例程设置在环境参数里提供的内核模式事件，并返回 STATUS_MORE_PROCESSING_REQUIRED。

IoCompletion 例程必须返回 STATUS_MORE_PROCESSING_REQUIRED 来阻止 I/O 管理器在此时调用由高层的驱动程序设置的 IoCompletion 例程，IoCompletion 例程使用这个状态阻止完成以便它的 DispatchPnP 例程能重获控制。当 DispatchPnP 例程完成 IRP 时，I/O 管理器为这个 IRP 恢复调用高层的 IoCompletion 例程。

9. I/O 管理器停止完成 IRP 并返回控制给 **IoCompleteRequest** 例程，例子中该例程即为总线驱动程序的 DispatchPnP 例程。

图 2.3 延迟 PnP IRP 的处理---第 2 部分

10. 总线驱动程序从它的带有状态指示 IRP 处理结果的 DispatchPnP 例程里返回：或者是 STATUS_SUCCESS 或者是一个错误状态。
11. **IoCallDriver** 返回控制给它的调用程序，在本例中即为功能驱动程序的 DispatchPnP 例程。
12. 功能驱动程序的 DispatchPnP 例程恢复处理 IRP。

如果 **IoCallDriver** 返回 STATUS_PENDING，DispatchPnP 例程已经在它的 IoCompletion 例程调用之前恢复执行。这样，DispatchPnP 例程必须等待它的 IoCompletion 例程通知内核事件。这确保 DispatchPnP 例程将不继续处理 IRP 直到所有低层的驱动程序已经完成它为止。

如果设置 **Irp->IoStatus.Status** 为一个错误，一个低层的驱动程序已经使 IRP 失效，且功能驱动程序不可以继续处理 IRP（除过任何必要的清理）。

13. 一旦低层的驱动程序已成功地完成 IRP，功能驱动程序处理 IRP。

对首先被父总线驱动程序处理的 IRP，总线驱动程序典型地在 **Irp->IoStatus.Status** 里设置一个成功状态，并有选择地在 **Irp->IoStatus.Information** 里设置一个值，除非功能和过滤器驱动程序使 IRP 失效，它们在 **IoStatus** 里留下值。

功能驱动程序的 DispatchPnP 例程调用 **IoCompleteRequest** 来完成 IRP，I/O 管理器恢复 I/O 完成处理。例子中，没有过滤器驱动程序在功能驱动程序之上，这样没有更多的 IoCompletion 例程将被调用。当 **IoCompleteRequest** 返回对功能驱动程序的 DispatchPnP 例程的控制时，DispatchPnP 例程返回状态。

对一些 IRP，如果一个功能或过滤器驱动程序在它的反方向设备堆栈上使 IRP 失效，PnP 管理器通知低层的驱动程序。例如，如果一个功能或过滤器驱动程序失效一个 IRP_MN_START_DEVICE，PnP 管理器给设备堆栈发送一个 IRP_MN_REMOVE_DEVICE。

第3章　启动、停止和删除设备

本章描述了何时 PnP 管理器启动、停止和删除设备及即插即用（PnP）驱动程序必须在它们的 DispatchPnP 例程做什么来支持这些活动，包括以下的主题：

3.1　启动设备

3.2　为了资源重新平衡而停止设备

3.3　删除设备

参看第 2 章获得处理 PnP IRP 的一般规则，参看《Windows2000 驱动程序开发参考》的卷一来获得每个 PnP IRP 的信息。

3.1 启动一设备

PnP 管理器发送一个 IRP_MN_START_DEVICE 给驱动程序，或者启动一个新被枚举的设备，或者重新启动一个存在但因资源重新平衡而停止的设备。

功能和过滤器驱动程序必须设置一个 IoCompletion 例程，向下传递 IRP_MN_START_DEVICE 到设备堆栈，并延迟它们的启动操作直到所有的低层驱动程序已经完成 IRP。父总线驱动程序，即设备堆栈里的底端驱动程序，在其他的驱动程序访问设备之前必须是执行设备的启动操作的第一个驱动程序。

为保证启动操作的适当时序，微软 Windows2000 PnP 管理器的延迟暴露设备接口，并且阻塞设备建立的请求直到启动 IRP 成功。

如果一个设备的驱动程序失效 IRP_MN_START_DEVICE，PnP 管理器发送一个 IRP_MN_REMOVE_DEVICE IRP 给设备堆栈（在 Windows2000 上）。为了响应这个 IRP，设备的驱动程序放弃它们的启动操作（如果它们成功的启动了 IRP），放弃它们的 AddDevice 操作，并与设备堆栈分离，PnP 管理器标记这样的一个设备为“失败的启动”。

这一部分包括以下的主题：

3.1.1 启动功能驱动程序里的一个设备

3.1.2 启动过滤器驱动程序里的一个设备

3.1.3 启动总线驱动程序里的一个设备

3.1.4 启动设备设计注意

3.1.1 启动功能驱动程序里的一个设备

功能驱动程序设置一个 IoCompletion 例程，向下传递一个 IRP_MN_START_DEVICE 到设备堆栈，并延迟它们的启动操作直到所有的低层驱动程序已经完成 IRP。参见第 2 章的 2.3 节获得使用一个内核事件和一个 IoCompletion 例程来延迟 IRP 处理的详细信息。

在所有的低层驱动程序已经完成 IRP 之后 DispatchPnP 例程重新得到控制时，功能驱动程序为了启动设备而执行它的任务，一个功能驱动程序用如下的过程启动设备：

1. 如果一个低层驱动程序使 IRP 失效（**IoCallDriver** 返回的一个错误），不要继续处理 IRP，做

一些必要的清理并从 DispatchPnP 例程返回（去本列表的最后一步）。

2. 如果低层驱动程序成功地处理 IRP，启动该设备。

启动一个设备的确切步骤因设备不同而变化，这些步骤应该包括映射 I/O 空间、初始化硬件寄存器、设置设备为 D0 电源状态和用 **IoConnectInterrupt** 连接中断。如果该驱动程序在一个 IRP_MN_STOP_DEVICE 请求之后正重新启动一个设备，该驱动程序可能有设备状态需要保存。

在任何驱动程序能够访问设备之前，该设备应该被加电，参见第 3 部分第 1 章关于加电一个设备的更多信息。

如果设备能够被唤醒，它的电源策略所有者（通常为功能驱动程序）应该在它加电设备之后且在它完成 IRP_MN_START_DEVICE 请求之前发送一个等待/唤醒 IRP。想要详细的资料，参见第 3 部分第 4 章中关于发送一个等待/唤醒 IRP。

3. 在 IRP-holding 队列中启动 IRP。

清空驱动程序定义的 HOLD_NEW_REQUESTS 标记并且启动 IRP-holding 队列中的 IRP。当首次启动一个设备和在一个查询停止或停止 IRP 之后重新启动一个设备时，驱动程序应该能够做这些工作。参看 3.2.5 节获得更多的信息。

4. [可选择的]通过调用 **IoSetDeviceInterfaceState**，启用设备的接口。

如果有接口，则启动驱动程序以前在 AddDevice 例程里注册的接口（或者在一个 INF 里，或者通过另外一个组件，比如协同安装程序）。

在 Windows2000 上，直到 IRP_MN_START_DEVICE IRP 完成，PnP 管理器才发送设备接口到达的通知，表明设备的所有驱动程序已经完成它们的启动操作。PnP 管理器也使任何建立的请求失效，这些请求在设备所有的驱动程序完成启动 IRP 之前到达。

5. 完成 IRP。

功能驱动程序的 IoCompletion 例程返回 STATUS_MORE_PROCESSING_REQUIRED，如 2.3 节所描述的，这样功能驱动程序的 DispatchPnP 例程必须调用 **IoCompleteRequest** 来恢复 I/O 完成处理。

如果功能驱动程序的启动操作成功，驱动程序设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS，用一个 IO_NO_INCREMENT 的优先增强调用 **IoCompleteRequest**，并从它的 DispatchPnP 例程返回 STATUS_SUCCESS。

如果功能驱动程序在它的启动操作过程里遇到一个错误，驱动程序在 IRP 里设置错误状态，用 IO_NO_INCREMENT 调用 **IoCompleteRequest**，并从它的 DispatchPnP 例程里返回错误。

如果一个较低层驱动程序使 IRP 失效（**IoCallDriver** 返回的一个错误），功能驱动程序用 IO_NO_INCREMENT 调用 **IoCompleteRequest** 并从它的 DispatchPnP 例程里返回 **IoCallDriver** 错误。功能驱动程序在这种情况下不设置 **Irp->IoStatus.Status**，这是由于状态已经由以前使 IRP 失效的低层驱动程序所设置。

当一个功能驱动程序接受到一个 IRP_MN_START_DEVICE 请求时，它应该在 **IrpSp->Parameters.StartDevice.AllocatedResources** 和 **IrpSp->Parameters.StartDevice.AllocatedResourcesTranslated** 上检查结构，该结构描述了 PnP 管理器已经分别地指定给设备的原始的和转换的资源。驱动程序应该出于一个调试帮助的目的，在设备扩展里保存每个资源列表的一份拷贝。

资源列表是成对的 CM_RESOURCE_LIST 结构，在这个结构里，每个原始列表的元素对应于转换列表里相同的元素，例如，如果 **AllocatedResource.List[0]**描述了一个原始的 I/O 端口范围，那么，**AllocatedResourceTranslated.List[0]**描述了转换后的同样的范围，每个转换的资源包括一个物理地址和资源类型。

如果分配驱动程序一个转换了的内存资源（**CmResourceTypeMemory**），它必须调用

MmMapIoSpace 来映射该物理地址到一个虚拟地址，通过该虚拟地址，驱动程序能够访问设备寄存器。对一个以平台独立的方式操作的驱动程序，如果需要的话，它应该检查每个返回的、转换的资源并映射它。

一个功能驱动程序应该做下面的工作来响应一个 IRP_MN_START_DEVICE 以确保访问所有的设备资源：

1. 拷贝 IrpSp->Parameters.StartDevice.AllocatedResources 到设备扩展。
2. 拷贝 IrpSp->Parameters.StartDevice.AllocatedResourcesTranslated 到设备扩展。
3. 在一个循环里，在 **AllocatedResourcesTranslated** 里检查每个描述符元素，如果描述符资源类型是 **CmResourceTypeMemory**，调用 **MmMapIoSpace**，并传递物理地址和转换的资源的长度。

当某一个驱动程序接收到一个 IRP_MN_STOP_DEVICE、IRP_MN_REMOVE_DEVICE 或者 IRP_MN_SURPRISE_REMOVAL 请求时，它必须通过在一个相似的循环里调用 **MmUnmapIoSpace** 来释放映射。如果该驱动程序必须失效 IRP_MN_START_DEVICE 请求时，它也应该调用 **MmUnmapIoSpace**。

参见在《内核模式驱动程序设计指南》的第 16 章里“重新映射总线相关的内存空间地址到虚拟地址”来获得更多的信息。

3.1.2 启动过滤器驱动程序里的一个设备

一个顶层的过滤器驱动程序应该增强功能驱动程序的任何启动活动，一个低层的驱动程序典型地增强了设备特性并可能参与启动设备。

3.1.3 启动总线驱动程序里的一个设备

在一个总线驱动程序的 DispatchPnP 例程里，它用一个如下的程序启动一个子设备（子代 PDO）：

1. 启动设备。

确切的步骤因设备不同而异，举例来说，PCI 总线驱动程序对它的映射寄存器编程以便 PCI 总线上的请求能够实现，PnP ISA 总线驱动程序能够使 PnP ISA 卡可用，这样功能驱动程序能访问它。

2. 完成 IRP。

如果总线驱动程序的启动操作成功，驱动程序设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS，用一个 IO_NO_INCREMENT 的优先增强调用 **IoCompleteRequest**，并从它的 DispatchPnP 例程返回 STATUS_SUCCESS。

如果功能驱动程序在其启动操作过程里遇到一个错误，驱动程序在 IRP 里设置错误状态，用 IO_NO_INCREMENT 调用 **IoCompleteRequest**，并从其 DispatchPnP 例程里返回错误。

如果一个总线驱动程序需要一些时间来启动设备，它能够标记 IRP 为待定，并返回 STATUS_PENDING。

3.1.4 启动设备设计注意

- PnP 管理器失效了设备的创始请求直到 IRP_MN_START_IRP 完成为止，表明设备的所有的驱动程序已经执行了它们的启动操作。
- 由于 DispatchPnP 例程在一个系统线程的 IRQL PASSIVE_LEVEL 的环境里运行，只要驱动程序没有控制拥有一个系统分页文件的设备，由 **ExAllocatePoolWithTag** 分配的、在初始化过程中专用的任何内存能够从分页的池得到。这样一个内存分配必须在返回控制之前由 **ExFreePool** 释放。
- 甚至在设备启动过程中，一个 WDM 设备驱动程序的 ISR 应该能够决定是否它已经被伪中断调用。从 **IoConnectInterrupt** 调用返回时，在处理 IRP_MN_START_DEVICE 的代码路径里，如果中断在设备上是允许的，该 ISR 能够被立即调用。

3.2 为了资源重新平衡而停止一个设备

PnP 管理器指示驱动程序停止一个设备以便它能够重新平衡正被设备使用的硬件资源，当一个新的设备被枚举时，这需要已经在使用的资源，重新平衡典型情况下是必要的。本部分有以下的主题：

3.2.1 理解何时停止发布 IRP
3.2.2 处理 IRP_MN_QUERY_STOP_DEVICE 请求
3.2.3 处理 IRP_MN_STOP_DEVICE 请求
3.2.4 处理 IRP_MN_CANCEL_STOP_DEVICE 请求
3.2.5 当一个设备被暂停时，保留进入的 IRP

3.2.1 理解何时停止发布 IRP

图 3.1 给出了涉及设备的停止和重新启动的序列，参看第 1 章获得 PnP 设备状态从枚举到删除的广泛的讨论。

图 3.1 停止一个设备

下面的注释对应图 3.1 中的被圈起来的数字：

1. PnP 管理器发布一个 IRP_MN_QUERY_STOP_DEVICE 来询问是否一个设备的驱动程序能够停止该设备并且释放它的硬件资源。

   如果设备堆栈里的所有驱动程序返回 STATUS_SUCCESS，驱动程序已经设置设备为一状态（停止待定），从该状态，设备能够被快速地停止。

   PnP 管理器查询所需要的尽量多的设备堆栈来重新平衡必要的资源。
2. PnP 管理器发布一个 IRP_MN_STOP_DEVICE 来停止该设备。

   仅仅如果一个设备前面的查询停止 IRP 成功完成，PnP 管理器发送一个停止 IRP。为了响应一个停止 IRP，驱动程序释放了设备的硬件资源（比如它的 I/O 端口），并保留需要访问的设备的

任何 IRP。

3. 在成功地重新平衡资源之后，PnP 管理器发布 IRP_MN_START_DEVICE 请求来重新启动它在重新平衡过程中停止的设备。
4. 否则，PnP 管理器通过发送一个 IRP_MN_CANCEL_STOP_DEVICE 来取消一个查询停止 IRP。

为了响应一个 IRP_MN_CANCEL_STOP_DEVICE，一个设备的驱动程序返回设备启动的状态，并恢复处理设备的 I/O 请求。

如果堆栈里的一个驱动程序使请求失效或如果全面的重新平衡操作失败。并且正在取消它所有的查询停止请求，PnP 管理器取消设备堆栈的查询停止。当 PnP 管理器刚好取消了一个设备堆栈的查询停止时，由于附着在已失败查询的驱动程序之上的任何驱动程序有设备处于停止待定状态，所以，PnP 管理器发送 IRP_MN_CANCEL_STOP_DEVICE 请求。当 IRP_MN_CANCEL_STOP_DEVICE 成功时，驱动程序已经返回设备为启动的状态。

5. 在资源的重新平衡之后，如果一个驱动程序重新启动设备失效，PnP 管理器发送删除 IRP 给设备堆栈（在 Windows2000 上）。

PnP 管理器首先发送一个 IRP_MN_SURPRISE_REMOVAL 请求，然后发送一个 IRP_MN_REMOVE_DEVICE 请求，但是仅仅在设备的所有打开处理被关闭之后。

一个 PnP 设备的硬件资源的重新平衡对应用程序和最终用户必须是透明的，用户可能在操作上经历一个短暂的滞后，但是数据不能够被丢失，当你处理停止 IRP 时，你必须考虑到这一点。

3.2.2 处理 IRP_MN_QUERY_STOP_DEVICE 请求

一个 IRP_MN_QUERY_STOP_DEVICE 请求首先被设备堆栈里的顶端驱动程序所处理，然后被每下一个低层的驱动程序所处理，一个驱动程序在它的 DispatchPnP 例程里处理停止 IRP。

在响应一个 IRP_MN_QUERY_STOP_DEVICE 时，一个驱动程序必须做下面的工作：

1. 决定是否设备能够被停止，以及它的硬件资源被释放而没有不利的影响。

如果下面的任何选项是真，一个驱动程序必须失效一个查询停止 IRP。

- 已经通知一个驱动程序（通过 IRP_MN_DEVICE_USAGE_NOTIFICATION）设备在一个分页、休眠或崩溃转储文件的路径里。
- 不能够释放设备的硬件资源。

如果下面的选项是真，一个驱动程序可能使一个查询停止 IRP 失效。

- 驱动程序不应扔掉扔掉 I/O 请求和没有排队 IRP 的机制。

当设备处于停止状态时，一个驱动程序必须保留需要访问设备的 IRP。如果一个驱动程序没有排队 IRP，它不会允许停止设备，那样的话，必须使一个查询停止使 IRP 失效。

这个规则的一个例外是设备被允许扔掉 I/O，这样的一个设备的驱动程序能够成功查询停止并停止请求而没有排队 IRP。

2. 如果设备不能够被停止，使查询停止 IRP 失效。

设置 **Irp->IoStatus.Status** 为一适当的错误状态，用 IO_NO_INCREMENT 调用 **IoCompleteRequest**，并从驱动程序的 DispatchPnP 例程里返回错误。不要传递 IRP 给下一个较低层的驱动程序。

3. 如果设备能够被停止并且驱动程序排队 IRP，在设备扩展里设置 HOLD_NEW_REQUESTS 标记以便随后的 IRP 将被排队。（参见 3.2.5 节。）

可选择的，一个设备的驱动程序能够拖延完全地暂停设备直到驱动程序接收到随后的

IRP_MN_QUERY_STOP_DEVICE 请求。然而，这样的驱动程序必须排队一些请求，当它到达时，这些请求可阻止它们立即地成功停止 IRP。直到设备被重新启动，这样的驱动程序必须排队如下的请求：

- IRP_MN_DEVICE_USAGE_NOTIFICATION 请求（举个例子，在设备上放置一个分页文件）
- 同步传送的请求
- 生成能够阻止驱动程序成功一个停止 IRP 的请求

4. 如果设备在进度失效里不能有一个 IRP，要确保传递给其余驱动程序例程和给较低层的驱动程序的任何未完成的请求已经完成。

一个驱动程序能够取得这一点的一个方法是使用一个引用计数和一个事件，确保所有请求已经完成：

- 在它的 AddDevice 例程里，驱动程序在设备扩展里定义一个 I/O 引用计数并初始化计数为一。
- 也在它的 AddDevice 例程里，驱动程序用 **KeInitializeEvent** 生成一个事件并用 **KeClearEvent** 初始化事件为 Not-Signaled 状态。
- 它处理一个 IRP 的每个时刻，驱动程序用 **InterlockedIncrement** 增大引用计数。
- 它完成一个请求的每个时刻，驱动程序用 **InterlockedDecrement** 减少引用计数。

如果有一个请求，或者刚好在调用 IoCallDriver 之后，如果驱动程序使用了请求的非 IoCompletion 例程，驱动程序在 IoCompletion 例程里减少引用计数。

- 当驱动程序接收到一个 IRP_MN_QUERY_STOP_DEVICE 时，它用 **InterlockedDecrement** 减少了引用计数。如果没有未完成的请求，这将缩减引用计数为零。
- 当引用计数为零时，驱动程序用 **KeSetEvent** 设置事件，并发送信号以便查询停止代码能够继续。

作为以上程序的一个替换对象，一个驱动程序能够在进程中的任何 IRP 之后串行化 IRP_MN_QUERY_STOP_DEVICE IRP。

5. 执行放置设备在一个停止待定状态里所必要的任何其他步骤。

在一个驱动程序使一查询停止 IRP 成功之后，它必须准备使一个 IRP_MN_STOP_DEVICE 成功。

6. 完成 IRP。

在一个功能或过滤器驱动程序里：

- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。
- 用 **IoSkipCurrentIrpStackLocation** 定位下一个堆栈位置，并用 **IoCallDriver** 传递 IRP 给下一个较低层驱动程序。
- 当从 DispatchPnP 例程返回状态时，从 **IoCallDriver** 传播这个状态。
- 不要完成 IRP。

在一个总线驱动程序里：

- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。

然而，如果在总线上的设备使用硬件资源，重新评价总线和子设备的资源需求。如果任何资源需求已经改变，返回 STATUS_RESOURCE_REQUIREMENTS_CHANGED 代替 STATUS_SUCCESS。这个状态表示成功，但是请求 PnP 管理器在发送一个停止 IRP 之前重新查询你的资源。

- 用 IO_NO_INCREMENT 完成 IRP（**IoCompleteRequest**）。
- 从 DispatchPnP 例程返回。

如果在设备堆栈里的任何一个驱动程序使 IRP_MN_QUERY_STOP_DEVICE 失效，PnP 管理

器发送一个 IRP_MN_CANCEL_STOP_DEVICE 给设备堆栈，这阻止了驱动程序为一个查询停止 IRP 而需要一个 IoCompletion 例程，以便侦测是否一个较低层的驱动程序以前使 IRP 失效过。

3.2.3 处理 IRP_MN_STOP_DEVICE 请求

一个 IRP_MN_STOP_DEVICE 请求首先被设备堆栈里的顶层驱动程序所处理，然后被每下一个较低层的驱动程序所处理。一个驱动程序在它的 DispatchPnP 例程里处理停止 IRP。

一个驱动程序以如下程序处理一个 IRP_MN_STOP_DEVICE 请求：

1. 确保设备被暂停。

如果一个驱动程序没有完全暂停设备来响应 IRP_MN_QUERY_STOP 请求，它现在必须这样作了。在设备扩展里设置一个 HOLD_NEW_REQUESTS 标记并执行任何其他必要的操作来暂停设备。

在资源重新平衡操作过程中设备有可能失去电力，因而有可能失去设备状态。设备的驱动程序应该保存任何设备状态信息，并且当它们接收到随后的 IRP_MN_START_DEVICE 请求时重新保存它。

2. 释放设备的硬件资源。

在一个功能驱动程序里，确切的操作取决于设备和驱动程序，但是能够包括用 **IoDisconnectInterrupt** 断开一个中断，用 **MmUnmapIoSpace** 释放一个物理地址范围和释放 I/O 端口。

如果一个过滤程序或总线驱动程序以前获得设备的任何硬件资源，驱动程序必须释放资源来响应一个 IRP_MN_STOP_DEVICE 请求。

3. 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。
4. 传递 IRP 给下一个较低层的驱动程序或者完成该 IRP。

- 在一个功能或过滤器驱动程序里，用 **IoSkipCurrentIrpStackLocation** 确定下一个堆栈位置，用 **IoCallDriver** 传递 IRP 给下一个较低层的驱动程序，并象从 DispatchPnP 例程返回状态一样从 **IoCallDriver** 返回状态。不要完成 IRP。
- 在一个总线驱动程序里，使用 **IoCompleteRequest** 的 IO_NO_INCREMENT 完成 IRP 并从 DispatchPnP 例程返回。

当设备为了 PnP 资源重新平衡而停止时，驱动程序不能启动访问设备的任何 IRP。驱动程序必须如 3.2.5 节里描述的一样排队这样的 IRP，或者，如果驱动程序不应扔掉 I/O 请求和不能够实现一个 IRP-holding 队列，则使它们失效。

3.2.4 处理 IRP_MN_CANCEL_STOP_DEVICE 请求

一个 IRP_MN_CANCEL_STOP_DEVICE 请求首先被一个设备的父总线驱动程序所处理，然后被每个下一个更高级的驱动程序所处理。一个驱动程序在它的 DispatchPnP 例程里处理停止 IRP。

为响应一个 IRP_MN_CANCEL_STOP_DEVICE 请求，一个驱动程序必须返回设备它的启动状态并且恢复正常操作。驱动程序必须成功于一个取消停止 IRP。

一个驱动程序用如下的程序处理一个 IRP_MN_CANCEL_STOP_DEVICE 请求：

1. 延迟重新启动设备直到更低层的驱动程序已经完成它们的重新启动操作。（看第 2 章 2.3 节）
2. 在较低层的驱动程序完成之后，返回设备它的被启动的状态。

   确切的操作取决于设备和驱动程序。
3. 在 IRP-holding 队列里启动 IRP。

   如果当设备处于停止待定状态时驱动程序正保留请求，清除 HOLD_NEW_REQUESTS 标记并在 IRP-holding 队列里启动 IRP。参看 3.2.5 节获得更多的信息。
4. 用 **IoCompleteRequest** 完成 IRP。

■ 在一个功能或过滤器驱动程序里：

象 2.3 节里所描述的一样，驱动程序的 IoCompletion 例程在以前返回 STATUS_MORE_PROCESSING_REQUIRED，这样驱动程序的 DispatchPnP 例程必须调用 **IoCompleteRequest** 来恢复 I/O 完成处理。

驱动程序设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。用一个 IO_NO_INCREMENT 的优先增强调用 **IoCompleteRequest**，并从它的 DispatchPnP 例程里返回 STATUS_SUCCESS。

一个总线驱动程序不应该使这个操作失效，如果一个驱动程序使重新启动 IRP 失效，设备处于一个不一致的状态，将不能正常操作。

■ 在一个父总线驱动程序里：

如果总线驱动程序的重新启动操作成功，通过指定的一个 IO_NO_INCREMENT 的优先增强，驱动程序设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS 并调用 **IoCompleteRequest**。总线驱动程序从它的 DispatchPnP 例程里返回 STATUS_SUCCESS。

一个总线驱动程序不应该使这个操作失效过，如果一个驱动程序使重新启动 IRP 失效，设备处于一个不一致的状态，将不能正常操作。

当设备启动和运行时，驱动程序接收到一个伪取消停止请求。这可能发生，例如，如果一个驱动程序（或者设备堆栈里的一个更高级的驱动程序）使一个 IRP_MN_QUERY_STOP_DEVICE 请求失效。当一个设备启动和运行时，驱动程序能够安全地使设备的伪取消停止请求成功。

3.2.5 当一个设备被暂停时，保留进入的 IRP

当驱动程序的资源正被重新平衡时，一个设备的驱动程序必须暂停设备。在资源重新平衡的过程中，一些驱动程序暂停设备来响应一个 IRP_MN_QUERY_STOP_DEVICE 请求，其他的驱动程序则拖延暂停设备直到它们接收到 IRP_MN_STOP_DEVICE 请求。在每一种情况下，当 IRP_MN_STOP_DEVICE 成功时，必须暂停设备。

驱动程序必须在进程中完成任何设备的 IRP 并阻止启动任何需要访问设备的新的 IRP。

当一个设备被暂停时，为保留 IRP，一个驱动程序执行如下的过程：

1. 在它的 AddDevice 例程里，用象 HOLD_NEW_REQUESTS 的一个名字在设备扩展里定义一个设备标志。清除标志。
2. 为了保留 IRP 生成一个 FIFO 队列。

如果驱动程序已经排队 IRP，由于在暂停一个设备之前，驱动程序对完成任何未完成的请求是必要的，所以它能够再次使用相同的队列。

如果驱动程序还没有一个 IRP 队列，它必须在它的 AddDevice 例程里生成一个。它生成什么类型的队列取决于驱动程序怎么处理队列。一个驱动程序可能使用一个内部锁定的、双链接的列表和 **ExInterlocked..List** 例程。

3. 在它的 IRP_MN_QUERY_STOP_DEVICE（或 IRP_MN_STOP_DEVICE）的 DispatchPnP 代码里，完成任何未完成的请求，并设置 HOLD_NEW_REQUESTS 标志。
4. 在一个访问设备的调度例程里，如 DispatchWrite 或 DispatchRead，检查 HOLD_NEW_REQUESTS 标志是否设置。如果设置的话，驱动程序必须标记 IRP 为待定并排队它。（参见内核模式驱动程序设计指南获得有关标记 IRP 为待定的更多信息。）

驱动程序的 DispatchPnP 例程必须继续处理 PnP IRP 而不是保留它们，并且 DispatchPower 例程必须继续处理电源 IRP。

5. 在 DispatchPnP 里，为响应一个启动或取消停止 IRP，清除 HOLD_NEW_REQUESTS 标志并在 IRP-holding 队列里启动 IRP。

这些动作可能是处理这些 PnP IRP 的最后步骤，例如，为响应一个启动 IRP，驱动程序必须首先执行一些操作来启动设备，然后，它能够在 IRP-holding 队列里启动 IRP。

在 IRP-holding 队列里处理 IRP 中的错误没有影响从启动或取消停止 IRP 返回的状态。

参见《Windows2000 驱动程序开发参考》的卷二获得关于 **ExInterlocked..List** 例程和 **IoMarkIrpPending** 的信息。

3.3 删除一个设备

当设备已经或正在物理地从机器删除时，PnP 管理器指令驱动程序删除一个设备的设备对象。当一个用户请求更新一个设备的驱动程序时，PnP 管理器也发送一个删除 IRP。

下面的部分描述了 PnP 管理器何时发布删除 IRP 和驱动程序该做什么响应这些 IRP。本部分涵盖以下的主题：

3.3.1 理解何时发布删除 IRP
3.3.2 处理 IRP_MN_QUERY_REMOVE_DEVICE 请求
3.3.3 处理 IRP_MN_REMOVE_DEVICE 请求
3.3.4 处理 IRP_MN_CANCEL_REMOVE_DEVICE 请求
3.3.5 处理 IRP_MN_SURPRISE_REMOVAL 请求

3.3.1 理解何时发布删除 IRP

图 3.2 给出了涉及删除一个设备的驱动程序里的 IRP 的典型序列。参见第 2 部分第 1 章获得设备状态从枚举到删除的更广泛的讨论。

下面的注释相应于图 3.2 中的被圈起来的数字：

1. 查询删除

PnP 管理器发布一个 IRP_MN_QUERY_REMOVE_DEVICE 来询问是否一个设备能够被删除而没有使机器崩溃，当一个用户请求更新一个设备的驱动程序时，它也发送这个 IRP。

图 3.2 典型地删除 IRP 转换

如果设备堆栈里所有的驱动程序返回 STATUS_SUCCESS，驱动程序已经使设备为删除待定状态。在这种状态下，驱动程序不可以启动阻止设备被删除的任何操作。

在这个“干净的”删除状况下，在发送一个删除 IRP 之前，PnP 管理器发送一个查询删除 IRP。看第五步对“突然的”删除的描述。

尽管它没有在上面的结构图中给出，一个总线驱动程序可能为没有启动的设备接收一个 IRP_MN_QUERY_REMOVE_DEVICE。如果一个用户请求动态地删除一个设备，且该设备物理地存在于机器但不可用时，这种情况能够发生。

2. 在成功查询之后删除

PnP 管理器发布一个 IRP_MN_REMOVE_DEVICE 来删除一个设备的驱动程序。

驱动程序必须使这个请求成功，设备的驱动程序执行任何必要的清空，从设备堆栈分离，并删除 FDO 和任何过滤程序 DO。父总线驱动程序保留 PDO 直到用户物理地从机器删除设备。

注意到驱动程序可能删除 IRP 之前接收到一个 IRP_MN_STOP_DEVICE，但这不是必须的。为了资源的重新平衡，IRP_MN_STOP_DEVICE 被用来暂停一个设备；这不是一个向着删除方向的一步。当设备停止时，如果一个用户删除设备硬件，PnP 管理器在停止 IRP 之后的某个时刻发送一个删除 IRP，但是停止不是一个删除的先决条件。

3. 重新枚举设备

如果在驱动程序已经删除它们的设备对象之后重新枚举设备的话，PnP 管理器调用驱动程序的 AddDevice 例程并发布一个 IRP_MN_START_DEVICE 来保存设备。（也看图 1.1）

4. 取消一个查询删除

PnP 管理器发布一个 IRP_MN_CANCEL_REMOVE_DEVICE 来取消一个查询删除请求。为响应一个 IRP_MN_CANCEL_REMOVE_DEVICE，驱动程序使设备返回到它的启动状态。

5. 突然删除(Windows2000)

在 Windows2000 上，如果一个用户从机器拔掉一个设备而没有首先通过 Unplug 或 Eject Hardware 应用程序，则 PnP 管理器发送一个 IRP_MN_SURPRISE_REMOVAL IRP。

由于驱动程序没有提前接收到警告，这个状况被称作“突然的”删除。

为响应一个 IRP_MN_SURPRISE_REMOVAL IRP，设备的驱动程序使任何未决的 I/O 失效，并释放为设备所使用的硬件资源。驱动程序必须确保没有组件试图访问这些设备，由于设备不再存在。

所有的驱动程序必须处理一个 IRP_MN_SURPRISE_REMOVAL IRP，并必须设置状态为 STATUS_SUCCESS。

不能够取消一个 IRP_MN_SURPRISE_REMOVAL IRP。

6. 在突然删除之后删除(Windows2000)

当设备的所有打开的处理被关闭时，Windows2000 的 PnP 管理器发送一个 IRP_MN_REMOVE_DEVICE 请求给设备的驱动程序。每个驱动程序与设备堆栈分离并删除它的设备对象。

7. 突然删除(Windows98)

在 Windows98 上，当一个设备被删除而没有警告时，一个驱动程序没有接收到一个 IRP_MN_SURPRISE_REMOVAL 。Windows98 的 PnP 管理器仅仅发送一个 IRP_MN_REMOVE_DEVICE，WDM 驱动程序必须有代码来处理两个：即一个 IRP_MN_SURPRISE_REMOVAL 和一个 IRP_MN_REMOVE_DEVICE 而没有一个前面的突然删除 IRP（Windows98 的行为）。这里 IRP_MN_SURPRISE_REMOVAL 后面跟着一个 IRP_MN_REMOVE_DEVICE（Windows2000 的突然删除行为）。

8. 在一个失败的启动之后删除(Windows2000)

如果设备的一个驱动程序使 IRP_MN_START_DEVICE 失效，PnP 管理器发送一个

IRP_MN_REMOVE_DEVICE 请求给设备堆栈。这样一个删除 IRP 确保设备的所有驱动程序被通知说该设备没有成功地启动。为响应 IRP_MN_REMOVE_DEVICE IRP，设备的驱动程序取消它们的启动操作（如果它们成功地启动了 IRP）和 AddDevice 操作。PnP 管理器标记这样一个设备为“失败的启动”。

这个行为仅用于 Windows2000 平台，在 Windows98 上，PnP 管理器发送一个 IRP_MN_STOP_DEVICE 来响应一个失败的启动。

PnP 设备的一个驱动程序能够比图 3.2 所示的在更多的条件下接收到一个 IRP_MN_SURPRISE_REMOVAL。举例而言，一个用户可以给机器插入一个 PC 卡，然后在设备启动之前删除它。在这样的情况下，在驱动的添加例程被调用之后，而在发布 IRP_MN_STOP_DEVICE 请求之前，PnP 管理器发布一个突然删除 IRP。在驱动程序添加例程调用之后的任何时候，PnP 设备的一个驱动程序必须准备处理删除 IRP。

3.3.2 处理一个 IRP_MN_QUERY_REMOVE_DEVICE 请求

PnP 管理器发送这个 IRP 来通知驱动程序一个设备将要从机器里被删除并询问是否一个设备能够被删除而不会使机器崩溃。当一个用户请求更新设备的驱动程序时，它也发送这个 IRP。

PnP 管理器在系统线程环境的 IRQL PASSIVE_LEVEL 上发送这个 IRP。

在发送这个 IRP 给设备的驱动程序之前，它做下面的工作：

- 通知所有的用户模式应用程序，这些应用程序已注册接收设备上（或者一个相关的设备）的通知。

这包括被注册接收通知的应用程序，它可能在一个设备上、在一个设备的后代上（子代设备、子代之子代等等）或者在一个设备删除的关系上。通过调用 **RegisterDeviceNotification**，一个应用程序为这样的通知而注册。

为响应这些通知，一个应用程序或者准备设备删除（关闭设备的处理），或者使查询失效。

- 通知所有的内核模式驱动程序，这些驱动程序被注册用于设备上（或者一个相关的设备）的通知。

这包括被注册用于通知的驱动程序，它可能在一个设备上、在一个设备的后代上或者在一个设备删除的关系上。通过用一个 **EventCategoryTargetDeviceChange** 的事件种类调用 **IoRegisterPlugPlayNotification**，一个驱动程序为这样的通知所注册。

为响应这些通知，一个驱动程序或者准备设备删除（关闭设备的处理），或者使查询失效。

- 发送 IRP_MN_QUERY_REMOVE_DEVICE IRP 给设备的后代驱动程序。
- （在 Windows2000 上）如果一个文件系统被安装在设备上，PnP 管理器给文件系统和一些文件系统过滤程序发送一个查询删除请求。如果有设备的打开句柄，文件系统典型地使查询删除请求失效。如果没有的话，文件系统则典型地锁定卷来阻止将来成功地建立。如果一个安装的文件系统不支持一个查询删除请求，PnP 管理器使设备的查询删除请求失效。

如果上面所有的步骤都成功，PnP 管理器给设备的驱动程序发送 IRP_MN_QUERY_REMOVE_DEVICE。

一个 IRP_MN_QUERY_REMOVE_DEVICE 请求首先被设备堆栈里的顶层驱动程序所处理，然后被每下一个较低层驱动程序所处理。一个驱动程序在它的 DispatchPnP 例程里处理删除 IRP。

为响应一个 IRP_MN_QUERY_REMOVE_DEVICE，一个驱动程序必须作下面的工作：

1. 决定一个设备是否能够从机器中被删除而没有使操作崩溃。

如果下面的任何选项为真的话，一个驱动程序必须使一个查询删除 IRP 失效：

- 删除设备可能导致数据丢失。
- 一个组件有设备的一个打开句柄。（这仅是 Windows98 的一个论点，如果在 IRP_MN_QUERY_REMOVE_DEVICE 完成之后没有打开句柄，Windows2000 跟踪打开句柄并使请求失效。）
- 已经通知一个驱动程序（通过一个 IRP_MN_DEVICE_USAGE_NOTIFICATION IRP）设备在一个分页的、崩溃的转储或者休眠的文件路径里。
- 驱动程序相对于设备有一个未完成的接口引用，这就是说，驱动程序提供一个接口来响应一个 IRP_MN_QUERY_INTERFACE 请求并且接口没有被间接访问。

如果设备能够唤醒，驱动程序应该取消等待唤醒并使查询删除请求成功。

2. 如果不能够删除设备，使查询删除 IRP 失效。

设置 **Irp->IoStatus.Status** 为一个适当的错误状态（典型地为 STATUS_UNSUCCESSFUL），用 IO_NO_INCREMENT 调用 **IoCompleteRequest**，并从驱动程序的 DispatchPnP 例程里返回。不要传递 IRP 给下一个较低层的驱动程序。

3. 记录设备以前的 PnP 状态。

当它收到 IRP_MN_QUERY_REMOVE_DEVICE 请求时，驱动程序应当记录设备 PnP 状态，因为如果查询被取消（IRP_MN_CANCEL_REMOVE_DEVICE），驱动程序必须使设备返回到那个状态。从前的状态是典型地“启动的”，但也许不是，例如，如果设备被禁用的时候。

4. 完成 IRP：

在一个功能或过滤器驱动程序里：

- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。
- 用 **IoSkipCurrentIrpStackLocation** 定位下一个堆栈位置，并用 **IoCallDriver** 传递 IRP 给下一个较低层驱动程序。
- 当从 DispatchPnP 例程返回状态时，从 **IoCallDriver** 传播此状态。
- 不要完成 IRP。

在一个总线驱动程序里：

- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。
- 用 IO_NO_INCREMENT 完成 IRP（**IoCompleteRequest**）。
- 从 DispatchPnP 例程返回。

如果设备堆栈里的任何驱动程序使一个 IRP_MN_QUERY_REMOVE_DEVICE 失效，PnP 管理器发送一个 IRP_MN_CANCEL_REMOVE_DEVICE 给设备堆栈，这一过程阻止了为一个查询删除 IRP 而需要一个 IoCompletion 例程，从而可以检测是否一个低层的驱动程序在以前使 IRP 失效过。

一旦一个驱动程序使一个 IRP_MN_QUERY_REMOVE_DEVICE 成功，且它认为设备处于删除待定状态时，驱动程序必须为设备使一些随后的生成请求失效。驱动程序如通常一样处理其余所有的 IRP，直到驱动程序接收到一个 IRP_MN_CANCEL_REMOVE_DEVICE 或 IRP_MN_REMOVE_DEVICE 为止。

3.3.3 处理一个 IRP_MN_REMOVE_DEVICE 请求

PnP 管理器使用这个 IRP 来指令驱动程序删除一个设备的软件表示（设备对象等等）。当一个

设备已经用一个有序的方式（例如，在 Unplug 或者 Eject Hardware 小应用程序里被一个用户初始化）被删除时，通过突然（一个用户从插槽里拔出设备而没有预先的警告），或者用户请求更新驱动程序时，PnP 管理器发送这个 IRP。

在发送这个 IRP 到设备的驱动程序之前，PnP 管理器做下面的工作：

- 如果有子设备，发送 IRP_MN_REMOVE_DEVICE 请求给设备的子设备。
- 通知任何用户模式组件和为通知而注册的内核模式驱动程序，设备正在被删除。PnP 管理器在处理一个设备时调用任何为对象设备通知而注册的用户模式组件，并调用任何为 **EventCategoryTargetDeviceChange** 而注册的内核模式驱动程序。
- （在 Windows2000 上）如果一个文件系统被安装在设备上，PnP 管理器给文件系统和任何文件系统过滤程序发送一个删除请求，作为响应，一个文件系统典型地卸下卷。

设备堆栈里的顶层驱动程序处理一个删除 IRP 并传递它给下一个较低层的驱动程序，一个设备的父总线驱动程序是执行它的删除设备操作的最后一个驱动程序。一个驱动程序在它 DispatchPnP 例程里的处理删除 IRP。

在一个驱动程序为一个 IRP_MN_REMOVE_DEVICE 请求返回成功之前，它必须保证设备的所有资源已经释放，这个 IRP 在驱动程序卸载之前可能是最后一个调用。

删除一个设备可能产生删除一系列其他设备的需要，PnP 管理器负责协调从顶层向下到根设备级删除另外的设备对象。

这一部分描述了：

3.3.1.1 删除功能驱动程序里的一个设备

3.3.1.2 删除过滤器驱动程序里的一个设备

3.3.1.3 删除总线驱动程序里的一个设备

3.3.3.1 删除功能驱动程序里的一个设备

当删除一个设备时，一个功能驱动程序必须取消它以前执行的用于添加和启动设备的任何操作。这部分讨论包括外设的功能驱动程序和总线设备的功能驱动程序。

一个功能驱动程序在它的 DispatchPnP 例程里使用一个如下的过程删除一个设备：

1. 这是一个总线设备的一个功能驱动程序吗？

   如果是这样的话，可能地删除总线上的设备任何未完成的子 PDO。

   如果总线驱动程序处理子设备的一个从前的 IRP_MN_SURPRISE_REMOVAL 请求，但是驱动程序还没有接收到随后的 IRP_MN_REMOVE_DEVICE 请求时，驱动程序保留子 PDO 不动。在以后的某个时候，当子设备的所有处理关闭时，PnP 管理器将发送子设备的删除 IRP，并且，总线驱动程序将在此时删除子 PDO。

   如果总线驱动程序处理设备一个从前的 IRP_MN_REMOVE_DEVICE 请求，且驱动程序还没有接收到随后的 IRP_MN_SURPRISE_REMOVAL 请求时，总线驱动程序删除子 PDO。在这种情况下，PnP 管理器保证在它发送一个删除 IRP 给父总线设备之前，任何功能和过滤器驱动程序已经从子设备（FDO 和过滤程序 DO 已经被删除）删除。子 PDO 可能仍旧存在，这样总线驱动程序必须在它删除总线设备之前删除子 PDO。

2. 是否驱动程序已经处理过这个 FDO 从前的一个 IRP_MN_SURPRISE_REMOVAL 请求？

   如果是这样的话，执行任何尚存的清理并跳到第七步，**IoCallDriver**。

   一个驱动程序在设备扩展里典型地维护一个标记，这个标记说明了是否驱动程序已经为设备处理过一个 IRP_MN_SURPRISE_REMOVAL 请求。

3. 保证设备不运行。

如果在响应一个前面的 IRP_MN_QUERY_REMOVE_DEVICE 时设备还不运行的话，驱动程序必须标记设备以便不接收新的请求，并且必须完成在驱动程序里排队的任何请求。驱动程序必须使任何需要访问设备的未完成请求失效。

一个驱动程序能够使用 **Io..RemoveLock..**例程来枚举待定的 I/O，并设置一个事件来表明删除处理能够继续。参见《Windows2000 驱动程序开发参考》的卷一获得这些例程的信息。

4. 执行一些掉电操作。

设备的每个驱动程序执行它的掉电操作，如果有，当它接收到 IRP_MN_REMOVE_DEVICE IRP 时。设备的电源策略所有者，典型地为功能驱动程序，并不发送一个独立的 IRP_MN_SET_POWER IRP 来设置电源状态为 D3。父总线驱动程序典型地使插槽掉电，并当总线驱动程序得到删除 IRP 时，用 **PoSetPowerState** 通知电源管理器。看第 3 部分获得电源管理的更多信息。

5. 用 **IoSetDeviceInterfaceState** 禁止任何设备接口。

6. 为驱动程序所使用的设备释放任何硬件资源。

确切的操作取决于设备和驱动程序，但是也包括用 **IoDisconnectInterrupt** 断开一个中断，用 **MmUnmapIoSpace** 释放物理地址范围，并释放 I/O 端口。

7. 向下传递 IRP_MN_REMOVE_DEVICE 请求给下一个驱动程序。

用 **IoSkipCurrentIrpStackLocation** 为下一个较低层的驱动程序设定堆栈位置，并用 **IoCallDriver** 传递 IRP 给下一个驱动程序。

在继续它的删除行为之前，一个驱动程序不是非要等待基本的驱动程序来完成它们的删除操作。

8. 用 **IoDetachDevice** 从设备堆栈删除设备对象。

给下一个较低层的设备对象如 *TargetDevice* 参数指定一个指针。驱动程序在它的 AddDevice 例程里的 **IoSkipCurrentIrpStackLocation** 调用接收到这样的一个指针。

9. 清除设备专用的分配、内存、事件等等。

10. 用 **IodeleteDevice** 释放 FDO。

11. 从 DispatchPnP 例程里返回，并从 **IoCallDriver** 传播返回的状态。

一个功能驱动程序没有为一个删除 IRP 指定一个 IoCompletion 例程，也没有完成 IRP。删除 IRP 由父总线驱动程序完成。

3.3.3.2 删除过滤器驱动程序里的一个设备

当删除一个设备时，一个过滤器驱动程序必须取消它执行用于添加和启动设备的任何操作。当删除一个设备时，一个过滤器驱动程序遵从与功能驱动程序基本相同的步骤。

3.3.3.3 删除总线驱动程序里的一个设备

当删除一个子设备（子 PDO）时，父总线驱动程序必须取消它执行来添加和启动设备的任何操作。

一个总线驱动程序在它的 DispatchPnP 例程里用如下的步骤删除一个子设备：

1. 是否驱动程序已经处理过这个 PDO 的一个从前的请求？

如果是这样的话，执行任何尚存的清理并跳到第四步。

一个驱动程序在设备扩展里典型地维护一个标记，这个标记说明了是否驱动程序已经处理过设备的一个 IRP_MN_SURPRISE_REMOVAL 请求。

2. 完成在驱动程序里排队等待的任何请求。
3. 从设备删除电源，如果总线驱动程序能够这样做的话，通过调用 **PoSetPowerState** 通知电源管理器。

总线驱动程序掉电子设备，如果可能的话，通知电源管理器电源状态里设备的改变。总线驱动程序做这些来响应 IRP_MN_REMOVE_DEVICE IRP；当设备被删除时，设备的电源策略所有者并不发送一个 IRP_MN_SET_POWER IRP。看第 3 部分获得电源管理的更多信息。

4. 如果总线驱动程序在它最近为 **BusRelations** 而对 IRP_MN_QUERY_DEVICE_RELATIONS 请求的响应中报告过这个设备，设备仍旧物理地存在于机器上。在这种状况下，总线驱动程序将：

- 为设备保留 PDO 直到设备已经物理地删除
- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS
- 用 **IoCompleteRequest** 完成 IRP
- 从 DispatchPnP 例程返回

总线驱动程序继续在随后的枚举里（**BusRelations** 的 IRP_MN_QUERY_DEVICE_RELATIONS）报告这个设备，直到设备被物理地删除。PnP 管理器跟踪是否一个枚举的设备已经被添加和启动。

5. 如果设备以前不包括在总线驱动程序为了 **BusRelations** 而响应大部分最近的一个 IRP_MN_QUERY_DEVICE_RELATIONS 请求时，总线驱动程序认为设备将要物理地从机器上删除。在这样的状况下，总线驱动程序做下面的工作：

- 清除设备专用的分配、内存、事件等等
- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS
- 用 **IoCompleteRequest** 完成 IRP
- 用 **IoDeleteDevice** 释放 PDO

如果驱动程序过去忽略了它的大多数最近的 **BusRelations** 列表里的设备，总线驱动程序必须删除 PDO。如果一个用户再次把设备插入到机器里，总线驱动程序必须生成一个新的 PDO 来响应下一个 **BusRelations** 查询。如果一个总线驱动程序为一个新的设备实例重新利用相同的 PDO 时，机器将不能正常操作。

- 从 DispatchPnP 例程返回

如果当 PnP 管理器发送 IRP_MN_REMOVE_DEVICE 请求时设备仍旧存在，则总线驱动程序保留 PDO。如果后来的某个时候设备物理地从总线里删除，PnP 管理器发送另外的一个 IRP_MN_REMOVE_DEVICE。在接收到随后的删除 IRP 之后，总线驱动程序删除设备的 PDO。

总线驱动程序必须能够处理已经删除的设备的一个 IRP_MN_REMOVE_DEVICE，并且这个设备的 PDO 标记为删除。为响应这样的一个 IRP，总线驱动程序能够使 IRP 成功或者返回 STATUS_NO_SUCH_DEVICE。尽管总线驱动程序以前对 **IoDeleteDevice** 的调用，由于一些组件仍旧有这个对象的引用，设备的 PDO 在这种情况下还没有删除。这样，在处理第二个删除 IRP 过程中，总线驱动程序能够访问 PDO。总线驱动程序不可以为了 PDO 而两次调用 **IoDeleteDevice**；当 I/O 系统的引用计数为零时，它删除 PDO。

直到总线驱动程序接收到一个设备的一个 IRP_MN_REMOVE_DEVICE 请求时，它才删除一个子设备的数据结构。一个总线驱动程序可能发现一个设备已经删除并且调用 **IoInvalidateDeviceRelations**，但是，直到发送一个 IRP_MN_REMOVE_DEVICE 请求，它才可以

删除设备的 PDO。

3.3.4 处理一个 IRP_MN_CANCEL_REMOVE_DEVICE 请求

为响应一个 IRP_MN_CANCEL_REMOVE_DEVICE 请求，一个设备的驱动程序必须返回设备到这种状态，即驱动程序接收 IRP_MN_QUERY_REMOVE_DEVICE 请求之前的状态。典型地，驱动程序返回设备为启动的状态。

除过发送一个 IRP_MN_CANCEL_REMOVE_DEVICE 给设备之外，如果有的话，PnP 管理器发送 IRP 给设备已删除的相关设备。PnP 管理器也发送一个取消删除 IRP 给设备的子代。

在 IRP_MN_CANCEL_REMOVE_DEVICE 请求完成之后，PnP 管理器调用任何 **EventCategoryTargetDeviceChange** 通知回调。通过调用 **IoRegisterPlugPlayNotification**，这样的回调注册在设备上。PnP 管理器也通过调用 **RegisterDeviceNotification** 来调用为这样的通知所注册的任何用户模式组件。

一个 IRP_MN_CANCEL_REMOVE_DEVICE 请求必须首先被一个设备的父总线驱动程序所处理，然后被设备堆栈里每个更高级的驱动程序所处理。一个驱动程序在它的 DispatchPnP 例程里处理删除 IRP。

一个驱动程序在它的 DispatchPnP 例程里使用一个如下的步骤处理一个 IRP_MN_CANCEL_REMOVE_DEVICE 请求：

1. 在一个功能或过滤器驱动程序里，延迟启动设备直到更低层的驱动程序已经完成它们的重新启动操作。

   一个功能或过滤器驱动程序设置一个 IoCompletion 例程，向下传递 IRP_MN_CANCEL_REMOVE_DEVICE 到设备堆栈，并且延迟它的启动操作直到更低层的驱动程序已经处理完 IRP。（看第 2 章 2.3 节。）
2. 在一个更低层的驱动程序完成之后，返回设备到以前的 PnP 状态。

   驱动程序返回设备到在它接收到 IRP_MN_QUERY_REMOVE_DEVICE 请求之前的状态。典型地，驱动程序返回设备为启动的状态。确切的操作取决于设备和驱动程序。
3. 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS 和用 **IoCompleteRequest** 完成 IRP。

   当与任何 PnP IRP 一起时，一个总线驱动程序完成 IRP。

   一个功能或过滤器驱动程序也完成了 IRP，在这种状况下，由于驱动程序的 IoCompletion 例程通过返回 STATUS_MORE_PROCESSING_REQUIRED 中断了完成处理。

驱动程序必须使这个 IRP 成功，如果任何驱动程序使这个 IRP 失效，设备将处于一个不一致的状态。

当设备启动和运行时，一个驱动程序也许会接收到一个伪取消删除请求。这种情况可能发生，例如，如果一个驱动程序（或设备堆栈里的一个更高级的驱动程序）在过去使一个 IRP_MN_QUERY_REMOVE_DEVICE 请求失效过。当一个设备启动和运行时，一个驱动程序简单地使一个设备的伪取消删除请求成功。

3.3.5 处理一个 IRP_MN_SURPRISE_REMOVAL 请求

Windows2000 PnP 管理器发送这个 IRP 来通知驱动程序一个设备已经意外地从机器中被删

除，并且将不再能够处理 I/O（“突然的”删除）。

所有的 PnP 驱动程序必须处理这个 IRP，适当的 IRP 操作能够使驱动程序和 PnP 管理器来做下面的事情：

1. 释放分配给设备的硬件资源并使它们对另一个设备可用。

只要一个设备从机器中物理地删除，它的硬件资源就应该被释放。然后，PnP 管理器能够重新分配资源给另一个设备，包括相同的设备，这可能是一个用户热插回到机器中的设备。

2. 最小化数据丢失和系统崩溃的风险。

当一个用户删除一个热插拔设备而没有首先使用 Unplug 或 Eject Hardware 小应用程序，突然删除能够挂起机器，使机器崩溃，和/或引起数据丢失。

避免这样崩溃最好的方法是对设备厂商来说为热插拔设备实现一个锁定机制。这样一个机制保证设备在一个受控制的形式中删除。PnP 管理器警告访问设备的组件正在删除设备，这样它们能够执行有序的清除和删除。

假如一个设备没有一个锁定机制，或者它被附着到另外一个没有锁定的设备上，它的驱动程序必须处理一个 IRP_MN_SURPRISE_REMOVAL 请求。处理这个 IRP 没有阻止一个突然的删除崩溃，但是只要删除被检测到且驱动程序处理了 IRP，它就通过停止设备的 I/O 而最小化了影响。

在下面的任一方法中一个突然的删除被检测：

- 如果一个总线有热插拔通知，它将设备已经消失的消息通知给设备的父总线驱动程序，总线驱动程序调用 **IoInvalidateDeviceRelations**。作为响应，PnP 管理器查询它的子设备的总线驱动程序（**BusRelations** 的 IRP_MN_QUERY_DEVICE_RELATIONS）。PnP 管理器确定设备不在子设备的新列表里，并且初始化设备的突然删除操作。
- 总线因另外一个的原因被枚举，突然被删除的设备不包括在它的子设备列表里。PnP 管理器初始化它的突然删除操作。
- 设备的功能驱动程序判定设备不再存在（例如，由于它的请求重复地超时），总线也许是可枚举的，但是它没有热插拔通知。在这种状况下，功能驱动程序调用 **IoInvalidateDeviceState**。作为响应，PnP 管理器给设备堆栈发送一个 IRP_MN_QUERY_PNP_DEVICE_STATE 请求。功能驱动程序在 PNP_DEVICE_STATE 的位屏蔽里设置 PNP_DEVICE_FAILED 标记来说明设备已经失效。

PnP 管理器在系统线程环境的 IRQL PASSIVE_LEVEL 上发送一个 IRP_MN_SURPRISE_REMOVAL。

在通知用户模式应用程序和其他的内核模式组件之前 PnP 管理器给驱动程序发送这个 IRP。在完成 IRP 之后，PnP 管理器用为设备上这个通知而注册的内核模式组件的 GUID_TARGET_DEVICE_REMOVE_COMPLETE，发送一个 **EventCategoryTargetDevice-Change** 通知。

IRP_MN_SURPRISE_REMOVAL IRP 首先被设备堆栈里顶层的驱动程序所处理，然后被每个下一个较低层的驱动程序所处理。所有的 PnP 驱动程序必须处理这个 IRP，并必须设置 STATUS_SUCCESS。

在驱动程序的 AddDevice 例程调用之后，一个 PnP 设备的驱动程序必须准备随时处理一个 IRP_MN_SURPRISE_REMOVAL。

为响应一个 IRP_MN_SURPRISE_REMOVAL，一个驱动程序必须做下面的工作：

1. 阻止设备的任何新的 I/O。

一个驱动程序应该处理随后的关闭、清除、电源和 PnP 请求，但是驱动程序必须阻止设备的任何新的 I/O。驱动程序使用后续的 IRP（如果设备存在，驱动程序将处理这些 IRP。）失效，包

括关闭、清除和 PnP IRP。

一个驱动程序能够在设备扩展里设置一个位来说明设备已经被突然的删除，驱动程序的调度例程应该检查这个位。

2. 失效设备上的未完成的 I/O 请求。
3. 继续向下传递任何驱动程序没有为设备处理的 IRP。
4. 在一个父总线驱动程序里，如果驱动程序能够这样做的话，掉电总线插槽。调用 **PoSetPowerState** 来通知电源管理器。
5. 用 **IoSetDeviceInterfaceState** 使设备接口不可用。
6. 释放设备的硬件资源（中断、I/O 端口、内存寄存器和 DMA 通道）。
7. 清除任何设备专用的分配、内存、事件等等。

一个驱动程序可能延迟这样的清除直到它接收到随后的 IRP_MN_REMOVE_DEVICE 请求，但是如果一个早期的组件有一个不能够被关闭的打开句柄，删除 IRP 将永不会被发送。

8. 使设备对象附着到设备堆栈。

   不分离和删除设备对象直到随后的 IRP_MN_REMOVE_DEVICE 请求为止。
9. 完成 IRP。

   在一个功能或过滤器驱动程序里：

- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。
- 用 **IoSkipCurrentIrpStackLocation** 定位下一个堆栈位置并用 **IoCallDriver** 传递 IRP 给下一个较低层驱动程序。
- 当从 DispatchPnP 例程返回状态时，从 **IoCallDriver** 传播此状态。
- 不要完成 IRP。

  在一个总线驱动程序里（该驱动程序处理这个 IRP 的一个子 PDO）：

- 设置 **Irp->IoStatus.Status** 为 STATUS_SUCCESS。
- 用 IO_NO_INCREMENT 完成 IRP（**IoCompleteRequest**）。
- 从 DispatchPnP 例程返回。

在这个 IRP 成功以及设备的所有的打开句柄关闭之后，PnP 管理器发送一个 IRP_MN_REMOVE_DEVICE 请求给设备堆栈。为响应这个删除 IRP，驱动程序从堆栈里分离它们的设备对象并删除它们。如果一个早期的组件有一个设备的打开句柄，尽管 I/O 失效，却让句柄打开，则 PnP 管理器从不发送删除 IRP。

所有的驱动程序应该处理这个 IRP，并注意到设备已经物理地从机器中删除。然而，如果一些驱动程序没有处理 IRP，它们将不会引起不利的结果。举例而言，一个没有消耗系统硬件资源并留存于一个协议基础的总线上的设备，如 USB 或 1394，由于它没有消耗任何资源，所以不能够与硬件资源连接。由于功能和过滤器驱动程序仅仅通过父总线驱动程序访问设备，在设备已经被删除之后，驱动程序试图访问设备的努力没有危险。由于总线支持删除通知，当设备消失和总线驱动程序使随后所有的访问设备的努力失效时，通知父总线驱动程序。

Windows98 PnP 管理器没有发送这个 IRP。如果一个用户删除一个设备而没有首先使用适当的用户接口，PnP 管理器仅仅发送一个 IRP_MN_REMOVE_DEVICE 请求给设备的驱动程序。所有的 WDM 驱动程序必须处理 IRP_MN_SURPRISE_REMOVAL 和 IRP_MN_REMOVE_DEVICE。IRP_MN_REMOVE_DEVICE 的代码应该检查是否一个驱动程序接收到一个前期的突然删除 IRP，并应该处理这两种情况。

第4章　使用 PnP 通知

在一个即插即用（PnP）的环境里，驱动程序和应用程序需要对机器里的设备配置的变化作出反应。例如，一个应用程序需要知道何时感兴趣的设备已经被添加到机器上，而一个驱动程序需要知道何时一个变化发生在一个特定的设备上。

当特定的 PnP 事件发生时，PnP 管理器提供一个机制来通知驱动程序和应用程序。。本章描述了在内核模式代码里怎样使用 PnP 通知。用户模式应用程序的编写者应该看平台 SDK 文档获得关于 **RegisterDeviceNotification** 功能和相关的功能的信息。

本章包括以下的主题：

4.1　PnP 通知总览
4.2　编写 PnP 通知回调例程指南
4.3　使用 PnP 设备接口改变通知
4.4　使用 PnP 目标设备改变通知
4.5　使用 PnP 硬件 profile 改变通知
4.6　使用 PnP 定制通知

参见《Windows2000 驱动程序开发参考》的卷一获得当一个设备的电源状态变化时关于使用 **PoRegisterDeviceNotify** 来为通知注册。为了了解系统状态、时间和策略变化的通知的有关内容，请看内核模式驱动程序设计指南里有关系统定义的回调的信息。

4.1 PnP 通知总览

当特定的事件发生在指定的设备上或者在整个系统上时，PnP 管理器提供一个机制来通知的驱动程序和应用程序。一个驱动程序能够为下面种类的事件的通知而注册。

- **EventCategoryInterfaceChange**

当一个驱动程序为设备接口上这个种类的事件而注册时，PnP 管理器通知驱动程序下面的事件：

- GUID_DEVICE_INTERFACE_ARRIVAL

表明一个特定类的设备接口已经启用。例如，一个用户给机器添加一个新的硬盘和卷管理器使一个新的卷可用（类“卷”的一个设备接口）。

- GUID_DEVICE_INTERFACE_REMOVAL

表明指定类的一个设备接口已经被禁止。

在《Windows2000 驱动程序开发参考》的卷一中参看 **IoRegisterDeviceInterface** 和相关的例程获得设备接口的更多信息。

- **EventCategoryTargetDeviceChange**

当一个驱动程序为设备上这类事件注册时，当在设备上发生下面的事件时，PnP 管理器通知驱动程序：

- GUID_TARGET_DEVICE_QUERY_REMOVE

表明 PnP 管理器准备删除设备的驱动程序。几种操作能够引起这个事件，包括一个用户请求

从机器上删除指定的设备，或者一个用户发布设备的一个更新驱动程序请求。这个通知请求设备的驱动程序或者支持或者否决到来的删除操作。

- GUID_TARGET_DEVICE_REMOVE_COMPLETE

表明指定的设备已经从机器中删除，或者一个用户正在改变设备的驱动程序。

- GUID_TARGET_DEVICE_REMOVE_CANCELLED

表明指定的设备上的一个到来的删除操作已经被取消。

- GUID_Xxx（定制事件）

表明一个定制事件已经发生在一个指定的设备上。

你能够定义设备的一个定制事件。当驱动程序（或另外的相关组件）通知 PnP 管理器定制事件已经发生时，PnP 管理器通知设备上为目标设备的改变通知而注册的任何组件。

不象设备接口的改变的注册过程，它可被认为是设备接口里一个“被动的”权利，而为目标设备的改变的注册表明设备里一个“主动的”权利。

- **EventCategoryHardwareProfileChange**

本类包括下面的事件：

- GUID_HWPROFILE_QUERY_CHANGE

表明一个用户已经请求改变机器的硬件 profile。PnP 管理器使用这个通知询问已注册的组件，是否它能够改变硬件 profile 而没有中断系统操作。已注册的组件典型地使这些查询请求成功。

- GUID_HWPROFILE_CHANGE_COMPLETE

表明一个机器的硬件 profile 已经改变。如果一个驱动程序维护 profile 专用的设置，它应该在一个硬件 profile 改变之后更新这些设置。

- GUID_HWPROFILE_CHANGE_CANCELLED

表明一个到来的硬件 profile 改变已经被取消。

为了内核模式组件，PnP 通知如下工作：

1. 通过调用 **IoRegisterPlugPlayNotification**，驱动程序注册一个种类事件的通知。

   一个 PnP 通知的回调例程保留被注册状态直到驱动程序明确地删除注册。
2. 当已注册的种类里有一个事件发生时，PnP 管理器调用驱动程序的回调例程。
3. 驱动程序通过调用 **IoUnregisterPlugPlayNotification** 删除回调注册。

对一个驱动程序来说，在一个关闭的过程中生成一个同步事件或者等待一个同步事件是非法的。

为获得 PnP 通知的更多信息，看本章里下面的部分：

4.1 编写 PnP 通知回调例程指南
4.2 使用 PnP 设备接口改变通知
4.3 使用 PnP 目标设备改变通知
4.4 使用 PnP 硬件 profile 改变通知
4.5 使用 PnP 定制通知

4.2 编写 PnP 通知回调例程的指南

PnP 管理器在 IRQL PASSIVE_LEVEL 上调用通知回调例程。

为保证 PnP 子系统的平稳操作，一个 PnP 通知回调例程必须遵循下面的指南：

1. 一个通知回调例程不能够阻塞。

2. 一个通知回调例程不能够调用、或者引起对同步例程或任何例程的调用。在这里，同步例程生成 PnP 事件；这些任何例程则阻塞了等待设备安装或删除。

在一个通知回调过程中调用这样的例程能够引起系统死锁。

举个例子，一个驱动程序不能够在一个通知回调例程里调用 **IoReportTargetDeviceChange**，相反，它应该调用 **IoReportTargetDeviceChangeAsynchronous**。

3. 一个通知回调例程应该为它没有明确失效的任何事件返回成功状态。

当一个驱动程序为一个事件种类上的通知注册时，PnP 管理器通知驱动程序那个种类里的所有事件，包括当前的和将来的。如果一个驱动程序为它没有处理的事件返回一个错误状态，驱动程序因错误而将冒险使一个新的查询事件失效。

例如，当驱动程序使一个查询通知失效而否决正被处理的事件时，一个驱动程序正确地返回一个错误状态。

4. 一个通知回调例程应该是可分页的代码。

4.3 使用 PnP 设备接口改变通知

一个驱动程序为 **EventCategoryDeviceInterfaceChange** 通知而注册，以便当机器上一个特别类的设备接口到达（可用的）或者被删除（不能用的）时，它能够被通知。例如，一个合成的电池驱动程序可能为电池类的设备接口的通知来注册，这样它就能够提供完整可用的电池电力信息给操作系统。

下面的子部分讨论怎样注册设备接口的改变通知和怎样在一个 PnP 通知回调例程里处理设备接口改变事件。

在《Windows2000 驱动程序开发参考》的卷一中参看 **IoRegisterDeviceInterface** 和相关的例程，来获得设备接口的有关信息。

4.3.1 为设备接口改变通知注册

通过调用 **IoRegisterPlugPlayNotification**，一个驱动程序为设备接口的到达和删除事件的通知注册。

在 Windows2000 驱动程序开发参考的卷一中参看 **IoRegisterPlugPlayNotification** 获得关于调用例程的总体信息，下面的信息应用于设备接口改变通知而调用这个例程：

- 指定 **EventCategoryDeviceInterfaceChange** 的一个 *EventCategory*。
- *EventCategoryData* 必须指向设备接口类的 GUID。

一个接口类的 GUID 在典型情况下带有支持接口的结构、常数等等的头文件里被定义。

- 指定 PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES 的一个 *EventCategoryFlags*。

这个标记指示 PnP 管理器，为指定类的设备接口在将来的到达和离开而注册回调例程，并立即为已经运行的相关设备接口调用回调例程。

一个驱动程序能够调用 **IoGetDeviceInterfaces** 来得到一个特定类存在的接口的一个列表，然后注册它的回调例程而不用这个标记，但是使用标记变得更容易，且避免了潜在的时钟问题。

- 指定一个驱动程序定义的环境，如果适当的话，PnP 管理器将传递它给回调例程。

打开一个设备的句柄来响应设备接口的到达通知的驱动程序，将在设备上为 **EventCategoryTargetDeviceChange** 事件注册。（参见 4.4 节。）

一个驱动程序通过用由 **IoRegisterPlugPlayNotification** 返回的 *NotificationEntry* 调用 **IoUnregisterPlugPlayNotification** 来取消通知注册。

4.3.2 处理设备接口改变事件

当一个驱动程序或者一个用户模式组件能够允许或者禁止一个设备接口，PnP 管理器调用设备接口类上为 **EventCategoryDeviceInterfaceChange** 注册的任何通知回调例程。PnP 管理器指定 GUID_DEVICE_INTERFACE_ARRIVAL 或 GUID_DEVICE_INTERFACE_REMOVAL 的一个 *NotificationStructrue.***Event**。

当处理一个 GUID_DEVICE_INTERFACE_ARRIVAL 事件时，为了处理新的接口，一个通知回调例程应该执行驱动程序定义的任务。

例如，一个回调例程可能排队一个工作者例程来打开设备。一个回调例程不必直接打开设备。如果接口的提供者引起了阻塞 PnP 事件，且如果通知回调例程在回调线程里试图打开设备时，它能够引起一个锁死。

一个回调例程可能使它自己的一个设备接口能够响应新设备接口的可用性。

当处理一个 GUID_DEVICE_INTERFACE_REMOVAL 事件时，当接口启用时，一个通知回调例程应该取消它执行的任何操作。

4.4 使用 PnP 目标设备改变通知

一个驱动程序在一个设备上为 **EventCategoryTargetDeviceChange** 通知而注册，这样当驱动程序将要被删除时，驱动程序能够被通知。例如，如果一个驱动打开设备的一个句柄，驱动程序应该在一个设备上为 **EventCategoryTargetDeviceChange** 通知而注册，这样当 PnP 管理器需要删除设备时，驱动程序能够关闭它的句柄。

驱动程序也能够为定制通知使用 **EventCategoryTargetDeviceChange** 通知。（参见 4.6 节）

下面的子部分讨论了怎样注册目标设备的改变通知，和怎样在一个 PnP 通知回调例程里处理目标设备改变事件。

4.4.1 注册目标设备改变通知

4.4.2 处理一个 GUID_TARGET_DEVICE_QUERY_REMOVE 事件

4.4.3 处理一个 GUID_TARGET_DEVICE_REMOVE_COMPLETE 事件

4.4.4 处理一个 GUID_TARGET_DEVICE_REMOVE_CANCELLED 事件

参见第 3 章当 PnP 管理器发送删除 IRP 给一个设备的驱动程序时，启动、停止和删除设备的有关描述。

4.4.1 注册目标设备改变通知

通过调用 **IoRegisterPlugPlayNotification**，一个驱动程序注册了目标设备改变事件的通知。

在《Windows2000 驱动程序开发参考》的卷一中阅读 **IoRegisterPlugPlayNotification** 获得调

用例程的总体信息。下面的信息应用于目标设备改变通知的调用例程：

- 指定 **EventCategoryTargetDeviceChange** 的一个 *EventCategory*。
- *EventCategoryData* 必须为了设备指向文件对象，在这个设备上通知被请求。

如果驱动程序的回调例程需要访问文件对象，驱动程序应该在调用 **IoRegisterPlugPlayNotification** 之前删除文件对象上的一个引用。

如果驱动程序的回调例程不需要访问文件对象，驱动程序不许要引用该对象。

在文件对象关闭之后，驱动程序继续为设备接收通知直到驱动程序删除它的通知注册。举个例子，这个设计允许驱动程序接收 GUID_TARGET_DEVICE_REMOVE_CANCELLED 事件的通知。

- 指定一个驱动程序定义的、PnP 管理器将传递给回调例程的环境。

一个驱动程序可能使用环境参数来维护文件对象当前的状态信息（如是否它已经被关闭/删除）。

一个驱动程序也可能使用环境来存储它最初用来打开设备的路径。在一个取消的删除操作之后，一个驱动程序能够使用这个例程重新打开设备。（参见 4.4.4 节获得更多信息。）

一个驱动程序通过用由 **IoRegisterPlugPlayNotification** 返回的 *NotificationEntry* 调用 **IoUnregisterPlugPlayNotification** 来删除一个通知注册。如果过去当驱动程序为通知注册时已经删除文件对象上的一个引用，且该引用仍旧未完成，驱动程序必须在它删除注册表之后释放引用。

4.4.2 处理一个 GUID_TARGET_DEVICE_QUERY_REMOVE 事件

在 PnP 管理器给一个设备的驱动程序发送一个 IRP_MN_QUERY_REMOVE_DEVICE IRP 之前，它调用设备上为 **EventCategoryTargetDeviceChange** 而注册的任何通知回调例程。PnP 管理器指定 GUID_TARGET_DEVICE_QUERY_REMOVE 的一个 *NotificationStructrue.***Event**。
作为这样的一个通知的响应，回调例程决定设备是否可被删除而没有中断系统。

如果不应该删除设备，回调例程返回 STATUS_SUCCESSFUL。为响应这个状态，PnP 管理器放弃查询删除处理，且设备不会被删除。

如果能够被删除设备，回调例程应该执行任何适当的操作来准备删除设备，如关闭设备上的任何句柄打开（如果可能的话）。如果在设备上句柄维持打开状态，PnP 管理器不能够删除设备并放弃查询删除处理。

当成功地处理一个 GUID_TARGET_DEVICE_QUERY_REMOVE 事件时，一个通知回调例程应该：

- 关闭设备的任何打开句柄。
- 如果驱动程序在文件对象上有一个未完成的引用，间接访问文件对象。
- 保留将来的 **EventCategoryTargetDeviceChange** 通知的注册状态，由于到来的删除操作也许会取消，所以这是重要的。

关闭设备的一个句柄没有取消为 PnP 目标设备改变通知而注册一个驱动程序。PnP 管理器仍旧能够调用驱动程序的通知回调例程，但是在这样的调用里，在 *NotificationStructure* 里的文件对象是不合法的。

4.4.3 处理一个 GUID_TARGET_DEVICE_REMOVE_COMPLETE 事件

PnP 管理器在给一个设备的驱动程序发送 IRP_MN_REMOVE_DEVICE IRP 之前，它调用设备上为 **EventCategoryTargetDeviceChange** 注册的任何内核模式通知回调例程。PnP 管理器指定 GUID_TARGET_DEVICE_REMOVE_COMPLETE 的一个 *NotificationStructrue.***Event**。

当处理一个 GUID_TARGET_DEVICE_REMOVE_COMPLETE 事件时，一个通知回调例程应该：

- 删除设备上的通知注册。

  设备已经删除，这样驱动程序调用 **IoUnregisterPlugPlayNotification** 来删除通知注册。

  机器中设备可能仍旧物理地存在，但是所有的设备对象已经被删除，且设备是不可用的。
- 如果驱动程序没有接收到一个从前的查询删除通知时，执行突然删除处理。

如果一个驱动程序是突然的删除，PnP 管理器给已注册的驱动程序发送一个删除完成通知而不必有一个先前的查询删除通知。在这种状况下，一个驱动程序必须执行任何必要的清除，如关闭设备的任何句柄并删除任何文件对象的未完成引用。

4.4.4 处理一个 GUID_TARGET_DEVICE_REMOVE_CANCELLED 事件

如果一个 IRP_MN_QUERY_REMOVE_DEVICE 请求失效，PnP 管理器给设备的驱动程序发送一个 IRP_MN_CANCEL_REMOVE_DEVICE IRP。在取消删除 IRP 成功完成之后，调用设备上为 **EventCategoryTargetDeviceChange** 注册的任何通知回调例程。PnP 管理器指定一个 GUID_TARGET_DEVICE_REMOVE_CANCELLED 的 *NotificationStructrue.***Event**。

当处理一个 GUID_TARGET_DEVICE_REMOVE_CANCELLED 事件时，一个通知回调例程应该为目标设备通知注册。

由于驱动程序关闭了以前的注册句柄来响应查询删除通知，驱动程序必须打开一个新的句柄。驱动程序必须做下面的事情：

1. 用 **IoUnregisterPlugPlayNotification** 删除旧的注册。
2. 打开设备的一个新的句柄。
3. 用 **IoRegisterPlugPlayNotification** 为新的句柄上的通知而注册。

4.5 使用 PnP 硬件 profile 改变通知

一个驱动程序为 **EventCategoryTargetDeviceChange** 通知注册，以便当机器从一个硬件 profile 转变为另一个时驱动程序能够被通知，例如，当一个袖珍计算机被连接或断开连接时，一个驱动程序能够使用这个机制被通知。

下面的子部分讨论怎样注册硬件 profile 改变通知和怎样在一个 PnP 通知回调例程里处理硬件

profile 改变事件。
4.3.1 注册硬件 profile 改变通知
4.3.2 处理硬件 profile 改变事件

4.5.1 注册硬件 profile 改变通知

一个驱动程序通过调用 **IoRegisterPlugPlayNotification** 来注册硬件 profile 的改变通知。

在《Windows2000 驱动程序开发参考》的卷一中参看 **IoRegisterPlugPlayNotification** 获得调用例程的总体信息。下面的信息应用于为硬件 profile 改变通知而调用这个例程：

- 指定 **EventCategoryHardwareProfileChange** 的一个 *EventCategory*。
- *EventCategoryData* 必须为 NULL。
- 指定一个驱动程序定义的环境，如果适当，PnP 管理器将传递它给回调例程。

一个驱动程序通过用由 **IoRegisterPlugPlayNotification** 返回的 *NotificationEntry* 调用 **IoUnregisterPlugPlayNotification** 来删除通知注册。

4.5.2 处理硬件 profile 改变事件

在一个硬件 profile 改变的特定时间，PnP 管理器调用为 **EventCategoryHardwareProfileChange** 而注册的回调例程：

- 在机器的硬件 profile 改变之前，PnP 管理器调用已注册的通知回调例程，并指定 GUID_HWPROFILE_QUERY_CHANGE 的一个 *NotificationStructrue.***Event**。
- 在一个机器的硬件 profile 改变完成之后，PnP 管理器调用已注册的通知回调例程，并指定 GUID_HWPROFILE_CHANGE_COMPLETE 的一个 *NotificationStructrue.***Event**。
- 如果机器的硬件 profile 改变取消，PnP 管理器调用已注册的通知回调例程，并指定 GUID_HWPROFILE_CHANGE_CANCELLED 的一个 *NotificationStructrue.***Event**。

对一个 GUID_HWPROFILE_QUERY_CHANGE 事件，PnP 管理器调用用户模式回调例程，然后调用内核模式回调例程。为响应一个 GUID_HWPROFILE_QUERY_CHANGE 事件，典型情况下，一个驱动程序的通知回调例程返回 STATUS_SUCCESS。

对一个 GUID_HWPROFILE_CHANGE_COMPLETE 事件，PnP 管理器调用内核模式回调例程，然后调用用户模式回调例程。为响应这样的一个事件，一个驱动程序的回调例程可能更新它的硬件 profile 专用设置。

对一个 GUID_HWPROFILE_CHANGE_CANCELLED 事件，PnP 管理器调用内核模式回调例程，然后调用用户模式回调例程。为响应这样的一个事件，典型情况下，一个驱动程序的回调例程返回 STATUS_SUCCESS。如果驱动程序执行任何操作来响应 GUID_HWPROFILE_QUERY_CHANGE 事件，驱动程序取消这些操作来响应取消事件。

4.6 使用 PnP 定制通知

通知能够使用目标设备改变通知机制的驱动程序设备上的定制事件。

程序编辑器定义了定制事件必须作下面的事情：

1. 为定制事件定义一个新的 GUID。

用 *uuidgen* 或 *guidgen* 生成，在一个适当的头文件和文档里发布 GUID。

2. 编写代码来触发定制事件。

在内核模式里，一个驱动程序用定制 GUID 和设备的一个 PDO 指针调用 **IoReportTargetDeviceChange**，定制事件仅能够从内核模式被触发。

一个驱动程序编写者在一个如下的步骤里使用定制通知：

1. 驱动程序（或应用程序）为定制事件的通知注册。

在内核模式里，一个驱动程序调用 **IoRegisterPlugPlayNotification**，并为设备上的一个 **EventCategoryTargetDeviceChange** 注册。

在用户模式里，一个应用程序注册使用 **RegisterDeviceNotification**。参见 SDK 平台获得更多的信息。

2. 一个内核模式组件触发定制事件。
3. PnP 管理器调用设备上已注册的通知例程。

它先调用被注册的用户模式回调例程，然后调用内核模式回调例程。

4. 当用户模式通知完成时，内核模式驱动程序通知回调例程响应该定制事件。

参看 4.2 节获得通知回调例程的总体指南，除过这些指南外，一个定制通知回调例程不必从回调例程线程内打开一个设备的句柄。

第5章　支持多功能设备

一个多功能设备占用其父总线上的一个位置，但是有不止一个功能。在一个多功能设备里，单个的功能是独立的：它们不能够有启动顺序依赖；一个功能的资源需求不能够按照另外一个功能资源表达（例如，功能 1 使用 I/O 端口 X 和功能 2 使用端口 X+200）；且每个功能必须能够作为一个分割的设备被操作，即使它作为另一个功能驱动程序而被相同的驱动程序所服务。

为了一个多功能设备在微软的 Windows2000 上正确地配置，设备上的每个功能必须被枚举并且每个功能的资源需求必须与 PnP 管理器通信。这里也必须有 INF 文件和每个功能的驱动程序 。负责这些的任务的每个组件依靠设备的父总线的多功能标准，设备遵照标准决定其依赖的范围，且其能力取决于父总线驱动程序。

本章包括下面的部分：

5.1　支持多功能 PC Card 设备
5.2　支持多功能 PCI 设备
5.3　支持其他总线上的多功能设备
5.4　使用系统提供的 mf.sys
5.5　为多功能设备建立资源图

在《Windows2000 驱动程序开发参考》的卷一第 3 部分里参看 INF 文件部分和命令来获得 INF 文件语义的全面信息。

参看本指南的第 1 部分获得诸如功能驱动程序、总线驱动程序和它们相关联的设备对象等的 PnP 术语信息。

本章描述了怎样在 Windows2000 平台上支持多功能设备。

5.1 支持多功能 PC Card 设备

PC Card 多功能标准说明一个多功能设备为每个功能在属性内存里有一套配置寄存器。这些寄存器允许 PCMCIA 总线驱动程序，例如，能够使每个功能独立并定义一个不包括在每个功能里的 I/O 资源的范围。该标准也说明一个多功能设备在属性内存里包含每套配置寄存器的地址。这些地址能够使 PCMCIA 总线驱动程序对配置寄存器编程。

如果一个 16 位 PC Card 设备完全地和正确地执行 PC Card 多功能标准，这样的一个设备的厂商有最小的 INF 和驱动程序需求来保证设备在 Windows2000 上正确配置。参见 5.1.1 节获得更多的信息。

如果一个 16 位 PC Card 设备没有完全执行 PC Card 多功能标准，厂商必须在 INF 文件里提供丢失的信息。有两个方法可以使一个 PC Card 多功能设备可能不执行多功能标准：

1. 设备为每个功能实现一套多功能配置寄存器，但是不包含所有套的寄存器在它的属性内存里的定位。
2. 设备没有为每个功能实现一套多功能配置寄存器。

如果一个设备有以上列出的限制，且如果设备的 INF 在 *DDInstall*.**LogConfigOverride** 部分里有必须的信息，PCMCIA 总线驱动程序能够编程配置寄存器。阅读下面的部分获得更多的信息：

5.1.2　支持有不完全配置寄存器地址的 PC Card

5.1.3　支持有不完全配置寄存器的 PC Card

插件总线设备主要地遵循多功能规则，参见支持多功能 PC Card 设备。

5.1.1 支持遵照多功能标准的 PC Card

如果一个 16 位 ISA 类型的 PC Card 设备完全地和正确地执行 PC Card 多功能标准，这样的一个设备的厂商在 Windows2000 上能够依靠下面系统提供的组件处理多功能语义的软件方面：

1. 支持多功能设备的一个 INF 文件。（系统提供的）

PCMCIA 总线驱动程序为设备指定了一个设备硬件 ID，这引起配置管理器使用系统提供的多功能 INF 文件（*mf.inf*）来配置设备。*mf.inf* 文件指定了类“多功能”和它相关联的 GUID（如在 *devguid.h* 里定义的）。

2. 支持多功能设备的一个功能驱动程序。（系统提供的）

mf.inf 文件指定了系统提供的多功能总线驱动程序（*mf.sys*）作为支持设备的功能驱动程序。

mf.sys 总线驱动程序枚举设备的功能数目，PCMCIA 总线驱动程序读设备上的配置寄存器来决定每个功能的资源需求。

参见 5.4 节获得使用系统提供的 *mf.sys* 驱动程序的更多信息。

一个遵照标准的多功能 PC Card 设备的厂商必须为单个的功能提供下面的支持：

1. 设备的每个功能有一个 PnP 功能驱动程序。（厂商提供的）

由于多功能总线驱动程序处理多功能语义，因此功能驱动程序和这个功能被包装为单独的设备而使用的驱动程序相同。

2. 设备的每个功能一个 INF 文件。（厂商提供的）

如果功能被打包为单个的设备的话，INF 文件能够是将被使用的相同的文件。INF 文件不需要任何特别的多功能语义。

PCMCIA 总线驱动程序建立子功能硬件 ID

对一个真的多功能 PC Card 设备，PCMCIA 总线驱动程序与 *mf.sys* 一道，生成子设备功能的硬件 ID，这些 ID 有下面的格式：

<Manufacturer-name>-<Producu-ID-string>-DEV<number>-CRC

在这个格式里，<*number*>是功能从 0 开始的数字。

例如，PCMCIA 总线驱动程序生成如下的子设备硬件 ID：

3COM_Coporation-3C562D/3C563D-DEV0-4893

3COM_Coporation-3C562D/3C563D-DEV1-4893

一个多功能 PC Card 设备的一个子功能 INF 文件必须指定硬件 ID，该硬件 ID 由 PCMCIA 总线驱动程序和 *mf.sys* 报告。

5.1.2 支持有不完全配置寄存器地址的 PC Card

Register Address

如果一个多功能 16 位 PC Card 设备有为每个功能的配置寄存器，但是在属性内存里不包含所有寄存器设置（不支持 LONGLINK_MFC 元组）的指针，这样的设备厂商在 Windows2000 上能够

使用系统提供的多功能总线驱动程序（*mf.sys*），但是必须提供一个定制的 INF 文件和单个功能的支持。

这样的设备的厂商能够在 Windows2000 上为多功能设备使用系统提供的功能驱动程序。

设备的一个定制的 INF 必须指定 *mf.sys* 作为设备的功能驱动程序，系统提供的 *mf.sys* 驱动程序然后将枚举设备的功能。

参看 5.4 部分获得使用系统提供的 *mf.sys* 驱动程序的更多信息。

这样的设备的厂商在 Windows2000 上必须提供下面的设备：

1. 多功能设备的一个定制的 INF 文件。（厂商提供的）

厂商必须提供一个多功能 INF 文件，该文件指定 *mf.sys* 作为多功能总线驱动程序，指定类“多功能”（和在 *devguid.h* 里定义的相关联的 GUID），并提供丢失的配置寄存器地址。在本部分的后面看进一步的信息。

2. 设备的每个功能有一个 PnP 功能驱动程序。（厂商提供的）

由于多功能总线驱动程序处理多功能语义，当功能被打包为单个的设备的话，功能驱动程序能够与被使用的驱动程序相同。

3. 设备的每个功能有一个 INF 文件。（厂商提供的）

当功能被打包为单个的设备的话，INF 文件能够是相同的文件。INF 文件不需要任何特别的多功能语义。

这样一个多功能设备的定制的 INF 必须包含至少一个 *DDInstall*.**LogConfigOverride** 表项。重载部分必须包含每个功能的一个 **MfCardConfig** 表项，这样可以标识每套配置寄存器的位置。

如果由于重载 configs 存在于 INF，INF 必须重述由设备指定的所有资源需求。PnP 管理器没有使用设备的任何设备资源需求。

使用下面的语法指定 **MfCardConfig** 表项：

MfCardConfig=ConfigRegBase:ConfigOptions[:IoConfigIndex][(attrs)]

ConfigRegBase

为设备的一个功能指定配置寄存器属性偏移。

ConfigOptions

指定 8 位 PCMCIA 配置选项寄存器。

IoConfigIndex

可选择地给支持总线驱动程序的 **IOConfig** 资源描述符指定一个索引，使用它来为配置 I/O 基础编程和限定寄存器。该索引是基于 0 的，这样零代表着 *DDInstall*.**LogConfigOverride** 节里第一个 **IOConfig** 表项。

如果在配置选项寄存器里设置一位（0x02），那么一个 *IoConfigIndex* 必须指定。如果该位被清空，则该参数被忽略。

attrs

可选择地指定一套属性标记，可能的标记包括：

A

在配置和状态寄存器里打开音频。

例如，为包含有一个调制解调器和一个网络适配器的多功能 PC Card 设备考虑一个定制 INF 下面的摘录：

; …

```
[DDInstall.LogConfigOverride]
LogConfig = DDInstall.Override0

[DDInstall.Override]
IOConfig        =    3F8-3FF                    ;   Com1
IOConfig        =    10@100-FFFF%FFF0           ;   NIC   I/O
IRQConfig       =    3,4,5,7,9,10,11            ;   IRQ
MemConfig       =    2000@0-FFFFFFFF%FFFFE000   ;   Memory   Descriptor   0
MemConfig       =    1000@0-FFFFFFFF%FFFFE000   ;   Memory   Descriptor   1
MfCardConfig    =    1000:47:0 (A)
MfCardConfig    =    1000:47:1
;…
```

例子给出了两个 **MfCardConfig** 表项，每一个为设备的每个功能。第一个 **MfCardConfig** 表项包含下面的信息：

1000(the ConfigRegBase)

指定在卡的属性内存里有一套配置寄存器位于偏移 0x1000。在这个例子中，这些寄存器里的信息描述了卡上调制解调器的功能。

47(the ConfigOptions)

为总线驱动程序指定十六进制的值来编程使配置选项寄存器位于 ConfigRegBase 偏移（0x1000）。

0(the IoConfigIndex)

指定这个功能的 I/O 资源被列在本部分里的第一个 **IOConfig** 表项。一个零索引表示第一个表项，在此例中即为“IOConfig=3F8-3FF”。

A(the attrs)

指示总线驱动程序能够为这个功能打开音频，对一个调制解调器来说是典型的。

第二个 **MfCardConfig** 表项包含关于设备上第二个功能的信息（在这个例子中即为网络适配器），这个表项指出这里有第二套配置寄存器位于偏移 0x1080。总线驱动程序将写值为 0x47 的 *ConfigOptions* 给这个功能的配置选项寄存器。*IoConfigIndex* 值为一指令总线驱动程序使用这部分里的第二个 **IOConfig** 表项（**IOConfig=**10@100-FFFF%FFF0）编程 I/O 基础，并限定这个功能的寄存器。

在 INF 里包括不止一个 *DDInstall*.**LogConfigOverride***N* 节，将指定有不止一个的非序列的 I/O 端口范围的选择。

如果设备使用一个不基于零的内存窗口，*DDInstall*.**LogConfigOverride***N* 节必须包括一个 **PcCardConfig** 表项。如果一个重载部分有一个 **MfCardConfig** 表项和 **PcCardConfig** 表项，PCMCIA 总线驱动程序在 **PcCardConfig** 表项里忽略了 *ConfigIndex* 值，并只是使用 *MemoryCardBaseN* 信息。看《Windows2000 驱动程序开发参考》卷一的 5.1.3 节获得有关 **PcCardConfig** 表项的进一步的信息。

5.1.3 支持有不完全配置寄存器的 PC Card

如果一个多功能 16 位 PC Card 设备没有每个功能的配置寄存器，这样的设备的厂商能够使用系统提供的多功能总线驱动程序（*mf.sys*），但是必须提供一个定制 INF 文件和支持单个的功能。

这样的设备的厂商在 Windows2000 上能够使用下面的系统提供的组件：

1. 多功能设备的一个功能驱动程序。（系统提供的）

设备的一个定制 INF 必须指定 *mf.sys* 作为设备的功能驱动程序。系统提供的 *mf.sys* 驱动程序然后将枚举设备的功能。

参见 5.4 节获得使用系统提供的 *mf.sys* 驱动程序的更多信息。

这样的设备的厂商在 Windows2000 上必须提供下面的设备：

1. 多功能设备的一个定制的 INF 文件。（厂商提供的）

厂商必须提供一个多功能 INF 文件，该文件指定 *mf.sys* 作为多功能总线驱动程序，指定类“多功能”（和在 *devguid.h* 里定义的相关联的 GUID），并提供丢失的配置寄存器地址。在本部分的后面看进一步的信息。

2. 设备的每个功能一个 PnP 功能驱动程序。（厂商提供的）

由于多功能总线驱动程序处理多功能语义，当功能被打包为单个的设备的话，功能驱动程序能够与被使用的驱动程序相同。

3. 设备的每个功能一个 INF 文件。（厂商提供的）

当功能被打包为单个的设备的话，INF 文件与被使用的文件能够是相同的。INF 文件不需要任何特别的多功能语义。

厂商为这样的一个设备提供的定制 INF 必须指定：

- *mf.sys* 作为设备的服务。

  参看 5.4 节获得更多的信息。
- 多功能设备的资源需求。

  在 *DDInstall*.**LogConfigOverride** 节里指定资源需求。
- 每个设备功能的硬件 ID。

  在一个 *DDInstall*.**HW** 节里指定硬件 ID。
- 每个设备功能的资源图，标明了每个子功能对父资源的需求

  在一个 *DDInstall*.**HW** 部分里指定资源图，参看 5.5 节获得有关生成资源图的更多信息。

如果由于重载 configs 存在于 INF 里，INF 必须重述设备指定的所有资源需求。PnP 管理器没有使用设备的任何设备资源需求。

对这样的一个设备，配置选项寄存器能够使用一个 **PcCardConfig** 表项而被编程，相似于编程一个单功能设备。**PcCardConfig** 表项包含应用于整个设备的信息，**PcCardConfig** 表项在《Windows2000 驱动程序开发参考》卷一的第 3 部分的 *INF LogConfig Directive* 文档里。

当为一个多功能设备指定一个 **PcCardConfig** 表项时，*ConfigIndex* 的配置与定义的一个单功能设备的配置相同。单功能 PC Card 的配置寄存器包含在设备的属性里定义的一套资源的一个索引。这个命令也能够用于一定的多功能设备，这些多功能设备使用配置选项寄存器的基于索引的格式。

例 5.1 给出了一个 INF 文件来安装一个多功能设备，该设备使用了 *mf.sys* 作为它的总线驱动程序，并且有一个不完整的配置寄存器。

例 5.1 一个具有不完整的配置寄存器的一个多功能 PcCard 设备的 INF 样本

```
; MFSupra.inf
; 这个文件安装了 Supra Dual 56K 调制解调器
; Copyright 1999 Microsoft Corporation

[version]
Signature     =   "$Windows NT$"
Provider      =   %MSFT%
Class         =   Multifunction          ;系统定义的类
ClassGUID     =   {4d36e971-e325-11ce-bfc1-08002be10318}

[ControlFlags]
ExcludeFromSelect=*SUP2440          ;不要在将要手动安装的设备列表里包含 PnP 设备

[Manufacturer]
%M_Supra% = Supra

[Supra]
%Supra1% = Sup2231GoCard.mf, *SUP2440

[Sup2231GoCard.mf.NT]
Include = mf.inf                    ;说明这个设备需要 mf.sys
Needs = MFINSTALL.mf

[Sup2231GoCard.mf.NT.HW]
AddReg=Sup2231.mf.RegHW

[Sup2231.mf.RegHW]
HKR, Child0000, HardwareID,    , MF\Shotgun_DEV0      ;modem1
HKR, Child0000, ResourceMap, 1, 00, 02
HKR, Child0001, HardwareID,    , MF\Shotgun_DEV1      ;modem2
HKR, Child0001, ResourceMap, 1, 01, 02

[Sup2231GoCard.mf.NT.Services]
Include = mf.inf
Needs = MFINSTALL.mf. Services

[Sup2231GoCard.mf.NT.LogConfigOverride]
LogConfig = Sup223x.mf.Override0,    Sup223x.mf.Override1,    \
            Sup223x.mf.Override2,    Sup223x.mf.Override3

[Sup223x.mf.Override0]
```

```
ConfigPriority = NORMAL
IOConfig        =2F8-2FF                          ; Com2
IOConfig        =20@100-FFFF%FFE0                 ; NIC    I/O
IRQConfig       =3,4,5,7,9,10,11,12,15            ; IRQ
MemConfig       =1000@0-FFFFFFFF%FFFFF000         ; Memory Descriptor
PcCardConfig    =59 (w)                           ; ConfigIndex

[Sup223x.mf.Override1]
ConfigPriority = NORMAL
IOConfig        =3E8-3EF                          ; Com3
IOConfig        =20@100-FFFF%FFE0                 ; NIC    I/O
IRQConfig       =3,4,5,7,9,10,11,12,15            ; IRQ
MemConfig       =1000@0-FFFFFFFF%FFFFF000         ; Memory Descriptor
PcCardConfig    =69 (w)                           ; ConfigIndex

[Sup223x.mf.Override2]
ConfigPriority = NORMAL
IOConfig        =2E8-2EF                          ; Com4
IOConfig        =20@100-FFFF%FFE0                 ; NIC    I/O
IRQConfig       =3,4,5,7,9,10,11,12,15            ; IRQ
MemConfig       =1000@0-FFFFFFFF%FFFFF000         ; Memory Descriptor
PcCardConfig    =79 (w)                           ; ConfigIndex

[Sup223x.mf.Override3]
ConfigPriority = NORMAL
IOConfig        =3F8-3FF                          ; Com1
IOConfig        =20@100-FFFF%FFE0                 ; NIC    I/O
IRQConfig       =3,4,5,7,9,10,11,12,15            ; IRQ
MemConfig       =1000@0-FFFFFFFF%FFFFF000         ; Memory Descriptor
PcCardConfig    =49 (w)                           ; ConfigIndex

[strings]
MSFT = “Microsoft”
M_Supra= “Supra”
Supra1 = “Supra Dual 56K modem”
```

象上面给出的一个 INF 拷贝子功能的 ID 和资源信息到注册表中。当 *mf.sys* 驱动程序枚举设备的子功能数目时，它从注册表里提取信息。

5.2 支持多功能 PCI 设备

如果一个多功能 PCI 设备完全遵照 PCI 多功能标准，PCI 总线驱动程序枚举单个的功能。PCI 总线驱动程序管理这个事实，即有不止一个功能驻留在单个的设备位置里。对系统的其余部分，单个的功能象独立的设备一样操作。

一个多功能 PCI 设备的厂商在 Windows2000 上必须做下面的事情：

1. 保证设备遵照多功能规范。
2. 提供设备的每个功能一个 PnP 功能驱动程序。

由于多功能总线驱动程序处理多功能语义，当功能被打包为单个的设备的话，功能驱动程序能够与被使用的驱动程序相同。

3. 提供设备每个功能的一个 INF 文件。

当功能被打包为单个的设备的话，INF 文件能够与被使用的文件是相同的。INF 文件不需要任何特别的多功能语义。

例如，图 5.1 给出了可能为具有 ISDN 和调制解调器功能的多功能 PCI 设备而建立的设备堆栈的一个例子，如图 5.1 所示的，不是枚举了一个多功能设备，PCI 驱动程序枚举了两个子设备。PnP 管理器象对待一个典型的设备一样对待每个子设备，定位 INF 文件，装载适当的驱动程序，调用它们的 AddDevice 例程等等，直到一个设备堆栈为每个设备建立为止。PCI 驱动程序为子设备仲裁资源，并管理设备的任何其他的多功能方面。

图 5.1　一个其父设备枚举每个功能的多功能设备的设备堆栈实例

多功能卡的厂商为 ISDN 和调制解调器设备提供功能驱动程序和 INF，就如同它们是分离的设备。

图例集中于功能驱动程序和总线驱动程序，说明每个功能和它们相关联的 FDO 和 PDO，为简单起见，忽略任何过滤器驱动程序（和过滤程序 DO）。

5.3 支持其他总线上的多功能设备

对一个 PnP ISA、USB 或 1394 总线上的一个多功能设备，如果设备遵照总线标准的话，父总线驱动程序枚举单个的功能。

对这样的一个设备，父总线驱动程序管理这个事实，即有不止一个设备驻留在单个的总线里的。对剩下系统，单个的功能象独立的设备一样操作。

这样类型的多功能设备厂商必须做下面的事情：

1. 保证设备遵照总线的规范，这里，设备将留驻于这个总线上。
2. 提供设备的每个功能的一个 PnP 功能驱动程序。

由于多功能总线驱动程序处理多功能语义，当功能被打包为单个的设备的话，功能驱动程序能够与被使用的驱动程序相同。

3. 提供设备的每个功能的一个 INF 文件。

当功能被打包为单个的设备的话，INF 文件能够与被使用的文件是相同的。INF 文件不需要任何特别的多功能语义。

5.4 使用系统提供的 mf.sys

如果一个设备的基础总线支持一个多功能总线标准，如 PC Card，这样的设备的厂商在 Windows2000 上能够使用系统提供的多功能总线驱动程序（*mf.sys*）来支持设备。

mf.sys 总线驱动程序处理设备功能的 PnP 枚举，并在两个功能之间仲裁资源，如 I/O 端口和 IRQs。*mf.sys* 驱动程序通过电源管理父多功能设备来为子功能处理电源管理。

为使用 *mf.sys*，一个多功能设备必须满足下面的需求：

- 设备的基础总线必须有一个多功能总线标准。
- 子功能的 DEVICE_CAPABILITIES 必须是相同的且必须匹配这些父设备。当查询一个子设备功能（IRP_MN_QUERY_CAPABILITIES）的设备能力时，*mf.sys* 驱动程序报告父设备的设备能力。
- 多功能设备驻留的总线的驱动程序，如 *pcmcia.sys*，必须处理任何 IRP_MN_READ_CONFIG 和 IRP_MN_WRITE_CONFIG 请求。*mf.sys* 驱动程序只是传递这些 IRP 给父总线驱动程序。
- 功能必须是独立的：它们不能够有启动顺序依赖；一个功能的资源需求不能够按照另外一个功能资源被表达（例如，功能 1 使用 I/O 端口 X 和功能 2 使用端口 X+200）；并且每个功能必须能够作为一个独立的设备操作，即使它们作为另一个功能而被相同的驱动程序所服务。

为使用 *mf.sys*，一个厂商为多功能设备提供一个 INF，这个多功能设备指定 *mf.sys* 作为设备的驱动程序的。如果一个设备完全地和正确地遵照它的基础总线的多功能标准，这样的设备的厂商能够使用系统提供的 *mf.sys*。如果一个设备没有完全地遵照这个标准，厂商必须提供一个定制 INF。

在任何一种情况下，厂商也为设备上的单个功能提供驱动程序和 INF 文件。

下面的一个定制多功能 INF 骨架说明了必要的语义，来指定 *mf.sys* 作为一个多功能设备的驱动程序：

```
[Version]
Signature = “$Windows    NT$”
;…
Class = Multifunction          ;系统定义的 MF 设备类
ClassGUID = {4d36e971-e325-11ce-bfc1-08002be10318}     ;MF 的 GUID
;…
;…
[ControlFlags]
ExcludeFromSelect = *          ;不要让设备列表里的 PnP 设备被手动安装

[Manufacturer]
;…
;…
[ModelSection]                 ;models    section
;…
;…
[DDInstall]          ;install    section
```

```
Include = mf.inf                ;说明这个设备需要 mf.sys
Needs = MFINSTALL.mf
;…
[DDInstall.HW]
AddReg= DDInstall.RegHW

[DDInstall.RegHW]
;在这里放置具有子功能硬件 ID 的表项
;…

;在这里放置重载节
;…

[Strings]
;…
```

考虑一个综合 LAN/调制解调器 PC Card 设备，没有任何特定的多功能支持，这样一个设备可能被 PCMCIA 总线驱动程序作为一个单个的调制解调器设备来报告。有另外的一个多功能 INF 和 *mf.sys* 总线驱动程序的支持，设备的两种功能被枚举。图 5.2 给出了设备堆栈的例子，该堆栈可能为这样一个具有必要的多功能支持的 combo PC Card 而建立。

如图 5.2 所示的，驱动程序为多功能设备驻留的总线枚举一个设备。在多功能 INF 文件里的硬件 ID 引起 PnP 管理器装载 *mf.sys* 总线驱动程序作为设备的功能驱动程序。*mf.sys* 总线驱动程序枚举两个子设备，一个 LAN 设备和一个调制解调器。

图 5.2 被 *mf.sys* 枚举的一个多功能设备的设备堆栈实例

PnP 管理器象对待一个典型的设备一样对待每个子设备，定位 INF 文件，装载适当的驱动程序，调用它们的 AddDevice 例程等等，直到一个设备堆栈为每个设备而建立为止。*mf.sys* 总线驱动程序为子设备仲裁资源，并管理设备的任何其他的多功能方面。多功能卡的设备的厂商为多个功能（LAN 和调制解调器）提供功能驱动程序和 INFs，就如同它们是分开的设备一样。

图例集中于功能驱动程序和父总线驱动程序和它们相关联的 FDO 和 PDO，为简单起见，忽略任何过滤器驱动程序（和过滤程序 DO）。

5.5 为一个多功能设备生成资源图

一个资源图在一个 INF *add-registry-section* 里标识了被一个子设备功能所使用的一个多功能资源设备。如果一个多功能设备需要资源图，设备的 INF 典型地包括一个每个设备功能的资源图。

本部分描述了怎样构建资源图。参看前节来决定哪一个多功能设备需要资源图。

为了在一个子功能资源图里引用一个父资源，当父资源出现在 *log-config-section(s)*里时，它被隐含地标数为 00-*N*。

例如，用下面的 *DDInstall*.**LogConfigOverride** 节考虑一个多功能 PC Card：

```
[DDInstall.Override0]                          ;com2
IOConfig=2f8-2ff                               ;资源 00
IOConfig=20@100-FFFF%FFE0                      ;资源 01
IRQConfig=3,4,5,7,9,10,11                      ;资源 02
MemConfig=4000@0-FFFFFFFF%FFFFC000             ;资源 03
PcCardConfig=41:100000 (w)                     ;资源 04
```

在该例中的设备有五个资源，当构建一个资源图时，它们被标数为 00-04。如果有多于一个 *DDInstall*.**LogConfigOverride** 节，资源在每个部分里必须以相同的顺序列表。

如果一个子功能（Child0000）需要上面的第一个和第三个资源列表，这个子功能的资源图将是：00,02。如果另一个子功能（Child0001）需要所有的五个资源，那么它的资源图将是：00,01,02,03,04。例子中，资源 00（IoConfig=2f8-2ff）和 02（IRQConfig=3,4,5, 7, 9,10,11）被共享。这些资源图将在 INF 里按如下指定：

```
[DDInstall.RegHW]
                    ;每个“子”功能列表硬件和资源图
HKR,Child0000,HardwareID,,child0000-hardware-ID
HKR,Child0000,ResourceMap,1,00,02              ;子 0000 图
HKR,Child0001,HardwareID,,child0001-hardware-ID
HKR,Child0001,ResourceMap,1,00,01,02,03,04     ;子 0001 图
```

紧随 ResourceMap 参数的“1”指定注册表表项是一个 REG_BINARY，紧随“1”的数字是资源图值。

如果在 INF 里没有 *DDInstall*.**LogConfigOverride** 节，父资源以资源需求被基础总线驱动程序构建的顺序来标数。对 PC Card 来说，总线驱动程序以这样的顺序报告资源：IRQ，I/O 端口，内存地址。对多个 I/O 和内存需求来说，它们以与卡上的元组同样的顺序被标数。

第三部分　　电源管理

第1章　在驱动程序内支持电源管理

本书的第1部分介绍了即插即用(PnP)驱动程序的一般要求。这一章介绍为支持电源管理对驱动程序的要求。它包括下列各节：

1.1　内核模式下的电源管理组件

下面的内容在电源管理中对三种内核模式组件的作用提供了简要的介绍。

1.1.1　ACPI BIOS

以Microsoft® Windows® 2000和Windows 98支持的集成电源管理特性仅在拥有高级配置和电源接口(ACPI)BIOS的系统上可用。 ACPI BIOS在硬件层上提供了一些特性使软件电源管理成为

可能。 在启动时，操作系统装载器检查这种BIOS是否被装载。 如果这种BIOS已经装载，ACPI就能被激活于操作系统中。如果没有装载，ACPI就被禁止，同时由不健壮的高级电源管理(APM)模型代替使用。

在缺省情况下，Windows 2000假定任何ACPI BIOS的给定日期为1999年1月1日，或者稍晚,即Windows 2000准备完毕时。 但是，如果已告知这种BIOS有ACPI问题，装载器禁止ACPI并由APM代替使用。

设备管理器显示一个计算机是否支持ACPI。

1.1.2 ACPI 驱动程序

微软提供ACPI驱动程序作为操作系统的一部分。 在拥有ACPI BIOS的系统上，HAL使ACPI驱动程序以设备树为基础在系统启动时装载，其中HAL作为操作系统和BIOS之间的接口。 ACPI 驱动程序对其他驱动程序是透明的。

ACPI 驱动程序的职责包括支持PnP和电源管理。例如，一个系统上的ACPI任务可以包括为COM端口再编写资源或者为系统唤醒重新激活USB控制器。

对于每一个在ACPI名称空间(硬件供应商建立的ACPI BIOS的一部分)中所描述的设备，ACPI驱动程序建立一种过滤设备对象(过滤DO)或者一种物理设备对象(PDO)。 如果设备集成在系统板卡上，驱动程序建立过滤设备对象，代表ACPI总线过滤，并且立即将它附加到总线驱动程序(PDO)之上的设备栈中。 对于其他在ACPI中所描述的名称空间中而不在系统板上的设备，ACPI建立PDO。ACPI通过这些设备对象向设备栈提供电源管理和PnP特性。

ACPI建立了设备对象的设备，依赖于机器的BIOS和主插件板（母板）的配置，在不同系统中有所不同。注意，一个ACPI总线过滤程序只为在ACPI名称空间中描述并集成在系统板中的设备装载。并非所有的母板设备都有ACPI总线过滤程序。

所有ACPI的功能对高层驱动程序都是透明的。在任何给定的设备栈中，驱动程序都不能假定存在（或不存在）ACPI过滤器。

1.1.3 电源管理器

电源管理器负责为系统管理电源的使用。在第一部分中已提到，它管理系统范围的电源策略，并且通过系统跟踪电源IRP的路径。

电源管理器通过把IRP_MJ_POWER请求发送到驱动程序中来请求电源操作。一个请求可以指定一种新的电源状态或者能查询在电源状态方面的变化是否是可行的。

当要求休眠，冬眠，或者关机时，电源管理器把IRP_MJ_POWER请求发送到每一个要求适当电源行动的设备树中的叶节点。 电源管理器在确定该系统是否应该休眠，冬眠，或者关机时要考虑下列几点：

。系统活动电平
。系统电池电平
。应用程序请求关机，冬眠，还是休眠
。用户行动，诸如按下电源按钮
。控制面板设置

电源管理器也为驱动程序提供了接口，这主要包括PoXxx例程，电源管理IRP，和所需的驱动程序入口点。 驱动程序使用这个接口管理这些设备的电源，并且为系统电源管理去响应电源管理器的请求。

1.1.4 电源管理中驱动程序的作用

驱动程序在两个方面支持电源管理：

1.驱动程序响应由电源管理器所发出的系统范围的电源请求。

2.驱动程序为其独立设备管理电源。

所有驱动程序都必须有DispatchPower例程来处理IRP_MJ_POWER请求，就象在第1部分中所描述的。在其DispatchPower例程中，驱动程序检查每一电源IRP，或者是处理它，或者把它传递到相邻较低层驱动程序中。

对于参与电源管理的设备，此设备在设备栈中的所有驱动程序必须作出响应或者适当地传递电源IRP。单个驱动程序的失败会导致电源管理在整个系统中被禁止。

每一设备有一个驱动程序为其设备管理电源策略。这个驱动程序能把电源IRP发送到它自己的设备栈中，以在其设备上执行电源操作。电源策略管理器负责发出符合系统电源IRP的设备电源IRP。此外，驱动程序还可以执行一定的电源任务，诸如，在启动或关闭时管理设备的电源，而不必接收电源IRP，要考虑这些隐含的电源请求。

1.2 驱动程序的电源管理职能

支持电源管理的驱动程序，其职能如下：

。保证电源管理性能能在PnP列举过程中有详细的报告

。为电源管理在设备对象中设置标记

。由电源管理器或者驱动程序处理电源IRP的发送

。当系统启动或者停止空闲状态时立即给设备通电

。当系统关闭或空闲时，断电或让设备休眠

。如果设备支持唤醒能力，激活设备的唤醒能力

并非设备栈中的每一个驱动程序都执行这些任务。一般来说，总线驱动程序报告能力，设置标记，操纵物理设备，设备电源策略管理器(通常是功能驱动程序)发出请求使设备处于休眠状况并且激活唤醒。

只有极少例外情况，驱动程序响应电源IRP对设备进行加电，断电以及激活唤醒能力，即主代码为IRP_MJ_POWER的IRP。电源管理器可以发送电源IRP，在某些情况下驱动程序也可以发送电源IRP。

1.2.1 报告设备的电源性能

在列举过程中，驱动程序报告设备的信息以响应PnP IRP_MN_QUERY_CAPABILITIES请求。 与

这些信息一起，驱动程序在DEVICE_CAPABILITIES 结构中，报告设备的电源管理能力。总线驱动程序一般都要填写这个结构。

高层驱动程序应该为查询能力IRP设置IoCompletion 例程，这样这些驱动程序就能做一个这种结构的本地拷贝，并且保证它包含适当的值。一般来说，高层驱动程序不能改变这些值。 但是，如果必须改变的话，驱动程序能更进一步地限制设备的能力，但不能增加能力。换句话说，驱动程序能限制规则而不能放松规则。

在IRP完成且所有驱动程序的完成例程都运行之后，此结构被放入高速缓存，而且驱动程序不能改变其内容。

DEVICE_CAPABILITIES 的以下字段属于电源管理：

。DeviceD1和DeviceD2

。WakeFromD0，WakeFromD1，WakeFromD2，和WakeFromD3

。DeviceState

。SystemWake

。DeviceWake

。D1Latency，D2Latency，和D3Latency

1.2.1.1 DeviceD1 和 DeviceD2

DeviceD1和DeviceD2字段表明设备硬件是否支持这些设备状态。 它们都只占用一位，这些位置位或清零表示设备支持这种状态还是不支持这种状态。 操作系统假定所有设备支持D0和D3状态。

1.2.1.2 WakeFromD0 ， WakeFromD1 ， WakeFromD2 ， 和 WakeFromD3

每一个这样的字段表明当设备在指定状态下时，能否被外部信号唤醒。

对于支持所有四种设备状态(D0，D1，D2，D3)但是只能从状态D0和D1中唤醒的设备，设置WakeFromD0和WakeFromD1位，同时对WakeFromD2和WakeFromD3位清零。

1.2.1.3 DeviceState

DeviceState字段是一个由DEVICE_POWER_STATE值构成的数组,按SYSTEM_POWER_STATE值进行索引,其范围为从PowerSystemWorking到PowerSystemShutdown。此数组中的每一个元素包含为索引中的系统状态设备支持的最大(最大功率)的设备状态，如果不支持系统状态，则置为PowerDeviceUnspecified。

例如，在仅仅支持S0，S4，和S5的系统上，支持D0和D3状态的设备的DeviceState数组包含下列情况：

DeviceState 元素	值
DeviceState[PowerSystemWorking]	PowerDeviceD0
DeviceState[PowerSystemSleeping1]	PowerDeviceUnspecified

DeviceState[PowerSystemSleeping2]	PowerDeviceUnspecified
DeviceState[PowerSystemSleeping3]	PowerDeviceUnspecified
DeviceState[PowerSystemHibernate]	PowerDeviceD3
DeviceState[PowerSystemShutdown]	PowerDeviceD3

在支持所有系统电源状态的系统上，对于当系统变为休眠状态时必须处于D2状态，当系统在冬眠状态时处于D3状态的设备DeviceState数组的设置如下：

DeviceState 元素	值
DeviceState[PowerSystemWorking]	PowerDeviceD0
DeviceState[PowerSystemSleeping1]	PowerDeviceD2
DeviceState[PowerSystemSleeping2]	PowerDeviceD2
DeviceState[PowerSystemSleeping3]	PowerDeviceD2
DeviceState[PowerSystemHibernate]	PowerDeviceD3
DeviceState[PowerSystemShutdown]	PowerDeviceD3

注意，DeviceState数组的项目表示为相应系统状态设备支持的最大功率状态。 在前面的例子中，对任何系统电源状态，设备都可以处于D3状态，对系统状态PowerSystemWorking 到PowerSystemSleeping3，设备可处于D2状态，对系统状态PowerSystemWorking，设备可处于D1状态。

总线驱动程序或者 ACPI过滤程序基于双亲设备节点的能力，设置这些值。

一般来说，高层驱动程序不应该改变这些值。 然而，在一些很少出现的情况下，这种变化也是可能的，设备能指定一个比总线驱动程序或者比ACPI过滤程序的最初返回值低（较小功率）的状态。例如，在上面的表中，假定DeviceState[PowerSystemSleeping1]为 PowerDeviceD2。驱动程序能把此值改变为PowerDeviceD3，但不能改变为PowerDeviceD1或者PowerDeviceD0。

1.2.1.4 SystemWake

SystemWake字段包含设备能唤醒系统时的最低(最小功率)系统电源状态，或者当设备不能唤醒系统时此字段值为PowerSystemUndefined。

在列举中总线驱动程序设置了此值。 一个高层驱动程序能把此值改变为较高功率状态，但是不能把它改变为较低功率状态。 例如，如果总线驱动程序把SystemWake设置为S3，而设备栈上层的驱动程序仅仅能支持从S2唤醒，则高层驱动程序能改变此值为S2。 如果驱动程序改变了SystemWake，那么它也不得不改变DeviceWake，就像下面所解释的。

驱动程序很少需要将值的改变通知设备栈。 因为这些改变的值更加限制了设备能力，较低层的驱动程序不会收到他们不能处理的请求。 在前面的例子中，高层驱动程序从低功率状态（比S2低）唤醒系统的请求失败，因此较低层驱动程序不会收到这样的请求。但是一个驱动程序必须能意识到任何改变，这样它就能在IRP_MN_START_DEVICE处理中发送PnP IRP_MN_QUERY_CAPABILITIES到它自己的设备栈中。

如果SystemWake和DeviceWake两个字段都非零(即不是PowerSystemUnspecified)，则设备和其驱动程序支持此系统上的唤醒。

在非ACPI硬件上，这个字段总包含零(PowerSystemUnspecified)。

1.2.1.5 DeviceWake

DeviceWake字段包含设备能响应唤醒事件的最低(最小功率)设备电源状态，或者如果设备不能被外部信号唤醒，则置为PowerDeviceUnspecified。

总线驱动程序设置此值。 高层驱动程序能改变此值为一个较大功率状态。例如，如果总线驱动程序把DeviceWake设置为D3，而设备栈上层的驱动程序仅仅能支持从D2唤醒，则高层驱动程序能把此值改变为D2。

注意,如果一个驱动程序改变DeviceWake,则它也不得不改变SystemWake,以避免DeviceState数组中的系统到设备映像发生冲突。 例如，假定总线驱动程序设置为下列情况：

。DeviceState[PowerSystemSleeping1]=PowerDeviceD1

。DeviceState[PowerSystemSleeping2]=PowerDeviceD3

。DeviceWake=PowerDeviceD3

。SystemWake=PowerSystemSleeping2

如果一个高层驱动程序确定其设备不能从D3状态唤醒系统，而只能从D2或者更高状态，那么它能改变DeviceWake为D2。 然而，这种变化引起从S2到D3的映像成为不可能的。DeviceState数组为给定系统状态列出了设备支持的最高功率状态。如果这个例子中的系统状态是PowerSystemSleeping2，则设备状态就不能是PowerDeviceD2。为了解决这个问题，驱动程序也必须把SystemWake改变为PowerSystemSleeping1。这对WakeFromDx和DeviceDx的设置也是成立的。驱动程序必须保证它对SystemWake或者DeviceWake的所有改变都与WakeFromDx和DeviceDx值不冲突。WakeFromDx和DeviceDx的值反映驱动程序不能变化的硬件特性。

如果SystemWake和DeviceWake两个字段都是非零(即不是 PowerSystemUnspecified),那么设备和其驱动程序都支持在这个系统上唤醒。

在非 ACPI硬件上，DeviceWake字段包含零(PowerSystemUnspecified)。

1.2.1.6 D1Latency，D2Latency，和 D3Latency

D1Latency，D2Latency，和D3Latency字段包含从D0状态到每个休眠状态要求的近似时间(以100微秒为单位)。为不支持的任何设备状态驱动程序将等待时间置为零。

1.2.2 对电源管理设置设备对象标记

在AddDevice例程中，每一驱动程序建立一个设备对象(过滤DO，FDO或者PDO)，并且根据需要设置DO_Xxxx标记。 下列的设备对象标记属于电源管理：

。DO_POWER_INRUSH

。DO_POWER_PAGABLE

设备对象标记一般是在总线驱动程序为设备创建PDO时由总线驱动程序设置的。 然而，一些功能驱动程序可能需要改变这些标记的值,把它们作为这些驱动程序AddDevice例程的一部分。总之，标记的值应该在整个设备栈的所有设备对象中是一致的。

在Windows 2000中，分页路径上的设备驱动程序不能设置DO_POWER_PAGABLE 标记。不设置标记的驱动程序必须在IRQL DISPATCH_LEVEL 下被调用。

总之，驱动程序不应改变总线驱动程序的标记值，如果较低层驱动程序清除了这个标记，驱动程序就决不能设置此标记。当处理包括PnP分页请求的转变时(一般来说，响应带有IRP_MN_DEVICE_USAGE_NOTIFlCATION 请求的IRP_MJ_PNP)，存储驱动程序必须认真安排其标记的设置和清除的顺序。

可分页的Windos 2000驱动程序必须设置DO_POWER_PAGABLE标记。 电源管理器在IRQL PASSIVE_LEVEL 下调用这些驱动程序。 所有WDM和Windows 98驱动程序必须设置DO_POWER_PAGABLE。此外，Windows 98驱动程序不能在IRQL DISPATCH_LEVEL 下完成电源IRP。

启动时要求inrash启动功率的设备驱动程序必须在设备对象中设置DO_POWER_INRUSH 标记，而且要在此之后清除DO_DEVICE_INITIALIZING 标记。 在设备栈中只有一个设备（一般是总线驱动程序(PDO)）需要设置DO_POWER_INRUSH。此标记通知电源管理器必须同时给这些设备上电，

的设备按顺序依次上电。 电源管理器保证在任何给定时间内系统中只有一个IRP是活动的。

电源管理器能在IRQL DISPATCH_LEVEL 下发送启动功率IRP。 当处理一个IRP时，驱动程序不能访问任何可分页代码或者数据。

驱动程序不能同时设置 DO_POWER_PAGABLE 和 DO_POWER_INRUSH。

1.2.3 处理电源 IRP

驱动程序在DispatchPower 例程中处理电源IRP。 所有电源管理请求都有主IRP代码IRP_MJ_POWER 和下列之一的次代码：

。IRP_MN_QUERY_POWER----通过查询确定电源状态的改变是否可行。

。IRP_MN_SET_POWER----请求从一种电源状态变化到另一种电源状态。

。IRP_MN_WAIT_WAKE----请求激活设备对自己或者对系统的唤醒。

。IRP_MN_POWER_SEQUENCE----请求特定设备的优化电源恢复信息。

支持IRP_MN_SET_POWER 和IRP_MN_QUERY_POWER 是需要的。所有驱动程序都必须准备处理这些IRP。

设备栈中所有能被外部信号唤醒的设备都支持IRP_MN_WAIT_WAKE。驱动程序发送此IRP以唤醒设备。

支持IRP_MN_POWER_SEQUENCE是可选的。 这个IRP为需要长时间恢复电源的设备提供优化。

电源IRP能指定系统电源操作或者设备电源操作。系统电源IRP和设备电源IRP采取略有不同的路径通过设备栈，如下列各段所述。

1.2.3.1 系统电源 IRP

系统电源IRP指定主IRP代码IRP_MJ_POWER，指定下面列出的次电源IRP代码之一，以及IRP栈的Power.Type成员值SystemPowerState。只有电源管理器能发送这样一个IRP；驱动程序不能发送系统电源IRP。

由于下列原因之一，电源管理器发送系统电源IRP：

。改变系统电源状态以响应一个空闲超时、一个系统活动变化、一个用户请求、或者电池电

压过低(IRP_MN_SET_POWER)

。查询设备以确定系统是否能进入休眠(IRP_MN_QUERY_POWER)

。在查询之后重新确定当前系统电源状态(IRP_MN_SET_POWER)

电源管理器代表系统发送IRP_MN_QUERY_POWER 和 IRP_MN_SET_POWER请求。驱动程序可以使IRP_MN_QUERY_POWER请求失败，但是不能使IRP_MN_SET_POWER请求失败。

例如，为了改变系统电源状态，在设备树的每一设备节点中，电源管理器把系统电源IRP发送到栈中的顶端驱动程序。 图1.1显示在一个设备栈内的驱动程序如何处理系统电源IRP。

图1.1表明：

1.电源管理器调用I/O管理器把系统电源IRP发送到设备树中的每一叶节点。

2. 如果可能，驱动程序处理IRP，必要时设置IoCompletion例程，并向栈底转发IRP (使用PoCallDriver)。如果驱动程序必须使 IRP失败，它会立即这样做并且完成IRP。 驱动程序能使IRP_MN_QUERY_POWER IRP失败，但是不能使IRP_MN_SET_POWER IRP失败。

3.当拥有设备电源策略的驱动程序接收到IRP时，驱动程序必须在发送IRP之前为系统设置IoCompletion例程。

4.最后，总线驱动程序完成系统IRP。I/O管理器调用IoCompletion例程。

5.在IoCompletion例程中，设备电源策略所有者调用PoRequest PowerIrp发送设备电源IRP，指定驱动程序设置一个在设备电源IRP完成后调用的回调例程。

如果有必要的话，驱动程序在 DEVICE_CAPABILITIES结构的可缓冲拷贝中参考DeviceState字段(参见“报告设备的电源性能”)，确定哪个设备电源状态符合在IRP中的系统电源状态。

6.在设备IRP完成且任何设备IRP完成例程都已运行时，调用电源策略所有者的回调例程。 在回调例程中，驱动程序将返回状态复制到系统IRP中。它先调用PoStartNextPowerIrp，然后调用IoCompleteRequest，完成系统IRP。

详见第3章。

图1.1 系统电源IRP的路径

由于一些设备在加电时需要有启动功率，系统启动功率IRP在整个系统中同步且串行地被处理。在同一时间内仅有一个这样的IRP是活跃的。详见1.3.1节。

1.2.3.2 独立设备的电源 IRP

设备电源IRP指定主IRP代码IRP_MJ_POWER、下面列出的次电源IRP代码之一、以及Power.Type成员的值DevicePowerState。

。IRP_MN_QUERY_POWER

。IRP_MN_SET_POWER

。IRP_MN_WAIT_WAKE

。IRP_MN_POWER_SEQUENCE

所有设备栈中的驱动程序都接收这样的IRP；一般来说,只有设备电源策略管理器能发送这些IRP。 然而，当代表设备执行空闲检测时，电源管理器也能发送设备电源IRP，就像在“使用电源管理器程序进行空闲检测”中所解释的一样。

驱动程序发送设备电源IRP有以下原因：

。为了查询或者改变设备电源状态以响应系统电源IRP

。为了设备进入休眠状态中以节能

。为了将已进入休眠状态的设备返回到工作状态
。为了响应外部信号以唤醒设备
。为了在给设备上电时得到的电源顺序值

正如图1.2所示，设备电源IRP按以下步骤发送、转发和完成：

1.设备电源策略所有者调用PoRequestPowerIrp分配设备电源IRP,在参数中指定IRP完成时要调用的回调例程。

2.电源管理器分配设备电源IRP，并且为目标PDO把它发送到设备栈的顶端驱动程序中。

3.驱动程序执行任何需要的动作，如果需要,设置IoCompletion例程，调用PoStartNextPowerIrp，通知系统已准备好处理另一个电源IRP，并且调用PoCallDriver把IRP传递到相邻较低层驱动程序中。每一个栈中的驱动程序都这样做直到IRP到达总线驱动程序中。如果驱动程序必须使IRP失败，那么它将立即这样做并调用IoCompleteRequest。

4. 保持设备PDO的总线驱动程序，执行请求的动作，然后调用IoCompleteRequest完成IRP。

5.I/O管理器调用IoCompletion例程,此例程是驱动程序在IRP向栈底传送时设置的。在所有IoCompletion例程被调用后,调用回调例程。

关于设备电源IRP更多的信息，参见第3章和第4章。关于上电顺序IRP的细节情况，参见卷1的“Windows2000 驱动程序开发指南”中的IRP_MN_POWER_SEQUENCE。

图1.2 设备电源IRP的路径

1.2.4 设备的上电

当总线驱动程序为其子设备之一处理PnP IRP_MN_START_DEVICE请求时，它应该给设备加电，并调用PoSetPowerState以向电源管理器报告设备状态。设备加电是设备启动时的必须部分。设备电源策略所有者不为PowerDeviceD0发送IRP_MN_SET_POWER请求,因此驱动程序不必期待在启动时收到这些IRP。

当为了节能将设备关闭时，它的驱动程序应在I/O请求到达时给设备上电。在这种情况下，设备电源策略所有者必须发送IRP_MN_SET_POWER把设备返回到工作的状态。当IRP完成之后，驱动程序停止I/O排队，开始处理队列中的请求。

1.2.5 设备的掉电

除非设备能够被唤醒，否则当系统关闭时，其驱动程序会关闭其电源。设备必须是掉电删除或者意外删除。

当设备被删除时，PnP管理器发送IRP_MN_REMOVE_DEVICE请求到设备栈。在响应IRP时，设备驱动程序应该保证设备掉电。设备掉电是删除处理中必须的部分；设备电源策略所有者不要求为PowerDeviceD3发送IRP_MN_SET_POWER。

当驱动程序处理IRP_MN_REMOVE_DEVICE请求时，他们等候未决I/O的完成，执行任何必要的删除处理，调用PoSetPowerState通知电源管理器设备在状态D3中，并且删除它们为这种设备建立的

设备对象。一般来说，总线驱动程序关闭设备的电源。

如果设备从Windows 2000系统中被意外删除,PnP管理模块发送IRP_MN_SURPRISE_REMOVAL请求。作为对这个IRP的响应，驱动程序应该关闭设备的电源，并且执行附加的删除处理，就象在“处理IRP_MN_SURPRISE_REMOVAL请求”中所描述的。

系统关闭时，电源管理器为系统电源状态(S4或者S5)发送IRP_MN_SET_POWER。 当设备电源策略所有者收到这个IRP时，它应该为PowerDeviceD3发送IRP_MN_SET_POWER，这样低层驱动程序就能完成它们的工作并关闭设备的电源。

驱动程序能选择性地执行对其设备的空闲检测，或者请求电源管理器执行空闲检测，以便在不使用时能关闭设备的电源。详见第2章中的“检测空闲的设备”。

1.2.6 激活设备唤醒能力

如果设备支持唤醒，其电源策略所有者必须能够激活或禁止设备的唤醒能力。驱动程序通过发送IRP_MJ_POWER及次功能代码IRP_MN_WAIT_WAKE激活唤醒能力，并且通过取消以前发送的IRP_MN_WAIT_WAKE禁止唤醒能力。设备在某一时间内只能有一个IRP_MN_WAIT_WAKE是未决的。

为了确定其设备是否支持唤醒、从哪个设备状态可接收唤醒信号、在哪个系统状态可唤醒系统,驱动程序参考DEVICE_CAPABILITIES结构中的SystemWake，DeviceWake，和WakeFromDx字段。

详见第4章，关于在驱动程序中激活、禁止和响应唤醒信号。

1.3 处理电源 IRP 的规则

支持电源管理的驱动程序必须在下列情况中符合一定的规则：

。使用PoCallDriver，而不是IoCallDriver，传递电源IRP

。把电源IRP传递到相邻较低层驱动程序中

。当设备正在休眠时，排队I/O请求

。处理没有支持或者不可识别的电源IRP

下面各节讲述驱动程序应该如何执行这些任务。

1.3.1 使用 PoCallDriver

驱动程序必须使用PoCallDriver把电源IRP传递到相邻较低层驱动程序上，而不使用IoCallDrivcr。PoRequestPowerIrp和PoStartNextPowerIrp与PoCallDriver保证电源管理器在整个系统中适当地同步电源IRP。

系统限制活动的电源IRP的数量如下：

。在任何给定的时间内,对每个物理设备对象都不能有多于一个以上的系统电源IRP(IRP_MN_SET_POWER，IRP_MN_QUERY_POWER)活动。

。在任何给定的时间内,对每个物理设备对象都不能有多于一个以上的设备设置电源IRP(IRP_MN_SET_POWER)活动。

。在任何给定的时间内,系统中都不能有一个以上的请求启动功率电源IRP活动。

为了保证两种需启动功率的设备不试图同时上电，电源管理器在整个系统中追踪活动的需启动功率电源IRP,并且允许一次只有一个是活动的。当活动的启动功率IRP已经完成时，其他的启动功率IRP才能开始。

由于这些对启动功率IRP的限制,一个设备电源IRP可能会有其他设备的启动功率IRP完成时被阻塞。驱动程序编写者在调试时应该及时了解到这种行为。

1.3.2 传递电源 IRP

必须把电源IRP通过设备栈的所有路径向下传递到PDO，以保证电源转换是被“干净”地管理。当IRP在栈中向下传递时，驱动程序处理减少设备能量的IRP。当IRP从栈中向上返回时，驱动程序在IoCompletion例程中处理给设备增加能量的IRP。

图1.3显示向下传递电源IRP的有关步骤。

图1.3 向下传递一个电源IRP

由图中所示，驱动程序必须做到下列各内容：

1.如果驱动程序没有设置IoCompletion例程，则调用PoStartNextPowerIrp。

2.为相邻较低层驱动程序设置IRP栈位置。没有设置IoCompletion例程的驱动程序应该使用IoSkipCurrentIrpStackLocation。

设置IoCompletion例程的驱动程序应该使用IoCopyCurrentIrpStackLocationToNext。拷贝当前栈的位置，以保证当IoCompletion例程运行时，IRP栈指针设置到正确的位置。

3.必要时，调用IoSetCompletionRoutine来设置IoCompletion例程。在IoCompletion例程中，驱动程序能调用PoStartNextPowerIrp以表明它已准备好处理下一个电源IRP。

4.调用PoCallDriver ，把IRP传递到栈中的相邻较低层驱动程序中。

驱动程序必须使用PoCallDriver而非IoCallDriver(就其他IRP而论),以保证系统适当地同步电源IRP。详见1.3.1节。

当前IRP栈位置指向当前驱动程序时，驱动程序必须调用PoStartNextPowerIrp。 这个例程能从IoCompletion例程或从DispatchPower例程中调用。 然而，PoStartNextPowerIrp必须在IoCompleteRequest，IoSkipCurrentIrpStackLocation和PoCallDriver之前调用。若按其他顺序调用，程序可能会造成系统死锁。

即使某一驱动程序使IRP失败，这个驱动程序也必须调用PoStartNextPowerIrp通知电源管理器，它已准备好处理另一个电源IRP。

记住,IoCompletion例程能在IRQL DISPATCH_LEVEL中调用。因此，在带有IRP的低层驱动程序完成之后，如果在IRQL PASSIVE_LEVEL下驱动程序请求附加处理，驱动程序的完成例程应该对工作项目排队，然后返回STATUS_MORE_PROCESSING_REQUIRED。线程必须完成此IRP。

处理电源IRP时不能阻塞

驱动程序在处理电源IRP时决不能造成长时间的延迟。

当向下传递电源IRP时，驱动程序应该在调用PoStartNextPowerIrp和PoCallDriver之后，尽可能快地从DispatchPower例程中返回。驱动程序在返回之前不能等待一个内核事件或用其他原因延迟。如果驱动程序不能在一个短的时间内处理电源IRP，它将返回STATUS_PENDING并且对所有的

IRP进行排队，直到电源IRP完成(注意,这种行为不同于PnP IRP和DispatchPnP例程，这两个例程允许阻塞)。

如果驱动程序必须等待设备栈下层另一个驱动程序的电源行动，则此驱动程序应该从其DispatchPower例程中返回STATUS_PENDING，并且为电源IRP设置IoCompletion例程。在IoCompletion例程中，驱动程序能执行所需的任何任务，然后调用PoStartNextPowerIrp和IoCompleteRequest。例如，当保持系统电源IRP以便设置设备电源状态适合于被请求的系统状态时,设备电源策略所有者常常必须发送一个设备电源IRP。

在这种情况下,电源策略所有者应该在系统电源IRP中设置IoCompletion例程,把系统电源IRP传递到相邻较低层驱动程序中，并且从其DispatchPower例程中返回STATUS_PENDING。在IoCompletion例程中，它调用PoRequestPowerIrp去发送设备电源IRP，把指针传递到请求中的回调例程。IoCompletion例程应该返回STATUS_MORE_PROCESSING_REQUIRED。最后，驱动程序从回调例程中向下传递系统IRP。驱动程序不能在DispatchPower例程中的内核事件上等待，对当前正在处理的IRP也不能用IoCompletion例程通知信号；否则，可能发生系统的死锁。详见第3章中关于“在设备电源策略所有者中处理一个系统设置电源IRP”。

类似的情况，当系统将要休眠时，电源策略所有者在发送设备IRP使其设备掉电之前可以根据需要完成一些未决的I/O。驱动程序应该对工作项目进行排队，然后从DispatchPower例程中返回STATUS_PENDING，而不是在I/O完成时通知一个事件并在DispatchPower例程中等待。在工人线程中，驱动程序等候I/O完成，然后发送设备电源IRP。详见在线DDK中的IoAllocateWorkItem (这个例程不适用于Windows98)。

1.3.3 设备休眠时排队 I/O 请求

当设备处于休眠时，在其设备栈中的驱动程序应该对设备的I/O请求排队。IoAllocateWorkItem，IoQueueWorkItem，和IoFree WorkItem支持例程提供一种对延迟处理排队IRP的方法。这一例子参见第2部分第3章“当设备暂停时保持内入的IRP”中的为PnP驱动程序的排队机制。

只有当设备在Working(D0)状态下时，驱动程序能访问其设备。当设备在休眠状态时，驱动程序不能触及任何设备寄存器，设备必须首先被返回到工作状态。

1.3.4 处理未被支持的或者无法识别的电源 IRP

如果驱动程序不支持特定的电源IRP，那么它必须在设备栈中将IRP传递到相邻较低层驱动程序中。栈中更低的驱动程序可以准备处理IRP，但必须有机会这样做。

为了传递未被支持的或者无法识别的电源IRP，在“传递电源IRP”所描述顺序的中，驱动程序调用 PoStartNextPowerIrp，IoSkipCurrentIrpStackLocation，和PoCallDriver。驱动程序在向下传递IRP之前，不应该在IRP中改变任何事物。

第2章　独立设备的电源管理

作为对系统电源状态变化的响应，驱动程序管理电源，检测和关闭空闲设备，并在需要时给设备上电。本章介绍与设备电源管理有关的以下内容：

关于处理系统电源状态转变的内容，参见第3章“处理系统电源状态请求”，关于IRP_MN_WAIT_WAKE请求的内容，参见第4章“支持具有唤醒能力的设备”。

2.1　设备电源状态

设备电源状态描述特定设备的电源状态。设备电源状态用下列标准术语定义为：

。能源消耗：　多少能源供设备使用？

。设备环境：　设备在此状态中需保持多少操作环境？

。设备驱动程序行为：设备驱动程序必须为设备做些什么才能使设备恢复成为完全操作状态？

。恢复时间：　设备恢复成为完全操作状态需要多长时间？

。唤醒能力：设备能从这种状态请求唤醒吗？通常，如果设备能从一种给定的电源状态(例如 D2)请求唤醒，那么它也能从任何大功率的状态(D1)请求唤醒。

电源状态的确切定义是设备相关的。并非所有设备都能定义所有状态；许多设备仅仅定义D0和D3状态。要了解特定设备的设备状态的定义，以及每种状态的操作需求是什么，参见有关的“设备类型电源管理的参考规范”(可在微软网站www.microsoft.com上找到)。

设备的电源状态不必与系统的电源状态匹配。例如，即使系统是在（S0）工作状态下，某些设备也可以在（D3）关闭状态中。通常，因为许多设备是从系统中得到能量的，所以设备的电源状态没有系统的高。然而，能唤醒系统的设备是例外情况；一般来说，当系统是在休眠状态下时，这些设备就处于一种低潜伏休眠状态(D1或者D2)。

注意，某些设备在一种单一的设备电源状态之内时，有若干不同的低能模式。如果其驱动程序能自动地把设备从一方式转换到另一种方式，而不改变设备电源状态，那么这个设备就能使用这些模式。但是，通常如果在这些方式中没有用户可觉查到的区别，设备应该只使用最低能模式。如果一个低能模式，诸如低速率，影响性能或者除设备驱动程序外对软件是不透明的，硬件不应该自动地使用它。详见“设备类型电源管理的参考规范”。

驱动程序或者电源管理器可以请求设备电源状态的转变，而且所有驱动程序都必须准备处理请求转变的IRP。详见“为设备电源状态发送IRP_MN_QUERY_POWER或IRP_MN_SET_POWER，为设备电源状态处理IRP_MN_QUERY_POWER和IRP_MN_SET_POWER”。

和系统一样，设备也能从工作状态(DO)转变到任何休眠状态(D1-D3)，也能从任何休眠状态转变到工作状态。

2.1.1 设备工作状态 DO

在DO状态下，设备是完全打开和操作的。

能源消耗

设备有最大的连续能源消耗

设备环境

保持所有的环境

设备驱动程序行为

正常操作

恢复时间

不可用

唤醒能力

不可用

2.1.2 设备休眠状态 D1，D2，和 D3

设备状态D1，D2，和D3是设备休眠状态。

D1和D2是休眠状态。许多设备的类别都不定义这些状态。所有设备必须定义状态D3。

设备状态D1

设备状态D1是最高功率的休眠状态。它有下列的特性

能源消耗

其消耗处在小于DO状态而大于或等于D2状态。

设备环境

一般来说，设备环境是由硬件保存，且不需要由驱动程序恢复。

设备驱动程序行为

驱动程序必须能保存、恢复或者再初始化硬件丢失的环境。但是，通常是在设备进入这种状态时就会丢失少量的环境。

恢复时间

一般来说，从D1到D0恢复设备所需要的时间应该小于从D2到D0恢复设备所需要的时间。

唤醒能力

状态D1的设备能够请求唤醒。

为了最大限度地减少用户觉察到的延迟，设备从D1恢复到D0应有可能的最小延迟。在状态转变中，最大限度地减少延迟比减少能源消耗更重要。

设备状态D2

状态D2是具有下列特性的一种中间休眠状态：

能源消耗

消耗小于或者等于在D1状态下时的消耗。

设备环境

一般来说，大多数的设备环境被硬件丢失。

设备驱动程序行为

设备驱动程序必须能保存、恢复或者再初始化被硬件丢失的环境。当它进入D2时，通常设备丢失大多数环境。

恢复时间

从D2到D0恢复设备至少和从D1到D0恢复设备的时间一样长。

唤醒能力

状态D2中的设备能够请求唤醒。

在状态D2中，能源消耗下降到可恢复的最低水平，驱动程序可在合理的时间内恢复设备，即大多数用户不会发现此设备有明显的延迟。状态D1优先考虑用户对延迟感知,状态D2优先考虑节能。因此，通常从状态D2到状态D0的恢复时间超过从状态D1到状态D0的恢复时间。例如，在状态D2中，减少总线上的能量可以使设备关闭一些它的功能。这样，需要更多时间以重新启动设备，并且恢复设备。

许多类型的设备不定义这种状态。

设备状态D3

设备状态D3是最节能的休眠状态.所有设备都必须支持这种状态。它有下列的特征：

能源消耗

设备中和/或者系统中完全断电。

设备环境

设备驱动程序仅仅负责恢复设备环境。驱动程序必须保存、恢复设备环境或者必须在转变成不同的电源状态时重新初始化设备。

设备驱动程序行为

设备驱动程序仅仅负责恢复设备环境，通常从最近的工作配置。

恢复时间

总的恢复时间是其他所有设备状态中最高的。

唤醒能力

在状态D3中的设备不能唤醒一个休眠的系统。

在状态D3中，设备中完全断电或者仅有最小限度的节拍电流可供使用。驱动程序和硬件必须为断电作准备。当电源恢复之后，操作系统提供必要的硬件复位信号。为了使设备返回到工作状态，设备驱动程序必须能恢复和重新初始化设备而不依赖于所运行的BIOS代码。

电源管理器可以为进入D3的父设备中消除系统电源。

所有类型的设备都定义这种状态。

2.1.3 设备电源状态所需的支持

要了解正在工作的设备定义了哪些设备状态，以及每个状态的操作需求是什么，参见“设备类别电源管理的参考规范”。这些规范参见微软网站。

对没有电源管理规范的早期设备和其他设备应该遵循“缺省类别设备电源管理规范”。实质上，缺省规范有以下要求：

。支持D0和D3状态

。当设备被加电时，驱动程序存储、恢复或者重新初始化设备环境

。驱动程序管理设备电源策略

类别驱动程序和端口驱动程序一般支持电源管理。如果你正在写链接到这样一种驱动程序上的微驱动程序，那么检查可类别或者端口驱动程序说明，以查明在微驱动程序中要求的电源管理支持程度。使用下列一般的方法：

。网络适配器驱动程序必须符合NDIS 5.0规范，以及电源管理接口，参见“设备类型电源管理的参考规范”和在线DDK中的“网络驱动程序设计指南”。这些驱动程序不使用卷1“Windows 2000驱动程序开发参考”所描述的接口。

。流式驱动程序在流式类别驱动程序中使用电源管理接口来处理设备状态D0和D3。为了处理设备状态D1和D2，这些驱动程序必须使用在这里以及卷1“Windows 2000驱动程序开发参考”所描述的电源管理接口。

。SCSI端口驱动程序为微端口管理大多数的PnP和电源管理需求。SCSI微端口驱动程序必须支持Microsoft®Windows®2000 PnP和带有相关例程（如HwScsiAdapterControl）的电源管理接口。详见在线DDK中“内核模式驱动程序设计指南”。

。视频端口驱动程序为微端口管理大多数的PnP和电源管理需求。视频微端口驱动程序必须支持微端口特定例程，参见在线DDK中“图表驱动程序设计指南和参考”。视频微端口驱动程序并不使用卷1中的“Windows 2000驱动程序开发参考”所描述的接口。

。所有其他的WDM驱动程序(USB、IEEE 1394等等)和Windows 2000 PnP驱动程序使用类别相关的电源管理接口。参见在线DDK中“核心模式驱动程序设计指南”)或卷1“Windows 2000驱动程序开发参考”所描述的接口，

2.2 管理设备电源策略

正象电源管理器为系统保持和管理电源策略一样，在设备栈中的每一设备都有一个驱动程序为设备保持和管理电源策略。这个驱动程序就是设备的“设备电源策略所有者”。

设备电源策略所有者是拥有有关设备使用和电源状态的大多数信息的驱动程序。实际上，它不必设置给设备加电或断电的设备寄存器，但是必须确定设备何时正被使用、何时处于空闲以及何时能改变电源状态。

一般来说，设备的功能驱动程序是其电源策略所有者，尽管对某些设备其他驱动程序或者系统组件可能会担当上述这个角色。与电源管理有关的驱动程序类型的信息，参见第1部分第1章“驱

动程序的类型”。

一些驱动程序为设备(创建FDO)充当功能驱动程序，为列举的子设备充当总线驱动程序(创建PDO)。在电源和PnP IRP的Dispatch例程中，这种驱动程序必须能区别发送到FDO的IRP的处理和发送到PDO的IRP的处理。

例如，对SCSI适配器，驱动程序可以为适配器本身充当功能驱动程序(创建FDO)，为附属于适配器的磁盘充当总线驱动程序(创建PDO)。此驱动程序作为SCSI适配器的功能驱动程序/策略所有者，它为SCSI适配器接收系统IRP和请求设备IRP。作为磁盘的总线驱动程序，它处理和完成建立磁盘PDO的设备IRP。虽然驱动程序拥有设备(FDO)的电源策略，但并不意味着它也拥有子设备(PDO)的电源策略。

设备电源策略所有者负责下列情况：

。当它处理PnP管理的IRP_MN_START_DEVICE请求时，调用PoSetPowerState设置设备的初始电源状态为D0。

需要时，设备应该加电；例如，设备必须加电去处理一个I/O请求。设备电源策略所有者负责确定何时需要设备，设备是否已经加电，以及设置正确的设备电源状态。通常，PnP启动设备IRP经完成时，设备应该是加电的。

一般来说，大多数设备在没有使用时应该是断电的，甚至系统在工作状态下。

。通过调用PoRequestPowerIrp发送设备电源请求，以响应系统电源请求。

例如，当策略所有者接收一个系统设置电源IRP时，它就发送一个设备设置电源IRP。当系统进入任何休眠状态时，大多数设备进入D3。在DEVICE_CAPABILITY结构中的DeviceState数组列出设备能为每个系统状态保持的最大功率状态(参见第1章中“报告设备电源能力”)。

。检测设备何时是空闲的并将其置于休眠状态，以节能。

电源管理器或是设备策略所有者都能检测空闲设备，并且发送设备电源IRP以改变其状态。详见“检测空闲的设备”。

。需要时，把其设备返回到工作状态。

对一个休眠设备，当一个I/O请求到达时，设备的驱动程序应该把它返回到工作状态。

。需要时，为设备激活或禁止唤醒能力。

设备电源策略所有者发送和取消等待/唤醒IRP，参见第4章“支持具有唤醒能力的设备”。

2.3 为设备电源状态处理 IRP_MN_SET_POWER

设备设置电源IRP为单一设备请求改变状态，并且被发送到此设备的设备栈中的所有驱动程序上。这样的IRP指定I/O栈位置的Power.Type成员为DevicePowerState。

当掉电IRP到达栈中时，驱动程序就处理这些IRP。当上电IRP到达栈中时，驱动程序设置IoCompletion例程，然后当IRP在栈中向上回传时，在IoCompletion例程中处理这些IRP。在典型的设备栈中，驱动程序处理设备设置电源IRP如下：

。大多数过滤驱动程序仅仅把IRP传递到相邻较低层驱动程序中(参见第1章中的“传递电源IRP”)，它传送PoCallDriver的状态作为其DispatchPower例程的返回状态。然而，一些过滤驱动程序，可能还首先需要执行一些设备的相关任务，诸如对内入IRP的排队或者保存设备状态。

。功能驱动程序执行设备的相关任务(诸如完成未决的I/O请求，内入I/O请求的排队，保存设备环境，或者改变设备能量)，如有必要，设置一个IoCompletion例程，并且把设备电源IRP传递到相

邻较低层驱动程序中。它传送PoCallDriver的状态作为其DispatchPower例程的返回状态。
。如果总线驱动程序有能力改变设备电源状态，则调用PoSetPowerState，通知电源管理器新的设备状态，并且调用PoStartNextPowerIrp开始下一电源IRP。然后，完成IRP，指定IO_NO_INCREMENT。如果驱动程序不能立即完成IRP，则调用IoMarkIrpPending，从其DispatchPower例程中返回STATUS_PENDING，然后稍迟完成IRP。

驱动程序不应该失败IRP_MN_SET_POWER请求。

即使目标设备已处于请求的电源状态下，每一功能驱动程序或者过滤驱动程序都必须把IRP传递到相邻较低层驱动程序中。每一个设置电源IRP必须在设备栈中向下传递到能完成IRP的总线驱动程序中。

每当设备改变状态时，其驱动程序都必须调用PoSetPowerState以通知电源管理器此变化。同有关上电、掉电的其他驱动程序任务一样,在设备上电之后(如果新的状态是D0)或者在设备掉电之前(如果新的状态是其他任何状态)调用PoSetPowerState。

每一驱动程序应该追踪其设备的电源状态。电源管理器不向驱动程序提供此信息。

当为设备电源状态处理IRP_MN_SET_POWER请求时，驱动程序应该尽可能迅速地从DispatchPower例程中返回。

驱动程序不能在其DispatchPower例程中,等待由处理同一IRP的代码设置信号量的内核事件。因为电源IRP在整个系统中是同步的，可能会发生死锁。

驱动程序在处理电源IRP必须采取的确切步骤，依赖于设备是在上电还是掉电，如在下列各节中所描述的：
。2.3.1 处理设备掉电IRP
。2.3.2 处理设备上电IRP。

2.3.1 处理设备掉电 IRP

设备掉电IRP指定次功能代码IRP_MN_SET_POWER和设备状态(PowerDeviceD0,PowerDeviceD1，PowerDeviceD2，或PowerDeviceD3),这些状态必须是小功率的或者是等于当前设备电源状态的。当IRP在栈中向下传递时，驱动程序必须处理掉电IRP。高层驱动程序必须在低层驱动程序之前处理IRP。没有设备相关任务执行的驱动程序,应该立即把IRP传递到相邻较低层驱动程序中。

图2.1显示了处理这样的IRP的有关的步骤：

如果IRP指定PowerDeviceD3，通常功能驱动程序将执行下列的任务：
。调用IoAcquireRemoveLock，传递当前IRP，以确保当处理电源IRP时，驱动程序不接收PnP IRP_MN_REMOVE_DEVICE请求。

如果IoAcquireRemoveLock返回一个失败状态，则驱动程序不应该继续处理IRP。相反，它应该调用PoStartNextPowerIrp，然后完成IRP(IoCompleteRequest)，返回失败状态。
。执行任何设备必须在设备断电之前完成的特定任务,，例如关闭设备，完成或者更新任何未决的I/O，禁止中断，排队内入的IRP，以及保存设备环境(可用保存的环境恢复或重新初始化设备)。

当时处理IRP时，驱动程序不应该造成长时间的延迟(例如，对于这种类型的设备用户可以检测到不合理的延迟)。

在设备返回到工作状态之前，驱动程序应该对任何内入的I/O请求进行排队。

图2.1 处理设备掉电请求

。有可能检查Parameters.Power.ShutdownType值。如果系统设置电源IRP是活动的，则ShutdownType提供关于系统IRP的信息。关于这个值,详见第3章中“系统电源行动”。

在冬眠路径上的设备驱动程序必须检查此值。如果ShutdownType是PowerActionHibernate,驱动程序应该保存任何需要的环境以恢复设备，而不应该使设备掉电。

。改变设备的物理电源状态，如果驱动程序有能力这样做，并且改变是适当的。

。调用PoSetPowerState，向电源管理器通知新的设备电源状态。

。调用PoStartNextPowerIrp，表明它已准备好处理下一个电源IRP。

。为相邻较低层驱动程序建立栈位置。

当设备在任何休眠状态下时，驱动程序不能与其设备通信。设备进入休眠状态时，IoCompletion例程是很少使用的。如果驱动程序没有在这个IRP中设置IoCompletion例程，那么它能用IoSkipCurrentIrpStackLocation为相邻较低层驱动程序设置栈位置。否则，驱动程序必须使用IoCopyCurrentIrpStackLocationtoNext。

。调用PoCallDriver，把IRP传递到相邻较低层驱动程序中。IRP必须向下传递到完成IRP的总线驱动程序中。

。调用IoReleaseRemoveLock，释放以前拥有的锁。

。由PoCallDriver传送返回的状态作为从其DispatchPower例程的返回状态。

驱动程序必须保存所有设备环境信息，并且在传发IRP之前设置新的电源状态。环境信息至少应该包含请求新的电源状态。它也应该包括在上电时驱动程序需要的有关任何附加信息。在完成IRP之后，且设备已经掉电的情况下，驱动程序不能再访问设备，同时设备环境不再有用了。

每一驱动程序都必须把IRP传递到相邻较低层驱动程序中。当IRP到达总线驱动程序时，总线驱动程序关闭设备的电源(如果它有这个能力的话)，调用PoSetPowerState通知电源管理器，并完成IRP。

然而，如果总线驱动程序服务于冬眠设备，则它应该检查IRP中的ShutdovnType值是否是PowerSystemHibernate。如果是，总线驱动程序将调用PoSetPowerState以便报告PowerDeviceD3，但是不应该使设备掉电。在冬眠文件以及系统的其余部分保存之后，设备将掉电。

最终，其子设备是要掉电的，总线驱动程序也可以选择使其总线掉电。这样的行为称为设备依赖。

如果IRP指定任何其他状态(D0，D1或者D2)，那么需要的驱动程序行动是设备依赖的。一般来说，支持这些状态的设备当I/O请求到达时，能迅速返回到工作状态。这种设备的驱动程序必须完成任何未决的I/O请求，对所有新的请求进行排队，在转发IRP到相邻较低层驱动程序之前，要保存所有必要的环境。当IRP到达总线驱动程序时，它将硬件设置为请求状态。当它休眠时，驱动程序不能访问设备。

在某些环境下，当设备已处于D0状态时，功能驱动程序或者过滤驱动程序可以收到指定PowerDeviceD0状态的设备电源IRP。驱动程序与处理其他设置电源IRP一样的方法处理此IRP； 完成未决的I/O请求，对内入的I/O请求进行排队，设置一个IoCompletion例程，并且把IRP传递到相邻较低层驱动程序中。但驱动程序不能改变设备的硬件设置。当总线驱动程序收到IRP时，它仅仅完成此IRP。当IRP完成之后，功能驱动程序和过滤驱动程序能处理任何排队请求。当一个高层驱动程序试图I/O时，对I/O请求排队, 直到IRP完成,以消除低层驱动程序试图改变设备寄存器的可能性。

2.3.2 处理设备上电 IRP

设备上电IRP指定IRP_MN_SET_POWER和比当前设备状态更多能量的设备状态。一般来说，一个上电IRP指定设备工作状态PowerDeviceD0。

首先由下层的总线驱动程序为设备处理上电请求，然后通过各层驱动程序在栈中向上回传。

图2.2显示处理上电IRP的有关步骤。

图2.2 处理一个设备上电请求

当为了上电处理一个IRP_MN_SET_POWER请求时，功能驱动程序或者过滤驱动程序必须做到:

。调用IoAcquireRemoveLock，确保驱动程序在处理上电IRP时不会收到IRP_MN_REMOVE_DEVICE请求。

如果IoAcquireRemoveLock返回失败状态，驱动程序不应该继续处理IRP。相反，它应该调用PoStartNextPowerIrp，接着完成IRP(IoCompleteRequest)，然后返回失败状态。

。调用IoSetCompletionRoutine，建立一个上电IoCompletion例程。

当处理设备上电IRP时,在IRP已经完成且设备上电之后,驱动程序应该设置IoCompletion例程以便恢复环境，释放删除的锁，并且执行其他需要的任务。在IRP完成之前，驱动程序不应该恢复环境。详见“设备电源IRPIoCompletion例程”。

。调用IoCopyCurrentIrpStackLocationToNext，设置IRP栈位置。记住，如果一个驱动程序建立IoCompletion例程，则它不能调用IoSkipCurrentIrpStackLocation。

。调用PoStartNextPowerIrp，开始下一个电源IRP。

当IRP栈位置指向当前驱动程序时，驱动程序必须调用PoStartNextPowerIrp。因此，在调用PoCallDriver之前，驱动程序能从DispatchPower程序中或者从IoCompletion程序调用它。

。调用PoCallDriver，把IRP传递到相邻较低层驱动程序中。

IRP必须在设备栈中向下传递到总线驱动程序中，只有总线驱动程序可以完成它。

当总线驱动程序收到IRP时，应该首先检查以确认设备仍然存在并且在休眠时未被移去或替换。如果设备不存在，总线驱动程序将在父设备上调用IoInvalidateDeviceRelations，并通知PnP管理模块设备已经消失。在这种情况下，总线驱动程序可以使电源IRP失败。

如果设备仍然存在，总线驱动程序则执行请求的任务，把设备返回到操作状态中，然后调用PoSetPowerState通知电源管理器新状态,并且完成IRP(IoCompleteRequest)。如果设备正在休眠，驱动程序已排队I/O，或者设备请求启动功率，则总线驱动程序给设备供电。否则，一旦总线驱动程序必须与设备通信时，它立即就给设备供电。

2.3.3 设备电源 IRPIoCompletion 例程

在总线驱动程序完成IRP之后，I/O管理器调用由高层驱动程序注册的IoCompletion例程。

当设备进入D0状态时，不论是从休眠状态返回还是因系统启动而进入D0状态，每个驱动程序通常都建立一个IoCompletion例程，用以完成返回工作状态所需执行的大多数任务。如图2.3中所显示。

图2.3 设备上电IoCompletion例程

这些任务包括：

。根据需要恢复设备状态或者再初始化设备。

。调用PoSetPowerState，向电源管理器通知设备处于D0电源状态。

。当设备处于非工作状态下时,准备处理由驱动程序排队的I/O。

。如果驱动程序最初没有发送当前的电源IRP,则调用PoStartNextPowerIrp去接收下一个电源IRP。

。调用IoCompleteRequest完成IRP。

。释放为设备环境分配的内存。

。当驱动程序收到IRP时，调用IoReleaseRemoveLock，释放驱动程序在其DispatchPower例程中获得的锁。

在总线驱动程序或者高层驱动程序与设备进行通信之前，总线驱动程序不会给设备上电。

当其设备进入休眠状态时，驱动程序能够建立IoCompletion例程，但这样的例程用处不大；当设备不在D0状态时驱动程序不能访问它。

2.4 为设备电源状态处理 IRP_MN_QUERY_POWER

一个设备查询电源IRP用来查询单个设备状态的变化,并且把它发送到设备栈中的所有驱动程序上。这个IRP在I/O栈位置Power Type成员中指定DevicePowerState。

在查询电源IRP向栈底传送期间驱动程序处理它们。

如果下面的一切都成立，功能驱动程序或者过滤驱动程序就能使IRP_MN_QUERY_POWER请求失败：

。设备被激活了唤醒能力，同时请求的电源状态是在设备能唤醒系统的状态之下。例如，从D2而不是从D3唤醒系统的设备可以使D3的查询失败，但可以使D2的查询成功。

。进入请求的状态会迫使驱动程序放弃丢失数据的操作，诸如一个开放的调制解调器连接。驱动程序很少会因为这个原因使查询失败；在大多数情况下，都是应用程序处理的。

为了使一个IRP_MN_QUERY_POWER请求失败，驱动程序采取下列的措施：

1.调用PoStartNextPowerIrp，表明其准备好处理下一个电源IRP。

2.将Irp->IoStatus.Status置为一个失败状态，并且调用IoCompleteRequest，在调用中指定IO_NO_INCREMENT。驱动程序不把IRP在栈中再向下传递。

3.从其DispatchPower例程中返回一个错误状态。

如果驱动程序使查询IRP成功，为了防止其成功地完成后面一个查询电源状态的IRP_MN_SET_POWER请求，它不能开始任何操作或者采取任何行动。

成功完成IRP的驱动程序必须为查询的状态准备一个设置电源IRP，并且向下传递查询IRP，如下：

1.完成任何未完成的I/O操作。

2.排队内入的I/O请求。

3.避免开始任何可能干扰指定电源状态转变的新的活动。然而，驱动程序不会保存设备的环境，也不会为关闭采取其他步骤。

4.调用PoStartNextPowerIrp，表明准备处理下一个电源IRP。

5.调用IoSkipCurrentIrpStackLocation，为相邻较低层驱动程序设置IRP栈位置。

6.调用PoCallDriver，把查询IRP传递到相邻较低层驱动程序上。不要完成IRP。
7.传送由PoCallDriver返回的状态作为从其DispatchPower例程中的返回状态。驱动程序不能改变在Irp->IoStatus.Status中的值。

当查询IRP到达总线驱动程序时，总线驱动程序调用PoStartNextPowerIrp，如果它能变为指定的电源状态，则设置Irp->IoStatus.Status 为STATUS_SUCCESS。如果它不能变为指定的电源状态，则设置一个失败状态。然后总线驱动程序调用IoCompleteRequest，指定IO_NO_INCREMENT。

在一个典型的设备栈中的驱动程序处理设备查询电源IRP，其情形如下：
。大多数的过滤驱动程序仅仅把IRP传递到相邻较低层驱动程序中(参见第1章中“传递电源IRP”)，并且传送由PoCallDriver返回的状态作为从其DispatchPower例程中的返回状态。然而，某些过滤驱动程序，可能首先需要执行设备特定任务，诸如排队内入的IRP或者保存设备状态。
。功能驱动程序执行设备特定任务(诸如完成未决的I/O请求，排队内入的I/O请求，保存设备环境，或者改变设备状态)，必要时建立一个IoCompletion例程，并且把设备电源IRP传递到相邻较低层驱动程序。传送由PoCallDriver返回的状态作为从其DispatchPower例程中的返回状态。
。总线驱动程序调用PoStartNextPowerIrp开始下一个电源IRP。接着它完成IRP，指定IO_NO_INCREMENT。如果驱动程序不能立即完成IRP，则它调用IoMarkIrpPending，从其DispatchPower程序中返回STATUS_PENDING，然后稍迟再完成IRP。

即使目标设备已处于被查询的电源状态下，每一个功能驱动程序或者过滤驱动程序也必须排队I/O，并且把IRP向下传递到相邻较低层驱动程序中。IRP必须在设备栈中向下传递到完成它的总线驱动程序上。

当处理一个IRP_MN_QUERY_POWER请求时，驱动程序会尽可能迅速地从DispatchPower例程中返回。驱动程序不必在DispatchPower例程中等待由处理相同IRP的代码所发出的核心事件信号，因为电源IRP在整个系统中是同步的，可能会发生死锁。

2.5 为设备电源状态发送IRP_MN_QUERY_POWER或者发送IRP_MN_SET_POWER

设备电源策略所有者发送设备查询电源IRP(IRP_MN_QUERY_POWER)，以确定低层驱动程序是否能提供设备电源状态的变化和改变设备电源状态的设备设置电源IRP(IRP_MN_SET_POWER)。(这个驱动程序也能发送一个等待/唤醒IRP，使其设备唤醒以响应一个外部信号，详见第4章。)

当下面的任意一个条件成立时，驱动程序就会发送一个IRP_MN_QUERY_POWER请求：
。驱动程序收到一个系统查询电源IRP。
。驱动程序正在准备把一个空闲设备置于休眠状态，因此必须查询低层驱动程序，确定这样做是否可行。

当下面的任意一个条件成立时，驱动程序就会发送一个IRP_MN_SET_POWER请求：
。驱动程序已确定设备是空闲的、且能置为休眠状态。
。设备正在休眠，必须重新进入工作状态，处理等待的I/O。
。驱动程序收到一个系统设置电源IRP。

驱动程序不能分配自己的电源IRP；电源管理器为此目的提供PoRequestPowerIrp例程。PoRequestPowerIrp分配和发送IRP，并且与PoCallDriver一同，保证所有电源请求适当地同步(参

见第1章“处理电源IRP的规则”)。PoRequestPowerIrp的调用者必须正在IRQL<=DISPATCH_LEVEL下运行。

下列是这个例程的原型：

```
NTSTATUS
PoRequestPowerIrp(
     IN PDEVICE_OBJECT DeviceObject,
     IN UCHAR MinorFunction,
     IN POWER_STATE PowerState,
     IN PREQUEST_POWER_COMPLETE CompletionFunction,
     IN PVOID Context,
     QUT PIRP*Irp OPTIONAL
     );
```

为了发送IRP，驱动程序调用PoRequestPowerIrp，在DeviceObject中指定一个指向目标设备对象的指针,在MinorFunction中指定次IRP代码IRP_MN_SET_POWER或者在IRP_MN_QUERY_POWER，PowerState.Type 中指定值DevicePowerState，在PowerState.State中指定设备电源状态。在Windows 98中，DeviceObject必须指定下层设备的PDO；在Windows 2000中，这个值即能指向PDO也能指向同一设备栈中驱动程序的FDO。

当其他驱动程序都已完成IRP之后，如果此驱动程序必须执行附加的任务，那么它必须在CompletionFunction中传递一个指向电源完成函数的指针。当驱动程序把IRP向下传递到设备栈中时，I/O管理器在调用所有由驱动程序建立的IoCompletion程序之后调用CompletionFunction。

只要设备的电源策略所有者发送设备电源查询IRP，那么它接着就从调用PoRequestPowerIrp中指定回调例程(CompletionFunction)上发送一个设备设置电源IRP。如果查询成功,设置电源IRP将设备设置在查询的电源状态中。如果查询失败，设置电源IRP重新确认当前设备电源状态。重新确认当前状态是很重要的，因为驱动程序排队I/O以响应查询；策略所有者必须发送设置电源IRP，把其设备栈返回到工作状态。

记住，设备的策略所有者不仅发送设备电源IRP，而且当IRP向下传递到设备栈时也处理IRP。因此，这样的驱动程序常常建立一个IoCompletion例程，(和IoSetCompletionRoutine一起)作为其IRP处理代码的一部分，尤其是当设备正在上电时。在CompletionFunction之前，用由其他驱动程序建立的IoCompletion例程按顺序调用IoCompletion例程。详见2.3.3节“设备电源IRP IoCompletion例程”。

因为当调用CompletionFunction时，所有驱动程序都完成IRP，所以CompletionFunction不能用它创建的IRP 调用PoStarNextPowerIrp或者PoCallDriver(但是它调用不同于这些IRP的例程)。相反，这些例程执行由创建IRP的驱动程序请求的附加行为。

如果驱动程序发送响应系统IRP的设备IRP，CompletionFunction可以完成系统IRP(参见第3章中的“在设备电源策略所有者中处理一个系统设置电源IRP”)。

为了响应PoRequestPowerIrp的调用，电源管理器分配电源IRP并且把它发送到设备的设备栈顶端。电源管理器返回一个指向分配的IRP的指针。

如果没有错误发生，PoRequestPowerIrp返回STATUS_PENDING，意思是已经成功地发送IRP，并且是未决完成。如果电源管理器不能分配IRP，或者如果调用者指定了一个无效的电源次IRP代码,则此调用失败。

设备加电的请求，首先必须由位于下面的总线驱动程序为设备处理，然后由栈中的每一个高层驱动程序连续地处理。因此，当发送一个PowerDeviceDO请求时，驱动程序必须保证在IRP完成之

后，其CompletionFunction执行请求的任务，同时设备已经上电。

当关闭设备电源(PowerDeviceD3)时，设备栈中的每一个驱动程序必须保存其所有必要的环境，并且在把IRP发送到相邻较低层驱动程序之前做必要的清理。环境信息的范围和清理范围依赖于驱动程序的类型。功能驱动程序必须保存硬件环境；过滤驱动程序可以按需要保存其自己的软件环境。在这种情形中的CompletionFunction设置能采取与完成的IRP有关的行动，但是驱动程序不能访问设备。

2.6 检测空闲的设备

每一种设备的电源策略所有者负责确定设备何时空闲、并且把它置为休眠状态以节能。策略所有者有两种方法检测空闲设备：

。用电源管理器例程进行空闲检测

。自己执行设备特定的空闲检测

2.6.1 用于空闲检测的电源管理器程序

电源管理器通过PoRegisterDeviceForIdle和PoSetDeviceBusy例程为空闲检测提供支持。

为了使设备能进行空闲检测，设备电源策略所有者调用PoRegisterDeviceForIdleDetection，并指定下列情况。

。系统为追求性能进行优化时用的空闲超时值

。系统为追求节能进行优化时用的空闲超时值

。空闲时设备应该转向的设备电源状态

PoRegisterDeviceForIdleDetection返回一个空闲计数器指针，以后驱动程序将用此指针禁止空闲检测。PoRegisterDeviceForIdleDetection的调用者必须在IRQL<DISPATCH_LEVEL上运行。

在设备已经启动、且已准备好处理设备电源IRP之后，驱动程序能在任何时间为空闲检测注册其设备。例如，对一个PnP启动设备IRP，驱动程序可以激活空闲检测作为其IoCompletion例程的一部分。

任何给定设备的超时值应该与设备的上电等待时间成正比，并基于观测到的设备行为。对于一定类型的设备，驱动程序能指定节能和性能超时值为-1，以对设备的类型使用标准电源策略超时值。详见设备特定文档。

当设备使用时，驱动程序必须调用PoSetDeviceBusy，传递由PoRegisterDeviceForIdleDetection返回的指针。PoSetDeviceBusy复位空闲计数，因此为设备重新启动空闲计数。驱动程序应该在每一个I/O操作中调用PoSetDeviceBusy。

为了确定设备是否空闲，电源管理器为当前系统电源策略(节能或者性能)把空闲计数器的值与驱动程序指定的空闲超时值进行比较。对系统电源策略常用的功能参见“平台SDK”。

当设备满足超时值时，电源管理器发送一个设备设置电源IRP，指定驱动程序在PoRegisterDeviceForIdleDetection调用中传送的设备电源状态。电源管理器在发送此设置电源IRP之前不发送查询IRP。在栈中的驱动程序处理此设置电源IRP。驱动程序必须以某种适当的方式

完成它，而且不能失败。(参见“处理设备掉电IRP”)。

当驱动程序不再请求空闲检测或者不再准备处理设备掉电IRP时，它就会再一次调用PoRegisterDeviceForIdleDetection为超时值传递零值。为节能(电池)和性能(AC)电源策略，设置超时值为零，以禁止空闲检测。驱动程序能迅速地重新激活空闲检测； 它只需要用非零超时值调用PoRegisterDeviceForIdleDetection。一旦驱动程序注册了设备，它就能够通过改变超时值激活或禁止空闲检测，即使设备已经掉电或者再次被唤醒。

2.6.2 执行设备特定的空闲检测

驱动程序能够执行基于设备特定标准的自己的空闲检测，而不使用电源管理器的空闲检测例程。

这种驱动程序应该将其空闲设备放进最低可能的休眠状态中，此状态对当前系统状态是有效的。为此，驱动程序用次IRP代码IRP_MN_SET_POWER请求电源IRP(PoRequestPowerIrp)，把设备电源状态指定到设备应该转变的状态上。

第3章　处理系统电源状态请求

所有驱动程序都必须能够响应系统电源状态的请求，而不管系统是成功地处于休眠，冬眠，还是在唤醒状态。

如果任何驱动程序都不支持系统电源管理，独立设备可以休眠或者唤醒，但是电源管理器不能使整个系统进入休眠状态中。

本章描述驱动程序应该如何响应改变或者查询系统电源状态的IRP。 它包括下列主题：

3.1　系统电源状态

系统电源状态描述整体系统的能源消耗。操作系统支持六种系统电源状态，即从S0(充分加电和操作)到S5（断电）。每一状态具有下列特征：

。能源消耗：计算机使用多少功率电源？

。软件恢复：操作系统从什么点重新启动？

。硬件等待时间：计算机返回工作状态需要多少时间？

。系统环境：要保持多少系统环境？操作系统必须重新启动以返回工作状态吗？

状态S0是工作状态。状态S1 、S2 、S3，和S4是休眠状态，因为要降低能源消耗，所以在这些状态中，计算机显示关闭形式，可是它保持着足够的环境，这样不用重新启动操作系统就能返回到工作状态。状态S5是中断或者关机状态。

当系统从关机状态(S5)或者任何一个休眠状态(S1-S4)转变到工作状态(S0)时，系统进入唤醒

状态。当系统从工作状态转变到任何一个休眠状态或者关机状态时，系统进入休眠状态。图3.1显示系统电源状态的几种可能的转变。

图3.1 系统电源状态转变的可能

从图中可以看到，系统不能从一种休眠状态直接进入另一种休眠状态；它必须在进入另一种休眠状态之前先进入工作状态。例如，系统不能从状态S2转变到S4，也不能从状态S4转变到S2。它必须首先返回到状态S0，然后才能再进入下一个休眠状态。 因为处于一种中间休眠状态的系统已经失去某种操作环境，在它进行其他状态转变之前，它必须返回工作状态以恢复该环境。

3.1.1 系统工作状态 S0

系统状态S0是系统的工作状态，具有下列特点：

能源消耗：

最大值。当然，单独设备的电源状态能够动态地改变在个别设备基础上的电源保护。未使用的设备可以根据需要掉电或者加电。

软件恢复：

不适应。

硬件等待时间：

没有。

系统环境：

保持所有环境。

3.1.2 系统休眠状态 S1，S2，S3，S4

S1，S2，S3，和S4状态都是休眠状态。在这些状态中的系统不执行任何计算任务，并且显示关闭形式。但是它不同于关机状态(S5)中的系统，休眠系统保持存储状态，或者是在硬件上或者是在磁盘上。操作系统不需要重新启动就能使计算机返回到工作状态。当某些事件发生时，一些设备能从休眠状态唤醒系统，例如对调制解调器的调用。此外，在一些计算机上，外部指示器还告诉用户系统只是处于休眠状态。

从S1到S4，在顺序的每一种休眠状态中，大多数计算机关机。所有ACPI顺序计算机在S1中关闭处理器时钟，并且在S4丢失系统环境(除非在关机之前已写好“冬眠”文件)，就像本节下面几页中所列出的。休眠状态中间的一些细节情况主要取决于生产商如何设计机器。例如，在一些机器的某些芯片上，主板在状态S3上可能会掉电，而在其他一些类似的芯片上，直到状态S4还将保持电源。进一步说，一些设备或许只能从状态S1唤醒系统，而不能从较深层的休眠状态唤醒系统。

系统状态S1

系统状态S1是具有下列特点的一种休眠状态：

能源消耗：

小于在S0中的消耗，而大于在其他休眠状态的消耗。处理器时钟是关闭的，总线时钟是停止

的。

软件恢复：

控制在其停止的地方重新启动。

硬件等待时间：

一般来说，不超过两秒时钟。

系统环境：

所有环境都由硬件保持和维护着。

系统状态S2：

除了CPU环境不同以外，系统状态S2类似于S1。由于处理器失去较低，所以系统高速缓存的内容被丢失。状态S2具有下列特点：

能源消耗：

小于在状态S1中的消耗，而大于在S3中的消耗。处理器时钟是关闭的，总线时钟是停止的。一些总线可能会掉电。

软件恢复：

在唤醒之后，从处理器的重置矢量开始控制。

硬件等待时间：

两秒或者更长时间，大于或者等于S1的等待时间。

系统环境：

CPU环境和系统高速缓存内容都已丢失。

系统状态S3：

系统状态S3是具有下列特点的一种休眠状态：

能源消耗：

小于状态S2中的消耗。处理器是关闭的，主板上的一些芯片也可能是关闭的。

软件恢复

在唤醒事件之后，控制从处理器的重置矢量开始。

硬件等待时间：

几乎与S2没有什么区别。

系统环境：

只有系统存储器被保留，而CPU环境、高速缓存内容、芯片上设置的环境都已丢失。

系统状态S4：

系统状态S4，即冬眠状态，是最低功率的休眠状态，并且有最长的唤醒等待时间。为了把能源消耗降低到最小，所有设备的硬件电源都是关闭。然而，操作系统环境被保持着，在进入S4状态之前系统就把它写在磁盘的那个冬眠文件上(内存映像)。当重新启动时，装入例程阅读此文件，并且跳到系统的上一个位置，即冬眠前的位置。

如果在状态S1，S2或者S3中的计算机失去了所有AC或者电池电源，那么它也就失去了系统环境，因此必须重新启动返回到S0。然而在状态S4中的计算机，甚至在它失去电池或者AC电源之后还能从其以前的位置重新启动。因为系统环境保存在冬眠文件中。在冬眠状态的计算机不使用电源(稀疏的电流是个例外情况)。

状态S4具有下列的特点：

能源消耗：

除了稀疏的电流会流到电源开关和类似的设备上之外，其余都是关闭的。

软件恢复：

系统从保存的冬眠文件中重新启动。 如果冬眠文件不能被装载，就要求重新启动。当系统在S4状态下时，重新配置的硬件可能会导致某些变化，阻止了冬眠文件的正确装载。

硬件等待时间：

等待时间长且未定义。只有物理互动才使系统返回到工作状态。这种互动可以包括：用户按下“ON”开关；如果有适当的硬件而且又能被唤醒，LAN上的调制解调器或活动的输入环等。如果硬件支持它，那么机器也能从恢复的定时器中唤醒。

系统环境：

硬件中没有保留任何内容。在掉电之前，系统在冬眠文件中写了一个存储映象。当操作系统被装载时，它阅读此文件并且跳到其以前的位置上。

3.1.3 系统关机状态 S5

在S5状态或者称为关机状态中，机器没有存储状态，并且不执行任何计算任务。

状态S4和S5之间唯一的区别是：在状态S4中计算机能从冬眠文件中重新启动，而在状态S5中重新启动需要重新引导系统。

状态S5具有下列特点：

能源消耗：

除了流到电源按钮的稀疏电流之外，其余全是关闭的。

软件恢复：

在唤醒时要求引导。

硬件等待时间：

等待时间长且未定义。只有物理互动，诸如用户按下“ON”开关，使系统返回到工作状态。如果系统这样配置，BIOS也能从恢复的定时器上被唤醒。

系统环境：

没有保留任何内容。

3.1.4 系统电源动作

当电源管理器发送IRP设置或者查询系统电源状态时，它与附加的参数一同指定系统电源状态，给出有关电源状态变化的信息。这个参数，在Irp->Parameters.Power.ShutdownType中是一个POWER_ACTION类型的枚举变量。这个枚举变量描述系统电源状态的请求，如表3.1中所示：

表3.1系统电源动作

POWER_ACTION 枚举变量	要求的系统电源状态
PowerActionNone	S0或者无系统电源IRP活动
PowerActionSleep	S1 、S2，或者S3
PowerActionHibernate	S4

PowerActionShutdown(只有Microsoftz@Windows@2000)	S5
PowerActionShutdownReset	S5
PowerActionShutdownOff	S5

对于状态S5， 当驱动程序收到一个系统查询IRP或者设置电源IRP时，它能对更多关于请求关机的信息检查ShutdownType。当机器正在重置而不是不确定的关闭电源时，驱动程序能用这个信息来优化它的关机序列。当系统重置时，大多数设备的驱动程序保持着电源。然而，对某些设备，诸如执行DMA的视频流设备，当系统重置时，驱动程序可以选择其设备掉电，这样就可以停止任何正在进行的I/O操作。

当设备电源策略所有者把设备电源IRP发送到其设备栈中以响应系统电源IRP时，驱动程序能使用ShutdounType参数得到有关当前系统电源IRP的信息。在这种情况下，ShutdownType值表明当前请求的系统电源状态，或者在系统要求并不明显时它是PowerActionNone。然而，如果设备IRP请求状态D0，驱动程序应该不依赖于这个信息。

在Windows 98上，当IRP请求设备电源状态时，这个字段总包含PowerActionNone。

3.2 系统电源策略

在系统电源策略管理模块中，电源管理器追踪系统的活动，确定适当的系统电源状态，并且发送IRP_MJ_POWER请求，以查询或者改变系统电源状态。它也通过应用程序读写电源策略的设置提供接口(参见平台SDK)。

电源管理器支持两项不同的电源策略—— 一个是对AC（畴壁电流）另一个是对DC(电池或者UPS)，并且取决于当前电源在这两种电源策略之间的自动转换。一般来说，AC电源策略注重保护超过注重性能，而DC电源策略注重性能超过注重保护。为了查明系统何时从一种策略变成另一种策略，驱动程序能够注册带有系统\Callback\PowerState回调对象的通知。详见“Windows2000驱动程序开发参考”第2卷中的ExCreateCallback和“内核模式驱动程序设计指南”第1部分第3章中的Callback对象。

按照APCI说明的计算机能自动地从AC转换到电池电源，或者从一个电池转换到另一个电池，就象每一个这类电源断电一样。如果计算机硬件允许操作系统选择电源，那么电源管理器跟踪最近充电且功能依旧的电池，并且选择此电池作为计算机的电源。

一旦AC电源变得可供使用，计算机硬件便自动地开始给电池充电。如果硬件允许操作系统选择为哪一个电池充电，那么电源管理器就会选择放电最少的电池去再充电；这样任何时候系统至少具有一个充好电的电池的机会就增大了。

无论其他设置是什么，只要电池具有再充电能力或者提供的系统电源报告硬件条件“临界”，并且在两秒或者更长时间是处于放电状态中，那么电源管理器为临界电池执行DC电源策略。这种状态下的电源策略一般要求转变为冬眠状态或者关机状态。

3.3 防止系统电源状态变化

虽然驱动程序不能直接设置系统电源策略，可是电源管理器提供三个例程使驱动程序能防止系统的转变超越工作状态之外：PoSetSystemState，PoRegisterSystemState，和PoUnregisterSystemState。

通过调用PoRegisterSystemState或者PoSetSystemState,驱动程序能够向电源管理器通知用户已经到场或者驱动程序请求使用系统或者显示。

PoRegisterSystemState允许驱动程序注册一个连续地繁忙状态。由于驱动程序能稍迟改变其设置，它返回一个回柄。一旦状态注册生效，电源管理器就不能试图使系统进入休眠状态。驱动程序通过调用PoUnregisterSystemState撤消状态注册。

有了PoSetSystemState，驱动程序就能通知相同条件的电源管理器(用户在场，系统请求，显示请求)，但是这个设置不是连续的。它具有重新启动与指定条件有关的递减计数器的作用。

使用这些例程，驱动程序能阻止许多、但并非所有超越工作状态之外的转变。当电源损失是非常紧迫的、或者当用户明确请求关机时，电源管理器总是关闭系统。

3.4 为系统电源状态处理 IRP_MN_QUERY_P0WER

电源管理器发送带有次IRP码IRP_MN_QUERY_POWER和Parameters.Power.Type中的SystemPowerState的电源IRP，以确定它是否能安全地变为一个指定系统电源状态(S1-S5)，并且允许驱动程序为这一变化做准备。

只要可能,电源管理器在发送一个较低(更小功率)状态请求IRP_MN_SET_POWER之前首先查询。然而，在失效的电池或者电源损失非常紧迫的情况下，电源管理器不先查询就直接发送设置电源IRP。在发送一个IRP将系统设置在工作状态之前，电源管理器决不发送查询请求。

对系统电源状态的IRP_MN_QUERY_POWER请求并不要求动作,除非驱动程序必须使IRP失效或者为其设备栈管理电源策略。大多数过滤驱动程序仅仅把IRP传递到相邻较低层驱动程序中，直到它到达完成它的总线驱动程序。驱动程序决不发送设备IRP_MN_SET_POWER请求以响应系统查询；它仅在收到系统设置电源请求之后，才请求这个IRP。

关于如何处理系统电源查询的更多信息参见下列内容：

。3.4.1 使系统查询电源IRP失效

。3.4.2 在设备电源策略所有者中处理系统查询电源IRP

。3.4.3 在总线驱动程序中处理系统查询电源IRP

。3.4.4 在过滤驱动程序中处理系统查询电源IRP

因为电源管理器把系统查询IRP发送到系统的每一个设备栈中,一种设备的驱动程序可能使查询失效，而其他设备的驱动程序则能成功地完成它。也可能电池用尽而查询激活,要求立即关机。因此，在查询IRP之后，每一个驱动程序必须准备接收下列任何电源IRP：

。查询状态的IRP_MN_SET_POWER

。不同电源状态的IRP_MN_SET_POWER

。当前电源状态的IRP_MN_SET_POWER

。任何状态的IRP_MN_QUERY_POWER

然而，一般来说，驱动程序收到一个跟随系统查询IRP之后的系统设置电源IRP。如果查询成功，则设置电源IRP指定查询状态。如果查询失效，则设置电源IRP重新确定当前电源状态。

3.4.1 使系统查询电源 IRP 失效

如果下列两者之一为真,功能驱动程序或者过滤驱动程序就使IRP_MN_QUERY_POWER请求失效：
。设备为唤醒被激活，并且请求系统电源状态的功率比SystemWake的值小，SystemWake的值指定了设备能唤醒系统的最小功率状态。例如，从S2而不是从S3中能唤醒系统的设备，对状态S3将是一个失效的查询，但是对状态S2可能是一个成功的查询。
。进入符合请求状态的设备状态会使驱动程序放弃丢失数据的操作，诸如打开调制解调器连接。由于这个原因，驱动程序很少丢失查询；在大多数情况下，应用例程处理这样的情况。

为了使IRP_MN_QUERY_POWER请求失效，驱动程序应该采取下列措施：
1.调用PoStartNextPowerIrp，表明其已准备处理下一个电源IRP。
2.设置Irp->IoStatus.Status为失效状态，并且调用IoCompleteRequest，指定IO_NO_INCREMENT。不把IRP传递到更下面的设备栈中。
3.调用IoReleaseRemoveLock，释放以前获取的锁。
4.从其DispatchPower例程中返回一个失效状态。

3.4.2 在设备电源策略所有者中处理系统查询电源 IRP

当设备电源策略所有者为系统状态接收IRP_MN_QUERY_POWER时,它通过传递下来的查询请求，并且在IoCompletion例程中为设备电源状态发送一个IRP_MN_QUERY_POWER作为响应。当栈中所有的驱动程序都已完成设备查询时，设备电源策略所有者完成系统查询。拥有设备电源策略的驱动程序应该采取下列措施在其DispatchPower例程中响应一个系统查询请求：
1.调用IoAcquireRemoveLock，传递当前的IRP，保证在处理电源IRP时，驱动程序不接收PnP IRP_MN_REMOVE_DEVICE请求。如果IoAcquireRemoveLock返回失效状态，驱动程序不应该继续处理这个IRP。相反，它应该调用PoStartNextPowerIrp，然后，完成这个IRP(IoCompleteRequest)，并且返回失效状态。
2.保证驱动程序能支持查询的系统电源状态，如在“使一个系统查询电源IRP失效”中所描述的。如果不能保证，就用在那节中所描述的失效状态完成这个IRP。

然而，如果设备具有唤醒能力，但它不能从冬眠状态唤醒系统，则其驱动程序对状态S4不能使一个查询请求失效。在这种情况下，对于驱动程序(发送IRP_MN_WAIT_WAKE)电源策略所有者必须撤消这个等待唤醒的IRP，并且完成系统查询。详见第4章中“撤消一个等待/唤醒的IRP”。
3.如果驱动程序能支持查询系统电源状态，则在系统查询电源IRP中设置一个IoCompletion例程。
4. 通过调用IoCopyCurrentIrpStackLocationToNext，为相邻较低层驱动程序创建IRP栈位置。
5.调用PoCallDriver，把IRP传递到相邻较低层驱动程序中。
6.从其DispatchPower例程中，由IoCallDriver传递返回的状态。

IoCompletion例程应该做下列事情：

1。 检查Irp->IoStatus.Status，保证低层驱动程序成功地完成IRP。如果任何驱动程序已失效于IRP，调用IoCompleteRequest完成此IRP，并且返回失效状态。
2.如果低层驱动程序已经成功地完成这个IRP，调用PoRequestPowerIrp，为所查询的系统状态是有效的设备状态发送一个设备查询电源IRP。如果必要的话，根据在DEVICE_CAPABILITIES结构中的DVICE_STATE数组，确定查询的系统状态中哪一个设备状态是有效的。
3.在调用PoRequestPowerIrp中,指定一个回调例程(CompletionFunction)并且在环境区域内传递系统IRP。
4.从IoCompletion例程中返回STATUS_MORE_PROCESSING_REQUIRED,这样驱动程序在回调例程中能完成处理中的系统查询IRP。

在IRP已经完成之后，并且在IRP处理过程中所创建的所有IoCompletion例程都已运行的情况下，电源管理器通过I/O管理器调用电源策略管理模块的回调例程。此回调例程必须依次执行下列操作：
1. 调用 PoStartNextPowerIrp，开始下一个电源IRP。
2. 对设备查询电源IRP，用返回的状态完成系统查询电源IRP(IoCompleteRequest)。
3. 调用IoReleaseRemoveLock，释放以前所获得的锁。
4. 用完成的查询IRP返回状态。

设备电源策略所有者不仅发送设备查询，而且还必须在其进入设备栈时处理它。详见第2章中的“为设备电源状态处理IRP_MN_QUERY_POWER”。

3.4.3 在总线驱动程序中处理系统查询电源 IRP

当系统查询IRP到达总线驱动程序中时,总线驱动程序保证它能支持符合查询系统状态的设备状态；同时，如果能够等待/唤醒，则查询系统状态将不阻止其设备唤醒系统。

总线驱动程序调用PoStartNextPowerIrp，如果它能变为指定的电源状态，则在STATUS_SUCCESS中设置Irp->IoStatus.Status；如果它不能变为指定的电源状态，则设置一个失效状态。然后总线驱动程序调用IoCompleteRequest，指定IO_NO_INCREMENT。

在总线驱动程序完成IRP之后，电源管理器调用IoCompletion例程，这个例程是由传递IRP到栈中的其他驱动程序所创建的。

3.4.4 在过滤驱动程序中处理系统查询电源 IRP

对于所有过滤驱动程序和功能驱动程序，如果它们并不拥有其设备栈的电源策略，那么它们就应该把一个系统查询电源IRP传递到相邻较低层驱动程序中，其步骤如下：
1.调用IoAcquireRemoveLock，传递当前IRP，保证驱动程序当处理电源IRP时不能接收PnP IRP_MN_REMOVE_DEVICE请求。

如果IoAcquireRemoveLock返回失效状态，则驱动程序不应该继续处理此IRP。相反，驱动程序应该调用PoStartNextPowerIrp，然后完成此IRP，并且返回失效状态。
2.确定它是否应该使查询失效。其方法和完整的处理过程参见3.4.1“使系统查询电源IRP失效”。

3.调用PoStartNextPowerIrp。
4.设置此IRP栈位置(IoSkipCurrentIrpStackLocation或者IoCopyIrpStackLocationToNext)。虽然驱动程序能在此IRP中创建一个IoCompletion例程，但是这样做没有太大的必要。
5.把这个IRP传递到相邻较低层驱动程序中(PoCallDriver)。
6.调用IoReleaseRemoveLock。如果驱动程序为此IRP创建了一个IoCompletion例程，那么用IoCompletion例程代替这次的调用。
7. 传递由PoCallDriver返回的状态作为其DispatchPower例程的返回状态。

3.5 为系统电源状态处理 IRP_MN_SET_POWER

电源管理器发送一个电源IRP，指定次代码IRP_MN_SET_POWER和一个系统电源状态,由于下列原因之一：
。改变系统电源状态
。在一个失效的IRP_MN_QUERY_POWER请求之后，重新指定当前的电源状态

通过I/O管理器，电源管理器在每一个PnP设备节点上把IRP发送到设备栈中的顶端驱动程序。这个IRP通知当前系统电源状态栈中的所有驱动程序。

为了保证正常启动，电源管理器对系统加电IRP排序，所以在子设备加电之前，父设备有机会加电。电源管理器在发送系统电源IRP之前不进行查询。

为了保证计算机能正常休眠或关机，电源管理器发送系统IRP，指定休眠、冬眠，或者关机都处于一种定义好的序列中，这样离根较远的设备要比离根近的设备先掉电。只要可能，电源管理器在发送这样一个IRP之前要进行查询。详见“为系统电源状态处理IRP_MN_QUERY_POWER”。

系统电源IRP并不直接请求改变电源状态——它只是发出一个通知。驱动程序不能直接响应系统电源IRP以改变其设备的电源状态,而只能响应设备电源IRP来改变其设备的电源状态(设备电源策略所有者发送设备电源IRP，参见“在设备电源策略所有者中处理系统设置电源IRP”)。

对请求的系统电源状态，即使设备已经在一个有效的设备电源状态中，每一个驱动程序也必须把系统设置电源IRP传递到相邻较低层驱动程序中，直到它到达总线驱动程序。只有总线驱动程序允许完成这个IRP。驱动程序根据它在设备栈中的作用，处理这个IRP，在下面几节中描述：
。3.5.1在设备电源策略所有者中处理系统设置电源IRP
。3.5.2在总线驱动程序中处理系统设置电源IRP
。3.5.3在过滤驱动程序中处理系统设置电源IRP

驱动程序不能使IRP_MN_SET_POWER请求失效。电源管理器忽略此IRP返回的任何失效状态。

3.5.1 在设备电源策略所有者中处理系统设置电源 IRP

对每一个系统设置电源IRP中指定的系统电源状态,在每一个设备栈中的设备电源策略所有者负责将其设备栈放到一个适当的设备电源状态中。

当设备电源策略所有者为系统电源状态接收IRP_MN_SET_POWER，它通过传递下来的IRP响应，同时在IoCompletion例程中，为一个相应的设备电源状态发送IRP_MN_SET_POWER。当栈中所有的

驱动程序都已完成这个设备IRP时，设备电源策略所有者才完成系统IRP。

详见下列两节：

。3.5.1.1确定当前的设备电源状态

。3.5.1.2发送一个响应系统设置电源IRP的设备设置电源IRP

3.5.1.1 确定当前的设备电源状态

电源策略所有者能在DEVICE_CAPABILITY结构中,根据Device_State数组对每一系统状态确定设备电源的有效范围。对每个系统状态，数组列出位于下层的设备能支持的最大功率。

当在这个范围中选择特定状态时，需考虑下列情况：

。当系统进入S0状态时，大多数设备进入S0状态。

。当系统进入任何休眠状态时，大多数设备进入D3状态。然而，能够被唤醒的设备如果支持这些状态，则可以请求代替进入D1或者D2。详见第1章中“报告设备电源能力”。

。申请特殊规则的设备将掌握冬眠文件。如果系统IRP请求PowerSystemHibernate，掌握冬眠文件的设备决不能掉电。这个设备的电源策略所有者应该请求设备状态D3，并且保存环境。但是设备的驱动程序决不能使设备掉电。

如果系统IRP请求PowerSystemHibernate，驱动程序将在Irp->Parameter.Power.ShutdownType中检查POWER_ACTION的值,以获得更多关于状态变化原因的信息。详见3.1.4节“系统电源行动”。

设备电源策略所有者必须为每一个系统设置电源IRP发送设备设置电源IRP，即使设备已经处于当前的设备状态中。如果设备预先已经暂停操作响应查询电源IRP，则设置电源IRP就会通知设备停止IRP的排队，并且为其当前的电源状态返回正常的操作。 当设备处于D3状态时，唯一的例外情况是驱动程序不必对D3发送一个附加的IRP_MN_SET_POWER请求。

3.5.1.2 发送一个响应系统设置电源 IRP 的设备设置电源 IRP

设备电源策略所有者应该采取下列措施来响应一个系统设置电源IRP：

1.调用IoAcquireRemoveLock，传递当前的IRP，保证当处理电源IRP时驱动程序不接收PnP IRP_MN_REMOVE_DEVICE请求。

如果IoAcquireRemoveLock返回失效状态，驱动程序不应该继续处理IRP。相反，它应该调用PoStartNextPowerIrp，完成IRP(IoCompleteRequest)，并且返回失效状态。

2. 在系统设置电源IRP中创建IoCompletion例程。

3. 通过调用IoCopyCurrentIrpStackLocationToNext,为相邻较低层驱动程序创建IRP栈位置。

4.调用PoCallDriver，把IRP传递到相邻较低层驱动程序。

5.传递由PoCallDriver返回的状态作为其DispatchPower例程的返回值。

在IoCompletion例程中，驱动程序发送一个设备设置电源IRP的过程如下：

1.首先检查IRP得到请求的系统电源状态，然后为此系统电源状态选择一种适当的设备电源状态。详见3.5.1.1节“确定当前的设备电源状态”。

2.调用PoRequestPowerIrp，为以步骤1所确定的设备电源状态发送IRP_MN_SET_POWER。 即使设备已经处在那种状态中，电源策略所有者也必须发送设备SET_POWER请求。

3.在调用PoRequestPowerIrp指定回调例程(CompletionFunction)，并且在环境区域中传递系统IRP。
4.从IoCompletion例程中返回STATUS_MORE_PROCESSING_REQUIRED,这样驱动程序能在回调例程中完成处理系统设置电源IRP。

记住,设备电源策略所有者不仅能发送设备设置电源IRP,而且当此IRP穿过设备栈还必须能处理它。因而，设备电源策略所有者不仅可以包含一个与设备设置电源IRP有关的回调例程和一个系统设置电源IRP所创建的IoCompletion例程，而且还可以包含一个为设备设置电源IRP所创建的IoCompletion例程。详见第2章中“为设备电源状态处理IRP_MN_SET_POWER”。

当设备设置电源IRP传递到设备栈中时，所创建的IoCompletion例程已经运行，回调例程被调用。这时，栈中的所有驱动程序都已完成设备设置电源IRP，同时设备电源的转变也已完成。

回调例程必须处理下列情况：
1.调用PoStartNextPowerIrp，开始下一个电源IRP。
2. 对于设备设置电源IRP，用返回的状态完成系统设置电源IRP(IoCompleteRequest)。
3. 调用 IoReleaseRemoveLock，释放以前获取的锁。
4. 用完成的设置电源IRP返回此状态

3.5.2 在总线驱动程序中处理系统设置电源 IRP

当总线驱动程序接收一个系统设置电源IRP时，它必须采取下列的措施：
1. 调用PoStartNextPowerIrp，开始下一个电源IRP。
2. 把Irp->IoStatus.Status设置为STATUS_SUCCESS。驱动程序不能使系统设置电源IRP失效。
3. 调用IoCompleteRequest，指定IO_NO_INCREMENT，完成IRP。

总线驱动程序不能改变设备电源的设置，直到它收到一个请求设备电源状态的电源IRP。

3.5.3 在过滤驱动程序中处理系统设置电源 IRP

对于所有过滤驱动程序和功能驱动程序，如果它们并不拥有其设备栈的电源策略，那么它们就仅仅把系统设置电源IRP传递到相邻较低层驱动程序中，其步骤如下：
1. 调用IoAcquireRemoveLock，传递当前IRP，保证当处理电源IRP时驱动程序不能接收PnP IRP_MN_REMOV_DEVICE请求。

如果IoAcquireRemoveLock返回失效状态，则驱动程序不应该继续处理IRP。相反，驱动程序应该调用PoStartNextPowerIrp,然后完成IRP，并且返回失效状态。
2. 调用PoStartNextPowerIrp。
3. 设置此IRP栈位置(IoSkipCurrentIrpStackLocation或者IoCopyIrpStackLocationToNext)。虽然驱动程序能在此IRP中创建IoCompletion例程，但是这样做的必要性不大。
4. 把这个IRP传递到相邻较低层驱动程序中(PoCallDriver)。
5. 调用IoReleaseRemoveLock。如果驱动程序为此IRP创建了一个IoCompletion例程，那么用此IoCompletion例程代替这次的调用。

6. 传递由PoCallDriver返回的状态作为其DispatchPower例程的返回状态。

第4章　支持具有唤醒能力的设备

能响应外部唤醒信号设备的驱动程序必须能够处理IRP_MN_WAIT_WAKE请求(等待/唤醒IRP)。这种设备电源策略所有者必须能够发送一个IRP_MN_WAIT_WAKE请求。

一般来说，无论什么原因，设备的唤醒信号也是设备的一个正常的服务事件。例如，用户输入时，可以使键盘唤醒系统，键盘和其驱动程序就是一个正常的服务事件。

所有驱动程序的开发者都应该阅读本章的第一段，它概述了等待/唤醒操作。后面各段提供关于处理和发送等待/唤醒IRP。

本章包括下列的主题：

4.1 等待/唤醒操作综述

实际上，Microsoft@Windows@2000和WDM唤醒机制的工作情况如图所示。

图4.1　IRP_MN_WAIT_WAKE处理综述

1.当系统和设备处于工作状态时，设备的电源策略所有者确定其设备应该何时被唤醒(“待命”)。电源策略所有者请求一个电源IRP(带有次代码IRP_MN_WAIT_WAKE的PoRequestPowerIrp)，这个电源IRP被发送到PDO用以通知其设备栈中所有驱动程序的。在这个请求中，策略所有者指定一个回调例程(并非与IoCompletion例程一样)。

2.电源管理器，通过I/O管理器，把IRP发送到设备栈的顶端。

3.驱动程序设置IoCompletion例程，并且向下传递IRP直到它到达总线驱动程序。

4.总线驱动程序能够唤醒物理设备，如果确定能够唤醒，标明IRP为未决状态。必要的话，总线驱动程序也为其父设备请求一个等待/唤醒IRP。

5.稍后，外部唤醒信号到达。
6.总线驱动程序完成IRP_MN_WAIT_WAKE。
7. 当驱动程序把IRP向下传递到栈中时，I/O管理器调用所创建的IoCompletion例程。
8.当I/O管理器请求此IRP时，它就调用由策略所有者创建的回调例程。

IRP_MN_WAIT_WAKE请求不能改变设备或者系统的电源状态。如果设备进入一种适当的休眠状态，一个外部信号将会使得设备(和可能的系统)唤醒。所以它只能唤醒设备，而且要稍迟一些。

当唤醒信号到达时，不论设备是唤醒系统还是唤醒自己,驱动程序的动作是相同的。如果设备有唤醒能力，并且系统正处在设备能唤醒它的休眠状态下时，设备将唤醒此系统。 如果设备有唤醒能力，并且系统正处在工作状态下，则只有设备将被唤醒。

因为计算机和设备在设计方面有所不同，特别是在电源启动方面，支持系统和设备电源状态——能够支持等待/唤醒的状态，在硬件配置方面都是不相同的。因此，拥有其设备电源策略的任何驱动程序和所有的总线驱动程序，必须注意正在运行中的设备的单独配置能力。详见“确定设备是否能唤醒系统”。

对于等待/唤醒操作上的进一步的细节，参见“理解通过设备树的等待/唤醒IRP的路径”。

4.1.1 确定设备是否能唤醒系统

一些设备，诸如键盘，调制解调器以及网卡等，在设备休眠状态下能响应外部信号。作为其电源管理技术的一部分，操作系统为这些设备唤醒一个休眠系统提供了一个方法，即能恢复系统以前的环境。软件唤醒机制依靠系统和设备硬件以及BIOS中的支持,允许系统从除S5(PowerSystemShutdown)之外的任何状态中被唤醒。处于状态S5的系统必须重新启动。

虽然操作系统设计成可以由任何中间的休眠状态被唤醒，但是其精确唤醒能力从计算机到计算机以及从设备到设备都是不同的。并非所有计算机都支持所有系统休眠状态；因此，对一些计算机而言，从某些状态唤醒的能力是毫无意义的。

同样地,大多数设备既不支持所有设备状态(D0到D3),也不支持从这些设备确实支持的所有设备状态而来的唤醒状态。

设备能进入的休眠状态与它支持唤醒的状态一同由总线驱动程序列举描述，并且被存储在DEVICE_CAPABILITY结构中。表4.1列出这种结构中与等待/唤醒支持有关的成员。

表4.1 DEVICE_CAPABILITY结构中唤醒信息

成员	描述
DeviceD1	如果设备支持状态PowerDeviceD1为真
DeviceD2	如果设备支持状态PowerDeviceD2为真
WakeFromD0	如果设备能从PowerDeviceD0中唤醒为真
WakeFromD1	如果设备能从PowerDeviceD1中唤醒为真
WakeFromD2	如果设备能从PowerDeviceD2中唤醒为真
WakeFromD3	如果设备能从PowerDeviceD3中唤醒
DeviceState[PowerSystemMaximum]	指定这种设备能为每一系统状态支持的最大功率的设备状态，PowerSystemUnspecified 到 PowerSystemShutdown

SystemWake　　指定系统能被唤醒的最低的电源状(S0到S4)。
DeviceWake　　指定设备能唤醒的最低的电源状态(D0到D3)。

DeviceWake项目列出设备能响应唤醒信号的最低电源设备状态。PowerDeviceUnspecified的值表示设备不能唤醒系统。SystemWake项列出系统能被唤醒的最低电源状态。由于这些值是基于父设备节点的能力，所以驱动程序不能改变它们。详见第1章中“报告设备的电源能力”。

总之，如果下列条件满足，设备就能唤醒系统：

。处于电源状态下的设备，其功率等于或大于 DeviceWake的值。

。处于电源状态下的系统，其功率等于或大于SystemWake的值

。

4.1.2　理解通过设备树的等待/唤醒 IRP 的路径

在一种独立设备栈内，电源策略所有者发送一个等待/唤醒IRP，同时所有驱动程序都处理这个等待/唤醒IRP，这个过程在“等待/唤醒操作综述”中概要介绍了，其详细内容分别在“发送等待/唤醒IRP”和“处理等待/唤醒IRP的步骤”。

在设备树(叶节点、父节点、祖父节点等等)组成的分支之内，驱动程序必须相互合作，以保证等待/唤醒IRP到达一个能使所有必要的硬件都能够唤醒的驱动程序中。 在ACPI计算机上，ACPI负责系统特定的一般目的事件(GPE)寄存器，其寄存器与每一叶片设备来的唤醒信号有关。因而，驱动程序必须请求并且传递等待/唤醒IRP,直到到达一个ACPI过滤驱动程序(在开始插入在设备栈中)或者位于下面的ACPI驱动程序中。ACPI使寄存器能够控制这个未决的IRP，直到信号到达，然后再完成这个IRP。 因为ACPI能响应唤醒信号，所以它不能把IRP传递到一个较低层驱动程序中。

ACPI过滤驱动程序，和位于下面的ACPI驱动程序本身一样，对其他驱动程序是透明的。为了在硬件中提供最大限度的灵活性，任何设备栈中的ACPI过滤驱动程序的确切位置是设备或系统特有的。在设计驱动程序时，关于设备栈中的ACPI过滤例程的存在或者位置不能做任何假设。

如前所述子驱动程序为每个子设备创建一个PDO，为父设备创建一个FDO。这样，驱动程序为子设备和功能驱动程序以及策略所有者，为父设备充当总线驱动程序。因此，只要总线驱动程序为了子PDO接收一个等待/唤醒的IRP时，它就应该为其父PDO请求另一个等待/唤醒的IRP。

图4.2显示了这样一种状态发生的配置样例状况。

图4.2　USB的配置样例

在这个配置样例中，键盘和调制解调器都是USB集线器的子系统，而USB集线器又是USB主机控制器的子系统，PCI总线列举出USB主机控制器。在下一页的图4.3显示配置样例中的键盘设备栈。

根据图4.3显示，从底部向上读：

1.ACPI驱动程序为PCI创建PDO。

2.PCI驱动程序创建PCI FDO和USB主机控制器PDO，并且拥有PCI设备栈的策略。

3.USB主机控制器驱动程序(主机控制器/总线驱动程序对)创建USB主机控制器FDO和USB集线器PDO。它拥有USB主机控制器设备栈的策略。注意，ACPI驱动程序也在这个栈中创建一个过滤例程DO。

4. USB集线器驱动程序创建USB集线器FDO和键盘PDO。这个驱动程序拥有USB集线器设备栈的电源策略。

5. 键盘的功能驱动程序是USBHID类的驱动程序/微驱动程序对。这个驱动程序为键盘创建FDO并且拥有其电源策略。因为键盘没有子设备，所以驱动程序不创建PDO。

注意,每一设备栈可以包括不被显示的附加的和可选的过滤DOs。为了允许键盘输入唤醒系统，策略所有者为键盘的PDO请求IRP_MN_WAIT_WAKE。这个IRP发送出一连串的其他等待/唤醒IRP。如图4.4所示

图4.4 等待/唤醒IRP请求的USB配置样例

当总线驱动程序接收一个为它所创建的PDO的IRP_MN_WAIT_WAKE目标,那么总线驱动程序必须为它拥有电源策略的设备栈请求另一个IRP_MN_WAIT_WAKE，同时创建一个FDO。

由图4.4所示：

1.键盘驱动程序调用PoRequestPowerIrp，发送等待/唤醒IRP(IRP1)到它的PDO。

电源管理器分配此IRP，并且通过I/O管理器把IRP发送到键盘设备栈的顶端。 驱动程序创建IoCompletion例程，并且在栈中向下传递IRP直到它到达键盘PDO。USB集线器驱动程序，为键盘充当总线驱动程序，并控制未决的IRP1。

2.因为当唤醒信号到达时，USB集线器驱动程序不能唤醒系统，所以USB集线器驱动程序必须调用PoRequestPowerIrp，为USB集线器设备栈请求一个等待/唤醒的IRP(IRP2)。

电源管理器把这IRP发送到USB集线器设备栈的顶端。在此栈中的驱动程序创建IoCompletion例程，并且把IRP传递到USB主机控制器驱动程序中(为USB集线器充当总线驱动程序)。USB主机控制器驱动程序控制未决的IRP2，直到键盘发出一个唤醒事件信号。

3.同样地，USB主机控制器驱动程序不能唤醒系统，因此USB主机控制器驱动程序调用PoRequestPowerIrp，发送一个等待/唤醒IRP(IRP3)到USB主机控制器设备栈。

电源管理器把此IRP发送到USB主机控制器设备栈的顶端，在那里驱动程序创建IoCompletion例程，并且把IRP传递到PCI驱动程序中(为USB集线器充当总线驱动程序)。PCI驱动程序控制未决的IRP3，直到键盘发出一个唤醒事件信号。

4.PCI驱动程序不能唤醒系统，因此PCI驱动程序调用PoRequestPowerIrp，并且发送一个等待/唤醒IRP(IRP4)到PCI设备栈中。其父设备是根设备，ACPI是总线驱动程序。

电源管理器把IRP发送到PCI总线设备栈的顶端;其驱动程序创建完成例程并且把的IRP传递到ACPI驱动程序中。

5.ACPI驱动程序能唤醒系统，因此它不发送等待/唤醒IRP到其他任何PDO。ACPI驱动程序控制未决的IRP4，直到一个唤醒信号到达。

当键盘确定唤醒信号时，ACPI驱动程序立刻阻止它。但是ACPI不能限制键盘确定这个信号，仅能让信号穿过根设备。接着，ACPI驱动程序完成IRP4，同时I/O管理器调用IoCompletion例程，支持PCI设备栈。当IRP4已经完成并且所有IoCompletion例程都已运行时，这个PCI驱动程序的回调例程被调用。在其回调例程中，PCI驱动程序限制信号穿过USB主机控制器。接着，PCI驱动程序完成IRP3。相同的事件通过USB主机控制器栈和USB集线器栈继续发生，直到键盘驱动程序收到IRP1。这时，如有必要，键盘驱动程序可以为唤醒事件服务。

每次当驱动程序发送一个等待/唤醒IRP到父PDO时，它必须为其自己的IRP创建一个Cancel例程。如果触发它的IRP被撤消，则创建Cancel例程，可以使驱动程序有机会撤消新的IRP。在这个USB例子中，如果键盘驱动程序撤消其等待/唤醒IRP(这样就禁止键盘的唤醒能力)，USB集线器，USB主机控制器和PCI驱动程序必须撤消他们所发送的IRP，以此作为键盘IRP的结果。详见“等待/

唤醒的IRP Cancel例程”。

虽然一个父驱动程序可以列举一个以上的能够进行等待/唤醒的子例程，但是对PDO仅有一个等待/唤醒IRP是未决的。在这种情况下，父驱动程序应该随时确保它保持未决的等待/唤醒IRP，以使它们任何设备都能够被唤醒。为此，每次驱动程序收到一个等待/唤醒IRP时，它就会增加一个内部计数器的计数。而每次驱动程序完成一个等待/唤醒IRP时，它就会减少计数器的计数；而且，如果计数器的值是零，则应该发送另一个等待/唤醒IRP到它的设备栈中。

例如，以前在图4.2中所显示的USB配置中，USB集线器列举了两种设备，键盘和调制解调器。当USB集线器驱动程序为键盘PDO接收一个等待/唤醒IRP时，它在为其PDO请求一个IRP之前，增加一个等待/唤醒IRP的计数。如果调制解调器的策略所有者稍迟能够唤醒调制解调器，那么当调制解调器的等待/唤醒IRP到达时，USB集线器设备栈中已经有未决的IRP2。因为它不能同时控制两个未决的等待/唤醒IRP，所以USB集线器驱动程序增加其等待/唤醒参考计数，并且使新的IRP失效。当唤醒信号从键盘或者调制解调器到达时，USB集线器驱动程序首先确定是哪一个设备发信号，然后完成IRP2，并且减少其参考数。因为两种设备都能够被唤醒(由于其参考数不是零)，它必须发送自己的另一个等待/唤醒IRP为唤醒“重新武装”它自己。(对USB主机控制器和PCI驱动程序同样适用。)

然而，一个驱动程序不能把自己的IRP发送到唤醒信号刚刚到达的相同设备上，使其再次等待/唤醒。只有设备电源策略管理模块能这样做。因为再次等待/唤醒不能进行。

4.1.3 完成等待/唤醒 IRP 综述

当一个唤醒信号到达时，一个等待/唤醒IRP就完成了。唤醒信号是设备特定的，但是通常又是设备的一个正常服务事件。例如，一个引入的圆环可能使一台休眠的调制解调器醒来。

图4.5显示完成等待/唤醒IRP的步骤。

当信号发生时，在总线检测设备已经唤醒的位置上控制重新进入总线驱动程序。总线驱动程序为请求事件服务，并且调用IoCompleteRequest，为它的PDO完成 IRP_MN_WAIT_WAKE IRP。

接着，在设备栈中I/O管理器调用相邻较高层驱动程序所创建的IoCompletion例程。在IoCompletion例程中，驱动程序在必要时为唤醒信号服务，并且调用IoCompleteRequest完成IRP。I/O管理器继续调用IoCompletion例程工作以支持设备栈，直到所有驱动程序都完成IRP。

图4.5完成一个等待/唤醒IRP的步骤

在它的IoCompletion例程中,列举多于一种子设备(创建多于一个PDO)的任何设备和从多于一种的这样设备已经接收到等待/唤醒请求的任何设备必须为在另一个子设备上的等待/唤醒发送自己的一个等待/唤醒IRP以“重新武装”自己。详见“理解通过设备树的等待/唤醒IRP的路径”。

在调用由驱动程序创建的IoCompletion例程之后，当它把IRP传递到栈中时，I/O管理器在它请求等待/唤醒IRP时，调用由电源策略所有者所创建的回调例程。在回调例程中，策略所有者应该把其设备返回到工作状态，并且为其子PDO完成一个未决的等待/唤醒的IRP。在子设备栈中完成其子IRP，使得I/O管理器调用由驱动程序所创建的IoCompletion例程，如此等等。最后，在设备节点上，原来的等待/唤醒IRP开始的策略所有者确定其设备被确定的唤醒信号，并且完成所有未决的等待/唤醒IRP。

4.2 处理等待/唤醒 IRP 的步骤

所有PnP驱动程序必须准备接收带有次IRP码IRP_MN_WAIT_WAKE的电源IRP。驱动程序如何处理等待/唤醒IRP，取决于其设备栈中的位置、它所控制的设备类型以及其设备支持唤醒的特定状态。

下列各段提供了处理这个IRP所基于的驱动程序的类型和支持等待/唤醒的水平。

4.2.1 在功能驱动程序或者过滤驱动程序 (FDO)中处理等待/唤醒 IRP

当创建FDO或者过滤DO驱动程序为相关的PDO接收到一个IRP_MN_WAIT_WAKE的请求时，它仅仅把IRP传递到相邻较低层驱动程序中，或者在传递IRP之前采取一定的行动。

对于支持唤醒的设备

在上面接收一个等待/唤醒IRP时，功能驱动程序或者过滤驱动程序应该采取下列的措施：

1. 调用IoAcquireRemoveLock，传递当前IRP，保证在处理等待/唤醒IRP时不接收PnP IRP_MN_RWMOVE_DEVICE请求。

如果IoAcquireRemoveLock返回一个失效状态，那么驱动程序不应该继续处理这个IRP。相反，它应该调用PoStartNextPowerIrp，完成这个IRP(IoCompleteRequest)，并且返回失效状态。

2. 检查Irp->Parameters.WaitWake.SyslemWake中的值，并且在DEVICE_CAPABILITY结构中比较当前设备电源状态与DeviceState[SystemWake]。

如果设备支持唤醒，但是等待/唤醒IRP即不是从指定的SystemWake状态中来，也不是从当前设备电源状态中来，那么驱动程序应该使这个IRP失效，如下：

。在Irp->IoStatus.Status中设置STATUS_INVALID_DEVICE_STATE。

。调用PoStartNextPowerIrp。

。完成IRP(IoCompleteRequest)，指定IO_NO_INCREMENT的优先级提升。

。在Irp->IoStatus.Status中，从DispatchPower例程返回状态设置。

3. 否则，为IRP创建IoCompletion例程，使用IoSetCompletionRoutine。

IoCompletion例程应该执行驱动程序请求的任务，把设备返回到工作状态。

如果要撤消IRP，则也要调用IoCompletion例程。

4. 在其IoCompletion例程中，保存驱动程序可能需要的任何信息。

5.调用PoStartNextPowerIrp，开始下一个电源IRP。

6.调用PoCallDriver，把等待/唤醒IRP传递到相邻较低层驱动程序。

7.调用IoReleaseRemoveLock，释放以前获取的锁。

8.从DispatchPower例程中返回STATUS_PENDING。当驱动程序控制IRP时，驱动程序不能在Irp->IoStatus.Status中改变值。

对于不支持唤醒的设备

对不支持唤醒的设备,如果功能驱动程序或者过滤驱动程序接收一个等待/唤醒的IRP时,则驱动程序应该使这个IRP失效，其过程如下：

1. 调用PoStartNextPowerIrp。

2. 完成IRP(IoCompleteRequest)，指定IO_NO_INCREMENT的优先级提升。
3. 在Irp->IoStatus.Ststus中，从DispatchPower例程返回状态设置。

4.2.2 在总线驱动程序(PDO)中处理等待/唤醒 IRP

像其他电源IRP一样,每一个等待/唤醒IRP必须通过设备栈向下传递到总线驱动程序(PDO)中，最后PDO负责完成IRP。接收上面的IRP时，总线驱动程序能立即使IRP失效或者处理未决的IRP，稍后完成它。下面是总线驱动程序必须采取的步骤：

1. 检查Irp->Parameters.WaitWake.SystemWake中的值。如果设备支持唤醒，而IRP即不是来自指定的系统唤醒状态,也不是来自当前设备电源状态，则驱动程序应该使这个IRP失效，如下：
。在Irp->IoStatus.Status中设置STATUS_INVALID_DEVICE_STATE。
。调用PoStartNextPowerIrp。
。完成IRP(IoCompleteRequest)，指定IO_NO_INCREMENT的优先级提升。
。在Irp->IoStatus.Status中从DispatchPower例程返回状态设置。
2. 检查一个等待/唤醒IRP对PDO是否是未决的。如果是，给STATUS_DEVICE_BUSY设置Irp->IoStatus.status，增加驱动程序内部的等待/唤醒IRP的计数，并且按以前步骤所描述的方法完成IRP。

对一个PDO，只有一个等待/唤醒IRP可以是未决的。

3. 如果驱动程序支持指定系统状态的唤醒，并且没有等待/唤醒IRP已经成为未决的，调用IoMarkIrpPending以指明对I/O管理器，IRP将被完成或者稍晚被撤消。不创建IoCompletion例程。
4. 设置激活设备硬件唤醒能力。

总线驱动程序激活其硬件唤醒能力，这种机制是属于设备依赖的。对于一个PCI设备，pci.sys负责设置PME位，因为这个驱动程序拥有PME寄存器。对于其他设备，参阅设备类别特定文档。

5. 如果PDO是FDO的子设备，则为FDO请求一个等待/唤醒IRP，保证为当前IRP创建一个Cancel例程(它所控制的未决IRP)。不要试图传递或者再使用当前IRP。
6. 从DispatchPower例程中返回STATUS_PENDING。
7. 当一个唤醒信号到达时，调用PoStartNextPowerIrp。然后调用IoCompleteRequest完成未决的等待/唤醒IRP，把Irp->IoStatus.Status设置到STATUS_SUCCESS，并且指定IO_NO_INCREMENT的优先级提升。

对于不支持唤醒的设备

如果设备不支持唤醒，则总线驱动程序(PDO)将进行如下操作：

1. 开始下一个电源IRP(PoStartNextPowerIrp)。
2. 通过调用IoCompleteRequest完成等待/唤醒IRP，指定IO_NO_INCREMENT。
3. 从DispatchPower例程中返回，传递Irp->IoStatus.Status中的值作为其返回值。

4.2.3 等待/唤醒 IRPIoCompletion 例程

在设备栈中的相邻较低层驱动程序已经完成等待/唤醒IRP之后,I/O管理器调用驱动程序的等

待/唤醒IoCompletino例程。处理等待/唤醒IRP的每个功能驱动程序和过滤(FDO)驱动程序应该为此IRP创建一个IoCompletion例程。

每个功能驱动程序或者过滤驱动程序在处理进入设备栈的等待/唤醒IRP时，创建一个IoCompletion例程。因此发送IRP的设备电源策略所有者除了回调例程之外还有一个IoCompletion例程。要记住，回调例程要在IoCompletion例程之后调用，而且这两个例程有不同的要求。详见“等待/唤醒回调例程”。

在一个等待/唤醒IoCompletion例程中所要求的动作依赖于设备和驱动程序的类型。下面驱动程序在其等待/唤醒IoCompletion例程中可能需要执行的一些动作：

1. 在设备扩展中重置相关域。例如，当一个等待/唤醒IRP是未决的时，大多数驱动程序设置一个标记并且保持指针指向设备扩展中的IRP。
2. 重置IoCancle例程，对于IRP，通过调用IoSetCancelPoutine，为例程指定一个NULL指针。
3. 调用PoStartNextPowerIrp，开始下一个电源IRP。
4. 调用IoCompleteRequest，指定IO_NO_INCREMENT，以完成IRP。

当每一个驱动程序连续的完成IRP时，I/O管理器把控制传递到相邻较高层驱动程序的IoCompletion例程中，支持设备栈。

当驱动程序传递等待/唤醒IRP到设备栈时，调用由驱动程序所创建的IoCompletion例程，在这之后，I/O管理器调用回调例程，并由发送IRP的驱动程序传递IRP到PoRequestPowerIrp。详见“等待/唤醒回调例程”。

4.3 发送等待/唤醒 IRP

次电源IRP码IRP_MN_WAIT_WAKE为唤醒设备或者唤醒系统提供必要条件。能唤醒自己或者唤醒系统的设备驱动程序发送IRP_MN_WAIT_WAKE请求。系统只对总是唤醒系统的设备发送IRP_MN_WAIT_WAKE请求，诸如加电开关。

因为下面两个原因之一，驱动程序发送IRP_MN_WAIT_WAKE请求：

1.其设备必须能够从休眠状态返回到工作状态，以响应外部唤醒信号。

例如，为保存能量，在设置一个IRP进入电源中状态D1之前，调制解调器的驱动程序发送一个等待/唤醒IRP。这个等待/唤醒IRP使得调制解调器响应一个到达的调用。

2.其设备在响应一个唤醒信号时必须能够唤醒系统。

当系统将要休眠时，带有未决的IRP_MN_WAIT_WAKE的调制解调器可能仍然在状态D1中。在这种情况下，一个到达的调用将唤醒系统及调制解调器。

无论设备是准备唤醒自己还是唤醒系统，其驱动程序必须采取相同的动作。两者的主要区别在于设备和系统硬件如何响应初始唤醒信号。不论哪一种情况，驱动程序的动作是相同的。

4.3.1 确定何时发送等待/唤醒 IRP

拥有设备电源策略的驱动程序发送等待/唤醒IRP代表其设备。当下列情况之一发生时，这个驱动程序必须发送一个等待/唤醒IRP：

。驱动程序正在使设备进入休眠，但是当响应一个外部唤醒信号时，设备必须能够被唤醒。
。系统将进入休眠，并且设备必须能够唤醒它。
。驱动程序接收一个带有次代码IRP_MN_WMI的IRP_MJ_SYSTEM_CONTPOL请求，而此次代码设置DEVICE_WAKE_ENABLE为TRUE。为获得更多关于WMI的信息，参见在线DDK中“内核模式驱动程序设计指南和参考”。

在这样的条件出现之前，电源策略所有者应该发送等待/唤醒IRP。当它的设备在状态D0的任何时间时，它都能发送这个IRP。但当它处理另一个设置电源IRP或者查询电源IRP时，它不能发送这样的IRP。通常，在驱动程序已经初始化且设备已开始之后，驱动程序在PNP管理模块的IRP_MN_START_DEVICE请求的期间，应该发送这个IRP。

4.3.2 等待/唤醒 IRP 请求

为了发送一个IRP_MN_WAIT_WAKE，驱动程序调用PoRequestPowerIrp，传递(在其他参数中间)一个指针到目标PDO中，传递一个系统电源状态和一个指针到回调例程中。

系统电源状态指定能唤醒系统IRP的最小功率。其值必须是等于或者大于在DEVICE_CAPABILITIES结构中SystemWake状态的功率。例如，如果驱动程序传递在IRP中的PowerSystemSleeping2，相关的IRP可以使系统从状态S0，S1和S2中唤醒。在这样一种情况下，系统必须支持S0和S2(在最高功率和最低功率范围中),但不需要支持S1。

请求一个等待/唤醒IRP的驱动程序应该指定一个回调例程，而此回调例程在所有其他驱动程序完成IRP之后被调用。在这个例程中，驱动程序能使其设备返回到工作状态。

在响应PoRequestPowerIrp时,电源管理器分配一个带有次代码IRP_MN_WAIT_WAKE的电源IRP，并且把它发送到目标PDO的设备栈的顶端。调用者返回一个指向分配的IRP的指针，如果不得不撤消该IRP，则可延迟使用。

如果没有错误发生，PoRequestPowerIrp返回STATUS_PENDING。这个状态意味IRP成功地被发送并且是未决的完成。

一个等待/唤醒IRP不能改变系统的电源状态或者设备的电源状态。它仅仅能激活一个设备的唤醒信号。此IRP仍然保持未决状态，直到一个外部信号使系统或者设备唤醒。

4.3.3 等待/唤醒的回调例程

当一个驱动程序请求等待/唤醒IRP时，它必须指定一个回调例程，当唤醒事件发生时它就能使设备返回到工作的状态(D0)。在唤醒事件发生之后，并且所有驱动程序已经完成IRP时，系统调用回调例程传递到PoRequestPowerIrp。

因为这个回调例程被设置为代表开始IRP的驱动程序，而不是对正在处理的IRP，所以它不能调用PoStartNextPowerIrp：只有当驱动程序把IRP传递到栈中时，IoCompletion例程设置才应该开始下一个电源IRP。记住，策略所有者不仅发送这个IRP而且还处理它。因此，当它把IRP传递到设备栈中时，它还可以创建一个IoCompletion例程；除此之外，当它请求等待/唤醒IRP时，又创建了一个回调例程。

回调例程具有下列职责：

1.如果驱动程序控制多于一个的设备，确定哪一个设备发出唤醒信号。

2.引起唤醒信号事件的服务。

3.设置发出唤醒信号的驱动程序，在D0状态通过调用PoRequestPowerIrp发送一个PowerDeviceD0请求。驱动程序也必须调用PoSetPowerState以通知新设备状态的电源管理器。详见“为设备电源状态发送IRP_MN_QUETY_POWER或者IRP_MN_SET_POWER”。

4.如果驱动程序为IRP创建一个Cancel例程，调用IoSetCancelRoutine将Cancel例程重置为NULL。

5.如果驱动程序为多于一种的设备拥有电源策略，则减少它的等待/唤醒参考数。 如果该数非零，表明以前另一个设备曾经发送了一个等待/唤醒IRP，那么为它的PDO请求另一个等待/唤醒IRP(PoRequestPowerIrp)。

例如,PCI设备可以使调制解调器与网络接口卡(NIC)两个设备都有等待/唤醒的能力。如果NIC唤醒系统(这样完成IRP)，那么PCI FDO必须发送另一个等待/唤醒IRP到它自己中，以便调制解调器仍然能够被唤醒。

因为请求等待/唤醒IRP的驱动程序为它的设备栈控制电源策略，所以当IRP完成时，它负责将其设备放进工作状态。虽然较低层驱动程序实际上已经把电源用到设备上，但是策略所有者必须为设备状态D0调用PoRequestPowerIrp，以发送一个IRP_MN_SET_POWER IRP。只有设备栈中所有驱动程序都已处理这个加电IRP，设备才返回到工作状态。

4.3.4 撤消等待/唤醒 IRP

只有发送等待/唤醒IRP的驱动程序能撤消那个IRP。

在下面的环境下，驱动程序可以根据需要撤消一个未决的等待/唤醒的IRP：

。驱动程序为设备接收一个PnP IRP_MN_STOP_DEVICE，IRP_MN_QUERY_REMOVE_DEVICE，IRP_MN_REMOVE_DEVICE，或者IRP_MN_SURPRISE_REMOVAL请求。如果驱动程序计算它自己未决的等待/唤醒IRP作为未完成I/O，那么当它为设备接收一个IRP_MN_QUERY_REMOVE_DEVICE请求时，它就应该撤消此IRP。在设备被重新启动之后，驱动程序应该再发送等待/唤醒IRP(PoRequestPowerIrp)。

。系统将进入休眠，而设备不能唤醒系统。例如，USB集线器驱动程序在设备启动时，可以发送一个IRP_MN_WAIT_WAKE请求，以防它稍晚将其输入设备中的一个设备放进休眠状态。当系统在工作状态下时，从设备而来的唤醒信号把设备返回到工作状态(但是对系统电源状态没有作用)。如果设备将不允许唤醒系统，那么当系统准备关机时，USB集线器驱动程序撤消这IRP。

。系统正在进入一个设备不能唤醒它的休眠状态，即它正在进入一个比在其DEVICE_CAPABILITIES结构所指定的SystemWake值的功率更小的状态。

。设备正在进入一个它不能响应唤醒信号的电源状态，即它正在进入一个比其DEVICE_CAPABILITIES结构所指定的DeviceWake值的功率更小的状态。

。驱动程序接收一个带有次代码IRP_MN_WMI的IRP_MJ_SYSTEM_CONTROL请求，此次代码将DEVICE_WAKE_ENABLE设置为FALSE。详见在线DDK中“内核模式驱动程序设计指南和参考”。

为了撤消等待/唤醒IRP，发送IRP的驱动程序调用IoCancelIrp，把指针传递到以前驱动程序调用PoRequestPowerIrp时返回的IRP中。驱动程序不可以撤消一个它没有发送的等待/唤醒IRP。

4.4 等待/唤醒 IRP 的 Cancel 例程

许多功能驱动程序和总线驱动程序应该为未决的等待/唤醒IRP创建Cancel例程；下面是必须创建Cancel例程的驱动程序类型：

。改变设备设置为激活或者禁止唤醒能力的驱动程序

。发送IRP_MN_WAIT_WAKE请求到父设备驱动程序的驱动程序。

Cancel例程允许驱动程序为其设备禁止唤醒能力，并且清除有关未决的等待/唤醒IRP的任何数据。为父设备请求等待/唤醒IRP的驱动程序也能撤消那些IRP。

在其等待/唤醒Cancel例程中，驱动程序采取下列措施：

1.调用IoSetCancelRoutine，重置IRP的Cancel例程为NULL。

2.调用IoReleaseCancelSpinLock，传递在IRP中指定的CancelIRQL，以释放IRP的撤消自旋锁。

3. 在设备扩展中重置相关域。例如，当一个等待/唤醒IRP是未决的时，大多数驱动程序设置标记，并且在设备扩展中保持IRP的指针。注意，驱动程序可能在收到一个等待/唤醒IRP的同时，也正在撤消另一个这样的IRP。驱动程序必须检查在自旋锁(或者其相似物)的保护下，它是否已经有一个IRP。如果已经有，驱动程序必须仔细地同步它的处理，以保证它撤消当前的IRP。为得到更多关于自旋锁在Cancel例程中的使用的内容，参见第12章“内核模式驱动程序设计指南”中的“Cancel例程”。

4.改变任何需要的设备设置。例如，调制解调器驱动程序可以丧失设备的唤醒设置能力。

5.把Irp->IoStatus.Status设置到STATUS_CANCELED。

6.调用PoStartNextPowerIrp，开始下一个电源IRP。

7.调用IoCompleteRequest，以完成等待/唤醒IRP，指定IO_NO_INCREMENT。

8.如果驱动程序以前为父设备请求相关的IRP_MN_WAIT_WAKE，那么驱动程序就应该从其Cancel例程中撤消那个IRP。驱动程序必须在它撤消父设备的IRP之前，释放这个撤消自旋锁。

例如，为设备充当总线驱动程序并且为父设备拥有电源策略驱动程序的驱动程序，应该撤消相关的等待/唤醒IRP,而此IRP是稍早发送到其父设备的。调用的IoCancelIrp请求父设备的Cancel例程，进入设备栈，如此等等。详见第12章“内核模式驱动程序设计指南”中的“Cancel例程”。

第四部分　设置

第1章 设备安装总览

在安装了微软 Windows 2000 的计算机上，Setup 和其他系统提供的组件以及厂商提供的组件一起工作来安装设备。当计算机启动或者启动后的任何时候用户增加了一个即插即用（PnP）设备时，Setup 安装设备（或者手工安装一个非即插即用设备）。

为了支持 PnP，Setup 基于机器中的设备来进行安装过程而不是按照驱动程序的顺序来安装。例如，不是装载一批驱动程序，然后让这些驱动程序检测它们所支持的设备。而是由 Setup 确定机器中提供了什么设备，再装载和调用相应的驱动程序。驱动程序，比如 ACPI 驱动程序和其他 PnP 总线驱动程序帮助 Setup 确定机器中有什么样的设备。

这一章包含了以下的信息：

1.1 设备安装组件

图 1.1 显示了在 Windows 2000 机器上安装设备所涉及的软件组件。

图 1.1 中的阴影矩形代表由 IHV 和 OEM 提供的组件，其他设备安装组件由微软提供。

图 1.1 设备安装设置组件

以下描述了设备安装中各种组件所扮演的角色：

内核模式的 PnP 管理器

内核模式的 PnP 管理器通知用户模式的 PnP 管理器在机器中有一个新设备存在并需要被安装。内核模式的 PnP 管理器也调用设备驱动程序的 DriverEntry 和 AddDevice 例程并发送 IRP_MN_START_DEVICE 请求以启动设备。

PnP 管理器有两部分：一部分运行在用户模式，一部分运行在内核模式。内核模式的 PnP 管理器向用户模式的 PnP 管理器报告 PnP 事件，用户模式的 PnP 管理器向内核模式的 PnP 管理器发送控制请求。

驱动程序

PnP 驱动程序执行由 PnP 管理器指示的设备安装操作。例如，当 PnP 管理器为 BusRelations 发出 IRP_MN_QUERY_DEVICE_RELATIONS 请求时，PnP 总线驱动程序返回它的子设备的列表。当被 PnP 管理器调用时，PnP 驱动程序执行它们的 DriverEntry 和 AddDevice 例程。PnP 驱动程序也处理由 PnP 管理器发送的其他请求，比如 IRP_MN_QUERY_CAPABILITIES，IRP_MN_START_DEVICE 和 IRP_MN_REMOVE_DEVICE。

PnP 驱动程序还能检测非即插即用设备，并使用 IoReportDetectedDevice 向 PnP 管理器报告。

用户模式的 PnP 管理器

用户模式的 PnP 管理器从内核模式的 PnP 管理器接受设备安装请求，调用其他的用户模式 Setup 组件以开始设备安装任务，并向内核模式的 PnP 管理器发出控制请求（比如“启动设备”）。

用户模式的 PnP 管理器和内核模式的 PnP 管理器一起维护设备树（device tree）。

用户模式的 PnP 管理器尽力在一个可信任的进程上下文中安装一个设备，而不要求用户对对话框做出响应（“服务器端”的安装）。这样的自动安装提供了更好的用户接口。如果 PnP 管理器不能完成可信任的安装，比如因为安装程序提供了一个定制的“完成”页面，则 PnP 管理器放弃可信任的安装。在这种情况下，当用户以系统管理员权限登录，PnP 管理器启动 New Device DLL 中的发现新硬件向导重新开始设备安装（“客户端”安装）。

Setup API

Setup API 包括 SetupXxx 函数和 SetupDiXxx 函数。这些函数执行许多设备安装任务，如搜索 INF 文件，为设备构造潜在的驱动程序列表，拷贝驱动程序文件，向注册表中写信息，注册设备的协同安装程序等等。大多数 Setup 组件调用这些函数以完成工作。

SetupDiXxx 函数有时也被称为设备安装程序。但是这个术语是含混的，因为不只一个组件涉及到设备的安装。

更多的信息参见《Windows 2000 驱动程序开发参考》第 1 卷中的第 2 章和 3 章。

CfgMgr API

配置管理 API 提供了基本的安装和配置操作，它们在 Setup API 中没有被提供。配置管理函数执行底层任务，比如取得设备节点的状态以及管理资源的描述符。这些函数主要被 Setup API 调用，但也能被其他 Setup 组件调用。

协同安装程序和类安装程序

类安装程序在一个特定的设备设置类别中执行适用于设备的安装操作。例如，端口类安装程序负责分配一个 COM 端口名给端口设置类别中的一个设备。如果在特别的设置类别中的设备不要求任何特殊的安装操作，对这个设置类就不需要类安装程序。

微软为大多数系统定义的设备设置类别提供了类安装程序。更多的信息，参见第 5 章---编写类安装程序。IHV 和 OEM 可以提供类安装程序，但是它们通常提供协同安装程序。

协同安装程序执行对一个特别的设备或者一个设置类别设备的安装操作。更多的信息参见第 4 章---编写协同安装程序。

INF 文件和目录文件

INF 文件和目录文件提供了关于要安装的设备和驱动程序的信息。更多的信息参见第 3 章---建立 INF 文件。

设备管理器

设备管理器允许用户察看和管理机器中的设备。例如，用户可以察看设备状态并设置设备属性。如果用户请求更新驱动程序，设备管理器调用 New Device DLL 中的更新驱动程序向导。

更多的信息参见设备管理器的联机帮助。

增加/删除硬件向导

增加/删除硬件向导允许用户增加、删除、拔掉和修理设备。在 systray 中的 Unplug 或 Eject 硬件小程序是这个向导的“拔掉”任务的快捷方式。

New Device DLL

New Device DLL 包含发现新硬件向导、更新驱动程序向导和 UpdateDriverForPlugAndPlayDevices 函数。

用户模式的 PnP 管理器在有适当权限的用户上下文中调用发现新硬件向导，以开始客户端的新设备的安装。当用户选择设备的驱动程序属性页上的“更新驱动程序...”按钮时，设备管理器调用更新驱动程序向导。发现新硬件向导、更新驱动程序向导调用 Setup API 和配置管理器 API 完成它们的任务。

定制 Setup 应用程序

IHV 和 OEM 可以提供定制的设置应用程序。更多的信息参见第 6 章---编写定制的设备安装应用程序。

1.2 PnP 设备安装示例

为了说明在设备安装涉及的组件间的交互,这一节分步骤的描述了热插拔 PnP 设备的安装(使用随书的 CD 中的支持文件)。

图 1.2 显示了参加 PnP 设备安装的软件组件。有阴影的组件是设备厂商提供的。

图 1.2 PnP 设备安装示例

以下的注释对应于图 1.2 中标出的带圈的数字:

1. 用户将设备插入机器。

   如果设备和总线支持热插拔通知,用户可以在机器运行时插入新设备。

2. 设备被枚举。

总线驱动程序在总线支持下,接受新设备的热插拔通知。总线驱动程序通过调用 IoInvalidateDeviceRelations 通知内核模式的 PnP 管理器,总线上的设备已经改变。这种情况下,改变是总线上的新设备。

内核模式的 PnP 管理器通过为 BusRelations 发送 IRP_MJ_PNP/IRP_MN_QUERY_DEVICE_RELATIONS 向总线驱动程序来查询总线上存在的设备。总线驱动程序以当前总线上的设备列表响应此 IRP。

内核模式的 PnP 管理器比较新列表和以前的列表,以确定总线上有一个新的设备。

内核模式的 PnP 管理器向总线驱动程序发出 IRP 以收集关于新设备的信息,比如设备的硬件 ID、兼容 ID 和设备性能。这些 IRP 包括 IRP_MN_QUERY_ID 和 IRP_MN_QUERY_CAPABILITIES。参见第 2 部分的第 1 章---动态添加新的 PnP 设备以获得更多有关内核模式 PnP 管理器的枚举活动的信息。

3. 内核模式 PnP 管理器通知用户模式 PnP 管理器,有一个设备要被安装。

用户模式 PnP 管理器尽力执行可信任的安装(服务器端安装),但是在这个例子中不能进行,因为它需要显示一个用户界面,询问用户设备安装磁盘的信息。

4. 用户模式 PnP 管理器使用 rundll32.exe 建立一个新进程,并启动 newdev.dll 安装新设备。

5. New Device DLL 调用 Setup API 和 CfgMgr API 函数以执行它的安装任务。

New Device DLL 调用 SetupDiBuildDriverInfoList,为设备建立可能的驱动程序列表。在本例子中,机器中的 INF 没有和新设备相匹配的,因此返回的列表是空的。New Device DLL 显示发现新设备向导。用户提供驱动程序的位置(软盘、CD、Windows Update)。在本例子中,驱动文件在 CD 上,所以用户装入 CD 并点击向导中“下一步”按钮。New Device DLL 再次调用 SetupDiBuildDriverInfoList,从 CD 接收到包含驱动程序的列表。

参见下一节获得更多有关驱动程序选择的信息。

6. 类安装程序和协同安装程序通过处理 DIF 请求,可以参加到安装过程中。

例如,New Device DLL 调用 SetupDiCallClassInstaller 以发送 DIF_SELECTBESTCOMPATDRV 安装请求。SetupDiCallClassInstaller 对任何为此设备注册的类安装程序和协同安装程序发送 DIF 请求。

New Device DLL 发送一系列的 DIF 编码，包括 DIF_SELECTBESTCOMPATDRV，DIF_ALLOW_INSTALL，DIF_INSTALLDEVICEFILES ， DIF_REGISTER_COINSTALLERS ， DIF_INSTALLINTERFACES 和 DIF_INSTALLDEVICE。类安装程序和协同安装程序可以参加每一个操作。

Setup 使用被选择驱动程序的 INF 文件的 Version section 中的 Class 和 ClassGUID 条目以确定设备的设置类别。设置类别确定了设备的类安装程序和协同安装程序，设备专用的协同安装程序在 INF DDInstall.Coinstaller section 中列出。更多的关于安装程序的信息，参见第 4 章---编写协同安装程序和第 5 章---编写类安装程序。

7. Setup 将控制转移给内核模式以装载驱动程序并启动设备。

一旦 Setup 为设备选择了最佳的驱动程序，拷贝了适合的驱动程序文件，注册了任何设备专用协同安装程序，注册了任何设备接口等等，它将控制转移给内核模式以装载驱动程序并启动设备。适当的 CfgMgr 函数对用户模式的 PnP 管理器发送请求，用户模式的 PnP 管理器再将它传给内核模式的 PnP 管理器。

8. PnP 管理器为设备装入适宜的功能驱动程序和任何可选的过滤器驱动程序。

PnP 管理器为任何已请求但还没有被装入的驱动程序调用 DriverEntry。PnP 管理器然后为每个驱动程序调用 AddDevice，开始于下层的过滤程序，然后是功能驱动程序，最后是顶层的过滤程序。PnP 管理器分配资源给设备，如果需要，它发送 IRP_MN_START_DEVICE 给设备的驱动程序。

9. 安装程序可以提供向导页以改变设备设置。

Setup 在它显示标准完成页之前发送 DIF_NEWDEVICEWIZARD_FINISHINSTALL 请求。如果必要的话，对安装程序这是一个机会以增加定制页以改变设备的设置。

1.3 Setup 如何为设备选择驱动程序?

Windows 2000 Setup 为每个设备选择驱动程序，除非设备正在被以原始方式使用。这一节描述了 Setup 如何做出这个选择。Setup 对设备的概念实际上是“驱动程序节点”，它包括了对一个设备的所有支持，如任何服务、设备特定的协同安装程序、注册表项等等。设备的服务可以包括一套 PnP 驱动程序（一个功能驱动程序和任何顶层、下层过滤器驱动程序）。

一些设备要求厂商提供为那个设备专门设计的驱动程序。另一些设备能够被厂商提供的、设计用来支持一个设备家族的驱动程序所驱动。还有一些设备能被系统提供的、能支持给定设备设置类别的所有设备的驱动程序所驱动。为了为设备选择驱动程序，Setup 优先选择和设备特别匹配的驱动程序。如果不能找到这样的驱动程序，它从后继的更一般的驱动程序中选择。

为了确定驱动程序匹配，Setup 比较设备的硬件 ID、兼容性 ID 和由设备的父总线驱动程序报告的机器中 INF 文件所列出的硬件 ID、兼容性 ID。（如果 Setup 在任何已安装的 INF 文件中不能找到匹配项，它就会显示发现新硬件向导并询问用户设备的驱动程序。）

Setup 为设备建立可能的驱动程序列表---即那些在 Models 项包含的 ID 和设备的 ID 之一相匹配的驱动程序。Setup 为每个可能的驱动程序分配优先等级。等级指示了驱动程序对设备的匹配程度。等级号越低，就越匹配。0 等级是最佳匹配，0xFFFF 等级是最差匹配。Setup 指定的等级如下：

驱动程序等级	描述
0-FFF	在 INF 的 Models 项的硬件 ID 和设备的硬件 ID 匹配。等级 N 匹配意味着设备的 N+1 硬件 ID 和 INF 硬件 ID 匹配。

1000-1FFF	在 INF 的 Models 项的兼容性 ID 和设备的硬件 ID 之一匹配。
2000-2FFF	在 INF 的 Models 项的硬件 ID 和设备的兼容性 ID 之一匹配。
3000-3FFF	在 INF 的 Models 项的兼容性 ID 和设备的兼容性 ID 之一匹配。
FFFF	最差的匹配。这是个特殊的值，被如协同安装程序的组件使用。

等级在 0-FFF 范围内的驱动程序被称为硬件 ID 匹配，因为它们是硬件 ID 和硬件 ID 的匹配。这是最好的匹配。所有其他等级被称为兼容性 ID 匹配，因为它们在匹配中至少涉及一项兼容性 ID 匹配。当 Setup 在一个管理员用户进程的上下文中安装设备（客户端安装），如果驱动程序等级仅仅是兼容性 ID 匹配,则 Setup 显示发现新硬件向导以询问用户是否他/她有更好的驱动程序。

例如，考虑一个有两个硬件 ID 和两个兼容性 ID 的设备。ID 的语法在术语集中定义；对于这个讨论，用以下的缩写来代表 ID：

HwID_1，HwID_2，CID_1，CID_2

在硬件 ID 列表的第一个硬件 ID 对设备是最精确的标识。在本例子中就是 HwID_1。

在本例子中，有一个 INF 文件，它的 Models 节有类似下面的表项：

DeviceDesc1 = InstallSection1，INF_HWID_1，INF_CID_1

INF_XXX_N 是对在 INF Models 节列出的硬件和兼容性 ID 的缩写。

下表列出了当比较示例设备的 ID 和 INF 文件的 ID 时，Setup 可以设置的可能的匹配等级：

设备的 ID	**INF 文件 Models 节的 ID**	
	INF_HWID_1	**INF_CID_1**
HwID_1	0 级	1000-1FFF 级
HwID_2	1 级	1000-1FFF 级
CID_1	2000-2FFF 级	3000-3FFF 级
CID_2	2000-2FFF 级	3000-3FFF 级

在这个例子中，Setup 开始取出设备的第一个硬件 ID，HwID_1，并把它和机器中的 INF 中的 ID 比较。如果 Setup 匹配了 HwID_1 和 INF 文件 Models 节列出的第一个硬件 ID，它增加此驱动程序节点到它的潜在驱动程序列表中，并将此匹配等级置为 0 级。如果 Setup 匹配了任何其他 INF 文件 Models 节的 ID 和 HwID_1,它给这些匹配分配等级并将它们增加到它的潜在驱动程序列表中。

接着，Setup 取出设备附加的硬件 ID，并和 INF 中的 ID 比较。如果它找到任何匹配，它给这些匹配分配等级并将它们增加到它的设备的潜在驱动程序列表中。然后它为设备的兼容性 ID 进行同样的过程（如果有兼容性 ID）。

在 Setup 构造了它的完整的潜在驱动程序列表后，它为设备选择最佳的驱动程序（最小等级的驱动程序）。如果不只一个驱动程序有相同的最小等级，Setup 选择最近日期的驱动程序。驱动程序的日期由驱动程序 INF 文件的 DriverVer 项说明。如果驱动程序包没有数字签名，Setup 忽略任何 DriverVer 项并使用默认的日期 00/00/0000。如果两个驱动程序有相同的等级和相同的日期，Setup 选择任何一个。

通过处理 DIF_SELECTBESTCOMPATDRV 请求（或者对于手工安装的设备是 DIF_SELECTDEVICE 请求），类安装程序和协同安装程序能参与驱动程序的选择。在 Setup 构造了它的潜在驱动程序列表后但是在它选择最佳的驱动程序前，Setup 发送这个 DIF 请求。当处理这些 DIF 编码时，类安装程序和协同安装程序对潜在的驱动程序设置 DNF_BAD_DRIVER 标记,如果安装程序能够判断驱动程序不能支持设备。如果潜在的驱动程序节点 DNF_BAD_DRIVER 标记被设置，Setup 做选择时会忽略此驱动程序。

当安装程序处理 DIF_SELECTBESTCOMPATDRV 请求（或者 DIF_SELECTDEVICE）时，安装程序通过改变潜在驱动程序的等级也可以参与驱动程序的选择。但是，安装程序改变驱动程序的等级的

唯一方式是设置等级为 0xFFFF。这个等级值指示驱动程序可以为设备使用，但是仅作为最后的选择。安装程序不应该把设备等级设为中间值，因为它们会干扰适当的驱动程序选择，也因为等级范围容易改变。

不是更改驱动程序的选择过程，类安装程序和协同安装程序可以重载 Setup 的条目选择过程并直接为设备选择驱动程序。但是，这并不被推荐，因为 Setup 的选择算法在将来版本中可能会被加强,它自己选择驱动程序也会引起安装程序阻止Setup的选择操作的进行。参见《Windows 2000 驱动程序开发参考》卷 1 中的 DIF_SELECTBESTCOMPATDRV 和 DIF_SELECTDEVICE 获得更多信息。

为了说明驱动程序选择过程，考虑一个示例的视频设备，它匹配 INF 文件的三个驱动程序节点。设备的父总线驱动程序（pci.sys）报告以下的设备 ID：

设备的硬件 ID：

PCI\VEN_FFFF&DEV_493D&SUBSYS_001C105D&REV_00

PCI\VEN_FFFF&DEV_493D&SUBSYS_001C105D

PCI\VEN_FFFF&DEV_493D&CC_030000

PCI\VEN_FFFF&DEV_493D&CC_0300

设备的兼容性 ID：

PCI\VEN_FFFF&DEV_493D&REV_00

PCI\VEN_FFFF&DEV_493D

PCI\VEN_FFFF&CC_030000

PCI\VEN_FFFF&CC_0300

PCI\VEN_FFFF

PCI\CC_030000

PCI\CC_0300

在这个例子中，Setup 在 INF 文件中找到三个驱动程序节点可以匹配设备的 ID。第一个匹配的驱动程序的节点如下：

%Sample% = Sample.DDInstall，PCI\VEN_FFFF&DEV_493D&CC_0300

设备的第四个硬件 ID 和上面的 INF 项的硬件 ID 匹配,所以这是个等级为 3 的驱动程序匹配。Setup 增加这个驱动程序到它的潜在驱动程序列表中。这个驱动程序支持这种类型的任何视频设备（有任何子系统厂商 ID 和任何硬件版本号的视频设备）。

Setup 为设备匹配的第二个 INF ID 在它的 Models 表项中有以下的驱动程序节点：

%Sample2% = Sample2.DDInstall，PCI\VEN_FFFF&DEV_493D& SUBSYS_001C105D

设备的第二个硬件 ID 和上面的 INF 项的硬件 ID 匹配,所以这是个等级为 1 的驱动程序匹配。Setup 增加这个驱动程序到它的潜在驱动程序列表中。这个驱动程序支持有这个特定子系统厂商 ID 及任何硬件版本号的设备。

Setup 为设备匹配的第三个 INF ID 在它的 Models 表项中有以下的驱动程序节点：

%Sample3% = vga，PCI\CC_0300

设备的最后一个兼容性 ID 和上面的 INF 项的硬件 ID 匹配，所以这个驱动程序匹配的等级在 2000-2FFF 范围内。Setup 增加这个驱动程序到它的潜在驱动程序列表中。这个驱动程序是一个支持任何 PCI 视频设备的通用驱动程序（VGA 驱动程序）。

对于这个例子，如果类安装程序和协同安装程序不更改 Setup 给驱动程序匹配指定的等级，Setup 选择等级为 1 的驱动程序，%Sample2%。

1.4 系统设置阶段

对于大多数设备，你应该设计设备和支持文件以便 Setup 在系统启动的适当时间或者启动后机器正在运行的任何时候能安装设备。但是，某些设备必须在系统启动的特殊阶段被安装。为协助这些设备的厂商，这一节提供了一些系统设置阶段和设备安装相关的背景信息。

当 Windows 2000 启动时，Setup 程序的初始阶段仅安装 Windows 2000 运行所需要的最小数目的设备，如键盘、鼠标、显示适配器、SCSI/Disk 和 Machine/HAL。这个阶段的 Setup 程序是文本模式的设置。对于在文本模式设置期间安装设备的用户，你的分发盘必须包括一个名为 txtsetup.oem 的文本文件。更多的信息，参见第 9 章---安装一个要求重新启动机器的设备。

在文本模式设置完成后，Setup 程序启动 Windows 2000 并继续安装的 GUI 模式阶段。Setup 在 GUI 模式设置阶段安装大多数设备。

在 GUI 模式设置阶段没有安装的设备包括那些需要用户交互的安装。例如，如果这是第一次设备被设置并且一个此设备的协同安装程序提供了定制一个完成页，Setup 必须向用户显示此页。当用户以管理员权限登录时，Setup 执行这种设备安装。

一旦机器启动并运行，用户可以安装新设备：

• 为安装一个新的 PnP 设备，将设备插入机器中。

如果设备和总线支持热插拔通知，设备安装被自动初始化。

如果用户需要关掉机器，打开机箱以插入设备，设备将在系统启动时被识别和配置。

当系统正在运行时，如果设备能被插入，并且设备和总线不支持热插拔通知，使用增加/删除硬件向导以初始化设备配置。（选择“安装”）

• 安装一个非 PnP 设备，使用增加/删除硬件向导。

如果 Setup 能在系统进程的可信任的上下文中安装设备，用户不需要管理员权限来安装设备。如果 Setup 需要提示用户信息，它要求在管理员用户的进程上下文中安装设备。

为了更新已安装设备的驱动程序，使用设备管理器或者适宜的控制面板小程序中的硬件页。（点击控制面板小程序中的硬件页的属性按钮以启动设备的设备管理器。）

第2章 为设备提供驱动程序

这一章讨论了驱动程序包的文件集并讨论了驱动程序作为一个整体的事宜。此文档的其他部分讨论了对驱动程序文件内容的要求和指南。

这一章包括以下信息：

- 2.1 驱动程序文件
- 2.2 注册表中的驱动程序信息
- 2.3 指定驱动程序装载顺序
- 2.4 安装过滤器驱动程序
- 2.5 为设备安装 Null 驱动程序

2.1 驱动程序文件

支持一个特定的设备的软件依赖于设备的种类和设备被使用的方式。典型情况下，厂商在驱动程序包中提供下面的软件以支持设备：

1. 一个设备安装信息文件（INF 文件）。

一个 INF 文件包含系统 Setup 组件使用的信息以安装支持设备的文件。当 Setup 安装驱动程序时，它拷贝这些文件到%windir%\inf 目录中。这个文件是必须的。

更多的信息，参见第 3 章，建立一个 INF 文件。

1. 设备的一个或多个驱动程序。

一个.sys 文件时驱动程序的映像文件。当驱动程序被安装时，Setup 拷贝此文件到%windir%\system32\drivers 目录中。对大多数设备，驱动程序是必须的；某些设备不需要驱动程序，比如原始能力的设备。

更多的信息，参见这一章的剩余部分和其他章节。

2. 驱动程序包的数字签名（驱动程序目录文件）。

驱动程序目录文件包含数字签名。所有的驱动程序包都应被签名。

厂商通过提交它的驱动程序包给 Windows 硬件质量实验室（WHQL）测试、签名，从而获得数字签名。WHQL 将包返回，并附带一个目录文件（.cat 文件）。厂商必须为设备在 INF 文件中列出这个目录文件。更多的信息参见《Windows 2000 驱动程序开发参考》卷 1 第 3 部分第 1 章的 INF Version 节。参考 WHQL 指南以获得如何提交驱动程序包以测试、签名的信息。

3. 一个或多个协同安装程序。

协同安装程序是一个 Win32 DLL，它辅助设备在微软 Windows 2000 系统中的安装。例如，一个 IHV 也许会提供一个协同安装程序，从而将设备特定的信息写入注册表中，而这是不能被 INF 处理的。协同安装程序是可选的。更多的信息参见第 4 章，编写协同安装程序。

4. 其他文件。

一个驱动程序包可以包含其他文件，比如定制的设置应用程序，一个设备图标，一个驱动程序库文件（例如对视频驱动程序），等等。更多的信息参见第 6 章，编写定制的设备安装应用程序；第 7 章，提供设备属性页；第 9 章，安装一个要求重新启动机器的设备；以及图形驱动程序设计指南的第 1 部分，图形驱动程序。

2.2 注册表中的驱动程序信息

操作系统和驱动程序将关于驱动程序和设备的信息存储在注册表中。总的来说，驱动程序应使用注册表存储机器重启动后仍必须维护的数据。此外，驱动程序可以访问注册表以获得系统或者其他程序或者驱动程序存储在其中的信息。更多的关于注册表的信息，参见平台 SDK 文档。

以下的注册表中的树是驱动程序开发者特别感兴趣的(这里 HKLM 代表 HKEY_LOCAL_MACHINE)：

- HKLM\SYSTEM\CurrentControlSet\Services
- HKLM\SYSTEM\CurrentControlSet\Control
- HKLM\SYSTEM\CurrentControlSet\Enum
- HKLM\SYSTEM\CurrentControlSet\HardwareProfiles

驱动程序和用户模式的设置组件必须使用系统例程如 IoGetDeviceProperty 和 SetupDiGetDeviceRegistryProperty 访问注册表的 Plug and Play（PnP）主键。驱动程序不应直接访问这些主键。以下注册表信息的讨论仅仅是为调试一个设备的安装/配置问题。

在 HKLM\SYSTEM\CurrentControlSet 下的主键是保留驱动程序重要数据的安全地方，因为数据被存在系统区。系统采取特别的预防措施以保护系统区（例如做多个备份）。

HKLM\SYSTEM\CurrentControlSet\Services 树

这个树存储机器每个服务的信息。每个驱动程序有一个形式为 HKLM\SYSTEM\CurrentControlSet\Services\DriverName 的主键。当 PnP 管理器调用驱动程序的 DriverEntry 例程时，它在 RegistryPath 参数中将此路径传给驱动程序。驱动程序可以将全局驱动程序定义的数据存储在它的 Services 树主键下。在此主键下存储的信息在它初始化时对驱动程序是可用的。

以下的主键和值表项是特别有意义的：

ImagePath

说明驱动程序映像文件全路径的值表项。Setup 使用驱动程序 INF 文件的必须的 ServiceBinary 表项建立这个值。这个表项在驱动程序的 INF AddService 指令中引用的 service-install-section 中。这个路径的典型值是 %windir%\system32\Drivers\DriverName.sys，此处 DriverName 是驱动程序的 Services 主键名称。

Parameters

用来存储驱动程序专用数据的主键。对一些类型的驱动程序，系统期望找到专用的值表项。你可以使用驱动程序的 INF 文件的 AddReg 表项来对这个子键添加值表项。

Performance

对可选的性能监视信息说明的主键。在此主键下的值说明了驱动程序的性能 DLL 的名称和在此 DLL 中某些导出的函数的名称。你可以使用驱动程序的 INF 文件的 AddReg 表项来对这个子键添加值表项。

HKLM\SYSTEM\CurrentControlSet\Control 树

注册表的这个树包含了控制系统启动和一些设备配置方面的信息。以下的子键特别令人感兴趣：

Class

包含关于计算机上的设备设置类别的信息。对每个类有一个子键，使用设置类别的 GUID 命名。每个子键包含一个设置类别的信息，如类安装程序（如果有的话），注册的类顶层过滤器驱动程序，

注册的类底层过滤器驱动程序，等等。每个类别子键为每个安装在系统中的类设备实例包含软件主键（驱动程序的主键）。

CoDeviceInstallers

包含关于特定类别的、以注册的协同安装程序的信息。

DeviceClasses

包含关于机器中设备接口的信息。对每个设备接口类有一个子键，每个已注册到设备接口类的接口的实例，在那些子键下有一个表项。

HKLM\SYSTEM\CurrentControlSet\Enum 树

注册表的这个树包含了关于机器中设备的信息。PnP 管理器为每个设备建立一个子键，名字是 HKLM\SYSTEM\CurrentControlSet\Enum\enumerator\deviceID 的形式。在这些主键下是一个子键，是机器中现有的每个设备实例。设备实例的子键具有如设备描述、硬件 ID、兼容性 ID、资源要求等等的信息。

Enum 树为操作系统组件的使用而保留，并且它的布局容易改变。驱动程序和用户模式的 Setup 组件必须使用系统 API，如 IoGetDeviceProperty 和 SetupDiGetDeviceRegistryProperty，以从树中提取信息。驱动程序和 Setup 应用程序不应直接访问 Enum 树。当调试驱动程序时，你可以使用注册表编辑器察看 Enum 树。

HKLM\SYSTEM\CurrentControlSet\HardwareProfiles

注册表的这个树包含了计算机上关于硬件 profiles 的信息。

2.3 指定驱动程序装载顺序

对于大多数设备，Setup 和 PnP 管理器装载驱动程序的顺序由机器中设备的物理层次所决定。Setup 和 PnP 管理器从系统根设备的配置开始，然后是根设备的子设备（例如，PCI 适配器），然后是这些子设备的子设备等等。如果驱动程序还没有为另一个设备装入，PnP 管理器为每个设备装入驱动程序。

在 INF 文件中的设置可以影响驱动程序的装入顺序。这一节描述了厂商应当在在驱动程序的 INF AddService 指令中引用的 service-install-section 中说明的相关值。这一节讨论了 StartType、LoadOrderGroup、Dependencies 表项。

为说明 StartType，驱动程序应该遵守这些规则：

• PnP 驱动程序

PnP 驱动程序应该有一个 SERVICE_DEMAND_START（0x3）的开始类型，说明无论什么时候 PnP 管理器发现一个驱动程序可以服务的设备时，它都可以装入驱动程序。

• 必须要起动机器的设备的驱动程序

如果设备要求重启动机器，设备的驱动程序应有一个 SERVICE_BOOT_START（0x0）的开始类型。

• 检测不是 PnP 可枚举的且不要求起动机器的设备的驱动程序

对于非即插即用设备，驱动程序通过调用 IoReportDetectedDevice 向 PnP 管理器报告此设备。这样的驱动程序有一个 SERVICE_SYSTEM_START（0x01）的开始类型，这样 Setup 在系统初始化时将装入驱动程序。

仅仅报告非即插即用设备的驱动程序应该设置这种开始类型。如果驱动程序同时为 PnP 设备和非 PnP 设备服务，它应当设置这种开始类型。

·必须被服务控制管理器启动的非 PnP 的驱动程序

这样的驱动程序有一个 SERVICE_AUTO_START（0x02）的开始类型。PnP 驱动程序不应设置这种类型。

PnP 驱动程序的开发者应使此驱动程序在 Setup 配置驱动程序所服务的设备的任何时候，都能被加载。相反，在 PnP 管理器确定驱动程序所服务的设备不存在时，驱动程序应能在任何时候被卸载。PnP 驱动程序所依赖的唯一驱动程序装载顺序如下：

·子设备的驱动程序可以依赖于已被加载的父设备的驱动程序。

·在设备栈的驱动程序依赖于在它之下的已被加载的驱动程序。(例如，功能驱动程序要确定任何低层的过滤器驱动程序已经被加载。)注意在设备栈的驱动程序不能依赖于在任何较低层驱动程序之后被加载，因为当另一个此类设备被配置时，驱动程序也许已经被装载了。在过滤组之内的过滤器驱动程序不能预测它们的装载顺序。例如，如果一个设备有三个注册的顶层过滤器驱动程序，这三个驱动程序会在功能驱动程序之后被加载，但是在它们的过滤组之内，装载顺序是任意的。

如果一个驱动程序的装载顺序明显依赖于另一个驱动程序,这个依赖性通过父亲/孩子关系来实现。子设备的驱动程序被加载以前，它依赖于父设备的驱动程序已被加载。

为了增加设置正确的 StartType 值的重要性，下表描述了 Setup 和 PnP 管理器如何在 INF 文件中使用 StartType 表项：

1. 在系统启动时，操作系统装载程序在将控制转移给内核以前，装入 SERVICE_BOOT_START 类型的驱动程序。当内核得到控制时，这些驱动程序已在内存中。Boot-start 类型的驱动程序使用 INF LoadOrderGroup 表项以安排装载顺序。(Boot-start 类型的驱动程序在大多数设备被配置以前装入，这样它们的装入顺序不能被设备层次所决定。）操作系统忽略 Boot-start 类型的驱动程序 INF 的 Dependencies 表项。
2. PnP 管理器调用 SERVICE_BOOT_START 类型的驱动程序的 DriverEntry 例程，这样驱动程序可以服务 boot 类型的设备。

如果 boot 类型的驱动程序的设备有子设备，这些设备就被枚举。如果子设备的驱动程序也是 Boot-start 类型的驱动程序，它们被配置和启动。如果一个设备的驱动程序不全是 Boot-start 类型的驱动程序，PnP 管理器为设备建立设备节点但并不启动设备。

3. 在所有的 boot 类型的驱动程序被装入以及 boot 类型的设备被启动后，PnP 管理器配置其余的 PnP 设备并装入它们的驱动程序。

PnP 管理器遍历设备树并为还未启动的设备节点装入驱动程序（即任何在前面步骤还没有启动的设备节点)。当每个设备启动时，PnP 管理器枚举设备的子设备（如果它有子设备)。

在 PnP 管理器配置这些设备时，它为设备装入驱动程序并启动设备，而不管驱动程序的 StartType 值。许多这些驱动程序是 SERVICE_DEMAND_START 类型的，但是它们可以有任何 StartType 值。

PnP 管理器忽略在这一步骤装入驱动程序中由于 INF Dependencies 表项和 LoadOrderGroup 表项而建立的注册表表项。装载顺序基于物理设备层次。

在这一步骤的之后，所有的设备已经被配置，除了不是即插即用可枚举的设备和这些设备的子设备之外。(子设备也许是、也许不是即插即用可枚举的。)

4. PnP 管理器装入 StartType 为 SERVICE_SYSTEM_START 的、还没有被装入的驱动程序。

这些驱动程序检测并报告它们的非 PnP 设备。PnP 管理器处理由于这些驱动程序的 INF LoadOrderGroup 表项而导致的注册表表项。它忽略由于这些驱动程序的 INF Dependencies 表项而建立的注册表表项。

5. 服务控制管理器装入 StartType 为 SERVICE_AUTO_START 的、还没有被装入的驱动程序。

服务控制管理器处理关于服务的 DependOnGroup 和 DependOnService 的服务数据库信息。这个信息来自于 INF AddService 表项的 Dependencies 表项。注意 Dependencies 信息仅仅为非 PnP 驱动程序处理，因为任何必须的 PnP 驱动程序在系统启动的前面步骤中已经被装入。服务控制管理器忽略 INF LoadOrderGroup 信息。

参见平台 SDK 文档获得更多关于服务控制管理器的信息。

2.4 安装过滤器驱动程序

一个 PnP 过滤器驱动程序能支持一个特定的设备或者支持一个设置类的所有设备以及可以依附在设备的功能驱动程序之下（底层过滤程序）或者之上（顶层过滤程序）。参见第 1 部分的第 1 章驱动程序种类以获得更多关于 PnP 驱动程序层的信息。

为了注册一个设备专用的过滤器驱动程序，通过设备的 INF 文件的 DDInstall.HW 节的 AddReg 表项建立一个注册表表项。对于设备专用的顶层过滤，建立名为 UpperFilters 的表项。对于设备专用的底层过滤，建立名为 LowerFilters 的表项。例如，以下 INF 摘录了安装 cdaudio 作为 cdrom 驱动程序的顶层过滤：

```
:
; Installation section for cdaudio.Sets cdrom as the service
; and adds cdaudio as a PnP upper filter driver.
;
[cdaudio_install]
CopyFiles = cdaudio_copyfiles,cdrom_copyfiles
[cdaudio_install.HW]
AddReg = cdaudio_addreg

[cdaudio_install.Services]
AddService = cdrom,0x00000002,cdrom_ServiceInstallSection
AddService = cdaudio,,cdaudio_ServiceInstallSection
:

[cdaudio_addreg]
HKR,,"UpperFilters",0x00010000,"cdaudio";REG_MULTI_SZ value
:

[cdaudio_ServiceInstallSection]
DisplayName = %cdaudio_ServiceDesc%
ServiceType   = 1       ; SERVICE_KERNEL_DRIVER
StartType     = 3       ; SERVICE_DEMAND_START
ErrorControl  = 1       ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\cdaudio.sys
:
```

为了安装一个设备设置类别的类范围的顶层/底层过滤程序，你可以提供一个安装必要服务的设置应用程序。设置应用程序然后为希望的设备设置类别注册作为顶层或底层过滤的服务。为拷贝服务的二进制文件，设置应用程序可以使用 SetupInstallFiltersFromInfSection。为安装服务，设置应用程序可以使用 SetupInstallServicesFromInfSection。为了为特别的设备设置类别注册作为顶层或底层过滤的服务，设置应用程序为每个感兴趣的设备设置类别调用 SetupInstallFromInfSection，使用从 SetupDiOpenClassRegKey 提取的注册表主键句柄作为 RelativeKeyRoot 参数。例如，考虑以下的 INF 节：

```
:

[DestinationDirs]
upperfilter_copyfiles = 12

[upperfilter_inst]
CopyFiles = upperfilter_copyfiles
AddReg = upperfilter_addreg

[upperfilter_copyfiles]
upperfilt.sys

[upperfilter_addreg]
;append this service to existing REG_MULTI_SZ list,if any
HKR,,UpperFilters,0x00010008,upperfilt

[upperfilter_inst.Services]
AddService = upperfilt,,upperfilter_service

[upperfilter_service]
DisplayName = %upperfilter_ServiceDesc%
ServiceType   = 1       ; SERVICE_KERNEL_DRIVER
StartType     = 3       ; SERVICE_DEMAND_START
ErrorControl  = 1       ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\upperfilt.sys
:
```

设置应用程序将为[upperfilter_inst]节调用 SetupInstallFilesFromInfSection。接下来，它为[upperfilter_inst.Services]节调用 SetupInstallServicesFromInfSection。最后，为[upperfilter_inst]节调用 SetupInstallFromInfSection，为每个它想注册的顶层过滤服务的类主键分别调用一次。每个调用会为 Flags 参数说明 SPINST_REGISTRY 以指示仅仅注册表的修改需要被执行。

2.5 为设备安装 Null 驱动程序

如果设备在机器中没有使用并且设备不会被启动，你也许为设备安装一个 null 驱动程序。这样的设备典型情况下并不存在，但是如果存在，你可以安装一个 null 驱动程序，此设备的 INF 的表项如下：

```
:
[MyModels]
%MyDeviceDescription% = MyNullInstallSection,&BadDeviceHardwareID%
:
[MyNullInstallSection]
;the install section must be empty

[MyNullInstallSection.Services]
AddService = ,2    ; no value for the service name
:
```

在 Models 节的设备的硬件 ID 应该具体的标识此设备，使用子系统厂商 ID 和相关的其他信息。操作系统为设备建立设备节点，但是如果设备不是具有未加工能力（raw capable），操作系统不会启动设备。

如果设备有启动配置，那些资源会被保留。

第3章 建立 INF 文件

为了安装 Microsoft Windows 2000 驱动程序，必须有一个 INF 文件。一个 INF 文件是一个文本文件，包含了关于设备和要安装的文件的必要信息，如驱动程序映像、注册表信息、版本信息等等，这些都被 Setup 组件所使用。

Windows 2000 INF 文件不包含安装脚本。安装程序是 Win32 安装应用程序的一部分，如新设备向导和增加/删除硬件向导，每个 INF 文件作为一个资源。

包括对设备的安装支持，INF 文件为设备初始化一个新的设置类别，比如 INF 有一个 ClassInstall32 节。

这一章包含以下信息：

- 3.1 INF 文件总体指南
- 3.2 为设备文件说明源和目标位置
- 3.3 建立跨平台的和/或者双操作系统的 INF 文件
- 3.4 建立国际化的 INF 文件
- 3.5 在设备的 INF 文件中加强打开文件的安全性
- 3.6 从应用程序访问 INF 文件

参见《Windows 2000 驱动程序开发参考》第 1 卷中的第 3 部分的第 1 章---INF 文件的节和指示以获得 INF 文件格式的完整描述。

3.1 INF 文件总体指南

INF 文件有一些公共部分并遵守一套单一的语法规则，但是由于 Windows 支持的各种设备，它们也是有区别的。编写 INF 文件时，你应该求教于以下的信息资源。

- 这一章和 INF 文件节和指令参考资料。
- 你的设备的类别的文档。

例如，如果你的设备是图形适配器，参见 DDK 文档的图形驱动程序一节。

- INF 文件的 DDK 工具。

DDK 提供了一些工具以协助编写 INF 文件。参见 DDK 的 tools 目录中的关于工具如 GenINF 和 ChkINF 的信息。

- INF 示例文件和类似设备的 INF 文件。

DDK 包括了它的示例驱动程序的 INF 文件。分析这些示例文件，看看是否有设备的 INF 文件类似于你的设备。

你可以使用任何编辑器建立或者修改 INF 文件。如果你的 INF 文件包含非 ASCII 字符，将文件存成 Unicode 文件。注意 ChkINF 不支持 Unicode 文件。

一个和 Windows 2000 操作系统一起发行的 INF 文件必须有 xxxxxxxx.inf 的文件名，这里 xxxxxxxx 不能超过 8 个字符。独立于操作系统发行的 INF 文件的名称不受 8 个字符的限制。

3.2 为设备文件说明源和目标位置

当 Setup 处理 INF 文件中的拷贝、更名、删除文件语句时，它为文件确定源和目标位置。为了决定这些位置，它估计驱动程序是和操作系统一起发行还是单独发行，并检查 INF 文件的各节和表项，包括 SourceDisksNames，SourceDisksFiles，LayoutFile，Include，Needs，DestinationDirs。这一节描述了 Setup 如何确定源和目标位置以及提供指南以帮助你正确的说明这些位置。

独立于操作系统发行的 INF 的源介质

独立于操作系统发行的设备的 INF 文件使用 SourceDisksNames 和 SourceDisksFiles 节来说明文件的位置。如果这样的 INF 在 DDInstall 节包含 Include 和 Needs 表项，那些表项就说明了另外的可能的源位置。

如果 INF 有 SourceDisksNames 和 SourceDisksFiles 节，而没有 Include 表项，那么 SourceDisksNames 和 SourceDisksFiles 节必须列出所有的除了目录和 INF 文件以外的原介质和源文件。(Windows 98 要求目录文件被列在 SourceDisksFiles 节；Windows 2000 忽略这个表项。)目录文件必须和 INF 文件处在同一个位置。目录文件不能被压缩。

和操作系统一起发行的 INF 的源介质

如果支持设备的文件被包括在操作系统中，INF 必须在 Version 节说明 LayoutFile 表项。这个表项说明文件在操作系统介质的何处。说明 LayoutFile 表项的 INF 不能包括 SourceDisksNames 和 SourceDisksFiles 节。

仅仅系统提供的 INF 可以直接引用 layout.inf。和操作系统一起发行的 OEM 和 IHV 文件可以引用系统提供的 INF（如 mf.inf）中的公共 install 节。

源介质和包含 Include 和 Needs 表项的 INF

如果 INF 有 SourceDisksNames 和 SourceDisksFiles 节，以及 Include 和 Needs 表项，Setup 使用主 INF 文件加上任何“包括”的 INF 文件以确定源介质的位置。当说明源介质和源文件位置时，包含的文件要尽可能的精确是特别的重要。

考虑在图 3.1 所示的包含的 INF 的层次。

图 3.1　包含的 INF 文件的层次示例

在图 3.1，对于一个多功能设备的 INF，MyMfDevice.inf，包含了系统提供的 mf.inf 文件。反过来，系统提供的 mf.inf 文件包含了系统提供的文件 layout.inf.。当 Setup 搜索源介质以便从它拷贝在 MyMfDevice.inf 所引用的文件时，它在 MyMfDevice.inf 和任何引用被拷贝文件的包含的 INF 文件中寻找 SourceDisksFiles 节。Setup 首先搜索 MyMfDevice.inf，但是并不保证对包含的 INF 文件的搜索顺序。

修饰的 SourceDisksFiles 节比未修饰的节优先，即使修饰的节在一个包含的文件中。例如，图 3.1 所示的 INF 文件，当在一个 x86 机器上安装时，如果 layout.inf.包含[SourceDisksFiles.x86]节并且 MyMfDevice.inf 仅仅包含了一个未修饰的[SourceDisksFiles]节，Setup 首先从 layout.inf.使用修饰的节。因此，包含其他 INF 的 INF 应包含使用平台扩展的某个节名。

典型情况下，厂商提供的 INF 应当说明它的驱动程序包的文件的位置，不应当使 Setup 为了文件位置而搜索包含的 INF 文件。换句话说，拷贝文件的厂商 INF 应说明 SourceDisksNames 和

SourceDisksFiles 节，那些节应当用平台扩展来修饰，也应包含被 INF 直接拷贝的所有文件的信息。另外，如果厂商文件名字尽可能的具体是有益的，除非安装程序正在替换和操作系统一起发行的厂商提供的文件。

为设备安装文件说明目标位置

INF 文件用 DestinationDirs 节来说明设备文件的目标位置。这一节应当总是被说明的，并且和拷贝、更名、删除语句的节在同一个 INF 文件中。

DestinationDirs 节应当包含 DefaultDestDir 表项。

如果一个 INF 有拷贝、更名、删除语句的节但是没有 DestinationDirs 节并且这个 INF 包含了其他 INF，Setup 搜索包含的 INF 以确定目标位置信息。但是，Setup 搜索包含的 INF 的顺序是不可预测的。因此，Setup 会冒一个风险，例如，拷贝文件到一个违背 INF 开发者意愿的位置。为避免这样的混淆，总是在包含拷贝、更名、删除语句的节中说明 DestinationDirs 节。DestinationDirs 节应当至少包括 DefaultDestDir 表项，并能列出 INF 中的拷贝、更名、删除文件的节。

3.3 建立跨平台的和/或者双操作系统的 INF 文件

你可以使用系统定义的扩展以建立一个单一的跨平台的安装和/或者双操作系统安装的 INF 文件。扩展能使你建立修饰节名称和修饰指令。这个修饰说明了哪个 DDInstall 节和相关的指令对每个平台和/或者操作系统是重要的。你可以建立一个 INF，对在 x86 平台的 Windows 2000 设备给与安装支持，对未来平台的 Windows 2000 给与安装支持，对所有平台的 Windows 2000 给与安装支持，对 Windows 9x 平台的支持，或者对所有以上平台的支持。

例如，如果你的设备支持 Windows 2000 和 Windows 9x 平台，你将建立一个有两个或多个“并行”DDInstall 节的 INF，它有下面的扩展以控制在每个平台和/或者每个操作系统安装什么：

- [install-section-name.NTx86]包含在所有的 Windows 2000 x86 平台上为安装设备或者一套设备兼容的模块而要求的 Windows 2000 专用指令。
- [install-section-name.NT]包含在所有的 Windows 2000 平台上为安装设备或者一套设备兼容的模块而要求的 Windows 2000 专用指令。

当前，.NT 代表了由.ntx86 定义的相同的一套平台。如果在将来 Windows 2000 支持非 x86 平台，.NT 将代表包含.ntx86 定义的平台和那些新平台。

- [install-section-name]包含 Windows 9x 专用指令。

未修饰的节在 Windows 2000 平台也适用。Setup 在 Windows 2000 上使用未修饰的节如果它可以定位合适的修饰节。但是为了防止 Windows 2000 的节错误的被用在 Windows 9x 上，你应当对任何 Windows 2000 特定的 DDInsatll 节和相关的节以.ntx86 或者.nt 进行修饰。

Windows Setup 如下处理 DDInsatll 节：

- 在 Windows 98，Setup 处理未修饰的节并忽略以.nt*平台扩展修饰的任何节。
- 在 Windows 2000：

Setup 检查特定平台扩展的 install 节，如.ntx86。如果有一个存在，Setup 为当前平台处理合适的节。Setup 在正被处理的 INF 和任何包含的 INF 文件（即在 Include 表项中包含的 INF）中检查特定平台的 install 节。

如果没有特定平台扩展，Setup 检查在 INF 或者任何包含的 INF 中检查 DDInstall.nt 节。如果有一个，Setup 处理 DDInstall.nt 节。

如果没有 DDInstall.nt 节，Setup 处理未修饰的解。

有平台扩展的 DDInstall 节也可以设置 INF 的附加的、每个设备节。特别的，对于设备和驱动程序安装，设置 Windows 2000 要求的 DDInstall.Services 以及可能 DDInstall.HW，DDInstall.CoInstallers，DDInstall.LogConfigOverride，和/或者 DDInstall.Interface 节。在双操作系统和/或者跨平台设备/驱动程序 INF 文件中，在 INF 开发者定义的节的名称后立即说明适宜的扩展，例如，install-section-name.NTx86.HW。

如果在 INF 文件中没有一个或多个修饰的 DDInstall.NT 和/或者 DDInstall.NTx86 节，INF 必须有并行的修饰和未修饰的附加的每个设备节。即，如果一个双操作系统 INF 有 DDInstall.NT 和 DDInstall 节以及 DDInstall.HW 节，它也必须有并行的 DDInstall.NT.HW 节（如果设备或者驱动程序要求一个 Windows 2000 .HW 节）。

INF 文件节和指令文档说明了这些扩展，它作为正规语法语句的一部分，例如：

[DDInstall.HW]

[DDInstall.nt.HW]

[DDInstall.ntx86.HW]

但是，这样一个正规语法语句指示这些扩展是合法的，但是对基本节可能是可选的。它并不指示任何有 DDInstall.nt.HW 节的 INF 也必须有每一个其他特定平台的 DDInstall.ntXXX.HW 节。Windows 2000 跨平台和/或者双操作系统 INF 文件的编写者可以使用任何这些系统定义的扩展的子集以说明特别的跨平台和/或者双操作系统设备/驱动程序安装所要求的最小数目的修饰节。

3.4 建立国际化的 INF 文件

将在国际市场使用的 INF 应当对所有用户可见的文本使用%strkey%令牌。字符串令牌被定义在[Strings]节，它通常在 INF 文件的末尾。

为了建立一个单一的国际化的 INF 文件,你应当包含一套本地专用的 Strings.LanguageID 节，在 INF Strings 节的参考页中有描述。

可选的，你也可以为每种本地化建立分离的 INF。为减少重复和易于维护，除了 Strings 节，你建立一个有必须节和表项的主 INF 文件，。然后建立第二套文件，这里每个文件为支持的本地化仅包含 Strings 节。将主文件和每个字符串文件相连以产生本地专用的 INF 文件。

如果一个 INF 包含在 ASCII 范围之外的字符（在 0-127 之外），INF 应当是 Unicode 格式。为建立这样的文件，例如，从应用程序如记事本程序将 INF 存成 Unicode 格式文件。如果 INF 不是 Unicode 格式，Setup 使用当前的本地化翻译字符。如果 INF 是 Unicode 格式，Setup 使用全字符集。

3.5 在设备的 INF 文件中加强打开文件的安全性

对 Windows 2000，微软为某些设备类别加强了在类安装程序的 INF 中的打开文件的安全性，包括 CDROM、DiskDrive、FDC、FloppyDisk、HDC 和 SCSIAdapter。如果你不能确定类安装程序是否为你的设备加强了文件打开时的安全性，你应在设备 INF 中加强安全性。

在设备的 INF 文件中，在[Xxx_AddReg]节增加以下的表项：

[install-section-name.HW] |

```
[install-section-name.nt.HW] |
[install-section-name.ntx86.HW]
...
AddReg = Xxx_AddReg
...
[Xxx_AddReg]
...
HKR,,DeviceCharacteristics,0x10001,characteristics
```

Characteristics 值是赋给设备的特性数字值。这个值是定义在 wdm.h 或者 ntddk.h 中的一个或多个 FILE_* 文件特性的逻辑或（OR）的结果。对于安全性重要的特性是 FILE_DEVICE_SECURE_OPEN，它的数字值为 0x100。设置这个特性使 I/O 管理器对所有设备的打开请求执行安全检查。FILE_DEVICE_SECURE_OPEN 特性在 wdm.h 和 ntddk.h 中定义，并被 Windows 2000 平台支持，但是不被 Win9x 支持。

特性值为 0 使 Setup 忽略类范围内的设备特性（在相关联的类安装程序 INF 中说明）。

在设备安装期间，PnP 管理器以如下方式为设备对象确定设备特性：

它提取设备类的特性设置（在类安装程序 INF 中说明）。

它提取设备的特性（在设备 INF 中说明）。特定设备的设置覆盖类的设置。

或者直接或者通过 IoCreateDevice，它将特定 INF 的设置和那些由驱动程序在设备对象中的设置进行逻辑或。

它将 FDO 的设备特性设置和任何过滤 DO 逻辑或，并传播这些设置到 FDO 和过滤 DO。如果设备在原始方式，那么就没有 FDO，PnP 管理器传播设备特性到 PDO 和任何过滤 DO。

以上的步骤确保了，如果一个驱动程序为设备设置了 FILE_DEVICE_SECURE_OPEN，那么这个特性被传播该所有的驱动程序所服务的设备的设备对象。

PnP 管理器在它已经为设备的驱动程序调用 AddDevice 例程之后，以及在驱动程序已经建立了它们的设备对象并把它们连接到设备堆栈之后，但是在它发送 IRP_MN_START_DEVICE 请求之前，逻辑或并传播这些特性。

当 PnP 管理器传播 FILE_DEVICE_SECURE_OPEN 时，它也传播以下的设备特性：FILE_ROMOVABLE_MEDIA，FILE_READ_ONLY_DEVICE，FILE_FLOPPY_DISKETTE。

一个有 ClassInstall32 节的 INF 文件可以为设备的设置类设置 FILE_DEVICE_SECURE_OPEN 特性。参见第 5 章，在一个类安装程序的 INF 文件中加强打开文件的安全性。

3.6 从应用程序访问 INF 文件

大多数 INF 文件由系统类安装程序和其他系统应用程序所使用。这一节包含了厂商提供的应用程序必须访问 INF 文件的信息。

这一节描述了可以用 INF 文件执行几个公共操作以及通用的 Setup 函数。对于如何使用这些函数的完整信息，参见平台 SDK 文档。

3.6.1 打开和关闭 INF 文件

在一个应用程序访问 INF 文件之前，它调用 SetupOpenInfFile 打开文件。这个函数返回一个 INF 文件的句柄。

如果你不知道需要打开的 INF 文件的名称，使用 SetupGetInfFileList 获得在目录中所有 INF 文件的列表。

一旦应用程序打开一个 INF 文件，它可以使用 SetupOpenAppendInfFile 将另外的 INF 文件插入到打开的 INF 文件之后。当后续的 Setup 函数引用打开的 INF 文件，他们也可以访问存储在任何插入的文件中的信息。

如果在调用 SetupOpenAppendInfFile 中没有 INF 文件被说明，这个函数在调用 SetupOpenInfFile 中打开 INF 文件的 Version 节的 LayoutFile 表项所说明的文件。

当在 INF 文件中的信息不再需要时，应用程序应当调用 SetupCloseInfFile 以释放在 SetupOpenInfFile 所分配的资源。

3.6.2 从 INF 文件提取信息

一旦你有了 INF 文件的句柄，你可以按不同的方式提取信息。诸如 SetupGetInfInfomatiom，SetupQueryInfFileInfomatiom，SetupQueryInfVersionInfomatiom 等函数从特定的 INF 文件中提取信息。

别的函数，例如 SetupGetSourceInfo 和 SetupGetTargetPath，从源文件和目标目录得到信息。

另外的函数，例如 SetupGetLineText 和 SetupGetStringField，能使你直接访问存在 INF 文件的一行或一个域的信息。这些函数被更高一层的 Setup 函数内部使用，但是如果你需要直接访问行或域级其他信息，也可以使用这些函数。

第4章　编写协同安装程序

协同安装程序是微软公司开发的 win32 的 DLL，它是用来帮助在 Windows 2000 系统上进行设备安装。它被 Setup API 调用作为类安装程序的“助手”。例如，供应商可以提供协同安装程序将特定设备信息写入 INF 文件无法处理的注册表中。

本章内容：

- 4.1　协同安装程序总览
- 4.2　协同安装程序界面
- 4.3　协同安装程序操作
- 4.4　注册协同安装程序

4.1　协同安装程序总览

由 Setup API 调用的协同安装程序如图 4-1 所示。

图 4-1　协同安装程序在设备安装中的分工

带有阴影的方框表示可由 IHV 和 OEM 提供的组件，其他组件则由 OS 提供。参见第 1 章“*设备安装总览*”可以了解更多的有关安装组件的信息。

协同安装程序可以是设备专用或类专用的。Setup API 只在安装协同安装程序为其注册的设备时才调用一个设备专用的协同安装程序。操作系统(OS)及供应商可以为一个设备注册零个或多个设备专用的协同安装程序。在为协同安装程序注册安装设备设置类的任何设备时，Setup API 调用类协同安装程序。操作系统及供应商可以为一个设备设置类注册一个或多个类协同安装程序。除此之外，类协同安装程序还可以为一个或多个设置类注册。

GUI 模式设置、新设备 DLL 以及定制设置应用程序通过调用带有设备安装函数代码(DIF 代码)的 **SetupDiCallClassInstaller** 来安装设备。对于每一个 DIF 请求，**SetupDiCallClassInstaller** 调用为设备设置类注册的任何类协同安装程序，调用为特定设备注册的任何设备协同安装程序，以及由系统提供用于设备设置类的类安装程序(如果有的话)。

定制设置应用程序须调用 **SetupDiCallClassInstaller** 而不是直接调用协同安装程序或类安装程序。这个函数可以保证所有注册的协同安装程序都能被正确调用。

类协同安装程序一般都在设备安装之前注册，而设备专用的协同安装程序则是作为设备安装的一部分被注册的。因此类协同安装程序总是在第一次构建时就被添加到协同安装程序列表之中，并在设备安装时被所有 DIF 请求调用。特定设备协同安装程序则是在为该设备完成 DIF_REGISTER_COINSTALLERS 请求之后被添加到重安装列表之中的(或是在调用了 **SetupDiRegisterCoDeviceInstallers** 之后)。特定设备协同安装程序并不参与以下 DIF 请求：DIF_ALLOWINSTALL、DIF_INSTALLDEVICEFILES 及 DIF_SELECTBESTCOMPATDRV。

如果协同安装程序需要响应以下任何一个 DIF 请求，它就必须是一个类协同安装程序(而不是设备专用协同安装程序)：

- DIF_FIRSTTIMESETUP，DIF_DETECT*

- DIF_NEWDEVICEWIZARD_PRESELECT
- DIF_NEWDEVICEWIZARD_SELECT
- DIF_NEWDEVICEWIZARD_PREANALYZE
- DIF_NEWDEVICEWIZARD_POSTANALYZE

设备协同安装程序并不适用于这样的上下文，这是因为并没有标识出某个特定的设备，或是因为在安装的这个初始阶段还没有注册过设备安装程序。

图 4-2 显示了在注册了任意一个特定设备的协同安装程序之后，**SetupDiCallClassInstaller** 调用协同安装程序及类安装程序的顺序。

图 4-2　为 DIF 请求调用协同安装程序的处理及后处理示例

在图 4-2 所演示的示例中，为该设备的设置类注册了两个类协同安装程序以及一个特定设备协同安装程序。以下步骤对应于图 4-2 中的带圆圈的数字标号：

1. **SetupDiCallClassInstaller** 调用第一个类协同安装程序，同时指定一个表示安装请求正在处理中的 DIF 代码(在本例中是 DIF_INSTALLDEVICE)。协同安装程序在安装请求中有参与的选择权。本例中，第一个注册的类协同安装程序返回 NO_ERROR。
2. 接下来，**SetupDiCallClassInstaller** 调用任意额外注册的类协同安装程序。在本例中，第二个类安装程序返回了 ERROR_DI_POSTPROCESSING_REQUIRED，它要求在后处理之后再调用协同安装程序。
3. **SetupDiCallClassInstaller** 调用任意注册的设备专用协同安装程序。
4. 在调用了所有的注册过的协同安装程序后，如果设备的设置类有一个系统提供的类安装程序，**SetupDiCallClassInstaller** 就调用它。在本例中，类安装程序返回 ERROR_DI_DO_DEFAULT，这是类安装程序的一个典型返回值。
5. 如果有一个缺省的设备处理驱动程序，**SetupDiCallClassInstaller** 就为安装请求调用它。DIF_INSTALLDEVICE 有一个缺省的设备处理驱动程序 **SetupDiInstallDevice**，它是 Setup API 的一部分。
6. **SetupDiCallClassInstaller** 调用任何要求后处理的协同安装程序。在本例中，第二个类协同安装程序要求了后处理。

除了协同安装程序在它的单个入口点被再一次调用外，协同安装程序的后处理与驱动程序的 IoCompletion 例程相似。当 **SetupDiCallClassInstaller** 为后处理调用协同安装程序时，它将 *PostProcessing* 设为 TRUE，并将 *InstallResult* 设为 *Context* 参数中的恰当值。在本例中，Context.InstallResult 是 NO_ERROR，这是因为成功地执行了缺省的设备处理驱动程序。

在后处理中，**SetupDicallClassInstaller** 反向调用了协同安装程序。如果图 4-2 中的所有协同安装程序都已返回了 ERROR_DI_POSTPROCESSING_REQUIRED，那么 **SetupDiCallClassInstaller** 就会为后处理先调用 Device_Coinstaller_1，之后再是 Class_Coinstaller_2，和 Class_Coinstaller_1。类安装程序并不要求后处理，只有协同安装程序才要求。

即使先前的协同安装程序在安装请求中失败，要求后处理的协同安装程序也会被调用。

4.2 安装程序界面

协同安装程序具有以下原型：

```
typedef
DWORD
( CALLBACK* COINSTALLER_PROC)  (
      IN DI_FUNCTION   InstallFunction,
      IN HDEVINFO   DeviceInfoSet,
      IN PSP_DEVINFO_DADA   DeviceInfoData   OPTIONAL,
      IN OUT PCOINSTALLER_CONTEXT_DATA   Context
      );
```

InstallFunction 指定了正被处理的设备安装请求，在其中协同安装程序具有参与的选择权。例如，DIF_INSTALLDEVICE。参见《Windows 2000 Driver Development Reference》一书的第一卷第 3 部分的第 5 章“安装功能代码”有关“在 DIF 代码上的文档处理”内容。

DeviceInfoSet 提供了一个设备信息集的标识值。

DeviceInfoData 有选择性地标识作为设备安装请求的目标设备。如果这个参数是非 NULL 的，它就在设备信息集中标识一个元素。当 **SetupDiCallClassInstaller** 调用一个特定设备协同安装程序时，*DeviceInfoData* 为非 NULL。特定类协同安装程序可以同一个具有 NULL *DeviceInfoData* 的 DIF 请求（如 DIF_DETECT 或 DIF_FIRSTTIMESETUP）一起被调用。

Context 指向该安装请求的特定协同安装程序上下文结构。这个上下文信息的格式如下：

```
Typedef   struct
_COINSTALLER_CONTEXT_DATA   {
      BOOLEAN   PostProcessing;
      DWORD   InstallResult;
      PVOID      PrivateData;
}   COINSTALLER_CONTEXT_DATA,   *PCOINSTALLER_CONTEXT_DATA;
```

当在恰当的类安装程序（如有的话）处理了 *InstallFunction* 之后再调用协同安装程序时，*PostProcessing* 为 TRUE。*PostProcessing* 对协同安装程序是只读的。

如果 *PostProcessing* 为 FALSE，则 *InstallResult* 不相关。如果 *PostProcessing* 为 TRUE，*InstallResult* 就是安装请求的当前状态。该值为 NO_ERROR 或是一个由为该安装请求调用的先前部分返回的错误状态。协同安装程序可以为它的函数返回通过返回该值传送状态，或者可以返回其他状态。InstallResult 对协同安装程序是只读的。

PrivateData 指向一个被分配的协同安装程序缓冲。如果协同安装程序设置该指针并要求后处理，那么当 **SetupDiCallClassInstaller** 为后处理调用协同安装程序时，将该指针传给这个协同安装程序。

设备协同安装程序返回以下一个值：

- **NO_ERROR**

协同安装程序对特定的 *InstallFunction* 执行恰当的动作，或协同安装程序决定它无需为该请求执行任何操作。

- **ERROR_DI_POSTPROCESSING_REQUIRED**

协同安装程序对特定的 *InstallFunction* 执行任何恰当的操作，同时在类安装已处理了该请求之后并要求被再次调用。

- **A Win32 error**

协同安装程序遇到一个错误。

协同安装程序不会设置 ERROR_DI_DO_DEFAULT 的返回状态。这个状态只能由类安装程序使用。如果一个协同安装程序返回了这样的状态，那么 **SetupDiCallClassInstaller** 将不能正确地处理 DIF_Xxx 请求。协同安装程序也可能在后处理传送中*传输* ERROR_DE_DO_DEFAULT 的一个返回状态，但是它永远不会*设置*这个值。

4.3 协同安装程序操作

协同安装程序是用户模式的 Win32 DLL，它为注册编写额外的配置信息或执行要求动态信息的其他安装任务，而该动态信息无法通过编写 INF 来得到。

协同安装程序可以完成以下一些或所有的任务：

- 打开 InstallFunction 来处理仅一个或少量的 DIF_Xxx 请求。
- 根据它是否被后处理调用来执行不同的操作（也就是 **Contex->PostProcessing** 是否为 TRUE?）
- 当为后处理调用时，检查 **Context->InstallResult**。如果它不是 NO_ERROR，就进行任何必需的清除并返回 **InstallResult**。

协同安装程序必须不能给最终用户显示任何的 UI。协同安装程序应该为设备提供恰当的缺省值。如果它没有缺省值并要求用户输入，那么其他的设备支持组件就应提示用户稍后的所需输入。例如，若调制解调器没有正确的拨号属性设置，则需在用户使用调制解调器而不是在设备设置时提示他们。

4.3.1 处理 DIF 代码

每个 DIF 代码的参考页都继续了以下一些或全部的部分：

何时发送

描述 Setup 应用程序发送该 DIF 请求的典型时间及原因。

由谁处理

指出允许哪些安装程序处理该请求。该安装程序包括了类安装程序、类协同安装程序（设置-类范围的协同安装程序），以及设备协同安装程序（特定设备协同安装程序）。

输入

SetupDiCallClassInstaller 通过在它的主入口点调用安装程序给一个安装程序发送一个 DIF 请求。除了 DIF 代码之外，这个功能提供与某请求相关的额外信息。参见每个 DIF 代码的参考页可得到与每个请求一起提供的信息细节。以下列表包括了额外输入的一般描述，还列出了用于处理参数的 **SetupDiX**xx 函数：

- **DeviceInfoSet**

为设备信息集提供一个标识值。

该标识值是不透明的。利用这个标识值，例如，在调用中将设备信息集标识到 **SetupDiX**xx 函数。

DeviceInfoSet 可能具有相联的设备设置类。如果是这样的，则调用 **SetupDiGetDeviceInfoListClass** 以得到类 GUID。

- **DeviceInfoData**

有选择性地给一个 SP_DEVINFO_DATA 结构提供一个指针，该结构在设备信息集中标识了一个设备。

- Device Installation Parameters

这些非直接的参数为 SP_DEVINSTALL_PARAMS 结构中的设备安装提供了信息。如果 DeviceInfoData 是非 NULL，就有与 DeviceInfoData 相关的设备安装参数。如果 DeviceInfoData 为 NULL，则设备安装参数就与 DeviceInfoSet 相关的设备安装参数。

调用 **SetupDiGetDeviceInstallParams** 以得到设备安装参数。

- Class Installation Parameters

将可选的非直接参数指定给某个 DIF 请求。它们尤其是“DIF 请求参数”。例如，一个 DIF_REMOVE 安装请求的类安装参数被包含在一个 SP_REMOVEDEVICE_PARAMS 结构中。

每个 SP_XXX_PARAMS 结构开始于一个固定大小的 SP_CLASSINSTALL-HEADER。

调用 **SetupDiGetClassInstallParams** 以得到类安装参数。

如果 DIF 请求具有类安装参数，就有与 *DeviceInfoSet* 相关的参数集，及与 DeviceInfoData 相关的另一个参数集（如果 DIF 请求指定了 DeviceInfoData）。SetupDiGetClassInstallParams 返回了可得到的最特定参数。

- **上下文 （Context）**

协同安装程序具有一个可选的上下文参数。

- **输出 （Output）**

描述这个 DIF 代码所需的输出。

如果安装程序修改了设备安装参数，那么在返回之前安装程序必须调用 **SetupDiSetDeviceInstallParams** 来应用改变。类似地，如果安装程序修改 DIF 代码的类安装参数，安装程序必须调用 **SetupDiSetClassInstallParams**。

- **返回值 (Return Value)**

指定 DIF 代码的恰当返回值。参见图 4-3 中有关返回值的更多信息。

- **缺省处理程序 (Default Handler)**

指定 **SetupDi** 函数，它执行 DIF 代码的系统定义操作。并非所有的 DIF 代码都具有缺省处理程序。除非协同安装程序或类安装程序采取步骤阻碍调用缺省处理程序，**SetupDiCallClassInstaller** 才会在它调用类安装程序之后再调用 DIF 代码的缺省处理程序（但却是在它调用任何为后处理注册的协同安装程序之前）。

操作 (Operation)

描述安装程序可能用来处理 DIF 请求的典型任务。

其他 (See Also)

相关信息源的列表。

图 4-3 中是 **SetupDiCallClassInstaller** 中处理 DIF 代码的事件序列。

操作系统执行每个 DIF 代码的一些操作。由供应商提供的协同安装程序及类安装程序可以参

与安装行为。请注意即使 DIF 代码失败了，**SetupDiCallClassInstaller** 也调用了为后处理注册的协同安装程序。

4.4 注册协同安装程序

协同安装程序可以为某个设置类的单个或全部设备注册。当特定设备中的一个已被安装时，这些设备的协同安装程序就通过 INF 文件动态注册。类协同安装程序被手工注册或由定制的设置应用程序及一个 INF 注册。

如要了解更多的信息，可参见《Registering a Device-Specific Coinstaller》及《Registering a Class Coinstaller》。

如要更新协同安装程序，DLL 的每个新版本都需有一个新的文件名，这是因为当用户在设备属性页上点击 Update Driver 按纽时，尤其要用到 DLL。

图 4-3　在 SetupDiCallClassInstaller 中的 DIF 代码处理流程图

4.4.1 注册设备专用的协同安装程序

为了注册设备专用的协同安装程序，将以下部分添加到设备的 INF 文件中：

```
;     :
;     :
[DestinationDirs]
XxxCopyFilesSection = 11                    \\DIRID_SYSTEM
                                            \\    Xxx = driver or dev .    prefix
;       :
;       :
[XxxInstall . OS-platporm.CoInstallers]              \\    OS-platform is optional
CopyFiles = XxxCopyFilesSection
AddReg = Xxx.OS-platform. CoInstallers_AddReg

[XxxCopyFilesSection]
XxxCoInstall.dll

{Xxx. OS-platform.CoInstallers_AddReg}
HKR,,CoInstallers32.0x00010000.”XxxCoInstall.dll. \
      XxxCoInstallEntryPoint”
```

DestinationDirs 部分中的项说明 XxxCopyFiles 部分中列出的文件将被复制到系统目录下。Xxx 前缀标识出驱动程序、设备或设备组（如 cdrom_CopyFilesSection）。Xxx 前缀应是唯一的。

协同安装程序安装节的名称可以用可选的操作系统/架构扩展（如

cdrom_install.NTx86.CoInstallers）来修饰。

Xxx_**AddReg** 部分中的项在设备驱动程序密钥中建立一个 **CoInstallers32** 值项。该项包含了协同安装程序 DLL，而且可选地还可有一个特定入口点。如果忽略这个输入点，则缺省为 CoDeviceInstall。十六进制标志参数（0x00010000）指明这是 REG_MULTI_SZ 值项。

为了给设备注册多于一个的特定设备协同安装程序，复制每个协同安装程序的文件并在注册项中包含至少一个信息串。例如，为了注册两个协同安装程序，建立如下部分：

```
;    :
;    :
[DestinationDirs]
XxxCopyFilesSection = 11                    \\DIRID_SYSTEM
                                            \\   Xxx = driver or dev .   prefix
;      :
;      :
[XxxInstall . OS-platporm.CoInstallers]              \\   OS-platform is optional
CopyFiles = XxxCopyFilesSection
AddReg = Xxx.OS-platform. CoInstallers_AddReg

[XxxCopyFilesSection]
XxxCoInstall.dll                              \\   copy 1st coinst. file
 YyyCoinstall.dll                             \\   copy 2nd coinst. file

[Xxx. OS-platform.CoInstallers_AddReg]
HKR..CoInstallers32.0x00010000.                          \
     "XxxCoInstall.dll.    XxxCoInstallEntryPoint".        \
     "YyyCoInstall.dll.    YyyCoInstallEntryPoint"
                                              \\   add both to registry
```

当执行设备专用的协同安装程序 INF 部分时，该协同安装程序是在安装一个设备的过程中被注册的。接着 Setup API 在安装过程中的每个随后步骤上调用协同安装程序。如果为一个设备注册多个协同安装程序，那么 Setup API 按其在注册中所列顺序调用它们。

4.4.2 注册类协同安装程序

如要为某个设置类的每个设备都注册一个协同安装程序，可按以下所列建立一个注册表项 **HKLM\System\CurrentControlSet\Control\CoDeviceInstallers** subkey:

{setup-class-GUID}: REG_MULTI_SZ : "XyzCoInstall.dll. XyzCoInstallEntryPoint\0\0"

该系统建立了 **CoDeviceInstallers** 密钥。Setup-class-GUID 为设备设置类指定 GUID。如果协同安装程序提供设备的多个类，它就建立每个设置类的单个值项。

我们不能覆盖先前写给 setup-class-GUID 密钥的其他协同安装程序。读取这个密钥，将自己的协同安装程序信息串附加到 REG_MULTI_SZ 列表中，并将该密钥写回到注册表中。

如果忽略 CoInstallEntryPoint，则缺省为 CoDeviceInstall。

协同安装程序 DLL 必须也被复制到系统目录下。

一旦复制了文件且做出了注册表项，类协同安装程序就可被用来调用相关设备和服务。

不用手工建立注册项来注册一个类协同安装程序，就可以利用 INF 文件注册它，如以下所示。

```
[version]
signature = “$Windows NT$”

 [DestinationDirs]
 DefaultDestDir = 11      / / DIRID_SYSTEM

 [DefaultInstall]
 CopyFiles = @classXcoinst.dll
 AddReg = CoInstaller_AddReg

 [CoInstaller_AddReg]
 HKLM.System\CurrentControlSet\Control\CoDeviceInstallers, \
    {setup-class-GUID}, 0x00010008, “classXcoinst.dll,classXCoInstaller’
 ; above line uses the line continuation character (\)
```

这个例子 INF 将文件 classXcoinst.dll 复制到系统目录下并在 **CoDeviceInstallers** 密钥下建立了一个 setup-class-GUID 类的项。Xxx-AddReg 部分的项指示两个标志：”00010000”标志表示这个项是 REG_MULTI_SZ，而”00000008”标志表示新值将被附加到任何已有的值上（如果新值并未存在于信息串中）。

这样一个注册表类协同安装程序的 INF 可由右点击安装或通过 **SetupInstallFromInfSection** 应用程序激活。

第5章　编写类安装程序

类安装程序 DLL 执行在某个设备设置类中应用设备的操作。例如，端口类安装程序负责将 COM 端口名赋给端口设置类中的一个设备。如果某个设置类中的设备不要求任何特别的安装操作，那么对于该设置类也不要求类安装程序。

微软提供系统定义的 *device setup classes*（它要求类安装程序）类安装程序。IHV 或 OEM 可以提供一个协同安装程序来执行任何所需的设置操作（参见第 4 章“编写协同安装程序”）。

如果为自己设备定义了一个新的设备设置类，那么可能会需要写出一个类安装程序。但是，很少需要定义一个新的设备设置类，这是因为几乎所有的设备都可以与系统定义的 *device setup classes* 中的一个相关。例如，数字相机的制造者可能会认为它需要一个新的设置类，但是相机却是属于图象类的。

本章包含以下内容：

- 5.1 类安装程序界面
- 5.2 设备安装函数小结
- 5.3 注册类安装程序
- 5.4 在类安装程序的 INF 文件中加强文件打开的安全性

大家还可参见“编写协同安装程序”一章以了解处理 DIF 代码的一般信息。

5.1 类安装程序界面

每个类安装程序都输出一个类安装程序函数。该函数的缺省名为 **ClassInstall**。不论何时当 Setup API 需要类安装程序来执行它的设置类设备的特定安装任务，它都会调用这个函数。

安装函数的原型如下：

```
typedef DWORD   (CALLBACK* CLASS_INSTALL_PROC)   (
    IN DI_FUNCTION          InstallFunction,
    IN HDEVINFO             DeviceInfoSet,
    IN PSP_DEVINFO_DATA     DeviceInfoData    /* optional */
```

类安装函数接受两个或三个自变量：*InstallFunction*，它标识出要执行的安装请求；另一个是设备信息集的标识值；还有一个可选自变量，即指向 SP_DEVINFO_DATA 结构的指针，该结构包含了有关设备安装的信息。

类协同安装程序返回以下的一个值：

- **ERROR_DI_DO_DEFAULT**

典型地，类安装程序返回 ERROR_DI_DO_DEFAULT，它构建出设置来执行 DIF 请求的缺省动作。

这个返回状态并不真是一个错误，而就是一个有效的返回值。ERROR_DI_DO_DEFAULT 表明类安装程序有可能执行了一些过滤或增大安装的动作，但是类安装程序的动作并不能取代缺省动作。

例如，类安装程序可能设置一些标志对应于 DIF_INSTALLDEVICE 请求并返回 ERROR_DI_DO_DEFAULT。对应于这个返回值，设置执行了 DIF_INSTALLDEVICE 的缺省动作(调用 **SetupDiInstallDevice**)之后再调用要求后处理的任何协同安装程序。

如果已知了所给的 DIF 要求并没有缺省动作，那么就应该仍是返回 ERROR_DI_DO_DEFAULT。DIF 请求可能会在今后发行的操作系统中具有缺省动作。

- **A Win32 error**

  类安装程序可以返回一个恰当的 Win32 错误。

- **NO_ERROR**

  当类安装程序决定它已完整地完成了设备安装任务，它就仅返回 NO_ERROR，该任务包括了调用任何缺省处理程序或代替缺省操作。

在诸如下面的情况下，类安装程序可能会调用缺省处理程序。如果缺省处理程序已运行了，安装程序就需要执行某个动作。如果类安装程序调用缺省处理程序，则安装程序必须不返回 ERROR_DI_DO_DEFAULT；它必须返回 NO_ERROR 或 Win32 错误。

5.2 设备安装函数小结

安装程序可以使用由 Setup API 提供的 *device installation functions* 来执行安装操作。设备安装函数允许安装程序搜索 INF 文件兼容的驱动程序，通过选择对话框向用户显示驱动程序选择，并执行真正的驱动程序安装。

大多数的设备安装函数依赖于 SP_DEVINFO_DATA 结构中的信息来执行安装任务。每个设备都具有一个与之相关的 SP_DEVINFO_DATA 结构。我们可以重新得到设备信息集的一个句柄(HDEVINFO)，该信息集通过 **SetupDiGetClassDevs** 函数将所有被安装的设备包含在一个特定的类中。我们可以利用 **SetupDiCreateDeviceInfo** 函数将新的设备添加到设备信息集中。还可以通过 **SetupDiDestroyDeviceInfoList** 函数将所有的 SP_DEVINFO_DATA 结构都释放到设备信息集中。这个函数还释放任意的兼容设备及可能已被添加到结构中的类设备列表。

通过使用 **SetupDiBuildDriverInfoList** 函数，可以生成一个列表，安装程序或用户可从该列表中选择驱动程序或设备来安装。**SetupDiBuildDriverInfoList** 建立一个兼容驱动程序列表或特定类的所有设备列表。

一旦有了这样的一个兼容驱动程序的列表之后，就可提示用户利用 **SetupDiSelectDevice** 函数从该列表中进行选择。这个函数显示一个包括了设备信息集中每个设备信息的对话框。我们可以通过使用 **SetupDiInstallDevice** 函数安装选定的驱动程序。这个函数使用驱动程序的 INF 文件信息来创建任何所需的新注册表项，设置设备硬件的配置，并将驱动程序文件复制到恰当的目录下。

安装程序可能需要检查，并在注册密钥下设置将要安装设备的值。我们可以利用 **SetupDiCreateDevRegKey** 或 **SetupDiOpenDevRegKey** 函数打开设备的硬件或驱动程序密钥。

可以通过 **SetupDiInstallClass** 函数安装设备的新类。这个函数从一个包含了 **ClassInstall32** 部分的 INF 文件安装设备的新类。

可以通过 **SetupDiRemoveDevice** 函数从系统中删除设备。这个函数删除设备注册项，如果可能还停止该设备的运行。如果不能动态停止设备，那么函数就设置提示最终导致用户关闭系统的标志。

5.3 注册类安装程序

通过添加 **Installer32** 和 **Icon** 项到类注册中可以注册类安装程序。微软的 Windows 2000 在 **HKEY_LOCAL_MACHINE** 下维持一个有关每个类信息的 **System\CurrentControlSet\Control\Class** 分支。(Windows9x 在 **HKEY_LOCAL_MACHINE** 下维持一个有关每个类信息的 **System\CurrentControlSet\Services\Class** 分支)。该密钥的值包含了类的本地描述。我们可以在一个由 *INF ClassInstall32* 部分引用的 **AddReg** 部分中做出恰当的类安装注册项。

Installer32 和 **Icon** 项具有以下形式：

Installer32=安装程序-DLL-名[,安装程序入口点]

Icon=序号

系统使用图标表示用户的安装程序并调用它来在类中安装设备。如果 **Icon** 的值为负，它的绝对值代表类注册中的一个预定义的图标。如果 **Icon** 值为正，它表示将被抽取出的可执行的类安装程序中的图标。值 1 被保留。参见《*Windows 2000 Driver Development Reference*》卷一中的 **SetupDiDrawMiniIcon** 来查看预定义的小图标。

5.4 在类安装程序的 INF 文件中加强文件打开的安全性

对于 Windows 2000，可以通过向类安装程序 INF 添加新的项来加强设备的一个类文件打开安全性。新项指定类中设备新的 FILE_DEVICE_SECURE_OPEN 特性。

新项出现在由 **ClassInstall32** 部分引用的 **AddReg** 部分中并有以下的句法：

```
[ClassInstall32]  |  [ClassInstall32.ntx86]
…
AddReg = Xxx_AddReg
…

[Xxx_AddReg]
…
HKR,, DeviceCharacteristics,0x10001 , characteristics     ; new
```

characteristics 值是赋给设备特性的数字值。该值是一个或多个 FILE_*文件特性被定义到 wdm.h 或 ntddk.h 中的 Oring 的结果。与安全性相关的重要新特性是 FILE_DEVICE_SECURE_OPEN，其数字值是 0x100。该特性的设置将 I/O Manager 直接用于执行设置类的设备的所有打开请求的安全性检查。FILE_DEVICE_SECURE_OPEN 特性被定义在 *ntddk.h* 和 *wdm.h* 中，且仅被 Windows 2000 平台支持。

DeviceCharacteristics 是 HKR **AddReg** 项的特殊关键字。设置将特定的设备特性储存到注册表中的个人站点，它是从由 Setup 建立并由其他 HKR **AddReg** 项指定的用户定义的注册项中分离出来的。

如果合适的话，通过指定它的设备 INF 文件中的”HKR,,DeviceCharacteristics,…”项，类中的单个设备可以覆盖由类安装程序设置的特性。参见 Plug and Play、电源管理及设置引用中的 INF AddReg Directive 部分以了解更多的内容。

第6章　编写定制设备安装应用程序

定制设备安装应用程序应调用 **UpdatdDriverForPlugAndPlayDevices** 函数来安装新设备而不是自己发送所有必要的 DIF 代码。若已给出了一个 INF 文件和硬件 ID，这个函数就在机器上扫描设备，并尝试着将 *FullInfPath* 上指定的设备安装到与所给的匹配的任何设备上。如果设置应用程序调用 **UpdeateDriverForPlugAndPlayDevices** 而不是发送 DIF 代码，则该应用程序则更有可能是向上兼容的。例如，如果今后发行的操作系统对设备安装序列作了改变，那么微软也会修改 **UpdateDriverForPlugAndPlayDevices** 来发送恰当的 DIF 代码。

若由于某种原因，定制设备安装应用程序不能调用 **UpdateDriverForPlugAndPlayDevices**，这样的应用程序就应通过调用 **SetupDiCallClassInstaller** 来发送 DIF 代码。这个函数保证了调用任何的协同安装程序、类安装程序及 **SetupDiX**xx 缺省设备驱动程序且顺序是是恰当的。

6.1　安装与驱动程序共用的软件实用程序

你的定制设备安装应用程序应该处理以下两种情况：

- 用户插接进硬件中并插入到分配媒介
- 用户插入到分配媒介以安装设备支持并插接进硬件

如要解决第一种情况，就应提供一个调用 **UpdateDriverForPlugAndPlayDevices** 的自动运行的设备程序，并运行你的设备实用程序或应用程序的设置程序。

如要解决第二种情况，则按以下步骤设置程序：

1. 在目标系统中，给驱动程序文件建立一个目录。如果设置程序安装了一个应用程序，驱动程序文件就应被储存在应用程序目录下的子目录中。
2. 由分配媒介向步骤 1 中的目录复制驱动程序包中的所有的文件。驱动程序包包括了驱动程序、INF 文件、目录文件等。
3. 调用步骤 1 创建的目录中的 INF 文件的 **SetupCopyOEMInf**。指定 OEMSourceMediaType 的 SPOST_PATH 参数，并将 OEMSourceMediaLocation 参数指定为 NULL。**SetupCopyOEMInf** 复制驱动程序包的 INF 文件到目标系统的%windir%\Inf 目录下，并指示 Setup API 将 INF 文件的源位置储存到它的预处理 INF 文件列表中。**SetupCopyOEMInf** 还处理目录文件，所以 PnP Manager 将会在它下一次识别 INF 文件列出的设备时安装驱动程序。

当用户插接到设备中时，PnP Manager 识别该设备，找出由 **SetupCopyOEMInf** 复制的 INF 文件，并安装步骤 2 所复制的驱动程序。

6.2　定制安装应用程序指导

定制安装应用程序*必须*进行以下操作：

- 支持应用程序卸载。作为卸载的一部分，应用程序的设置程序应查看是否有相关的设备仍出现在系统中，如果有，则向用户发出警告。
- 列出 Add/Remove Programs 中所有已安装的应用程序，这样用户如果需要就可以卸载了。

- 按照微软 Windows 2000 应用程序的指导。如需了解更多的信息，可参看 MSDN™ 站点 (http://msdn.microsoft.com)。
- 将软件应用程序提交给恰当的软件 Windows Logo 程序。参见 MSDN 站点以了解更多信息。

定制安装应用程序*必须不*进行以下操作：

- 指导用户复制或覆盖任何文件，尤其是.inf 和.sys 文件。
- 即使是硬件已被删除，在卸载操作中从系统中删除已安装的驱动程序。
- 使用 RunOnce Registry 表项来生成设置程序，该程序要求强制的重启动。
- 强制任何不必要的系统重启。**UpdateDriverForPlugAndPlayDevices** 函数通过它的 bRebootRequired 参数表明重启的需要。重启在安装 PnP 设备或软件应用程序时一般是不需要的。
- 使用设备或类协同安装程序来生成应用程序设置程序，这是因为在设备安装时不能保证系统状态对于软件应用程序安装是安全的。尤其是如果设备安装及设置程序是在服务器一方运行，系统将会被挂起。
- 由于类似的原因，使用类安装程序来引起应用程序的设置程序。
- 使用 Startup Group 来生成设置程序。
- 使用 win.ini 项生成设置程序。
- 除了在以下情况下，强制用户安装任何应用程序：
  - 如果 Windows 不能给内置的应用程序提供使用设备标准功能的所需功能，例如，如果 Windows 不为一个实用程序提供设置可配置的键盘密钥。
  - 如果应用程序被特别要求使用这个特定设备-----例如，如果要求一个实用程序为调制解调器设置国家代码。

第7章　提供设备属性页

如果设备或类具有任何用户可以设置的单个属性，那么协同安装程序或类安装程序应该提供一个定制设备属性页。这样的属性可能会包括 CD-ROM 驱动器的缺省回放卷或调制解调器的麦克风卷。当用户在设备的 Properties 上点击时，Device Manager 都会显示属性页。在 Microsoft Windows NT 的早先版本中，用户通过 Control Panel applets 设置这样的信息。相反地，Windows 2000 驱动程序应提供属性页。

尽管你也可以写出提供定制属性页的类安装程序，一般多选的是提供协同安装程序的这个机能，以及其他设备或设备-类-特定特征。

Platform SDK 文件提供属性页的复合文档以及操作它们的 Win32 函数。本章还给 Platform SDK 文件补充了以下与设备安装特定相关的信息：

- 7.1 设备属性页所需的支持
- 7.2 处理 DIF_ADDPROPERTYPAGE_ADVANCED 请求
- 7.3 属性页的回调函数
- 7.4 处理属性页的 Windows 信息

7.1 设备属性页所需的支持

定制的属性页提供者必须做到以下几点：

- 处理 DIF_ADDPROPERTYPAGE_ADVANCED 请求
- 如果必须给属性页分配及释放额外的存储量，就要提供处理 PSPCB_CREATE 和 PSPCB_RELEASE 信息的回调。
- 提供处理每个定制属性页的 Windows 信息的对话框过程

图 7-1 是定制属性页的一个样本。

图 7-1　定制属性页的一个示例

在图中，一个安装程序提供名称为 Properties 的属性页。缺省地，系统如图提供 General 和 Driver 书签。在合适时，系统也提供 Resources 和 Power 书签。

设置组件可以为它的设备或类定义超过一个的属性页。每个书签都应有一个该页目的的简洁描述。系统并不检查来确保名称是唯一的。

7.2 处理 DIF_ADDPROPERTYPAGE_ADVANCED

提供一个或多个定制属性页的安装程序必须处理 DIF_ADDPROPERTYPAGE_ADVANCED 设备安装函数代码。当用户在 Device Manager 或在 Control Panel 应用程序中的设备 Properties 上点击时，Device Manager 通过 Setupapi.dll 发出这个请求。

为了响应这个请求，安装程序提供其每个定制属性页的信息、创建页并将创建的页添加到设备动态属性页列表中。

安装程序应采取以下步骤：

1. 调用 **SetupDiGetClassInstallParams** 以得到设备的当前类安装参数。例如：

```
SP_ADDPROPERTYPAGE_DATAAddPropertyPageData;
:
ZeroMemory (&AddPropertyPageData, sizeof( SP_ADDPROPERTYPAGE_DATA) );
AddPropertyPageData. ClassInstallHeader.cbSize =
        sizeof(SP_CLASSINSTALL_HEADER);
  if  (SetupDiGetClassInstallParams(DeviceInfoSet, DeviceInfoData,
        (PSP_CLASSINSTALL_HEADER)&AddPropertyPageData,
        sizeof(SP-ADDPROPERTYPAGE_DATA),  NULL  ) )  {
```

示例零初始化结构，其中的类安装头参数将被返回，并设定 **SetupDiGetClassInstallParams** 要求 **cbSize** 字段中的类安装头的大小。类安装头是每个类安装参数结构的第一个成分。

当一个 DIF_ADDPROPERTYPAGE_ADVANCED 请求是激活的，这个函数返回一个在 *ClassInstallParams* 上的 SP_ADDPROPERTYPAGE_DATA 结构。因此，对 **SetupDi** 函数的调用可以将返回的 ClassInstallParams 放到这个类型的结构中。

2. 保证尚未遇到设备动态页的最大数字，且它还带有如下的语句：

```
if  (AddPropertyPageData.NumDynamicPages <
     MAX_INSTALLWIZARD_DYNAPAGES)
```

如果测试失败，就不要初始化或建立页了。类似地会返回 NO_ERROR。

3. 为存放任何特定设备数据分配内存，该数据将在后面的对话框过程中用到，且初始化这个区域也会使用到该数据。这个数据必须包括与请求一起传送的 DeviceInfoSet 和 DeviceInfoData。

例如，原型页供应者可以定义并使用以下的结构：

```
typedef struct TESTPROP_PAGE_DATA  {
HDEVINFO DeviceInfoSet;
PSP_DEVINFO_DATA DeviceInfoData;
} TEST_PROP_PAGE_DATA, *PTEST_PROP_PAGE_DATA;

PTEST_PROP_PAGE_DAT    pMyPropPageData;

pMyPropPageData = HeapAlloc (GetProcessHeap( ), 0,
      sizeof(TESTPROP_PAGE_DATA) );
```

4. 利用定制属性页初始化 PROPSHEETPAGE 结构：
   - 在 **dwFlags** 中，设置 PSP_USECALLBACK 标志以及定制页所要求的任意其他标志。PSP_USECALLBACK 标志表明回调函数已被提供。
   - 在 **pfnCallback** 中，设置指向属性页的回调函数的指针。
   - 在 **pfnDlgProc** 中，设置指向属性页的对话框过程的指针。
   - 在 **lParam** 中，传送一个分配给内存区域并在步骤 3 中初始化的指针。
   - 将其他成分设为与定制的属性页相符。参见 Platform SDK 文档以了解有关 PROPSHEETPAGE 结构的更多内容。
5. 调用 **CreatePropertySheetPage** 以建立新页。
6. 将新页添加到类安装参数的 **DynamicPages** 元素中的动态属性页列表中并添加

NumDynamicPages 成分。

7. 对于每个额外的定制属性页，重复步骤 2 到步骤 6。
8. 调用 **SetupDiSetClassInstallParams** 以设置新的类安装参数，该参数包括了更新的属性页结构。
9. 返回 NO_ERROR。

设置增加新建立的属性页到设备属性表，且 Device Manager 使 Win32 API 调用来建立该表。当显示属性页时，系统调用在 PROPSHEETPAGE 结构中指定的对话框过程。

7.3 属性页回调函数

当安装程序设置它的设备或设备类的属性页时，它提供了一个回调函数的指针。当建立属性页时，回调函数被调用一次，当属性页被破坏时则再被调用一次。

回调是 **PropSheetPageProc** 函数，它的描述在 Platform SDK 文档中，它须能处理 PSPCB_CREATE 和 PSPCB_RELEASE 动作。

在建立属性页时，回调是同 PSPCB_CREATE 信息一起被调用的。与这个信息对应，回调可以分配与页相关的数据内存。函数应返回 TRUE 来继续建立页，或由于某个原因，该页不能被建立时，则返回 FALSE。

当用户点击页上的对话框中的 OK 或 Cancel 时，或是点击 Drivers 书签上的卸载时，设备的属性页会被破坏。

当属性页正被破坏时，回调会与一个 PSPCB_RELEASE 信息一起被调用。该函数应释放建立属性页时被分配的所有数据。典型地，这涉及到 PROPSHEETPAGE 结构中 **IParam** 上的数据。当页正被破坏时，返回值被忽略。

7.4 处理属性页的 Windows 信息

当提供属性页的安装程序处理一个 DIF_ADDPROPERTYPAGE_ADVANCED 请求时，它设置了属性页的对话框过程的地址。当对话框过程得到一个 WM_INITDIALOG 信息时，它必须初始化属性页，而且当它得到一个 WM_MOTIFY 信息时，它还须准备处理设备属性的变化。该过程还可以处理它可能要求的任何其他这样的信息，这同 Platform SDK 文件中描述的相同。

与 WM_INITDIALOG 信息对应，对话框过程初始化属性页中的信息。这样的信息可能包括了表示设备的一个图标、设备的友好名称，以及它的 PnP 设备描述。

SetupDiLoadClassIcon 装载特定设备类的图标并返回被装载的大图标的句柄，该大图标可被用在 **SendDlgItemMessage** 的随后调用之中。例如：

```
if (SetupDiLoadClassIcon(
        &pTestPropPageData->DeviceInfoData->ClassGuid, &ClassIcon,
        NULL0 ) {
    OldIcon = (HICON)SendDlgItemMessage(hDlg, IDC_TEST_ICON,
        STM_SETICON,   (WPARAM)ClassIcon, 0 );
    if (OldIcon)   {
                DestroyIcon(OldIcon);
```

```
        }
    }
```

在 ClassIcon 中被返回的句柄可以被放置到由 **SendDlgItemMessage** 函数要求的 WPARAM 中。在示例中，IDC_TEST_ICON 表示了接收 STM_SETICON 信息的对话框中的控制。IDC_TEST_ICON 的值必须被定义在安装程序中。参见《Windows 2000 Driver Development Reference》一书第一卷第 3 部分的第 3 章“设备安装函数”了解操作图标和位图的其他 **SetupDi** 函数，并参见 Platform SDK 文档了解有关 **SendDlgItemMessage** 和 **DestroyIcon** 的有关信息，此外还可以利用对话框中的图标。

除了图标表示设备之外，典型设备属性页还可以包括设备的描述或“友好的名称”并显示设备属性的当前设置。PnP Manager 将每个设备的 PnP 属性存在注册表中。属性页供应者可以调用 **SetupDiGetDeviceRegistryProperty** 来得到任意此类属性的值。如果也将指定设备或指定类的配置信息也作为安装过程的一部分储存在注册表中，属性页供应者就可利用其他的 **SetupDi** 函数来抽取该信息显示了。参见《Windows 2000 Driver Development Reference》一书第一卷第 3 部分的第 3 章“设备安装函数”的详细内容。

当页中出现了某种类型的变化，属性表给对话框过程发送 WM_NOTIFY 信息。对话框过程应准备从信息参数中抽取通知代码并作出恰当的反应。

PSN_APPLY 通知是安装程序带来的一大好处。当用户点击 OK 或 Apply 时，属性表发送该信息。相应地，安装程序应证实该变化并在设备安装参数中设置 DI_FLAGSEX_PROPCHANGE_PENDING 标志，如下所示：

1 如果未做好它，取得设备的设备安装参数(SP_DEVINSTALL_PARAMS 结构)指针。通过调用 **SetupDiGetDeviceInstallParams** 及传送保存的 *DeviceInfoSet* 和 *DeviceInfoData*(已在 lParam 的区域上传送)可得到该结构。
2 保证用户的更改是有效的。
3 在返回的 SP_DEVINSTALL_PARAMS 结构的 **FlagsEx** 字段中设置 DI_FLAGSEX_PROPCHANGE_PENDING 标志。但是如果安装程序能够绝对保证更改不要求设备的驱动程序停止及再重新启动，也就无需设置该标志。
4 与改变了的 SP_DEVINSTALL_PARAMS 结构一起调用 **SetupDiSetDeviceInstallParams** 来设置新参数。

在安装程序设置了新参数之后，Setup 或 Device Manager 发送一个 DIF_PROPERTYCHANGE 请求。

当用户点击 Cancel 时，属性表发送一个 PSN_RESET 通知信息。与该信息相对应，对话框过程应删除由用户做出的任何更改。

参见 Platform SDK 文档中有关其他通知(可能会遇到这样的过程以及它如何处理它们)的更多内容。

第8章　设备安装疑难解答

为了证实设备是正确安装的，则应运行测试程序来操作设备并利用 Device Manager 观察设备的系统信息。

为了启动 Device Manager，在 My Computer 图标上右击，从菜单上选择 Manage，再在随后显示列出的 System Tools 中选择 Device Manager。Device Manager 显示每个设备的信息，这其中包括了设备类型、设备状态、制造商、特定设备属性及设备驱动程序信息。从 Device Manager 在线帮助可以了解更多的内容。

当测试新 PnP 设备安装时，使 Device Manager 列表隐藏在设备之后是有用的。如要了解更多的信息，可参看 Device Manager 中的 Displaying Hidden Devices。

如果你的设备是一个引导程序设备，则设备安装有可能会出现一个阻止机器引导的问题。在这种情况下，就需要利用内核调试程序对设备安装进行故障检修。如想了解更多的内容，可以参看与 Microsoft 调试程序一起提供的在线文件，但它与这个 DDK 不在一起。

如果你的设备不要求来引导机器，就特别要注意设备安装的一个问题：设备并没有正常工作而且设备被 Device Manager 中的一个黄色的警告点（黄色的感叹号）标记出来。典型的安装问题包括：

- “没有安装该设备的驱动程序。（代码 28）”或“设备未配置好。”

  这些 Device Manager 错误表明 Setup 没有找到与设备相匹配的 INF，或是虽然找到了一个 INF，但却无法成功安装设备。此时可以考虑在你的 INF 文件上运行 ChkINF 并查看 Setup API 日志。(参见*利用 SetupAPI 登录*。)
- “这个设备没有正确配置。（代码 00）”

  这个 Device Manager 错误尤指用户删除了 Found New Hardware 向导。
- “本设备由于 Windows 未能装载设备所需的驱动程序而不能正常工作。(代码 31)”

  这个 Device Manager 错误表示 Setup 找到了一个与设备匹配的 INF 也很可能找到了驱动程序，但或是由于你的 INF 没有列出所有所需复制的文件，因此设备在 **DriverEntry** 中失败。或是由于驱动程序在其 AddDevice 例程中失败。观察驱动程序项及装载到设备中的驱动程序的事件日志及 Setup API 日志。可能还需要与内核调试程序相连以调试失败的驱动程序。
- “这个设备无法找到足够多可用的自由资源。(代码 12)。如果要使用该设备，则需使本系统中另一个设备不工作。”

  这个 Device Manager 错误表示 Setup 找到了与设备匹配的 INF 并装载了恰当的驱动程序，但是它无法给设备分配所需的资源。

  用户可以单击设备 Device Manager 属性表 General 标签上的 Troubleshooter 按钮。这个故障检修应帮助用户给设备找出资源并停止无用设备的工作，这样新的设备才能运行。
- “这个设备不能启动。(代码 10)”

  这个 Device Manager 错误表示 Setup 找到一个与设备匹配的 INF，并已装载了恰当的驱动程序，分配了资源(如果有的话)，并发出 IRP_MN_START_DEVICE 请求。但是，设备的一个驱动程序无法执行启动请求。观察相关项的事件日志。可能还需要与内核调试程序相连来调试失败的驱动程序。在调试程序中，设置 IRP_MN_START_DEVICE 调度例程中的断点。
- “设备无法使用。(代码 22)”

  有这个错误的设备在 Device manager 中标为红色的”X”，而不是黄色的感叹号。这个错误表示

用户无法使用设备或 Setup 无法安装设备。

协同安装程序(或类安装程序)可以提供帮助用户对设备中的故障进行检修的故障检修程序。可参见《Windows 2000 Driver Development Reference》一书第一卷第 3 部分的第 5 章中的 DIF_TROUBLESHOOTER 了解更多知识。

8.1 使用 SetupAPI 记录日志

Windows 2000 Setup and Device Installer Services，即 SetupAPI，包括了控制 Setup 及设备安装的 Windows 函数。当 Setup 运行时，Setup 函数(**Setup**Xxx 函数)及 device 安装函数(**SetupDi**Xxx 函数)建立了一个提供对设备安装问题进行故障检修有用信息的日志文件。

在 Windows 2000 中，SetupAPI 登录到文件%systemroot%\setupapi.log。日志文件是一个纯文本文件。如要重新设置日志，对该文件重命名或删除即可。

本节包括以下内容：

- 8.1.1 设置 SetupAPI 记录日志级别
- 8.1.2 解释 SetupAPI 日志文件示例

8.1.1 设置 SetupAPI 记录日志级别

如要改变写到 SetupAPI 日志中的信息级，在以下注册密钥中建立(或修改)值即可：

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersions\Setup

通过在这个密钥下设置值，可以选择被记录的错误级别，还可修改日志的冗余性，或关掉日志。还可以将信息记录到调试程序及日志文件。

该密钥下的 SetuAPI 记录日志的重要值是 **LogLevel** 和 **AppLogLevels**。SetupAPI 对这两个值有如下的解释：

- LogLevel = LoggingFlags

这个 DWORD 值设置了为 Setup 和设备安装而被记录的错误级别。如果该值未被指定或为零，SetupAPI 使用以下缺省行为：

- 对于 Setup，记录所有的错误及一些警告。
- 对于设备安装，记录所有错误及所有警告。

LoggingFlags DWORD 由 3 个部分组成：0xGGGGDDSS、下八位——由掩模 0x000000FF 表示，设置 Setup 的记录级别、下一高八位——由掩模 0x0000FF00 表示，设置设备安装的记录级别。最高位是一般标志。

Setup 标志如下解释：

Setup 标志	**意义**
0x00000000	使用缺省设置(当前为 0x20)
0x00000001	关掉(没有 Setup 记录)
0x00000020	登录错误及警告
0x000000FF	所有项上带有时间信息的冗余登录。该设置在设备安装测试时有用

设备安装标志解释如下：

设备标志	**意义**

设备标志	意义
0x00000000	使用缺省设置(当前为 0x3000)
0x00000100	关掉(没有设备安装记录)
0x00003000	记录错误、警告及信息
0x0000FF00	所有项上带有时间信息的冗余记录。该设置在设备安装测试时有用

SetupAPI 如下解释一般标志：

一般标志	意义
0x10000000	转送所有的上下文信息(尤其是未用的)
0x80000000	将输出发送到调试程序及日志文件中

例如，SetupAPI 如下解释 LoggingFlags 例值：

- 0x00000000 意味着缺省登录
- 0x0000FFFF 意味着冗余记录
- 0x8000FF00 意味着将冗余设备安装信息记录到日志文件及调试程序中
- AppLogLevels\<appname>=LoggingFlags

如果存在，这个 DWORD 值能够记录到每个应用程序基础上。这个 *LoggingFlags* 值与前面的 **LogLevel** 值相同。例如，AppLogLevels\rundll32.exe=0xFFFF 能够对 Rundll32 冗余记录。

为了在 Windows 2000 清除安装时修改缺省的 SetupAPI 记录级，就在文本模式 Setup 和 GUI 模式 Setup 之间进行编辑。以下步骤描述了该过程。这些步骤假设正在安装 D:\Winnt 并在其他部分上有一个 Windows 2000 的工作构件。如下改变 SetupAPI 记录级：

1. 开始安装正在测试的清除构件。
2. 在文本模式 Setup 之后(即在 GUI 模式 Setup 之前)的第一次引导程序中停止设置过程。
3. 通过从启动菜单中选择启动进入工作构件来进行该操作。
4. 找到 D:\Winnt\System32\config 中的注册处(文件)。在该情况下，需要修改 Software.sav 中的注册处。
5. 运行 Regedt32，选择“Local Machine 上的 HKEY_LOCAL_MACHINE”窗口，并选择 HKEY_LOCAL_MACHINE 键。再在 Registry 菜单上单击并选择 Load Hive。
6. 浏览文件并选择 D:\Winnt\System32\config\software.sav。当提示要输入密钥名时，输入”_sw.sav.”。
7. 打开 HKEY_LOCAL_MACHINE 下的 _sw.sav 密钥并找到如下密钥：
   HKEY_LOCAL_MACHINE_sw.sav\Microsoft\Windows\CurrentVersion\Setup
   打开 Security 下拉菜单，选择 Permissions，并对所登录进的账号赋于全部的控制权（例如 Administrator）。
8. 利用 Edit->AddValue 在这个密钥下添加必要的登录值。例如，添加“LogLevel=0xFFFF”以进行完全冗余登录。
9. 选择 HKEY_LOCAL_MACHINE_sw.sav，并从菜单中选择 Registry->Unload Hive (_sw.sav 应消失)。
10. 将 D:\Winnt\System32\config\software.sav 复制到 D:\Winnt\System32\config\soft
    ware 中。
11. 重启并接着再进行 Setup。
12. 为了证实该变化，在 GUI 模式 Setup 中按下 SHIFT+F10，运行 Regedt32，并检查登录级。

8.1.2 解释 SetupAPI 日志文件示例

以下的日志样本解释了包含在 SetupAPI 日志文件中的信息。

一般地，安装的所有部分都会一起显示在日志文件中。日志中的安装部分开始格式的一个项[year/month/day time process-id.instance description]，其中 instance 是保证两个部分被同时安装的一个数，这是因为相同的过程是唯一的。

在以下的日志文件样本中，缩进排列的部分是注释。日志样本是由冗余输出的 **LogLevel**=0xFFFF 生成的：

[1999/10/15 15:14:25 376.1560 Driver Install]

SetupAPI lists the hardware compatible Ids of the device being installed.The first ID listed is the most exact match.Lines without a time stamp provide context for time-stamped entries.

Searching for hardware ID(s):acpi\pnp0200,*pnp0200

Enumerating files E:\WINNT\inf*.inf

@15:14:25.637: Opened PNF of “E:\WINNT\inf\1394.inf”.

PNFs are preprocessed INF files.

:

Searching other INFs…

:

@15:14:26.688: Opened PNF of “E:\WINNT\inf\machine.inf”.

Setup found a match here,so it logs the INF in which it found the match,what PNP id it matched on,descriptive information.,and the install section.

@15:14:26.698: Found *PNP0200 in E:\WINNT\inf\machine.inf; Device: Direct memory access controller ; Driver: Direct memory access controller;Provider: Microsoft; Mfg: (Standard system devices); Section :NO_DRV_X

@15:14:26.698: Decorated section name: NO_DRV_X

:

Setup searches other INFs

:

@15:14:31.575: Opened PNF of “E:\WINNT\inf\wordpad.inf”.

Setup has finished searching INFs.Next, it loads class coinstaller and class installers that are registered for this device’s setup class.

Loading co-installer modules for Direct memory access controller.

@15:14:31.615: Verifying file: E:\WINNT\System32\SysSetup.dll using key: SysSetup.Dll in catalog: -.

@15:14:31.695: Obtained module “ E:\WINNT\System32\SysSetup.dll”procedure “CriticalDeviceCoInstaller” for use.

:

Setup performs various device installation oprations before it sends the DIF_INSTALLDEVICEFILES request.

:

@15:14:32.026: Device install function : DIF_INSTALLDEVICEFILES .

Setup logs each device installation function code .

CoInstallers are numbered in the order that they are loaded.

@15:14:32.026: Executing co-installer 1 of 1.

@15:14:32.036: Completed co-installer 1 of 1.

In this example ,there is no class installer for this device's setup class.You can tell ths because there is no entry for "Executing class-installer."

@15:14:32.046: Completed class-installer.

Now Setup calls the default handler for this DIF request . If "Executing default installer" is listed before the class installer completes, that means the class installer called the default handler function directly.

@15:14:32.046: Executing default installer.

@15:14:32.056: Install Device : Begin.

Setup lists the devnode here.

@15:14:32.066: Doing copy-only install of ACPI\PNP0200\4&12BA42B2&0.

Setup lists the INF and decorated INF section being used.

@15:14:32.136: Installing section NO_DRV_X from e:\winnt\inf\machine.inf.

@15:14:32.146: Install Device: Queueing files from INF(s).

@15:14:32.156: Install Device: Queueing co-installer files from INF(s).

Processing coinstaller registeration section NO_DRV_X.CoInstallers.

@15:14:32.236: Verifying file: E:\WINNT\Certcals.inf using key: certclas.inf in catalog:-.

@15:14:32.346: Co-Installer Registered.

@15:14:32.356: Installing section NO_DRV_X.Interface from e:\winnt\inf\machine.inf.

@15:14:32.366: Interface installed.

@15:14:32.376: Device install finished successfully(ACPI\PNP0200\4&12BA42B2&0)

@15:14:32.386: Install Device : end.

@15:14:32.396: Completed default installer.

The install pass for copying files has finished successfully. Now carry on with the rest of the install operation for this device.

@15:14:32.457: Verifying file: e:\winnt\inf\machine.inf using key: machine.inf in catalog:-.

@15:14:32.497: Verifying file: e:\winnt\inf\layout.inf using key: layout.inf in catalog:-.

@15:14:32.557: Pruning Files: Verifying catalogs/infs.

@15:14:32.567: Pruning Files: Verifying catalogs/infs completed.

:

Setup performs various other device installation functions preceeding DIF_INSTALLDEVICE.

:

@15:14:33.498: Device install function: DIF_INSTALLDEVICE.

@15:14:33.498: Executing co-installer 1 of 1.

@15:14:33.518: Completed co-installer 1 of 1.

@15:14:33.528: Completed class-installer.

@15:14:33.538: Executing default installer.

@15:14:33.538: Install Device : Begin.

@15:14:33.548: Doing full install of ACPI\PNP0200\4&12BA42B2&0.
@15:14:33.628: Install Device: Changing registery settings specified by INF(s).
@15:14:33.638: Install Device: Writing driver specific registery settings.
@15:14:33.668: Install Device: Installing required windows services.
@15:14:33.678: Install Device: Writing drive descriptive registery settings.
@15:14:33.708: Install Device: Restarting Device.
@15:14:33.768: Install Device: Restarting Device completed.
@15:14:33.788: Device install finished successfully(ACPI\PNP0200\4&12BA42B2&0)
@15:14:33.798: Install Device : end.
@15:14:33.798: Completed default installer.
@15:14:33.808: Executing co-installer (Post Processing).
@15:14:33.838: Completed co-installer 1 (Post Processing).
:
Setup performs various device installation functions after DIF_INSTALLDEVICE.
:

如要了解 DIF 代码的更多信息，参见《Windows 2000 Driver Development Reference》一书第 1 卷第 5 章“设备安装函数代码”。

8.2 显示 Device Manager 中的隐藏设备

Device Manager 列出了安装在机器上的设备。缺省地，某些设备并未出现在列表中。这些“隐藏”的设备包括了 devnode 状态位 DN_NO_SHOW_IN_DM 设置的设备及作为设置类一部分的设备，该设置类在注册表中标为 **NoDisplayClass** (例如，打印机和非 PnP 驱动程序)。

如要在 Device manager 显示中包括了隐藏设备，单击 View 并选择 Show Hidden Devices。

并没有缺省地显示在 Device Manager 中的设备的其他目录包括了从机器中被物理删除的设备，但是机器的注册表项并未被删除。这些设备可被认为是“不存在的”设备。用户无需看到这些设备，这是因为不存在的设备不需要被注意到且也不会带来什么问题。如果一个用户需要查看并未显示的设备，那么对于驱动程序的设计可能会有所影响。但在测试中，却有可能需要查看这些设备。为了看到未显示的设备，输入以下的 MS-DOS 命令：“set DEVMGR_SHOW_NONPRESENT_DEVICES=1.”(还可以利用系统属性表中的 Advanced 标签将该值设置到永久环境值中。)在看到这个值后，运行 Device Manager 并选择 Show Hidden Devices。

第9章 安装一个要求重新启动机器的设备

在 Windows 2000 安装的文本模式设置阶段中，Windows 2000 Setup 程序为要求重新启动机器的设备安装了驱动程序。这些驱动程序大多数都包含在操作系统中。用户可以在文本模式设置时通过按恰当的键及插入由第三方供应商提供的软盘来安装另一个驱动程序。第三方供应商提供放在软盘上的 *txtsetup.oem* 文件以保证用户能在文本模式设置时安装这样的驱动程序。

为了安装一个要求重新启动机器的设备，用户就要安装设备硬件、打开机器，并(如果未被包括在操作系统中)安装设备的驱动程序。当机器正在引导时，用户在设置装载阶段开始时按 F*n* 键来安装另一个设备的驱动程序。在这个阶段中，系统显示与“设置正在检查计算机的硬件配置。”类似的信息。如是海量储存器，用户按下 F6 键。如是 HAL，用户则按 F5 键。

当机器无法引导时就会出现某个错误信息，它指明找不到引导驱动程序。例如，如果机器无法引导并显示“无法引导设备”的信息，这有可能是引导程序磁盘是要求驱动程序的其他海量储存器设备并未被包含在操作系统中，如果机器无法引导且显示信息“设置无法决定你的机器类型”，这有可能是因为要求新的 HAL 驱动程序。在大多数机器上并不会出现这后一种错误；它多会出现在高端服务器上。

注意这个过程是用来安装一个并未被操作系统包含在“在盒子中”的驱动程序。不要使用这个过程来代替或更新操作系统中的驱动程序。如要替代或更新已包含的驱动程序，可等到系统引导并利用 Device Manager 在设备上执行“更新驱动程序”操作。

如果供应商为在引导中所需的设备分配了一个驱动程序，并要求用户在文本模式设置时安装它，那么除了其他的驱动程序文件之外，该供应商所分配的磁盘必须包括一个名为 *txtsetup.oem* 的文件。*txtsetup.oem* 文件是一个包括了以下信息的文本文件：

- 由 *txtsetup.oem* 文件支持的硬件组件列表
- 由每个组件的分配盘复制的文件列表
- 注册密钥列表及用以建立每个组件的值列表

带有单个 *txtsetup.oem* 文件的单个分配盘可以支持多个硬件设备。这样的文件可以包括带有指定对每个硬件类型缺省选择的一个 **Defaults** 部分。该文件也可以包括每个硬件类型及与 **Files.***Hwcomponent.ID* 及 **Config.***DriverKey* 部分恰当相关的 *HwComponent* 项。

以下示例显示了 PnP SCSI 控制程序的 *txtsetup.oem* 文件样本：

```
[Disks]
d1 = “OEM SCSI Driver disk”,\disk1.tag,\

[Defaults]
SCSI = oemscsi

[SCSI]
oemscsi = “OEM Fast SCSI Controller”

[Files.SCSI.oemscsi]
driver = d1,oemfs2.sys,OEMSCSI
```

```
inf = d1,oemsetup.inf
dll = d1,oemdrv.dll
catalog = d1,oemdrv.cat
[Config.OEMSCSI]
value = “”,tag,REG_DWORD,5
value = parameters\PnpInterface,5,REG_DWORD,1
```

参见《Windows 2000 驱动程序开发参考》一书第一卷第 9 章第 3 节中有关 *txtsetup.oem* 文件格式的详细内容。