

Writing Windows WDM Device Drivers

Covers NT 4, Win 98, and Win 2000

Chris Cant

**R&D Books
Lawrence, Kansas 66046**

**R&D Books
Miller Freeman, Inc.
1601 W. 23rd Street, Suite 200
Lawrence, KS 66046
USA**

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where R&D is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

Copyright © 1999 by Miller Freeman, Inc., except where noted otherwise. Published by R&D Books, Miller Freeman, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

The programs in this book are presented for instructional value. The programs have been carefully tested, but are not guaranteed for any particular purpose. The publisher does not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher assumes no liability for damages resulting from the use of the information in this book or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Cover art created by Robert Ward.

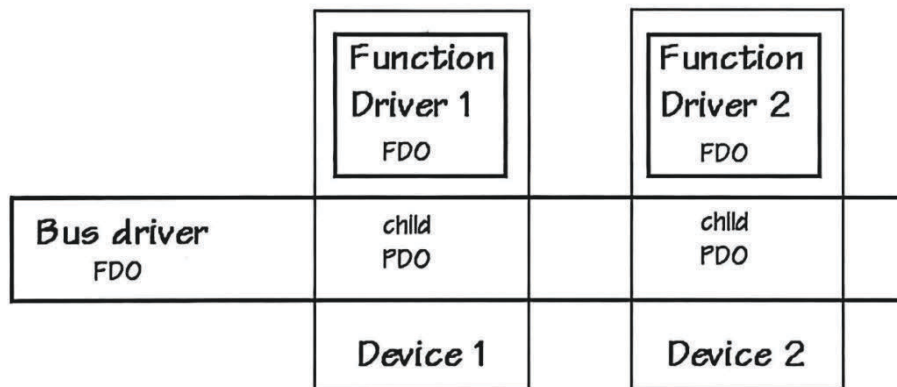
Distributed in the U.S. and Canada by:
Publishers Group West
1700 Fourth Street
Berkeley, CA 94710
1-800-788-3123
ISBN: 0-87930-565-7

***un* Miller Freeman**
A United News & Media publication

Just to reiterate, each **function** driver knows how to control the functions in a device. However, a function driver usually uses the facilities of its underlying **bus** driver to talk to its device.

Even a virtual device driver like **Wdm1** has a device stack. **Wdm1** is a function driver and so its device object is an **FDO**. There is a bus driver underneath **Wdm1**, the system **Unknown** driver. The **Unknown** bus driver makes a **PDO** to represent each **Wdm1** device. The **Unknown** driver does some important jobs, even though it does not correspond to a hardware bus. The following chapter describes the jobs that even a minimal bus driver like **Unknown** must do.

Figure 8.1 A bus driver, two devices, and their function drivers



© 1999 PHD Computer Consultants Ltd

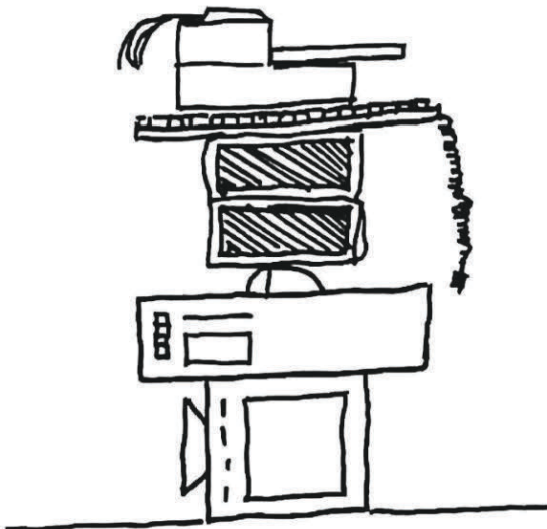
Device Stacks

When one driver layers its device over another, it forms a *device stack*. Each device in the stack usually plays some part in processing user requests.

Figure 8.2 shows a generic device stack. A bus driver is always at the bottom of the stack. One or more function drivers do the main device control functions.

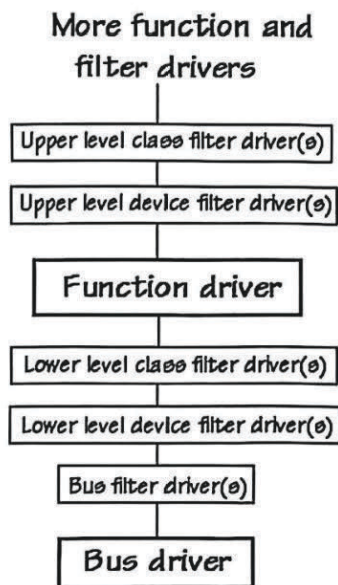
Filter drivers are used to modify the behavior of standard device drivers. Rather than rewrite a whole bus or class driver, a filter driver modifies its actions in the area of interest. As the figure shows, there are various types of filter driver. A bus filter driver acts on all devices that attach to a particular bus driver. Class filter drivers are installed in the stack for each device of the specified class. Finally, a device filter driver is installed for one particular device. *Lower-level* filter drivers are below the function driver, while *upper-level* drivers are above the function driver in the stack.

Filter drivers are only installed when a device is first installed, so they cannot be inserted once a device stack has been built.



"You said you wanted a device stack."

Figure 8.2 Bus, function, and filter drivers



© 1995 PHD Computer Consultants Ltd

Monolithic and Layered Drivers

When a standard keyboard is found, Windows loads the appropriate driver. This talks directly to the keyboard through its controller electronics. The keyboard driver is described as *monolithic* because it takes over all the processing required to handle the keyboard (i.e., it does not use other drivers to do its job).

In contrast, USB drivers use a layered approach. To use a USB keyboard, the keyboard driver uses the services of the USB class drivers.

As shown in Chapter 21, the USB class drivers expose an upper edge that drivers higher than it must use. Such drivers can issue USB Request Blocks (URBs), which transmit or receive information from a USB device. The keyboard driver does not care how the USB class drivers do their job.

The situation for the USB class drivers themselves is probably slightly different³. They are told where the USB bus controller electronics live, and they make use of the services of the PCI bus driver. However, I suspect that the USB class drivers talk directly to their own hardware. Asking the PCI bus driver to write or read memory would be too slow.

A different approach would have been to let each USB driver talk directly to the USB hardware in a monolithic driver. This would lead to two problems. The first is that each USB driver would need a large amount of code. Duplicating this code would waste memory and writing large drivers is decidedly error prone. The second problem is almost of more significance: how would all these different drivers coordinate their activities? The answer is, of course, with difficulty. Using different layers of drivers solves both these problems.

Plug and Play Messages

This section looks at the messages that a PnP driver receives. Its AddDevice routine is called when a device is added. After that, messages are sent using the PnP IRP. The PnP messages have the same major function code IRP_MJ_PNP, but each message uses a different minor function code. Each message has different parameters in the IRP.

A PnP driver must implement an AddDevice routine and handle various PnP IRPs. Table 8.2 is a list of all the minor function codes. The first eight of these minor codes are the most important. The other minor function codes are usually just handled by bus drivers.

Table 8.2 Plug and Play minor function codes

Common PnP IRPs	
IRP_MN_START_DEVICE	Assign resources and start a device
IRP_MN_QUERY_REMOVE_DEVICE	Ask if a device can be removed
IRP_MN_CANCEL_REMOVE_DEVICE	Cancel a query remove request
IRP_MN_REMOVE_DEVICE	Device has been unplugged or uninstalled Deallocate resources and remove a device
IRP_MN_SURPRISE_REMOVAL (W2000 only)	A user has unexpectedly unplugged a device
IRP_MN_QUERY_STOP_DEVICE	Ask if a device can be stopped
IRP_MN_CANCEL_STOP_DEVICE	Cancel a query stop request
IRP_MN_STOP_DEVICE	Stop a device for resource reallocation

3. I don't know for sure because I don't know how they work. And I don't care!

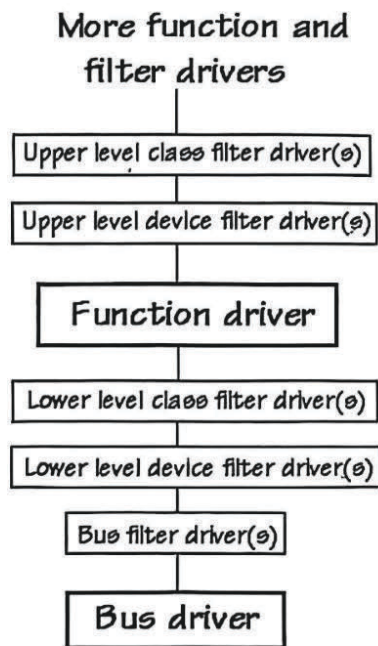
to this common device, and, for example, use some IOCTLs to change the general behavior of your driver.

The complication to this approach is that your IRP dispatch routines will receive requests both for your ordinary devices and for your common device. Control requests for the ordinary devices should be rejected, and vice versa. The simplest way to achieve this functionality is to have a flag in the device extension that indicates the device type. You could have two separate device extensions, with the device type flag in a common portion of each structure.

Filter Drivers

Filter drivers let you modify the behavior of standard system drivers. A WDM filter driver is inserted into the device stack when the device stack is built. Figure 19.2 shows the different types of filter driver. This book does not cover filter drivers.

Figure 19.2 WDM filter drivers



© 1998 PHD Computer Consultants Ltd

Filter drivers can also be written to work in Windows NT 4 and NT 3.51. However, WDM device stacks are not available in these older operating systems. Instead, the `IoGetDeviceObjectPointer` or `IoAttachDevice` routines can be used to layer your device transparently above existing device objects. This book does not cover these types of filter driver, either.

NT Layering

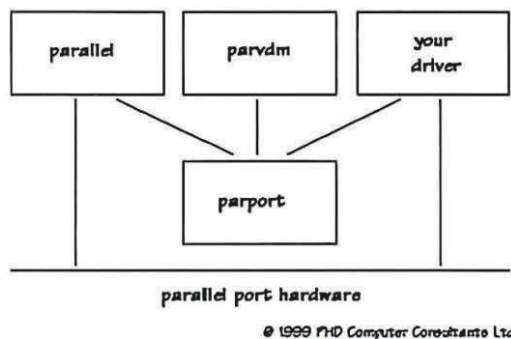
Windows NT 4, NT 3.51, and 2000 use layers of NT style drivers. This is not the layering of Plug and Play devices seen in most of this book. Instead, each driver must be loaded in the correct order and use any existing drivers.

Parallel Port Drivers

As an example, Figure 19.3 shows the layering of the parallel port drivers. The parport driver is loaded first. parport arbitrates access to the parallel port hardware. The parallel and parvdm drivers are then loaded. When these drivers want to access the parallel port hardware, they must ask parport for access rights. Once access rights have been obtained, they can talk to the parallel port electronics directly.

New parallel port drivers should use the same technique to get access to the hardware². The DDK recommends that you grab access to the port on a per-IRP basis. For most applications, I recommend taking control of the port for the duration of a larger transaction (e.g., while a handle is open to your device).

Figure 19.3 Parallel port layering



The parport driver creates a named kernel device object for each parallel port it finds. These device objects have kernel names starting with `\Device\ParallelPort0`. Use `IoGetDeviceObjectPointer` to obtain a pointer to the relevant device object³. Store this in your device extension for later use.

The parport driver supports several Internal IOCTLs. `IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO` returns information about the parallel port (the base register address and port allocation routines). When the information is safely stored away, a similar (but undocumented) `IOCTL_INTERNAL_RELEASE_PARALLEL_PORT_INFO` call is made to release the information.

Two port allocation routines are available. `TryAllocatePort` tries to allocate the port and returns straightaway. If this succeeds, `FreePort` is used (later) to return the port. Call `TryAllocatePort` when you want to start accessing a parallel port and `FreePort` when you are finished with it.

When you install a driver that uses the parport driver, you must make sure that your driver loads after parport has started. Do this by including a `DependOnService` string value in your driver's registry entry that is set to parport.

Microsoft may be introducing a new WDM architecture for these parallel port drivers. A parallel port bus driver would replace parport. It would enumerate the ports available on the "bus". This is particularly important, as several IEEE 1284.3 printers may be daisy-chained

2. The only alternative is to remove the parport, parallel, and parvdm drivers, which would stop any existing parallel port access.

3. See Chapter 23 for details of `IoGetDeviceObjectPointer`.

on a single parallel port connection. The header file `parallel.h` in the W2000 DDK `src\kernel\inc` directory gives a hint of how the new interface will work.

Conclusion

Writing a client driver to access standard buses has been made much easier with the provision of several important system drivers. In addition, you can write minidrivers to interface these standard drivers to your own specialized hardware.

Let's now start looking at the standard system drivers by considering the Universal System Bus.