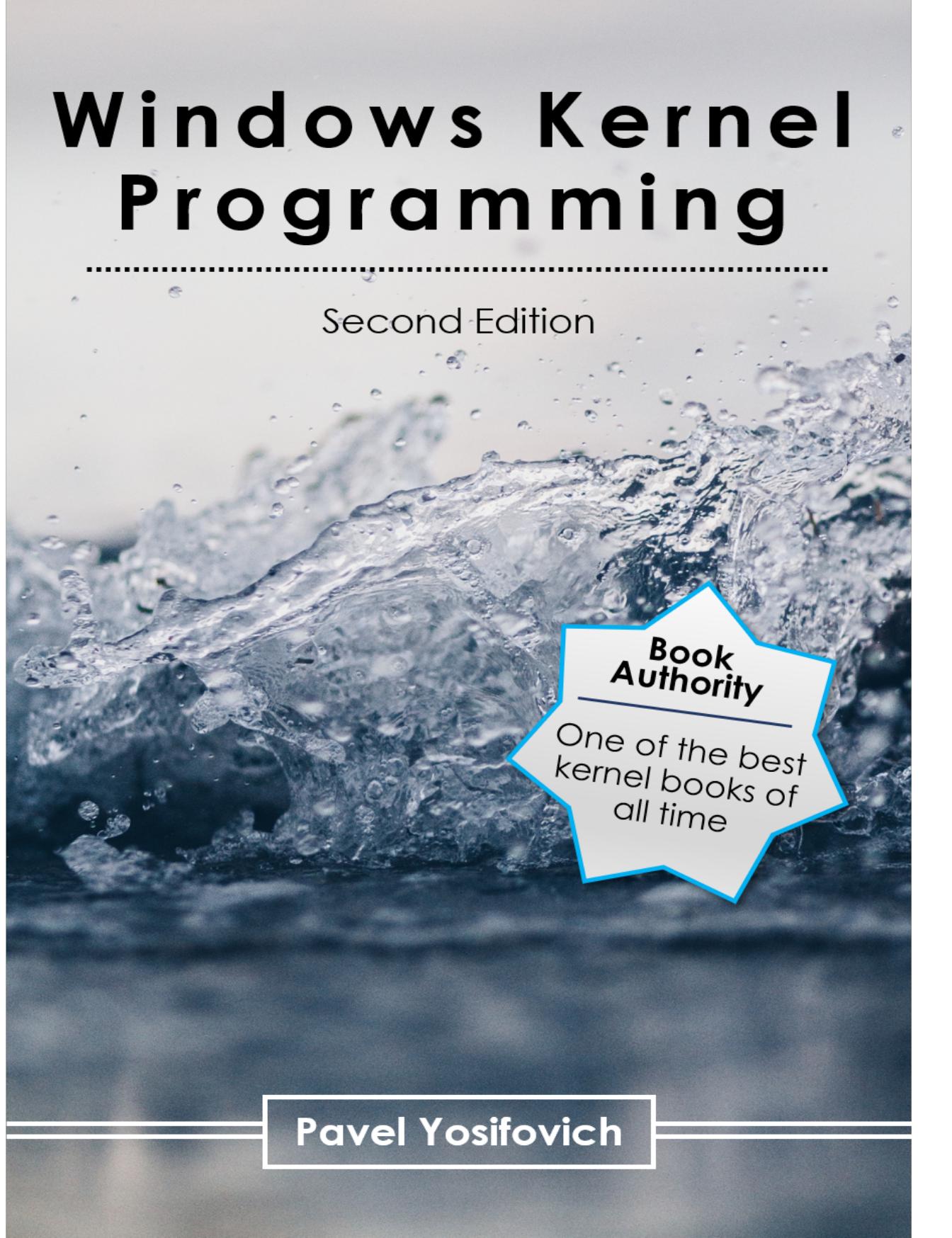


Windows Kernel Programming

Second Edition



Book
Authority

One of the best
kernel books of
all time

Pavel Yosifovich

Windows Kernel Programming, Second Edition

Pavel Yosifovich

This book is for sale at <http://leanpub.com/windowskernelprogrammingsecondedition>

This version was published on 2023-02-26



This is a **Leanpub** book. Leanpub empowers authors and publishers with the Lean Publishing process. **Lean Publishing** is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 - 2023 Pavel Yosifovich

Contents

Introduction	1
Who Should Read This Book	1
What You Should Know to Use This Book	1
Book Contents	1
Sample Code	2
Chapter 1: Windows Internals Overview	4
Processes	4
Virtual Memory	6
Page States	8
System Memory	8
Threads	9
Thread Stacks	10
System Services (a.k.a. System Calls)	12
General System Architecture	14
Handles and Objects	16
Object Names	17
Accessing Existing Objects	20
Chapter 2: Getting Started with Kernel Development	24
Installing the Tools	24
Creating a Driver Project	25
The DriverEntry and Unload Routines	26
Deploying the Driver	28
Simple Tracing	32
Summary	34
Chapter 3: Kernel Programming Basics	35
General Kernel Programming Guidelines	35
Unhandled Exceptions	36
Termination	36
Function Return Values	37
IRQL	37
C++ Usage	37
Testing and Debugging	38
Debug vs. Release Builds	39

CONTENTS

The Kernel API	39
Functions and Error Codes	40
Strings	41
Dynamic Memory Allocation	43
Linked Lists	46
The Driver Object	48
Object Attributes	49
Device Objects	53
Opening Devices Directly	56
Summary	59
Chapter 4: Driver from Start to Finish	60
Introduction	60
Driver Initialization	61
Passing Information to the Driver	63
Client / Driver Communication Protocol	64
Creating the Device Object	64
Client Code	67
The Create and Close Dispatch Routines	69
The Write Dispatch Routine	70
Installing and Testing	74
Summary	78
Chapter 5: Debugging and Tracing	79
Debugging Tools for Windows	79
Introduction to <i>WinDbg</i>	80
Tutorial: User mode debugging basics	81
Kernel Debugging	98
Local Kernel Debugging	98
Local kernel Debugging Tutorial	100
Full Kernel Debugging	108
Using a Virtual Serial Port	108
Using the Network	112
Kernel Driver Debugging Tutorial	113
Asserts and Tracing	118
Asserts	118
Extended DbgPrint	120
Other Debugging Functions	125
Trace Logging	126
Viewing ETW Traces	129
Summary	134
Chapter 6: Kernel Mechanisms	135
Interrupt Request Level (IRQL)	135
Raising and Lowering IRQL	138
Thread Priorities vs. IRQLs	139

CONTENTS

Deferred Procedure Calls	139
Using DPC with a Timer	142
Asynchronous Procedure Calls	143
Critical Regions and Guarded Regions	144
Structured Exception Handling	144
Using __try/__except	146
Using __try/__finally	148
Using C++ RAII Instead of __try / __finally	150
System Crash	152
Crash Dump Information	154
Analyzing a Dump File	158
System Hang	161
Thread Synchronization	163
Interlocked Operations	163
Dispatcher Objects	165
Mutex	167
Fast Mutex	173
Semaphore	173
Event	174
Named Events	175
Executive Resource	177
High IRQL Synchronization	179
The Spin Lock	181
Queued Spin Locks	184
Work Items	185
Summary	187
Chapter 7: The I/O Request Packet	188
Introduction to IRPs	188
Device Nodes	189
IRP Flow	193
IRP and I/O Stack Location	194
Viewing IRP Information	197
Dispatch Routines	201
Completing a Request	202
Accessing User Buffers	203
Buffered I/O	204
Direct I/O	208
User Buffers for IRP_MJ_DEVICE_CONTROL	213
Putting it All Together: The <i>Zero</i> Driver	215
Using a Precompiled Header	216
The DriverEntry Routine	218
The Create and Close Dispatch Routines	220
The Read Dispatch Routine	220
The Write Dispatch Routine	221
Test Application	222

CONTENTS

Read/Write Statistics	223
Summary	227
Chapter 8: Advanced Programming Techniques (Part 1)	228
Driver Created Threads	228
Memory Management	230
Pool Allocations	230
Secure Pools	233
Overloading the new and delete Operators	235
Lookaside Lists	237
The “Classic” Lookaside API	237
The Newer Lookaside API	239
Calling Other Drivers	242
Putting it All Together: The Melody Driver	244
Client Code	260
Invoking System Services	261
Example: Enumerating Processes	263
Summary	266
Chapter 9: Process and Thread Notifications	267
Process Notifications	267
Implementing Process Notifications	270
The DriverEntry Routine	274
Handling Process Exit Notifications	276
Handling Process Create Notifications	279
Providing Data to User Mode	283
The User Mode Client	285
Thread Notifications	288
Image Load Notifications	291
Final Client Code	298
Remote Thread Detection	301
The Detector Client	310
Summary	311
Chapter 10: Object and Registry Notifications	312
Object Notifications	312
Pre-Operation Callback	314
Post-Operation Callback	317
The Process Protector Driver	318
Object Notification Registration	318
Managing Protected Processes	320
The Pre-Callback	324
The Client Application	324
Registry Notifications	328
Registry Overview	328
Using Registry Notifications	332

CONTENTS

Handling Pre-Notifications	334
Handling Post-Operations	334
Extending the SysMon Driver	335
Handling Registry Callback	336
Modified Client Code	342
Performance Considerations	344
Miscellaenous Notes	344
Summary	345
Chapter 11: Advanced Programming Techniques (Part 2)	346
Timers	346
Kernel Timers	346
Timer Resolution	348
High-Resolution Timers	350
I/O Timer	354
Generic Tables	355
Splay Trees	355
Tables Sample Driver	359
Testing the Tables Driver	369
AVL Trees	372
Hash Tables	373
Singly Linked Lists	373
Sequenced Singly-Linked Lists	374
Callback Objects	376
Chapter 12: File System Mini-Filters	380
Introduction	381
Loading and Unloading	382
Initialization	384
Operations Callback Registration	387
The Altitude	391
Installation	393
Installing the Driver	396
Processing I/O Operations	396
Pre Operation Callbacks	397
Post Operation Callbacks	399
File Names	401
File Name Parts	402
RAII FLT_FILE_NAME_INFORMATION wrapper	405
The Delete Protector Driver	407
Handling Pre-Create	412
Handling Pre-Set Information	414
DelProtect Configuration	416
Testing the Modified Driver	418
The Directory Hiding Driver	419
Managing Directories	419

CONTENTS

Phase 1: Prevent Access	423
Phase 2: Making a Directory Invisible	425
Contexts	437
Managing Contexts	439
Initiating I/O Requests	441
The File Backup Driver	442
The Post Create Callback	445
The Pre-Write Callback	449
The Post-Cleanup Callback	456
Testing the Driver	457
Restoring Backups	457
File Copying with a Section Object	459
User Mode Communication	462
Creating the Communication Port	463
User Mode Connection	464
Sending and Receiving Messages	465
Enhanced Backup Driver	467
The User Mode Client	470
Debugging	471
Exercises	475
Summary	476
Chapter 13: The Windows Filtering Platform	477
WFP Overview	477
The WFP API	487
User-Mode Examples	489
Enumerating Objects	489
Adding Filters	491
Callout Drivers	498
Callout Driver Basics	499
Callout Registration	499
Demo: Callout Driver	507
The Driver	507
Managing Processes	511
Callout Callbacks	514
Demo: User-Mode Client	518
Testing	527
Debugging	528
Summary	528
Chapter 14: Introduction to KMDF	529
Introduction to WDF	529
Introduction to KMDF	531
KMDF Objects	531
Core Object Types	532
Object Creation	533

CONTENTS

Context Memory	536
The Booster KMDF Driver	536
Driver Initialization	537
Device I/O Control Handling	543
The INF File	545
The Install Sections	547
Device Installation	548
The User-Mode Client	549
Installing and Testing	552
Registering a Device Class	558
Summary	560
Chapter 15: Miscellaneous Topics	561
Driver Signing	561
Driver Verifier	565
Example Driver Verifier Sessions	569
Filter Drivers	573
Filter Driver Implementation	575
Attaching Filters	576
Attaching Filters at Arbitrary Time	578
Filter Cleanup	580
More on Hardware-Based Filter Drivers	581
Device Monitor	582
Adding a Device to Filter	584
Removing a Filter Device	589
Initialization and Unload	591
Handling Requests	592
Testing the Driver	596
Results of Requests	600
Driver Hooking	602
Kernel Libraries	604
Summary	605
Appendix: The Kernel Template Library	607
Standard Library	607
Synchronization	607
Memory	607
Strings	608
Containers	609
File System Mini-Filters	609

Introduction

Windows kernel programming is considered by many a dark art, available to select few that manage to somehow unlock the mysteries of the Windows kernel. Kernel development, however, is no different than user-mode development, at least in general terms. In both cases, a good understanding of the platform is essential for producing high quality code.

The book is a guide to programming within the Windows kernel, using the well-known Visual Studio integrated development environment (IDE). This environment is familiar to many developers in the Microsoft space, so that the learning curve is restricted to kernel understanding, coding and debugging, with less friction from the development tools.

The book targets software device drivers, a term I use to refer to drivers that do not deal with hardware. Software kernel drivers have full access to the kernel, allowing these to perform any operation allowed by the kernel. Some software drivers are more specific, such as file system mini filters, also described in the book.

Who Should Read This Book

The book is intended for software developers that target the Windows kernel, and need to write kernel drivers to achieve their goals. Common scenarios where kernel drivers are employed are in the Cyber Security space, where kernel drivers are the chief mechanism to get notified of important events, with the power to intercept certain operations. The book uses C and C++ for code examples, as the kernel API is all C. C++ is used where it makes sense, where its advantages are obvious in terms of maintenance, clarity, resource management, or any combination of these. The book does not use complex C++ constructs, such as template metaprogramming. The book is not about C++, it's about Windows kernel drivers.

What You Should Know to Use This Book

Readers should be very comfortable with the C programming language, especially with pointers, structures, and its standard library, as these occur very frequently when working with kernel APIs. Basic C++ knowledge is highly recommended, although it is possible to traverse the book with C proficiency only.

Book Contents

Here is a quick rundown of the chapters in the book:

- Chapter 1: **Windows Internals Overview** - provides the fundamentals of the internal workings of the Windows OS at a high level, enough to get the fundamentals without being bogged down by too many details.

- Chapter 2: **Getting Started with Kernel Development** - describes the tools and procedures needed to set up a development environment for developing kernel drivers. A very simple driver is created to make sure all the tools and procedures are working correctly.
- Chapter 3: **Kernel Programming Basics** - looks at the fundamentals of writing drivers, including basic kernel APIs, handling of common programming tasks involving strings, linked lists, dynamic memory allocations, and more.
- Chapter 4: **Driver from Start to Finish** - shows how to build a complete driver that performs some useful functionality, along with a client application to drive it.
- Chapter 5: **Debugging and Tracing** - shows how to use *WinDbg* to debug user-mode and especially kernel-mode code. It also looks at tracing driver code.
- Chapter 6: **Kernel Mechanisms** - looks at various kernel mechanisms that a driver developer must be familiar with, such IRQLs, BSODs, and synchronization.
- Chapter 7: **The I/O Request Packet** - discussed the details of handling IRPs, accessing user-mode buffers in a safe way, and other aspects of handling I/O requests, which is the main work of a typical driver.
- Chapter 8: **Advanced Programming Techniques (Part 1)** - discussed various kernel programming techniques, including thread management, memory management and using system calls.
- Chapter 9: **Process and Thread Notifications** - shows how drivers can be notified when processes and threads are created or destroyed.
- Chapter 10: **Object and Registry Notifications** - shows how drivers can be notified when handles are opened to certain types of objects. The chapter also shows how to be notified when Registry operations are invoked.
- Chapter 11: **Advanced Programming Techniques (Part 2)** - shows more techniques useful for driver writers, such as using timers and trees.
- Chapter 12: **File System Mini-Filters** - discussed the support provided by Windows and the Filter Manager to handle file system notifications.
- Chapter 13: **The Windows Filtering Platform** - shows how to the WFP to intercept network operations.
- Chapter 14: **Introduction to KMDF** - introduces the basics of the Kernel Mode Driver Framework.
- Chapter 15: **Miscellaneous Topics** - discusses other topics of interest, such as generic filter drivers, and hooking drivers.
- Appendix: **The Kernel Template Library** - summaries the usage of a set of classes supporting many generic aspects of kernel development that has been developed specifically for this book.

If you are new to Windows kernel development, you should read chapters 1 to 7 in order. Chapter 8 contains some advanced material you may want to go back to after you have built a few simple drivers. Chapters 9 onward describe specialized techniques, and in theory at least, can be read in any order.

Sample Code

All the sample code from the book is freely available on the book's Github repository at <https://github.com/zodiacon/windowskernelprogrammingbook2e>. Updates to the code samples will be pushed to

this repository. It's recommended the reader clone the repository to the local machine, so it's easy to experiment with the code directly.

All code samples have been compiled with Visual Studio 2019. It's possible to compile most code samples with earlier versions of Visual Studio if desired. There might be few features of the latest C++ standards that may not be supported in earlier versions, but these should be easy to fix.

Happy reading!

Pavel Yosifovich

March 2023

Chapter 1: Windows Internals Overview

This chapter describes the most important concepts in the internal workings of Windows. Some of the topics will be described in greater detail later in the book, where it's closely related to the topic at hand. Make sure you understand the concepts in this chapter, as these make the foundations upon which any driver and even user mode low-level code, is built.

In this chapter:

- Processes
 - Virtual Memory
 - Threads
 - System Services
 - System Architecture
 - Handles and Objects
-

Processes

A process is a containment and management object that represents a running instance of a program. The term “process runs” which is used fairly often, is inaccurate. Processes don't run – processes manage. Threads are the ones that execute code and technically run. From a high-level perspective, a process owns the following:

- An executable program, which contains the initial code and data used to execute code within the process. This is true for most processes, but some special ones don't have an executable image (created directly by the kernel).
- A private virtual address space, used for allocating memory for whatever purposes the code within the process needs it.
- An access token (called *primary token*), which is an object that stores the security context of the process, used by threads executing in the process (unless a thread assumes a different token by using *impersonation*).
- A private handle table to executive objects, such as events, semaphores, and files.

- One or more threads of execution. A normal user-mode process is created with one thread (executing the classic `main`/`WinMain` function). A user mode process without threads is mostly useless, and under normal circumstances will be destroyed by the kernel.

These elements of a process are depicted in figure 1-1.

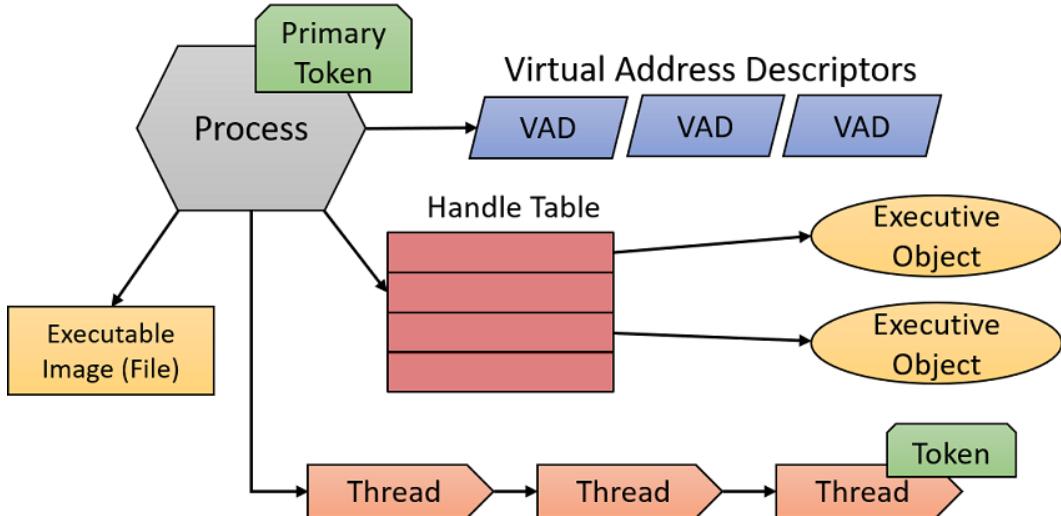
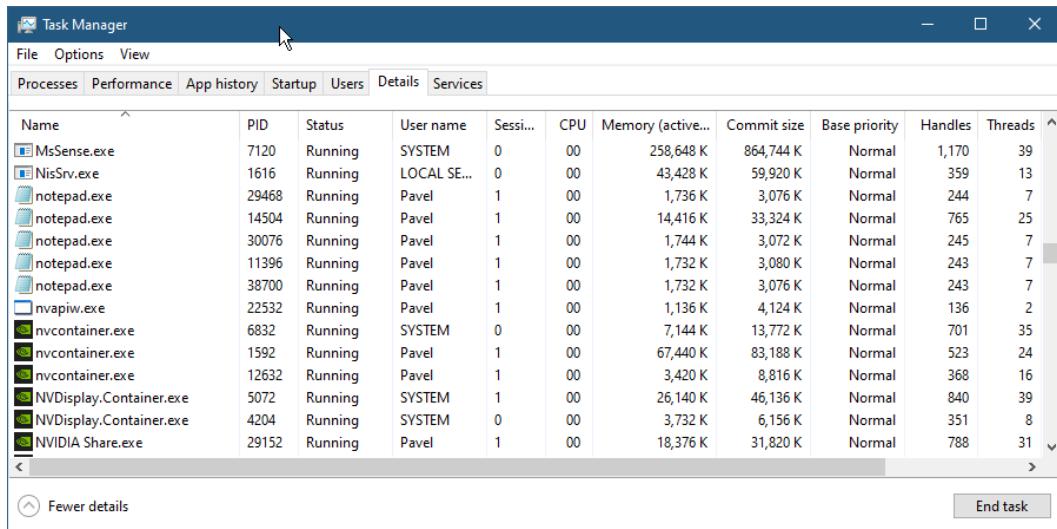


Figure 1-1: Important ingredients of a process

A process is uniquely identified by its Process ID, which remains unique as long as the kernel process object exists. Once it's destroyed, the same ID may be reused for new processes. It's important to realize that the executable file itself is not a unique identifier of a process. For example, there may be five instances of *notepad.exe* running at the same time. Each of these *Notepad* instances has its own address space, threads, handle table, process ID, etc. All those five processes are using the same image file (*notepad.exe*) as their initial code and data. Figure 1-2 shows a screenshot of *Task Manager*'s Details tab showing five instances of *Notepad.exe*, each with its own attributes.



The screenshot shows the Windows Task Manager window with the 'Details' tab selected. The table lists various processes, with several instances of 'notepad.exe' running under different user accounts (SYSTEM, LOCAL SE..., Pavel). The columns include Name, PID, Status, User name, Session, CPU, Memory (active...), Commit size, Base priority, Handles, and Threads.

Name	PID	Status	User name	Sessi...	CPU	Memory (active...)	Commit size	Base priority	Handles	Threads
MsSense.exe	7120	Running	SYSTEM	0	00	258,648 K	864,744 K	Normal	1,170	39
NisSrv.exe	1616	Running	LOCAL SE...	0	00	43,428 K	59,920 K	Normal	359	13
notepad.exe	29468	Running	Pavel	1	00	1,736 K	3,076 K	Normal	244	7
notepad.exe	14504	Running	Pavel	1	00	14,416 K	33,324 K	Normal	765	25
notepad.exe	30076	Running	Pavel	1	00	1,744 K	3,072 K	Normal	245	7
notepad.exe	11396	Running	Pavel	1	00	1,732 K	3,080 K	Normal	243	7
notepad.exe	38700	Running	Pavel	1	00	1,732 K	3,076 K	Normal	243	7
nvapiw.exe	22532	Running	Pavel	1	00	1,136 K	4,124 K	Normal	136	2
nvcontainer.exe	6832	Running	SYSTEM	0	00	7,144 K	13,772 K	Normal	701	35
nvcontainer.exe	1592	Running	Pavel	1	00	67,440 K	83,188 K	Normal	523	24
nvcontainer.exe	12632	Running	Pavel	1	00	3,420 K	8,816 K	Normal	368	16
NVDisplay.Container.exe	5072	Running	SYSTEM	1	00	26,140 K	46,136 K	Normal	840	39
NVDisplay.Container.exe	4204	Running	SYSTEM	0	00	3,732 K	6,156 K	Normal	351	8
NVIDIA Share.exe	29152	Running	Pavel	1	00	18,376 K	31,820 K	Normal	788	31

Figure 1-2: Five instances of notepad

Virtual Memory

Every process has its own virtual, private, linear address space. This address space starts out empty (or close to empty, since the executable image and NtD11.D11 are the first to be mapped, followed by more subsystem DLLs). Once execution of the main (first) thread begins, memory is likely to be allocated, more DLLs loaded, etc. This address space is private, which means other processes cannot access it directly. The address space range starts at zero (technically the first and last 64KB of the address space cannot be committed), and goes all the way to a maximum which depends on the process “bitness” (32 or 64 bit) and the operating system “bitness” as follows:

- For 32-bit processes on 32-bit Windows systems, the process address space size is 2 GB by default.
- For 32-bit processes on 32-bit Windows systems that use the *increase user virtual address space* setting, it can be configured to have up to 3GB of address space per process. To get the extended address space, the executable from which the process was created must have been marked with the LARGEADDRESSAWARE linker flag in its PE header. If it was not, it would still be limited to 2 GB.
- For 64-bit processes (on a 64-bit Windows system, naturally), the address space size is 8 TB (Windows 8 and earlier) or 128 TB (Windows 8.1 and later).
- For 32-bit processes on a 64-bit Windows system, the address space size is 4 GB if the executable image has the LARGEADDRESSAWARE flag in its PE header. Otherwise, the size remains at 2 GB.



The requirement of the LARGEADDRESSAWARE flag stems from the fact that a 2 GB address range requires 31 bits only, leaving the most significant bit (MSB) free for application use. Specifying this flag indicates that the program is not using bit 31 for anything and so having that bit set (which would happen for addresses larger than 2 GB) is not an issue.

Each process has its own address space, which makes any process address relative, rather than absolute. For example, when trying to determine what lies in address 0x20000, the address itself is not enough; the process to which this address relates to must be specified.

The memory itself is called *virtual*, which means there is an indirect relationship between an address and the exact location where it's found in physical memory (RAM). A buffer within a process may be mapped to physical memory, or it may temporarily reside in a file (such as a page file). The term *virtual* refers to the fact that from an execution perspective, there is no need to know if the memory about to be accessed is in RAM or not; if the memory is indeed mapped to RAM, the CPU will perform the virtual-to-physical translation before accessing the data. If the memory is not resident (specified by a flag in the translation table entry), the CPU will raise a page fault exception that causes the memory manager's page fault handler to fetch the data from the appropriate file (if indeed it's a valid page fault), copy it to RAM, make the required changes in the page table entries that map the buffer, and instruct the CPU to try again. Figure 1-3 shows this conceptual mapping from virtual to physical memory for two processes.

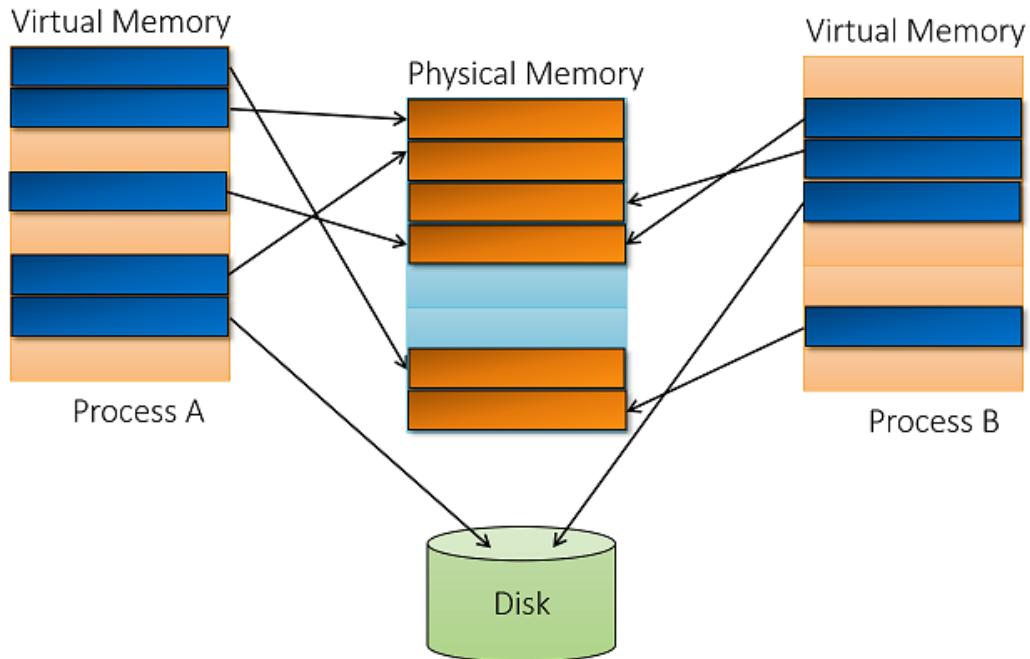


Figure 1-3: virtual memory mapping

The unit of memory management is called a *page*. Every attribute related to memory is always at a page's granularity, such as its protection or state. The size of a page is determined by CPU type (and on some processors, may be configurable), and in any case, the memory manager must follow suit. Normal (sometimes called small) page size is 4 KB on all Windows-supported architectures.

Apart from the normal (small) page size, Windows also supports large pages. The size of a large page is 2 MB (x86/x64/ARM64) or 4 MB (ARM). This is based on using the Page Directory Entry (PDE) to map the large page without using a page table. This results in quicker translation, but most importantly better use of the *Translation Lookaside Buffer* (TLB) – a cache of recently translated pages maintained

by the CPU. In the case of a large page, a single TLB entry maps significantly more memory than a small page.



The downside of large pages is the need to have the memory contiguous in RAM, which can fail if memory is tight or very fragmented. Also, large pages are always non-pageable and can only use read/write protection.

Huge pages (1 GB in size) are supported on Windows 10 and Server 2016 and later. These are used automatically with large pages if an allocation is at least 1 GB in size, and that size can be located as contiguous in RAM.

Page States

Each page in virtual memory can be in one of three states:

- Free – the page is not allocated in any way; there is nothing there. Any attempt to access that page would cause an access violation exception. Most pages in a newly created process are free.
- Committed – the reverse of free; an allocated page that can be accessed successfully (assuming non-conflicting protection attributes; for example, writing to a read-only page causes an access violation). Committed pages are mapped to RAM or to a file (such as a page file).
- Reserved – the page is not committed, but the address range is reserved for possible future commitment. From the CPU's perspective, it's the same as Free – any access attempt raises an access violation exception. However, new allocation attempts using the `VirtualAlloc` function (or `NtAllocateVirtualMemory`, the related native API) that does not specify a specific address would not allocate in the reserved region. A classic example of using reserved memory to maintain contiguous virtual address space while conserving committed memory usage is described later in this chapter in the section "Thread Stacks".

System Memory

The lower part of the address space is for user-mode processes use. While a particular thread is executing, its associated process address space is visible from address zero to the upper limit as described in the previous section. The operating system, however, must also reside somewhere – and that somewhere is the upper address range that's supported on the system, as follows:

- On 32-bit systems running without the *increase user virtual address space* setting, the operating system resides in the upper 2 GB of virtual address space, from address `0x80000000` to `0xFFFFFFFF`.
- On 32-bit systems configured with the *increase user virtual address space* setting, the operating system resides in the address space left. For example, if the system is configured with 3 GB user address space per process (the maximum), the OS takes the upper 1 GB (from address `0xC0000000` to `0xFFFFFFFF`). The component that suffers mostly from this address space reduction is the file system cache.
- On 64-bit systems running Windows 8, Server 2012 and earlier, the OS takes the upper 8 TB of virtual address space.

- On 64-bit systems running Windows 8.1, Server 2012 R2 and later, the OS takes the upper 128 TB of virtual address space.

Figure 1-4 shows the virtual memory layout for the two “extreme” cases: 32-bit process on a 32-bit system (left) and a 64-bit process on a 64-bit system (right).

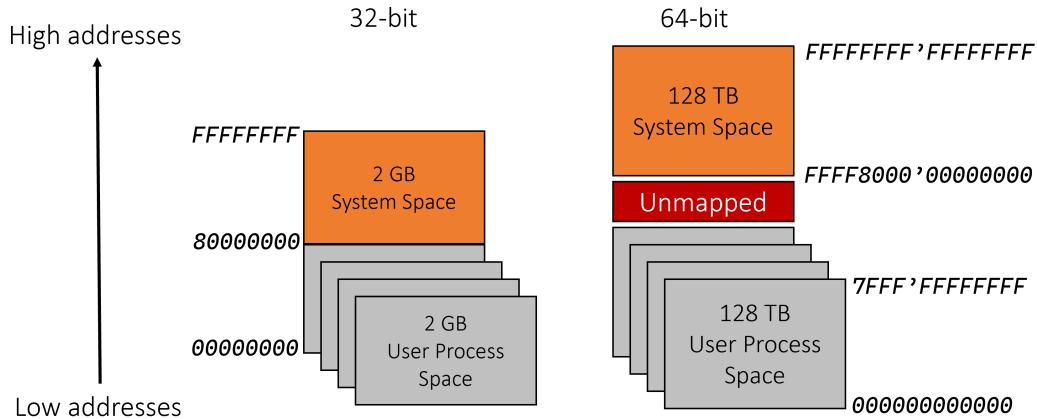


Figure 1-4: virtual memory layout

System space is not process-relative – after all, it’s the same system, the same kernel, the same drivers that service every process on the system (the exception is some system memory that is on a per-session basis but is not important for this discussion). It follows that any address in system space is absolute rather than relative, since it “looks” the same from every process context. Of course, actual access from user mode into system space results in an access violation exception.

System space is where the kernel itself, the Hardware Abstraction Layer (HAL), and kernel drivers reside once loaded. Thus, kernel drivers are automatically protected from direct user mode access. It also means they have a potentially system-wide impact. For example, if a kernel driver leaks memory, that memory will not be freed even after the driver unloads. User-mode processes, on the other hand, can never leak anything beyond their lifetime. The kernel is responsible for closing and freeing everything private to a dead process (all handles are closed and all private memory is freed).

Threads

The actual entities that execute code are threads. A Thread is contained within a process, using the resources exposed by the process to do work (such as virtual memory and handles to kernel objects). The most important details a thread owns are the following:

- Current access mode, either user or kernel.
- Execution context, including processor registers and execution state.
- One or two stacks, used for local variable allocations and call management.
- Thread Local Storage (TLS) array, which provides a way to store thread-private data with uniform access semantics.

- Base priority and a current (dynamic) priority.
- Processor affinity, indicating on which processors the thread is allowed to run on.

The most common states a thread can be in are:

- Running – currently executing code on a (logical) processor.
- Ready – waiting to be scheduled for execution because all relevant processors are busy or unavailable.
- Waiting – waiting for some event to occur before proceeding. Once the event occurs, the thread goes to the Ready state.

Figure 1-5 shows the state diagram for these states. The numbers in parenthesis indicate the state numbers, as can be viewed by tools such as *Performance Monitor*. Note that the Ready state has a sibling state called Deferred Ready, which is similar, and exists to minimize internal locking.

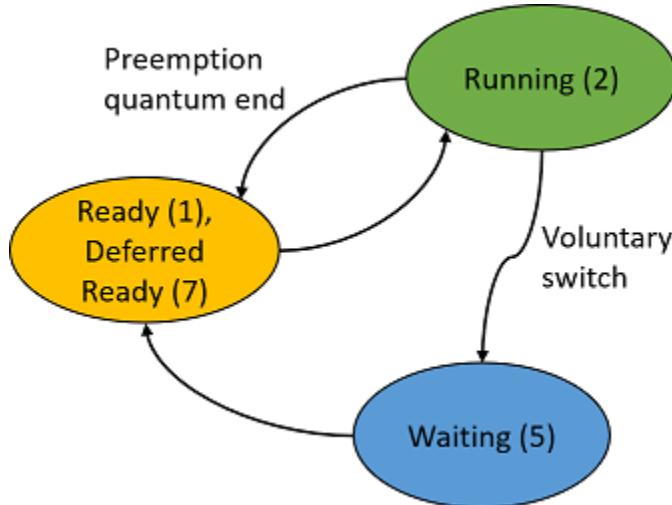


Figure 1-5: Common thread states

Thread Stacks

Each thread has a stack it uses while executing, used to store local variables, parameters passed to functions (in some cases), and where return addresses are stored prior to making function calls. A thread has at least one stack residing in system (kernel) space, and it's pretty small (default is 12 KB on 32-bit systems and 24 KB on 64-bit systems). A user-mode thread has a second stack in its process user-space address range and is considerably larger (by default can grow up to 1 MB). An example with three user-mode threads and their stacks is shown in figure 1-6. In the figure, threads 1 and 2 are in process A, and thread 3 is in process B.

The kernel stack always resides in RAM while the thread is in the Running or Ready states. The reason for this is subtle and will be discussed later in this chapter. The user-mode stack, on the other hand, may be paged out, just like any other user-mode memory.

The user-mode stack is handled differently than the kernel-mode stack in terms of its size. It starts out with a certain amount of committed memory (could be as small as a single page), where the next page is committed with a PAGE_GUARD attribute. The rest of the stack address space memory is reserved, thus not wasting memory. The idea is to grow the stack in case the thread's code needs to use more stack space. If the thread needs more stack space it would access the guard page which would throw a page-guard exception. The memory manager then removes the guard protection, and commits an additional page, marking it with a PAGE_GUARD attribute. This way, the stack grows as needed, avoiding the entire stack memory being committed upfront. Figure 1-7 shows this layout.

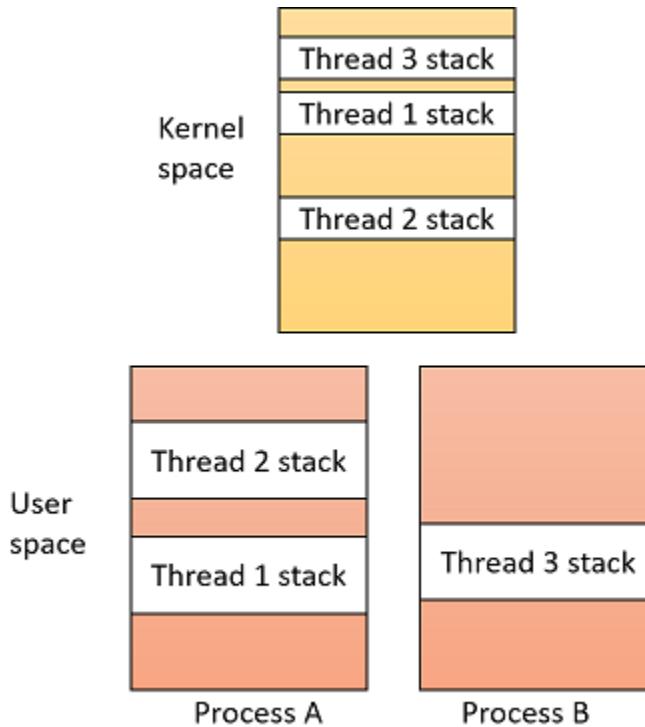


Figure 1-6: User mode threads and their stacks

Technically, Windows uses 3 guard pages rather than one in most cases.



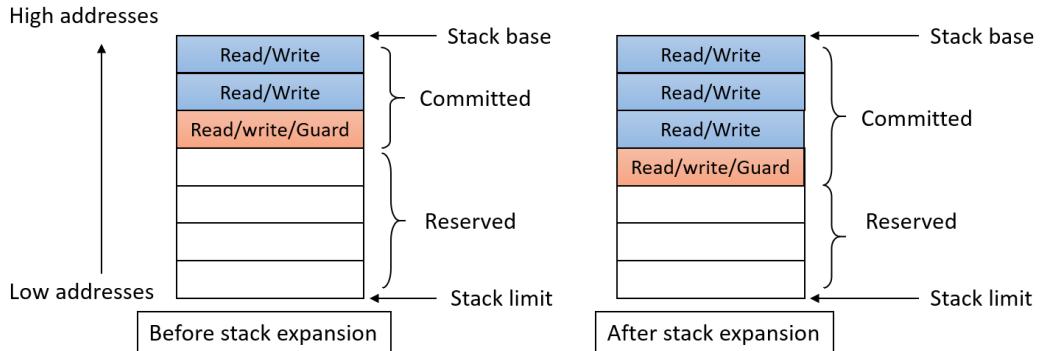


Figure 1-7: Thread's stack in user space

The sizes of a thread's user-mode stack are determined as follows:

- The executable image has a stack commit and reserved values in its Portable Executable (PE) header. These are taken as defaults if a thread does not specify alternative values. These are always used for the first thread in the process.
- When a thread is created with `CreateThread` (or similar functions), the caller can specify its required stack size, either the upfront committed size or the reserved size (but not both), depending on a flag provided to the function; specifying zero uses the defaults set in the PE header.



Curiously enough, the functions `CreateThread` and `CreateRemoteThread(Ex)` only allow specifying a single value for the stack size and can be the committed or the reserved size, but not both. The native (undocumented) function, `NtCreateThreadEx` allows specifying both values.

System Services (a.k.a. System Calls)

Applications need to perform various operations that are not purely computational, such as allocating memory, opening files, creating threads, etc. These operations can only be ultimately performed by code running in kernel mode. So how would user-mode code be able to perform such operations?

Let's take a common (simple) example: a user running a *Notepad* process uses the *File / Open* menu to request opening a file. *Notepad*'s code responds by calling the `CreateFile` documented Windows API function. `CreateFile` is documented as implemented in `kernel32.dll`, one of the Windows subsystem DLLs. This function still runs in user mode, so there is no way it can directly open a file. After some error checking, it calls `NtCreateFile`, a function implemented in `NTDLL.dll`, a foundational DLL that implements the API known as the *Native API*, and is the lowest layer of code which is still in user mode. This function (documented in the *Windows Driver Kit* for device driver developers) is the one that makes the transition to kernel mode. Before the actual transition, it puts a number, called system service number, into a CPU register (EAX on Intel/AMD architectures). Then it

issues a special CPU instruction (`syscall` on x64 or `sysenter` on x86) that makes the actual transition to kernel mode while jumping to a predefined routine called the *system service dispatcher*.

The system service dispatcher, in turn, uses the value in that EAX register as an index into a *System Service Dispatch Table* (SSDT). Using this table, the code jumps to the system service (system call) itself. For our *Notepad* example, the SSDT entry would point to the `NtCreateFile` function, implemented by the kernel's I/O manager. Notice the function has the same name as the one in `NTDLL.dll`, and has the same parameters as well. On the kernel side is the real implementation. Once the system service is complete, the thread returns to user mode to execute the instruction following `sysenter/syscall`. This sequence of calls is depicted in figure 1-8.

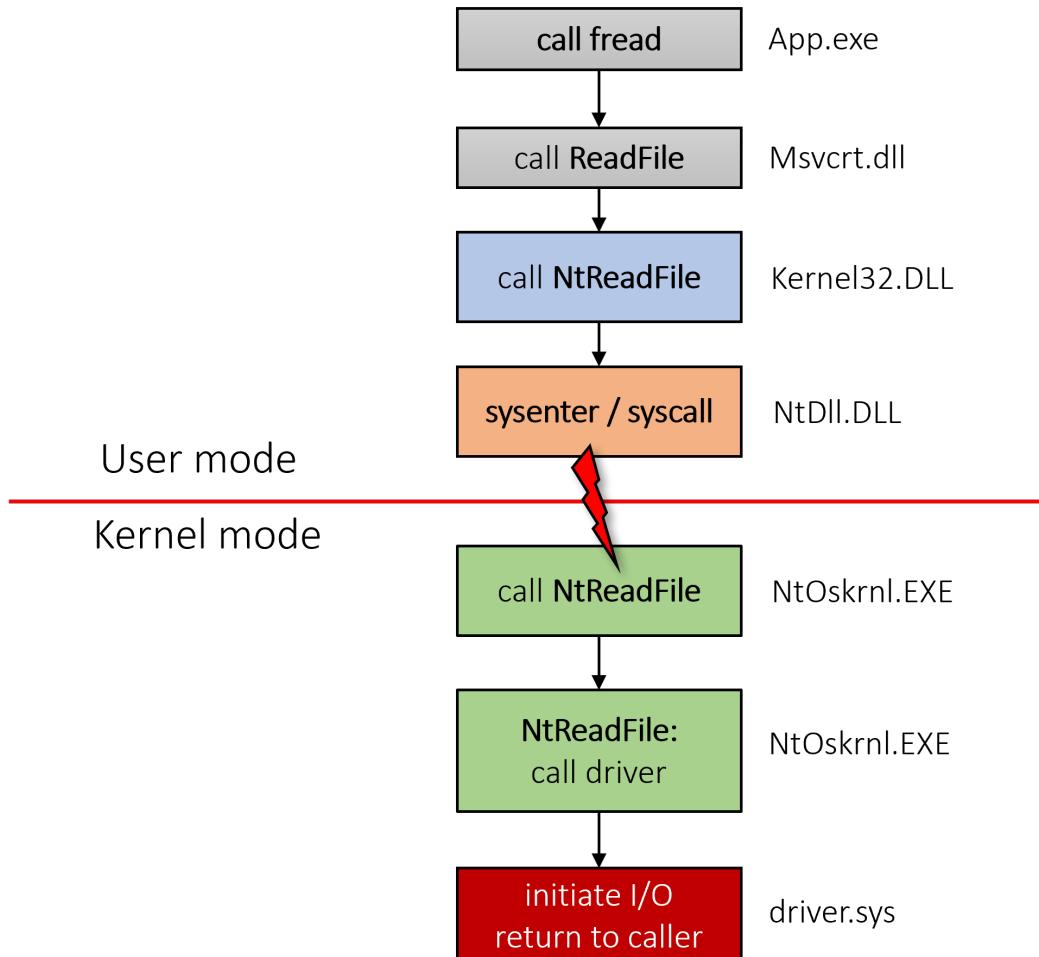


Figure 1-8: System service function call flow

General System Architecture

Figure 1-9 shows the general architecture of Windows, comprising of user-mode and kernel-mode components.

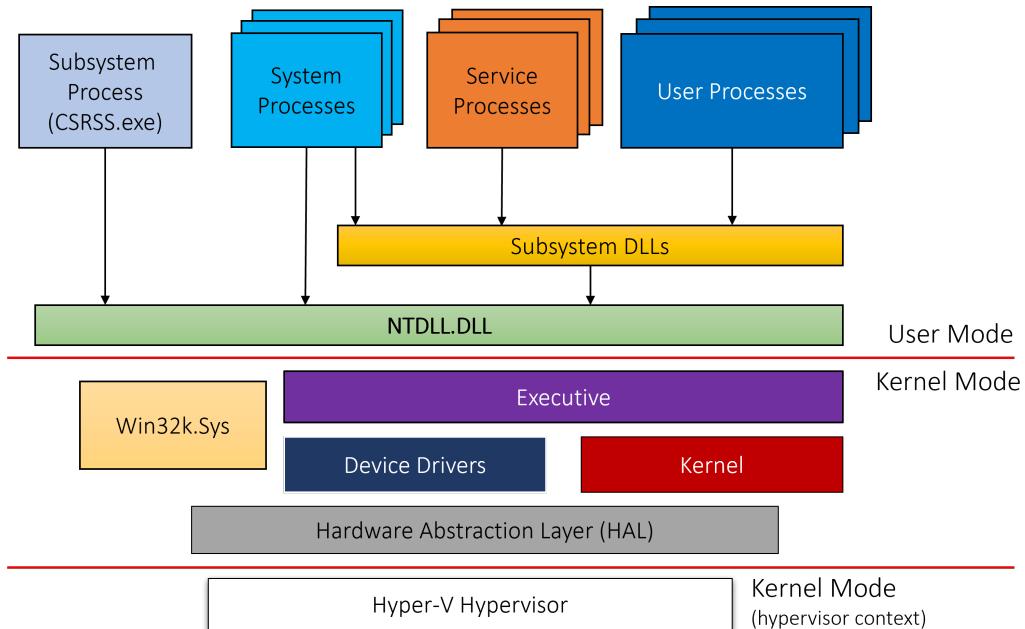


Figure 1-9: Windows system architecture

Here's a quick rundown of the named boxes appearing in figure 1-9:

- **User processes**

These are normal processes based on image files, executing on the system, such as instances of *Notepad.exe*, *cmd.exe*, *explorer.exe*, and so on.

- **Subsystem DLLs**

Subsystem DLLs are dynamic link libraries (DLLs) that implement the API of a subsystem. A subsystem is a particular view of the capabilities exposed by the kernel. Technically, starting from Windows 8.1, there is only a single subsystem – the Windows Subsystem. The subsystem DLLs include well-known files, such as *kernel32.dll*, *user32.dll*, *gdi32.dll*, *advapi32.dll*, *combase.dll*, and many others. These include mostly the officially documented API of Windows.

- **NTDLL.DLL**

A system-wide DLL, implementing the Windows native API. This is the lowest layer of code which is still in user mode. Its most important role is to make the transition to kernel mode for system call invocation. NTDLL also implements the Heap Manager, the Image Loader and some part of the user mode thread pool.

- **Service Processes**

Service processes are normal Windows processes that communicate with the Service Control Manager (SCM, implemented in services.exe) and allow some control over their lifetime. The SCM can start, stop, pause, resume and send other messages to services. Services typically execute under one of the special Windows accounts – local system, network service or local service.

- **Executive**

The Executive is the upper layer of NtOskrnl.exe (the “kernel”). It hosts most of the code that is in kernel mode. It includes mostly the various “managers”: Object Manager, Memory Manager, I/O Manager, Plug & Play Manager, Power Manager, Configuration Manager, etc. It’s by far larger than the lower Kernel layer.

- **Kernel**

The Kernel layer implements the most fundamental and time-sensitive parts of kernel-mode OS code. This includes thread scheduling, interrupt and exception dispatching, and implementation of various kernel primitives such as mutexes and semaphores. Some of the kernel code is written in CPU-specific machine language for efficiency and for getting direct access to CPU-specific details.

- **Device Drivers**

Device drivers are loadable kernel modules. Their code executes in kernel mode and so has the full power of the kernel. This book is dedicated to writing certain types of kernel drivers.

- **Win32k.sys**

This is the kernel-mode component of the Windows subsystem. Essentially, it’s a kernel module (driver) that handles the user interface part of Windows and the classic *Graphics Device Interface* (GDI) APIs. This means that all windowing operations (CreateWindowEx, GetMessage, PostMessage, etc.) are handled by this component. The rest of the system has little-to-none knowledge of UI.

- **Hardware Abstraction Layer (HAL)**

The HAL is a software abstraction layer over the hardware closest to the CPU. It allows device drivers to use APIs that do not require detailed and specific knowledge of things like Interrupt Controllers or DMA controller. Naturally, this layer is mostly useful for device drivers written to handle hardware devices.

- **System Processes**

System processes is an umbrella term used to describe processes that are typically “just there”, doing their thing where normally these processes are not communicated with directly. They are important nonetheless, and some in fact, critical to the system’s well-being. Terminating some of them is fatal and causes a system crash. Some of the system processes are native processes, meaning they use the native API only (the API implemented by NTDLL). Example system processes include *Smss.exe*, *Lsass.exe*, *Winlogon.exe*, and *Services.exe*.

- **Subsystem Process**

The Windows subsystem process, running the image *Csrss.exe*, can be viewed as a helper to the kernel for managing processes running under the Windows subsystem. It is a critical process, meaning if killed, the system would crash. There is one *Csrss.exe* instance per session, so on a standard system two instances would exist – one for session 0 and one for the logged-on user session (typically 1). Although *Csrss.exe* is the “manager” of the Windows subsystem (the only one left these days), its importance goes beyond just this role.

- **Hyper-V Hypervisor**

The Hyper-V hypervisor exists on Windows 10 and server 2016 (and later) systems if they support *Virtualization Based Security* (VBS). VBS provides an extra layer of security, where the normal OS is a virtual machine controlled by Hyper-V. Two distinct *Virtual Trust Levels* (VTLs) are defined, where VTL 0 consists of the normal user-mode/kernel-mode we know of, and VTL 1 contains the secure kernel and *Isolated User Mode* (IUM). VBS is beyond the scope of this book. For more information, check out the *Windows Internals* book and/or the Microsoft documentation.



Windows 10 version 1607 introduced the *Windows Subsystem for Linux* (WSL). Although this may look like yet another subsystem, like the old POSIX and OS/2 subsystems supported by Windows, it is not like that at all. The old subsystems were able to execute POSIX and OS/2 apps if these were compiled using a Windows compiler to use the PE format and Windows system calls. WSL, on the other hand, has no such requirement. Existing executables from Linux (stored in ELF format) can be run as-is on Windows, without any recompilation.

To make something like this work, a new process type was created – the Pico process together with a Pico provider. Briefly, a Pico process is an empty address space (minimal process) that is used for WSL processes, where every system call (Linux system call) must be intercepted and translated to the Windows system call(s) equivalent using that Pico provider (a device driver). There is a true Linux (the user-mode part) installed on the Windows machine.

The above description is for WSL version 1. Starting with Windows 10 version 2004, Windows supports a new version of WSL known as WSL 2. WSL 2 is not based on pico processes anymore. Instead, it's based on a hybrid virtual machine technology that allows installing a full Linux system (including the Linux kernel), but still see and share the Windows machine's resources, such as the file system. WSL 2 is faster than WSL 1 and solves some edge cases that didn't work well in WSL 1, thanks to the real Linux kernel handling Linux system calls.

Handles and Objects

The Windows kernel exposes various types of objects for use by user-mode processes, the kernel itself and kernel-mode drivers. Instances of these types are data structures in system space, created by the Object Manager (part of the executive) when requested to do so by user-mode or kernel-mode code. Objects are reference counted – only when the last reference to the object is released will the object be destroyed and freed from memory.

Since these object instances reside in system space, they cannot be accessed directly by user mode. User mode must use an indirect access mechanism, known as handles. A handle is an index to an entry in a table maintained on a process basis, stored in kernel space, that points to a kernel object residing in system space. There are various *Create** and *Open** functions to create/open objects and retrieve back handles to these objects. For example, the *CreateMutex* user-mode function allows creating or opening a mutex (depending on whether the object is named and exists). If successful, the

function returns a handle to the object. A return value of zero means an invalid handle (and a function call failure). The `OpenMutex` function, on the other hand, tries to open a handle to a named mutex. If the mutex with that name does not exist, the function fails and returns null (0).

Kernel (and driver) code can use either a handle or a direct pointer to an object. The choice is usually based on the API the code wants to call. In some cases, a handle given by user mode to the driver must be turned into a pointer with the `ObReferenceObjectByHandle` function. We'll discuss these details in a later chapter.



Most functions return null (zero) on failure, but some do not. Most notably, the `CreateFile` function returns `INVALID_HANDLE_VALUE` (-1) if it fails.

Handle values are multiples of 4, where the first valid handle is 4; Zero is never a valid handle value.

Kernel-mode code can use handles when creating/opening objects, but they can also use direct pointers to kernel objects. This is typically done when a certain API demands it. Kernel code can get a pointer to an object given a valid handle using the `ObReferenceObjectByHandle` function. If successful, the reference count on the object is incremented, so there is no danger that if the user-mode client holding the handle decided to close it while kernel code holds a pointer to the object would now hold a dangling pointer. The object is safe to access regardless of the handle-holder until the kernel code calls `ObDereferenceObject`, which decrements the reference count; if the kernel code missed this call, that's a resource leak which will only be resolved in the next system boot.

All objects are reference counted. The object manager maintains a handle count and total reference count for objects. Once an object is no longer needed, its client should close the handle (if a handle was used to access the object) or dereference the object (if kernel client using a pointer). From that point on, the code should consider its handle(pointer) to be invalid. The Object Manager will destroy the object if its reference count reaches zero.

Each object points to an object type, which holds information on the type itself, meaning there is a single type object for each type of object. These are also exposed as exported global kernel variables, some of which are defined in the kernel headers and are needed in certain cases, as we'll see in later chapters.

Object Names

Some types of objects can have names. These names can be used to open objects by name with a suitable `Open` function. Note that not all objects have names; for example, processes and threads don't have names – they have IDs. That's why the `OpenProcess` and `OpenThread` functions require a process/thread identifier (a number) rather than a string-base name. Another somewhat weird case of an object that does not have a name is a file. The file name is not the object's name – these are different concepts.



Threads appear to have a name (starting from Windows 10), that can be set with the user-mode API `SetThreadDescription`. This is not, however, a true name, but rather a friendly name/description most useful in debugging, as Visual Studio shows a thread's description, if there is any.

From user-mode code, calling a *Create* function with a name creates the object with that name if an object with that name does not exist, but if it exists it just opens the existing object. In the latter case, calling `GetLastError` returns `ERROR_ALREADY_EXISTS`, indicating this is not a new object, and the returned handle is yet another handle to an existing object.

The name provided to a *Create* function is not actually the final name of the object. It's prepended with `\Sessions\x\BaseNamedObjects\` where x is the session ID of the caller. If the session is zero, the name is prepended with `\BaseNamedObjects\`. If the caller happens to be running in an AppContainer (typically a Universal Windows Platform process), then the prepended string is more complex and consists of the unique AppContainer SID: `\Sessions\x\AppContainerNamedObjects\{AppContainerSID}`.

All the above means is that object names are session-relative (and in the case of AppContainer – package relative). If an object must be shared across sessions it can be created in session 0 by prepending the object name with `Global\`; for example, creating a mutex with the `CreateMutex` function named `Global\MyMutex` will create it under `\BaseNamedObjects`. Note that AppContainers do not have the power to use session 0 object namespace.

This hierarchy can be viewed with the Sysinternals `WinObj` tool (run elevated) as shown in figure 1-10.

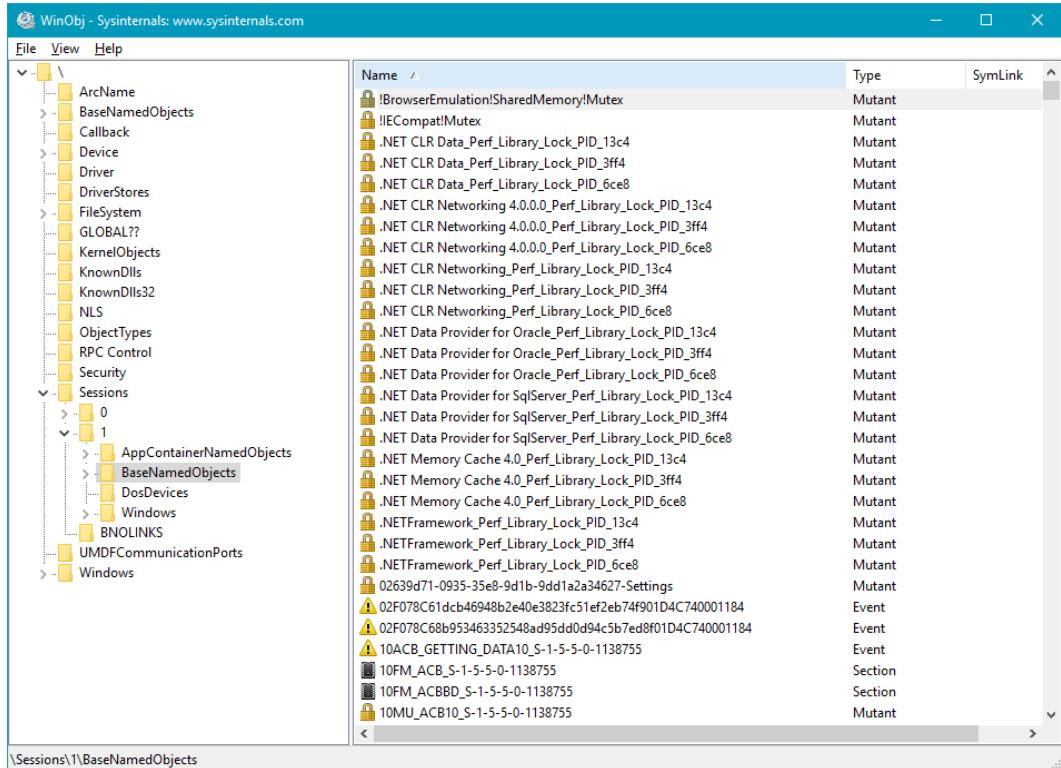


Figure 1-10: Sysinternals WinObj tool

The view shown in figure 1-10 is the object manager namespace, comprising of a hierarchy of named objects. This entire structure is held in memory and manipulated by the Object Manager (part of the *Executive*) as required. Note that unnamed objects are not part of this structure, meaning the objects seen in *WinObj* do not comprise all the existing objects, but rather all the objects that were created with a name.

Every process has a private handle table to kernel objects (whether named or not), which can be viewed with the *Process Explorer* and/or *Handles* Sysinternals tools. A screenshot of Process Explorer showing handles in some process is shown in figure 1-11. The default columns shown in the handles view are the object type and name only. However, there are other columns available, as shown in figure 1-11.

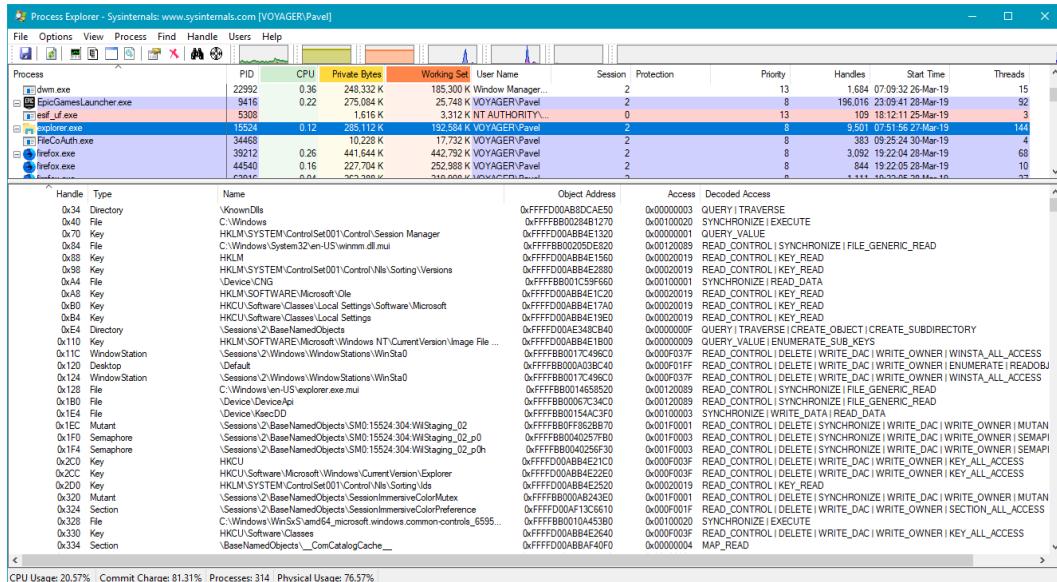


Figure 1-11: Viewing handles in processes with Process Explorer

By default, Process Explorer shows only handles for objects, which have names (according to *Process Explorer*'s definition of a name, discussed shortly). To view all handles in a process, select *Show Unnamed Handles and Mappings* from *Process Explorer*'s *View* menu.

The various columns in the handle view provide more information for each handle. The handle value and the object type are self explanatory. The name column is tricky. It shows true object names for Mutexes (Mutants), Semaphores, Events, Sections, ALPC Ports, Jobs, Timers, Directory (object manager Directories, not file system directories), and other, less used object types. Yet others are shown with a name that has a different meaning than a true named object:

- Process and Thread objects, the name is shown as their unique ID.
- For File objects, it shows the file name (or device name) pointed to by the file object. It's not the same as an object's name, as there is no way to get a handle to a file object given the file name - only a new file object may be created that accesses the same underlying file or device (assuming sharing settings for the original file object allow it).
- (Registry) Key objects names are shown with the path to the registry key. This is not a name, for the same reasoning as for file objects.
- Token object names are shown with the user name stored in the token.

Accessing Existing Objects

The Access column in *Process Explorer*'s handles view shows the access mask which was used to open or create the handle. This access mask is key to what operations are allowed to be performed with a specific handle. For example, if client code wants to terminate a process, it must call the *OpenProcess* function first, to obtain a handle to the required process with an access mask of (at least) *PROCESS_TERMINATE*, otherwise there is no way to terminate the process with that handle. If the call succeeds,

then the call to `TerminateProcess` is bound to succeed.

Here's a user-mode example for terminating a process given its process ID:

```
bool KillProcess(DWORD pid) {
    //
    // open a powerful-enough handle to the process
    //
    HANDLE hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pid);
    if (!hProcess)
        return false;

    //
    // now kill it with some arbitrary exit code
    //
    BOOL success = TerminateProcess(hProcess, 1);

    //
    // close the handle
    //
    CloseHandle(hProcess);

    return success != FALSE;
}
```

The *Decoded Access* column provides a textual description of the access mask (for some object types), making it easier to identify the exact access allowed for a particular handle.

Double-clicking a handle entry (or right-clicking and selecting *Properties*) shows some of the object's properties. Figure 1-12 shows a screenshot of an example event object properties.

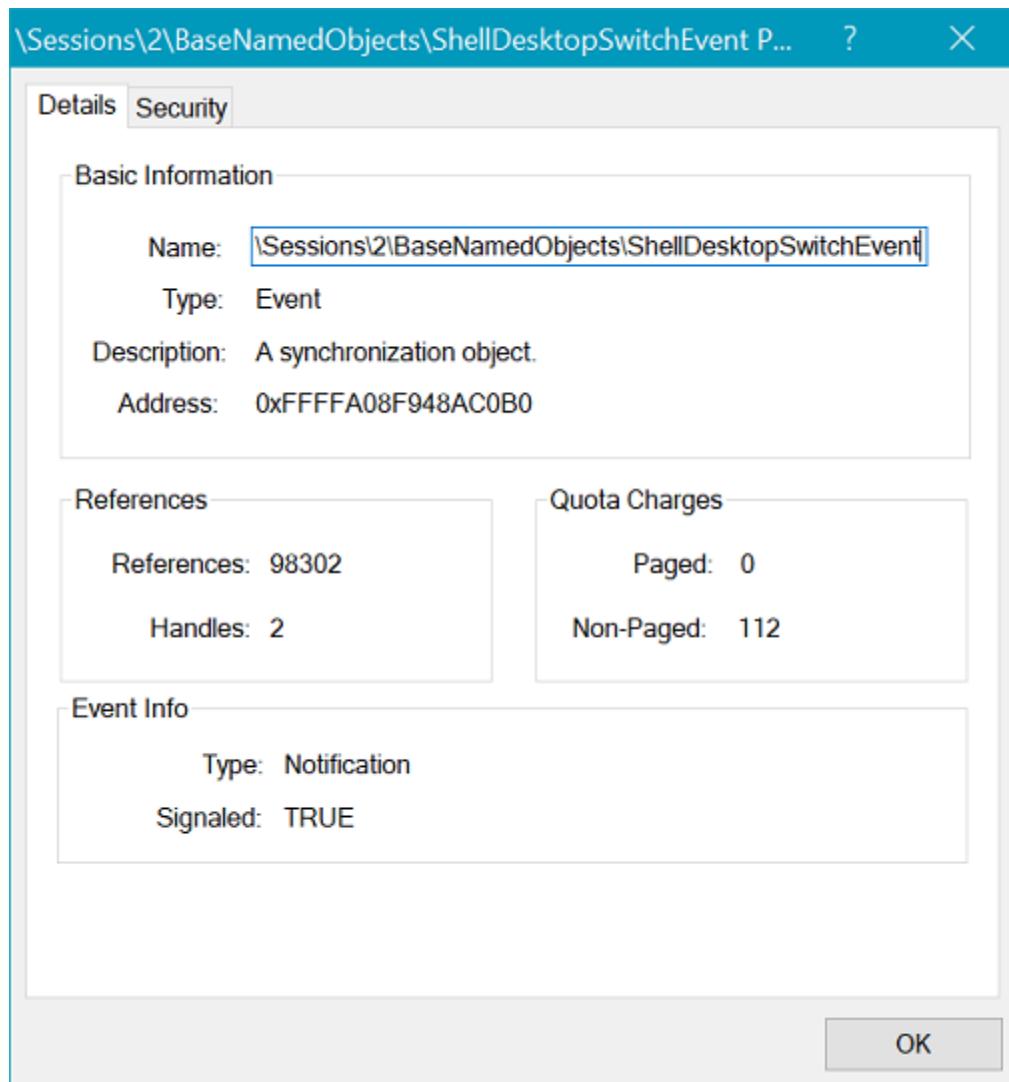


Figure 1-12: Object properties in Process Explorer

Notice that the dialog shown in figure 1-12 is for the object's properties, rather than the handle's. In other words, looking at an object's properties from any handle that points to the same object shows the same information.

The properties in figure 1-12 include the object's name (if any), its type, a short description, its address in kernel memory, the number of open handles, and some specific object information, such as the state and type of the event object shown. Note that the *References* shown do not indicate the actual number

of outstanding references to the object (it does prior to Windows 8.1). A proper way to see the actual reference count for the object is to use the kernel debugger's !trueref command, as shown here:

```
1kd> !object 0xFFFFA08F948AC0B0
Object: fffffa08f948ac0b0  Type: (fffffa08f684df140) Event
    ObjectHeader: fffffa08f948ac080 (new version)
    HandleCount: 2  PointerCount: 65535
    Directory Object: fffff90839b63a700  Name: ShellDesktopSwitchEvent
1kd> !trueref fffffa08f948ac0b0
fffffa08f948ac0b0: HandleCount: 2  PointerCount: 65535  RealPointerCount: 3
```

We'll take a closer look at the attributes of objects and the kernel debugger in later chapters.

In the next chapter, we'll start writing a very simple driver to show and use many of the tools we'll need later in this book.

Chapter 2: Getting Started with Kernel Development

This chapter deals with the fundamentals needed to get up and running with kernel driver development. During the course of this chapter, you'll install the necessary tools and write a very basic driver that can be loaded and unloaded.

In this chapter:

- Installing the Tools
 - Creating a Driver Project
 - The `DriverEntry` and `Unload` routines
 - Deploying the Driver
 - Simple Tracing
-

Installing the Tools

In the old days (pre-2012), the process of developing and building drivers included using a dedicated build tool from the Device Driver Kit (DDK), without having an integrated development experience developers were used to when developing user-mode applications. There were some workarounds, but none of them was perfect nor officially supported by Microsoft.

Fortunately, starting with Visual Studio 2012 and Windows Driver Kit 8, Microsoft officially supports building drivers with Visual Studio (with `msbuild`), without the need to use a separate compiler and build tools.

To get started with driver development, the following tools must be installed (in this order) on your development machine:

- Visual Studio 2019 with the latest updates. Make sure the C++ workload is selected during installation. Note that any SKU will do, including the free Community edition.
- Windows 11 SDK (generally, the latest is recommended). Make sure at least the *Debugging Tools for Windows* item is selected during installation.
- Windows 11 Driver Kit (WDK) - it supports building drivers for Windows 7 and later versions of Windows. Make sure the wizard installs the project templates for Visual Studio at the end of the installation.

- The *Sysinternals* tools, which are invaluable in any “internals” work, can be downloaded for free from <http://www.sysinternals.com>. Click on *Sysinternals Suite* on the left of that web page and download the Sysinternals Suite zip file. Unzip to any folder, and the tools are ready to go.



The SDK and WDK versions must match. Follow the guidelines in the WDK download page to load the corresponding SDK with the WDK.

A quick way to make sure the WDK templates are installed correctly is to open Visual Studio and select New Project and look for driver projects, such as “Empty WDM Driver”.

Creating a Driver Project

With the above installations in place, a new driver project can be created. The template you’ll use in this section is “WDM Empty Driver”. Figure 2-1 shows what the New Project dialog looks like for this type of driver in Visual Studio 2019. Figure 2-2 shows the same initial wizard with Visual Studio 2019 if the *Classic Project Dialog* extension is installed and enabled. The project in both figures is named “Sample”.

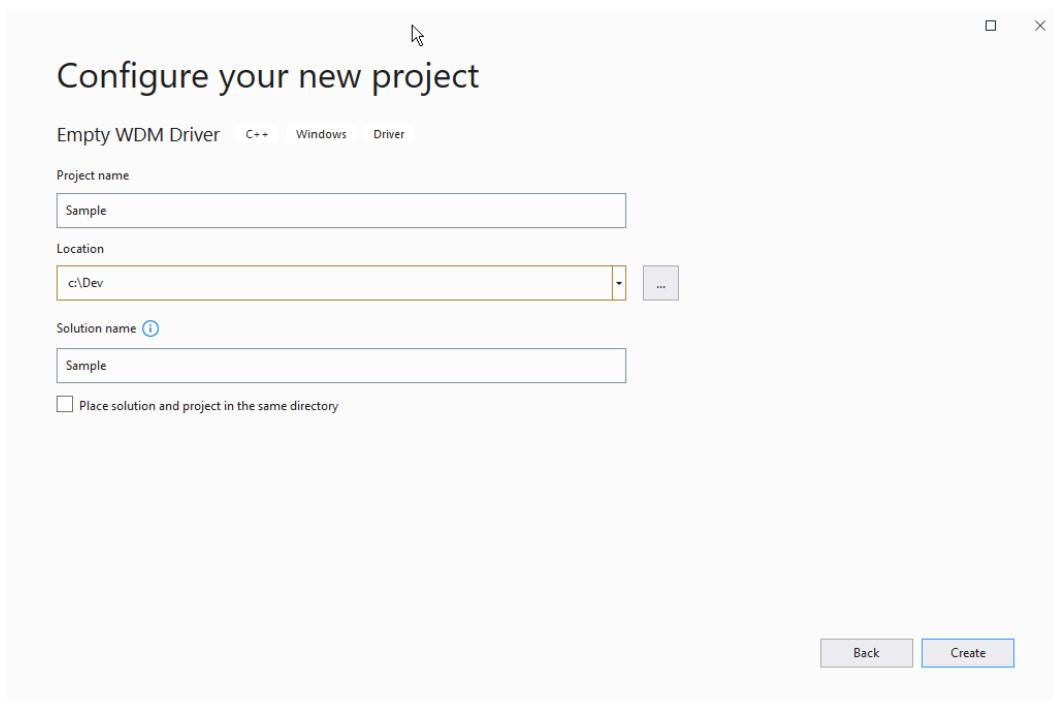


Figure 2-1: New WDM Driver Project in Visual Studio 2019

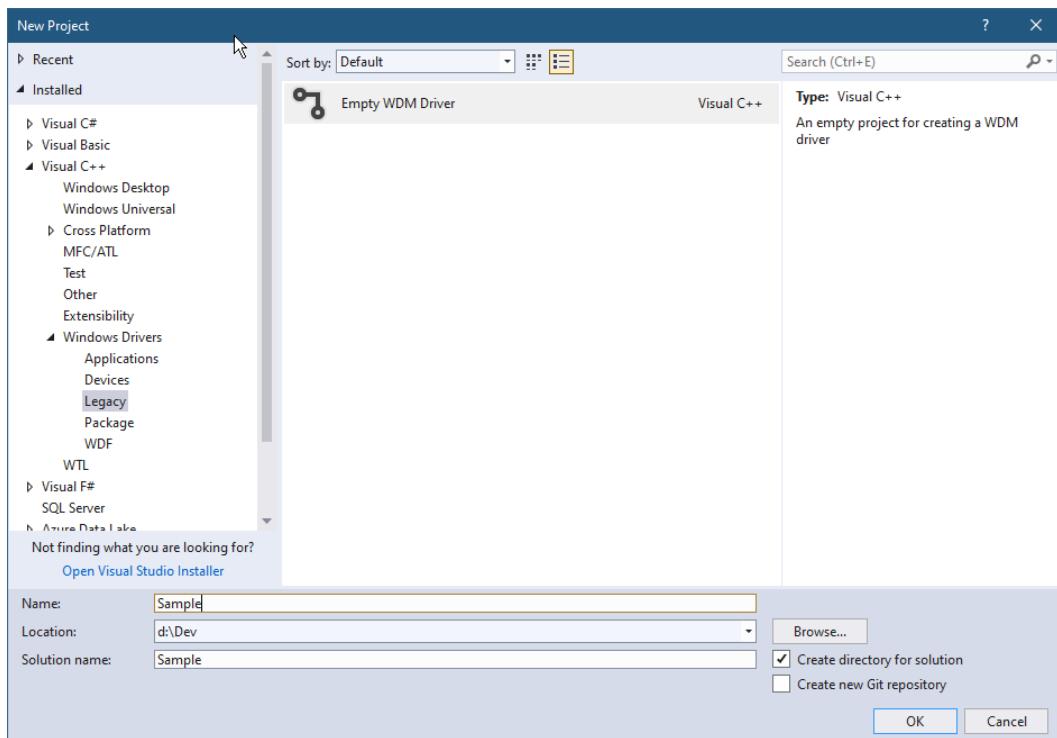


Figure 2-2: New WDM Driver Project in Visual Studio 2019 with the *Classic Project Dialog* extension

Once the project is created, the *Solution Explorer* shows a single file within the *Driver Files* filter - `Sample.inf`. You won't need this file in this example, so simply delete it (right-click and select *Remove* or press the *Del* key).

Now it's time to add a source file. Right-click the *Source Files* node in *Solution Explorer* and select *Add / New Item...* from the File menu. Select a C++ source file and name it `Sample.cpp`. Click *OK* to create it.

The DriverEntry and Unload Routines

Every driver has an entry point called `DriverEntry` by default. This can be considered the “main” function of the driver, comparable to the classic `main` of a user-mode application. This function is called by a system thread at IRQL `PASSIVE_LEVEL` (0). (IRQLs are discussed in detail in chapter 8.)

`DriverEntry` has a predefined prototype, shown here:

```
NTSTATUS DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath);
```

The `_In_` annotations are part of the *Source (Code) Annotation Language* (SAL). These annotations are transparent to the compiler, but provide metadata useful for human readers and static analysis

tools. I may remove these annotations in code samples to make it easier to read, but you should use SAL annotations whenever possible.

A minimal DriverEntry routine could just return a successful status, like so:

```
NTSTATUS DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject, _In_ PUNICODE_STRING RegistryPath) {
    return STATUS_SUCCESS;
}
```

This code would not yet compile. First, you'll need to include a header that has the required definitions for the types present in DriverEntry. Here's one possibility:

```
#include <ntddk.h>
```

Now the code has a better chance of compiling, but would still fail. One reason is that by default, the compiler is set to treat warnings as errors, and the function does not make use of its given arguments. Removing *treat warnings as errors* from the compiler's options is not recommended, as some warnings may be errors in disguise. These warnings can be resolved by removing the argument names entirely (or commenting them out), which is fine for C++ files. There is another, more "classic" way to solve this, which is to use the UNREFERENCED_PARAMETER macro:

```
NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);

    return STATUS_SUCCESS;
}
```

As it turns out, this macro actually references the argument given just by writing its value as is, and this shuts the compiler up, making the argument technically "referenced".

Building the project now compiles fine, but causes a linker error. The DriverEntry function must have C-linkage, which is not the default in C++ compilation. Here's the final version of a successful build of the driver consisting of a DriverEntry function only:

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);

    return STATUS_SUCCESS;
}
```

At some point, the driver may be unloaded. At that time, anything done in the `DriverEntry` function must be undone. Failure to do so creates a leak, which the kernel will not clean up until the next reboot. Drivers can have an `Unload` routine that is automatically called before the driver is unloaded from memory. Its pointer must be set using the `DriverUnload` member of the driver object:

```
DriverObject->DriverUnload = SampleUnload;
```

The `Unload` routine accepts the driver object (the same one passed to `DriverEntry`) and returns `void`. As our sample driver has done nothing in terms of resource allocation in `DriverEntry`, there is nothing to do in the `Unload` routine, so we can leave it empty for now:

```
void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
}
```

Here is the complete driver source at this point:

```
#include <ntddk.h>

void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
}

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    return STATUS_SUCCESS;
}
```

Deploying the Driver

Now that we have a successfully compiled `Sample.sys` driver file, let's install it on a system and then load it. Normally, you would install and load a driver on a virtual machine, to remove the risk of crashing your primary machine. Feel free to do so, or take the slight risk with this minimalist driver.

Installing a software driver, just like installing a user-mode service, requires calling the `CreateService` API with proper arguments, or using a comparable tool. One of the well-known tools for this purpose is `Sc.exe` (short for *Service Control*), a built-in Windows tool for managing services. We'll use this tool to install and then load the driver. Note that installation and loading of drivers is a privileged operation, normally available for administrators.

Open an elevated command window and type the following (the last part should be the path on your system where the `SYS` file resides):

```
sc create sample type= kernel binPath= c:\dev\sample\x64\debug\sample.sys
```

Note there is no space between *type* and the equal sign, and there is a space between the equal sign and *kernel*; same goes for the second part.

If all goes well, the output should indicate success. To test the installation, you can open the registry editor (*regedit.exe*) and look for the driver details at *HKLM\System\CurrentControlSet\Services\Sample*. Figure 2-3 shows a screenshot of the registry editor after the previous command.

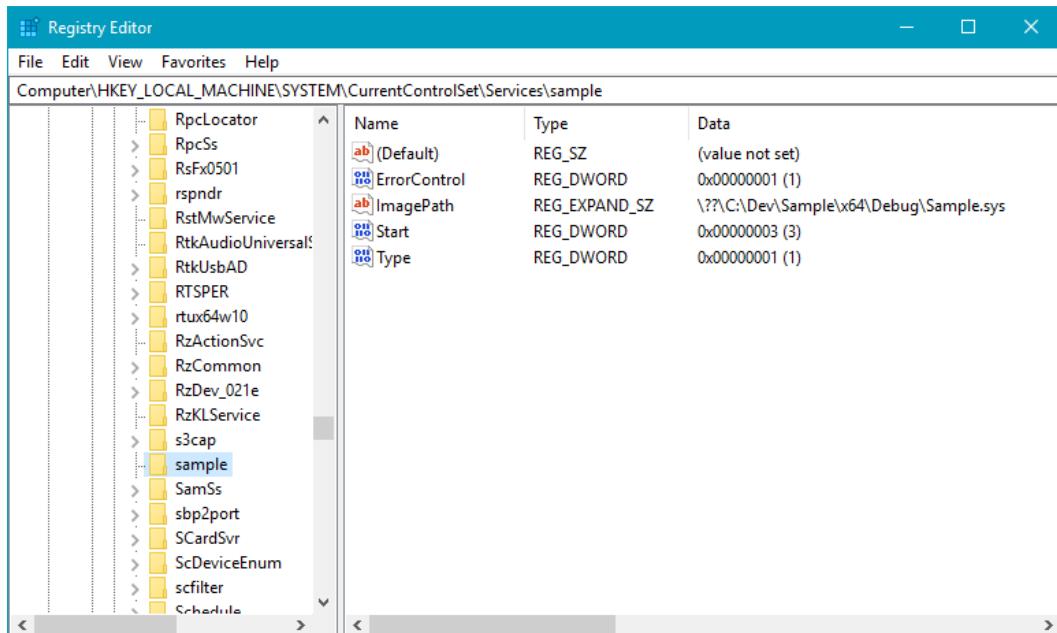


Figure 2-3: Registry for an installed driver

To load the driver, we can use the *Sc.exe* tool again, this time with the *start* option, which uses the *StartService* API to load the driver (the same API used to load services). However, on 64 bit systems drivers must be signed, and so normally the following command would fail:

```
sc start sample
```

Since it's inconvenient to sign a driver during development (maybe even not possible if you don't have a proper certificate), a better option is to put the system into test signing mode. In this mode, unsigned drivers can be loaded without a hitch.

With an elevated command window, test signing can be turned on like so:

```
bcdedit /set testsigning on
```

Unfortunately, this command requires a reboot to take effect. Once rebooted, the previous start command should succeed.



If you are testing on a Windows 10 (or later) system with *Secure Boot* enabled, changing the test signing mode will fail. This is one of the settings protected by Secure Boot (local kernel debugging is also protected by Secure Boot). If you can't disable Secure Boot through BIOS setting, because of IT policy or some other reason, your best option is to test on a virtual machine.

There is yet another setting that you may need to specify if you intend to test the driver on pre-Windows 10 machine when using Visual Studio 2019 (or earlier) only. In this case, you have to set the target OS version in the project properties dialog, as shown in figure 2-4. Notice that I have selected all configurations and all platforms, so that when switching configurations (Debug/Release) or platforms (x86/x64/ARM/ARM64), the setting is maintained.

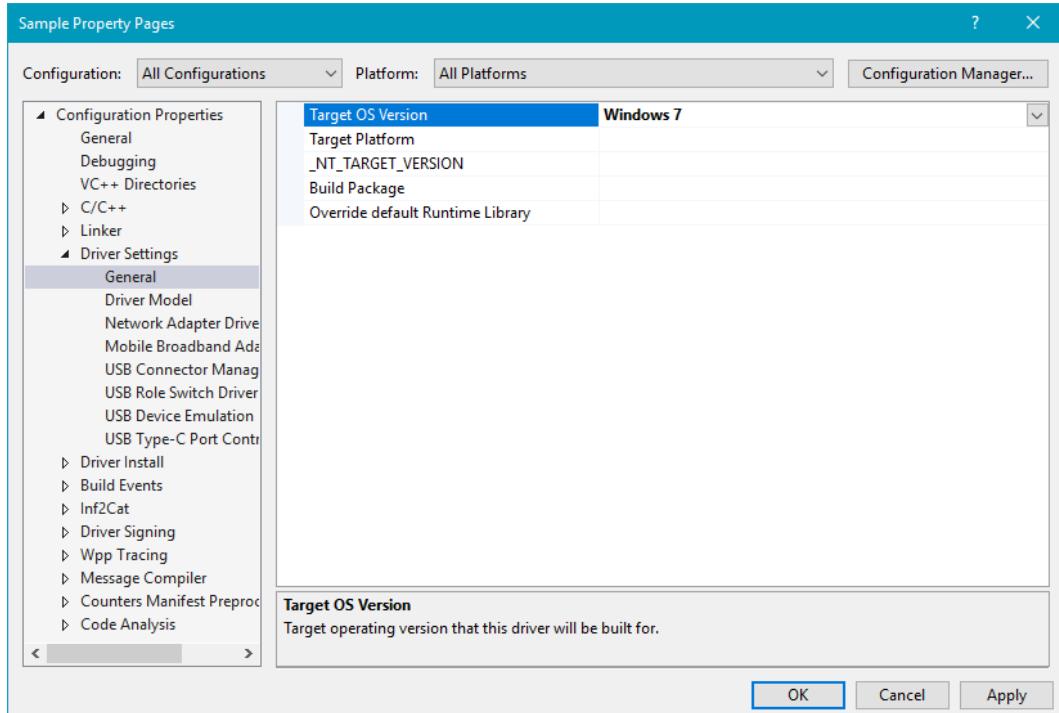


Figure 2-4: Setting Target OS Platform in the project properties

Once test signing mode is on, and the driver is loaded, this is the output you should see:

```
c:/>sc start sample
SERVICE_NAME: sample
    TYPE               : 1   KERNEL_DRIVER
    STATE              : 4   RUNNING
                           (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0   (0x0)
    SERVICE_EXIT_CODE : 0   (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x0
    PID                : 0
    FLAGS              :
```



With Visual Studio 2022, you can only build drivers for Windows 10 and later.

This means everything is well, and the driver is loaded. To confirm, we can open *Process Explorer* and find the *Sample.Sys* driver image file. Figure 2-5 shows the details of the sample driver image loaded into system space.

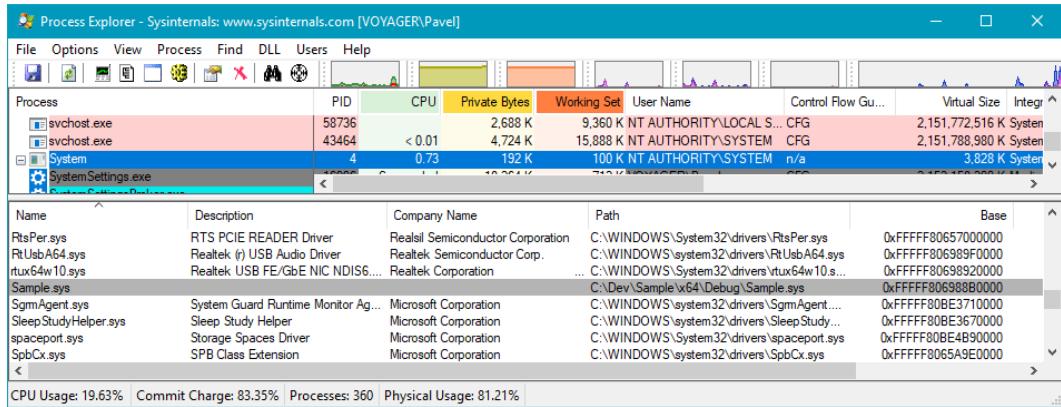


Figure 2-5: sample driver image loaded into system space

At this point, we can unload the driver using the following command:

```
sc stop sample
```

Behind the scenes, *sc.exe* calls the *ControlService* API with the *SERVICE_CONTROL_STOP* value. Unloading the driver causes the *Unload* routine to be called, which at this time does nothing. You can verify the driver is indeed unloaded by looking at *Process Explorer* again; the driver image entry should not be there anymore.

Simple Tracing

How can we know for sure that the `DriverEntry` and `Unload` routines actually executed? Let's add basic tracing to these functions. Drivers can use the `DbgPrint` function to output `printf`-style text that can be viewed using the kernel debugger, or some other tool.

Here is updated versions for `DriverEntry` and the `Unload` routine that use `DbgPrint` to trace the fact their code executed:

```
void SampleUnload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    DbgPrint("Sample driver Unload called\n");
}

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;

    DbgPrint("Sample driver initialized successfully\n");

    return STATUS_SUCCESS;
}
```

A more typical approach is to have these outputs in Debug builds only. This is because `Dbgprint` has some overhead that you may want to avoid in Release builds. `KdPrint` is a macro that is only compiled in Debug builds and calls the underlying `DbgPrint` kernel API. Here is a revised version that uses `KdPrint`:

```
void SampleUnload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    KdPrint(("Sample driver Unload called\n"));
}

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = SampleUnload;
```

```

KdPrint(("Sample driver initialized successfully\n"));

return STATUS_SUCCESS;
}

```

Notice the double parenthesis when using KdPrint. This is required because KdPrint is a macro, but apparently accepts any number of arguments, a-la printf. Since macros cannot receive a variable number of parameters, a compiler trick is used to call the DbgPrint function that does accept a variable number of parameters.

With these statements in place, we would like to load the driver again and see these messages. We'll use a kernel debugger in chapter 4, but for now we'll use a useful Sysinternals tool named *DebugView*. Before running *DebugView*, you'll need to make some preparations. First, starting with Windows Vista, DbgPrint output is not actually generated unless a certain value is in the registry. You'll have to add a key named *Debug Print Filter* under *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager* (the key typically does not exist). Within this new key, add a DWORD value named *DEFAULT* (not the default value that exists in any key) and set its value to 8 (technically, any value with bit 3 set will do). Figure 2-6 shows the setting in *RegEdit*. Unfortunately, you'll have to restart the system for this setting to take effect.

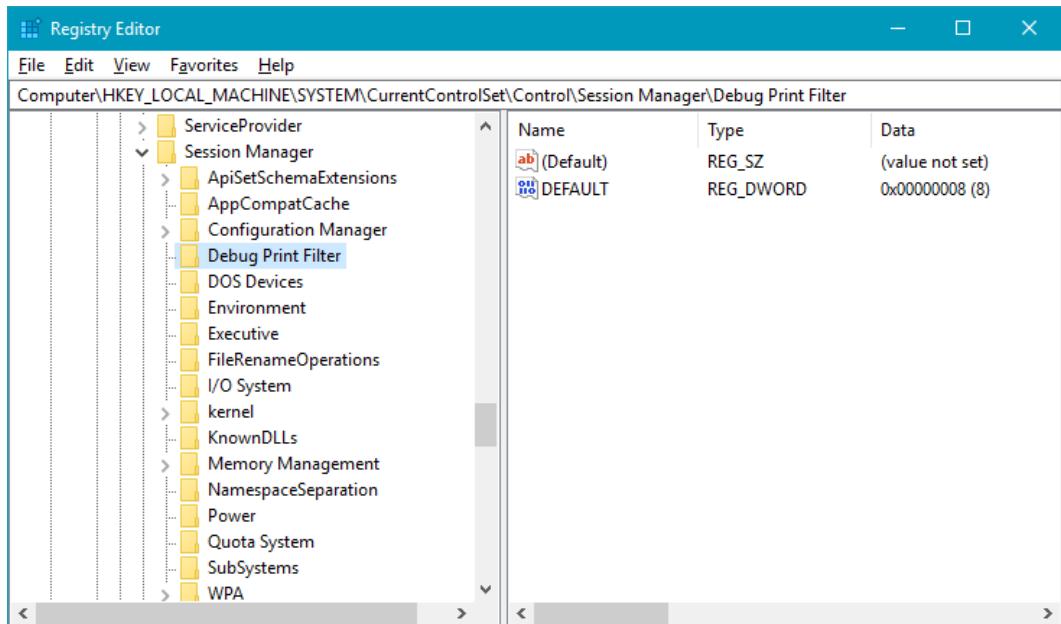


Figure 2-6: Debug Print Filter key in the registry

Once this setting has been applied, run *DebugView* (*DbgView.exe*) elevated. In the *Options* menu, make sure *Capture Kernel* is selected (or press *Ctrl+K*). You can safely deselect *Capture Win32* and *Capture Global Win32*, so that user-mode output from various processes does not clutter the display.



DebugView is able to show kernel debug output even without the Registry value shown in figure 2-6 if you select *Enable Verbose Kernel Output* from its *Capture* menu. However, it seems this option does not work on Windows 11, and the Registry setting is necessary.

Build the driver, if you haven't already. Now you can load the driver again from an elevated command window (`sc start sample`). You should see output in *DebugView* as shown in figure 2-7. If you unload the driver, you'll see another message appearing because the Unload routine was called. (The third output line is from another driver and has nothing to do with our sample driver)

A screenshot of the DebugView application window. The title bar reads "DebugView on \\VOYAGER (local)". The menu bar includes File, Edit, Capture, Options, Computer, and Help. Below the menu is a toolbar with various icons. The main pane displays a table of kernel debug output. The columns are labeled "#", "Time", and "Debug Print". The data shows three entries:

#	Time	Debug Print
1	21:17:44	Sample driver initialized successfully
2	21:17:48	Sample driver Unload called
3	21:17:48	received arp request through adapter 1, for 158.14.168.192

A scroll bar is visible on the right side of the main pane.

Figure 2-7: Sysinternals *DebugView* Output



Add code to the sample `DriverEntry` to output the Windows OS version: major, minor, and build number. Use the `RtlGetVersion` function to retrieve the information. Check the results with *DebugView*.

Summary

We've seen the tools you need to have for kernel development and wrote a very minimalistic driver to prove the basic tools work. In the next chapter, we'll look at the fundamental building blocks of kernel APIs, concepts, and fundamental structures.

Chapter 3: Kernel Programming Basics

In this chapter, we'll dig deeper into kernel APIs, structures, and definitions. We'll also examine some of the mechanisms that invoke code in a driver. Finally, we'll put all that knowledge together to create our first functional driver and client application.

In this chapter:

- General Kernel Programming Guidelines
 - Debug vs. Release Builds
 - The Kernel API
 - Functions and Error Codes
 - Strings
 - Dynamic Memory Allocation
 - Linked Lists
 - Object Attributes
 - The Driver Object
 - Device Objects
-

General Kernel Programming Guidelines

Developing kernel drivers requires the Windows Driver Kit (WDK), where the appropriate headers and libraries needed are located. The kernel APIs consist of C functions, very similar in essence to user-mode APIs. There are several differences, however. Table 3-1 summarizes the important differences between user-mode programming and kernel-mode programming.

Table 3-1: Differences between user mode and kernel mode development

	User Mode	Kernel Mode
Unhandled Exceptions	Unhandled exceptions crash the process	Unhandled exceptions crash the system
Termination	When a process terminates, all private memory and resources are freed automatically	If a driver unloads without freeing everything it was using, there is a leak, only resolved in the next boot
Return values	API errors are sometimes ignored	Should (almost) never ignore errors

Table 3-1: Differences between user mode and kernel mode development

User Mode	Kernel Mode
IRQL	Always PASSIVE_LEVEL (0)
Bad coding	Typically localized to the process
Testing and Debugging	Typical testing and debugging done on the developer's machine
Libraries	Can use almost any C/C++ library (e.g. STL, boost)
Exception Handling	Can use C++ exceptions or Structured Exception Handling (SEH)
C++ Usage	Full C++ runtime available
	No C++ runtime

Unhandled Exceptions

Exceptions occurring in user-mode that are not caught by the program cause the process to terminate prematurely. Kernel-mode code, on the other hand, being implicitly trusted, cannot recover from an unhandled exception. Such an exception causes the system to crash with the infamous *Blue screen of death* (BSOD) (newer versions of Windows have more diverse colors for the crash screen). The BSOD may first appear to be a form of punishment, but it's essentially a protection mechanism. The rationale being it, is that allowing the code to continue execution could cause irreversible damage to Windows (such as deleting important files or corrupting the registry) that may cause the system to fail boot. It's better, then, to stop everything immediately to prevent potential damage. We'll discuss the BSOD in more detail in chapter 6.

All this leads to at least one conclusion: kernel code must be meticulously programmed, and no details like error checking should be skipped.

Termination

When a process terminates, for whatever reason - either normally, because of an unhandled exception, or terminated by external code - it never leaks anything: all private memory is freed, and all handles are closed. Of course, premature handle closing may cause some loss of data, such as a file handle being closed before flushing some data to disk - but there are no resource leaks beyond the lifetime of the process; this is guaranteed by the kernel.

Kernel drivers, on the other hand, don't provide such a guarantee. If a driver unloads while still holding onto allocated memory or open kernel handles - these resources will not be freed automatically, only released at the next system boot.

Why is that? Can't the kernel keep track of a driver's allocations and resource usage so these can be freed automatically when the driver unloads?

Theoretically, this would have been possible to achieve (although currently the kernel does not track such resource usage). The real issue is that it would be too dangerous for the kernel to attempt such cleanup. The kernel has no way of knowing whether the driver leaked those resources for a reason;

for example, the driver could allocate some buffer and then pass it to another driver, with which it cooperates. That second driver may use the memory buffer and free it eventually. If the kernel attempted to free the buffer when the first driver unloads, the second driver would cause an access violation when accessing that now-freed buffer, causing a system crash.

This emphasizes the responsibility of a kernel driver to properly clean up allocated resources; no one else will do it.

Function Return Values

In typical user-mode code, return values from API functions are sometimes ignored, the developer being somewhat optimistic that the called function is unlikely to fail. This may or may not be appropriate for one function or another, but in the worst case, an unhandled exception would later crash the process; the system, however, remains intact.

Ignoring return values from kernel APIs is much more dangerous (see the previous Termination section), and generally should be avoided. Even seemingly “innocent” looking functions can fail for unexpected reasons, so the golden rule here is - always check return status values from kernel APIs.

IRQL

Interrupt Request Level (IRQL) is an important kernel concept that will be further discussed in chapter 6. Suffice it to say at this point that normally a processor’s IRQL is zero, and in particular it’s always zero when user-mode code is executing. In kernel mode, it’s still zero most of the time - but not all the time. Some restrictions on code execution exist at IRQL 2 and higher, which means the driver writer must be careful to use only allowed APIs at that high IRQL. The effects of higher than zero IRQLs are discussed in chapter 6.

C++ Usage

In user mode programming, C++ has been used for many years, and it works well when combined with user-mode Windows APIs. With kernel code, Microsoft started officially supporting C++ with Visual Studio 2012 and WDK 8. C++ is not mandatory, of course, but it has some important benefits related to resource cleanup, with a C++ idiom called *Resource Acquisition Is Initialization* (RAII). We’ll use this RAII idiom quite a bit to make sure we don’t leak resources.

C++ as a language is almost fully supported for kernel code. But there is no C++ runtime in the kernel, and so some C++ features just cannot be used:

- The new and delete operators are not supported and will fail to compile. This is because their normal operation is to allocate from a user-mode heap, which is irrelevant within the kernel. The kernel API has “replacement” functions that are more closely modeled after the C functions malloc and free. We’ll discuss these functions later in this chapter. It is possible, however, to overload the new and delete operators similarly as is sometimes done in user-mode, and invoke the kernel allocation and free functions in the implementation. We’ll see how to do that later in this chapter as well.

- Global variables that have non-default constructors will not be called - there is no C/C++ runtime to call these constructors. These situations must be avoided, but there are some workarounds:
 - Avoid any code in the constructor and instead create some `Init` function to be called explicitly from driver code (e.g. from `DriverEntry`).
 - Allocate a pointer only as a global (or static) variable, and create the actual instance dynamically. The compiler will generate the correct code to invoke the constructor. This works assuming the `new` and `delete` operators have been overloaded, as described later in this chapter.
- The C++ exception handling keywords (`try`, `catch`, `throw`) do not compile. This is because the C++ exception handling mechanism requires its own runtime, which is not present in the kernel. Exception handling can only be done using *Structured Exception Handling* (SEH) - a kernel mechanism to handle exceptions. We'll take a detailed look at SEH in chapter 6.
- The standard C++ libraries are not available in the kernel. Although most are template-based, these do not compile, because they may depend on user-mode libraries and semantics. That said, C++ templates as a language feature work just fine. One good usage of templates is to create alternatives for a kernel-mode library types, based on similar types from the user-mode standard C++ library, such as `std::vector<>`, `std::wstring`, etc.

The code examples in this book make some use of C++. The features mostly used in the code examples are:

- The `nullptr` keyword, representing a true NULL pointer.
- The `auto` keyword that allows type inference when declaring and initializing variables. This is useful to reduce clutter, save some typing, and focus on the important pieces.
- Templates will be used where they make sense.
- Overloading of the `new` and `delete` operators.
- Constructors and destructors, especially for building RAII types.

Any C++ standard can be used for kernel development. The Visual Studio setting for new projects is to use C++ 14. However, you can change the C++ compiler standard to any other setting, including C++ 20 (the latest standard as of this writing). Some features we'll use later will depend on C++ 17 at least.

Strictly speaking, kernel drivers can be written in pure C without any issues. If you prefer to go that route, use files with a C extension rather than CPP. This will automatically invoke the C compiler for these files.

Testing and Debugging

With user-mode code, testing is generally done on the developer's machine (if all required dependencies can be satisfied). Debugging is typically done by attaching the debugger (Visual Studio in most cases) to the running process or launching an executable and attaching to the process.

With kernel code, testing is typically done on another machine, usually a virtual machine hosted on the developer's machine. This ensures that if a BSOD occurs, the developer's machine is unaffected.

Debugging kernel code must be done with another machine, where the actual driver is executing. This is because hitting a breakpoint in kernel-mode freezes the entire machine, not just a particular process. The developer's machine hosts the debugger itself, while the second machine (again, usually a virtual machine) executes the driver code. These two machines must be connected through some mechanism so data can flow between the host (where the debugger is running) and the target. We'll look at kernel debugging in more detail in chapter 5.

Debug vs. Release Builds

Just like with user-mode projects, building kernel drivers can be done in Debug or Release mode. The differences are similar to their user-mode counterparts - Debug builds use no compiler optimizations by default, but are easier to debug. Release builds utilize full compiler optimizations by default to produce the fastest and smallest code possible. There are a few differences, however.

The terms in kernel terminology are *Checked* (Debug) and *Free* (Release). Although Visual Studio kernel projects continue to use the Debug/Release terms, older documentation uses the Checked/Free terms. From a compilation perspective, kernel Debug builds define the symbol `DBG` and set its value to 1 (compared to the `_DEBUG` symbol defined in user mode). This means you can use the `DBG` symbol to distinguish between Debug and Release builds with conditional compilation. This is, for example, what the `KdPrint` macro does: in Debug builds, it compiles to calling `DbgPrint`, while in Release builds it compiles to nothing, resulting in `KdPrint` calls having no effect in Release builds. This is usually what you want because these calls are relatively expensive. We'll discuss other ways of logging information in chapter 5.

The Kernel API

Kernel drivers use exported functions from kernel components. These functions will be referred to as the *Kernel API*. Most functions are implemented within the kernel module itself (`NtOskrnl.exe`), but some may be implemented by other kernel modules, such as the HAL (`hal.dll`).

The Kernel API is a large set of C functions. Most of these start with a prefix suggesting the component implementing that function. Table 3-2 shows some of the common prefixes and their meaning:

Table 3-2: Common kernel API prefixes

Prefix	Meaning	Example
Ex	General executive functions	<code>ExAllocatePoolWithTag</code>
Ke	General kernel functions	<code>KeAcquireSpinLock</code>
Mm	Memory manager	<code>MmProbeAndLockPages</code>
Rtl	General runtime library	<code>RtlInitUnicodeString</code>
FsRtl	file system runtime library	<code>FsRtlGetFileSize</code>
Flt	file system mini-filter library	<code>FltCreateFile</code>
Ob	Object manager	<code>ObReferenceObject</code>

Table 3-2: Common kernel API prefixes

Prefix	Meaning	Example
Io	I/O manager	IoCompleteRequest
Se	Security	SeAccessCheck
Ps	Process manager	PsLookupProcessByProcessId
Po	Power manager	PoSetSystemState
Wmi	Windows management instrumentation	WmiTraceMessage
Zw	Native API wrappers	ZwCreateFile
Hal	Hardware abstraction layer	HalExamineMBR
Cm	Configuration manager (registry)	CmRegisterCallbackEx

If you take a look at the exported functions list from *NtOsKrnL.exe*, you'll find many functions that are not documented in the Windows Driver Kit; this is just a fact of a kernel developer's life - not everything is documented.

One set of functions bears discussion at this point - the *Zw* prefixed functions. These functions mirror native APIs available as gateways from *NtDll.dll* with the actual implementation provided by the Executive. When an *Nt* function is called from user mode, such as *NtCreateFile*, it reaches the Executive at the actual *NtCreateFile* implementation. At this point, *NtCreateFile* might do various checks based on the fact that the original caller is from user mode. This caller information is stored on a thread-by-thread basis, in the undocumented *PreviousMode* member in the *KTHREAD* structure for each thread.

You can query the previous processor mode by calling the documented *ExGetPreviousMode* API.

On the other hand, if a kernel driver needs to call a system service, it should not be subjected to the same checks and constraints imposed on user-mode callers. This is where the *Zw* functions come into play. Calling a *Zw* function sets the previous caller mode to *KernelMode* (0) and then invokes the native function. For example, calling *ZwCreateFile* sets the previous caller to *KernelMode* and then calls *NtCreateFile*, causing *NtCreateFile* to bypass some security and buffer checks that would otherwise be performed. The bottom line is that kernel drivers should call the *Zw* functions unless there is a compelling reason to do otherwise.

Functions and Error Codes

Most kernel API functions return a status indicating success or failure of an operation. This is typed as *NTSTATUS*, a signed 32-bit integer. The value *STATUS_SUCCESS* (0) indicates success. A negative value indicates some kind of error. You can find all the defined *NTSTATUS* values in the file *<ntstatus.h>*.

Most code paths don't care about the exact nature of the error, and so testing the most significant bit is enough to find out whether an error occurred. This can be done with the NT_SUCCESS macro. Here is an example that tests for failure and logs an error if that is the case:

```
NTSTATUS DoWork() {
    NTSTATUS status = CallSomeKernelFunction();
    if(!NT_SUCCESS(status)) {
        KdPrint((L"Error occurred: 0x%08X\n", status));
        return status;
    }

    // continue with more operations

    return STATUS_SUCCESS;
}
```

In some cases, NTSTATUS values are returned from functions that eventually bubble up to user mode. In these cases, the STATUS_xxx value is translated to some ERROR_yyy value that is available to user-mode through the GetLastError function. Note that these are not the same numbers; for one, error codes in user-mode have positive values (zero is still success). Second, the mapping is not one-to-one. In any case, this is not generally a concern for a kernel driver.

Internal kernel driver functions also typically return NTSTATUS to indicate their success/failure status. This is usually convenient, as these functions make calls to kernel APIs and so can propagate any error by simply returning the same status they got back from the particular API. This also implies that the “real” return values from driver functions is typically returned through pointers or references provided as arguments to the function.



Return NTSTATUS from your own functions. It will make it easier and consistent to report errors.

Strings

The kernel API uses strings in many scenarios as needed. In some cases, these strings are simple Unicode pointers (`wchar_t*` or one of their `typedefs` such as `WCHAR*`), but most functions dealing with strings expect a structure of type `UNICODE_STRING`.

The term *Unicode* as used in this book is roughly equivalent to UTF-16, which means 2 bytes per character. This is how strings are stored internally within kernel components. *Unicode* in general is a set of standards related to character encoding. You can find more information at <https://unicode.org>.

The `UNICODE_STRING` structure represents a string with its length and maximum length known. Here is a simplified definition of the structure:

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWCH   Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
typedef const UNICODE_STRING *PCUNICODE_STRING;
```

The `Length` member is in bytes (not characters) and does not include a Unicode-NUL terminator, if one exists (a NUL terminator is not mandatory). The `MaximumLength` member is the number of bytes the string can grow to without requiring a memory reallocation.

Manipulating `UNICODE_STRING` structures is typically done with a set of *Rtl* functions that deal specifically with strings. Table 3-3 lists some of the common functions for string manipulation provided by the *Rtl* functions.

Table 3-3: Common `UNICODE_STRING` functions

Function	Description
<code>RtlInitUnicodeString</code>	Initializes a <code>UNICODE_STRING</code> based on an existing C-string pointer. It sets <code>Buffer</code> , then calculates the <code>Length</code> and sets <code>MaximumLength</code> to the same value. Note that this function does not allocate any memory - it just initializes the internal members.
<code>RtlCopyUnicodeString</code>	Copies one <code>UNICODE_STRING</code> to another. The destination string pointer (<code>Buffer</code>) must be allocated before the copy and <code>MaximumLength</code> set appropriately.
<code>RtlCompareUnicodeString</code>	Compares two <code>UNICODE_STRING</code> s (equal, less, greater), specifying whether to do a case sensitive or insensitive comparison.
<code>RtlEqualUnicodeString</code>	Compares two <code>UNICODE_STRING</code> s for equality, with case sensitivity specification.
<code>RtlAppendUnicodeToString</code>	Appends one <code>UNICODE_STRING</code> to another.
<code>RtlAppendUnicodeToString</code>	Appends <code>UNICODE_STRING</code> to a C-style string.

In addition to the above functions, there are functions that work on C-string pointers. Moreover, some of the well-known string functions from the C Runtime Library are implemented within the kernel as well for convenience: `wcsncpy_s`, `wcsncat_s`, `wcslen`, `wcsncpy_s`, `wcschr`, `strncpy`, `strncpy_s` and others.



The `wcs` prefix works with C Unicode strings, while the `str` prefix works with C Ansi strings. The suffix `_s` in some functions indicates a *safe* function, where an additional argument indicating the maximum length of the string must be provided so the function would not transfer more data than that size.



Never use the non-safe functions. You can include `<dontuse.h>` to get errors for deprecated functions if you do use these in code.

Dynamic Memory Allocation

Drivers often need to allocate memory dynamically. As discussed in chapter 1, kernel thread stack size is rather small, so any large chunk of memory should be allocated dynamically.

The kernel provides two general memory pools for drivers to use (the kernel itself uses them as well).

- Paged pool - memory pool that can be paged out if required.
- Non-Paged Pool - memory pool that is never paged out and is guaranteed to remain in RAM.

Clearly, the non-paged pool is a “better” memory pool as it can never incur a page fault. We’ll see later in this book that some cases require allocating from non-paged pool. Drivers should use this pool sparingly, only when necessary. In all other cases, drivers should use the paged pool. The `POOL_TYPE` enumeration represents the pool types. This enumeration includes many “types” of pools, but only three should be used by drivers: `PagedPool`, `NonPagedPool`, `NonPagedPoolNx` (non-page pool without execute permissions).

Table 3-4 summarizes the most common functions used for working with the kernel memory pools.

Table 3-4: Functions for kernel memory pool allocation

Function	Description
<code>ExAllocatePool</code>	Allocate memory from one of the pools with a default tag. This function is considered obsolete. The next function in this table should be used instead
<code>ExAllocatePoolWithTag</code>	Allocate memory from one of the pools with the specified tag
<code>ExAllocatePoolZero</code>	Same as <code>ExAllocatePoolWithTag</code> , but zeroes out the memory block
<code>ExAllocatePoolWithTagQuota</code>	Allocate memory from one of the pools with the specified tag and charge the current process quota for the allocation
<code>ExFreePool</code>	Free an allocation. The function knows from which pool the allocation was made

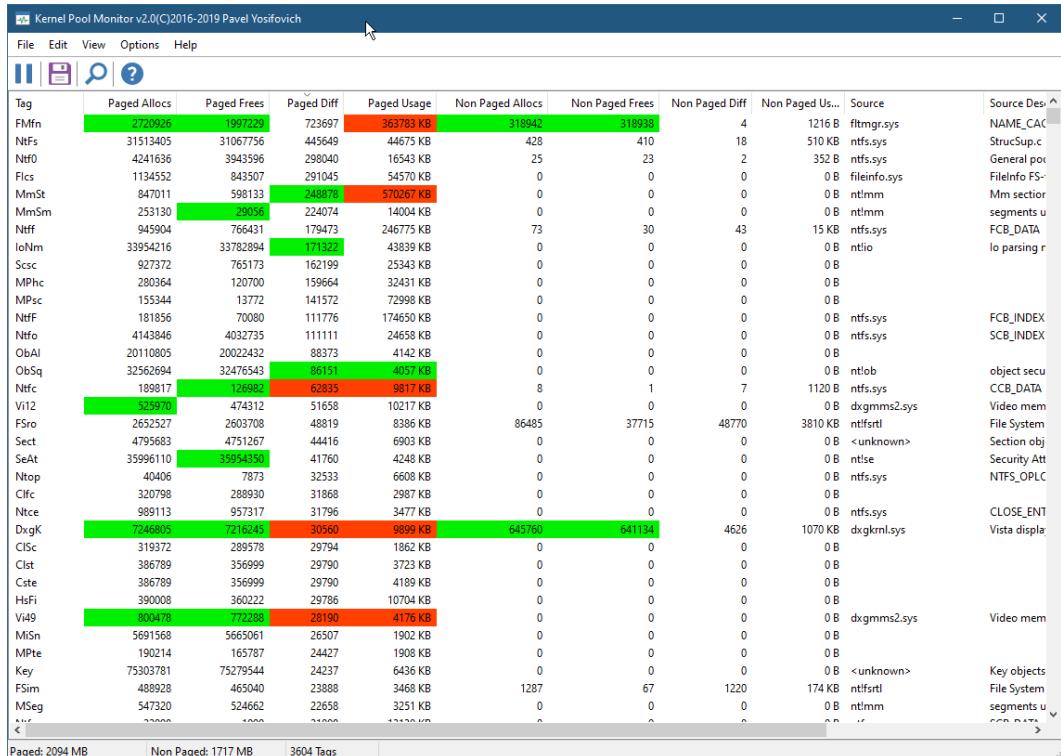


ExAllocatePool calls ExAllocatePoolWithTag using the tag enoN (the word “none” in reverse). Older Windows versions used ‘mdW (WDM in reverse). You should avoid this function and use ExAllocatePoolWithTag instead.

ExAllocatePoolZero is implemented inline in `wdm.h` by calling ExAllocatePoolWithTag and adding the POOL_ZERO_ALLOCATION (=1024) flag to the pool type.

Other memory management functions are covered in chapter 8, “Advanced Programming Techniques”.

The tag argument allows “tagging” an allocation with a 4-byte value. Typically this value is comprised of up to 4 ASCII characters logically identifying the driver, or some part of the driver. These tags can be used to help identify memory leaks - if any allocations tagged with the driver’s tag remain after the driver is unloaded. These pool allocations (with their tags) can be viewed with the *Poolmon* WDK tool, or my own *PoolMonXv2* tool (downloadable from <http://www.github.com/zodiacon/AllTools>). Figure 3-1 shows a screenshot of *PoolMonXv2*.



The screenshot shows the Kernel Pool Monitor v2.0(C)2016-2019 Pavel Yosifovich application window. The main pane displays a table of memory usage statistics. The columns include Tag, Paged Allocs, Paged Frees, Paged Diff, Paged Usage, Non Paged Allocs, Non Paged Frees, Non Paged Diff, Non Paged Us., Source, and Source Desc. The table lists numerous kernel objects like FMfn, NtFs, Ntf0, Flcs, MnSt, MmSm, Ntff, IoNm, Scsc, MPhc, MPsc, NtFf, NtFo, ObAl, ObSq, Ntfc, Vi12, FSro, Sect, SeAt, Ntop, Clfc, Ntce, DrgK, C1Sc, Clst, Cste, HsFi, V49, MiSn, MPte, Key, FSim, MSeg, etc. Each row shows the memory usage for that object, with some values highlighted in green or red. The bottom of the window shows status bars for Paged: 2094 MB, Non Paged: 1717 MB, and 3604 Tags.

Tag	Paged Allocs	Paged Frees	Paged Diff	Paged Usage	Non Paged Allocs	Non Paged Frees	Non Paged Diff	Non Paged Us..	Source	Source Desc
FMfn	2720926	1997229	723697	363783 KB	318942	318938	4	1216 B	fltmgr.sys	NAME_CAC
NtFs	31513405	31067756	445649	44675 KB	428	410	18	510 KB	ntfs.sys	StrucSup.c
Ntf0	4241636	3943596	298040	16543 KB	25	23	2	352 B	ntfs.sys	General por
Flcs	1134552	843507	291045	54570 KB	0	0	0	0 B	fileinfo.sys	FileInfo FS-
MnSt	847011	598133	248978	570267 KB	0	0	0	0 B	ntlm	Mm sector
MmSm	253130	29056	224074	14004 KB	0	0	0	0 B	ntlm	segments u
Ntff	945904	766431	179473	246775 KB	73	30	43	15 KB	ntfs.sys	FCB_DATA
IoNm	33954216	33782894	171322	43839 KB	0	0	0	0 B	ntio	Io parsing r
Scsc	927372	765173	162199	25343 KB	0	0	0	0 B		
MPhc	280364	120700	159664	32431 KB	0	0	0	0 B		
MPsc	155344	13772	141572	72998 KB	0	0	0	0 B		
NtFf	181856	70080	111776	174650 KB	0	0	0	0 B	ntfs.sys	FCB_INDEX
NtFo	4143846	4032735	111111	24658 KB	0	0	0	0 B	ntfs.sys	SCB_INDEX
ObAl	20110805	20024242	88373	4142 KB	0	0	0	0 B		
ObSq	32562694	32476543	86151	4057 KB	0	0	0	0 B	ntlob	object secu
Ntfc	189817	126962	62435	9817 KB	8	1	7	1120 B	ntfs.sys	CCB_DATA
Vi12	525970	474312	51658	10217 KB	0	0	0	0 B	dkgmms2.sys	Video mem
FSro	2652527	2603708	48819	8396 KB	86485	37715	48770	3810 KB	ntfslrt	File System
Sect	4795683	4751267	44416	6903 KB	0	0	0	0 B	<unknown>	Section obj
SeAt	35996110	35954350	41760	4248 KB	0	0	0	0 B	ntse	Security Att
Ntop	40406	7873	32533	6608 KB	0	0	0	0 B	ntfs.sys	NTFS_OPLC
Clfc	320798	288930	31868	2987 KB	0	0	0	0 B		
Ntce	989113	957317	31796	3477 KB	0	0	0	0 B	ntfs.sys	CLOSE_ENT
DrgK	7246805	7216245	30560	6899 KB	645760	641134	4626	1070 KB	dxgkrm.sys	Vista displa
C1Sc	319372	289578	29794	1862 KB	0	0	0	0 B		
Clst	386789	356999	29790	3723 KB	0	0	0	0 B		
Cste	386789	356999	29790	4189 KB	0	0	0	0 B		
HsFi	390008	360222	29786	10704 KB	0	0	0	0 B		
V49	800478	772288	28190	4176 KB	0	0	0	0 B	dxgmm2.sys	Video mem
MiSn	5691568	5665061	26507	1902 KB	0	0	0	0 B		
MPte	190214	165787	24427	1908 KB	0	0	0	0 B		
Key	75303781	75279544	24237	6436 KB	0	0	0	0 B	<unknown>	Key objects
FSim	488928	465040	23888	3468 KB	1287	67	1220	174 KB	ntfslrt	File System
MSeg	547320	524662	22658	3251 KB	0	0	0	0 B	ntlm	segments u
...

Figure 3-1: PoolMonXv



You must use tags comprised of printable ASCII characters. Otherwise, running the driver under the control of the *Driver Verifier* (described in chapter 11) would lead to *Driver Verifier* complaining.

The following code example shows memory allocation and string copying to save the registry path passed to `DriverEntry`, and freeing that string in the `Unload` routine:

```
// define a tag (because of little endianness, viewed as 'abcd')
```

```
#define DRIVER_TAG 'dcba'
```

```
UNICODE_STRING g_RegistryPath;
```

```
extern "C" NTSTATUS
```

```
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    DriverObject->DriverUnload = SampleUnload;
```

```

g_RegistryPath.Buffer = (WCHAR*)ExAllocatePoolWithTag(PagedPool,
    RegistryPath->Length, DRIVER_TAG);
if (g_RegistryPath.Buffer == nullptr) {
    KdPrint(("Failed to allocate memory\n"));
    return STATUS_INSUFFICIENT_RESOURCES;
}

g_RegistryPath.MaximumLength = RegistryPath->Length;
RtlCopyUnicodeString(&g_RegistryPath,
    (PCUNICODE_STRING)RegistryPath);

// %wZ is for UNICODE_STRING objects
KdPrint(("Original registry path: %wZ\n", RegistryPath));
KdPrint(("Copied registry path: %wZ\n", &g_RegistryPath));
//...
return STATUS_SUCCESS;
}

void SampleUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);

    ExFreePool(g_RegistryPath.Buffer);
    KdPrint(("Sample driver Unload called\n"));
}

```

Linked Lists

The kernel uses circular doubly linked lists in many of its internal data structures. For example, all processes on the system are managed by EPROCESS structures, connected in a circular doubly linked list, where its head is stored in the kernel variable PsActiveProcessHead.

All these lists are built in the same way, centered around the LIST_ENTRY structure defined like so:

```

typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY;

```

Figure 3-2 depicts an example of such a list containing a head and three instances.

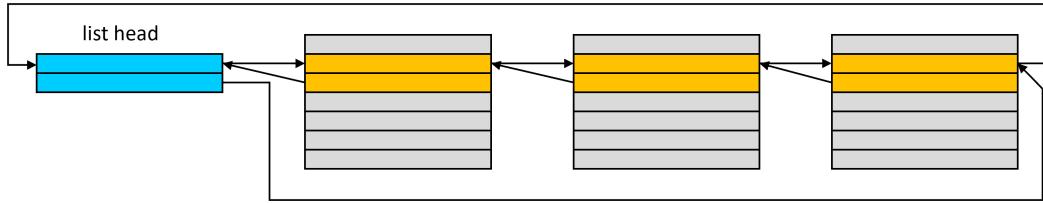


Figure 3-2: Circular linked list

One such structure is embedded inside the real structure of interest. For example, in the EPROCESS structure, the member ActiveProcessLinks is of type LIST_ENTRY, pointing to the next and previous LIST_ENTRY objects of other EPROCESS structures. The head of a list is stored separately; in the case of the process, that's PsActiveProcessHead.

To get the pointer to the actual structure of interest given the address of a LIST_ENTRY can be obtained with the CONTAINING_RECORD macro.

For example, suppose you want to manage a list of structures of type *MyDataItem* defined like so:

```
struct MyDataItem {
    // some data members
    LIST_ENTRY Link;
    // more data members
};
```

When working with these linked lists, we have a head for the list, stored in a variable. This means that natural traversal is done by using the Flink member of the list to point to the next LIST_ENTRY in the list. Given a pointer to the LIST_ENTRY, what we're really after is the *MyDataItem* that contains this list entry member. This is where the CONTAINING_RECORD comes in:

```
MyDataItem* GetItem(LIST_ENTRY* pEntry) {
    return CONTAINING_RECORD(pEntry, MyDataItem, Link);
}
```

The macro does the proper offset calculation and does the casting to the actual data type (*MyDataItem* in the example).

Table 3-5 shows the common functions for working with these linked lists. All operations use constant time.

Table 3-5: Functions for working with circular linked lists

Function	Description
InitializeListHead	Initializes a list head to make an empty list. The forward and back pointers point to the forward pointer.
InsertHeadList	Insert an item to the head of the list.
InsertTailList	Insert an item to the tail of the list.
IsListEmpty	Check if the list is empty.
RemoveHeadList	Remove the item at the head of the list.
RemoveTailList	Remove the item at the tail of the list.
RemoveEntryList	Remove a specific item from the list.
ExInterlockedInsertHeadList	Insert an item at the head of the list atomically by using the specified spinlock.
ExInterlockedInsertTailList	Insert an item at the tail of the list atomically by using the specified spinlock.
ExInterlockedRemoveHeadList	Remove an item from the head of the list atomically by using the specified spinlock.

The last three functions in table 3-4 perform the operation atomically using a synchronization primitive called a *spin lock*. Spin locks are discussed in chapter 6.

The Driver Object

We've already seen that the `DriverEntry` function accepts two arguments, the first is a driver object of some kind. This is a semi-documented structure called `DRIVER_OBJECT` defined in the WDK headers. "Semi-documented" means that some of its members are documented for driver's use and some are not. This structure is allocated by the kernel and partially initialized. Then it's provided to `DriverEntry` (and before the driver unloads to the `Unload` routine as well). The role of the driver at this point is to further initialize the structure to indicate what operations are supported by the driver.

We've seen one such "operation" in chapter 2 - the `Unload` routine. The other important set of operations to initialize are called *Dispatch Routines*. This is an array of function pointers, stored in the `MajorFunction` member of `DRIVER_OBJECT`. This set specifies which operations the driver supports, such as `Create`, `Read`, `Write`, and so on. These indices are defined with the `IRP_MJ_` prefix. Table 3-6 shows some common major function codes and their meaning.

Table 3-6: Common major function codes

Major function	Description
IRP_MJ_CREATE (0)	Create operation. Typically invoked for <code>CreateFile</code> or <code>ZwCreateFile</code> calls.
IRP_MJ_CLOSE (2)	Close operation. Normally invoked for <code>CloseHandle</code> or <code>ZwClose</code> .
IRP_MJ_READ (3)	Read operation. Typically invoked for <code>ReadFile</code> , <code>ZwReadFile</code> and similar read APIs.
IRP_MJ_WRITE (4)	Write operation. Typically invoked for <code>WriteFile</code> , <code>ZwWriteFile</code> , and similar write APIs.
IRP_MJ_DEVICE_CONTROL (14)	Generic call to a driver, invoked because of <code>DeviceIoControl</code> or <code>ZwDeviceIoControlFile</code> calls.
IRP_MJ_INTERNAL_DEVICE_CONTROL (15)	Similar to the previous one, but only available for kernel-mode callers.
IRP_MJ_SHUTDOWN (16)	Called when the system shuts down if the driver has registered for shutdown notification with <code>IoRegisterShutdownNotification</code> .
IRP_MJ_CLEANUP (18)	Invoked when the last handle to a file object is closed, but the file object's reference count is not zero.
IRP_MJ_PNP (31)	Plug and play callback invoked by the Plug and Play Manager. Generally interesting for hardware-based drivers or filters to such drivers.
IRP_MJ_POWER (22)	Power callback invoked by the Power Manager. Generally interesting for hardware-based drivers or filters to such drivers.

Initially, the `MajorFunction` array is initialized by the kernel to point to a kernel internal routine, `IopInvalidDeviceRequest`, which returns a failure status to the caller, indicating the operation is not supported. This means the driver, in its `DriverEntry` routine only needs to initialize the actual operations it supports, leaving all the other entries in their default values.

For example, our Sample driver at this point does not support any dispatch routines, which means there is no way to communicate with the driver. A driver must at least support the `IRP_MJ_CREATE` and `IRP_MJ_CLOSE` operations, to allow opening a handle to one of the device objects for the driver. We'll put these ideas into practice in the next chapter.

Object Attributes

One of the common structures that shows up in many kernel APIs is `OBJECT_ATTRIBUTES`, defined like so:

```

typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;           // SECURITY_DESCRIPTOR
    PVOID SecurityQualityOfService;    // SECURITY_QUALITY_OF_SERVICE
} OBJECT_ATTRIBUTES;
typedef OBJECT_ATTRIBUTES *POBJECT_ATTRIBUTES;
typedef CONST OBJECT_ATTRIBUTES *PCOBJECT_ATTRIBUTES;

```

The structure is typically initialized with the `InitializeObjectAttributes` macro, that allows specifying all the structure members except `Length` (set automatically by the macro), and `SecurityQualityOfService`, which is not normally needed. Here is the description of the members:

- `ObjectName` is the name of the object to be created/located, provided as a pointer to a `UNICODE_STRING`. In some cases it may be ok to set it to `NULL`. For example, the `ZwOpenProcess` allows opening a handle to a process given its PID. Since processes don't have names, the `ObjectName` in this case should be initialized to `NULL`.
- `RootDirectory` is an optional directory in the object manager namespace if the name of the object is relative one. If `ObjectName` specifies a fully-qualified name, `RootDirectory` should be set to `NULL`.
- `Attributes` allows specifying a set of flags that have effect on the operation in question. Table 3-7 shows the defined flags and their meaning.
- `SecurityDescriptor` is an optional security descriptor (`SECURITY_DESCRIPTOR`) to set on the newly created object. `NULL` indicates the new object gets a default security descriptor, based on the caller's token.
- `SecurityQualityOfService` is an optional set of attributes related to the new object's impersonation level and context tracking mode. It has no meaning for most object types. Consult the documentation for more information.

Table 3-7: Object attributes flags

Flag (OBJ_)	Description
INHERIT (2)	The returned handle should be marked as inheritable
PERMANENT (0x10)	The object created should be marked as permanent. Permanent objects have an additional reference count that prevents them from dying even if all handles to them are closed
EXCLUSIVE (0x20)	If creating an object, the object is created with exclusive access. No other handles can be opened to the object. If opening an object, exclusive access is requested, which is granted only if the object was originally created with this flag

Table 3-7: Object attributes flags

Flag (OBJ_)	Description
CASE_INSENSITIVE (0x40)	When opening an object, perform a case insensitive search for its name. Without this flag, the name must match exactly
OPENIF (0x80)	Open the object if it exists. Otherwise, fail the operation (don't create a new object)
OPENLINK (0x100)	If the object to open is a symbolic link object, open the symbolic link object itself, rather than following the symbolic link to its target
KERNEL_HANDLE (0x200)	The returned handle should be a kernel handle. Kernel handles are valid in any process context, and cannot be used by user mode code
FORCE_ACCESS_CHECK (0x400)	Access checks should be performed even if the object is opened in KernelMode access mode
IGNORE_IMPERSONATED_DEVICEMAP (0x800)	Use the process device map instead of the user's if it's impersonating (consult the documentation for more information on device maps)
DONT_REPARSE (0x1000)	Don't follow a reparse point, if encountered. Instead an error is returned (STATUS_REPARSE_POINT_ENCOUNTERED). <i>Reparse points</i> are briefly discussed in chapter 11

A second way to initialize an OBJECT_ATTRIBUTES structure is available with the RTL_CONSTANT_OBJECT_ATTRIBUTES macro, that uses the most common members to set - the object's name and the attributes.

Let's look at a couple of examples that use OBJECT_ATTRIBUTES. The first one is a function that opens a handle to a process given its process ID. For this purpose, we'll use the ZwOpenProcess API, defined like so:

```
NTSTATUS ZwOpenProcess (
    _Out_     PHANDLE ProcessHandle,
    _In_      ACCESS_MASK DesiredAccess,
    _In_      POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_   PCLIENT_ID ClientId);
```

It uses yet another common structure, CLIENT_ID that holds a process and/or a thread ID:

```
typedef struct _CLIENT_ID {
    HANDLE UniqueProcess; // PID, not handle
    HANDLE UniqueThread; // TID, not handle
} CLIENT_ID;
typedef CLIENT_ID *PCLIENT_ID;
```

To open a process, we need to specify the process ID in the UniqueProcess member. Note that although the type of UniqueProcess is HANDLE, it is the unique ID of the process. The reason for

the HANDLE type is that process and thread IDs are generated from a private handle table. This also explains why process and thread IDs are always multiple of four (just like normal handles), and why they don't overlap.

With these details at hand, here is a process opening function:

```
NTSTATUS
OpenProcess(ACCESS_MASK accessMask, ULONG pid, PHANDLE phProcess) {
    CLIENT_ID cid;
    cid.UniqueProcess = ULongToHandle(pid);
    cid.UniqueThread = nullptr;

    OBJECT_ATTRIBUTES procAttributes =
        RTL_CONSTANT_OBJECT_ATTRIBUTES(nullptr, OBJ_KERNEL_HANDLE);
    return ZwOpenProcess(phProcess, accessMask, &procAttributes, &cid);
}
```

The ULongToHandle function performs the required casts so that the compiler is happy (HANDLE is 64-bit on a 64-bit system, but ULONG is always 32-bit). The only member used in the above code from OBJECT_ATTRIBUTES is the Attributes flags.

The second example is a function that opens a handle to a file for read access, by using the ZwOpenFile API, defined like so:

```
NTSTATUS ZwOpenFile(
    _Out_    PHANDLE FileHandle,
    _In_     ACCESS_MASK DesiredAccess,
    _In_     POBJECT_ATTRIBUTES ObjectAttributes,
    _Out_    PIO_STATUS_BLOCK IoStatusBlock,
    _In_     ULONG ShareAccess,
    _In_     ULONG OpenOptions);
```

A full discussion of the parameters to ZwOpenFile is reserved for chapter 11, but one thing is obvious: the file name itself is specified using the OBJECT_ATTRIBUTES structure - there is no separate parameter for that. Here is the full function opening a handle to a file for read access:

```
NTSTATUS OpenFileForRead(PCWSTR path, PHANDLE phFile) {
    UNICODE_STRING name;
    RtlInitUnicodeString(&name, path);

    OBJECT_ATTRIBUTES fileAttributes;
    InitializeObjectAttributes(&fileAttributes, &name,
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, nullptr, nullptr);
    IO_STATUS_BLOCK ioStatus;
```

```
    return ZwOpenFile(phFile, FILE_GENERIC_READ,
                      &fileAttributes, &iostatus, FILE_SHARE_READ, 0);
}
```

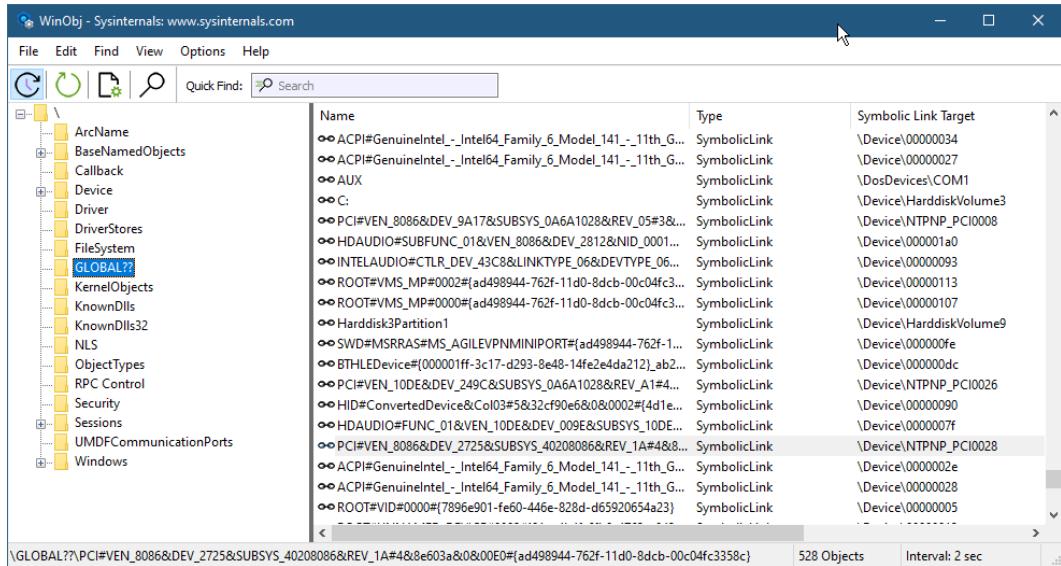
InitializeObjectAttributes is used to initialize the OBJECT_ATTRIBUTES structure, although the RTL_CONSTANT_OBJECT_ATTRIBUTES could have been used just as well, since we're only specifying the name and attributes. Notice the need to turn the passed-in NULL-terminated C-string pointer into a UNICODE_STRING with Rt1InitUnicodeString.

Device Objects

Although a driver object may look like a good candidate for clients to talk to, this is not the case. The actual communication endpoints for clients are device objects. Device objects are instances of the semi-documented DEVICE_OBJECT structure. Without device objects, there is no one to talk to. This means that at least one device object should be created by the driver and given a name, so that it may be contacted by clients.

The CreateFile function (and its variants) accepts a first argument which is called “file name” in the documentation, but really this should point to a device object’s name, where an actual file system file is just one particular case. The name CreateFile is somewhat misleading - the word “file” here means “file object”. Opening a handle to a file or device creates an instance of the kernel structure FILE_OBJECT, another semi-documented structure.

More precisely, CreateFile accepts a *symbolic link*, a kernel object that knows how to point to another kernel object. (You can think of a symbolic link as similar in principle to a file system shortcut.) All the symbolic links that can be used from the user mode CreateFile or CreateFile2 calls are located in the *Object Manager* directory named ?? You can see the contents of this directory with the Sysinternals WinObj tool. Figure 3-3 shows this directory (named Global?? in WinObj).

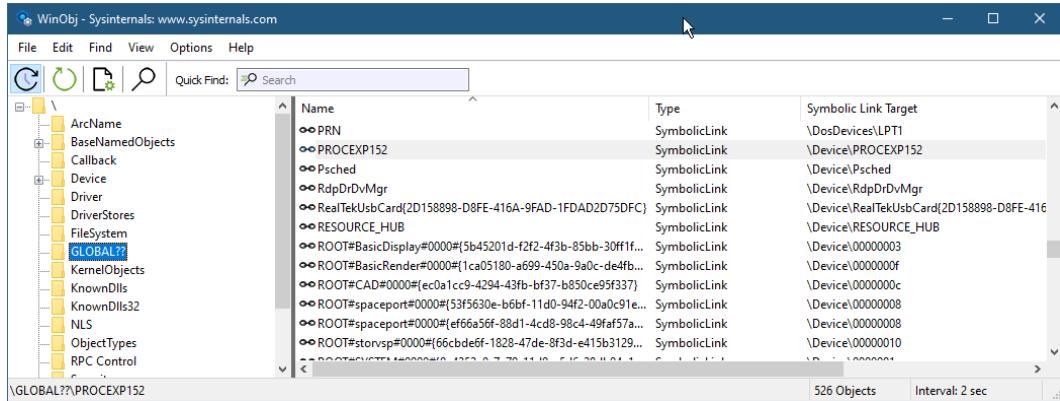
Figure 3-3: Symbolic links directory in *WinObj*

Some of the names seem familiar, such as *C*, *Aux*, *Con*, and others. Indeed, these are valid “file names” for `CreateFile` calls. Other entries look like long cryptic strings, and these in fact are generated by the I/O system for hardware-based drivers that call the `IoRegisterDeviceInterface` API. These types of symbolic links are not useful for the purpose of this book.

Most of the symbolic links in the `\??` directory point to an internal device name under the `\Device` directory. The names in this directory are not directly accessible by user-mode callers. But they can be accessed by kernel callers using the `IoGetDeviceObjectPointer` API.

A canonical example is the driver for *Process Explorer*. When *Process Explorer* is launched with administrator rights, it installs a driver. This driver gives *Process Explorer* powers beyond those that can be obtained by user-mode callers, even if running elevated. For example, *Process Explorer* in its *Threads* dialog for a process can show the complete call stack of a thread, including functions in kernel mode. This type of information is not possible to obtain from user mode; its driver provides the missing information.

The driver installed by *Process Explorer* creates a single device object so that *Process Explorer* is able to open a handle to that device and make requests. This means that the device object must be named, and must have a symbolic link in the `??` directory; and it’s there, called `PROCEXP152`, probably indicating driver version 15.2 (at the time of writing). Figure 3-4 shows this symbolic link in *WinObj*.

Figure 3-4: *Process Explorer's* symbolic link in *WinObj*

Notice the symbolic link for *Process Explorer*'s device points to \Device\PROCEXP152, which is the internal name only accessible to kernel callers (and the native APIs NtOpenFile and NtCreateFile, as shown in the next section). The actual CreateFile call made by *Process Explorer* (or any other client) based on the symbolic link must be prepended with \\.\\. This is necessary so that the I/O manager's parser will not assume the string "PROCEXP152" refers to a file with no extension in the current directory. Here is how *Process Explorer* would open a handle to its device object (note the double backslashes because of the backslash being an escape character in C/C++):

```
HANDLE hDevice = CreateFile(L"\\.\\" PROCEXP152",
    GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING,
    0, nullptr);
```



With C++ 11 and later, you can write strings without escaping the backslash character. The device path in the above code can be written like so: LR"(\\.\\" PROCEXP152)". L indicates Unicode (as always), while anything between R"(and)" is not escaped.

You can try the above code yourself. If *Process Explorer* has run elevated at least once on the system since boot, its driver should be running (you can verify with the tool itself), and the call to CreateFile will succeed if the client is running elevated.

A driver creates a device object using the IoCreateDevice function. This function allocates and initializes a device object structure and returns its pointer to the caller. The device object instance is stored in the DeviceObject member of the DRIVER_OBJECT structure. If more than one device object is created, they form a singly linked list, where the member NextDevice of the DEVICE_OBJECT points to the next device object. Note that the device objects are inserted at the head of the list, so the first

device object created is stored last; its NextDevice points to NULL. These relationships are depicted in figure 3-5.

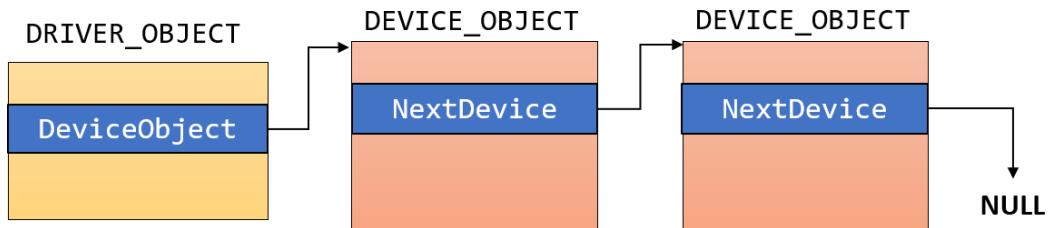


Figure 3-5: Driver and Device objects

Opening Devices Directly

The existence of a symbolic link makes it easy to open a handle to a device with the documented `CreateFile` user-mode API (or from the `ZwOpenFile` API in the kernel). It is sometimes useful, however, to be able to open device objects without going through a symbolic link. For example, a device object might not have a symbolic link, because its driver decided (for whatever reason) not to provide one.

The native `NtOpenFile` (and `NtCreateFile`) function can be used to open a device object directly. Microsoft never recommends using native APIs, but this function is somewhat documented for user-mode use. Its definition is available in the `<Winternl.h>` header file:

```
NTAPI NtOpenFile (
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG ShareAccess,
    IN ULONG OpenOptions);
```

Notice the similarity to the `ZwOpenFile` we used in an earlier section - this is the same function prototype, just invoked here from user mode, eventually to land at `NtOpenFile` within the I/O manager. The function requires usage of an `OBJECT_ATTRIBUTES` structure, described earlier in this chapter.

The above prototype uses old macros such as IN, OUT and others. These have been replaced by SAL annotations. Unfortunately, some header files were not yet converted to SAL.

To demonstrate using `NtOpenFile` from user mode, we'll create an application to play a single sound. Normally, the `Beep` Windows user-mode API provides such a service:

```
BOOL Beep(  
    _In_ DWORD dwFreq,  
    _In_ DWORD dwDuration);
```

The function accepts the frequency to play (in Hertz), and the duration to play, in milliseconds. The function is synchronous, meaning it does not return until the duration has elapsed.

The Beep API works by calling a device named `\Device\Beep` (you can find it in `WinObj`), but the beep device driver does not create a symbolic link for it. However, we can open a handle to the beep device using `NtOpenFile`. Then, to play a sound, we can use the `DeviceIoControl` function with the correct parameters. Although it's not too difficult to reverse engineer the beep driver workings, fortunately we don't have to. The SDK provides the `<ntddbeep.h>` file with the required definitions, including the device name itself.

We'll start by creating a C++ Console application in Visual Studio. Before we get to the `main` function, we need some `#includes`:

```
#include <Windows.h>  
#include <winternl.h>  
#include <stdio.h>  
#include <ntddbeep.h>
```

`<winternl.h>` provides the definition for `NtOpenFile` (and related data structures), while `<ntddbeep.h>` provides the beep-specific definitions.

Since we will be using `NtOpenFile`, we must also link against `NtDll.Dll`, which we can do by adding a `#pragma` to the source code, or add the library to the linker settings in the project's properties. Let's go with the former, as it's easier, and is not tied to the project's properties:

```
#pragma comment(lib, "ntdll")
```



Without the above linkage, the linker would issue an “unresolved external” error.

Now we can start writing `main`, where we accept optional command line arguments indicating the frequency and duration to play:

```

int main(int argc, const char* argv[]) {
    printf("beep [<frequency> <duration_in_msec>]\n");
    int freq = 800, duration = 1000;
    if (argc > 2) {
        freq = atoi(argv[1]);
        duration = atoi(argv[2]);
    }
}

```

The next step is to open the device handle using `NtOpenFile`:

```

HANDLE hFile;
OBJECT_ATTRIBUTES attr;
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\Device\Beep");
InitializeObjectAttributes(&attr, &name, OBJ_CASE_INSENSITIVE,
    nullptr, nullptr);
IO_STATUS_BLOCK ioStatus;
NTSTATUS status = ::NtOpenFile(&hFile, GENERIC_WRITE, &attr, &ioStatus, 0, 0);

```

The line to initialize the device name can be replaced with:

```
RtlInitUnicodeString(&name, DD_BEEP_DEVICE_NAME_U);
```

The `DD_BEEP_DEVICE_NAME_U` macro is conveniently supplied as part of `<ntddbeep.h>`.

If the call succeeds, we can play the sound. To do that, we call `DeviceIoControl` with a control code defined in `<ntddbeep.h>` and use a structure defined there as well to fill in the frequency and duration:

```

if (NT_SUCCESS(status)) {
    BEEP_SET_PARAMETERS params;
    params.Frequency = freq;
    params.Duration = duration;
    DWORD bytes;
    //
    // play the sound
    //
    printf("Playing freq: %u, duration: %u\n", freq, duration);
    ::DeviceIoControl(hFile, IOCTL_BEEP_SET, &params, sizeof(params),
        nullptr, 0, &bytes, nullptr);

    //
    // the sound starts playing and the call returns immediately
    // Wait so that the app doesn't close
}

```

```
//  
::Sleep(duration);  
::CloseHandle(hFile);  
}
```

The input buffer passed to `DeviceIoControl` should be a `BEEP_SET_PARAMETERS` structure, which we pass in along with its size. The last piece of the puzzle is to use the `Sleep` API to wait based on the duration, otherwise the handle to the device would be closed and the sound cut off.



Write an application that plays an array of sounds by leveraging the above code.

Summary

In this chapter, we looked at some of the fundamental kernel data structures, concepts, and APIs. In the next chapter, we'll build a complete driver, and a client application, expanding on the information presented thus far.

Chapter 4: Driver from Start to Finish

In this chapter, we'll use many of the concepts we learned in previous chapters and build a simple, yet complete, driver, and an associated client application, while filling in some of the missing details from previous chapters. We'll deploy the driver and use its capabilities - perform some operation in kernel mode that is difficult, or impossible to do, in user mode.

In this chapter:

- **Introduction**
 - **Driver Initialization**
 - **Client Code**
 - **The Create and Close Dispatch Routines**
 - **The Write Dispatch Routine**
 - **Installing and Testing**
-

Introduction

The problem we'll solve with a simple kernel driver is the inflexibility of setting thread priorities using the Windows API. In user mode, a thread's priority is determined by a combination of its process *Priority Class* with an offset on a per thread basis, that has a limited number of levels.

Changing a process *priority class* (shown as *Base priority* column in *Task Manager*) can be achieved with the `SetPriorityClass` function that accepts a process handle and one of the six supported priority classes. Each priority class corresponds to a priority level, which is the default priority for threads created in that process. A particular thread's priority can be changed with the `SetThreadPriority` function, accepting a thread handle and one of several constants corresponding to offsets around the base priority class. Table 4-1 shows the available thread priorities based on the process priority class and the thread's priority offset.

Table 4-1: Legal values for thread priorities with the Windows APIs

Priority Class	- Sat	-2	-1	0 (default)	+1	+2	+ Sat	Comments
Idle	1	2	3	4	5	6	15	<i>Task Manager</i> refers to <i>Idle</i> as “Low”
Below Normal	1	4	5	6	7	8	15	
Normal	1	6	7	8	9	10	15	
Above Normal	1	8	9	10	11	12	15	
High	1	11	12	13	14	15	15	Only six levels are available (not seven).
Real-time	16	22	23	24	25	26	31	All levels between 16 to 31 can be selected.

The values acceptable to `SetThreadPriority` specify the offset. Five levels correspond to the offsets -2 to +2: `THREAD_PRIORITY_LOWEST` (-2), `THREAD_PRIORITY_BELOW_NORMAL` (-1), `THREAD_PRIORITY_NORMAL` (0), `THREAD_PRIORITY_ABOVE_NORMAL` (+1), `THREAD_PRIORITY_HIGHEST` (+2). The remaining two levels, called *Saturation* levels, set the priority to the two extremes supported by that priority class: `THREAD_PRIORITY_IDLE` (-Sat) and `THREAD_PRIORITY_TIME_CRITICAL` (+Sat).

The following code example changes the current thread’s priority to 11:

```
SetPriorityClass(GetCurrentProcess(),
    ABOVE_NORMAL_PRIORITY_CLASS);           // process base=10
SetThreadPriority(GetCurrentThread(),
    THREAD_PRIORITY_ABOVE_NORMAL);         // +1 offset for thread
```



The Real-time priority class does not imply Windows is a real-time OS; Windows does not provide some of the timing guarantees normally provided by true real-time operating systems. Also, since Real-time priorities are very high and compete with many kernel threads doing important work, such a process must be running with administrator privileges; otherwise, attempting to set the priority class to Real-time causes the value to be set to High.

There are other differences between the real-time priorities and the lower priority classes. Consult the *Windows Internals* book for more information.

Table 4-1 shows the problem we will address quite clearly. Only a small set of priorities are available to set directly. We would like to create a driver that would circumvent these limitations and allow setting a thread’s priority to any number, regardless of its process priority class.

Driver Initialization

We’ll start building the driver in the same way we did in chapter 2. Create a new *WDM Empty Project* named *Booster* (or another name of your choosing) and delete the INF file created by the wizard. Next, add a new source file to the project, called *Booster.cpp* (or any other name you prefer). Add the basic `#include` for the main WDK header and an almost empty `DriverEntry`:

```
#include <ntddk.h>

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    return STATUS_SUCCESS;
}
```

Most software drivers need to do the following in `DriverEntry`:

- Set an *Unload* routine.
- Set dispatch routines the driver supports.
- Create a device object.
- Create a symbolic link to the device object.

Once all these operations are performed, the driver is ready to take requests.

The first step is to add an *Unload* routine and point to it from the driver object. Here is the new `DriverEntry` with the *Unload* routine:

```
// prototypes

void BoosterUnload(PDRIVER_OBJECT DriverObject);

// DriverEntry

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    DriverObject->DriverUnload = BoosterUnload;

    return STATUS_SUCCESS;
}

void BoosterUnload(PDRIVER_OBJECT DriverObject) {
    // empty for now
}
```

We'll add code to the *Unload* routine as needed when we do actual work in `DriverEntry` that needs to be undone.

Next, we need to set up the dispatch routines that we want to support. Practically all drivers must support `IRP_MJ_CREATE` and `IRP_MJ_CLOSE`, otherwise there would be no way to open a handle to any device for this driver. So we add the following to `DriverEntry`:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = BoosterCreateClose;
DriverObject->MajorFunction[IRP_MJ_CLOSE] = BoosterCreateClose;
```

We're pointing the Create and Close major functions to the same routine. This is because, as we'll see shortly, they will do the same thing: simply approve the request. In more complex cases, these could be separate functions, where in the Create case the driver can (for instance) check to see who the caller is and only let approved callers succeed with opening a handle.

All major functions have the same prototype (they are part of an array of function pointers), so we have to add a prototype for `BoosterCreateClose`. The prototype for these functions is as follows:

```
NTSTATUS BoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp);
```

The function must return `NTSTATUS`, and accepts a pointer to a device object and a pointer to an *I/O Request Packet* (IRP). An IRP is the primary object where the request information is stored, for all types of requests. We'll dig deeper into an IRP in chapter 7, but we'll look at the basics later in this chapter, since we require it to complete our driver.

Passing Information to the Driver

The Create and Close operations we set up are required, but certainly not enough. We need a way to tell the driver which thread and to what value to set its priority. From a user-mode client's perspective, there are three basic functions it can use: `WriteFile`, `ReadFile`, and `DeviceIoControl`.

For our driver's purposes, we can use either `WriteFile` or `DeviceIoControl`. Read doesn't make sense, because we're passing information *to* the driver, rather than from the driver. So which is better, `WriteFile` or `DeviceIoControl`? This is mostly a matter of taste, but the general wisdom here is to use `Write` if it's really a write operation (logically); for anything else - `DeviceIoControl` is preferred, as it's a generic mechanism for passing data to and from the driver.

Since changing a thread's priority is not a purely Write operation, `DeviceIoControl` makes more sense, but we'll use `WriteFile`, as it's a bit easier to handle. We'll look at all the details in chapter 7. `WriteFile` has the following prototype:

```
BOOL WriteFile(
    _In_         HANDLE hFile,
    _In_reads_bytes_opt_(nNumberOfBytesToWrite) LPCVOID lpBuffer,
    _In_         DWORD nNumberOfBytesToWrite,
    _Out_opt_    LPDWORD lpNumberOfBytesWritten,
    _Inout_opt_  LPOVERLAPPED lpOverlapped);
```

Our driver has to export its handling of a write operation capability by assigning a function pointer to the `IRP_MJ_WRITE` index of the `MajorFunction` array in the driver object:

```
DriverObject->MajorFunction[IRP_MJ_WRITE] = BoosterWrite;
```

BoosterWrite must have the same prototype as all major function code handlers:

```
NTSTATUS BoosterWrite(PDEVICE_OBJECT DeviceObject, PIRP Irp);
```

Client / Driver Communication Protocol

Given that we use `WriteFile` for client/driver communication, we now must define the actual semantics. `WriteFile` allows passing in a buffer, for which we need to define proper semantics. This buffer should contain the two pieces of information required so the driver can do its thing: the thread id and the priority to set for it.

These pieces of information must be usable both by the driver and the client. The client would supply the data, and the driver would act on it. This means these definitions must be in a separate file that must be included by both the driver and client code.

For this purpose, we'll add a header file named `BoosterCommon.h` to the driver project. This file will also be used later by the user-mode client.

Within this file, we need to define the data structure to pass to the driver in the `WriteFile` buffer, containing the thread ID and the priority to set:

```
struct ThreadData {  
    ULONG ThreadId;  
    int Priority;  
};
```

We need the thread's unique ID and the target priority. Thread IDs are 32-bit unsigned integers, so we select `ULONG` as the type. The priority should be a number between 1 and 31, so a simple 32-bit integer will do.

We cannot normally use `DWORD` - a common type defined in user mode headers - because it's not defined in kernel mode headers. `ULONG`, on the other hand, is defined in both. It would be easy enough to define it ourselves, but `ULONG` is the same anyway.

Creating the Device Object

We have more initializations to do in `DriverEntry`. Currently, we don't have any device object and so there is no way to open a handle and reach the driver. A typical software driver needs just one device object, with a symbolic link pointing to it, so that user-mode clients can obtain handles easily with `CreateFile`.

Creating the device object requires calling the `IoCreateDevice` API, declared as follows (some SAL annotations omitted/simplified for clarity):

```
NTSTATUS IoCreateDevice(
    _In_      PDRIVER_OBJECT DriverObject,
    _In_      ULONG DeviceExtensionSize,
    _In_opt_  PUNICODE_STRING DeviceName,
    _In_      DEVICE_TYPE DeviceType,
    _In_      ULONG DeviceCharacteristics,
    _In_      BOOLEAN Exclusive,
    _Outptr_  PDEVICE_OBJECT *DeviceObject);
```

The parameters to `IoCreateDevice` are described below:

- *DriverObject* - the driver object to which this device object belongs to. This should be simply the driver object passed to the `DriverEntry` function.
- *DeviceExtensionSize* - extra bytes that would be allocated in addition to `sizeof(DEVICE_OBJECT)`. Useful for associating some data structure with a device. It's less useful for software drivers creating just a single device object, since the state needed for the device can simply be managed by global variables.
- *DeviceName* - the internal device name, typically created under the `\Device` Object Manager directory.
- *DeviceType* - relevant to some type of hardware-based drivers. For software drivers, the value `FILE_DEVICE_UNKNOWN` should be used.
- *DeviceCharacteristics* - a set of flags, relevant for some specific drivers. Software drivers specify zero or `FILE_DEVICE_SECURE_OPEN` if they support a true namespace (rarely needed by software drivers). More information on device security is presented in chapter 8.
- *Exclusive* - should more than one file object be allowed to open the same device? Most drivers should specify `FALSE`, but in some cases `TRUE` is more appropriate; it forces a single client at a time for the device.
- *DeviceObject* - the returned pointer, passed as an address of a pointer. If successful, `IoCreateDevice` allocates the structure from non-paged pool and stores the resulting pointer inside the dereferenced argument.

Before calling `IoCreateDevice` we must create a `UNICODE_STRING` to hold the internal device name:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\Booster");
// alternatively,
// RtlInitUnicodeString(&devName, L"\Device\Booster");
```

The device name could be anything but should be in the `\Device` object manager directory. There are two ways to initialize a `UNICODE_STRING` with a constant string. The first is using `RtlInitUnicodeString`, which works just fine. But `RtlInitUnicodeString` must count the number of characters in the string to initialize the `Length` and `MaximumLength` appropriately. Not a big deal in this case, but there is a quicker way - using the `RTL_CONSTANT_STRING` macro, which calculates the length of the string statically (at compile time), meaning it can only work correctly with literal strings.

Now we are ready to call the `IoCreateDevice` function:

```
PDEVICE_OBJECT DeviceObject;
NTSTATUS status = IoCreateDevice(
    DriverObject,           // our driver object
    0,                      // no need for extra bytes
    &devName,               // the device name
    FILE_DEVICE_UNKNOWN,    // device type
    0,                      // characteristics flags
    FALSE,                 // not exclusive
    &DeviceObject);        // the resulting pointer
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create device object (0x%08X)\n", status));
    return status;
}
```

If all goes well, we now have a pointer to our device object. The next step is to make this device object accessible to user-mode callers by providing a symbolic link. Creating a symbolic link involves calling `IoCreateSymbolicLink`:

```
NTSTATUS IoCreateSymbolicLink(
    _In_ PUNICODE_STRING SymbolicLinkName,
    _In_ PUNICODE_STRING DeviceName);
```

The following lines create a symbolic link and connect it to our device object:

```
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\Booster");
status = IoCreateSymbolicLink(&symLink, &devName);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to create symbolic link (0x%08X)\n", status));
    IoDeleteDevice(DeviceObject); // important!
    return status;
}
```

The `IoCreateSymbolicLink` does the work by accepting the symbolic link and the target of the link. Note that if the creation fails, we must undo everything done so far - in this case just the fact the device object was created - by calling `IoDeleteDevice`. More generally, if `DriverEntry` returns any failure status, the `Unload` routine is **not** called. If we had more initialization steps to do, we would have to remember to undo everything until that point in case of failure. We'll see a more elegant way of handling this in chapter 6.

Once we have the symbolic link and the device object set up, `DriverEntry` can return success, indicating the driver is now ready to accept requests.

Before we move on, we must not forget the `Unload` routine. Assuming `DriverEntry` completed successfully, the `Unload` routine must undo whatever was done in `DriverEntry`. In our case, there are two things to undo: device object creation and symbolic link creation. We'll undo them in reverse order:

```

void BoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\Booster");
    // delete symbolic link
    IoDeleteSymbolicLink(&symLink);

    // delete device object
    IoDeleteDevice(DriverObject->DeviceObject);
}

```

Notice the device object pointer is extracted from the driver object, as it's the only argument we get in the Unload routine. It's certainly possible to store the device object pointer in a global variable and access it here directly, but there is no need. Global variables usage should be kept to a minimum.

Client Code

At this point, it's worth writing the user-mode client code. Everything we need for the client has already been defined.

Add a new C++ Console Application project to the solution named *Boost* (or some other name of your choosing). The Visual Studio wizard should create a single source file with some “hello world” type of code. You can safely delete all the contents of the file.

First, we add the required #includes to the *Boost.cpp* file:

```

#include <windows.h>
#include <stdio.h>
#include "..\Booster\BoosterCommon.h"

```

Note that we include the common header file created by the driver to be shared with the client.

Change the `main` function to accept command line arguments. We'll accept a thread ID and a priority using command line arguments and request the driver to change the priority of the thread to the given value.

```

int main(int argc, const char* argv[]) {
    if (argc < 3) {
        printf("Usage: Boost <threadid> <priority>\n");
        return 0;
    }

    //
    // extract from command line
    //
    int tid = atoi(argv[1]);
    int priority = atoi(argv[2]);
}

```

Next, we need to open a handle to our device. The “file name” to `CreateFile` should be the symbolic link prepended with “`\.\.`”. The entire call should look like this:

```
HANDLE hDevice = CreateFile(L"\\.\.\Booster", GENERIC_WRITE,
    0, nullptr, OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE)
    return Error("Failed to open device");
```

The `Error` function simply prints some text with the last Windows API error:

```
int Error(const char* message) {
    printf("%s (error=%u)\n", message, GetLastError());
    return 1;
}
```

The `CreateFile` call should reach the driver in its `IRP_MJ_CREATE` dispatch routine. If the driver is not loaded at this time - meaning there is no device object and no symbolic link - we’ll get error number 2 (file not found).

Now that we have a valid handle to our device, it’s time to set up the call to `Write`. First, we need to create a `ThreadData` structure and fill in the details:

```
ThreadData data;
data.ThreadId = tid;
data.Priority = priority;
```

Now we’re ready to call `WriteFile` and close the device handle afterwards:

```
DWORD returned;
BOOL success = WriteFile(hDevice,
    &data, sizeof(data),           // buffer and length
    &returned, nullptr);
if (!success)
    return Error("Priority change failed!");

printf("Priority change succeeded!\n");

CloseHandle(hDevice);
```

The call to `WriteFile` reaches the driver by invoking the `IRP_MJ_WRITE` major function routine.

At this point, the client code is complete. All that remains is to implement the dispatch routines we declared on the driver side.

The Create and Close Dispatch Routines

Now we're ready to implement the three dispatch routines defined by the driver. The simplest by far are the Create and Close routines. All that's needed is completing the request with a successful status. Here is the complete Create/Close dispatch routine implementation:

```
NTSTATUS BoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    UNREFERENCED_PARAMETER(DeviceObject);

    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

Every dispatch routine accepts the target device object and an *I/O Request Packet* (IRP). We don't care much about the device object, since we only have one, so it must be the one we created in `DriverEntry`. The IRP on the other hand, is extremely important. We'll dig deeper into IRPs in chapter 6, but we need to take a quick look at IRPs now.

An IRP is a semi-documented structure that represents a request, typically coming from one of the managers in the Executive: the I/O Manager, the Plug & Play Manager, or the Power Manager. With a simple software driver, that would most likely be the I/O Manager. Regardless of the creator of the IRP, the driver's purpose is to handle the IRP, which means looking at the details of the request and doing what needs to be done to complete it.

Every request to the driver always arrives wrapped in an IRP, whether that's a Create, Close, Read, Write, or any other IRP. By looking at the IRP's members, we can figure out the type and details of the request (technically, the dispatch routine itself was pointed to based on the request type, so in most cases you already know the request type). It's worth mentioning that an IRP never arrives alone; it's accompanied by one or more structures of type `IO_STACK_LOCATION`. In simple cases like our driver, there is a single `IO_STACK_LOCATION`. In more complex cases where there are filter drivers above or below us, multiple `IO_STACK_LOCATION` instances exist, one for each layer in the device stack. (We'll discuss this more thoroughly in chapter 7). Simply put, some of the information we need is in the base IRP structure, and some is in the `IO_STACK_LOCATION` for our "layer" in the device stack.

In the case of Create and Close, we don't need to look into any members. We just need to set the completion status of the IRP in its `IoStatus` member (of type `IO_STATUS_BLOCK`), which has two members:

- *Status* (`NTSTATUS`) - indicating the status this request should complete with.
- *Information* (`ULONG_PTR`) - a polymorphic member, meaning different things in different request types. In the case of Create and Close, a zero value is just fine.

To complete the IRP, we call `IoCompleteRequest`. This function has a lot to do, but basically it propagates the IRP back to its creator (typically the I/O Manager), and that manager notifies the

client that the operation has completed and frees the IRP. The second argument is a temporary priority boost value that a driver can provide to its client. In most cases for a software driver, a value of zero is fine (`IO_NO_INCREMENT` is defined as zero). This is especially true since the request is completed synchronously, so no reason the caller should get a priority boost. More information on this function is provided in chapter 7.

The last thing to do is return the same status as the one put into the IRP. This may seem like a useless duplication, but it is necessary (the reason will be clearer in a later chapter).



You may be tempted to write the last line of `BoosterCreateClose` like so:

```
return Irp->IoStatus.Status; So that the returned value is always the same as the
one stored in the IRP. This code is buggy, however, and will cause a BSOD in most
cases. The reason is that after IoCompleteRequest is invoked, the IRP pointer should
be considered "poison", as it's more likely than not that it has already been deallocated by
the I/O manager.
```

The Write Dispatch Routine

This is the crux of the matter. All the driver code so far has led to this dispatch routine. This is the one doing the actual work of setting a given thread to a requested priority.

The first thing we need to do is check for errors in the supplied data. In our case, we expect a structure of type `ThreadData`. The first thing to do is retrieve the current IRP stack location, because the size of the buffer happens to be stored there:

```
NTSTATUS BoosterWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto status = STATUS_SUCCESS;
    ULONG_PTR information = 0; // track used bytes

    // irpSp is of type PIO_STACK_LOCATION
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
```

The key to getting the information for any IRP is to look inside the `IO_STACK_LOCATION` associated with the current device layer. Calling `IoGetCurrentIrpStackLocation` returns a pointer to the correct `IO_STACK_LOCATION`. In our case, there is just one `IO_STACK_LOCATION`, but in the general case there could be more (in fact, a filter may be above our device), so calling `IoGetCurrentIrpStackLocation` is the right thing to do.

The main ingredient in an `IO_STACK_LOCATION` is a monstrous union identified with the member named `Parameters`, which holds a set of structures, one for each type of IRP. In the case of `IRP_MJ_WRITE`, the structure to look at is `Parameters.Write`.

Now we can check the buffer size to make sure it's at least the size we expect:

```
do {
    if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
}
```

The do keyword opens a simple `do/while(false)` block that allows using the `break` keyword to bail out early in case of an error. We'll discuss this technique in greater detail in chapter 7.

Next, we need to grab the user buffer's pointer, and check if the priority value is in the legal range (0 to 31). We also check if the pointer itself is NULL, as it's possible for the client to pass a NULL pointer for the buffer, but the length may be greater than zero. The buffer's address is provided in the `UserBuffer` member of the IRP:

```
auto data = static_cast<ThreadData*>(Irp->UserBuffer);
if (data == nullptr || data->Priority < 1 || data->Priority > 31) {
    status = STATUS_INVALID_PARAMETER;
    break;
}
```

`UserBuffer` is typed as a `void` pointer, so we need to cast it to the expected type. Then we check the priority value, and if not in range change the status to `STATUS_INVALID_PARAMETER` and break out of the "loop".



Notice the order of checks: the pointer is compared to `NULL` first, and only if non-`NULL`, the next check takes place. If `data` is `NULL`, however, no further checks are made. This behavior is guaranteed by the C/C++ standard, known as *short circuit evaluation*.



The use of `static_cast` asks the compiler to check if the cast makes sense. Technically, the C++ compiler allows casting a `void` pointer to any other pointer, so it doesn't look that useful in this case, and perhaps a C-style cast would be simpler to write. Still, it's a good habit to have, as it can catch some errors at compile time (rather than nasty bugs at runtime).

We're getting closer to our goal. The API we would like to use is `KeSetPriorityThread`, prototyped as follows:

```
KPRIORITy KeSetPriorityThread(
    _Inout_ PKTHREAD Thread,
    _In_     KPRIORITy Priority);
```

The KPRORITY type is just an 8-bit integer. The thread itself is identified by a pointer to a KTHREAD object. KTHREAD is one part of the way the kernel manages threads. It's completely undocumented, but we need the pointer value anyway. We have the thread ID from the client, and need to somehow get a hold of a pointer to the real thread object in kernel space. The function that can look up a thread by its ID is aptly named `PsLookupThreadByThreadId`. To get its definition, we need to add another `#include`:

```
#include <ntifs.h>
```



You must add this `#include` before `<ntddk.h>`, otherwise you'll get compilation errors. In fact, you can remove `<ntddk.h>` entirely, as it's included by `<ntifs.h>`.

Here is the definition for `PsLookupThreadByThreadId`:

```
NTSTATUS PsLookupThreadByThreadId(
    _In_          HANDLE ThreadId,
    _Outptr_       PETHREAD *Thread);
```

Again, we see that a thread ID is required, but its type is `HANDLE` - but it is the ID that we need nonetheless. The resulting pointer is typed as `PETHREAD` or pointer to `ETHREAD`. `ETHREAD` is completely opaque. Regardless, we seem to have a problem since `KeSetPriorityThread` accepts a `PKTHREAD` rather than `PETHREAD`. It turns out these are the same, because the first member of an `ETHREAD` is a `KTHREAD` (the member is named `Tcb`). We'll prove all this in the next chapter when we use the kernel debugger. Here is the beginning of the definition of `ETHREAD`:

```
typedef struct _ETHREAD {
    KTHREAD Tcb;
    // more members
} ETHREAD;
```

The bottom line is we can safely switch `PKTHREAD` for `PETHREAD` or vice versa when needed without a hitch.

Now we can turn our thread ID into a pointer:

```
PETHREAD thread;
status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId),
    &thread);
if (!NT_SUCCESS(status))
    break;
```

The call to `PsLookupThreadByThreadId` can fail, the main reason being that the thread ID does not reference any thread in the system. If the call fails, we simply break and let the resulting NTSTATUS propagate out of the “loop”.

We are finally ready to change the thread’s priority. But wait - what if after the last call succeeds, the thread is terminated, just before we set its new priority? Rest assured, this cannot happen. Technically, the thread can terminate (from an execution perspective) at that point, but that will not make our pointer a dangling one. This is because the lookup function, if successful, increments the reference count on the kernel thread object, so it cannot die until we explicitly decrement the reference count. Here is the call to make the priority change:

```
auto oldPriority = KeSetPriorityThread(thread, data->Priority);
KdPrint(("Priority change for thread %u from %d to %d succeeded!\n",
         data->ThreadId, oldPriority, data->Priority));
```

We get back the old priority, which we output with `KdPrint` for debugging purposes. All that’s left to do now is decrement the thread object’s reference; otherwise, we have a leak on our hands (the thread object will never die), which will only be resolved in the next system boot. The function that accomplishes this feat is `ObDereferenceObject`:

```
ObDereferenceObject(thread);
```

We should also report to the client that we used the buffer provided. This is where the `information` variable is used:

```
information = sizeof(data);
```

We’ll write that value to the IRP before completing it. This is the value returned as the second to last argument from the client’s `WriteFile` call. All that’s left to do is to close the while “loop” and complete the IRP with whatever status we happen to have at this time.

```
// end the while "loop"
} while (false);

//
// complete the IRP with the status we got at this point
//
Irp->IoStatus.Status = status;
Irp->IoStatus.Information = information;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
return status;
}
```

And we’re done! For reference, here is the complete `IRP_MJ_WRITE` handler:

```

NTSTATUS BoosterWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto status = STATUS_SUCCESS;
    ULONG_PTR information = 0;

    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    do {
        if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }
        auto data = static_cast<ThreadData*>(Irp->UserBuffer);
        if (data == nullptr
            || data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        PETHREAD thread;
        status = PsLookupThreadByThreadId(
            ULongToHandle(data->ThreadId), &thread);
        if (!NT_SUCCESS(status)) {
            break;
        }
        auto oldPriority = KeSetPriorityThread(thread, data->Priority);
        KdPrint(("Priority change for thread %u from %d to %d succeeded!\n",
            data->ThreadId, oldPriority, data->Priority));

        ObDereferenceObject(thread);
        information = sizeof(data);
    } while (false);

    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = information;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

Installing and Testing

At this point, we can build the driver and client successfully. Our next step is to install the driver and test its functionality. You can try the following on a virtual machine, or if you're feeling brave enough - on your development machine.

First, let's install the driver. Copy the resulting *booster.sys* file to the target machine (if it's not your

development machine). On the target machine, open an elevated command window and install the driver using the `sc.exe` tool as we did back in chapter 2:

```
c:\> sc create booster type= kernel binPath= c:\Test\Booster.sys
```

Make sure `binPath` includes the full path of the resulting SYS file. The name of the driver (*booster*) in the example is the name of the created Registry key, and so must be unique. It doesn't have to be related to the SYS file name.

Now we can load the driver:

```
c:\> sc start booster
```

If all is well, the driver would have started successfully. To make sure, we can open *WinObj* and look for our device name and symbolic link. Figure 4-1 shows the symbolic link in *WinObj*.

	Name	Type	Symbolic Link Target
GLOBAL?\Booster	ACPI_ROOT_OBJECT	SymbolicLink	\Device\00000014
	AgileVPN	SymbolicLink	\Device\AgileVPN
	ahcache	SymbolicLink	\Device\ahcache
	AUX	SymbolicLink	\DosDevices\COM1
	BitLocker	SymbolicLink	\Device\BitLocker
	Booster	SymbolicLink	\Device\Booster
	BootPartition	SymbolicLink	\Device\HarddiskVolume4
	C:	SymbolicLink	\Device\HarddiskVolume4
	CimfsControl	SymbolicLink	\Device\cimfs\control
	COM2	SymbolicLink	\Device\Serial0
	CON	SymbolicLink	\Device\ConDrv\Console
	CONIN\$	SymbolicLink	\Device\ConDrv\Currentin
	CONOUT\$	SymbolicLink	\Device\ConDrv\CurrentOut
	Disk{6587d739-dd09-449b-0626-01d0ced5e42}	SymbolicLink	\Device\Harddisk\DR0
	DISPLAY#Default_Monitor#1&c528b8a&3&UID256#{10910c...	SymbolicLink	\Device\0000007d
	DISPLAY#Default_Monitor#1&c528b8a&3&UID256#{866519...	SymbolicLink	\Device\0000007d
	DISPLAY#Default_Monitor#1&c528b8a&3&UID256#{ef07b...	SymbolicLink	\Device\0000007d
	DISPLAY#MSH062E#5&1a097cd88&0&UID5527112#{10910c...	SymbolicLink	\Device\00000069
	DISPLAY#MSH062E#5&1a097cd88&0&UID5527112#{866519b...	SymbolicLink	\Device\00000069
	DISPLAY#MSH062E#5&1a097cd88&0&UID5527112#{ef07b5f...	SymbolicLink	\Device\00000069
	DISPLAY1	SymbolicLink	\Device\Video0
	DISPLAY2	SymbolicLink	\Device\Video1
	FltMgr	SymbolicLink	\FileSystem\Filters\FltMgr

Figure 4-1: Symbolic Link in *WinObj*

Now we can finally run the client executable. Figure 4-2 shows a thread in *Process Explorer* of a `cmd.exe` process selected as an example for which we want set priority to a new value.

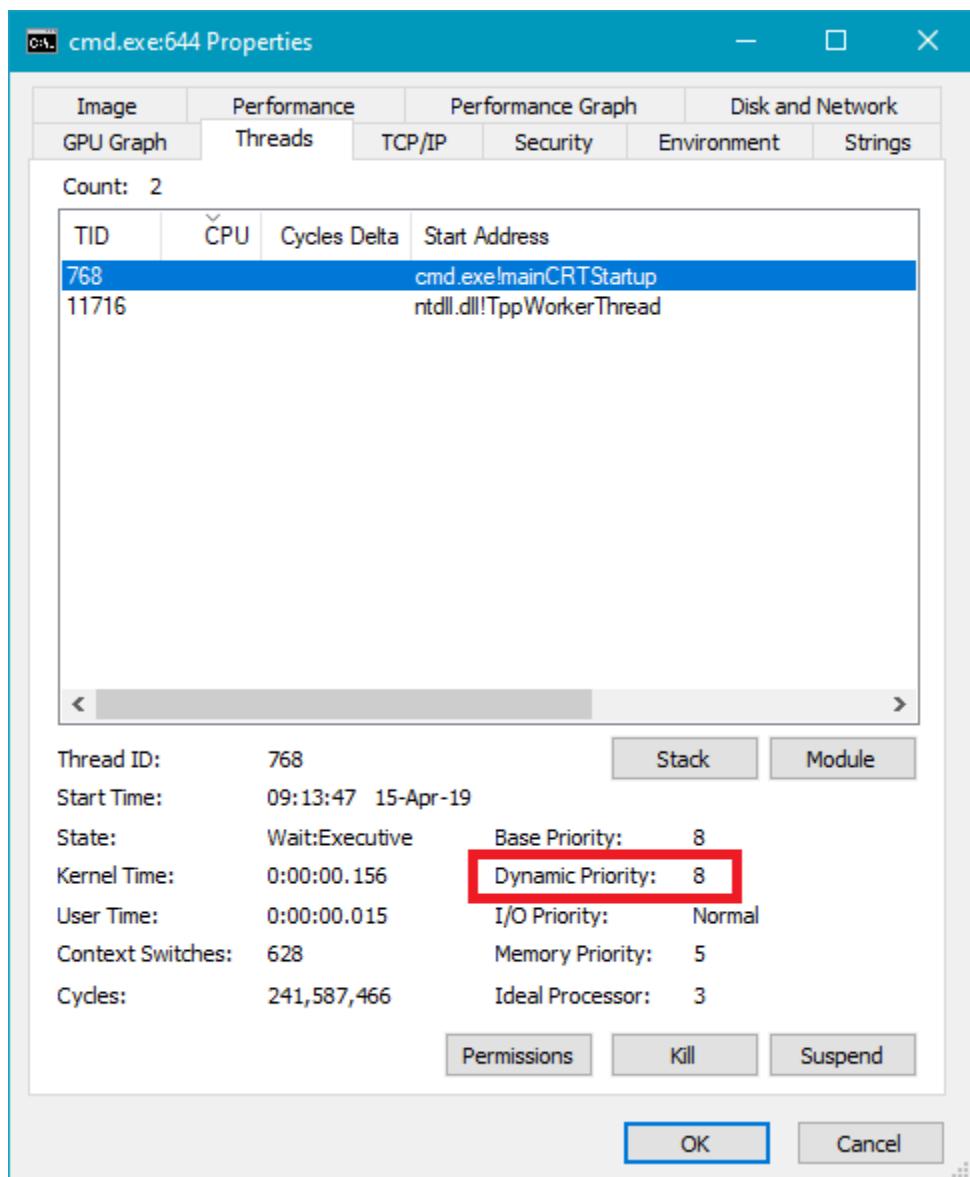


Figure 4-2: Original thread priority

Run the client with the thread ID and the desired priority (replace the thread ID as needed):

```
c:\Test> boost 768 25
```



If you get an error trying to run the executable (usually it's a Debug build), you may need to set the runtime library to a static one instead of a DLL. Go to *Project properties* in Visual Studio for the client application, C++ node, *Code Generation*, *Runtime Library*, and select **Multithreaded Debug**. Alternatively, you can compile the client in *Release* build, and that should run without any changes.

And voila! See figure 4-3.

You should also run *DbgView* and see the output when a successful priority change occurs.

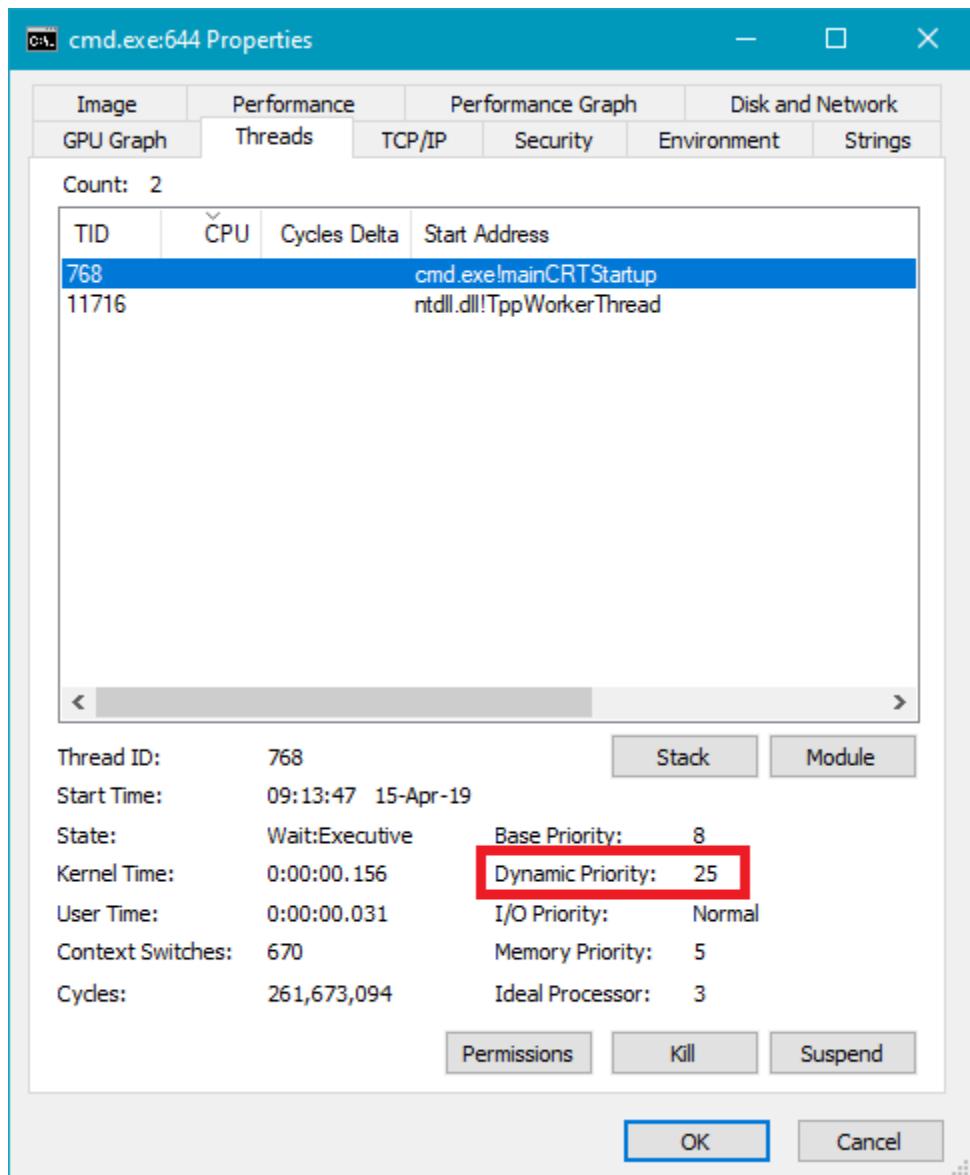


Figure 4-3: Modified thread priority

Summary

We've seen how to build a simple, yet complete, driver, from start to finish. We created a user-mode client to communicate with the driver. In the next chapter, we'll tackle debugging, which is something we're bound to do when writing drivers that may not behave as we expect.

Chapter 5: Debugging and Tracing

Just like with any software, kernel drivers tend to have bugs. Debugging drivers, as opposed to user-mode debugging, is more challenging. Driver debugging is essentially debugging an entire machine, not just a specific process. This requires a somewhat different mindset. This chapter discusses user-mode and kernel-mode debugging using the *WinDbg* debugger.

In this chapter:

- Debugging Tools for Windows
 - Introduction to *WinDbg*
 - Kernel Debugging
 - Full Kernel Debugging
 - Kernel Driver Debugging Tutorial
 - Asserts and Tracing
-

Debugging Tools for Windows

The *Debugging Tools for Windows* package contains a set of debuggers, tools, and documentation focusing on the debuggers within the package. This package can be installed as part of the Windows SDK or the WDK, but there is no real “installation” done. The installation just copies files but does not touch the Registry, meaning the package depends only on its own modules and the Windows built-in DLLs. This makes it easy to copy the entire directory to any other directory including removable media.

The package contains four debuggers: *Cdb.exe*, *Ntsd.exe*, *Kd.exe*, and *WinDbg.exe*. Here is a rundown of the basic functionality of each debugger:

- *Cdb* and *Ntsd* are user-mode, console-based debuggers. This means they can be attached to processes, just like any other user-mode debugger. Both have console UI - type in a command, get a response, and repeat. The only difference between the two is that if launched from a console window, *Cdb* uses the same console, whereas *Ntsd* always opens a new console window. They are otherwise identical.
- *Kd* is a kernel debugger with a console user interface. It can attach to the local kernel (*Local Kernel Debugging*, described in the next section), or to another machine for a full kernel debugging experience.
- *WinDbg* is the only debugger with a graphical user interface. It can be used for user-mode debugging or kernel debugging, depending on the selection performed with its menus or the command line arguments passed to it when launched.

A relatively recent alternative to the classic *WinDbg* is *Windbg Preview*, available through the Microsoft store. This is a remake of the classic debugger with a much better user interface. It can be installed on Windows 10 version 1607 or later. From a functionality standpoint, it's similar to the classic *WinDbg*. But it is somewhat easier to use because of the modern, convenient UI, and in fact has also solved some bugs that still plague the classic debugger. All the commands we'll see in this chapter work equally well with either debugger.

Although these debuggers may seem different from one another, the user-mode debuggers are essentially the same, as are the kernel debuggers. They are all based around a single debugger engine implemented as a DLL (*DbgEng.Dll*). The various debuggers are able to use *extension DLLs*, that provide most of the power of the debuggers by loading new commands.

The Debugger Engine is documented to a large extent in the *Debugging tools for Windows* documentation, which makes it possible to write new debuggers (or other tools) that utilize the debugger engine.

Other tools that are part of the package include the following (partial list):

- *Gflags.exe* - the Global Flags tool that allows setting some kernel flags and image flags.
- *ADPlus.exe* - generate a dump file for a process crash or hang.
- *Kill.exe* - a simple tool to terminate process(es) based on process ID, name, or pattern.
- *Dumpchk.exe* - tool to do some general checking of dump files.
- *TList.exe* - lists running processes on the system with various options.
- *Umdh.exe* - analyzes heap allocations in user-mode processes.
- *UsbView.exe* - displays a hierarchical view of USB devices and hubs.

Introduction to *WinDbg*

This section describes the fundamentals of *WinDbg*, but bear in mind everything is essentially the same for the console debuggers, with the exception of the GUI windows.

WinDbg is built around commands. The user enters a command, and the debugger responds with text describing the results of the command. With the GUI, some of these results are depicted in dedicated windows, such as locals, stack, threads, etc.

WinDbg supports three types of commands:

- Intrinsic commands - these commands are built-in into the debugger (part of the debugger engine), and they operate on the target being debugged.

- Meta commands - these commands start with a period (.) and they operate on the debugging environment, rather than directly on the target being debugged.
- Extension commands (sometimes called *bang commands*) - these commands start with an exclamation point (!), providing much of the power of the debugger. All extension commands are implemented in external DLLs. By default, the debugger loads a set of predefined extension DLLs, but more can be loaded from the debugger directory or another directory with the .load meta command.

Writing extension DLLs is possible and is fully documented in the debugger docs. In fact, many such DLLs have been created and can be loaded from their respective source. These DLLs provide new commands that enhance the debugging experience, often targeting specific scenarios.

Tutorial: User mode debugging basics

If you have experience with *WinDbg* usage in user-mode, you can safely skip this section.

This tutorial is aimed at getting a basic understanding of *WinDbg* and how to use it for user-mode debugging. Kernel debugging is described in the next section.

There are generally two ways to initiate user-mode debugging - either launch an executable and attach to it, or attach to an already existing process. We'll use the latter approach in this tutorial, but except for this first step, all other operations are identical.

- Launch *Notepad*.
- Launch *WinDbg* (either the Preview or the classic one. The following screenshots use the Preview).
- Select *File / Attach To Process* and locate the *Notepad* process in the list (see figure 5-1). Then click *Attach*. You should see output similar to figure 5-2.

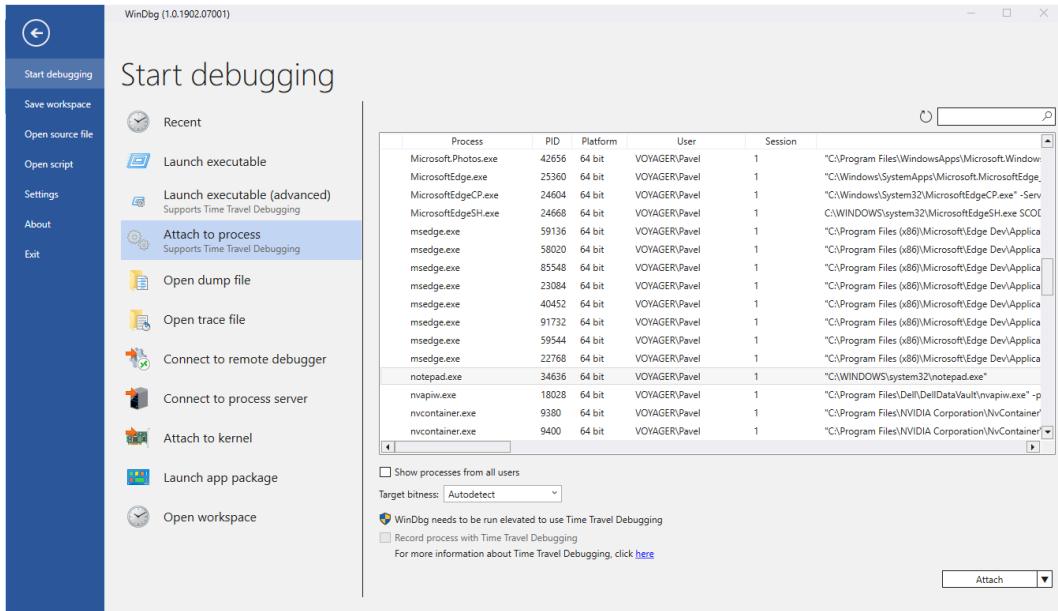


Figure 5-1: Attaching to a process with WinDbg

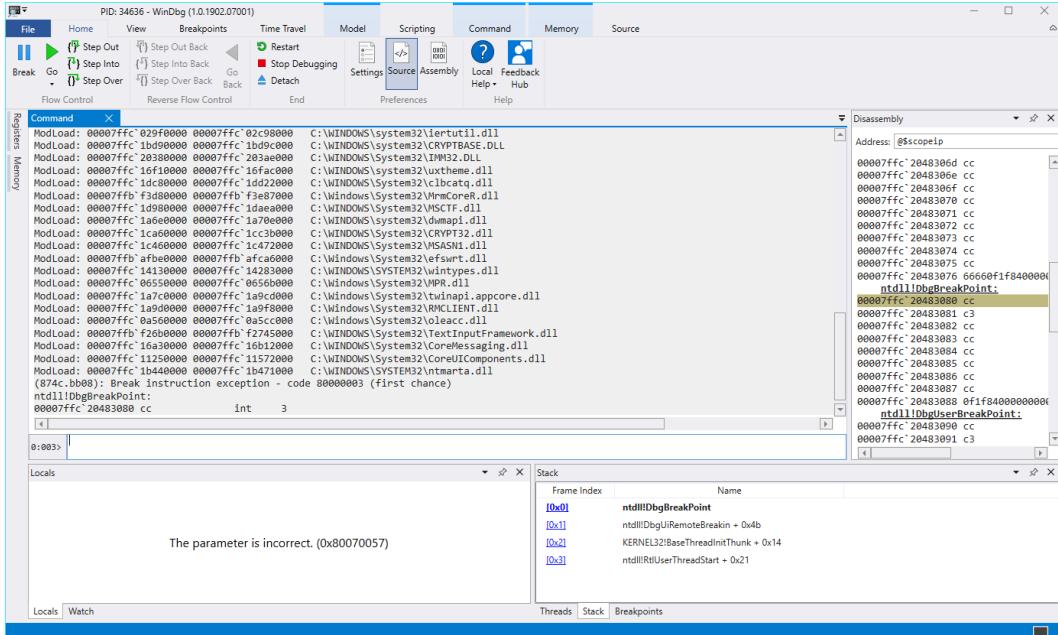


Figure 5-2: First view after process attach

The Command window is the main window of interest - it should always be open. This is the one showing the various responses of commands. Typically, most of the time in a debugging session is spent interacting with this window.

The process is suspended - we are in a breakpoint induced by the debugger.

- The first command we'll use is `~`, which shows information about all threads in the debugged process:

```
0:003> ~
0  Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
1  Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
2  Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
. 3  Id: 874c.bb08 Suspend: 1 Teb: 00000001`222ab000 Unfrozen
```

The exact number of threads you'll see may be different than shown here.

One thing that is very important is the existence of proper symbols. Microsoft provides a public symbol server, which allows locating symbols for most modules by produced by Microsoft. This is essential in any low-level debugging.

- To set symbols quickly, enter the `.symfix` command.
- A better approach is to set up symbols once and have them available for all future debugging sessions. To do that, add a system environment variable named `_NT_SYMBOL_PATH` and set it to a string like the following:

```
SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
```

The middle part (between asterisks) is a local path for caching symbols on your local machine; you can select any path you like (including a network share, if sharing with a team is desired). Once this environment variable is set, next invocations of the debugger will find symbols automatically and load them from the Microsoft symbol server as needed.



The debuggers in the *Debugging Tools for Windows* are not the only tools that look for this environment variables. *Sysinternals* tools (e.g. *Process Explorer*, *Process Monitor*), Visual Studio, and others look for the same variable as well. You set it once, and get its benefit using multiple tools.

- To make sure you have proper symbols, enter the `!m` (loaded modules) command:

```
0:003> lm
start           end             module name
00007ff7`53820000 00007ff7`53863000  notepad    (deferred)
00007ffb`afbe0000 00007ffb`afca6000  efswrt     (deferred)
...
00007ffc`1db00000 00007ffc`1dba8000  shcore     (deferred)
00007ffc`1dbb0000 00007ffc`1dc74000  OLEAUT32   (deferred)
00007ffc`1dc80000 00007ffc`1dd22000  clbcatq   (deferred)
00007ffc`1dd30000 00007ffc`1de57000  COMDLG32   (deferred)
00007ffc`1de60000 00007ffc`1f350000  SHELL32    (deferred)
00007ffc`1f500000 00007ffc`1f622000  RPCRT4    (deferred)
00007ffc`1f630000 00007ffc`1f6e3000  KERNEL32   (pdb symbols)  c:\symbols\ker\
ne132.pdb\3B92DED9912D874A2BD08735BC0199A31\kernel132.pdb
00007ffc`1f700000 00007ffc`1f729000  GDI32      (deferred)
00007ffc`1f790000 00007ffc`1f7e2000  SHLWAPI   (deferred)
00007ffc`1f8d0000 00007ffc`1f96e000  sechost    (deferred)
00007ffc`1f970000 00007ffc`1fc9c000  combase    (deferred)
00007ffc`1fca0000 00007ffc`1fd3e000  msrvct    (deferred)
00007ffc`1fe50000 00007ffc`1fef3000  ADVAPI32   (deferred)
00007ffc`20380000 00007ffc`203ae000  IMM32      (deferred)
00007ffc`203e0000 00007ffc`205cd000  ntd11     (pdb symbols)  c:\symbols\ntd\
11.pdb\E7EEB80BFAA91532B88FF026DC6B9F341\ntd11.pdb
```

The list of modules shows all modules (DLLs and the EXE) loaded into the debugged process at this time. You can see the start and end virtual addresses into which each module is loaded. Following the module name you can see the symbol status of this module (in parenthesis). Possible values include:

- *deferred* - the symbols for this module were not needed in this debugging session so far, and so are not loaded at this time. The symbols will be loaded when needed (for example, if a call stack contains a function from that module). This is the default value.
- *pdb symbols* - proper public symbols have been loaded. The local path of the PDB file is displayed.
- *private pdb symbols* - private symbols are available. This would be the case for your own modules, compiled with Visual Studio. For Microsoft modules, this is very rare (at the time of writing, *combase.dll* is provided with private symbols). With private symbols, you have information about local variables and private types.
- *export symbols* - only exported symbols are available for this DLL. This typically means there are no symbols for this module, but the debugger is able to use the exported symbols. It's better than no symbols at all, but could be confusing, as the debugger will use the closest export it can find, but the real function is most likely different.
- *no symbols* - this module's symbols were attempted to be located, but nothing was found, not even exported symbols (such modules don't have exported symbols, as is the case of an executable or driver files).

You can force loading of a module's symbols using the following command:

```
.reload /f modulename.dll
```

This will provide definitive evidence to the availability of symbols for this module.

Symbol paths can also be configured in the debugger's settings dialog.

Open the *File / Settings* menu and locate *Debugging Settings*. You can then add more paths for symbol searching. This is useful if debugging your own code, so you would like the debugger to search your directories where relevant PDB files may be found (see figure 5-3).

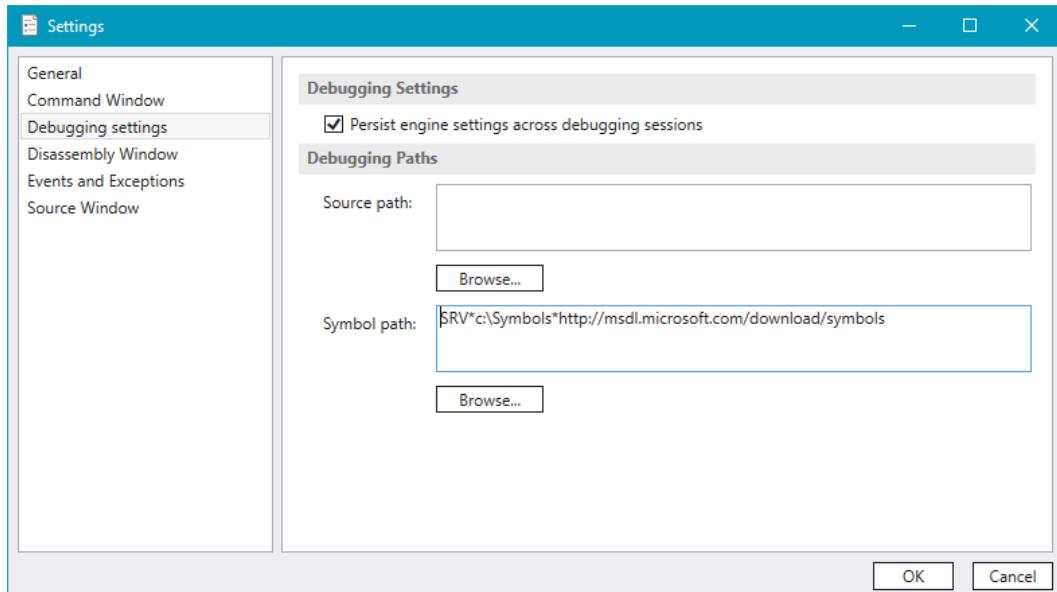


Figure 5-3: Symbols and source paths configuration

Make sure you have symbols configured correctly before you proceed. To diagnose any issues, you can enter the `!sym noisy` command that logs detailed information for symbol load attempts.

Back to the thread list - notice that one of the threads has a dot in front of its data. This is the current thread as far as the debugger is concerned. This means that any command issued that involves a thread, where the thread is not explicitly specified, will work on that thread. This "current thread" is also shown in the prompt - the number to the right of the colon is the current thread index (3 in this example).

Enter the `k` command, that shows the stack trace for the current thread:

```
0:003> k
# Child-SP          RetAddr          Call Site
00 00000001`224ffbd8 00007ffc`204aef5b ntdll!DbgBreakPoint
01 00000001`224ffbe0 00007ffc`1f647974 ntdll!DbgUiRemoteBreakin+0x4b
02 00000001`224ffc10 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
03 00000001`224ffc40 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```



How can you tell that you don't have proper symbols except using the `!m` command? If you see very large offsets from the beginning of a function, this is probably not the real function name - it's just the closest one the debugger knows about. "Large offsets" is obviously a relative term, but a good rule of thumb is that a 4-hex digit offset is almost always wrong.

You can see the list of calls made on this thread (user-mode only, of course). The top of the call stack in the above output is the function `DbgBreakPoint` located in the module `ntdll.dll`. The general format of addresses with symbols is `modulename!functionname+offset`. The offset is optional and could be zero if it's exactly the start of this function. Also notice the module name is without an extension.

In the output above, `DbgBreakpoint` was called by `DbgUiRemoteBreakIn`, which was called by `BaseThreadInitThunk`, and so on.

This thread, by the way, was injected by the debugger in order to break into the target forcefully.

To switch to a different thread, use the following command: `~ns` where *n* is the thread index. Let's switch to thread 0 and then display its call stack:

```
0:003> ~0s
win32u!NtUserGetMessage+0x14:
00007ffc`1c4b1164 c3          ret
0:000> k
* Child-SP      RetAddr      Call Site
00 00000001`2247f998 00007ffc`1d802fdb win32u!NtUserGetMessage+0x14
01 00000001`2247f9a0 00007ff7`5382449f USER32!GetMessageW+0x2d
02 00000001`2247fa00 00007ff7`5383ae07 notepad!WinMain+0x267
03 00000001`2247fb00 00007ffc`1f647974 notepad!__mainCRTStartup+0x19f
04 00000001`2247fb00 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
05 00000001`2247fbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

This is *Notepad*'s main (first) thread. The top of the stack shows the thread waiting for UI messages (`win32u!NtUserGetMessage`). The thread is actually waiting in kernel mode, but this is invisible from a user-mode debugger's view.

An alternative way to show the call stack of another thread without switching to it, is to use the tilde and thread number before the actual command. The following output is for thread 1's stack:

```
0:000> ~1k
# Child-SP          RetAddr          Call Site
00 00000001`2267f4c8 00007ffc`204301f4 ntdll!NtWaitForWorkViaWorkerFactory+0x14
01 00000001`2267f4d0 00007ffc`1f647974 ntdll!TppWorkerThread+0x274
02 00000001`2267f7c0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
03 00000001`2267f7f0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```



The above call stack is very common, and indicates a thread that is part of the thread pool. TppWorkerThread is the thread entry point for thread pool threads (Tpp is short for “Thread Pool Private”).

Let's go back to the list of threads:

```
. 0  Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
  1  Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
  2  Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
# 3  Id: 874c.bb08 Suspend: 1 Teb: 00000001`222ab000 Unfrozen
```

Notice the dot has moved to thread 0 (current thread), revealing a hash sign (#) on thread 3. The thread marked with a hash (#) is the one that caused the last breakpoint (which in this case was our initial debugger attach).

The basic information for a thread provided by the ~ command is shown in figure 5-4.

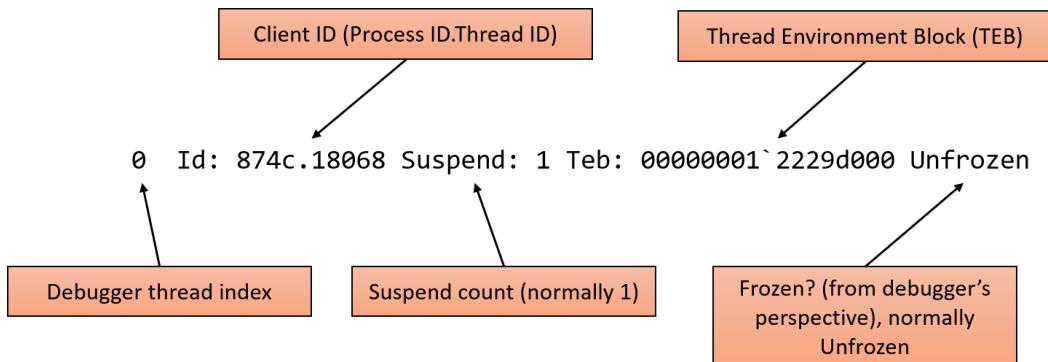


Figure 5-4: Thread information for the ~ command

Most numbers reported by WinDbg are hexadecimal by default. To convert a value to decimal, you can use the ? (evaluate expression) command.

Type the following to get the decimal process ID (you can then compare to the reported PID in Task Manager):

```
0:000> ? 874c  
Evaluate expression: 34636 = 00000000`0000874c
```

You can express decimal numbers with the `0n` prefix, so you can get the inverse result as well:

```
0:000> ? 0n34636  
Evaluate expression: 34636 = 00000000`0000874c
```



The `0y` prefix can be used in *WinDbg* to specify binary values. For example, using `0y1100` is the same as `0n12` as is `0xc`. You can use the `?` command to see the converted values.

You can examine the TEB of a thread by using the `!teb` command. Using `!teb` without an address shows the TEB of the current thread:

```
0:000> !teb  
TEB at 000000012229d000  
ExceptionList: 0000000000000000  
StackBase: 0000000122480000  
StackLimit: 000000012246f000  
SubSystemTib: 0000000000000000  
FiberData: 0000000000001e00  
ArbitraryUserPointer: 0000000000000000  
Self: 000000012229d000  
EnvironmentPointer: 0000000000000000  
ClientId: 000000000000874c . 0000000000018068  
RpcHandle: 0000000000000000  
Tls Storage: 000001c93676c940  
PEB Address: 000000012229c000  
LastErrorValue: 0  
LastStatusValue: 8000001a  
Count Owned Locks: 0  
HardErrorMode: 0  
0:000> !teb 00000001`222a5000  
TEB at 00000001222a5000  
ExceptionList: 0000000000000000  
StackBase: 0000000122680000  
StackLimit: 000000012266f000  
SubSystemTib: 0000000000000000  
FiberData: 0000000000001e00  
ArbitraryUserPointer: 0000000000000000  
Self: 00000001222a5000
```

```

EnvironmentPointer: 0000000000000000
ClientId: 000000000000874c . 00000000000046ac
RpcHandle: 0000000000000000
Tls Storage: 000001c936764260
PEB Address: 000000012229c000
LastErrorValue: 0
LastStatusValue: c0000034
Count Owned Locks: 0
HardErrorMode: 0

```

Some data shown by the !teb command is relatively known or easy to guess:

- *StackBase* and *StackLimit* - user-mode current stack base and stack limit for the thread.
- *ClientId* - process and thread IDs.
- *LastErrorCode* - last Win32 error code (`GetLastError`).
- *TlsStorage* - *Thread Local Storage* (TLS) array for this thread (full explanation of TLS is beyond the scope of this book).
- *PEB Address* - address of the Process Environment Block (PEB), viewable with the !peb command.
- *LastStatusValue* - last NTSTATUS value returned from a system call.
- The !teb command (and similar commands) shows parts of the real data structure behind the scenes, in this case _TEB. You can always look at the real structure using the dt (display type) command:

```

0:000> dt ntdll!_teb
+0x000 NtTib : _NT_TIB
+0x038 EnvironmentPointer : Ptr64 Void
+0x040 ClientId : _CLIENT_ID
+0x050 ActiveRpcHandle : Ptr64 Void
+0x058 ThreadLocalStoragePointer : Ptr64 Void
+0x060 ProcessEnvironmentBlock : Ptr64 _PEB
...
+0x1808 LockCount : UInt4B
+0x180c WowTebOffset : Int4B
+0x1810 ResourceRetValue : Ptr64 Void
+0x1818 ReservedForWdf : Ptr64 Void
+0x1820 ReservedForCrt : UInt8B
+0x1828 EffectiveContainerId : _GUID

```

Notice that *WinDbg* is **not** case sensitive when it comes to symbols. Also, notice the structure name starting with an underscore; this is the way most structures are defined in Windows (user-mode and

kernel-mode). Using the typedef name (without the underscore) may or may not work, so always using the underscore is recommended.



How do you know which module defines a structure you wish to view? If the structure is documented, the module would be listed in the docs for the structure. You can also try specifying the structure without the module name, forcing the debugger to search for it. Generally, you “know” where the structure is defined with experience and sometimes context.

If you attach an address to the previous command, you can get the actual values of data members:

```
0:000> dt ntdll!_teb 00000001`2229d000
+0x000 NtTib          : _NT_TIB
+0x038 EnvironmentPointer : (null)
+0x040 ClientId       : _CLIENT_ID
+0x050 ActiveRpcHandle : (null)
+0x058 ThreadLocalStoragePointer : 0x0000001c9`3676c940 Void
+0x060 ProcessEnvironmentBlock : 0x00000001`2229c000 _PEB
+0x068 LastErrorValue   : 0
...
+0x1808 LockCount      : 0
+0x180c WowTebOffset    : 0n0
+0x1810 ResourceRetValue : 0x0000001c9`3677fd00 Void
+0x1818 ReservedForWdf  : (null)
+0x1820 ReservedForCrt  : 0
+0x1828 EffectiveContainerId : _GUID {00000000-0000-0000-0000-000000000000}
```

Each member is shown with its offset from the beginning of the structure, its name, and its value. Simple values are shown directly, while structure values (such as `NtTib` above) are shown with a hyperlink. Clicking this hyperlink provides the details of the structure.

Click on the `NtTib` member above to show the details of this data member:

```
0:000> dx -r1 (*((ntdll!_NT_TIB *)0x12229d000))
(*((ntdll!_NT_TIB *)0x12229d000)) [Type: _NT_TIB]
[+0x000] ExceptionList    : 0x0 [Type: _EXCEPTION_REGISTRATION_RECORD *]
[+0x008] StackBase        : 0x122480000 [Type: void *]
[+0x010] StackLimit       : 0x12246f000 [Type: void *]
[+0x018] SubSystemTib     : 0x0 [Type: void *]
[+0x020] FiberData        : 0x1e00 [Type: void *]
[+0x020] Version          : 0x1e00 [Type: unsigned long]
[+0x028] ArbitraryUserPointer : 0x0 [Type: void *]
[+0x030] Self              : 0x12229d000 [Type: _NT_TIB *]
```

The debugger uses the newer `dx` command to view data. See the section “Advanced Debugging with WinDbg” later in this chapter for more on the `dx` command.

If you don't see hyperlinks, you may be using a very old *WinDbg*, where Debugger Markup Language (DML) is not on by default. You can turn it on with the `.prefer_dml 1` command.

Now let's turn our attention to breakpoints. Let's set a breakpoint when a file is opened by *notepad*.

- Type the following command to set a breakpoint in the `CreateFile` API function:

```
0:000> bp kernel32!createfilew
```

Notice the function name is in fact `CreateFileW`, as there is no function called `CreateFile`. In code, this is a macro that expands to `CreateFileW` (wide, Unicode version) or `CreateFileA` (ASCII or Ansi version) based on a compilation constant named `UNICODE`. *WinDbg* responds with nothing. This is a good thing.



The reason there are two sets of functions for most APIs where strings are involved is a historical one. In any case, Visual Studio projects define the `UNICODE` constant by default, so Unicode is the norm. This is a good thing - most of the A functions convert their input to Unicode and call the W function.

You can list the existing breakpoints with the `b1` command:

```
0:000> b1
0 e Disable Clear 00007ffc`1f652300 0001 (0001) 0:***** KERNEL32!CreateFileW
```

You can see the breakpoint index (0), whether it's enabled or disabled (e=enabled, d=disabled), and you get DML hyperlinks to disable (`bd` command) and delete (`bc` command) the breakpoint.

Now let *notepad* continue execution, until the breakpoint hits:

Type the `g` command or press the *Go* button on the toolbar or hit *F5*:

You'll see the debugger showing **Busy** in the prompt and the command area shows **Debuggee is running**, meaning you cannot enter commands until the next break.

Notepad should now be alive. Go to its *File* menu and select *Open....* The debugger should spew details of module loads and then break:

```
Breakpoint 0 hit
KERNEL32!CreateFileW:
00007ffc`1f652300 ff25aa670500    jmp     qword ptr [KERNEL32!_imp_CreateFileW \
(00007ffc`1f6a8ab0)] ds:00007ffc`1f6a8ab0={KERNELBASE!CreateFileW (00007ffc`1c7\
5e260)}
```

- We have hit the breakpoint! Notice the thread in which it occurred. Let's see what the call stack looks like (it may take a while to show if the debugger needs to download symbols from Microsoft's symbol server):

```
0:002> k
# Child-SP          RetAddr           Call Site
00 00000001`226fab08 00007ffc`061c8368 KERNEL32!CreateFileW
01 00000001`226fab10 00007ffc`061c5d4d mscoreei!RuntimeDesc::VerifyMainRuntimeM\
odule+0x2c
02 00000001`226fab60 00007ffc`061c6068 mscoreei!FindRuntimesInInstallRoot+0x2fb
03 00000001`226fb3e0 00007ffc`061cb748 mscoreei!GetOrCreateSxSProcessInfo+0x94
04 00000001`226fb460 00007ffc`061cb62b mscoreei!CLRMetaHostPolicyImpl::GetReque\
stedRuntimeHelper+0xfc
05 00000001`226fb740 00007ffc`061ed4e6 mscoreei!CLRMetaHostPolicyImpl::GetReque\
stedRuntime+0x120
...
21 00000001`226fede0 00007ffc`1df025b2 SHELL32!CFSIconOverlayManager::LoadNonlo\
adedOverlayIdentifiers+0xaa
22 00000001`226ff320 00007ffc`1df022af SHELL32!EnableExternalOverlayIdentifiers\
+0x46
23 00000001`226ff350 00007ffc`1def434e SHELL32!CFSIconOverlayManager::RefreshOv\
erlayImages+0xff
24 00000001`226ff390 00007ffc`1cf250a3 SHELL32!SHELL32_GetIconOverlayManager+0x\
6e
25 00000001`226ff3c0 00007ffc`1ceb2726 windows_storage!CFSFolder::_GetOverlayIn\
fo+0x12b
26 00000001`226ff470 00007ffc`1cf3108b windows_storage!CAutoDestItemsFolder::Ge\
tOverlayIndex+0xb6
27 00000001`226ff4f0 00007ffc`1cf30f87 windows_storage!CRegFolder::_GetOverlayI\
nfo+0xbff
28 00000001`226ff5c0 00007ffb`df8fc4d1 windows_storage!CRegFolder::GetOverlayIn\
dex+0x47
29 00000001`226ff5f0 00007ffb`df91f095 explorerframe!CNscOverlayTask::_Extract+\
0x51
2a 00000001`226ff640 00007ffb`df8f70c2 explorerframe!CNscOverlayTask::InternalR\
esumeRT+0x45
```

```
2b 00000001`226ff670 00007ffc`1cf7b58c explorerframe!CRunnableTask::Run+0xb2  
2c 00000001`226ff6b0 00007ffc`1cf7b245 windows_storage!CShellTask::TT_Run+0x3c  
2d 00000001`226ff6e0 00007ffc`1cf7b125 windows_storage!CShellTaskThread::Thread\  
Proc+0xdd  
2e 00000001`226ff790 00007ffc`1db32ac6 windows_storage!CShellTaskThread::s_Thre\  
adProc+0x35  
2f 00000001`226ff7c0 00007ffc`204521c5 shcore!ExecuteWorkItemThreadProc+0x16  
30 00000001`226ff7f0 00007ffc`204305c4 ntdll!RtlpTpWorkCallback+0x165  
31 00000001`226ff8d0 00007ffc`1f647974 ntdll!TppWorkerThread+0x644  
32 00000001`226ffbc0 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14  
33 00000001`226ffb0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Your call stack may be different, as it depends on the Windows version, and any extensions that may be loaded and used by the open file dialog box.

What can we do at this point? You may wonder what file is being opened. We can get that information based on the calling convention of the `CreateFileW` function. Since this is a 64-bit process (and the processor is Intel/AMD), the calling convention states that the first integer/pointer arguments are passed in the *RCX*, *RDX*, *R8*, and *R9* registers (in this order). Since the file name in `CreateFileW` is the first argument, the relevant register is *RCX*.

You can get more information on calling conventions in the Debugger documentation (or in several web resources).

Display the value of the *RCX* register with the `r` command (you'll get a different value):

```
0:002> r rcx  
rcx=00000001226fabf8
```

We can view the memory pointed by *RCX* with various `d` (display) family of commands. Here is the `db` command, interpreting the data as bytes.

```
0:002> db 00000001226fabf8
00000001`226fabf8  43 00 3a 00 5c 00 57 00-69 00 6e 00 64 00 6f 00  C...\.W.i.n\
.d.o.
00000001`226fac08  77 00 73 00 5c 00 4d 00-69 00 63 00 72 00 6f 00  w.s.\.M.i.c\
.r.o.
00000001`226fac18  73 00 6f 00 66 00 74 00-2e 00 4e 00 45 00 54 00  s.o.f.t...N\
.E.T.
00000001`226fac28  5c 00 46 00 72 00 61 00-6d 00 65 00 77 00 6f 00  \.F.r.a.m.e\
.w.o.
00000001`226fac38  72 00 6b 00 36 00 34 00-5c 00 5c 00 76 00 32 00  rk.6.4.\.\
.v.2.
00000001`226fac48  2e 00 30 00 2e 00 35 00-30 00 37 00 32 00 37 00  ..0...5.0.7\
.2.7.
00000001`226fac58  5c 00 63 00 6c 00 72 00-2e 00 64 00 6c 00 6c 00  \.c.1.r...d\
.1.1.
00000001`226fac68  00 00 76 1c fc 7f 00 00-00 00 00 00 00 00 00 00  ..v.....\
....
```

The db command shows the memory in bytes, and ASCII characters on the right. It's pretty clear what the file name is, but because the string is Unicode, it's not very convenient to see.

Use the du command to view Unicode string more conveniently:

```
0:002> du 00000001226fabf8
00000001`226fabf8  "C:\Windows\Microsoft.NET\Framework"
00000001`226fac38  "rk64\v2.0.50727\clr.dll"
```

You can use a register value directly by prefixing its name with @:

```
0:002> du @rcx
00000001`226fabf8  "C:\Windows\Microsoft.NET\Framework"
00000001`226fac38  "rk64\v2.0.50727\clr.dll"
```

Similarly, you can view the value of the second argument by looking at the rdx register.

Now let's set another breakpoint in the native API that is called by CreateFileW - NtCreateFile:

```
0:002> bp ntdll!ntcreatefile
0:002> b1
  0 e Disable Clear  00007ffc`1f652300  0001 (0001)  0:**** KERNEL32!CreateFil\
eW
  1 e Disable Clear  00007ffc`20480120  0001 (0001)  0:**** ntdll!NtCreateFile
```

Notice the native API never uses W or A - it always works with Unicode strings (in fact it expects UNICODE_STRING structures, as we've seen already).

Continue execution with the g command. The debugger should break:

```
Breakpoint 1 hit
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1      mov     r10,rcx
```

Check the call stack again:

```
0:002> k
# Child-SP          RetAddr           Call Site
00 00000001`226fa938 00007ffc`1c75e5d6 ntdll!NtCreateFile
01 00000001`226fa940 00007ffc`1c75e2c6 KERNELBASE!CreateFileInternal+0x2f6
02 00000001`226faab0 00007ffc`061c8368 KERNELBASE!CreateFileW+0x66
03 00000001`226fab10 00007ffc`061c5d4d mscoreei!RuntimeDesc::VerifyMainRuntimeModule+0x2c
04 00000001`226fab60 00007ffc`061c6068 mscoreei!FindRuntimesInInstallRoot+0x2fb
05 00000001`226fb3e0 00007ffc`061cb748 mscoreei!GetOrCreateSxSProcessInfo+0x94
...
...
```

List the next 8 instructions that are about to be executed with the **u** (unassemble or disassemble) command:

```
0:002> u
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1      mov     r10,rcx
00007ffc`20480123 b855000000  mov     eax,55h
00007ffc`20480128 f604250803fe7f01 test   byte ptr [SharedUserData+0x308 (0000`0000`7ffe0308)],1
00007ffc`20480130 7503       jne    ntdll!NtCreateFile+0x15 (00007ffc`204\80135)
00007ffc`20480132 0f05       syscall
00007ffc`20480134 c3         ret
00007ffc`20480135 cd2e       int    2Eh
00007ffc`20480137 c3         ret
```

Notice the value 0x55 is copied to the *EAX* register. This is the system service number for *NtCreateFile*, as described in chapter 1. The *syscall* instruction shown is the one causing the transition to kernel-mode, and then executing the *NtCreateFile* system service itself.

You can step over the next instruction with the *p* command (step - hit *F10* as an alternative). You can step into a function (in case of assembly, this is the *call* instruction) with the *t* command (trace - hit *F11* as an alternative):

```

0:002> p
Breakpoint 1 hit
ntdll!NtCreateFile:
00007ffc`20480120 4c8bd1      mov     r10,rcx
0:002> p
ntdll!NtCreateFile+0x3:
00007ffc`20480123 b855000000      mov     eax,55h
0:002> p
ntdll!NtCreateFile+0x8:
00007ffc`20480128 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (0000\
0000`7ffe0308)],1 ds:00000000`7ffe0308=00
0:002> p
ntdll!NtCreateFile+0x10:
00007ffc`20480130 7503      jne     ntdll!NtCreateFile+0x15 (00007ffc`204\
80135) [br=0]
0:002> p
ntdll!NtCreateFile+0x12:
00007ffc`20480132 0f05      syscall

```

Stepping inside a `syscall` is not possible, as we're in user-mode. When we step over/into it, all is done and we get back a result.

```

0:002> p
ntdll!NtCreateFile+0x14:
00007ffc`20480134 c3      ret

```

The return value of functions in x64 calling convention is stored in `EAX` or `RAX`. For system calls, it's an `NTSTATUS`, so `EAX` contains the returned status:

```

0:002> r eax
eax=c0000034

```

Zero means success, and a negative value (in two's complement, most significant bit is set) means an error. We can get a textual description of the error with the `!error` command:

```

0:002> !error @eax
Error code: (NTSTATUS) 0xc0000034 (3221225524) - Object Name not found.

```

This means the file wasn't found on the system.

Disable all breakpoints and let *Notepad* continue execution normally:

```
0:002> bd *
0:002> g
```

Since we have no breakpoints at this time, we can force a break by clicking the *Break* button on the toolbar, or hitting *Ctrl+Break* on the keyboard:

```
874c.16a54): Break instruction exception - code 80000003 (first chance)
ntdll!DbgBreakPoint:
00007ffc`20483080 cc           int      3
```

Notice the thread number in the prompt. Show all current threads:

```
0:022> ~
  0  Id: 874c.18068 Suspend: 1 Teb: 00000001`2229d000 Unfrozen
  1  Id: 874c.46ac Suspend: 1 Teb: 00000001`222a5000 Unfrozen
  2  Id: 874c.152cc Suspend: 1 Teb: 00000001`222a7000 Unfrozen
  3  Id: 874c.f7ec Suspend: 1 Teb: 00000001`222ad000 Unfrozen
  4  Id: 874c.145b4 Suspend: 1 Teb: 00000001`222af000 Unfrozen
...
  18 Id: 874c.f0c4 Suspend: 1 Teb: 00000001`222d1000 Unfrozen
  19 Id: 874c.17414 Suspend: 1 Teb: 00000001`222d3000 Unfrozen
  20 Id: 874c.c878 Suspend: 1 Teb: 00000001`222d5000 Unfrozen
  21 Id: 874c.d8c0 Suspend: 1 Teb: 00000001`222d7000 Unfrozen
  22 Id: 874c.16a54 Suspend: 1 Teb: 00000001`222e1000 Unfrozen
  23 Id: 874c.10838 Suspend: 1 Teb: 00000001`222db000 Unfrozen
  24 Id: 874c.10cf0 Suspend: 1 Teb: 00000001`222dd000 Unfrozen
```

Lots of threads, right? These were created by the common open dialog, so not the direct fault of *Notepad*.

Continue exploring the debugger in any way you want!



Find out the system service numbers for `NtWriteFile` and `NtReadFile`.

If you close *Notepad*, you'll hit a breakpoint at process termination:

```

ntdll!NtTerminateProcess+0x14:
00007ffc`2047fc14 c3          ret
0:000> k
# Child-SP      RetAddr      Call Site
00 00000001`2247f6a8 00007ffc`20446dd8 ntdll!NtTerminateProcess+0x14
01 00000001`2247f6b0 00007ffc`1f64d62a ntdll!RtlExitUserProcess+0xb8
02 00000001`2247f6e0 00007ffc`061cee58 KERNEL32!ExitProcessImplementation+0xa
03 00000001`2247f710 00007ffc`0644719e mscoreei!RuntimeDesc::ShutdownAllActiveR\
untimes+0x287
04 00000001`2247fa00 00007ffc`1fcda291 mscoreei!ShellShim_CorExitProcess+0x11e
05 00000001`2247fa30 00007ffc`1fcda2ad msvcrt!_crtCorExitProcess+0x4d
06 00000001`2247fa60 00007ffc`1fcda925 msvcrt!_crtExitProcess+0xd
07 00000001`2247fa90 00007ff7`5383ae1e msvcrt!doexit+0x171
08 00000001`2247fb00 00007ffc`1f647974 notepad!__mainCRTStartup+0x1b6
09 00000001`2247fb00 00007ffc`2044a271 KERNEL32!BaseThreadInitThunk+0x14
0a 00000001`2247fbf0 00000000`00000000 ntdll!RtlUserThreadStart+0x21

```

You can use the `q` command to quit the debugger. If the process is still alive, it will be terminated. An alternative is to use the `.detach` command to disconnect from the target without killing it.

Kernel Debugging

User-mode debugging involves the debugger attaching to a process, setting breakpoints that cause the process' threads to become suspended, and so on. Kernel-mode debugging, on the other hand, involves controlling the entire machine with the debugger. This means that if a breakpoint is set and then hit, the entire machine is frozen. Clearly, this cannot be achieved with a single machine. In full kernel debugging, two machines are involved: a host (where the debugger runs) and a target (being debugged). The target can, however, be a virtual machine hosted on the same machine (host) where the debugger executes. Figure 5-5 shows a host and target connected via some connection medium.

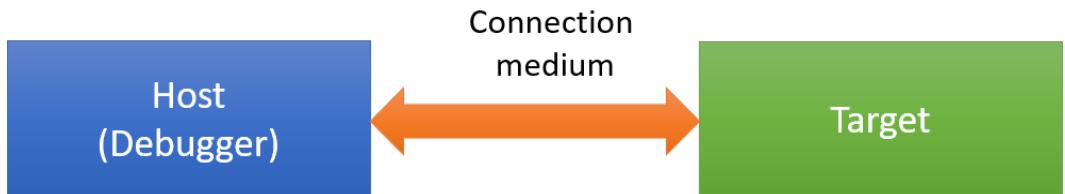


Figure 5-5: Host-target connection

Before we get into full kernel debugging, we'll take a look at its simpler cousin - local kernel debugging.

Local Kernel Debugging

Local kernel debugging (LKD) allows viewing system memory and other system information on the local machine. The primary difference between local and full kernel debugging, is that with LKD there

is no way to set up breakpoints, which means you're always looking at the current state of the system. It also means that things change, even while commands are being executed, so some information may be stale or unreliable. With full kernel debugging, commands can only be entered while the target system is in a breakpoint, so system state is unchanged.

To configure LKD, enter the following in an elevated command prompt and then restart the system:

```
bcdedit /debug on
```



Local Kernel Debugging is protected by *Secure Boot* on Windows 10, Server 2016, and later. To activate LKD you'll have to disable Secure Boot in the machine's BIOS settings. If, for whatever reason, this is not possible, there is an alternative using the Sysinternals *LiveKd* tool. Copy *LiveKd.exe* to the *Debugging Tools for Windows* main directory. Then launch *WinDbg* using LiveKd with the following command: `livekd -w`. The experience is not the same, as data may become stale because of the way *Livekd* works, and you may need to exit the debugger and relaunch from time to time.

After the system is restarted, launch *WinDbg* elevated (the 64-bit one, if you are on a 64-bit system). Select the menu *File / Attach To Kernel* (*WinDbg* preview) or *File / Kernel Debug...* (classic *WinDbg*). Select the *Local* tab and click *OK*. You should see output similar to the following:

```
Microsoft (R) Windows Debugger Version 10.0.22415.1003 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Connected to Windows 10 22000 x64 target at (Wed Sep 29 10:57:30.682 2021 (UTC \
+ 3:00)), ptr64 TRUE

***** Path validation summary *****
Response                      Time (ms)      Location
Deferred
osoft.com/download/symbols
Symbol search path is: SRV*c:\symbols*https://msdl.microsoft.com/download/symbo\
ls
Executable search path is:
Windows 10 Kernel Version 22000 MP (6 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Edition build lab: 22000.1.amd64fre.co_release.210604-1628
Machine Name:
Kernel base = 0xfffff802`07a00000 PsLoadedModuleList = 0xfffff802`08629710
Debug session time: Wed Sep 29 10:57:30.867 2021 (UTC + 3:00)
System Uptime: 0 days 16:44:39.106
```

Note the prompt displays *lkd*. This indicates Local Kernel Debugging is active.

Local kernel Debugging Tutorial

If you're familiar with kernel debugging commands, you can safely skip this section.

You can display basic information for all processes running on the system with the process 0 0 command:

```
1kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS fffffd104936c8040
    SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
    DirBase: 006d5000 ObjectTable: fffffa58d3cc44d00 HandleCount: 3909.
    Image: System

PROCESS fffffd104936e2080
    SessionId: none Cid: 0058 Peb: 00000000 ParentCid: 0004
    DirBase: 0182c000 ObjectTable: fffffa58d3cc4ea40 HandleCount: 0.
    Image: Secure System

PROCESS fffffd1049370a080
    SessionId: none Cid: 0090 Peb: 00000000 ParentCid: 0004
    DirBase: 011b6000 ObjectTable: fffffa58d3cc65a80 HandleCount: 0.
    Image: Registry

PROCESS fffffd10497dd0080
    SessionId: none Cid: 024c Peb: bc6c2ba000 ParentCid: 0004
    DirBase: 10be4b000 ObjectTable: fffffa58d3d49ddc0 HandleCount: 60.
    Image: smss.exe

...
```

For each process, the following information is displayed:

- The address attached to the *PROCESS* text is the EPROCESS address of the process (in kernel space, of course).
- *SessionId* - the session the process is running under.
- *Cid* - (client ID) the unique process ID.
- *Peb* - the address of the *Process Environment Block* (PEB). This address is in user space, naturally.
- *ParentCid* - (parent client ID) the process ID of the parent process. Note that it's possible the parent process no longer exists, so this ID may belong to some process created after the parent process terminated.
- *DirBase* - physical address of the Master Page Directory for this process, used as the basis for virtual to physical address translation. On x64, this is known as *Page Map Level 4*, and on x86 it's *Page Directory Pointer Table* (PDPT).
- *ObjectTable* - pointer to the private handle table for the process.

- *HandleCount* - number of handles in the handle table for this process.
- *Image* - executable name, or special process name for those not associated with an executable (such as *Secure System*, *System*, *Mem Compression*).

The !process command accepts at least two arguments. The first indicates the process of interest using its EPROCESS address or the unique Process ID, where zero means “all or any process”. The second argument is the level of detail to display (a bit mask), where zero means the least amount of detail. A third argument can be added to search for a particular executable. Here are a few examples:

List all processes running *explorer.exe*:

```
1kd> !process 0 0 explorer.exe
PROCESS fffffd1049e118080
SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3208.
Image: explorer.exe

PROCESS fffffd104a14e2080
SessionId: 1 Cid: 2548 Peb: 005c1000 ParentCid: 0314
DirBase: 140fe9000 ObjectTable: fffffa58d46a99500 HandleCount: 2613.
Image: explorer.exe
```

List more information for a specific process by specifying its address and a higher level of detail:

```
1kd> !process fffffd1049e7a60c0 1
PROCESS fffffd1049e7a60c0
SessionId: 1 Cid: 1374 Peb: d3e343000 ParentCid: 0314
DirBase: 37eb97000 ObjectTable: fffffa58d58a9de00 HandleCount: 224.
Image: dllhost.exe
VadRoot fffffd104b81c7db0 Vads 94 Clone 0 Private 455. Modified 2. Locked 0.
DeviceMap fffffa58d41354230
Token fffffa58d466e0060
ElapsedTime 01:04:36.652
UserTime 00:00:00.015
KernelTime 00:00:00.015
QuotaPoolUsage[PagedPool] 201696
QuotaPoolUsage[NonPagedPool] 13048
Working Set Sizes (now,min,max) (4330, 50, 345) (17320KB, 200KB, 1380KB)
PeakWorkingSetSize 4581
VirtualSize 2101383 Mb
PeakVirtualSize 2101392 Mb
PageFaultCount 5427
MemoryPriority BACKGROUND
BasePriority 8
```

CommitCharge	678
Job	fffffd104a05ed380

As can be seen from the above output, more information on the process is displayed. Some of this information is hyperlinked, allowing easy further examination. For example, the job this process is part of (if any) is a hyperlink, executing the !job command if clicked.

Click on the Job address hyperlink:

```
1kd> !job fffffd104a05ed380
Job at fffffd104a05ed380
  Basic Accounting Information
    TotalUserTime:          0x0
    TotalKernelTime:        0x0
    TotalCycleTime:         0x0
    ThisPeriodTotalUserTime: 0x0
    ThisPeriodTotalKernelTime: 0x0
    TotalPageFaultCount:   0x0
    TotalProcesses:         0x1
    ActiveProcesses:        0x1
    FreezeCount:            0
    BackgroundCount:        0
    TotalTerminatedProcesses: 0x0
    PeakJobMemoryUsed:      0x2f5
    PeakProcessMemoryUsed:   0x2f5
  Job Flags
    [wake notification allocated]
    [wake notification enabled]
    [timers virtualized]
  Limit Information (LimitFlags: 0x800)
  Limit Information (EffectiveLimitFlags: 0x403800)
    JOB_OBJECT_LIMIT_BREAKAWAY_OK
```



A Job is a kernel object that manages one or more processes, for which it can apply various limits and get accounting information. A discussion of jobs is beyond the scope of this book. More information can be found in the *Windows Internals 7th edition, part 1* and *Windows 10 System Programming, Part 1* books.

As usual, a command such as !job hides some information available in the real data structure. In this case, the type is EJOB. Use the command dt nt!_ejob with the job address to see all the details.

The PEB of a process can be viewed as well by clicking its hyperlink. This is similar to the !peb command used in user mode, but the twist here is that the correct process context must be set first, as the address is in user space. Click the Peb hyperlink. You should see something like this:

```
1kd> .process /p fffffd1049e7a60c0; !peb d3e343000
Implicit process is now fffffd104`9e7a60c0
PEB at 0000000d3e343000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: No
  ImageBaseAddress: 00007ff661180000
  NtGlobalFlag: 0
  NtGlobalFlag2: 0
  Ldr 00007ffb37ef9120
  Ldr.Initialized: Yes
  Ldr.InInitializationOrderModuleList: 000001d950004560 . 000001d95005a960
  Ldr.InLoadOrderModuleList: 000001d9500046f0 . 000001d95005a940
  Ldr.InMemoryOrderModuleList: 000001d950004700 . 000001d95005a950
          Base TimeStamp           Module
  7ff661180000 93f44fbf Aug 29 00:12:31 2048 C:\WINDOWS\system32\DllH\
ost.exe
  7ffb37d80000 50702a8c Oct 06 15:56:44 2012 C:\WINDOWS\SYSTEM32\ntd1\
1.dll
  7ffb36790000 ae0b35b0 Jul 13 01:50:24 2062 C:\WINDOWS\System32\KERN\
EL32.DLL
  ...

```

The correct process context is set with the .process meta command, and then the PEB is displayed. This is a general technique you need to use to show memory that is in user space - always make sure the debugger is set to the correct process context.

Execute the !process command again, but with the second bit set for the details:

```
1kd> !process fffffd1049e7a60c0 2
PROCESS fffffd1049e7a60c0
  SessionId: 1 Cid: 1374 Peb: d3e343000 ParentCid: 0314
  DirBase: 37eb97000 ObjectTable: fffffa58d58a9de00 HandleCount: 221.
  Image: dllhost.exe

  THREAD fffffd104a02de080 Cid 1374.022c Teb: 0000000d3e344000 Win32Thread: \
fffffd104b82ccbb0 WAIT: (UserRequest) UserMode Non-Alertable
    fffffd104b71d2860 SynchronizationEvent

  THREAD fffffd104a45e8080 Cid 1374.0f04 Teb: 0000000d3e352000 Win32Thread: \
fffffd104b82cccd90 WAIT: (WrUserRequest) UserMode Non-Alertable
    fffffd104adc5e0c0 QueueObject

  THREAD fffffd104a229a080 Cid 1374.1ed8 Teb: 0000000d3e358000 Win32Thread: \

```

```

fffffd104b82cf900 WAIT: (UserRequest) UserMode Non-Alertable
    fffffd104b71dfb60  NotificationEvent
    fffffd104ad02a740  QueueObject

    THREAD fffffd104b78ee040  Cid 1374.0330  Teb: 0000000d3e37a000 Win32Thread: \
0000000000000000 WAIT: (WrQueue) UserMode Alertable
    fffffd104adc4f640  QueueObject

```

Detail level 2 shows a summary of the threads in the process along with the object(s) they are waiting on (if any).

You can use other detail values (4, 8), or combine them, such as 3 (1 or 2).

Repeat the !process command again, but this time with no detail level. More information is shown for the process (the default in this case is full details):

```

1kd> !process fffffd1049e7a60c0
PROCESS fffffd1049e7a60c0
SessionId: 1  Cid: 1374      Peb: d3e343000  ParentCid: 0314
DirBase: 37eb97000  ObjectTable: fffffa58d58a9de00  HandleCount: 223.
Image: dllhost.exe
VadRoot fffffd104b81c7db0 Vads 94 Clone 0 Private 452. Modified 2. Locked 0.
DeviceMap fffffa58d41354230
Token                      fffffa58d466e0060
ElapsedTime                01:10:30.521
UserTime                   00:00:00.015
KernelTime                 00:00:00.015
QuotaPoolUsage[PagedPool]   201696
QuotaPoolUsage[NonPagedPool] 13048
Working Set Sizes (now,min,max) (4329, 50, 345) (17316KB, 200KB, 1380KB)
PeakWorkingSetSize          4581
VirtualSize                2101383 Mb
PeakVirtualSize             2101392 Mb
PageFaultCount              5442
MemoryPriority              BACKGROUND
BasePriority                8
CommitCharge                678
Job                         fffffd104a05ed380

THREAD fffffd104a02de080  Cid 1374.022c  Teb: 0000000d3e344000 Win32Thread: \
fffffd104b82ccbb0 WAIT: (UserRequest) UserMode Non-Alertable
    fffffd104b71d2860  SynchronizationEvent
    Not impersonating
DeviceMap                  fffffa58d41354230

```

```

Owning Process      fffffd1049e7a60c0      Image:      dllhost.exe
Attached Process    N/A       Image:      N/A
Wait Start TickCount 3641927      Ticks: 270880 (0:01:10:32.500)
Context Switch Count 27       IdealProcessor: 2
UserTime            00:00:00.000
KernelTime          00:00:00.000
Win32 Start Address 0x00007ff661181310
Stack Init fffffbe88b4bdf630 Current fffffbe88b4bdf010
Base fffffbe88b4be0000 Limit fffffbe88b4bd9000 Call 0000000000000000
Priority 8 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Kernel stack not resident.

THREAD fffffd104a45e8080 Cid 1374.0f04 Teb: 0000000d3e352000 Win32Thread: \
fffffd104b82cccd90 WAIT: (WrUserRequest) UserMode Non-Alertable
    fffffd104adc5e0c0 QueueObject
Not impersonating
DeviceMap           fffffa58d41354230
Owning Process      fffffd1049e7a60c0      Image:      dllhost.exe
Attached Process    N/A       Image:      N/A
Wait Start TickCount 3910734      Ticks: 2211 (0:00:00:34.546)
Context Switch Count 2684       IdealProcessor: 4
UserTime            00:00:00.046
KernelTime          00:00:00.078
Win32 Start Address 0x00007ffb3630f230
Stack Init fffffbe88b4c87630 Current fffffbe88b4c86a10
Base fffffbe88b4c88000 Limit fffffbe88b4c81000 Call 0000000000000000
Priority 10 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP            RetAddr        Call Site
fffffbe88`b4c86a50 ffffff802`07c5dc17  nt!KiSwapContext+0x76
fffffbe88`b4c86b90 ffffff802`07c5fac9  nt!KiSwapThread+0x3a7
fffffbe88`b4c86c70 ffffff802`07c59d24  nt!KiCommitThreadWait+0x159
fffffbe88`b4c86d10 ffffff802`07c8ac70  nt!KeWaitForSingleObject+0x234
fffffbe88`b4c86e00 ffffff9da`6d577d46  nt!KeWaitForMultipleObjects+0x540
fffffbe88`b4c86f00 ffffff99c`c175d920  0xfffffff9da`6d577d46
fffffbe88`b4c86f08 ffffff99c`c175d920  0xfffffff99c`c175d920
fffffbe88`b4c86f10 00000000`00000001  0xfffffff99c`c175d920
fffffbe88`b4c86f18 fffffd104`9a423df0  0x1
fffffbe88`b4c86f20 00000000`00000001  0xfffffd104`9a423df0
fffffbe88`b4c86f28 fffffbe88`b4c87100  0x1
fffffbe88`b4c86f30 00000000`00000000  0xfffffbe88`b4c87100
...

```

The command lists all threads within the process. Each thread is represented by its ETHREAD address attached to the text "THREAD". The call stack is listed as well - the module prefix "nt" represents the

kernel - there is no need to use the real kernel module name.

One of the reasons to use “nt” instead of explicitly stating the kernel’s module name is because these are different between 64 and 32 bit systems (*ntoskrnl.exe* on 64 bit, and *ntkrnlpa.exe* on 32 bit); and it’s a lot shorter.

User-mode symbols are not loaded by default, so thread stacks that span to user mode show just numeric addresses. You can load user symbols explicitly with `.reload /user` after setting the process context to the process of interest with the `.process` command:

```
1kd> !process 0 0 explorer.exe
PROCESS fffffd1049e118080
SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3217.
Image: explorer.exe

PROCESS fffffd104a14e2080
SessionId: 1 Cid: 2548 Peb: 005c1000 ParentCid: 0314
DirBase: 140fe9000 ObjectTable: fffffa58d46a99500 HandleCount: 2633.
Image: explorer.exe

1kd> .process /p fffffd1049e118080
Implicit process is now fffffd104^9e118080
1kd> .reload /user
Loading User Symbols
.....
1kd> !process fffffd1049e118080
PROCESS fffffd1049e118080
SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3223.
Image: explorer.exe
...
THREAD fffffd1049e47c400 Cid 1780.1754 Peb: 000000000078c000 Win32Thread: \
fffffd1049e5da7a0 WAIT: (WrQueue) UserMode Alertable
fffffd1049e076480 QueueObject
IRP List:
fffffd1049fbea9b0: (0006,0478) Flags: 00060000 Mdl: 00000000
fffffd1049efd6aa0: (0006,0478) Flags: 00060000 Mdl: 00000000
fffffd1049efee010: (0006,0478) Flags: 00060000 Mdl: 00000000
fffffd1049f3ef8a0: (0006,0478) Flags: 00060000 Mdl: 00000000
Not impersonating
```

```

DeviceMap          fffffa58d41354230
Owning Process    fffffd1049e118080      Image:      explorer.exe
Attached Process   N/A           Image:      N/A
Wait Start TickCount 3921033      Ticks: 7089 (0:00:01:50.765)
Context Switch Count 16410       IdealProcessor: 5
UserTime           00:00:00.265
KernelTime         00:00:00.234
Win32 Start Address ntdll!TppWorkerThread (0x00007ffb37d96830)
Stack Init fffffbe88b5fc7630 Current fffffbe88b5fc6d20
Base fffffbe88b5fc8000 Limit fffffbe88b5fc1000 Call 0000000000000000
Priority 9 BasePriority 8 PriorityDecrement 0 IoPriority 2 PagePriority 5
Child-SP          RetAddr          Call Site
fffffbe88`b5fc6d60 ffffff802`07c5dc17 nt!KiSwapContext+0x76
fffffbe88`b5fc6ea0 ffffff802`07c5fac9 nt!KiSwapThread+0x3a7
fffffbe88`b5fc6f80 ffffff802`07c62526 nt!KiCommitThreadWait+0x159
fffffbe88`b5fc7020 ffffff802`07c61f38 nt!KeRemoveQueueEx+0x2b6
fffffbe88`b5fc70d0 ffffff802`07c6479c nt!IoRemoveIoCompletion+0x98
fffffbe88`b5fc71f0 ffffff802`07e25075 nt!NtWaitForWorkViaWorkerFactory+0x\

39c
fffffbe88`b5fc7430 00007ffb`37e26e84 nt!KiSystemServiceCopyEnd+0x25 (Tra\
pFrame @ fffffbe88`b5fc74a0)
00000000`03def858 00007ffb`37d96b0f ntdll!NtWaitForWorkViaWorkerFactory\
+0x14
00000000`03def860 00007ffb`367a54e0 ntdll!TppWorkerThread+0x2df
00000000`03defb50 00007ffb`37d8485b KERNEL32!BaseThreadInitThunk+0x10
00000000`03defb80 00000000`00000000 ntdll!RtlUserThreadStart+0x2b

...

```

Notice the thread above has issued several IRPs as well. We'll discuss this in greater detail in chapter 7.

A thread's information can be viewed separately with the !thread command and the address of the thread. Check the debugger documentation for the description of the various pieces of information displayed by this command.

Other generally useful/interesting commands in kernel-mode debugging include:

- !pcr - display the Process Control Region (PCR) for a processor specified as an additional index (processor 0 is displayed by default if no index is specified).
- !vm - display memory statistics for the system and processes.
- !running - displays information on threads running on all processors on the system.

We'll look at more specific commands useful for debugging drivers in subsequent chapters.

Full Kernel Debugging

Full kernel debugging requires configuration on the host and target. In this section, we'll see how to configure a virtual machine as a target for kernel debugging. This is the recommended and most convenient setup for kernel driver work (when not developing device drivers for hardware). We'll go through the steps for configuring a Hyper-V virtual machine. If you're using a different virtualization technology (e.g. VMWare or VirtualBox), please consult that product's documentation or the web for the correct procedure to get the same results.

The target and host machines must communicate using some communication media. There are several options available. The fastest communication option is to use the network. Unfortunately, this requires the host and target to run Windows 8 at a minimum. Since Windows 7 is still a viable target, there is another convenient option - the COM (serial) port, which can be exposed as a named pipe to the host machine. All virtualization platforms allow redirecting a virtual serial port to a named pipe on the host. We'll look at both options.



Just like Local Kernel Debugging, the target machine cannot use *Secure Boot*. With full kernel debugging, there is no workaround.

Using a Virtual Serial Port

In this section, we'll configure the target and host to use a virtual COM port exposed as a named pipe to the host. In the next section, we'll configure kernel debugging using the network.

Configuring the Target

The target VM must be configured for kernel debugging, similar to local kernel debugging, but with the added connection media set to a virtual serial port on that machine.

One way to do the configuration is using *bcdedit* in an elevated command window:

```
bcdedit /debug on  
bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

Change the debug port number according to the actual virtual serial number (typically 1).

The VM must be restarted for these configurations to take effect. Before you do that, we can map the serial port to a named pipe. Here is the procedure for Hyper-V virtual machines:

If the Hyper-V VM is Generation 1 (older), there is a simple UI in the VM's settings to do the configuration. Use the *Add Hardware* option to add a serial port if there are none defined. Then configure the serial port to be mapped to a named port of your choosing. Figure 5-6 shows this dialog.

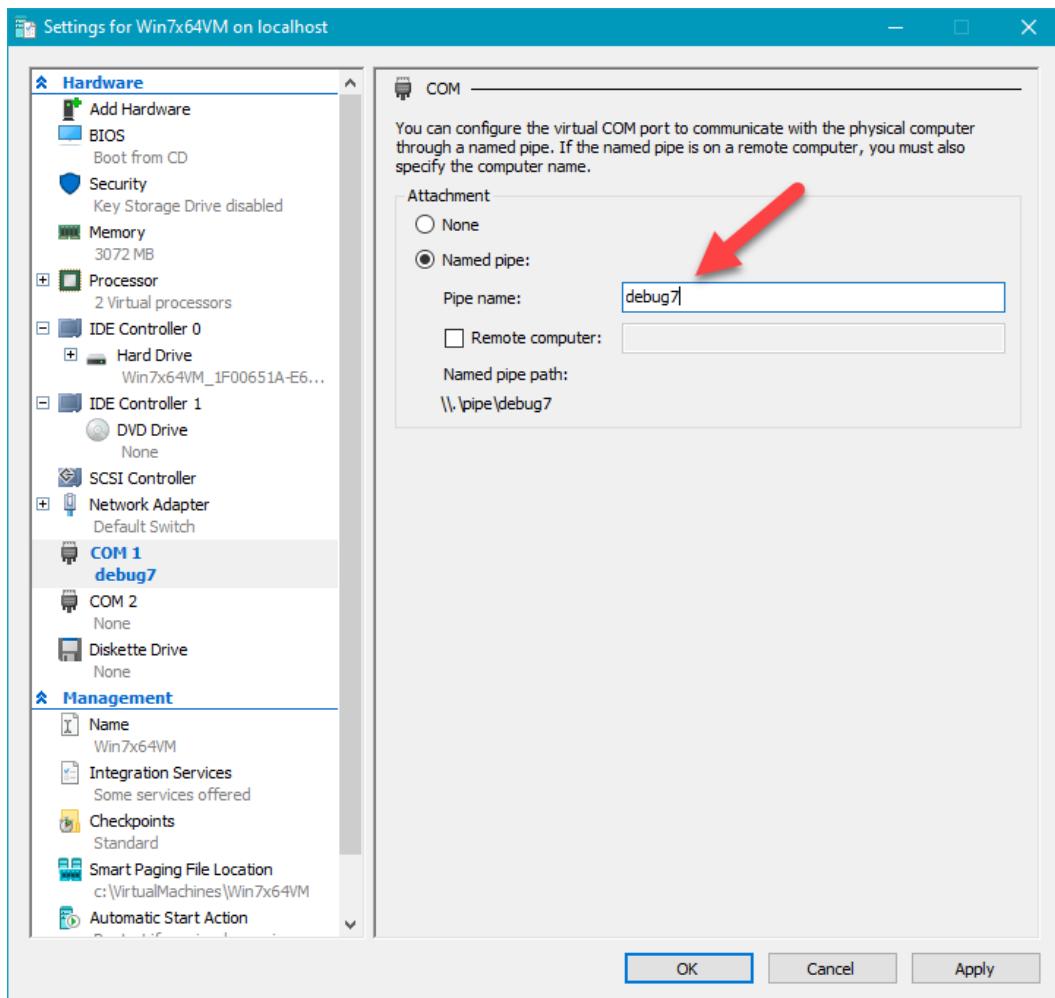


Figure 5-6: Mapping serial port to named pipe for Hyper-V Gen-1 VM

For Generation 2 VMs, no UI is currently available. To configure this, make sure the VM is shut down, and open an elevated *PowerShell* window.

Type the following to set a serial port mapped to a named pipe:

```
PS C:\>Set-VMComPort myvmname -Number 1 -Path "\.\pipe\debug"
```

Change the VM name appropriately and the COM port number as set inside the VM earlier with *bcdedit*. Make sure the pipe path is unique.

You can verify the settings are as expected with *Get-VMComPort*:

```
PS C:\>Get-VMComPort myvmname
```

VMName	Name	Path
myvmname	COM 1	\.\pipe\debug
myvmname	COM 2	

You can boot the VM - the target is now ready.

Configuring the Host

The kernel debugger must be properly configured to connect with the VM on the same serial port mapped to the same named pipe exposed on the host.

Launch the kernel debugger elevated, and select *File / Attach To Kernel*. Navigate to the *COM* tab. Fill in the correct details as they were set on the target. Figure 5-7 shows what these settings look like.

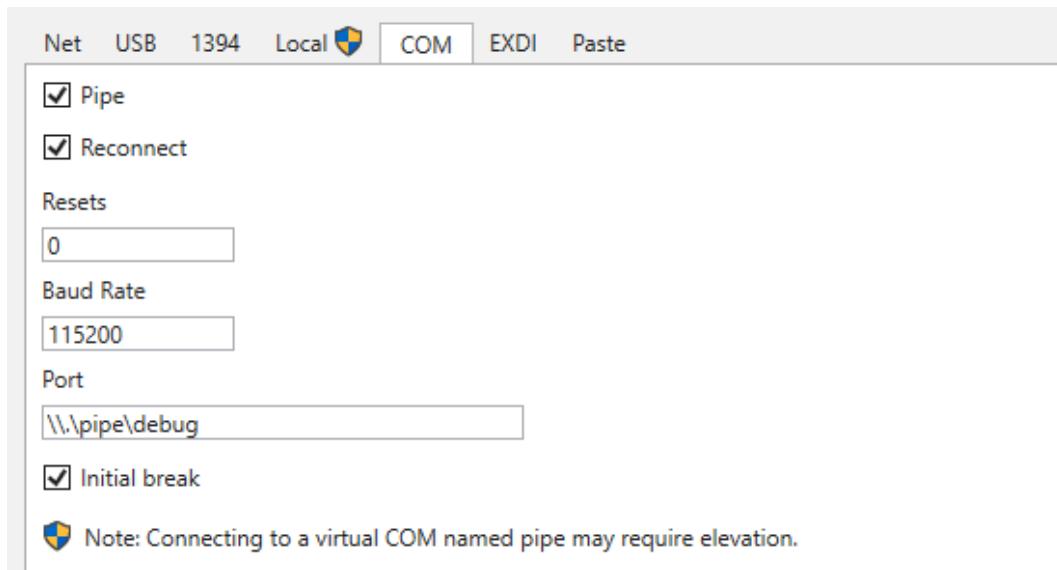


Figure 5-7: Setting host COM port configuration

Click OK. The debugger should attach to the target. If it does not, click the *Break* toolbar button. Here is some typical output:

```
Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Opened \\.\pipe\debug
Waiting to reconnect...
Connected to Windows 10 18362 x64 target at (Sun Apr 21 11:28:11.300 2019 (UTC \
+ 3:00)), ptr64 TRUE
Kernel Debugger connection established. (Initial Breakpoint requested)

***** Path validation summary *****
Response           Time (ms)    Location
Deferred          SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
soft.com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffff801`36a09000 PsLoadedModuleList = 0xfffff801`36e4c2d0
Debug session time: Sun Apr 21 11:28:09.669 2019 (UTC + 3:00)
System Uptime: 1 days 0:12:28.864
Break instruction exception - code 80000003 (first chance)
*****
*
*   You are seeing this message because you pressed either
*       CTRL+C (if you run console kernel debugger) or,
*       CTRL+BREAK (if you run GUI kernel debugger),
*   on your debugger machine's keyboard.
*
*
*               THIS IS NOT A BUG OR A SYSTEM CRASH
*
*
* If you did not intend to break into the debugger, press the "g" key, then
* press the "Enter" key now. This message might immediately reappear. If it
* does, press "g" and "Enter" again.
*
*****
nt!DbgBreakPointWithStatus:
fffff801`36bcd580 cc           int     3
```

Note the prompt has an index and the word *kd*. The index is the current processor that induced the break. At this point, the target VM is completely frozen. You can now debug normally, bearing in mind anytime you break somewhere, the entire machine is frozen.

Using the Network

In this section, we'll configure full kernel debugging using the network, focusing on the differences compared to the virtual COM port setup.

Configuring the Target

On the target machine, running with an elevated command window, configure network debugging using the following format with `bcdedit`:

```
bcdedit /dbgsettings net hostip:<ip> port: <port> [key: <key>]
```

The *hostip* must be the IP address of the host accessible from the target. *port* can be any available port on the host, but the documentation recommends working with port 50000 and up. The key is optional. If you don't specify it, the command generates a random key. For example:

```
bcdedit /dbgsettings net hostip:10.100.102.53 port:51111  
Key=1rhvit77hdpv7.rxgwjdhxj7v.312gs2roip4sf.3w25wrjeocobh
```

The alternative is provide your own key for simplicity, which must be in the format *a.b.c.d*. This is acceptable from a security standpoint when working with local virtual machines:

```
bcdedit /dbgsettings net hostip:10.100.102.53 port:51111 key:1.2.3.4  
Key=1.2.3.4
```

You can always display the current debug configuration with `/dbgsettings` alone:

```
bcdedit /dbgsettings  
key 1.2.3.4  
debugtype NET  
hostip 10.100.102.53  
port 51111  
dhcp Yes  
The operation completed successfully.
```

Finally, restart the target.

Configuring the Host

On the host machine, launch the debugger and select the *File / Attach the Kernel* option (or *File / Kernel Debug...* in the classic *WinDbg*). Navigate to the *NET* tab, and enter the information corresponding to your settings (figure 5-7).



Figure 5-8: Attach to kernel dialog

You may need to click the *Break* button (possibly multiple times) to establish a connection. More information and troubleshooting tips can be found at <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection>.

Kernel Driver Debugging Tutorial

Once host and target are connected, debugging can begin. We will use the *Booster* driver we developed in chapter 4 to demonstrate full kernel debugging.

Install (but don't load) the driver on the target as was done in chapter 4. Make sure you copy the driver's PDB file alongside the driver SYS file itself. This simplifies getting correct symbols for the driver.

Let's set a breakpoint in `DriverEntry`. We cannot load the driver just yet because that would cause `DriverEntry` to execute, and we'll miss the chance to set a breakpoint there. Since the driver is not loaded yet, we can use the `bu` command (unresolved breakpoint) to set a future breakpoint. Break into the target if it's currently running, and type the following command in the debugger:

```
0: kd> bu booster!driverentry
0: kd> b1
0 e Disable Clear u          0001 (0001) (booster!driverentry)
```

The breakpoint is unresolved at this point, since our module (driver) is not yet loaded. The debugger will re-evaluate the breakpoint any time a new module is loaded.

Issue the `g` command to let the target continue execution, and load the driver with `sc start booster` (assuming the driver's name is *booster*). If all goes well, the breakpoint should hit, and the source file should open automatically, showing the following output in the command window:

```
0: kd> g
Breakpoint 0 hit
Booster!DriverEntry:
fffff802`13da11c0 4889542410      mov     qword ptr [rsp+10h],rdx
```



The index on the left of the colon is the CPU index running the code when the breakpoint hit (CPU 0 in the above output).

Figure 5-9 shows a screenshot of *WinDbg Preview* source window automatically opening and the correct line marked. The *Locals* window is also shown as expected.

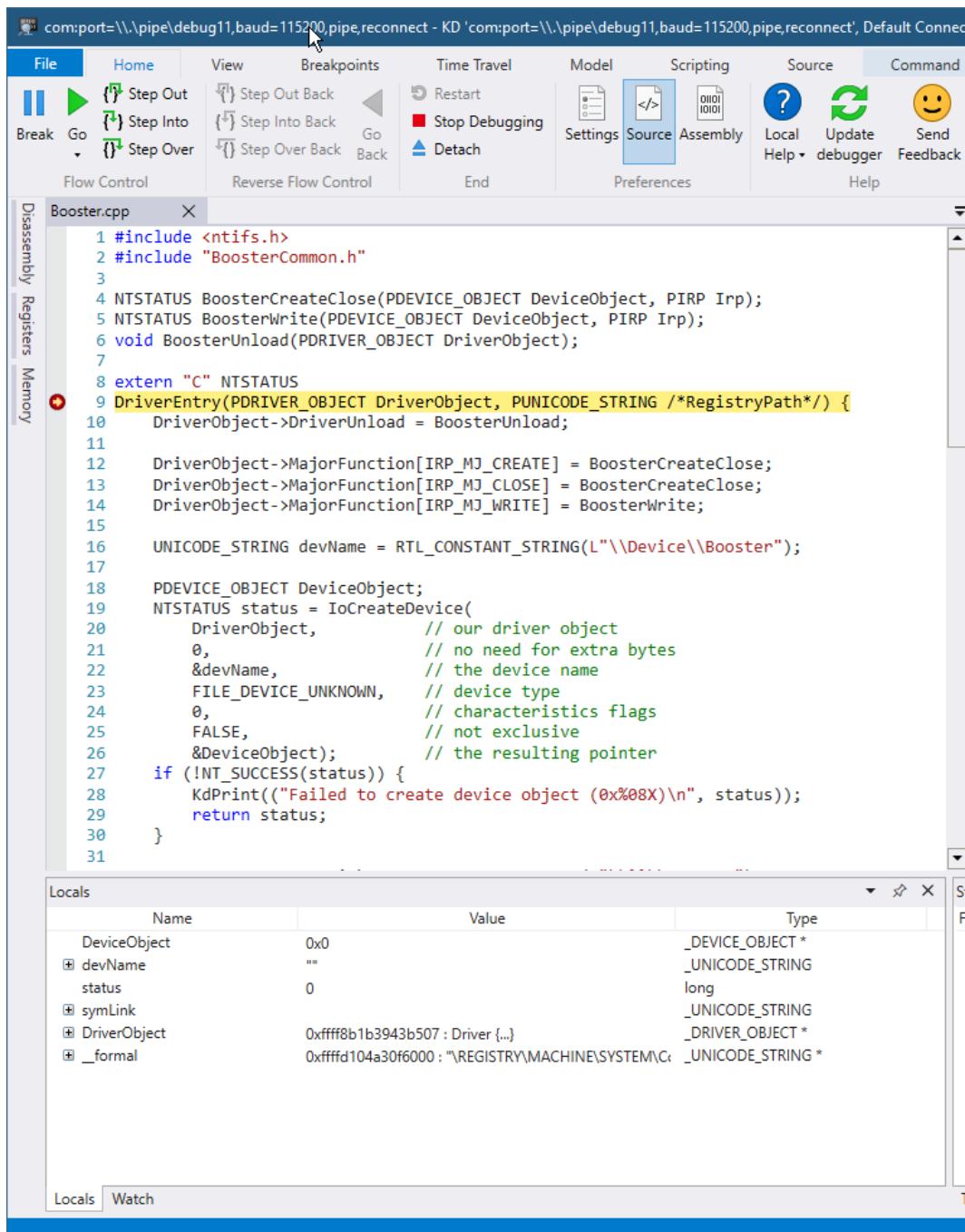


Figure 5-9: Breakpoint hit in DriverEntry

At this point, you can step over source lines, look at variables in the *Locals* window, and even add expressions to the *Watch* window. You can also change values using the *Locals* window just like you

would normally do with other debuggers.

The *Command* window is still available as always, but some operations are just easier with the GUI. Setting breakpoints, for example, can be done with the normal `bp` command, but you can simply open a source file (if it's not already open), go to the line where you want to set a breakpoint, and hit `F9` or click the appropriate button on the toolbar. Either way, the `bp` command will be executed in the *Command* window. The *Breakpoints* window can serve as a quick overview of the currently set breakpoints.

- Issue the `k` command to see how `DriverEntry` is being invoked:

```
0: kd> k
# Child-SP          RetAddr           Call Site
00 fffffbe88`b3f4f138 ffffff802`13da5020  Booster!DriverEntry [D:\Dev\windowsk\
ernelprogrammingbook2e\Chapter04\Booster\Booster.cpp @ 9]
01 fffffbe88`b3f4f140 ffffff802`081caf0  Booster!GsDriverEntry+0x20 [minkerne\
l\tools\gs_support\kmode\gs_support.c @ 128]
02 fffffbe88`b3f4f170 ffffff802`080858e2  nt!PnPCallDriverEntry+0x4c
03 fffffbe88`b3f4f1d0 ffffff802`081aeab7  nt!IopLoadDriver+0x8ba
04 fffffbe88`b3f4f380 ffffff802`07c48aaaf  nt!IopLoadUnloadDriver+0x57
05 fffffbe88`b3f4f3c0 ffffff802`07d5b615  nt!ExpWorkerThread+0x14f
06 fffffbe88`b3f4f5b0 ffffff802`07e16c24  nt!PspSystemThreadStartup+0x55
07 fffffbe88`b3f4f600 00000000`00000000  nt!KiStartSystemThread+0x34
```



If breakpoints fail to hit, it may be a symbols issue. Execute the `.reload` command and see if the issues are resolved. Setting breakpoints in user space is also possible, but first execute `.reload /user` to force the debugger to load user-mode symbols.

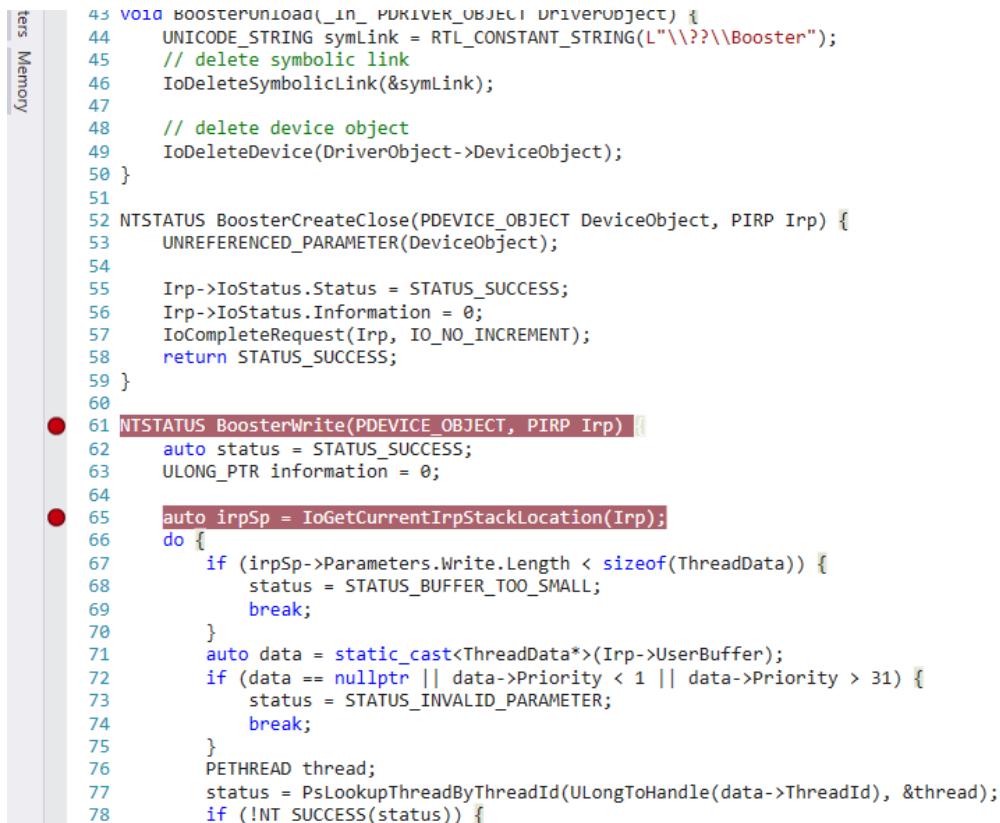
It may be the case that a breakpoint should hit only when a specific process is the one executing the code. This can be done by adding the `/p` switch to a breakpoint. In the following example, a breakpoint is set only if the process is a specific `explorer.exe`:

```
0: kd> !process 0 0 explorer.exe
PROCESS fffffd1049e118080
SessionId: 1 Cid: 1780 Peb: 0076b000 ParentCid: 16d0
DirBase: 362ea5000 ObjectTable: fffffa58d45891680 HandleCount: 3918.
Image: explorer.exe

PROCESS fffffd104a14e2080
SessionId: 1 Cid: 2548 Peb: 005c1000 ParentCid: 0314
DirBase: 140fe9000 ObjectTable: fffffa58d46a99500 HandleCount: 4524.
Image: explorer.exe
```

```
0: kd> bp /p fffffd1049e118080 booster!boosterwrite
0: kd> bl
 0 e Disable Clear fffff802`13da11c0 [D:\Dev\Chapter04\Booster\Booster.cp\
p @ 9] 0001 (0001) Booster!DriverEntry
 1 e Disable Clear fffff802`13da1090 [D:\Dev\Chapter04\Booster\Booster.cp\
p @ 61] 0001 (0001) Booster!BoosterWrite
Match process data fffffd104`9e118080
```

Let's set a normal breakpoint somewhere in the `BoosterWrite` function, by hitting *F9* on the line in source view, as shown in figure 5-10 (the earlier conditional breakpoint is shown as well).



```
43 void BoosterUnload(_In_ PDRIVER_OBJECT DriverObject) {
44     UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\Booster");
45     // delete symbolic link
46     IoDeleteSymbolicLink(&symLink);
47
48     // delete device object
49     IoDeleteDevice(DriverObject->DeviceObject);
50 }
51
52 NTSTATUS BoosterCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
53     UNREFERENCED_PARAMETER(DeviceObject);
54
55     Irp->IoStatus.Status = STATUS_SUCCESS;
56     Irp->IoStatus.Information = 0;
57     IoCompleteRequest(Irp, IO_NO_INCREMENT);
58     return STATUS_SUCCESS;
59 }
60
61 NTSTATUS BoosterWrite(PDEVICE_OBJECT, PIRP Irp) {
62     auto status = STATUS_SUCCESS;
63     ULONG_PTR information = 0;
64
65     auto irpSp = IoGetCurrentIrpStackLocation(Irp);
66     do {
67         if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
68             status = STATUS_BUFFER_TOO_SMALL;
69             break;
70         }
71         auto data = static_cast<ThreadData*>(Irp->UserBuffer);
72         if (data == nullptr || data->Priority < 1 || data->Priority > 31) {
73             status = STATUS_INVALID_PARAMETER;
74             break;
75         }
76         PETHREAD thread;
77         status = PsLookupThreadByThreadId(ULongToHandle(data->ThreadId), &thread);
78         if (!NT_SUCCESS(status)) {
```

Figure 5-10: Breakpoint hit in DriverEntry

Listing the breakpoints reflect the new breakpoint with the offset calculated by the debugger:

```
0: kd> b1
0 e Disable Clear fffff802`13da11c0 [D:\Dev\Chapter04\Booster\Booster.cpp @\
9] 0001 (0001) Booster!DriverEntry
1 e Disable Clear fffff802`13da1090 [D:\Dev\Chapter04\Booster\Booster.cpp @\
61] 0001 (0001) Booster!BoosterWrite
Match process data fffffd104`9e118080
2 e Disable Clear fffff802`13da10af [D:\Dev\Chapter04\Booster\Booster.cpp @\
65] 0001 (0001) Booster!BoosterWrite+0x1f
```

Enter the `g` command to release the target, and then run the *boost* application with some thread ID and priority:

```
c:\Test> boost 5964 30
```

The breakpoint within `BoosterWrite` should hit:

```
Breakpoint 2 hit
Booster!BoosterWrite+0x1f:
fffff802`13da10af 488b4c2468      mov     rcx,qword ptr [rsp+68h]
```

You can continue debugging normally, looking at local variables, stepping over/into functions, etc.

Finally, if you would like to disconnect from the target, enter the `.detach` command. If it does not resume the target, click the *Stop Debugging* toolbar button (you may need to click it multiple times).

Asserts and Tracing

Although using a debugger is sometimes necessary, some coding can go a long way in making a debugger less needed. In this section we'll examine asserts and powerful logging that is suitable for both debug and release builds of a driver.

Asserts

Just like in user mode, asserts can be used to verify that certain assumptions are correct. An invalid assumption means something is very wrong, so it's best to stop. The WDK header provides the `NT_ASSERT` macro for this purpose.

`NT_ASSERT` accepts something that can be converted to a Boolean value. If the result is non-zero (true), execution continues. Otherwise, the assertion has failed, and the system takes one of the following actions:

- If a kernel debugger is attached, an assertion failure breakpoint is raised, allowing debugging the assertion.
- If a kernel debugger is not attached, the system bugchecks. The resulting dump file will point to the exact line where the assertion has failed.

Here is a simple assert usage added to the `DriverEntry` function in the *Booster* driver from chapter 4:

```

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    DriverObject->DriverUnload = BoosterUnload;

    DriverObject->MajorFunction[IRP_MJ_CREATE] = BoosterCreateClose;
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = BoosterCreateClose;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = BoosterWrite;

    UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\Booster");

    PDEVICE_OBJECT DeviceObject;
    NTSTATUS status = IoCreateDevice(
        DriverObject,                      // our driver object
        0,                                // no need for extra bytes
        &devName,                          // the device name
        FILE_DEVICE_UNKNOWN,               // device type
        0,                                // characteristics flags
        FALSE,                            // not exclusive
        &DeviceObject);                  // the resulting pointer
    if (!NT_SUCCESS(status)) {
        KdPrint(("Failed to create device object (0x%08X)\n", status));
        return status;
    }

    NT_ASSERT(DeviceObject);

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\Booster");
    status = IoCreateSymbolicLink(&symLink, &devName);
    if (!NT_SUCCESS(status)) {
        KdPrint(("Failed to create symbolic link (0x%08X)\n", status));
        IoDeleteDevice(DeviceObject);
        return status;
    }

    NT_ASSERT(NT_SUCCESS(status));
    return STATUS_SUCCESS;
}

```

The first assert makes sure the device object pointer is non-NULL:

```
NT_ASSERT(DeviceObject);
```

The second makes sure the status at the end of DriverEntry is a successful one:

```
NT_ASSERT(NT_SUCCESS(status));
```

NT_ASSERT only compiles its expression in Debug builds, which makes using asserts practically free from a performance standpoint, as these will not be part of the final released driver. This also means you need to be careful that the expression inside NT_ASSERT has no side effects. For example, the following code is wrong:

```
NT_ASSERT(NT_SUCCESS(IoCreateSymbolicLink(...)));
```

This is because the call to IoCreateSymbolicLink will disappear completely in *Release* build. The correct way to assert would be something like the following:

```
status = IoCreateSymbolicLink(...);  
NT_ASSERT(NT_SUCCESS(status));
```

Asserts are useful and should be used liberally because they only have an effect in *Debug* builds.

Extended DbgPrint

We've seen usage of the DbgPrint function (and the KdPrint macro) to generate output that can be viewed with the kernel debugger or a comparable tool, such as *DebugView*. This works, and is simple to use, but has some significant downsides:

- All the output is generated - there is no easy way to filter output to show just some output (such as errors and warnings only). This is partially mitigated with the extended DbgPrintEx function described in the next paragraph.
- DbgPrint(Ex) is a relatively slow function, which is why it's mostly used with KdPrint so that the overhead is removed in *Release* builds. But output in *Release* builds could be very important. Some bugs may only happen in Release builds, where good output could be useful for diagnosing issues.
- There is no semantic meaning associated with DbgPrint - it's just text. There is no way to add values with property name or type information.
- There is no built-in way to save the output to a file rather than just see it in the debugger. If using *DebugView*, it allows saving its output to a file.



The output from DbgPrint(Ex) is limited to 512 bytes. Any remaining bytes are lost.

The DbgPrintEx function (and the associated KdPrintEx macro) were added to provide some filtering support for DbgPrint output:

```
ULONG DbgPrintEx (
    _In_ ULONG ComponentId,
    _In_ ULONG Level,
    _In_z_ _Printf_format_string_ PCSTR Format,
    ...);           // any number of args
```

A list of component IDs is present in the *<dpfilter.h>* header (common to user and kernel mode), currently containing 155 valid values (0 to 154). Most values are used by the kernel and Microsoft drivers, except for a handful that are meant to be used by third-party drivers:

- DPFLTR_IHVVIDEO_ID (78) - for video drivers.
- DPFLTR_IHVAUDIO_ID (79) - for audio drivers.
- DPFLTR_IHVNETHWORK_ID (80) - for network drivers.
- DPFLTR_IHVSTREAMING_ID (81) - for streaming drivers.
- DPFLTR_IHVBUS_ID (82) - for bus drivers.
- DPFLTR_IHVDRIVER_ID (77) - for all other drivers.
- DPFLTR_DEFAULT_ID (101) - used with DbgPrint or if an illegal component number is used.

For most drivers, the DPFLTR_IHVDRIVER_ID component ID should be used.

The Level parameter indicates the severity of the message (error, warning, information, etc.), but can technically mean anything you want. The interpretation of this value depends on whether the value is between 0 and 31, or greater than 31:

- 0 to 31 - the level is a single bit formed by the expression `1 << Level`. For example, if Level is 5, then the value is 32.
- Anything greater than 31 - the value is used as is.

<dpfilter.h> defines a few constants that can be used as is for Level:

```
#define DPFLTR_ERROR_LEVEL    0
#define DPFLTR_WARNING_LEVEL   1
#define DPFLTR_TRACE_LEVEL     2
#define DPFLTR_INFO_LEVEL      3
```

You can define more (or different) values as needed. The final result of whether the output will make its way to its destination depends on the component ID, the bit mask formed by the Level argument, and on a global mask read from the *Debug Print Filter* Registry key at system startup. Since the *Debug Print Filter* key does not exist by default, there is a default value for all component IDs, which is zero. This means that actual level value is 1 (`1 << 0`). The output will go through if either of the following conditions is true (value is the value specified by the Level argument to DbgPrintEx):

- If `value & (Debug print Filter value for that component)` is non-zero, the output goes through. With the default, it's `(value & 1) != 0`.

- If the result of the value ANDed with the Level of the ComponentId is non-zero, the output goes through.

If neither is true, the output is dropped.

Setting the component ID level can be done in one of three ways:

- Using the *Debug Print Filter* key under *HKLM\System\CCS\Control\Session Manager*. DWORD values can be specified where their name is the macro name of a component ID without the prefix or suffix. For example, for *DPFLTR_IHVVIDEO_ID*, you would set the name to “*IHVVIDEO*”.
- If a kernel debugger is connected, the level of a component can be changed during debugging. For example, the following command changes the level of *DPFLTR_IHVVIDEO_ID* to 0x1ff:

```
ed Kd_IHVVIDEO_Mask 0x1ff
```



The *Debug Print Filter* value can also be changed with the kernel debugger by using the global kernel variable *Kd_WIN2000_Mask*.

- The last option is to make the change through the *NtSetDebugFilterState* native API. It's undocumented, but it may be useful in practice. The *Dbgkflt* tool, available in the *Tools* folder in the book's samples repository, makes use of this API (and its query counterpart, *NtQueryDebugFilterState*), so that changes can be made even if a kernel debugger is not attached.

If *NtSetDebugFilterState* is called from user mode, the caller must have the *Debug* privilege in its token. Since administrators have this privilege by default (but not non-admin users), you must run *dbgkflt* from an elevated command window for the change to succeed.



The kernel-mode APIs provided by the *<wdm.h>* are *DbgQueryDebugFilterState* and *DbgSetDebugFilterState*. These are still undocumented, but at least their declaration is available. They use the same parameters and return type as their native invokers. This means you can call these APIs from the driver itself if desired (perhaps based on configuration read from the Registry).

Using *Dbgkflt*

Running *Dbgkflt* with no arguments shows its usage.

To query the effective level of a given component, add the component name (without the prefix or suffix). For example:

```
dbgkflt default
```

This returns the effective bits for the DPFLTR_DEFAULT_ID component. To change the value to something else, specify the value you want. It's always ORed with 0x80000000 so that the bits you specify are directly used, rather than interpreting numbers lower than 32 as $(1 \ll \text{number})$. For example, the following sets the first 4 bits for the DEFAULT component:

```
dbgkflt default 0xf
```

DbgPrint is just a shortcut that calls *DbgPrintEx* with the DPFLTR_DEFAULT_ID component like so (this is conceptual and will not compile):

```
ULONG DbgPrint (PCSTR Format, arguments) {
    return DbgPrintEx(DPFLTR_DEFAULT_ID, DPFLTR_INFO_LEVEL, Format, arguments);
}
```

This explains why the DWORD named DEFAULT with a value of 8 ($1 \ll \text{DPFLTR_INFO_LEVEL}$) is the value to write in the Registry to get *DbgPrint* output to go through.

Given the above details, a driver can use *DbgPrintEx* (or the *KdPrintEx* macro) to specify different levels so that output can be filtered as needed. Each call, however, may be somewhat verbose. For example:

```
DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_INFO_LEVEL,
    "Booster: DriverEntry called. Registry Path: %wZ\n", RegistryPath);
```

Obviously, we might prefer a simpler function that always uses DPFLTR_IHVDRIVER_ID (the one that should be used for generic third-party drivers), like so:

```
Log(DPFLTR_INFO_LEVEL,
    "Booster: DriverEntry called. Registry Path: %wZ\n", RegistryPath);
```

We can go even further by defining specific functions that use a log level implicitly:

```
LogInfo("Booster: DriverEntry called. Registry Path: %wZ\n", RegistryPath);
```

Here is an example where we define several bits to be used by creating an enumeration (there is no necessity to use the defined ones):

```
enum class LogLevel {
    Error = 0,
    Warning,
    Information,
    Debug,
    Verbose
};
```

Each value is associated with a small number (below 32), so that the values are interpreted as powers of two by `DbgPrintEx`. Now we can define functions like the following:

```
ULONG Log(LogLevel level, PCSTR format, ...);
ULONG LogError(PCSTR format, ...);
ULONG LogWarning(PCSTR format, ...);
ULONG LogInfo(PCSTR format, ...);
ULONG LogDebug(PCSTR format, ...);
```

and so on. `Log` is the most generic function, while the others use a predefined log level. Here is the implementation of the first two functions:

```
#include <stdarg.h>

ULONG Log(LogLevel level, PCSTR format, ...) {
    va_list list;
    va_start(list, format);
    return vDbgPrintEx(DPFLTR_IHVDRIVER_ID,
        static_cast<ULONG>(level), format, list);
}

ULONG LogError(PCSTR format, ...) {
    va_list list;
    va_start(list, format);
    return vDbgPrintEx(DPFLTR_IHVDRIVER_ID,
        static_cast<ULONG>(LogLevel::Error), format, list);
}
```



The use of `static_cast` in the above code is required in C++, as scoped enums don't automatically convert to integers. You can use a C-style cast instead, if you prefer. If you're using pure C, change the scoped enum to a standard enum (remove the `class` keyword).



The return value from the various `DbgPrint` variants is typed as a `ULONG`, but is in fact a standard `NTSTATUS`.

The implementation uses the classic C variable arguments ellipsis (...) and implements these as you would in standard C. The implementation calls vDbgPrintEx that accepts a va_list, which is necessary for this to work correctly.



It's possible to create something more elaborate using the C++ variadic template feature.
This is left as an exercise to the interested (and enthusiastic) reader.

The above code can be found in the *Booster2* project, part of the samples for this chapter. As part of that project, here are a few examples where these functions are used:

```
// in DriverEntry
Log(LogLevel::Information, "Booster2: DriverEntry called. Registry Path: %wZ\n\" \
,
    RegistryPath);

// unload routine
LogInfo("Booster2: unload called\n");

// when an error is encountered creating a device object
LogError("Failed to create device object (0x%08X)\n", status);

// error locating thread ID
LogError("Failed to locate thread %u (0x%08X)\n",
    data->ThreadId, status);

// success in changing thread priority
LogInfo("Priority for thread %u changed from %d to %d\n",
    data->ThreadId, oldPriority, data->Priority);
```

Other Debugging Functions

The previous section used vDbgPrintEx, defined like so:

```
ULONG vDbgPrintEx(
    _In_ ULONG ComponentId,
    _In_ ULONG Level,
    _In_z_ PCCH Format,
    _In_ va_list arglist);
```

It's identical to DbgPrintEx, except its last argument is an already constructed va_list. A wrapper macro exists as well - vKdPrintEx (compiled in *Debug* builds only).

Lastly, there is yet another extended function for printing - vDbgPrintExWithPrefix:

```
ULONG vDbgPrintExWithPrefix (
    _In_z_ PCCH Prefix,
    _In_ ULONG ComponentId,
    _In_ ULONG Level,
    _In_z_ PCCH Format,
    _In_ va_list arglist);
```

It adds a prefix (first parameter) to the output. This is useful to distinguish our driver from other drivers using the same functions. It also allows easy filtering in tools such as *DebugView*. For example, this code snippet shown earlier uses an explicit prefix:

```
LogInfo("Booster2: unload called\n");
```

We can define one as a macro, and use it as the first word in any output like so:

```
#define DRIVER_PREFIX "Booster2: "
LogInfo(DRIVER_PREFIX "unload called\n");
```

This works, but it could be nicer by adding the prefix in every call automatically, by calling `vDbgPrintExWithPrefix` instead of `vDbgPrintEx` in the Log implementations. For example:

```
ULONG Log(LogLevel level, PCSTR format, ...) {
    va_list list;
    va_start(list, format);
    return vDbgPrintExWithPrefix("Booster2", DPFLTR_IHVDRIVER_ID,
        static_cast<ULONG>(level), format, list);
}
```



Complete the implementation of the Log functions variants.

Trace Logging

Using `DbgPrint` and its variants is convenient enough, but as discussed earlier has some drawbacks. Trace logging is a powerful alternative (or complementary) that uses *Event Tracing for Windows* (ETW) for logging purposes, that can be captured live or to a log file. ETW has the additional benefits of being performant (can be used to log thousands of events per second without any noticeable delay), and has semantic information not available with the simple strings generated by the `DbgPrint` functions.



Trace logging can be used in exactly the same way in user mode as well.



ETW is beyond the scope of this book. You can find more information in the official documentation or in my book “Windows 10 System Programming, Part 2”.

To get started with trace logging, an ETW provider has to be defined. Contrary to “classic” ETW, no provider registration is necessary, as trace logging ensures the even metadata is part of the logged information, and as such is self-contained.

A provider must have a unique GUID. You can generate one with the *Create GUID* tool available with Visual Studio (*Tools* menu). Figure 5-11 shows a screenshot of the tool with the second radio button selected, as it’s the closest to the format we need. Click the *Copy* button to copy that text to the clipboard.

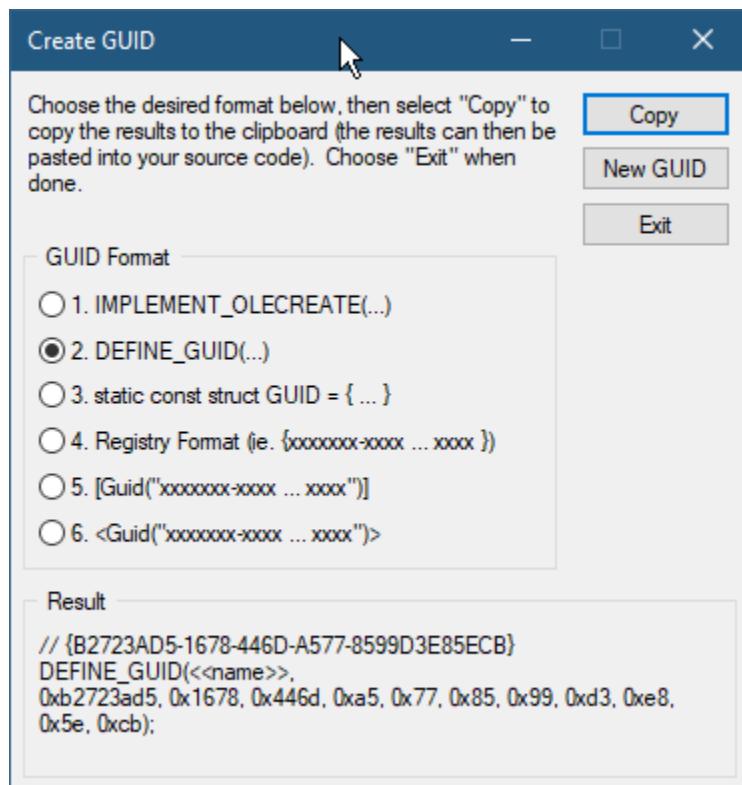


Figure 5-11: The *Create GUID* tool

Paste the text to the main source file of the driver and change the pasted macro to TRACELOGGING_-
DEFINE_PROVIDER to look like this:

```
// {B2723AD5-1678-446D-A577-8599D3E85ECB}
TRACELOGGING_DEFINE_PROVIDER(g_Provider, "Booster", \
    (0xb2723ad5, 0x1678, 0x446d, 0xa5, 0x77, 0x85, 0x99, 0xd3, 0xe8, 0x5e, 0xcb\
));
```

`g_Provider` is a global variable created to represent the ETW provider, where “Booster” is set as its friendly name.

You will need to add the following `#includes` (these are common with user-mode):

```
#include <TraceLoggingProvider.h>
#include <evntrace.h>
```

In `DriverEntry`, call `TraceLoggingRegister` to register the provider:

```
TraceLoggingRegister(g_Provider);
```

Similarly, the provider should be deregistered in the unload routine like so:

```
TraceLoggingUnregister(g_Provider);
```

The logging is done with the `TraceLoggingWrite` macro that is provided a variable number of arguments using another set of macros that provide convenient usage for typed properties. Here is an example of a logging call in `DriverEntry`:

```
TraceLoggingWrite(g_Provider, "DriverEntry started", // provider, event name
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION), // log level
    TraceLoggingValue("Booster Driver", "DriverName"), // value, name
    TraceLoggingUnicodeString(RegistryPath, "RegistryPath")); // value, name
```

The above call means the following:

- Use the provider described by `g_Provider`.
- The event name is “DriverEntry started”.
- The logging level is *Information* (several levels are defined).
- A property named “`DriverName`” has the value “`Boster Driver`”.
- A property named “`RegistryPath`” has the value of the `RegistryPath` variable.

Notice the usage of the `TraceLoggingValue` macro - it’s the most generic and uses the type inferred by the first argument (the value). Many other type-safe macros exist, such as the `TraceLoggingUnicodeString` macro above that ensures its first argument is indeed a `UNICODE_STRING`.

Here is another example - if symbolic link creation fails:

```
TraceLoggingWrite(g_Provider, "Error",
    TraceLoggingLevel(TRACE_LEVEL_ERROR),
    TraceLoggingValue("Symbolic link creation failed", "Message"),
    TraceLoggingNTStatus(status, "Status", "Returned status"));
```

You can use any “properties” you want. Try to provide the most important details for the event.

Here are a couple of more examples, taken from the *Booster* project part of the samples for this chapter:

```
// Create/Close dispatch IRP
TraceLoggingWrite(g_Provider, "Create/Close",
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
    TraceLoggingValue(
        IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_CREATE ?
        "Create" : "Close", "Operation"));

// success in changing priority
TraceLoggingWrite(g_Provider, "Boosting",
    TraceLoggingLevel(TRACE_LEVEL_INFORMATION),
    TraceLoggingUInt32(data->ThreadId, "ThreadId"),
    TraceLoggingInt32(oldPriority, "OldPriority"),
    TraceLoggingInt32(data->Priority, "NewPriority"));
```

Viewing ETW Traces

Where do all the above traces go to? Normally, they are just dropped. Someone has to configure listening to the provider and log the events to a real-time session or a file. The WDK provides a tool called *TraceView* that can be used for just that purpose.

You can open a Developer’s Command window and run *TraceView.exe* directly. If you can’t locate it, it’s installed by default in a directory such as *C:\Program Files (x86)\Windows Kits\10\bin\10.0.22000.0\x64*.

You can copy the executable to the target machine where the driver is supposed to run. When you run *TraceView.exe*, an empty window is shown (figure 5-12).

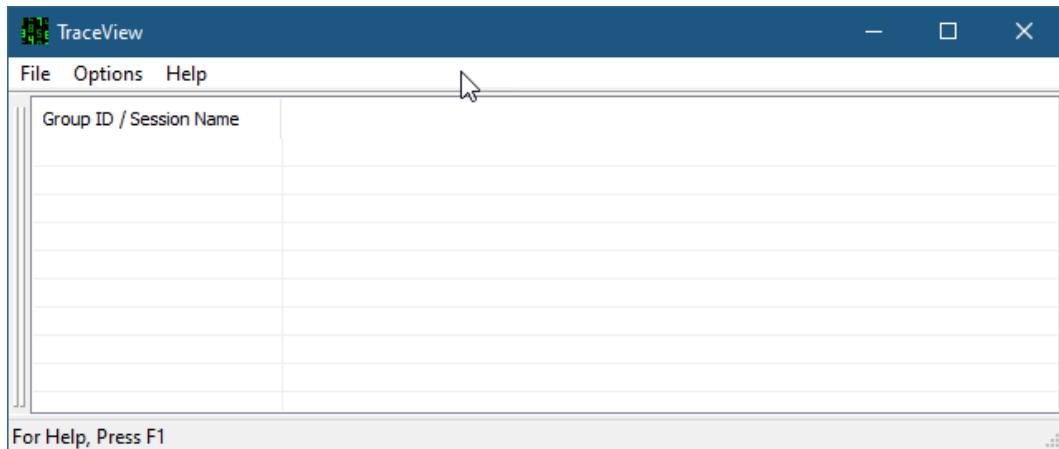


Figure 5-12: The *TraceView.exe* tool

Select the *File / Create New log Session* menu to create a new session. This opens up the dialog shown in figure 5-13.

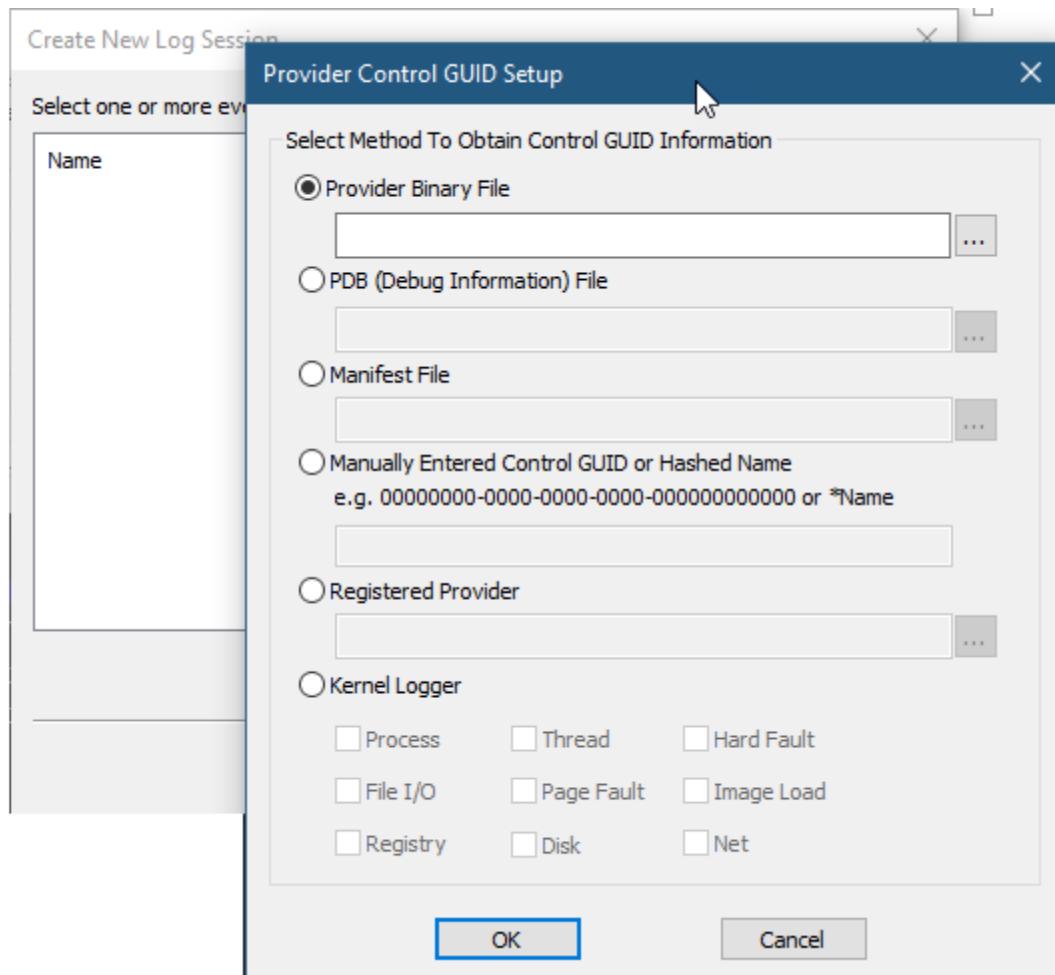


Figure 5-13: New session dialog with a new provider

TraceView provides several methods of locating providers. We can add multiple providers to the same session to get information from other components in the system. For now, we'll add our provider by using the *Manually Entered Control GUID* option, and type in our GUID (figure 5-14):

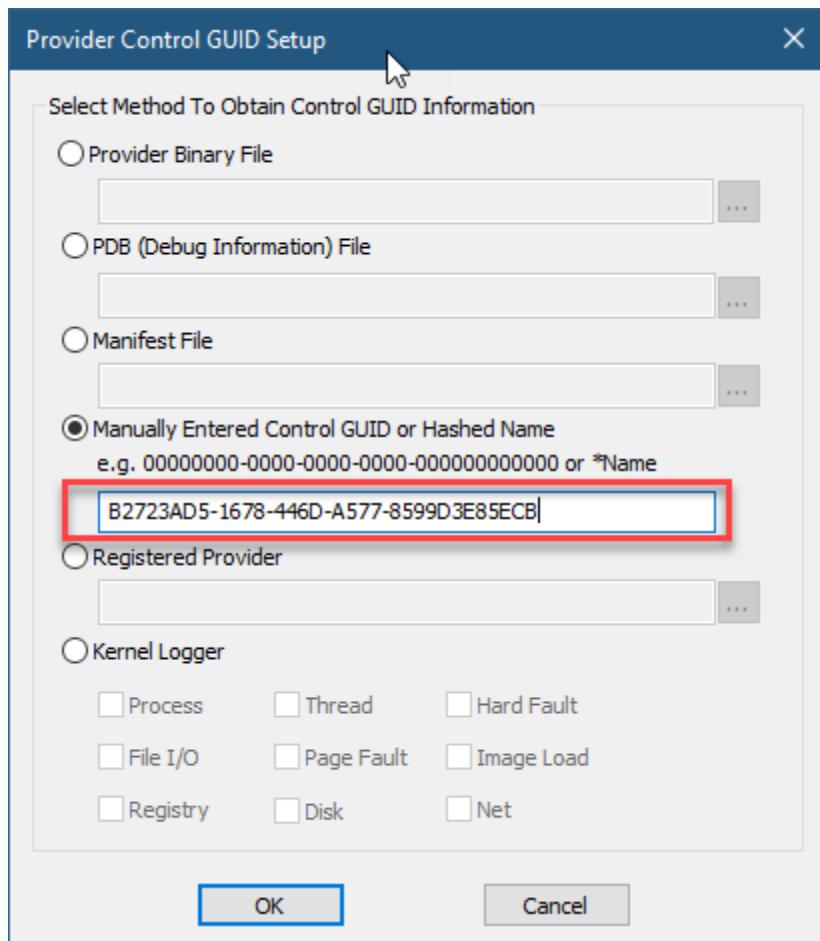


Figure 5-14: Adding a provider GUID manually

Click *OK*. A dialog will pop up asking the source for decoding information. Use the default *Auto* option, as trace logging does not require any outside source. You'll see the single provider in the *Create New Log Session* dialog. Click the *Next* button. The last step of the wizard allows you to select where the output should go to: a real-time session (shown with *TraceView*), a file, or both (figure 5-15).

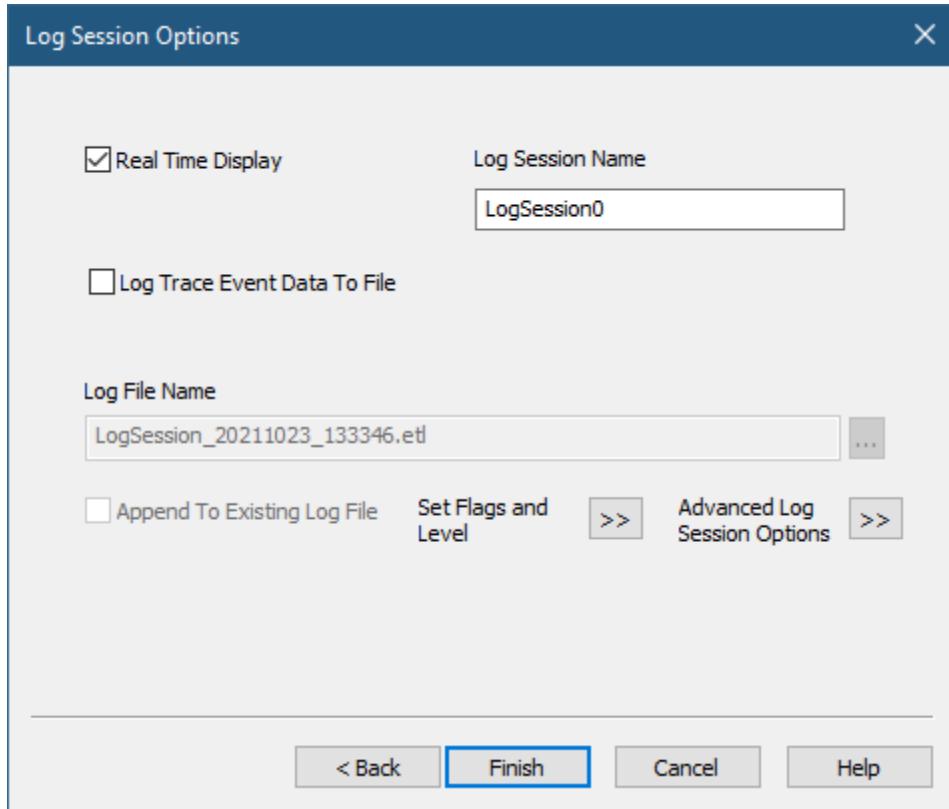


Figure 5-15: Output selection for a session

Click *Finish*. Now you can load/use the driver normally. You should see the output generated in the main *TraceView* window (figure 5-16).

TraceView										
File		Options		Help						
Group ID / Session Name	State	Event Count	Events Lost	Buffers Read	Flags	Max Buf	Min Buf	Level	KD Filter	Ignore TraceView
0 Booster	RUNNING	8	0	6	0xFFFFFFFFFFFFFF	21	4	5	FALSE	FALSE
M	Msg #	Name	Process ID	Thread ID	CPU #	Sequence #	System Time	Message		
00000001	Booster	4	272	1	0	0	10/16/2021-12:48:53:830	{"Driver Name": "Booster Driver", "Registry Path": "\REGISTRY\{MACHINE\}\SYSTEM\ControlSet001\{Services\}booster", "Meta": {"provider": "Booster", "event": "DriverOpen"}, "Event": "Create", "Time": "2021-10-16T12:48:49.746", "CPU": 2, "PID": 4860, "BD": 7380, "Channel": 11, "Level": 4}		
00000002	Booster	4860	7380	2	0	0	10/16/2021-12:49:47:246	{"Operation": "Create", "meta": {"provider": "Booster", "event": "Create/Close", "time": "2021-10-16T12:49:47.246", "cpu": 2, "pid": 4860, "bd": 7380, "channel": 11, "level": 4}}		
00000003	Booster	4860	7380	1	0	0	10/16/2021-12:49:47:247	{"Operation": "Close", "meta": {"provider": "Booster", "event": "Create/Close", "time": "2021-10-16T12:49:47.247", "cpu": 1, "pid": 4860, "bd": 7380, "channel": 11, "level": 4}}		
00000004	Booster	8508	4620	0	0	0	10/16/2021-12:50:21:51	{"Operation": "Create", "meta": {"provider": "Booster", "event": "Create/Close", "time": "2021-10-16T12:50:21.051", "cpu": 0, "pid": 8508, "bd": 4620, "channel": 11, "level": 4}}		
00000005	Booster	8508	4620	1	0	0	10/16/2021-12:50:21:52	{"Operation": "Close", "meta": {"provider": "Booster", "event": "Create/Close", "time": "2021-10-16T12:50:21.052", "cpu": 1, "pid": 8508, "bd": 4620, "channel": 11, "level": 4}}		
00000006	Booster	6640	9260	2	0	0	10/16/2021-12:51:00:345	{"Operation": "Create", "meta": {"provider": "Booster", "event": "Create/Close", "time": "2021-10-16T12:51:00.345", "cpu": 2, "pid": 6640, "bd": 9260, "channel": 11, "level": 4}}		
00000007	Booster	6640	9260	2	0	0	10/16/2021-12:51:00:345	{"ThreadID": 6400, "OldPriority": 9, "NewPriority": 22, "Meta": {"provider": "Booster", "event": "Boosting", "time": "2021-10-16T12:51:00.345", "cpu": 2, "pid": 6640, "bd": 9260, "channel": 11, "level": 4}}		
00000008	Booster	6640	9260	2	0	0	10/16/2021-12:51:00:345	{"Operation": "Close", "meta": {"provider": "Booster", "event": "Create/Close", "time": "2021-10-16T12:51:00.345", "cpu": 2, "pid": 6640, "bd": 9260, "channel": 11, "level": 4}}		

Figure 5-16: ETW real-time session in action

You can see the various properties shown in the *Message* column. When logging to a file, you can open the file later with *TraceView* and see what was logged.

There are other ways to use *TraceView*, and other tools to record and view ETW information. You could also write your own tools to parse the ETW log, as the events have semantic information and

so can easily be analyzed.

Summary

In this chapter, we looked at the basics of debugging with *WinDbg*, as well as tracing activities within the driver. Debugging is an essential skill to develop, as software of all kinds, including kernel drivers, may have bugs.

In the next chapter, we'll delve into some kernel mechanisms we need to get acquainted with, as these come up frequently while developing and debugging drivers.

Chapter 6: Kernel Mechanisms

This chapter discusses various mechanisms the Windows kernel provides. Some of these are directly useful for driver writers. Others are mechanisms that a driver developer needs to understand as it helps with debugging and general understanding of activities in the system.

In this chapter:

- Interrupt Request Level
 - Deferred Procedure Calls
 - Asynchronous Procedure Calls
 - Structured Exception Handling
 - System Crash
 - Thread Synchronization
 - High IRQL Synchronization
 - Work Items
-

Interrupt Request Level (IRQL)

In chapter 1, we discussed threads and thread priorities. These priorities are taken into consideration when more threads want to execute than there are available processors. At the same time, hardware devices need to notify the system that something requires attention. A simple example is an I/O operation that is carried out by a disk drive. Once the operation completes, the disk drive notifies completion by requesting an interrupt. This interrupt is connected to an Interrupt Controller hardware that then sends the request to a processor for handling. The next question is, which thread should execute the associated Interrupt Service Routine (ISR)?

Every hardware interrupt is associated with a priority, called *Interrupt Request Level* (IRQL) (not to be confused with an interrupt physical line known as IRQ), determined by the HAL. Each processor's context has its own IRQL, just like any register. IRQLs may or may not be implemented by the CPU hardware, but this is essentially unimportant. IRQL should be treated just like any other CPU register.

The basic rule is that a processor executes the code with the highest IRQL. For example, if a CPU's IRQL is zero at some point, and an interrupt with an associated IRQL of 5 comes in, it will save its state (context) in the current thread's kernel stack, raise its IRQL to 5 and then execute the ISR associated with the interrupt. Once the ISR completes, the IRQL will drop to its previous level, resuming the previously executed code as though the interrupt never happened. While the ISR is executing, other

interrupts coming in with an IRQL of 5 or less cannot interrupt this processor. If, on the other hand, the IRQL of the new interrupt is above 5, the CPU will save its state again, raise IRQL to the new level, execute the second ISR associated with the second interrupt and when completed, will drop back to IRQL 5, restore its state and continue executing the original ISR. Essentially, raising IRQL blocks code with equal or lower IRQL temporarily. The basic sequence of events when an interrupt occurs is depicted in figure 6-1. Figure 6-2 shows what interrupt nesting looks like.

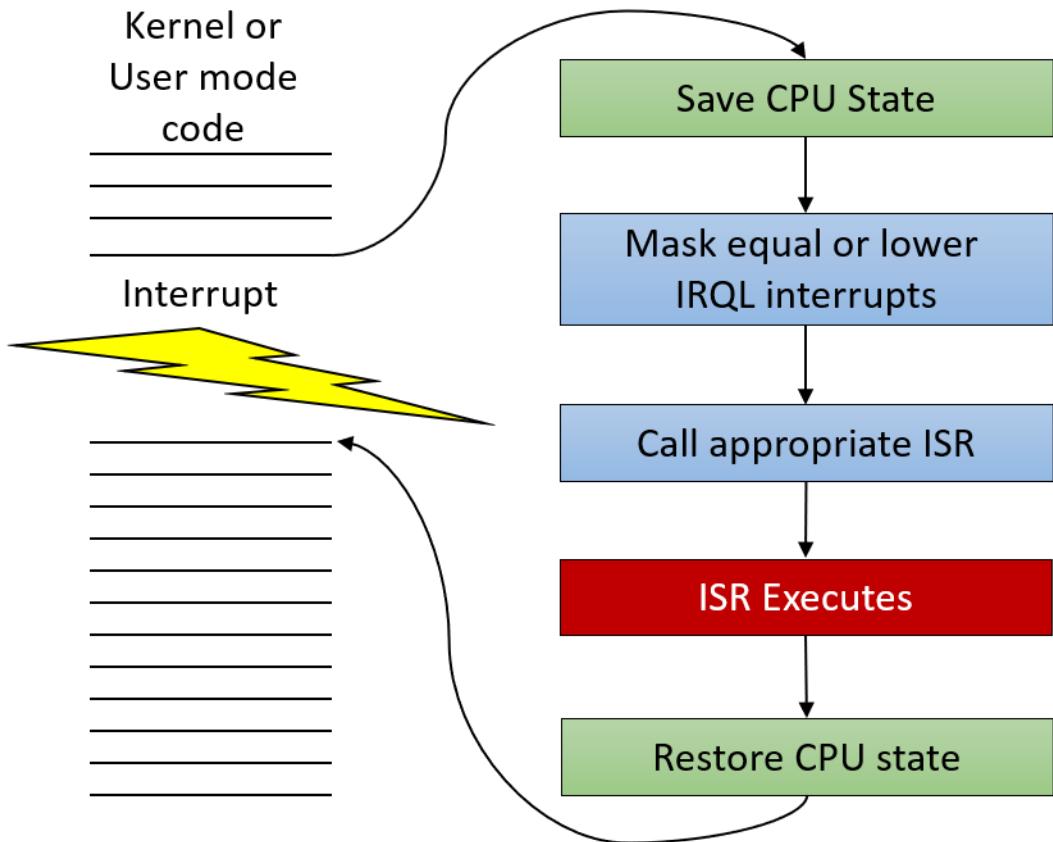


Figure 6-1: Basic interrupt dispatching

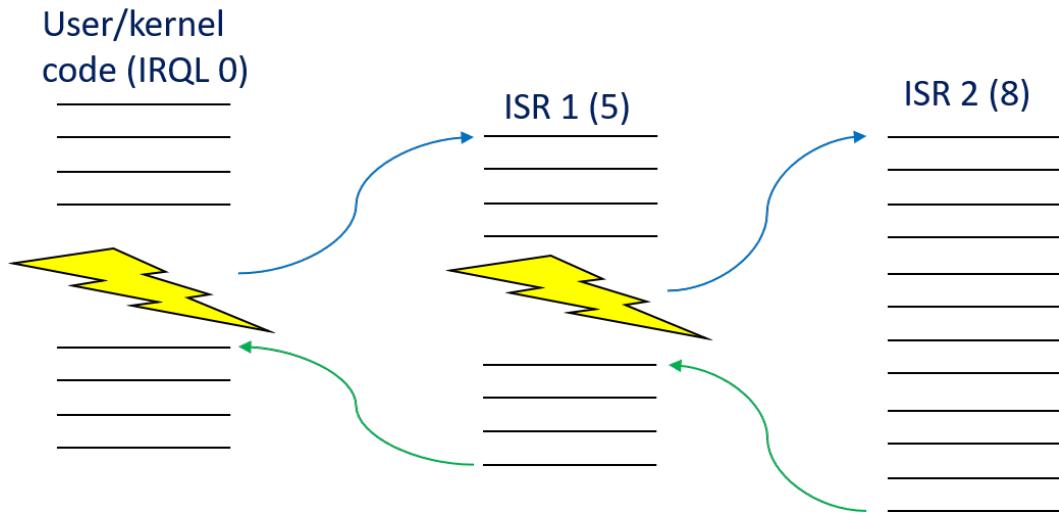


Figure 6-2: Nested interrupts

An important fact for the depicted scenarios in figures 6-1 and 6-2 is that execution of all ISRs is done by the same thread - which got interrupted in the first place. Windows does not have a special thread to handle interrupts; they are handled by whatever thread was running at that time on the interrupted processor. As we'll soon discover, context switching is not possible when the IRQL of the processor is 2 or higher, so there is no way another thread can sneak in while these ISRs execute.

The interrupted thread does not get its quantum reduced because of these “interruptions”. It's not its fault, so to speak.

When user-mode code is executing, the IRQL is always zero. This is one reason why the term IRQL is not mentioned in any user-mode documentation - it's always zero and cannot be changed. Most kernel-mode code runs with IRQL zero as well. It's possible, however, in kernel mode, to raise the IRQL on the current processor.

The important IRQLs are described below:

- `PASSIVE_LEVEL` in WDK (0) - this is the “normal” IRQL for a CPU. User-mode code always runs at this level. Thread scheduling works normally, as described in chapter 1.
- `APC_LEVEL` (1) - used for special kernel APCs (*Asynchronous Procedure Calls* will be discussed later in this chapter). Thread scheduling works normally.
- `DISPATCH_LEVEL` (2) - this is where things change radically. The scheduler cannot wake up on this CPU. Paged memory access is not allowed - such access causes a system crash. Since the scheduler cannot interfere, waiting on kernel objects is not allowed (causes a system crash if used).
- Device IRQL - a range of levels used for hardware interrupts (3 to 11 on x64/ARM/ARM64, 3 to 26 on x86). All rules from IRQL 2 apply here as well.

- Highest level (`HIGH_LEVEL`) - this is the highest IRQL, masking all interrupts. Used by some APIs dealing with linked list manipulation. The actual values are 15 (x64/ARM/ARM64) and 31 (x86).

When a processor's IRQL is raised to 2 or higher (for whatever reason), certain restrictions apply on the executing code:

- Accessing memory not in physical memory is fatal and causes a system crash. This means accessing data from non-paged pool is always safe, whereas accessing data from paged pool or from user-supplied buffers is not safe and should be avoided.
- Waiting on any kernel object (e.g. mutex or event) causes a system crash, unless the wait timeout is zero, which is still allowed. (we'll discuss dispatcher object and waiting later in this chapter in the *Thread Synchronization* section.)

These restrictions are due to the fact that the scheduler “runs” at IRQL 2; so if a processor's IRQL is already 2 or higher, the scheduler cannot wake up on that processor, so context switches (replacing the running thread with another on this CPU) cannot occur. Only higher level interrupts can temporarily divert code into an associated ISR, but it's still the same thread - no context switch can occur; the thread's context is saved, the ISR executes and the thread's state resumes.



The current IRQL of a processor can be viewed while debugging with the `!irq1` command.
An optional CPU number can be specified, which shows the IRQL of that CPU.

You can view the registered interrupts on a system using the `!idt` debugger command.

Raising and Lowering IRQL

As previously discussed, in user mode the concept of IRQL is not mentioned and there is no way to change it. In kernel mode, the IRQL can be raised with the `KeRaiseIrql` function and lowered back with `KeLowerIrql`. Here is a code snippet that raises the IRQL to `DISPATCH_LEVEL` (2), and then lowers it back after executing some instructions at this IRQL.

```
// assuming current IRQL <= DISPATCH_LEVEL

KIRQL oldIrql;          // typedefed as UCHAR
KeRaiseIrql(DISPATCH_LEVEL, &oldIrql);

NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);

// do work at IRQL DISPATCH_LEVEL

KeLowerIrql(oldIrql);
```



If you raise IRQL, make sure you lower it in the same function. It's too dangerous to return from a function with a higher IRQL than it was entered. Also, make sure KeRaiseIrql actually raises the IRQL and KeLowerIrql actually lowers it; otherwise, a system crash will follow.

Thread Priorities vs. IRQLs

IRQL is an attribute of a processor. Priority is an attribute of a thread.

Thread priorities only have meaning at $\text{IRQL} < 2$. Once an executing thread raised IRQL to 2 or higher, its priority does not mean anything anymore - it has theoretically an infinite quantum - it will continue execution until it lowers the IRQL to below 2.

Naturally, spending a lot of time at $\text{IRQL} \geq 2$ is not a good thing; user mode code is not running for sure. This is just one reason there are severe restrictions on what executing code can do at these levels.

Task Manager shows the amount of CPU time spent in IRQL 2 or higher using a pseudo-process called *System Interrupts*; *Process Explorer* calls it *Interrupts*. Figure 6-3 shows a screenshot from *Task Manager* and figure 6-4 shows the same information in *Process Explorer*.

Processes	Performance	App history	Startup	Users	Details	Services					
Name	PID	Status	User name	Ses...	CPU	Memory (a...)	Commit size	Base priority	H...	Th...	Description
System interrupts	-	Running	SYSTEM	0	01	0 K	0 K	N/A	-	-	Deferred procedure calls and interrupt service routines
System Idle Process	0	Running	SYSTEM	0	86	8 K	60 K	N/A	-	12	Percentage of time the processor is idle
System	4	Running	SYSTEM	0	00	20 K	204 K	N/A	9,...	382	NT Kernel & System
Secure System	88	Running	SYSTEM	n	n	80,272 K	104 K	N/A	-	-	NT Kernel & System

Figure 6-3: IRQL 2+ CPU time in Task Manager

Process	PID	CPU	Private Bytes	Working Set	Description	User Name
Interrupts	n/a	1.06	0 K	0 K	Hardware Interrupts and DPCs	
System Idle Process	0	83.28	60 K	8 K		NT AUTHORITY\SYSTEM
System	4	0.88	204 K	3,932 K		NT AUTHORITY\SYSTEM
Secure System	88	Suspended	184 K	80,372 K		NT AUTHORITY\SYSTEM

Figure 6-4: IRQL 2+ CPU time in Process Explorer

Deferred Procedure Calls

Figure 6-5 shows a typical sequence of events when a client invokes some I/O operation. In this figure, a user mode thread opens a handle to a file, and issues a read operation using the `ReadFile` function. Since the thread can make an asynchronous call, it regains control almost immediately and can do other work. The driver receiving this request, calls the file system driver (e.g. NTFS), which may call other drivers below it, until the request reaches the disk driver, which initiates the operation on the actual disk hardware. At that point, no code needs to execute, since the hardware “does its thing”.

When the hardware is done with the read operation, it issues an interrupt. This causes the Interrupt Service Routine associated with the interrupt to execute at Device IRQL (note that the thread handling the request is arbitrary, since the interrupt arrives asynchronously). A typical ISR accesses the device’s hardware to get the result of the operation. Its final act should be to complete the initial request.

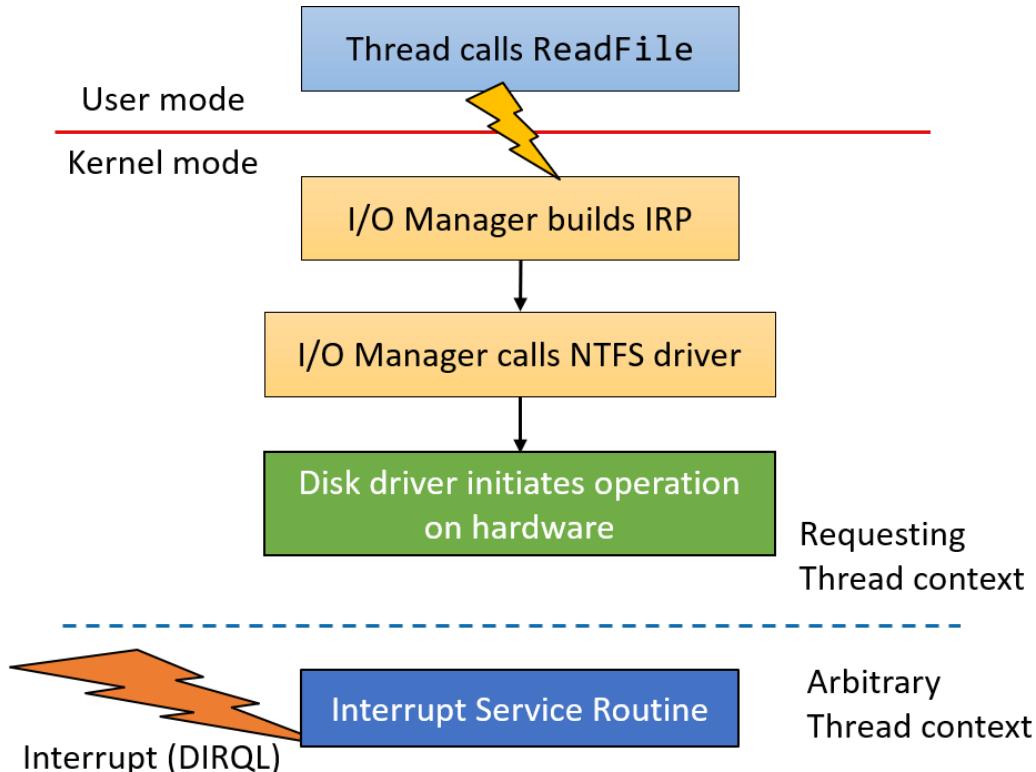


Figure 6-5: Typical I/O request processing (part 1)

As we've seen in chapter 4, completing a request is done by calling `IoCompleteRequest`. The problem is that the documentation states this function can only be called at `IRQL <= DISPATCH_LEVEL` (2). This means the ISR cannot call `IoCompleteRequest` or it will crash the system. So what is the ISR to do?



You may wonder why there is such a restriction. One of the reasons has to do with the work done by `IoCompleteRequest`. We'll discuss this in more detail in the next chapter, but the bottom line is that this function is relatively expensive. If the call would have been allowed, that would mean the ISR will take substantially longer to execute, and since it executes in a high IRQL, it will mask off other interrupts for a longer period of time.

The mechanism that allows the ISR to call `IoCompleteRequest` (and other functions with similar limitations) as soon as possible is using a *Deferred Procedure Call* (DPC). A DPC is an object encapsulating a function that is to be called at IRQL `DISPATCH_LEVEL`. At this IRQL, calling `IoCompleteRequest` is permitted.



You may wonder why the ISR does not simply lower the current IRQL to `DISPATCH_LEVEL`, call `IoCompleteRequest`, and then raise the IRQL back to its original value. This can cause a deadlock. We'll discuss the reason for that later in this chapter in the section *Spin Locks*.

The driver which registered the ISR prepares a DPC in advance, by allocating a KDPC structure from non-paged pool and initializing it with a callback function using `KeInitializeDpc`. Then, when the ISR is called, just before exiting the function, the ISR requests the DPC to execute as soon as possible by queuing it using `KeInsertQueueDpc`. When the DPC function executes, it calls `IoCompleteRequest`. So the DPC serves as a compromise - it's running at IRQL DISPATCH_LEVEL, meaning no scheduling can occur, no paged memory access, etc. but it's not high enough to prevent hardware interrupts from coming in and being served on the same processor.

Every processor on the system has its own queue of DPCs. By default, `KeInsertQueueDpc` queues the DPC to the current processor's DPC queue. When the ISR returns, before the IRQL can drop back to zero, a check is made to see whether DPCs exist in the processor's queue. If there are, the processor drops to IRQL DISPATCH_LEVEL (2) and then processes the DPCs in the queue in a First In First Out (FIFO) manner, calling the respective functions, until the queue is empty. Only then can the processor's IRQL drop to zero, and resume executing the original code that was disturbed at the time the interrupt arrived.



DPCs can be customized in some ways. Check out the docs for the functions `KeSetImportantDpc` and `KeSetTargetProcessorDpc`.

Figure 6-6 augments figure 6-5 with the DPC routine execution.

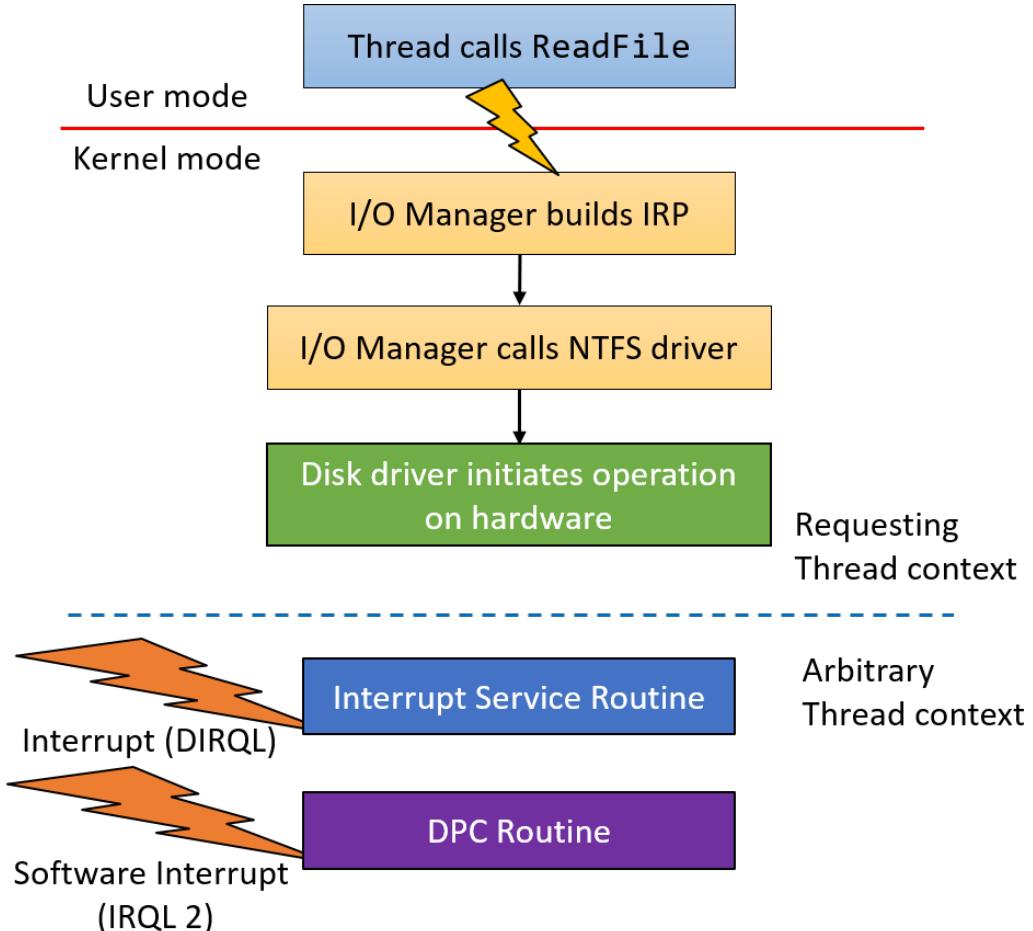


Figure 6-6: Typical I/O request processing (part 2)

Using DPC with a Timer

DPCs were originally created for use by ISRs. However, there are other mechanisms in the kernel that utilize DPCs.

One such use is with a kernel timer. A kernel timer, represented by the KTIMER structure allows setting up a timer to expire some time in the future, based on a relative interval or absolute time. This timer is a dispatcher object and so can be waited upon with `KeWaitForSingleObject` (discussed later in this chapter in the section “Synchronization”). Although waiting is possible, it’s inconvenient for a timer. A simpler approach is to call some callback when the timer expires. This is exactly what the kernel timer provides using a DPC as its callback.

The following code snippet shows how to configure a timer and associate it with a DPC. When the timer expires, the DPC is inserted into a CPU’s DPC queue and so executes as soon as possible. Using

a DPC is more powerful than a zero IRQL based callback, since it is guaranteed to execute before any user mode code (and most kernel mode code).

```
KTIMER Timer;
KDPC TimerDpc;

void InitializeAndStartTimer(ULONG msec) {
    KeInitializeTimer(&Timer);
    KeInitializeDpc(&TimerDpc,
        OnTimerExpired,           // callback function
        nullptr);                // passed to callback as "context"

    // relative interval is in 100nsec units (and must be negative)
    // convert to msec by multiplying by 10000

    LARGE_INTEGER interval;
    interval.QuadPart = -10000LL * msec;
    KeSetTimer(&Timer, interval, &TimerDpc);
}

void OnTimerExpired(KDPC* Dpc, PVOID context, PVOID, PVOID) {
    UNREFERENCED_PARAMETER(Dpc);
    UNREFERENCED_PARAMETER(context);

    NT_ASSERT(KeGetCurrentIrql() == DISPATCH_LEVEL);

    // handle timer expiration
}
```

Asynchronous Procedure Calls

We've seen in the previous section that DPCs are objects encapsulating a function to be called at IRQL DISPATCH_LEVEL. The calling thread does not matter, as far as DPCs are concerned.

Asynchronous Procedure Calls (APCs) are also data structures that encapsulate a function to be called. But contrary to a DPC, an APC is targeted towards a particular thread, so only that thread can execute the function. This means each thread has an APC queue associated with it.

There are three types of APCs:

- User mode APCs - these execute in user mode at IRQL PASSIVE_LEVEL only when the thread goes into alertable state. This is typically accomplished by calling an API such as `SleepEx`, `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx` and similar APIs. The last argument to these functions can be set to TRUE to put the thread in alertable state. In this state it looks at its APC queue, and if not empty - the APCs now execute until the queue is empty.

- Normal kernel-mode APCs - these execute in kernel mode at IRQL PASSIVE_LEVEL and preempt user-mode code (and user-mode APCs).
- Special kernel APCs - these execute in kernel mode at IRQL APC_LEVEL (1) and preempt user-mode code, normal kernel APCs, and user-mode APCs. These APCs are used by the I/O manager to complete I/O operations as will be discussed in the next chapter.

The APC API is undocumented in kernel mode (but has been reversed engineered enough to allow usage if desired).



User-mode can use (user mode) APCs by calling certain APIs. For example, calling `ReadFileEx` or `WriteFileEx` start an asynchronous I/O operation. When the operation completes, a user-mode APC is attached to the calling thread. This APC will execute when the thread enters an *alertable* state as described earlier. Another useful function in user mode to explicitly generate an APC is `QueueUserAPC`. Check out the Windows API documentation for more information.

Critical Regions and Guarded Regions

A *Critical Region* prevents user mode and normal kernel APCs from executing (special kernel APCs can still execute). A thread enters a critical region with `KeEnterCriticalSection` and leaves it with `KeLeaveCriticalSection`. Some functions in the kernel require being inside a critical region, especially when working with executive resources (see the section “Executive Resources” later in this chapter).

A *Guarded Region* prevents all APCs from executing. Call `KeEnterGuardedRegion` to enter a guarded region and `KeLeaveGuardedRegion` to leave it. Recursive calls to `KeEnterGuardedRegion` must be matched with the same number of calls to `KeLeaveGuardedRegion`.



Raising the IRQL to APC_LEVEL disables delivery of all APCs.



Write RAII wrappers for entering/leaving critical and guarded regions.

Structured Exception Handling

An exception is an event that occurs because of a certain instruction that did something that caused the processor to raise an error. Exceptions are in some ways similar to interrupts, the main difference being that an exception is synchronous and technically reproducible under the same conditions, whereas

an interrupt is asynchronous and can arrive at any time. Examples of exceptions include division by zero, breakpoint, page fault, stack overflow and invalid instruction.

If an exception occurs, the kernel catches this and allows code to handle the exception, if possible. This mechanism is called *Structured Exception Handling* (SEH) and is available for user-mode code as well as kernel-mode code.

The kernel exception handlers are called based on the *Interrupt Dispatch Table* (IDT), the same one holding mapping between interrupt vectors and ISRs. Using a kernel debugger, the !idt command shows all these mappings. The low numbered interrupt vectors are in fact exception handlers. Here's a sample output from this command:

```
1kd> !idt

Dumping IDT: fffff8011d941000

00: fffff8011dd6c100 nt!KiDivideErrorFaultShadow
01: fffff8011dd6c180 nt!KiDebugTrapOrFaultShadow      Stack = 0xFFFFF8011D9459D0
02: fffff8011dd6c200 nt!KiNmiInterruptShadow        Stack = 0xFFFFF8011D9457D0
03: fffff8011dd6c280 nt!KiBreakpointTrapShadow
04: fffff8011dd6c300 nt!KiOverflowTrapShadow
05: fffff8011dd6c380 nt!KiBoundFaultShadow
06: fffff8011dd6c400 nt!KiInvalidOpcodeFaultShadow
07: fffff8011dd6c480 nt!KiNpxNotAvailableFaultShadow
08: fffff8011dd6c500 nt!KiDoubleFaultAbortShadow     Stack = 0xFFFFF8011D9453D0
09: fffff8011dd6c580 nt!KiNpxSegmentOverrunAbortShadow
0a: fffff8011dd6c600 nt!KiInvalidTssFaultShadow
0b: fffff8011dd6c680 nt!KiSegmentNotPresentFaultShadow
0c: fffff8011dd6c700 nt!KiStackFaultShadow
0d: fffff8011dd6c780 nt!KiGeneralProtectionFaultShadow
0e: fffff8011dd6c800 nt!KiPageFaultShadow
10: fffff8011dd6c880 nt!KiFloatingErrorFaultShadow
11: fffff8011dd6c900 nt!KiAlignmentFaultShadow

(truncated)
```

Note the function names - most are very descriptive. These entries are connected to Intel/AMD (in this example) faults. Some common examples of exceptions include:

- Division by zero (0)
- Breakpoint (3) - the kernel handles this transparently, passing control to an attached debugger (if any).
- Invalid opcode (6) - this fault is raised by the CPU if it encounters an unknown instruction.
- Page fault (14) - this fault is raised by the CPU if the page table entry used for translating virtual to physical addresses has the Valid bit set to zero, indicating (as far as the CPU is concerned) that the page is not resident in physical memory.

Some other exceptions are raised by the kernel as a result of a previous CPU fault. For example, if a page fault is raised, the Memory Manager's page fault handler will try to locate the page that is not resident in RAM. If the page happens not to exist at all, the Memory Manager will raise an Access Violation exception.

Once an exception is raised, the kernel searches the function where the exception occurred for a handler (except for some exceptions which it handles transparently, such as Breakpoint (3)). If not found, it will search up the call stack, until such handler is found. If the call stack is exhausted, the system will crash.

How can a driver handle these types of exceptions? Microsoft added four keywords to the C language to allow developers to handle such exceptions, as well as have code execute no matter what. Table 6-1 shows the added keywords with a brief description.

Table 6-1: Keywords for working with SEH

Keyword	Description
<code>__try</code>	Starts a block of code where exceptions may occur.
<code>__except</code>	Indicates if an exception is handled, and provides the handling code if it is.
<code>__finally</code>	Unrelated to exceptions directly. Provides code that is guaranteed to execute no matter what - whether the <code>__try</code> block is exited normally, with a <code>return</code> statement, or because of an exception.
<code>__leave</code>	Provides an optimized mechanism to jump to the <code>__finally</code> block from somewhere within a <code>__try</code> block.

The valid combination of keywords is `__try/__except` and `__try/__finally`. However, these can be combined by using nesting to any level.



These same keywords work in user mode as well, in much the same way.

Using `__try/__except`

In chapter 4, we implemented a driver that accesses a user-mode buffer to get data needed for the driver's operation. We used a direct pointer to the user's buffer. However, this is not guaranteed to be safe. For example, the user-mode code (say from another thread) could free the buffer, just before the driver accesses it. In such a case, the driver would cause a system crash, essentially because of a user's error (or malicious intent). Since user data should never be trusted, such access should be wrapped in a `__try/__except` block to make sure a bad buffer does not crash the driver.

Here is the important part of a revised `IRP_MJ_WRITE` handler using an exception handler:

```

do {
    if (irpSp->Parameters.Write.Length < sizeof(ThreadData)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    auto data = (ThreadData*)Irp->UserBuffer;
    if (data == nullptr) {
        status = STATUS_INVALID_PARAMETER;
        break;
    }
    __try {
        if (data->Priority < 1 || data->Priority > 31) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        PETHREAD Thread;
        status = PsLookupThreadByThreadId(
            ULONGToHandle(data->ThreadId), &Thread);
        if (!NT_SUCCESS(status))
            break;
        KeSetPriorityThread((PKTHREAD)Thread, data->Priority);
        ObDereferenceObject(Thread);
        KdPrint(("Thread Priority change for %d to %d succeeded!\n",
            data->ThreadId, data->Priority));
        break;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // probably something wrong with the buffer
        status = STATUS_ACCESS_VIOLATION;
    }
} while(false);

```

Placing EXCEPTION_EXECUTE_HANDLER in __except says that any exception is to be handled. We can be more selective by calling GetExceptionCode and looking at the actual exception. If we don't expect this, we can tell the kernel to continue looking for handlers up the call stack:

```

__except (GetExceptionCode() == STATUS_ACCESS_VIOLATION
    ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    // handle exception
}

```

Does all this mean that the driver can catch any and all exceptions? If so, the driver will never cause a system crash. Fortunately (or unfortunately, depending on your perspective), this is not the case.

Access violation, for example, is something that can only be caught if the violated address is in user space. If it's in kernel space, it cannot be caught and still cause a system crash. This makes sense, since something bad has happened and the kernel will not let the driver get away with it. User mode addresses, on the other hand, are not at the control of the driver, so such exceptions can be caught and handled.

The SEH mechanism can also be used by drivers (and user-mode code) to raise custom exceptions. The kernel provides the generic function `ExRaiseStatus` to raise any exception and some specific functions like `ExRaiseAccessViolation`:

```
void ExRaiseStatus(NTSTATUS Status);
```

A driver can also crash the system explicitly if it concludes that something really bad going on, such as data being corrupted from underneath the driver. The kernel provides the `KeBugCheckEx` for this purpose:

```
VOID KeBugCheckEx(
    _In_ ULONG BugCheckCode,
    _In_ ULONG_PTR BugCheckParameter1,
    _In_ ULONG_PTR BugCheckParameter2,
    _In_ ULONG_PTR BugCheckParameter3,
    _In_ ULONG_PTR BugCheckParameter4);
```

`KeBugCheckEx` is the normal kernel function that generates a crash. `BugCheckCode` is the crash code to be reported, and the other 4 numbers can provide more details about the crash. If the bugcheck code is one of those documented by Microsoft, the meaning of the other 4 numbers must be provided as documented. (See the next section *System Crash* for more details).

Using `__try`/`__finally`

Using a block of `__try` and `__finally` is not directly related to exceptions. This is about making sure some piece of code executes no matter what - whether the code exits cleanly or mid-way because of an exception. This is similar in concept to the `finally` keyword popular in some high level languages (e.g. Java, C#). Here is a simple example to show the problem:

```
void foo() {
    void* p = ExAllocatePoolWithTag(PagedPool, 1024, DRIVER_TAG);
    if(p == nullptr)
        return;

    // do something with p

    ExFreePool(p);
}
```

The above code seems harmless enough. However, there are several issues with it:

- If an exception is thrown between the allocation and the release, a handler in the caller will be searched, but the memory will not be freed.
- If a `return` statement is used in some conditional between the allocation and release, the buffer will not be freed. This requires the code to be careful to make sure all exit points from the function pass through the code freeing the buffer.

The second bullet can be implemented with careful coding, but is a burden best avoided. The first bullet cannot be handled with standard coding techniques. This is where `__try`/`__finally` come in. Using this combination, we can make sure the buffer is freed no matter what happens in the `__try` block:

```
void foo() {
    void* p = ExAllocatePoolWithTag(PagedPool, 1024, DRIVER_TAG);
    if(p == nullptr)
        return;
    __try {
        // do something with p
    }
    __finally {
        // called no matter what
        ExFreePool(p);
    }
}
```

With the above code in place, even if `return` statements appear within the `__try` body, the `__finally` code will be called before actually returning from the function. If some exception occurs, the `__finally` block runs first before the kernel searches up the call stack for possible handlers.

`__try`/`__finally` is useful not just with memory allocations, but also with other resources, where some acquisition and release need to take place. One common example is when synchronizing threads accessing some shared data. Here is an example of acquiring and releasing a fast mutex (fast mutex and other synchronization primitives are described later in this chapter):

```
FAST_MUTEX MyMutex;

void foo() {
    ExAcquireFastMutex(&MyMutex);
    __try {
        // do work while the fast mutex is held
    }
    __finally {
        ExReleaseFastMutex(&MyMutex);
    }
}
```

Using C++ RAII Instead of __try / __finally

Although the preceding examples with __try/__finally work, they are not terribly convenient. Using C++ we can build RAII wrappers that do the right thing without the need to use __try/__finally. C++ does not have a finally keyword like C# or Java, but it doesn't need one - it has destructors.

Here is a very simple, bare minimum, example that manages a buffer allocation with a RAII class:

```
template<typename T = void>
struct kunique_ptr {
    explicit kunique_ptr(T* p = nullptr) : _p(p) {}
    ~kunique_ptr() {
        if (_p)
            ExFreePool(_p);
    }

    T* operator->() const {
        return _p;
    }

    T& operator*() const {
        return *_p;
    }

private:
    T* _p;
};
```

The class uses templates to allow working easily with any type of data. An example usage follows:

```
struct MyData {
    ULONG Data1;
    HANDLE Data2;
};

void foo() {
    // take charge of the allocation
    kunique_ptr<MyData> data((MyData*)ExAllocatePool(PagedPool, sizeof(MyData))\

);
    // use the pointer
    data->Data1 = 10;
    // when the object goes out of scope, the destructor frees the buffer
}
```

If you don't normally use C++ as your primary programming language, you may find the above code confusing. You can continue working with `_try/_finally`, but I recommend getting acquainted with this type of code. In any case, even if you struggle with the implementation of `kunique_ptr` above, you can still use it without needing to understand every little detail.

The `kunique_ptr` type presented above is a bare minimum. You should also remove the copy constructor and copy assignment, and allow move copy and assignment (C++ 11 and later, for ownership transfer). Here is a more complete implementation:

```
template<typename T = void>
struct kunique_ptr {
    explicit kunique_ptr(T* p = nullptr) : _p(p) {}

    // remove copy ctor and copy = (single owner)
    kunique_ptr(const kunique_ptr&) = delete;
    kunique_ptr& operator=(const kunique_ptr&) = delete;

    // allow ownership transfer
    kunique_ptr(kunique_ptr&& other) : _p(other._p) {
        other._p = nullptr;
    }

    kunique_ptr& operator=(kunique_ptr&& other) {
        if (&other != this) {
            Release();
            _p = other._p;
            other._p = nullptr;
        }
        return *this;
    }

    ~kunique_ptr() {
        Release();
    }

    operator bool() const {
        return _p != nullptr;
    }

    T* operator->() const {
```

```
        return _p;
    }

T& operator*() const {
    return *_p;
}

void Release() {
    if (_p)
        ExFreePool(_p);
}

private:
    T* _p;
};
```

We'll build other RAII wrappers for synchronization primitives later in this chapter.



Using C++ RAII wrappers has one missing piece - if an exception occurs, the destructor will **not** be called, so a leak of some sort occurs. The reason this does not work (as it does in user-mode), is the lack of a C++ runtime and the current inability of the compiler to set up elaborate code with `__try/__finally` to mimic this effect. Even so, it's still very useful, as in many cases exceptions are not expected, and even if they are, no handler exists in the driver for that and the system should probably crash anyway.

System Crash

As we already know, if an unhandled exception occurs in kernel mode, the system crashes, typically with the “Blue Screen of Death” (BSOD) showing its face (on Windows 8+, that’s literally a face - *saddy* or *frowny* - the inverse of *smiley*). In this section, we’ll discuss what happens when the system crashes and how to deal with it.

The system crash has many names, all meaning the same thing - “Blue screen of Death”, “System failure”, “Bugcheck”, “Stop error”. The BSOD is not some punishment, as may seem at first, but a protection mechanism. If kernel code, which is supposed to be trusted, did something bad, stopping everything is probably the safest approach, as perhaps letting the code continue roaming around may result in an unbootable system if some important files or Registry data is corrupted.

Recent versions of Windows 10 have some alternate colors for when the system crashes. Green is used for insider preview builds, and I actually encountered a pink as well (power-related errors).

If the crashed system is connected to a kernel debugger, the debugger will break. This allows examining the state of the system before other actions take place.

The system can be configured to perform some operations if the system crashes. This can be done with the *System Properties* UI on the *Advanced* tab. Clicking *Settings...* at the *Startup and Recovery* section brings the *Startup and Recovery* dialog where the *System Failure* section shows the available options. Figure 6-7 shows these two dialogs.

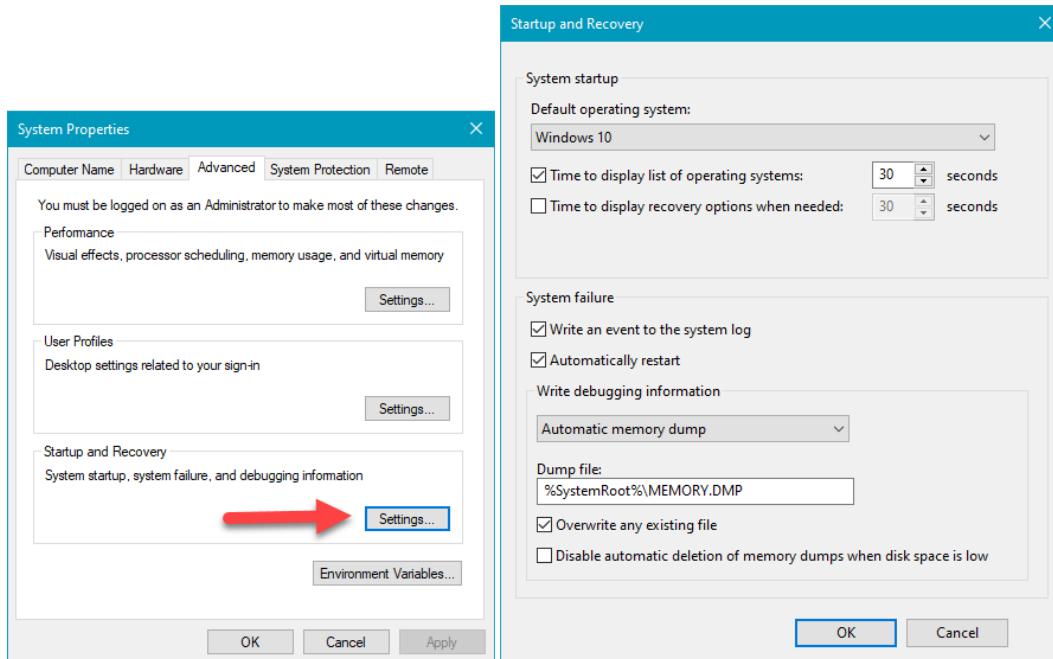


Figure 6-7: Startup and recovery settings

If the system crashes, an event entry can be written to the event log. It's checked by default, and there is no good reason to change it. The system is configured to automatically restart; this has been the default since Windows 2000.

The most important setting is the generation of a dump file. The dump file captures the system state at the time of the crash, so it can later be analyzed by loading the dump file into the debugger. The type of the dump file is important since it determines what information will be present in the dump. The dump is not written to the target file at crash time, but instead written to the first page file. Only when the system restarts, the kernel notices there is dump information in the page file, and it copies the data to the target file. The reason has to do with the fact that at system crash time it may be too dangerous to write something to a new file (or overwrite an existing file); the I/O system may not be stable enough. The best bet is to write the data to a page file, which is already open anyway. The downside is that the page file must be large enough to contain the dump, otherwise the dump file will not be generated.



The dump file contains physical memory only.

The dump type determines what data would be written and hints at the page file size that may be required. Here are the options:

- *Small memory dump* (256 KB on Windows 8 and later, 64 KB on older systems) - a very minimal dump, containing basic system information and information on the thread that caused the crash. Usually this is too little to determine what happened in all but the most trivial cases. The upside is that the file is small, so it can be easily moved.
- *Kernel memory dump* - this is the default on Windows 7 and earlier versions. This setting captures all kernel memory but no user memory. This is usually good enough, since a system crash can only be caused by kernel code misbehaving. It's extremely unlikely that user-mode had anything to do with it.
- *Complete memory dump* - this provides a dump of all physical memory, user memory and kernel memory. This is the most complete information available. The downside is the size of the dump, which could be gigantic depending on the size of RAM (the total size of the final file). The obvious optimization is not to include unused pages, but *Complete Memory Dump* does not do that.
- *Automatic memory dump* (Windows 8+) - this is the default on Windows 8 and later. This is the same as kernel memory dump, but the kernel resizes the page file on boot to a size that guarantees with high probability that the page file size would be large enough to contain a kernel dump. This is only done if the page file size is specified as "System managed" (the default).
- *Active memory dump* (Windows 10+) - this is similar to a complete memory dump, with two exceptions. First, unused pages are not written. Second, if the crashed system is hosting guest virtual machines, the memory they were using at the time is not captured (as it's unlikely these have anything to do with the host crashing). These optimizations help in reducing the dump file size.

Crash Dump Information

Once you have a crash dump in hand, you can open it in *WinDbg* by selecting *File/Open Dump File* and navigating to the file. The debugger will spew some basic information similar to the following:

```
Microsoft (R) Windows Debugger Version 10.0.18317.1001 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Loading Dump File [C:\Windows\MEMORY.DMP]
Kernel Bitmap Dump File: Kernel address space is available, User address space \
may not be available.
```

```
***** Path validation summary *****
Response                      Time (ms)      Location
Deferred                         0             SRV*c:\Symbols*http://msdl.micro\
soft.com/download/symbols
Symbol search path is: SRV*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
```

```
Windows 10 Kernel Version 18362 MP (4 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 18362.1.amd64fre.19h1_release.190318-1202
Machine Name:
Kernel base = 0xfffffff803`70abc000 PsLoadedModuleList = 0xfffffff803`70eff2d0
Debug session time: Wed Apr 24 15:36:55.613 2019 (UTC + 3:00)
System Uptime: 0 days 0:05:38.923
Loading Kernel Symbols
.....Page 2001b5efc too large to be in the dump \
file.
Page 20001ebfb too large to be in the dump file.
.
.
.
Loading User Symbols
PEB is paged out (Peb.Ldr = 00000054`34256018). Type ".hh dbgerr001" for detail\
ls
Loading unloaded module list
.
.
.
For analysis of this file, run !analyze -v
nt!KeBugCheckEx:
fffff803`70c78810 48894c2408      mov     qword ptr [rsp+8],rcx ss:fffff988`53b\
0f6b0=000000000000000a
```

The debugger suggests running `!analyze -v` and it's the most common thing to do at the start of dump analysis. Notice the call stack is at `KeBugCheckEx`, which is the function generating the bugcheck.

The default logic behind `!analyze -v` performs basic analysis on the thread that caused the crash and shows a few pieces of information related to the crash dump code:

```
2: kd> !analyze -v
*****
*                                         Bugcheck Analysis
*
*****
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high. This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: fffffd907b0dc7660, memory referenced
Arg2: 0000000000000002, IRQL
Arg3: 0000000000000000, value 0 = read operation, 1 = write operation
```

Arg4: fffff80375261530, address which referenced memory

Debugging Details:

(truncated)

DUMP_TYPE: 1

BUGCHECK_P1: fffffd907b0dc7660

BUGCHECK_P2: 2

BUGCHECK_P3: 0

BUGCHECK_P4: fffff80375261530

READ_ADDRESS: Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool

Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool

fffffd907b0dc7660 Paged pool

CURRENT_IRQL: 2

FAULTING_IP:

myfault+1530

fffff803`75261530 8b03 mov eax,dword ptr [rbx]

(truncated)

ANALYSIS_VERSION: 10.0.18317.1001 amd64fre

TRAP_FRAME: fffff98853b0f7f0 -- (.trap 0xfffffff98853b0f7f0)

NOTE: The trap frame does not contain all registers.

Some register values may be zeroed or incorrect.

rax=0000000000000000 rbx=0000000000000000 rcx=fffffd90797400340

rdx=0000000000000880 rsi=0000000000000000 rdi=0000000000000000

rip=fffff80375261530 rsp=fffff98853b0f980 rbp=0000000000000002

r8=fffffd9079c5cec10 r9=0000000000000000 r10=fffffd907974002c0

r11=fffffd907b0dc1650 r12=0000000000000000 r13=0000000000000000

r14=0000000000000000 r15=0000000000000000

iopl=0 nv up ei ng nz na po nc

myfault+0x1530:

fffff803`75261530 8b03 mov eax,dword ptr [rbx] ds:00000000`00000\

000=????????

Resetting **default scope**

LAST_CONTROL_TRANSFER: from fffff80370c8a469 to fffff80370c78810

STACK_TEXT:

```
fffff988`53b0f6a8 fffff803`70c8a469 : 00000000`0000000a fffffd907`b0dc7660 00000`00000002 00000000`00000000 : nt!KeBugCheckEx
fffff988`53b0f6b0 fffff803`70c867a5 : ffff8788`e4604080 ffffff4c`c66c7010 00000`00000003 00000000`00000880 : nt!KiBugCheckDispatch+0x69
fffff988`53b0f7f0 fffff803`75261530 : ffffff4c`c66c7000 00000000`00000000 fffff`988`53b0f9e0 00000000`00000000 : nt!KiPageFault+0x465
fffff988`53b0f980 fffff803`75261e2d : fffff988`00000000 00000000`00000000 ffff8`788`ec7cf520 00000000`00000000 : myfault+0x1530
fffff988`53b0f9b0 fffff803`75261f88 : ffffff4c`c66c7010 00000000`000000f0 00000`00000001 ffffff30`21ea80aa : myfault+0x1e2d
fffff988`53b0fb00 fffff803`70ae3da9 : ffff8788`e6d8e400 00000000`00000001 00000`000`83360018 00000000`00000001 : myfault+0x1f88
fffff988`53b0fb40 fffff803`710d1dd5 : fffff988`53b0fec0 ffff8788`e6d8e400 00000`00000001 ffff8788`ecdb6690 : nt!IoCallDriver+0x59
fffff988`53b0fb80 fffff803`710d172a : ffff8788`00000000 00000000`83360018 00000`00000000 fffff988`53b0fec0 : nt!IoPynchronousServiceTail+0x1a5
fffff988`53b0fc20 fffff803`710d1146 : 00000054`344feb28 00000000`00000000 00000`00000000 00000000`00000000 : nt!IoP XxxControlFile+0x5ca
fffff988`53b0fd60 fffff803`70c89e95 : ffff8788`e4604080 fffff988`53b0fec0 00000`054`344feb28 fffff988`569fd630 : nt!NtDeviceIoControlFile+0x56
fffff988`53b0fdd0 00007ff8`ba39c147 : 00000000`00000000 00000000`00000000 00000`00000000 00000000`00000000 : nt!KiSystemServiceCopyEnd+0x25
00000054`344feb48 00000000`00000000 : 00000000`00000000 00000000`00000000 00000`00000000 00000000`00000000 : 0x000007ff8`ba39c147
```

(truncated)

FOLLOWUP_IP:

myfault+1530

```
fffff803`75261530 8b03          mov     eax,dword ptr [rbx]
```

FAULT_INSTR_CODE: 8d48038b

SYMBOL_STACK_INDEX: 3

SYMBOL_NAME: myfault+1530

```
FOLLOWUP_NAME: MachineOwner  
  
MODULE_NAME: myfault  
  
IMAGE_NAME: myfault.sys  
  
(truncated)
```

Every crash dump code can have up to 4 numbers that provide more information about the crash. In this case, we can see the code is DRIVER_IRQL_NOT_LESS_OR_EQUAL (0xd1) and the next four numbers named *Arg1* through *Arg4* mean (in order): memory referenced, the IRQL at the time of the call, read vs. write operation and the accessing address.

The command clearly recognizes *myfault.sys* as the faulting module (driver). That's because this is an easy crash - the culprit is on the call stack as can be seen in the *STACK TEXT* section above (you can also simply use the *k* command to see it again).



The `!analyze -v` command is extensible and it's possible to add more analysis to that command using an extension DLL. You may be able to find such extensions on the web. Consult the debugger API documentation for more information on how to add your own analysis code to this command.

More complex crash dumps may show calls from the kernel only on the call stack of the offending thread. Before you conclude that you found a bug in the Windows kernel, consider this more likely scenario: A driver did something that was not fatal in itself, such as experience a buffer overflow - wrote data beyond its allocated buffer, but unfortunately ,the memory following that buffer was allocated by some other driver or the kernel, and so nothing bad happened at that time. Some time later, the kernel accessed that memory and got bad data and caused a system crash. But the faulting driver is nowhere to be found on any call stack; this is much harder to diagnose.



One way to help diagnose such issues is using *Driver Verifier*. We'll look at the basics of Driver Verifier in module 12.

Once you get the crash dump code, it's helpful to look in the debugger documentation at the topic "Bugcheck Code Reference", where common bugcheck codes are explained more fully with typical causes and ideas on what to investigate next.

Analyzing a Dump File

A dump file is a snapshot of a system's memory. Other than that, it's the same as any other kernel debugging session. You just can't set breakpoints, and certainly cannot use any *go* command. All other commands are available as usual. Commands such as `!process`, `!thread`, `!m`, `k` can be used normally. Here are some other commands and tips:

- The prompt indicates the current processor. Switching processors can be done with the command `~n` where `n` is the CPU index (it looks like switching threads in user mode).
- The `!running` command can be used to list the threads that were running on all processors at the time of the crash. Adding `-t` as an option shows the call stack for each thread. Here is an example with the above crash dump:

```
2: kd> !running -t
```

```
System Processors: (000000000000000f)
```

```
Idle Processors: (0000000000000002)
```

	Prcbs	Current	(pri)	Next	(pri)	Idle
0	fffffb000c1944180	fffff8788e91cf080	(8)			fffffb000c1\
048400					
	# Child-SP	RetAddr		Call Site		
00	00000094`ed6ee8a0	00000000`00000000	0x00007ff8`b74c4b57			
955140					
	# Child-SP	RetAddr		Call Site		
00	fffff988`53b0f6a8	fffff803`70c8a469	nt!KeBugCheckEx			
01	fffff988`53b0f6b0	fffff803`70c867a5	nt!KiBugCheckDispatch+0x69			
02	fffff988`53b0f7f0	fffff803`75261530	nt!KiPageFault+0x465			
03	fffff988`53b0f980	fffff803`75261e2d	myfault+0x1530			
04	fffff988`53b0f9b0	fffff803`75261f88	myfault+0x1e2d			
05	fffff988`53b0fb00	fffff803`70ae3da9	myfault+0x1f88			
06	fffff988`53b0fb40	fffff803`710d1dd5	nt!IofCallDriver+0x59			
07	fffff988`53b0fb80	fffff803`710d172a	nt!IopSynchronousServiceTail+0x1a5			
08	fffff988`53b0fc20	fffff803`710d1146	nt!IopXxxControlFile+0x5ca			
09	fffff988`53b0fd60	fffff803`70c89e95	nt!NtDeviceIoControlFile+0x56			
0a	fffff988`53b0fdd0	00007ff8`ba39c147	nt!KiSystemServiceCopyEnd+0x25			
0b	00000054`344feb48	00000000`00000000	0x00007ff8`ba39c147			
3	fffffb000c1c80180	fffff8788e917e0c0	(5)			fffffb000c1\
c91140					
	# Child-SP	RetAddr		Call Site		
00	fffff988`5683ec38	fffff803`70ae3da9	Ntfs!NtfsFsdClose			
01	fffff988`5683ec40	fffff803`702bb5de	nt!IofCallDriver+0x59			
02	fffff988`5683ec80	fffff803`702b9f16	FLTMGR!FltpLegacyProcessingAfterPreCallb\			

```

acksCompleted+0x15e
03 fffff988`5683ed00 fffff803`70ae3da9 FLTMGR!FltpDispatch+0xb6
04 fffff988`5683ed60 fffff803`710cfe4d nt!IofCallDriver+0x59
05 fffff988`5683eda0 fffff803`710de470 nt!IopDeleteFile+0x12d
06 fffff988`5683ee20 fffff803`70aea9d4 nt!ObpRemoveObjectRoutine+0x80
07 fffff988`5683ee80 fffff803`723391f5 nt!ObfDereferenceObject+0xa4
08 fffff988`5683eec0 fffff803`72218ca7 Ntfs!NtfsDeleteInternalAttributeStream+0\
x111
09 fffff988`5683ef00 fffff803`722ff7cf Ntfs!NtfsDecrementCleanupCounts+0x147
0a fffff988`5683ef40 fffff803`722fe87d Ntfs!NtfsCommonCleanup+0xadf
0b fffff988`5683f390 fffff803`70ae3da9 Ntfs!NtfsFsdCleanup+0x1ad
0c fffff988`5683f6e0 fffff803`702bb5de nt!IofCallDriver+0x59
0d fffff988`5683f720 fffff803`702b9f16 FLTMGR!FltpLegacyProcessingAfterPreCallb\
acksCompleted+0x15e
0e fffff988`5683f7a0 fffff803`70ae3da9 FLTMGR!FltpDispatch+0xb6
0f fffff988`5683f800 fffff803`710ccc38 nt!IofCallDriver+0x59
10 fffff988`5683f840 fffff803`710d4bf8 nt!IopCloseFile+0x188
11 fffff988`5683f8d0 fffff803`710d9f3e nt!ObCloseHandleTableEntry+0x278
12 fffff988`5683fa10 fffff803`70c89e95 nt!NtClose+0xde
13 fffff988`5683fa80 00007ff8`ba39c247 nt!KiSystemServiceCopyEnd+0x25
14 000000b5`aacf9df8 00000000`00000000 0x00007ff8`ba39c247

```

The command gives a pretty good idea of what was going on at the time of the crash.

- The !stacks command lists all thread stacks for all threads by default. A more useful variant is a search string that lists only threads where a module or function containing this string appears. This allows locating driver's code throughout the system (because it may not have been running at the time of the crash, but it's on some thread's call stack). Here's an example for the above dump:

```

2: kd> !stacks
Proc.Thread .Thread Ticks ThreadState Blocker
[fffff803710459c0 Idle]
0.000000 fffff80371048400 0000003 RUNNING nt!KiIdleLoop+0x15e
0.000000 fffffb000c17b1140 0000ed9 RUNNING hal!HalProcessorIdle+0xf
0.000000 fffffb000c1955140 0000b6e RUNNING nt!KiIdleLoop+0x15e
0.000000 fffffb000c1c91140 000012b RUNNING nt!KiIdleLoop+0x15e
[fffff8788d6a81300 System]
4.000018 fffff8788d6b8a080 0005483 Blocked nt!PopFxEmergencyWorker+0x3e
4.00001c fffff8788d6bc5140 0000982 Blocked nt!ExpWorkQueueManagerThread+0x\
127
4.000020 fffff8788d6bc9140 000085a Blocked nt!KeRemovePriQueue+0x25c

```

```
(truncated)

2: kd> !stacks 0 myfault
Proc.Thread .Thread Ticks ThreadState Blocker
[fffff803710459c0 Idle]
[fffff8788d6a81300 System]

(truncated)
[fffff8788e99070c0 notmyfault64.exe]
af4.00160c ffff8788e4604080 0000006 RUNNING nt!KeBugCheckEx

(truncated)
```

The address next to each line is the thread's ETHREAD address that can be fed to the !thread command.

System Hang

A system crash is the most common type of dump that is typically investigated. However, there is yet another type of dump that you may need to work with: a hung system. A hung system is a non-responsive or near non-responsive system. Things seem to be halted or deadlocked in some way - the system does not crash, so the first issue to deal with is how to get a dump of the system.

A dump file contains some system state, it does not have to be related to a crash or any other bad state. There are tools (including the kernel debugger) that can generate a dump file at any time.

If the system is still responsive to some extent, the Sysinternals *NotMyFault* tool can force a system crash and so force a dump file to be generated (this is in fact the way the dump in the previous section was generated). Figure 6-8 shows a screenshot of *NotMyFault*. Selecting the first (default) option and clicking *Crash* immediately crashes the system and will generate a dump file (if configured to do so).

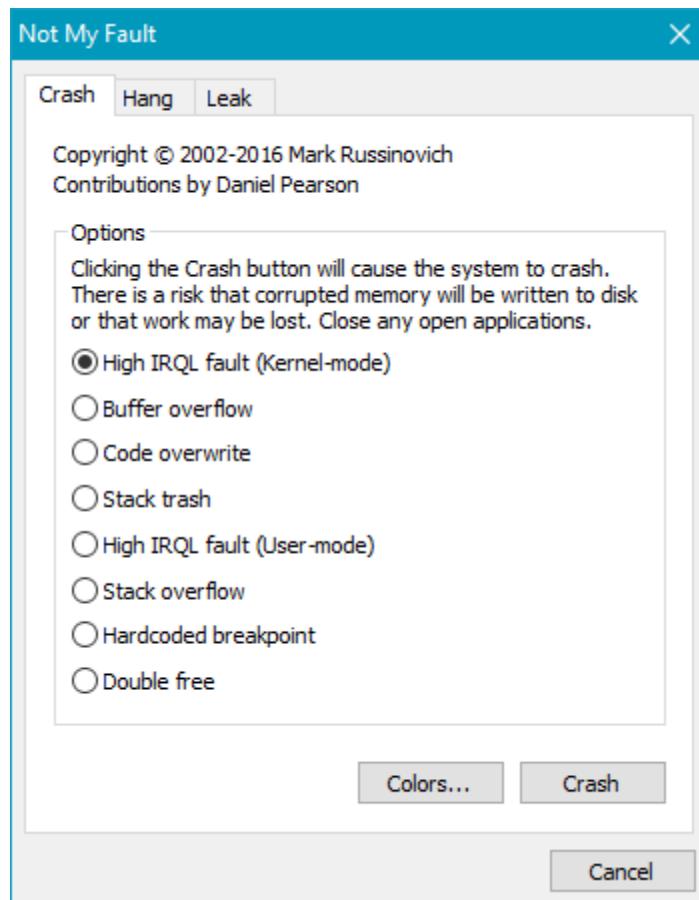


Figure 6-8: NotMyFault

NotMyFault uses a driver, *myfault.sys* that is actually responsible for the crash.



NotMyFault has 32 and 64 bit versions (the later file name ends with “64”). Remember to use the correct one for the system at hand, otherwise its driver will fail to load.

If the system is completely unresponsive, and you can attach a kernel debugger (the target was configured for debugging), then debug normally or generate a dump file using the .dump command.

If the system is unresponsive and a kernel debugger cannot be attached, it's possible to generate a crash manually if configured in the Registry beforehand (this assumes the hang was somehow expected). When a certain key combination is detected, the keyboard driver will generate a crash. Consult [this link¹](#) to get the full details. The crash code in this case is 0xe2 (MANUALLY_INITIATED_CRASH).

¹<https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/forcing-a-system-crash-from-the-keyboard>

Thread Synchronization

Threads sometimes need to coordinate work. A canonical example is a driver using a linked list to gather data items. The driver can be invoked by multiple clients, coming from many threads in one or more processes. This means manipulating the linked list must be done atomically, so it's not corrupted. If multiple threads access the same memory where at least one is a writer (making changes), this is referred to as a *data race*. If a data race occurs, all bets are off and anything can happen. Typically, within a driver, a system crash occurs sooner or later; data corruption is practically guaranteed.

In such a scenario, it's essential that while one thread manipulates the linked list, all other threads back off the linked list, and wait in some way for the first thread to finish its work. Only then another thread (just one) can manipulate the list. This is an example of thread synchronization.

The kernel provides several primitives that help in accomplishing proper synchronization to protect data from concurrent access. The following discussed various primitives and techniques for thread synchronization.

Interlocked Operations

The Interlocked set of functions provide convenient operations that are performed atomically by utilizing the hardware, which means no software objects are involved. If using these functions gets the job done, then they should be used, as these are as efficient as they can possibly be.

Technically, these `Interlocked`-family of functions are called *compiler intrinsics*, as they are instructions to the processor, disguised as functions.



The same functions (intrinsics) are available in user-mode as well.

A simple example is incrementing an integer by one. Generally, this is not an atomic operation. If two (or more) threads try to perform this at the same time on the same memory location, it's possible (and likely) some of the increments will be lost. Figure 6-9 shows a simple scenario where incrementing a value by 1 done from two threads ends up with result of 1 instead of 2.

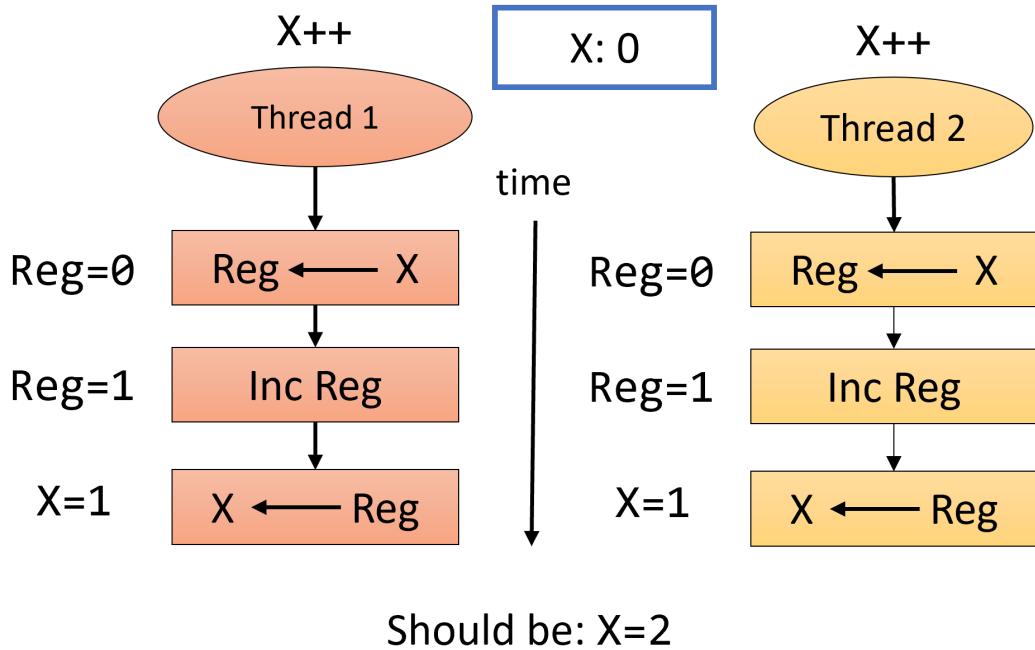


Figure 6-9: Concurrent increment



The example in figure 6-9 is extremely simplistic. With real CPUs there are other effects to consider, especially caching, which makes the shown scenario even more likely. CPU caching, store buffers, and other aspects of modern CPUs are non-trivial topics, well beyond the scope of this book.

Table 6-2 lists some of the Interlocked functions available for drivers use.

Table 6-2: Some **Interlocked** functions

Function	Description
InterlockedIncrement /	Atomically increment a 32/16/64 bit integer by one
InterlockedIncrement16 /	
InterlockedIncrement64	
InterlockedDecrement / 16 / 64	Atomically decrement a 32/16/64 bit integer by one.
InterlockedAdd / InterlockedAdd64	Atomically add one 32/64 bit integer to a variable.
InterlockedExchange / 8 / 16 / 64	Atomically exchange two 32/8/16/64 bit values.
InterlockedCompareExchange / 64 / 128	Atomically compare a variable with a value. If equal exchange with the provided value and return TRUE; otherwise, place the current value in the variable and return FALSE.



The `InterlockedCompareExchange` family of functions are used in *lock-free programming*, a programming technique to perform complex atomic operations without using software objects. This topic is well beyond the scope of this book.

 The functions in table 6-2 are also available in user mode, as these are not really functions, but rather CPU *intrinsics* - special instructions to the CPU.

Dispatcher Objects

The kernel provides a set of primitives known as *Dispatcher Objects*, also called *Waitable Objects*. These objects have a state, either **signaled** or **non-signaled**, where the meaning of signaled and non-signaled depends on the type of object. They are called “waitable” because a thread can wait on such objects until they become signaled. While waiting, the thread does not consume CPU cycles as it’s in a *Waiting* state.

The primary functions used for waiting are `KeWaitForSingleObject` and `KeWaitForMultipleObjects`. Their prototypes (with simplified SAL annotations for clarity) are shown below:

```
NTSTATUS KeWaitForSingleObject (
    _In_ PVOID Object,
    _In_ KWAIT_REASON WaitReason,
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout);

NTSTATUS KeWaitForMultipleObjects (
    _In_ ULONG Count,
    _In_reads_(Count) PVOID Object[],
    _In_ WAIT_TYPE WaitType,
    _In_ KWAIT_REASON WaitReason,
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_opt_ PLARGE_INTEGER Timeout,
    _Out_opt_ PKWAIT_BLOCK WaitBlockArray);
```

Here is a rundown of the arguments to these functions:

- *Object* - specifies the object to wait for. Note these functions work with objects, not handles. If you have a handle (maybe provided by user mode), call `ObReferenceObjectByHandle` to get the pointer to the object.

- *WaitReason* - specifies the wait reason. The list of wait reasons is pretty long, but drivers should typically set it to `Executive`, unless it's waiting because of a user request, and if so specify `UserRequest`.
- *WaitMode* - can be `UserMode` or `KernelMode`. Most drivers should specify `KernelMode`.
- *Alertable* - indicates if the thread should be in an alertable state during the wait. Alertable state allows delivering of user mode *Asynchronous Procedure Calls* (APCs). User mode APCs can be delivered if wait mode is `UserMode`. Most drivers should specify `FALSE`.
- *Timeout* - specifies the time to wait. If `NULL` is specified, the wait is indefinite - as long as it takes for the object to become signaled. The units of this argument are in 100nsec chunks, where a negative number is relative wait, while a positive number is an absolute wait measured from January 1, 1601 at midnight.
- *Count* - the number of objects to wait on.
- *Object[]* - an array of object pointers to wait on.
- *WaitType* - specifies whether to wait for all object to become signaled at once (`WaitAll`) or just one object (`WaitAny`).
- *WaitBlockArray* - an array of structures used internally to manage the wait operation. It's optional if the number of objects is \leq `THREAD_WAIT_OBJECTS` (currently 3) - the kernel will use the built-in array present in each thread. If the number of objects is higher, the driver must allocate the correct size of structures from non-paged memory, and deallocate them after the wait is over.

The main return values from `KeWaitForSingleObject` are:

- `STATUS_SUCCESS` - the wait is satisfied because the object state has become signaled.
- `STATUS_TIMEOUT` - the wait is satisfied because the timeout has elapsed.



Note that all return values from the wait functions pass the `NT_SUCCESS` macro with `true`.

`KeWaitForMultipleObjects` return values include `STATUS_TIMEOUT` just as `KeWaitForSingleObject`. `STATUS_SUCCESS` is returned if `WaitAll` wait type is specified and all objects become signaled. For `WaitAny` waits, if one of the objects became signaled, the return value is `STATUS_WAIT_0` plus its index in the array of objects (Note that `STATUS_WAIT_0` is defined to be zero).



There are some fine details associated with the wait functions, especially if wait mode is `UserMode` and the wait is alertable. Check the WDK docs for the details.

Table 6-3 lists some of the common dispatcher objects and the meaning of *signaled* and *non-signaled* for these objects.

Table 6-3: Object Types and signaled meaning

Object Type	Signaled meaning	Non-Signaled meaning
Process	process has terminated (for whatever reason)	process has not terminated
Thread	thread has terminated (for whatever reason)	thread has not terminated
Mutex	mutex is free (unowned)	mutex is held
Event	event is set	event is reset
Semaphore	semaphore count is greater than zero	semaphore count is zero
Timer	timer has expired	timer has not yet expired
File	asynchronous I/O operation completed	asynchronous I/O operation is in progress



All the object types from table 6-3 are also exported to user mode. The primary waiting functions in user mode are `WaitForSingleObject` and `WaitForMultipleObjects`.

The following sections will discuss some of common object types useful for synchronization in drivers. Some other objects will be discussed as well that are not dispatcher objects, but support waiting as well.

Mutex

Mutex is the classic object for the canonical problem of one thread among many that can access a shared resource at any one time.



Mutex is sometimes referred to as *Mutant* (its original name). These are the same thing.

A mutex is signaled when it's free. Once a thread calls a wait function and the wait is satisfied, the mutex becomes non-signaled and the thread becomes the owner of the mutex. Ownership is critical for a mutex. It means the following:

- If a thread is the owner of a mutex, it's the only one that can release the mutex.
- A mutex can be acquired more than once by the same thread. The second attempt succeeds automatically since the thread is the current owner of the mutex. This also means the thread needs to release the mutex the same number of times it was acquired; only then the mutex becomes free (signaled) again.

Using a mutex requires allocating a KMUTEX structure from non-paged memory. The mutex API contains the following functions working on that KMUTEX:

- `KeInitializeMutex` or `KeInitializeMutant` must be called once to initialize the mutex.

- One of the waiting functions, passing the address of the allocated KMUTEX structure.
- KeReleaseMutex is called when a thread that is the owner of the mutex wants to release it.

Here are the definitions of the APIs that can initialize a mutex:

```
VOID KeInitializeMutex (
    _Out_ PKMUTEX Mutex,
    _In_ ULONG Level);
VOID KeInitializeMutant ( // defined in ntifs.h
    _Out_ PKMUTANT Mutant,
    _In_ BOOLEAN InitialOwner);
```

The Level parameter in KeInitializeMutex is not used, so zero is a good value as any. KeInitializeMutant allows specifying if the current thread should be the initial owner of the mutex. KeInitializeMutex initializes the mutex to be unowned.

Releasing the mutex is done with KeReleaseMutex:

```
LONG KeReleaseMutex (
    _Inout_ PKMUTEX Mutex,
    _In_ BOOLEAN Wait);
```

The returned value is the previous state of the mutex object (including recursive ownership count), and should mostly be ignored (although it may sometimes be useful for debugging purposes). The Wait parameter indicates whether the next API call is going to be one of the wait functions. This is used as a hint to the kernel that can optimize slightly if the thread is about to enter a wait state.



As part of calling KeReleaseMutex, the IRQL is raised to DISPATCH_LEVEL. If Wait is TRUE, the IRQL is not lowered, which would allow the next wait function (KeWaitForSingleObject or KeWaitForMultipleObjects) to execute more efficiently, as no context switch can interfere.

Given the above functions, here is an example using a mutex to access some shared data so that only a single thread does so at a time:

```
KMUTEX MyMutex;
LIST_ENTRY DataHead;

void Init() {
    KeInitializeMutex(&MyMutex, 0);
}

void DoWork() {
```

```
// wait for the mutex to be available

KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);

// access DataHead freely

// once done, release the mutex

KeReleaseMutex(&MyMutex, FALSE);
}
```

It's important to release the mutex no matter what, so it's better to use `__try / __finally` to make sure it's executed however the `__try` block is exited:

```
void DoWork() {
    // wait for the mutex to be available

    KeWaitForSingleObject(&MyMutex, Executive, KernelMode, FALSE, nullptr);

    __try {
        // access DataHead freely

    }
    __finally {
        // once done, release the mutex

        KeReleaseMutex(&MyMutex, FALSE);
    }
}
```

Figure 6-10 shows two threads attempting to acquire the mutex at roughly the same time, as they want to access the same data. One thread succeeds in acquiring the mutex, the other has to wait until the mutex is released by the owner before it can acquire it.

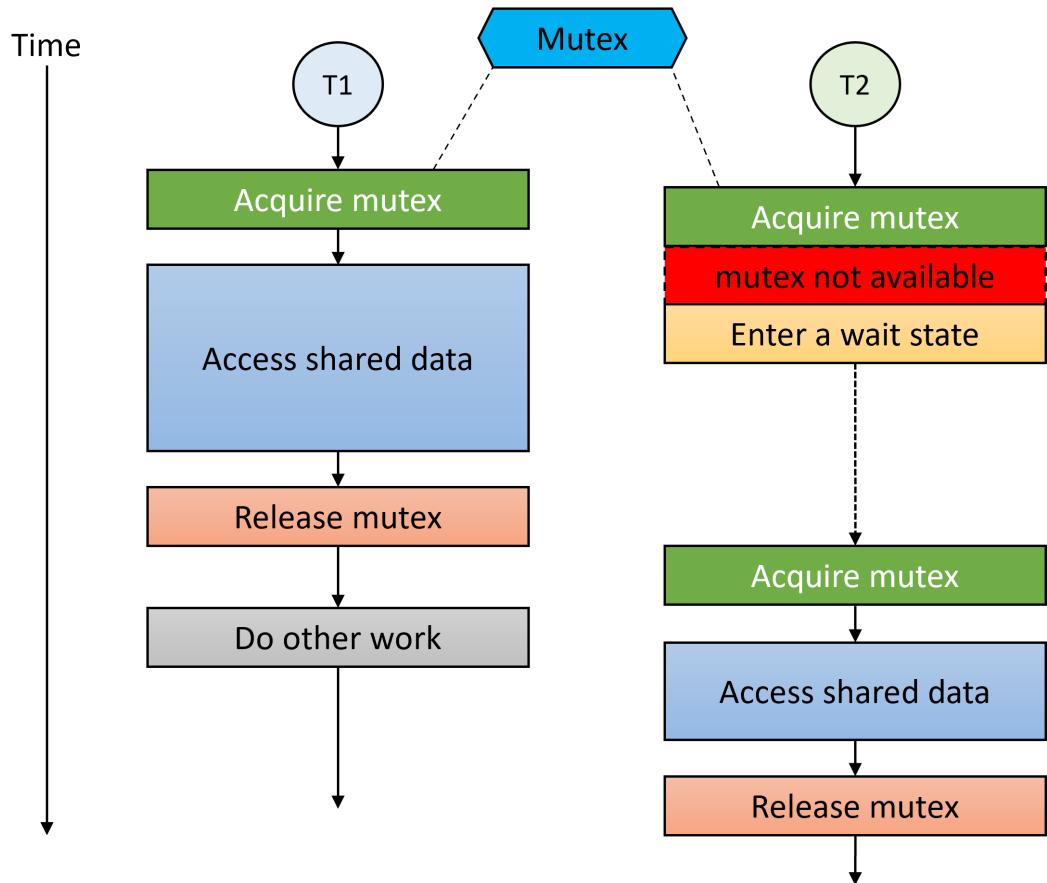


Figure 6-10: Acquiring a mutex

Since using `__try/__finally` is a bit awkward, we can use C++ to create a RAII wrapper for waits. This could also be used for other synchronization primitives.

First, we'll create a mutex wrapper that provides functions named `Lock` and `Unlock`:

```
struct Mutex {
    void Init() {
        KeInitializeMutex(&_mutex, 0);
    }

    void Lock() {
        KeWaitForSingleObject(&_mutex, Executive, KernelMode, FALSE, nullptr);
    }

    void Unlock() {
        KeReleaseMutex(&_mutex, FALSE);
    }
}
```

```
    }

private:
    KMUTEX _mutex;
};
```

Then we can create a generic RAII wrapper for waiting for any type that has a `Lock` and `Unlock` functions:

```
template<typename TLock>
struct Locker {
    explicit Locker(TLock& lock) : _lock(lock) {
        lock.Lock();
    }

    ~Locker() {
        _lock.Unlock();
    }

private:
    TLock& _lock;
};
```

With these definitions in place, we can replace the code using the mutex with the following:

```
Mutex MyMutex;

void Init() {
    MyMutex.Init();
}

void DoWork() {
    Locker<Mutex> locker(MyMutex);

    // access DataHead freely
}
```



Since locking should be done for the shortest time possible, you can use an artificial C/C++ scope containing `Locker` and the code to execute while the mutex is owned, to acquire the mutex as late as possible and release it as soon as possible.

With C++ 17 and later, `Locker` can be used without specifying the type like so:

```
Locker locker(MyMutex);
```

Since Visual Studio currently uses C++ 14 as its default language standard, you'll have to change that in the project properties under the General node / C++ Language Standard.

We'll use the same `Locker` type with other synchronization primitives in subsequent sections.

Abandoned Mutex

A thread that acquires a mutex becomes the mutex owner. The owner thread is the only one that can release the mutex. What happens to the mutex if the owner thread dies for whatever reason? The mutex then becomes an *abandoned mutex*. The kernel explicitly releases the mutex (as no thread can do it) to prevent a deadlock, so another thread would be able to acquire that mutex normally. However, the returned value from the next successful wait call is `STATUS_ABANDONED` rather than `STATUS_SUCCESS`. A driver should log such an occurrence, as it frequently indicates a bug.

Other Mutex Functions

Mutexes support a few miscellaneous functions that may be useful at times, mostly for debugging purposes. `KeReadStateMutex` returns the current state (recursive count) of the mutex, where 0 means “unowned”:

```
LONG KeReadStateMutex (_In_ PKMUTEX Mutex);
```

Just remember that after the call returns, the result may no longer be correct as the mutex state may have changed because some other thread has acquired or released the mutex before the code gets to examine the result. The benefit of this function is in debugging scenarios only.

You can get the current mutex owner with a call to `KeQueryOwnerMutant` (defined in `<ntifs.h>`) as a `CLIENT_ID` data structure, containing the thread and process IDs:

```
VOID KeQueryOwnerMutant (
    _In_ PKMUTANT Mutant,
    _Out_ PCLIENT_ID ClientId);
```

Just like with `KeReadStateMutex`, the returned information may be stale if other threads are doing work with that mutex.

Fast Mutex

A fast mutex is an alternative to the classic mutex, providing better performance. It's not a dispatcher object, and so has its own API for acquiring and releasing it. A fast mutex has the following characteristics compared with a regular mutex:

- A fast mutex cannot be acquired recursively. Doing so causes a deadlock.
- When a fast mutex is acquired, the CPU IRQL is raised to APC_LEVEL (1). This prevents any delivery of APCs to that thread.
- A fast mutex can only be waited on indefinitely - there is no way to specify a timeout.

Because of the first two bullets above, the fast mutex is slightly faster than a regular mutex. In fact, most drivers requiring a mutex use a fast mutex unless there is a compelling reason to use a regular mutex.



Don't use I/O operations while holding on to a fast mutex. I/O completions are delivered with a special kernel APC, but those are blocked while holding a fast mutex, creating a deadlock.

A fast mutex is initialized by allocating a FAST_MUTEX structure from non-paged memory and calling `ExInitializeFastMutex`. Acquiring the mutex is done with `ExAcquireFastMutex` or `ExAcquireFastMutexUnsafe` (if the current IRQL happens to be APC_LEVEL already). Releasing a fast mutex is accomplished with `ExReleaseFastMutex` or `ExReleaseFastMutexUnsafe`.

Semaphore

The primary goal of a semaphore is to limit something, such as the length of a queue. The semaphore is initialized with its maximum and initial count (typically set to the maximum value) by calling `KeInitializeSemaphore`. While its internal count is greater than zero, the semaphore is signaled. A thread that calls `KeWaitForSingleObject` has its wait satisfied, and the semaphore count drops by one. This continues until the count reaches zero, at which point the semaphore becomes non-signaled.

Semaphores use the KSEMAPHORE structure to hold their state, which must be allocated from non-paged memory. Here is the definition of `KeInitializeSemaphore`:

```
VOID KeInitializeSemaphore (
    _Out_ PRKSEMAPHORE Semaphore,
    _In_ LONG Count,           // starting count
    _In_ LONG Limit);        // maximum count
```

As an example, imagine a queue of work items managed by the driver. Some threads want to add items to the queue. Each such thread calls `KeWaitForSingleObject` to obtain one "count" of the semaphore. As long as the count is greater than zero, the thread continues and adds an item to the queue, increasing its length, and semaphore "loses" a count. Some other threads are tasked with processing work items from the queue. Once a thread removes an item from the queue, it calls `KeReleaseSemaphore` that

increments the count of the semaphore, moving it to the signaled state again, allowing potentially another thread to make progress and add a new item to the queue.

`KeReleaseSemaphore` is defined like so:

```
LONG KeReleaseSemaphore (
    _Inout_ PRKSEMAPHORE Semaphore,
    _In_ KPRIORITY Increment,
    _In_ LONG Adjustment,
    _In_ BOOLEAN Wait);
```

The `Increment` parameter indicates the priority boost to apply to the thread that has a successful waiting on the semaphore. The details of how this boost works are described in the next chapter. Most drivers should provide the value 1 (that's the default used by the kernel when a semaphore is released by the user mode `ReleaseSemaphore` API). `Adjustment` is the value to add to the semaphore's current count. It's typically one, but can be a higher value if that makes sense. The last parameter (`Wait`) indicates whether a wait operation (`KeWaitForSingleObject` or `KeWaitForMultipleObjects`) immediately follows (see the information bar in the mutex discussion above). The function returns the old count of the semaphore.



Is a semaphore with a maximum count of one equivalent to a mutex? At first, it seems so, but this is not the case. A semaphore lacks ownership, meaning one thread can acquire the semaphore, while another can release it. This is a strength, not a weakness, as described in the above example. A Semaphore's purpose is very different from that of a mutex.

You can read the current count of the semaphore by calling `KeReadStateSemaphore`:

```
LONG KeReadStateSemaphore (_In_ PRKSEMAPHORE Semaphore);
```

Event

An event encapsulates a boolean flag - either true (signaled) or false (non-signaled). The primary purpose of an event is to signal something has happened, to provide *flow synchronization*. For example, if some condition becomes true, an event can be set, and a bunch of threads can be released from waiting and continue working on some data that perhaps is now ready for processing.

There are two types of events, the type being specified at event initialization time:

- Notification event (manual reset) - when this event is set, it releases any number of waiting threads, and the event state remains set (signaled) until explicitly reset.
- Synchronization event (auto reset) - when this event is set, it releases at most one thread (no matter how many are waiting for the event), and once released the event goes back to the reset (non-signaled) state automatically.

An event is created by allocating a `KEVENT` structure from non-paged memory and then calling `KeInitializeEvent` to initialize it, specifying the event type (`NotificationEvent` or `SynchronizationEvent`) and the initial event state (signaled or non-signaled):

```
VOID KeInitializeEvent (
    _Out_ PRKEVENT Event,
    _In_ EVENT_TYPE Type,           // NotificationEvent or SynchronizationEvent
    _In_ BOOLEAN State);          // initial state (signaled=TRUE)
```

Notification events are called *Manual-reset* in user-mode terminology, while Synchronization events are called *Auto-reset*. Despite the name changes, these are the same.

Waiting for an event is done normally with the KeWaitXxx functions. Calling KeSetEvent sets the event to the signaled state, while calling KeResetEvent or KeClearEvent resets it (non-signaled state) (the latter function being a bit quicker as it does not return the previous state of the event):

```
LONG KeSetEvent (
    _Inout_ PRKEVENT Event,
    _In_ KPRIORITY Increment,
    _In_ BOOLEAN Wait);
VOID KeClearEvent (_Inout_ PRKEVENT Event);
LONG KeResetEvent (_Inout_ PRKEVENT Event);
```

Just like with a semaphore, setting an event allows providing a priority boost to the next successful wait on the event.

Finally, the current state of an event (signaled or non-signaled) can be read with KeReadStateEvent:

```
LONG KeReadStateEvent (_In_ PRKEVENT Event);
```

Named Events

Event objects can be named (as can mutexes and semaphores). This can be used as an easy way of sharing an event object with other drivers or with user-mode clients. One way of creating or opening a named event by name is with the helper functions IoCreateSynchronizationEvent and IoCreateNotificationEvent APIs:

```
PKEVENT IoCreateSynchronizationEvent(
    _In_ PUNICODE_STRING EventName,
    _Out_ PHANDLE EventHandle);
PKEVENT IoCreateNotificationEvent(
    _In_ PUNICODE_STRING EventName,
    _Out_ PHANDLE EventHandle);
```

These APIs create the named event object if it does not exist and set its state to signaled, or obtain another handle to the named event if it does exist. The name itself is provided as a normal `UNICODE_STRING` and must be a full path in the Object Manager's namespace, as can be observed in the Sysinternals *WinObj* tool.

These APIs return two values: the pointer to the event object (direct returned value) and an open handle in the `EventHandle` parameter. The returned handle is a kernel handle, to be used by the driver only. The functions return `NULL` on failure.

You can use the previously described events API to manipulate the returned event by address. Don't forget to close the returned handle (`ZwClose`) to prevent a leak. Alternatively, you can call `ObReferenceObject` on the returned pointer to make sure it's not prematurely destroyed and close the handle immediately. In that case, call `ObDereferenceObject` when you're done with the event.

Built-in Named Kernel Events

One use of the `IoCreateNotificationEvent` API is to gain access to a bunch of named event objects the kernel provides in the `\KernelObjects` directory. These events provide various notifications for memory related status, that may be useful for kernel drivers.

Figure 6-11 shows the named events in *WinObj*. Note that the lower symbolic links are actually events, as these are internally implemented as *Dynamic Symbolic Links* (see more details at <https://scorpionsoftware.net/2021/04/30/dynamic-symbolic-links/>).

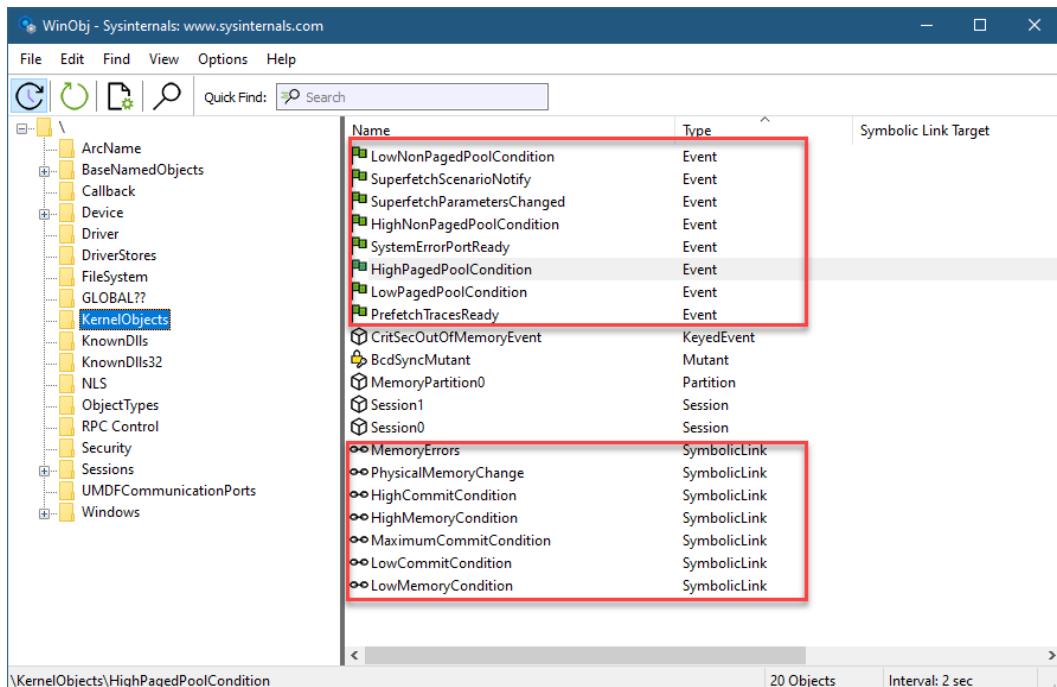


Figure 6-11: Kernel Named Events

All the events shown in figure 6-11 are Notification events. Table 6-5 lists these events with their meaning.

Table 6-5: Named kernel events

Name	Description
HighMemoryCondition	The system has lots of free physical memory
LowMemoryCondition	The system is low on physical memory
HighPagedPoolCondition	The system has lots of free paged pool memory
LowPagedPoolCondition	The system is low on paged pool memory
HighNonPagedPoolCondition	The system has lots of free non-paged pool memory
LowNonPagedPoolCondition	The system is low on non-paged pool memory
HighCommitCondition	The system has lots of free memory in RAM and paging file(s)
LowCommitCondition	The system is low on RAM and paging file(s)
MaximumCommitCondition	The system is almost out of memory, and no further increase in page files size is possible

Drivers can use these events as hints to either allocate more memory or free memory as required. The following example shows how to obtain one of these events and wait for it on some thread (error handling omitted):

```
UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\"KernelObjects"\LowCommitCondition");
HANDLE hEvent;
auto event = IoCreateNotificationEvent(&name, &hEvent);

// on some driver-created thread...
KeWaitForSingleObject(event, Executive, KernelMode, FALSE, nullptr);
// free some memory if possible...
//
// close the handle
ZwClose(hEvent);
```



Write a driver that waits on all these named events and uses `DbgPrint` to indicate a signaled event with its description.

Executive Resource

The classic synchronization problem of accessing a shared resource by multiple threads was dealt with by using a mutex or fast mutex. This works, but mutexes are pessimistic, meaning they allow a

single thread to access a shared resource. That may be unfortunate in cases where multiple threads access a shared resource by reading only.

In cases where it's possible to distinguish data changes (writes) vs. just looking at the data (reading) - there is a possible optimization. A thread that requires access to the shared resource can declare its intentions - read or write. If it declares read, other threads declaring read can do so concurrently, improving performance. This is especially useful if the shared data changes infrequently, i.e. there are considerably more reads than writes.

Mutexes by their very nature are pessimistic locks, since they enforce a single thread at a time execution. This makes them always work at the expense of possible performance gains with concurrency.

The kernel provides yet another synchronization primitive that is geared towards this scenario, known as *single writer, multiple readers*. This object is the *Executive Resource*, another special object which is not a dispatcher object.

Initializing an executive resource is done by allocating an ERESOURCE structure from non-paged pool and calling ExInitializeResourceLite. Once initialized, threads can acquire either the exclusive lock (for writes) using ExAcquireResourceExclusiveLite or the shared lock by calling ExAcquireResourceSharedLite. Once done the work, a thread releases the executive resource with ExReleaseResourceLite (no matter whether it acquired as exclusive or not).

The requirement for using the acquire and release functions is that normal kernel APCs must be disabled. This can be done with KeEnterCrticalRegion just before the acquire call, and then KeLeaveCrticalRegion just after the release call. The following code snippet demonstrates that:

```
ERESOURCE resource;

void WriteData() {
    KeEnterCrticalRegion();
    ExAcquireResourceExclusiveLite(&resource, TRUE);      // wait until acquired

    // Write to the data

    ExReleaseResourceLite(&resource);
    KeLeaveCrticalRegion();
}
```

Since these calls are so common when working with executive resources, there are functions that perform both operations with a single call:

```
void WriteData() {
    ExEnterCriticalRegionAndAcquireResourceExclusive(&resource);

    // Write to the data

    ExReleaseResourceAndLeaveCriticalSection(&resource);
}
```

A similar function exists for shared acquisition, `ExEnterCriticalSectionAndAcquireResourceShared`. Finally, before freeing the memory the resource occupies, call `ExDeleteResourceLite` to remove the resource from the kernel's resource list:

```
NTSTATUS ExDeleteResourceLite(
    _Inout_ PERESOURCE Resource);
```

You can query the number of waiting threads for exclusive and shared access of a resource with the functions `ExGetExclusiveWaiterCount` and `ExGetSharedWaiterCount`, respectively.

There are other functions for working with executive resources for some specialized cases. Consult the WDK documentation for more information.



Create appropriate C++ RAII wrappers for executive resources.

High IRQL Synchronization

The sections on synchronization so far have dealt with threads waiting for various types of objects. However, in some scenarios, threads cannot wait - specifically, when the processor's IRQL is `DISPATCH_LEVEL` (2) or higher. This section discusses these scenarios and how to handle them.

Let's examine an example scenario: A driver has a timer, set up with `KeSetTimer` and uses a DPC to execute code when the timer expires. At the same time, other functions in the driver, such as `IRP_MJ_DEVICE_CONTROL` may execute at the same time (runs at IRQL 0). If both these functions need to access a shared resource (e.g. a linked list), they must synchronize access to prevent data corruption.

The problem is that a DPC cannot call `KeWaitForSingleObject` or any other waiting function - calling any of these is fatal. So how can these functions synchronize access?

The simple case is where the system has a single CPU. In this case, when accessing the shared resource, the low IRQL function just needs to raise IRQL to DISPATCH_LEVEL and then access the resource. During that time a DPC cannot interfere with this code since the CPU's IRQL is already 2. Once the code is done with the shared resource, it can lower the IRQL back to zero, allowing the DPC to execute. This prevents execution of these routines at the same time. Figure 6-12 shows this setup.

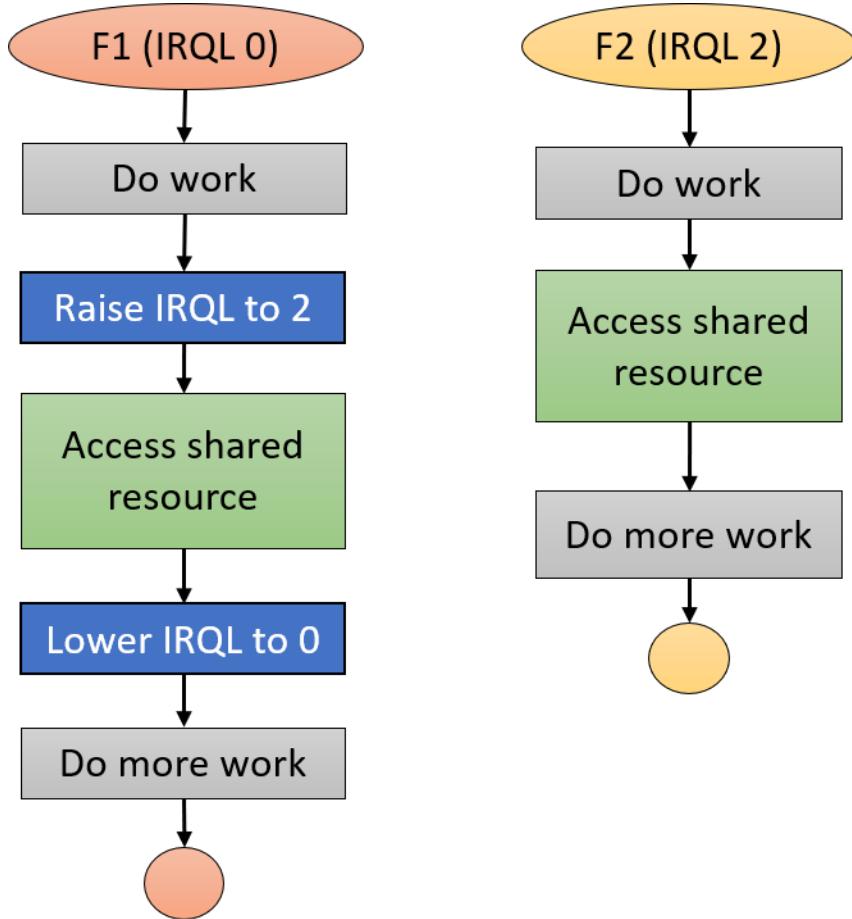


Figure 6-12: High-IRQL synchronization by manipulating IRQL

In standard systems, where there is more than one CPU, this synchronization method is not enough, because the IRQL is a CPU's property, not a system-wide property. If one CPU's IRQL is raised to 2, if a DPC needs to execute, it can execute on another CPU whose IRQL may be zero. In this case, it's possible that both functions execute at the same time, accessing the shared data, causing a data race.

How can we solve that? We need something like a mutex, but that can synchronize between processors - not threads. That's because when the CPU's IRQL is 2 or higher, the thread itself loses meaning because the scheduler cannot do work on that CPU. This kind of object exists - the *Spin Lock*.

The Spin Lock

The Spin Lock is just a bit in memory that is used with atomic test-and-set operations via an API. When a CPU tries to acquire a spin lock, and that spin lock is not currently free (the bit is set), the CPU keeps spinning on the spin lock, busy waiting for it to be released by another CPU (remember, putting the thread into a waiting state cannot be done at IRQL DISPATCH_LEVEL or higher).

In the scenario depicted in the previous section, a spin lock would need to be allocated and initialized. Each function that requires access to the shared data needs to raise IRQL to 2 (if not already there), acquire the spin lock, perform the work on the shared data, and finally release the spin lock and lower IRQL back (if applicable; not so for a DPC). This chain of events is depicted in figure 6-13.

Creating a spin lock requires allocating a KSPIN_LOCK structure from non-paged pool, and calling KeInitializeSpinLock. This puts the spin lock in the unowned state.

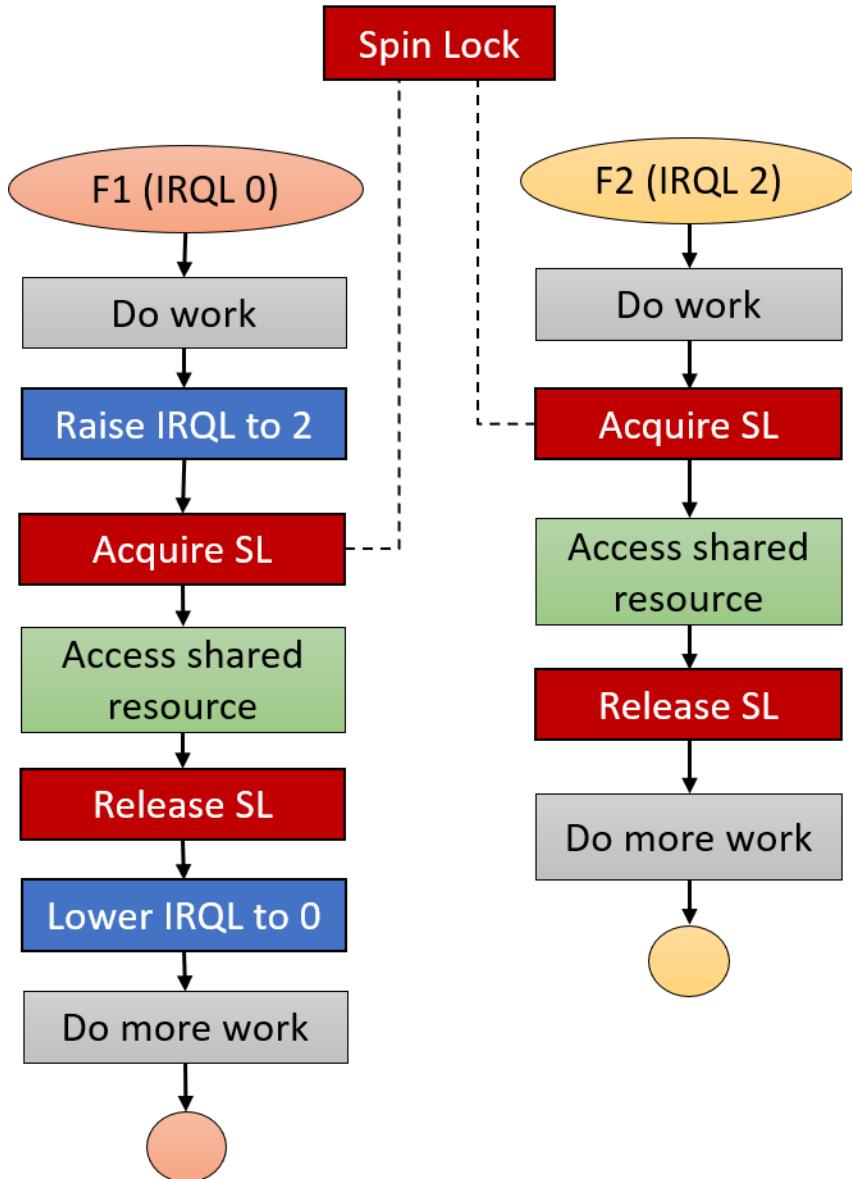


Figure 6-13: High-IRQL synchronization with a Spin Lock

Acquiring a spin lock is always a two-step process: first, raise the IRQL to the proper level, which is the highest level of any function trying to synchronize access to a shared resource. In the previous example, this *associated IRQL* is 2. Second, acquire the spin lock. These two steps are combined by using the appropriate API. This process is depicted in figure 6-14.

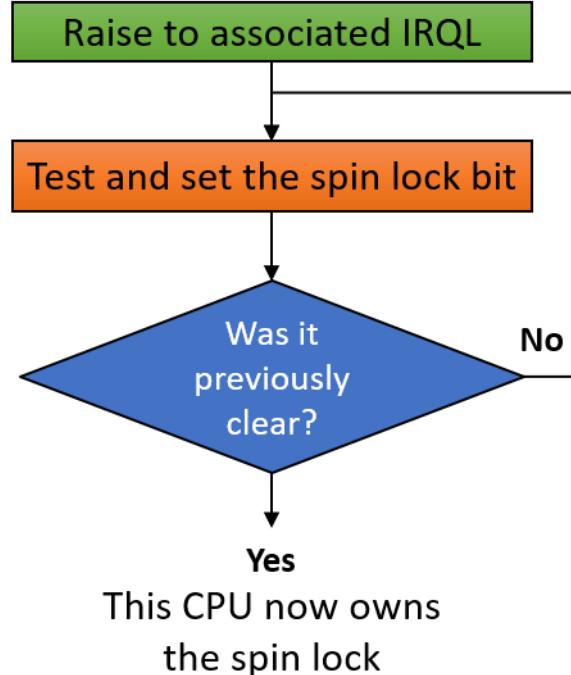


Figure 6-14: Acquiring a Spin Lock

Acquiring and releasing a spin lock is done using an API that performs the two steps outlined in figure 6-12. Table 6-4 shows the relevant APIs and the associated IRQL for the spin locks they operate on.

Table 6-4: APIs for working with spin locks

IRQL	Acquire function	Release function	Remarks
DISPATCH_LEVEL (2)	KeAcquireSpinLock	KeReleaseSpinLock	
DISPATCH_LEVEL (2)	KeAcquireSpinLockAtDpcLevel	KeReleaseSpinLockFromDpcLevel	(a)
Device IRQL	KeAcquireInterruptSpinLock	KeReleaseInterruptSpinLock	(b)
Device IRQL	KeSynchronizeExecution	(none)	(c)
HIGH_LEVEL	ExInterlockedXxx	(none)	(d)

Remarks on table 6-4:

- (a) Can be called at IRQL 2 only. Provides an optimization that just acquires the spin lock without changing IRQLs. The canonical scenario is calling these APIs within a DPC routine.
- (b) Useful for synchronizing an ISR with any other function. Hardware-based drivers with an interrupt source use these routines. The argument is an interrupt object (KINTERRUPT), where the spin lock is part of it.
- (c) KeSynchronizeExecution acquires the interrupt object spin lock, calls the provided callback and releases the spin lock. The net effect is the same as calling the pair KeAcquireInterruptSpinLock /

`KeReleaseInterruptSpinLock`.

(d) A set of three functions for manipulating `LIST_ENTRY`-based linked lists. These functions use the provided spin lock and raise IRQL to `HIGH_LEVEL`. Because of the high IRQL, these routines can be used in any IRQL, since raising IRQL is always a safe operation.



If you acquire a spin lock, be sure to release it in the same function. Otherwise, you're risking a deadlock or a system crash.



Where do spin locks come from? The scenario described here requires the driver to allocate its own spin lock to protect concurrent access to its own data from high-IRQL functions. Some spin locks exist as part of other objects, such as the `KINTERRUPT` object used by hardware-based drivers that handle interrupts. Another example is a system-wide spin lock known as the *Cancel spin lock*, which is acquired by the kernel before calling a cancellation routine registered by a driver. This is the only case where a driver released a spin lock it has not acquired explicitly.

If several CPUs try to acquire the same spin lock at the same time, which CPU gets the spin lock first? Normally, there is no order - the CPU with fastest electrons wins :). The kernel does provide an alternative, called *Queued spin locks* that serve CPUs on a FIFO basis. These only work with IRQL `DISPATCH_LEVEL`. The relevant APIs are `KeAcquireInStackQueuedSpinLock` and `KeReleaseInStackQueuedSpinLock`. Check the WDK documentation for more details.



Write a C++ wrapper for a `DISPATCH_LEVEL` spin lock that works with the `Locker` RAII class defined earlier in this chapter.

Queued Spin Locks

A variant on classic spin locks are *queued spin locks*. These behave the same as normal spin locks, with the following differences:

- Queued spin locks always raise to IRQL `DISPATCH_LEVEL` (2). This means they cannot be used for synchronizing with an ISR, for example.
- There is a queue of CPU waiting to acquire the spin lock, on a FIFO basis. This is more efficient when high contention is expected. Normal spin locks provide no guarantee as to the order of acquisition when multiple CPUs attempt to acquire a spin lock.

A queued spin lock is initialized just like a normal spin lock (`KeInitializeSpinLock`). Acquiring and releasing a queued spin lock is achieved with different APIs:

```
void KeAcquireInStackQueuedSpinLock (
    _Inout_ PKSPIN_LOCK SpinLock,
    _Out_ PKLOCK_QUEUE_HANDLE LockHandle);
void KeReleaseInStackQueuedSpinLock (
    _In_ PKLOCK_QUEUE_HANDLE LockHandle);
```

Except for a spin lock, the caller provides an opaque `KLOCK_QUEUE_HANDLE` structure that is filled in by `KeAcquireInStackQueuedSpinLock`. The same one must be passed to `KeReleaseInStackQueuedSpinLock`.

Just like with normal dispatch-level spin locks, shortcuts exist if the caller is already at IRQL `DISPATCH_LEVEL`. `KeAcquireInStackQueuedSpinLockAtDpcLevel` acquires the spin lock with no IRQL changes, while `KeReleaseInStackQueuedSpinLockFromDpcLevel` releases it.



Write a C++ RAII wrapper for a queued spin lock.

Work Items

Sometimes there is a need to run a piece of code on a different thread than the executing one. One way to do that is to create a thread explicitly and task it with running the code. The kernel provides functions that allow a driver to create a separate thread of execution: `PsCreateSystemThread` and `IoCreateSystemThread` (available in Windows 8+). These functions are appropriate if the driver needs to run code in the background for a long time. However, for time-bound operations, it's better to use a kernel-provided thread pool that will execute your code on some system worker thread.

`PsCreateSystemThread` and `IoCreateSystemThread` are discussed in chapter 8.



`IoCreateSystemThread` is preferred over `PsCreateSystemThread`, because it allows associating a device or driver object with the thread. This makes the I/O system add a reference to the object, which makes sure the driver cannot be unloaded prematurely while the thread is still executing.

A thread created by `PsCreateSystemThread` must terminate itself eventually by calling `PsTerminateSystemThread` (from within the thread). This function never returns if successful.

Work items is the term used to describe functions queued to the system thread pool. A driver can allocate and initialize a work item, pointing to the function the driver wishes to execute, and then the work item can be queued to the pool. This may seem very similar to a DPC, the primary difference

being work items always execute at IRQL PASSIVE_LEVEL (0). Thus, work items can be used by IRQL 2 code (such as DPCs) to perform operations not normally allowed at IRQL 2 (such as I/O operations).

Creating and initializing a work item can be done in one of two ways:

- Allocate and initialize the work item with `IoAllocateWorkItem`. The function returns a pointer to the opaque `IO_WORKITEM`. When finished with the work item it must be freed with `IoFreeWorkItem`.
- Allocate an `IO_WORKITEM` structure dynamically with size provided by `IoSizeofWorkItem`. Then call `IoInitializeWorkItem`. When finished with the work item, call `IoUninitializeWorkItem`.

These functions accept a device object, so make sure the driver is not unloaded while there is a work item queued or executing.



There is another set of APIs for work items, all start with `Ex`, such as `ExQueueWorkItem`. These functions do not associate the work item with anything in the driver, so it's possible for the driver to be unloaded while a work item is still executing. These APIs are marked as *deprecated* - always prefer using the `Io` functions.

To queue the work item, call `IoQueueWorkItem`. Here is its definition:

```
viud IoQueueWorkItem(
    _Inout_ PIO_WORKITEM IoWorkItem,           // the work item
    _In_     PIO_WORKITEM_ROUTINE WorkerRoutine, // the function to be called
    _In_     WORK_QUEUE_TYPE QueueType,         // queue type
    _In_opt_ PVOID Context);                   // driver-defined value
```

The callback function the driver needs to provide has the following prototype:

```
IO_WORKITEM_ROUTINE WorkItem;

void WorkItem(
    _In_     PDEVICE_OBJECT DeviceObject,
    _In_opt_ PVOID Context);
```

The system thread pool has several queues (at least logically), based on the thread priorities that serve these work items. There are several levels defined:

```
typedef enum _WORK_QUEUE_TYPE {
    CriticalWorkQueue,           // priority 13
    DelayedWorkQueue,            // priority 12
    HyperCriticalWorkQueue,      // priority 15
    NormalWorkQueue,             // priority 8
    BackgroundWorkQueue,         // priority 7
    RealTimeWorkQueue,           // priority 18
    SuperCriticalWorkQueue,      // priority 14
    MaximumWorkQueue,
    CustomPriorityWorkQueue = 32
} WORK_QUEUE_TYPE;
```

The documentation indicates `DelayedWorkQueue` must be used, but in reality, any other supported level can be used.



There is another function that can be used to queue a work item: `IoQueueWorkItemEx`. This function uses a different callback that has an added parameter which is the work item itself. This is useful if the work item function needs to free the work item before it exits.

Summary

In this chapter, we looked at various kernel mechanisms driver developers should be aware of and use. In the next chapter, we'll take a closer look at *I/O Request Packets* (IRPs).

Chapter 7: The I/O Request Packet

After a typical driver completes its initialization in `DriverEntry`, its primary job is to handle requests. These requests are packaged as the semi-documented *I/O Request Packet* (IRP) structure. In this chapter, we'll take a deeper look at IRPs and how a driver handles common IRP types.

In This chapter:

- **Introduction to IRPs**
 - Device Nodes
 - IRP and I/O Stack Location
 - Dispatch Routines
 - Accessing User Buffers
 - Putting it All Together: The Zero Driver
-

Introduction to IRPs

An IRP is a structure that is allocated from non-paged pool typically by one of the “managers” in the Executive (I/O Manager, Plug & Play Manager, Power Manager), but can also be allocated by the driver, perhaps for passing a request to another driver. Whichever entity allocating the IRP is also responsible for freeing it.

An IRP is never allocated alone. It’s always accompanied by one or more I/O Stack Location structures (`IO_STACK_LOCATION`). In fact, when an IRP is allocated, the caller must specify how many I/O stack locations need to be allocated with the IRP. These I/O stack locations follow the IRP directly in memory. The number of I/O stack locations is the number of device objects in the device stack. We’ll discuss device stacks in the next section. When a driver receives an IRP, it gets a pointer to the IRP structure itself, knowing it’s followed by a set of I/O stack location, one of which is for the driver’s use. To get the correct I/O stack location, a driver calls `IoGetCurrentIrpStackLocation` (actually a macro). Figure 7-1 shows a conceptual view of an IRP and its associated I/O stack locations.

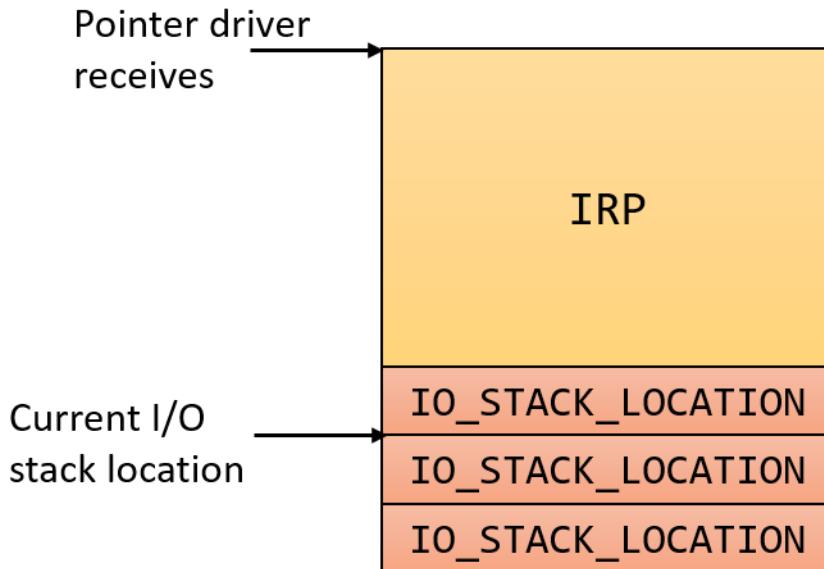


Figure 7-1: IRP and its I/O stack locations

The parameters of the request are somehow “split” between the main IRP structure and the current IO_STACK_LOCATION.

Device Nodes

The I/O system in Windows is device-centric, rather than driver-centric. This has several implications:

- Device objects can be named, and handles to device objects can be opened. The `CreateFile` function accepts a symbolic link that leads to a device object. `CreateFile` cannot accept a driver’s name as an argument.
- Windows supports device layering - one device can be layered on top of another. Any request destined for a lower device will reach the uppermost device first. This layering is common for hardware-based devices, but it works with any device type.

Figure 7-2 shows an example of several layers of devices, “stacked” one on top of the other. This set of devices is known as a *device stack*, sometimes referred to as *device node* (although the term *device node* is often used with hardware device stacks). Figure 7-1 shows six layers, or six devices. Each of these devices is represented by a `DEVICE_OBJECT` structure created by calling the standard `IoCreateDevice` function.

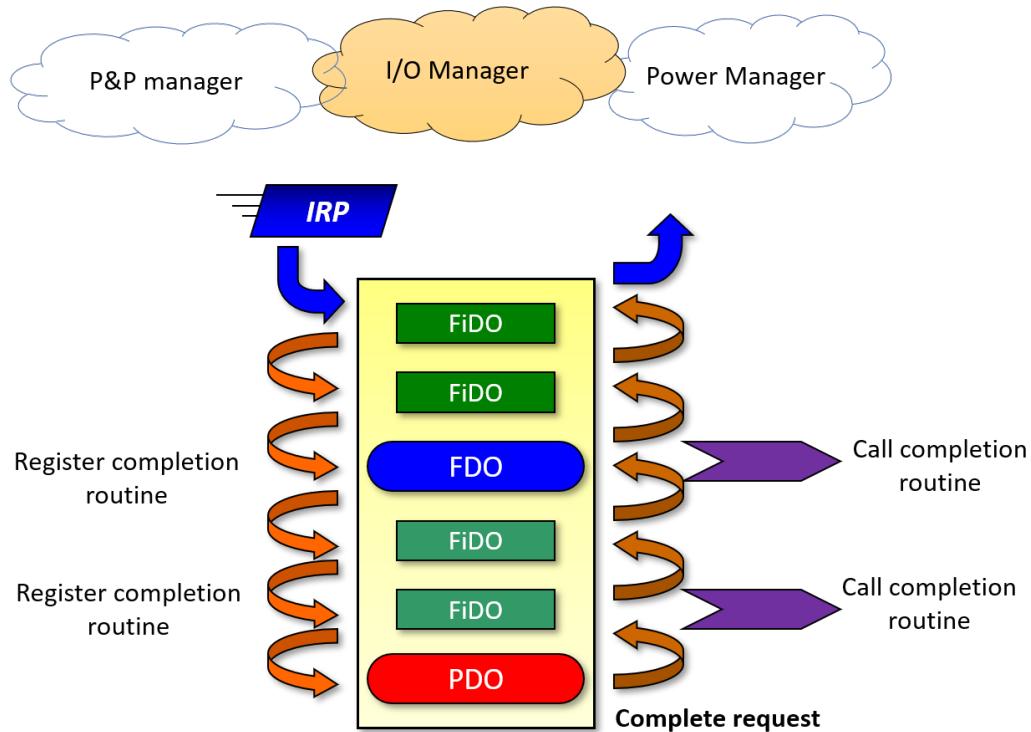


Figure 7-2: Layered devices

The different device objects that comprise the device node (devnode) layers are labeled according to their role in the devnode. These roles are relevant in a hardware-based devnode.

All the device objects in figure 7-2 are just `DEVICE_OBJECT` structures, each created by a different driver that is in charge of that layer. More generically, this kind of device node does not have to be related to hardware-based device drivers.

Here is a quick rundown of the meaning of the labels present in figure 7-2:

- **PDO (Physical Device Object)** - Despite the name, there is nothing “physical” about it. This device object is created by a bus driver - the driver that is in charge of the particular bus (e.g. PCI, USB, etc.). This device object represents the fact that there is some device in that slot on that bus.
- **FDO (Functional Device Object)** - This device object is created by the “real” driver; that is, the driver typically provided by the hardware’s vendor that understands the details of the device intimately.
- **FiDO (Filter Device Object)** - These are optional filter devices created by filter drivers.

The Plug & Play (P&P) manager, in this case, is responsible for loading the appropriate drivers, starting from the bottom. As an example, suppose the devnode in figure 7-2 represents a set of drivers that manage a PCI network card. The sequence of events leading to the creation of this devnode can be summarized as follows:

1. The PCI bus driver (*pci.sys*) recognizes the fact that there is something in that particular slot. It creates a PDO (`IoCreateDevice`) to represent this fact. The bus driver has no idea whether this a network card, video card or something else; it only knows there is something there and can extract basic information from its controller, such as the Vendor ID and Device ID of the device.
2. The PCI bus driver notifies the P&P manager that it has changes on its bus (calls `Io InvalidateDeviceRelations` with the `BusRelations` enumeration value).
3. The P&P manager requests a list of PDOs managed by the bus driver. It receives back a list of PDOs, in which this new PDO is included.
4. Now the P&P manager's job is to find and load the proper driver that should manage this new PDO. It issues a query to the bus driver to request the full hardware device ID.
5. With this hardware ID in hand, the P&P manager looks in the Registry at `HKLM\System\CurrentControlSet\Enum\PCI\HardwareID`. If the driver has been loaded before, it will be registered there, and the P&P manager will load it. Figure 7-3 shows an example hardware ID in the registry (NVIDIA display driver).
6. The driver loads and creates the FDO (another call to `IoCreateDevice`), but adds an additional call to `IoAttachDeviceToDeviceStack`, thus attaching itself over the previous layer (typically the PDO).

We'll see how to write filter drivers that take advantage of `IoAttachDeviceToDeviceStack` in chapter 13.

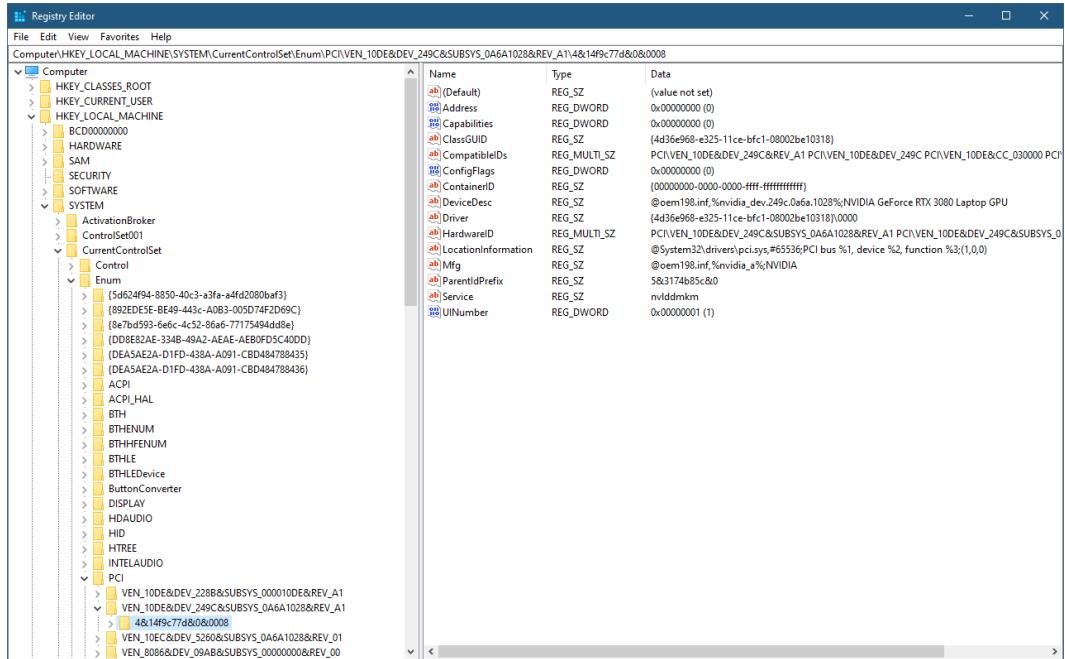


Figure 7-3: Hardware ID information



The value **Service** in figure 7-3 indirectly points to the actual driver at `HKLM\System\CurrentControlSet\Services\{ServiceName}` where all drivers must be registered.

The filter device objects are loaded as well, if they are registered correctly in the Registry. Lower filters (below the FDO) load in order, from the bottom. Each filter driver loaded creates its own device object and attaches it on top of the previous layer. Upper filters work the same way but are loaded after the FDO. All this means that with operational P&P devnodes, there are at least two layers - PDO and FDO, but there could be more if filters are involved. We'll look at basic filter development for hardware-based drivers in chapter 13.

Full discussion of Plug & Play and the exact way this kind of devnode is built is beyond the scope of this book. The previous description is incomplete and glances over some details, but it should give you the basic idea. Every devnode is built from the bottom up, regardless of whether it is related to hardware or not.

Lower filters are searched in two locations: the hardware ID key shown in figure 7-3 and in the corresponding class based on the **ClassGUID** value listed under `HKLM\System\CurrentControlSet\Control\Classes`. The value name itself is **LowerFilters** and is a multiple string value holding service names, pointing to the same **Services** key. Upper filters are searched in a similar manner, but the value name is

UpperFilters. Figure 7-4 shows the registry settings for the *DiskDrive* class, which has a lower filter and an upper filter.

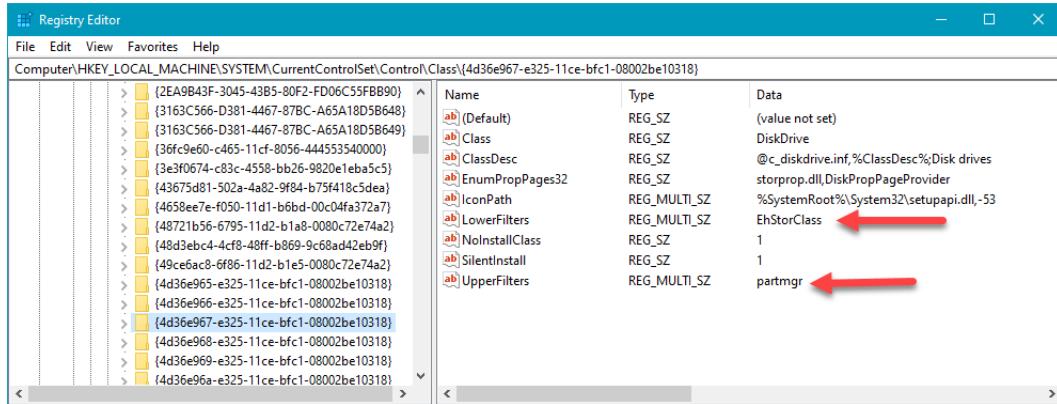


Figure 7-4: The *DiskDrive* class key

IRP Flow

Figure 7-2 shows an example devnode, whether related to hardware or not. An IRP is created by one of the managers in the Executive - for most of our drivers that is the I/O Manager.

The manager creates an IRP with its associated IO_STACK_LOCATIONS - six in the example in figure 7-2. The manager initializes the main IRP structure and the first I/O stack location only. Then it passes the IRP's pointer to the uppermost layer.

A driver receives the IRP in its appropriate dispatch routine. For example, if this is a Read IRP, then the driver will be called in its IRP_MJ_READ index of its MajorFunction array from its driver object. At this point, a driver has several options when dealing with IRP:

- Pass the request down - if the driver's device is not the last device in the devnode, the driver can pass the request along if it's not interesting for the driver. This is typically done by a filter driver that receives a request that it's not interested in, and in order not to hurt the functionality of the device (since the request is actually destined for a lower-layer device), the driver can pass it down. This must be done with two calls:
 - Call `IoSkipCurrentIrpStackLocation` to make sure the next device in line is going to see the same information given to this device - it should see the same I/O stack location.
 - Call `IoCallDriver` passing the lower device object (which the driver received at the time it called `IoAttachDeviceToDeviceStack`) and the IRP.

Before passing the request down, the driver must prepare the next I/O stack location with proper information. Since the I/O manager only initializes the first I/O stack location, it's the responsibility of each driver to initialize the next one. One way to do that is to call `IoCopyIrpStackLocationToNext` before calling `IoCallDriver`. This works, but is a bit wasteful if the driver just wants the lower layer to see the same information. Calling `IoSkipCurrentIrpStackLocation` is an optimization

which decrements the current I/O stack location pointer inside the IRP, which is later incremented by `IoCallDriver`, so the next layer sees the same `IO_STACK_LOCATION` this driver has seen. This decrement/increment dance is more efficient than making an actual copy.

- Handle the IRP fully - the driver receiving the IRP can just handle the IRP without propagating it down by eventually calling `IoCompleteRequest`. Any lower devices will never see the request.
- Do a combination of the above options - the driver can examine the IRP, do something (such as log the request), and then pass it down. Or it can make some changes to the next I/O stack location, and then pass the request down.
- Pass the request down (with or without changes) and be notified when the request completes by a lower layer device - Any layer (except the lowest one) can set up an I/O completion routine by calling `IoSetCompletionRoutine` before passing the request down. When one of the lower layers completes the request, the driver's completion routine will be called.
- Start some asynchronous IRP handling - the driver may want to handle the request, but if the request is lengthy (typical of a hardware driver, but also could be the case for a software driver), the driver may mark the IRP as pending by calling `IoMarkIrpPending` and return a `STATUS_PENDING` from its dispatch routine. Eventually, it will have to complete the IRP.

Once some layer calls `IoCompleteRequest`, the IRP turns around and starts “bubbling up” towards the originator of the IRP (typically one of the I/O System Managers). If completion routines have been registered, they will be invoked in reverse order of registration.

In most drivers in this book, layering will not be considered, since the driver is most likely the single device in its devnode. The driver will handle the request then and there or handle it asynchronously; it will not pass it down, as there is no device underneath.

We'll discuss other aspects of IRP handling in filter drivers, including completion routines, in chapter 13.

IRP and I/O Stack Location

Figure 7-5 shows some of the important fields in an IRP.

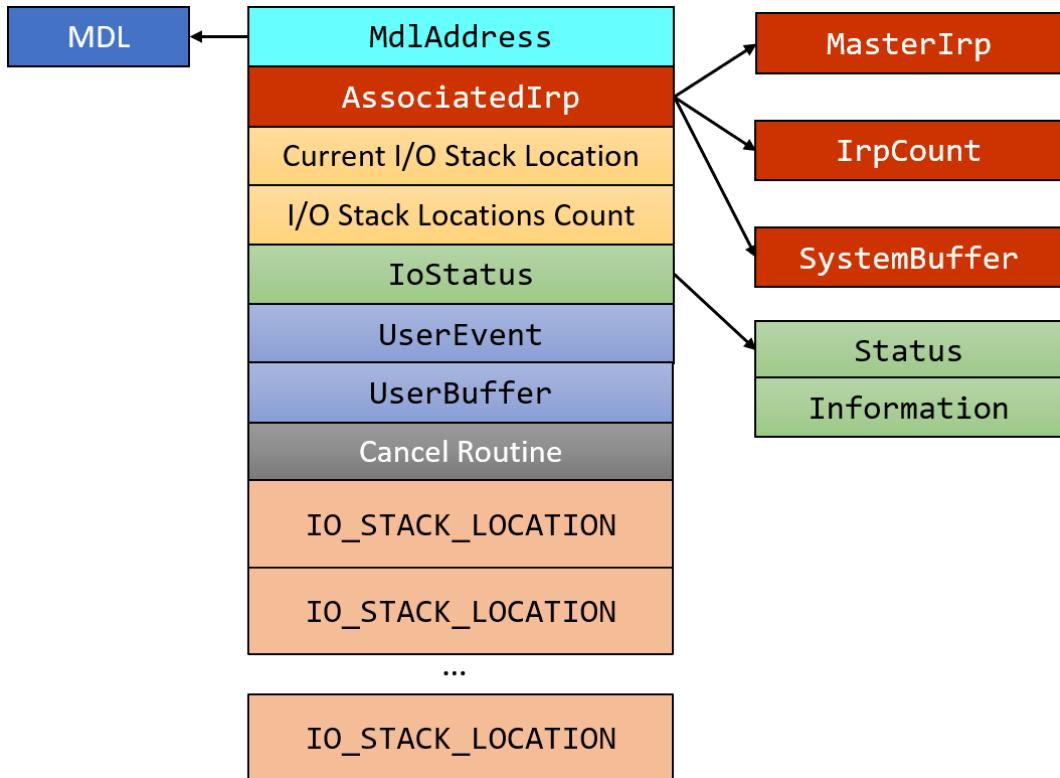


Figure 7-5: Important fields of the IRP structure

Here is a quick rundown of these fields:

- **IoStatus** - contains the **Status** (NT_STATUS) of the IRP and an **Information** field. The **Information** field is a polymorphic one, typed as ULONG_PTR (32 or 64-bit integer), but its meaning depends on the type of IRP. For example, for Read and Write IRPs, its meaning is the number of bytes transferred in the operation.
- **UserBuffer** - contains the raw buffer pointer to the user's buffer for relevant IRPs. Read and Write IRPs, for instance, store the user's buffer pointer in this field. In DeviceIoControl IRPs, this points to the output buffer provided in the request.
- **UserEvent** - this is a pointer to an event object (KEVENT) that was provided by a client if the call is asynchronous and such an event was supplied. From user mode, this event can be provided (with a HANDLE) in the OVERLAPPED structure that is mandatory for invoking I/O operations asynchronously.
- **AssociatedIrp** - this union holds three members, only one (at most) of which is valid:

* **SystemBuffer** - the most often used member. This points to a system-allocated non-paged pool buffer used for Buffered I/O operations. See the section “Buffered I/O” later in this chapter for the details.

* **MasterIrp** - A pointer to a “master” IRP, if this IRP is an *associated IRP*. This idea is supported by the I/O manager, where one IRP is a “master” that may have several “associated” IRPs. Once all

the associated IRPs complete, the master IRP is completed automatically. `MasterIrp` is valid for an associated IRP - it points to the master IRP.

* `IrpCount` - for the master IRP itself, this field indicates the number of associated IRPs associated with this master IRP.

Usage of master and associated IRPs is pretty rare. We will not be using this mechanism in this book.

- **Cancel Routine** - a pointer to a cancel routine that is invoked (if not `NULL`) if the driver is asked to cancel the IRP, such as with the user mode functions `CancelIo` and `CancelIoEx`. Software drivers rarely need cancellation routines, so we will not be using those in most examples.
- **MdlAddress** - points to an optional *Memory Descriptor List* (MDL). An MDL is a kernel data structure that knows how to describe a buffer in RAM. `MdlAddress` is used primarily with Direct I/O (see the section “Direct I/O” later in this chapter).

Every IRP is accompanied by one or more `IO_STACK_LOCATION`s. Figure 7-6 shows the important fields in an `IO_STACK_LOCATION`.

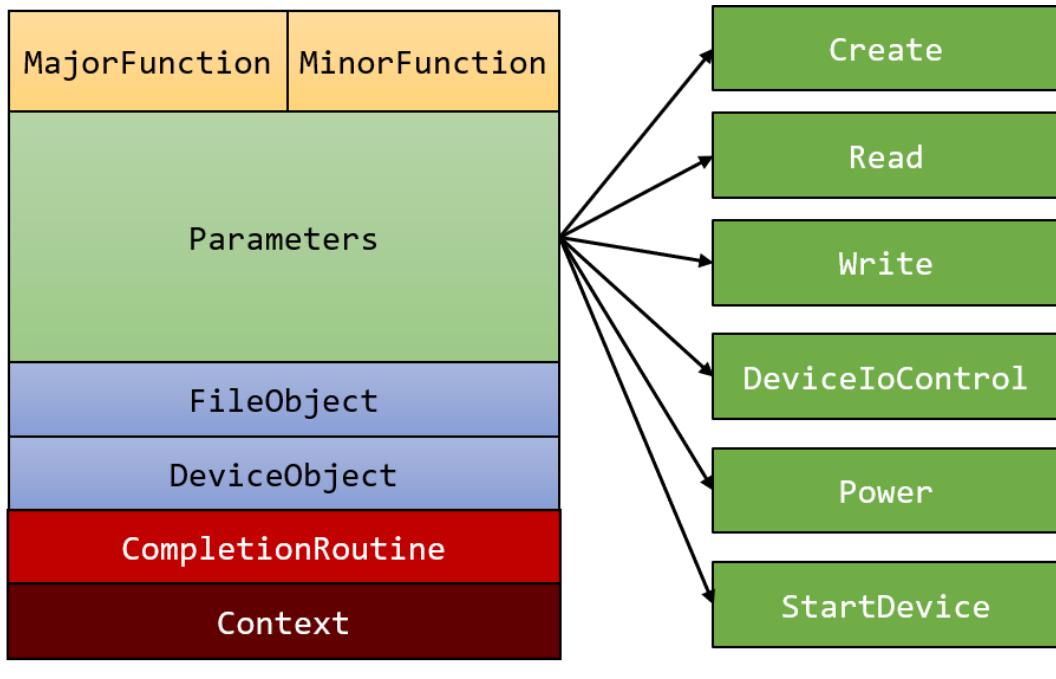


Figure 7-6: Important fields of the `IO_STACK_LOCATION` structure

Here's a rundown of the fields shown in figure 7-6:

- **MajorFunction** - this is the major function of the IRP (IRP_MJ_CREATE, IRP_MJ_READ, etc.). This field is sometimes useful if the driver points more than one major function code to the same handling routine. In that routine, the driver may want to distinguish between the major function codes using this field.
- **MinorFunction** - some IRP types have minor functions. These are IRP_MJ_PNP, IRP_MJ_POWER and IRP_MJ_SYSTEM_CONTROL (WMI). Typical code for these handlers has a switch statement based on the MinorFunction. We will not be using these types of IRPs in this book, except in the case of filter drivers for hardware-based devices, which we'll examine in some detail in chapter 13.
- **FileObject** - the FILE_OBJECT associated with this IRP. Not needed in most cases, but is available for dispatch routines that need it.
- **DeviceObject** - the device object associated with this IRP. Dispatch routines receive a pointer to this, so typically accessing this field is not required.
- **CompletionRoutine** - the completion routine that is set for the *previous* (upper) layer (set with IoSetCompletionRoutine), if any.
- **Context** - the argument to pass to the completion routine (if any).
- **Parameters** - this monster union contains multiple structures, each valid for a particular operation. For example, in a Read (IRP_MJ_READ) operation, the Parameters.Read structure field should be used to get more information about the Read operation.

The current I/O stack location obtained with `IoGetCurrentIrpStackLocation` hosts most of the parameters of the request in the `Parameters` union. It's up to the driver to access the correct structure, as we've already seen in chapter 4 and will see again in this and subsequent chapters.

Viewing IRP Information

While debugging or analyzing kernel dumps, a couple of commands may be useful for searching or examining IRPs.

The `!irpfind` command can be used to find IRPs - either all IRPs, or IRPs that meet certain criteria. Using `!irpfind` without any arguments searches the non-paged pool(s) for all IRPs. Check out the debugger documentation on how to specify specific criteria to limit the search. Here's an example of some output when searching for all IRPs:

```
1kd> !irpfind
Unable to get offset of nt!_MI_VISIBLE_STATE.SpecialPool
Unable to get value of nt!_MI_VISIBLE_STATE.SessionSpecialPool

Scanning large pool allocation table for tag 0x3f707249 (Irp?) (fffffbf0a8761000\
0 : fffffbf0a87910000)

          Irp      [ Thread ]      irpStack: (Mj,Mn)    DevObj      [Driver\
]      MDL Process
fffffbf0aa795ca30 [fffffbf0a7fcde080] irpStack: (c, 2)  fffffbf0a74d20050 [ \File\
System\Ntfs]
```

```
fffffbf0a9a8ef010 [fffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\  
System\Ntfs]  
fffffbf0a8e68ea20 [fffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\  
System\Ntfs]  
fffffbf0a90deb710 [fffffbf0a808a1080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\  
System\Ntfs]  
fffffbf0a99d1da90 [0000000000000000] Irp is complete (CurrentLocation 10 > Stack\  
Count 9)  
fffffbf0a74cec940 [0000000000000000] Irp is complete (CurrentLocation 8 > StackC\  
ount 7)  
fffffbf0aa0640a20 [fffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\  
System\Ntfs]  
fffffbf0a89acf4e0 [fffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\  
System\Ntfs]  
fffffbf0a89acfaf50 [fffffbf0a7fcde080] irpStack: ( c, 2) fffffbf0a74d20050 [ \File\  
System\Ntfs]  
(truncated)
```

Faced with a specific IRP, the command !irp examines the IRP, providing a nice overview of its data. As always, the dt command can be used with the nt!_IRP type to look at the entire IRP structure. Here's an example of one IRP viewed with !irp:

```
kd> !irp fffffbf0a8bbada20  
Irp is active with 13 stacks 12 is current (= 0xfffffbf0a8bbade08)  
No Mdl: No System Buffer: Thread fffffbf0a7fcde080: Irp stack trace.  
    cmd   flg cl Device   File      Completion-Context  
[N/A(0), N/A(0)]  
    0 0 00000000 00000000 00000000-00000000  
  
    Args: 00000000 00000000 00000000 00000000  
[N/A(0), N/A(0)]  
    0 0 00000000 00000000 00000000-00000000  
  
(truncated)  
  
    Args: 00000000 00000000 00000000 00000000  
[N/A(0), N/A(0)]  
    0 0 00000000 00000000 00000000-00000000  
  
    Args: 00000000 00000000 00000000 00000000  
>[IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]  
    0 e1 fffffbf0a74d20050 fffffbf0a7f52f790 fffff8015c0b50a0-fffffbf0a91d99010 Su\  
ccess Error Cancel pending
```

```
\FileSystem\Ntfs
  Args: 00004000 00000051 00000000 00000000
[IRP_MJ_DIRECTORY_CONTROL(c), N/A(2)]
  0 0 fffffbf0a60e83dc0 fffffbf0a7f52f790 00000000-00000000
    \FileSystem\FltMgr
  Args: 00004000 00000051 00000000 00000000
```

The !irp commands lists the I/O stack locations and the information stored in them. The current I/O stack location is marked with a > symbol (see the IRP_MJ_DIRECTORY_CONTROL line above).

The details for each IO_STACK_LOCATION are as follows (in order):

- first line:
 - Major function code (e.g. IRP_MJ_DEVICE_CONTROL).
 - Minor function code.
- second line:
 - Flags (mostly unimportant)
 - Control flags
 - Device object pointer
 - File object pointer
 - Completion routine (if any)
 - Completion context (for the completion routine)
 - Success, Error, Cancel indicate the IRP completion cases where the completion routine would be invoked
 - “pending” if the IRP was marked as pending (SL_PENDING_RETURNED flag is set in the Control flags)
- Driver name for that layer
- “Args” line:
 - The value of Parameters.Others.Argument1 in the I/O stack location. Essentially the first pointer-size member in the Parameters union.
 - The value of Parameters.Others.Argument2 in the I/O stack location (the second pointer-size member in the Parameters union)
 - Device I/O control code (if IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL). It’s shown as a DML link that invokes the !ioct1decode command to decode the control code (more on device I/O control codes later in this chapter). For other major function codes, shows the third pointer-size member (Parameters.Others.Argument3)
 - The forth pointer-size member (Parameters.Others.Argument4)

The !irp command accepts an optional *details* argument. The default is zero, which provides the output described above (considered a summary). Specifying 1 provides additional information in a concrete form. Here is an example for an IRP targeted towards the console driver (you can locate those easily by looking for *cmd.exe* processes):

```

1kd> !irp fffffdb899e82a6f0 1
Irp is active with 2 stacks 1 is current (= 0xfffffdb899e82a7c0)
No Mdl: System buffer=fffffdb89c1c84ac0: Thread fffffdb89b6efa080: Irp stack tr\
ace.
Flags = 00060030
ThreadListEntry.Flink = fffffdb89b6efa530
ThreadListEntry.Blink = fffffdb89b6efa530
IoStatus.Status = 00000000
IoStatus.Information = 00000000
RequestorMode = 00000001
Cancel = 00
CancelIrql = 0
ApcEnvironment = 00
UserIosb = 73d598f420
UserEvent = 00000000
Overlay.AsynchronousParameters.UserApcRoutine = 00000000
Overlay.AsynchronousParameters.UserApcContext = 00000000
Overlay.AllocationSize = 00000000 - 00000000
CancelRoutine = fffff8026f481730
UserBuffer = 00000000
&Tail.Overlay.DeviceQueueEntry = fffffdb899e82a768
Tail.Overlay.Thread = fffffdb89b6efa080
Tail.Overlay.AuxiliaryBuffer = 00000000
Tail.Overlay.ListEntry.Flink = ffff8006d16437b8
Tail.Overlay.ListEntry.Blink = ffff8006d16437b8
Tail.Overlay.CurrentStackLocation = fffffdb899e82a7c0
Tail.Overlay.OriginalFileObject = fffffdb89c1c0a240
Tail.Apc = 8b8b7240
Tail.CompletionKey = 15f8b8b7240
    cmd   flg cl Device     File      Completion-Context
>[N/A(f), N/A(7)]
        0  1 00000000 00000000 00000000-00000000      pending

        Args: ffff8006d1643790 15f8d92c340 0xa0e666b0 fffffdb899e7a53c0
[IRP_MJ_DEVICE_CONTROL(e), N/A(0)]
        5  0 fffffdb89846f9e10 fffffdb89c1c0a240 00000000-00000000
        \Driver\condrv
        Args: 00000000 00000060 0x500016 00000000

```

Additionally, specifying detail value of 4 shows *Driver Verifier* information related to the IRP (if the driver handling this IRP is under the verifier's microscope). *Driver Verifier* will be discussed in chapter 13.

Dispatch Routines

In chapter 4, we have seen an important aspect of `DriverEntry` - setting up dispatch routines. These are the functions connected with major function codes. The `MajorFunction` field in `DRIVER_OBJECT` is the array of function pointers index by the major function code.

All dispatch routines have the same prototype, repeated here for convenience using the `DRIVER_DISPATCH` typedef from the WDK (somewhat simplified for clarity):

```
typedef NTSTATUS DRIVER_DISPATCH (
    _In_     PDEVICE_OBJECT DeviceObject,
    _Inout_   PIRP Irp);
```

The relevant dispatch routine (based on the major function code) is the first routine in a driver that sees the request. Normally, it's called in the requesting thread context, i.e. the thread that called the relevant API (e.g. `ReadFile`) in IRQL `PASSIVE_LEVEL` (0). However, it's possible that a filter driver sitting on top of this device sent the request down in a different context - it may be some other thread unrelated to the original requestor and even in higher IRQL, such as `DISPATCH_LEVEL` (2). Robust drivers need to be ready to deal with this kind of situation, even though for software drivers this "inconvenient" context is rare. We'll discuss the way to properly deal with this situation in the section "Accessing User Buffers", later in this chapter.

The first thing a typical dispatch routine does is check for errors. For example, read and write operations contain buffers - do these buffers have appropriate size? For `DeviceIoControl`, there is a control code in addition to potentially two buffers. The driver needs to make sure the control code is something it recognizes. If any error is identified, the IRP is typically completed immediately with an appropriate status.

If all checks turn up ok, then the driver can deal with performing the requested operation.

Here is the list of the most common dispatch routines for a software driver:

- `IRP_MJ_CREATE` - corresponds to a `CreateFile` call from user mode or `ZwCreateFile` in kernel mode. This major function is essentially mandatory, otherwise no client will be able to open a handle to a device controlled by this driver. Most drivers just complete the IRP with a success status.
- `IRP_MJ_CLOSE` - the opposite of `IRP_MJ_CREATE`. Called by `CloseHandle` from user mode or `ZwClose` from kernel mode when the last handle to the file object is about to be closed. Most drivers just complete the request successfully, but if something meaningful was done in `IRP_MJ_CREATE`, this is where it should be undone.
- `IRP_MJ_READ` - corresponds to a read operation, typically invoked from user mode by `ReadFile` or kernel mode with `ZwReadFile`.
- `IRP_MJ_WRITE` - corresponds to a write operation, typically invoked from user mode by `Writefile` or kernel mode with `ZwWriteFile`.
- `IRP_MJ_DEVICE_CONTROL` - corresponds to the `DeviceIoControl` call from user mode or `ZwDeviceIoControlFile` from kernel mode (there are other APIs in the kernel that can generate `IRP_MJ_DEVICE_CONTROL` IRPs).
- `IRP_MJ_INTERNAL_DEVICE_CONTROL` - similar to `IRP_MJ_DEVICE_CONTROL`, but only available to kernel callers.

Completing a Request

Once a driver decides to handle an IRP (meaning it's not passing down to another driver), it must eventually complete it. Otherwise, we have a leak on our hands - the requesting thread cannot really terminate and by extension, its containing process will linger on as well, resulting in a “zombie process”.

Completing a request means calling `IoCompleteRequest` after setting the request status and extra information. If the completion is done in the dispatch routine itself (a common case for software drivers), the routine must return the same status that was placed in the IRP.

The following code snippet shows how to complete a request in a dispatch routine:

```
NTSTATUS MyDispatchRoutine(PDEVICE_OBJECT, PIRP Irp) {
    //...
    Irp->IoStatus.Status = STATUS_XXX;
    Irp->IoStatus.Information = bytes;      // depends on request type
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_XXX;
}
```



Since the dispatch routine must return the same status as was placed in the IRP, it's tempting to write the last statement like so: `return Irp->IoStatus.Status;` This, however, will likely result in a system crash. Can you guess why?

After the IRP is completed, touching any of its members is a bad idea. The IRP has probably already been freed and you're touching deallocated memory. It can actually be worse, since another IRP may have been allocated in its place (this is common), and so the code may return the status of some random IRP.

The `Information` field should be zero in case of an error (a failure status). Its exact meaning for a successful operation depends on the type of IRP.

The `IoCompleteRequest` API accepts two arguments: the IRP itself and an optional value to temporarily boost the original thread's priority (the thread that initiated the request in the first place). In most cases, for software drivers, the thread in question is the executing thread, so a thread boost is inappropriate. The value `IO_NO_INCREMENT` is defined as zero, so no increment in the above code snippet.

However, the driver may choose to give the thread a boost, regardless of whether it's the calling thread or not. In this case, the thread's priority jumps with the given boost, and then it's allowed to execute one quantum with that new priority before the priority decreases by one, it can then get another quantum with the reduced priority, and so on, until its priority returns to its original level. Figure 7-7 illustrates this scenario.

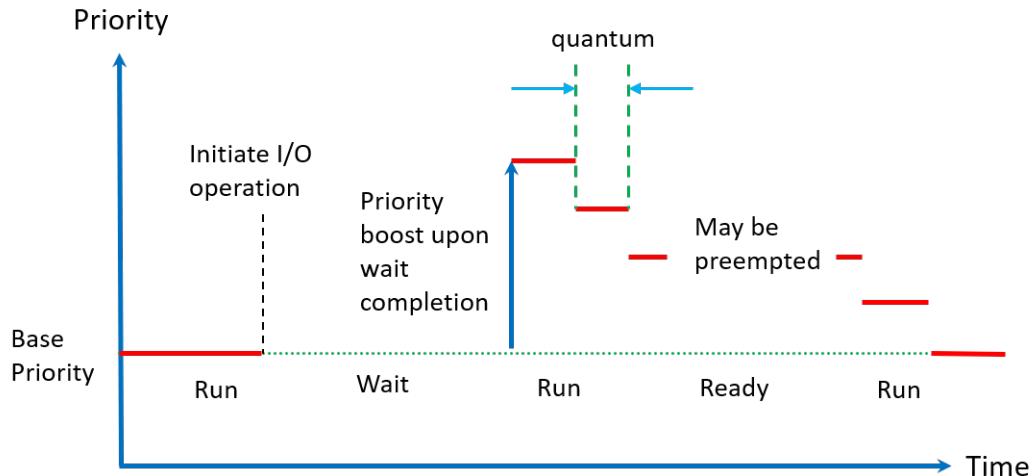


Figure 7-7: Thread priority boost and decay

i The thread's priority after the boost can never go above 15. If it's supposed to, it will be 15. If the original thread's priority is above 15 already, boosting has no effect.

Accessing User Buffers

A given dispatch routine is the first to see the IRP. Some dispatch routines, mainly `IRP_MJ_READ`, `IRP_MJ_WRITE` and `IRP_MJ_DEVICE_CONTROL` accept buffers provided by a client - in most cases from user mode. Typically, a dispatch routine is called in IRQL 0 and in the requesting thread context, which means the buffers pointers provided by user mode are trivially accessible: the IRQL is 0, so page faults are handled normally, and the thread is the requestor, so the pointers are valid in this process context.

However, there could be issues. As we've seen in chapter 6, even in this convenient context (requesting thread and IRQL 0), it's possible for another thread in the client's process to free the passed-in buffer(s), before the driver gets a chance to examine them, and so cause an access violation. The solution we've used in chapter 6 is to use a `__try / __except` block to handle any access violation by returning failure to the client.

In some cases, even that is not enough. For example, if we have some code running at IRQL 2 (such as a DPC running as a result of timer expiration), we cannot safely access the user's buffers in this context. In general, there are two potential issues here:

- IRQL of the calling CPU is 2 (or higher), meaning no page fault handling can occur.
- The thread calling the driver may be some arbitrary thread, and not the original requestor. This means that the buffer pointer(s) provided are meaningless, since the wrong process address space is accessible.

Using exception handling in such a case will not work as expected, because we'll be accessing some memory location that is essentially invalid in this random process context. Even if the access succeeds (because that memory happens to be allocated in this random process and is resident in RAM), you'll be accessing random memory, and certainly not the original buffer provided to the client.

All this means that there must be some good way to access the original user's buffer in such an inconvenient context. In fact, there are two such ways provided by the I/O manager for this purpose, called *Buffered I/O* and *Direct I/O*. In the next two sections, we'll see what each of these schemes mean and how to use them.



Some data structures are always safe to access, since they are allocated from non-paged pool (and are in system space). Common examples are device objects (created with `IoCreateDevice`) and IRPs.

Buffered I/O

Buffered I/O is the simplest of the two ways. To get support for Buffered I/O for Read and Write operations, a flag must be set on the device object like so:

```
DeviceObject->Flags |= DO_BUFFERED_IO; // DO = Device Object
```

`DeviceObject` is the allocated pointer from a previous call to `IoCreateDevice` (or `IoCreateDeviceSecure`).

For `IRP_MJ_DEVICE_CONTROL` buffers, see the section “User Buffers for `IRP_MJ_DEVICE_CONTROL`” later in this chapter.

Here are the steps taken by the I/O Manager and the driver when a read or write request arrives:

1. The I/O Manager allocates a buffer from non-paged pool with the same size as the user's buffer. It stores the pointer to this new buffer in the `AssociatedIrp->SystemBuffer` member of the IRP. (The buffer size can be found in the current I/O stack location's `Parameters.Read.Length` or `Parameters.Write.Length`.)
2. For a write request, the I/O Manager copies the user's buffer to the system buffer.
3. Only now the driver's dispatch routine is called. The driver can use the system buffer pointer directly without any checks, because the buffer is in system space (its address is absolute - the same from any process context), and in any IRQL, because the buffer is allocated from non-paged pool, so it cannot be paged out.
4. Once the driver completes the IRP (`IoCompleteRequest`), the I/O manager (for read requests) copies the system buffer back to the user's buffer (the size of the copy is determined by the `IoStatus.Information` field in the IRP set by the driver).
5. Finally, the I/O Manager frees the system buffer.



You may be wondering how does the I/O Manager copy back the system buffer to the original user's buffer from `IoCompleteRequest`. This function can be called from any thread, in IRQL ≤ 2 . The way it's done is by queuing a *special kernel APC* to the thread that requested the operation. Once this thread is scheduled for execution, the first thing it does is run this APC which performs the actual copying. The requesting thread is obviously in the correct process context, and the IRQL is 1, so page faults can be handled normally.

Figures 7-8a to 7-8e illustrate the steps taken with Buffered I/O.

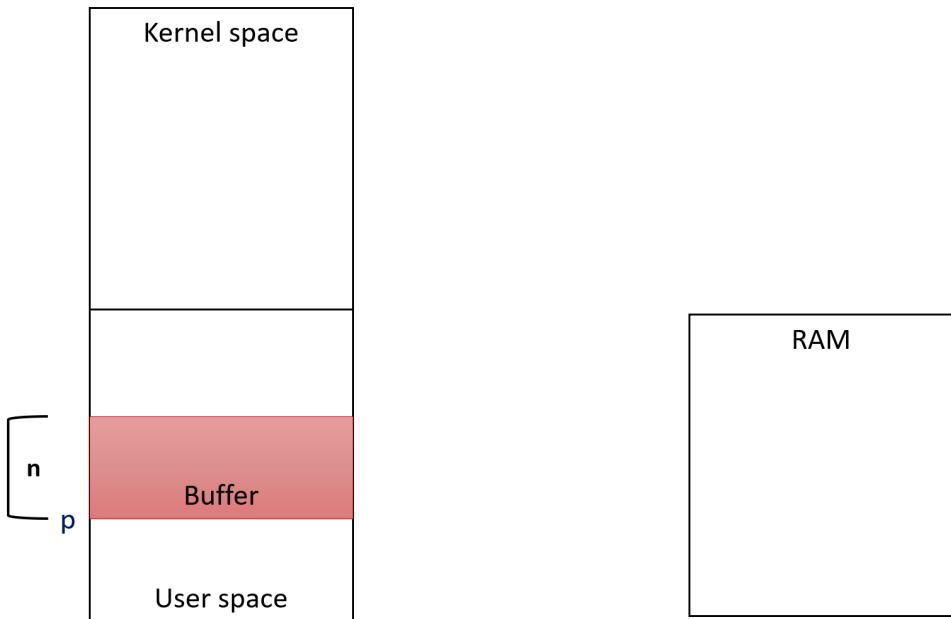


Figure 7-8a: Buffered I/O: initial state

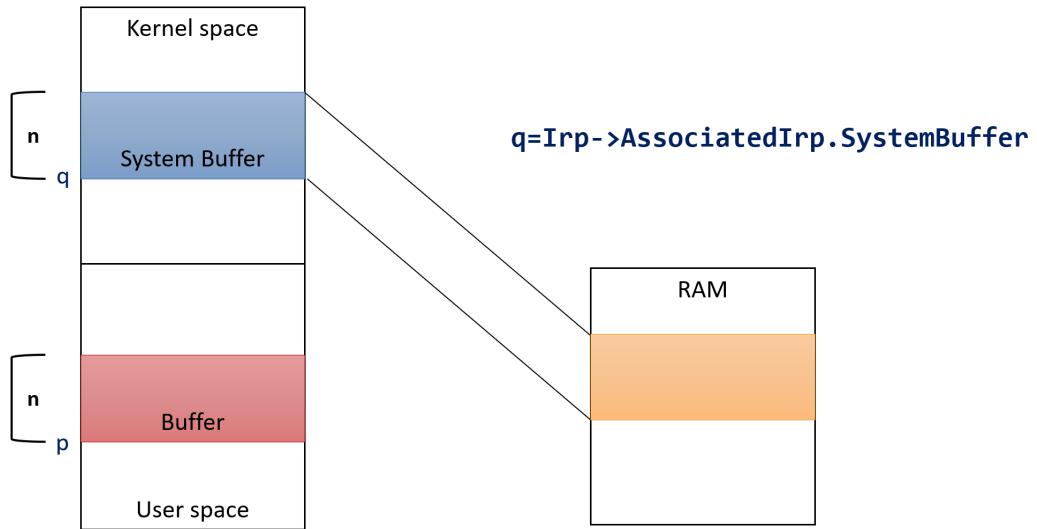


Figure 7-8b: Buffered I/O: system buffer allocated

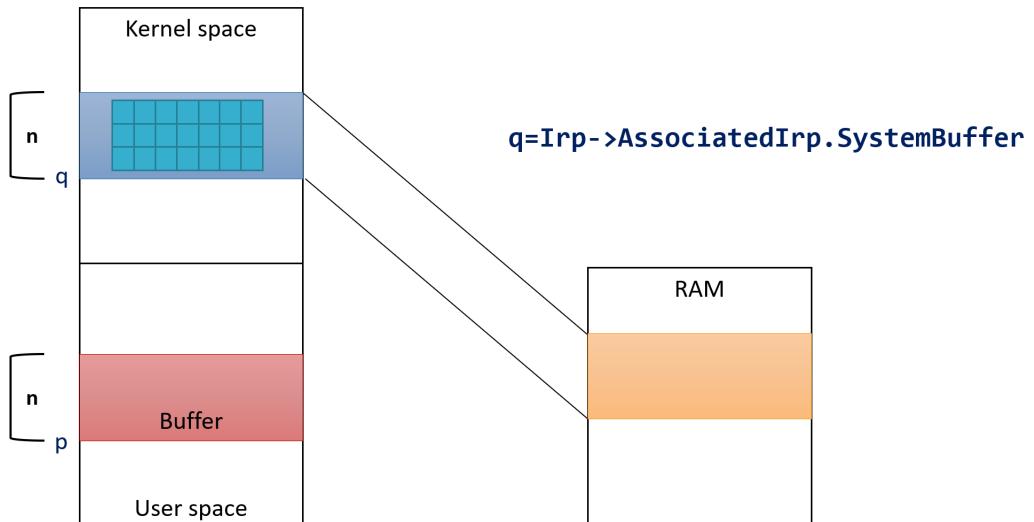


Figure 7-8c: Buffered I/O: driver accesses system buffer

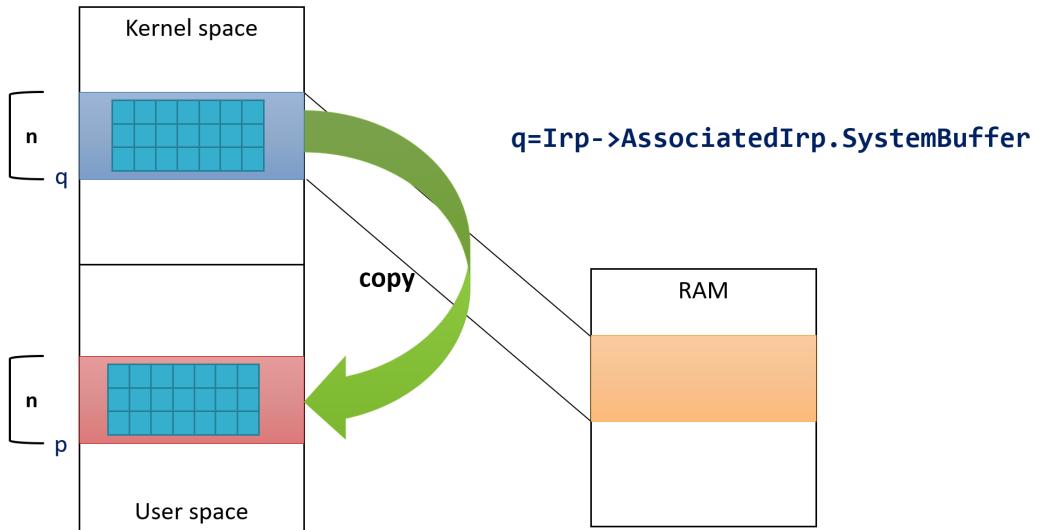


Figure 7-8d: Buffered I/O: on IRP completion, I/O manager copies buffer back (for read)

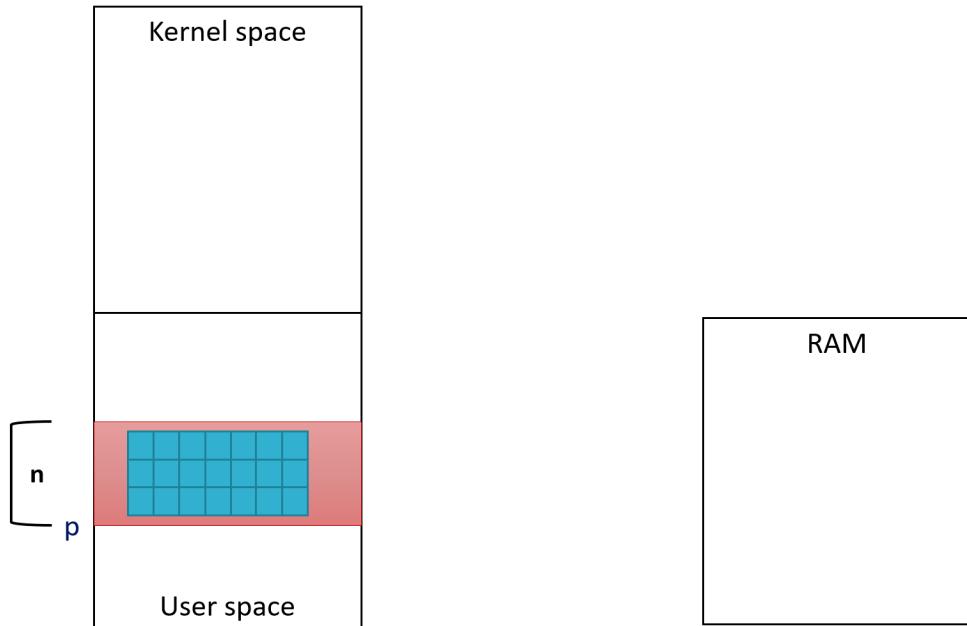


Figure 7-8e: Buffered I/O: final state - I/O manager frees system buffer

Buffered I/O has the following characteristics:

- Easy to use - just specify the flag in the device object, and everything else is taken care of by

the I/O Manager.

- It always involves a copy - which means it's best used for small buffers (typically up to one page). Large buffers may be expensive to copy. In this case, the other option, Direct I/O, should be used instead.

Direct I/O

The purpose of Direct I/O is to allow access to a user's buffer in any IRQL and any thread but without any copying going around.

For read and write requests, selecting Direct I/O is done with a different flag of the device object:

```
DeviceObject->Flags |= DO_DIRECT_IO;
```

As with Buffered I/O, this selection only affects read and write requests. For `DeviceIoControl` see the next section.

Here are the steps involved in handling Direct I/O:

1. The I/O Manager first makes sure the user's buffer is valid and then pages it into physical memory (if it wasn't already there).
2. It then locks the buffer in memory, so it cannot be paged out until further notice. This solves one of the issues with buffer access - page faults cannot happen, so accessing the buffer in any IRQL is safe.
3. The I/O Manager builds a *Memory Descriptor List* (MDL), a data structure that describes a buffer in physical memory. The address of this data structure is stored in the `MdlAddress` field of the IRP.
4. At this point, the driver gets the call to its dispatch routine. The user's buffer, although locked in RAM, cannot be accessed from an arbitrary thread just yet. When the driver requires access to the buffer, it must call a function that maps the same user buffer to a system address, which by definition is valid in any process context. So essentially, we get two mappings to the same memory buffer. One is from the original address (valid only in the context of the requestor process) and the other in system space, which is always valid. The API to call is `MmGetSystemAddressForMdlSafe`, passing the MDL built by the I/O Manager. The return value is the system address.
5. Once the driver completes the request, the I/O Manager removes the second mapping (to system space), frees the MDL, and unlocks the user's buffer, so it can be paged normally just like any other user-mode memory.

The MDL is in actually a list of MDL structures, each one describing a piece of the buffer that is contiguous in physical memory. Remember, that a buffer that is contiguous in virtual memory is not necessary contiguous in physical memory (the smallest piece is a page size). In most cases, we don't need to care about this detail. One case where this matters is in *Direct Memory Access* (DMA) operations. Fortunately, this is in the realm of hardware-based drivers.

Figures 7-9a to 7-9f illustrate the steps taken with Direct I/O.

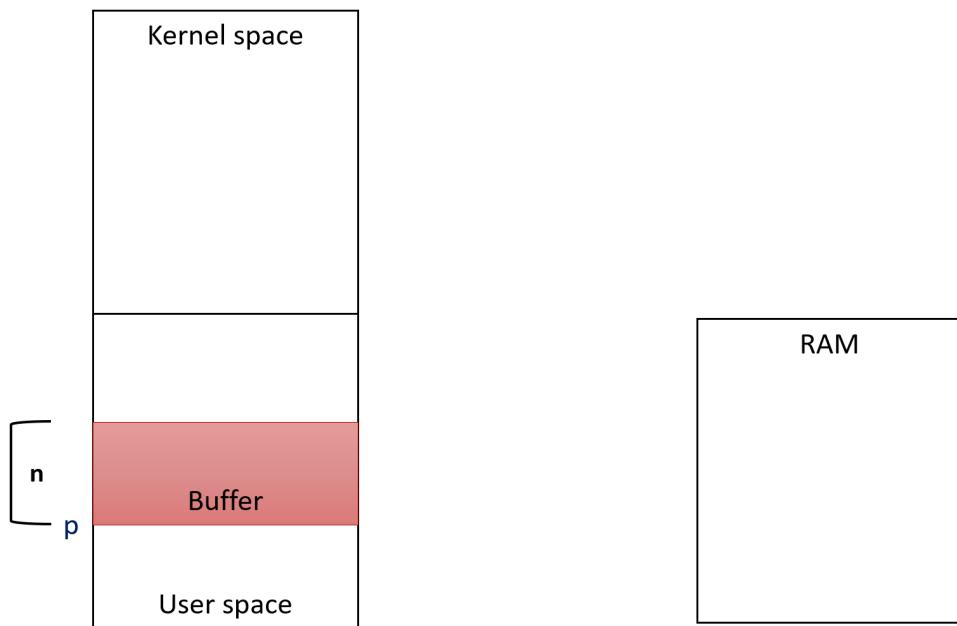


Figure 7-9a: Direct I/O: initial state

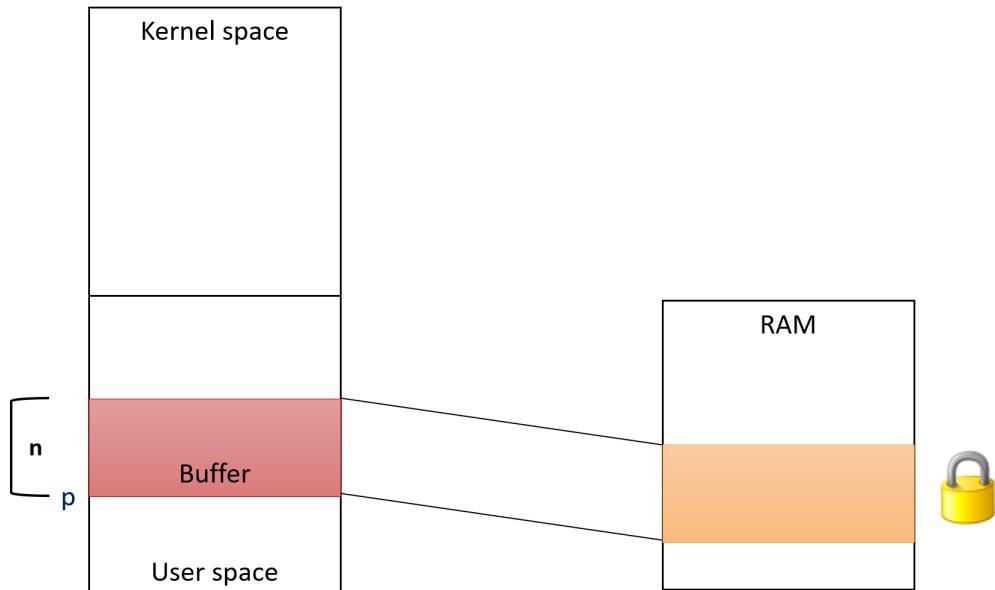


Figure 7-9b: Direct I/O: I/O manager faults buffer's pages to RAM and locks them

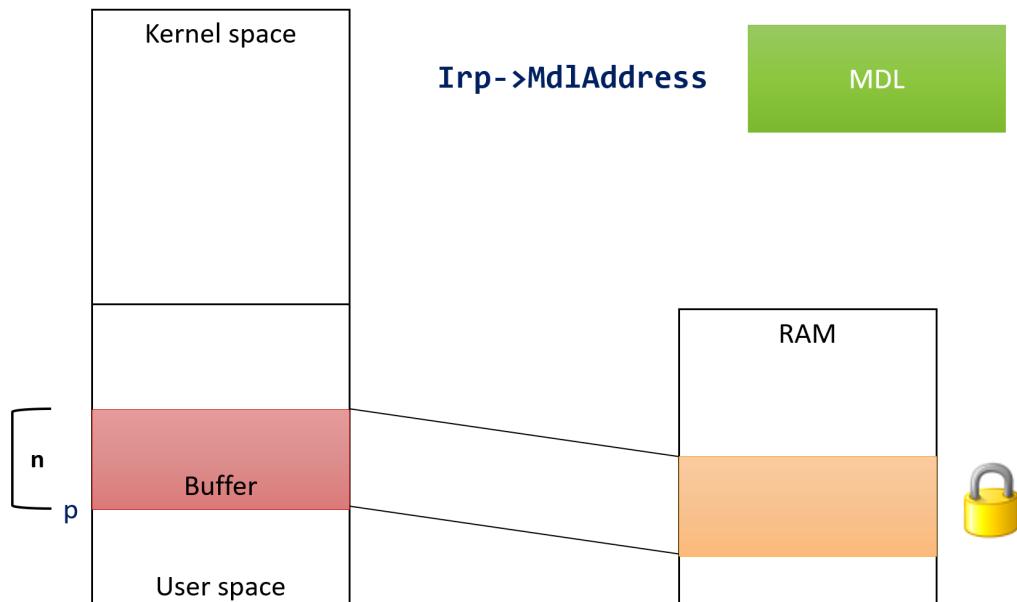


Figure 7-9c: Direct I/O: the MDL describing the buffer is stored in the IRP

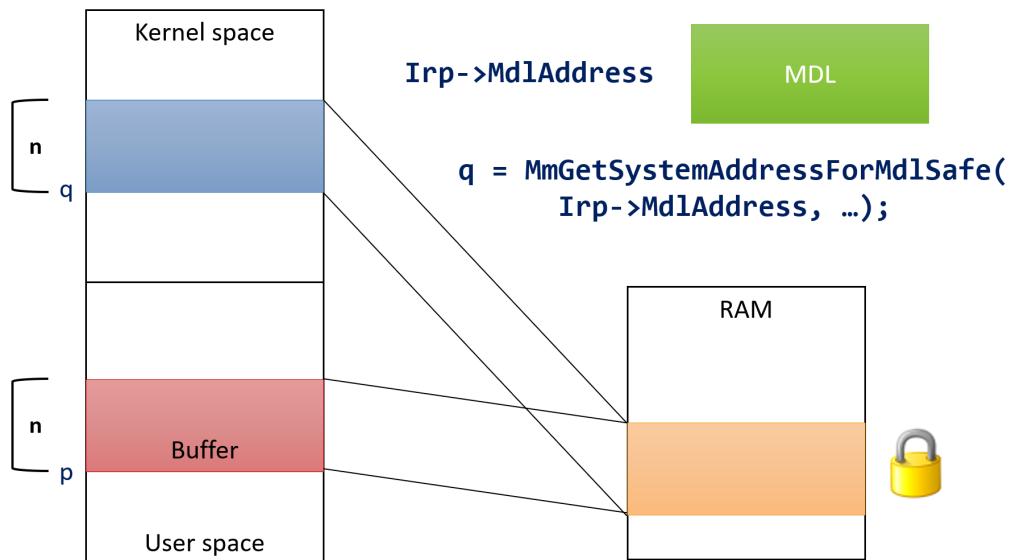


Figure 7-9d: Direct I/O: the driver double-maps the buffer to a system address

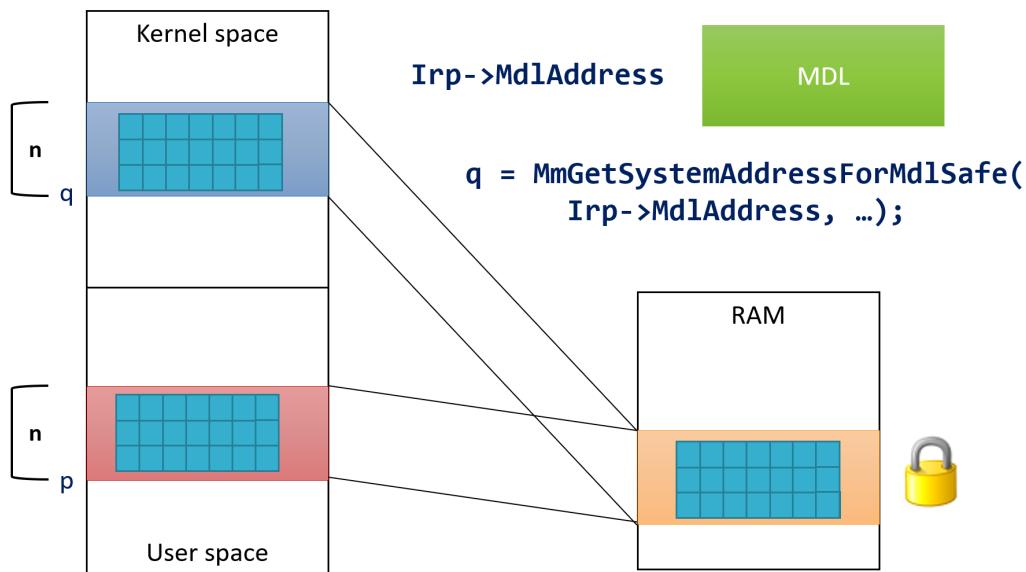


Figure 7-9e: Direct I/O: the driver accesses the buffer using the system address

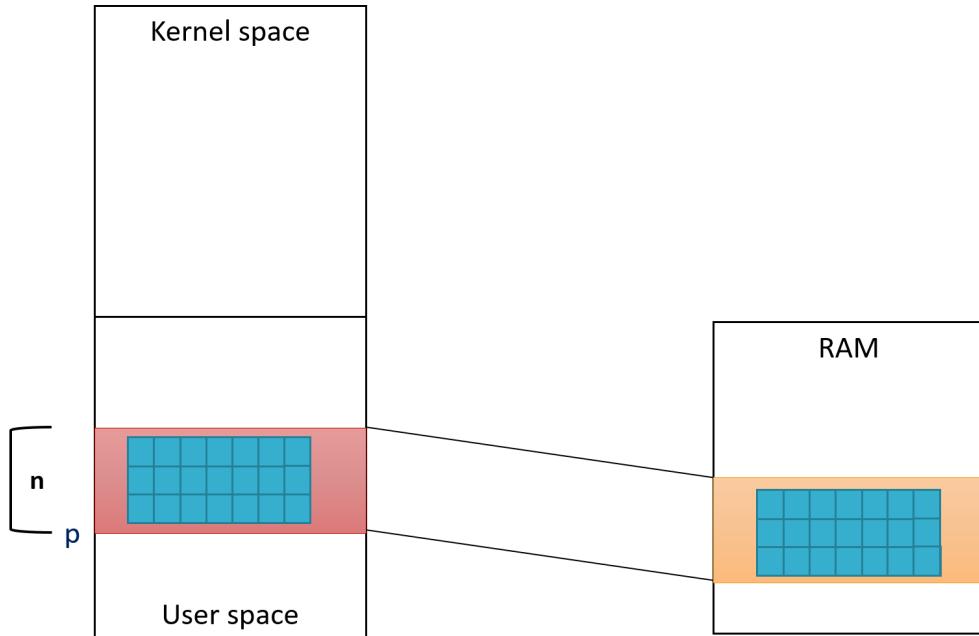


Figure 7-9f: Direct I/O: when the IRP is completed, the I/O manager frees the mapping, the MDL and unlocks the buffer

Notice there is no copying at all. The driver just reads/writes to the user's buffer directly, using the system address.



Locking the user's buffer is done with the `MmProbeAndLockPages` API, fully documented in the WDK. Unlocking is done with `MmUnlockPages`, also documented. This means a driver can use these routines outside the narrow context of Direct I/O.



Calling `MmGetSystemAddressForMdlSafe` can be done multiple times. The MDL stores a flag indicating whether the system mapping has already been done. If so, it just returns the existing pointer.

Here is the prototype of `MmGetSystemAddressForMdlSafe`:

```
PVOID MmGetSystemAddressForMdlSafe (
    _Inout_ PMDL Mdl,
    _In_     ULONG Priority);
```

The function is implemented inline within the `wdm.h` header by calling the more generic `MmMapLockedPagesSpecifyCache` function:

```
PVOID MmGetSystemAddressForMd1Safe(PMDL Md1, ULONG Priority) {
    if (Md1->Md1Flags & (MDL_MAPPED_TO_SYSTEM_VA|MDL_SOURCE_IS_NONPAGED_POOL)) {
        return Md1->MappedSystemVa;
    } else {
        return MmMapLockedPagesSpecifyCache(Md1, KernelMode, MmCached,
                                            NULL, FALSE, Priority);
    }
}
```

`MmGetSystemAddressForMd1Safe` accepts the MDL and a page priority (`MM_PAGE_PRIORITY` enumeration). Most drivers specify `NormalPagePriority`, but there is also `LowPagePriority` and `HighPagePriority`. This priority gives a hint to the system of the importance of the mapping in low memory conditions. Check the WDK documentation for more information.

If `MmGetSystemAddressForMd1Safe` fails, it returns `NULL`. This means the system is out of system page tables or very low on system page tables (depends on the priority argument above). This should be a rare occurrence, but still can happen in low memory conditions. A driver must check for this; if `NULL` is returned, the driver should complete the IRP with the status `STATUS_INSUFFICIENT_RESOURCES`.



There is a similar function, called `MmGetSystemAddressForMd1`, which if it fails, crashes the system. Do not use this function.

You may be wondering why doesn't the I/O manager call `MmGetSystemAddressForMd1Safe` automatically, which would be simple enough to do. This is an optimization, where the driver may not need to call this function at all if there is any error in the request, so that the mapping doesn't have to occur at all.

Drivers that don't set either of the flags `DO_BUFFERED_IO` nor `DO_DIRECT_IO` in the device object flags implicitly use *Neither I/O*, which simply means the driver doesn't get any special help from the I/O manager, and it's up to the driver to deal with the user's buffer.

User Buffers for `IRP_MJ_DEVICE_CONTROL`

The last two sections discussed Buffered I/O and Direct I/O as they pertain to read and write requests. For `IRP_MJ_DEVICE_CONTROL` (and `IRP_MJ_INTERNAL_DEVICE_CONTROL`), the buffering access method is supplied on a control code basis. Here is the prototype of the user-mode API `DeviceIoControl` (it's similar with the kernel function `ZwDeviceIoControlFile`):

```
BOOL DeviceIoControl(
    HANDLE hDevice,           // handle to device or file
    DWORD dwIoControlCode,    // IOCTL code (see <winiocrtl.h>)
    PVOID lpInBuffer,         // input buffer
    DWORD nInBufferSize,      // size of input buffer
    PVOID lpOutBuffer,        // output buffer
    DWORD nOutBufferSize,     // size of output buffer
    PDWORD lpdwBytesReturned, // # of bytes actually returned
    LPOVERLAPPED lpOverlapped); // for async. operation
```

There are three important parameters here: the I/O control code, and optional two buffers designated “input” and “output”. As it turns out, the way these buffers are accessed depends on the control code, which is very convenient, because different requests may have different requirements related to accessing the user’s buffer(s).

The control code defined by a driver must be built with the `CTL_CODE` macro, defined in the WDK and user-mode headers, defined like so:

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method))
```

The first parameter, `DeviceType` can be one of a set of constants defined by Microsoft for various known device types (such as `FILE_DEVICE_DISK` and `FILE_DEVICE_KEYBOARD`). For custom devices (like the ones we are writing), it can be any value, but the documentation states that the minimum value for custom codes should be `0x8000`.

The second parameter, `Function`, is a running index, that should be different between multiple control codes defined by the same driver. If all other components of the macro are same (possible), at least the `Function` would be a differentiating factor. Similarly to device type, the official documentation states that custom devices should use values starting from `0x800`.

The third parameter (`Method`) is the key to selecting the buffering method for accessing the input and output buffers provided with `DeviceIoControl`. Here are the options:

- `METHOD_NEITHER` - this value means no help is required of the I/O manager, so the driver is left dealing with the buffers on its own. This could be useful, for instance, if the particular code does not require any buffer - the control code itself is all the information needed - it’s best to let the I/O manager know that it does not need to do any additional work.
 - In this case, the pointer to the user’s input buffer is stored in the current I/O stack location’s `Parameters.DeviceIoControl.Type3InputBuffer` field, and the output buffer is stored in the IRP’s `UserBuffer` field.
- `METHOD_BUFFERED` - this value indicates Buffered I/O for both the input and output buffer. When the request starts, the I/O manager allocates the system buffer from non-paged pool with the size that is the maximum of the lengths of the input and output buffers. It then copies the input buffer to the system buffer. Only now the `IRP_MJ_DEVICE_CONTROL` dispatch routine is invoked. When the request completes, the I/O manager copies the number of bytes indicated with the `IoStatus.Information` field in the IRP to the user’s output buffer.

- The system buffer pointer is at the usual location: `AssociatedIrp.SystemBuffer` inside the IRP structure.
- `METHOD_IN_DIRECT` and `METHOD_OUT_DIRECT` - contrary to intuition, both of these values mean the same thing as far as buffering methods are concerned: the input buffer uses Buffered I/O and the output buffer uses Direct I/O. The only difference between these two values is whether the output buffer can be read (`METHOD_IN_DIRECT`) or written (`METHOD_OUT_DIRECT`).



The last bullet indicates that the output buffer can also be treated as input by using `METHOD_IN_DIRECT`.

Table 7-1 summarizes these buffering methods.

Table 7-1: Buffering method based on control code `Method` parameter

Method	Input buffer	Output buffer
<code>METHOD_NEITHER</code>	Neither	Neither
<code>METHOD_BUFFERED</code>	Buffered	Buffered
<code>METHOD_IN_DIRECT</code>	Buffered	Direct
<code>METHOD_OUT_DIRECT</code>	Buffered	Direct

Finally, the `Access` parameter to the macro indicates the direction of data flow. `FILE_WRITE_ACCESS` means from the client to the driver, `FILE_READ_ACCESS` means the opposite, and `FILE_ANY_ACCESS` means bi-directional access (the input and output buffers are used). You should always use `FILE_ANY_ACCESS`. Beside simplifying the control code building, you guarantee that if later on, once the driver is already deployed, you may want to use the other buffer, you wouldn't need to change the `Access` parameter, and so not disturb existing clients that would not know about the control code change.



If a control code is built with `METHOD_NEITHER`, the I/O manager does nothing to help with accessing the buffer(s). The values for the input and output buffer pointers provided by the client are copied as-is to the IRP. No checking is done by the I/O manager to make sure these pointers point to valid memory. A driver should not use these pointers as memory pointers, but they can be used as two arbitrary values propagating to the driver that may mean something.

Putting it All Together: The *Zero* Driver

In this section, we'll use what we've learned in this (and earlier) chapter and build a driver and a client application. The driver is named *Zero* and has the following characteristics:

- For read requests, it zeros out the provided buffer.
- For write requests, it just consumes the provided buffer, similar to a classic *null* device.

The driver will use Direct I/O so as not to incur the overhead of copies, as the buffers provided by the client can potentially be very large.

We'll start the project by creating an "Empty WDM Project" in Visual Studio and name it *Zero*. Then we'll delete the created INF file, resulting in an empty project, just like in previous examples.

Using a Precompiled Header

One technique that we can use that is not specific to driver development, but is generally useful, is using a *precompiled header*. Precompiled headers is a Visual Studio feature that helps with faster compilation times. The precompiled header is a header file that has `#include` statements for headers that rarely change, such as `ntddk.h` for drivers. The precompiled header is compiled once, stored in an internal binary format, and used in subsequent compilations, which become considerably faster.



Many user mode projects created by Visual Studio already use precompiled headers. Kernel-mode projects provided by the WDK templates currently don't use precompiled headers. Since we're starting with an empty project, we have to set up precompiled headers manually anyway.

Follow these steps to create and use a precompiled header:

- Add a new header file to the project and call it *pch.h*. This file will serve as the precompiled header. Add all rarely-changing `#includes` here:

```
// pch.h

#pragma once

#include <ntddk.h>
```

- Add a source file named *pch.cpp* and put a single `#include` in it: the precompiled header itself:

```
#include "pch.h"
```

- Now comes the tricky part. Letting the compiler know that *pch.h* is the precompiled header and *pch.cpp* is the one creating it. Open project properties, select *All Configurations* and *All Platforms* so you won't need to configure every configuration/platform separately, navigate to *C/C++ / Precompiled Headers* and set *Precompiled Header* to **Use** and the file name to "*pch.h*" (see figure 7-10). Click OK and to close the dialog box.

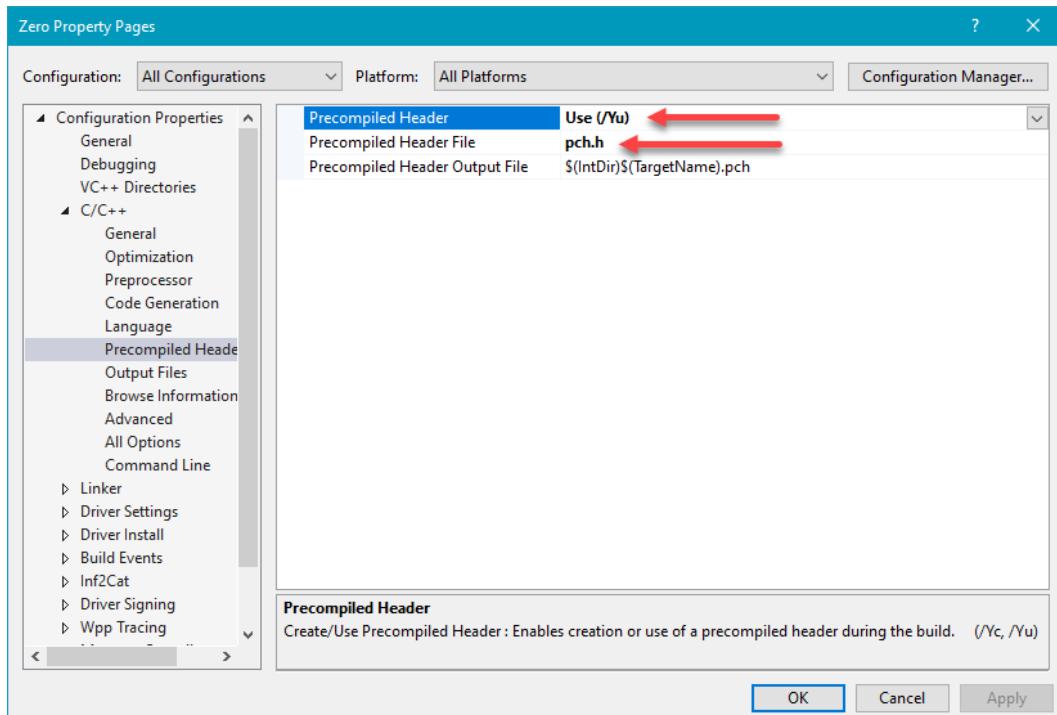


Figure 7-10: Setting precompiled header for the project

- The *pch.cpp* file should be set as the creator of the precompiled header. Right click this file in *Solution Explorer*, and select *Properties*. Navigate to *C/C++ / Precompiled Headers* and set *Precompiled Header* to **Create** (see figure 7-11). Click **OK** to accept the setting.

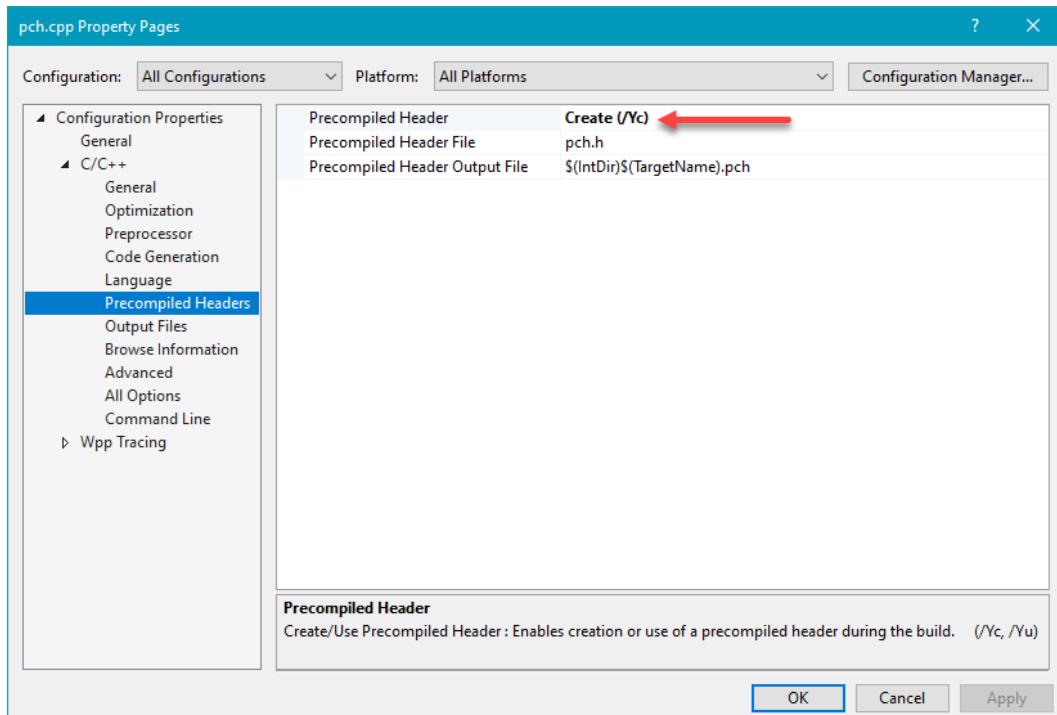


Figure 7-10: Setting precompiled header for pch.cpp

From this point on, every C/CPP file in the project must `#include "pch.h"` as the first thing in the file. Without this include, the project will not compile.



Make sure there is nothing before this `#include "pch.h"` in a source file. Anything before this line does not get compiled at all!

The DriverEntry Routine

The `DriveEntry` routine for the *Zero* driver is very similar to the one we created for the driver in chapter 4. However, in chapter 4's driver the code in `DriverEntry` had to undo any operation that was already done in case of a later error. We had just two operations that could be undone: creation of the device object and creation of the symbolic link. The *Zero* driver is similar, but we'll create a more robust and less error-prone code to handle errors during initialization. Let's start with the basics of setting up an unload routine and the dispatch routines:

```
#define DRIVER_PREFIX "Zero: "

// DriverEntry

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    DriverObject->DriverUnload = ZeroUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = ZeroCreateClose;
    DriverObject->MajorFunction[IRP_MJ_READ] = ZeroRead;
    DriverObject->MajorFunction[IRP_MJ_WRITE] = ZeroWrite;
```

Now we need to create the device object and symbolic link and handle errors in a more general and robust way. The trick we'll use is a do / while(false) block, which is not really a loop, but it allows getting out of the block with a simple break statement in case something goes wrong:

```
UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\Zero");
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\Device\Zero");
PDEVICE_OBJECT DeviceObject = nullptr;
auto status = STATUS_SUCCESS;

do {
    status = IoCreateDevice(DriverObject, 0, &devName, FILE_DEVICE_UNKNOWN,
                           0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\n", status));
        break;
    }

    // set up Direct I/O
    DeviceObject->Flags |= DO_DIRECT_IO;

    status = IoCreateSymbolicLink(&symLink, &devName);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create symbolic link (0x%08X)\n",
                 status));
        break;
    }
} while (false);

if (!NT_SUCCESS(status)) {
```

```

if (DeviceObject)
    IoDeleteDevice(DeviceObject);
}
return status;

```

The pattern is simple: if an error occurs in any call, just break out of the “loop”. Outside the loop, check the status, and if it’s a failure, undo any operations done so far. With this scheme in hand, it’s easy to add more initializations (which we’ll need in more complex drivers), while keeping the cleanup code localized and appearing just once.

It’s possible to use goto statements instead of the do / while(false) approach, but as the great Dijkstra wrote, “goto considered harmful”, so I tend to avoid it if I can.

Notice we’re also initializing the device to use Direct I/O for our read and write operations.

The Create and Close Dispatch Routines

Before we get to the actual implementation of IRP_MJ_CREATE and IRP_MJ_CLOSE (pointing to the same function), let’s create a helper function that simplifies completing an IRP with a given status and information:

```

NTSTATUS CompleteIrp(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS,
    ULONG_PTR info = 0) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

```

Notice the default values for the status and information. The Create/Close dispatch routine implementation becomes almost trivial:

```

NTSTATUS ZeroCreateClose(PDEVICE_OBJECT, PIRP Irp) {
    return CompleteIrp(Irp);
}

```

The Read Dispatch Routine

The Read routine is the most interesting. First we need to check the length of the buffer to make sure it’s not zero. If it is, just complete the IRP with a failure status:

```
NTSTATUS ZeroRead(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Read.Length;
    if (len == 0)
        return CompleteIrp(Irp, STATUS_INVALID_BUFFER_SIZE);
```

Note that the length of the user's buffer is provided through the `Parameters.Read` member inside the current I/O stack location.

We have configured Direct I/O, so we need to map the locked buffer to system space using `MmGetSystemAddressForMdlSafe`:

```
NT_ASSERT(Irp->MdlAddress);           // make sure Direct I/O flag was set
auto buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress, NormalPagePriority);
if (!buffer)
    return CompleteIrp(Irp, STATUS_INSUFFICIENT_RESOURCES);
```

The functionality we need to implement is to zero out the given buffer. We can use a simple `memset` call to fill the buffer with zeros and then complete the request:

```
memset(buffer, 0, len);

return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

If you prefer a more “fancy” function to zero out memory, call `RtlZeroMemory`. It’s a macro, defined in terms of `memset`.

It's important to set the `Information` field to the length of the buffer. This indicates to the client the number of bytes transferred in the operation (returned in the second to last parameter to `ReadFile`). This is all we need for the read operation.

The Write Dispatch Routine

The write dispatch routine is even simpler. All it needs to do is complete the request with the buffer length provided by the client (essentially swallowing the buffer):

```
NTSTATUS ZeroWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Write.Length;

    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

Note that we don't even bother calling `MmGetSystemAddressForMdlSafe`, as we don't need to access the actual buffer. This is also the reason this call is not made beforehand by the I/O manager: the driver may not even need it, or perhaps need it in certain conditions only; so the I/O manager prepares everything (the MDL) and lets the driver decide when and if to map the buffer.

Test Application

We'll add a new console application project to the solution to test the read and write operations. Here is some simple code to test these operations:

```
int Error(const char* msg) {
    printf("%s: error=%u\n", msg, ::GetLastError());
    return 1;
}

int main() {
    HANDLE hDevice = CreateFile(L"\\\\.\Zero", GENERIC_READ | GENERIC_WRITE,
        0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE) {
        return Error("Failed to open device");
    }

    // test read
    BYTE buffer[64];

    // store some non-zero data
    for (int i = 0; i < sizeof(buffer); ++i)
        buffer[i] = i + 1;

    DWORD bytes;
    BOOL ok = ReadFile(hDevice, buffer, sizeof(buffer), &bytes, nullptr);
    if (!ok)
        return Error("failed to read");
    if (bytes != sizeof(buffer))
        printf("Wrong number of bytes\n");
```

```
// check that all bytes are zero
for (auto n : buffer)
    if (n != 0) {
        printf("Wrong data!\n");
        break;
    }

// test write
BYTE buffer2[1024];      // contains junk
ok = WriteFile(hDevice, buffer2, sizeof(buffer2), &bytes, nullptr);
if (!ok)
    return Error("failed to write");
if (bytes != sizeof(buffer2))
    printf("Wrong byte count\n");
CloseHandle(hDevice);
}
```

Read/Write Statistics

Let's add some more functionality to the *Zero* driver. We may want to count the total bytes read/written throughout the lifetime of the driver. A user-mode client should be able to read these statistics, and perhaps even zero them out.

We'll start by defining two global variables to keep track of the total number of bytes read/written (in *Zero.cpp*):

```
long long g_TotalRead;
long long g_TotalWritten;
```

You could certainly put these in a structure for easier maintenance and extension. The `long long` C++ type is a signed 64-bit value. You can add `unsigned` if you wish, or use a `typedef` such as `LONG64` or `ULONG64`, which would mean the same thing. Since these are global variables, they are zeroed out by default.

We'll create a new file that contains information common to user-mode clients and the driver called *ZeroCommon.h*. here is where we define the control codes we support, as well as data structures to be shared with user-mode.

First, we'll add two control codes: one for getting the stats and another for clearing them:

```
#define DEVICE_ZERO 0x8022

#define IOCTL_ZERO_GET_STATS \
CTL_CODE(DEVICE_ZERO, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_ZERO_CLEAR_STATS \
CTL_CODE(DEVICE_ZERO, 0x801, METHOD_NEITHER, FILE_ANY_ACCESS)
```

The DEVICE_ZERO is defined as some number from 0x8000 as the documentation recommends. The *function* number starts with 0x800 and incremented with each control code. METHOD_BUFFERED is used for getting the stats, as the size of the returned data is small (2 x 8 bytes). Clearing the stats requires no buffers, so METHOD_NEITHER is selected.

Next, we'll add a structure that can be used by clients (and the driver) for storing the stats:

```
struct ZeroStats {
    long long TotalRead;
    long long TotalWritten;
};
```

In DriverEntry, we add a dispatch routine for IRP_MJ_DEVICE_CONTROL like so:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ZeroDeviceControl;
```

All the work is done in ZeroDeviceControl. First, some initialization:

```
NTSTATUS ZeroDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG_PTR len = 0;
```

The details for IRP_MJ_DEVICE_CONTROL are located in the current I/O stack location in the Parameters.DeviceIoControl structure. The status is initialized to an error in case the control code provided is unsupported. len keeps track of the number of valid bytes returned in the output buffer.

Implementing the IOCTL_ZERO_GET_STATS is done in the usual way. First, check for errors. If all goes well, the stats are written to the output buffer:

```

switch (dic.IoControlCode) {
    case IOCTL_ZERO_GET_STATS:
    {
        // artificial scope so the compiler does not complain
        // about defining variables skipped by a case
        if (dic.OutputBufferLength < sizeof(ZeroStats)) {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        }

        auto stats = (ZeroStats*)Irp->AssociatedIrp.SystemBuffer;
        if (stats == nullptr) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        //
        // fill in the output buffer
        //
        stats->TotalRead = g_TotalRead;
        stats->TotalWritten = g_TotalWritten;
        len = sizeof(ZeroStats);
        //
        // change status to indicate success
        //
        status = STATUS_SUCCESS;
        break;
    }
}

```

Once out of the `switch`, the IRP would be completed. Here is the stats clearing Ioctl handling:

```

case IOCTL_ZERO_CLEAR_STATS:
    g_TotalRead = g_TotalWritten = 0;
    status = STATUS_SUCCESS;
    break;
}

```

All that's left to do is complete the IRP with whatever the status and length values are:

```
return CompleteIrp(Irp, status, len);
```

For easier viewing, here is the complete `IRP_MJ_DEVICE_CONTROL` handling:

```
NTSTATUS ZeroDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG_PTR len = 0;

    switch (dic.IoControlCode) {
        case IOCTL_ZERO_GET_STATS:
        {
            if (dic.OutputBufferLength < sizeof(ZeroStats)) {
                status = STATUS_BUFFER_TOO_SMALL;
                break;
            }

            auto stats = (ZeroStats*)Irp->AssociatedIrp.SystemBuffer;
            if (stats == nullptr) {
                status = STATUS_INVALID_PARAMETER;
                break;
            }
            stats->TotalRead = g_TotalRead;
            stats->TotalWritten = g_TotalWritten;
            len = sizeof(ZeroStats);
            status = STATUS_SUCCESS;
            break;
        }

        case IOCTL_ZERO_CLEAR_STATS:
        {
            g_TotalRead = g_TotalWritten = 0;
            status = STATUS_SUCCESS;
            break;
        }
    }

    return CompleteIrp(Irp, status, len);
}
```

The stats have to be updated when data is read/written. It must be done in a thread safe way, as multiple clients may bombard the driver with read/write requests. Here is the updated ZeroWrite function:

```
NTSTATUS ZeroWrite(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto len = stack->Parameters.Write.Length;
    // update the number of bytes written
    InterlockedAdd64(&g_TotalWritten, len);
    return CompleteIrp(Irp, STATUS_SUCCESS, len);
}
```

The change to ZeroRead is very similar.

Astute readers may question the safety of the Ioclt implementations. For example, is reading the total number of bytes read/written with no multithreaded protection (while possible read/write operations are in effect) a correct operation, or is it a data race? Technically, it's a data race, as the driver might be updating to the stats globals while some client is reading the values, that could result in torn reads. One way to resolve that is by dispensing with the interlocked instructions and use a mutex or a fast mutex to protect access to these variables. Alternatively, There are functions to deal with these scenario, such as ReadAcquire64. Their implementation is CPU dependent. For x86/x64, they are actually normal reads, as the processor provides safety against such torn reads. On ARM CPUs, this requires a memory barrier to be inserted (memory barriers are beyond the scope of this book).



Save the number of bytes read/written to the Registry before the driver unloads. Read it back when the driver loads.

Replace the Interlocked instructions with a fast mutex to protect access to the stats.

Here is some client code to retrieve these stats:

```
ZeroStats stats;
if (!DeviceIoControl(hDevice, IOCTL_ZERO_GET_STATS,
    nullptr, 0, &stats, sizeof(stats), &bytes, nullptr))
    return Error("failed in DeviceIoControl");

printf("Total Read: %lld, Total Write: %lld\n",
    stats.TotalRead, stats.TotalWritten);
```

Summary

In this chapter, we learned how to handle IRPs, which drivers deal with all the time. Armed with this knowledge, we can start leveraging more kernel functionality, starting with process and thread callbacks in chapter 9. Before getting to that, however, there are more techniques and kernel APIs that may be useful for a driver developer, described in the next chapter.

Chapter 8: Advanced Programming Techniques (Part 1)

In this chapter we'll examine various techniques of various degrees of usefulness to driver developers.

In this chapter:

- Driver Created Threads
 - Memory Management
 - Calling Other Drivers
 - Putting it All Together: The Melody Driver
 - Invoking System Services
-

Driver Created Threads

We've seen how to create work items in chapter 6. Work items are useful when some code needs to execute on a separate thread, and that code is "bound" in time - that is, it's not too long, so that the driver doesn't "steal" a thread from the kernel worker threads. For long operations, however, it's preferable that drivers create their own separate thread(s). Two functions are available for this purpose:

```
NTSTATUS PsCreateSystemThread(  
    _Out_ PHANDLE ThreadHandle,  
    _In_ ULONG DesiredAccess,  
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,  
    _In_opt_ HANDLE ProcessHandle,  
    _Out_opt_ PCLIENT_ID ClientId,  
    _In_ PKSTART_ROUTINE StartRoutine,  
    _In_opt_ PVOID StartContext);  
  
NTSTATUS IoCreateSystemThread( // Win 8 and later  
    _Inout_ PVOID IoObject,  
    _Out_ PHANDLE ThreadHandle,  
    _In_ ULONG DesiredAccess,
```

```
_In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
_In_opt_ HANDLE ProcessHandle,
_Out_opt_ PCLIENT_ID ClientId,
_In_ PKSTART_ROUTINE StartRoutine,
_In_opt_ PVOID StartContext);
```

Both functions have the same set of parameters except the additional first parameter to `IoCreateSystemThread`. The latter function takes an additional reference on the object passed in (which must be a device object or a driver object), so the driver is not unloaded prematurely while the thread is alive. `IoCreateSystemThread` is only available for Windows 8 and later systems. Here is a description of the other parameters:

- `ThreadHandle` is the address of a handle to the created thread if successful. The driver must use `ZwClose` to close the handle at some point.
- `DesiredAccess` is the access mask requested. Drivers should simply use `THREAD_ALL_ACCESS` to get all possible access with the resulting handle.
- `ObjectAttributes` is the standard `OBJECT_ATTRIBUTES` structure. Most members have no meaning for a thread. The most common attributes to request of the returned handle is `OBJ_KERNEL_HANDLE`, but it's not needed if the thread is to be created in the System process - just pass `NULL`, which will always return a kernel handle.
- `ProcessHandle` is a handle to the process where this thread should be created. Drivers should pass `NULL` to indicate the thread should be part of the System process so it's not tied to any specific process' lifetime.
- `ClientId` is an optional output structure, providing the process and thread ID of the newly created thread. In most cases, this information is not needed, and `NULL` can be specified.
- `StartRoutine` is the function to execute in a separate thread of execution. This function must have the following prototype:

```
VOID KSTART_ROUTINE (_In_ PVOID StartContext);
```

The `StartContext` value is provided by the last parameter to `PsCreateSystemThread`. This could be anything (or `NULL`) that would give the new thread data to work with.

The function indicated by `StartRoutine` will start execution on a separate thread. It's executed with the IRQL being `PASSIVE_LEVEL` (0) in a critical region (where normal kernel APCs are disabled).

For `PsCreateSystemThread`, exiting the thread function is not enough to terminate the thread. An explicit call to `PsTerminateSystemThread` is required to properly manage the thread's lifetime:

```
NTSTATUS PsTerminateSystemThread(_In_ NTSTATUS ExitStatus);
```

The exit status is the exit code of the thread, which can be retrieved with `PsGetThreadExitStatus` if desired.

For `IoCreateSystemThread`, exiting the thread function is sufficient, as `PsTerminateSystemThread` is called on its behalf when the thread function returns. The exit code of the thread is always `STATUS_SUCCESS`.



`IoCreateSystemThread` is a wrapper around `PsCreateSystemThread` that increments the ref count of the passed in device/driver object, calls `PsCreateSystemThread` and then decrements the ref count and calls `PsTerminateSystemThread`.

Memory Management

We have looked at the most common functions for dynamic memory allocation in chapter 3. The most useful is `ExAllocatePoolWithTag`, which we have used multiple times in previous chapters. There are other functions for dynamic memory allocation you might find useful. Then, we'll examine lookaside lists, that allow more efficient memory management if fixed-size chunks are needed.

Pool Allocations

In addition to `ExAllocatePoolWithTag`, the Executive provides an extended version that indicates the importance of an allocation, taken into account in low memory conditions:

```
typedef enum _EX_POOL_PRIORITY {
    LowPoolPriority,
    LowPoolPrioritySpecialPoolOverrun = 8,
    LowPoolPrioritySpecialPoolUnderrun = 9,
    NormalPoolPriority = 16,
    NormalPoolPrioritySpecialPoolOverrun = 24,
    NormalPoolPrioritySpecialPoolUnderrun = 25,
    HighPoolPriority = 32,
    HighPoolPrioritySpecialPoolOverrun = 40,
    HighPoolPrioritySpecialPoolUnderrun = 41
} EX_POOL_PRIORITY;

VOID ExAllocatePoolWithTagPriority (
    _In_ POOL_TYPE PoolType,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag,
    _In_ EX_POOL_PRIORITY Priority);
```

The priority-related values indicate the importance of succeeding an allocation if system memory is low (`LowPoolPriority`), very low (`NormalPoolPriority`), or completely out of memory (`HighPoolPriority`). In any case, the driver should be prepared to handle a failure.

The “special pool” values tell the Executive to make the allocation at the end of a page (“Overrun” values) or beginning of a page (“Underrun”) values, so it’s easier to catch buffer overflow or underflow. These values should only be used while tracking memory corruptions, as each allocation costs at least one page.

Starting with Windows 10 version 1909 (and Windows 11), two new pool allocation functions are supported. The first is `ExAllocatePool2` declared like so:

```
PVOID ExAllocatePool2 (
    _In_ POOL_FLAGS Flags,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag);
```

Where the `POOL_FLAGS` enumeration consists of a combination of values shown in table 8-1:

Table 8-1: Flags for `ExAllocatePool2`

Flag (<code>POOL_FLAG</code>)	Must recognize?	Description
<code>USE_QUOTA</code>	Yes	Charge allocation to calling process
<code>UNINITIALIZED</code>	Yes	Contents of allocated memory is not touched. Without this flag, the memory is zeroed out
<code>CACHE_ALIGNED</code>	Yes	Address should be CPU-cache aligned. This is “best effort”
<code>RAISE_ON_FAILURE</code>	Yes	Raises an exception (<code>STATUS_INSUFFICIENT_RESOURCES</code>) instead of returning <code>NULL</code> if allocation fails
<code>NON_PAGED</code>	Yes	Allocate from non-paged pool. The memory is executable on x86, and non-executable on all other platforms
<code>PAGED</code>	Yes	Allocate from paged pool. The memory is executable on x86, and non-executable on all other platforms
<code>NON_PAGED_EXECUTABLE</code>	Yes	Non paged pool with execute permissions
<code>SPECIAL_POOL</code>	No	Allocates from “special” pool (separate from the normal pool so it’s easier to find memory corruptions)

The *Must recognize?* column indicates whether failure to recognize or satisfy the flag causes the function to fail.

The second allocation function, `ExAllocatePool3`, is extensible, so new functions of this sort are unlikely to pop up in the future:

```
PVOID ExAllocatePool3 (
    _In_ POOL_FLAGS Flags,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag,
    _In_reads_opt_(ExtendedParametersCount)
        PCPOOL_EXTENDED_PARAMETER ExtendedParameters,
    _In_ ULONG ExtendedParametersCount);
```

This function allows customization with an array of “parameters”, where the supported parameter types may be extended in future kernel versions. The currently available parameters are defined with the POOL_EXTENDED_PARAMETER_TYPE enumeration:

```
typedef enum POOL_EXTENDED_PARAMETER_TYPE {
    PoolExtendedParameterInvalidType = 0,
    PoolExtendedParameterPriority,
    PoolExtendedParameterSecurePool,
    PoolExtendedParameterNumaNode,
    PoolExtendedParameterMax
} POOL_EXTENDED_PARAMETER_TYPE, *PPOOL_EXTENDED_PARAMETER_TYPE;
```

The array provided to ExAllocatePool3 consists of structures of type POOL_EXTENDED_PARAMETER, each one specifying one parameter:

```
typedef struct _POOL_EXTENDED_PARAMETER {
    struct {
        ULONG64 Type : 8;
        ULONG64 Optional : 1;
        ULONG64 Reserved : 64 - 9;
    };

    union {
        ULONG64 Reserved2;
        PVOID Reserved3;
        EX_POOL_PRIORITY Priority;
        POOL_EXTENDED_PARAMS_SECURE_POOL* SecurePoolParams;
        POOL_NODE_REQUIREMENT PreferredNode; // ULONG
    };
} POOL_EXTENDED_PARAMETER, *PPOOL_EXTENDED_PARAMETER;
```

The Type member indicates which of the union members is valid for this parameter (POOL_EXTENDED_PARAMETER_TYPE). Optional indicates if the parameter set is optional or required. An optional parameter that fails to be satisfied does not cause the ExAllocatePool3 to fail. Based on Type, the correct member in the union must be set. Currently, these parameters are available:

- Priority of the allocation (Priority member)
- Preferred NUMA node (PreferredNode member)
- Use secure pool (discussed later, SecurePoolParams member)

The following example shows using ExAllocatePool3 to achieve the same effect as ExAllocatePoolWithTagPriority for non-paged memory:

```
PVOID AllocNonPagedPriority(ULONG size, ULONG tag, EX_POOL_PRIORITY priority) {
    POOL_EXTENDED_PARAMETER param;
    param.Optional = FALSE;
    param.Type = PoolExtendedParameterPriority;
    param.Priority = priority;

    return ExAllocatePool3(POOL_FLAG_NON_PAGED, size, tag, &param, 1);
}
```

Secure Pools

Secure pools introduced in Windows 10 version 1909 allow kernel callers to have a memory pool that cannot be accessed by other kernel components. This kind of protection is internally achieved by the Hyper-V hypervisor, leveraging its power to protect memory access even from the kernel, as the memory is part of *Virtual Trust Level* (VTL) 1 (the secure world). Currently, secure pools are not fully documented, but here are the basic steps to use a secure pool.



Secure pools are only available if *Virtualization Based Security* (VBS) is active (meaning Hyper-V exists and creates the two worlds - normal and secure). Discussion of VBS is beyond the scope of this book. Consult information online (or the Windows Internals books) for more on VBS.

A secure pool can be created with ExCreatePool, returning a handle to the pool:

```
#define POOL_CREATE_FLG_SECURE_POOL      0x1
#define POOL_CREATE_FLG_USE_GLOBAL_POOL  0x2
#define POOL_CREATE_FLG_VALID_FLAGS   (POOL_CREATE_FLG_SECURE_POOL | \
                                         POOL_CREATE_FLG_USE_GLOBAL_POOL)

NTSTATUS ExCreatePool (
    _In_ ULONG Flags,
    _In_ ULONG_PTR Tag,
    _In_opt_ POOL_CREATE_EXTENDED_PARAMS* Params,
    _Out_ HANDLE* PoolHandle);
```

Currently, flags should be `POOL_CREATE_FLG_VALID_FLAGS` (both supported flags), and `Params` should be `NULL`. `PoolHandle` contains the pool handle if the call succeeds.

Allocating from a secure pool must be done with `ExAllocatePool3`, described in the previous section with a `POOL_EXTENDED_PARAMS_SECURE_POOL` structure as a parameter:

```
#define SECURE_POOL_FLAGS_NONE      0x0
#define SECURE_POOL_FLAGS_FREEABLE   0x1
#define SECURE_POOL_FLAGS_MODIFIABLE 0x2

typedef struct _POOL_EXTENDED_PARAMS_SECURE_POOL {
    HANDLE SecurePoolHandle;        // pool handle
    PVOID Buffer;                  // initial data
    ULONG_PTR Cookie;              // for validation
    ULONG SecurePoolFlags;         // flags above
} POOL_EXTENDED_PARAMS_SECURE_POOL;
```

`Buffer` points to existing data to be initially stored in the new allocation. `Cookie` is used for validation, by calling `ExSecurePoolValidate`. Freeing memory from a secure pool must be done with a new function, `ExFreePool2`:

```
VOID ExFreePool2 (
    _Pre_notnull_ PVOID P,
    _In_ ULONG Tag,
    _In_reads_opt_(ExtendedParametersCount)
        PCPOOL_EXTENDED_PARAMETER ExtendedParameters,
    _In_ ULONG ExtendedParametersCount);
```

If `ExtendedParameters` is `NULL` (and `ExtendedParametersCount` is zero), the call is diverted to the normal `ExFreePool1`, which will fail for a secure pool. For a secure pool, a single `POOL_EXTENDED_PARAMETER` is required that has the pool parameters with the pool handle only. `Buffer` should be `NULL`.

Updating the memory in the pool requires its own call:

```
NTSTATUS ExSecurePoolUpdate (
    _In_ HANDLE SecurePoolHandle,
    _In_ ULONG Tag,
    _In_ PVOID Allocation,
    _In_ ULONG_PTR Cookie,
    _In_ SIZE_T Offset,
    _In_ SIZE_T Size,
    _In_ PVOID Buffer);
```

Finally, a secure pool must be destroyed with `ExDestroyPool`:

```
VOID ExDestroyPool (_In_ HANDLE PoolHandle);
```

Overloading the new and delete Operators

We know there is no C++ runtime in the kernel, which means some C++ features that work as expected in user mode don't work in kernel mode. One of these features are the new and delete C++ operators. Although we can use the dynamic memory allocation functions, new and delete have a couple of advantages over calling the raw functions:

- new causes a constructor to be invoked, and delete causes the destructor to be invoked.
- new accepts a type for which memory must be allocated, rather than specifying a number of bytes.

Fortunately, C++ allows overloading the new and delete operators, either globally or for specific types. new can be overloaded with extra parameters that are needed for kernel allocations - at least the pool type must be specified. The first argument to any overloaded new is the number of bytes to allocate, and any extra parameters can be added after that. These are specified with parenthesis when actually used. The compiler inserts a call to the appropriate constructor, if exists.

Here is a basic implementation of an overloaded new operator that calls ExAllocatePoolWithTag:

```
void* __cdecl operator new(size_t size, POOL_TYPE pool, ULONG tag) {
    return ExAllocatePoolWithTag(pool, size, tag);
}
```

The `__cdecl` modifier indicates this should be using the C calling convention (rather than the `__stdcall` convention). It only matters in x86 builds, but still should be specified as shown.

Here is an example usage, assuming an object of type MyData needs to be allocated from paged pool:

```
MyData* data = new (PagedPool, DRIVER_TAG) MyData;
if(data == nullptr)
    return STATUS_INSUFFICIENT_RESOURCES;
// do work with data
```

The size parameter is never specified explicitly as the compiler inserts the correct size (which is essentially `sizeof(MyData)` in the above example). All other parameters must be specified. We can make the overload simpler to use if we default the tag to a macro such as `DRIVER_TAG`, expected to exist:

```
void* __cdecl operator new(size_t size, POOL_TYPE pool,
    ULONG tag = DRIVER_TAG) {
    return ExAllocatePoolWithTag(pool, size, tag);
}
```

And the corresponding usage is simpler:

```
MyData* data = new (PagedPool) MyData;
```

In the above examples, the default constructor is invoked, but it's perfectly valid to invoke any other constructor that exists for the type. For example:

```
struct MyData {
    MyData(ULONG someValue);
    // details not shown
};

auto data = new (PagedPool) MyData(200);
```

We can easily extend the overloading idea to other overloads, such as one that wraps `ExAllocatePoolWithTagPriority`:

```
void* __cdecl operator new(size_t size, POOL_TYPE pool,
    EX_POOL_PRIORITY priority, ULONG tag = DRIVER_TAG) {
    return ExAllocatePoolWithTagPriority(pool, size, tag, priority);
}
```

Using the above operator is just a matter of adding a priority in parenthesis:

```
auto data = new (PagedPool, LowPoolPriority) MyData(200);
```

Another common case is where you already have an allocated block of memory to store some object (perhaps allocated by a function out of your control), but you still want to initialize the object by invoking a constructor. Another `new` overload can be used for this purpose, known as *placement new*, since it does not allocate anything, but the compiler still adds a call to a constructor. Here is how to define a placement `new` operator overload:

```
void* __cdecl operator new(size_t size, void* p) {
    return p;
}
```

And an example usage:

```
void* SomeFunctionAllocatingObject();

MyData* data = (MyData*)SomeFunctionAllocatingObject();
new (data) MyData;
```

Finally, an overload for `delete` is required so the memory can be freed at some point, calling the destructor if it exists. Here is how to overload the `delete` operator:

```
void __cdecl operator delete(void* p, size_t) {
    ExFreePool(p);
}
```

The extra size parameter is not used in practice (zero is always the value provided), but the compiler requires it.



Remember that you cannot have global objects that have default constructors that do something, since there is no runtime to invoke them. The compiler will report a warning if you try. A way around it (of sorts) is to declare the global variable as a pointer, and then use an overloaded new to allocate and invoke a constructor in `DriverEntry`. Of course, you must remember to call `delete` in the driver's unload routine.

Another variant of the `delete` operator the compiler might insist on if you set the compiler conformance to C++17 or newer is the following:

```
void __cdecl operator delete(void* p, size_t, std::align_val_t) {
    ExFreePool(p);
}
```

You can look up the meaning of `std::align_val_t` in a C++ reference, but it does not matter for our purposes.

Lookaside Lists

The dynamic memory allocation functions discussed so far (the `ExAllocatePool*` family of APIs) are generic in nature, and can accommodate allocations of any size. Internally, managing the pool is non-trivial: various lists are needed to manage allocations and deallocations of different sizes. This management aspect of the pools is not free.

One fairly common case that leaves room for optimizations is when fixed-sized allocations are needed. When such allocation is freed, it's possible to not really free it, but just mark it as available. The next allocation request can be satisfied by the existing block, which is much faster to do than allocating a fresh block. This is exactly the purpose of lookaside lists.

There are two APIs to use for working with lookaside lists. The original one, available from Windows 2000, and a newer available from Vista. I'll describe both, as they are quite similar.

The “Classic” Lookaside API

The first thing to do is to initialize the data structure managing a lookaside list. Two functions are available, which are essentially the same, selecting the paged pool or non-paged pool where the allocations should be coming from. Here is the paged pool version:

```
VOID ExInitializePagedLookasideList (
    _Out_ PPAGED_LOOKASIDE_LIST Lookaside,
    _In_opt_ PALLOCATE_FUNCTION Allocate,
    _In_opt_ PFREE_FUNCTION Free,
    _In_ ULONG Flags,
    _In_ SIZE_T Size,
    _In_ ULONG Tag,
    _In_ USHORT Depth);
```

The non-paged variant is practically the same, with the function name being `ExInitializeNPagedLookasideList`.

The first parameter is the resulting initialized structure. Although, the structure layout is described in `wdm.h` (with a macro named `GENERAL_LOOKASIDE_LAYOUT` to accommodate multiple uses that can't be shared in other ways using the C language), you should treat this structure as opaque.

The `Allocate` parameter is an optional allocation function that is called by the lookaside implementation when a new allocation is required. If specified, the allocation function must have the following prototype:

```
PVOID AllocationFunction (
    _In_ POOL_TYPE PoolType,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag);
```

The allocation function receives the same parameters as `ExAllocatePoolWithTag`. In fact, if the allocation function is not specified, this is the call made by the lookaside list manager. If you don't require any other code, just specify `NULL`. A custom allocation function could be useful for debugging purposes, for example. Another possibility is to call `ExAllocatePoolWithTagPriority` instead of `ExAllocatePoolWithTag`, if that makes sense for your driver.

If you do provide an allocation function, you might need to provide a de-allocation function in the `Free` parameter. If not specified, the lookaside list manager calls `ExFreePool`. Here is the expected prototype for this function:

```
VOID FreeFunction (
    _In_ __drv_freesMem(Mem) PVOID Buffer);
```

The next parameter, `Flags` can be zero or `POOL_RAISE_IF_ALLOCATION_FAILURE` (Windows 8 and later) that indicates an exception should be raised (`STATUS_INSUFFICIENT_RESOURCE`) if an allocation fails, instead of returning `NULL` to the caller.

The `Size` parameter is the size of chunks managed by the lookaside list. Usually, you would specify it as `sizeof` some structure you want to manage. `Tag` is the tag to use for allocations. Finally, the last parameter, `Depth`, indicates the number of allocations to keep in a cache. The documentation indicates this parameter is "reserved" and should be zero, which makes the lookaside list manager to choose something appropriate. Regardless of the number, the "depth" is adjusted based on the allocation patterns used with the lookaside list.

Once a lookaside list is initialized, you can request a memory block (of the size specified in the initialization function, of course) by calling `ExAllocateFromPagedLookasideList`:

```
VOID ExAllocateFromPagedLookasideList (
    _Inout_ PPAGED_LOOKASIDE_LIST Lookaside)
```

Nothing could be simpler - no special parameters are required, since everything else is already known. The corresponding function for a non-paged pool lookaside list is `ExAllocateFromNPagedLookasideList`. The opposite function used to free an allocation (or return it to the cache) is `ExFreeToPagedLookasideList`:

```
VOID ExFreeToPagedLookasideList (
    _Inout_ PPAGED_LOOKASIDE_LIST Lookaside,
    _In_ __drv_freesMem(Mem) PVOID Entry)
```

The only value required is the pointer to free (or return to the cache). As you probably guess, the non-paged pool variant is `ExFreeToNPagedLookasideList`.

Finally, when the lookaside list is no longer needed, it must be freed by calling `ExDeletePagedLookasideList`:

```
VOID ExDeletePagedLookasideList (
    _Inout_ PPAGED_LOOKASIDE_LIST Lookaside);
```

One nice benefit of lookaside lists is that you don't have to return all allocations to the list by repeatedly calling `ExFreeToPagedLookasideList` before calling `ExDeletePagedLookasideList`; the latter is enough, and will free all allocated blocks automatically. `ExDeleteNPagedLookasideList` is the corresponding non-paged variant.



Write a C++ class wrapper for lookaside lists using the above APIs.

The Newer Lookaside API

The newer API provides two main benefits over the classic API:

- Uniform API for paged and non-paged blocks.
- The lookaside list structure itself is passed to the custom allocate and free functions (if provided), that allows accessing driver data (example shown later).

Initializing a lookaside list is accomplished with `ExInitializeLookasideListEx`:

```
NTSTATUS ExInitializeLookasideListEx (
    _Out_ PLOOKASIDE_LIST_EX Lookaside,
    _In_opt_ PALLOCATE_FUNCTION_EX Allocate,
    _In_opt_ PFREE_FUNCTION_EX Free,
    _In_ POOL_TYPE PoolType,
    _In_ ULONG Flags,
    _In_ SIZE_T Size,
    _In_ ULONG Tag,
    _In_ USHORT Depth);
```

PLOOKASIDE_LIST_EX is the opaque data structure to initialize, which must be allocated from non-paged memory, regardless of whether the lookaside list is to manage paged or non-paged memory.

The allocation and free functions are optional, just as they are with the classic API. These are their prototypes:

```
PVOID AllocationFunction (
    _In_ POOL_TYPE PoolType,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag,
    _Inout_ PLOOKASIDE_LIST_EX Lookaside);
VOID FreeFunction (
    _In_ __drv_freesMem(Mem) PVOID Buffer,
    _Inout_ PLOOKASIDE_LIST_EX Lookaside);
```

Notice the lookaside list itself is a parameter. This could be used to access driver data that is part of a larger structure containing the lookaside list. For example, suppose the driver has the following structure:

```
struct MyData {
    ULONG SomeData;
    LIST_ENTRY SomeHead;
    LOOKASIDELIST_EX Lookaside;
};
```

The driver creates an instance of that structure (maybe globally, or on a per-client basis). Let's assume it's created dynamically for every client creating a file object to talk to a device the driver manages:

```

// if new is overridden as described earlier in this chapter
MyData* pData = new (NonPagedPool) MyData;
// or with a standard allocation call
MyData* pData = (MyData*)ExAllocatePoolWithTag(NonPagedPool,
    sizeof(MyData), DRIVER_TAG);

// initialize the lookaside list
ExInitializeLookasideListEx(&pData->Lookaside, MyAlloc, MyFree, ...);

```

In the allocation and free functions, we can get a pointer to our `MyData` object that contains whatever lookaside list is being used at the time:

```

VOID MyAlloc(POOL_TYPE type, SIZE_T size, ULONG tag,
    PLOOKASIDE_LIST_EX lookaside) {
    MyData* data = CONTAINING_RECORD(lookaside, MyData, Lookaside);
    // access members
    //...
}

```

The usefulness of this technique is if you have multiple lookaside lists, each one could have their own “context” data. Obviously, if you just have one such list stored globally, you can just access whatever global variables you need.

Continuing with `ExInitializeLookasideListEx` - `PoolType` is the pool type to use; this is where the driver selects where allocations should be made from. `Size`, `Tag` and `Depth` have the same meaning as they do in the classic API.

The `Flags` parameter can be zero, or one of the following:

- `EX_LOOKASIDE_LIST_EX_FLAGS_RAISE_ON_FAIL` - raise an exception instead of returning `NULL` to the caller in case of an allocation failure.
- `EX_LOOKASIDE_LIST_EX_FLAGS_FAIL_NO_RAISE` - this flag can only be specified if a custom allocation routine is specified, which causes the pool type provided to the allocation function to be ORed with the `POOL_QUOTA_FAIL_INSTEAD_OF_RAISE` flag that causes a call to `ExAllocationPoolWithQuotaTag` to return `NULL` on quota limit violation instead of raising the `POOL_QUOTA_FAIL_INSTEAD_OF_RAISE` exception. See the docs for more details.



The above flags are mutually exclusive.

Once the lookaside list is initialized, allocation and deallocation are done with the following APIs:

```
PVOID ExAllocateFromLookasideListEx (_Inout_ PLOOKASIDE_LIST_EX Lookaside);
VOID ExFreeToLookasideListEx (
    _Inout_ PLOOKASIDE_LIST_EX Lookaside,
    _In_ __drv_freesMem(Entry) PVOID Entry);
```

Of course, the terms “allocation” and “deallocation” are in the context of a lookaside list, meaning allocations could be reused, and deallocations might return the block to the cache.

Finally, a lookaside list must be deleted with `ExDeleteLookasideListEx`:

```
VOID ExDeleteLookasideListEx (_Inout_ PLOOKASIDE_LIST_EX Lookaside);
```

Calling Other Drivers

One way to talk to other drivers is to be a “proper” client by calling `ZwOpenFile` or `ZwCreateFile` in a similar manner to what a user-mode client does. Kernel callers have other options not available for user-mode callers. One of the options is creating IRPs and sending them to a device object directly for processing.

IRPs are typically created by one of the three managers, part of the Executive: I/O manager, Plug & Play manager, and Power manager. In the cases we’ve seen so far, the I/O manager is the one creating IRPs for create, close, read, write, and device I/O control request types. Drivers can create IRPs as well, initialize them and then send them directly to another driver for processing. This could be more efficient than opening a handle to the desired device, and then making calls using `ZwReadFile`, `ZwWriteFile` and similar APIs we’ll look at in more detail in a later chapter. In some cases, opening a handle to a device might not even be an option, but obtaining a device object pointer might still be possible.

The kernel provides a generic API for building IRPs, starting with `IoAllocateIrp`. Using this API requires the driver to register a completion routine so the IRP can be properly freed. We’ll examine these techniques in a later chapter (“Advanced Programming Techniques (Part 2)”). In this section, I’ll introduce a simpler function to build a device I/O control IRP using `IoBuildDeviceIoControlRequest`:

```
PIRP IoBuildDeviceIoControlRequest(
    _In_          ULONG IoControlCode,
    _In_          PDEVICE_OBJECT DeviceObject,
    _In_opt_       PVOID InputBuffer,
    _In_          ULONG InputBufferLength,
    _Out_opt_     PVOID OutputBuffer,
    _In_          ULONG OutputBufferLength,
    _In_          BOOLEAN InternalDeviceIoControl,
    _In_opt_     PKEVENT Event,
    _Out_         PIO_STATUS_BLOCK IoStatusBlock);
```

The API returns a proper IRP pointer on success, including filling in the first `IO_STACK_LOCATION`, or `NULL` on failure. Some of the parameters to `IoBuildDeviceIoControlRequest` are the same provided

to the `DeviceIoControl` user-mode API (or to its kernel equivalent, `ZwDeviceIoControlFile`) - `IoControlCode`, `InputBuffer`, `InputBufferLength`, `OutputBuffer` and `OutputBufferLength`.

The other parameters are the following:

- `DeviceObject` is the target device of this request. It's needed so the API can allocate the correct number of `IO_STACK_LOCATION` structures that accompany any IRP.
- `InternalDeviceControl` indicates whether the IRP should set its major function to `IRP_MJ_INTERNAL_DEVICE_CONTROL` (TRUE) or `IRP_MJ_DEVICE_CONTROL` (FALSE). This obviously depends on the target device's expectations.
- `Event` is an optional pointer to an event object that gets signaled when the IRP is completed by the target device (or some other device the target may send the IRP to). An event is needed if the IRP is sent for synchronous processing, so that the caller can wait on the event if the operation has not yet completed. We'll see a complete example in the next section.
- `IoStatusBlock` returns the final status of the IRP (status and information), so the caller can examine it if it so wishes.

The call to `IoBuildDeviceIoControlRequest` just builds the IRP - it is not sent anywhere at this point. To actually send the IRP to a device, call the generic `IoCallDriver` API:

```
NTSTATUS IoCallDriver(
    _In_ PDEVICE_OBJECT DeviceObject,
    _Inout_ PIRP Irp);
```

`IoCallDriver` advances the current I/O stack location to the next, and then invokes the target driver's major function dispatch routine. It returns whatever is returned from that dispatch routine. Here is a very simplified implementation:

```
NTSTATUS IoCallDriver(PDEVICE_OBJECT DeviceObject, PIRP Irp {
    // update the current layer index
    DeviceObject->CurrentLocation--;
    auto irpSp = IoGetNextIrpStackLocation(Irp);
    // make the next stack location the current one
    Irp->Tail.Overlay.CurrentStackLocation = irpSp;
    // update device object
    irpSp->DeviceObject = DeviceObject;

    return (DeviceObject->DriverObject->MajorFunction[irpSp->MajorFunction])
        (DeviceObject, Irp);
}
```

The main question remaining is how to we get a pointer to a device object in the first place? One way is by calling `IoGetDeviceObjectPointer`:

```
NTSTATUS IoGetDeviceObjectPointer(
    _In_  PUNICODE_STRING ObjectName,
    _In_  ACCESS_MASK DesiredAccess,
    _Out_ PFILE_OBJECT *FileObject,
    _Out_ PDEVICE_OBJECT *DeviceObject);
```

The *ObjectName* parameter is the fully-qualified name of the device object in the Object Manager's namespace (as can be viewed with the *WinObj* tool from *Sysinternals*). Desired access is usually FILE_READ_DATA, FILE_WRITE_DATA or FILE_ALL_ACCESS. Two values are returned on success: the device object pointer (in *DeviceObject*) and an open file object pointing to the device object (in *FileObject*).

The file object is not usually needed, but it should be kept around as a means of keeping the device object referenced. When you're done with the device object, call *ObDereferenceObject* on the file object pointer to decrement the device object's reference count indirectly. Alternatively, you can increment the device object's reference count (*ObReferenceObject*) and then decrement the file object's reference count so you don't have to keep it around.

The next section demonstrates usage of these APIs.

Putting it All Together: The Melody Driver

The *Melody* driver we'll build in this section demonstrates many of the techniques shown in this chapter. The melody driver allows playing sounds asynchronously (contrary to the *Beep* user-mode API that plays sounds synchronously). A client application calls *DeviceIoControl* with a bunch of notes to play, and the driver will play them as requested without blocking. Another sequence of notes can then be sent to the driver, those notes queued to be played after the first sequence is finished.

It's possible to come up with a user-mode solution that would do essentially the same thing, but this can only be easily done in the context of a single process. A driver, on the other hand, can accept calls from multiple processes, having a "global" ordering of playback. In any case, the point is to demonstrate driver programming techniques, rather than managing a sound playing scenario.

We'll start by creating an empty WDM driver, as we've done in previous chapters, named *KMelody*. Then we'll add a file named *MelodyPublic.h* to serve as the common data to the driver and a user-mode client. This is where we define what a note looks like and an I/O control code for communication:

```
// MelodyPublic.h
#pragma once

#define MELODY_SYMLINK L"\\?\?\KMelody"

struct Note {
    ULONG Frequency;
    ULONG Duration;
    ULONG Delay{ 0 };
    ULONG Repeat{ 1 };
};

#define MELODY_DEVICE 0x8003

#define IOCTL_MELODY_PLAY \
    CTL_CODE(MELODY_DEVICE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

A note consists of a frequency (in Hertz) and duration to play. To make it a bit more interesting, a delay and repeat count are added. If Repeat is greater than one, the sound is played Repeat times, with a delay of Delay between repeats. Duration and Delay are provided in milliseconds.

The architecture we'll go for in the driver is to have a thread created when the first client opens a handle to our device, and that thread will perform the playback based on a queue of notes the driver manages. The thread will be shut down when the driver unloads.

It may seem asymmetric at this point - why not create the thread when the driver loads? As we shall see shortly, there is a little "snag" that we have to deal with that prevents creating the thread when the driver loads.

Let's start with `DriverEntry`. It needs to create a device object and a symbolic link. Here is the full function:

```
PlaybackState* g_State;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(RegistryPath);

    g_State = new (PagedPool) PlaybackState;
    if (g_State == nullptr)
        return STATUS_INSUFFICIENT_RESOURCES;
```

```

auto status = STATUS_SUCCESS;
PDEVICE_OBJECT DeviceObject = nullptr;
UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\KMelody");

do {
    UNICODE_STRING name = RTL_CONSTANT_STRING(L"\Device\KMelody");
    status = IoCreateDevice(DriverObject, 0, &name, FILE_DEVICE_UNKNOWN,
                           0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status))
        break;

    status = IoCreateSymbolicLink(&symLink, &name);
    if (!NT_SUCCESS(status))
        break;
} while (false);

if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "Error (0x%08X)\n", status));
    delete g_State;
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    return status;
}

DriverObject->DriverUnload = MelodyUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = MelodyCreateClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MelodyDeviceControl;

return status;
}

```

Most of the code should be familiar by now. The only new code is the creation of an object of type `PlaybackState`. The new C++ operator is overloaded as described earlier in this chapter. If allocating a `PlaybackState` instance fails, `DriverEntry` returns `STATUS_INSUFFICIENT_RESOURCES`, reporting a failure to the kernel.

The `PlaybackState` class is going to manage the list of notes to play and most other functionality specific to the driver. Here is its declaration (in `PlaybackState.h`):

```
struct PlaybackState {
    PlaybackState();
    ~PlaybackState();

    NTSTATUS AddNotes(const Note* notes, ULONG count);
    NTSTATUS Start(PVOID IoObject);
    void Stop();

private:
    static void PlayMelody(PVOID context);
    void PlayMelody();

    LIST_ENTRY m_head;
    FastMutex m_lock;
    PAGED_LOOKASIDE_LIST m_lookaside;
    KSEMAPHORE m_counter;
    KEVENT m_stopEvent;
    HANDLE m_hThread{ nullptr };
};
```

`m_head` is the head of the linked list holding the notes to play. Since multiple threads can access this list, it must be protected with a synchronization object. In this case, we'll go with a fast mutex. `FastMutex` is a wrapper class similar to the one we saw in chapter 6, with the added twist that it's initialized in its constructor rather than a separate `Init` method. This is convenient, and possible, because `PlaybackState` is allocated dynamically, causing its constructor to be invoked, along with constructors for data members (if any).

The note objects will be allocated from a lookaside list (`m_lookaside`), as each note has a fixed size, and there is a strong likelihood of many notes coming and going. `m_stopEvent` is an event object that will be used as a way to signal our playback thread to terminate. `m_hThread` is the playback thread handle. Finally, `m_counter` is a semaphore that is going to be used in a somewhat counter-intuitive way, its internal count indicating the number of notes in the queue.

As you can see, the event and semaphore don't have wrapper classes, so we need to initialize them in the `PlaybackState` constructor. Here is the constructor in full (in *PlaybackState.cpp*) with an addition of a type that is going to hold a single node:

```

struct FullNote : Note {
    LIST_ENTRY Link;
};

PlaybackState::PlaybackState() {
    InitializeListHead(&m_head);
    KeInitializeSemaphore(&m_counter, 0, 1000);
    KeInitializeEvent(&m_stopEvent, SynchronizationEvent, FALSE);
    ExInitializePagedLookasideList(&m_lookaside, nullptr, nullptr, 0,
        sizeof(FullNote), DRIVER_TAG, 0);
}

```

Here are the initialization steps taken by the constructor:

- Initialize the linked list to an empty list (`InitializeListHead`).
- Initialize the semaphore to a value of zero, meaning no notes are queued up at this point, with a maximum of 1000 queued notes. Of course, this number is arbitrary.
- Initialize the stop event as a `SynchronizationEvent` type in the non-signaled state (`KeInitializeEvent`). Technically, a `NotificationEvent` would have worked just as well, as just one thread will be waiting on this event as we'll see later.
- Initialize the lookaside list to managed paged pool allocations with size of `sizeof(FullNote)`. `FullNote` extends `Note` to include a `LIST_ENTRY` member, otherwise we can't store such objects in a linked list. The `FullNote` type should not be visible to user-mode, which is why it's defined privately in the driver's source files only.

`DRIVER_TAG` and `DRIVER_PREFIX` are defined in the file *KMelody.h*.

Before the driver finally unloads, the `PlaybackState` object is going to be destroyed, invoking its destructor:

```

PlaybackState::~PlaybackState() {
    Stop();
    ExDeletePagedLookasideList(&m_lookaside);
}

```

The call to `Stop` signals the playback thread to terminate as we'll see shortly. The only other thing left to do in terms of cleanup is to free the lookaside list.

The unload routine for the driver is similar to ones we've seen before with the addition of freeing the `PlaybackState` object:

```

void MelodyUnload(PDRIVER_OBJECT DriverObject) {
    delete g_State;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\KMelody");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}

```

The IRP_MJ_DEVICE_CONTROL handler is where notes provided by a client need to be added to the queue of notes to play. The implementation is pretty straightforward because the heavy lifting is performed by the `PlaybackState::AddNotes` method. Here is `MelodyDeviceControl` that validates the client's data and then invokes `AddNotes`:

```

NTSTATUS MelodyDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG info = 0;

    switch (dic.IoControlCode) {
        case IOCTL_MELODY_PLAY:
            if (dic.InputBufferLength == 0 ||
                dic.InputBufferLength % sizeof(Note) != 0) {
                status = STATUS_INVALID_BUFFER_SIZE;
                break;
            }
            auto data = (Note*)Irp->AssociatedIrp.SystemBuffer;
            if (data == nullptr) {
                status = STATUS_INVALID_PARAMETER;
                break;
            }

            status = g_State->AddNotes(data,
                dic.InputBufferLength / sizeof(Note));
            if (!NT_SUCCESS(status))
                break;
            info = dic.InputBufferLength;
            break;
    }
    return CompleteRequest(Irp, status, info);
}

```

`CompleteRequest` is a helper that we've seen before that completes the IRP with the given status and information:

```
NTSTATUS CompleteRequest(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS, ULONG_PTR info = 0);
//...
NTSTATUS CompleteRequest(PIRP Irp, NTSTATUS status, ULONG_PTR info) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}
```

`PlaybackState::AddNotes` needs to iterate over the provided notes. Here is the beginning of the function:

```
NTSTATUS PlaybackState::AddNotes(const Note* notes, ULONG count) {
    KdPrint((DRIVER_PREFIX "State::AddNotes %u\n", count));

    for (ULONG i = 0; i < count; i++) {
```

For each note, it needs to allocate a `FullNote` structure from the lookaside list:

```
auto fullNote = (FullNote*)ExAllocateFromPagedLookasideList(&m_lookaside);
if (fullNote == nullptr)
    return STATUS_INSUFFICIENT_RESOURCES;
```

If successful, the note data is copied to the `FullNote` and is added to the linked list under the protection of the fast mutex:

```
//
// copy the data from the Note structure
//
memcpy(fullNote, &notes[i], sizeof(Note));

//
// insert into the linked list
//
Locker locker(m_lock);
InsertTailList(&m_head, &fullNote->Link);
}
```

`Locker<T>` is the same type we looked at in chapter 6. The notes are inserted at the back of the list with `InsertTailList`. This is where we must provide a pointer to a `LIST_ENTRY` object, which is why `FullNote` objects are used instead of just `Note`. Finally, when the loop is completed, the semaphore must be incremented by the number of notes to indicate there are `count` more notes to play:

```

//  

// make the semaphore signaled (if it wasn't already) to  

// indicate there are new note(s) to play
//  

KeReleaseSemaphore(&m_counter, 2, count, FALSE);  

KdPrint((DRIVER_PREFIX "Semaphore count: %u\n",
    KeReadStateSemaphore(&m_counter)));

```

The value 2 used in KeReleaseSemaphore is the temporary priority boost a driver can provide to a thread that is released because of the semaphore becoming signaled (the same thing happens with the second parameter to IoCompleteRequest). I've chosen the value 2 arbitrarily. The value 0 (IO_NO_INCREMENT) is fine as well.

For debugging purposes, it may be useful to read the semaphore's count with KeReadStateSemaphore as was done in the above code. Here is the full function (without the comments):

```

NTSTATUS PlaybackState::AddNotes(const Note* notes, ULONG count) {
    KdPrint((DRIVER_PREFIX "State::AddNotes %u\n", count));

    for (ULONG i = 0; i < count; i++) {
        auto fullNote =
            (FullNote*)ExAllocateFromPagedLookasideList(&m_lookaside);
        if (fullNote == nullptr)
            return STATUS_INSUFFICIENT_RESOURCES;

        memcpy(fullNote, &notes[i], sizeof(Note));

        Locker locker(m_lock);
        InsertTailList(&m_head, &fullNote->Link);
    }
    KeReleaseSemaphore(&m_counter, 2, count, FALSE);
    KdPrint((DRIVER_PREFIX "Semaphore count: %u\n",
        KeReadStateSemaphore(&m_counter)));

    return STATUS_SUCCESS;
}

```

The next part to look at is handling IRP_MJ_CREATE and IRP_MJ_CLOSE. In earlier chapters, we just completed these IRPs successfully and that was it. This time, we need to create the playback thread when the first client opens a handle to our device. The initialization in DriverEntry points both indices to the same function, but the code is slightly different between the two. We could separate them to different functions, but if the difference is not great we might decide to handle both within the same function.

For IRP_MJ_CLOSE, there is nothing to do but complete the IRP successfully. For IRP_MJ_CREATE, we want to start the playback thread the first time the dispatch routine is invoked. Here is the code:

```
NTSTATUS MelodyCreateClose(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    auto status = STATUS_SUCCESS;
    if (IoGetCurrentIrpStackLocation(Irp)->MajorFunction == IRP_MJ_CREATE) {
        //
        // create the "playback" thread (if needed)
        //
        status = g_State->Start(DeviceObject);
    }
    return CompleteRequest(Irp, status);
}
```

The I/O stack location contains the IRP major function code we can use to make the distinction as required here. In the Create case, we call `PlaybackState::Start` with the device object pointer that would be used to keep the driver object alive as long as the thread is running. Let's see what that method looks like.

```
NTSTATUS PlaybackState::Start(PVOID IoObject) {
    Locker locker(m_lock);
    if (m_hThread)
        return STATUS_SUCCESS;

    return IoCreateSystemThread(
        IoObject,           // Driver or device object
        &m_hThread,         // resulting handle
        THREAD_ALL_ACCESS, // access mask
        nullptr,            // no object attributes required
        NtCurrentProcess(), // create in the current process
        nullptr,            // returned client ID
        PlayMelody,         // thread function
        this);              // passed to thread function
}
```

Acquiring the fast mutex ensures that a second thread is not created (as `m_hThread` would already be non-NUL). The thread is created with `IoCreateSystemThread`, which is preferred over `PsCreateSystemThread` because it ensures that the driver is not unloaded while the thread is executing (this does require Windows 8 or later).

The passed-in I/O object is the device object provided by the `IRP_MJ_CREATE` handler. The most common way of creating a thread by a driver is to run it in the context of the *System* process, as it normally should not be tied to a user-mode process. Our case, however, is more complicated because we intend to use the *Beep* driver to play the notes. The *Beep* driver needs to be able to handle multiple users (that might be connected to the same system), each one playing their own sounds. This is why when asked to play a note, the *Beep* driver plays in the context of the caller's session. If we create the thread in the *System* process, which is always part of session zero, we will not hear any sound, because session 0 is not an interactive user session.

This means we need to create our thread in the context of some process running under the caller's session - Using the caller's process directly (`NtCurrentProcess`) is the simplest way to get it working. You may frown at this, and rightly so, because the first process calling the driver to play something is going to have to host that thread for the lifetime of the driver. This has an unintended side effect: the process will not die. Even if it may seem to terminate, it will still show up in *Task Manager* with our thread being the single thread still keeping the process alive. We'll find a more elegant solution later in this chapter.

Yet another consequence of this arrangement is that we only handle one session - the first one where one of its processes happens to call the driver. We'll fix that as well later on.

The thread created starts running the `PlayMelody` function - a static function in the `PlaybackState` class. Callbacks must be global or static functions (because they are directly C function pointers), but in this case we would like to access the members of this instance of `PlaybackState`. The common trick is to pass the `this` pointer as the thread argument, and the callback simply invokes an instance method using this pointer:

```
// static function
void PlaybackState::PlayMelody(PVOID context) {
    ((PlaybackState*)context)->PlayMelody();
}
```

Now the instance method `PlaybackState::PlayMelody` has full access to the object's members.



There is another way to invoke the instance method without going through the intermediate static by using C++ *lambda functions*, as non-capturing lambdas are directly convertible to C function pointers:

```
IoCreateSystemThread(..., [](auto param) {
    ((PlaybackState*)param)->PlayMelody();
}, this);
```

The first

order of business in the new thread is to obtain a pointer to the *Beep* device using `IoGetDeviceObjectPointer`:

```
#include <ntddbeep.h>

void PlaybackState::PlayMelody() {
    PDEVICE_OBJECT beepDevice;
    UNICODE_STRING beepDeviceName = RTL_CONSTANT_STRING(DD_BEEP_DEVICE_NAME_U);
    PFILE_OBJECT beepFileObject;
    auto status = IoGetDeviceObjectPointer(&beepDeviceName, GENERIC_WRITE,
                                           &beepFileObject, &beepDevice);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to locate beep device (0x%X)\n",
                 status));
    }
}
```

```

    return;
}

```

The *Beep* device name is \Device\Beep as we've seen in chapter 2. Conveniently, the provided header *ntddbeep.h* declares everything we need in order to work with the device, such as the DD_BEEP_-DEVICE_NAME_U macro that defines the Unicode name.

At this point, the thread should loop around while it has notes to play and has not been instructed to terminate. This is where the semaphore and the event come in. The thread must wait until one of them is signaled. If it's the event, it should break out of the loop. If it's the semaphore, it means the semaphore's count is greater than zero, which in turn means the list of notes is not empty:

```

PVOID objects[] = { &m_counter, &m_stopEvent };
IO_STATUS_BLOCK ioStatus;
BEEP_SET_PARAMETERS params;

for (;;) {
    status = KeWaitForMultipleObjects(2, objects, WaitAny, Executive,
        KernelMode, FALSE, nullptr, nullptr);
    if (status == STATUS_WAIT_1) {
        KdPrint((DRIVER_PREFIX "Stop event signaled. Exiting thread...\n"));
        break;
    }

    KdPrint((DRIVER_PREFIX "Semaphore count: %u\n",
        KeReadStateSemaphore(&m_counter)));
}

```

The required function call is to `KeWaitForMultipleObjects` with the event and semaphore. They are put in an array, since this is the requirement for `KeWaitForMultipleObjects`. If the returned status is `STATUS_WAIT_1` (which is the same as `STATUS_WAIT_0 + 1`), meaning index number 1 is the signaled object, the loop is exited with a `break` instruction.

Now we need to extract the next note to play:

```

PLIST_ENTRY link;
{
    Locker locker(m_lock);
    link = RemoveHeadList(&m_head);
    NT_ASSERT(link != &m_head);
}

auto note = CONTAINING_RECORD(link, FullNote, Link);
KdPrint((DRIVER_PREFIX "Playing note Freq: %u Dur: %u Rep: %u Delay: %u\n",
    note->Frequency, note->Duration, note->Repeat, note->Delay));

```

We remove the head item from the list, and doing so under the fast mutex' protection. The assert ensures we are in a consistent state - remember that removing an item from an empty list returns the pointer to its head.

The actual `FullNote` pointer is retrieved with the help of the `CONTAINING_RECORD` macro, that moves the `LIST_ENTRY` pointer we received from `RemoveHeadList` to the containing `FullNode` that we are actually interested in.

The next step is to handle the note. If the note's frequency is zero, let's consider that as a "silence time" with the length provided by `Delay`:

```
if (note->Frequency == 0) {
    //
    // just do a delay
    //

    NT_ASSERT(note->Duration > 0);
    LARGE_INTEGER interval;
    interval.QuadPart = -10000LL * note->Duration;
    KeDelayExecutionThread(KernelMode, FALSE, &interval);
}
```

`KeDelayExecutionThread` is the rough equivalent of the `Sleep/SleepEx` APIs from user-mode. Here is its declaration:

```
NTSTATUS KeDelayExecutionThread (
    _In_ KPROCESSOR_MODE WaitMode,
    _In_ BOOLEAN Alertable,
    _In_ PLARGE_INTEGER Interval);
```

We've seen all these parameters as part of the wait functions. The most common invocation is with `KernelMode` and `FALSE` for `WaitMode` and `Alertable`, respectively. The interval is the most important parameter, where negative values mean relative wait in 100nsec units. Converting from milliseconds means multiplying by `-10000`, which is what you see in the above code.

If the frequency in the note is not zero, then we need to call the *Beep* driver with proper IRP. We already know that we need the `IOCTL_BEEP_SET` control code (defined in *ntddbeep.h*) and the `BEEP_SET_PARAMETERS` structure. All we need to do is build an IRP with the correct information using `IoBuildDeviceIoControlRequest`, and send it to the beep device with `IoCallDriver`:

```

else {
    params.Duration = note->Duration;
    params.Frequency = note->Frequency;
    int count = max(1, note->Repeat);

    KEVENT doneEvent;
    KeInitializeEvent(&doneEvent, NotificationEvent, FALSE);

    for (int i = 0; i < count; i++) {
        auto irp = IoBuildDeviceIoControlRequest(IOCTL_BEEP_SET, beepDevice,
            &params, sizeof(params),
            nullptr, 0, FALSE, &doneEvent, &ioStatus);
        if (!irp) {
            KdPrint((DRIVER_PREFIX "Failed to allocate IRP\n"));
            break;
        }

        status = IoCallDriver(beepDevice, irp);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "Beep device playback error (0x%X)\n",
                status));
            break;
        }
        if (status == STATUS_PENDING) {
            KeWaitForSingleObject(&doneEvent, Executive, KernelMode,
                FALSE, nullptr);
        }
    }
}

```

We loop around based on the Repeat member (which is usually 1). Then the IRP_MJ_DEVICE_CONTROL IRP is built with `IoBuildDeviceIoControlRequest`, supplying the frequency to play and the duration. Then, `IoCallDriver` is invoked with the *Beep* device pointer we obtained earlier, and the IRP. Unfortunately (or fortunately, depending on your perspective), the *Beep* driver just starts the operation, but does not wait for it to finish. It might (and in fact, always) returns `STATUS_PENDING` from the `IoCallDriver` call, which means the operation is not yet complete (the actual playing has not yet begun). Since we don't have anything else to do until then, the `doneEvent` event provided to `IoBuildDeviceIoControlRequest` is signaled automatically by the I/O manager when the operation completes - so we wait on the event.

Now that the sound is playing, we have to wait for the duration of that note with `KeDelayExecutionThread`:

```

LARGE_INTEGER delay;
delay.QuadPart = -10000LL * note->Duration;
KeDelayExecutionThread(KernelMode, FALSE, &delay);

```

Finally, if `Repeat` is greater than one, then we might need to wait between plays of the same note:

```
// perform the delay if specified,  
// except for the last iteration  
//  
if (i < count - 1 && note->Delay != 0) {  
    delay.QuadPart = -10000LL * note->Delay;  
    KeDelayExecutionThread(KernelMode, FALSE, &delay);  
}  
}  
}
```

At this point, the note data can be freed (or just returned to the lookaside list) and the code loops back to wait for the availability of the next note:

```
    ExFreeToPagedLookasideList(&m_lookaside, note);  
}
```

The loop continues until the thread is instructed to stop by signaling `stopEvent`, at which point it breaks from the infinite loop and cleans up by dereferencing the file object obtained from `IoGetDeviceObjectPointer`:

```
    ObDereferenceObject(beepFileObject);  
}
```

Here is the entire thread function for convenience (comments and KdPrint removed):

```
void PlaybackState::PlayMelody() {
    PDEVICE_OBJECT beepDevice;
    UNICODE_STRING beepDeviceName = RTL_CONSTANT_STRING(DD_BEEP_DEVICE_NAME_U);
    PFILE_OBJECT beepFileObject;
    auto status = IoGetDeviceObjectPointer(&beepDeviceName, GENERIC_WRITE,
                                           &beepFileObject, &beepDevice);
    if (!NT_SUCCESS(status)) {
        return;
    }

    PVOID objects[] = { &m_counter, &m_stopEvent };
    IO_STATUS_BLOCK ioStatus;
    BEEP_SET_PARAMETERS params;

    for (;;) {
        status = KeWaitForMultipleObjects(2, objects, WaitAny, Executive,
                                         KernelMode, FALSE, nullptr, nullptr);
        if (status == STATUS_WAIT_1) {
```

```
        break;
    }

PLIST_ENTRY link;
{
    Locker locker(m_lock);
    link = RemoveHeadList(&m_head);
    NT_ASSERT(link != &m_head);
}

auto note = CONTAINING_RECORD(link, FullNote, Link);
if (note->Frequency == 0) {
    NT_ASSERT(note->Duration > 0);
    LARGE_INTEGER interval;
    interval.QuadPart = -10000LL * note->Duration;
    KeDelayExecutionThread(KernelMode, FALSE, &interval);
}
else {
    params.Duration = note->Duration;
    params.Frequency = note->Frequency;
    int count = max(1, note->Repeat);

    KEVENT doneEvent;
    KeInitializeEvent(&doneEvent, SynchronizationEvent, FALSE);

    for (int i = 0; i < count; i++) {
        auto irp = IoBuildDeviceIoControlRequest(IOCTL_BEEP_SET,
            beepDevice, &params, sizeof(params),
            nullptr, 0, FALSE, &doneEvent, &ioStatus);
        if (!irp) {
            break;
        }
        NT_ASSERT(irp->UserEvent == &doneEvent);

        status = IoCallDriver(beepDevice, irp);
        if (!NT_SUCCESS(status)) {
            break;
        }
        if (status == STATUS_PENDING) {
            KeWaitForSingleObject(&doneEvent, Executive,
                KernelMode, FALSE, nullptr);
        }
    }

    LARGE_INTEGER delay;
```

```

        delay.QuadPart = -10000LL * note->Duration;
        KeDelayExecutionThread(KernelMode, FALSE, &delay);

        if (i < count - 1 && note->Delay != 0) {
            delay.QuadPart = -10000LL * note->Delay;
            KeDelayExecutionThread(KernelMode, FALSE, &delay);
        }
    }
}
ExFreeToPagedLookasideList(&m_lookaside, note);
}
ObDereferenceObject(beepFileObject);
}
}

```

The last piece of the puzzle is the `PlaybackState::Stop` method that signals the thread to exit:

```

void PlaybackState::Stop() {
    if (m_hThread) {
        //
        // signal the thread to stop
        //
        KeSetEvent(&m_stopEvent, 2, FALSE);

        //
        // wait for the thread to exit
        //
        PVOID thread;
        auto status = ObReferenceObjectByHandle(m_hThread, SYNCHRONIZE,
                                                *PsThreadType, KernelMode, &thread, nullptr);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "ObReferenceObjectByHandle error (0x%X)\n",
                     status));
        }
        else {
            KeWaitForSingleObject(thread, Executive, KernelMode, FALSE, nullptr\
);
            ObDereferenceObject(thread);
        }
        ZwClose(m_hThread);
        m_hThread = nullptr;
    }
}

```

If the thread exists (`m_hThread` is non-NULL), then we set the event (`KeSetEvent`). Then we wait for the thread to actually terminate. This is technically unnecessary because the thread was created with `IoCreateSystemThread`, so there is no danger the driver is unloaded prematurely. Still, it's worthwhile showing how to get the pointer to the thread object given a handle (since `KeWaitForSingleObject` requires an object). It's important to remember to call `ObDereferenceObject` once we don't need the pointer anymore, or the thread object will remain alive forever (keeping its process and other resources alive as well).

Client Code

Here are some examples for invoking the driver (error handling omitted):

```
#include <Windows.h>
#include <stdio.h>
#include "..\KMelody\MelodyPublic.h"

int main() {
    HANDLE hDevice = CreateFile(MELODY_SYMLINK, GENERIC_WRITE, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);

    Note notes[10];
    for (int i = 0; i < _countof(notes); i++) {
        notes[i].Frequency = 400 + i * 30;
        notes[i].Duration = 500;
    }
    DWORD bytes;
    DeviceIoControl(hDevice, IOCTL_MELODY_PLAY, notes, sizeof(notes),
        nullptr, 0, &bytes, nullptr);

    for (int i = 0; i < _countof(notes); i++) {
        notes[i].Frequency = 1200 - i * 100;
        notes[i].Duration = 300;
        notes[i].Repeat = 2;
        notes[i].Delay = 300;
    }
    DeviceIoControl(hDevice, IOCTL_MELODY_PLAY, notes, sizeof(notes),
        nullptr, 0, &bytes, nullptr);

    CloseHandle(hDevice);
    return 0;
}
```

I recommend you build the driver and the client and test them. The project names are *KMelody* and *Melody* in the solution for this chapter. Build your own music!



1. Replace the call to `IoCreateSystemThread` with `PsCreateSystemThread` and make the necessary adjustments.
2. Replace the lookaside list API with the newer API.

Invoking System Services

System Services (system calls) are normally invoked indirectly from user mode code. For example, calling the Windows `CreateFile` API in user mode invokes `NtCreateFile` from `NtDll.Dll`, which is a system call. This call traverses the user/kernel boundary, eventually calling the “real” `NtCreateFile` implementation within the executive.

We already know that drivers can invoke system calls as well, using the `Nt` or the `Zw` variant (which sets the previous execution mode to `KernelMode` before invoking the system call). Some of these system calls are fully documented in the driver kit, such as `NtCreateFile/ZwCreateFile`. Others, however, are not documented or sometimes partially documented.

For example, enumerating processes in the system is fairly easy to do from user-mode - in fact, there are several APIs one can use for this purpose. They all invoke the `NtQuerySystemInformation` system call, which is not officially documented in the WDK. Ironically, it’s provided in the user-mode header `Winternl.h` like so:

```
NTSTATUS NtQuerySystemInformation (
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
    OUT PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength OPTIONAL);
```

The macros `IN` and `OUT` expand to nothing. These were used in the old days before SAL was invented to provide some semantics for developers. For some reason, `Winternl.h` uses these macros rather than the modern SAL annotations.

We can copy this definition and tweak it a bit by turning it into its Zw variant, more suitable for kernel callers. The SYSTEM_INFORMATION_CLASS enumeration and associated data structures are the real data we're after. Some values are provided in user-mode and/or kernel-mode headers. Most of the values have been “reversed engineered” and can be found in open source projects, such as *Process Hacker*². Although these APIs might not be officially documented, they are unlikely to change as Microsoft's own tools depend on many of them.

If the API in question only exists in certain Windows versions, it's possible to query dynamically for the existence of a kernel API with `MmGetSystemRoutineAddress`:

```
PVOID MmGetSystemRoutineAddress (_In_ PUNICODE_STRING SystemRoutineName);
```

You can think of `MmGetSystemRoutineAddress` as the kernel-mode equivalent of the user-mode `GetProcAddress` API.

Another very useful API is `NtQueryInformationProcess`, also defined in *Winternl.h*:

```
NTAPI NtQueryInformationProcess (
    IN HANDLE ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID ProcessInformation,
    IN ULONG ProcessInformationLength,
    OUT PULONG ReturnLength OPTIONAL);
```

Curiously enough, the kernel-mode headers provide many of the `PROCESSINFOCLASS` enumeration values, along with their associated data structures, but not the definition of this system call itself. Here is a partial set of values for `PROCESSINFOCLASS`:

```
typedef enum _PROCESSINFOCLASS {
    ProcessBasicInformation = 0,
    ProcessDebugPort = 7,
    ProcessWow64Information = 26,
    ProcessImageFileName = 27,
    ProcessBreakOnTermination = 29
} PROCESSINFOCLASS;
```



A more complete list is available in *ntddk.h*. A full list is available within the *Process Hacker* project.

The following example shows how to query the current process image file name. `ProcessImageFileName` seems to be the way to go, and it expects a `UNICODE_STRING` as the buffer:

²<https://github.com/processhacker/phnt>

```
ULONG size = 1024;
auto buffer = ExAllocatePoolWithTag(PagedPool, size, DRIVER_TAG);
auto status = ZwQueryInformationProcess(NtCurrentProcess(),
    ProcessImageFileName, buffer, size, nullptr);
if(NT_SUCCESS(status)) {
    auto name = (UNICODE_STRING*)buffer;
    // do something with name...
}
ExFreePool(buffer);
```

Example: Enumerating Processes

The *EnumProc* driver shows how to call `ZwQuerySystemInformation` to retrieve the list of running processes. `DriverEntry` calls the `EnumProcesses` function that does all the work and dumps information using simple `DbgPrint` calls. Then `DriverEntry` returns an error so the driver is unloaded.

First, we need the definition of `ZwQuerySystemInformation` and the required enum value and structure which we can copy from *Winternl.h*:

```
#include <ntddk.h>

// copied from <Winternl.h>
enum SYSTEM_INFORMATION_CLASS {
    SystemProcessInformation = 5,
};

typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    UCHAR Reserved1[48];
    UNICODE_STRING ImageName;
    KPRIORITY BasePriority;
    HANDLE UniqueProcessId;
    PVOID Reserved2;
    ULONG HandleCount;
    ULONG SessionId;
    PVOID Reserved3;
    SIZE_T PeakVirtualSize;
    SIZE_T VirtualSize;
    ULONG Reserved4;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    PVOID Reserved5;
```

```

SIZE_T QuotaPagedPoolUsage;
PVOID Reserved6;
SIZE_T QuotaNonPagedPoolUsage;
SIZE_T PagefileUsage;
SIZE_T PeakPagefileUsage;
SIZE_T PrivatePageCount;
LARGE_INTEGER Reserved7[6];
} SYSTEM_PROCESS_INFORMATION, * PSYSTEM_PROCESS_INFORMATION;

```

```

extern "C" NTSTATUS ZwQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS info,
    PVOID buffer,
    ULONG size,
    PULONG len);

```

Notice there are lots of “reserved” members in `SYSTEM_PROCESS_INFORMATION`. We’ll manage with what we get, but you can find the full data structure in the *Process Hacker* project.

`EnumProcesses` starts by querying the number of bytes needed by calling `ZwQuerySystemInformation` with a null buffer and zero size, getting the last parameter as the required size:

```

void EnumProcesses() {
    ULONG size = 0;
    ZwQuerySystemInformation(SystemProcessInformation, nullptr, 0, &size);
    size += 1 << 12;      // 4KB, just to make sure the next call succeeds
}

```

We want to allocate some more in case new processes are created between this call and the next “real” call. We can write the code in a more robust way and have a loop that queries until the size is large enough, but the above solution is robust enough for most purposes.

Next, we allocate the required buffer and make the call again, this time with the real buffer:

```

auto buffer = ExAllocatePoolWithTag(PagedPool, size, 'cprP');
if (!buffer)
    return;

if (NT_SUCCESS(ZwQuerySystemInformation(SystemProcessInformation,
    buffer, size, nullptr))) {

```

if the call succeeds, we can start iterating. The returned pointer is to the first process, where the next process is located `NextEntryOffset` bytes from this offset. The enumeration ends when `NextEntryOffset` is zero:

```

auto info = (SYSTEM_PROCESS_INFORMATION*)buffer;
ULONG count = 0;
for (;;) {
    DbgPrint("PID: %u Session: %u Handles: %u Threads: %u Image: %wZ\n",
        HandleToULong(info->UniqueProcessId),
        info->SessionId, info->HandleCount,
        info->NumberOfThreads, info->ImageName);
    count++;
    if (info->NextEntryOffset == 0)
        break;

    info = (SYSTEM_PROCESS_INFORMATION*)((PUCHAR)info + info->NextEntryOffset);
}
DbgPrint("Total Processes: %u\n", count);

```

We output some of the details provided in the `SYSTEM_PROCESS_INFORMATION` structure and count the number of processes while we're at it. The only thing left to do in this simple example is to clean up:

```

}
ExFreePool(buffer);
}

```

As mentioned, `DriverEntry` is simple:

```

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);

    EnumProcesses();
    return STATUS_UNSUCCESSFUL;
}

```

Given this knowledge, we can make the *KMelody* driver a bit better by creating our thread in a `Csrss.exe` process for the current session, instead of the first client process that comes in. This is better, since `Csrss` always exists, and is in fact a *critical process* - one that if killed for whatever reason, causes the system to crash.

Killing `Csrss` is not easy, since it's a protected process starting with Windows 8.1, but kernel code can certainly do that.



1. Modify the *KMelody* driver to create the thread in a *Csrss* process for the current session. Search for *Csrss* with *ZwQuerySystemInformation* and create the thread in that process.
2. Add support for multiple sessions, where there is one playback thread per session. Hint: call *ZwQueryInformationProcess* with *ProcessSessionId* to find out the session a process is part of. Manage a list of *PlaybackState* objects, one for each session. You can also use the undocumented (but exported) *PsGetCurrentProcessSessionId* API.

Summary

In this chapter, we were introduced to some programming techniques that are useful in many types of drivers. We're not done with these techniques - there will be more in chapter 11. But for now, we can begin using some kernel-provided notifications, starting with Process and Thread notifications in the next chapter.

Chapter 9: Process and Thread Notifications

One of the powerful mechanisms available for kernel drivers is the ability to be notified when certain important events occur. In this chapter, we'll look into some of these events, namely process creation and destruction, thread creation and destruction, and image loads.

In this chapter:

- **Process Notifications**
 - **Implementing Process Notifications**
 - **Providing Data to User Mode**
 - **Thread Notifications**
 - **Image Load Notifications**
 - **Remote Thread Detection**
-

Process Notifications

Whenever a process is created or destroyed, interested drivers can be notified by the kernel of that fact. This allows drivers to keep track of processes, possibly associating some data with these processes. At the very minimum, these allow drivers to monitor process creation/destruction in real-time. By “real-time” I mean that the notifications are sent “in-line”, as part of process creation; the driver cannot miss any processes that may be created and destroyed quickly.

For process creation, drivers also have the power to stop the process from being fully created, returning an error to the caller that initiated process creation. This kind of power can only be directly achieved in kernel mode.

Windows provides other mechanisms for being notified when processes are created or destroyed. For example, using *Event Tracing for Windows* (ETW), such notifications can be received by a user-mode process (running with elevated privileges). However, there is no way to prevent a process from being created. Furthermore, ETW has an inherent notification delay of about 1-3 seconds (it uses internal buffers for performance reasons), so a short-lived process may exit before the creation notification arrives. Opening a handle to the created process at that time would no longer be possible.

The main API for registering for process notifications is `PsSetCreateProcessNotifyRoutineEx`, defined like so:

```
NTSTATUS PsSetCreateProcessNotifyRoutineEx (
    _In_ PCREATE_PROCESS_NOTIFY_ROUTINE_EX NotifyRoutine,
    _In_ BOOLEAN Remove);
```



There is currently a system-wide limit of 64 registrations, so it's theoretically possible for the registration function to fail.

The first argument is the driver's callback routine, having the following prototype:

```
void ProcessNotifyCallback(
    _Inout_     PEPPROCESS Process,
    _In_        HANDLE ProcessId,
    _Inout_opt_ PPS_CREATE_NOTIFY_INFO CreateInfo);
```

The second argument to `PsSetCreateProcessNotifyRoutineEx` indicates whether the driver is registering or unregistering the callback (FALSE indicates the former). Typically, a driver will call this API with FALSE in its `DriverEntry` routine and call the same API with TRUE in its `Unload` routine.

The parameters to the process notification routine are as follows:

- *Process* - the process object of the newly created process, or the process being destroyed.
- *Process Id* - the unique process ID of the process. Although it's declared with type `HANDLE`, it's in fact an ID.
- *CreateInfo* - a structure that contains detailed information on the process being created. If the process is being destroyed, this argument is NULL.

For process creation, the driver's callback routine is executed by the creating thread (running as part of the creating process). For process exit, the callback is executed by the last thread to exit the process. In both cases, the callback is called inside a critical region (where normal kernel APCs are disabled).

Starting with Windows 10 version 1607, there is another function for process notifications: `PsSetCreateProcessNotifyR`. This “extended” function sets up a callback similar to the previous one, but the callback is also invoked for *Pico processes*. Pico processes are those used to host Linux processes for the *Windows Subsystem for Linux* (WSL) version 1. If a driver is interested in such processes, it must register with the extended function.

A driver using these callbacks must have the IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY flag in its *Portable Executable* (PE) image header. Without it, the call to the registration function returns STATUS_ACCESS_DENIED (unrelated to driver test signing mode). Currently, Visual Studio does not provide UI for setting this flag. It must be set in the linker command-line options with /integritycheck. Figure 9-1 shows the project properties where this setting is specified.

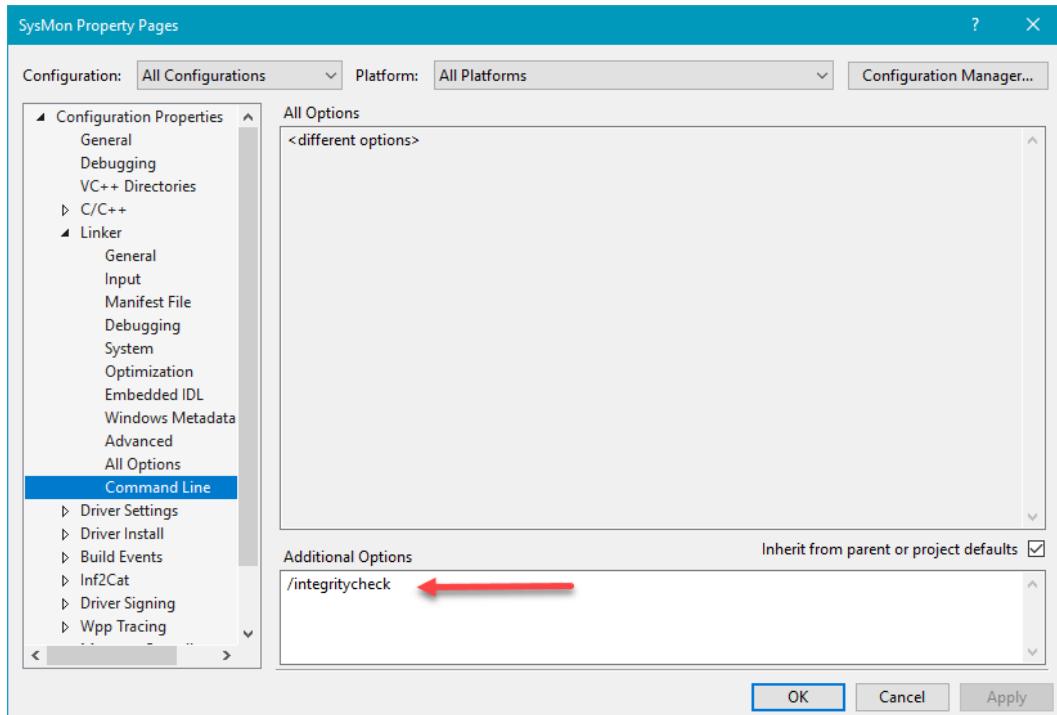


Figure 9-1: /integritycheck linker switch in Visual Studio

The data structure provided for process creation is defined like so:

```
typedef struct _PS_CREATE_NOTIFY_INFO {
    _In_ SIZE_T Size;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG FileOpenNameAvailable : 1;
            _In_ ULONG IsSubsystemProcess : 1;
            _In_ ULONG Reserved : 30;
        };
    };
};
```

```

_In_ HANDLE ParentProcessId;
_In_ CLIENT_ID CreatingThreadId;
_Inout_ struct _FILE_OBJECT *FileObject;
_In_ PCUNICODE_STRING ImageFileName;
_In_opt_ PCUNICODE_STRING CommandLine;
_Inout_ NTSTATUS CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;

```

Here is a description of the important fields in this structure:

- *CreatingThreadId* - a combination of thread and process Id of the creator of the process.
- *ParentProcessId* - the parent process ID (not a handle). This process is usually the same as provided by CreateThreadId.UniqueProcess, but may be different, as it's possible, as part of process creation, to pass in a different parent process to inherit certain properties from. See the user-mode documentation for UpdateProcThreadAttribute with the PROC_THREAD_ATTRIBUTE_PARENT_PROCESS attribute.
- *ImageFileName* - the image file name of the executable, available if the flag FileOpenNameAvailable is set.
- *CommandLine* - the full command line used to create the process. Note that in some cases it may be NULL.
- *IsSubsystemProcess* - this flag is set if this process is a Pico process. This can only happen if the driver registered using PsSetCreateProcessNotifyRoutineEx2.
- *CreationStatus* - this is the status that would return to the caller. It's set to STATUS_SUCCESS when the callback is invoked. This is where the driver can stop the process from being created by placing some failure status in this member (e.g. STATUS_ACCESS_DENIED). If the driver fails the creation, subsequent drivers that may have set up their own callbacks will not be called.

Implementing Process Notifications

To demonstrate process notifications, we'll build a driver that gathers information on process creation and destruction and allow this information to be consumed by a user-mode client. This is similar to tools such as *Process Monitor* and *SysMon* from *Sysinternals*, which use process and thread notifications for reporting process and thread activity. During the course of implementing this driver, we'll leverage some of the techniques we learned in previous chapters.

Our driver name is going to be *SysMon* (unrelated to the *SysMon* tool). It will store all process creation/destruction information in a linked list. Since this linked list may be accessed concurrently by multiple threads, we need to protect it with a mutex or a fast mutex; we'll go with fast mutex, as it's slightly more efficient.

The data we gather will eventually find its way to user mode, so we should declare common structures that the driver produces and a user-mode client consumes. We'll add a common header file named *SysMonPublic.h* to the driver project and define a few structures. We start with a common header for all information structures we need to collect:

```

enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit
};

struct ItemHeader {
    ItemType Type;
    USHORT Size;
    LARGE_INTEGER Time;
};

```



The `ItemType` enum defined above uses the C++ 11 *scoped enum* feature, where enum values have a scope (`ItemType` in this case). These enums can also have a non-int size - `short` in the example. If you're using C, you can use classic enums, or even `#defines` if you prefer.

The `ItemHeader` structure holds information common to all event types: the type of the event, the time of the event (expressed as a 64-bit integer), and the size of the payload. The size is important, as each event has its own information. If we later wish to pack an array of these events and (say) provide them to a user-mode client, the client needs to know where each event ends and the next one begins.

Once we have this common header, we can derive other data structures for particular events. Let's start with the simplest - process exit:

```

struct ProcessExitInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ExitCode;
};

```

For process exit event, there is just one interesting piece of information (besides the header and the thread ID) - the exit status (code) of the process. This is normally the value returned from a user-mode `main` function.



If you're using C, then inheritance is not available to you. However, you can simulate it by having the first member be of type `ItemHeader` and then adding the specific members; The memory layout is the same.

```

struct ProcessExitInfo {
    ItemHeader Header;
    ULONG ProcessId;
};

```

The type used for a process ID is `ULONG` - process IDs (and thread IDs) cannot be larger than 32-bit. `HANDLE` is not a good idea, as user mode may be confused by it. Also, `HANDLE` has a different size in a 32-bit process as opposed to a 64-bit process, so it's best to avoid "bitness"-affected members. If you're familiar with user-mode programming, `DWORD` is a common `typedef` for a 32-bit unsigned integer. It's not used here because `DWORD` is not defined in the WDK headers. Although it's pretty easy to define it explicitly, it's simpler just to use `ULONG`, which means the same thing and is defined in user-mode and kernel-mode headers.

Since we need to store every such structure as part of a linked list, each data structure must contain a `LIST_ENTRY` instance that points to the next and previous items. Since these `LIST_ENTRY` objects should not be exposed to user-mode, we will define extended structures containing these entries in a different file, that is not shared with user-mode.

There are several ways to define a "bigger" structure to hold the `LIST_ENTRY`. One way is to create templated type that has a `LIST_ENTRY` at the beginning (or end) like so:

```
template<typename T>
struct FullItem {
    LIST_ENTRY Entry;
    T Data;
};
```

The layout of `FullItem<T>` is shown in figure 9-2.

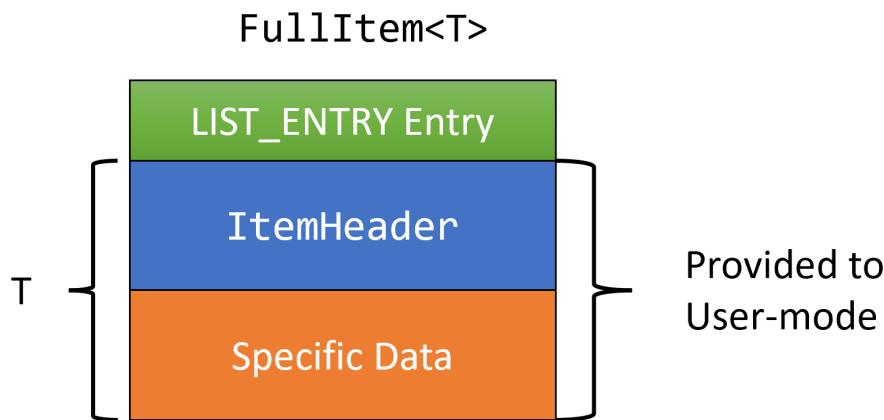


Figure 9-2: `FullItem<T>` layout

A templated class is used to avoid creating a multitude of types, one for each specific event type. For example, we could create the following structure specifically for a process exit event:

```
struct FullProcessExitInfo {
    LIST_ENTRY Entry;
    ProcessExitInfo Data;
};
```

We could even inherit from `LIST_ENTRY` and then just add the `ProcessExitInfo` structure. But this is not elegant, as our data has nothing to do with `LIST_ENTRY`, so inheriting from it is artificial and should be avoided.

The `FullItem<T>` type saves the hassle of creating these individual types.



IF you're using C, then templates are not available, and you must use the above structure approach. I'm not going to mention C again in this chapter - there is always a workaround that can be used if you have to.

Another way to accomplish something similar, without using templates is by using a union to hold on to all the possible variants. For example:

```
struct ItemData : ItemHeader {
    union {
        ProcessCreateInfo ProcessCreate;      // TBD
        ProcessExitInfo ProcessExit;
    };
};
```

Then we just extend the list of data members in the union. The full item would be just a simple extension:

```
struct FullItem {
    LIST_ENTRY Entry;
    ItemData Data;
};
```

The rest of the code uses the first option (with the template). The reader is encouraged to try the second option.

The head of our linked list must be stored somewhere. We'll create a data structure that will hold all the global state of the driver, instead of creating separate global variables. Here is the definition of our structure (in `Globals.h` in the smaple code for this chapter):

```
#include "FastMutex.h"

struct Globals {
    void Init(ULONG maxItems);
    bool AddItem(LIST_ENTRY* entry);
    LIST_ENTRY* RemoveItem();

private:
    LIST_ENTRY m_ItemsHead;
    ULONG m_Count;
    ULONG m_MaxCount;
    FastMutex m_Lock;
};

};
```

The `FastMutex` type used is the same one we developed in chapter 6.

`Init` is used to initialize the data members of the structure. Here is its implementation (in `Globals.cpp`):

```
void Globals::Init(ULONG maxCount) {
    InitializeListHead(&m_ItemsHead);
    m_Lock.Init();
    m_Count = 0;
    m_MaxCount = maxCount;
}
```

`m_MaxCount` holds the maximum number of elements in the linked list. This will be used to prevent the list from growing arbitrarily large if a client does not request data for a while. `m_Count` holds the current number of items in the list. The list itself is initialized with the normal `InitializeListHead` API. Finally, the fast mutex is initialized by invoking its own `Init` method as implemented in chapter 6.

The DriverEntry Routine

The `DriverEntry` for the `SysMon` driver is similar to the one in the `Zero` driver from chapter 7. We have to add process notification registration and proper initialization of our `Globals` object:

```
// in SysMon.cpp
Globals g_State;

extern "C"
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    auto status = STATUS_SUCCESS;

    PDEVICE_OBJECT DeviceObject = nullptr;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\sysmon");
    bool symLinkCreated = false;

    do {
        UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\sysmon");
        status = IoCreateDevice(DriverObject, 0, &devName,
                               FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to create device (0x%08X)\n",
                     status));
            break;
        }
        DeviceObject->Flags |= DO_DIRECT_IO;

        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "failed to create sym link (0x%08X)\n",
                     status));
            break;
        }
        symLinkCreated = true;

        status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX
                     "failed to register process callback (0x%08X)\n",
                     status));
            break;
        }
    } while (false);

    if (!NT_SUCCESS(status)) {
        if (symLinkCreated)
            IoDeleteSymbolicLink(&symLink);
        if (DeviceObject)
```

```

        IoDeleteDevice(DeviceObject);
    return status;
}

g_State.Init(10000);           // hard-coded limit for now

DriverObject->DriverUnload = SysMonUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = SysMonCreateClose;
DriverObject->MajorFunction[IRP_MJ_READ] = SysMonRead;

return status;
}

```

The device object's flags are adjusted to use Direct I/O for read/write operations (DO_DIRECT_IO). The device is created as exclusive, so that only a single client can exist to the device. This makes sense, otherwise multiple clients might be getting data from the device, which would mean each client getting parts of the data. In this case, I decided to prevent that by creating the device as exclusive (TRUE value in the second to last argument). We'll use the read dispatch routine to return event information to a client.

The create and close dispatch routines are handled in the simplest possible way - just completing them successfully, with the help of `CompleteRequest` we have encountered before:

```

NTSTATUS CompleteRequest(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS, ULONG_PTR info = 0) {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return status;
}

NTSTATUS SysMonCreateClose(PDEVICE_OBJECT, PIRP Irp) {
    return CompleteRequest(Irp);
}

```

Handling Process Exit Notifications

The process notification function in the code above is `OnProcessNotify` and has the prototype outlined earlier in this chapter. This callback handles process creation and exit. Let's start with process exit, as it's much simpler than process creation (as we shall soon see). The basic outline of the callback is as follows:

```
void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFOCreateInfo) {
    if (CreateInfo) {
        // process create
    }
    else {
        // process exit
    }
}
```

For process exit we have just the process ID we need to save, along with the header data common to all events. First, we need to allocate storage for the full item representing this event:

```
auto info = (FullItem<ProcessExitInfo>*)ExAllocatePoolWithTag(PagedPool,
    sizeof(FullItem<ProcessExitInfo>), DRIVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "failed allocation\n"));
    return;
}
```

If the allocation fails, there is really nothing the driver can do, so it just returns from the callback. Now it's time to fill the generic information: time, item type and size, all of which are easy to get:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessExit;
item.Size = sizeof(ProcessExitInfo);
item.ProcessId = HandleToULong(ProcessId);
item.ExitCode = PsGetProcessExitStatus(Process);

PushItem(&info->Entry);
```

First, we dig into the data item itself (bypassing the `LIST_ENTRY`) with the `item` variable. Next, we fill the header information: The item type is well-known, since we are in the branch handling a process exit notification; the time can be obtained with `KeQuerySystemTimePrecise` that returns the current system time (UTC, not local time) as a 64-bit integer counting from January 1, 1601 at midnight Universal Time. Finally, the item size is constant and is the size of the user-facing data structure (not the size of the `FullItem<ProcessExitInfo>`).



Notice the `item` variable is a reference to the data; without the reference (`&`), a copy would have been created, which is **not** what we want.



The KeQuerySystemTimePrecise API is available starting with Windows 8. For earlier versions, the KeQuerySystemTime API should be used instead.

The specific data for a process exit event consists of the process ID and the exit code. The process ID is provided directly by the callback itself. The only thing to do is call `HandleToULong` so the correct cast is used to turn a HANDLE value into an unsigned 32-bit integer. The exit code is not given directly, but it's easy to retrieve with `PsGetProcessExitStatus`:

```
NTSTATUS PsGetProcessExitStatus(_In_ PEPPROCESS Process);
```

All that's left to do now is add the new item to the end of our linked list. For this purpose, we'll define and implement a function named `AddItem` in the `Globals` class:

```
void Globals::AddItem(LIST_ENTRY* entry) {
    Locker locker(m_Lock);
    if (m_Count == m_MaxCount) {
        auto head = RemoveHeadList(&m_ItemsHead);
        ExFreePool(CONTAINING_RECORD(head,
            FullItem<ItemHeader>, Entry));
        m_Count--;
    }

    InsertTailList(&m_ItemsHead, entry);
    m_Count++;
}
```

`AddItem` uses the `Locker<T>` we saw in earlier chapters to acquire the fast mutex (and release it when the variable goes out of scope) before manipulating the linked list. Remember to set the C++ standard to C++ 17 at least in the project's properties so that `Locker` can be used without explicitly specifying the type it works on (the compiler makes the inference).

We'll add new items to the tail of the list. If the number of items in the list is at its maximum, the function removes the first item (from the head) and frees it with `ExFreePool`, decrementing the item count.

This is not the only way to handle the case where the number of items is too large. Feel free to use other ways. A more "precise" way might be tracking the number of bytes used, rather than number of items, because each item is different in size.



We don't need to use atomic increment/decrement operations in the `AddItem` function because manipulation of the item count is always done under the protection of the fast mutex.

With `AddItem` implemented, we can call it from our process notify routine:

```
g_State.AddItem(&info->Entry);
```



Implement the limit by reading from the registry in `DriverEntry`. Hint: you can use APIs such as `ZwOpenKey` or `IoOpenDeviceRegistryKey` and then `ZwQueryValueKey`. We'll look at these APIs more closely in chapter 11.

Handling Process Create Notifications

Process create notifications are more complex because the amount of information varies. The command line length is different for different processes. First we need to decide what information to store for process creation. Here is a first try:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    WCHAR CommandLine[1024];
};
```

We choose to store the process ID, the parent process ID and the command line. Although this structure can work and is fairly easy to deal with because its size is known in advance.



What might be an issue with the above declaration?

The potential issue here is with the command line. Declaring the command line with constant size is simple, but not ideal. If the command line is longer than allocated, the driver would have to trim it, possibly hiding important information. If the command line is shorter than the defined limit, the structure is wasting memory.



Can we use something like this?

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    UNICODE_STRING CommandLine; // can this work?
};
```

This can-

not work. First, `UNICODE_STRING` is not normally defined in user mode headers. Secondly (and much worse), the internal pointer to the actual characters normally would point to system space, inaccessible to user-mode. Thirdly, how would that string be eventually freed?

Here is another option, which we'll use in our driver:

```
struct ProcessCreateInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ParentProcessId;
    ULONG CreatingThreadId;
    ULONG CreatingProcessId;
    USHORT CommandLineLength;
    WCHAR CommandLine[1];
};
```

We'll store the command line length and copy the actual characters at the end of the structure, starting from `CommandLine`. The array size is specified as 1 just to make it easier to work with in the code. The actual number of characters is provided by `CommandLineLength`.

Given this declaration, we can begin implementation for process creation (`CreateInfo` is non-NULL):

```
USHORT allocSize = sizeof(FullItem<ProcessCreateInfo>);
USHORT commandLineSize = 0;
if (CreateInfo->CommandLine) {
    commandLineSize = CreateInfo->CommandLine->Length;
    allocSize += commandLineSize;
}
auto info = (FullItem<ProcessCreateInfo>*)ExAllocatePoolWithTag(
    PagedPool, allocSize, DRIVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "failed allocation\n"));
    return;
}
```

The total size for an allocation is based on the command line length (if any). Now it's time to fill in the fixed-size details:

```
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessCreate;
item.Size = sizeof(ProcessCreateInfo) + commandLineSize;
item.ProcessId = HandleToULong(ProcessId);
item.ParentProcessId = HandleToULong(CreateInfo->ParentProcessId);
item.CreatingProcessId = HandleToULong(
    CreateInfo->CreatingThreadId.UniqueProcess);
item.CreatingThreadId = HandleToULong(
    CreateInfo->CreatingThreadId.UniqueThread);
```

The item size must be calculated to include the command line length.

Next, we need to copy the command line to the address where `CommandLine` begins, and set the correct command line length:

```

if (commandLineSize > 0) {
    memcpy(item.CommandLine, CreateInfo->CommandLine->Buffer, commandLineSize);
    item.CommandLength = commandLineSize / sizeof(WCHAR); // len in WCHARs
}
else {
    item.CommandLength = 0;
}
g_State.AddItem(&info->Entry);

```

The command line length is stored in characters, rather than bytes. This is not mandatory, of course, but would probably be easier to use by user mode code. Notice the command line is not NULL terminated - it's up to the client not read too many characters. As an alternative, we can make the string null terminated to simplify client code. In fact, if we do that, the command line length is not even needed.



Make the command line NULL-terminated and remove the command line length.



Astute readers may notice that the calculated data length is actually one character longer than needed, perfect for adding a NULL-terminator. Why? `sizeof(ProcessCreateInfo)` includes one character of the command line.

For easier reference, here is the complete process notify callback implementation:

```

void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo) {
    if (CreateInfo) {
        USHORT allocSize = sizeof(FullItem<ProcessCreateInfo>);
        USHORT commandLineSize = 0;
        if (CreateInfo->CommandLine) {
            commandLineSize = CreateInfo->CommandLine->Length;
            allocSize += commandLineSize;
        }
        auto info = (FullItem<ProcessCreateInfo>*)ExAllocatePoolWithTag(
            PagedPool, allocSize, DRIVER_TAG);
        if (info == nullptr) {
            KdPrint((DRIVER_PREFIX "failed allocation\n"));
            return;
        }

        auto& item = info->Data;
    }
}

```

```
KeQuerySystemTimePrecise(&item.Time);
item.Type = ItemType::ProcessCreate;
item.Size = sizeof(ProcessCreateInfo) + commandLineSize;
item.ProcessId = HandleToULong(ProcessId);
item.ParentProcessId = HandleToULong(CreateInfo->ParentProcessId);
item.CreatingProcessId = HandleToULong(
    CreateInfo->CreatingThreadId.UniqueProcess);
item.CreatingThreadId = HandleToULong(
    CreateInfo->CreatingThreadId.UniqueThread);

if (commandLineSize > 0) {
    memcpy(item.CommandLine, CreateInfo->CommandLine->Buffer,
        commandLineSize);
    item.CommandLength = commandLineSize / sizeof(WCHAR);
}
else {
    item.CommandLength = 0;
}
g_State.AddItem(&info->Entry);
}

else {
    auto info = (FullItem<ProcessExitInfo>*)ExAllocatePoolWithTag(
        PagedPool, sizeof(FullItem<ProcessExitInfo>), DRIVER_TAG);
    if (info == nullptr) {
        KdPrint((DRIVER_PREFIX "failed allocation\n"));
        return;
    }

    auto& item = info->Data;
    KeQuerySystemTimePrecise(&item.Time);
    item.Type = ItemType::ProcessExit;
    item.ProcessId = HandleToULong(ProcessId);
    item.Size = sizeof(ProcessExitInfo);
    item.ExitCode = PsGetProcessExitStatus(Process);

    g_State.AddItem(&info->Entry);
}
}
```

Providing Data to User Mode

The next thing to consider is how to provide the gathered information to a user-mode client. There are several options that could be used, but for this driver we'll let the client poll the driver for information using a read request. The driver will fill the user-provided buffer with as many events as possible, until either the buffer is exhausted or there are no more events in the queue.

We'll start the read request by obtaining the address of the user's buffer with Direct I/O (set up in `DriverEntry`):

```
NTSTATUS SysMonRead(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto len = irpSp->Parameters.Read.Length;
    auto status = STATUS_SUCCESS;
    ULONG bytes = 0;
    NT_ASSERT(Irp->MdlAddress); // we're using Direct I/O

    auto buffer = (PUCHAR)MmGetSystemAddressForMdlSafe(
        Irp->MdlAddress, NormalPagePriority);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
    }
}
```

Now we need to access our linked list and pull items from its head. We'll add this support to the `Global` class by implementing a method that removed an item from the head and returns it. If the list is empty, it returns `NULL`:

```
LIST_ENTRY* Globals::RemoveItem() {
    Locker locker(m_Lock);
    auto item = RemoveHeadList(&m_ItemsHead);
    if (item == &m_ItemsHead)
        return nullptr;

    m_Count--;
    return item;
}
```

If the linked list is empty, `RemoveHeadList` returns the head itself. It's also possible to use `IsEmpty` to make that determination. Lastly, we can check if `m_Count` is zero - all these are equivalent. If there is an item, it's returned as a `LIST_ENTRY` pointer.

Back to the Read dispatch routine - we can now loop around, getting an item out, copying its data to the user-mode buffer, until the list is empty or the buffer is full:

```

else {
    while (true) {
        auto entry = g_State.RemoveItem();
        if (entry == nullptr)
            break;

        //
        // get pointer to the actual data item
        //
        auto info = CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry);
        auto size = info->Data.Size;
        if (len < size) {
            //
            // user's buffer too small, insert item back
            //
            g_State.AddHeadItem(entry);
            break;
        }
        memcpy(buffer, &info->Data, size);
        len -= size;
        buffer += size;
        bytes += size;
        ExFreePool(info);
    }
}

return CompleteRequest(Irp, status, bytes);

```

`Globals::RemoveItem` is called to retrieve the head item (if any). Then we have to check if the remaining bytes in the user's buffer are enough to contain the data of this item. If not, we have to push the item back to the head of the queue, accomplished with another method in the `Globals` class:

```

void Globals::AddHeadItem(LIST_ENTRY* entry) {
    Locker locker(m_Lock);
    InsertHeadList(&m_ItemsHead, entry);
    m_Count++;
}

```

If there is enough room in the buffer, a simple `memcpy` is used to copy the actual data (everything except the `LIST_ENTRY` to the user's buffer). Finally, the variables are adjusted based on the size of this item and the loop repeats.

Once out of the loop, the only thing remaining is to complete the request with whatever status and information (`bytes`) have been accumulated thus far.

We need to take a look at the unload routine as well. If there are items in the linked list, they must be freed explicitly; otherwise, we have a leak on our hands:

```

void SysMonUnload(PDRIVER_OBJECT DriverObject) {
    PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);

    LIST_ENTRY* entry;
    while ((entry = g_State.RemoveItem()) != nullptr)
        ExFreePool(CONTAINING_RECORD(entry, FullItem<ItemHeader>, Entry));

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\sysmon");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}

```

The linked list items are freed by repeatedly removing items from the list and calling `ExFreePool` on each item.

The User Mode Client

Once we have all this in place, we can write a user mode client that polls data using `ReadFile` and displays the results.

The `main` function calls `ReadFile` in a loop, sleeping a bit so that the thread is not always consuming CPU. Once some data arrives, it's sent for display purposes:

```

#include <Windows.h>
#include <stdio.h>
#include <memory>
#include <string>
#include "..\SysMon\SysMonPublic.h"

int main() {
    auto hFile = CreateFile(L"\?\?\SysMon", GENERIC_READ, 0,
                           nullptr, OPEN_EXISTING, 0, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        return Error("Failed to open file");

    int size = 1 << 16;           // 64 KB
    auto buffer = std::make_unique<BYTE[]>(size);

    while (true) {
        DWORD bytes = 0;
        // error handling omitted
        ReadFile(hFile, buffer.get(), size, &bytes, nullptr);

        if (bytes)

```

```
    DisplayInfo(buffer.get(), bytes);

    // wait a bit before polling again
    Sleep(400);
}

// never actually reached
CloseHandle(hFile);
return 0;
}
```

The `DisplayInfo` function must make sense of the buffer it's given. Since all events start with a common header, the function distinguishes the various events based on the `ItemType`. After the event has been dealt with, the `Size` field in the header indicates where the next event starts:

```
void DisplayInfo(BYTE* buffer, DWORD size) {
    while (size > 0) {
        auto header = (ItemHeader*)buffer;
        switch (header->Type) {
            case ItemType::ProcessExit:
            {
                DisplayTime(header->Time);
                auto info = (ProcessExitInfo*)buffer;
                printf("Process %u Exited (Code: %u)\n",
                    info->ProcessId, info->ExitCode);
                break;
            }

            case ItemType::ProcessCreate:
            {
                DisplayTime(header->Time);
                auto info = (ProcessCreateInfo*)buffer;
                std::wstring commandline(info->CommandLine,
                    info->CommandLineLength);
                printf("Process %u Created. Command line: %ws\n",
                    info->ProcessId, commandline.c_str());
                break;
            }

        }
        buffer += header->Size;
        size -= header->Size;
    }
}
```

```
}
```

To extract the command line properly, the code uses the C++ `wstring` class constructor that can build a string based on a pointer and the string length. The `DisplayTime` helper function formats the time in a human-readable way:

```
void DisplayTime(const LARGE_INTEGER& time) {
    //
    // LARGE_INTEGER and FILETIME have the same size
    // representing the same format in our case
    //
    FILETIME local;

    //
    // convert to local time first (KeQuerySystemTime(Procise) returns UTC)
    //
    FileTimeToLocalFileTime((FILETIME*)&time, &local);
    SYSTEMTIME st;
    FileTimeToSystemTime(&local, &st);
    printf("%02d:%02d:%02d.%03d: ",
           st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);
}
```

`SYSTEMTIME` is a convenient structure to work with, as it contains all ingredients of a date and time. In the above code, only the time is displayed, but the date components are present as well.

That's all we need to begin testing the driver and the client.

The driver can be installed and started as done in earlier chapters, similar to the following:

```
sc create sysmon type= kernel binPath= C:\Test\SysMon.sys

sc start sysmon
```

Here is some sample output when running `SysMonClient.exe`:

```
16:18:51.961: Process 13124 Created. Command line: "C:\Program Files (x86)\Microsoft\Edge\Application\97.0.1072.62\identity_helper.exe" --type=utility --utility-sub-type=winrt_app_id.mojom.WinrtAppIdService --field-trial-handle=2060,1091\8786588500781911,4196358801973005731,131072 --lang=en-US --service-sandbox-type\=none --mojo-platform-channel-handle=5404 /prefetch:8
16:18:51.967: Process 13124 Exited (Code: 3221226029)
16:18:51.969: Process 6216 Created. Command line: "C:\Program Files (x86)\Microsoft\Edge\Application\97.0.1072.62\identity_helper.exe" --type=utility --utility\
```

```
y-sub-type=winrt_app_id.mojom.WinrtAppIdService --field-trial-handle=2060,10918\
786588500781911,4196358801973005731,131072 --lang=en-US --service-sandbox-type=\
none --mojo-platform-channel-handle=5404 /prefetch:8
16:18:53.836: Thread 12456 Created in process 10720
16:18:58.159: Process 10404 Exited (Code: 1)
16:19:02.033: Process 6216 Exited (Code: 0)
16:19:28.163: Process 9360 Exited (Code: 0)
```

Thread Notifications

The kernel provides thread creation and destruction callbacks, similarly to process callbacks. The API to use for registration is `PsSetCreateThreadNotifyRoutine` and for unregistering there is another API, `PsRemoveCreateThreadNotifyRoutine`:

```
NTSTATUS PsSetCreateThreadNotifyRoutine(
    _In_ PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine);
NTSTATUS PsRemoveCreateThreadNotifyRoutine (
    _In_ PCREATE_THREAD_NOTIFY_ROUTINE NotifyRoutine);
```

The arguments provided to the callback routine are the process ID, thread ID and whether the thread is being created or destroyed:

```
typedef void (*PCREATE_THREAD_NOTIFY_ROUTINE)(
    _In_ HANDLE ProcessId,
    _In_ HANDLE ThreadId,
    _In_ BOOLEAN Create);
```

If a thread is created, the callback is executed by the creator thread; if the thread exits, the callback executes on that thread.

We'll extend the existing *SysMon* driver to receive thread notifications as well as process notifications. First, we'll add enum values for thread events and a structure representing the information, all in the *SysMonCommon.h* header file:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit
};

struct ThreadCreateInfo : ItemHeader {
```

```

    ULONG ThreadId;
    ULONG ProcessId;
};

struct ThreadExitInfo : ThreadCreateInfo {
    ULONG ExitCode;
};

```

It's convenient to have `ThreadExitInfo` inherit from `ThreadCreateInfo`, as they share the thread and process IDs. It's certainly not mandatory, but it makes the thread notification callback a bit simpler to write.

Now we can add the proper registration to `DriverEntry`, right after registering for process notifications:

```

status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set thread callbacks (0x%08X)\n",
             status));
    break;
}

```

Conversley, a call to `PsRemoveCreateThreadNotifyRoutine` is needed in the Unload routine:

```
// in SysMonUnload
PsRemoveCreateThreadNotifyRoutine(OnThreadNotify);
```

The callback routine itself is simpler than the process notification callback, since the event structures have fixed sizes. Here is the thread callback routine in its entirety:

```

void OnThreadNotify(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create) {
    //
    // handle create and exit with the same code block, tweaking as needed
    //
    auto size = Create ? sizeof(FullItem<ThreadCreateInfo>)
        : sizeof(FullItem<ThreadExitInfo>);
    auto info = (FullItem<ThreadExitInfo*>)ExAllocatePoolWithTag(
        PagedPool, size, DRIVER_TAG);
    if (info == nullptr) {
        KdPrint((DRIVER_PREFIX "Failed to allocate memory\n"));
        return;
    }
    auto& item = info->Data;

```

```

KeQuerySystemTimePrecise(&item.Time);
item.Size = Create ? sizeof(ThreadCreateInfo) : sizeof(ThreadExitInfo);
item.Type = Create ? ItemType::ThreadCreate : ItemType::ThreadExit;
item.ProcessId = HandleToULong(ProcessId);
item.ThreadId = HandleToULong(ThreadId);
if (!Create) {
    PETHREAD thread;
    if (NT_SUCCESS(PsLookupThreadByThreadId(ThreadId, &thread))) {
        item.ExitCode = PsGetThreadExitStatus(thread);
        ObDereferenceObject(thread);
    }
}
g_State.AddItem(&info->Entry);
}

```

Most of this code should look pretty familiar. The slightly complex part is retrieving the thread exit code.

`PsGetThreadExitStatus` can be used for that, but that API requires a thread object pointer rather than an ID. `PsLookupThreadByThreadId` is used to obtain the thread object that is passed to `PsGetThreadExitStatus`. It's important to remember to call `ObDereferenceObject` on the thread object or else it will linger in memory until the next system restart.

To complete the implementation, we'll add code to the client that knows how to display thread creation and destruction (in the switch block inside `DisplayInfo`):

```

case ItemType::ThreadCreate:
{
    DisplayTime(header->Time);
    auto info = (ThreadCreateInfo*)buffer;
    printf("Thread %u Created in process %u\n",
           info->ThreadId, info->ProcessId);
    break;
}

case ItemType::ThreadExit:
{
    DisplayTime(header->Time);
    auto info = (ThreadExitInfo*)buffer;
    printf("Thread %u Exited from process %u (Code: %u)\n",
           info->ThreadId, info->ProcessId, info->ExitCode);
    break;
}

```

Here is some sample output given the updated driver and client:

```
16:19:41.500: Thread 10512 Created in process 9304
16:19:41.500: Thread 10512 Exited from process 9304 (Code: 0)
16:19:41.500: Thread 4424 Exited from process 9304 (Code: 0)
16:19:41.501: Thread 10180 Exited from process 9304 (Code: 0)
16:19:41.777: Process 14324 Created. Command line: "C:\WINDOWS\system32\defrag.\exe" -p bf8 -s 00000000000003BC -b -OnlyPreferred C:
16:19:41.777: Thread 8120 Created in process 14324
16:19:41.780: Process 11572 Created. Command line: \??\C:\WINDOWS\system32\conh\ost.exe 0xffffffff -ForceV1
16:19:41.780: Thread 7952 Created in process 11572
16:19:41.784: Thread 8748 Created in process 11572
16:19:41.784: Thread 6408 Created in process 11572
```



Add client code that displays the process image name for thread create and exit.

Windows 10 adds another registration function that provides additional flexibility.

```
typedef enum _PSCREATETHREADNOTIFYTYPE {
    PsCreateThreadNotifyNonSystem = 0,
    PsCreateThreadNotifySubsystems = 1
} PSCREATETHREADNOTIFYTYPE;

NTSTATUS PsSetCreateThreadNotifyRoutineEx(
    _In_ PSCREATETHREADNOTIFYTYPE NotifyType,
    _In_ PVOID NotifyInformation); // PCREATE_THREAD_NOTIFY_ROUTINE
```

Using `PsCreateThreadNotifyNonSystem` indicates the callback for new threads should execute on the newly created thread, rather than the creator.

Image Load Notifications

The last callback mechanism we'll look at in this chapter is image load notifications. Whenever a PE image (EXE, DLL, driver) file loads, the driver can receive a notification.

The `PsSetLoadImageNotifyRoutine` API registers for these notifications, and `PsRemoveImageNotifyRoutine` is used for unregistering:

```
NTSTATUS PsSetLoadImageNotifyRoutine(
    _In_ PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine);
NTSTATUS PsRemoveLoadImageNotifyRoutine(
    _In_ PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine);
```

The callback function has the following prototype:

```
typedef void (*PLOAD_IMAGE_NOTIFY_ROUTINE)(
    _In_opt_ PUNICODE_STRING FullImageName,
    _In_ HANDLE ProcessId,      // pid into which image is being mapped
    _In_ PIMAGE_INFO ImageInfo);
```

Curiously enough, there is no callback mechanism for image unloads.

The *FullImageName* argument is somewhat tricky. As indicated by the SAL annotation, it's optional and can be NULL. Even if it's not NULL, it doesn't always produce the correct image file name before Windows 10. The reasons for that are rooted deep in the kernel, it's I/O system and the file system cache. In most cases, this works fine, and the format of the path is the internal NT format, starting with something like "\Device\HadrdiskVolumex\..." rather than "c:\...". Translation can be done in a few ways, we'll see one way when we look at the client code.

The *ProcessId* argument is the process ID into which the image is loaded. For drivers (kernel modules), this value is zero.

The *ImageInfo* argument contains additional information on the image, declared as follows:

```
#define IMAGE_ADDRESSING_MODE_32BIT      3

typedef struct _IMAGE_INFO {
    union {
        ULONG Properties;
        struct {
            ULONG ImageAddressingMode : 8; // Code addressing mode
            ULONG SystemModeImage : 1; // System mode image
            ULONG ImageMappedToAllPids : 1; // Image mapped into all processes
            ULONG ExtendedInfoPresent : 1; // IMAGE_INFO_EX available
            ULONG MachineTypeMismatch : 1; // Architecture type mismatch
            ULONG resourcesignatureLevel : 4; // Signature level
            ULONG resourcesignatureType : 3; // Signature type
            ULONG ImagePartialMap : 1; // Nonzero if entire image is not \
mapped
        }
    }
}
```

```

    ULONG Reserved : 12;
};

};

PVOID ImageBase;
ULONG resourceselector;
SIZE_T resourcesize;
ULONG resourcesectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;

```

Here is quick rundown of the important fields in this structure:

- *SystemModeImage* - this flag is set for a kernel image, and unset for a user mode image.
- *resourcesignatureLevel* - signing level for *Protected Processes Light* (PPL) (Windows 8.1 and later). See `SE_SIGNING_LEVEL_` constants in the WDK.
- *resourcesignatureType* - signature type for PPL (Windows 8.1 and later). See the `SE_IMAGE_SIGNATURE_TYPE` enumeration in the WDK.
- *ImageBase* - the virtual address into which the image is loaded.
- *ImageSize* - the size of the image.
- *ExtendedInfoPresent* - if this flag is set, then `IMAGE_INFO` is part of a larger structure, `IMAGE_INFO_EX`, shown here:

```

typedef struct _IMAGE_INFO_EX {
    SIZE_T Size;
    IMAGE_INFO ImageInfo;
    struct _FILE_OBJECT *FileObject;
} IMAGE_INFO_EX, *PIMAGE_INFO_EX;

```

To access this larger structure, a driver can use the `CONTAINING_RECORD` macro like so:

```

if (ImageInfo->ExtendedInfoPresent) {
    auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
    // access FileObject
}

```

The extended structure adds just one meaningful member - the file object used to open the image. This may be useful for retrieving the file name in pre-Windows 10 machines, as we'll soon see.

As with the process and thread notifications, we'll add the needed code to register in `DriverEntry` and the code to unregister in the `Unload` routine. Here is the full `DriverEntry` function (with `KdPrint` calls removed for brevity):

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    auto status = STATUS_SUCCESS;

    PDEVICE_OBJECT DeviceObject = nullptr;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\sysmon");
    bool symLinkCreated = false;
    bool processCallbacks = false, threadCallbacks = false;

    do {
        UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\sysmon");
        status = IoCreateDevice(DriverObject, 0, &devName,
                               FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
        if (!NT_SUCCESS(status)) {
            break;
        }
        DeviceObject->Flags |= DO_DIRECT_IO;

        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status)) {
            break;
        }
        symLinkCreated = true;

        status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
        if (!NT_SUCCESS(status)) {
            break;
        }
        processCallbacks = true;

        status = PsSetCreateThreadNotifyRoutine(OnThreadNotify);
        if (!NT_SUCCESS(status)) {
            break;
        }
        threadCallbacks = true;

        status = PsSetLoadImageNotifyRoutine(OnImageLoadNotify);
        if (!NT_SUCCESS(status)) {
            break;
        }
    } while (false);

    if (!NT_SUCCESS(status)) {
```

```

    if (threadCallbacks)
        PsRemoveCreateThreadNotifyRoutine(OnThreadNotify);
    if (processCallbacks)
        PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);
    if (symLinkCreated)
        IoDeleteSymbolicLink(&symLink);
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    return status;
}

g_State.Init(10000);

DriverObject->DriverUnload = SysMonUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = SysMonCreateClose;
DriverObject->MajorFunction[IRP_MJ_READ] = SysMonRead;

return status;
}

```

We'll add an event type to the `ItemType` enum:

```

enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit,
    ImageLoad
};

```

As before, we need a structure to contain the information we can get from image load:

```

const int MaxImageFileSize = 300;

struct ImageLoadInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ImageSize;
    ULONG64 LoadAddress;
    WCHAR ImageFileName[MaxImageFileSize + 1];
};

```

For variety, `ImageLoadInfo` uses a fixed size array to store the path to the image file. The interested reader should change that to use a scheme similar to process create notifications.

The image load notification starts by not storing information on kernel images:

```
void OnImageLoadNotify(PUNICODE_STRING FullImageName,
    HANDLE ProcessId, PIMAGE_INFO ImageInfo) {
    if (ProcessId == nullptr) {
        // system image, ignore
        return;
    }
```

This is not necessary, of course. You can remove the above check so that kernel images are reported as well. Next, we allocate the data structure and fill in the usual information:

```
auto size = sizeof(FullItem<ImageLoadInfo>);
auto info = (FullItem<ImageLoadInfo>*)ExAllocatePoolWithTag(PagedPool, size, DR\
IVER_TAG);
if (info == nullptr) {
    KdPrint((DRIVER_PREFIX "Failed to allocate memory\n"));
    return;
}

auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Size = sizeof(item);
item.Type = ItemType::ImageLoad;
item.ProcessId = HandleToULong(ProcessId);
item.ImageSize = (ULONG)ImageInfo->ImageSize;
item.LoadAddress = (ULONG64)ImageInfo->ImageBase;
```

The interesting part is the image path. The simplest option is to examine `FullImageName`, and if non-NULL, just grab its contents. But since this information might be missing or not 100% reliable, we can try something else first, and fall back on `FullImageName` if all else fails.

The secret is to use `FltGetFileNameInformationUnsafe` - a variant on `FltGetFileNameInformation` that is used with File System Mini-filters, as we'll see in chapter 12. The "Unsafe" version can be called in non-file-system contexts as is our case. A full discussion on `FltGetFileNameInformation` is saved for chapter 12. For now, let's just use if the file object is available:

```

item.ImageFileName[0] = 0; // assume no file information
if (ImageInfo->ExtendedInfoPresent) {
    auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
    PFLT_FILE_NAME_INFORMATION nameInfo;
    if (NT_SUCCESS(FltGetFileNameInformationUnsafe(exinfo->FileObject,
        nullptr, FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
        &nameInfo))) {
        // copy the file path
        wcscpy_s(item.ImageFileName, nameInfo->Name.Buffer);
        FltReleaseFileNameInformation(nameInfo);
    }
}

```

`FltGetFileNameInformationUnsafe` requires the file object that can be obtained from the extended `IMAGE_INFO_EX` structure. `wcscpy_s` ensures we don't copy more characters than are available in the buffer. `FltReleaseFileNameInformation` must be called to free the `PFLT_FILE_NAME_INFORMATION` object allocated by `FltGetFileNameInformationUnsafe`.

To gain access to these functions, add `#include <FltKernel.h>` and add `FlgMgr.lib` into the Linker Input / Additional Dependencies line.

Finally, if this method does not produce a result, we fall back to using the provided image path:

```

if (item.ImageFileName[0] == 0 && FullImageName) {
    wcscpy_s(item.ImageFileName, FullImageName->Buffer);
}

g_State.AddItem(&info->Entry);

```

Here is the full image load notification code for easier reference (`KdPrint` removed):

```

void OnImageLoadNotify(PUNICODE_STRING FullImageName, HANDLE ProcessId, PIMAGE_\
INFO ImageInfo) {
    if (ProcessId == nullptr) {
        // system image, ignore
        return;
    }

    auto size = sizeof(FullItem<ImageLoadInfo>);
    auto info = (FullItem<ImageLoadInfo>*)ExAllocatePoolWithTag(

```

```
    PagedPool, size, DRIVER_TAG);
if (info == nullptr)
    return;

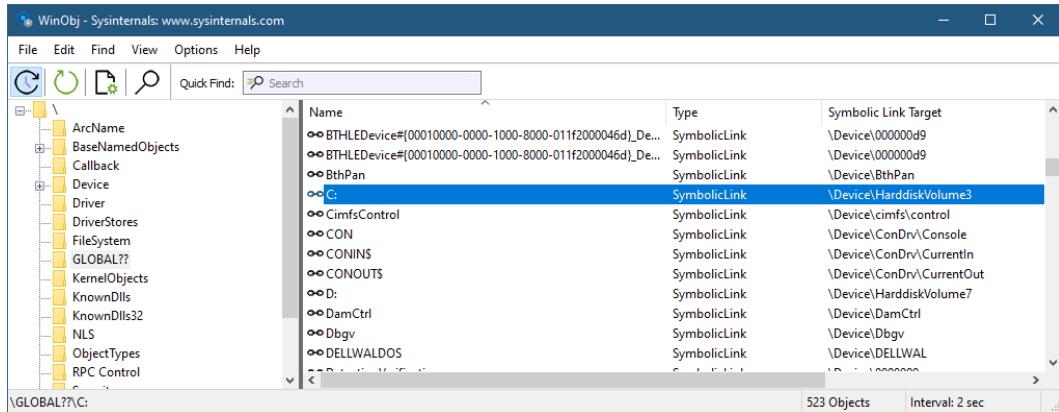
auto& item = info->Data;
KeQuerySystemTimePrecise(&item.Time);
item.Size = sizeof(item);
item.Type = ItemType::ImageLoad;
item.ProcessId = HandleToULong(ProcessId);
item.ImageSize = (ULONG)ImageInfo->ImageSize;
item.LoadAddress = (ULONG64)ImageInfo->ImageBase;
item.ImageFileName[0] = 0;

if (ImageInfo->ExtendedInfoPresent) {
    auto exinfo = CONTAINING_RECORD(ImageInfo, IMAGE_INFO_EX, ImageInfo);
    PFLT_FILE_NAME_INFORMATION nameInfo;
    if (NT_SUCCESS(FltGetFileNameInformationUnsafe(
        exinfo->FileObject, nullptr,
        FLT_FILE_NAME_NORMALIZED | FLT_FILE_NAME_QUERY_DEFAULT,
        &nameInfo))) {
        wcscpy_s(item.ImageFileName, nameInfo->Name.Buffer);
        FltReleaseFileNameInformation(nameInfo);
    }
}
if (item.ImageFileName[0] == 0 && FullImageName) {
    wcscpy_s(item.ImageFileName, FullImageName->Buffer);
}

g_State.AddItem(&info->Entry);
}
```

Final Client Code

The client code must be extended for image loads. It seems easy enough except for one snag: the resulting image path retrieved in the image load notification is in NT Device form, instead of the more common, “DOS based” form with drive letters, which in fact are symbolic links. We can see these mappings in tools such as *WinObj* from *Sysinternals* (figure 9-3).

Figure 9-3: Symbolic links in *WinObj*

Notice the device name targets for C: and D: in figure 9-3. A file like `c:\temp\mydll.dll` will be reported as `\Device\DeviceHarddiskVolume3\temp\mydll.dll`. It would be nice if the display would show the common mappings instead of the NT device name.

One way of getting these mappings is by calling `QueryDosDevice`, which retrieves the target of a symbolic link stored in the “?” Object Manager directory. We are already familiar with these symbolic links, as they are valid strings to the `CreateFile` API.

Based on `QueryDosDevice`, we can loop over all existing drive letters and store the targets. Then, we can lookup every device name and find its drive letter (symbolic link). Here is a function to do that. If we can't find a match, we'll just return the original string:

```
#include <unordered_map>

std::wstring GetDosNameFromNTName(PCWSTR path) {
    if (path[0] != L'\\')
        return path;

    static std::unordered_map<std::wstring, std::wstring> map;
    if (map.empty()) {
        auto drives = GetLogicalDrives();
        int c = 0;
        WCHAR root[] = L"X:";
        WCHAR target[128];
        while (drives) {
            if (drives & 1) {
                root[0] = 'A' + c;
                if (QueryDosDevice(root, target, _countof(target))) {
                    map.insert({ target, root });
                }
            }
            c++;
            drives >>= 1;
        }
    }
    return map[path];
}
```

```

        drives >= 1;
        c++;
    }
}

auto pos = wcschr(path + 1, L'\\');
if (pos == nullptr)
    return path;

pos = wcschr(pos + 1, L'\\');
if (pos == nullptr)
    return path;

std::wstring ntname(path, pos - path);
if (auto it = map.find(ntname); it != map.end())
    return it->second + std::wstring(pos);

return path;
}

```

I will let the interested reader figure out how this code works. In any case, since user-mode is not the focus of this book, you can just use the function as is, as we'll do in our client.

Here is the part in `DisplayInfo` that handles image load notifications (within the `switch`):

```

case ItemType::ImageLoad:
{
    DisplayTime(header->Time);
    auto info = (ImageLoadInfo*)buffer;
    printf("Image loaded into process %u at address 0x%llx (%ws)\n",
           info->ProcessId, info->LoadAddress,
           GetDosNameFromNTName(info->ImageFileName).c_str());
    break;
}

```

Here is some example output when running the full driver and client:

```
18:59:37.660: Image loaded into process 12672 at address 0x7FFD531C0000 (C:\Win\dows\System32\msvcp110_win.dll)
18:59:37.661: Image loaded into process 12672 at address 0x7FFD5BF30000 (C:\Win\dows\System32\advapi32.dll)
18:59:37.676: Thread 11416 Created in process 5820
18:59:37.676: Thread 12496 Created in process 4824
18:59:37.731: Thread 6636 Created in process 3852
18:59:37.731: Image loaded into process 12672 at address 0x7FFD59F70000 (C:\Win\dows\System32\ntmarta.dll)
18:59:37.735: Image loaded into process 12672 at address 0x7FFD51340000 (C:\Win\dows\System32\policymanager.dll)
18:59:37.735: Image loaded into process 12672 at address 0x7FFD531C0000 (C:\Win\dows\System32\msvcp110_win.dll)
18:59:37.737: Image loaded into process 12672 at address 0x7FFD51340000 (C:\Win\dows\System32\policymanager.dll)
18:59:37.737: Image loaded into process 12672 at address 0x7FFD531C0000 (C:\Win\dows\System32\msvcp110_win.dll)
18:59:37.756: Thread 6344 Created in process 704
```



Add the process name in image load notifications.

Create a driver that monitors process creation and allows a client application to configure executable paths that should not be allowed to execute.

Remote Thread Detection

One interesting example of using process and thread notifications is to detect *remote threads*. A remote thread is one that is created (injected) to a process different than its creator. This fairly well-known technique can be used (for example) to force the new thread to load a DLL, essentially injecting that DLL into another process.

This scenario is not necessarily malicious, but it could be. The most common example where this happens is when a debugger attaches to a target and wants to break into the target. This is done by creating a thread in the target process (by the debugger process) and pointing the thread function to an API such as `DebugBreak` that forces a breakpoint, allowing the debugger to gain control.

Anti-malware systems know how to detect these scenarios, as these may be malicious. Let's build a driver that can make that kind of detection. At first, it seems to be very simple: when a thread is created, compare its creator's process ID with the target process where the thread is created, and if they are different - you have a remote thread on your hands.

There is a small dent in the above description. The first thread in any process is "remote" by definition, because it's created by some other process (typically the one calling `CreateProcess`), so this "natural" occurrence should not be considered a remote thread creation.



If you feel up to it, code this driver on your own!

The core of the driver are process and thread notification callbacks. The most important is the thread creation callback, where the driver's job is to determine whether a created thread is a remote one or not. We must keep an eye for new processes as well, because the first thread in a new process is technically remote, but we need to ignore it.

The data maintained by the driver and later provided to the client contains the following (*Detector-Public.h*):

```
struct RemoteThread {
    LARGE_INTEGER Time;
    ULONG CreatorProcessId;
    ULONG CreatorThreadId;
    ULONG ProcessId;
    ULONG ThreadId;
};
```

Here is the data we'll store as part of the driver (in *KDetector.h*):

```
struct RemoteThreadItem {
    LIST_ENTRY Link;
    RemoteThread Remote;
};

const ULONG MaxProcesses = 32;

ULONG NewProcesses[MaxProcesses];
ULONG NewProcessesCount;
ExecutiveResource ProcessesLock;
LIST_ENTRY RemoteThreadsHead;
FastMutex RemoteThreadsLock;
LookasideList<RemoteThreadItem> Lookaside;
```

There are a few class wrappers for kernel APIs we haven't seen yet. `FastMutex` is the same we used in the *SysMon* driver. `ExecutiveResource` is a wrapper for an ERESOURCE structure and APIs we looked at in chapter 6. Here is its declaration and definition:

```
// ExecutiveResource.h
```

```
struct ExecutiveResource {
    void Init();
    void Delete();

    void Lock();
    void Unlock();

    void LockShared();
    void UnlockShared();

private:
    ERESOURCE m_res;
    bool m_CritRegion;
};
```

```
// ExecutiveResource.cpp
```

```
void ExecutiveResource::Init() {
    ExInitializeResourceLite(&m_res);
}

void ExecutiveResource::Delete() {
    ExDeleteResourceLite(&m_res);
}

void ExecutiveResource::Lock() {
    m_CritRegion = KeAreApcsDisabled();
    if(m_CritRegion)
        ExAcquireResourceExclusiveLite(&m_res, TRUE);
    else
        ExEnterCriticalSectionAndAcquireResourceExclusive(&m_res);
}

void ExecutiveResource::Unlock() {
    if (m_CritRegion)
        ExReleaseResourceLite(&m_res);
    else
        ExReleaseResourceAndLeaveCriticalSection(&m_res);
}

void ExecutiveResource::LockShared() {
```

```

m_CritRegion = KeAreApcsDisabled();
if (m_CritRegion)
    ExAcquireResourceSharedLite(&m_res, TRUE);
else
    ExEnterCriticalSectionAndAcquireResourceShared(&m_res);
}

void ExecutiveResource::UnlockShared() {
    Unlock();
}

```

A few things are worth noting:

- Acquiring an Executive Resource must be done in a critical region (when normal kernel APCs are disabled). The call to `KeAreApcsDisabled` returns true if normal kernel APCs are disabled. In that case a simple acquisition will do; otherwise, a critical region must be entered first, so the “shortcuts” to enter a critical region and acquire the Executive Resource are used.



A similar API, `KeAreAllApcsDisabled` returns true if all APCs are disabled (essentially whether the thread is in a guarded region).

- An Executive Resource is used to protect the `NewProcesses` array from concurrent write access. The idea is that more reads than writes are expected for this data. In any case, I wanted to show a possible wrapper for an Executive Resource.
- The class presents an interface that can work with the `Locker<TLock>` type we have been using for exclusive access. For shared access, the `LockShared` and `UnlockShared` methods are provided. To use them conveniently, a companion class to `Locker<>` can be written to acquire the lock in a shared manner. Here is its definition (in `Locker.h` as well):

```

template<typename TLock>
struct SharedLocker {
    SharedLocker(TLock& lock) : m_lock(lock) {
        lock.LockShared();
    }
    ~SharedLocker() {
        m_lock.UnlockShared();
    }

private:
    TLock& m_lock;
};

```

`LookasideList<T>` is a wrapper for lookaside lists we met in chapter 8. It’s using the new API, as it’s easier for selecting the pool type required. Here is its definition (in `LookasideList.h`):

```

template<typename T>
struct LookasideList {
    NTSTATUS Init(POOL_TYPE pool, ULONG tag) {
        return ExInitializeLookasideListEx(&m_lookaside, nullptr, nullptr,
            pool, 0, sizeof(T), tag, 0);
    }

    void Delete() {
        ExDeleteLookasideListEx(&m_lookaside);
    }

    T* Alloc() {
        return (T*)ExAllocateFromLookasideListEx(&m_lookaside);
    }

    void Free(T* p) {
        ExFreeToLookasideListEx(&m_lookaside, p);
    }

private:
    LOOKASIDE_LIST_EX m_lookaside;
};


```

Going back to the data members for this driver. The purpose of the `NewProcesses` array is to keep track of new processes before their first thread is created. Once the first thread is created, and identified as such, the array will drop the process in question, because from that point on, any new thread created in that process from another process is a remote thread for sure. We'll see all that in the callbacks implementations.

The driver uses a simple array rather than a linked list, because I don't expect a lot of processes with no threads to exist for more than a tiny fraction, so a fixed sized array should be good enough. However, you can change that to a linked list to make this bulletproof.

When a new process is created, it should be added to the `NewProcesses` array since the process has zero threads at that moment:

```

void OnProcessNotify(PEPROCESS Process, HANDLE ProcessId,
    PPS_CREATE_NOTIFY_INFO CreateInfo) {
    UNREFERENCED_PARAMETER(Process);

    if (CreateInfo) {
        if (!AddNewProcess(ProcessId)) {
            KdPrint((DRIVER_PREFIX "New process created, no room to store\n"));
        }
        else {

```

```
        KdPrint((DRIVER_PREFIX "New process added: %u\n", HandleToULong(ProcAddress)));
    }
}
}
```

AddProcess locates an empty “slot” in the array and puts the process ID in it:

```
bool AddNewProcess(HANDLE pid) {
    Locker locker(ProcessesLock);
    if (NewProcessesCount == MaxProcesses)
        return false;

    for(int i = 0; i < MaxProcesses; i++)
        if (NewProcesses[i] == 0) {
            NewProcesses[i] = HandleToUlong(pid);
            break;
        }
    NewProcessesCount++;
    return true;
}
```

Now comes the interesting part: the thread create/exit callback.



1. Add process names to the data maintained by the driver for each remote thread. A remote thread is when the creator (the caller) is different than the process in which the new thread is created. We also have to remove some false positives:

```
void OnThreadNotify(HANDLE ProcessId, HANDLE ThreadId, BOOLEAN Create) {
    if (Create) {
        bool remote = PsGetCurrentProcessId() != ProcessId
            && PsInitialSystemProcess != PsGetCurrentProcess()
            && PsGetProcessId(PsInitialSystemProcess) != ProcessId;
```

The second and third checks make sure the source process or target process is not the *System* process. The reasons for the *System* process to exist in these cases are interesting to investigate, but are out of scope for this book - we'll just remove these false positives. The question is how to identify the *System* process. All versions of Windows from XP have the same PID for the *System* process: 4. We

could use that number because it's unlikely to change in the future, but there is another way, which is foolproof and also allows me to introduce something new.

The kernel exports a global variable, `PsInitialSystemProcess`, which always points to the *System* process' EPROCESS structure. This pointer can be used just like any other opaque process pointer.

If the thread is indeed remote, we must check if it's the first thread in the process, and if so, discard this as a remote thread:

```
if (remote) {
    //
    // really remote if it's not a new process
    //
    bool found = FindProcess(ProcessId);
```

`FindProcess` searches for a process ID in the `NewProcesses` array:

```
bool FindProcess(HANDLE pid) {
    auto id = HandleToUlong(pid);
    SharedLocker locker(ProcessesLock);
    for (int i = 0; i < MaxProcesses; i++)
        if (NewProcesses[i] == id)
            return true;
    return false;
}
```

If the process is found, then it's the first thread in the process and we should remove the process from the new processes array so that subsequent remote threads (if any) can be identified as such:

```
if (found) {
    //
    // first thread in process, remove process from new processes array
    //
    RemoveProcess(ProcessId);
}
```

`RemoveProcess` searches for the PID and removes it from the array by zeroing it out:

```

bool RemoveProcess(HANDLE pid) {
    auto id = HandleToUlong(pid);
    Locker locker(ProcessesLock);
    for (int i = 0; i < MaxProcesses; i++) {
        if (NewProcesses[i] == id) {
            NewProcesses[i] = 0;
            NewProcessesCount--;
            return true;
        }
    }
    return false;
}

```

If the process isn't found, then it's not new and we have a real remote thread on our hands:

```

else {
    //
    // really a remote thread
    //
    auto item = Lookaside.Alloc();
    auto& data = item->Remote;
    KeQuerySystemTimePrecise(&data.Time);
    data.CreatorProcessId = HandleToULong(PsGetCurrentProcessId());
    data.CreatorThreadId = HandleToULong(PsGetCurrentThreadId());
    data.ProcessId = HandleToULong(ProcessId);
    data.ThreadId = HandleToULong(ThreadId);

    KdPrint((DRIVER_PREFIX
        "Remote thread detected. (PID: %u, TID: %u) -> (PID: %u, TID: %u)\n",
        data.CreatorProcessId, data.CreatorThreadId,
        data.ProcessId, data.ThreadId));

    Locker locker(RemoteThreadsLock);
    // TODO: check the list is not too big
    InsertTailList(&RemoteThreadsHead, &item->Link);
}

```

Getting the data to a user mode client can be done in the same way as we did for the *SysMon* driver:

```
NTSTATUS DetectorRead(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto len = irpSp->Parameters.Read.Length;
    auto status = STATUS_SUCCESS;
    ULONG bytes = 0;
    NT_ASSERT(Irp->Md1Address);

    auto buffer = (PUCHAR)MmGetSystemAddressForMdlSafe(
        Irp->Md1Address, NormalPagePriority);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
    }
    else {
        Locker locker(RemoteThreadsLock);
        while (true) {
            //
            // if the list is empty, there is nothing else to give
            //
            if (IsListEmpty(&RemoteThreadsHead))
                break;

            //
            // if remaining buffer size is too small, break
            //
            if (len < sizeof(RemoteThread))
                break;

            auto entry = RemoveHeadList(&RemoteThreadsHead);
            auto info = CONTAINING_RECORD(entry, RemoteThreadItem, Link);
            ULONG size = sizeof(RemoteThread);
            memcpy(buffer, &info->Remote, size);
            len -= size;
            buffer += size;
            bytes += size;
            //
            // return data item to the lookaside list
            //
            Lookaside.Free(info);
        }
    }
    return CompleteRequest(Irp, status, bytes);
}
```

Because there is just one type of “event” and it has a fixed size, the code is simpler than in the *SysMon* case.

The full driver code is in the *KDetector* project in the solution for this chapter.

The Detector Client

The client code is very similar to the *SysMon* client, but simpler, because all “events” have the same structure and are even fixed-sized. Here are the `main` and `DisplayData` functions:

```
void DisplayData(const RemoteThread* data, int count) {
    for (int i = 0; i < count; i++) {
        auto& rt = data[i];
        DisplayTime(rt.Time);
        printf("Remote Thread from PID: %u TID: %u -> PID: %u TID: %u\n",
               rt.CreatorProcessId, rt.CreatorThreadId, rt.ProcessId, rt.ThreadId);
    }
}

int main() {
    HANDLE hDevice = CreateFile(L"\\\\.\\"kdetector", GENERIC_READ, 0,
                               nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE)
        return Error("Error opening device");

    RemoteThread rt[20];      // fixed array is good enough

    for (;;) {
        DWORD bytes;
        if (!ReadFile(hDevice, rt, sizeof(rt), &bytes, nullptr))
            return Error("Failed to read data");

        DisplayData(rt, bytes / sizeof(RemoteThread));
        Sleep(1000);
    }

    CloseHandle(hDevice);
    return 0;
}
```

The `DisplayTime` is the same one from the *SysMonClient* project.

We can test the driver by installing it and starting it normally, and launching our client (or we can use *DbgView* to see the remote thread outputs). The classic example of a remote thread (as mentioned

earlier) is when a debugger wishes to forcefully break into a target process. Here is one way to do that:

1. Run some executable, say Notepad.exe.
2. Launch WinDbg.
3. Use WinDbg to attach to the Notepad process. A remote thread notification should appear.

Here are some examples of output when the detector client is running:

```
13:08:15.280: Remote Thread from PID: 7392 TID: 4788 -> PID: 8336 TID: 9384  
13:08:58.660: Remote Thread from PID: 7392 TID: 13092 -> PID: 8336 TID: 13288  
13:10:52.313: Remote Thread from PID: 7392 TID: 13092 -> PID: 8336 TID: 12676  
13:11:25.207: Remote Thread from PID: 15268 TID: 7564 -> PID: 1844 TID: 6688  
13:11:25.209: Remote Thread from PID: 15268 TID: 15152 -> PID: 1844 TID: 7928
```

You might find some remote thread entries surprising (run *Process Explorer* for a while, for example)

The full code of the client is in the *Detector* project.



Display process names in the client.

Summary

In this chapter we looked at some of the callback mechanisms provided by the kernel: process, thread and image loads. In the next chapter, we'll continue with more callback mechanisms - opening handles to certain object types, and Registry notifications.

Chapter 10: Object and Registry Notifications

The kernel provides more ways to intercept certain operations. First, we'll examine object notifications, where obtaining handles to some types of objects can be intercepted. Then, we'll look at Registry operations interception.

In this chapter:

- Object Notifications
 - The Process Protector Driver
 - Registry Notifications
 - Extending the *SysMon* Driver
 - Exercises
-

Object Notifications

The kernel provides a mechanism to notify interested drivers when attempts to open or duplicate a handle to certain object types. The officially supported object types are process, thread, and for Windows 10 - desktop as well.

Desktop Objects

A desktop is a kernel object contained in a Window Station, yet another kernel object, which is in itself part of a Session. A desktop contains windows, menus, and hooks. The hooks referred to here are user-mode hooks available with the `SetWindowsHookEx` API.

Normally, when a user logs in, two desktops are created. A desktop named “Winlogon” is created by *Winlogon.exe*. This is the desktop that you see when pressing the *Secure Attention Sequence* key combination(SAS, normally *Ctrl+Alt+Del*). The second desktop is named “default” and is the normal desktop we are familiar with, where normal windows are shown and used. Switching to another desktop is done with the `SwitchDesktop` API. For some more details, read [this blog post^a](#).

^a<https://scorpionsoftware.net/2019/02/17/windows-10-desktops-vs-sysinternals-desktops/>

The registration API to call is `ObRegisterCallbacks`, prototyped like so:

```
NTSTATUS ObRegisterCallbacks (
    _In_ POB_CALLBACK_REGISTRATION CallbackRegistration,
    _Outptr_ PVOID *RegistrationHandle);
```

Prior to registration, an OB_CALLBACK_REGISTRATION structure must be initialized, which provides the necessary details about what the driver is registering for. The *RegistrationHandle* is the return value upon a successful registration, which is just an opaque pointer used for unregistering by calling ObUnRegisterCallbacks.

Drivers using `ObRegisterCallbacks` must be linked with the `/integritycheck` switch.

Here is the definition of OB_CALLBACK_REGISTRATION:

```
typedef struct _OB_CALLBACK_REGISTRATION {
    _In_ USHORT             Version;
    _In_ USHORT             OperationRegistrationCount;
    _In_ UNICODE_STRING     Altitude;
    _In_ PVOID               RegistrationContext;
    _In_ OB_OPERATION_REGISTRATION *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

Version is just a constant that must be set to OB_FLT_REGISTRATION_VERSION (currently 0x100). Next, the number of operations that are being registered is specified by *OperationRegistrationCount*. This determines the number of OB_OPERATION_REGISTRATION structures that are pointed to by *OperationRegistration*. Each one of these provides information on an object type of interest (process, thread or desktop).

The *Altitude* argument is interesting. It specifies a number (in string form) that affects the order of callbacks invocation for this driver. This is necessary because other drivers may have their own callbacks and the question of which driver is invoked first is answered by the altitude - the higher the altitude, the earlier in the call chain the driver is invoked.

What value should the altitude be? It shouldn't matter in most cases, as there is no obvious to know what values other drivers are using. The altitude provided must not collide with altitudes specified by previously registered drivers. The altitude does not have to be an integer number. In fact, it's an infinite precision decimal number, and this is why it's specified as a string. To avoid collision, the altitude should be set to something with random numbers after a decimal point, such as "12345.1762389". The chances of collision in this case are slim. The driver can even truly generate random digits to avoid collisions. If the registration fails with a status of STATUS_FLT_INSTANCE_ALTITUDE_COLLISION, this means altitude collision, so the careful driver can adjust its altitude and try again.

The concept of Altitude is also used for registry filtering (see “Registry Notifications” later in this chapter) and file system mini-filters (see chapter 12).

Finally, *RegistrationContext* is a driver defined value that is passed as-is to the callback routine(s).

The OB_OPERATION_REGISTRATION structure(s) is where the driver sets up its callbacks, indicates which object types and operations are of interest. It's defined like so:

```
typedef struct _OB_OPERATION_REGISTRATION {
    _In_ POBJECT_TYPE             *ObjectType;
    _In_ OB_OPERATION             Operations;
    _In_ POB_PRE_OPERATION_CALLBACK PreOperation;
    _In_ POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

ObjectType is a pointer to the object type for this instance registration - process, thread or desktop. These pointers are exported as global kernel variables: PsProcessType, PsThreadType, and ExDesktopObjectType, respectively.

The *Operations* field must specify one or two flags (OB_OPERATION), selecting create/open (OB_OPERATION_HANDLE_CREATE) and/or duplicate (OB_OPERATION_HANDLE_DUPLICATE).

OB_OPERATION_HANDLE_CREATE refers to calls to user mode functions such as CreateProcess, OpenProcess, CreateThread, OpenThread, CreateDesktop, OpenDesktop and similar functions for these object types. OB_OPERATION_HANDLE_DUPLICATE refers to handle duplication for these objects (such as using the DuplicateHandle user-mode API).

The APIs intercepted are not user-mode only; kernel APIs are intercepted as well (the callbacks parameters do indicate if the handle being created/duplicated is a kernel handle). Kernel APIs such as ZwOpenProcess, PsCreateSystemThread, and ZwDuplicateObject are examples of affected functions.

Any time one of these calls is made, one or two callbacks can be registered: a pre-operation callback (PreOperation field) and/or a post-operation callback (PostOperation).

Pre-Operation Callback

The pre-operation callback is invoked before the actual create/open/duplicate operation completes, giving a chance to the driver to make changes to the operation's result. The pre-operation callback receives a OB_PRE_OPERATION_INFORMATION structure, defined as shown here:

```

typedef struct _OB_PRE_OPERATION_INFORMATION {
    _In_ OB_OPERATION                      Operation;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG KernelHandle:1;
            _In_ ULONG Reserved:31;
        };
    };
    _In_ PVOID                               Object;
    _In_ POBJECT_TYPE                        ObjectType;
    _Out_ PVOID                               CallContext;
    _In_ POB_PRE_OPERATION_PARAMETERS    Parameters;
} OB_PRE_OPERATION_INFORMATION, *POB_PRE_OPERATION_INFORMATION;

```

Here is a rundown of the structure's members:

- *Operation* - indicates what operation is this (OB_OPERATION_HANDLE_CREATE or OB_OPERATION_HANDLE_DUPLICATE).
- *KernelHandle* (inside *Flags*) - indicates this is a kernel handle. Kernel handles can only be created and used by kernel code. This allows the driver to ignore kernel requests if it so desires.
- *Object* - the pointer to the actual object for which a handle is being created/opened/duplicated. For processes, this is the EPROCESS address, for thread it's the PTHREAD address.
- *ObjectType* - points to the object type: *PsProcessType, *PsThreadType or *ExDesktopObjectType.
- *CallContext* - a driver-defined value, that is propagated to the post-callback for this instance (if exists).
- *Parameters* - a union specifying additional information based on the *Operation*. This union is defined like so:

```

typedef union _OB_PRE_OPERATION_PARAMETERS {
    _Inout_ OB_PRE_CREATE_HANDLE_INFORMATION     CreateHandleInformation;
    _Inout_ OB_PRE_DUPLICATE_HANDLE_INFORMATION   DuplicateHandleInformation;
} OB_PRE_OPERATION_PARAMETERS, *POB_PRE_OPERATION_PARAMETERS;

```

The driver should inspect the appropriate field based on the operation. For Create operations, the driver receives the following information:

```

typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
    _Inout_ ACCESS_MASK  DesiredAccess;
    _In_ ACCESS_MASK     OriginalDesiredAccess;
} OB_PRE_CREATE_HANDLE_INFORMATION, *POB_PRE_CREATE_HANDLE_INFORMATION;

```

The *OriginalDesiredAccess* is the access mask specified by the caller. Consider this user-mode code to open a handle to an existing process:

```
HANDLE OpenHandleToProcess(DWORD pid) {
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
        FALSE, pid);
    if(!hProcess) {
        // failed to open a handle
    }
    return hProcess;
}
```

In this example, the client tries to obtain a handle to a process with the specified access mask, indicating what are its “intentions” towards that process. The driver’s pre-operation callback receives this value in the *OriginalDesiredAccess* field. This value is also copied to *DesiredAccess*. Normally, the kernel will determine, based on the client’s security context and the process’ security descriptor whether the client can be granted the access it desires.

The driver can, based on its own logic, modify *DesiredAccess* for example by removing some of the access requested by the client:

```
OB_PREOP_CALLBACK_STATUS OnPreOpenProcess(PVOID /* RegistrationContext */,
    POB_PRE_OPERATION_INFORMATION Info) {

    if(/* some logic */) {
        Info->Parameters->CreateHandleInformation.DesiredAccess &=
            ~PROCESS_VM_READ;
    }
    return OB_PREOP_SUCCESS;
}
```

The above code snippet removes the `PROCESS_VM_READ` access mask before letting the operation continue normally. If it eventually succeeds, the client will get back a valid handle, but only with the `PROCESS_QUERY_INFORMATION` access mask.



You can find the complete list of process, thread and desktop access masks in the MSDN documentation.



You cannot add new access mask bits that were not requested by the client.

For duplicate operations, the information provided to the driver is the following:

```
typedef struct _OB_PRE_DUPLICATE_HANDLE_INFORMATION {
    _Inout_ ACCESS_MASK DesiredAccess;
    _In_ ACCESS_MASK OriginalDesiredAccess;
    _In_ PVOID SourceProcess;
    _In_ PVOID TargetProcess;
} OB_PRE_DUPLICATE_HANDLE_INFORMATION, *POB_PRE_DUPLICATE_HANDLE_INFORMATION;
```

The `DesiredAccess` field can be modified as before. The extra information provided is the source process (from which a handle is being duplicated) and the target process (the process the new handle will be duplicated into). This allows the driver to query various properties of these processes before making a decision on how to modify (if at all) the desired access mask.



Notice that although both structures in the union are different, the first two members are the same, so they have the same layout in memory. This is useful for handling create and duplicate operations with the same code.

Post-Operation Callback

Post-operation callbacks are invoked after the operation completes. At this point, the driver cannot make any modifications, it can only look at the results. The post-operation callback receives the following structure:

```
typedef struct _OB_POST_OPERATION_INFORMATION {
    _In_ OB_OPERATION Operation;
    union {
        _In_ ULONG Flags;
        struct {
            _In_ ULONG KernelHandle:1;
            _In_ ULONG Reserved:31;
        };
    };
    _In_ PVOID Object;
    _In_ POBJECT_TYPE ObjectType;
    _In_ PVOID CallContext;
    _In_ NTSTATUS ReturnStatus;
    _In_ POB_POST_OPERATION_PARAMETERS Parameters;
} OB_POST_OPERATION_INFORMATION, *POB_POST_OPERATION_INFORMATION;
```

This looks similar to the pre-operation callback information, except for the following:

- The final status of the operation is returned in `ReturnStatus`. If successful, it means the client will get back a valid handle (possibly with a reduced access mask).
- The `Parameters` union provided has just one piece of information: the access mask granted to the client (assuming the status is successful).

The Process Protector Driver

The Process Protector driver is an example of using object callbacks. Its purpose is to protect certain processes from termination by denying the PROCESS_TERMINATE access mask from any client that requests it.

The driver should keep a list of protected processes. In this driver we'll use a simple limited array to hold the process IDs under the driver's protection. Here is the structure used to hold the driver's global data (defined in *Protector.h*):

```
#define PROCESS_TERMINATE 1

const int MaxPids = 256;

struct Globals {
    ULONG PidsCount;           // currently protected process count
    ULONG Pids[MaxPids];       // protected PIDs
    ExecutiveResource Lock;
    PVOID RegHandle;

    void Init() {
        Lock.Init();
    }

    void Term() {
        Lock.Delete();
    }
};
```



Notice that we must define PROCESS_TERMINATE explicitly, since it's not defined in the WDK headers (only PROCESS_ALL_ACCESS is defined). It's fairly easy to get its definition from user mode headers or documentation.

The ExecutiveResource type is the same used in chapter 9. It's important to use an Executive Resource here and not a (fast) mutex because we anticipate many more "reads" (checks if a process is under the driver's termination protection) than "writes" (adding or removing processes), so there is a clear advantage to an Executive Resource in this case. The main file (*Protector.cpp*) declares a global variable of type `Globals` named `g_Data`, calls `Init` in `DriverEntry`, and calls `Term` in the `Unload` routine, as we'll see shortly.

Object Notification Registration

The `DriverEntry` routine must include the registration to object callbacks for process objects. Here is the start of `DriverEntry`:

```
extern "C"
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    g_Data.Init();
```

Next, we prepare the structures for registration:

```
OB_OPERATION_REGISTRATION operation = {
    PsProcessType,           // object type
    OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE,
    OnPreOpenProcess,        nullptr      // pre, post
};

OB_CALLBACK_REGISTRATION reg = {
    OB_FLT_REGISTRATION_VERSION,
    1,                      // operation count
    RTL_CONSTANT_STRING(L"12345.6171"), // altitude
    nullptr,            // context
    &operation             // single operation
};
```

The registration is for process objects only, with a pre-callback provided. This callback should remove the PROCESS_TERMINATE access mask from the desired access requested by the client.

Now we're ready to do perform all standard initializatio, including objack callback registration:

```
auto status = STATUS_SUCCESS;
UNICODE_STRING deviceName = RTL_CONSTANT_STRING(L"\Device\KProtect");
UNICODE_STRING symName = RTL_CONSTANT_STRING(L"\?\?\Device\KProtect");
PDEVICE_OBJECT DeviceObject = nullptr;

do {
    status = ObRegisterCallbacks(&reg, &g_Data.RegHandle);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to register callbacks (0x%08X)\n",
                 status));
        break;
    }

    status = IoCreateDevice(DriverObject, 0, &deviceName, FILE_DEVICE_UNKNOWN,
                           0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "failed to create device object (0x%08X)\n",
                 status));
        break;
    }
}
```

```

    }

status = IoCreateSymbolicLink(&symName, &deviceName);
if (!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to create symbolic link (0x%08X)\n",
              status));
    break;
}
} while (false);

```

The rest of `DriverEntry` is nothing new, shown here for completeness:

```

if (!NT_SUCCESS(status)) {
    if (g_Data.RegHandle)
        ObUnRegisterCallbacks(g_Data.RegHandle);
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    return status;
}

DriverObject->DriverUnload = ProtectUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = ProtectCreateClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ProtectDeviceControl;

return status;
}

```

Managing Protected Processes

The driver maintains an array of process IDs for processes under its protection. Managing these process IDs is done by exposing three control codes (in `ProtectorPublic.h`):

```

#define KPROTECT_DEVICE 0x8101

#define IOCTL_PROTECT_ADD_PID \
    CTL_CODE(KPROTECT_DEVICE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROTECT_REMOVE_PID \
    CTL_CODE(KPROTECT_DEVICE, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_PROTECT_REMOVE_ALL \
    CTL_CODE(KPROTECT_DEVICE, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)

```

Before implementing the I/O Control codes, we should write functions to add processes, remove processes, and find whether a specific PID is under the driver's protection. Here is the function to add an array of process IDs:

```
ULONG AddProcesses(const ULONG* pids, ULONG count) {
    ULONG added = 0;
    ULONG current = 0;

    Locker locker(g_Data.Lock);
    for (int i = 0; i < MaxPids && added < count; i++) {
        if (g_Data.Pids[i] == 0) {
            g_Data.Pids[i] = pids[current++];
            added++;
        }
    }
    g_Data.PidsCount += added;
    return added;
}
```

The function acquires the Executive Resource exclusively, as it is going to change the the PIDs array. The loop body looks for an “empty” slot (where the PID is zero). If it finds one, it changes the value to the current PID to house, and then moves on to the next. Finally, AddProcesses returns the number of added PIDs.

The function does not check if the PID was already added. It doesn’t cause any particular issues, but it might be nice to check for duplication, at the expense of a higher running time.

The opposite function to remove an array of PIDs is RemoveProcesses:

```
ULONG RemoveProcesses(const ULONG* pids, ULONG count) {
    ULONG removed = 0;

    Locker locker(g_Data.Lock);
    for (int i = 0; i < MaxPids && removed < count; i++) {
        auto pid = g_Data.Pids[i];
        if(pid) {
            for (ULONG c = 0; c < count; c++) {
                if (pid == pids[c]) {
                    g_Data.Pids[i] = 0;
                    removed++;
                    break;
                }
            }
        }
    }
}
```

```

    g_Data.PidsCount -= removed;
    return removed;
}

```

This function does the reverse - when it finds a non-zero PID, it searches the PIDs to remove with the current PID, and if found, removes the PID by zeroing the entry in the array.

Lastly, `FindProcess` searches for a PID in the array:

```

int FindProcess(ULONG pid) {
    SharedLocker locker(g_Data.Lock);
    ULONG exist = 0;
    for (int i = 0; i < MaxPids && exist < g_Data.PidsCount; i++) {
        if (g_Data.Pids[i] == 0)
            continue;
        if (g_Data.Pids[i] == pid)
            return i;
        exist++;
    }
    return -1;
}

```

This is a function we expect to be called many more times than `AddProcesses` or `RemoveProcesses` - it should be called any time clients call `OpenProcess` or `DuplicateHandle` with a process handle to duplicate. Any number of threads can be making such calls at any time. This is why it's important to make the function as efficient as possible.

The function does not change the PIDs array, which is why it can acquire the Executive Resource in shared mode (and thus improve concurrency). Then the PID is searched in the array, returning its index if found, or -1 if it can't be found. Failing to find the PID should be the common case since the driver is likely to protect a small number of processes. This is why the number of non-zero PIDs is counted, and if it reaches the number of PIDs protected (`g_Data.PidsCount`), the loop can be exited early before the entire `MaxPids` elements are traversed.

Now we're ready to implement the `IRP_MJ_DEVICE_CONTROL` dispatch routine. We'll start normally, by preparing the information we need:

```

NTSTATUS ProtectDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG info = 0;
    auto inputLen = dic.InputBufferLength;

```

Adding and removing PIDs IOCTLs accept the same information - an array of `ULONG` values representing one or more PIDs. We can share their implementation like so:

```

switch (dic.IoControlCode) {
    case IOCTL_PROTECT_ADD_PID:
    case IOCTL_PROTECT_REMOVE_PID:
    {
        if (inputLen == 0 || inputLen % sizeof(ULONG) != 0) {
            status = STATUS_INVALID_BUFFER_SIZE;
            break;
        }
        auto pids = (ULONG*)Irp->AssociatedIrp.SystemBuffer;
        if (pids == nullptr) {
            status = STATUS_INVALID_PARAMETER;
            break;
        }
        ULONG count = inputLen / sizeof(ULONG);
        auto added = dic.IoControlCode == IOCTL_PROTECT_ADD_PID
            ? AddProcesses(pids, count) : RemoveProcesses(pids, count);
        status = added == count ? STATUS_SUCCESS : STATUS_NOT_ALL_ASSIGNED;
        info = added * sizeof(ULONG);
        break;
    }
}

```

First we have the usual checks for a proper buffer size and the system buffer being non-NULL. Then, it's just a matter of calling `AddProcesses` or `RemoveProcesses` as needed. The final status is set to `STATUS_SUCCESS` if all the provided PIDs are added or removed. Otherwise, `STATUS_NOT_ALL_ASSIGNED` is set as the error value. This status is returned from trying to enable privileges in a token, hijacked here as a convenience (or more likely laziness on my part).

Removing all processes is fairly simple, done directly in the `case` itself:

```

case IOCTL_PROTECT_REMOVE_ALL:
    Locker locker(g_Data.Lock);
    Rt1ZeroMemory(g_Data.Pids, sizeof(g_Data.Pids));
    g_Data.PidsCount = 0;
    status = STATUS_SUCCESS;
    break;
}

return CompleteRequest(Irp, status, info);
}

```

Removing all PIDs is just clearing the PIDs array and resetting the count of protected processes to zero.

Finally, `CompleteRequest` is used to complete the IRP with the current status and information, the same helper function we used in chapter 9.

The Pre-Callback

The most important part of the driver is removing the PROCESS_TERMINATE access mask for PIDs that are currently being protected:

```
OB_PREOP_CALLBACK_STATUS
OnPreOpenProcess(PVOID, POB_PRE_OPERATION_INFORMATION Info) {
    if(Info->KernelHandle)
        return OB_PREOP_SUCCESS;

    auto process = (PEPROCESS)Info->Object;
    auto pid = HandleToULong(PsGetProcessId(process));

    AutoLock locker(g_Data.Lock);
    if (FindProcess(pid)) {
        // found in list, remove terminate access
        Info->Parameters->CreateHandleInformation.DesiredAccess &=
            ~PROCESS_TERMINATE;
    }

    return OB_PREOP_SUCCESS;
}
```

If the handle is a kernel handle, we let the operation continue normally, since we don't want to stop kernel code from working properly.

Now we need the process ID for which a handle is being opened. The data provided in the callback as the object pointer. Fortunately, getting the PID is simple with the PsGetProcessId API. It accepts a PEPROCESS and returns its ID.

The last part is checking whether we're actually protecting this particular process or not, so we call FindProcess under the protection of the lock. If found, we remove the PROCESS_TERMINATE access mask.

The Client Application

The client application should be able to add, remove and clear processes by issuing correct DeviceIoControl calls. The command line interface is demonstrated by the following commands (assuming the executable is *Protect.exe*):

Protect.exe add 1200 2820 (protect PIDs 1200 and 2820)

Protect.exe remove 2820 (remove protection from PID 2820)

Protect.exe clear (remove all PIDs from protection)

Here is the main function:

```
int wmain(int argc, const wchar_t* argv[]) {
    if(argc < 2)
        return PrintUsage();

    enum class Options {
        Unknown,
        Add, Remove, Clear
    };
    Options option;
    if (::wcsicmp(argv[1], L"add") == 0)
        option = Options::Add;
    else if (::wcsicmp(argv[1], L"remove") == 0)
        option = Options::Remove;
    else if (::wcsicmp(argv[1], L"clear") == 0)
        option = Options::Clear;
    else {
        printf("Unknown option.\n");
        return PrintUsage();
    }

    HANDLE hFile = ::CreateFile(L"\\\\\\\\.\\\" PROCESS_PROTECT_NAME,
        GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hFile == INVALID_HANDLE_VALUE)
        return Error("Failed to open device");

    std::vector<DWORD> pids;
    BOOL success = FALSE;
    DWORD bytes;
    switch (option) {
        case Options::Add:
            pids = ParsePids(argv + 2, argc - 2);
            success = ::DeviceIoControl(hFile, IOCTL_PROCESS_PROTECT_BY_PID,
                pids.data(), static_cast<DWORD>(pids.size()) * sizeof(DWORD),
                nullptr, 0, &bytes, nullptr);
            break;

        case Options::Remove:
            pids = ParsePids(argv + 2, argc - 2);
            success = ::DeviceIoControl(hFile, IOCTL_PROCESS_UNPROTECT_BY_PID,
                pids.data(), static_cast<DWORD>(pids.size()) * sizeof(DWORD),
                nullptr, 0, &bytes, nullptr);
            break;
    }
}
```

```

    case Options::Clear:
        success = ::DeviceIoControl(hFile, IOCTL_PROCESS_PROTECT_CLEAR,
            nullptr, 0, nullptr, 0, &bytes, nullptr);
        break;

    }

    if (!success)
        return Error("Failed in DeviceIoControl");

    printf("Operation succeeded.\n");

    ::CloseHandle(hFile);

    return 0;
}

```

The *ParsePids* helper function parses process IDs and returns them as a `std::vector<DWORD>` that is easy to pass as an array by using the `data()` method on `std::vector<T>`:

```

std::vector<DWORD> ParsePids(const wchar_t* buffer[], int count) {
    std::vector<DWORD> pids;
    for (int i = 0; i < count; i++)
        pids.push_back(_wtoi(buffer[i]));
    return pids;
}

```

Finally, the *Error* function is the same we used in previous projects, while *PrintUsage* just displays simple usage information.

The driver is installed in the usual way, and then started:

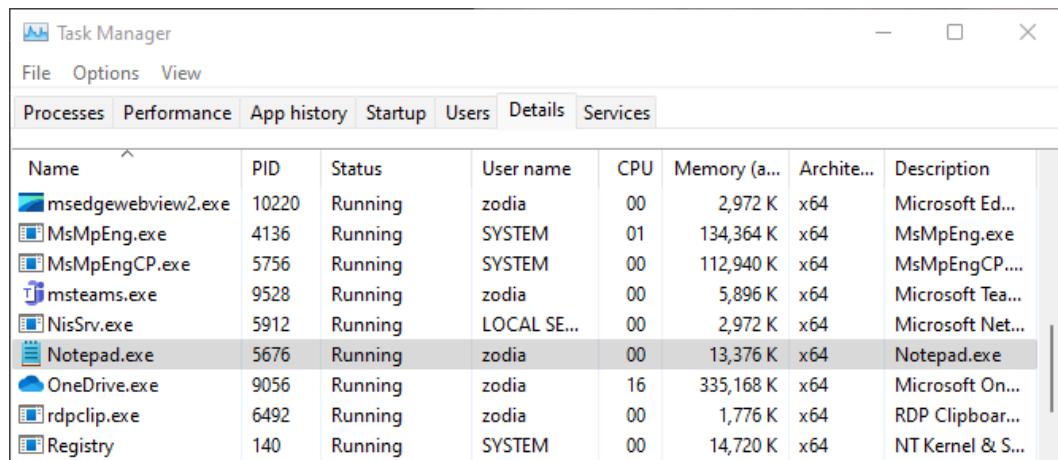
```

sc create protect type= kernel binPath= c:\book\processprotect.sys

sc start protect

```

Let's test it by launching a process (*Notepad.exe*) as an example, protecting it, and then trying to kill it with *Task Manager*. Figure 10-1 shows the notepad instance running.



The screenshot shows the Windows Task Manager window with the 'Details' tab selected. The table lists various processes, including 'Notepad.exe' which is highlighted. The columns show Name, PID, Status, User name, CPU, Memory (a...), Archite..., and Description. The 'Notepad.exe' row has a PID of 5676, is running, and is owned by user 'zodia'.

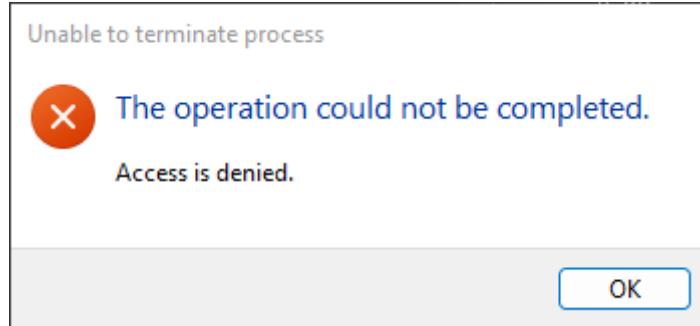
Name	PID	Status	User name	CPU	Memory (a...)	Archite...	Description
msedgewebview2.exe	10220	Running	zodia	00	2,972 K	x64	Microsoft Ed...
MsMpEng.exe	4136	Running	SYSTEM	01	134,364 K	x64	MsMpEng.exe
MsMpEngCP.exe	5756	Running	SYSTEM	00	112,940 K	x64	MsMpEngCP....
msteams.exe	9528	Running	zodia	00	5,896 K	x64	Microsoft Tea...
NisSrv.exe	5912	Running	LOCAL SE...	00	2,972 K	x64	Microsoft Net...
Notepad.exe	5676	Running	zodia	00	13,376 K	x64	Notepad.exe
OneDrive.exe	9056	Running	zodia	16	335,168 K	x64	Microsoft On...
rdclip.exe	6492	Running	zodia	00	1,776 K	x64	RDP Clipboar...
Registry	140	Running	SYSTEM	00	14,720 K	x64	NT Kernel & S...

Figure 10-1: Notepad running

Now protect it:

```
protect add 5676
```

Clicking End task in *Task Manager*, pops up an error, shown in Figure 10-2.

Figure 10-2: Attempting to terminate *notepad*

We can remove the protection and try again. This time the process is terminated as expected.

```
protect remove 5676
```

In the case of notepad, even with protection, clicking the window close button or selecting *File/Exit* from the menu would terminate the process. This is because it's being done internally by calling `ExitProcess` which does not involve any handles being opened. This means the protection mechanism we devised here is good for processes without any user interface.



Add a control code that allows querying the currently protected processes.

Registry Notifications

Somewhat similar to object notifications, the *Configuration Manager* (the part in the *Executive* that manages the Registry) can be used to register for notifications when Registry keys or values are accessed.

Before we look at Registry callbacks, some background on the Registry itself might be helpful.

Registry Overview

The Registry is a fairly well-known artifact in Windows; it's a hierarchical database, used to store system-wide and user-related information. Most of the data in the Registry is persisted in files, but some is generated dynamically and not persisted (volatile).

The typical tool used to examine the Registry is *RegEdit*, part of Windows. Figure 10-3 shows the *hives* shown when running *RegEdit*. The documented user-mode APIs use this layout of the Registry in order to access keys.

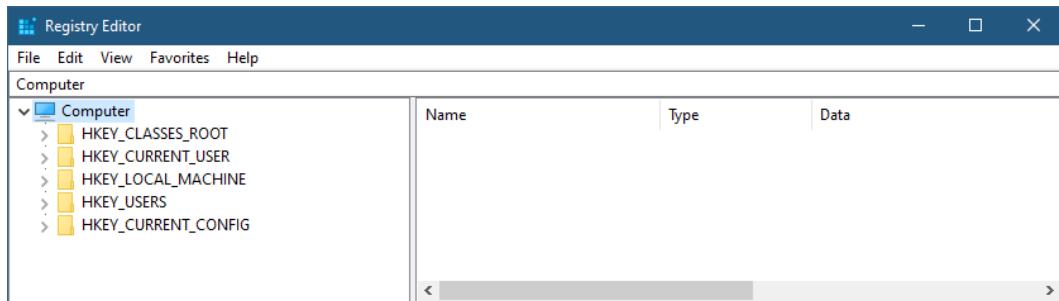
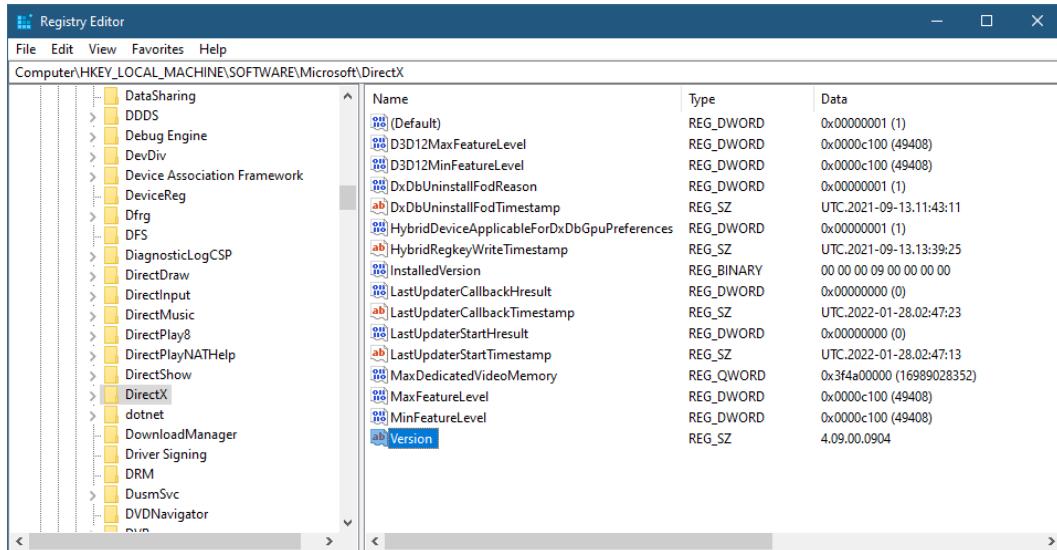


Figure 10-3: The *hives* shown in *RegEdit*

The following user-mode example shows how to open the *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX* key for read access, and read in the *Version* value, which happens to be a string (figure 10-4):

Figure 10-4: The `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX` key

```

HKEY hKey;
DWORD error = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    L"SOFTWARE\\Microsoft\\DirectX", 0, KEY_READ, &hKey);
if (ERROR_SUCCESS == error) {
    WCHAR version[64];
    ULONG count = sizeof(version);
    error = RegQueryValueEx(hKey, L"Version", nullptr, nullptr,
        (BYTE*)version, &count);
    if (ERROR_SUCCESS == error) {
        printf("DirectX version: %ws\n", version);
    }
    RegCloseKey(hKey);
}

```

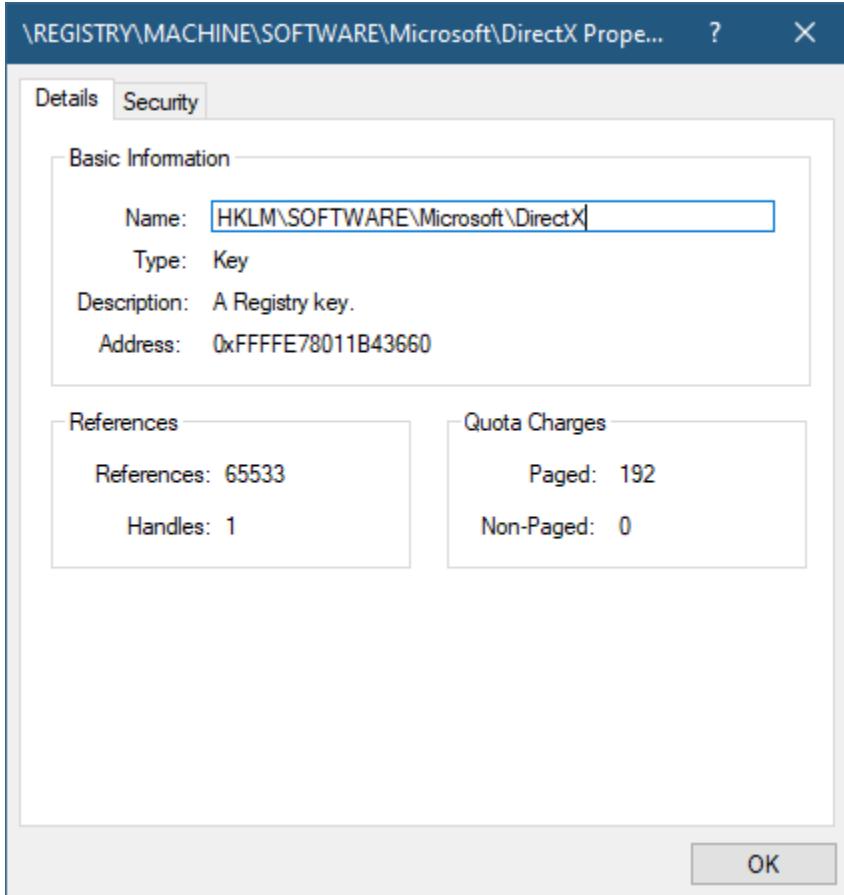
More details about the user-mode Registry API can be found in chapter 15 of my book “Windows 10 System Programming, part 2”.

If you run this little piece of code, and examine the key handle returned from `RegOpenKeyEx` in *Process Explorer*, you’ll see something like figure 10-5. The key “name” seems to be what we have used.

0x00000084	Key	HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions	0x00020019	0xFFFFE78011B5430	READ_CONTROL KEY_READ	
0x00000088	Key	HKLM\SYSTEM\ControlSet001\Control\Session Manager	0x00000001	0xFFFFE78011B5B3D0	QUERY_VALUE	
0x000000A0	Key	HKLM	0x00020019	0xFFFFE78011B43440	READ_CONTROL KEY_READ	
0x000000A4	Key	HKLM\SOFTWARE\Microsoft\DirectX	0x00020019	0xFFFFE78011B43660	READ_CONTROL KEY_READ	

Figure 10-5: Registry key handle in *Process Explorer*

However, if you double-click the handle to show the object's (key) properties, you'll see something similar to figure 10-6.

Figure 10-6: Registry key properties *Process Explorer*

Notice the key name in the title bar. We can confirm the name by copying the real object address and feeding it to a kernel debugger using the !object command:

```
1kd> !object 0xFFFFE78011B43660
Object: fffffe78011b43660  Type: (fffffb90f07d8a220) Key
  ObjectHeader: fffffe78011b43630 (new version)
  HandleCount: 1  PointerCount: 32767
  Directory Object: 00000000 Name: \REGISTRY\MACHINE\SOFTWARE\MICROSOFT\DIRECTX
```

The “real” key name starts with “REGISTRY”, which is in fact a named kernel object stored at the root of the Object Manager’s namespace (figure 10-7).

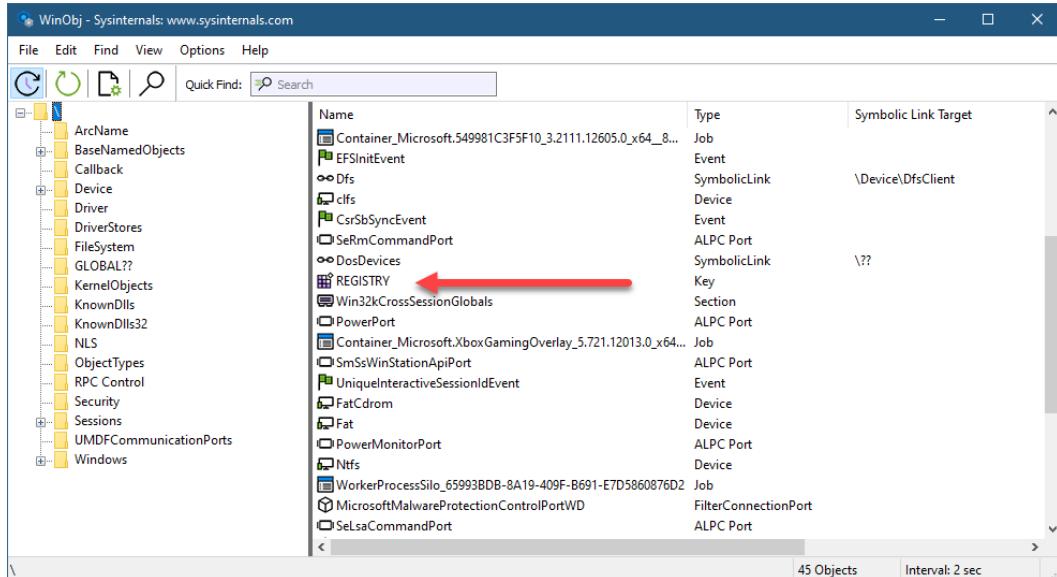


Figure 10-7: The Registry key object in *WinObj*

Clearly, the names used to access keys from documented Windows APIs go through some “translation”, changing *HKEY_LOCAL_MACHINE* to *REGISTRY\MACHINE*. To see the entire picture, showing the “real” Registry, you can use my *RegExp* tool, downloadable from my Github repo (figure 10-8). It shows both the Registry as observed by user-mode APIs (upper part) and the real Registry (lower part), as used internally within the kernel.

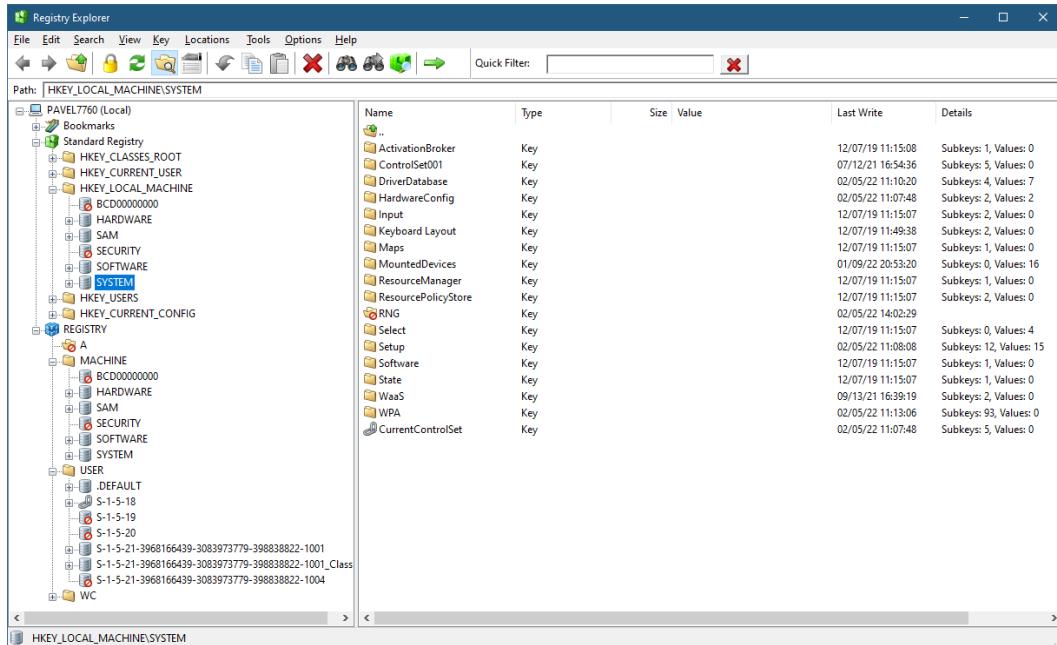
Figure 10-8: The *Registry Explorer* tool

Table 10-1 shows the “translations” for common key names.

Table 10-1: Registry keys

User-facing key name	Real key name	Notes
HKEY_LOCAL_MACHINE	REGISTRY\MACHINE	
HKEY_USERS	REGISTRY\USERS	
HKEY_CURRENT_USER	REGISTRY\USER\{userSID}	
(no equivalence)	REGISTRY\A	Root of private process keys
(no equivalence)	REGISTRY\WC	Root of keys for Windows Containers (silos)

All the key names received/handled with the following Registry notifications always use the real key names.

Using Registry Notifications

The `CmRegisterCallbackEx` API is used to register for such notifications. Its prototype is as follows:

```
NTSTATUS CmRegisterCallbackEx (
    _In_      PEX_CALLBACK_FUNCTION Function,
    _In_      PCUNICODE_STRING     Altitude,
    _In_      PVOID                Driver,           // PDRIVER_OBJECT
    _In_opt_   PVOID                Context,
    _Out_     PLARGE_INTEGER       Cookie,
    _Reserved_ PVOID               Reserved
)
```

Function is the callback itself, which we'll look at in a moment. *Altitude* is the driver's callback altitude, which essentially has the same meaning as it has with object callbacks. The *Driver* argument should be the driver object provided to `DriverEntry`. *Context* is a driver-defined value passed as-is to the callback. Finally, *Cookie* is the result of the registration if successful. This cookie should be passed to `CmUnregisterCallback` to unregister.

It's a bit annoying that all the various registration APIs are inconsistent with respect to registration/unregistration: `CmRegisterCallbackEx` returns a `LARGE_INTEGER` as representing the registration; `ObRegisterCallbacks` returns a `PVOID`; process and thread registration functions return nothing (internally use the address of the callback itself to identify the registration). Finally, process and thread unregistration is done with asymmetric APIs; Oh well.

The callback function is very generic, shown here:

```
NTSTATUS RegistryCallback (
    _In_      PVOID CallbackContext,
    _In_opt_   PVOID Argument1,
    _In_opt_   PVOID Argument2);
```

CallbackContext is the *Context* argument passed to `CmRegisterCallbackEx`. The first generic argument is really an enumeration, `REG_NOTIFY_CLASS`, describing the operation for which the callback is being invoked. The second argument is a pointer to a specific structure relevant to this type of notification. A driver will typically switch on the notification type like so:

```
NTSTATUS OnRegistryNotify(PVOID, PVOID Argument1, PVOID Argument2) {
    switch ((REG_NOTIFY_CLASS)(ULONG_PTR)Argument1) {
        //...
    }
}
```

The callback is called at IRQL PASSIVE_LEVEL (0) by the thread performing the operation.

Table 10-2 shows some values from the `REG_NOTIFY_CLASS` enumeration and the corresponding structure passed in as *Argument2*.

Table 10-2: Some registry notifications and associated structures

Notification	Associated structure
RegNtPreDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPostDeleteKey	REG_POST_OPERATION_INFORMATION
RegNtPreSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPostSetValueKey	REG_POST_OPERATION_INFORMATION
RegNtPreCreateKey	REG_PRE_CREATE_KEY_INFORMATION
RegNtPostCreateKey	REG_POST_CREATE_KEY_INFORMATION

Handling Pre-Notifications

The callback is called for pre-operations before these are carried out by the Configuration Manager. At that point, the driver has the following options:

- Returning STATUS_SUCCESS from the callback instructs the Configuration Manager to continue processing the operation normally (including calling other drivers that have registered for notifications).
- Return some failure status from the callback. In this case, the Configuration Manager returns to the caller with that status, and the post-operation will not be invoked.
- Handle the request in some way, and then return STATUS_CALLBACK_BYPASS from the callback. The Configuration Manager returns success to the caller and does not invoke the post-operation. The driver must take care to set proper values in the REG_xxx_KEY_INFORMATION structure provided in the callback.

Handling Post-Operations

After the operation is completed, and assuming the driver did not prevent the post-operation from occurring, the callback is invoked after the Configuration Manager performs the requested operation. The structure provided for many post operations is shown here:

```
typedef struct _REG_POST_OPERATION_INFORMATION {
    PVOID     Object;           // input
    NTSTATUS  Status;          // input
    PVOID     PreInformation;   // The pre information
    NTSTATUS  ReturnStatus;    // can change the outcome of the operation
    PVOID     CallContext;
    PVOID     ObjectContext;
    PVOID     Reserved;
} REG_POST_OPERATION_INFORMATION,*PREG_POST_OPERATION_INFORMATION;
```

The callback has the following options for a post-operation:

- Look at the operation result and do something benign (log it, for instance).
- Modify the return status by setting a new status value in the `ReturnStatus` field of the post-operation structure, and return `STATUS_CALLBACK_BYPASS` from the callback. The Configuration Manager returns this new status to the caller.
- Modify the output parameters in the `REG_xxx_KEY_INFORMATION` structure and return `STATUS_SUCCESS`. The Configuration Manager returns this new data to the caller.



The `PreInformation` member of the post-operation structure points to the pre-information structure associated with that operation.



Care must be taken if data is changed when a post-operation, or if a successful status is changed to a failed one or vice versa. This might require the driver to deallocate or allocate key objects.

Extending the *SysMon* Driver

We'll extend our *SysMon* driver from chapter 9 to include notifications for a Registry operation. As an example, we'll add notifications for write operations to anywhere under `HKEY_LOCAL_MACHINE`.

First, we'll define a data structure that would include the reported information (in `SysMonPublic.h`):

```
struct RegistrySetValueInfo : ItemHeader {
    ULONG ProcessId;
    ULONG ThreadId;
    USHORT KeyNameOffset; // from beginning of structure
    USHORT ValueNameOffset; // from beginning of structure
    ULONG DataType; // REG_xxx
    ULONG DataSize; // actual size
    USHORT DataOffset;
    USHORT ProvidedDataSize;
};
```

Key names, value names and values could be large, so it's best not to use fixed-size arrays (although that would be much simpler), but store offsets to the names and value. Each name will be NULL-terminated, which avoids the need to store lengths of strings (as we did in the command line case in chapter 9). The data itself could be arbitrarily large, so we'll have to decide on a maximum length to copy as part of the notification.

`DataType` is one of the `REG_xxx` type constants, such as `REG_SZ`, `REG_DWORD`, `REG_BINARY`, etc. These values are the same as used with user-mode APIs.

Next, we'll add a new event type for this notification:

```
enum class ItemType : short {
    None,
    ProcessCreate,
    ProcessExit,
    ThreadCreate,
    ThreadExit,
    ImageLoad,
    RegistrySetValue // new value
};
```

It's possible to subdivide Registry notifications further by defining a Registry item type and then define specific items for different Registry operations. In this example, we just add one specific Registry operation, but you may want to take the more generic approach if multiple Registry operations are of interest.

In `DriverEntry`, we need to add registry callback registration as part of the `do/while(false)` block. The returned cookie representing the registration is stored in a global variable:

```
UNICODE_STRING altitude = RTL_CONSTANT_STRING(L"7657.124");
status = CmRegisterCallbackEx(OnRegistryNotify, &altitude, DriverObject,
    nullptr, &g_RegCookie, nullptr);
if(!NT_SUCCESS(status)) {
    KdPrint((DRIVER_PREFIX "failed to set registry callback (%08X)\n",
        status));
    break;
}
```



It would have been better to encapsulate all state in the `Globals` structure and provide methods for initializing and uninitializing all the callbacks within this class. This is left as an exercise to the reader.

We must also unregister the notification in the `Unload` routine:

```
CmUnRegisterCallback(g_RegCookie);
```

Handling Registry Callback

Our callback should only care about writes done to `HKEY_LOCAL_MACHINE`. First, we switch on the operation of interest:

```
NTSTATUS OnRegistryNotify(PVOID context, PVOID arg1, PVOID arg2) {
    UNREFERENCED_PARAMETER(context);

    switch ((REG_NOTIFY_CLASS)(ULONG_PTR)arg1) {
        case RegNtPostSetValueKey:
            // ...
    }
    return STATUS_SUCCESS;
}
```

In this driver we don't care about any other operation, so after the `switch` we simply return a successful status. Note that we examine the post-operation, since only the result is interesting for this driver. Next, inside the `case` we care about, we cast the second argument to the post-operation data and check if the operation succeeded:

```
auto args = (REG_POST_OPERATION_INFORMATION*)arg2;
if (!NT_SUCCESS(args->Status))
    break;
```

If the operation is not successful, we bail out. This is just an arbitrary decision for this driver; a different driver might be interested in these failed attempts.

Next, we need to check if the key in question is under `HKEY_LOCAL_MACHINE`, which as we've seen is in actuality `\REGISTRY\MACHINE`.

The key path is not stored in the post-structure and not even stored in the pre-structure directly. Instead, the Registry key object itself is provided as part of the post-information structure. We then need to extract the key name with `CmCallbackGetKeyObjectIDEx` (Windows 8+) or `CmCallbackGetKeyObjectID` (earlier versions), and see if it's starting with `\REGISTRY\MACHINE`. These APIs are declared as follows:

```
NTSTATUS CmCallbackGetKeyObjectID (
    _In_          PLARGE_INTEGER     Cookie,
    _In_          PVOID             Object,
    _Out_opt_     PULONG_PTR       ObjectID,
    _Outptr_opt_ PCUNICODE_STRING *ObjectName);

NTSTATUS CmCallbackGetKeyObjectIDEx (
    _In_          PLARGE_INTEGER     Cookie,
    _In_          PVOID             Object,
    _Out_opt_     PULONG_PTR       ObjectID,
    _Outptr_opt_ PCUNICODE_STRING *ObjectName,
    _In_          ULONG            Flags); // must be zero
```

`Cookie` identifies the registration cookie returned from `CmRegisterCallbackEx`, identifying the driver. `Object` is the Registry key whose name we need. `ObjectID` is an optional returned value

that provides the unique identifier of the key in question. Finally, *ObjectName* is a pointer to a `UNICODE_STRING` pointer retruned with the full key name itself.

The two APIs are identical from a parameter perspective, as the *Flags* argument to `CmCallbackGetKeyObjectIDEx` must be zero. There are differences in implementation, however:

First, The returned key name from `CmCallbackGetKeyObjectID` is valid until the last handle of the key is closed. With `CmCallbackGetKeyObjectIDEx`, the name must be freed by calling `CmCallbackReleaseKeyObjectIDEx`:

```
VOID CmCallbackReleaseKeyObjectIDEx (_In_ PCUNICODE_STRING ObjectName);
```

Second, if the name of the Registry key is changed after it's been obtained with `CmCallbackGetKeyObjectID`, subsequent calls to `CmCallbackGetKeyObjectID` will return the old, stale, name. In contrast, `CmCallbackReleaseKeyObjectIDEx` always returns the current key name.



Call `CmCallbackReleaseKeyObjectIDEx` if you're targeting Windows 8 and later.

Here is the call to obtain the key name and checking if it's part of *HKLM*:

```
static const WCHAR machine[] = L"\REGISTRY\MACHINE\";  
PCUNICODE_STRING name;  
if (NT_SUCCESS(CmCallbackGetKeyObjectIDEx(&g_RegCookie, args->Object,  
    nullptr, &name, 0))) {  
    if (wcsncmp(name->Buffer, machine, ARRSIZE(machine) - 1) == 0) {
```

If the condition holds, then we need to capture the information of the operation into our notification structure and add it to the queue. The needed information (data type, value name, actual value, etc.) is provided with the pre-information structure that is luckily available as part of the post-information structure we receive directly.

```
auto preInfo = (REG_SET_VALUE_KEY_INFORMATION*)args->PreInformation;  
NT_ASSERT(preInfo);
```

Calculating the correct size to allocate is more involved than previous cases, as we have several variable-length strings to deal with. We can start with the base data structure size and then add the sizes (in bytes) of the strings (not forgetting to leave room for a terminating NULL):

```

USHORT size = sizeof(RegistrySetValueInfo);
USHORT keyNameLen = name->Length + sizeof(WCHAR);
USHORT valueNameLen = preInfo->ValueName->Length + sizeof(WCHAR);
//
// restrict copied data to 256 bytes
//
USHORT valueSize = (USHORT)min(256, preInfo->DataSize);
size += keyNameLen + valueNameLen + valueSize;

```

The driver stores the data itself, and since it's unbounded in theory, we decide to store no more than 256 bytes. We will still report the true size of the data - the data itself may be truncated.

Now comes the real work of making the allocation and filling all the details. First, the fixed-size data, including the header:

```

auto info = (FullItem<RegistrySetValueInfo>*)ExAllocatePoolWithTag(PagedPool,
    size + sizeof(LIST_ENTRY), DRIVER_TAG);
if (info) {
    auto& data = info->Data;
    KeQuerySystemTimePrecise(&data.Time);
    data.Type = ItemType::RegistrySetValue;
    data.Size = size;
    data.DataType = preInfo->Type;
    data.ProcessId = HandleToULong(PsGetCurrentProcessId());
    data.ThreadId = HandleToUlong(PsGetCurrentThreadId());
    data.ProvidedDataSize = valueSize;
    data.DataSize = preInfo->DataSize;
}

```

Next, we copy the strings and set the offsets:

```

// first offset starts at the end of the structure
//
USHORT offset = sizeof(data);
data.KeyNameOffset = offset;
wcsncpy_s((PWSTR)((PUCHAR)&data + offset),
    keyNameLen / sizeof(WCHAR), name->Buffer,
    name->Length / sizeof(WCHAR));
offset += keyNameLen;
data.ValueNameOffset = offset;
wcsncpy_s((PWSTR)((PUCHAR)&data + offset),
    valueNameLen / sizeof(WCHAR), preInfo->ValueName->Buffer,
    preInfo->ValueName->Length / sizeof(WCHAR));
offset += valueNameLen;

```

```

data.DataOffset = offset;
memcpy((PUCHAR)&data + offset, preInfo->Data, valueSize);

// finally, add the item
g_State.AddItem(&info->Entry);

```

Using `wcsncpy_s` to copy the strings is a good choice in this case, since it appends NULL at the end of strings (if there is enough space, and we made sure of that).

Finally, if `CmCallbackGetKeyObjectIDEx` succeeds, the resulting key name must be explicitly freed:

```
CmCallbackReleaseKeyObjectIDEx(name);
```

Here is the full function for convenience:

```

NTSTATUS OnRegistryNotify(PVOID context, PVOID arg1, PVOID arg2) {
    UNREFERENCED_PARAMETER(context);

    switch ((REG_NOTIFY_CLASS)(ULONG_PTR)arg1) {
        case RegNtPostSetValueKey:
            auto args = (REG_POST_OPERATION_INFORMATION*)arg2;
            if (!NT_SUCCESS(args->Status))
                break;

            static const WCHAR machine[] = L"\\"REGISTRY"\\"MACHINE\\";

            PCUNICODE_STRING name;
            if (NT_SUCCESS(CmCallbackGetKeyObjectIDEx(
                &g_RegCookie, args->Object, nullptr, &name, 0))) {
                //
                // look for HKLM subkeys
                //

                if (wcsncmp(name->Buffer, machine, ARRAYSIZE(machine) - 1) == 0) {
                    auto preInfo = (REG_SET_VALUE_KEY_INFORMATION*)args->PreInformation;
                    USHORT size = sizeof(RegistrySetValueInfo);
                    USHORT keyNameLen = name->Length + sizeof(WCHAR);
                    USHORT valueNameLen = preInfo->ValueName->Length + sizeof(WCHAR);
                    //
                    // restrict copied data to 256 bytes
                    //

                    USHORT valueSize = (USHORT)min(256, preInfo->DataSize);
                    size += keyNameLen + valueNameLen + valueSize;
                    auto info = (FullItem<RegistrySetValueInfo>*)
                }
            }
    }
}

```

```
        ExAllocatePoolWithTag(PagedPool,
            size + sizeof(LIST_ENTRY), DRIVER_TAG);
    if (info) {
        auto& data = info->Data;
        KeQuerySystemTimePrecise(&data.Time);
        data.Type = ItemType::RegistrySetValue;
        data.Size = size;
        data.DataType = preInfo->Type;
        data.ProcessId = HandleToULong(PsGetCurrentProcessId());
        data.ThreadId = HandleToUlong(PsGetCurrentThreadId());
        data.ProvidedDataSize = valueSize;
        data.DataSize = preInfo->DataSize;
        //
        // first offset starts at the end of the structure
        //
        USHORT offset = sizeof(data);
        data.KeyNameOffset = offset;
        wcsncpy_s((PWSTR)((PUCHAR)&data + offset),
            keyNameLen / sizeof(WCHAR), name->Buffer,
            name->Length / sizeof(WCHAR));
        offset += keyNameLen;
        data.ValueNameOffset = offset;
        wcsncpy_s((PWSTR)((PUCHAR)&data + offset),
            valueNameLen / sizeof(WCHAR), preInfo->ValueName->Buffer,
            preInfo->ValueName->Length / sizeof(WCHAR));
        offset += valueNameLen;
        data.DataOffset = offset;
        memcpy((PUCHAR)&data + offset, preInfo->Data, valueSize);
        g_State.AddItem(&info->Entry);
    }
    else {
        KdPrint((DRIVER_PREFIX
            "Failed to allocate memory for registry set value\n"));
    }
}
CmCallbackReleaseKeyObjectIDEx(name);
}
break;
}
return STATUS_SUCCESS;
}
```

Modified Client Code

The client application should be modified to support this new event type. Here is the case added as part of `DisplayInfo`:

```
case ItemType::RegistrySetValue:
{
    DisplayTime(header->Time);
    auto info = (RegistrySetValueInfo*)buffer;
    printf("Registry write PID=%u, TID=%u: %ws\\%ws type: %d size: %d data: ",
        info->ProcessId, info->ThreadId,
        (PCWSTR)((PBYTE)info + info->KeyNameOffset),
        (PCWSTR)((PBYTE)info + info->ValueNameOffset),
        info->DataType, info->DataSize);
    DisplayRegistryValue(info);
    break;
}
```

The data itself is displayed by a helper function, `DisplayRegistryValue`:

```
void DisplayRegistryValue(const RegistrySetValueInfo* info) {
    auto data = (PBYTE)info + info->DataOffset;
    switch (info->DataType) {
        case REG_DWORD:
            printf("0x%08X (%u)\n", *(DWORD*)data, *(DWORD*)data);
            break;

        case REG_SZ:
        case REG_EXPAND_SZ:
            printf("%ws\n", (PCWSTR)data);
            break;

        // add other cases... (REG_QWORD, REG_LINK, etc.)

        default:
            DisplayBinary(data, info->ProvidedDataSize);
            break;
    }
}
```

`DisplayBinary` is a simple helper function that shows binary data as a series of hex values shown here for completeness:

```

void DisplayBinary(const BYTE* buffer, DWORD size) {
    printf("\n");
    for (DWORD i = 0; i < size; i++) {
        printf("%02X ", buffer[i]);
        //
        // go to new line every 16 values
        //
        if ((i + 1) % 16 == 0)
            printf("\n");
    }
    printf("\n");
}

```

Here is some output for this enhanced client and driver:

```

11:14:13.991: Registry write PID=5076, TID=9532: \REGISTRY\MACHINE\SOFTWARE\Mic\
rosoft\Windows\CurrentVersion\Diagnostics\DiagTrack\Aggregation\Instrumentation\
\CodecAppSvcAggregator\HbActiveMillis type: 11 size: 8 data:
4E 88 2B 05 00 00 00 00
11:14:13.991: Registry write PID=5076, TID=9532: \REGISTRY\MACHINE\SOFTWARE\Mic\
rosoft\Windows\CurrentVersion\Diagnostics\DiagTrack\Aggregation\Instrumentation\
\CodecAppSvcAggregator\HbErrorMillis type: 11 size: 8 data:
00 00 00 00 00 00 00 00
11:14:13.991: Registry write PID=5076, TID=9532: \REGISTRY\MACHINE\SOFTWARE\Mic\
rosoft\Windows\CurrentVersion\Diagnostics\DiagTrack\Aggregation\Instrumentation\
\CodecAppSvcAggregator\HbSeq type: 4 size: 4 data: 0x00000005 (5)
Err type: 1 size: 30 data: ProcTerminated
11:14:13.991: Registry write PID=5076, TID=9532: \REGISTRY\MACHINE\SOFTWARE\Mic\
rosoft\Windows\CurrentVersion\Diagnostics\DiagTrack\Aggregation\Instrumentation\
\UpdateHeartbeatScan\HbErr type: 4 size: 4 data: 0x00000000 (0)
11:14:36.838: Registry write PID=7148, TID=8648: \REGISTRY\MACHINE\SOFTWARE\Mic\
rosoft\Windows NT\CurrentVersion\Notifications\Data\418A073AA3BC1C75 type: 3 si\
ze: 464 data:
90 05 00 00 00 00 00 00 04 00 04 00 01 00 01 00
01 01 00 00 A5 AD CF 00 4F 00 02 00 00 00 01 91
40 01 02 99 66 00 03 03 DD 01 03 89 A8 01 0D 28
C7 01 0D D3 F9 00 0E BA CD 00 0F 16 8C 01 10 FF
88 01 1E C3 30 02 22 78 CE 00 24 AC C7 00 29 45
00 02 29 45 01 01 2F A8 FF 01 31 48 4F 00 36 1E
E1 01 3E 5B ED 01 46 48 B6 00 48 3B DB 01 4E 12

```



Enhance *SysMon* by adding I/O control codes to enable/disable certain notification types (processes, threads, image loads, Registry).

Performance Considerations

The Registry callback is invoked for every registry operation; there is no apriori way to request filtering of certain operations only. This means the callback needs to be as quick as possible since the caller is waiting. Also, there may be more than one driver in the chain of callbacks.

Some Registry operations, especially read operations happen in large quantities, so it's better for a driver to avoid processing read operations, if possible. If it must process read operations, it should at least limit its processing to certain keys of interest, such as anything under `HKLM\System\CurrentControlSet` (just an example). If processing can be done asynchronously, a work item could be used.

Write and create operations are used much less often, so in these cases the driver can do more if needed.

Miscellaenous Notes

- The documentation provides some warnings when dealing with Registry notifications, worth repeating here.

Certain Registry operations are lightly-documented because they are not very useful. Modifying the following operations should be avoided as it's difficult and error-prone: `NtRestoreKey`, `NtSaveKey`, `NtSaveKeyEx`, `NtLoadKeyEx`, `NtUnloadKey2`, `NtUnloadKeyEx`, `NtReplaceKey`, `NtRenameKey`, `NtSetInformationKey`

- The operations `RegNtPostCreateKeyEx` and `RegNtPostOpenKeyEx` provide a Registry key object (*Object* member in `REG_POST_OPERATION_INFORMATION`). This member is valid only if the *Status* member is `STATUS_SUCCESS`. Otherwise, its value is undefined.
- For some operations, the *Object* member points to a Registry key that is being destroyed (its internal reference count is zero). These are the operations:
 - `RegNtPreKeyHandleClose` (`REG_KEY_HANDLE_CLOSE_INFORMATION` structure)
 - `RegNtPostKeyHandleClose` (`REG_POST_OPERATION_INFORMATION` structure)
 - `RegNtCallbackObjectContextCleanup` (`REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION` structure)

The *Object* member should not be passed to general kernel routines (such as `ObReferenceObjectByPointer`). However, for the first two cases, the object can still be used within the callback by calling Configuration Manager functions (e.g. `CmCallbackGetKeyObjectIDEx`).



1. Implement a driver that protects a Registry key from modifications. A client can send the driver registry keys to protect or unprotect.
2. Implement a driver that redirects Registry write operations coming from selected processes (configured by a client application) to their own private key if they access `HKEY_LOCAL_MACHINE`. If the app is writing data, it goes to its private store. If it's reading data, first check the private store, and if no such value is found, go to the real Registry key.

Summary

In this chapter, we looked at two callback mechanisms supported by the kernel - obtaining handles to certain object types, and Registry access. In the next chapter, we'll look at more techniques that may be useful for a driver developer.

Chapter 11: Advanced Programming Techniques (Part 2)

In this chapter we'll continue to examine techniques of various degrees of usefulness to driver developers.

In this chapter:

- Timers
 - Generic Tables
 - Hash Tables
 - Singly Linked Lists
 - Callback Objects
-

Timers

We have briefly seen an example that uses a kernel timer in chapter 6. In this section, we'll cover kernel timers in more detail, as well as high-resolution timers, which have been introduced in Windows 8.1.

Kernel Timers

A kernel timer is represented by the `KTIMER` structure that must be allocated from non-paged memory. The timer can be set to one shot or periodic. The interval itself can be relative or absolute, making it quite flexible. A kernel timer is a dispatcher object, which means it can be waited upon with `KeWaitForSingleObject` and similar APIs. Once a `KTIMER` is allocated, it must be initialized by calling `KeInitializeTimer` or `KeInitializeTimerEx`:

```

VOID KeInitializeTimer (_Out_ PKTICKER Timer);

typedef enum _TIMER_TYPE {
    NotificationTimer,
    SynchronizationTimer
} TIMER_TYPE;

VOID KeInitializeTimerEx (
    _Out_ PKTICKER Timer,
    _In_ TIMER_TYPE Type);

```

There are two kinds of timers (similar to the two kinds of event kernel object types) - `NotificationTimer` that releases any number of waiting threads, and remains in the signaled state, or a `SynchronizationTimer`, that after releasing a single thread goes to the non-signaled state automatically. `KeInitializeTimer` is a shortcut that initializes a notification timer.

Once the timer is initialized, its interval can be set with `KeSetTimer` (one shot) or `KeSetTimerEx` (periodic):

```

BOOLEAN KeSetTimer (
    _Inout_ PKTICKER Timer,
    _In_ LARGE_INTEGER DueTime,
    _In_opt_ PKDPC Dpc);
BOOLEAN KeSetTimerEx (
    _Inout_ PKTICKER Timer,
    _In_ LARGE_INTEGER DueTime,
    _In_ LONG Period,
    _In_opt_ PKDPC Dpc);

```

Both functions set the timer interval based on a `LARGE_INTEGER` structure, that is set to a negative number for a relative count, and a positive number for an absolute count from January 1, 1601, at midnight GMT. The number (whether positive or negative) is specified as 100nsec units. For example, `1msec` equals $1000 \times 100\text{nsec}$ units. Here is how to specify a relative interval of 10 milliseconds:

```

LARGE_INTEGER interval;
interval.QuadPart = -10 * 10000;      // 10 msec

```

We have encountered these units before when discussing `KeDelayExecutionThread` in chapter 8.

The `Period` argument in `KeSetTimerEx` indicates the period the timer should count repeatedly from its first signaling. Curiously enough, it's specified in milliseconds. Finally, a DPC object can be

specified as an alternative to waiting. If one is provided, it will be inserted in a CPU's DPC queue and run just like any other DPC.

Both functions return TRUE if the timer is already in the system's timer queue. If it was there before the call, it's implicitly cancelled and set to the new specified time. With KeSetTimer, once the timer expires, it won't restart unless another call to KeSetTimer(Ex) is made. Regardless, a timer can be cancelled by calling KeCancelTimer:

```
BOOLEAN KeCancelTimer (_Inout_ PKTIMER);
```

KeCancelTimer returns TRUE if the timer was in the system's timer queue - which is always TRUE for a periodic timer.

Another available API to set a timer's interval is KeSetCoalescableTimer:

```
BOOLEAN KeSetCoalescableTimer (
    _Inout_ PKTIMER Timer,
    _In_ LARGE_INTEGER DueTime,
    _In_ ULONG Period,
    _In_ ULONG TolerableDelay,
    _In_opt_ PKDPC Dpc);
```

Most parameters are the same as KeSetTimerEx, except for the additional TolerableDelay. This parameter allows a caller to set some "tolerance" interval in milliseconds that indicates that it's ok to program the timer to expire slightly after the provided DueTime by no more than the tolerance delay. The period (if non-zero) can be up to the tolerance higher or lower. The point of a coalescable timer is to allow the system to save energy by not waking up too often to signal timers. Close-enough timers will be "coalesced" by the system, so that a single wakeup can signal multiple timers if their tolerance allows it.

Finally, you can query a timer's signaled state by calling KeReadStateTimer (may be useful for debugging purposes):

```
BOOLEAN KeReadStateTimer (_In_ PKTIMER Timer);
```

Timer Resolution

It may seem from the KeSetTimer(Ex) APIs that the timer's resolution can be really high, as the units are very small. For example, it seems you can set a timer to expire after 1 microsecond by specifying the value -10 for DueTime. This does not work as expected, however.

There is a default timer resolution, which is typically 15.625 milliseconds in today's systems. This is the default (and maximum) resolution, that is also used by the kernel's scheduler. This resolution can be changed, however. A quick way to determine the clock's resolution is to run the *Sysinternals ClockRes.exe* command line tool. Here is an example run:

```
C:\>clockres
```

```
Clockres v2.1 - Clock resolution display utility
Copyright (C) 2016 Mark Russinovich
Sysinternals

Maximum timer interval: 15.625 ms
Minimum timer interval: 0.500 ms
Current timer interval: 1.000 ms
```

The current timer interval is the active one, and is (more often than not) lower than the default. This is because user mode processes can change the clock's resolution to get better timing in wait operations, sleep calls, and timers. For example, the `timeBeginPeriod` or `timeSetEvent` user mode multimedia APIs allow setting up a timer with up to 1 millisecond resolution (both call the `NtSetTimerResolution` native API). This causes the clock's resolution to be reprogrammed to cater for the client process. The system keeps track of processes that request resolution changes, and so has to make sure the clock is using the highest resolution (lowest interval) requested by any process.

A kernel driver can specify its own request for a resolution value by calling `ExSetTimerResolution`:

```
ULONG ExSetTimerResolution (
    _In_ ULONG DesiredTime,
    _In_ BOOLEAN SetResolution);
```

The `DesiredTime` is in 100-nanosecond (nsec) units. If `SetResolution` is TRUE, the system adjusts the resolution to the closest value it can support, and returns the actual set value. If `SetResolution` is FALSE, the system decrements an internal counter (incremented for each `ExSetTimerResolution` call with TRUE), and if zero is reached, resets the resolution to its initial value. Of course, this will not occur as long as there are user mode processes that requested a higher resolution than the default.

With Windows 8 and later, you can also query the current resolution without making any changes with `ExQueryTimerResolution`:

```
void ExQueryTimerResolution (
    _Out_ PULONG MaximumTime,
    _Out_ PULONG MinimumTime,
    _Out_ PULONG CurrentTime);
```

The returned values are in 100-nsec units. Converted to milliseconds, these numbers are the same ones displayed by `ClockRes`.



The `KeQueryTimeIncrement` function returns the same value as the maximum timer resolution.



Write a C++ RAII wrapper for working with timers.

High-Resolution Timers

Starting with Windows 8.1, the kernel provides support for another type of timer - high-resolution timers, that can be used instead of the “standard” timers. These newer timers offer the following benefits over standard timers:

- There is no need to set the timer resolution explicitly - it will be set as required based on the provided interval (and revert automatically as well).
- High resolution timers never expire earlier than their set time.
- There is no need to set up an explicit DPC to be used as callback - the callback is specified directly as part of setting the timer. The system will invoke the callback at IRQL DISPATCH_LEVEL (2).

A high-resolution timer must be first allocated by calling `ExAllocateTimer`:

```
PEX_TIMER ExAllocateTimer (
    _In_opt_ PEXT_CALLBACK Callback,
    _In_opt_ PVOID CallbackContext,
    _In_ ULONG Attributes);
```

The callback provided must have the following prototype:

```
VOID EXT_CALLBACK (
    _In_ PEX_TIMER Timer,
    _In_opt_ PVOID Context);
```

The `CallbackContext` parameter to `ExAllocateTimer` is passed as-is to the callback function, along with the timer object itself. The attributes provided can be zero or the following:

- `EX_TIMER_HIGH_RESOLUTION` - specifies that the timer should be a high-resolution one. Without this flag, the timer is similar in terms of accuracy to a standard timer.
- `EX_TIMER_NO_WAKE` - indicates the timer should expire at its interval plus its tolerance delay (set with `ExSetTimer` discussed shortly). This flag conflicts with the previous one.
- `EX_TIMER_NOTIFICATION` - creates the timer as a notification timer as opposed to a synchronization timer (if this flag is not specified). The timer object can be waited upon just like standard timers.

`ExAllocateTimer` returns an opaque pointer to the allocated timer object that must be eventually freed with `ExDeleteTimer` (shown later).

The next step is to set the timer interval and start it by calling `ExSetTimer`:

```
BOOLEAN ExSetTimer (
    _In_ PEX_TIMER Timer,
    _In_ LONGLONG DueTime,
    _In_ LONGLONG Period,
    _In_opt_ PEXT_SET_PARAMETERS Parameters);
```

High-resolution timers only work with relative time, meaning `DueTime` must be a negative value (in the usual 100 nsec units). The optional `Period` parameter is the period for a periodic timer. It's specified in the same 100 nsec units (contrary to a standard timer where the period is specified in milliseconds). Finally, `Parameters` can be `NULL` or a pointer to `EXT_SET_PARAMETERS`:

```
typedef struct _EXT_SET_PARAMETERS_V0 {
    ULONG Version;
    ULONG Reserved;
    LONGLONG NoWakeTolerance;
} EXT_SET_PARAMETERS, *PEXT_SET_PARAMETERS;
```

The only parameter of interest is `NoWakeTolerance`, indicates the timer's maximum tolerance for waking a processor. If the value is set to `EX_TIMER_UNLIMITED_TOLERANCE`, the timer never wakes a processor in a low power state. Initializing this structure must be done with `ExInitializeSetTimerParameters` that sets the `Version` member to the correct value, `Reserved` and `NoWakeTolerance` to zero. Here is a typical way of working with `EXT_SET_PARAMETERS` if desired:

```
EXT_SET_PARAMETERS params;
ExInitializeSetTimerParameters(&params);
params.NoWakeTolerance = -5000;      // 0.5 msec
ExSetTimer(timer, -15000, 0, &params); // 1.5 msec interval
```

`ExSetTimer` cancels any previous timer that may have been active and sets the new values. If the timer was active, the function returns `TRUE`. Otherwise, it returns `FALSE`.

As with standard timers, it's possible to cancel a high-resolution timer with `ExCancelTimer`:

```
BOOLEAN ExCancelTimer (
    _Inout_ PEX_TIMER Timer,
    _In_opt_ PEXT_CANCEL_PARAMETERS Parameters);
```

The function returns `TRUE` if the timer was actually cancelled, or `FALSE` if the timer was inactive - nothing to cancel. `Parameters` must be `NULL`.

Finally, a timer object must be deleted with `ExDeleteTimer`:

```
BOOLEAN ExDeleteTimer (
    _In_     PEX_TIMER Timer,
    _In_     BOOLEAN Cancel,
    _In_     BOOLEAN Wait,
    _In_opt_ PEXT_DELETE_PARAMETERS Parameters);
```

`Cancel` indicates whether to cancel the timer (if active). If `Cancel` is set to TRUE, then `Wait` can be set to TRUE as well to wait until the timer has been cancelled. If `Wait` is set to TRUE, so must `Cancel`. Similar to `ExSetTimer`, an optional `EXT_DELETE_PARAMETERS` structure can be provided, that includes an optional callback to be invoked when the timer is finally deleted. `ExDeleteTimer` returns TRUE if `Cancel` is TRUE and the timer was cancelled.



Write a C++ RAI^II wrapper for High-Resolution timers.

You can find examples for using standard and high-resolution timers in the *Timers* project, part of the source code for this chapter. The example driver has a few I/O control codes to set up a standard timer and a high-resolution timer. Here is an excerpt for creating a high-resolution timer:

```
// in TimersPublic.h

struct PeriodicTimer {
    ULONG Interval;
    ULONG Period;
};

// in DriverEntry
// g_HiRes is PEX_TIMER

g_HiRes = ExAllocateTimer(HiResCallback, nullptr,
    EX_TIMER_HIGH_RESOLUTION);

//...
case IOCTL_TIMERS_SET_HIRES:
    //check buffer... and then
    auto data = (PeriodicTimer*)Irp->AssociatedIrp.SystemBuffer;
    ExSetTimer(g_HiRes, -10000LL * data->Interval,
        10000LL * data->Period, nullptr);
    status = STATUS_SUCCESS;
    break;
//...
```

```

void HiResCallback(PEX_TIMER, PVOID) {
    auto counter = KeQueryPerformanceCounter(nullptr);
    DbgPrint(DRIVER_PREFIX "Hi-Res Timer DPC: IRQL=%d Counter=%lld\n",
        (int)KeGetCurrentIrql(), counter.QuadPart);
}

```

The *TimersTest* user-mode application can be used to test the timers. Here is the entire code:

```

#include <Windows.h>
#include <stdio.h>
#include "..\Timers\TimersPublic.h"

int main(int argc, const char* argv[]) {
    if (argc < 2) {
        printf("Usage: TimersTest [query | stop | set [hires] "
            "[interval(ms)] [period(ms)]]\n");
    }

    HANDLE hDevice = CreateFile(L"\\\\.\\Timers", GENERIC_READ | GENERIC_WRITE,
        0, nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE) {
        printf("Error opening device (%u)\n", GetLastError());
        return 1;
    }

    DWORD bytes;
    if (argc < 2 || _strcmp(argv[1], "query") == 0) {
        TimerResolution res;
        if (DeviceIoControl(hDevice, IOCTL_TIMERS_GET_RESOLUTION, nullptr,
            0, &res, sizeof(res), &bytes, nullptr)) {
            printf("Timer resolution (100nsec): Max: %u Min: %u "
                "Current: %u Inc: %u\n",
                res.Maximum, res.Minimum, res.Current, res.Increment);
            float factor = 10000.0f;
            printf("Timer resolution (msec):      Max: %.3f Min: %.3f "
                "Current: %.3f Inc: %.3f\n",
                res.Maximum / factor, res.Minimum / factor,
                res.Current / factor, res.Increment / factor);
        }
    }
    else if (_strcmp(argv[1], "set") == 0 && argc > 2) {
        int arg = 2;
        bool hires = false;
    }
}

```

```

    if (_stricmp(argv[2], "hires") == 0) {
        hires = true;
        arg++;
    }
    PeriodicTimer data{};
    if (argc > arg) {
        data.Interval = atoi(argv[arg]);
        arg++;
        if (argc > arg) {
            data.Period = atoi(argv[arg]);
        }
        if (!DeviceIoControl(hDevice,
            hires ? IOCTL_TIMERS_SET_HIRES : IOCTL_TIMERS_SET_PERIODIC,
            &data, sizeof(data), nullptr, 0, &bytes, nullptr))
            printf("Error setting timer (%u)\n", GetLastError());
    }
}
else if (_stricmp(argv[1], "stop") == 0) {
    DeviceIoControl(hDevice, IOCTL_TIMERS_STOP,
        nullptr, 0, nullptr, 0, &bytes, nullptr);
}
else {
    printf("Unknown option.\n");
}
CloseHandle(hDevice);

return 0;
}

```

I/O Timer

There is yet another type of timer that can be used by a driver, known as an *I/O Timer*. This timer exists for every device object (just one per device). When started, it runs a callback at IRQL_DISPATCH_LEVEL every second. There is no way to further customize it. It can be used as a “watchdog” of some sort, when high resolution is not required.

The first step in using an I/O timer is to initialize it:

```

NTSTATUS IoInitializeTimer(
    _In_      PDEVICE_OBJECT DeviceObject,
    _In_      PIO_TIMER_ROUTINE TimerRoutine,
    _In_opt_  PVOID Context);

```

Notice the device object parameter - this is how the I/O timer is identified. TimerRoutine has the following prototype:

```
VOID IO_TIMER_ROUTINE (
    _In_ struct _DEVICE_OBJECT *DeviceObject,
    _In_opt_ PVOID Context);
```

To start the timer, call `IoStartTimer`. To stop it, call `IoStopTimer`:

```
VOID IoStartTimer(_In_ PDEVICE_OBJECT DeviceObject);
VOID IoStopTimer(_In_ PDEVICE_OBJECT DeviceObject);
```

Generic Tables

The term “generic tables” is used by the kernel API to refer to two binary tree implementations available to device driver writers (and the kernel itself). The first type is a *Splay Tree* implementation, referred to as simply Generic Tables. The second implementation is using *AVL trees*, referred to as AVL tables.

Splay trees are binary search trees where frequently used items move closer to the root and thus are faster to access. On the downside, the tree is not self-balancing in the sense that it can have any depth. *AVL trees* (named after *Georgy Adelson-Velsky* and *Evgenii Landis*) are self-balancing binary search trees, keeping their depth logarithmic on the number of items (in base 2). They are similar to *red-black* trees, but are faster in retrieval. You can find more information online.

Both implementations have an almost identical API. We’ll start with Splay trees, and then look at the differences compared to AVL trees.

Splay Trees

The most common functions related to generic tables are shown in table 11-1.

Table 11-1: Common functions for working with generic tables

Function	Description
RtlInitializeGenericTable	Initialize a new generic table
RtlInsertElementGenericTable	Insert a new item into the table
RtlLookupElementGenericTable	Lookup an item by key (logarithmic)
RtlNumberGenericTableElements	Return the number of items in the table
RtlGetElementGenericTable	Return an item by index
RtlDeleteElementGenericTable	Delete an item from the table
RtlEnumerateGenericTable	Enumerate the items in the table

It's important to note that the tables API provide no inherent synchronization. It's the job of the driver to make sure thread/CPU safety exists. You can use any appropriate synchronization primitive we looked at, such as a (fast) mutex, Executive Resource, or spin lock.

The first step when using a generic table is to initialize it by calling `RtlInitializeGenericTable`:

```
VOID RtlInitializeGenericTable (
    _Out_ PRTL_GENERIC_TABLE Table,
    _In_ PRTL_GENERIC_COMPARE_ROUTINE CompareRoutine,
    _In_ PRTL_GENERIC_ALLOCATE_ROUTINE AllocateRoutine,
    _In_ PRTL_GENERIC_FREE_ROUTINE FreeRoutine,
    _In_opt_ PVOID TableContext);
```

A generic table is managed by an `RTL_GENERIC_TABLE` structure, that although provided in the headers, should be treated as opaque. A driver allocates such a structure and calls the initialization API. The function requires three callbacks to be specified (all of which are mandatory).

`CompareRoutine` is a function that should tell which element is greater (or equal) given two elements. This is the basis of any binary search tree implementation. The routine must have the following prototype:

```
typedef enum _RTL_GENERIC_COMPARE_RESULTS {
    GenericLessThan,
    GenericGreaterThan,
    GenericEqual
} RTL_GENERIC_COMPARE_RESULTS;

RTL_GENERIC_COMPARE_RESULTS CompareFunction (
    _In_ struct _RTL_GENERIC_TABLE *Table,
    _In_ PVOID FirstStruct,
    _In_ PVOID SecondStruct);
```

The returned value is a simple enumeration. The provided arguments should be cast to the actual data stored in the table and compared using some key present in that data. The returned value must be consistent - using the key for comparison in a consistent way - otherwise the table APIs cannot work as expected.

The `AllocateRoutine` and `FreeRoutine` are needed to implement the method of allocating and freeing memory for the nodes managed by the table. These include the data item itself the driver wishes to store and any other metadata required by the table implementation. Here are the prototypes:

```
PVOID AllocateFunction (
    _In_ struct _RTL_GENERIC_TABLE *Table,
    _In_ CLONG ByteSize);
VOID FreeFunction (
    _In_ struct _RTL_GENERIC_TABLE *Table,
    _In_ PVOID Buffer);
```

The byte size provided to the allocation function is properly calculated to include any metadata required by the tables API. As we'll soon see, the insert API specifies the driver's data size and automatically adds the required overhead before calling the allocation function.

As for the implementation itself - you can use any memory APIs discussed, such as ExAllocatePoolWithTag, ExAllocatePool2, or even lookaside lists. You can use the paged pool or non-paged pool, as needed. The deallocation function must free the allocation appropriately.

Finally, the TableContext parameter allows adding some context pointer that may be useful for the driver. It can be retrieved by accessing the TableContext member of RTL_GENERIC_TABLE. It's also possible to allocate a structure that starts with a RTL_GENERIC_TABLE member, and add driver-specific members, so that access is possible by casting to the larger structure.



Although the RTL_GENERIC_TABLE is supposed to be opaque, there is no other way to get to the table context except accessing the TableContext member directly.

Once the table is initialized, items can be inserted (based on a key) by calling Rt1InsertElementGenericTable:

```
PVOID Rt1InsertElementGenericTable (
    _In_ PRTL_GENERIC_TABLE Table,
    _In_reads_bytes_(BufferSize) PVOID Buffer,
    _In_ CLONG BufferSize,
    _Out_opt_ PBOOLEAN NewElement);
```

The provided Buffer should be the data to be placed in the table, which should contain the key to be used for comparison. The function calls the compare function to figure out if the element already exists in the table. If it does, its address is returned and no insertion takes place. If it doesn't exist, it's inserted by copying the provided buffer to the "real" buffer allocated (by calling the registered allocation routine). BufferSize should specify the number of bytes in the data structure to copy. The returned pointer in this case is the address of the stored object within the table.

For example, suppose the driver wants to keep some data on a per-process basis, keyed by the process ID. The data structure could look something like the following (full example is shown in the next section):

```
struct ProcessData {
    ULONG Id;           // serves as the key
    // data to be tracked per process...
};
```

Inserting an item would be done with the following code:

```
void AddProcessData(ULONG pid) {
    ProcessData data;
    data.Id = pid;
    // fill more members...

    PVOID item = RtlInsertElementGenericTable(&g_table,
                                              &data, sizeof(data), nullptr);
}
```

There is no need to store the returned pointer - the driver can get it later by performing a lookup. Notice that the provided data is on the stack - it doesn't matter, as it's copied to the dynamically-allocated buffer anyway.

The final optional parameter to `RtlInsertElementGenericTable` (`NewElement`) returns if a new item was inserted (TRUE) or the item was already in the table (FALSE).

Retrieving an item based on the key is accomplished with `RtlLookupElementGenericTable`:

```
PVOID RtlLookupElementGenericTable (
    _In_ PRTL_GENERIC_TABLE Table,
    _In_ PVOID Buffer);
```

The provided `Buffer` should be the key data that will be used by the called compare routine. It doesn't have to include a full blown item if the key members are first in the data structure. In the previous example, providing a simple `ULONG` is enough, as it's the first member of `ProcessData`. `RtlLookupElementGenericTable` returns the pointer to the data within the table, or `NULL` if the item cannot be located.

The table API provides an additional way to retrieve items - by index:

```
PVOID RtlGetElementGenericTable(
    _In_ PRTL_GENERIC_TABLE Table,
    _In_ ULONG Index);
```

This is sometimes useful for enumeration purposes, although the order is not generally predictable. You can get the number of items in the table with the simple `RtlNumberGenericTableElements`. To get a predictable enumeration (ordered by key), you can call `RtlEnumerateGenericTable`:

```
PVOID RtlEnumerateGenericTable (
    _In_ PRTL_GENERIC_TABLE Table,
    _In_ BOOLEAN Restart);
```

Set `Restart` to `TRUE` when initializing enumeration, and iterate until the returned pointer is `NULL`. Here is an example:

```
for (PVOID ptr = RtlEnumerateGenericTable(Table, TRUE);
     ptr;
     ptr = RtlEnumerateGenericTable(Table, FALSE)) {
    // process ptr
}
```

`RtlEnumerateGenericTable` flattens the tree into a linked list and provides the items as required. A similar API, `RtlEnumerateGenericTableWithoutSplaying` will not perturb the splay links.

Finally, to delete an item from the table, call `RtlDeleteElementGenericTable`:

```
BOOLEAN RtlDeleteElementGenericTable (
    _In_ PRTL_GENERIC_TABLE Table,
    _In_ PVOID Buffer);
```

The function returns `TRUE` if the item was found and was deleted, `FALSE` otherwise. You must be careful to delete all items from the table before the driver unloads, or the memory used by remaining items will leak. You can use the following loop to delete all items properly:

```
PVOID element;
while ((element = RtlGetElementGenericTable(&table, 0)) != nullptr) {
    RtlDeleteElementGenericTable(&table, element);
}
```



Write a RAII wrapper for generic tables. Use C++ templates if you can.

Tables Sample Driver

The *Tables* driver example shows a usage for the common generic table APIs. The driver tracks Registry access and counts certain Registry operations on a per-process basis.

The header file *TablesPublic.h* contains definitions for control codes and the data structure tracked per process (which is also returned to user mode upon request):

```
#define TABLES_DEVICE 0x8003

#define IOCTL_TABLES_GET_PROCESS_COUNT      \
    CTL_CODE(TABLES_DEVICE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_TABLES_GET_PROCESS_BY_ID      \
    CTL_CODE(TABLES_DEVICE, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_TABLES_GET_PROCESS_BY_INDEX   \
    CTL_CODE(TABLES_DEVICE, 0x802, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_TABLES_DELETE_ALL            \
    CTL_CODE(TABLES_DEVICE, 0x803, METHOD_NEITHER, FILE_ANY_ACCESS)
#define IOCTL_TABLES_START                \
    CTL_CODE(TABLES_DEVICE, 0x804, METHOD_NEITHER, FILE_ANY_ACCESS)
#define IOCTL_TABLES_STOP                 \
    CTL_CODE(TABLES_DEVICE, 0x805, METHOD_NEITHER, FILE_ANY_ACCESS)
#define IOCTL_TABLES_GET_ALL              \
    CTL_CODE(TABLES_DEVICE, 0x806, METHOD_OUT_DIRECT, FILE_ANY_ACCESS)

struct ProcessData {
    ULONG Id;
    LONG64 RegistrySetValueOperations;
    LONG64 RegistryCreateKeyOperations;
    LONG64 RegistryRenameOperations;
    LONG64 RegistryDeleteOperations;
};

};
```

Every time a process makes one of these operation, the relevant counter is incremented. A generic table is used to quickly lookup a process making a Registry operation based on the process' ID.

The process generic table and other data is stored in the following structure (in *Tables.h*):

```
struct Globals {
    void Init();

    RTL_GENERIC_TABLE ProcessTable;
    FastMutex Lock;
    LARGE_INTEGER RegCookie;
};
```

A global instance is created in *Tables.cpp*. *Init* is used to initialize the fast mutex (a RAII wrapper similar to the one we saw in chapter 6) and the table itself:

```
#define DRIVER_PREFIX "Tables: "
#define DRIVER_TAG 'lbaT'

Globals g_Globals;

void Globals::Init() {
    Lock.Init();
    RtlInitializeGenericTable(&ProcessTable,
        CompareProcesses, AllocateProcess, FreeProcess, nullptr);
}

extern Globals g_Globals;
```

CompareProcesses uses the process ID for comparison:

```
RTL_GENERIC_COMPARE_RESULTS
CompareProcesses(PRTL_GENERIC_TABLE, PVOID first, PVOID second) {
    auto p1 = (ProcessData*)first;
    auto p2 = (ProcessData*)second;

    if (p1->Id == p2->Id)
        return GenericEqual;

    return p1->Id > p2->Id ? GenericGreaterThan : GenericLessThan;
}
```

Allocation and deallocation are performed in a straightforward manner with ExAllocatePool2 and ExFreePool:

```
PVOID AllocateProcess(PRTL_GENERIC_TABLE, CLONG bytes) {
    return ExAllocatePool2(POOL_FLAG_PAGED | POOL_FLAG_UNINITIALIZED,
        bytes, DRIVER_TAG);
}

void FreeProcess(PRTL_GENERIC_TABLE, PVOID buffer) {
    ExFreePool(buffer);
}
```

POOL_FLAG_UNINITIALIZED is used to skip zeroing out the structure, as the table API will copy the provided data anyway.

DriverEntry is fairly standard, with two additions. One is a Registry notification callback for tracking Registry operations. The other is a process notification callback, so that when a process exits, the stats

kept for the process are removed from the generic table. This is partly because process IDs may be reused and that would track multiple processes that happen to have the same ID with the same data structure.



If you would want to track all processes without losing stats, it's possible to use a combination of the process ID and its creation time as a unique key. Another option for a unique key is a process key available with `PsGetProcessStartKey` (from Windows 10 version 1703). Another idea would be to push dead processes to a separate list.

Here is the complete `DriverEntry`:

```
extern "C"
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    NTSTATUS status;
    PDEVICE_OBJECT devObj = nullptr;
    UNICODE_STRING link = RTL_CONSTANT_STRING(L"\?\?\Tables");
    bool symLinkCreated = false, procRegistered = false;

    do {
        UNICODE_STRING name = RTL_CONSTANT_STRING(L"\Device\Tables");
        status = IoCreateDevice(DriverObject, 0, &name, FILE_DEVICE_UNKNOWN,
                               0, FALSE, &devObj);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX
                     "Failed in IoCreateDevice (0x%X)\n", status));
            break;
        }

        status = IoCreateSymbolicLink(&link, &name);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX
                     "Failed in IoCreateSymbolicLink (0x%X)\n", status));
            break;
        }
        symLinkCreated = true;
        g_Globals.Init();

        //
        // set process notification routine
        //
        status = PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, FALSE);
        if (!NT_SUCCESS(status))
            break;
    }
```

```

procRegistered = true;

//
// Registry notifications
//
UNICODE_STRING altitude = RTL_CONSTANT_STRING(L"123456.789");
status = CmRegisterCallbackEx(OnRegistryNotify,
    &altitude, DriverObject, nullptr,
    &g_Globals.RegCookie, nullptr);
} while (false);

if (!NT_SUCCESS(status)) {
    if (procRegistered)
        PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);
    if (!symLinkCreated)
        IoDeleteSymbolicLink(&link);
    if (devObj)
        IoDeleteDevice(devObj);
    return status;
}

DriverObject->DriverUnload = TablesUnload;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = TablesCreateClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = TablesDeviceControl;

return status;
}

```

The Registry notification callback first tests for the interesting operations:

```

NTSTATUS OnRegistryNotify(PVOID, PVOID Argument1, PVOID Argument2) {
    UNREFERENCED_PARAMETER(Argument2);

    auto type = (REG_NOTIFY_CLASS)(ULONG_PTR)Argument1;
    switch (type) {
        case RegNtPostSetValueKey:
        case RegNtPostCreateKey:
        case RegNtPostCreateKeyEx:
        case RegNtPostRenameKey:
        case RegNtPostDeleteValueKey:
        case RegNtPostDeleteKey:

```

At this point it's time to look for the current process in the generic table. If it's not there, then an entry needs to be created:

```
PVOID buffer;
auto pid = HandleToULong(PsGetCurrentProcessId());
{
    Locker locker(g_Globals.Lock);
    buffer = RtLookupElementGenericTable(&g_Globals.ProcessTable, &pid);
    if (buffer == nullptr) {
        //
        // process does not exist, create a new entry
        //
        ProcessData data{};
        data.Id = pid;
        buffer = RtInsertElementGenericTable(&g_Globals.ProcessTable,
            &data, sizeof(data), nullptr);
        if (buffer) {
            KdPrint((DRIVER_PREFIX
                "Added process %u from Registry callback\n", pid));
        }
    }
}
```

The `Locker` class is the same one we used in chapter 6 - acquiring the lock (fast mutex in this case) in the constructor and releasing in the destructor. Once the fast mutex is acquired, `RtLookupElementGenericTable` is called to look for the process ID. If not found (NULL returned), `RtInsertElementGenericTable` is called to insert a new item. Technically, it's possible to just call `RtInsertElementGenericTable` without doing a lookup first, as it would return the existing pointer if the item to insert already exists. Note that `data` is zeroed out before the ID is set, so that copying the data to the table would start all counters at zero.



The artificial scope is there to minimize the locking scope.

The next step is to increment the relevant counter:

```

if (buffer) {
    auto data = (ProcessData*)buffer;
    switch (type) {
        case RegNtPostSetValueKey:
            InterlockedIncrement64(&data->RegistrySetValueOperations);
            break;
        case RegNtPostCreateKey:
        case RegNtPostCreateKeyEx:
            InterlockedIncrement64(&data->RegistryCreateKeyOperations);
            break;
        case RegNtPostRenameKey:
            InterlockedIncrement64(&data->RegistryRenameOperations);
            break;
        case RegNtPostDeleteKey:
        case RegNtPostDeleteValueKey:
            InterlockedIncrement64(&data->RegistryDeleteOperations);
            break;
    }
}
}

```

The process notify callback should remove a dead process data structure:

```

void OnProcessNotify(PEPROCESS, HANDLE pid, PPS_CREATE_NOTIFY_INFO createInfo) {
    if (!createInfo) {
        //
        // process dead, remove from table
        //
        Locker locker(g_Globals.Lock);
        ProcessData data;
        data.Id = HandleToULong(pid);
        auto deleted = RtlDeleteElementGenericTable(
            &g_Globals.ProcessTable, &data);
        if (!deleted) {
            KdPrint((DRIVER_PREFIX
                "Failed to delete process with ID %u\n", data.Id));
        }
    }
}

```

Deleting could fail if the driver started after the process in question was already running. Note that there is no need to create a new item if a process is created - if the process does not perform the tracked Registry operations no item should be added as an optimization.

The IRP_MJ_DEVICE_CONTROL handler handles all client requests. It starts with the “usual” code:

```
NTSTATUS TablesDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    auto len = 0U;

    switch (dic.IoControlCode) {
```

After the switch, the IRP is completed with the status and len:

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

The CompleteRequest helper function is the same as used in chapter 8 (and others), completing the IRP with whatever status and information provided.

Here is the case for getting the number of elements (processes) being tracked:

```
case IOCTL_TABLES_GET_PROCESS_COUNT:
{
    if (dic.OutputBufferLength < sizeof(ULONG)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    Locker locker(g_Globals.Lock);
    *(ULONG*)Irp->AssociatedIrp.SystemBuffer =
        Rt1NumberGenericTableElements(&g_Globals.ProcessTable);
    len = sizeof(ULONG);
    status = STATUS_SUCCESS;
}
break;
```



The NULL check for the system buffer is missing in the above snippet.

Getting a process' data by ID requires lookup:

```

case IOCTL_TABLES_GET_PROCESS_BY_ID:
{
    if (dic.OutputBufferLength < sizeof(ProcessData) ||  

        dic.InputBufferLength < sizeof(ULONG)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    ULONG pid = *(ULONG*)Irp->AssociatedIrp.SystemBuffer;
    Locker locker(g_Globals.Lock);
    auto data = (ProcessData*)RtlLookupElementGenericTable(
        &g_Globals.ProcessTable, &pid);
    if (data == nullptr) {
        //  

        // invalid or non-tracked PID  

        //  

        status = STATUS_INVALID_CID;
        break;
    }
    memcpy(Irp->AssociatedIrp.SystemBuffer, data, len = sizeof(ProcessData));
    status = STATUS_SUCCESS;
}
break;

```

Getting all process information is a bit tricky, as we need to make sure not to overflow the user's buffer:

```

case IOCTL_TABLES_GET_ALL:
{
    if (dic.OutputBufferLength < sizeof(ProcessData)) {
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    Locker locker(g_Globals.Lock);
    auto count = RtlNumberGenericTableElements(&g_Globals.ProcessTable);
    if (count == 0) {
        status = STATUS_NO_DATA_DETECTED;
        break;
    }
    NT_ASSERT(Irp->MdlAddress);
    count = min(count, dic.OutputBufferLength / sizeof(ProcessData));
    auto buffer = (ProcessData*)MmGetSystemAddressForMdlSafe(
        Irp->MdlAddress, NormalPagePriority);
    if (buffer == nullptr) {

```

```

        status = STATUS_INSUFFICIENT_RESOURCES;
        break;
    }
    for (ULONG i = 0; i < count; i++) {
        auto data = (ProcessData*)RtlGetElementGenericTable(
            &g_Globals.ProcessTable, i);
        NT_ASSERT(data);
        memcpy(buffer, data, sizeof(ProcessData));
        buffer++;
    }
    len = count * sizeof(ProcessData);
    status = STATUS_SUCCESS;
}
break;
}

```

Here is where `RtlGetElementGenericTable` comes in handy. The code fills the user's buffer with as many `ProcessData` structures that would fit or all that exist if everything fits.

To delete all items (`IOCTL_TABLES_DELETE_ALL`), which is also needed in the Unload routine, `DeleteAllProcesses` is called:

```

void DeleteAllProcesses() {
    Locker locker(g_Globals.Lock);
    //
    // deallocate all objects still stored in the table
    //
    PVOID p;
    auto t = &g_Globals.ProcessTable;
    while ((p = RtlGetElementGenericTable(t, 0)) != nullptr) {
        RtlDeleteElementGenericTable(t, p);
    }
}

```

Finally, the Unload routine cleans everything up:

```

void TablesUnload(PDRIVER_OBJECT DriverObject) {
    CmUnRegisterCallback(g_Globals.RegCookie);
    PsSetCreateProcessNotifyRoutineEx(OnProcessNotify, TRUE);
    DeleteAllProcesses();
    UNICODE_STRING link = RTL_CONSTANT_STRING(L"\?\?\Tables");
    IoDeleteSymbolicLink(&link);
    IoDeleteDevice(DriverObject->DeviceObject);
}

```

See the full source code in the *Tables* project.

Testing the Tables Driver

The client application *TablesTest* uses command line arguments to work with the driver. Here is the complete `main` function:

```
int main(int argc, const char* argv[]) {
    enum class Command {
        GetProcessCount,
        DeleteAll,
        GetProcessById,
        GetProcessByIndex,
        GetAllProcesses,
        Start,
        Stop,
        Error = 99,
    };

    auto cmd = Command::GetProcessCount;
    int pid = 0;

    if (argc > 1) {
        if (_stricmp(argv[1], "help") == 0)
            return PrintUsage();
        if (_stricmp(argv[1], "delete") == 0)
            cmd = Command::DeleteAll;
        else if (_stricmp(argv[1], "count") == 0)
            cmd = Command::GetProcessCount;
        else if (_stricmp(argv[1], "start") == 0)
            cmd = Command::Start;
        else if (_stricmp(argv[1], "getall") == 0)
            cmd = Command::GetAllProcesses;
        else if (_stricmp(argv[1], "stop") == 0)
            cmd = Command::Stop;
        else if (_stricmp(argv[1], "get") == 0) {
            if (argc > 2)
                pid = atoi(argv[2]);
            cmd = Command::GetProcessById;
        }
    }
}
```

```
    }
    else {
        printf("Missing process ID\n");
        return 1;
    }
}
else if (_stricmp(argv[1], "geti") == 0) {
    if (argc > 2) {
        pid = atoi(argv[2]);
        cmd = Command::GetProcessByIndex;
    }
    else {
        printf("Missing index\n");
        return 1;
    }
}
else
    cmd = Command::Error;
}
if (cmd == Command::Error) {
    printf("Command error.\n");
    return PrintUsage();
}
auto hDevice = CreateFile(L"\\\\".\\Tables",
    GENERIC_READ | GENERIC_WRITE, 0, nullptr,
    OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE) {
    printf("Error opening device (%u)\n", GetLastError());
    return 1;
}

DWORD bytes;
BOOL success = FALSE;
switch (cmd) {
    case Command::GetProcessCount:
    {
        DWORD count;
        success = DeviceIoControl(hDevice,
            IOCTL_TABLES_GET_PROCESS_COUNT, nullptr, 0,
            &count, sizeof(count), &bytes, nullptr);
        if (success) {
            printf("Process count: %u\n", count);
        }
    }
}
```

```
        break;
    }

case Command::GetAllProcesses:
{
    DWORD count = 0;
    success = DeviceIoControl(hDevice,
        IOCTL_TABLES_GET_PROCESS_COUNT, nullptr, 0,
        &count, sizeof(count), &bytes, nullptr);
    if (count) {
        count += 10;      // in case more processes created
        auto data = std::make_unique<ProcessData[]>(count);
        success = DeviceIoControl(hDevice,
            IOCTL_TABLES_GET_ALL, nullptr, 0,
            data.get(), count * sizeof(ProcessData), &bytes, nullptr);
        if (success) {
            count = bytes / sizeof(ProcessData);
            printf("Returned %u processes\n", count);
            for (DWORD i = 0; i < count; i++)
                DisplayProcessData(data[i]);
        }
    }
    break;
}

case Command::DeleteAll:
{
    success = DeviceIoControl(hDevice, IOCTL_TABLES_DELETE_ALL,
        nullptr, 0, nullptr, 0, &bytes, nullptr);
    if (success)
        printf("Deleted successfully.\n");
    break;
}

case Command::GetProcessById:
case Command::GetProcessByIndex:
{
    ProcessData data;
    success = DeviceIoControl(hDevice,
        cmd == Command::GetProcessById ?
            IOCTL_TABLES_GET_PROCESS_BY_ID :
            IOCTL_TABLES_GET_PROCESS_BY_INDEX,
        &pid, sizeof(pid), &data, sizeof(data), &bytes, nullptr);
    if (success)
        DisplayProcessData(data);
```

```

        }
        break;
    }
}

if (!success) {
    printf("Error (%u)\n", GetLastError());
}
CloseHandle(hDevice);
return 0;
}

```

DisplayProcessData shows the counters:

```

void DisplayProcessData(ProcessData const& data) {
    printf("PID: %u\n", data.Id);
    printf("Registry set Value: %lld\n", data.RegistrySetValueOperations);
    printf("Registry delete: %lld\n", data.RegistryDeleteOperations);
    printf("Registry create key: %lld\n", data.RegistryCreateKeyOperations);
    printf("Registry rename: %lld\n", data.RegistryRenameOperations);
}

```



1. Add support for system-wide statistics for the implemented operations. Add control codes to retrieve them from user mode.
2. Save deleted processes stats in a list (so they don't get lost once a process is terminated), and provide this list to the client if requested.
3. Implement the start and stop control codes to allow pausing and resuming counting operations.

AVL Trees

The API for using AVL trees is virtually identical to the splay trees API with the addition of the suffix “Avl” to function names, such as `RtlInitializeGenericTableAvl`. In the AVL tree case, a different structure, `RTL_AVL_TABLE`, is used to manage the tree.

You may want to experience with both implementations and decide based on performance measurements for your scenario that one implementation is better than the other. Fortunately, the kernel headers provide a simple way to switch to AVL trees without changing any code by defining the macro `RTL_USE_AVL_TABLES` before including `<ntddk.h>`:

```
#define RTL_USE_AVL_TABLES  
  
#include <ntddk.h>
```

That's it! All calls to the Splay trees functions are redirected (the functions become macros) to the AVL tree implementation.



Try it out with the *Tables* driver.

Hash Tables

The Splay trees and AVL trees discussed are implemented as binary search trees. Another common way to perform quick lookup is by using hash tables. Hash tables are based around a *hash function* that, if properly implemented, provides a good distribution of values across keys - no greater/less than comparison required.

The WDK documentation does not document any hash functions, but the kernel API supports a hash table implementation. The functions are declared in `<ntddk.h>`, but are undocumented. As such, they are not described in this book. Feel free to investigate their usage, starting with the function `RtlInitHashTableContext`.

Singly Linked Lists

We have seen numerous times the use of doubly-linked lists, based on the `LIST_ENTRY` structure. The kernel API also supports singly-linked lists, where the full functionality of a doubly-linked list is not required. The structure to use is `SINGLE_LIST_ENTRY` defined like so:

```
typedef struct _SINGLE_LIST_ENTRY {  
    struct _SINGLE_LIST_ENTRY *Next;  
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;
```

This is as simple as a linked list can possibly get. Just as with doubly-linked lists, one of these is defined as the header of the list (`Next` is initialized to `NULL`), and the same structure is embedded in a larger structure where the real data is. For example:

```
struct MyData {
    ULONGLONG Time;
    ULONG ProcessId;
    SINGLE_LIST_ENTRY Link;
    ULONG ExitCode;
};
```

Since it's a singly-linked list, you can only add a new head and remove the current head (both implemented inline within *ntdef.h*):

```
VOID PushEntryList(
    _Inout_ PSINGLE_LIST_ENTRY ListHead,
    _Inout_ __drv_aliasesMem PSINGLE_LIST_ENTRY Entry);

PSINGLE_LIST_ENTRY PopEntryList(_Inout_ PSINGLE_LIST_ENTRY ListHead);
```

Just like doubly-linked lists, the `CONTAINING_RECORD` macro can be used to get to the “real” data given the pointer to `SINGLE_LIST_ENTRY`, the full structure type, and the name of the `SINGLE_LIST_ENTRY` member within the larger structure.

The affermented functions are not thread/CPU safe, so must be properly protected if appropriate. That said, APIs are provided for thread/CPU safe pushing and popping using a spin lock only:

```
PSINGLE_LIST_ENTRY ExInterlockedPopEntryList (
    _Inout_ PSINGLE_LIST_ENTRY ListHead,
    _Inout_ __Requires_lock_not_held_(*_Curr_) PKSPIN_LOCK Lock);

PSINGLE_LIST_ENTRY ExInterlockedPushEntryList (
    _Inout_ PSINGLE_LIST_ENTRY ListHead,
    _Inout_ __drv_aliasesMem PSINGLE_LIST_ENTRY ListEntry,
    _Inout_ __Requires_lock_not_held_(*_Curr_) PKSPIN_LOCK Lock);
```

The spin lock is acquired at `IRQL HIGH_LEVEL`, which makes it easy to use from any IRQL.

Sequenced Singly-Linked Lists

There is yet another implementation of atomic singly linked lists provided by the kernel. These use *Lock Free* techniques, which are more efficient than using a spin lock.

The basis of these lists is a header described by a `SLIST_HEADER`, which should be treated as opaque. The driver initializes the header with `InitializeSListHeader` (or `ExInitializeSListHeader` which is the same thing):

```
VOID InitializeSListHead (_Out_ PSLIST_HEADER SListHead);
```

To add an item, use an SLIST_ENTRY object (usually part of a bigger structure) by passing it to ExInterlockedPushEntrySList macro:

```
PSLIST_ENTRY ExInterlockedPushEntrySList (
    _Inout_ PSLIST_HEADER ListHead,
    _Inout_ __drv_aliasesMem PSLIST_ENTRY ListEntry,
    _Inout_opt_ _Requires_lock_not_held_(*_Curr_) PKSPIN_LOCK Lock);
```

The spin lock should be passed as NULL, as this macro expands to calling ExpInterlockedPushEntrySList:

```
PSLIST_ENTRY ExpInterlockedPushEntrySList (
    _Inout_ PSLIST_HEADER ListHead,
    _Inout_ __drv_aliasesMem PSLIST_ENTRY ListEntry);
```

As you can see, the spin lock is not used at all. It's not quite clear why the macro accepts a spin lock, but the documentation hints that this is only useful with doubly-linked lists, so the macro prototype is probably for consistency only.

Similarly, popping an item (from the head only) is available with ExInterlockedPopEntrySList:

```
PSLIST_ENTRY ExInterlockedPopEntrySList (
    _Inout_ PSLIST_HEADER ListHead,
    _Inout_opt_ _Requires_lock_not_held_(*_Curr_) PKSPIN_LOCK Lock);
```

Again, the spin lock is not needed.

To clean the list entirely, call ExInterlockedFlushSList:

```
PSLIST_ENTRY ExInterlockedFlushSList (_Inout_ PSLIST_HEADER ListHead);
```

The function simply replaces (atomically) the head with NULL (making the list empty), and returns the previous head. It's the responsibility of the driver to iterate through the list and free items that were dynamically allocated explicitly.

Finally, you can call ExQueryDepthSList to get the number of items in the list:

```
USHORT ExQueryDepthSList (_In_ PSLIST_HEADER SListHead);
```

It's a fast operation, as the count is stored as part of SLIST_HEAD.

Callback Objects

The kernel defines a *Callback* object type that can be used to provide notifications, while maintaining a higher level of abstraction, where the callback object hides the callback(s) that should be invoked. There are quite a few callback objects used on a normal system, which can be viewed with *Sysinternals WinObj* tool (figure 11-1).

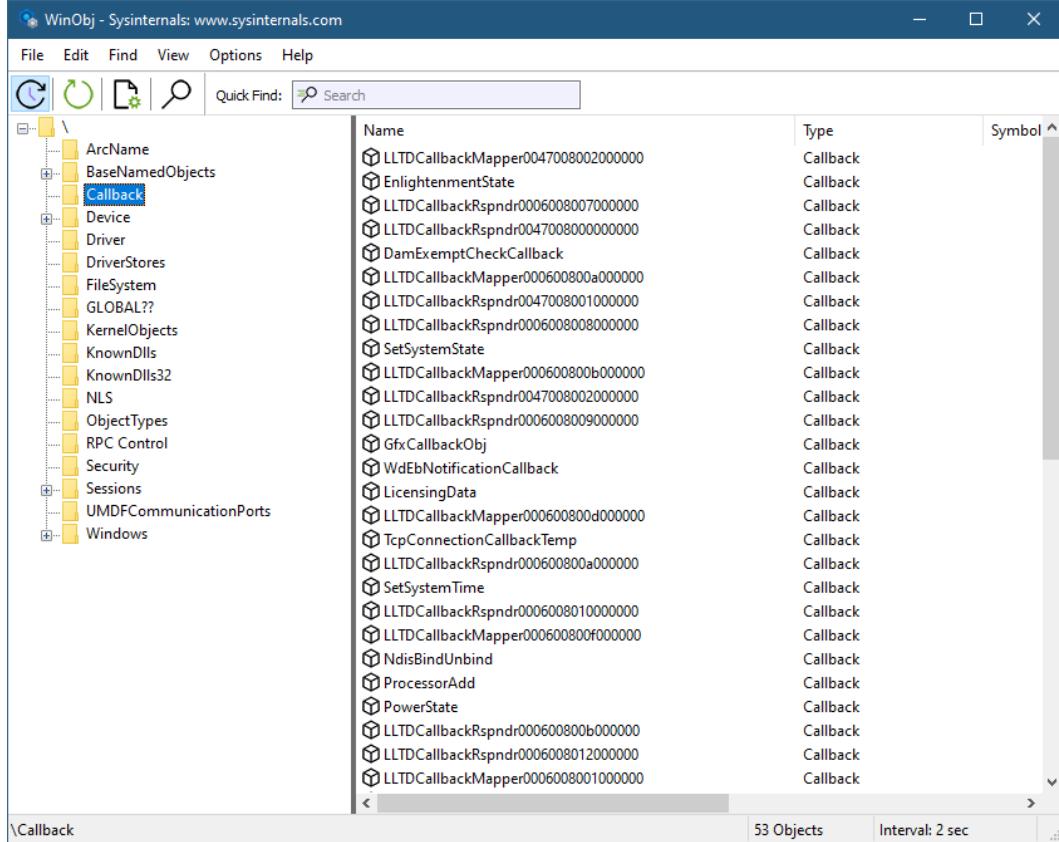


Figure 11-1: Callback objects

There are three existing (and documented) callback objects that drivers can use (all in the \Callback object manager directory):

- *ProcessorAdd* - callback invoked when a processor is hot-added to the system.
- *PowerState* - callback invoked when one of the following occurs: the system is about to go to a low power state, the system switches from AC to DC (or back), or the system power policy changes as a result of a user's or application's request.
- *SetSystemTime* - callback invoked when the system time is changed.

Working with an existing callback object, or when creating one is essentially the same. The first

step is to create the callback object with `ExCreateCallback`, giving it a name with the provided `OBJECT_ATTRIBUTES`:

```
NTSTATUS ExCreateCallback (
    _Outptr_ PCALLBACK_OBJECT *CallbackObject,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ BOOLEAN Create,
    _In_ BOOLEAN AllowMultipleCallbacks);
```

The `OBJECT_ATTRIBUTES` structure must be initialized with a name, and optionally other attributes, the most common being `OBJ_CASE_INSENSITIVE`. Set `Create` to TRUE to create a new callback object if such does not exist. If a new callback object is created, `AllowMultipleCallbacks` specifies whether multiple callbacks are allowed. If `Create` is FALSE or the object exists, this parameter is ignored. The returned object's (`CallbackObject`) reference count is incremented.

With a callback object in hand, an interested client can register a callback function with `ExRegisterCallback`:

```
PVOID ExRegisterCallback (
    _Inout_ PCALLBACK_OBJECT CallbackObject,
    _In_ PCALLBACK_FUNCTION CallbackFunction,
    _In_opt_ PVOID CallbackContext);
```

The function returns a registration cookie to be used to unregister with `ExUnregisterCallback`. The callback function itself must have the following prototype:

```
VOID CallbackFunction (
    _In_opt_ PVOID CallbackContext,
    _In_opt_ PVOID Argument1,
    _In_opt_ PVOID Argument2);
```

`CallbackContext` is whatever was passed in to `ExRegisterCallback`, and the two arguments are provided by whoever is invoking the callbacks - these can be anything, as determined by the invoker.

When using existing callback objects, that's all there is to it. If you are controlling the callback object, then you can invoke the callbacks that are currently registered with `ExNotifyCallback`:

```
VOID ExNotifyCallback (
    _In_ PVOID CallbackObject,
    _In_opt_ PVOID Argument1,
    _In_opt_ PVOID Argument2);
```

Finally, to unregister your callback (if you're a client), call `ExUnregisterCallback`, passing the registration cookie:

```
void ExUnregisterCallback (_Inout_ PVOID CallbackRegistration);
```

You must also decrement the reference count of the callback object with `ObDereferenceObject`, otherwise the callback object will leak. You can do that for the existing callback objects as soon as you don't need them.

The `Callbacks` driver demonstrates using a callback object with the `SetSystemTime` documented callback. Here is the entire driver:

```
void SystemTimeChanged(PVOID context, PVOID arg1, PVOID arg2);
void OnUnload(PDRIVER_OBJECT);

PVOID g_RegCookie;

extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    OBJECT_ATTRIBUTES attr;
    UNICODE_STRING name = RTL_CONSTANT_STRING(L"\\"CALLBACK"\SetSystemTime");
    InitializeObjectAttributes(&attr, &name,
        OBJ_CASE_INSENSITIVE, nullptr, nullptr);
    PCALLBACK_OBJECT callback;
    //
    // open the callback object
    //
    auto status = ExCreateCallback(&callback, &attr, FALSE, TRUE);
    if (!NT_SUCCESS(status)) {
        KdPrint(("Failed to create callback object (0x%X)\n", status));
        return status;
    }

    //
    // register our callback
    //
    g_RegCookie = ExRegisterCallback(callback, SystemTimeChanged, nullptr);
    if (g_RegCookie == nullptr) {
        ObDereferenceObject(callback);
        KdPrint(("Failed to register callback\n"));
        return STATUS_UNSUCCESSFUL;
    }

    //
    // callback object no longer needed
    //
    ObDereferenceObject(callback);
```

```
DriverObject->DriverUnload = OnUnload;

return STATUS_SUCCESS;
}

void SystemTimeChanged(PVOID context, PVOID arg1, PVOID arg2) {
    UNREFERENCED_PARAMETER(context);
    //
    // system time changed!
    // (arg1 and arg2 are always zero with this object)
    //
    DbgPrint("System time changed 0x%p 0x%p!\n", arg1, arg2);
}

void OnUnload(PDRIVER_OBJECT) {
    ExUnregisterCallback(g_RegCookie);
}
```

In this chapter we've looked at some potentially useful techniques a driver might want to use. In the next chapter, we'll turn our attention to file system mini-filters.

Chapter 12: File System Mini-Filters

File systems are targets for I/O operations to access files and other devices implemented as file systems (such as named pipes and mailslots). Windows supports several file systems, most notably NTFS, its native file system. File system filtering is the mechanism by which drivers can intercept calls destined to file systems. This is useful for many types of software, such as anti-viruses, backups, encryption, redirection, and more.

Windows supported for a long time a filtering model known as *file system filters*, which is now referred to as *legacy file system filters*. A newer model called *file system mini-filters* was developed to replace the legacy filter mechanism. Mini-filters are easier to write in many respects, and are the preferred way to develop file system filtering drivers. In this chapter we'll cover the basics of file system mini-filters.

This is a long chapter, so you may want to consume it in chunks. The example drivers get more complex as the chapter progresses.

In this chapter:

- **Introduction**
 - **Loading and Unloading**
 - **Initialization**
 - **Installation**
 - **Processing I/O Operations**
 - **File Names**
 - **The Delete Protector Driver**
 - **The Directory Hiding Driver**
 - **Contexts**
 - **Initiating I/O Requests**
 - **The File Backup Driver**
 - **User Mode Communication**
 - **Debugging**
 - **Exercises**
-

Introduction

Legacy file system filters are notoriously difficult to write. The driver writer has to take care of an assortment of little details, many of them boilerplate, complicating development. Legacy filters cannot be unloaded while the system is running which means the system had to be restarted to load an updated version of the driver. With the mini-filter model, drivers can be loaded and unloaded dynamically, thus streamlining the development workflow considerably.

Internally, a legacy filter provided by Windows called the *Filter Manager* is tasked with managing mini-filters. A typical filter layering is shown in figure 12-1.

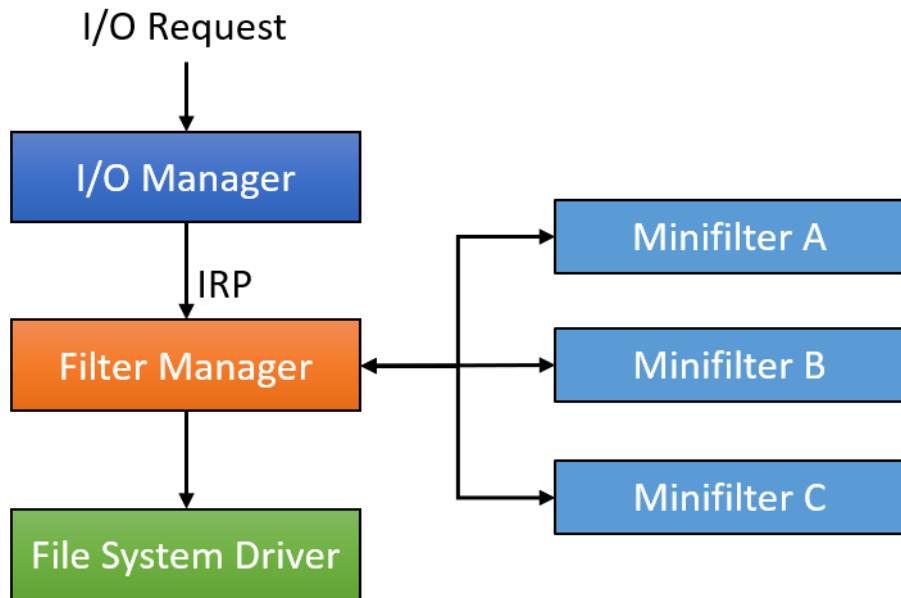


Figure 12-1: Mini-filters managed by the filter manager

Each mini-filter has its own *Altitude*, which determines its relative position in the device stack. The filter manager is the one receiving the IRPs just like any other legacy filter and then calls upon the mini-filters it's managing, in descending order of altitude.

In some unusual cases, there may be another legacy filter in the hierarchy, that may cause a mini-filter “split”, where some are higher in altitude than the legacy filter and some lower. In such a case, more than one instance of the filter manager will load, each managing its own mini-filters. Every such filter manager instance is referred to as a *Frame*. Figure 12-2 shows such an example with two frames.

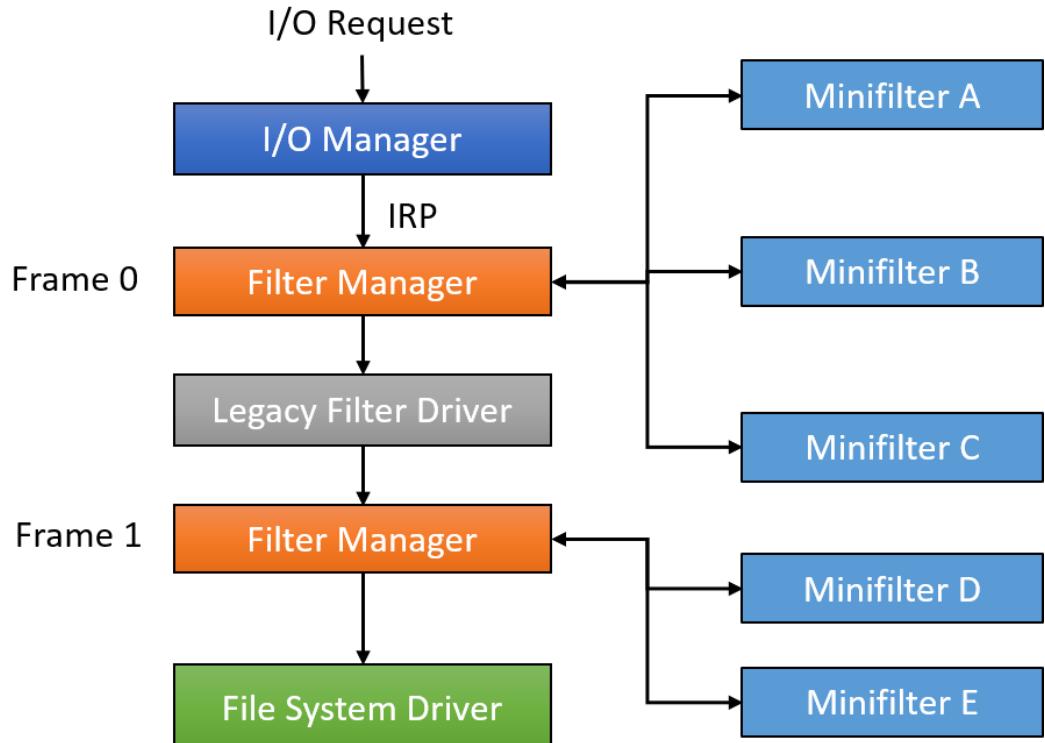


Figure 12-2: Mini-filters in two filter manager frames

Loading and Unloading

Mini-filter drivers must be loaded just like any other driver. The user mode API to use is `FilterLoad`, passing the driver's name (its key in the registry at `HKLM\System\CurrentControlSet\Services\drivername`). Internally, the kernel `F1tLoadFilter` API is invoked, with the same semantics. Just like any other driver, the `SeLoadDriverPrivilege` privilege must be present (and enabled) in the caller's token if called from user mode. By default, it's present in admin-level tokens, but not in standard users tokens.

Loading a mini-filter driver is equivalent to loading a standard software driver. Unloading, however, is not.

Unloading a mini-filter is accomplished with the `FilterUnload` API in user mode, or `F1tUnloadFilter` in kernel mode. This operation requires the same privilege as for loads, but is not guaranteed to succeed, because the mini-filter's *Filter unload callback* (discussed later) is called, which can fail the request so that driver remains loaded.

Although using APIs to load and unload filters has its uses, during development it's usually easier to use a built-in tool that can accomplish that (and more) called *fltmc.exe* (residing in the *System32* directory). Invoking it (from an elevated command window) without arguments lists the currently loaded mini-filters. Here is the output from a Windows 11 machine:

```
C:\WINDOWS\system32>fltmc
```

Filter Name	Num Instances	Altitude	Frame
bindflt	1	409800	0
wtd	5	385110	0
WdFilter	5	328010	0
storqosflt	0	244000	0
wcifs	0	189900	0
PrjFlt	0	189800	0
CldFlt	1	180451	0
bfs	7	150000	0
FileCrypt	0	141100	0
luafv	1	135000	0
npsvctrig	1	46000	0
Wof	3	40700	0
FileInfo	5	40500	0
WinSetupMon	2	40300	0

For each filter, the output shows the driver's name, the number of instances each filter has currently running (each instance is attached to a volume), its altitude and the filter manager frame it's part of.

You may be wondering why there are drivers with different number of instances. The short answer is that it's up to the driver to decide whether to attach to a given volume or not (we'll look at this in more detail later in this chapter).

Loading a driver with *fltmc.exe* is done with the *load* option, like so:

```
fltmc load myfilter
```

Conversely, unloading is done with the *unload* command line option:

```
fltmc unload myfilter
```

fltmc includes other options. Type *fltmc -?* to get the full list. For example, you can get the details of all instances for each driver using *fltmc instances*. Similarly, you can get a list of all volumes mounted on a system with *fltmc volumes*. We'll see later in this chapter how this information is conveyed to the driver.

File system drivers and filters are created in the *FileSystem* directory of the Object Manager namespace. Figure 12-3 shows this directory in *WinObj*.

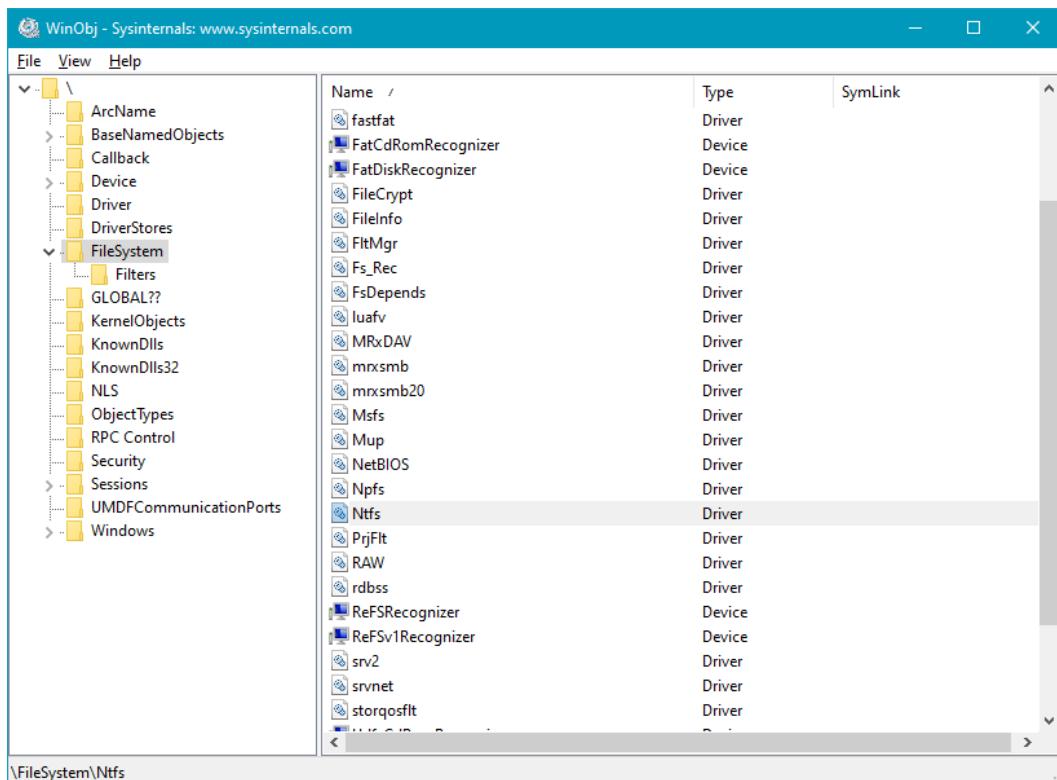


Figure 12-3: File system drivers, filters and mini-filters in WinObj

Initialization

A file system mini-filter driver has a `DriverEntry` routine, just like any other driver. The driver must register itself as a mini-filter with the filter manager, specifying various settings, such as what operations it wishes to intercept. The driver sets up appropriate structures and then calls `FltRegisterFilter` to register. If successful, the driver can do further initializations as needed and finally call `FltStartFiltering` to actually start filtering operations.

Note that the driver does **not** need to set up dispatch routines on its own (`IRP_MJ_READ`, `IRP_MJ_WRITE`, etc.). This is because the driver is not directly in the I/O path; the filter manager is.

`FltRegisterFilter` has the following prototype:

```
NTSTATUS FltRegisterFilter (
    _In_ PDRIVER_OBJECT Driver,
    _In_ const FLT_REGISTRATION *Registration,
    _Outptr_ PFLT_FILTER *RetFilter);
```

The required `FLT_REGISTRATION` structure provides all the necessary information for registration. It's defined like so:

```

typedef struct _FLT_REGISTRATION {
    USHORT Size;
    USHORT Version;

    FLT_REGISTRATION_FLAGS Flags;

    const FLT_CONTEXT_REGISTRATION *ContextRegistration;
    const FLT_OPERATION_REGISTRATION *OperationRegistration;

    PFLT_FILTER_UNLOAD_CALLBACK FilterUnloadCallback;
    PFLT_INSTANCE_SETUP_CALLBACK InstanceSetupCallback;
    PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK InstanceQueryTeardownCallback;
    PFLT_INSTANCE_TEARDOWN_CALLBACK InstanceTeardownStartCallback;
    PFLT_INSTANCE_TEARDOWN_CALLBACK InstanceTeardownCompleteCallback;

    PFLT_GENERATE_FILE_NAME GenerateFileNameCallback;
    PFLT_NORMALIZE_NAME_COMPONENT NormalizeNameComponentCallback;
    PFLT_NORMALIZE_CONTEXT_CLEANUP NormalizeContextCleanupCallback;

    PFLT_TRANSACTION_NOTIFICATION_CALLBACK TransactionNotificationCallback;
    PFLT_NORMALIZE_NAME_COMPONENT_EX NormalizeNameComponentExCallback;

#if FLT_MGR_WIN8
    PFLT_SECTION_CONFLICT_NOTIFICATION_CALLBACK SectionNotificationCallback;
#endif
} FLT_REGISTRATION, *PFLT_REGISTRATION;

```

There is a lot of information encapsulated in this structure. The most important fields are described below:

- *Size* must be set to the size of the structure, which may depend on the target Windows version (set in the project's properties). Drivers typically just specify `sizeof(FLT_REGISTRATION)`.
- *Version* is also based on the target Windows version. Drivers use `FLT_REGISTRATION_VERSION`.
- *Flags* can be zero or a combination of the following values:
 - `FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP` - the driver does not support a stop request, regardless of other settings.
 - `FLTFL_REGISTRATION_SUPPORT_NPFS_MSFS` - the driver is aware of named pipes and mailslots and wishes to filter requests to these file systems as well (see the sidebar “Pipes and Mailslots” for more information).
 - `FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME` (Windows 10 version 1607 and later) - the driver will support attaching to a Direct Access Volume (DAX), if such a volume is available (see the sidebar “Direct Access Volume”).

Pipes and Mailslots

A *named pipe* is a uni- or bi-directional communication mechanism from a server to one or more clients, implemented as a file system (*npfs.sys*). The Windows API provides specific functions for creating pipe servers. The `CreateNamedPipe` function can be used to create a named pipe server, to which clients can connect using the normal `CreateFile` API with a “file name” in this form: `\\\<server>\pipe\<pipename>`.

A *mailslot* is a uni-directional communication mechanism, implemented as a file system (*msfs.sys*), where a server process opens a mailslot (you can think of it as a mailbox), to which messages can be sent by clients. `CreateMailslot` is the Windows API to create a mailslot, while clients connect with `CreateFile` with a file name in the form `\\\<server>\mailslot\<mailslotname>`.

For more information check out the Microsoft documentation or my book “Windows 10 System Programming, Part 2”.

Direct Access Volume (DAX or DAS)

Direct access volumes is a relatively new capability added in Windows 10 version 1607 that provides support for a new kind of storage based on direct access to the underlying byte data. This is supported by new a type of storage hardware referred to as *Storage Class Memory* - a non-volatile storage medium with RAM-like performance. (more information can be found on the web.)

- *ContextRegistration* - an optional pointer to `FLT_CONTEXT_REGISTRATION` structure array, where each entry represents a context that driver may use in its work. *Context* refers to some driver-defined data that can be attached to file system entities, such as files and volumes. We’ll look at contexts later in this chapter. Some drivers don’t need any contexts, and can set this field to `NULL`.
- *OperationRegistration* - by far the most important field. This is a pointer to an array of `FLT_OPERATION_REGISTRATION` structures, each specifying the operation of interest and a pre and/or post callback the driver wishes to be called upon. The next section provides the details.
- *FilterUnloadCallback* - specifies a function to be called when the driver is about to be unloaded. If `NULL` is specified, the driver cannot be unloaded. If the driver sets a callback and returns a successful status, the driver is unloaded; in that case the driver must call `F1tUnregisterFilter` to unregister itself before being unloaded. Returning a non-success status does not unload the driver.
- *InstanceSetupCallback* - this callback allows the driver to be notified when an instance is about to be attached to a new volume. The driver may return `STATUS_SUCCESS` to attach or `STATUS_-FLT_DO_NOT_ATTACH` if the driver does not wish to attach to this volume.
- *InstanceQueryTeardownCallback* - an optional callback invoked before detaching from a volume. This can happen because of an explicit request to detach using `F1tDetachVolume` in kernel mode or `FilterDetach` in user mode. If `NULL` is specified by the callback, the detach operation is aborted.
- *InstanceTeardownStartCallback* - an optional callback invoked when teardown of an instance has started. The driver should complete any pending operations so that instance teardown can

complete. Specifying NULL for this callback does not prevent instance teardown (prevention can be achieved with the previous query teardown callback).

- *InstanceTeardownCompleteCallback* - an optional callback invoked after all the pending I/O operations complete or canceled.

The rest of the callback fields are all optional and seldom used. These are beyond the scope of this book.

Operations Callback Registration

A mini-filter driver must indicate which operations it's interested in. This is provided at mini-filter registration time with an array of `FLT_OPERATION_REGISTRATION` structures defined like so:

```
typedef struct _FLT_OPERATION_REGISTRATION {
    UCHAR MajorFunction;
    FLT_OPERATION_REGISTRATION_FLAGS Flags;
    PFLT_PRE_OPERATION_CALLBACK PreOperation;
    PFLT_POST_OPERATION_CALLBACK PostOperation;

    PVOID Reserved1;      // reserved
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

The operation itself is identified by a major function code, many of which are the same as the ones we met in previous chapters: `IRP_MJ_CREATE`, `IRP_MJ_READ`, `IRP_MJ_WRITE` and so on. However, there are other operations identified with a major function that do not have a real major function dispatch routine. This abstraction provided by the filter manager helps to isolate the mini-filter from knowing the exact source of the operation - it could be a real IRP or it could be another operation that is abstracted as an IRP. Furthermore, file systems support another mechanism for receiving requests, known as *Fast I/O*. Fast I/O is used for synchronous I/O with cached files. Fast I/O requests transfer data between user buffers and the system cache directly, bypassing the file system and storage driver stack, thus avoiding unnecessary overhead. The NTFS file system driver, as a canonical example, supports Fast I/O.

Fast I/O is initialized by allocating a `FAST_IO_DISPATCH` structure (containing a long list of callbacks), filling it in, and then setting the `FastIoDispatch` member of `DRIVER_OBJECT` to this structure.

This information can be viewed with a kernel debugger by using the `!drvobj` command as shown here for the NTFS file system driver:

```
1kd> !drvobj \filesystem\ntfs f
Driver object (fffffad8b19a60bb0) is for:
\FileSystem\Ntfs

Driver Extension List: (id , addr)

Device Object list:
fffffad8c22448050  fffffad8c476e3050  fffffad8c3943f050  fffffad8c208f1050
fffffad8b39e03050  fffffad8b39e87050  fffffad8b39e73050  fffffad8b39d52050
fffffad8b19fc9050  fffffad8b199f3d80

DriverEntry:    fffff8026b609010 Ntfs!GsDriverEntry
DriverStartIo:  00000000
DriverUnload:   00000000
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE           fffff8026b49bae0  Ntfs!NtfsFsdCreate
[01] IRP_MJ_CREATE_NAMED_PIPE fffff80269141d40  nt!IoPInvalidDeviceRequest
[02] IRP_MJ_CLOSE             fffff8026b49d730  Ntfs!NtfsFsdClose
[03] IRP_MJ_READ              fffff8026b3b3f80  Ntfs!NtfsFsdRead
...
[19] IRP_MJ_QUERY_QUOTA       fffff8026b49c700  Ntfs!NtfsFsdDispatchWait
[1a] IRP_MJ_SET_QUOTA         fffff8026b49c700  Ntfs!NtfsFsdDispatchWait
[1b] IRP_MJ_PNP               fffff8026b5143e0  Ntfs!NtfsFsdPnp

Fast I/O routines:
FastIoCheckIfPossible        fffff8026b5adff0  Ntfs!NtfsFastIoCheckIfPossible
FastIoRead                   fffff8026b49e080  Ntfs!NtfsCopyReadA
FastIoWrite                  fffff8026b46cb00  Ntfs!NtfsCopyWriteA
FastIoQueryBasicInfo         fffff8026b4d50d0  Ntfs!NtfsFastQueryBasicInfo
FastIoQueryStandardInfo      fffff8026b4d2de0  Ntfs!NtfsFastQueryStdInfo
FastIoLock                   fffff8026b4d6160  Ntfs!NtfsFastLock
FastIoUnlockSingle           fffff8026b4d6b40  Ntfs!NtfsFastUnlockSingle
FastIoUnlockAll               fffff8026b5ad2d0  Ntfs!NtfsFastUnlockAll
FastIoUnlockAllByKey          fffff8026b5ad590  Ntfs!NtfsFastUnlockAllByKey
ReleaseFileForNtCreateSection fffff8026b3c3670  Ntfs!NtfsReleaseForCreateSecti\
on
FastIoQueryNetworkOpenInfo nfo          fffff8026b4d4cb0  Ntfs!NtfsFastQueryNetworkOpenI\
nfo
AcquireForModWrite            fffff8026b3c4c20  Ntfs!NtfsAcquireFileForModWrite
MdlRead                      fffff8026b46b6a0  Ntfs!NtfsMdlReadA
MdlReadComplete               fffff8026911aca0  nt!FsRt1MdlReadCompleteDev
```

PrepareMdlWrite	fffff8026b46aae0	Ntfs!NtfsPrepareMdlWriteA
MdlWriteComplete	fffff802696c41e0	nt!FsRt1MdlWriteCompleteDev
FastIoQueryOpen	fffff8026b4d4940	Ntfs!NtfsNetworkOpenCreate
ReleaseForModWrite	fffff8026b3c5a40	Ntfs!NtfsReleaseFileForModWrite
AcquireForCcFlush	fffff8026b3a8690	Ntfs!NtfsAcquireFileForCcFlush
ReleaseForCcFlush	fffff8026b3c5610	Ntfs!NtfsReleaseFileForCcFlush

Device Object stacks:

```
!devstack fffffad8c22448050 :
!DevObj          !DrvObj                !DevExt          ObjectName
fffffad8c4adcba70  \FileSystem\FltMgr  fffffad8c4adcbbc0
> fffffad8c22448050  \FileSystem\Ntfs   fffffad8c224481a0
```

(truncated)

Processed 10 device objects.

The filter manager abstracts I/O operations, regardless of whether they are IRP-based or fast I/O based. Mini-filters can intercept any such request. If the driver is not interested in fast I/O, for example, it can query the actual request type provided by the filter manager with the `FLT_IS_FASTIO_OPERATION` and/or `FLT_IS_IRP_OPERATION` macros.

Table 12-1 lists some of the common major functions for file system mini-filters with a brief description for each.

Table 12-1: Common major functions

Major function	Dispatch routine?	Description
IRP_MJ_CREATE	Yes	Create or open a file/directory
IRP_MJ_READ	Yes	Read from a file
IRP_MJ_WRITE	Yes	Write to a file
IRP_MJ_QUERY_EA	Yes	Read extended attributes from a file/directory
IRP_MJ_DIRECTORY_CONTROL	Yes	Request sent to a directory
IRP_MJ_FILE_SYSTEM_CONTROL	Yes	File system device I/O control request
IRP_MJ_SET_INFORMATION	Yes	Various information setting for a file (e.g. delete, rename)
IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION	No	Section (memory mapped file) is being opened
IRP_MJ_OPERATION_END	No	signals the end of array of operations callbacks

The second field in `FLT_OPERATION_REGISTRATION` is a set of flags which can be zero or a combination of one of the following flags affecting read and write operations:

- `FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO` - do not invoke the callback(s) if it's cached I/O (such as fast I/O operations, which are always cached).
- `FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO` - do not invoke the callback(s) for paging I/O (IRP-based operations only).
- `FLTFL_OPERATION_REGISTRATION_SKIP_NON_DASD_IO` - do not invoke the callback(s) for DAX volumes.

The next two fields are the pre and post operation callbacks, where at least one must be non-NULL (otherwise, why have that entry in the first place?). Here is an example of initializing an array of `FLT_OPERATION_REGISTRATION` structures (for an imaginary driver called "Sample"):

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE, 0, nullptr, SamplePostCreateOperation },
    { IRP_MJ_WRITE, FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO,
        SamplePreWriteOperation, nullptr },
    { IRP_MJ_CLOSE, 0, nullptr, SamplePostCloseOperation },
    { IRP_MJ_OPERATION_END }
};
```

With this array at hand, registration for a driver that does not require any contexts could be done with the following code:

```
const FLT_REGISTRATION FilterRegistration = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0,                                // Flags
    nullptr,                           // Context
    Callbacks,                          // Operation callbacks
    ProtectorUnload,                   // MiniFilterUnload
    SampleInstanceSetup,                // InstanceSetup
    SampleInstanceQueryTeardown,       // InstanceQueryTeardown
    SampleInstanceTeardownStart,        // InstanceTeardownStart
    SampleInstanceTeardownComplete,     // InstanceTeardownComplete
};

PFLT_FILTER Filter;

NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    NTSTATUS status;
```

```

//... some code
status = FltRegisterFilter(DriverObject, &FilterRegistration, &Filter);
if(NT_SUCCESS(status)) {
    //
    // start I/O filtering
    //
    status = FltStartFiltering(Filter);
    if(!NT_SUCCESS(status))
        FltUnregisterFilter(Filter);
}
return status;
}

```

The Altitude

As we've seen already, file system mini-filters must have an altitude, indicating their relative "position" within the file system filters hierarchy. Contrary to the altitude we've already encountered with object and registry callbacks, a mini-filter's altitude value may be potentially significant.

First, the value of the altitude is not provided as part of mini-filter's registration, but is read from the registry. When the driver is installed, its altitude is written in the proper location in the registry. Figure 12-4 shows the registry entry for the built-in *Fileinfo* mini-filter driver; the Altitude is clearly visible, and is the same value shown earlier with the *fltmc.exe* tool.

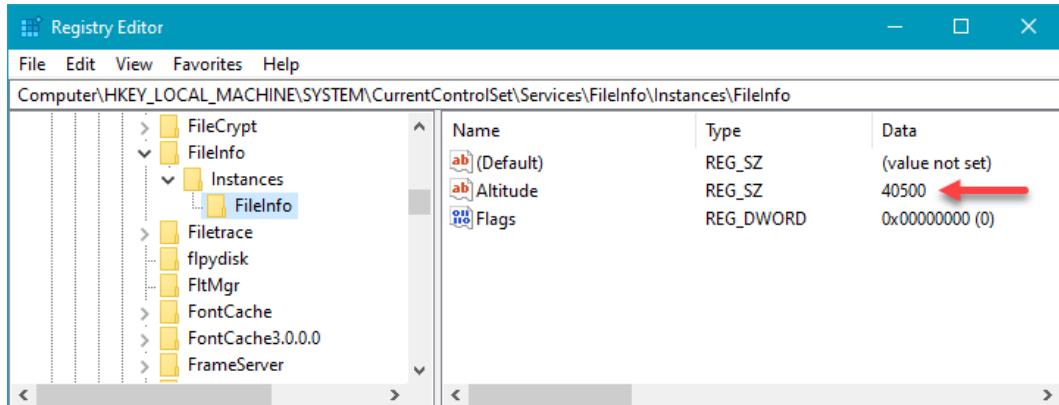


Figure 12-4: Altitude in the registry

Here is an example that should clarify why altitude matters. Suppose there is a mini-filter at altitude 10000 whose job is to encrypt data when written, and decrypt when read. Now suppose another mini-filter whose job is to check data for malicious activity is at altitude 9000. This layout is depicted in Figure 12-5.

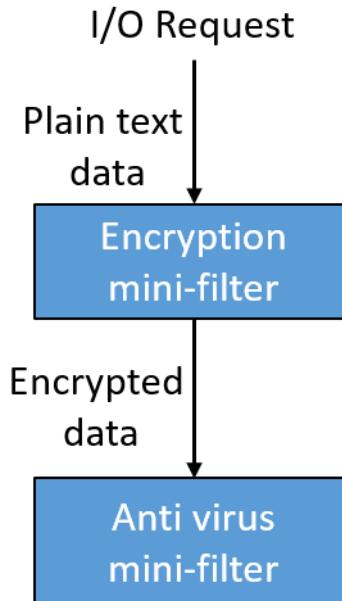


Figure 12-5: Two mini-filter layout

The encryption driver encrypts incoming data to be written, which is then passed on to the anti-virus driver. The anti-virus driver is in a problem, as it sees the encrypted data with no viable way of decrypting it (and even if it could, that would be wasteful). In such a case, the anti-virus driver must have an altitude higher than the encryption driver. How can such a driver guarantee this is in fact the case?

To rectify this (and other similar) situations, Microsoft has defined ranges of altitudes for drivers based on their requirements (and ultimately, their role). In order to obtain a proper altitude, the driver publisher must send an email to Microsoft (fsfcomm@microsoft.com) and ask an altitude be allocated for that driver based on its intended target. Check out [this link³](#) for the complete list of altitude ranges. In fact, the link shows all drivers that Microsoft has allocated an altitude for, with the file name, the altitude and the publishing company.



The altitude request email details are located at <https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/minifilter-altitude-request>.



For testing purposes, you can choose any appropriate altitude without going through Microsoft, but you *should* obtain an official altitude for production use.

Table 12-2 shows the list of groups and the altitude range for each group.

³<https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes>

Table 12-2: Altitude ranges and load order groups

Altitude range	Group name
420000 - 429999	Filter
400000 - 409999	FSFilter Top
360000 - 389999	FSFilter Activity Monitor
340000 - 349999	FSFilter Undelete
320000 - 329998	FSFilter Anti-Virus
300000 - 309998	FSFilter Replication
280000 - 289998	FSFilter Continuous Backup
260000 - 269998	FSFilter Content Screener
240000 - 249999	FSFilter Quota Management
220000 - 229999	FSFilter System Recovery
200000 - 209999	FSFilter Cluster File System
180000 - 189999	FSFilter HSM
170000 - 174999	FSFilter Imaging (ex: .ZIP)
160000 - 169999	FSFilter Compression
140000 - 149999	FSFilter Encryption
130000 - 139999	FSFilter Virtualization
120000 - 129999	FSFilter Physical Quota management
100000 - 109999	FSFilter Open File
80000 - 89999	FSFilter Security Enhancer
60000 - 69999	FSFilter Copy Protection
40000 - 49999	FSFilter Bottom
20000 - 29999	FSFilter System

Installation

Figure 12-4 shows that there are additional Registry entries that must be set, beyond what is possible with the standard `CreateService` installation API we've been using up until now (indirectly with the `sc.exe` tool). One way to install a file system mini-filter driver is to use an INF file. This approach was used in the first edition of the book, because at the time there was a driver project template for file system mini-filters provided with the WDK that used an INF file. Curiously enough, that template went away in recent WDKs without any explanation. Although it's possible to use an existing project from the first edition of the book as a basis for a driver that uses an INF file for installation, I will show another way that does not require an INF file at all.

If you want to see how to use an INF file to install a file system mini-filter, please see chapter 10 in the first edition of the book. Using an INF file is perfectly fine.

The alternative approach we'll use is to write the required Registry values directly as part of `DriverEntry` prior to calling `FltRegisterFilter`. The next driver example in this chapter, *DelProtect*, that will be discussed in an upcoming section, uses this technique. Here is the code (error handling omitted):

```
HANDLE hKey = nullptr, hSubKey = nullptr;
NTSTATUS status;

OBJECT_ATTRIBUTES keyAttr = RTL_CONSTANT_OBJECT_ATTRIBUTES(
    RegistryPath, OBJ_KERNEL_HANDLE);
status = ZwOpenKey(&hKey, KEY_WRITE, &keyAttr);

UNICODE_STRING subKey = RTL_CONSTANT_STRING(L"Instances");
OBJECT_ATTRIBUTES subKeyAttr;
InitializeObjectAttributes(&subKeyAttr, &subKey, OBJ_KERNEL_HANDLE, hKey,
    nullptr);
status = ZwCreateKey(&hSubKey, KEY_WRITE, &subKeyAttr, 0, nullptr, 0, nullptr);

// set "DefaultInstance" value. Any name is fine.
//
UNICODE_STRING valueName = RTL_CONSTANT_STRING(L"DefaultInstance");
WCHAR name[] = L"DelProtectDefaultInstance";
status = ZwSetValueKey(hSubKey, &valueName, 0, REG_SZ, name, sizeof(name));

//
// create "instance" key under "Instances"
//
UNICODE_STRING instKeyName;
RtlInitUnicodeString(&instKeyName, name);
HANDLE hInstKey;
InitializeObjectAttributes(&subKeyAttr, &instKeyName, OBJ_KERNEL_HANDLE,
    hSubKey, nullptr);
status = ZwCreateKey(&hInstKey, KEY_WRITE, &subKeyAttr, 0, nullptr, 0, nullptr);

//
// write out altitude
```

```
//  
WCHAR altitude[] = L"425342";  
UNICODE_STRING altitudeName = RTL_CONSTANT_STRING(L"Altitude");  
status = ZwSetValueKey(hInstKey, &altitudeName, 0, REG_SZ,  
    altitude, sizeof(altitude));  
  
//  
// write out flags  
//  
UNICODE_STRING flagsName = RTL_CONSTANT_STRING(L"Flags");  
ULONG flags = 0;  
status = ZwSetValueKey(hInstKey, &flagsName, 0, REG_DWORD,  
    &flags, sizeof(flags));  
  
ZwClose(hInstKey);
```

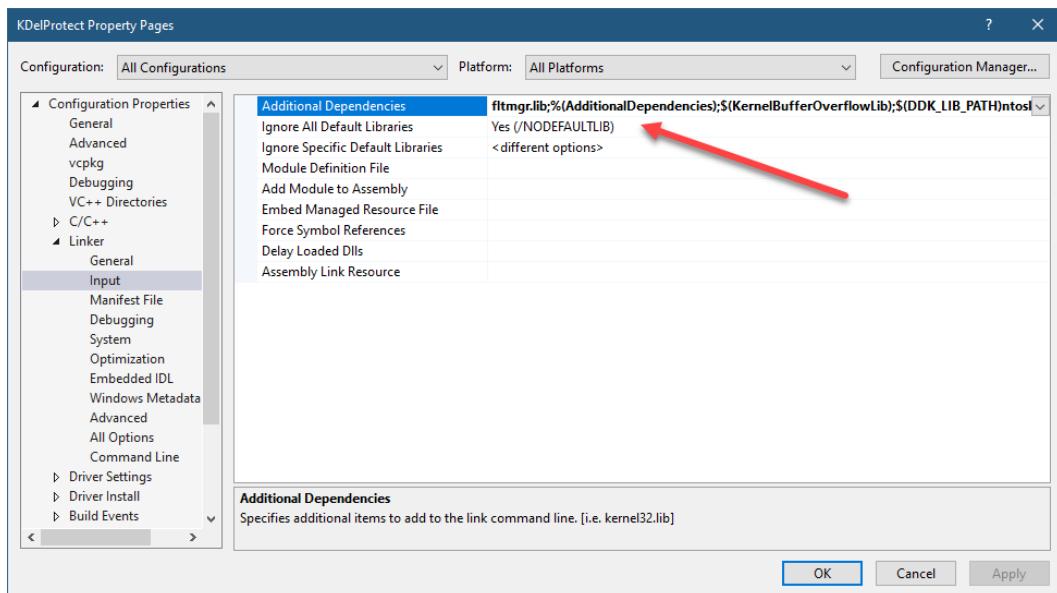
The *Flags* value in the Registry indicates what types of volume attach the driver is interested in. This can have one of the following values:

- 1 - the driver is not interested in automatic attachments.
- 2 - the driver is not interested in manual attachments (as a result of `FilterAttach`, `FilterAttachAtAltitude` or their kernel equivalents).
- 0 - the driver is interested in all attachments.



If you don't write the "Flags" value at all, `FltRegisterFilter` fails.

The last missing piece is the need to link with the Filter Manager API, implemented in *FltMgr.lib*. It must be added to the Linker input libraries as shown in figure 12-6.

Figure 12-6: *FltMgr.lib* in Linker options

Make sure you select “All Platforms” and “All Configurations”. You cannot add the *FltMgr.lib* in source code using a `#pragma comment(lib, "ftlmgm")` similarly to user mode. I don’t know why the linker does not accept this option.

Installing the Driver

With the Registry stuff being written by the driver itself, installing the file system mini-filter can be done with the same `CreateService` API call, or a tool such as `sc.exe`. The only difference is specifying the driver type to be file system-related rather than a generic driver. Here is the command for the *DelProtect* driver:

```
sc create delprotect type= filesys binPath= c:\Test\kdelprotect.sys
```

Notice the “`type= filesys`” instead of “`type= kernel`” we used in previous chapters. This writes a value of 2 in the *Type* value in the Registry, rather than 1. Does that really matter? As far as I can tell - it doesn’t, but still, it’s best to write the expected value.

Processing I/O Operations

The main function of a file system mini-filter is processing I/O operations by implementing pre and/or post callbacks for the operations of interest. Pre operations allow a mini-filter to reject an operation completely, while post operations allow looking at the result of the operation, and in some cases - making changes to the returned information.

Pre Operation Callbacks

All pre-operation callbacks have the same prototype as follows:

```
FLT_PREOP_CALLBACK_STATUS SomePreOperation (
    _Inout_     PFLT_CALLBACK_DATA Data,
    _In_        PCFLT_RELATED_OBJECTS FltObjects,
    _Outptr_    PVOID *CompletionContext);
```

First, let's look at the possible return values from a pre-operation, typed as the `FLT_PREOP_CALLBACK_STATUS` enumeration. Here are the common return values to use:

- `FLT_PREOP_COMPLETE` indicates the driver is completing the operation. The filter manager does not call the post-operation callback (if registered) and does not forward the request to lower-layer mini-filters.
- `FLT_PREOP_SUCCESS_NO_CALLBACK` indicates the pre-operation is done with the request and lets it continue flowing to the next filter. The driver does not want its post-operation callback to be called for this operation.
- `FLT_PREOP_SUCCESS_WITH_CALLBACK` indicates the driver allows the filter manager to propagate the request to lower-layer filters, but it wants its post-operation callback invoked for this operation.
- `FLT_PREOP_PENDING` indicates the driver is pending the operation. The filter manager does not continue processing the request until the driver calls `FltCompletePendedPreOperation` to let the filter manager know it can continue processing this request.
- `FLT_PREOP_SYNCHRONIZE` is similar to `FLT_PREOP_SUCCESS_WITH_CALLBACK`, but the driver asks the filter manager to invoke its post-callback on the same thread at `IRQL <= APC_LEVEL` (normally the post-operation callback can be invoked at `IRQL <= DISPATCH_LEVEL` by an arbitrary thread).

The `Data` argument provides all the information related to the I/O operation itself, as a `FLT_CALLBACK_DATA` structure defined like so:

```
typedef struct _FLT_CALLBACK_DATA {
    FLT_CALLBACK_DATA_FLAGS Flags;
    PTHREAD CONST Thread;
    PFILT_IO_PARAMETER_BLOCK CONST Iopb;
    IO_STATUS_BLOCK IoStatus;

    struct _FLT_TAG_DATA_BUFFER *TagData;
    union {
        struct {
            LIST_ENTRY QueueLinks;
            PVOID QueueContext[2];
        };
    };
}
```

```

    PVOID FilterContext[4];
};

KPROCESSOR_MODE RequestorMode;
} FLT_CALLBACK_DATA, *PFLT_CALLBACK_DATA;

```

This structure is also provided in the post-callback. Here is a rundown of the important members of this structure:

- * *Flags* may contain zero or a combination of flags, some of which are listed below:
 - * `FLTFL_CALLBACK_DATA_DIRTY` indicates the driver has made changes to the structure and then called `FltSetCallbackDataDirty`. Every member of the structure can be modified except `Thread` and `RequestorMode`.
 - * `FLTFL_CALLBACK_DATA_FAST_IO_OPERATION` indicates this is a fast I/O operation.
 - * `FLTFL_CALLBACK_DATA_IRP_OPERATION` indicates this is an IRP-based operation.
 - * `FLTFL_CALLBACK_DATA_GENERATED_IO` indicates this is an operation generated by another mini-filter.
 - * `FLTFL_CALLBACK_DATA_POST_OPERATION` indicates this is a post-operation, rather than a pre-operation.

- *Thread* is an opaque pointer to the thread requesting this operation.
- *IoStatus* is the status of the request. A pre-operation can set this value and then indicate the operation is complete by returning `FLT_PREOP_COMPLETE`. A post-operation can look at the final status of the operation.
- *RequestorMode* indicates whether the requestor of the operation is from user mode (`UserMode`) or kernel mode (`KernelMode`).
- *Iopb* is in itself a structure holding the detailed parameters of the request, defined like so:

```

ULONG IrpFlags;
UCHAR MajorFunction;
UCHAR MinorFunction;
UCHAR OperationFlags;
UCHAR Reserved;
PFILE_OBJECT TargetFileObject;
PFLT_INSTANCE TargetInstance;
FLT_PARAMETERS Parameters;
} FLT_IO_PARAMETER_BLOCK, *PFLT_IO_PARAMETER_BLOCK;

```

The useful member of this structure are the following:

- *TargetFileObject* is the file object that is the target of this operation; it's useful to have when invoking some APIs.
- *Parameters* is a monstrous union providing the actual data for the specific information (similar in concept to the *Paramters* member of an `IO_STACK_LOCATION`). The driver looks at the proper structure within this union to get to the information it needs. We'll look at some of these structures once we look at specific operation types, later in this chapter.

The second argument to the pre-callback is another structure of type `FLT RELATED OBJECTS`. This structure mostly contains opaque handles to the current filter, instance and volume, which are useful in some APIs. Here is the complete definition of this structure:

```
typedef struct _FLT RELATED OBJECTS {
    USHORT CONST Size;
    USHORT CONST TransactionContext;
    PFLT_FILTER CONST Filter;
    PFLT_VOLUME CONST Volume;
    PFLT_INSTANCE CONST Instance;
    PFILE_OBJECT CONST FileObject;
    PKTRANSACTION CONST Transaction;
} FLT RELATED OBJECTS, *PFLT RELATED OBJECTS;
```

The *FileObject* field is the same one accessed through the I/O parameter block's `TargetFileObject` field.

The last argument to the pre-callback is a context value that can be set by the driver. If set, this value is propagated to the post-callback routine for the same request (the default value is `NULL`).

Post Operation Callbacks

All post-operation callbacks have the same prototype as follows:

```
FLT_POSTOP_CALLBACK_STATUS SomePostOperation (
    _Inout_ PFLT_CALLBACK_DATA Data,
    _In_ PCFLT RELATED OBJECTS FltObjects,
    _In_opt_ PVOID CompletionContext,
    _In_ FLT_POST_OPERATION_FLAGS Flags);
```

The post-operation function is called at `IRQL <= DISPATCH_LEVEL` in an arbitrary thread context, unless the pre-callback routine returned `FLT_PREOP_SYNCHRONIZE`, in which case the filter manager guarantees the post-callback is invoked at `IRQL < DISPATCH_LEVEL` on the same thread that executed the pre-callback.

In the former case, the driver cannot perform certain types of operations because the IRQL is too high:

- Cannot access paged memory.
- Cannot use kernel APIs that only work at `IRQL < DISPATCH_LEVEL`.
- Cannot acquire synchronization primitives such as mutexes, fast mutexes, executive resources, semaphores, events, etc. (It can acquire spin locks, however.)
- Cannot set, get or delete contexts (see the section “Contexts” later in this chapter), but it can release contexts.

If the driver needs to do any of the above, it somehow must defer its execution to another routine called at `IRQL < DISPATCH_LEVEL`. This can be done in one of two ways:

- The driver calls `FltDoCompletionProcessingWhenSafe` which sets up a callback function that is invoked by a system worker thread at `IRQL < DISPATCH_LEVEL` (if the post-operation was called at `IRQL = DISPATCH_LEVEL`).
- The driver posts a work item by calling `FltQueueDeferredIoWorkItem`, which queues a work item that will eventually execute by a system worker thread at `IRQL = PASSIVE_LEVEL`. In the work item callback, the driver will eventually call `FltCompletePendedPostOperation` to signal the filter manager that the post-operation is complete.

Although using `FltDoCompletionProcessingWhenSafe` is easier, it has some limitations that prevent it from being used in some scenarios:

- Cannot be used for `IRP_MJ_READ`, `IRP_MJ_WRITE` or `IRP_MJ_FLUSH_BUFFERS` because it can cause a deadlock if these operations are completed synchronously by a lower layer.
- Can only be called for IRP-based operations (can check with the `FLT_IS_IRP_OPERATION` macro).



In any case, using one of these deferring mechanisms is not allowed if the flags argument is set to `FLTFL_POST_OPERATION_DRAINING`, which means the post-callback is part of volume detaching. In this case, the post callback is called at `IRQL < DISPATCH_LEVEL`.



Though it seems easy to just return `FLT_PREOP_SYNCHRONIZE` from the pre-callback to have the pos-callback run in a convenient context, it does carry some overhead with it, which the driver may want to avoid if possible.

The post-create operation (`IRP_MJ_CREATE`) is guaranteed to be called by the requesting thread at `IRQL PASSIVE_LEVEL`.

The returned value from the pos-callback is usually `FLT_POSTOP_FINISHED_PROCESSING` to indicate the driver is finished with this operation. However, if the driver needs to perform work in a work item (because of a high `IRQL`, for example), the driver can return `FLT_POSTOP_MORE_PROCESSING_REQUIRED` to tell the filer manager the operation is still pending completion, and in the work item call `FltCompletePendedPostOperation` to let the filter manager know it can continue processing this request.

There are many little details here, check out the WDK documentation for yet more details. We'll use some of the above mechanisms later in this chapter.

File Names

In some mini-filter callbacks, the name of the file being accessed is needed. At first, this seems like an easy detail to find: the FILE_OBJECT structure has a `FileName` member, which should be exactly what is needed.

Unfortunately, things are not that simple. Files may be opened with a full path or a relative one; rename operations on the same file may be occurring at the same time; some file name information is cached. For these and other internal reasons, the `FileName` field in the file object is not be trusted. In fact, it's only guaranteed to be valid in an IRP_MJ_CREATE pre-operation callback, and even there it's not necessarily in the format the driver needs.

To offset this issues, the filter manager provides the `FltGetFileNameInformation` API that can return the correct file name when needed. This function is prototyped as follows:

```
NTSTATUS FltGetFileNameInformation (
    _In_ PFLT_CALLBACK_DATA CallbackData,
    _In_ FLT_FILE_NAME_OPTIONS NameOptions,
    _Outptr_ PFLT_FILE_NAME_INFORMATION *FileNameInformation);
```

The `CallbackData` parameter is the one provided by the filter manager in any callback. The `NameOptions` parameter is a set of flags that specify (among other things) the requested file format. Typical value used by most drivers is `FLT_FILE_NAME_NORMALIZED` (full path name) ORed with `FLT_FILE_NAME_QUERY_DEFAULT` (locate the name in a cache, otherwise query the file system). The result from the call is provided by the last parameter, `FileNameInformation`. The result is an allocated structure that needs to be properly freed by calling `FltReleaseFileNameInformation`.

The `FLT_FILE_NAME_INFORMATION` structure is defined like so:

```
typedef struct _FLT_FILE_NAME_INFORMATION {
    USHORT Size;
    FLT_FILE_NAME_PARSED_FLAGS NamesParsed;
    FLT_FILE_NAME_OPTIONS Format;

    UNICODE_STRING Name;
    UNICODE_STRING Volume;
    UNICODE_STRING Share;
    UNICODE_STRING Extension;
    UNICODE_STRING Stream;
    UNICODE_STRING FinalComponent;
    UNICODE_STRING ParentDir;
} FLT_FILE_NAME_INFORMATION, *PFLT_FILE_NAME_INFORMATION;
```

The main ingredients are the several `UNICODE_STRING` structures that should hold the various components of a file name. Initially, only the `Name` field is initialized to the full file name (depending

on the flags used to query the file name information, “full” may be a partial name). If the request specified the flag `FLT_FILE_NAME_NORMALIZED`, then `Name` points to the full path name, in device form. *Device form* means that file such as `c:\mydir\myfile.txt` is stored with the internal device name to which “C:” maps to, such as `\Device\HarddiskVolume3\mydir\myfile.txt`. This makes the driver’s job a bit more complicated if it somehow depends on paths provided by user mode (more on that later).



The driver should never modify this structure, because the filter manager sometimes caches it for use with other drivers.

Since only the full name is provided by default (`Name` field), it’s often necessary to split the full path to its constituents. Fortunately, the filter manager provides such a service with the `FltParseFileNameInformation` API. This one takes the `FLT_FILE_NAME_INFORMATION` object and fills in the other `UNICODE_STRING` fields in the structure.

Note that `FltParseFileNameInformation` does not allocate anything. It just sets each `UNICODE_STRING`’s `Buffer` and `Length` to point to the correct parts in the full `Name` field. This means there is no “unparse” function and it’s not needed.

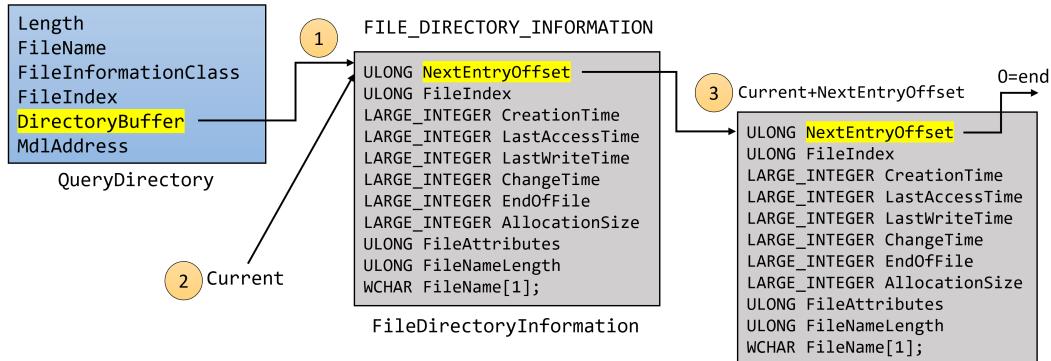


In scenarios where a simple C string is available for a full path, the simpler (and weaker) function `FltParseFileName` can be used for getting easy access to the file extension, stream and final component. It can also be used outside the scope of file system mini-filters.

File Name Parts

As can be seen from `FLT_FILE_NAME_INFORMATION` declaration, there are several components that make up a full file name. Here is an example for the local file “`c:\mydir1\mydir2\myfile.txt`”:

The volume is the actual device name for which the symbolic link “C:” maps to. Figure 12-8 shows `WinObj` showing the `C:` symbolic link and its target, which is `\Device\HarddiskVolume3` on that machine.

Figure 12-8: Driver Mapping in *WinObj*

The share string is empty for local files (**Length** is zero). **ParentDir** is set to the directory only. In our example that would be `\mydir1\mydir2\` (not the trailing backslash). The extension is just that, the file extension. In our example this is `.txt`.

The **FinalComponent** field stores the file name and stream name (if not using the default stream). For our example, it would be `myfile.txt`.

The **Stream** component bares some explanation. Some file systems (most notable NTFS) provide the ability to have multiple data “streams” in a single file. Essentially, this means several files can be stored into a single “physical” file. In NTFS, for instance, what we typically think of as a file’s data is in fact one of its streams named “\$DATA”, which is considered the default stream. But it’s possible to create/open another stream, that is stored in the same file, so to speak. Tools such as Windows Explorer do not look for these streams, and the sizes of any alternate streams are not shown or returned by standard APIs such as `GetFileSize`. Stream names are specified with a colon after the file name before the stream name itself. For example, the file name “myfile.txt:mystream” points to an alternate stream named “mystream” within the file “myfile.txt”. Alternate streams can be created with the command interpreter as the following example shows:

```
C:\temp>echo hello > hello.txt:mystream
```

```
C:\Temp>dir hello.txt
Volume in drive C is OS
Volume Serial Number is 1707-9837
```

```
Directory of C:\Temp
```

22-May-19 11:33	0	hello.txt
1 File(s)	0	bytes

Notice the zero size of the file. Is the data really in there? Trying to use the `type` command fails:

```
C:\Temp>type hello.txt:mystream
```

The filename, directory name, or volume label syntax is incorrect.

The type command interpreter does not recognize stream names. We can use the *SysInternals* tool *Streams.exe* to list the names and sizes of alternate streams in files. Here is the command with our *hello.txt* file:

```
C:\Temp>streams -nobanner hello.txt
```

```
C:\Temp\hello.txt:
```

```
:mystream:$DATA 8
```

The alternate stream content is not shown. To view (and optionally export to another file) the stream's data, we can use a tool called *NtfsStreams* available on my Github *AllTools* repository. Figure 12-7 shows *NtfsStreams* opening the *hello.txt* file from the previous example. We can clearly see stream's size and data.

The “\$DATA” shown is the stream type, where \$DATA is the normal data stream (there are other predefined stream types). Custom stream types are specifically used in reparse points (beyond the scope of this book).



Figure 12-7: Alternate Streams in *NtfsStreams*

Of course alternate streams can be created programmatically by passing the stream name at the end of the filename after a colon, to the *CreateFile* API. Here is an example (error handling omitted):

```

HANDLE hFile = ::CreateFile(L"c:\\temp\\myfile.txt:stream1",
    GENERIC_WRITE, 0, nullptr, OPEN_ALWAYS, 0, nullptr);

char data[] = "Hello, from a stream";
DWORD bytes;
::WriteFile(hFile, data, sizeof(data), &bytes, nullptr);
::CloseHandle(hFile);

```

Streams can also be deleted normally with DeleteFile and can be enumerated (this is what streams.exe and nftsstreams.exe do) with FindFirstStream and FileNextStream.

RAII FLT_FILE_NAME_INFORMATION wrapper

As discussed in the previous section, calling FltGetFileNameInformation requires calling its opposite function, FltReleaseFileNameInformation. This naturally leads to the possibility of creating a RAII wrapper to take care of this, making the surrounding code simpler and less error prone. Here is one possible declaration for such a wrapper:

```

enum class FileNameOptions {
    Normalized = FLT_FILE_NAME_NORMALIZED,
    Opened      = FLT_FILE_NAME_OPENED,
    Short       = FLT_FILE_NAME_SHORT,

    QueryDefault      = FLT_FILE_NAME_QUERY_DEFAULT,
    QueryCacheOnly    = FLT_FILE_NAME_QUERY_CACHE_ONLY,
    QueryFileSystemOnly = FLT_FILE_NAME_QUERY_FILESYSTEM_ONLY,

    RequestFromCurrentProvider = FLT_FILE_NAME_REQUEST_FROM_CURRENT_PROVIDER,
    DoNotCache        = FLT_FILE_NAME_DO_NOT_CACHE,
    AllowQueryOnReparse = FLT_FILE_NAME_ALLOW_QUERY_ON_REPARSE
};

DEFINE_ENUM_FLAG_OPERATORS(FileNameOptions);

struct FilterFileNameInformation {
    FilterFileNameInformation(PFLT_CALLBACK_DATA data, FileNameOptions options \
= FileNameOptions::QueryDefault | FileNameOptions::Normalized);
    ~FilterFileNameInformation();
}

```

```

operator bool() const {
    return _info != nullptr;
}

operator PFLT_FILE_NAME_INFORMATION() const {
    return Get();
}

PFLT_FILE_NAME_INFORMATION operator->() {
    return _info;
}

NTSTATUS Parse();

private:
    PFLT_FILE_NAME_INFORMATION _info;
};

```

The non-inline functions are defined below:

```

FilterFileNameInformation::FilterFileNameInformation(
    PFLT_CALLBACK_DATA data, FileNameOptions options) {
    auto status = FltGetFileNameInformation(data,
        (FLT_FILE_NAME_OPTIONS)options, &_info);
    if (!NT_SUCCESS(status))
        _info = nullptr;
}

FilterFileNameInformation::~FilterFileNameInformation() {
    if (_info)
        FltReleaseFileNameInformation(_info);
}

NTSTATUS FilterFileNameInformation::Parse() {
    return FltParseFileNameInformation(_info);
}

```

Using this wrapper can be something like the following:

```

FilterFileNameInformation nameInfo(Data);
if(nameInfo) { // operator bool()
    if(NT_SUCCESS(nameInfo.Parse())) {
        KdPrint(("Final component: %wZ\n", &nameInfo->FinalComponent));
    }
}

```

The Delete Protector Driver

it's time to put some of the information discussed so far into an actual driver. The driver we'll create will be able to protect certain files from deletion. We'll start by creating a new *Empty WDM Filter* project named *KDelProtect* (or another name of your choosing). Then we'll delete the INF file, since we are going to use the code presented earlier in this chapter to properly "register" the driver.

The main question we need to answer is: how does a file deletion manifested in a file system (and mini-filter)?

It turns out there are two way to delete a file. One way is to use IRP_MJ_SET_INFORMATION operation. This major function code provides a bag of operations, delete being one of them. Sending this request to a driver can be done by the user-mode APIs such as SetFileInformationByHandle and kernel APIs such as NtSetInformationFile. The second way to delete a file (and in fact the most common) is to open the file with the FILE_DELETE_ON_CLOSE option flag. The file then is deleted as soon as the last handle to it is closed.

This flag can be set from user mode in CreateFile with FILE_FLAG_DELETE_ON_CLOSE as one of the flags (second to last argument). The higher level function DeleteFile uses the same flag behind the scenes.

For our driver, we want to support both options to cover all our bases. The driver will protect files with client-defined extensions against deletion. A client can request to set a list of extensions, which means we also nee a "standard" device object (as we created many times before), sometimes reffered to as *Control Device Object* (CDO).

We'll start by adding a *Driver.h* file to contain private driver data. This file looks like the following:

```

#pragma once

#include "ExecutiveResource.h"

struct FilterState {
    PFLT_FILTER Filter;
    UNICODE_STRING Extentions;
    ExecutiveResource Lock;
};

```

```
PDRIVER_OBJECT DriverObject;
};

extern FilterState g_State;
```

The `Filter` member will hold the mini-filter registration handle. Extensions will hold the list of extensions we must protect from deletion - the format of that will be described later. Finally, any changes to the extensions list requires synchronization, so an Executive Resource is used (with a RAII wrapper that we saw in chapter 6). Since most of the time the extension list is read (rather than written), an Executive Resource is the best synchronization primitive to use.

Why do we need a driver object pointer stored in `FilterState`? This will become clear when we implement the driver's unload functionality.

Given the above declararion, we can create a global instance of the `FilterState` structure, initialize it, and proceed to create the CDO and a symbolic link. Here is the complete `DriverEntry` (in the file named *Driver.cpp*), with some `KdPrint` omitted for brevity:

```
FilterState g_State;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    auto status = g_State.Lock.Init();
    if (!NT_SUCCESS(status))
        return status;

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\""\?\?\Device\DelProtect");
    PDEVICE_OBJECT devObj = nullptr;
    bool symLinkCreated = false;
    do {
        status = InitMiniFilter(DriverObject, RegistryPath);
        if (!NT_SUCCESS(status)) {
            KdPrint((DRIVER_PREFIX "Failed to init mini-filter (0x%X)\n", status));
            break;
        }

        UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\DelProtect");
        status = IoCreateDevice(DriverObject, 0, &devName, FILE_DEVICE_UNKNOWN, 0, FALSE, &devObj);
        if (!NT_SUCCESS(status))
            break;

        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status))
```

```

        break;
symLinkCreated = true;

status = FltStartFiltering(g_State.Filter);
if (!NT_SUCCESS(status))
    break;
} while (false);

if (!NT_SUCCESS(status)) {
    g_State.Lock.Delete();
    if(g_State.Filter)
        FltUnregisterFilter(g_State.Filter);
    if (symLinkCreated)
        IoDeleteSymbolicLink(&symLink);
    if (devObj)
        IoDeleteDevice(devObj);
    return status;
}

g_State.DriverObject = DriverObject;

DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = OnCreateClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = OnDeviceControl;

return status;
}

```

The `InitMiniFilter` call is used to register the mini-filter. It's implemented in the `MiniFilter.cpp` file, to make the driver pieces more “managaeable” - not everything is in the same file. If the mini-filter is initialized successfully (and all other initializations succeed as well), the call to `FltStartFiltering` starts the mini-filter action.

Let's examine the initialization in `InitMiniFilter`. The first step is to initialize the “extensions” we protect. For demonstration and testing purposes we'll initialize it to a “PDF” extension. This is an arbitrary choice, but it allows easy testing of the driver even before we implement the client-facing functionality that allows changing the extensions being protected:

```

NTSTATUS
InitMiniFilter(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    WCHAR ext[] = L"PDF;";
    g_State.Extentions.Buffer = (PWSTR)ExAllocatePool2(POOL_FLAG_PAGED,
        sizeof(ext), DRIVER_TAG);
    if (g_State.Extentions.Buffer == nullptr)
        return STATUS_NO_MEMORY;

    memcpy(g_State.Extentions.Buffer, ext, sizeof(ext));
    g_State.Extentions.Length = g_State.Extentions.MaximumLength = sizeof(ext);
}

```

The string is allocated dynamically for consistency: if a client modifies the extensions later, the driver will free the existing string and then allocate a new one. To make it easier to work with multiple protected extensions, I decided to keep a single string in memory with the extensions stored in uppercase and separated by semicolons. For example, the string “PDF;DOCX;” indicates protecting PDF and DOCX files from deletion.

The next piece of code writes the correct Registry entries for the `FltRegisterFilter` to have a chance of success. The code is shown in the section “Installation”, earlier in this chapter, so I will not repeat it here. After the Registry values are written the filter can be registered. We have to prepare an array of callback structures based on what we need to support - namely `IRP_MJ_CREATE` (check for files opened with the “delete-on-close” flag), and `IRP_MJ_SET_INFORMATION` (if a file is deleted explicitly):

```

FLT_OPERATION_REGISTRATION const callbacks[] = {
    { IRP_MJ_CREATE, 0, DelProtectPreCreate, nullptr },
    { IRP_MJ_SET_INFORMATION, 0, DelProtectPreSetInformation, nullptr },
    { IRP_MJ_OPERATION_END }
};

}

```

We need pre-operations only, as our purpose is to prevent delete operations. Post-operations don’t make sense, as the “deed is already done” at that point. Now the main registration structure and the registration itself:

```

FLT_REGISTRATION const reg = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0, // Flags
    nullptr, // Context
    callbacks, // Operation callbacks
    DelProtectUnload, // MiniFilterUnload
    DelProtectInstanceSetup, // InstanceSetup
    DelProtectInstanceQueryTeardown, // InstanceQueryTeardown
    DelProtectInstanceTeardownStart, // InstanceTeardownStart
    DelProtectInstanceTeardownComplete, // InstanceTeardownComplete
}

```

```
};

status = FltRegisterFilter(DriverObject, &reg, &g_State.Filter);
```

The `DelProtectInstanceSetup` callback is where the the mini-filter decides (for each volume) to attach or to skip. In this example, let's decide to attach to NTFS volumes only:

```
NTSTATUS
DelProtectInstanceSetup(
    PCFLT RELATED OBJECTS FltObjects, FLT_INSTANCE_SETUP_FLAGS Flags,
    DEVICE_TYPE VolumeDeviceType, FLT_FILESYSTEM_TYPE VolumeFilesystemType) {
    UNREFERENCED_PARAMETER(FltObjects);
    UNREFERENCED_PARAMETER(Flags);
    UNREFERENCED_PARAMETER(VolumeDeviceType);

    return VolumeFilesystemType == FLT_FSTYPE_NTFS
        ? STATUS_SUCCESS : STATUS_FLT_DO_NOT_ATTACH;
}
```

`STATUS_FLT_DO_NOT_ATTACH` indicates the filter does not wish to attach to this volume, while `STATUS_SUCCESS` indicates that it does. Using the file system type is one way to make a decision, where the `VolumeDeviceType` is another. Consult the docs for the details.

The mini-filter unload callabck is where the mini-filter is unregistered. The driver *should not* add a normal unload routine by setting the `DriverUnload` member of the `DRIVER_OBJECT`. The reason is that the filter manager takes control of this callback. If you set it after `FltRegisterFilter`, some cleanup won't happen. If you set it before, it would simply be overridden by `FltRegisterFilter`. In summary, this is where our cleanup is done:

```
NTSTATUS DelProtectUnload(FLT_FILTER_UNLOAD_FLAGS Flags) {
    UNREFERENCED_PARAMETER(Flags);

    FltUnregisterFilter(g_State.Filter);
    g_State.Lock.Delete();
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\\""\?\?\DelProtect");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(g_State.DriverObject->DeviceObject);

    return STATUS_SUCCESS;
}
```

The remaining instance-related callbacks simply return `STATUS_SUCCESS`, but can be customized if desired.

Handling Pre-Create

The pre-create callback has the job to look for a file opened with the “delete-on-close” flag. The function itself has the same prototype like all pre-operation callbacks. It starts by not blocking kernel callers:

```
FLT_PREOP_CALLBACK_STATUS
DelProtectPreCreate(PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects, PVOID*) {
    UNREFERENCED_PARAMETER(FltObjects);

    if (Data->RequestorMode == KernelMode)
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Allowing kernel callers to move forward regardless is not mandatory of course, but in most cases we don’t want to prevent kernel code from doing work that may be necessary.

Next we need to check if the flag FILE_DELETE_ON_CLOSE exists in the creation request. The structure to look at is the Create field under the Parameters inside Iopb as follows:

```
const auto& params = Data->Iopb->Parameters.Create;
if (params.Options & FILE_DELETE_ON_CLOSE) {
    // delete flag
}
```

The above *params* variable references the Create structure defined like so:

```
struct {
    PIO_SECURITY_CONTEXT SecurityContext;
    //
    // The low 24 bits contains CreateOptions flag values.
    // The high 8 bits contains the CreateDisposition values.
    //
    ULONG Options;

    USHORT POINTER_ALIGNMENT FileAttributes;
    USHORT ShareAccess;
    ULONG POINTER_ALIGNMENT EaLength;

    PVOID EaBuffer;           //Not in IO_STACK_LOCATION parameters list
    LARGE_INTEGER AllocationSize; //Not in IO_STACK_LOCATION parameters list
} Create;
```

Generally, for any I/O operation, the documentation must be consulted to understand what's available and how to use it. In our case, the Options field is a combination of flags documented under the FltCreateFile function (which we'll use later in this chapter in an unrelated context). The code checks to see if this flag exists, and if so, it means a delete operation is being initiated.

If the file is opened for deletion, we need to examine the file name and check if its extension is one that we protect. If true, we have to fail the request. Here is the code:

```
auto status = FLT_PREOP_SUCCESS_NO_CALLBACK;

if (params.Options & FILE_DELETE_ON_CLOSE) {
    auto filename = &FltObjects->FileObject->FileName;
    KdPrint(("Delete on close: %wZ\n", filename));

    if (!IsDeleteAllowed(filename)) {
        Data->IoStatus.Status = STATUS_ACCESS_DENIED;
        status = FLT_PREOP_COMPLETE;
        KdPrint(("(Pre Create) Prevent deletion of %wZ\n", filename));
    }
}
return status;
}
```

The file name can be obtained by directly examining the file object - this is only allowed for a pre-create operation callback, which is exactly the callback we're in. In all other cases, FltGetFileNameInformation is the way to go.

IsDeleteAllowed is a private driver function to extract the extension and compare it to the list of extinctions the driver maintains:

```
bool IsDeleteAllowed(PCUNICODE_STRING filename) {
    UNICODE_STRING ext;
    if (NT_SUCCESS(FltParseFileName(filename, &ext, nullptr, nullptr))) {
        WCHAR uext[16] = { 0 };
        UNICODE_STRING suext;
        suext.Buffer = uext;
        //
        // save space for NULL terminator and a semicolon
        //
        suext.MaximumLength = sizeof(uext) - 2 * sizeof(WCHAR);
        RtlUpcaseUnicodeString(&suext, &ext, FALSE);
        RtlAppendUnicodeToString(&suext, L";");

        //
        // search for the prefix
    }
}
```

```

    //
    return wcsstr(g_State.Extensions.Buffer, uext) == nullptr;
}
return true;
}

```

The function starts by calling `FltParseFileName` to extract the extension. You may be thinking that getting to the extension should be fairly easy by calling something like `wcsrchr`, looking for a dot. However, if the file has a custom NTFS stream name, then finding the end of the extension would require looking for a colon - not too complex, but why bother when there exists an API that does the heavy lifting? Here is the prototype of `FltParseFileName`:

```

NTSTATUS FltParseFileName (
    _In_ PCUNICODE_STRING FileName,
    _Inout_opt_ PUNICODE_STRING Extension,
    _Inout_opt_ PUNICODE_STRING Stream,
    _Inout_opt_ PUNICODE_STRING FinalComponent);

```

The input is a `UNICODE_STRING`, with 3 outputs, all of them optional. This API does not allocate anything - it simply points the `UNICODE_STRING` objects to the `FileName`. We just need the extension, so the other arguments can be set to NULL.

The rest of the code does some juggling to convert the extension to uppercase (`RtlUpcaseUnicodeString`) so that `wcsstr` can be used to search for the extension in the `Extensions` member we maintain inside the `FilterState` structure. If the extension is not found (`wcsstr` returns NULL), the function returns true to indicate file deletion is allowed.

Handling Pre-Set Information

We are now ready to implement the pre-set information callback to cover our bases, so to speak, with the second way file deletion is implemented by file systems. We'll start by ignoring kernel callers as with `IRP_MJ_CREATE`:

```

FLT_PREOP_CALLBACK_STATUS DelProtectPreSetInformation(
    PFLT_CALLBACK_DATA Data, PCFLT RELATED OBJECTS FltObjects, PVOID*)
{
    UNREFERENCED_PARAMETER(FltObjects);

    if (Data->RequestorMode == KernelMode)
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
}

```

Since `IRP_MJ_SET_INFORMATION` is the way to do several types of operations, we need to check if this is in fact a delete operation. The driver must first access the proper structure in the `FLT_PARAMETERS` union, declared like so:

```

struct {
    ULONG Length;
    FILE_INFORMATION_CLASS *FileInformationClass;
    PFILE_OBJECT ParentOfTarget;
    union {
        struct {
            BOOLEAN ReplaceIfExists;
            BOOLEAN AdvanceOnly;
        };
        ULONG ClusterCount;
        HANDLE DeleteHandle;
    };
    PVOID InfoBuffer;
} SetFileInformation;

```

`FileInformationClass` indicates which type of operation this instance represents and so we need to check whether this is a delete operation:

```

auto status = FLT_PREOP_SUCCESS_NO_CALLBACK;
auto& params = Data->Iopb->Parameters.SetFileInformation;
if (params.FileInformationClass == FileDispositionInformation ||
    params.FileInformationClass == FileDispositionInformationEx) {

```

The `FileDispositionInformation` enumeration value indicates a delete operation. The `FileDispositionInformationEx` is similar, slightly extended, available in Windows 10 version 1607 and later.

If it is a delete operation, there is yet another check to do, by looking at the information buffer which is of type `FILE_DISPOSITION_INFORMATION(Ex)` for delete operations and checking the boolean flags stored there. Here are the structures and the relevant flag for the extended one:

```

typedef struct _FILE_DISPOSITION_INFORMATION {
    BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION, *PFILE_DISPOSITION_INFORMATION;

#define FILE_DISPOSITION_DELETE 0x00000001

typedef struct _FILE_DISPOSITION_INFORMATION_EX {
    ULONG Flags;
} FILE_DISPOSITION_INFORMATION_EX;

```

Checking for a value of one covers both cases well-enough:

```
auto info = (FILE_DISPOSITION_INFORMATION*)params.InfoBuffer;
if (info->DeleteFile & 1) {    // also covers FileDispositionInformationEx Flags
```

The next step is to check the file extension that is about to be deleted. Since this is not a pre-create callback, we must use FltGetFileNameInformation to get the file name, and then call IsDeleteAllowed as before:

```
PFLT_FILE_NAME_INFORMATION fi;
// 
// using FLT_FILE_NAME_NORMALIZED is important here for parsing purposes
//
if (NT_SUCCESS(FltGetFileNameInformation(
    Data, FLT_FILE_NAME_QUERY_DEFAULT | FLT_FILE_NAME_NORMALIZED, &fi))) {
    if (!IsDeleteAllowed(&fi->Name)) {
        Data->IoStatus.Status = STATUS_ACCESS_DENIED;
        KdPrint(("(Pre Set Information) Prevent deletion of %wZ\n",
            &fi->Name));
        status = FLT_PREOP_COMPLETE;
    }
    FltReleaseFileNameInformation(fi);
}
```

Now we can test the complete driver - we'll find that files of the selected extensions cannot be deleted. Here is an example command sequence once the driver is installed and PDF files are supposed to be protected:

```
c:\temp>dir
10/19/2022  01:13 PM    <DIR>          .
05/28/2022  01:09 PM    <DIR>          Test
10/19/2022  10:41 AM          5 hello1.pdf
10/19/2022  10:41 AM          5 hello2.txt
10/19/2022  10:41 AM          5 hello3.txt
```

```
C:\Temp>del hello2.txt
```

```
C:\Temp>del hello1.pdf
Access is denied.
```

DelProtect Configuration

Now that we have the basic driver working, we can add support for custom extensions. The driver can define a control code to be shared with user mode clients, defined in *DelProtectPublic.h*:

```
#define DEVICE_DELPROTECT 0x8009

#define IOCTL_DELPROTECT_SET_EXTENSIONS CTL_CODE( \
    DEVICE_DELPROTECT, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

The driver's IRP_MJ_DEVICE_CONTROL doesn't have anything we didn't see before. Here is its complete code:

```
NTSTATUS OnDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto& dic = irpSp->Parameters.DeviceIoControl;
    auto len = 0U;

    switch (dic.IoControlCode) {
        case IOCTL_DELPROTECT_SET_EXTENSIONS:
            auto ext = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
            auto inputLen = dic.InputBufferLength;
            if (ext == nullptr ||
                inputLen < sizeof(WCHAR) * 2 ||
                ext[inputLen / sizeof(WCHAR) - 1] != 0) {
                status = STATUS_INVALID_PARAMETER;
                break;
            }
            if (g_State.Extentions.MaximumLength <
                inputLen - sizeof(WCHAR)) {
                //
                // allocate a new buffer to hold the extensions
                //
                auto buffer = ExAllocatePool2(POOL_FLAG_PAGED,
                    inputLen, DRIVER_TAG);
                if (buffer == nullptr) {
                    status = STATUS_INSUFFICIENT_RESOURCES;
                    break;
                }
                g_State.Extentions.MaximumLength = (USHORT)inputLen;
                //
                // free the old buffer
                //
                ExFreePool(g_State.Extentions.Buffer);
                g_State.Extentions.Buffer = (PWSTR)buffer;
            }
            UNICODE_STRING ustr;
```

```

        Rt1InitUnicodeString(&ustr, ext);
        //
        // make sure the extensions are uppercase
        //
        Rt1UppercaseUnicodeString(&ustr, &ustr, FALSE);
        memcpy(g_State.Extentions.Buffer, ext, len = inputLen);
        g_State.Extentions.Length = (USHORT)inputLen;
        status = STATUS_SUCCESS;
        break;
    }
    return CompleteRequest(Irp, status, len);
}

```

Testing the Modified Driver

Earlier, we tested the driver by deleting files using *cmd.exe*, but that may not be generic enough, so we better create our own test application. There are three ways to delete a file with user mode APIs:

1. Call the `DeleteFile` function.
2. Call `CreateFile` with the flag `FILE_FLAG_DELETE_ON_CLOSE`.
3. Call `SetFileInformationByHandle` on an open file.

Internally, there are only two ways to delete a file - `IRP_MJ_CREATE` with the `FILE_DELETE_ON_CLOSE` flag and `IRP_MJ_SET_INFORMATION` with `FileDispositionInformation`. Clearly, in the above list, item (2) corresponds to the first option and item (3) corresponds to the second option. The only mystery left is `DeleteFile` - how does it delete a file?

From the driver's perspective it does not matter at all, since it must map to one of the two options the driver handles.

We'll create a console application project named *DelTest*, for which the usage text should be something like this:

```
c:\book>deltest
Usage: deltest.exe <method> <filename>
      Method: 1=DeleteFile, 2=delete on close, 3=SetFileInformation.
```

Let's examine the user mode code for each of these methods (assuming `filename` is a variable pointing to the file name provided in the command line).

Using `DeleteFile` is trivial:

```
BOOL success = DeleteFile(filename);
```

Opening the file with the delete-on-close flag can be achieved with the following:

```
HANDLE hFile = CreateFile(filename, DELETE, 0, nullptr, OPEN_EXISTING,
    FILE_FLAG_DELETE_ON_CLOSE, nullptr);
CloseHandle(hFile);
```

When the handle is closed, the file should be deleted (if the driver does not prevent it!)

Lastly, using `SetFileInformationByHandle`:

```
FILE_DISPOSITION_INFO info;
info.DeleteFile = TRUE;
HANDLE hFile = CreateFile(filename, DELETE, 0, nullptr,
    OPEN_EXISTING, 0, nullptr);
BOOL success = SetFileInformationByHandle(hFile, FileDispositionInfo,
    &info, sizeof(info));
CloseHandle(hFile);
```

The Directory Hiding Driver

The next driver we'll look at is more complex than the *DelProtect* driver. The Directory Hiding driver will hide a directory from the file system, making it not just inaccessible, but also “un-listable” - it will not be visible in directory listings (via the `dir` shell command, *File Explorer*, or whatever). We'll implement the driver in two phases. In the first phase, we'll make a directory (or directories) of choice inaccessible. In the second phase, we'll make it invisible.

Managing Directories

For the purpose of this driver, we'll hold on to a list of directories which should be hidden. This list can be implemented in several ways, such as the linked-lists we have used in previous drivers. To make it more interesting, we'll use a dynamic array of string objects, both of which are part of the *Kernel Template Library* (KTL), described in Appendix A, and available as part of the book's downloads. The idea is to build a reusable library, containing many of the expected types and functions as are available in the user-mode standard C++ library. The KTL is not nearly as broad as the C++ STL, and it's not supposed to be. What it should be, is convenient reusable code for use in driver projects.

To start off, we'll create an *Empty WDM Driver* project, as before, named *KHide*. The driver's state is going to be stored in the following structure declared in *MiniFilter.h*:

```
#include <ktl.h>

struct FilterState {
    FilterState();
    ~FilterState();

    PFLT_FILTER Filter;
    Vector<WString<PoolType::NonPaged>, PoolType::NonPaged> Files;
    ExecutiveResource Lock;
    PDRIVER_OBJECT DriverObject;
};

extern FilterState* g_State;
```

The *ktl.h* header contains all the #includes from other headers, also parts of the KTL. The `FilterState` structure has a default constructor and a destructor, which means we cannot create a global variable of that type and expect the constructor to be called (it won't). Instead, we'll use dynamic allocation to create an instance, which will force calling the constructor. The KTL has overloads for the `new` and `delete` operators.

The members include an Executive Resource (a RAII wrapper over the corresponding kernel object), a mini-filter handle, and a `Vector` of `WString`s. A `WString` is a null-terminated, Unicode string, automatically managed, with a convenient API. The `Vector` class is a templated type for holding a dynamic array of any type, used with a `WString` here. Both types require the pool type to use internally provided with the `PoolType` enumeration, which wraps the flags `POOL_FLAGS`, normally used with `ExAllocatePool2`:

```
enum class PoolType : ULONG64 {
    Paged = POOL_FLAG_PAGED,
    NonPaged = POOL_FLAG_NON_PAGED,
    NonPagedExecute = POOL_FLAG_NON_PAGED_EXECUTE,
    CacheAligned = POOL_FLAG_CACHE_ALIGNED,
    Uninitialized = POOL_FLAG_CACHE_ALIGNED,
    ChargeQuota = POOL_FLAG_USE_QUOTA,
    RaiseOnFailure = POOL_FLAG_RAISE_ON_FAILURE,
    Session = POOL_FLAG_SESSION,
    SpecialPool = POOL_FLAG_SPECIAL_POOL,
};

DEFINE_ENUM_FLAG_OPERATORS(PoolType);
```



A comprehensive coverage of the KTL is in *Appendix A*.

The constructor of FilterState should initialize the Executive Resource, while the destructor should delete it:

```
FilterState::FilterState() {
    Lock.Init();
    Filter = nullptr;
}

FilterState::~FilterState() {
    Lock.Delete();
}
```

The Vector will initialize itself in its default constructor (to an empty vector).

The DriverEntry function should be mostly familiar, using the same kind of code as the *DelProtect* driver for initializing the file system mini-filter, and creating a CDO to allow managing directories to hide. Here is the complete implementation (with some KdPrint calls removed):

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    g_State = new (PoolType::NonPaged) FilterState;
    if (!g_State)
        return STATUS_NO_MEMORY;

    PDEVICE_OBJECT devObj = nullptr;
    NTSTATUS status;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\Device\Hide");
    bool symLinkCreated = false;
    do {
        status = InitMiniFilter(DriverObject, RegistryPath);
        if (!NT_SUCCESS(status))
            break;

        UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\Hide");
        status = IoCreateDevice(DriverObject, 0, &devName,
                               FILE_DEVICE_UNKNOWN, 0, FALSE, &devObj);
        if (!NT_SUCCESS(status))
            break;

        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status))
            break;
        symLinkCreated = true;
```

```

        status = FltStartFiltering(g_State->Filter);
        if (!NT_SUCCESS(status))
            break;
    } while (false);

if (!NT_SUCCESS(status)) {
    if (g_State->Filter)
        FltUnregisterFilter(g_State->Filter);
    if (symLinkCreated)
        IoDeleteSymbolicLink(&symLink);
    if (devObj)
        IoDeleteDevice(devObj);
    if (g_State)
        delete g_State;
    return status;
}

g_State->DriverObject = DriverObject;
DriverObject->MajorFunction[IRP_MJ_CREATE] =
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = OnCreateClose;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = OnDeviceControl;

//  

// for testing purposes  

//  

#if DBG
    g_State->Files.Add(L"c:\\Temp");
#endif
return status;
}

```

The last line before the `return` statement adds an example directory (`c:\temp`) to make it easier to test the driver without the need to add a client, implement `IRP_MJ_DEVICE_CONTROL`, etc.

Initializing and registering the driver as a min-filter is very similar to the *DelProtect* driver. The operation we're concerned with is `IRP_MJ_DIRECTORY_CONTROL`, which is called when directory information is required by a client. Here is the registration code (in *MiniFilter.cpp*):

```

FLT_OPERATION_REGISTRATION const callbacks[] = {
    { IRP_MJ_DIRECTORY_CONTROL, 0, OnPreDirectoryControl, nullptr },
    { IRP_MJ_OPERATION_END }
};

FLT_REGISTRATION const reg = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0,                                // Flags
    nullptr,                           // Context
    callbacks,                         // Operation callbacks
    HideUnload,                        // MiniFilterUnload
    HideInstanceSetup,                 // InstanceSetup
    HideInstanceQueryTeardown,         // InstanceQueryTeardown
    HideInstanceTeardownStart,         // InstanceTeardownStart
    HideInstanceTeardownComplete,      // InstanceTeardownComplete
};
status = FltRegisterFilter(DriverObject, &reg, &g_State->Filter);

```

This driver only requires a single operation to intercept, and a single pre-callback for the first phase of the implementation.

Phase 1: Prevent Access

All we need to do is implement the IRP_MJ_DIRECTORY_CONTROL pre-operation callback. The first order of business is to allow kernel callers (no questions asked). Second, IRP_MJ_DIRECTORY_CONTROL has actually three minor function codes, only one of which we care about in this driver: IRP_MN_QUERY_DIRECTORY, IRP_MN_NOTIFY_CHANGE_DIRECTORY, and IRP_MN_NOTIFY_CHANGE_DIRECTORY_EX. As you probably have guessed, IRP_MN_QUERY_DIRECTORY is all we care about:

```

FLT_PREOP_CALLBACK_STATUS
OnPreDirectoryControl(PFLT_CALLBACK_DATA Data, PCFLT RELATED OBJECTS, PVOID*) {
    if (Data->RequestorMode == KernelMode ||
        Data->Iopb->MinorFunction != IRP_MN_QUERY_DIRECTORY)
        return FLT_PREOP_SUCCESS_NO_CALLBACK;
}

```

We expect the client to provide directory names from the usual user-mode vantage point using drive letters (often referred to as *DOS paths*), such as *c:\temp*. The kernel, however, provides names which are in *device form* (e.g. *\Device\HarddiskVolume4\Temp*). We can convert the user-provided paths to device form before storing them in the vector, or convert the device form path received from the filter manager to a DOS path. We'll take the latter approach in this driver (for versatility).

The term “DOS path” is a historic one, because of the “drive-colon” format used originally in DOS (*Disk Operating System*).

There are a few ways we could use to convert the device path to a DOS path. Probably the simplest option is the API `IoQueryFileDosDeviceName`:

```
NTSTATUS IoQueryFileDosDeviceName(
    _In_ PFILE_OBJECT FileObject,
    _Out_ POBJECT_NAME_INFORMATION *ObjectNameInformation);
```

It requires a `FILE_OBJECT` and returns a `POBJECT_NAME_INFORMATION`, filling it with the name. The latter structure is just a glorified `UNICODE_STRING`:

```
typedef struct _OBJECT_NAME_INFORMATION {
    UNICODE_STRING Name;
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;
```

The data is allocated dynamically and must be freed by calling `ExFreePool`.

At this point, we have the directory that needs querying - let’s convert it to a DOS path so we can easily compare it to our stored directory list:

```
POBJECT_NAME_INFORMATION nameInfo;
if (!NT_SUCCESS(IoQueryFileDosDeviceName(FltObjects->FileObject, &nameInfo)))
    return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Now we can acquire the Executive Resource in shared mode (just reading data), and compare the directory to any one in the list. If found, we can fail the request:

```
UNICODE_STRING path;
auto status = FLT_PREOP_SUCCESS_WITH_CALLBACK;
{
    SharedLocker locker(g_State->Lock);
    for (auto& name : g_State->Files) {
        name.GetUnicodeString(path);
        if (RtlEqualUnicodeString(&path, &nameInfo->Name, TRUE)) {
            //
            // found directory. fail request
            //
            Data->IoStatus.Status = STATUS_NOT_FOUND;
        }
    }
}
```

```

        Data->IoStatus.Information = 0;
        status = FLT_PREOP_COMPLETE;
        break;
    }
}
}
ExFreePool(nameInfo);
return status;
}

```

The SharedLocker class is a RAII wrapper around acquiring/releasing a shared lock of an Executive Resource. The Vector class is used here with the *range-based for* feature of C++ 11 (and later). This works, because Vector implements a begin and end methods (see *Appendix A* for more information). A UNICODE_STRING is initialized to prepare calling Rt1EqualUnicodeString which allows comparing two UNICODE_STRING objects for equality, optionally without case sensitivity (TRUE in the last argument), which is what we want. If a match is found, we set the final status of the IRP to STATUS_NOT_FOUND (technically any failure status would work), and change the final return value from the function to FLT_PREOP_COMPLETE, preventing any further propagation to lower-layered filters.

The driver is installed in the normal way:

```
c:\>sc create hide type= filesys binPath= c:\test\khide.sys
```

And is started just like any other file system mini-filter:

```
c:\>fltmcl load hide
```

Now trying to navigate to a hidden directory (e.g *c:\temp*) works, but the directory is always reported empty:

```
C:\Temp>dir
Volume in drive C has no label.
Volume Serial Number is E041-5DB0
```

```
Directory of C:\Temp
```

```
File Not Found
```

Phase 2: Making a Directory Invisible

When directory listing requested using IRP_MJ_DIRECTORY_CONTROL, it's the job of the file system driver to provide the list. One way to hide a directory (or a file for that matter), is to assume the role of the file system and produce such a listing, that would have the directory removed from it. This is

possible, but difficult. A better option is to let the file system driver “do its thing”, and then tweak the returned result before letting it bubble up to the client.

We’ll use the second approach. To that end, we need to respond to IRP_MJ_DIRECTORY_CONTROL after it has been processed by the I/O stack. This means we need a post callback. The driver’s mini-filter callback registration structure changes to the following:

```
FLT_OPERATION_REGISTRATION callbacks[] = {
    { IRP_MJ_DIRECTORY_CONTROL, 0,
        OnPreDirectoryControl, OnPostDirectoryControl },
    { IRP_MJ_OPERATION_END }
};
```

The post-callback does the heavy lifting. The idea is to look for a **parent** directory that contains the directory we wish to hide, and if this is the case - remove our directory name from the list somehow before it returns to the caller.

Let’s start, as before, but letting kernel callers have their way without interference:

```
FLT_POSTOP_CALLBACK_STATUS
OnPostDirectoryControl(PFLT_CALLBACK_DATA Data,
    PCFLT RELATED OBJECTS F1tObjects,
    PVOID, FLT_POST_OPERATION_FLAGS flags) {
    UNREFERENCED_PARAMETER(F1tObjects);

    if (Data->RequestorMode == KernelMode ||
        Data->Iopb->MinorFunction != IRP_MN_QUERY_DIRECTORY ||
        (flags & FLTFL_POST_OPERATION_DRAINING))
        return FLT_POSTOP_FINISHED_PROCESSING;
```

If the caller is from kernel mode, or the request is not “query directory” (IRP_MN_QUERY_DIRECTORY), we let the request continue normally. The last check is an optimization that looks at the Flags argument, where the value FLTFL_POST_OPERATION_DRAINING indicates the mini-filter instance is being detached, so no point in doing anything.

The information we get with IRP_MJ_DIRECTORY_CONTROL and IRP_MN_QUERY_DIRECTORY in the FLT_PARAMETERS union looks like the following:

```

struct {
    ULONG Length;
    PUNICODE_STRING FileName;
    FILE_INFORMATION_CLASS FileInformationClass;
    ULONG POINTER_ALIGNMENT FileIndex;
    PVOID DirectoryBuffer;
    PMDL MdlAddress;
} QueryDirectory;

```

`FileInformationClass` is the type of request. The `FILE_INFORMATION_CLASS` enumeration is a big one, but only a few are relevant to a query directory request. The docs list 8 of those. For each one, the `DirectoryBuffer` member points to a different kind of structure. Table 12-4 shows the enumeration values and the corresponding types as defined in the docs.

Table 12-4: Query directory file information class values and data

Enumeration	Structure Type
<code>FileBothDirectoryInformation</code>	<code>FILE_BOTH_DIR_INFORMATION</code>
<code>FileDirectoryInformation</code>	<code>FILE_DIRECTORY_INFORMATION</code>
<code>FileFullDirectoryInformation</code>	<code>FILE_FULL_DIR_INFORMATION</code>
<code>FileIdBothDirectoryInformation</code>	<code>FILE_ID_BOTH_DIR_INFORMATION</code>
<code>FileIdFullDirectoryInformation</code>	<code>FILE_ID_FULL_DIR_INFORMATION</code>
<code>FileNamesInformation</code>	<code>FILE_NAMES_INFORMATION</code>
<code>FileObjectIdInformation</code>	<code>FILE_OBJECTID_INFORMATION</code>
<code>FileReparsePointInformation</code>	<code>FILE_REPARSE_POINT_INFORMATION</code>

All the above data structures are similar in spirit, but not identical. Let's take one example:

```

typedef struct _FILE_DIRECTORY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    _Field_size_bytes_(FileNameLength) WCHAR FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;

```

All the structures listed in table 12-4 start with a `NextEntryOffset` member that points to the next same-kind structure. Its value must be added to the current pointer to this structure. The last instance has the `NextEntryOffset` set to zero, indicating there are no more instances. This idea is depicted in figure 12-8.

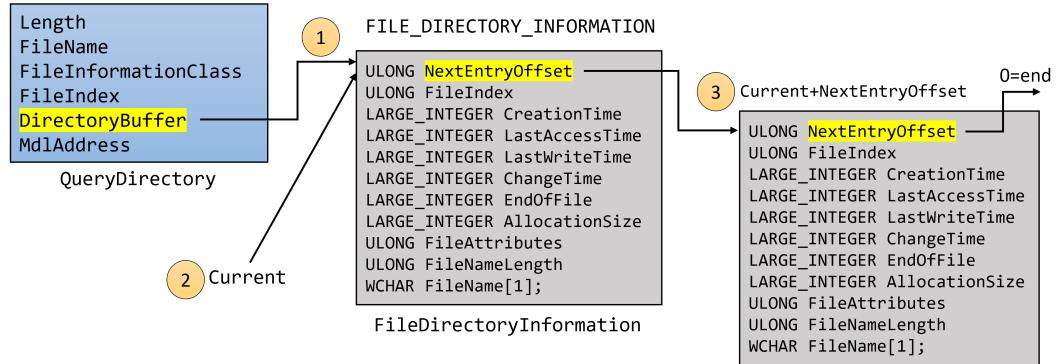


Figure 12-8: Directory information structures

The interesting part of the specific structure is the `FileName` member. This has the file or directory name for which some information is required or provided. This is not a full path - rather, it's just the final name relative to the immediate parent directory. For example, if a query directory is sent to a directory named `c:\Dir1\Dir2`, the `FileName` members would hold names like `file1.txt`, `mydir` (directory), and so on.

All the details above mean that in order to hide a directory from a listing, we first need to check if the parent directory being queried is a parent of any of the directories we are supposed to hide. Then we need to traverse the structure layout as described, looking for the directory name (its final component). If we find it, we can hide the directory by pointing the previous `NextEntryOffset` to the next one, skipping this one structure we want to "hide". This is depicted in figure 12-9.

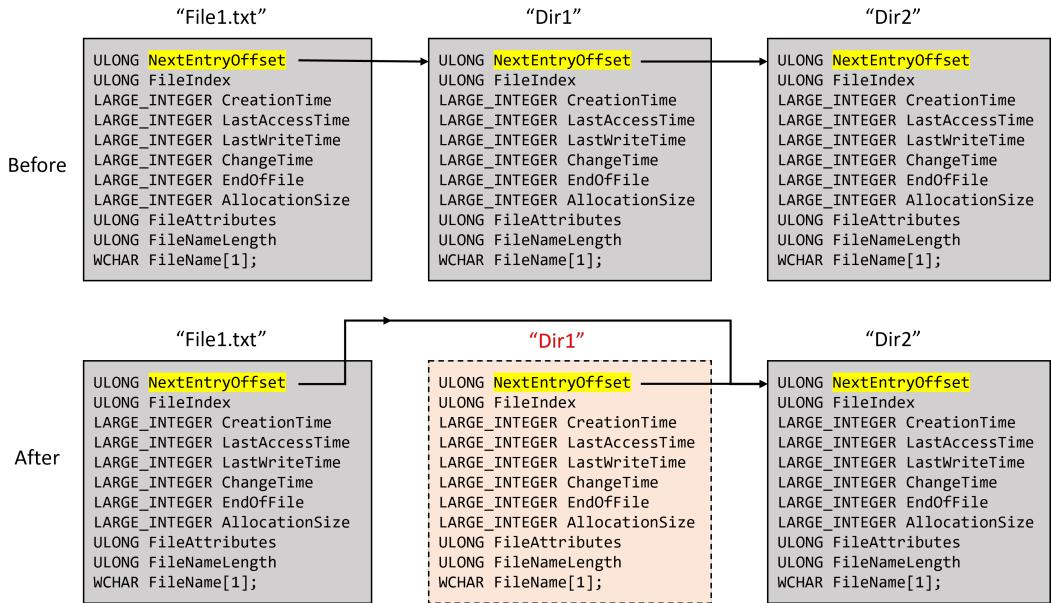


Figure 12-9: Directory "Dir1" is being hidden"

The example above is using `FILE_DIRECTORY_INFORMATION`, but we have to contend with all other 7 possible structures. The problem is that the `FileName` member is not located at the same offset in these structures! How can we deal with that in a sensible way?

Fortunately, in recent WDK versions, the `<ntifs.h>` header (where these structures are defined, and is included by `FltKernel.h`) provides several convenience macros that provide the offsets to key (common) members in these structures, namely `NextEntryOffset` (which is always zero in current versions), `FileName`, and `FileNameLength`. These macros initialize a structure named `FILE_INFORMATION_DEFINITION` to hold these offsets along with the corresponding `FileInfoClass`:

```
typedef struct _FILE_INFORMATION_DEFINITION {
    FILE_INFORMATION_CLASS Class;
    ULONG NextEntryOffset;
    ULONG FileNameLengthOffset;
    ULONG FileNameOffset;
} FILE_INFORMATION_DEFINITION, *PFILE_INFORMATION_DEFINITION;
```

Here is a definition for use with `FILE_DIRECTORY_INFORMATION`:

```
// from ntifs.h

#define FileDirectoryInformationDefinition { \
    FileDirectoryInformation, \
    FIELD_OFFSET(FILE_DIRECTORY_INFORMATION, NextEntryOffset), \
    FIELD_OFFSET(FILE_DIRECTORY_INFORMATION, FileName), \
    FIELD_OFFSET(FILE_DIRECTORY_INFORMATION, FileNameLength) \
}
```

Astute readers may notice a bug here. I didn't at first, as I assumed the definitions from WDK headers are correct. Can you spot the error?



The offsets of `FileName` and `FileNameLength` are in reverse order!

I reported the bug, but not sure if and when that will be fixed. It may very well be the case that the header you're using is already fixed. Please be aware that the next code snippets assume the error exists, and swap the usage of `FileNameLengthOffset` and `FileNameOffset`.

Back to the `QueryDirectory` structure. The `Length` member is the total length of the data pointed to by `DirectoryBuffer`. It's not usually needed, but can serve as a sanity check. The `MdlAddress` member provides an optional MDL that points to where `DirectoryBuffer` does. The docs indicate that the MDL should be used if provided (by calling `MmGetSystemAddressForMdlSafe`). The `DirectoryBuffer` address, by the way, points to user-mode memory when the query request is coming from user mode (such as from *Explorer.exe*).

Now that we have all the pieces for the plan, we can go ahead and implement the rest of the post-`IRP_MJ_DIRECTORY_CONTROL` callback.

We'll continue by setting up an array of the expected structures and information classes using the macros provided like `FileDirectoryInformationDefinition`:

```
static const FILE_INFORMATION_DEFINITION defs[] = {
    FileFullDirectoryInformationDefinition,
    FileBothDirectoryInformationDefinition,
    FileDirectoryInformationDefinition,
    FileNamesInformationDefinition,
    FileIdFullDirectoryInformationDefinition,
    FileIdBothDirectoryInformationDefinition,
    FileIdExtdDirectoryInformationDefinition,
    FileIdGlobalTxDirectoryInformationDefinition
};
```

Each item in the array is a `FILE_INFORMATION_DEFINITION` instance holding the correct offsets to locate `NextEntryOffset`, `FileName`, and `FileNameLength` in each corresponding structure.

Now we need to search and locate the actual information class handed to us:

```

const FILE_INFORMATION_DEFINITION* actual = nullptr;
for(auto const& def : defs)
    if (def.Class == params.FileInformationClass) {
        actual = &def;
        break;
    }

if (actual == nullptr) {
    KdPrint((DRIVER_PREFIX "Uninteresting info class (%u)\n",
        params.FileInformationClass));
    return FLT_POSTOP_FINISHED_PROCESSING;
}

```

The loop above might seem weird, but C++ 11 and later allow using *range-based for* for iterating through fixed sized arrays, as is the case here with `defs`. If that feels awkward, feel free to change to a class for loop with an index.

The actual pointer now points to the correct `FILE_INFORMATION_DEFINITION` that we need to use. Next, we need to grab the DOS path of the queried directory, and start comparing it to our list of directory parents:

```

POBJECT_NAME_INFORMATION dosPath = nullptr;
IoQueryFileDosDeviceName(FltObjects->FileObject, &dosPath);
if (dosPath) {
    PUCHAR base = nullptr;
    //
    // use MDL if available
    //
    if (params.MdlAddress)
        base = (PUCHAR)MmGetSystemAddressForMdlSafe(params.MdlAddress,
            NormalPagePriority);
    if (!base)
        base = (PUCHAR)params.DirectoryBuffer;
    if (base == nullptr) {
        //
        // the doc says DirectoryBuffer could be NULL
        //
        return FLT_POSTOP_FINISHED_PROCESSING;
    }
}

```

```

SharedLocker locker(g_State->Lock);
for (auto& name : g_State->Files) {
    //
    // look for a backslash so we can remove the final component
    //
    auto bs = wcsrchr(name, L'\\');
    if (bs == nullptr)
        continue;

    UNICODE_STRING copy;
    copy.Buffer = name.Data(); // C-pointer to the characters
    copy.Length = USHORT(bs - name + 1) * sizeof(WCHAR);
    //
    // copy now points to the parent directory
    // by making its Length shorter
    //
    if (copy.Length == sizeof(WCHAR) * 2) // Drive+colon only (e.g. C:)
        copy.Length += sizeof(WCHAR); // add the backslash

    if (RtlEqualUnicodeString(&copy, &dosPath->Name, TRUE)) {

```

To clarify the above code, suppose the DOS directory is `c:\Dir1\Dir2`. This means some client is asking about the contents in this directory. If one of the directories to hide is `c:\Dir1\Dir2\Dir3` (stored in one of the strings in our vector), we have to compare with its parent, which in this case should succeed.

The parent matches the directory queried, which means we have to iterate through the results, locate the final component in the list (`Dir3` in the above example), and “hide” the directory by changing the `NextEntryOffset` as described earlier. Here goes:

```

ULONG nextOffset = 0;
P UCHAR prev = nullptr;
auto str = bs + 1; // the final component beyond the backslash

do {
    //
    // due to a current bug in the definition of FILE_INFORMATION_DEFINITION
    // the file name and length offsets are switched in the definitions
    // of the macros that initialize FILE_INFORMATION_DEFINITION
    //
    auto filename = (PCWSTR)(base + actual->FileNameLengthOffset);
    auto filenameLen = *(PULONG)(base + actual->FileNameOffset);

    nextOffset = *(PULONG)(base + actual->NextEntryOffset);

```

```

if (filenameLen && _wcsnicmp(str, filename,
    filenameLen / sizeof(WCHAR)) == 0) {
    //
    // found it! hide it and exit
    //
    if (prev == nullptr) {
        //
        // first entry - move the buffer to the next item
        //
        params.DirectoryBuffer = base + nextOffset;

        //
        // notify the Filter Manager
        //
        FltSetCallbackDataDirty(Data);
    }
    else {
        //
        // Hide the directory!
        //
        *(PULONG)(prev + actual->NextEntryOffset) += nextOffset;
    }
    break;
}
prev = base;
base += nextOffset;
} while (nextOffset != 0);
break;

```

A few notes on the above code:

- We have to keep track of the previous pointer, so that we can manipulate it from the **current** node we're traversing. This is the role of the prev local variable.
- prev is defined as PUCHAR (pointer to unsigned character - a byte) to make sure adding any offset is interpreted as bytes. Remember, adding a number to a pointer advances the pointer by the number times the size of the item being pointed to. Same reasoning applies to the base variable.
- If the directory we need to hide happens to be the first, we need to change the DirectoryBuffer member itself (move it to the second item), and that requires notifying the filter manager by calling FltSetCallbackDataDirty. It can't really happen in this example, as the first item returned is always the “.” (dot) directory, referring to the current directory, but it's good to know about this practice that may be needed in other cases.

All that's left to do is free the DOS path and return FLT_POSTOP_FINISHED_PROCESSING from the callback.

The full code of the callback is presented here for convenience (with some of the earlier comments removed):

```
FLT_POSTOP_CALLBACK_STATUS
OnPostDirectoryControl(PFLT_CALLBACK_DATA Data,
    PCFLT RELATED OBJECTS FltObjects, PVOID,
    FLT_POST_OPERATION_FLAGS flags) {
    UNREFERENCED_PARAMETER(FltObjects);

    if (Data->RequestorMode == KernelMode ||
        Data->Iopb->MinorFunction != IRP_MN_QUERY_DIRECTORY ||
        (flags & FLTFL_POST_OPERATION_DRAINING))
        return FLT_POSTOP_FINISHED_PROCESSING;

    auto& params = Data->Iopb->Parameters.DirectoryControl.QueryDirectory;

    static const FILE_INFORMATION_DEFINITION defs[] = {
        FileFullDirectoryInformationDefinition,
        FileBothDirectoryInformationDefinition,
        FileDirectoryInformationDefinition,
        FileNamesInformationDefinition,
        FileIdFullDirectoryInformationDefinition,
        FileIdBothDirectoryInformationDefinition,
        FileIdExtdDirectoryInformationDefinition,
        FileIdGlobalTxDirectoryInformationDefinition
    };
    const FILE_INFORMATION_DEFINITION* actual = nullptr;
    for(auto const& def : defs)
        if (def.Class == params.FileInformationClass) {
            actual = &def;
            break;
        }

    if (actual == nullptr) {
        return FLT_POSTOP_FINISHED_PROCESSING;
    }

    POBJECT_NAME_INFORMATION dosPath = nullptr;
    IoQueryFileDosDeviceName(FltObjects->FileObject, &dosPath);
    if (dosPath) {
        PUCHAR base = nullptr;
        if (params.Md1Address)
            base = (PUCHAR)MmGetSystemAddressForMd1Safe(params.Md1Address,
```

```
        NormalPagePriority);

if (!base)
    base = (P UCHAR)params.DirectoryBuffer;
if (base == nullptr) {
    return FLT_POSTOP_FINISHED_PROCESSING;
}

SharedLocker locker(g_State->Lock);
for (auto& name : g_State->Files) {
    //
    // look for a backslash so we can remove the final component
    //
    auto bs = wcsrchr(name, L'\\');
    if (bs == nullptr)
        continue;

    UNICODE_STRING copy;
    copy.Buffer = name.Data();
    copy.Length = USHORT(bs - name + 1) * sizeof(WCHAR);
    //
    // copy now points to the parent directory
    // by making its Length shorter
    //
    if (copy.Length == sizeof(WCHAR) * 2)      // Drive+colon only
        copy.Length += sizeof(WCHAR);           // add the backslash

    if (RtlEqualUnicodeString(&copy, &dosPath->Name, TRUE)) {
        ULONG nextOffset = 0;
        P UCHAR prev = nullptr;
        auto str = bs + 1;      // the final component

        do {
        //
        // due to a current bug in the definition of FILE_INFORMATION_DEFINITION
        // the file name and length offsets are switched in the definitions
        // of the macros that initialize FILE_INFORMATION_DEFINITION
        //
        auto filename = (PCWSTR)(base +
            actual->FileNameLengthOffset);
        auto filenameLen = *(PULONG)(base +
            actual->FileNameOffset);

        nextOffset = *(PULONG)(base + actual->NextEntryOffset);
```

```

        if (filenameLen && _wcsnicmp(str, filename,
            filenameLen / sizeof(WCHAR)) == 0) {
        //
        // found it! hide it and exit
        //
        if (prev == nullptr) {
        //
        // first entry
        //
        params.DirectoryBuffer = base + nextOffset;
        FltSetCallbackDataDirty(Data);
        }
        else {
            *(PULONG)(prev + actual->NextEntryOffset)
                += nextOffset;
        }
        break;
    }
    prev = base;
    base += nextOffset;
} while (nextOffset != 0);
break;
}
}
ExFreePool(dosPath);
}
return FLT_POSTOP_FINISHED_PROCESSING;
}

```

Here is a directory listing when `c:\temp` is supposed to be hidden (before and after):

```

C:\>dir
Volume in drive C has no label.
Volume Serial Number is E041-5DB0

Directory of C:\

09/24/2022  02:42 PM           106,784 appverifUI.dll
10/02/2022  01:05 PM      <DIR>          DBG
10/30/2022  01:07 PM      <DIR>          Demos
10/10/2022  05:10 PM      <DIR>          dev
04/27/2022  07:53 AM      <DIR>  Program Files

```

```
10/18/2022  05:51 PM    <DIR>        Program Files (x86)
10/29/2022  01:42 PM    <DIR>        symbols
10/29/2022  10:43 PM    <DIR>        Temp
11/18/2021  05:16 AM    <DIR>        Users
09/24/2022  02:42 PM            61,376 vfcompat.dll
10/29/2022  03:29 PM    <DIR>        Windows
            3 File(s)      180,448 bytes
            13 Dir(s)   35,709,960,192 bytes free
```

```
C:\>fltmc load hide
```

```
C:\>dir
Volume in drive C has no label.
Volume Serial Number is E041-5DB0
```

```
Directory of C:\
```

```
09/24/2022  02:42 PM            106,784 appverifUI.dll
10/02/2022  01:05 PM    <DIR>        DBG
10/30/2022  01:07 PM    <DIR>        Demos
10/10/2022  05:10 PM    <DIR>        dev
04/27/2022  07:53 AM    <DIR>        Program Files
10/18/2022  05:51 PM    <DIR>        Program Files (x86)
10/29/2022  01:42 PM    <DIR>        symbols
11/18/2021  05:16 AM    <DIR>        Users
09/24/2022  02:42 PM            61,376 vfcompat.dll
10/29/2022  03:29 PM    <DIR>        Windows
            3 File(s)      180,448 bytes
            12 Dir(s)   35,707,621,376 bytes free
```

You can still navigate to the *Temp* directory with `cd temp`, but any `dir` inside would be empty. If you want to prevent *that*, you can handle the pre-callback for `IRP_MJ_CREATE` and fail access to any of the managed directories. I'll leave that as an exercise for the reader.

Contexts

In some scenarios it is desirable to attach some data to file system entities such as volumes and files. The filter manager provides this capability through *contexts*. A context is a data structure provided by the mini-filter driver that can be set and retrieved for any file system object. These contexts are connected to the objects they are set on, for as long as these objects are alive.

To use contexts, the driver must declare beforehand what contexts it may require and for what type of objects. This is done as part of the registration structure `FLT_REGISTRATION`. The `ContextRegistration`

field may point to an array of `FLT_CONTEXT_REGISTRATION` structures, each of which defines information for a single context. `FLT_CONTEXT_REGISTRATION` is declared as follows:

```
typedef struct _FLT_CONTEXT_REGISTRATION {
    FLT_CONTEXT_TYPE ContextType;
    FLT_CONTEXT_REGISTRATION_FLAGS Flags;
    PFLT_CONTEXT_CLEANUP_CALLBACK ContextCleanupCallback;
    SIZE_T Size;
    ULONG PoolTag;
    PFLT_CONTEXT_ALLOCATE_CALLBACK ContextAllocateCallback;
    PFLT_CONTEXT_FREE_CALLBACK ContextFreeCallback;
    PVOID Reserved1;
} FLT_CONTEXT_REGISTRATION, *PFLT_CONTEXT_REGISTRATION;
```

Here is a description of the above fields:

- *ContextType* identifies the object type this context would be attached to. The `FLT_CONTEXT_TYPE` is `typedef`ed as `USHORT` and can have one of the following values:

```
#define FLT_VOLUME_CONTEXT      0x0001
#define FLT_INSTANCE_CONTEXT     0x0002
#define FLT_FILE_CONTEXT         0x0004
#define FLT_STREAM_CONTEXT       0x0008
#define FLT_STREAMHANDLE_CONTEXT 0x0010
#define FLT_TRANSACTION_CONTEXT   0x0020
#if FLT_MGR_WIN8
#define FLT_SECTION_CONTEXT      0x0040
#endif // FLT_MGR_WIN8
#define FLT_CONTEXT_END           0xffff
```

As can be seen from the above definitions, a context can be attached to a volume, filter instance, file, stream, stream handle, transaction and section (on Windows 8 and later). The last value is a sentinel for indicating this is the end of the list of context definitions. The aside “Context Types” contains more information on the various context types.

Context Types

The filter manager supports several types of contexts:

- Volume contexts are attached to volumes, such as a disk partition (C:, D:, etc.).
- Instance contexts are attached to filter instances. A mini-filter can have several instances running, each attached to a different volume.
- File contexts can be attached to files in general (and not a specific file stream).

- Stream contexts can be attached to file streams, supported by some file systems, such as NTFS. File systems that support a single stream per file (such as FAT) treat stream contexts as file contexts.
- Stream handle contexts can be attached to a stream on a per FILE_OBJECT.
- Transaction contexts can be attached to a transaction that is in progress. Specifically, the NTFS file system supports transactions, and such so a context can be attached to a running transaction.
- Section contexts can be attached to section (file mapping) objects created with the function FltCreateSectionForDataScan (beyond the scope of this chapter).

Not all types of contexts are supported on all file systems. The filter manager provides APIs to query this dynamically if desired (for some context types), such as FltSupportsFileContexts, FltSupportsFileContextsEx and FltSupportsStreamContexts.

Context size can be fixed or variable. If fixed size is desired, it's specified in the *Size* field of FLT_CONTEXT_REGISTRATION. For a variable sized context, a driver specifies the special value FLT_VARIABLE_SIZED_CONTEXTS (-1). Using fixed-size contexts is more efficient, because the filter manager can use lookaside lists for managing allocations and deallocations.

The pool tag is specified with the *PoolTag* field of FLT_CONTEXT_REGISTRATION. This is the tag the filter manager will use when actually allocating the context. The next two fields are optional callbacks where the driver provides the allocation and deallocation functions. If these are non-NULL, then the *PoolTag* and *Size* fields are meaningless and not used.

Here is an example of building an array of context registration structure:

```
struct MyContext {
    //...
};

const FLT_CONTEXT_REGISTRATION ContextRegistration[] = {
    { FLT_FILE_CONTEXT, 0, nullptr, sizeof(MyContext), 'dcba',
        nullptr, nullptr, nullptr },
    { FLT_CONTEXT_END }
};
```

Managing Contexts

To actually use a context, a driver first needs to allocate it by calling FltAllocateContext, defined like so:

```
NTSTATUS FltAllocateContext (
    _In_ PFLT_FILTER Filter,
    _In_ FLT_CONTEXT_TYPE ContextType,
    _In_ SIZE_T ContextSize,
    _In_ POOL_TYPE PoolType,
    _Outptr_ PFLT_CONTEXT *ReturnedContext);
```

The *Filter* parameter is the filter's opaque pointer returned by `FltRegisterFilter` but also available in the `FLT_RELATED_OBJECTS` structure provided to all callbacks. *ContextType* is one of the supported context macros shown earlier, such as `FLT_FILE_CONTEXT`. *ContextSize* is the requested context size in bytes (must be greater than zero). *PoolType* can be `PagedPool` or `NonPagedPool`, depending on what IRQL the driver is planning to access the context (for volume contexts, `NonPagedPool` must be specified). Finally, the *ReturnedContext* field stores the returned allocated context; `PFLT_CONTEXT` is `typedef`ed as `PVOID`.

Once the context has been allocated, the driver can store in that data buffer anything it wishes. Then it must attach the context to an object (this is the reason to create the context in the first place) using one of several functions named `FltSetXxxContext` where "Xxx" is one of `File`, `Instance`, `Volume`, `Stream`, `StreamHandle`, or `Transaction`. The only exception is a section context which is set with `FltCreateSectionForDataScan`. Each of the `FltSetXxxContext` functions has the same generic makeup, shown here for the `File` case:

```
NTSTATUS FltSetFileContext (
    _In_ PFLT_INSTANCE Instance,
    _In_ PFILE_OBJECT FileObject,
    _In_ FLT_SET_CONTEXT_OPERATION Operation,
    _In_ PFLT_CONTEXT NewContext,
    _Outptr_ PFLT_CONTEXT *OldContext);
```

The function accepts the required parameters for the context at hand. In this file case it's the instance (actually needed in any set context function) and the file object representing the file that should carry this context. The *Operation* parameter can be either `FLT_SET_CONTEXT_REPLACE_IF_EXISTS` or `FLT_SET_CONTEXT_KEEP_IF_EXISTS`, which are pretty self explanatory.

NewContext is the context to set, and *OldContext* is an optional parameter that can be used to retrieve the previous context with the operation set to `FLT_SET_CONTEXT_REPLACE_IF_EXISTS`.

Contexts are reference counted. Allocating a context (`FltAllocateContext`) and setting a context increment its reference count. The opposite function is `FltReleaseContext` that must be called a matching number of times to make sure the context is not leaked. Although there is context delete function (`FltDeleteContext`), it's usually not needed as the filter manager will tear down the context once the file system object holding it is destroyed.



You must pay careful attention to context management, otherwise you may find that the driver cannot be unloaded because a positive reference counted context is still alive, and the file system object it's attached to has not yet been deleted (such as a file or volume). Clearly, this suggests a RAII context handling class could be useful.

The typical scenario would be to allocate a context, fill it, set it on the relevant object and then call `FltReleaseContext` once, keeping a reference count of one for the context. We will see a practical use of contexts in the “File Backup Driver” section later in this chapter.

Once a context has been set on an object, other callbacks may wish to get a hold of that context. A set of “get” functions provide access to the relevant context, all named in the form `FltGetXxxContext`, where “`Xxx`” is one of `File`, `Instance`, `Volume`, `Stream`, `StreamHandle`, `Transaction` or `Section`. The “get” functions increment the context’s reference count and so calling `FltReleaseContext` is necessary once working with the context is completed.

Initiating I/O Requests

File system mini-filters sometimes need to initiate their own I/O operations. Normally, kernel code would use functions such as `ZwCreateFile` or `ZwOpenFile` to open a handle to a file, and then issue I/O operations with functions such as `ZwReadFile`, `ZwWriteFile`, and `ZwDeviceIoControlFile`. Mini-filters don’t usually use `ZwCreateFile` if they need to issue an I/O operation from one of the filter manager’s callbacks. The reason has to do with the fact that the I/O operation will travel from the topmost filter down towards the file system itself, meeting the current mini-filter on the way! This is a form of *reentrancy*, which can cause issues if the driver is not careful. It also has a performance penalty because the entire file system stack of filters must be traversed.

Instead, mini-filters use filter manager routines to issue I/O operations that are sent to the next lower filter towards the file system, preventing reentrancy and a performance hit. These APIs start with `Flt` and are similar in concept to the “`Zw`” variants. The main function to use is `FltCreateFile` (or its extended friends, `FltCreateFileEx` and `FltCreateFileEx2`). Here is the prototype of `FltCreateFile`:

```
NTSTATUS FltCreateFile (
    _In_ PFLT_FILTER Filter,
    _In_opt_ PFLT_INSTANCE Instance,
    _Out_ PHANDLE FileHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _In_opt_ PLARGE_INTEGER AllocationSize,
    _In_ ULONG FileAttributes,
    _In_ ULONG ShareAccess,
    _In_ ULONG CreateDisposition,
    _In_ ULONG CreateOptions,
    _In_reads_bytes_opt_(EaLength) PVOID EaBuffer,
    _In_ ULONG EaLength,
    _In_ ULONG Flags);
```

Wow, that’s quite a mouthful - this function has many, many options. Fortunately, they are not difficult to understand, but they must be set just right, or the call will fail with some weird status.

As can be seen from the declaration, the first argument is the filter opaque address, used as the base layer for I/O operations through the resulting file handle. The main return value is the *FileHandle* to the open file if successful. We won't go over all the various parameters (refer to the WDK documentation), but we will use this function in the next section.

The extended function `FltCreateFileEx` has an additional output parameter which is the `FILE_OBJECT` pointer created by the function. `FltCreateFileEx2` has an additional input parameter of type `IO_DRIVER_CREATE_CONTEXT` used to specify additional information to the file system (refer to the WDK documentation for more information).

With the returned handle, the driver can call the standard I/O APIs such as `ZwReadFile`, `ZwWriteFile`, etc. The operation will still target lower layers only. Alternatively, the driver can use the returned `FILE_OBJECT` from `FltCreateFileEx` or `FltCreateFileEx2` with functions such as `FltReadFile` and `FltWriteFile` (the latter functions require the file object rather than a handle). The `Flt` functions are preferable, not just for consistency, but also because they are slightly faster, as they receive the file object directly rather than having a handle to look up to locate the file object.

Once the operation is done, `FltClose` must be called on the returned handle. If a file object was returned as well, its reference count must be decremented with `ObDereferenceObject` to prevent a leak.



`FltClose` just calls `ZwClose`; it's there for consistency.

The File Backup Driver

It's time to put what we learned into practice, specifically using contexts and I/O operations from within a mini-filter driver. The driver we'll build provides automatic backup of a file whenever that file is opened for write access, just before it's being written. In this way, it's possible to revert to the previous file state if desired. In effect - we have a single backup of the file at any point.

An important question is, where will that backup be stored? It's possible to create some "backup" directory within the directory of the file, or perhaps create a root directory for all backups and re-create the backup in the same folder structure of the original file, but starting from the backup root directory (the driver can even hide this directory from general access). These options are fine, but for this demo we'll use another option: we'll store the backup of the file *within the file itself*, in an alternate NTFS stream. So essentially, the file would contain its own backup. Then, if needed, we can swap the contexts of the alternate stream with the default stream, effectively restoring the file to its previous state.

We'll start, as before, with an *Empty WDM Driver* project named *KBackup*, and delete the INF file. We'll use the same infrastructure we've used in the *Delprotect* and *Hide* drivers. `DriverEntry` is going

to be simpler this time, as we will not implement any CDO, and just use hard-coded rules to decide which files should be backed up. Adding the required flexibility is saved as an exercise for the reader.

Here is `DriverEntry`:

```
PFLT_FILTER g_Filter;

extern "C"
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) \
{
    auto status = InitMiniFilter(DriverObject, RegistryPath);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed to init mini-filter (0x%X)\n", status));
        return status;
    }

    status = FltStartFiltering(g_Filter);
    if (!NT_SUCCESS(status)) {
        FltUnregisterFilter(g_Filter);
    }
    return status;
}
```

We just register the filter by calling `InitMiniFilter` (do be described momentarily), and call `FltStartFiltering` to get things going.

Registering the filter is mostly similar to earlier drivers, except that we'll need some context to be kept for files that we are going to back up. This means registration needs information about the context objects we plan to use. Here is the context structure we'll use:

```
struct FileContext {
    Mutex Lock;
    LARGE_INTEGER BackupTime;
    BOOLEAN Written;
};
```

We'll see the usage of this structure when we implement the callbacks. Registration is performed within `InitMiniFilter` after the standard Registry entries have been written:

```

FLT_OPERATION_REGISTRATION const callbacks[] = {
    { IRP_MJ_CREATE, 0, nullptr, OnPostCreate },
    { IRP_MJ_WRITE, 0, OnPreWrite },
    { IRP_MJ_CLEANUP, 0, nullptr, OnPostCleanup },
    { IRP_MJ_OPERATION_END }
};

const FLT_CONTEXT_REGISTRATION context[] = {
    { FLT_FILE_CONTEXT, 0, nullptr, sizeof(FileContext), DRIVER_TAG },
    { FLT_CONTEXT_END }
};

FLT_REGISTRATION const reg = {
    sizeof(FLT_REGISTRATION),
    FLT_REGISTRATION_VERSION,
    0,                                // Flags
    context,                           // Context
    callbacks,                         // Operation callbacks
    BackupUnload,                      // MiniFilterUnload
    BackupInstanceSetup,               // InstanceSetup
    BackupInstanceQueryTeardown,       // InstanceQueryTeardown
    BackupInstanceTeardownStart,        // InstanceTeardownStart
    BackupInstanceTeardownComplete,     // InstanceTeardownComplete
};
status = FltRegisterFilter(DriverObject, &reg, &g_Filter);

```

As far as contexts go, we'll need a context attached to certain files, so `FLT_FILE_CONTEXT` is the type of context required. As for callbacks, we need to intercept `IRP_MJ_CREATE` after a file object has been created to see whether it's an interesting file. `IRP_MJ_WRITE` is required, so we can write the contents of the file right before its contents are modified. The `IRP_MJ_CLEANUP` operation will be used to clean up our context objects.

Since we'll be using alternate streams, only NTFS can be used, as it's the only standard file system in Windows to support alternate file streams. This means the driver should not attach to a volume not using NTFS. We used similar code in earlier drivers to attach to NTFS volumes only:

```

NTSTATUS BackupInstanceSetup(
    PCFLT RELATED OBJECTS FltObjects, FLT_INSTANCE_SETUP_FLAGS Flags,
    DEVICE_TYPE VolumeDeviceType, FLT_FILESYSTEM_TYPE VolumeFilesystemType) {

    UNREFERENCED_PARAMETER(FltObjects);
    UNREFERENCED_PARAMETER(Flags);
    UNREFERENCED_PARAMETER(VolumeDeviceType);

    return VolumeFilesystemType == FLT_FSTYPE_NTFS
        ? STATUS_SUCCESS : STATUS_FLT_DO_NOT_ATTACH;
}

```

The Post Create Callback

Why do we even need a post-create callback? It is actually possible to write the driver without it, but it will help demonstrate some features we haven't seen before. Our goal for post-create is to allocate a file context for files we're interested in. For example, files that are not open for write access are of no interest to the driver.

Why do we use a post callback rather than a pre-callback? If a file open operation fails by some pre-create of another driver, we don't care. Only if the file is opened successfully, then our driver should examine the file further.

First, we'll bail if the flags argument indicates the instance is going away:

```

FLT_POSTOP_CALLBACK_STATUS OnPostCreate(
    PFLT_CALLBACK_DATA Data, PCFLT RELATED OBJECTS FltObjects,
    PVOID, FLT_POST_OPERATION_FLAGS Flags) {
    if (Flags & FLTFL_POST_OPERATION_DRAINING)
        return FLT_POSTOP_FINISHED_PROCESSING;
}

```

Next, let's extract the parameters of the create operation, and check if the file in question is a directory:

```

const auto& params = Data->Iopb->Parameters.Create;
BOOLEAN dir = FALSE;
FltIsDirectory(FltObjects->FileObject, FltObjects->Instance, &dir);

```

`FltIsDirectory` is a simple function provided by the filter manager that returns TRUE in the last boolean argument if the file object in question refers to a directory.

We are only interested in files opened for write access, not from kernel mode, and not new files (since new files do not require backup). Also, directories are not interesting:

```

if (dir
    || Data->RequestorMode == KernelMode
    || (params.SecurityContext->DesiredAccess & FILE_WRITE_DATA) == 0
    || Data->IoStatus.Status != STATUS_SUCCESS
    || Data->IoStatus.Information == FILE_CREATED) {
    //
    // kernel caller, not write access or a new file - skip
    //
    return FLT_POSTOP_FINISHED_PROCESSING;
}

```

The IO_STATUS_BLOCK.Information in a post-create callback returns how the file was created/opened (if the operation is successful). In the case of a new file being created, we don't care, as there is nothing to back up.

Check out the documentation for `FLT_PARAMETERS` for `IRP_MJ_CREATE` to get more information on the details shown above.

These kinds of checks are important, as they remove a lot of possible overhead for the driver. The driver should always strive to do as little as possible to reduce its performance impact.

Now that we have a file we care about, we need to prepare a context object to be attached to the file. This context will be needed later when we process the pre-write callback. First, we'll extract the name of the file. The driver needs to call the standard `FltGetFileNameInformation`. To make it a little easier and less error-prone, we'll use the RAII wrapper from the *KT*L.

Why don't we just create a backup for the file right here and now? The file was opened for write access, but there is no guarantee the client will actually *write* to the file; so we'll wait until we get a pre-write callback to perform the backup.

```

FilterFileNameInformation fileNameInfo(Data);
if (!fileNameInfo) {
    return FLT_POSTOP_FINISHED_PROCESSING;
}

```

In this driver, we'll backup files that have certain extensions - as mentioned already these will be hard coded to simplify the coding that has little to do with file system mini-filters. We'll call a helper function to determine if we should care about this file:

```
if (!ShouldBackupFile(fileNameInfo))
    return FLT_POSTOP_FINISHED_PROCESSING;
```

Here is the implementation of ShouldBackupFile:

```
bool ShouldBackupFile(FilterFileNameInformation& nameInfo) {
    if(!NT_SUCCESS(nameInfo.Parse()))
        return false;

    //
    // hard coded list of extensions
    //
    static PCWSTR extensions[] = {
        L"txt", L"docx", L"doc", L"jpg", L"png"
    };

    for (auto ext : extensions)
        if (nameInfo->Extension.Buffer != nullptr &&
            _wcsnicmp(ext, nameInfo->Extension.Buffer, wcslen(ext)) == 0)
            return true;

    return false;
}
```

FilterFileNameInformation::Parse calls FltParseFileNameInformation to get convenient access to the extension. If the file extension is found, true is returned, indicating this file is interesting.

Back at the post-create callback - we're not done. If the file has the right extension, but happens to use an alternate stream, we're not interested - we're only interested in the default stream (what is considered the "real" file's contents). The previous call to FltParseFileNameInformation within ShouldBackupFile gives back the stream, if any:

```
if (fileNameInfo->Stream.Length > 0)
    return FLT_POSTOP_FINISHED_PROCESSING;
```

Finally, we are ready to allocate our file context and initialize it. Allocation requires a call to FltAllocateContext and specifying the context type and other details:

```

FileContext* context;
auto status = FltAllocateContext(FltObjects->Filter,
    FLT_FILE_CONTEXT, sizeof(FileContext), PagedPool,
    (PFLT_CONTEXT*)&context);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to allocate file context (0x%08X)\n", status));
    return FLT_POSTOP_FINISHED_PROCESSING;
}

```

FltAllocateContext allocates a context with the required size and returns a pointer to the allocated memory. PFLT_CONTEXT is just a void* - we can cast it to whatever type we need. The returned context memory is not zeroed out, so all members must be initialized properly.

Now we can initialize the context and set it on the file object:

```

context->Written = FALSE;
context->Lock.Init();
context->BackupTime.QuadPart = 0;

// 
// set file context
//
status = FltSetFileContext(FltObjects->Instance,
    FltObjects->FileObject,
    FLT_SET_CONTEXT_REPLACE_IF_EXISTS,
    context, nullptr);

```

Why do we need this context in the first place? A typical client opens a file for write access and then calls WriteFile potentially multiple times. Before the first call to WriteFile the driver should back up the existing content of the file. This is why we need the boolean Written field - to make sure we make the backup just once before the first write attempt. This flag starts as FALSE and will turn TRUE after the first write operation. This turn of events is depicted in Figure 12-10.

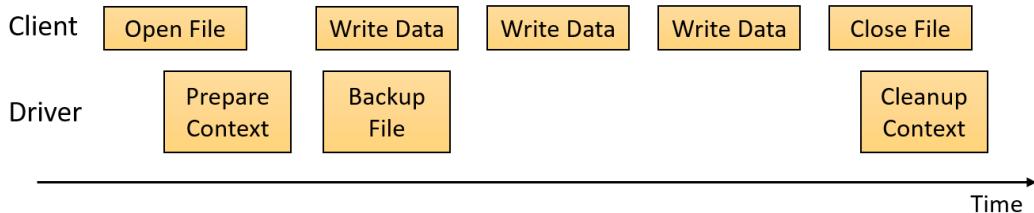


Figure 12-10: Client and driver operations for common write sequence

Why do we need a mutex? We need some synchronization in an unlikely, but possible case, where more than one thread within the client process write to the same file at roughly the same time. In such a case, we need to make sure we make a single backup of the data, otherwise our backup may become

corrupted. In all examples thus far where we needed such synchronization, we used a fast mutex, but here we're using a standard mutex. Why? The driver will perform I/O operations while holding the (fats) mutex. I/O operations can only be performed at IRQL PASSIVE_LEVEL (0). An acquired fast mutex raises IRQL to APC_LEVEL (1), which will cause a deadlock if I/O APIs are used.

The deadlock occurs because I/O operations are completed by sending a special kernel APC to the original thread. If that thread is waiting on a fast mutex (at IRQL APC_LEVEL=1), it will never run the APC (all APCs are blocked while the IRQL is APC_LEVEL), thus a deadlock.

The Mutex class is the same one shown in chapter 6 (part of the KTL as well). The BackupTime member is zeroed out and will be modified when we back up the file. In the current version of the driver, this information is not used, but it could be written to another stream in the file as some sort of "metadata".

Finally, FltReleaseContext must be called, which if all is well, sets the internal reference count of the context to 1 (+1 for allocate, +1 for set, -1 for release):

```
FltReleaseContext(context);

return FLT_POSTOP_FINISHED_PROCESSING;
}
```

The Pre-Write Callback

The pre-write callback's job is to make a copy of the file data just before the actual write operation is allowed to go through; this is why a pre-callback is needed here, otherwise in the post-callback the operation would already have completed.

We start by retrieving the file's context. If it does not exist, this means our post-create callback deemed the file uninteresting and we can just move on:

```
FLT_PREOP_CALLBACK_STATUS
OnPreWrite(PFLT_CALLBACK_DATA Data,
PCFLT RELATED OBJECTS FltObjects, PVOID*) {
    //
    // get the file context if exists
    //
    FileContext* context;

    auto status = FltGetFileContext(FltObjects->Instance,
        FltObjects->FileObject,
        (PFLT_CONTEXT*)&context);
    if (!NT_SUCCESS(status) || context == nullptr) {
```

```

//  

// no context, continue normally  

//  

return FLT_PREOP_SUCCESS_NO_CALLBACK;  

}

```

Once we have a context, we need to make a copy of the file data just once before the first write operation. First, we acquire the mutex and check the written flag from the context. if it's false, then a backup was not created yet and we call a helper function to make the backup:

```

do {  

    Locker locker(context->Lock);  

    if (context->Written) {  

        //  

        // already written, nothing to do  

        //  

        break;  

    }  

    FilterFileNameInformation name(Data);  

    if (!name)  

        break;  

    status = BackupFile(&name->Name, FltObjects);  

    if (!NT_SUCCESS(status)) {  

        KdPrint(("Failed to backup file! (0x%X)\n", status));  

    }  

    else {  

        KeQuerySystemTimePrecise(&context->BackupTime);  

    }  

    context->Written = TRUE;  

} while (false);  

FltReleaseContext(context);  

//  

// don't prevent the write regardless of any error  

//  

return FLT_PREOP_SUCCESS_NO_CALLBACK;  

}

```

`Locker` is the usual RAII type to acquire a synchronization object in its constructor and release in the destructor.

The `BackupFile` helper function is the key to making all this work. One might think that making a file copy is just an API away; unfortunately, it's not. There is no "CopyFile" function in the kernel. The `CopyFile` user mode API is a non-trivial function that does quite a bit of work to make copy work. Part of it is reading bytes from the source file and writing to the destination file. But that's not enough in the general case. First, there may be multiple streams to copy (in case of NTFS). Second, there is the question of the security descriptor from the original file which also needs to be copied in certain cases (refer to the documentation for `CopyFile` to get all details).

The bottom line is that there is no single `CopyFile` we can use, and we'll have to create our own file copy operation. Fortunately, we just need to copy a single file stream - the default stream to another stream inside the same physical file as our backup stream. Here is the start of our `BackupFile` function:

```
NTSTATUS BackupFile(PUNICODE_STRING path, PCFLT_RELATED_OBJECTS FltObjects) {
    //
    // get source file size
    //
    LARGE_INTEGER fileSize;
    auto status = FsRtlGetFileSize(FltObjects->FileObject, &fileSize);
    if (!NT_SUCCESS(status) || fileSize.QuadPart == 0)
        return status;
```

`FsRtlGetFileSize` is a simple API that returns the size of a file (default NTFS stream).

This API is recommended whenever the file size is needed given a `FILE_OBJECT` pointer. The alternative would be calling `ZwQueryInformationFile` or `FltQueryInformationFile` to obtain the file size (it has many other types of information it can retrieve). The `Zw` variant is less desirable as it requires a file handle and in some cases can cause a deadlock.

The route we'll take is to open two handles - one (source) handle pointing to the original file (with the default stream to back up) and the other (target) handle to the backup stream. Then, we'll read from the source and write to the target. This is conceptually simple, but as is often the case in kernel programming, the devil is in the details.

Now we're ready to open the source file with `FltCreateFileEx`. It's important **not** to use `ZwCreateFile`, so that the I/O requests are sent to the driver below this driver and not to the top of the file system driver stack:

```
HANDLE hSourceFile = nullptr;
HANDLE hTargetFile = nullptr;
PFILE_OBJECT sourceFile = nullptr;
PFILE_OBJECT targetFile = nullptr;
IO_STATUS_BLOCK ioStatus;
void* buffer = nullptr;

do {
    OBJECT_ATTRIBUTES sourceFileAttr;
```

```

InitializeObjectAttributes(&sourceFileAttr, path,
    OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);

status = FltCreateFileEx(
    FltObjects->Filter,           // filter object
    FltObjects->Instance,         // filter instance
    &hSourceFile,                // resulting handle
    &sourceFile,                 // resulting file object
    GENERIC_READ | SYNCHRONIZE,   // access mask
    &sourceFileAttr,              // object attributes
    &ioStatus,                   // resulting status
    nullptr, FILE_ATTRIBUTE_NORMAL, // allocation size, file attributes
    FILE_SHARE_READ | FILE_SHARE_WRITE, // share flags
    FILE_OPEN,                    // create disposition
    FILE_SYNCHRONOUS_IO_NONALERT | FILE_SEQUENTIAL_ONLY, // sync I/O
    nullptr, 0,                  // extended attributes, EA length
    IO_IGNORE_SHARE_ACCESS_CHECK); // flags

if (!NT_SUCCESS(status))
    break;

```

Before calling `FltCreateFileEx`, just like other APIs requiring a name, an `OBJECT_ATTRIBUTES` structure must be initialized properly with the file name provided to `BackupFile`. This is the default file stream that is about to change by a write operation and that's why we're making the backup. The important arguments in the call are:

- filter and instance objects, which provide the necessary information for the call to go to the next lower layer filter (or the file system) rather than go to the top of the file system stack.
- the returned handle, in `hSourceFile`.
- the returned `FILE_OBJECT`, to be used with `FltReadFile`.
- the access mask set to `GENERIC_READ` and `SYNCHRONIZE`.
- the create disposition, in this case indicating the file must exist (`FILE_OPEN`).
- the create options are set to `FILE_SYNCHRONOUS_IO_NONALERT` indicating synchronous operations through the resulting file handle. The `SYNCHRONIZE` access mask flag is required for synchronous operations to work.
- the flag `IO_IGNORE_SHARE_ACCESS_CHECK` is important, because the file in question was already opened by the client that most likely opened it with no sharing allowed. So we ask the file system to ignore share access checks for this call.

Read the documentation of `FltCreateFileEx` to gain a better understanding of all the various options this function provides.

Next we need to open or create the backup stream within the same file. We'll name the backup stream “:backup” and use another call to FltCreateFileEx to get a handle and file object to the target file:

```

//  

// open target file  

//  

UNICODE_STRING targetFileName;  

const WCHAR backupStream[] = L":backup";  

targetFileName.MaximumLength = path->Length + sizeof(backupStream);  

targetFileName.Buffer = (WCHAR*)ExAllocatePool2(POOL_FLAG_PAGED,  

    targetFileName.MaximumLength, DRIVER_TAG);  

if (targetFileName.Buffer == nullptr) {  

    status = STATUS_NO_MEMORY;  

    break;  

}  

RtlCopyUnicodeString(&targetFileName, path);  

RtlAppendUnicodeToString(&targetFileName, backupStream);  

OBJECT_ATTRIBUTES targetFileAttr;  

InitializeObjectAttributes(&targetFileAttr, &targetFileName,  

    OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);  

status = FltCreateFileEx(  

    FltObjects->Filter,           // filter object  

    FltObjects->Instance,         // filter instance  

    &hTargetFile,                // resulting handle  

    &targetFile,                 // resulting file object  

    GENERIC_WRITE | SYNCHRONIZE, // access mask  

    &targetFileAttr,              // object attributes  

    &ioStatus,                   // resulting status  

    nullptr, FILE_ATTRIBUTE_NORMAL,  

    0,                           // share flags  

    FILE_OVERWRITE_IF,           // create disposition  

    FILE_SYNCHRONOUS_IO_NONALERT | FILE_SEQUENTIAL_ONLY,  

    nullptr, 0,                // extended attributes, EA length  

    0);   // flags  

ExFreePool(targetFileName.Buffer);  

if (!NT_SUCCESS(status)) {  

    //  

    // could fail if a restore operation is in progress  

    //  

    break;
```

```
}
```

The file name is built by concatenating the base file name and the backup stream name. It is opened for write access (GENERIC_WRITE) and overwrites any data that may be present (FILE_OVERWRITE_IF).

With these file objects in hand, we can start from the source and writing to the target. A simple approach would be to allocate a buffer with the file size, and do the work with a single read and a single write. This could be problematic, however, if the file is very large, possibly causing memory allocation to fail.



There is also the risk of creating a backup for a very large file, possibly consuming lots of disk space. For this kind of driver, backup should probably be avoided when a file is too large (configurable in the Registry for instance) or avoid backup if the remaining disk space would be below a certain threshold (again could be configurable). This is left as an exercise for the reader.

A better option would be to allocate a relatively small buffer and just loop around until all the files chunks have been copied. This is the approach we'll use. First, allocate a buffer:

```
ULONG size = 1 << 20; // 1 MB
buffer = ExAllocatePool2(POOL_FLAG_PAGED, size, DRIVER_TAG);
if (!buffer) {
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
}
```

Now the loop:

```
ULONG bytes;
auto saveSize = fileSize;
while (fileSize.QuadPart > 0) {
    status = FltReadFile(
        FltObjects->Instance,
        sourceFile,           // source file object
        nullptr,              // byte offset
        (ULONG)min((LONGLONG)size, fileSize.QuadPart), // # of bytes
        buffer,
        0,                   // flags
        &bytes,               // bytes read
        nullptr, nullptr);   // no callback
    if (!NT_SUCCESS(status))
        break;
    //
```

```

// write to target file
//
status = FltWriteFile(
    FltObjects->Instance,
    targetFile,           // target file
    nullptr,              // offset
    bytes,                // bytes to write
    buffer,               // data to write
    0,                    // flags
    nullptr,              // written
    nullptr, nullptr);   // no callback

if (!NT_SUCCESS(status))
    break;

//
// update byte count remaining
//
fileSize.QuadPart -= bytes;
}

```

The loop keeps going as long as there are bytes to transfer. We start with the file size and then decrement it for every chunk transferred. The function that do the actual work are FltReadFile and FltWriteFile. We could have used ZwReadFile and ZwWriteFile (we have handles), but this is slightly less efficient. Notice the offsets are set to NULL, because we're using synchronous I/O, where the file objects track a file pointer automatically.

When all is done, there is one last thing to do. Since we may be overwriting a previous backup (that may have been larger than this one), we must set the end of file pointer to the current offset:

```

FILE_END_OF_FILE_INFORMATION info;
info.EndOfFile = saveSize;
status = FltSetInformationFile(FltObjects->Instance,
    targetFile, &info, sizeof(info), FileEndOfFileInformation);
} while (false);

```

Lastly, we need to cleanup everything:

```

    if (buffer)
        ExFreePool(buffer);
    if (hSourceFile)
        FltClose(hSourceFile);
    if (hTargetFile)
        FltClose(hTargetFile);
    if (sourceFile)
        ObDereferenceObject(sourceFile);
    if (targetFile)
        ObDereferenceObject(targetFile);

    return status;
}

```

The Post-Cleanup Callback

Why do we need another callback? Our context is attached to a file, which means it will only be deleted when the file is deleted, which may never happen. We need to free the context when the file is closed by the client.

There are two operations that seem relevant here, `IRP_MJ_CLOSE` and `IRP_MJ_CLEANUP`. The close operation seems most intuitive as it's supposed to be called when the last handle to the file is closed. However, due to caching, this does not always happen soon enough. A better approach is to handle `IRP_MJ_CLEANUP`, which essentially means the file object is no longer needed, as the last handle is has been closed but there is still outstanding reference to the file object itself. This is a good time to free our context (if exists).

A post-cleanup callback is similar to any other post-callback. We need to check if a context exists, and if so - delete it:

```

FLT_POSTOP_CALLBACK_STATUS
OnPostCleanup(PFLT_CALLBACK_DATA Data, PCFLT RELATED OBJECTS FltObjects,
    PVOID, FLT_POST_OPERATION_FLAGS Flags) {
    UNREFERENCED_PARAMETER(Flags);
    UNREFERENCED_PARAMETER(Data);

    FileContext* context;

    auto status = FltGetFileContext(FltObjects->Instance,
        FltObjects->FileObject, (PFLT_CONTEXT*)&context);
    if (!NT_SUCCESS(status) || context == nullptr) {
        //
        // no context, continue normally
        //
        return FLT_POSTOP_FINISHED_PROCESSING;
    }
}

```

```
}

FltReleaseContext(context);
FltDeleteContext(context);

return FLT_POSTOP_FINISHED_PROCESSING;
}
```

Testing the Driver

We can test the driver by deploying it to a target system as usual, and then manipulating files with one of the tracked extensions.

In the following example, I created a *hello.txt* file “Hello, world!”, saved the file, and then changed the contents to “Goodbye, world!” and saved again. Figure 12-11 shows the *Streams* command line tool, available with the source for this chapter:

```
C:\Demos>type c:\Temp\hello.txt
goodbye, world!

C:\Demos>streams -d c:\Temp\hello.txt
:backup:$DATA (15 bytes)
68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21 0D 0A          hello, world!..
```



The *Streams* tool uses the `FindFirstStreamW` and `FindNextStreamW` to iterate over the streams within a file. Check out the source code for more information.

Restoring Backups

How can we restore a backup? We need to copy the “:backup” stream contents over the “normal” file contents. Unfortunately, the `CopyFile` API cannot do this, as it does not accept alternate streams. Let’s write a utility to do the work.

We’ll create a new console application project named *Restore*. We’ll add the following `#includes` to the *Restore.cpp* file:

```
#include <Windows.h>
#include <stdio.h>
#include <string>
```

The `main` function should accept the file name as a command line argument:

```
int wmain(int argc, const wchar_t* argv[]) {
    if (argc < 2) {
        printf("Usage: Restore <filename>\n");
        return 0;
    }
}
```

Next, we'll open two files, one pointing to the “:backup” stream and the other to the “normal” file. Then, we'll copy in chunks, similarly to the driver's BackupFile code - but in user mode. The Error function just prints the provided text and whatever is returned from GetLastError:

```
std::wstring stream(argv[1]);
stream += L":backup";

HANDLE hSource = CreateFile(stream.c_str(), GENERIC_READ,
    FILE_SHARE_READ, nullptr, OPEN_EXISTING, 0, nullptr);
if (hSource == INVALID_HANDLE_VALUE)
    return Error("Failed to locate backup");

HANDLE hTarget = CreateFile(argv[1], GENERIC_WRITE, 0,
    nullptr, OPEN_EXISTING, 0, nullptr);
if (hTarget == INVALID_HANDLE_VALUE)
    return Error("Failed to locate file");

LARGE_INTEGER size;
if (!GetFileSizeEx(hSource, &size))
    return Error("Failed to get file size");

ULONG bufferSize = (ULONG)min((LONGLONG)1 << 21, size.QuadPart);
void* buffer = VirtualAlloc(nullptr, bufferSize,
    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
if (!buffer)
    return Error("Failed to allocate buffer");

DWORD bytes;
while (size.QuadPart > 0) {
    if (!ReadFile(hSource, buffer,
        (DWORD)(min((LONGLONG)bufferSize, size.QuadPart)),
        &bytes, nullptr))
        return Error("Failed to read data");

    if (!WriteFile(hTarget, buffer, bytes, &bytes, nullptr))
        return Error("Failed to write data");
    size.QuadPart -= bytes;
}
```

}



Extend the driver to store an additional stream in the file with the backup time and date.

File Copying with a Section Object

There is another to perform the copy operation happening in `BackupFile` using *Section* objects (known in user-mode as *Memory Mapped Files*). A full discussion of section objects is beyond the scope of this chapter, but here are the basics.

A *section* object can map a file (or part of a file) to memory, allowing access to the file's data using memory APIs, which are more flexible than I/O APIs. Furthermore, no buffers need to be allocated - the mapping to physical memory and write back (if needed) is managed automatically by the memory manager, and this is generally more efficient than using explicit I/O APIs.

Sections also support sharing memory between processes, or between a kernel driver and user-mode processes. Refer to the documentation of memory mapped files for more information.

Let's write an alternative `BackupFile` function that uses a session mapped to the input file for reading purposes. Using the same idea for writing to the target is also possible, and is left as an exercise for the reader.

We start the function as in the original:

```
NTSTATUS
BackupFileWithSection(PUNICODE_STRING path, PCFLT_RELATED_OBJECTS FltObjects) {
    LARGE_INTEGER fileSize;
    auto status = FsRtlGetFileSize(FltObjects->FileObject, &fileSize);
    if (!NT_SUCCESS(status) || fileSize.QuadPart == 0)
        return status;

    HANDLE hSourceFile = nullptr;
    HANDLE hTargetFile = nullptr;
    PFILE_OBJECT sourceFile = nullptr;
    PFILE_OBJECT targetFile = nullptr;
    IO_STATUS_BLOCK ioStatus;
    HANDLE hSection = nullptr;

    do {
        OBJECT_ATTRIBUTES sourceFileAttr;
        InitializeObjectAttributes(&sourceFileAttr, path,
            OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);
```

```
status = FltCreateFileEx(
    FltObjects->Filter,
    FltObjects->Instance,
    &hSourceFile,
    &sourceFile,
    GENERIC_READ | SYNCHRONIZE,
    &sourceFileAttr,
    &iostatus,
    nullptr, FILE_ATTRIBUTE_NORMAL,
    FILE_SHARE_READ | FILE_SHARE_WRITE,
    FILE_OPEN,
    FILE_SYNCHRONOUS_IO_NONALERT | FILE_SEQUENTIAL_ONLY,
    nullptr, 0,
    IO_IGNORE_SHARE_ACCESS_CHECK);
if (!NT_SUCCESS(status))
    break;

UNICODE_STRING targetFileName;
const WCHAR backupStream[] = L":backup";
targetFileName.MaximumLength = path->Length + sizeof(backupStream);
targetFileName.Buffer = (WCHAR*)ExAllocatePool2(POOL_FLAG_PAGED,
    targetFileName.MaximumLength, DRIVER_TAG);
if (targetFileName.Buffer == nullptr) {
    status = STATUS_NO_MEMORY;
    break;
}
RtlCopyUnicodeString(&targetFileName, path);
RtlAppendUnicodeToString(&targetFileName, backupStream);

OBJECT_ATTRIBUTES targetFileAttr;
InitializeObjectAttributes(&targetFileAttr, &targetFileName,
    OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, nullptr);

status = FltCreateFileEx(
    FltObjects->Filter,
    FltObjects->Instance,
    &hTargetFile,
    &targetFile,
    GENERIC_WRITE | SYNCHRONIZE,
    &targetFileAttr,
    &iostatus,
    nullptr, FILE_ATTRIBUTE_NORMAL,
```

```

    0,
    FILE_OVERWRITE_IF,
    FILE_SYNCHRONOUS_IO_NONALERT | FILE_SEQUENTIAL_ONLY,
    nullptr, 0, 0);

ExFreePool(targetFileName.Buffer);
if (!NT_SUCCESS(status)) {
    break;
}

```

Now comes the new stuff. We'll create a section object pointing to the source file:

```

OBJECT_ATTRIBUTES sectionAttributes = RTL_CONSTANT_OBJECT_ATTRIBUTES(
    nullptr, OBJ_KERNEL_HANDLE);
status = ZwCreateSection(&hSection, SECTION_MAP_READ | SECTION_QUERY,
    &sectionAttributes, nullptr,
    PAGE_READONLY, SEC_COMMIT, hSourceFile);
if (!NT_SUCCESS(status))
    break;

```

The section is created for read access, pointing to the source file (last argument). The loop needs to map a view into memory of file chunks (we'll go with 1 MB chunks as before), and then write the data based on the mapped pointer:

```

LARGE_INTEGER offset{};
auto saveSize = fileSize;
PVOID buffer = nullptr;
SIZE_T size = 1 << 20;
while (fileSize.QuadPart > 0) {
    buffer = nullptr;
    SIZE_T bytes = min((LONGLONG)size, fileSize.QuadPart);
    status = ZwMapViewOfSection(hSection, NtCurrentProcess(), &buffer, 0, 0 \
    , &offset, &bytes, ViewUnmap, 0, PAGE_READONLY);
    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "Failed in ZwMapViewOfSection (0x%X)\n", sta \
tus));
        break;
    }

    ULONG written;
    status = FltWriteFile(
        FltObjects->Instance,
        targetFile, nullptr,

```

```

        (ULONG)bytes, buffer,
        0, &written,
        nullptr, nullptr);

ZwUnmapViewOfSection(NtCurrentProcess(), buffer);
if (!NT_SUCCESS(status))
    break;

//
// update count and offset
//
fileSize.QuadPart -= written;
offset.QuadPart += written;
}

FILE_END_OF_FILE_INFORMATION info;
info.EndOfFile = saveSize;
status = FltSetInformationFile(FltObjects->Instance,
    targetFile, &info, sizeof(info), FileEndOfFileInformation);
} while(false);
}

```

ZwMapViewOfSection performs the mapping, returning the pointer to the mapped memory in `buffer`. Notice there is no buffer allocation anywhere - the data is just read directly.

Finally, we have to clean up, which is the same as the original code with the addition of the section handle:

```

if (hSection)
    ZwClose(hSection);
if (hSourceFile)
    FltClose(hSourceFile);
if (hTargetFile)
    FltClose(hTargetFile);
if (sourceFile)
    ObDereferenceObject(sourceFile);
if (targetFile)
    ObDereferenceObject(targetFile);

return status;
}

```

User Mode Communication

We saw in previous chapters one way of communicating between a driver and a user mode client: using `DeviceIoControl`. This is certainly a fine way and works well in many scenarios. One of its

drawbacks is that the user mode client must initiate the communication. If the driver has something to send to a user mode client (or clients), it cannot do so directly. It must store it and wait for the client to ask for the data.

The filter manager provides an alternative mechanism for bi-directional communication between a file system mini-filter and user mode clients, where any side can send information to the other and even wait for a reply.

The mini-filter creates a *filter communication port* object by calling `FltCreateCommunicationPort` to create such a port and register callbacks for client connection and messages. The user mode client connects to the port by calling `FilterConnectCommunicationPort`, receiving a handle to the port.

A mini-filter sends a message to its user mode client(s) with `FltSendMessage`. Conversely, a user mode client calls `FilterGetMessage` to wait until a message arrives, or calls `FilterSendMessage` to send a message to the driver. If the driver is expecting a reply, a user mode client calls `FilterReplyMessage` with the reply.

Creating the Communication Port

The `FltCreateCommunicationPort` function is declared as follows:

```
NTSTATUS FltCreateCommunicationPort (
    _In_ PFLT_FILTER Filter,
    _Outptr_ PFLT_PORT *ServerPort,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_ PVOID ServerPortCookie,
    _In_ PFLT_CONNECT_NOTIFY ConnectNotifyCallback,
    _In_ PFLT_DISCONNECT_NOTIFY DisconnectNotifyCallback,
    _In_opt_ PFLT_MESSAGE_NOTIFY MessageNotifyCallback,
    _In_ LONG MaxConnections);
```

Here is a description of the parameters to `FltCreateCommunicationPort`:

- *Filter* is the opaque pointer returned from `FltRegisterFilter`.
- *ServerPort* is an output opaque handle that is used internally to listen to incoming messages from user mode.
- *ObjectAttributes* is the standard attributes structure that must contain the server port name and a security descriptor that would allow user mode clients to connect (more on this later).
- *ServerPortCookie* is an optional driver-defined pointer that can be used to distinguish between multiple open ports in message callbacks.
- *ConnectNotifyCallback* is a callback the driver must provide, called when a new client connects to the port.
- *DisconnectNotifyCallback* is a callback called when a user mode client disconnects from the port.
- *MessageNotifyCallback* is the callback invoked when a message arrives on the port.

- *MaxConnections* indicates the maximum number of clients that can connect to the port. It must be greater than zero.

A successful call to `FltCreateCommunicationPort` requires the driver to prepare an object attributes and a security descriptor. The simplest security descriptor can be created with `FltBuildDefaultSecurityDescriptor` like so:

```
PSECURITY_DESCRIPTOR sd;
status = FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);
```

The security descriptor is necessary, otherwise no user-mode client would be able to open a handle successfully, as the port is too secure. The object attributes can then be initialized:

```
UNICODE_STRING portName = RTL_CONSTANT_STRING(L"\\\MyPort");
OBJECT_ATTRIBUTES portAttr;
InitializeObjectAttributes(&portAttr, &name,
    OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, sd);
```

The name of the port is in the object manager namespace, viewable with *WinObj* after port creation. The flags must include `OBJ_KERNEL_HANDLE`, otherwise the call fails. Notice the last argument being the security descriptor defined earlier. Now the driver is ready to call `FltCreateCommunicationPort`, typically done after the driver calls `FltRegisterFilter` (because the returned opaque filter object is needed for the call), but before `FltStartFiltering` so the port can be ready when actual filtering starts:

```
PFLT_PORT ServerPort;

status = FltCreateCommunicationPort(FilterHandle, &ServerPort, &portAttr, nullptr\tr,
    PortConnectNotify, PortDisconnectNotify, PortMessageNotify, 1);

// free security descriptor
FltFreeSecurityDescriptor(sd);
```

User Mode Connection

User mode clients call `FilterConnectCommunicationPort` to connect to an open port, declared like so:

```
HRESULT FilterConnectCommunicationPort (
    _In_ LPCWSTR lpPortName,
    _In_ DWORD dwOptions,
    _In_reads_bytes_opt_(wSizeOfContext) LPCVOID lpContext,
    _In_ WORD wSizeOfContext,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _Outptr_ HANDLE *hPort);
```

Here is a quick rundown of the parameters:

- *lpPortName* is the port name (such as “\MyPort”). Note that with the default security descriptor created by the driver, only admin level processes are able to connect.
- *dwOptions* is usually zero, but `FLT_PORT_FLAG_SYNC_HANDLE` in Windows 8.1 and later, indicating the returned handle should work synchronously only. It’s not clear why this is needed since the default usage is synchronous anyway.
- *lpContext* and *wSizeOfContext* support a way to send a buffer to the driver at connection time. This could be used as a means of authentication, for example, where some password or token is sent to the driver and the driver will fail requests to connect that don’t adhere to some predefined authentication mechanism. In a production driver this is generally a good idea, so that unknown clients could not “hijack” the communication port from legitimate clients.
- *lpSecurityAttributes* is the usual user mode `SECURITY_ATTRIBUTES`, typically set to `NULL`.
- *hPort* is the output handle used later by the client to send and receive messages.

This call invokes the driver’s client connection notify callback, declared as follows:

```
NTSTATUS PortConnectNotify(
    _In_ PFLT_PORT ClientPort,
    _In_opt_ PVOID ServerPortCookie,
    _In_reads_bytes_opt_(SizeOfContext) PVOID ConnectionContext,
    _In_ ULONG SizeOfContext,
    _Outptr_result_maybenull_ PVOID *ConnectionPortCookie);
```

ClientPort is a unique handle to the client’s port which the driver must keep around and use whenever it needs to communicate with that client. *ServerPortCookie* is the same one the driver specified in `FltCreateCommunicationPort`. The *ConnectionContex* and *SizeOfContex* parameters contain the optional buffer sent by the client. Finally, *ConnectionPortCookie* is an optional value the driver can return as representing this client; it’s passed in the client disconnect and message notification routines.

If the driver agrees to accept the client’s connection it returns `STATUS_SUCCESS`. Otherwise, the client will receive a failure `HRESULT` back at `FilterConnectCommunicationPort`.

Once the call to `FilterConnectCommunicationPort` succeeds, the client can start communicating with the driver, and vice-versa.

Sending and Receiving Messages

A mini-filter driver can send a message to clients with `FltSendMessage` declared like so:

```
NTSTATUS
FLTAPI
FltSendMessage (
    _In_ PFLT_FILTER Filter,
    _In_ PFLT_PORT *ClientPort,
    _In_ PVOID SenderBuffer,
    _In_ ULONG SenderBufferLength,
    _Out_ PVOID ReplyBuffer,
    _Inout_opt_ PULONG ReplyLength,
    _In_opt_ PLARGE_INTEGER Timeout);
```

The first two parameters should be known by now. The driver can send any buffer described by *SenderBuffer* with length *SenderBufferLength*. Typically the driver will define some structure in a common header file the client can include as well so that it can correctly interpret the received buffer. Optionally, the driver may expect a reply, and if so, the *ReplyBuffer* parameter should be non-NULL with the maximum reply length stored in *ReplyLength*. Finally, *Timeout* indicates how long the driver is willing to wait the message to reach the client (and wait for a reply, if one is expected). The timeout has the usual format, described here for convenience:

- if the pointer is NULL, the driver is willing to wait indefinitely.
- if the value is positive, then it's an absolute time in 100nsec units since January 1, 1601 at midnight.
- if the value is negative, it's relative time - the most common case - in the same 100nsec units. For example, to specify one second, specify -100000000. As another example, to specify *x* milliseconds, multiply *x* by -10000.

The driver should be careful not to specify NULL from within a callback, because it means that if the client is currently not listening, the thread blocks until it does, which may never happen. It's better to specify some limited value. Even better, if a reply is not needed right away, a work item can be used to send the message and wait for longer if needed (refer to chapter 6 for more information on work items, although the filter manager has its own work item APIs).

From the client's perspective, it can wait for a message from the driver with `FilterGetMessage`, specifying the port handle received when connecting, a buffer and size for the incoming message and an OVERLAPPED structure than can be used to make the call asynchronous (non-blocking). The received buffer always has a header of type `FILTER_MESSAGE_HEADER`, followed by the actual data sent by the driver. `FILTER_MESSAGE_HEADER` is defined like so:

```
typedef struct _FILTER_MESSAGE_HEADER {
    ULONG ReplyLength;
    ULONGLONG MessageId;
} FILTER_MESSAGE_HEADER, *PFILTER_MESSAGE_HEADER;
```

If a reply is expected, *ReplyLength* indicates how many bytes at most are expected. The *MessageId* field allows distinguishing between messages, which the client should use if it calls `FilterReplyMessage`.

A client can initiate its own message with `FilterSendMessage` which eventually lands in the driver's callback registered in `FltCreateCommunicationPort`. `FilterSendMessage` can specify a buffer comprising the message to send and an optional buffer for a reply that may be expected from the mini-filter.

See the documentation for `FilterSendMessage` and `FilterReplyMessage` for the complete details.

Enhanced Backup Driver

Let's enhance the file backup driver to send notifications to a user mode client when a file has been backed up. The source code is part of a different project, *KBackup2*, but the target file name is still *KBackup.sys*.

First, we'll define additional global variables to hold state related to the communication port:

```
PFLT_PORT g_Port;
PFLT_PORT g_ClientPort;
```

`g_Port` is the driver's server port and `g_ClientPort` is the client port once connected (we will allow a single client only).

We'll have to modify `DriverEntry` to create the communication port as described in the previous section. Here is the revised `DriverEntry`:

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    auto status = InitMiniFilter(DriverObject, RegistryPath);
    if (!NT_SUCCESS(status))
        return status;

    do {
        UNICODE_STRING name = RTL_CONSTANT_STRING(L"\\" BackupPort");
        PSECURITY_DESCRIPTOR sd;

        status = FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);
        if (!NT_SUCCESS(status))
            break;

        OBJECT_ATTRIBUTES attr;
        InitializeObjectAttributes(&attr, &name,
            OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE, nullptr, sd);
```

```

status = FltCreateCommunicationPort(g_Filter, &g_Port, &attr, nullptr,
    PortConnectNotify, PortDisconnectNotify, PortMessageNotify, 1);

FltFreeSecurityDescriptor(sd);
if (!NT_SUCCESS(status))
    break;

status = FltStartFiltering(g_Filter);
} while (false);

if (!NT_SUCCESS(status)) {
    FltUnregisterFilter(g_Filter);
}

return status;
}

```

The driver only allows a single client to connect to the port (the last 1 to `FltCreateCommunicationPort`) , which is quite common when a mini-filter works in tandem with a user mode service.

The `PortConnectNotify` callback is called when a client attempts to connect. Our driver simply stores the client's port and returns success:

```

NTSTATUS PortConnectNotify(
    PFLT_PORT ClientPort, PVOID ServerPortCookie,
    PVOID ConnectionContext, ULONG SizeOfContext,
    PVOID* ConnectionPortCookie) {
    UNREFERENCED_PARAMETER(ServerPortCookie);
    UNREFERENCED_PARAMETER(ConnectionContext);
    UNREFERENCED_PARAMETER(SizeOfContext);
    UNREFERENCED_PARAMETER(ConnectionPortCookie);

    g_ClientPort = ClientPort;

    return STATUS_SUCCESS;
}

```

When the client disconnects, the `PortDisconnectNotify` callback is invoked. It's important to close the client port at that time, otherwise the mini-filter will never be unloaded:

```
void PortDisconnectNotify(PVOID ConnectionCookie) {
    UNREFERENCED_PARAMETER(ConnectionCookie);

    FltCloseClientPort(g_Filter, &g_ClientPort);
    g_ClientPort = nullptr;
}
```

In this driver we don't expect any messages from the client - the driver is the only one sending messages - so the PostMessageNotify callback has an empty implementation.

Now we need to actually send a message when a file has been backed up successfully. For this purpose, we'll define a message structure common to the driver and the client in its own header file, *BackupCommon.h*:

```
struct FileBackupPortMessage {
    USHORT FileNameLength;
    WCHAR FileName[1];
};
```

The message contains the file name length and the file name itself. The message does not have a fixed size and depends on the file name length. In the pre-write callback after a file was backed up successfully we need to allocate and initialize a buffer to send:

```
if (g_ClientPort) {      // client connected
    USHORT nameLen = name->Name.Length;
    USHORT len = sizeof(FileBackupPortMessage) + nameLen;
    auto msg = (FileBackupPortMessage*)ExAllocatePool2(
        POOL_FLAG_PAGED, len, DRIVER_TAG);
    if (msg) {
        msg->FileNameLength = nameLen / sizeof(WCHAR);
        RtlCopyMemory(msg->FileName, name->Name.Buffer, nameLen);
        LARGE_INTEGER timeout;
        timeout.QuadPart = -10000 * 100; // 100 msec
        FltSendMessage(g_Filter, &g_ClientPort, msg, len,
            nullptr, nullptr, &timeout);
        ExFreePool(msg);
    }
}
```

First we check if any client is connected, and if so we allocate a buffer with the proper size to include the file name. Then we copy it to the buffer (*RtlCopyMemory*, the same as *memcpy*) before sending it on its way with a limited timeout to be received.

Finally, in the filter's unload routine we must close the filter communication port:

```
NTSTATUS BackupUnload(FLT_FILTER_UNLOAD_FLAGS Flags) {
    UNREFERENCED_PARAMETER(Flags);

    FltCloseCommunicationPort(g_Port);
    FltUnregisterFilter(g_Filter);

    return STATUS_SUCCESS;
}
```

The User Mode Client

Let's build a simple client that opens the port and listens to messages of files being backed up. We'll create a new console application named *BackupMon*, and add the following #includes:

```
#include <Windows.h>
#include <fltUser.h>
#include <stdio.h>
#include <string>
#include "..\KBackup2\BackupCommon.h"
```

fltuser.h is the user mode header where the FilterXxx functions are declared (they are not part of *windows.h*). In the cpp file we must add the import library for these functions:

```
#pragma comment(lib, "fltlb")
```



Alternatively, this library can be added in the project's properties in the Linker node, under Input. Putting this in the source file is easier and more robust, since changes to the project properties will not effect the setting. Without this library, “unresolved external” linker errors will show up.

Our `main` function needs first to open the communication port:

```
int main() {
    HANDLE hPort;
    auto hr = FilterConnectCommunicationPort(L"\\" BackupPort",
        0, nullptr, 0, nullptr, &hPort);
    if (FAILED(hr)) {
        printf("Error connecting to port (HR=0x%08X)\n", hr);
        return 1;
}
```

Now we can allocate a buffer for incoming messages and loop around forever waiting for messages. Once a message is received, we'll send it for handling:

```

BYTE buffer[1 << 12];      // 4 KB
auto message = (FILTER_MESSAGE_HEADER*)buffer;

for (;;) {
    hr = FilterGetMessage(hPort, message, sizeof(buffer), nullptr);
    if (FAILED(hr)) {
        printf("Error receiving message (0x%08X)\n", hr);
        break;
    }
    HandleMessage(buffer + sizeof(FILTER_MESSAGE_HEADER));
}

CloseHandle(hPort);

return 0;
}

```

The buffer here is allocated statically because the message just includes a file name, so a 4KB buffer should be more than enough. Once a message is received, we pass the message body to a helper function, `HandleMessage`, being careful to skip the always-present header.

All that's left now is to do something with the data:

```

void HandleMessage(const BYTE* buffer) {
    auto msg = (FileBackupPortMessage*)buffer;
    std::wstring filename(msg->FileName, msg->FileNameLength);

    printf("file backed up: %ws\n", filename.c_str());
}

```

We build the string based on the pointer and length (fortunately, the C++ standard `wstring` class has such a convenient constructor). This is important because the string is not necessarily NULL-terminated (although we could have zeroed out the buffer before each message receipt, thus making sure zeros are present at the end of the string).



The client application must be running elevated for the port open to succeed.

Debugging

Debugging file system mini-filter is no different than debugging any other kernel driver. However, the *Debugging Tools for Windows* package has a special extension DLL, `fltkd.dll`, with specific commands

to help with mini-filters. This DLL is not one of the default loaded extension DLLs, so the commands must be used with their “full name” that includes the *fltkd* prefix and the command. Alternatively, the DLL can be loaded explicitly with the .load command and then the commands can be directly used.

Table 12-3 shows some of the commands from *fltkd* with a brief description.

Table 12-3: fltkd.dll debugger commands

Command	Description
!help	shows the command list with brief descriptions
!filters	shows information on all loaded mini-filters
!filter	shows information for the specified filter address
!instance	shows information for the specified instance address
!volumes	shows all volume objects
!volume	shows detailed information on the specified volume address
!portlist	shows the server ports for the specified filter
!port	shows information on the specified client port

Here is an example session using some of the above commands:

```
2: kd> .load fltkd
2: kd> !filters

Filter List: ffff8b8f55bf60c0 "Frame 0"
  FLT_FILTER: ffff8b8f579d9010 "bindflt" "409800"
    FLT_INSTANCE: ffff8b8f62ea8010 "bindflt Instance" "409800"
  FLT_FILTER: ffff8b8f5ba06010 "CldFlt" "409500"
    FLT_INSTANCE: ffff8b8f550aaa20 "CldFlt" "180451"
  FLT_FILTER: ffff8b8f55ceca20 "WdFilter" "328010"
    FLT_INSTANCE: ffff8b8f572d6b30 "WdFilter Instance" "328010"
    FLT_INSTANCE: ffff8b8f575d5b30 "WdFilter Instance" "328010"
    FLT_INSTANCE: ffff8b8f585d2050 "WdFilter Instance" "328010"
    FLT_INSTANCE: ffff8b8f58bde010 "WdFilter Instance" "328010"
  FLT_FILTER: ffff8b8f5cdc6320 "storqosflt" "244000"
  FLT_FILTER: ffff8b8f550aca20 "wcifs" "189900"
    FLT_INSTANCE: ffff8b8f551a6720 "wcifs Instance" "189900"
  FLT_FILTER: ffff8b8f576cab30 "FileCrypt" "141100"
  FLT_FILTER: ffff8b8f550b2010 "luafv" "135000"
    FLT_INSTANCE: ffff8b8f550ae010 "luafv" "135000"
  FLT_FILTER: ffff8b8f633e8c80 "FileBackup" "100200"
    FLT_INSTANCE: ffff8b8f645df290 "FileBackup Instance" "100200"
```

```
FLT_INSTANCE: fffff8b8f5d1a7880 "FileBackup Instance" "100200"
FLT_FILTER: fffff8b8f58ce2be0 "npsvctrig" "46000"
    FLT_INSTANCE: fffff8b8f55113a60 "npsvctrig" "46000"
    FLT_FILTER: fffff8b8f55ce9010 "Wof" "40700"
        FLT_INSTANCE: fffff8b8f572e2b30 "Wof Instance" "40700"
        FLT_INSTANCE: fffff8b8f5bae7010 "Wof Instance" "40700"
    FLT_FILTER: fffff8b8f55ce8520 "FileInfo" "40500"
        FLT_INSTANCE: fffff8b8f579cea20 "FileInfo" "40500"
        FLT_INSTANCE: fffff8b8f577ee8a0 "FileInfo" "40500"
        FLT_INSTANCE: fffff8b8f58cc6730 "FileInfo" "40500"
        FLT_INSTANCE: fffff8b8f5bae2010 "FileInfo" "40500"
2: kd> !portlist fffff8b8f633e8c80

FLT_FILTER: fffff8b8f633e8c80
    Client Port List           : Mutex (fffff8b8f633e8ed8) List [fffff8b8f5949b7a0-f\ffff8b8f5949b7a0] mCount=1
    FLT_PORT_OBJECT: fffff8b8f5949b7a0
        FilterLink             : [fffff8b8f633e8f10-fffff8b8f633e8f10]
        ServerPort              : fffff8b8f5b195200
        Cookie                 : 0000000000000000
        Lock                   : (fffff8b8f5949b7c8)
        MsgQ                   : (fffff8b8f5949b800) NumEntries=1 Enabled
        MessageId              : 0x0000000000000000
        DisconnectEvent         : (fffff8b8f5949b8d8)
        Disconnected            : FALSE

2: kd> !volumes

Volume List: fffff8b8f55bf6140 "Frame 0"
    FLT_VOLUME: fffff8b8f579cb6b0 "\Device\Mup"
        FLT_INSTANCE: fffff8b8f572d6b30 "WdFilter Instance" "328010"
        FLT_INSTANCE: fffff8b8f579cea20 "FileInfo" "40500"
    FLT_VOLUME: fffff8b8f57af8530 "\Device\HarddiskVolume4"
        FLT_INSTANCE: fffff8b8f62ea8010 "bindflt Instance" "409800"
        FLT_INSTANCE: fffff8b8f575d5b30 "WdFilter Instance" "328010"
        FLT_INSTANCE: fffff8b8f551a6720 "wcifs Instance" "189900"
        FLT_INSTANCE: fffff8b8f550aaa20 "CldFlt" "180451"
        FLT_INSTANCE: fffff8b8f550ae010 "luafv" "135000"
        FLT_INSTANCE: fffff8b8f645df290 "FileBackup Instance" "100200"
        FLT_INSTANCE: fffff8b8f572e2b30 "Wof Instance" "40700"
        FLT_INSTANCE: fffff8b8f577ee8a0 "FileInfo" "40500"
    FLT_VOLUME: fffff8b8f58cc4010 "\Device\NamedPipe"
        FLT_INSTANCE: fffff8b8f55113a60 "npsvctrig" "46000"
```

```
FLT_VOLUME: fffff8b8f58ce8060 "\Device\Mailslot"
FLT_VOLUME: fffff8b8f58ce1370 "\Device\HarddiskVolume2"
    FLT_INSTANCE: fffff8b8f585d2050 "WdFilter Instance" "328010"
    FLT_INSTANCE: fffff8b8f58cc6730 "FileInfo" "40500"
FLT_VOLUME: fffff8b8f5b227010 "\Device\HarddiskVolume1"
    FLT_INSTANCE: fffff8b8f58bde010 "WdFilter Instance" "328010"
    FLT_INSTANCE: fffff8b8f5d1a7880 "FileBackup Instance" "100200"
    FLT_INSTANCE: fffff8b8f5bae7010 "Wof Instance" "40700"
    FLT_INSTANCE: fffff8b8f5bae2010 "FileInfo" "40500"

2: kd> !volume fffff8b8f57af8530

FLT_VOLUME: fffff8b8f57af8530 "\Device\HarddiskVolume4"
    FLT_OBJECT: fffff8b8f57af8530 [04000000] Volume
        RundownRef : 0x000000000000008b2 (1113)
        PointerCount : 0x000000001
        PrimaryLink : [fffff8b8f58cc4020-fffff8b8f579cb6c0]
        Frame : fffff8b8f55bf6010 "Frame 0"
        Flags : [00000164] SetupNotifyCalled EnableNameCaching Fi\
IteratorAttached +100!!
        FileSystemType : [00000002] FLT_FSTYPE_NTFS
        VolumeLink : [fffff8b8f58cc4020-fffff8b8f579cb6c0]
        DeviceObject : fffff8b8f573cab60
        DiskDeviceObject : fffff8b8f572e7b80
        FrameZeroVolume : fffff8b8f57af8530
        VolumeInNextFrame : 0000000000000000
        Guid : "\??\Volume{5379a5de-f305-4243-a3ec-311938a2df19}\

        CDODeviceName : "\Ntfs"
        CDODriverName : "\FileSystem\Ntfs"
        TargetedOpenCount : 1104
        Callbacks : (fffff8b8f57af8650)
        ContextLock : (fffff8b8f57af8a38)
        VolumeContexts : (fffff8b8f57af8a40) Count=0
        StreamListCtrls : (fffff8b8f57af8a48) rCount=29613
        FileListCtrls : (fffff8b8f57af8ac8) rCount=22668
        NameCacheCtrl : (fffff8b8f57af8b48)
        InstanceList : (fffff8b8f57af85d0)

            FLT_INSTANCE: fffff8b8f62ea8010 "bindflt Instance" "409800"
            FLT_INSTANCE: fffff8b8f575d5b30 "WdFilter Instance" "328010"
            FLT_INSTANCE: fffff8b8f551a6720 "wcifs Instance" "189900"
            FLT_INSTANCE: fffff8b8f550aaa20 "CldFlt" "180451"
            FLT_INSTANCE: fffff8b8f550ae010 "luafv" "135000"
```

```

FLT_INSTANCE: fffff8b8f645df290 "FileBackup Instance" "100200"
FLT_INSTANCE: fffff8b8f572e2b30 "Wof Instance" "40700"
FLT_INSTANCE: fffff8b8f577ee8a0 "FileInfo" "40500"

2: kd> !instance fffff8b8f5d1a7880

FLT_INSTANCE: fffff8b8f5d1a7880 "FileBackup Instance" "100200"
  FLT_OBJECT: fffff8b8f5d1a7880 [01000000] Instance
    RundownRef           : 0x0000000000000000 (0)
    PointerCount          : 0x00000001
    PrimaryLink           : [fffff8b8f5bae7020-fffff8b8f58bde020]
  OperationRundownRef   : fffff8b8f639c61b0
    Number                : 3
    PoolToFree             : fffff8b8f65aad590
    OperationsRefs         : fffff8b8f65aad5c0 (0)
      PerProcessor Ref[0]  : 0x0000000000000000 (0)
      PerProcessor Ref[1]  : 0x0000000000000000 (0)
      PerProcessor Ref[2]  : 0x0000000000000000 (0)
    Flags                 : [00000000]
    Volume                : fffff8b8f5b227010 "\Device\HarddiskVolume1"
    Filter                : fffff8b8f633e8c80 "FileBackup"
    TrackCompletionNodes  : fffff8b8f5f3f3cc0
    ContextLock            : (fffff8b8f5d1a7900)
    Context                : 0000000000000000
    CallbackNodes          : (fffff8b8f5d1a7920)
    VolumeLink             : [fffff8b8f5bae7020-fffff8b8f58bde020]
    FilterLink             : [fffff8b8f633e8d50-fffff8b8f645df300]

```

Exercises

1. Write a file system mini-filter that prevents file deletion from processes running certain image name (e.g. “cmd.exe”).
2. Extend the file system mini-filter from the previous item, but instead of deleting files, moves the files to the recycle bin.
3. Extend the file backup driver with the ability to choose the directories where backups will be created.
4. Extend the File Backup driver to include multiple backups, limited by some rule, such as file size, date or maximum number of backup copies.
5. Modify the File Backup driver to back up only the changed data instead of the entire file.
6. Come up with your own ideas for a file system mini-filter driver!

Summary

This chapter was all about file system mini-filters - powerful drivers capable of intercepting any and all file system activity. Mini-filters are a big topic, and this chapter should get you started on this interesting and powerful journey. You can find more information in the WDK documentation, and the WDK samples on *Github*.

In the next chapter, we'll switch gears to look at the *Windows Filtering Platform* (WFP), used for network filtering.

Chapter 13: The Windows Filtering Platform

The *Windows Filtering Platform* (WFP) provides flexible ways to control network filtering. It exposes user-mode and kernel-mode APIs, that interact with several layers of the networking stack. Some configuration and control is available directly from user-mode, without requiring any kernel-mode code (although it does require administrator-level access). WFP replaces older network filtering technologies, such as *Transport Driver Interface* (TDI) filters some types of NDIS filters.

If examining network packets (and even modification) is required, or blocking is needed based on some logic, a kernel-mode *Callout* driver can be written, which is what we'll be concerned with in this chapter. We'll begin with an overview of the main pieces of WFP, look at some user-mode code examples for configuring filters before diving into building a simple Callout driver that can use some logic to block access to the network.

This chapter is an introduction to WFP, as full treatment would probably require its own book.

In this chapter:

- **WFP Overview**
 - **The WFP API**
 - **User Mode Examples**
 - **Callout Drivers**
 - **Demo: Callout Driver**
 - **Demo: User-Mode Client**
 - **Summary**
-

WFP Overview

WFP is comprised of user-mode and kernel-mode components. A very high-level architecture is depicted in figure 13-1.

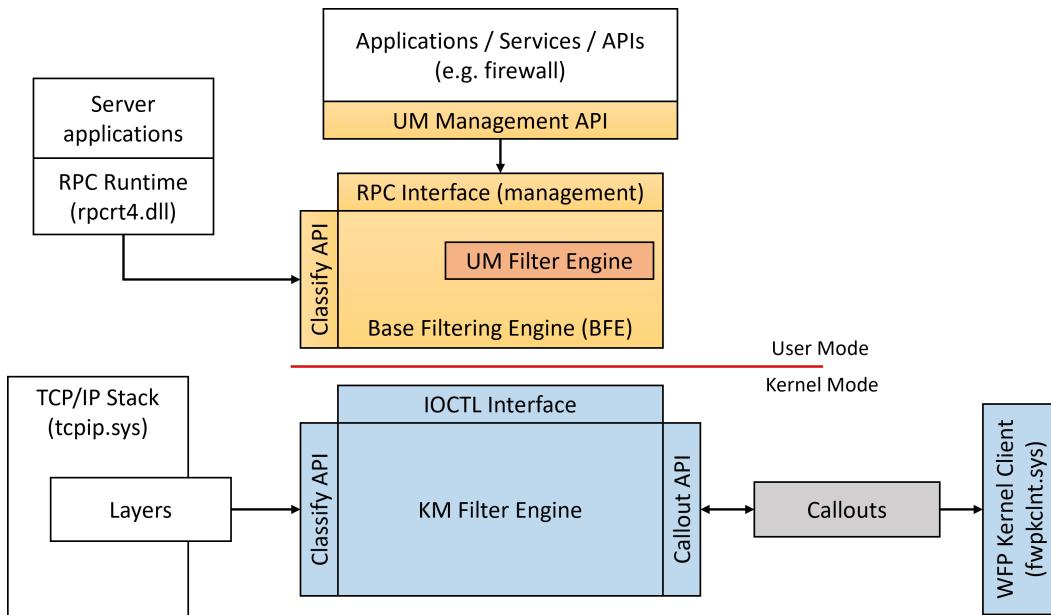


Figure 13-1: WFP Architecture

In user-mode, the WFP manager is the *Base Filtering Engine* (BFE), which is a service implemented by *bfe.dll* and hosted in a standard *svchost.exe* instance. It implements the WFP user-mode API, essentially managing the platform, talking to its kernel counterpart when needed. We'll examine some of these APIs in the next section.

User-mode applications, services and other components can utilize this user-mode management API to examine WFP objects state, and make changes, such as adding or deleting filters. A classic example of such "user" is the Windows Firewall, which is normally controllable by leveraging the *Microsoft Management Console* (MMC) that is provided for this purpose (see figure 13-2), but using these APIs from other applications is just as effective.

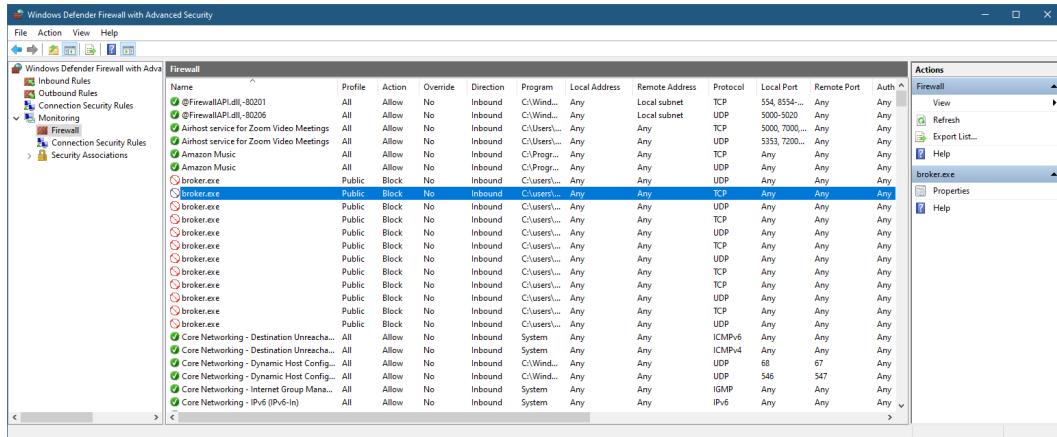


Figure 13-2: Windows Firewall MMC

The kernel-mode filter engine exposes various logical layers, where filters (and callouts) can be attached. Layers represent locations in the network processing of one or more packets. The TCP/IP driver makes calls to the WFP kernel engine so that it can decide which filters (if any) should be “invoked”.

For filters, this means checking the conditions set by the filter against the current request. If the conditions are satisfied, the filter’s action is applied. Common actions include blocking a request from being further processed, allowing the request to continue without further processing in this layer, continuing to the next filter in this layer (if any), and invoking a callout driver. Callouts can perform any kind of processing, such as examining and even modifying packet data.

The relationship between layers, filters, and callouts is depicted in figure 13-3.

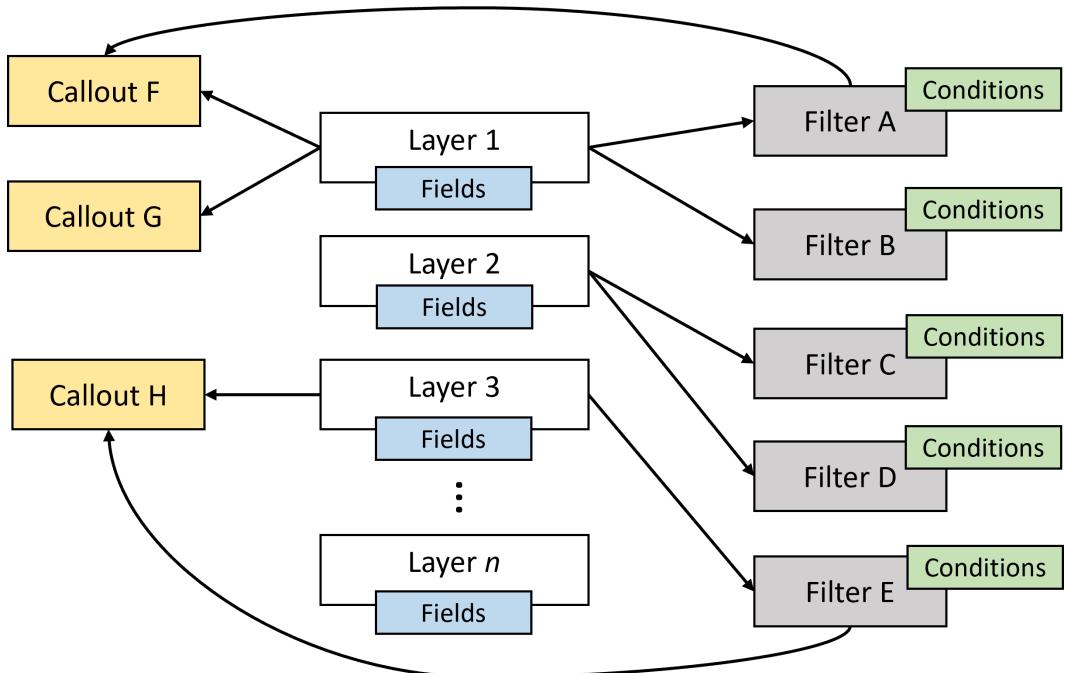


Figure 13-3: Layers, Filters and Callouts

As you can see in figure 13-3, each layer can have zero or more filters, and zero or more callouts. The number and meaning of the layers is fixed and provided out of the box by Windows. On most system, there are about 100 layers. Many of the layers are sets of pairs, where one is for IPv4 and the other (identical in purpose) is for IPv6.

The *WFP Explorer* tool I created provides some insight into what makes up WFP. Running the tool and selecting *View/Layers* from the menu (or clicking the *Layers* tool bar button) shows a view of all existing layers (figure 13-4).

You can download the *WFP Explorer* tool from its Github repository

(<https://github.com/zodiacon/WFPExplorer>) or the *AllTools* repository (<https://github.com/zodiacon/AllTools>). The screenshots shown may be slightly different as the tool may evolve after these screenshots were taken.

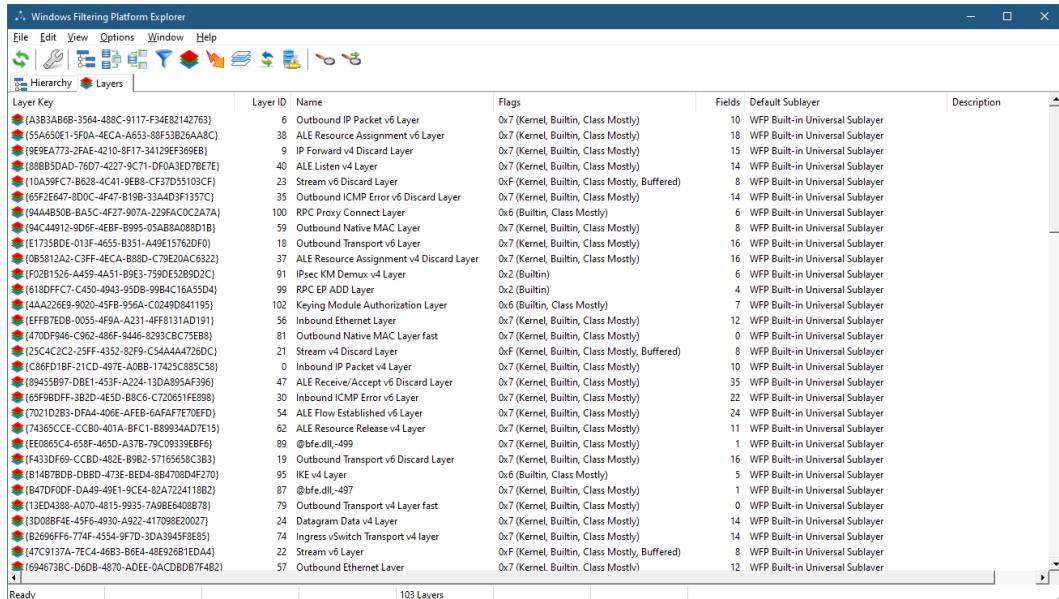


Figure 13-4: Layers in WFP Explorer

Each layer is uniquely identified by a GUID. Its Layer ID is used internally by the kernel engine as an identifier rather than the GUID, as it's smaller and so is faster (layer IDs are 16-bit only). Most layers have fields that can be used by filters to set conditions for invoking their actions. Double-clicking a layer shows its properties. Figure 13-5 shows the general properties of an example layer. Notice it has 382 filters and 2 callouts attached to it. Clicking the *Fields* tab shows the fields available in this layer, that can be used by filters to set conditions (figure 13-6).

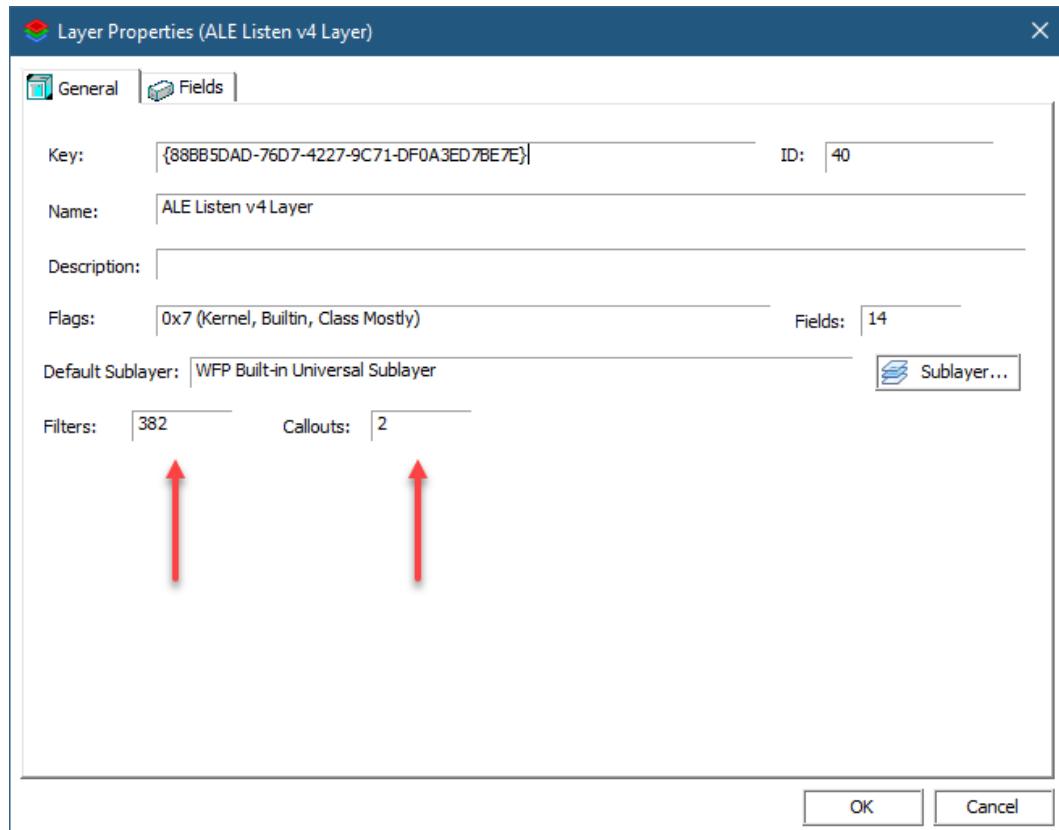


Figure 13-5: A Layer's general properties

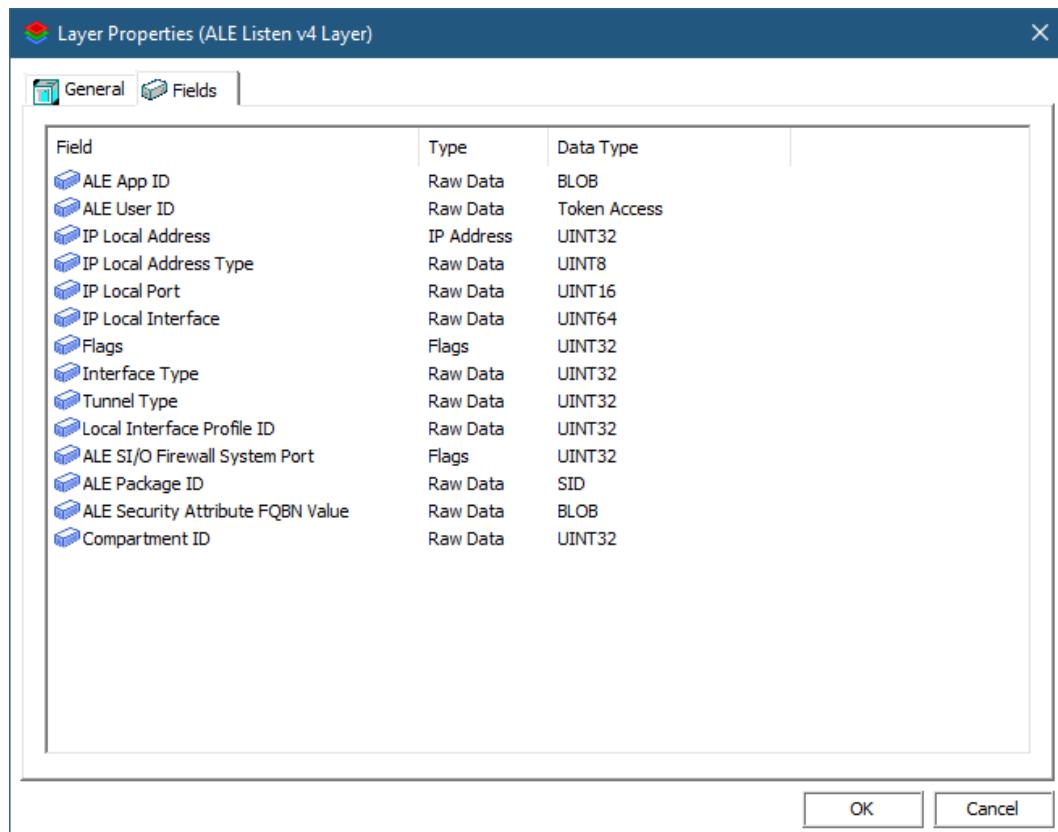


Figure 13-6: A Layer's fields

The meaning of the various layers, and the meaning of the fields for the layers are all documented in the official WFP documentation.

The currently existing filters can be viewed in *WFP Explorer* by selecting *Filters* from the *View* menu (figure 13-7). Layers cannot be added or removed, but filters can. Management code (user or kernel) can add and/or remove filters dynamically while the system is running. Figure 16-7 shows that on the system the tool is running on there are currently 2978 filters.

Windows Filtering Platform Explorer											
File	Edit	View	Options	Window	Help	Hierarchy	Filters				
Filter Key	Filter Name	Layer	Weight	Effective Weight	Filter Id	Conditions	Action	Action Filter/Callout	Flags		
(2E5C5C798-811D-4B09-A8E1-5AF2D708F35E)	File and Printer Sharing (L...	ALE Receive/Accept v4 Layer	0x9	0x3238C00000000000	0x28783	7 Permit	(None)	0x40 (Indexed)			
(1325A8a4-7065-4B91-B380-4C187CCB8873)	Foxit PDF CEF	ALE Receive/Accept v4 Layer	0xA	0x4a00000000000000	0x17424	2 Block	(None)	0x40 (Indexed)			
(AABF8935-255E-472A-9584-79B6BEE5F57)	HNS Container Networking -	ALE Receive/Accept v4 Layer	0x9	0x9032000000000000	0x17516	4 Permit	(None)	0x40 (Indexed)			
(67255FAF-E564-4408-961F-2D00C3556E99)	HNS Container Networking -	ALE Listen v4 Layer	0x9	0x9000000000000000	0x17418	3 Permit	(None)	0x40 (Indexed)			
(8120CE86-1774-4531-8551-9711B6AD0840)	File and Printer Sharing (Ech...	ALE Receive/Accept v4 Layer	0x9	0x9032000000000000	0x17802	4 Permit	(None)	0x40 (Indexed)			
(175953A5-E907-4441-A8C6-7E1ED0469A7)	File and Printer Sharing (Ech...	ALE Receive/Accept v4 Layer	0x9	0x9040000000000000	0x18850	5 Permit	(None)	0x40 (Indexed)			
(1900903C-8D70-40B4-A08A-90210C9E8073)	Airhost service for Zoom V...	ALE Receive/Accept v4 Layer	0x9	0x9003200000000000	0x17812	3 Permit	(None)	0x40 (Indexed)			
(787556A2-4A6B-4494-A81D-F78EC04CF11)	HNS Container Networking -	ALE Listen v4 Layer	0x9	0x9000000000000000	0x17761	3 Permit	(None)	0			
(7A94693C-110B-4147-950A-0E45A41D0480)	@FirewallAPI.dll - 0x2010	ALE Receive/Accept v4 Layer	0x9	0x9055412000000000	0x28747	7 Permit	(None)	0x40 (Indexed)			
(A0B1770C-91B8-CF11CF65238)	Core Networking - Time Esc...	ALE Receive/Accept v5 Layer	0x9	0x9000040000000000	0x17367	5 Permit	(None)	0x40 (Indexed)			
(DF24567E-CEDD-4616-A1B0-CEBDEA330D73)	Allow inbound firewall L...	ALE Receive/Accept v4 Layer	(Empty)	0x9000040000000000	0x17CD5	5 Permit	(None)	0x40 (Indexed)			
(2490E123-6759-4A14-98A8-9E381A5933E4)	HNS Container Networking -	ALE Receive/Accept v6 Layer	0x9	0x90000E0000000000	0x17850	4 Permit	(None)	0x40 (Indexed)			
(4AE8741C-641A-4882-97A7-EFC3BCF99087)	HNS Container Networking -	ALE Receive/Accept v6 Layer	0x9	0x90000E0000000000	0x17A02	4 Permit	(None)	0x40 (Indexed)			
(91C78877-11F4-4688-9FF3-E7A2E522CD09)	Allot Grouping to receive r...	ALE Receive/Accept v4 Layer	(Empty)	0x3800000000000000	0x17CA3	5 Permit	(None)	0x40 (Indexed)			
(84DE414B-80E6-4421-A04D-6355939C6F65)	HNSDefend Outbound for TCP	ALE Connect v4 Layer	(Empty)	0x1800000000000000	0x17099	3 Permit	(None)	0x40 (Indexed)			
(BCBA3C015-A09-4293-95D6-A7257AB5B007)	HNS Container Networking -	ALE Receive/Accept v6 Layer	0x9	0x9002000000000000	0x2869C	3 Permit	(None)	0x40 (Indexed)			
(1807053-98E0-4B62-9EF9-A2B07E830172)	vmaxima	ALE Receive/Accept v6 Layer	0xA	0x6059100000000000	0x177F8	2 Block	(None)	0x40 (Indexed)			
(9898759C-17C0-4E84-BD4-2088527C4646)	AppContainerBootImageFilter	ALE Receive/Accept v6 Layer	0x9	0xFFFFFFFF00000000	0x18394	1 Permit	(None)	0x01 (Persistent)			
(C1C09F8D-6636-4101-85DC-60F3FE41C02)	zoom.exe	ALE Receive/Accept v6 Layer	0xA	0x00000C0000000000	0x17A6E	2 Block	(None)	0x40 (Indexed)			
(EB18EE6D-8446-4C04-A399-7CD104ED095)	broker.exe	ALE Listen v4 Layer	0xA	0x0000000000000000	0x17725	1 Permit	(None)	0			
(F50B886A-4074-4783-A54F-C8CBDE8367A2)	WinDbg! engine process (...)	ALE Resource Assignment v6 Layer	0x9	0x9000F00000000000	0x17663	2 Permit	(None)	0			
(FF599E09-3849-4A1C-8924-EF6ACCE2864)	HP Smart	ALE Listen v4 Layer	0x9	0x9000000000000000	0x17892	2 Callout Terminat...	WFP_Built-in_Edge...	0			
(2926E609-E726-4362-A018-1481A8875519)	HNS Container Networking -	ALE Receive/Accept v4 Layer	(Empty)	0x9000000000000000	0x17786	4 Permit	(None)	0x40 (Indexed)			
(8F535262-5F1C-4841-8772-B10E3583D616)	Rivet IpV4 Outbound Transport	Outbound Transport v4 Layer	(Empty)	0x1800000000000000	0x18000	0 Callout Terminat...	p4 Egress Trans...	0			
(C755A4B18-92E8-4D99-9761-852B10C35507)	WinDbg! engine process (...)	ALE Listen v6 Layer	(Empty)	0x9000000000000000	0x17557	1 Permit	(None)	0			
(F0379071-8E5C-4047-A080-2E88751734C8)	HNS Container Networking -	ALE Receive/Accept v4 Layer	0x9	0x9000000000000000	0x17500	4 Permit	(None)	0x40 (Indexed)			
(E60A24B2-4E4F-4C5D-89C9-C718B70D8A77)	Print/Printer Sharing (SM...	ALE Receive/Accept v6 Layer	0x9	0x9040000000000000	0x20678	5 Permit	(None)	0x40 (Indexed)			
(15883E98-372D-4B51-8881-D66957939D)	Core Networking - Parallel...	ALE Receive/Accept v6 Layer	0x9	0x9000000000000000	0x17360	6 Permit	(None)	0x40 (Indexed)			
(4717B003-3709-4A00-8789-74C5C8027718)	Xbox Game Bar	ALE Receive/Accept v6 Layer	0x9	0x9000000000000000	0x17774	4 Permit	(None)	0x40 (Indexed)			
(F50D707F-F904-47D0-8477-75A1D040C018)	Allow outgoing WSD from P...	ALE Connect v4 Layer	(Empty)	0x1A133F0000000000	0x178C6	5 Permit	(None)	0x40 (Indexed)			
(35C74590-2CC2-4B2B-B4C6-88E7F2037777)	Allow outgoing WSD from P...	ALE Connect v4 Layer	(Empty)	0x1311C00000000000	0x28930	7 Permit	(None)	0x40 (Indexed)			
(11C10411-02D-444F-8CE4-E68E61BDB8A)	eclipse	ALE Listen v4 Layer	0xA	0x0000000000000000	0x17903	1 Permit	(None)	0			

Figure 13-7: Filters in WFP Explorer

Each filter is uniquely identified by a GUID, and just like layers has a “shorter” id (64-bit) that is used by the kernel engine to more quickly compare filter IDs when needed. Since multiple filters can be assigned to the same layer, some kind of ordering must be used when assessing filters. This is where the filter’s *weight* comes into play. A weight is a 64-bit value that is used to sort filters by priority. As you can see in figure 13-7, there are two weight properties - *weight* and *effective weight*. Weight is what is specified when adding the filter, but effective weight is the actual one used. There are three possible values to set for weight:

- A value between 0 and 15 is interpreted by WFP as a weight index, which simply means that the effective weight is going to start with 4 bits having the specified weight value and generate the other 60 bit. For example, if the weight is set to 5, then the effective weight is going to be between 0x5000000000000000 and 0x5FFFFFFFFFFFFF.
- An empty value tells WFP to generate an effective weight somewhere in the 64-bit range.
- A value above 15 is taken as is to become the effective weight.



What is an “empty” value? The weight is not really a number, but a FWP_VALUE type can hold all sorts of values, including holding no value at all (empty).

Double-clicking a filter in WFP Explorer shows its general properties, as shown in figure 13-8.

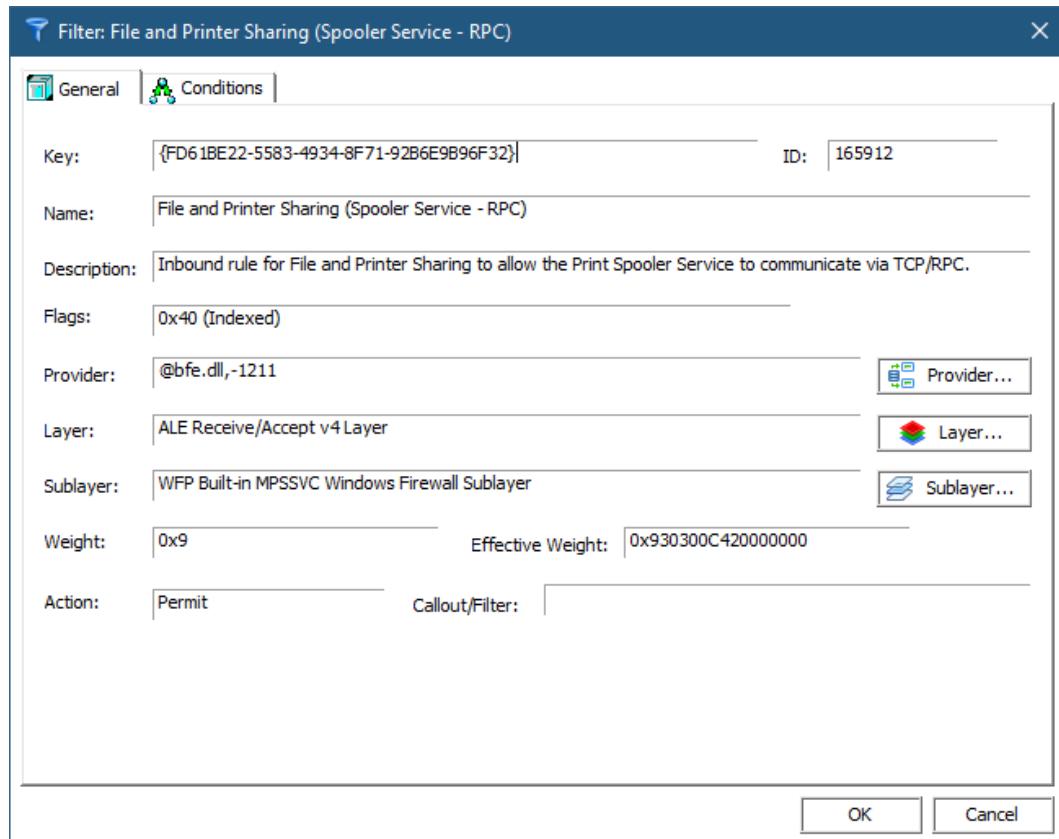


Figure 13-8: A filter's general properties

The *Conditions* tab shows the conditions this filter is configured with (figure 13-9). When all the conditions are met, the action of the filter is going to fire.

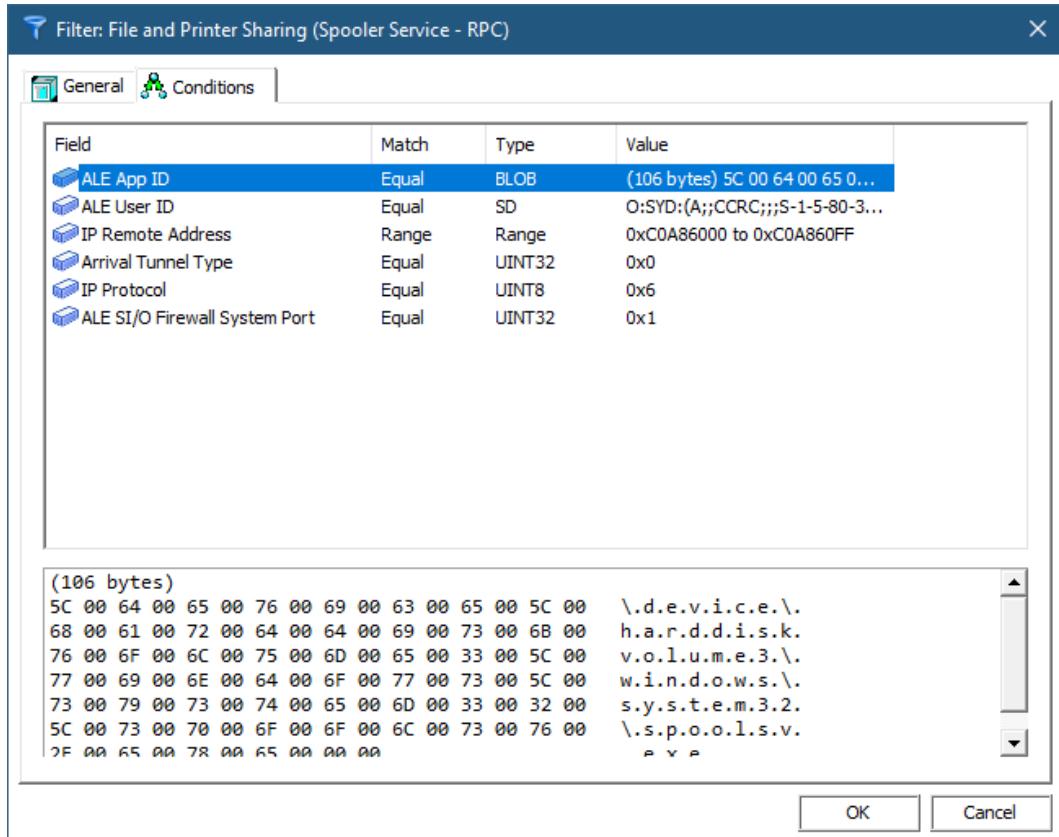


Figure 13-9: A filter's conditions

The list of fields used by a filter must be a subset of the fields exposed by the layer this filter is attached to. There are six conditions shown in figure 13-9 out of the possible 39 fields supported by this layer (“ALE Receive/Accept v4 Layer”). As you can see, there is a lot of flexibility in specifying conditions for fields - this is evident in the matching enumeration, FWPM_MATCH_TYPE:

```
typedef enum FWPM_MATCH_TYPE {
    FWPM_MATCH_EQUAL          = 0,
    FWPM_MATCH_GREATER,
    FWPM_MATCH_LESS,
    FWPM_MATCH_GREATER_OR_EQUAL,
    FWPM_MATCH_LESS_OR_EQUAL,
    FWPM_MATCH_RANGE,
    FWPM_MATCH_FLAGS_ALL_SET,
    FWPM_MATCH_FLAGS_ANY_SET,
    FWPM_MATCH_FLAGS_NONE_SET,
    FWPM_MATCH_EQUAL_CASE_INSENSITIVE,
    FWPM_MATCH_NOT_EQUAL,
```

```
FWP_MATCH_PREFIX,  
FWP_MATCH_NOT_PREFIX,  
FWP_MATCH_TYPE_MAX  
} FWP_MATCH_TYPE;
```

A filter can have zero conditions, which means it's always activated.

At this point, we have enough information to get acquainted with the WFP API.

The WFP API

The WFP API exposes its functionality for user-mode and kernel-mode callers. The header files used are different, to cater for differences in API expectations between user-mode and kernel-mode, but APIs in general are identical. For example, kernel APIs return NTSTATUS, whereas user-mode APIs return a simple LONG, that is the error value that is returned normally from `GetLastError`. Some APIs are provided for kernel-mode only, as they don't make sense for user mode.



The user-mode WFP APIs never set the last error, and always return the error value directly. Zero (ERROR_SUCCESS) means success, while other (positive) values mean failure.
Do not call `GetLastError` when using WFP - just look at the returned value.

WFP functions and structures use a versioning scheme, where function and structure names end with a digit, indicating version. For example, `FWPM_LAYER0` is the first version of a structure describing a layer. At the time of writing, this was the only structure for describing a layer. As a counter example, there are several versions of the function beginning with `FwpmNetEventEnum`: `FwpmNetEventEnum0` (for Vista+), `FwpmNetEventEnum1` (Windows 7+), `FwpmNetEventEnum2` (Windows 8+), `FwpmNetEventEnum3` (Windows 10+), `FwpmNetEventEnum4` (Windows 10 RS4+), and `FwpmNetEventEnum5` (Windows 10 RS5+). This is an extreme example, but there are others with less "versions". You can use any version that matches the target platform. To make it easier to work with these APIs and structures, a macro is defined with the base name that is expanded to the maximum supported version based on the target compilation platform. Here is the declarations for the macro `FwpmNetEventEnum`:

```
DWORD FwpmNetEventEnum0(  
    _In_ HANDLE engineHandle,  
    _In_ HANDLE enumHandle,  
    _In_ UINT32 numEntriesRequested,  
    _Outptr_result_buffer_(*numEntriesReturned) FWPM_NET_EVENT0*** entries,  
    _Out_ UINT32* numEntriesReturned);  
#if (NTDDI_VERSION >= NTDDI_WIN7)  
DWORD FwpmNetEventEnum1(  
    _In_ HANDLE engineHandle,  
    _In_ HANDLE enumHandle,
```

```
_In_ UINT32 numEntriesRequested,
_Outptr_result_buffer_(*numEntriesReturned) FWPM_NET_EVENT1*** entries,
_Out_ UINT32* numEntriesReturned);
#endif // (NTDDI_VERSION >= NTDDI_WIN7)
#if (NTDDI_VERSION >= NTDDI_WIN8)
DWORD FwpmNetEventEnum2(
    _In_ HANDLE engineHandle,
    _In_ HANDLE enumHandle,
    _In_ UINT32 numEntriesRequested,
    _Outptr_result_buffer_(*numEntriesReturned) FWPM_NET_EVENT2*** entries,
    _Out_ UINT32* numEntriesReturned);
#endif // (NTDDI_VERSION >= NTDDI_WIN8)
#if (NTDDI_VERSION >= NTDDI_WINTHRESHOLD)
DWORD FwpmNetEventEnum3(
    _In_ HANDLE engineHandle,
    _In_ HANDLE enumHandle,
    _In_ UINT32 numEntriesRequested,
    _Outptr_result_buffer_(*numEntriesReturned) FWPM_NET_EVENT3*** entries,
    _Out_ UINT32* numEntriesReturned);
#endif // (NTDDI_VERSION >= NTDDI_WINTHRESHOLD)
#if (NTDDI_VERSION >= NTDDI_WIN10_RS4)
DWORD FwpmNetEventEnum4(
    _In_ HANDLE engineHandle,
    _In_ HANDLE enumHandle,
    _In_ UINT32 numEntriesRequested,
    _Outptr_result_buffer_(*numEntriesReturned) FWPM_NET_EVENT4*** entries,
    _Out_ UINT32* numEntriesReturned);
#endif // (NTDDI_VERSION >= NTDDI_WIN10_RS4)
#if (NTDDI_VERSION >= NTDDI_WIN10_RS5)
DWORD FwpmNetEventEnum5(
    _In_ HANDLE engineHandle,
    _In_ HANDLE enumHandle,
    _In_ UINT32 numEntriesRequested,
    _Outptr_result_buffer_(*numEntriesReturned) FWPM_NET_EVENT5*** entries,
    _Out_ UINT32* numEntriesReturned);
#endif // (NTDDI_VERSION >= NTDDI_WIN10_RS5)
```

You can see that the differences in the functions relate to the structures returned as part of these APIs (FWPM_NET_EVENTx). It's recommended you use the macros, and only turn to specific versions if there is a compelling reason to do so.

The WFP APIs adhere to strict naming conventions that make it easier to use. All management functions start with Fwpm (Filtering Windows Platform Management), and all management structures start with FWPM. The function names themselves use the pattern *<Prefix><Object Type><Operation>*,

such as `FwpmFilterAdd` and `FwpmLayerGetByKey`.

It's curious that the prefixes used for functions, structures, and enums start with FWP rather than the (perhaps) expected WFP. I couldn't find a compelling reason for this.

WFP header files start with `fwp` and end with `u` for user-mode or `k` for kernel-mode. For example, `fwpmu.h` holds the management functions for user-mode callers, whereas `fwpmk.h` is the header for kernel callers. Two common files, `fwptypes.h` and `fwpmtypes.h` are used by both user-mode and kernel-mode headers. They are included by the “main” header files.

User-Mode Examples

Before making any calls to specific APIs, a handle to the WFP engine must be opened with `FwpmEngineOpen`:

```
DWORD FwpmEngineOpen0(
    _In_opt_ const wchar_t* serverName, // must be NULL
    _In_     UINT32 authnService,      // RPC_C_AUTHN_DEFAULT
    _In_opt_ SEC_WINNT_AUTH_IDENTITY_W* authIdentity,
    _In_opt_ const FWPM_SESSION0* session,
    _Out_    HANDLE* engineHandle);
```

Most of the arguments have good defaults when `NULL` is specified. The returned handle must be used with subsequent APIs. Once it's no longer needed, it must be closed:

```
DWORD FwpmEngineClose0(_Inout_ HANDLE engineHandle);
```

Enumerating Objects

What can we do with an engine handle? One thing provided with the management API is enumeration. These are the APIs used by *WFP Explorer* to enumerate layers, filters, sessions, and other object types in WFP. The following example displays some details for all the filters in the system (error handling omitted for brevity, the project `wfpfilters` has the full source code):

```
#include <Windows.h>
#include <fwpmu.h>
#include <stdio.h>
#include <string>

#pragma comment(lib, "Fwpuclnt")

std::wstring GuidToString(GUID const& guid) {
    WCHAR sguid[64];
    return ::StringFromGUID2(guid, sguid, _countof(sguid)) ? sguid : L"";
}

const char* ActionToString(FWPM_ACTION const& action) {
    switch (action.type) {
        case FWP_ACTION_BLOCK:           return "Block";
        case FWP_ACTION_PERMIT:          return "Permit";
        case FWP_ACTION_CALLOUT_TERMINATING: return "Callout Terminating";
        case FWP_ACTION_CALLOUT_INSPECTION: return "Callout Inspection";
        case FWP_ACTION_CALLOUT_UNKNOWN:   return "Callout Unknown";
        case FWP_ACTION_CONTINUE:         return "Continue";
        case FWP_ACTION_NONE:            return "None";
        case FWP_ACTION_NONE_NO_MATCH:    return "None (No Match)";
    }
    return "";
}

int main() {
    //
    // open a handle to the WFP engine
    //
    HANDLE hEngine;
    FwpmEngineOpen(nullptr, RPC_C_AUTHN_DEFAULT, nullptr, nullptr, &hEngine);

    //
    // create an enumeration handle
    //
    HANDLE hEnum;
    FwpmFilterCreateEnumHandle(hEngine, nullptr, &hEnum);

    UINT32 count;
    FWPM_FILTER** filters;
    //
    // enumerate filters
}
```

```

// FwpmFilterEnum(hEngine, hEnum,
8192,           // maximum entries,
&filters,        // returned result
&count);        // how many actually returned

for (UINT32 i = 0; i < count; i++) {
    auto f = filters[i];
    printf("%ws Name: %-40ws Id: 0x%016llx Conditions: %2u Action: %s\n",
        GuidToString(f->filterKey).c_str(),
        f->displayData.name,
        f->filterId,
        f->numFilterConditions,
        ActionToString(f->action));
}

// free memory allocated by FwpmFilterEnum
//
FwpmFreeMemory((void**)filters);

// close enumeration handle
//
FwpmFilterDestroyEnumHandle(hEngine, hEnum);

// close engine handle
//
FwpmEngineClose(hEngine);

return 0;
}

```

The enumeration pattern repeat itself with all other WFP object types (layers, callouts, sessions, etc.).



Enumerate all the layers in the system in a similar way.

Adding Filters

Let's see if we can add a filter to perform some useful function. Suppose we want to prevent network access from some process. We can add a filter at an appropriate layer to make it happen. Adding a

filter is a matter of calling `FwpmFilterAdd0`:

```
DWORD FwpmFilterAdd0(
    _In_ HANDLE engineHandle,
    _In_ const FWPM_FILTER0* filter,
    _In_opt_ PSECURITY_DESCRIPTOR sd,
    _Out_opt_ UINT64* id);
```

The main work is to fill a `FWPM_FILTER` structure defined like so:

```
typedef struct FWPM_FILTER0_ {
    GUID filterKey;
    FWPM_DISPLAY_DATA0 displayData;
    UINT32 flags;
    /* [unique] */ GUID *providerKey;
    FWP_BYTE_BLOB providerData;
    GUID layerKey;
    GUID subLayerKey;
    FWP_VALUE0 weight;
    UINT32 numFilterConditions;
    /* [unique][size_is] */ FWPM_FILTER_CONDITION0 *filterCondition;
    FWPM_ACTION0 action;
    /* [switch_is] */ /* [switch_type] */ union
    {
        /* [case()] */ UINT64 rawContext;
        /* [case()] */ GUID providerContextKey;
    } ;
    /* [unique] */ GUID *reserved;
    UINT64 filterId;
    FWP_VALUE0 effectiveWeight;
} FWPM_FILTER0;
```



The weird-looking comments are generated by the *Microsoft Interface Definition Language* (IDL) compiler when generating the header file from an IDL file. Although IDL is most commonly used by *Component Object Model* (COM) to define interfaces and types, WFP uses IDL to define its APIs, even though no COM interfaces are used; just plain C functions. The original IDL files are provided with the SDK, and they are worth checking out, since they may contain developer comments that are not “transferred” to the resulting header files.

Some members in `FWPM_FILTER` are necessary - `layerKey` to indicate the layer to attach this filter, any conditions needed to trigger the filter (`numFilterConditions` and the `filterCondition` array), and the action to take if the filter is triggered (`action` field).

Let's create some code that prevents the Windows Calculator from accessing the network. You may be wondering why would calculator require network access? No, it's not contacting Google to ask for the result of $2+2$. It's using the Internet for accessing current exchange rates (figure 13-10).

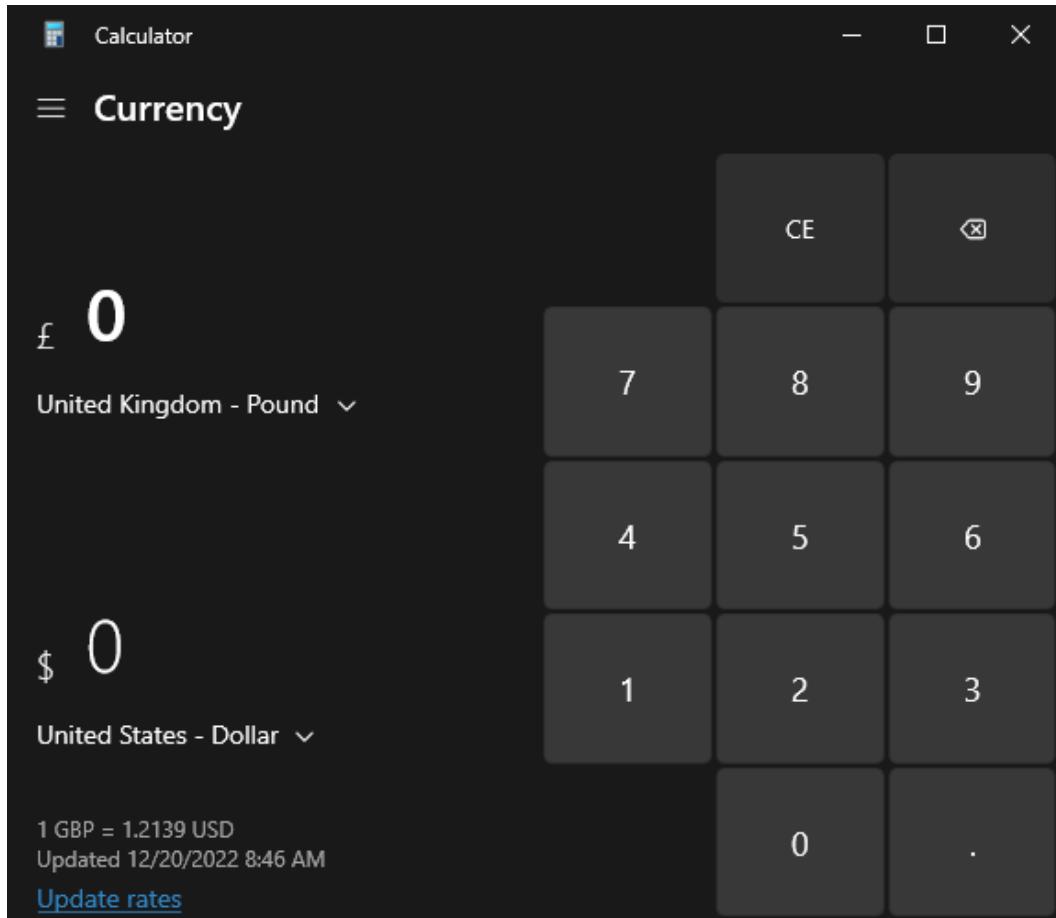


Figure 13-10: Windows Calculator as Currency Converter

Clicking the *Update Rates* button causes Calculator to consult the Internet for the updated exchange rate. We'll add a filter that prevents this.

We'll start as usual by opening handle to the WFP engine as was done in the previous example. Next, we need to fill the FWPM_FILTER structure. First, a nice display name:

```
FWPM_FILTER filter{}; // zero out the structure  
WCHAR filterName[] = L"Prevent Calculator from accessing the web";  
filter.displayData.name = filterName;
```

The name has no functional part - it just allows easy identification when enumerating filters. Now we need to select the layer. We'll also specify the action:

```
filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V4;
filter.action.type = FWP_ACTION_BLOCK;
```

There are several layers that could be used for blocking access, with the above layer being good enough to get the job done. Full description of the provided layers, their purpose and when they are used is provided as part of the WFP documentation.

The last part to initialize is the conditions to use. Without conditions, the filter is always going to be invoked, which will block all network access (or just for some processes, based on its effective weight). In our case, we only care about the application - we don't care about ports or protocols. The layer we selected has several fields, one of which is called *ALE App ID* (ALE stands for *Application Layer Enforcement*) - see figure 13-11.

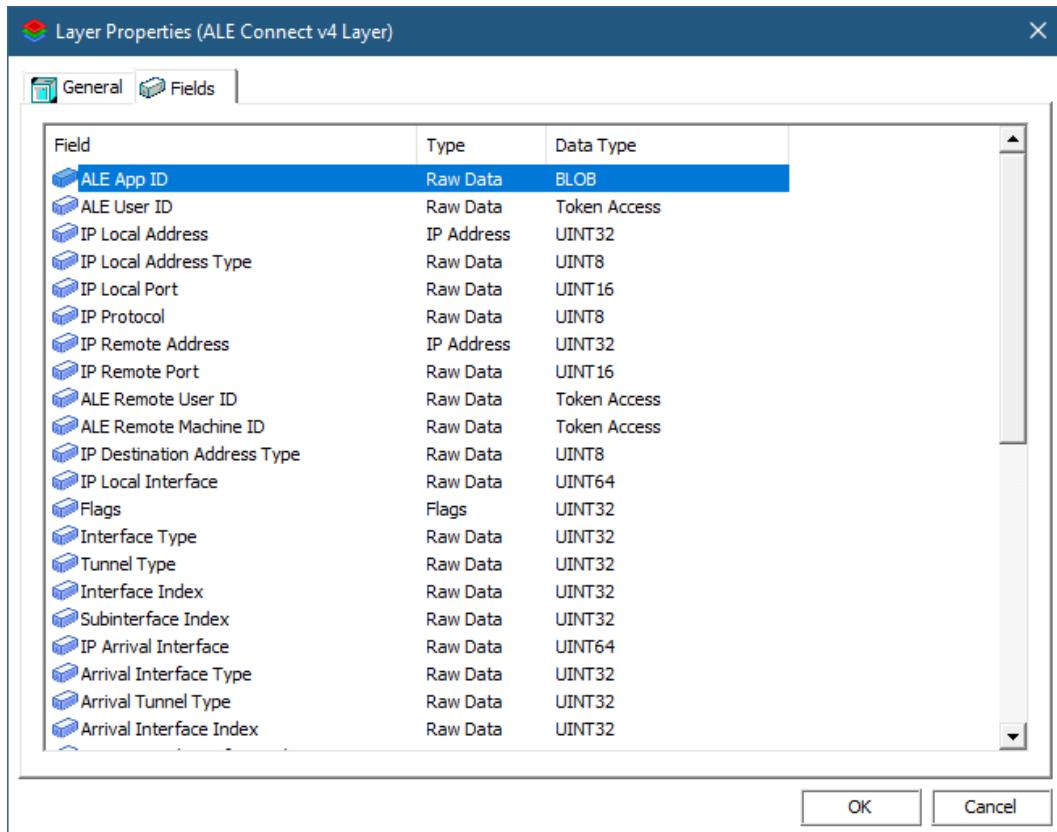


Figure 13-11: ALE Connect v4 Layer fields

This field can be used to identify an executable. To get that ID, we can use `FwpmGetAppIdFromFile`. Here is the code for Calculator's executable:

```
WCHAR filename[] = LR"(C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\  
_11.2210.0.0_x64_8wekyb3d8bbwe\CalculatorApp.exe)";  
FWP_BYTE_BLOB* appId;  
FwpmGetAppIdFromFileNmae(filename, &appId);
```

The code uses the path to the Calculator executable on my system - you should change that as needed because Calculator's version might be different.



A quick way to get the executable path is to run Calculator, open *Process Explorer*, open the resulting process properties, and copy the path from the *Image* tab.

The R"(" and closing parenthesis in the above snippet disables the "escaping" property of backslashes, making it easier to write file paths (C++ 14 feature).

The return value from `FwpmGetAppIdFromFileNmae` is a BLOB that needs to be freed eventually with `FwpmFreeMemory`.

Now we're ready to specify the one and only condition:

```
FWPM_FILTER_CONDITION cond;  
cond.fieldKey = FWPM_CONDITION_ALE_APP_ID;           // field  
cond.matchType = FWP_MATCH_EQUAL;  
cond.conditionValue.type = FWP_BYT_E_BLOB_TYPE;  
cond.conditionValue.byteBlob = appId;  
  
filter.filterCondition = &cond;  
filter.numFilterConditions = 1;
```

The `conditionValue` member of `FWPM_FILTER_CONDITION` is a `FWP_VALUE`, which is a generic way to specify many types of values. It has a `type` member that indicates the member in a big union that should be used. In our case, the type is a BLOB (`FWP_BYT_E_BLOB_TYPE`) and the actual value should be passed in the `byteBlob` union member.

Those familiar with COM may recognize this approach as similar to a VARIANT.

The last step is to add the filter, and repeat the exercise for IPv6, as we don't know how Calculator connects to the currency exchange server (we can find out, but it would be simpler and more robust to just block IPv6 as well):

```
FwpmFilterAdd(hEngine, &filter, nullptr, nullptr);  
  
filter.layerKey = FWPM_LAYER_ALE_AUTH_CONNECT_V6; // IPv6  
FwpmFilterAdd(hEngine, &filter, nullptr, nullptr);
```

We didn't specify any GUID for the filter. This causes WFP to generate a GUID. We didn't specify weight, either. WFP will generate them.

All that's left now is some cleanup:

```
FwpmFreeMemory((void**)appId);  
FwpmEngineClose(hEngine);
```

Running this code (elevated) should and trying to refresh the currency exchange rate with Calculator should fail (figure 13-12). Note that there is no need to restart Calculator - the effect is immediate.

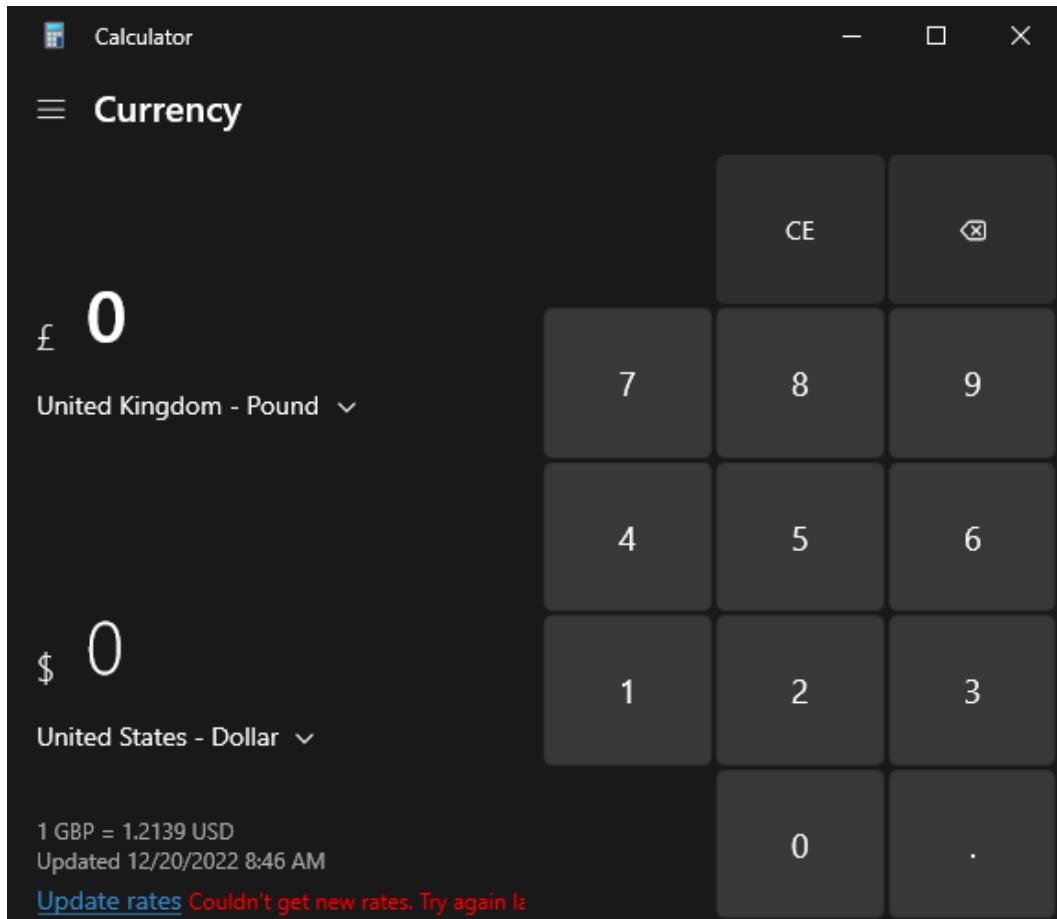


Figure 13-12: Calculator failing to update exchange rate

We can locate the filters added with *WFP Explorer* (figure 13-13):

Filter Key	Filter Name	Layer	Weight	Effective Weight	Filter Id	Conditions	Action
{6326F42E-9E50-4EDA-86EB-4DF2A2C724C7}	Port Scanning Prevention Filter	Outbound ICMP Error v6 Layer	0x8	0x800000000000000C	0x17DC5	4	Permit
{4F3D79F0-5C36-46C3-886F-20481694727B}	Port Scanning Prevention Filter	Outbound ICMP Error v4 Layer	0xFFFFFFF00000002	0xFFFFFFF0000000002	0x17DCC	4	Block
{D5521FCF-C172-48F9-885C-86C946BF6809}	Port Scanning Prevention Filter	Outbound ICMP Error v4 Layer	0x8	0x800000000000000C	0x17DC2	4	Permit
{B34EF5AF-19C9-43D0-BD03-0B066FB53F0}	Port Scanning Prevention Filter	Outbound ICMP Error v4 Layer	0xFFFFFFF00000002	0xFFFFFFF0000000002	0x17DD2	4	Block
{84659E8C3-BDEE-4CB4-8FF5-F98123F40C02}	Prevent Calculator from accessing the web	ALE Connect v6 Layer	(Empty)	0x1000000000	0x2C1A4	1	Block
{C60635D4-E92F-4058-A495-D006F2E57244}	Prevent Calculator from accessing the web	ALE Connect v4 Layer	(Empty)	0x4000000000	0x2C1A3	1	Block
{5862705C-9134-40F6-A7C7-75097C53303D}	pulseaudio.exe	ALE Listen v4 Layer	0xA	0x4000000000000000	0x17931	1	Permit
{992F08F3-15F8-40DC-AD9C-D64A8C321CA}	pulseaudio.exe	ALE Receive/Accept v4 Layer	0xA	0x4003008000000000	0x1792E	2	Block
{F111205DA-7426-4647-8C3D-FE58C0166C0B}	pulseaudio.exe	ALE Resource Assignment v6 Layer	0xA	0xA000FF8000000000	0x1792F	2	Permit
{1A094A827-2FFD-4173-A3D0-A87E32C0642C}	pulseaudio.exe	ALE Resource Assignment v4 Layer	0xA	0xA0007FE000000000	0x1792D	2	Permit

Figure 13-13: Calculator-related filters in WFP Explorer

Double-clicking one of the filters and selecting the *Conditions* tab shows the only condition where the App ID is revealed to be the full path of the executable in device form (figure 13-14). Of course,

you should not take any dependency on this format, as it may change in the future.

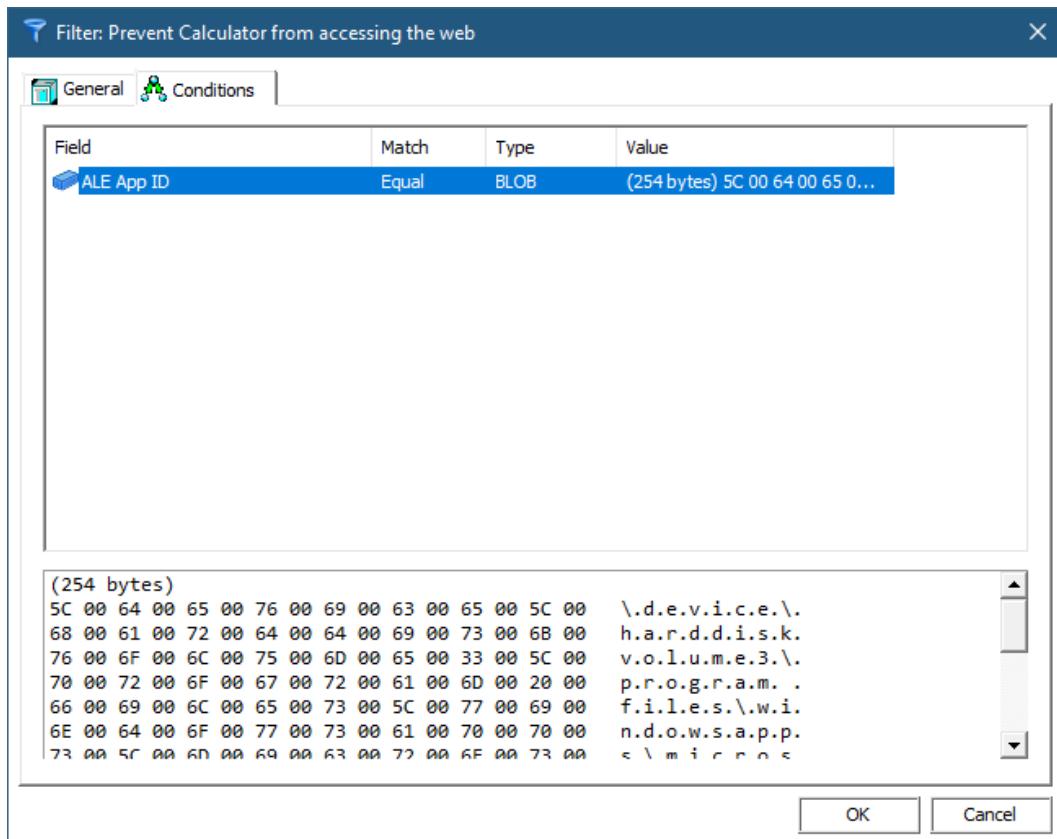


Figure 13-14: Filter condition with App ID

You can right-click the filters and delete them using *WFP Explorer*. The `FwpmFilterDeleteByKey` API is used behind the scenes. This will restore Calculator's exchange rate update functionality.

Callout Drivers

The existing WFP layers provides lots of flexibility when creating filters, thanks to the many fields and comparison options available. In many scenarios, you could get away with using the user mode API to add filters to get the functionality you need without resorting to writing a kernel driver. That said, some scenarios require more flexibility than can be provided by the built-in layers and callouts alone. Here are some examples that would require a callout driver:

- Checking some conditions that are not provided by fields in a required layer.
- Examining actual packet data, optionally modifying it.
- Pending an operation until a decision can be made whether to let the operation continue or not.

In the rest of this chapter, we'll look at some examples of callout drivers (as this is a kernel programming book).

Callout Driver Basics

A callout driver starts its life just like any other driver, with a `DriverEntry` routine. Using a callout driver requires three steps, one of which can only be performed by the driver:

1. Register a callout with the kernel WFP engine.
2. Add the callout to one or more applicable layers.
3. Use the callout as part of an action in filter(s).

The first step can only be done in a kernel driver, as this is where the callout specifies its callbacks, to be invoked by the WFP kernel engine when appropriate. The other two steps can be done from user-mode or kernel-mode, where usually user-mode makes more sense, as it provides flexibility of use, without the need to “disturb” the driver.

Technically, step 2 can be performed before step 1. If the callout is not registered, it will be treated as a “blocking” callout, meaning it will block whatever operation it's attached to.

Callout Registration

Registering a callout involves calling `FwpsCalloutRegister` with a callout description:

```
NTSTATUS FwpsCalloutRegister(
    _Inout_ void* deviceObject,
    _In_ const FWPS_CALLOUT* callout,
    _Out_opt_ UINT32* calloutId);
```

The function requires a device object, created normally with `IoCreateDevice`, as we have seen many times before. This is used to associate the callout with the device, so that the driver does not unload prematurely if any code is still executed by one of the callout's callbacks.

The important part of `FwpsCalloutRegister` is the callout structure provided:

```
typedef struct FWPS_CALLOUT_ {
    GUID calloutKey;
    UINT32 flags;
    FWPS_CALLOUT_CLASSIFY_FN classifyFn;
    FWPS_CALLOUT_NOTIFY_FN notifyFn;
    FWPS_CALLOUT_FLOW_DELETE_NOTIFY_FN flowDeleteFn;
} FWPS_CALLOUT;
```

calloutKey is a GUID used to identify the callout. This GUID should be generated once, typically with the *Create GUID* tool available as part of the Visual Studio Tools menu (figure 13-15). The same GUID must be used when adding the callout to a layer, and when using it as part of a filter action (as we'll soon see).

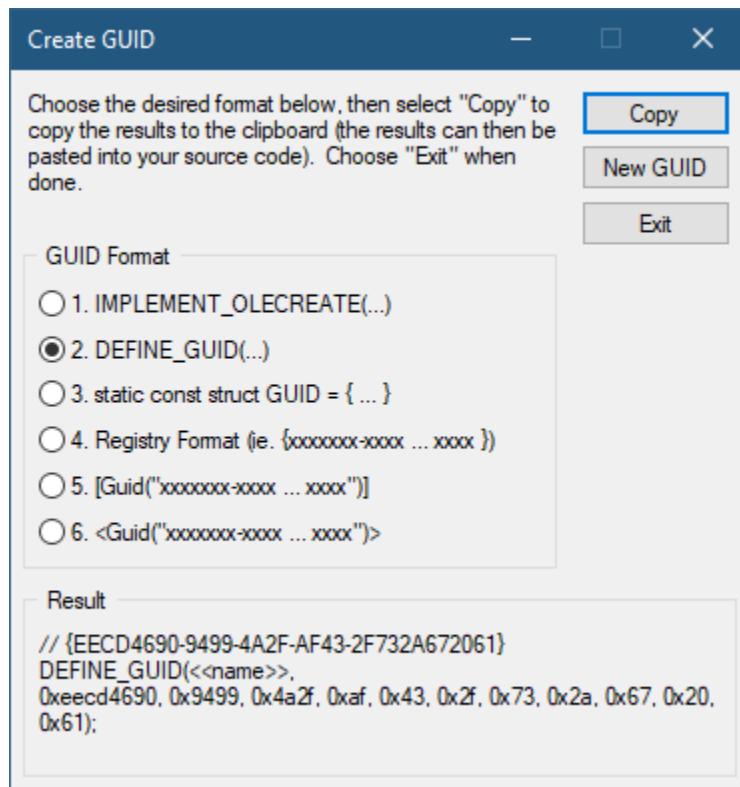


Figure 13-15: The *Create GUID* tool

flags can be zero, or a combination of flags. The list of flags has grown, indicated by the version of `FwpsCalloutRegister` called, with the associated `FWPS_CALLOUT` structure version. At the time of writing, `FwpsCalloutRegister0` to `FwpsCalloutRegister3` exist, with corresponding `FWPS_CALLOUT0` to `FWPS_CALLOUT3`. The data members are essentially the same (just “versioning” changes), and the flags list extended. Here are a couple of notable flags (read the docs for the full list):

- `FWP_CALLOUT_FLAG_ALLOW_OFFLOAD` indicates the callout is unaffected if network data pro-

cessing is offloaded to a capable network interface card (NIC). If this is flag is not specified, offloading will be disabled for any processing path involving filters that use this callout. Normally, you should set this flag.

- `FWP_CALLOUT_FLAG_ENABLE_COMMIT_ADD_NOTIFY` indicates the callout is able to receive notifications about objects and filters added inside a transaction. Once the transaction commits successfully, its callbacks will be invoked.

The last three members of `FWPS_CALLOUT` are callbacks that are invoked when appropriate. The most important one is `classifyFn`, which is the one used to “classify” in some way the request, and decide how processing should proceed. Here is the callback’s prototype:

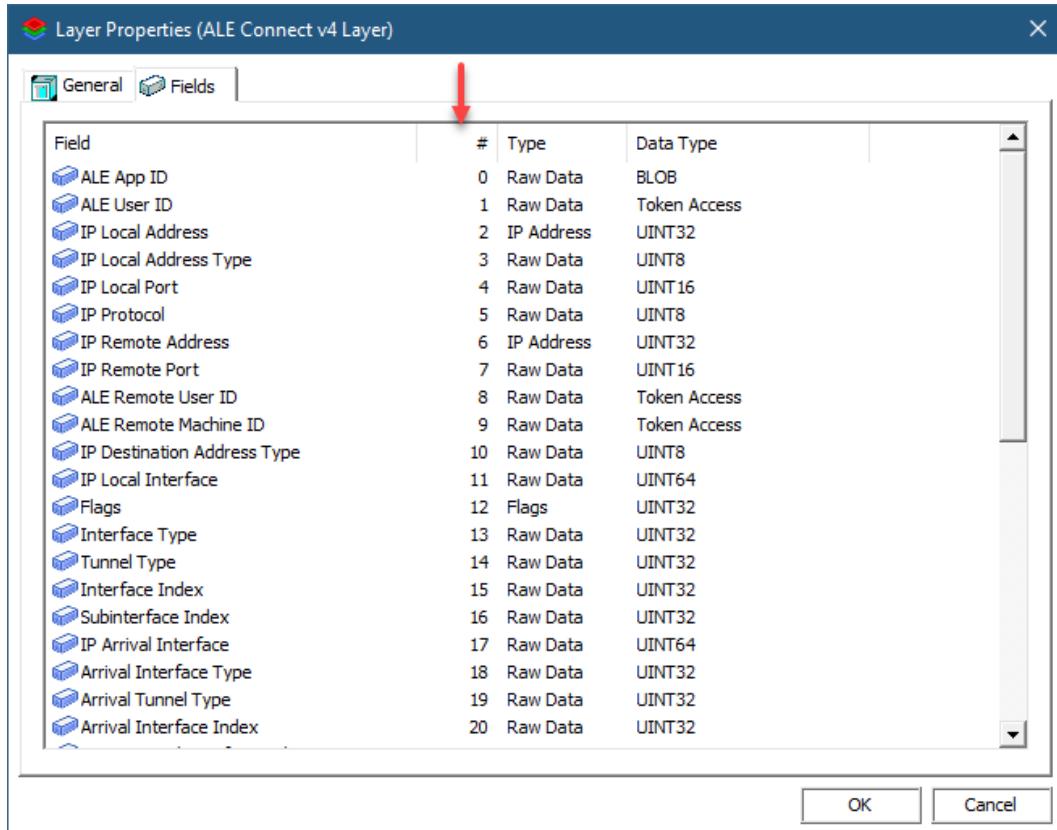
```
void ClassifyFunction(
    const FWPS_INCOMING_VALUES* inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES* inMetaValues,
    void* layerData,
    const void* classifyContext,
    const FWPS_FILTER* filter,
    UINT64 flowContext,
    FWPS_CLASSIFY_OUT* classifyOut);
```

It’s quite a callback, having a multitude of parameters, some of which point to their own structures. `inFixedValues` are the values set for the fields of the layer this callout is part of, wrapped in a `FWPS_INCOMING_VALUES` structure:

```
typedef struct FWPS_INCOMING_VALUE_ {
    FWP_VALUE value;
} FWPS_INCOMING_VALUE;

typedef struct FWPS_INCOMING_VALUES_ {
    UINT16 layerId;
    UINT32 valueCount;
    FWPS_INCOMING_VALUE *incomingValue;
} FWPS_INCOMING_VALUES;
```

The number of values (`valueCount`) is the same as the number of fields in the layer, and they are provided in order. *WFP Explorer* makes it easier to see the order thanks to the provided index in a layer’s properties (see figure 13-16 with an example layer).

Figure 13-16: Layer fields with indices in *WFP Explorer*

The field indices are also available in a set of enumerations, each enumeration describes one of the layers with the field indices provided in the correct order. For example, here is a subset from the same layer as shown in figure 13-16:

```
typedef enum FWPS_FIELDS_ALE_AUTH_CONNECT_V4_ {
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_ALE_APP_ID,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_ALE_USER_ID,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_LOCAL_ADDRESS,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_LOCAL_ADDRESS_TYPE,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_LOCAL_PORT,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_PROTOCOL,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_ADDRESS,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_REMOTE_PORT,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_ALE_REMOTE_USER_ID,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_ALE_REMOTE_MACHINE_ID,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_DESTINATION_ADDRESS_TYPE,
    FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_LOCAL_INTERFACE,
```

```

FWPS_FIELD_ALE_AUTH_CONNECT_V4_FLAGS,
FWPS_FIELD_ALE_AUTH_CONNECT_V4_INTERFACE_TYPE,
FWPS_FIELD_ALE_AUTH_CONNECT_V4_TUNNEL_TYPE,
FWPS_FIELD_ALE_AUTH_CONNECT_V4_INTERFACE_INDEX,
FWPS_FIELD_ALE_AUTH_CONNECT_V4_SUB_INTERFACE_INDEX,
FWPS_FIELD_ALE_AUTH_CONNECT_V4_IP_ARRIVAL_INTERFACE,
//...
FWPS_FIELD_ALE_AUTH_CONNECT_V4_MAX
} FWPS_FIELDS_ALE_AUTH_CONNECT_V4;

```

Next up is `inMetaValues`, pointing to a structure providing some general details of the operation (comments shown are from the header itself):

```

typedef struct FWPS_INCOMING_METADATA_VALUES_ {
    // Bitmask representing which values are set.
    UINT32 currentMetadataValues;
    // Internal flags;
    UINT32 flags;
    // Reserved for system use.
    UINT64 reserved;
    // Discard module and reason.
    FWPS_DISCARD_METADATA discardMetadata;
    // Flow Handle.
    UINT64 flowHandle;
    // IP Header size.
    UINT32 ipHeaderSize;
    // Transport Header size
    UINT32 transportHeaderSize;
    // Process Path.
    FWP_BYTE_BLOB* processPath;
    // Token used for authorization.
    UINT64 token;
    // Process Id.
    UINT64 processId;
    // Source and Destination interface indices for discard indications.
    UINT32 sourceInterfaceIndex;
    UINT32 destinationInterfaceIndex;
    // Compartment Id for injection APIs.
    ULONG compartmentId;
    // Fragment data for inbound fragments.
    FWPS_INBOUND_FRAGMENT_METADATA fragmentMetadata;
    // Path MTU for outbound packets (to enable calculation of fragments).
    ULONG pathMtu;
}

```

```
// Completion handle (required in order to be able to pend at this layer).
HANDLE completionHandle;
// Endpoint handle for use in outbound transport layer injection.
UINT64 transportEndpointHandle;
// Remote scope id for use in outbound transport layer injection.
SCOPE_ID remoteScopeId;
// Socket control data (and length) for use in outbound transport layer injection.
WSACMSGHDR* controlData;
ULONG controlDataLength;
// Direction for the current packet. Only specified for ALE re-authorization.
FWP_DIRECTION packetDirection;
// Raw IP header (and length) if the packet is sent with IP header from a RA\
W socket.
PVOID headerIncludeHeader;
ULONG headerIncludeHeaderLength;
IP_ADDRESS_PREFIX destinationPrefix;
UINT16 frameLength;
UINT64 parentEndpointHandle;
UINT32 icmpIdAndSequence;
// PID of the process that will be accepting the redirected connection
DWORD localRedirectTargetPID;
// original destination of a redirected connection
SOCKADDR* originalDestination;
HANDLE redirectRecords;
// Bitmask representing which L2 values are set.
UINT32 currentL2MetadataValues;
// L2 layer Flags;
UINT32 l2Flags;
UINT32 ethernetMacHeaderSize;
UINT32 wifiOperationMode;

NDIS_SWITCH_PORT_ID vSwitchSourcePortId;
NDIS_SWITCH_NIC_INDEX vSwitchSourceNicIndex;
NDIS_SWITCH_PORT_ID vSwitchDestinationPortId;

HANDLE vSwitchPacketContext;
PVOID subProcessTag;
// Reserved for system use.
UINT64 reserved1;
} FWPS_INCOMING_METADATA_VALUES;
```

I'll mention a few useful members. First, `currentMetadataValues` indicates which other members

have valid information. An extension to that is `currentL2MetadataValues`, simply because at some point 32 flags were not enough, so more were added in Windows 8 and later. Here are a few examples for `currentMetadataValues`:

- `FWPS_METADATA_FIELD_PROCESS_PATH` - process path of the accessing process is specified in the `processPath` member, as a `FWP_BYTE_BLOB*` - the same type used in an earlier section when calling `FwpmGetAppIdFromFile`.
- `FWPS_METADATA_FIELD_PROCESS_ID` - process ID of the accessing process given in the `processId` member.
- `FWPS_METADATA_FIELD_IP_HEADER_SIZE` - the IP header size (in bytes) is specified in `ipHeaderSize`, which indicates where the header ends and the actual packet data begins.

The next parameter to the `classify` function is `layerData`, providing the actual data that makes sense for this layer. Some layers don't have any associated data, so this pointer may be `NULL`. For a "Stream" layer (such as `FWPS_LAYER_STREAM_V4`), the pointer is to a `FWPS_STREAM_CALLOUT_IO_PACKET` structure. In all other cases, it points to a `NET_BUFFER_LIST`, which is the standard way of describing a network buffer. Clearly, there is a lot more to look into, some of which we'll do later in this chapter.

The next parameter, `classifyContext`, is an internal pointer used by the WFP infrastructure. It may be `NULL` for some layers. If not-`NULL`, it can be used to "pend" an operation - hold on to it, until a decision can be made later, outside of the context of the callback. This is beyond the scope of this chapter.

The next parameter, `filter` is the filter pointer used to invoke this callback. It's essentially the one used by a client code to set up this callout as an action target. Usually, filters are added from user mode with `FwpmFilterAdd`, but they can be added by kernel code in exactly the same way. Here is its generic definition (taking the version out of the equation):

```
typedef struct FWPS_FILTER_ {
    UINT64 filterId;
    FWP_VALUE weight;
    UINT16 subLayerWeight;
    UINT16 flags;
    UINT32 numFilterConditions;
    FWPS_FILTER_CONDITION *filterCondition;
    FWPS_ACTION action;
    UINT64 context;
    FWPM_PROVIDER_CONTEXT *providerContext;
} FWPS_FILTER;
```

Most members were set explicitly by whoever called `FwpmFilterAdd`. Consult the docs for the missing pieces.



Note that the "runtime" structures used the kernel WFP engine (starting with `Fwps`) are not the same ones used by the management functions (common to user-mode and kernel-mode, starting with `Fwpm`). For example, `filterId` in the above structure is a 64-bit value, instead of a GUID that is used to identify a filter with the management functions.

The next parameter, `flowContext`, represents a context associated with the data flow, if any. Some layers don't support data flow, which would make this parameter ignorable.

Finally, the last parameter to the `classify` callback, `classifyOut`, is a pointer to a structure where the result of the classification should be provided directly (unless the operation is pended). This is where the final "decision" of the callout is to be stored:

```
typedef struct FWPS_CLASSIFY_OUT_ {
    FWP_ACTION_TYPE actionType;
    UINT64 outContext;           // reserved
    UINT64 filterId;            // reserved
    UINT32 rights;
    UINT32 flags;
    UINT32 reserved;
} FWPS_CLASSIFY_OUT;
```

The most important member is `actionType`, where the driver decides the suggested fate of the operation. Possible values include:

- `FWP_ACTION_BLOCK` - block the operation.
- `FWP_ACTION_CONTINUE` - pass the decision to the next filter (if any).
- `FWP_ACTION_NONE` and `FWP_ACTION_NONE_NO_MATCH` - do nothing. Effectively the same as `FWP_ACTION_CONTINUE`, but provides different semantics in case someone cares.
- `FWP_ACTION_PERMIT` - allow the operation to continue.

Writing to `actionType` is "controlled" by the `rights` member. If it has the value `FWPS_RIGHT_ACTION_WRITE`, then the driver is allowed to set an action in `actionType`. If not, the driver is permitted to set an action of `FWP_ACTION_BLOCK` only to override an earlier filter's action. Technically, a callout can always write an action value, but it should follow the rules. A driver setting the action to block or permit should remove the `FWPS_RIGHT_ACTION_WRITE` flag from `rights` so that subsequent filters will not likely "interfere" with the callout's decision.

The only remaining member of `FWPS_CLASSIFY_OUT` yet to be discussed is `flags`, which can be set to a combination of values with the most useful being the following (see the docs for the other two possible flags):

- `FWPS_CLASSIFY_OUT_FLAG_ABSORB` - the data is silently dropped. This is typical for cases where the original packet is absorbed, to be replaced by a different one. The driver sets this value in such cases. We'll see an example using this flag later in this chapter.

Back to `FwpsCalloutRegister` and the `FWPS_CALLOUT` structure - the next member, `notifyFn`, is another callback the driver has to provide. It's called when filters that are using this callout are added or removed:

```
NTSTATUS NotifyCallback(
    _In_ FWPS_CALLOUT_NOTIFY_TYPE notifyType,
    _In_ const GUID* filterKey,
    _Inout_ FWPS_FILTER* filter);
```

notifyType can be either FWPS_CALLOUT_NOTIFY_ADD_FILTER or FWPS_CALLOUT_NOTIFY_DELETE_FILTER. filterKey is the GUID identifying this filter - it's NULL for a delete notification. Finally, filter is the “runtime” representation of the added/removed filter (the same one we examined earlier).

The return value from the callback does not matter for a *delete* operation. For an *add* operation, STATUS_SUCCESS indicates the driver is OK with the filter being added. Returning other status codes will cause the filter not to be added.

The last member in FWPS_CALLOUT is flowDeleteFn, an optional callback that is useful with data flow requests (out of scope for this chapter). Set this member to NULL if no data flow requests are processed in the callout.

Now that the callout is registered, it can be actively used in filters. Before the driver unloads, it should unregister its callout(s) by calling either FwpsCalloutUnregisterById or FwpsCalloutUnregisterByKey (whichever is more convenient):

```
NTSTATUS FwpsCalloutUnregisterById(_In_ const UINT32 calloutId);
NTSTATUS FwpsCalloutUnregisterByKey(_In_ const GUID* calloutKey);
```

The callout ID is an optional return value from FwpsCalloutRegister. The driver can store it for later use, such as for unregistering purposes. Using the GUID of the callout is just as good.

Demo: Callout Driver

In order to put the previous section information to good use, we'll create a callout driver that can block certain processes from accessing the network. Looking at various fields layers have, there is no “process ID” kind of field, which means we have to write a callout driver to accomplish this task.

The Driver

We start by creating a new *WDM Empty Driver* project as usual, named *ProcessNetFilter*. The INF file is deleted, as it's not needed. We'll keep the interesting state of the driver in a global class named *Globals* that will take care of all the WFP functionality (in *Globals.h*):

```

#include "Vector.h"
#include "SpinLock.h"

class Globals {
public:
    Globals();
    static Globals& Get();
    Globals(Globals const&) = delete;
    Globals& operator=(Globals const&) = delete;
    ~Globals();

    NTSTATUS RegisterCallouts(PDEVICE_OBJECT devObj);
    NTSTATUS AddProcess(ULONG pid);
    NTSTATUS DeleteProcess(ULONG pid);
    NTSTATUS ClearProcesses();
    bool IsProcessBlocked(ULONG pid) const;

    NTSTATUS DoCalloutNotify(
        _In_ FWPS_CALLOUT_NOTIFY_TYPE notifyType,
        _In_ const GUID* filterKey,
        _Inout_ FWPS_FILTER* filter);

    void DoCalloutClassify(
        _In_ const FWPS_INCOMING_VALUES* inFixedValues,
        _In_ const FWPS_INCOMING_METADATA_VALUES* inMetaValues,
        _Inout_opt_ void* layerData,
        _In_opt_ const void* classifyContext,
        _In_ const FWPS_FILTER* filter,
        _In_ UINT64 flowContext,
        _Inout_ FWPS_CLASSIFY_OUT* classifyOut);

private:
    Vector<ULONG, PoolType::NonPaged> m_Processes;
    mutable SpinLock m_ProcessesLock;
    inline static Globals* s_Globals;
};

```

The *vector.h* header implements a simple resizable array, which will not be described in this chapter. The *m_Processes* member is declared as such a vector of process IDs (ULONG) allocated from non-paged pool (PoolType::NonPaged enumeration value, defined in *Memory.h*). More information on the *Vector<>* class and other parts of the *KTL* can be found in the appendix.

A *Globals* pointer named *g_Data* is defined in *Main.cpp*. It's dynamically allocated with the *new* operator (overloaded) to allow the constructor to run, and by the same token, it's deleted with the

overloaded `delete` operator, causing its destructor to run.

To make it easier to access, a static variable (`s_Globals`) keeps track of the singleton `Globals` instance for easy access from anywhere using the static `Get` method. Here is the relevant code from *Globals.cpp*:

```
Globals::Globals() {
    s_Globals = this;
    m_ProcessesLock.Init();
}

Globals& Globals::Get() {
    return *s_Globals;
}
```

Let's now turn our attention to the `DriverEntry` function. The driver creates a normal named device object and a symbolic link in order to allow sending I/O controls for blocking and permitting network access for process IDs. Most of the code should be very familiar at this point:

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\ProcNetFilter");
    PDEVICE_OBJECT devObj;
    auto status = IoCreateDevice(DriverObject, 0, &devName,
        FILE_DEVICE_UNKNOWN, 0, FALSE, &devObj);
    if (!NT_SUCCESS(status))
        return status;

    bool symLinkCreated = false;
    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\ProcNetFilter");
    do {
        g_Data = new (PoolType::NonPaged) Globals;
        if (!g_Data) {
            status = STATUS_NO_MEMORY;
            break;
        }
        status = IoCreateSymbolicLink(&symLink, &devName);
        if (!NT_SUCCESS(status))
            break;
        symLinkCreated = true;

        status = g_Data->RegisterCallouts(devObj);
        if (!NT_SUCCESS(status))
            break;
    } while (false);
```

```

    if (!NT_SUCCESS(status)) {
        KdPrint((DRIVER_PREFIX "DriverEntry failed (0x%X)\n", status));
        if (symLinkCreated)
            IoDeleteSymbolicLink(&symLink);
        IoDeleteDevice(devObj);
        return status;
    }

    DriverObject->DriverUnload = ProcNetFilterUnload;
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
        DriverObject->MajorFunction[IRP_MJ_CLOSE] = ProcNetFilterCreateClose;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ProcNetFilterDeviceCon\

trol;

    return STATUS_SUCCESS;
}

```

The unfamiliar code is the call to `Globals::RegisterCallouts`. Registering callouts require calling `FwpsRegisterCallout` for each callout. Why would we need multiple callouts? When adding callouts (later), a callout is added at a specific layer. If the “same” callout behavior is required in several layers, different callouts (with different GUIDs) must be added separately. Since we’re interested in blocking network traffics for TCP and UDP, for IPv4 and IPv6, we require four callouts, even though the callouts’ callbacks will be the same:

```

NTSTATUS Globals::RegisterCallouts(PDEVICE_OBJECT devObj) {
    const GUID* guids[] = {
        &GUID_CALLOUT_PROCESS_BLOCK_V4,
        &GUID_CALLOUT_PROCESS_BLOCK_V6,
        &GUID_CALLOUT_PROCESS_BLOCK_UDP_V4,
        &GUID_CALLOUT_PROCESS_BLOCK_UDP_V6,
    };
    NTSTATUS status = STATUS_SUCCESS;

    for (auto& guid : guids) {
        FWPS_CALLOUT callout{};
        callout.calloutKey = *guid;
        callout.notifyFn = OnCalloutNotify;
        callout.classifyFn = OnCalloutClassify;
        status |= FwpsCalloutRegister(devObj, &callout, nullptr);
    }
    return status;
}

```

The GUIDs of these callouts are defined in the *ProcNetFilterPublic.h* header, shared with user mode, as it's more flexible to let user mode add callouts as needed.

The unload routine deletes the `g_Data` object, invoking the destructor, and then deletes the symbolic link and device object:

```
void ProcNetFilterUnload(PDRIVER_OBJECT DriverObject) {
    delete g_Data;

    UNICODE_STRING symLink = RTL_CONSTANT_STRING(L"\?\?\ProcNetFilter");
    IoDeleteSymbolicLink(&symLink);
    IoDeleteDevice(DriverObject->DeviceObject);
}
```

Here is the `Globals` destructor:

```
Globals::~Globals() {
    const GUID* guids[] = {
        &GUID_CALLOUT_PROCESS_BLOCK_V4,
        &GUID_CALLOUT_PROCESS_BLOCK_V6,
        &GUID_CALLOUT_PROCESS_BLOCK_UDP_V4,
        &GUID_CALLOUT_PROCESS_BLOCK_UDP_V6,
    };
    for(auto& guid : guids)
        FwpsCalloutUnregisterByKey(guid);
}
```

The destructor reverses the callout registration by unregistering the four callouts.

Managing Processes

Managing the process IDs that require blocking done by manipulating the `Vector<>`. Some functions in the `Globals` class are task with this work:

```
NTSTATUS Globals::AddProcess(ULONG pid) {
    //
    // check if the process exists
    //
    PEPPROCESS process;
    auto status = PsLookupProcessByProcessId(ULongToHandle(pid), &process);
    if (!NT_SUCCESS(status))
        return status;
```

```

{
    Locker locker(m_ProcessesLock);
    //
    // don't add if it's already there
    //
    if( !m_Processes.Contains(pid))
        m_Processes.Add(pid);
}
ObDereferenceObject(process);
return STATUS_SUCCESS;
}

NTSTATUS Globals::DeleteProcess(ULONG pid) {
    Locker locker(m_ProcessesLock);
    return m_Processes.Remove(pid) ? STATUS_SUCCESS : STATUS_NOT_FOUND;
}

NTSTATUS Globals::ClearProcesses() {
    Locker locker(m_ProcessesLock);
    m_Processes.Clear();
    return STATUS_SUCCESS;
}

bool Globals::IsProcessBlocked(ULONG pid) const {
    Locker locker(m_ProcessesLock);
    return m_Processes.Contains(pid);
}

```

`m_ProcessesLock` is of type `SpinLock` - a spin lock wrapper we've used in previous chapters. `Locker` is a generic locker we've used as well.

A nice touch in the `AddProcess` implementation is checking that the process actually exists by calling `PsLookupProcessByProcessId`.

Adding, removing and clearing the processes vector is done through I/O control codes. These are defined in `ProcNetFilterPublic.h` alongside the callout GUIDs:

```
#include <initguid.h>

#define PROCNETFILTER_DEVICE 0x8003

#define IOCTL_PNF_BLOCK_PROCESS           CTL_CODE(PROCNETFILTER_DEVICE, 0x800, METHOD_B \
UFFERED, FILE_ANY_ACCESS)
#define IOCTL_PNF_PERMIT_PROCESS CTL_CODE(PROCNETFILTER_DEVICE, 0x801, METHOD_B \
UFFERED, FILE_ANY_ACCESS)
#define IOCTL_PNF_CLEAR                 CTL_CODE(PROCNETFILTER_DEVICE, 0x802, METHOD_N \
EITHER, FILE_ANY_ACCESS)

// {5027C277-201A-4AAF-B8EC-95C05E857059}
DEFINE_GUID(GUID_CALLOUT_PROCESS_BLOCK_V4, 0x5027c277, 0x201a, 0x4aaaf, 0xb8, 0x \
ec, 0x95, 0xc0, 0x5e, 0x85, 0x70, 0x59);

// {CF51FD24-566F-4C6D-9BC9-8013E9875E7E}
DEFINE_GUID(GUID_CALLOUT_PROCESS_BLOCK_V6, 0xcf51fd24, 0x566f, 0x4c6d, 0x9b, 0x \
c9, 0x80, 0x13, 0xe9, 0x87, 0x5e, 0x7e);

// {200E35C6-7182-4F9C-97DF-34028A225BEC}
DEFINE_GUID(GUID_CALLOUT_PROCESS_BLOCK_UDP_V4, 0x200e35c6, 0x7182, 0x4f9c, 0x97 \
, 0xdf, 0x34, 0x02, 0x8a, 0x22, 0x5b, 0xec);

// {C8AF8E6D-1D0C-4547-A2A1-7593C3396BAF}
DEFINE_GUID(GUID_CALLOUT_PROCESS_BLOCK_UDP_V6, 0xc8af8e6d, 0x1d0c, 0x4547, 0xa2 \
, 0xa1, 0x75, 0x93, 0xc3, 0x39, 0x6b, 0xaf);
```

Handling the IOCTLs themselves is the job of ProcNetFilterDeviceControl:

```
NTSTATUS ProcNetFilterDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto irpSp = IoGetCurrentIrpStackLocation(Irp);
    auto const& dic = irpSp->Parameters.DeviceIoControl;
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG info = 0;

    switch (dic.IoControlCode) {
        case IOCTL_PNF_CLEAR:
            status = g_Data->ClearProcesses();
            break;

        case IOCTL_PNF_BLOCK_PROCESS:
        case IOCTL_PNF_PERMIT_PROCESS:
            if (dic.InputBufferLength < sizeof(ULONG)) {
```

```
        status = STATUS_BUFFER_TOO_SMALL;
        break;
    }
    auto pid = *(ULONG*)Irp->AssociatedIrp.SystemBuffer;
    status = dic.IoControlCode == IOCTL_PNF_BLOCK_PROCESS ?
        g_Data->AddProcess(pid) : g_Data->DeleteProcess(pid);
    if (NT_SUCCESS(status))
        info = sizeof(ULONG);
    break;
}

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

The above code should be familiar by now. `CompleteRequest` is a helper function we used before that simply completes an IRP given an optional status and information:

```
NTSTATUS CompleteRequest(
    PIRP Irp, NTSTATUS status = STATUS_SUCCESS, ULONG_PTR info = 0);

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

Callout Callbacks

The most interesting part in the driver is obviously the WFP related code. The callout registration done previously points the notify and classify callouts to `OnCalloutNotify` and `OnCalloutClassify`, respectively.

These two functions simply delegate their work to instance members of the `Globals` class so it would be easier to access the class members:

```

NTSTATUS OnCalloutNotify(FWPS_CALLOUT_NOTIFY_TYPE notifyType,
    const GUID* filterKey, FWPS_FILTER* filter) {
    return Globals::Get().DoCalloutNotify(notifyType, filterKey, filter);
}

void OnCalloutClassify(const FWPS_INCOMING_VALUES* inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES* inMetaValues,
    void* layerData, const void* classifyContext, const FWPS_FILTER* filter,
    UINT64 flowContext, FWPS_CLASSIFY_OUT* classifyOut) {
    Globals::Get().DoCalloutClassify(inFixedValues, inMetaValues,
        layerData, classifyContext, filter, flowContext, classifyOut);
}

```

The real work is done in the member functions with the names `DoCalloutNotify` and `DoCalloutClassify`.

The notify callback is mostly uninteresting, but must be implemented. The code just outputs the fact that a filter has been added or removed with its GUID if available:

```

NTSTATUS Globals::DoCalloutNotify(FWPS_CALLOUT_NOTIFY_TYPE notifyType,
    const GUID* filterKey, FWPS_FILTER* filter) {
    UNREFERENCED_PARAMETER(filter);

    UNICODE_STRING sguid = RTL_CONSTANT_STRING(L"<Noguid>");
    if (filterKey)
        RtlStringFromGUID(*filterKey, &sguid);

    if (notifyType == FWPS_CALLOUT_NOTIFY_ADD_FILTER) {
        KdPrint((DRIVER_PREFIX "Filter added: %wZ\n", sguid));
    }
    else if (notifyType == FWPS_CALLOUT_NOTIFY_DELETE_FILTER) {
        KdPrint((DRIVER_PREFIX "Filter deleted: %wZ\n", sguid));
    }
    if (filterKey)
        RtlFreeUnicodeString(&sguid);

    return STATUS_SUCCESS;
}

```

In most cases, the fact the filters are added or removed (that use one of the driver's callouts) is not important. Still, it may be useful in certain cases. For example, the driver can keep track of how many filters are currently using the driver for logging or other purposes.

The above code uses the helper `RtlStringFromGUID` API provided by the kernel to convert a GUID into a `UNICODE_STRING`. Memory is allocated by the routine, so `RtlFreeUnicodeString` must be

called to free the string. Note that in some cases the GUID of the filter is not provided, so care must be taken not to pass a NULL GUID to RtGetStringFromGUID, as it will crash the system.

The most important callback is the classify one. Its job is to determine if the request should be blocked. First, we need to check if a process ID is available as part of the “metadata” fields:

```
void Globals::DoCalloutClassify(const FWPS_INCOMING_VALUES* inFixedValues,
    const FWPS_INCOMING_METADATA_VALUES* inMetaValues,
    void* layerData, const void* classifyContext, const FWPS_FILTER* filter,
    UINT64 flowContext, FWPS_CLASSIFY_OUT* classifyOut) {
    UNREFERENCED_PARAMETER(flowContext);
    UNREFERENCED_PARAMETER(inFixedValues);
    UNREFERENCED_PARAMETER(layerData);
    UNREFERENCED_PARAMETER(filter);
    UNREFERENCED_PARAMETER(classifyContext);

    //
    // search for the PID (if available)
    //
    if ((inMetaValues->currentMetadataValues & FWPS_METADATA_FIELD_PROCESS_ID)
        == 0) return;
```

Now that we know a process ID is available, we’ll check if it’s on our list of PIDs to block:

```
bool block;
{
    Locker locker(m_ProcessesLock);
    block = m_Processes.Contains((ULONG)inMetaValues->processId);
}
```

The spin lock is acquired for the minimum possible interval, as multiple classify callbacks may be running at the same time. A spin lock is used (and not a fast mutex), because the classify callback is invoked at IRQL DISPATCH_LEVEL (2).

If we need to block, we set the action to “block” and tell downstream filters not to change the outcome:

```

if(block) {
    //
    // block
    //
    classifyOut->actionType = FWP_ACTION_BLOCK;

    //
    // ask other filters from overriding the block
    //
    classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

    KdPrint((DRIVER_PREFIX "Blocked process %u\n",
              (ULONG)inMetaValues->processId));
}

```

Removing the FWPS_RIGHT_ACTION_WRITE bit in the rights member is critical - otherwise next callouts in the chain might change the action to “permit”. It’s OK to change a “permit” action to “block” - but not vice-versa. Here is the full classify callout implementation for easier reference (comments removed):

```

void Globals::DoCalloutClassify(const FWPS_INCOMING_VALUES* inFixedValues,
                                const FWPS_INCOMING_METADATA_VALUES* inMetaValues,
                                void* layerData, const void* classifyContext, const FWPS_FILTER* filter,
                                UINT64 flowContext, FWPS_CLASSIFY_OUT* classifyOut) {
    UNREFERENCED_PARAMETER(flowContext);
    UNREFERENCED_PARAMETER(inFixedValues);
    UNREFERENCED_PARAMETER(layerData);
    UNREFERENCED_PARAMETER(filter);
    UNREFERENCED_PARAMETER(classifyContext);

    if ((inMetaValues->currentMetadataValues & FWPS_METADATA_FIELD_PROCESS_ID)
        == 0) return;

    bool block;
    {
        Locker locker(m_ProcessesLock);
        block = m_Processes.Contains((ULONG)inMetaValues->processId);
    }
    if(block) {
        classifyOut->actionType = FWP_ACTION_BLOCK;
        classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

        KdPrint((DRIVER_PREFIX "Blocked process %u\n",

```

```

        (ULONG)inMetaValues->processId));
    }
}

```

In order for the driver to link successfully, the *fwpkclnt.lib* import library must be added to the Linker's Input tab (see figure 13-17).



You may try to add the import through a pragma like so: `#pragma comment(lib, "fwpkclnt")`. This does

not have the desired effect. For some reason, this pragma only seems to work in user-mode projects.

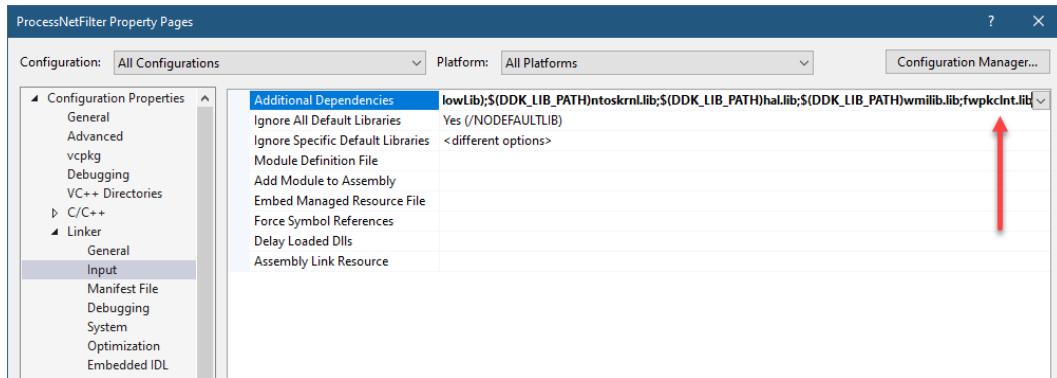


Figure 13-17: WFP kernel client library



For completeness, the driver should keep track of process destruction, and remove a destroyed process from the list of blocked processes (if listed). Add code to accomplish that.

Demo: User-Mode Client

The user-mode client needs to add the callouts to the correct layers, and add filters that use these callouts - otherwise the callouts play no role.

The project is a standard console application named *BlockProcess*. The `main` function starts by examining the command line:

```

int main(int argc, const char* argv[]) {
    if (argc < 2) {
        printf("Usage: blockprocess <block | permit | clear> [pid]\n");
        return 0;
    }
}

```

Next, it adds a new WFP provider to the system, to make it easier to identify callouts and filters that “belong” to the provider. Providers don’t play an active role in WFP, but they are useful for identifying different “sources” of filters or callouts:

```

if (DWORD error = RegisterProvider(); error != ERROR_SUCCESS) {
    printf("Error registering provider (%u)\n", error);
    return 1;
}

```



The feature where an initialization is permitted before a test with a semicolon in between as seen above is available from C++ 17. It also works with a `switch` statement. It’s useful in keeping the variable (`error` in the above code) constrained in scope of the `if` statement (and an `else` statement if exists).

Defining a provider requires generating a GUID to uniquely identify the provider. Here is the GUID defined at the top of the *BlockProcess.cpp* file:

```

// {7672D055-03C0-43F1-9E31-0392850BD07F}
DEFINE_GUID(WFP_PROVIDER_CHAPTER13,
    0x7672d055, 0x3c0, 0x43f1, 0x9e, 0x31, 0x3, 0x92, 0x85, 0xb, 0xd0, 0x7f);

```

Registering a provider must be done (as most operations) against the WFP engine:

```

DWORD RegisterProvider() {
    HANDLE hEngine;
    DWORD error = FwpmEngineOpen(nullptr, RPC_C_AUTHN_DEFAULT,
        nullptr, nullptr, &hEngine);
    if (error)
        return error;
}

```

Working with the WFP engine would require opening and closing it with code easily repeated. This project just repeats the code, but a good idea would be to create a wrapper class for the WFP engine. You can find one such example in the source code of *WFP Explorer*.

Next, we can check if the provider has already been registered. If so, no further action is needed. Otherwise, we go ahead and register it:

```

FWPM_PROVIDER* provider;
error = FwpmProviderGetByKey(hEngine, &WFP_PROVIDER_CHAPTER13, &provider);
if (error != ERROR_SUCCESS) {
    FWPM_PROVIDER reg{};
    WCHAR name[] = L"WKP2 Chapter 13";
    reg.displayData.name = name;
    reg.providerKey = WFP_PROVIDER_CHAPTER13;
    reg.flags = FWPM_PROVIDER_FLAG_PERSISTENT;

    error = FwpmProviderAdd(hEngine, &reg, nullptr);
}
else {
    FwpmFreeMemory((void**)provider);
}

```

If locating the provider by GUID (`FwpmProviderGetByKey`) fails, we need to fill a `FWPM_PROVIDER` structure. The display name is mandatory and so is the GUID used to uniquely identify it. The `FWPM_PROVIDER_FLAG_PERSISTENT` flag keeps the provider registered even if the system restarts. This is mostly needed if callouts/filters are needed at early stages of Windows boot before any user-mode code has any chance of running.

Finally, the engine should be closed:

```

FwpmEngineClose(hEngine);
return error;
}

```

Back to `main`. The next item on the agenda is opening a handle to the device. Without that, there are no registered callouts:

```

HANDLE hDevice = CreateFile(L"\\\.\ProcNetFilter",
    GENERIC_WRITE | GENERIC_READ, 0, nullptr, OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE) {
    printf("Error opening device (%u)\n", GetLastError());
    return 1;
}

```

Now we can add the four callouts if not yet added:

```

if (!AddCallouts()) {
    printf("Error adding callouts\n");
    return 1;
}

```

Adding the callouts allows them to be used in filters. If no filter is referencing the callouts, they are essentially useless.

AddCallouts opens a handle to the engine, and looks for one of the callouts. If it's already added, there is nothing else to do:

```

bool AddCallouts() {
    HANDLE hEngine;
    DWORD error = FwpmEngineOpen(nullptr, RPC_C_AUTHN_DEFAULT,
        nullptr, nullptr, &hEngine);
    if (error)
        return false;

    do {
        if (FWPM_CALLOUT* callout; FwpmCalloutGetByKey(hEngine,
            &GUID_CALLOUT_PROCESS_BLOCK_V4, &callout) == ERROR_SUCCESS) {
            FwpmFreeMemory((void**)&callout);
            break;
    }
}

```

Otherwise, the callouts must be added to the correct layers:

```

const struct {
    const GUID* guid;
    const GUID* layer;
} callouts[] = {
    { &GUID_CALLOUT_PROCESS_BLOCK_V4, &FWPM_LAYER_ALE_AUTH_CONNECT_V4 },
    { &GUID_CALLOUT_PROCESS_BLOCK_V6, &FWPM_LAYER_ALE_AUTH_CONNECT_V6 },
    { &GUID_CALLOUT_PROCESS_BLOCK_UDP_V4, &FWPM_LAYER_ALE_RESOURCE_ASSIGNME\NT_V4 },
    { &GUID_CALLOUT_PROCESS_BLOCK_UDP_V6, &FWPM_LAYER_ALE_RESOURCE_ASSIGNME\NT_V6 },
};

error = FwpmTransactionBegin(hEngine, 0);
if (error) break;

for (auto& co : callouts) {

```

```

FWPM_CALLOUT callout{};
callout.applicableLayer = *co.layer;
callout.calloutKey = *co.guid;
WCHAR name[] = L"Block PID callout";
callout.displayData.name = name;
callout.providerKey = (GUID*)&WFP_PROVIDER_CHAPTER13;

FwpmCalloutAdd(hEngine, &callout, nullptr, nullptr);
}
error = FwpmTransactionCommit(hEngine);
} while (false);
}

```

Each callout is added to the appropriate layer. For block/permit operations, the FWPM_LAYER_ALE_AUTH_CONNECT_V4/6 layer are the ones to use for TCP and FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4/6 for UDP. The WFP documentation lists all the available layers with their meaning.

For each callout, a display name is mandatory and so are the callout unique key (GUID) and the applicable layer. Filters added to this layer(s) only can use these callouts. The provider is set as well for easy identification.

Performing multiple operations against the engine can be done within the scope of a transaction that adheres to the classic “ACID” properties (atomicity, consistency, isolation and durability), meaning that either all operations within the transaction succeed or none do. `FwpmTransactionBegin` initiates a transaction and `FwpmTransactionCommit` commits it. `FwpmTransactionAbort` is available if aborting the transaction is desired. If the engine is closed prematurely, any transactions are aborted.

Finally, the engine is properly closed:

```

FwpmEngineClose(hEngine);
return error == ERROR_SUCCESS;
}

```

Back to `main`. The next thing to do is examine the command line arguments, and forward to the correct function for handling:

```

bool success = false;
if (_stricmp(argv[1], "block") == 0 && argc > 2) {
    success = BlockProcess(hDevice, atoi(argv[2]));
}
else if (_stricmp(argv[1], "permit") == 0 && argc > 2) {
    success = PermitProcess(hDevice, atoi(argv[2]));
}
else if (_stricmp(argv[1], "clear") == 0) {
    success = ClearAll(hDevice);
}

```

```

    else {
        printf("Unknown or bad command.\n");
        return 1;
    }
    if (success)
        printf("Operation completed successfully.\n");
    else
        printf("Error occurred: %u\n", GetLastError());
CloseHandle(hDevice);
return 0;
}

```

Let's examine each in turn, starting with `BlockProcess`. Its purpose is to add a PID to the list of blocked processes. First, it needs to add filters to the four layers if these have not been added before:

```

bool BlockProcess(HANDLE hDevice, DWORD pid) {
    if (!AddFilters()) {
        printf("Failed to add filters\n");
        return false;
    }
}

```

We need to add the filters just once, since they can serve any number of process IDs. This means it will be easier to give these four filters known GUIDs that we can then reference as needed. The following is set up at the top of `BlockProcess.cpp`:

```

// {C5C2DEC4-C0CD-4187-9BE9-C749ED53549D}
DEFINE_GUID(GUID_FILTER_V4, 0xc5c2dec4, 0xc0cd, 0x4187, 0x9b, 0xe9, 0xc7, 0x49,
0xed, 0x53, 0x54, 0x9d);
// {9E99EFD3-8E9E-496B-8F6D-63A69D2E84A7}
DEFINE_GUID(GUID_FILTER_V6, 0x9e99efd3, 0x8e9e, 0x496b, 0x8f, 0x6d, 0x63, 0xa6,
0x9d, 0x2e, 0x84, 0xa7);
// {EE870CB6-7D26-4580-A8F4-8CA7783A98F9}
DEFINE_GUID(GUID_FILTER_UDP_V4, 0xee870cb6, 0x7d26, 0x4580, 0xa8, 0xf4, 0x8c, 0\,
xa7, 0x78, 0x3a, 0x98, 0xf9);
// {C8EB1629-B3C7-4A37-95F5-1DA3495EC8F5}
DEFINE_GUID(GUID_FILTER_UDP_V6, 0xc8eb1629, 0xb3c7, 0x4a37, 0x95, 0xf5, 0x1d, 0\,
xa3, 0x49, 0x5e, 0xc8, 0xf5);

```

The alternative would be to let WFP assign GUIDs to added filters, but that would mean locating them would be more difficult, as it would require enumerating all filters and looking at the callout GUID they point to (if any), and/or identifying the provider.

The first step in `AddFilters` is checking if one was added before, and aborting if so:

```

bool AddFilters() {
    HANDLE hEngine;
    DWORD error = FwpmEngineOpen(nullptr, RPC_C_AUTHN_DEFAULT,
        nullptr, nullptr, &hEngine);
    if (error)
        return false;

    do {
        if (FWPM_FILTER* filter; FwpmFilterGetByKey(hEngine,
            &GUID_FILTER_V4, &filter) == ERROR_SUCCESS) {
            FwpmFreeMemory((void**)&filter);
            break;
    }
}

```

To add the filters, we open a transaction and call `FwpmFilterAdd` to add the four filters with their associated layers:

```

static const struct {
    const GUID* guid;
    const GUID* layer;
    const GUID* callout;
} filters[] = {
    { &GUID_FILTER_V4, &FWPM_LAYER_ALE_AUTH_CONNECT_V4, &GUID_CALLOUT_PROCESS_BLOCK_V4 },
    { &GUID_FILTER_V6, &FWPM_LAYER_ALE_AUTH_CONNECT_V6, &GUID_CALLOUT_PROCESS_BLOCK_V6 },
    { &GUID_FILTER_UDP_V4, &FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4, &GUID_CALLOUT_PROCESS_BLOCK_UDP_V4 },
    { &GUID_FILTER_UDP_V6, &FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6, &GUID_CALLOUT_PROCESS_BLOCK_UDP_V6 },
};

error = FwpmTransactionBegin(hEngine, 0);
if (error)
    break;

for (auto& fi : filters) {
    FWPM_FILTER filter{};
    filter.filterKey = *fi.guid;
    filter.providerKey = (GUID*)&WFP_PROVIDER_CHAPTER13;
    WCHAR filterName[] = L"Block filter based on PID";
    filter.displayData.name = filterName;
    filter.weight.uint8 = 8;
}

```

```

        filter.weight.type = FWP_UINT8;
        filter.layerKey = *fi.layer;
        filter.action.type = FWP_ACTION_CALLOUT_UNKNOWN;
        filter.action.calloutKey = *fi.callout;
        FwpmFilterAdd(hEngine, &filter, nullptr, nullptr);
    }
    error = FwpmTransactionCommit(hEngine);
} while (false);

```

For every filter, we set its unique key (filterKey member), a display name, our provider, a weight of 8 (“medium” weight), the layer GUID, and the action. The action bears some explanation.

The action has two parts - the type, and an optional callout key. Here are the valid values for the type member when adding filters:

- FWP_ACTION_BLOCK - block the operation.
- FWP_ACTION_PERMIT - allow the operation.
- FWP_ACTION_CALLOUT_TERMINATING - use a callout (provided in the calloutKey member), and the callout must classify with “block” or “permit”.
- FWP_ACTION_CALLOUT_INSPECTION - use a callout, that will not block nor permit - it will merely examine the request.
- FWP_ACTION_CALLOUT_UNKNOWN - use a callout that could have in any kind of outcome.

FWP_ACTION_BLOCK and FWP_ACTION_PERMIT only make sense if conditions are applied to the filter. Otherwise, they will categorically deny or permit everything. We’ve seen an example of using FWP_ACTION_BLOCK with a condition that involves an application ID at the beginning of this chapter to block a “calculator” application from accessing the network.

In our case, we use a callout, and since we only block if needed (and do nothing otherwise), the FWP_ACTION_CALLOUT_UNKNOWN value is the safest to use.

After the filters are added (if needed), BlockProcess sends the request to the driver. Here is the full function:

```

bool BlockProcess(HANDLE hDevice, DWORD pid) {
    if (!AddFilters()) {
        printf("Failed to add filters\n");
        return false;
    }

    DWORD ret;
    return DeviceIoControl(hDevice, IOCTL_PNF_BLOCK_PROCESS, &pid, sizeof(pid),
                           nullptr, 0, &ret, nullptr);
}

```

Similarly, PermitProcess removes a PID from the list of blocked processes by contacting the driver:

```
bool PermitProcess(HANDLE hDevice, DWORD pid) {
    DWORD ret;
    return DeviceIoControl(hDevice, IOCTL_PNF_PERMIT_PROCESS, &pid, sizeof(pid) \
    ,
        nullptr, 0, &ret, nullptr);
}
```

Finally, ClearAll deletes all filters and callouts, since they might not be needed anymore, and then tells the driver to clear its list of blocked processes:

```
bool ClearAll(HANDLE hDevice) {
    DeleteFilters();
    DeleteCallouts();
    DWORD ret;
    return DeviceIoControl(hDevice, IOCTL_PNF_CLEAR,
        nullptr, 0, nullptr, 0, &ret, nullptr);
}
```

DeleteFilters and DeleteCallouts open a handle to the WFP engine and call the appropriate API to delete a filter/callout by key:

```
bool DeleteFilters() {
    HANDLE hEngine;
    DWORD error = FwpmEngineOpen(nullptr, RPC_C_AUTHN_DEFAULT,
        nullptr, nullptr, &hEngine);
    if (error)
        return false;

    FwpmFilterDeleteByKey(hEngine, &GUID_FILTER_V4);
    FwpmFilterDeleteByKey(hEngine, &GUID_FILTER_V6);
    FwpmFilterDeleteByKey(hEngine, &GUID_FILTER_UDP_V4);
    FwpmFilterDeleteByKey(hEngine, &GUID_FILTER_UDP_V6);

    FwpmEngineClose(hEngine);
    return true;
}

bool DeleteCallouts() {
    HANDLE hEngine;
    DWORD error = FwpmEngineOpen(nullptr, RPC_C_AUTHN_DEFAULT,
        nullptr, nullptr, &hEngine);
    if (error)
        return false;
```

```

FwpmCalloutDeleteByKey(hEngine, &GUID_CALLOUT_PROCESS_BLOCK_V4);
FwpmCalloutDeleteByKey(hEngine, &GUID_CALLOUT_PROCESS_BLOCK_V6);
FwpmCalloutDeleteByKey(hEngine, &GUID_CALLOUT_PROCESS_BLOCK_UDP_V4);
FwpmCalloutDeleteByKey(hEngine, &GUID_CALLOUT_PROCESS_BLOCK_UDP_V6);

FwpmEngineClose(hEngine);
return true;
}

```

Testing

The driver is installed in the usual way using the `sc.exe` tool (running elevated), and then started:

```

sc.exe create procnetfilter type= kernel binPath= <path_to_sys_file>
sc.exe start procnetfilter

```

As an example, I ran calculator, but this time issued a block command based on its process id:

```
blockprocess block 10368
```

And verified that calculator is unable to update its currency exchange rates. Opening *WFP Explorer* and examining the callouts view shows the four added callouts (figure 13-18).

Windows Firewall: callout	270 {C3DBED20-0BB6-4BF3-82BD-96732E1E0028}	0x40000 (Registered)	Microsoft Corporation	ALE Listen v4 Layer
NDU Inbound Mac Frame Native Callout	266 {8E449837-F477-11DF-85CE-78E7D1810198}	0x403E2 (Registered)	Microsoft Corporation	Inbound Native MAC Layer fast
Interface Binding Callout	258 {8BC14E84-4287-4C9E-81D4-53233532058}	0x60300 (Register...	Microsoft Corporation	ALE Bind Redirect v4 Layer
Block PID callout	308 {CB4F8E6D-100C-4547-A2A1-7193C3398BAF}	0x40000 (Registered)	WKP2 Chapter 13	ALE Resource Assignment v6 Layer
Block PID callout	307 {200E35C6-7182-4F9C-97D9-34028A225BEC}	0x40000 (Registered)	WKP2 Chapter 13	ALE Resource Assignment v4 Layer
Block PID callout	306 {CF51FD24-566F-4C6D-9BC9-8013E9875F7E}	0x40000 (Registered)	WKP2 Chapter 13	ALE Connect v6 Layer
Block PID callout	305 {5027C277-281A-4AAF-B8E6-95C05E857859}	0x40000 (Registered)	WKP2 Chapter 13	ALE Connect v4 Layer
Block PID callout	304 {80C342E3-1E53-4D6F-9844-030F5AEE1E54}	0x60020 (Register...		Inbound Transport v6 Layer
WFP Built-in Psec Inbound Tunnel v6 La...	6			
IPXlet Forward IPv4 filter	300 {B255C296-7E0C-4115-95F3-87F24A8A1162}	0x40000 (Registered)		IP Forward v4 Layer

Figure 13-18: The added callouts

Similarly, we expect four filters to be added using these callouts (filters view in *WFP Explorer*, see figure 13-19).

WFP FILTERS	CALLOUT	PROVIDER	NAME	PERMISSION	OWNER
Workplace Gateway Manager (1.0.8308.23211)	ALE Receive/Accept v6 Layer	Microsoft Corporation	0x10AC0	0x9 0x9000002000000000	3 Permit (None)
Block filter based on PID	ALE Connect v4 Layer	WKP2 Chapter 13	0x100CA	0x8 0x8000000000000000	0 Callout Unknown Block PID callout
Block filter based on PID	ALE Resource Assignment v6 Layer	WKP2 Chapter 13	0x100CD	0x8 0x8000000000000000	0 Callout Unknown Block PID callout
Block filter based on PID	ALE Resource Assignment v4 Layer	WKP2 Chapter 13	0x100CC	0x8 0x8000000000000000	0 Callout Unknown Block PID callout
Block filter based on PID	ALE Connect v6 Layer	WKP2 Chapter 13	0x100CB	0x8 0x8000000000000000	0 Callout Unknown Block PID callout
winddef_stream_v4	Stream v4 Layer		0x10CE4	(Empty) 0x0	0 Callout Terminat... winddef_stream...
TCP_FILTFR	AI F Flow Established v4 Layer		0x10DB4	(Emptv) 0x7C00000000	1 Callout Inception NcAIFlowEstablish...

Figure 13-19: The added filters

You can now use the *permit* option to remove a process from being blocked:

```
blockprocess permit 10368
```

Or remove all processes from being blocked:

```
blockprocess clear
```

Debugging

The *WFP Explorer* tool proved to be very useful in debugging. Making sure the correct callouts and filters are being added is easy to see with this tool. Of course, you can write your own tools that are more specific to the task at hand. The WFP management API is fairly intuitive to use and is documented well enough. You may find the source code of *WFP Explorer* (<https://github.com/zodiacon/WFPExplorer>) useful for your own work with the management API.

Summary

WFP is a powerful platform that provides lots of flexibility in filtering network requests. In this chapter, we scratched the surface of WFP, but obviously there is a lot more, such as pending network operations, examining actual packets, and even modifying packets. All these will have to wait for another book.

Chapter 14: Introduction to KMDF

The *Kernel Mode Driver Framework* (KMDF) was first available with Windows Vista, and later ported to Windows XP and even Windows 2000. Its purpose is to provide a higher level of abstraction over WDM for the purpose of building drivers for hardware devices.

Up until now, we have used WDM only for writing drivers. This is perfectly acceptable since our drivers were not dealing with hardware devices. Using KMDF to write non-hardware drivers has marginal benefits, and at least one disadvantage, as it adds a dependency to the driver with potentially little value.

In this chapter, we'll examine the fundamentals of KMDF, and see how we can create the *Booster* driver from chapter 4 using KMDF. We will get some advantages when using KMDF, such as seeing (and managing) our device in *Device Manager*.

In this chapter:

- **Introduction to WDF**
 - **Introduction to KMDF**
 - Object Creation
 - The Booster KMDF Driver
 - The INF File
 - The User-Mode Client
 - Installing and Testing
 - Registering a Device Class
 - Summary
-

Introduction to WDF

The *Windows Driver Model* (WDM) we have been using throughout this book was released with Windows 2000 and Windows 98 (“Consumer Windows”) as a way to write source-compatible drivers for these two platforms. Windows NT 4 and Windows 95 had different driver models making it more difficult for hardware vendors to release drivers, as two separate drivers had to be written that did not share any code.

With WDM, many types of kernel drivers for hardware devices could be written with a shared source code, being compiled separately on Windows 2000 and Windows 98. This mostly worked well and

made it easier for hardware vendors to build kernel drivers for their hardware. The same process applied to subsequent operating systems, being Windows XP and Windows ME.

Obviously, today the Consumer Windows line of operating systems is no more, so the source code compatibility provided by WDM is no longer a true advantage. With time, some deficiencies of WDM were showing. The most important one was lack of built-in support for handling Plug & Play and Power Management IRPs properly. Most WDM drivers would copy such code from existing Microsoft samples that were close to what they needed, adjusting the code to the specifics of their hardware. In some cases, this “boilerplate” Plug & Play and Power code contributes to 50% of the entire driver’s source code.

Microsoft realized that WDM is too low-level for hardware-based drivers, so they came up with the *Windows Driver Frameworks* (WDF), formerly known as *Windows Driver Foundation* as a solution to these issues. WDF’s first version was released in 2006, coinciding with the release of Windows Vista. WDF consists of two parts:

- KMDF - the replacement of WDM; it’s a library that layers on top of WDM; WDM is still the fundamental kernel driver model in Windows.
- UMDF - the *User Mode Driver Framework*, which allows certain types of drivers to be written in user mode.

UMDF is not in scope of this book as it’s about writing driver in user-mode, contrary to what this book is about. See the sidebar for more on UMDF.

UMDF

UMDF allows writing drivers for relatively slow hardware devices, such as USB, in user-mode. Writing driver in user mode has several advantages:

- No system crash can ever happen, meaning the robustness of the system is maintained.
- Testing and debugging is easier, and can be done on the same machine.

A UMDF driver is a normal user-mode DLL, hosted by a system-provided host process, *UMDFHost.Exe*. If the DLL causes an exception to occur, the host process could crash, but the system remains intact. The driver then can be reloaded into a new host instance.

UMDF has two fundamental versions:

- Versions 1.x are based on the *Component Object Model* (COM), requiring the driver to implement various interfaces, while also getting implemented framework interfaces.
- Versions 2.x, supported from Windows 8.1 only, use the same APIs as KMDF, so that moving between KMDF and UMDF (in both directions) is much easier.

Does using UMDF imply that it’s possible to access kernel APIs from user-mode? No. The UMDF APIs communicate with a *Reflector* driver that sits in kernel mode, provided by Microsoft, which is the “go to guy” of the UMDF driver, for performing operations in kernel-mode; the fundamental rules cannot be broken.

UMDF is suitable for slow devices but is not good enough for devices that require handling of interrupts or other high-performance requirements, such as devices for PCI Express. Such drivers must be written as kernel mode drivers.

WDF has been opened-source by Microsoft, and is available at <https://github.com/microsoft/Windows-Driver-Frameworks>. It's even possible to step into this source code while debugging.

Introduction to KMDF

KMDF is a library, a layer on top of WDM. Every KMDF driver starts its life as a WDM driver. “Transforming” the driver into KMDF happens when a KMDF driver object is created in `DriverEntry`. Some of the benefits of KMDF include:

- Boilerplate Plug & Play and Power Management implemented within the framework.
- Consistent object model based on properties, methods and events (Callbacks).
- APIs have consistent naming conventions.
- Object hierarchy support and lifetime management using reference counting.
- Major versions of the framework can run side-by-side.

The KMDF header file to include is `wdf.h`, which should follow `<ntddk.h>` or `<ntifs.h>`, as it depends on their definitions.

KMDF Objects

Objects are the basis of KMDF. Although the APIs are C-based, their management and naming is object based. Example objects include driver, devices, queues and requests. Some object types correspond directly to their underlying WDM object (such as devices and requests), but others are new, providing a higher level of abstraction over some functionality. Each object is accessed via its API, while the object itself is provided as a “handle”, rather than a true pointer to a structure.



KMDF is implemented with C++, so each “handle” does correspond to a C++ object.

Objects have properties, methods, and events, having the following attributes:

- Properties - replace direct field access. Function names include Get or Set as part of the name (for properties that cannot fail), or Assign/Retrieve for properties that may fail. The format for property APIs is `Wdf<Object>Set/Get/Assign/Retrieve<Desc>`
- Methods - perform operations on objects. Naturally, these can have return values. Methods have the format `Wdf<ObjectType><Operation>`

- Events - can be registered by the driver, providing a callback to handle some scenario. Event names have the format Evt<ObjectType><Event>

Figure 14-1 shows the KMDF object hierarchy with the “handle” names for the various object types supported. We’ll use some of these object types later on, when we write a KMDF-equivalent driver to the *Booster* driver from chapter 4.

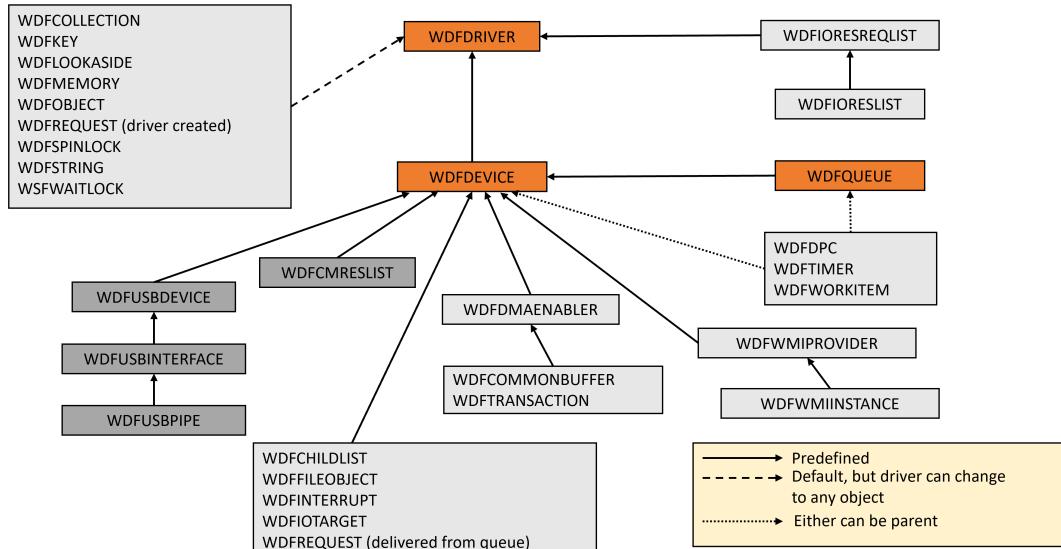


Figure 14-1: KMDF Object Hierarchy

KMDF objects are reference counted. Normally, the driver writer does not have to explicitly manage that lifetime, as a parent object will “release” its child objects when the parent is destroyed. Since all objects are somewhere in a hierarchy, manual referencing or dereferencing is not needed. There are cases, however, that a driver may wish to extend the lifetime of an object. For example, the driver might wish to log some information related to a KMDF object asynchronously using a work item. For this purpose KMDF provides two generic lifetime-management APIs:

```
void WdfObjectReference(WDFOBJECT object);
void WdfObjectDereference(WDFOBJECT object);
```

Every KMDF object supports two events related to its lifetime: EvtObjectCleanup and EvtObjectDestroy. The EvtObjectDestroy callback is invoked just before the object is destroyed - its reference count is zero. EvtObjectCleanup is raised earlier, when the object is in the process of being deleted, but there still might be outstanding references to it. The object should release any references it holds to other objects. The primary use case of this event is to break circular references, which is the primary concern in any reference-counting system.

Core Object Types

The most important and useful object types in KMDF are the following:

- WDFDRIVER - represents the driver. It's a wrapper over the WDM DRIVER_OBJECT object provided in DriverEntry. Creating a WDFDRIVER "transforms" the driver in KMDF.
- WDFDEVICE - represents a device (logical or physical). It's a wrapper around a WDM DEVICE_-OBJECT.
- WDFQUEUE - represents a queue of requests. There is no WDM equivalent to this object type, as its purpose is to allow handling IRPs in a driver-selected way. It supports three types of queues: sequential, parallel, and manual.
- WDFREQUEST - represents a request. It's a wrapper over a WDM IRP.

Object Creation

KMDF makes it relatively easy to create objects, as it follows a consistent pattern for all object types. Here is the `WdfDriverCreate` API as an example:

```
NTSTATUS WdfDriverCreate(
    _In_      PDRIVER_OBJECT DriverObject,
    _In_      PCUNICODE_STRING RegistryPath,
    _In_opt_  PWDF_OBJECT_ATTRIBUTES DriverAttributes,
    _In_      PWDF_DRIVER_CONFIG DriverConfig,
    _Out_opt_ WDFDRIVER* Driver);
```

The function starts with mandatory parameters (`DriverObject` and `RegistryPath` in this case), followed by two data structures. The first (`WDF_OBJECT_ATTRIBUTES`) is generic and appears in every "Create" KMDF API. The second is a specific structure for further customization (`WDF_DRIVER_CONFIG` in this case).

The generic structure pointer can be NULL, providing "default" behavior. Here is its declaration (with the source-provided comments intact):

```
typedef struct _WDF_OBJECT_ATTRIBUTES {
    //
    // Size in bytes of this structure
    //
    ULONG Size;
    //
    // Function to call when the object is deleted
    //
    PFN_WDF_OBJECT_CONTEXT_CLEANUP EvtCleanupCallback;
    //
    // Function to call when the objects memory is destroyed when the
    // the last reference count goes to zero
    //
    PFN_WDF_OBJECT_CONTEXT_DESTROY EvtDestroyCallback;
```

```

//  

// Execution level constraints for Object  

//  

WDF_EXECUTION_LEVEL ExecutionLevel;  

//  

// Synchronization level constraint for Object  

//  

WDF_SYNCHRONIZATION_SCOPE SynchronizationScope;  

//  

// Optional Parent Object  

//  

WDFOBJECT ParentObject;  

//  

// Overrides the size of the context allocated as specified by  

// ContextTypeInfo->ContextSize  

//  

size_t ContextSizeOverride;  

//  

// Pointer to the type information to be associated with the object  

//  

PCWDF_OBJECT_CONTEXT_TYPE_INFO ContextTypeInfo;  

} WDF_OBJECT_ATTRIBUTES, *PWDF_OBJECT_ATTRIBUTES;
}

```

Most of the members are self-explanatory, but not all. When using it, it's recommended to start with a sensible instance - this is what the `WDF_OBJECT_ATTRIBUTES_INIT` inline function is for:

```
VOID WDF_OBJECT_ATTRIBUTES_INIT(_Out_ PWDF_OBJECT_ATTRIBUTES Attributes);
```

Its source is provided directly - it sets the `Size` member to the `sizeof` the structure, zeroes out everything, and then sets two members to specific values: `ExecutionLevel` to `WdfExecutionLevelInheritFromParent` and `SynchronizationScope` to `WdfSynchronizationScopeInheritFromParent`, both of which can be considered "default". These enumerations define the value zero to be invalid.

Using `WDF_OBJECT_ATTRIBUTES_INIT` is not needed if nothing needs to change - passing `WDF_NO_OBJECT_ATTRIBUTES` (defined as `NULL`) for this pointer to the creation function is sufficient. Notice the `EvtCleanupCallback` and `EvtDestroyCallback` discussed earlier; this is where you would set these if needed.

Back to `WdfDriverCreate` - The second structure is a more specific one, where there is always a helper macro to initialize it - `WDF_DRIVER_CONFIG_INIT` in this case. It takes the "config" structure, and required parameters. After initialization, you can change other members in the structure. The `WDF_DRIVER_CONFIG` structure is defined like so:

```

typedef struct _WDF_DRIVER_CONFIG {
    //
    // Size of this structure in bytes
    //
    ULONG Size;
    //
    // Event callbacks
    //
    PFN_WDF_DRIVER_DEVICE_ADD EvtDriverDeviceAdd;
    PFN_WDF_DRIVER_UNLOAD     EvtDriverUnload;
    //
    // Combination of WDF_DRIVER_INIT_FLAGS values
    //
    ULONG DriverInitFlags;
    //
    // Pool tag to use for all allocations made by the framework on behalf of
    // the client driver.
    //
    ULONG DriverPoolTag;
} WDF_DRIVER_CONFIG, *PWDF_DRIVER_CONFIG;

```

The fact that only the driver’s “add device” handler is required in `WDF_DRIVER_CONFIG_INIT` (discussed later) indicates that the other members have sensible defaults.

The final parameter to a creation function is the resulting object handle. In the case of `WdfDriverCreate` it’s a `WDFDRIVER*` where the result should land. This last parameter is optional - specifying `NULL`, or more elegantly `WDF_NO_HANDLE` indicates the caller is not interested in the resulting handle. This is typical for cases where the handle can later be retrieved independently. We’ll see both cases later on.

Once the “pattern” of creation functions is understood, it makes it relatively easy to use any creation function. Here is another example, to solidify the pattern:

```

NTSTATUS WdfIoQueueCreate(
    _In_      WDFDEVICE Device,
    _In_      PWDF_IO_QUEUE_CONFIG Config,
    _In_opt_  PWDF_OBJECT_ATTRIBUTES QueueAttributes,
    _Out_opt_ WDFQUEUE* Queue);

```

`WdfIoQueueCreate` is used for creating queues - we’ll see a concrete example later - it has the ingredients discussed before: required parameters (`Device`), a specific structure (`WDF_IO_QUEUE_CONFIG`) initialized with `WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE` or `WDF_IO_QUEUE_CONFIG_INIT` (some extra flexibility here), and then the generic object attributes structure (`QueueAttributes`), with the final parameter being the returned queue handle. The two structures seem to be in reverse order compared to `WdfDriverCreate` - there is no good reason that I could find for this discrepancy.

Context Memory

When creating a device object in WDM, a device extension size can be specified with the second argument to `IoCreateDevice`. If the value is non-zero, the kernel will allocate the additional bytes at the end of the `DEVICE_OBJECT` structure and point the `DeviceExtension` member to the beginning of that block.

KMDF extends this idea by allowing any KMDF object to be associated with driver-specific memory block (context). This makes it easy to track any required state along with an associated object. The first step in allocating a some context memory associated with a KMDF object is to define the structure of that extra memory. For example:

```
struct MyDeviceContext {
    // members
};
```

Then, use a macro provided by KMDF that conveniently creates a function for accessing the memory:

```
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(MyDeviceContext, DeviceGetContext)
```

The macro creates the function `DeviceGetContext` that can be used to retrieve a pointer to the data (`MyDeviceContext`) after allocation. To make the actual allocation, the context size must be specified within the generic `WDF_OBJECT_ATTRIBUTES` structure. A convenient macro can initialize such structure before creating the actual object. Here is an example assuming a device object:

```
WDF_OBJECT_ATTRIBUTES devAttr;
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(&devAttr, MyDeviceContext);
status = WdfDeviceCreate(&DeviceInit, &devAttr, &device);
if(NT_SUCCESS(status)) {
    MyDeviceContext* context = DeviceGetContext(device);
    // use context
}
```

The Booster KMDF Driver

To demonstrate writing a KMDF driver, while bringing the previous sections into practical use, and showing other parts of KMDF such as request processing, we'll build a *Booster* driver (and client), similar to the one from chapter 4, but based on KMDF instead of WDM. While working on the driver, we'll compare and contrast the KMDF way versus WDM.

To begin, we'll create a new project named *Booster* of type *Kernel Mode Driver, Empty (KMDF)*, contrary to the *WDM Empty Driver* we've used until now. Note that there are other templates for KMDF, such as *Kernel Mode Driver (KMDF)*, which creates a non-empty project. Since we want to do everything from scratch, we'll use the "empty" template.

The project created is not truly empty - it has an INF file present. In the WDM case, we used to delete it. This time we'll keep it, as it's required to get some of the niceties, such as getting our device listed in *Device Manager*. Technically, we could have done that with WDM as well.

We'll examine the INF file later. For now, let's proceed with the main parts of the driver's code.

Driver Initialization

We'll add a standard C++ file to the project named *Booster.cpp*, and write the standard `DriverEntry` prototype. The first thing to do in a KMDF driver is "transform" it to such from WDM. This is done by creating the root KMDF driver object, wrapping the WDM-provided `DRIVER_OBJECT`:

```
extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath) {
    WDF_DRIVER_CONFIG config;
    WDF_DRIVER_CONFIG_INIT(&config, BoosterDeviceAdd);

    return WdfDriverCreate(DriverObject, RegistryPath,
        WDF_NO_OBJECT_ATTRIBUTES, &config, WDF_NO_HANDLE);
}
```

`WdfDriverCreate` accepts the driver object and Registry path passed to `DriverEntry` along with the "config" and "attributes" structures as discussed in the creation "pattern" earlier in this chapter.

Compared to a WDM driver, `DriverEntry` seems lacking - two crucial pieces are missing: device creation and symbolic link creation. Instead, `WDF_DRIVER_CONFIG_INIT` is used to initialize the "config" structure with callback function named `BoosterDeviceAdd`. This callback is called every time a device of this driver is "detected" in the system.



The Unload routine has been set up, as it's being handled by KMDF automatically.

Since our driver is not handling any hardware device, true Plug & Play cannot detect it. Instead, the INF file indicates (as we'll see later when we take a closer look at it) that whenever the driver is loaded, it should be treated as if its first (and only) device is "discovered", and so the `AddDevice` callback must be invoked (`BoosterAddDevice` in this case). This is where we'll create the device object and symbolic link.



KMDF vs. WDM

Behind the scenes, the `AddDevice` callback is stored in the `DriverObject->DriverExtension->AddDevice` member in the driver object.

The `AddDevice` callback is where all the magic happens. We need to do three things in that callback:

- Create a device object
- Create a symbolic link
- Create at least one queue

Let's see what each item entails. First, creating a device object:

```
NTSTATUS BoosterDeviceAdd(WDFDRIVER Driver, PWFDEVICE_INIT DeviceInit) {
    UNREFERENCED_PARAMETER(Driver);
    WDFDEVICE device;
    auto status = WdfDeviceCreate(&DeviceInit, WDF_NO_OBJECT_ATTRIBUTES,
        &device);
    if (!NT_SUCCESS(status))
        return status;
```

The *AddDevice* callback receives the driver object handle and a helper structure, `WDFDEVICE_INIT`, which is not publicly defined, but there are APIs to manage its contents. In our case, we don't need to do anything. `WdfDeviceCreate` accepts a pointer to it, which means it can replace it with a new object. The returned `WDFDEVICE` is a handle to the newly created device object.

What is missing compared to WDM? We used to pass a device name to `IoCreateDevice`, but no such name is provided in the above call. The reason will become clear with the next initialization - the symbolic link.

With the drivers we've written so far (not including filters), we provided an device name and an explicit symbolic link. In the hardware space (what KMDF was built for), that is unlikely to be a good idea. For example, suppose we're writing a driver for a printer device. What should the device name be? What should the symbolic link name be? "Printer1"? "MyPrinter"?

Using arbitrary strings has several drawbacks:

- The chosen name may collide with an existing name.
- Generating multiple names is challenging if more than one device of that type is connected to the system. We would have to manage "Printer1", "Printer2", etc. This is not easy, as "Printer1" might be later disconnected, and then reconnected again. What should its symbolic name be then?
- These strings don't mean anything to the system. How can a client application enumerate all (say) printers in the system? What "makes" a printer device?

All these above issues are mostly applicable to hardware-based devices. Our *Booster* device is going to be a singleton in the system (no other *Booster* devices can be connected), so perhaps the above concerns are irrelevant. But we will treat our booster device similar to a hardware device in this sense, to show the flexibility that we get if we adhere to that model.

What is that model? How can we solve the above issues? The I/O system provides the idea of *Device Interfaces*. A device interface is identified with a GUID, but from a conceptual perspective it's best to think of these just like interfaces in object-oriented code.

An interface is an abstraction that defined some kind of expected behavior, where multiple implementations of that behavior are possible. The way to solve the above issues is to register the device as “implementing” one (or more) interfaces. In a case of printers, and many other “standard” device, Microsoft has already defined those devie interfaces with well-known (and documented) GUIDs.

A printer driver can say “register my device as a printer”. If a driver is for a multifunction device, like a printer/scanner/fax set of devices which are part of the same hardware, then such a driver needs to register itself as “implementing” three interfaces - printer, scanner and fax. Each such registration creates a unique, repeatable, symbolic link, which is what we need.

With KMDF, the call to make (for each supported interface) is to `WdfDeviceCreateDeviceInterface`:

```
NTSTATUS WdfDeviceCreateDeviceInterface(
    _In_      WDFDEVICE Device,
    _In_      CONST GUID* InterfaceClassGUID,
    _In_opt_  PCUNICODE_STRING ReferenceString);
```

The above API requires the device object, the GUID to register it with, and the result is provided by `ReferenceString`, which is the resulted symbolic link. It is optional, since the driver has no use for it - instead, it’s the client that needs the symbolic link. How can a client get the symbolic link? It will have to use certain user-mode APIs to “locate” a device that implement the *Booster* “interface”. We’ll see those later when we write a user-mode client.

Since our *Booster* device is unique, there is no predefined device interface we can use. Instead, we’ll generate a GUID and consider that the *Booster*’s device interface. Think of it as “what does it mean to be a booster device?”. We’ll add that GUID to the header file shared with user-mode clients, as it’s needed in order to locate the device.

We’ll add a *BoosterCommon* header file to the project, which has the same pieces as the one from earlier versions - the supported control code and the `ThreadData` structure. Additionally, it will have our generated GUID:

```
#include <initguid.h>

#define BOOSTER_DEVICE 0x8001

#define IOCTL_BOOSTER_SET_PRIORITY \
CTL_CODE(BOOSTER_DEVICE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

struct ThreadData {
    ULONG ThreadId;
    int Priority;
};

// {49BDF7E8-8AD1-4852-9FB6-833279A1545F}
DEFINE_GUID(GUID_Booster, 0x49bd7e8, 0x8ad1, 0x4852, \
0x9f, 0xb6, 0x83, 0x32, 0x79, 0xa1, 0x54, 0x5f);
```

The GUID was generated with the *Create GUID* tool shown in figure 13-15.

Back to BoosterAddDevice - here is the call to `WdfDeviceCreateDeviceInterface`:

```
status = WdfDeviceCreateDeviceInterface(device, &GUID_Booster, nullptr);
if (!NT_SUCCESS(status))
    return status;
```



KMDF vs. WDM

`WdfDeviceCreateDeviceInterface` calls `IoRegisterDeviceInterface` behind the scenes.

The next step in the *AddDevice* callback is to create a request queue. A queue is an abstraction provided by KMDF for handling requests (IRPs). When a request comes in, such as IRP_MJ_CREATE, IRP_MJ_-READ or IRP_MJ_WRITE, KMDF takes control of the request. Internally, there are three “packages” used by KMDF for request processing:

- I/O Package - handles “standard” requests like Create, Read and Device I/O Control
- P&P/Power package - handles IRP_MJ_PNP (Plug & Play) and IRP_MJ_POWER (Power Management) requests
- WMI package - handles *Windows Management Instrumentation* (WMI) requests

Figure 14-2 shows the way these packages are logically connected internally and to request queues.

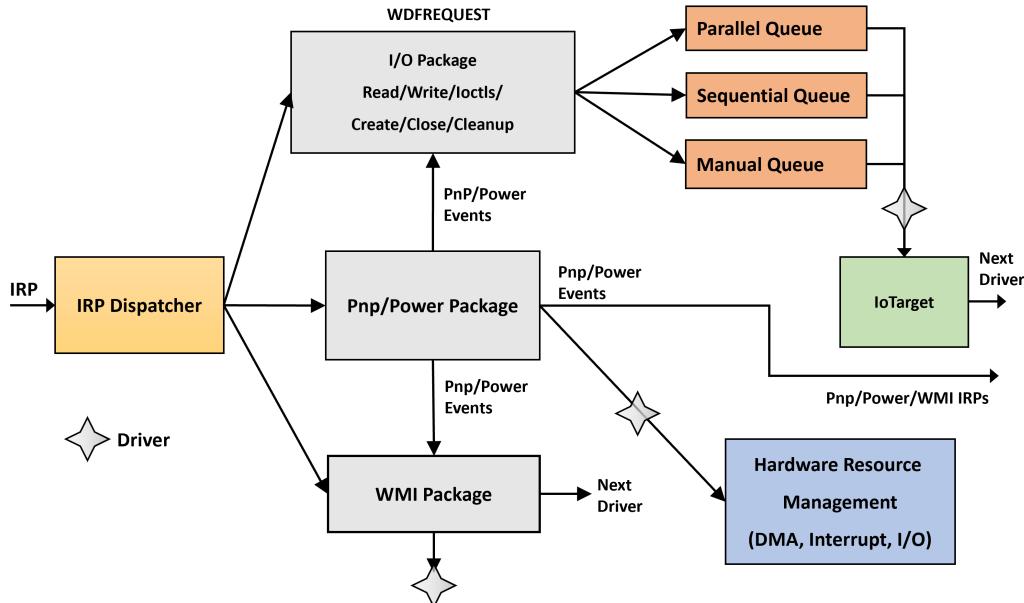


Figure 14-2: Request handling

Since the booster device is not Plug & Play, and doesn't support WMI, we only need to concern ourselves with "standard" requests. At least one queue is required to handle such requests. Three possible queues are provided:

- Sequential queue - guarantees that only one request is handled at a time
- Parallel queue - any number of requests can be thrown at the driver at the same time
- Manual queue - the driver decides when to pull the next request for processing

Since the booster driver holds no state, there is no particular limit to the number of requests that be handled concurrently - a parallel queue is the way to go. If there was some state, we could use a sequential queue that would make it easier to handle requests without manually adding synchronization, at the possible expense of lower performance with such requests, since they would be handles in a classic First-In-First-Out (FIFO) queue.

To create a queue, we need to initialize its configuration, which mostly means which requests should be handled by that queue, and then call `WdfIoQueueCreate`:

```
WDF_IO_QUEUE_CONFIG config;
WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&config, WdfIoQueueDispatchParallel);
config.EvtIoDeviceControl = BoosterDeviceControl;

WDFQUEUE queue;
status = WdfIoQueueCreate(device, &config, WDF_NO_OBJECT_ATTRIBUTES, &queue);
```

`WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE` initializes a `WDF_IO_QUEUE_CONFIG` structure to be a parallel queue (`WdfIoQueueDispatchParallel` enumeration), and set the queue to be the default. The default queue is used for any request that does not have a specific handler; one default queue must exist, and if there is just one queue, it must be the default.

The booster driver needs to handle `IRP_MJ_DEVICE_CONTROL`, which is why the `EvtIoDeviceControl` event (callback) is set to point to the driver's handler (`BoosterDeviceControl`). `WDF_IO_QUEUE_CONFIG` looks like so:

```
typedef struct _WDF_IO_QUEUE_CONFIG {
    ULONG Size;
    WDF_IO_QUEUE_DISPATCH_TYPE DispatchType;
    WDF_TRI_STATE PowerManaged;
    BOOLEAN AllowZeroLengthRequests;
    BOOLEAN DefaultQueue;
    PFN_WDF_IO_QUEUE_IO_DEFAULT EvtIoDefault;
    PFN_WDF_IO_QUEUE_IO_READ EvtIoRead;
    PFN_WDF_IO_QUEUE_IO_WRITE EvtIoWrite;
    PFN_WDF_IO_QUEUE_IO_DEVICE_CONTROL EvtIoDeviceControl;
    PFN_WDF_IO_QUEUE_IO_INTERNAL_DEVICE_CONTROL EvtIoInternalDeviceControl;
    PFN_WDF_IO_QUEUE_IO_STOP EvtIoStop;
```

```

PFN_WDF_IO_QUEUE_IO_RESUME           EvtIoResume;
PFN_WDF_IO_QUEUE_IO_CANCELED_ON_QUEUE EvtIoCanceledOnQueue;
union {
    struct {
        ULONG NumberOfPresentedRequests;
    } Parallel;
} Settings;
WDFDRIVER                           Driver;
} WDF_IO_QUEUE_CONFIG, *PWDF_IO_QUEUE_CONFIG;

```

You can see the EvtIo* callbacks for various requests and notifications, including EvtIoDefault which is a “catch all” handler for other requests not specified elsewhere.

You may be wondering about the IRP_MJ_CREATE and IRP_MJ_CLOSE handlers. These are handled automatically by the framework (in addition to IRP_MJ_CLEANUP). Create completed the request successfully.



Customizing handlers for Create, Close, and Cleanup is possible with event callbacks that can be applied on the DeviceInit structure using a WDFFILEOBJECT object. See the documentation for WDF_FILEOBJECT_CONFIG_INIT and WdfDeviceInitSetFileObjectConfig.

Here is the full *AddDevice* callback for easy reference:

```

NTSTATUS BoosterDeviceAdd(WDFDRIVER Driver, PWDFDEVICE_INIT DeviceInit) {
    UNREFERENCED_PARAMETER(Driver);
    WDFDEVICE device;
    auto status = WdfDeviceCreate(&DeviceInit, WDF_NO_OBJECT_ATTRIBUTES,
        &device);
    if (!NT_SUCCESS(status))
        return status;

    status = WdfDeviceCreateDeviceInterface(device, &GUID_Booster, nullptr);
    if (!NT_SUCCESS(status))
        return status;

    WDF_IO_QUEUE_CONFIG config;
    WDF_IO_QUEUE_CONFIG_INIT_DEFAULT_QUEUE(&config, WdfIoQueueDispatchParallel);
    config.EvtIoDeviceControl = BoosterDeviceControl;

    WDFQUEUE queue;
    status = WdfIoQueueCreate(device, &config, WDF_NO_OBJECT_ATTRIBUTES,
        &queue);

```

```

    return status;
}

```

Device I/O Control Handling

The booster's driver main job is to handle the single I/O control code. The `BoosterDeviceControl` handler that was set in `WDF_IO_QUEUE_CONFIG` must have the following prototype:

```

VOID EVT_WDF_IO_QUEUE_IO_DEVICE_CONTROL(
    _In_    WDFQUEUE Queue,
    _In_    WDFREQUEST Request,
    _In_    size_t OutputBufferLength,
    _In_    size_t InputBufferLength,
    _In_    ULONG IoControlCode);

```

As you can see, the function already provides most of what we need in order to process the request. There is no need to dig into the I/O stack location, as in WDM. The needed information is handed to us on silver platter, so to speak.

We'll start the implementation by examining the given control code:

```

VOID BoosterDeviceControl(WDFQUEUE Queue, WDFREQUEST Request,
    size_t OutputBufferLength, size_t InputBufferLength, ULONG IoControlCode) {
    UNREFERENCED_PARAMETER(InputBufferLength);
    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(Queue);

    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG info = 0;

    switch (IoControlCode) {
        case IOCTL_BOOSTER_SET_PRIORITY:

```

You may be wondering why the code uses `UNREFERENCED_PARAMETER` on the input buffer length. Shouldn't we be checking that as part of processing? As it turns out, even that is not strictly necessary for our case. Here are the next lines of code:

```

ThreadData* data;
status = WdfRequestRetrieveInputBuffer(Request, sizeof(ThreadData),
    (PVOID*)&data, nullptr);
if (!NT_SUCCESS(status))
    break;

```

`WdfRequestRetrieveInputBuffer` accepts the request object, the minimum size of the input buffer (`sizeof(ThreadData)`), the resulting pointer, and an optional variable to receive the actual input buffer size. If the buffer is too small, `WdfRequestRetrieveInputBuffer` returns an appropriate status. All we need to do is bail out if we get a failed status.

The next part of the handler is identical to the WDM case. This is what makes this driver unique:

```

if (data->Priority < 1 || data->Priority > 31) {
    status = STATUS_INVALID_PARAMETER;
    break;
}

PKTHREAD thread;
status = PsLookupThreadByThreadId(UlongToHandle(data->ThreadId), &thread);
if (!NT_SUCCESS(status))
    break;

KeSetPriorityThread(thread, data->Priority);
ObDereferenceObject(thread);
info = sizeof(ThreadData);
break;

```

All that's left to do is complete the request, for which KMDF has a bunch of APIs with different completion details, such as the `Information` and the priority boost. For *Booster*, the following is what is needed:

```

}
WdfRequestCompleteWithInformation(Request, status, info);

```

Here is the full device I/O control handler for easy reference:

```

VOID BoosterDeviceControl(WDFQUEUE Queue, WDFREQUEST Request,
    size_t OutputBufferLength, size_t InputBufferLength, ULONG IoControlCode) {
    UNREFERENCED_PARAMETER(InputBufferLength);
    UNREFERENCED_PARAMETER(OutputBufferLength);
    UNREFERENCED_PARAMETER(Queue);

    auto status = STATUS_INVALID_DEVICE_REQUEST;
    ULONG info = 0;

    switch (IoControlCode) {
        case IOCTL_BOOSTER_SET_PRIORITY:
            ThreadData* data;
            status = WdfRequestRetrieveInputBuffer(Request, sizeof(ThreadData),

```

```

        (PVOID*)&data, nullptr);
if (!NT_SUCCESS(status))
    break;

if (data->Priority < 1 || data->Priority > 31) {
    status = STATUS_INVALID_PARAMETER;
    break;
}

PKTHREAD thread;
status = PsLookupThreadByThreadId(
    UlongToHandle(data->ThreadId), &thread);
if (!NT_SUCCESS(status))
    break;

KeSetPriorityThread(thread, data->Priority);
ObDereferenceObject(thread);
info = sizeof(ThreadData);
break;
}
WdfRequestCompleteWithInformation(Request, status, info);
}

```

That concludes the *Booster*'s driver's code.

The INF File

We'll use an INF file to install the driver. This provides additional flexibility, as we can add files and Registry entries as part of installation without using any code. INF files are not restricted to KMDF, and we could have used them with WDM as well. KMDF practically requires an INF file to be used, as some details cannot be set in code and must be specified in the Registry.

The following is an introduction to INF files. It's far from complete, but should give you a good sense of how it works and how to customize certain parts.

INF files use the classic INI file syntax, where there are sections in square brackets, and underneath a section there are directives in the form "name=value". These entries are instructions to the installer that parses the file, essentially telling it to do two types of operations: copy files to specific locations and making changes to the Registry.

Although the INF file looks "flat" - just sections, with directives in each section, it in fact models a tree, where a directive in one section may point to another section by name.

It seems more appropriate to use a naturally hierarchical format, such as XML or JSON. When INF was invented, neither XML nor JSON existed. I would have expected Microsoft to adapt XML or JSON at some point, but this hasn't happened at the time of this writing, and unlikely to happen in the future.

The Version Section

The *Version* section is mandatory in an INF file. The following is generated by the WDK project wizard (for the empty KMDF project type) for the *Booster* project, also showing the provided comments (anything after a semicolon is considered a comment until the end of the line):

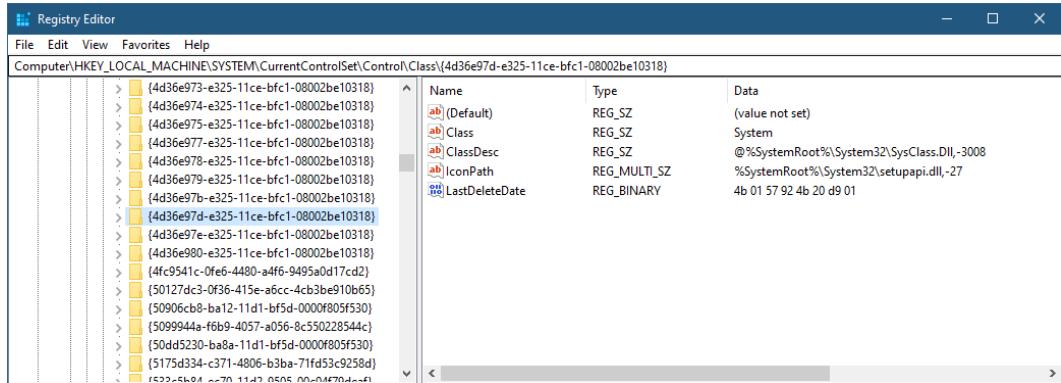
```
[Version]
Signature="$WINDOWS NT$"
Class=System ; TODO: specify appropriate Class
ClassGuid={4d36e97d-e325-11ce-bfc1-08002be10318} ; TODO: specify appropriate Cl\
assGuid
Provider=%ManufacturerName%
CatalogFile=Booster.cat
DriverVer= ; TODO: set DriverVer in stampinf property pages
PnpLockdown=1
```

The *Signature* directive must be set to the magic string "\$Windows NT\$". The reason for this name is historical, and not important for this discussion.

The *Class* and *ClassGuid* directives are mandatory and specify the class (type or group) to which this driver belongs to. The generated INF contains an example class, *System*, which is a predefined class defined by Microsoft long ago, with its associated GUID.

The "TODO" comments indicate that we should probably change that to an "appropriate" class. What is appropriate here? If the devices the driver manages are one of the predefined types (such as printer, disk, display, etc.), then that one should be used. These predefined *device classes* are listed in the WDK docs. For the *Booster* driver, it's more appropriate to generate our own *Booster* "category" (class), by generating another GUID. For the current driver, we'll stick with the default *System* class. We'll generate our own later in this chapter.

The *Class* is mostly useful for hardware-based drivers, as some functionality can be specified based on the driver's class, such as loading certain filters. The list of all classes and their properties can be found in the Registry under *HKLM\System\CurrentControlSet\Control\Class*. Each class is uniquely identified by a GUID; the string name is just a human-readable helper. Figure 14-3 shows the *System* class entry in the Registry.

Figure 14-3: The *System* device class in the Registry

Back to the *Version* section in the INF - the *Provider* directive is the name of the driver publisher. It doesn't mean much in practical terms, but might appear in some UI, so should be something meaningful. The value set by the WDK template is %ManufacturerName%. Anything within percent symbols is treated like a "macro" - to be replaced by the actual value specified in another section called *Strings*. Here is part of this section (traditionally the last section in the file):

```
[Strings]
SPSVCINST_ASSOCSERVICE= 0x00000002
ManufacturerName="Pavel Yosifovich"
DiskName = "Booster Installation Disk"
Booster.DeviceDesc = "Booster Device"
Booster.SVCDESC = "Booster Service"
```

As you can see, I have replaced the *ManufacturerName* with my name, and removed the original "TODO" set by the project template.

The Install Sections

Notice the comment "Install Section" in the INF. Following it are two sections like so:

```
[Manufacturer]
%ManufacturerName% = Standard, NT$ARCH$
```

```
[Standard.NT$ARCH$]
%Booster.DeviceDesc% = Booster_Device, Root\Booster ; TODO: edit hw-id
```

The *Manufacturer* section is mandatory, where the device installation sections must be listed. Typically there is just one, but technically an INF can install drivers for multiple devices. The string "Standard" forms a name for a section augmented with "NT\$ARCH\$" where "\$ARCH\$" is expanded to the platform name, such as "AMD64". This makes it easy to add sections that target specific architectures, if desired.

The pointed-to section, “Standard.NT\$ARCH\$” has directives pointing to specific device installation instructions (just one in this case). The left part (“%Booster.DeviceDesc%”) is shown in case the Plug & Play manager needs to show some User Interface with a description of the device, but otherwise is not important. The value after the equals sign is comprised of at least two parts. The first is a section name (“Booster_Device” in this case), where installation instructions continue. The second is the unique device ID for this device. The format is generally *Enumerator\ID*, where *Enumerator* is a type of bus in the hardware case (e.g. PCI), or a virtual bus, as it is in our case - the *Root* bus can be used to force a device to load always, which is what we want since the *Booster* device is not a hardware one.

The “TODO” comment indicates this can be changed if desired. We’ll keep the default since it’s basically what we need.

Device Installation

The base name “Booster_Device” is used in multiple sections, all working towards installing the driver with the correct settings. Here are the relevant sections:

```
[Booster_Device.NT]
CopyFiles=Drivers_Dir

[Drivers_Dir]
Booster.sys

;----- Service installation
[Booster_Device.NT.Services]
AddService = Booster,%SPSVCINST_ASSOCSERVICE%, Booster_Service_Inst

; ----- Booster driver install sections
[Booster_Service_Inst]
DisplayName      = %Booster.SVCDESC%
ServiceType     = 1                      ; SERVICE_KERNEL_DRIVER
StartType       = 3                      ; SERVICE_DEMAND_START
ErrorControl    = 1                      ; SERVICE_ERROR_NORMAL
ServiceBinary   = %12%\Booster.sys
```

Booster_Device.NT (applies for any architecture) has a *CopyFiles* directive, pointing to “Drivers_Dir” section that lists the files to copy (*Booster.sys* only in this case).

The *Booster_Device.NT.Services* serves the same purpose as the *CreateService* API (or the *sc.exe* tool we have been using). You can see the service information listed, including *DisplayName*, *ServiceType*, *StartType*, and *ErrorControl*. *ServiceBinary* sets the *ImagePath* value in the Registry, pointing to “%12%\Booster.sys”. This weird “%12%” value represents the *%SystemRoot%\System32\Drivers* directory. Table 14-1 shows some common directory names encoded with a number enclosed by percent symbols.

Table 14-1: Common number to directory mappings

Number	Directory
01	The directory from which the INF file is being installed
10	The Windows directory (same as %SystemRoot%)
11	The System directory (%SystemRoot\System32)
12	The Drivers directory (%SystemRoot\System32\Drivers)
17	The INF directory (%SystemDrive%INF)
20	The Fonts directory
24	Root directory of the system disk (e.g. C:\)
-1	Absolute path

There are a few more sections starting with “Booster_Device”, listed under a comment that reads “CoInstaller Installation”. A *co-installer* is a generic name for any additional installation that may be required besides the driver-specific files. In this case, it’s properly installing KMDF. These sections are boilerplate, and there is no need to touch them.

The User-Mode Client

Let’s now turn our attention to the user-mode client before attempting to install the driver. Most of the user-mode code should remain the same, since the client does not need to know how the driver is implemented.

There is one important change, however - the symbolic link name is not known in advance, and must be found dynamically. Here is the full main function of a client application called *boost* that is very similar to a client we’ve seen before:

```
int main(int argc, const char* argv[]) {
    if (argc < 3) {
        printf("Usage: boost <tid> <priority>\n");
        return 0;
    }

    auto name = FindBoosterDevice();
    if (name.empty()) {
        printf("Unable to locate Booster device\n");
        return 1;
    }

    HANDLE hDevice = CreateFile(name.c_str(), GENERIC_WRITE, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE) {
```

```

    printf("Error: %u\n", GetLastError());
    return 1;
}

ThreadData data;
data.ThreadId = atoi(argv[1]);
data.Priority = atoi(argv[2]);

DWORD bytes;
if (DeviceIoControl(hDevice, IOCTL_BOOSTER_SET_PRIORITY,
    &data, sizeof(data), nullptr, 0, &bytes, nullptr))
    printf("Success!\n");
else
    printf("Error: %u\n", GetLastError());

CloseHandle(hDevice);
return 0;
}

```

As you can see from the above code, the only change compared to a “classic” client is the way the symbolic link is obtained, by calling a helper function, `FindBoosterDevice`. If such a device is found, its symbolic link is returned as a `std::wstring`, which is just handed over to `CreateFile` like always. Learly, that function is the mystery.

We’ll start by adding the required includes:

```

#include <Windows.h>
#include <string>
#include <stdio.h>
#include "..\Booster\BoosterCommon.h"
#include <SetupAPI.h>

```

All the above includes should be familiar, except `<setupapi.h>`. This is where we’ll find the API used to search for devices based on some criteria. Next, we’ll need to add its import library, since it’s implemented in a separate DLL that is not referenced by default:

```
#pragma comment(lib, "setupapi")
```

Now we can start implementing the `FindBoosterDevice` function. Remember that the driver has registered itself with the `GUID_Booster` device interface, which is also provided in the common header file. We need to search for devices that “implement” that device interface:

```
std::wstring FindBoosterDevice() {
    HDEVINFO hDevInfo = SetupDiGetClassDevs(&GUID_Booster, nullptr, nullptr,
        DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
    if (!hDevInfo)
        return L"";
```

The `SetupDiGetClassDevs` API opens a handle to a “device information set” based on the supplied arguments. Here we specify `GUID_Booster` to hone in only on this GUID, and tell the API to search for existing devices only (`DIGCF_PRESENT`, without which the search would be extended to devices that are installed by not currently loaded), and the second flag (`DIGCF_DEVICEINTERFACE`) indicates the API should interpret `GUID_Booster` as a device interfaces, rather than a *device class* (which we’ll see later). Our device class is *System*, so looking for that would return too many results.

The next step is to enumerate the resulting list of devices (if any), where we expect to find one device or no device at all (if the *Booster* driver has not been loaded). The enumeration is done with `SetupDiEnumDeviceInfo` like so:

```
std::wstring result;
do {
    SP_DEVINFO_DATA data{ sizeof(data) };
    if (!SetupDiEnumDeviceInfo(hDevInfo, 0, &data))
        break;
```

The zero indicates the first item in the device set. We could enumerate more by incrementing the index until the call fails. Assuming a single *Booster* device is installed, zero is all we need. Once successful, we can proceed to locate the symbolic link from the first (and only) device interface supported for the *Booster* device:

```
SP_DEVICE_INTERFACE_DATA idata{ sizeof(idata) };
if (!SetupDiEnumDeviceInterfaces(hDevInfo, &data, &GUID_Booster, 0, &idata))
    break;
```

This retrieves the first device interface. Now we need the symbolic link:

```
BYTE buffer[1024];
auto detail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)buffer;
detail->cbSize = sizeof(*detail);
if (SetupDiGetDeviceInterfaceDetail(hDevInfo, &idata, detail,
    sizeof(buffer), nullptr, &data))
    result = detail->DevicePath;
} while (false);
SetupDiDestroyDeviceInfoList(hDevInfo);
return result;
}
```

`SetupDiGetDeviceInterfaceDetail` retrieves the symbolic link based on the information obtained thus far, returned in the `SP_DEVICE_INTERFACE_DETAIL_DATA` member `DevicePath`.

Here is the full function for easy reference:

```
std::wstring FindBoosterDevice() {
    HDEVINFO hDevInfo = SetupDiGetClassDevs(&GUID_Booster, nullptr, nullptr,
                                             DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
    if (!hDevInfo)
        return L"";

    std::wstring result;
    do {
        SP_DEVINFO_DATA data{ sizeof(data) };
        if (!SetupDiEnumDeviceInfo(hDevInfo, 0, &data))
            break;

        SP_DEVICE_INTERFACE_DETAIL_DATA idata{ sizeof(idata) };
        if (!SetupDiEnumDeviceInterfaces(hDevInfo, &data, &GUID_Booster,
                                         0, &idata))
            break;

        BYTE buffer[1024];
        auto detail = (PSP_DEVICE_INTERFACE_DETAIL_DATA)buffer;
        detail->cbSize = sizeof(*detail);
        if (SetupDiGetDeviceInterfaceDetail(hDevInfo, &idata, detail,
                                           sizeof(buffer), nullptr, &data))
            result = detail->DevicePath;
    } while (false);
    SetupDiDestroyDeviceInfoList(hDevInfo);
    return result;
}
```

That's it. This is all we need to get the symbolic link dynamically, based on the device interface we're after (`GUID_Booster`).

Installing and Testing

We have the driver and a client application. Installing the driver cannot be done with a simple `sc.exe create` as we did in the past. We have to tell some installer to parse the INF file and perform everything that is required.

First, we have to copy the files generated by the build process. These include `Booster.inf`, `Booster.sys`, and `Booster.cat`. The latter is a *catalog* file, containing signature information for the driver package. The INF file, by the way, has all its “\$ARCH\$” “macros” expanded.

Once these files are copied to some directory on the target system, we need to use a tool called *devcon.exe*, provided with the Windows SDK, to actually perform the installation. You can find it in a directory like *c:\Program Files (x86)\Windows Kits\10\Tools\10.0.25300.0\x64*. Open an elevated command window, navigate to above path and run the following:

```
devcon install c:\Demo\Booster.inf root\booster
```

The above command assumes that the driver files were copied to *c:\Demo*. The last argument must be the hardware ID specific earlier in the INF file. The reason it's required is that there could be multiple device IDs. I would have expected *DevCon* to select the whatever is in the INF file if there is just one. Currently, it's not doing that. When installing, you'll get the following dialog popping up (figure 14-4).

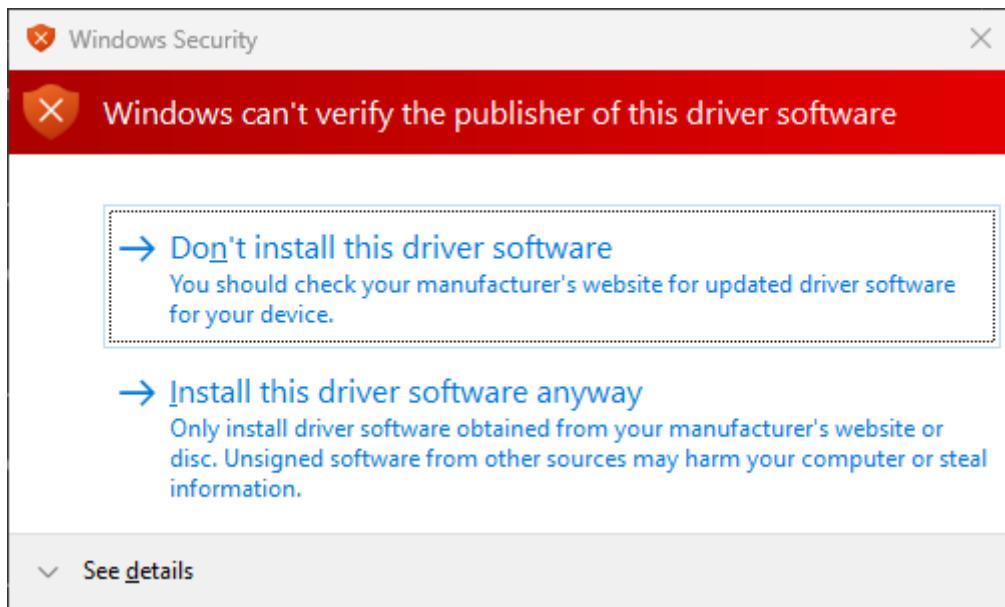


Figure 14-4: Warning installation dialog

The dialog color is based on whether the driver about to be installed is signed or not. In our case, it's unsigned (the system is in test-signing mode), so the color is bright red as a warning. Click the "Install this driver software anyway" option to proceed.



If you try to right-click the INF file and select *Install*, it won't work. This only works with a certain name for the install section (*DefaultInstall*), which is not the name given by the KMDF project template.

Once the driver is installed, you can open *Device Manager* and expand the *System Devices* node - remember the driver was listed in the *system* device class. The booster name should appear (figure 14-5).

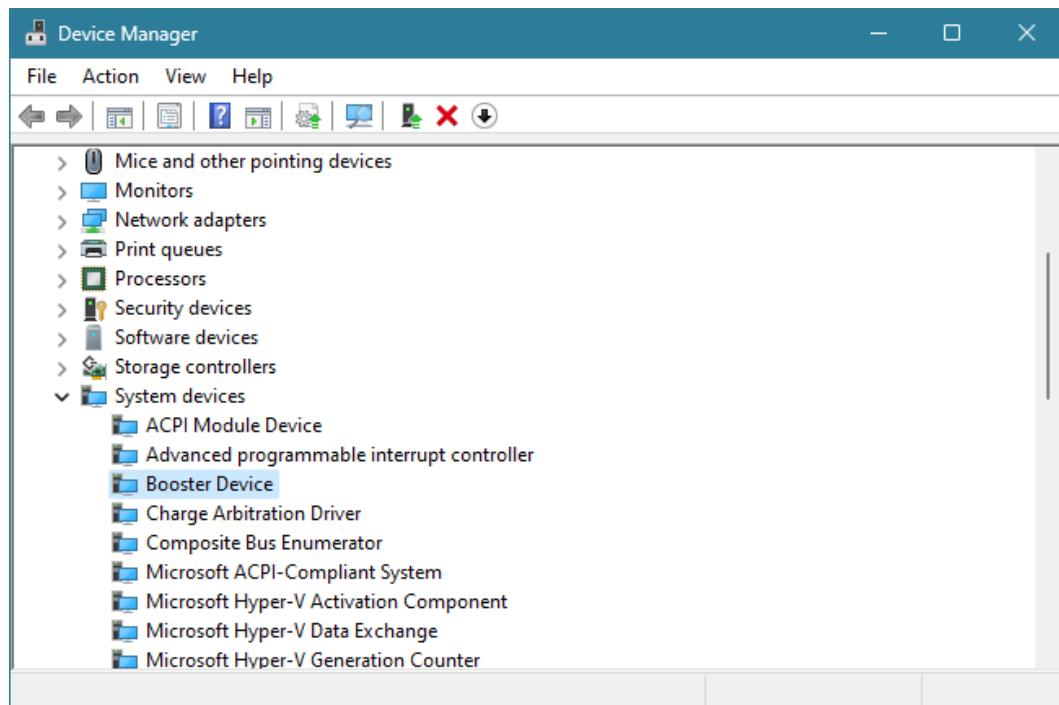


Figure 14-5: Booster device in *Device Manager*

Right-clicking the *Booster* node and selecting properties, and navigating to the *Details* tab, shows various properties of the device. Select *Hardware IDs* from the drop-down combobox and you'll see the familiar *root\booster* name (figure 14-6).

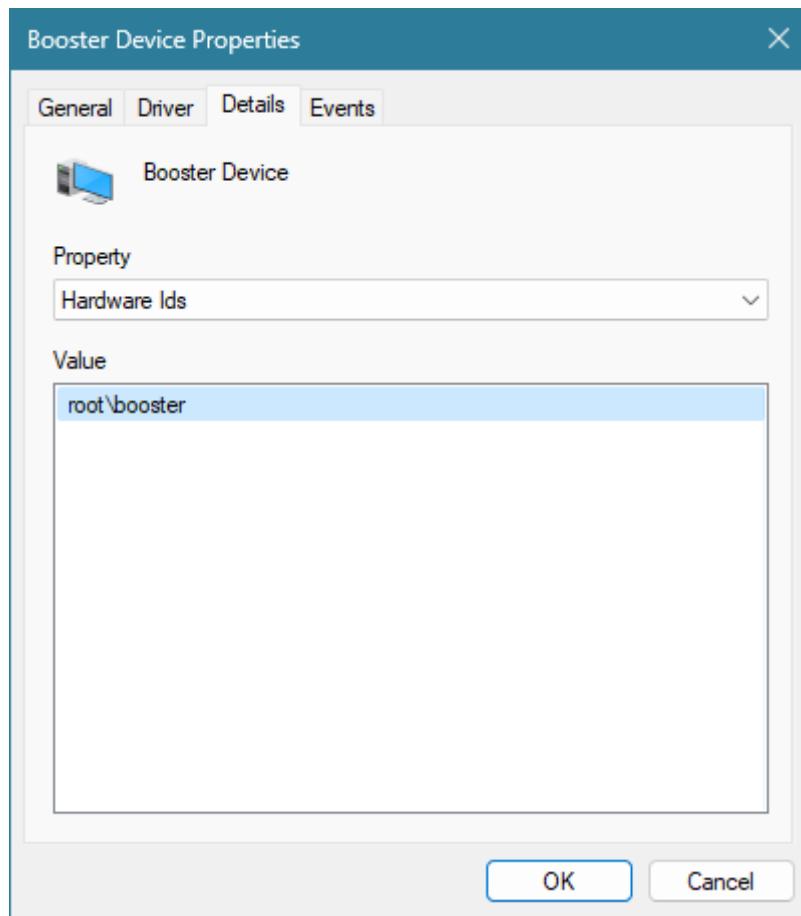


Figure 14-6: Booster's hardware ID

You can browse to the `System32\Drivers` directory where you'll find `Booster.sys`. You can also look in the Registry at the `Booster` key under the standard `Services` key.

What about the device name and symbolic link? Let's take a look using a local kernel debugger.

```
kd> !drvobj booster f
Driver object (fffffd287330f5550) is for:
  \Driver\Booster

Driver Extension List: (id , addr)
(fffff8001f2922a0 fffffd287472f76b0)
Device Object list:
fffffd28733edede0

DriverEntry:  fffff8003f6615c0          Booster
```

```
DriverStartIo: 00000000
DriverUnload: fffff8003f661760          Booster
AddDevice:     fffff8001f292050          Wdf01000!FxDriver::AddDevice

Dispatch routines:
[00] IRP_MJ_CREATE                  fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[01] IRP_MJ_CREATE_NAMED_PIPE      fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[02] IRP_MJ_CLOSE                  fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[03] IRP_MJ_READ                   fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[04] IRP_MJ_WRITE                  fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[05] IRP_MJ_QUERY_INFORMATION    fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[06] IRP_MJ_SET_INFORMATION      fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
...
[16] IRP_MJ_POWER                  fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[17] IRP_MJ_SYSTEM_CONTROL       fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[18] IRP_MJ_DEVICE_CHANGE        fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[19] IRP_MJ_QUERY_QUOTA         fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[1a] IRP_MJ_SET_QUOTA           fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock
[1b] IRP_MJ_PNP                  fffff8001f257ac0 Wdf01000!FxDevice::DispatchWithL\
ock

Device Object stacks:
```

```
!devstack fffffd28733edede0 :
  !DevObj          !DrvObj          !DevExt          ObjectName
  > fffffd28733edede0  \Driver\Booster   fffffd28745ec8fb0
    fffffd28743ef5b10  \Driver\PnpManager fffffd28743ef5c60  0000010a
!DevNode fffffd2872e9d3050 :
  DeviceInst is "ROOT\SYSTEM\0001"
  ServiceName is "Booster"
```

Processed 1 device objects.

You may need to enter .reload to force loading of KMDF symbols so you would see proper symbols. A few things to notice here:

- All the dispatch routines have been “hijacked” by KMDF.
- There is one device object, part of a device node where two devices exist - our *Booster* device and a device created by the Plug & Play manager (part of the virtual *root* bus).
- The device name is *00000010a*, which you can guess is generated by a running index. We expect some symbolic link to be pointing to it.

How do we see the symbolic link without running the client? We could examine the symbolic links directory using *WinObj* and look for a target of *\Device\00000010a* (figure 14-7).

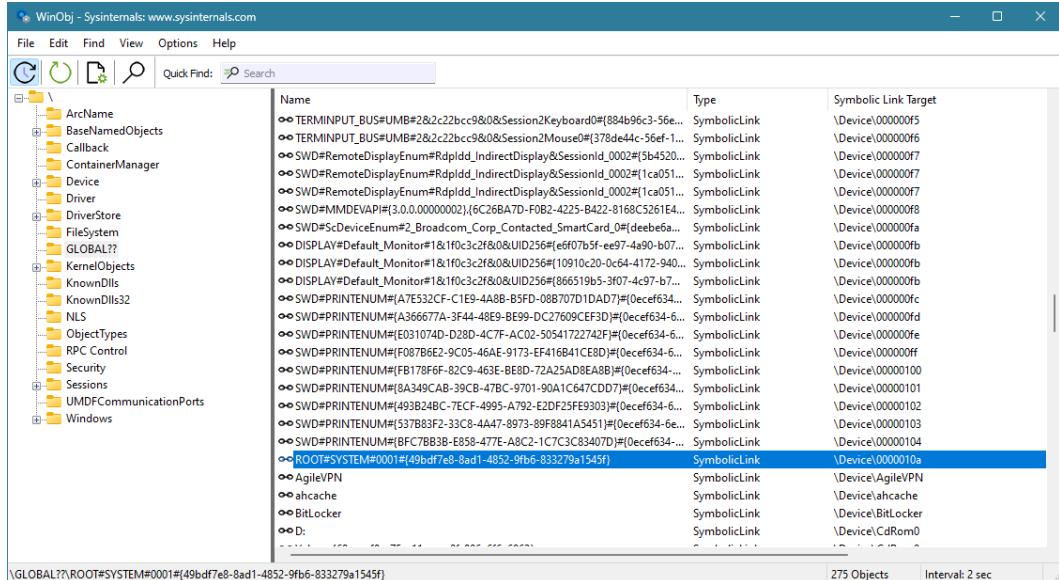


Figure 14-7: Booster’s symbolic link in *WinObj*

Note the name of the symbolic link: “*ROOT#SYSTEM#0001#*”. It consists of a “local” name (the same one shown as *DeviceInst* in the debugger output above where backslashes are replaced by pound signs), and then the *GUID_Booster* GUID in string form.

We can run the user-mode client normally:

```
boost.exe 7752 20
```



Examine other Booster device properties using *Device Manager*

Run *DevCon* again and install another device. What does that look like in *Device Manager* and *WinObj*?

If you're curious as to how the driver is installed via the INF file, you can look at the *DevCon* source code, which is provided as part of the WDK samples on Github. A couple of other tools you might find useful include *Device Explorer*, an enhanced version of *Device Manager* (you can find it in one of my Github repos), as well as *InstDrv* tool - a command line tool that can install a driver based on an INF file without the need to specify a hardware ID - it just installs the first one. The source code is part of the *Device Explorer*'s solution.

Registering a Device Class

Our *Booster* device was registered with the *System* device class, which is inappropriate. It should have its own device class, since it's a "special" device - it is unlike any other device type in the system.

To that end, we can register a new device class by adding a couple of sections in the INF file and setting the information for the new device class. Here are the sections to add:

```
; define new device class
[ClassInstall32]
AddReg=DevClass_AddReg

[DevClass_AddReg]
HKR,,,MyDeviceClassName      ; change as needed
HKR,,SilentInstall,,1
```

And of course the *Version* section has to use the new class and newly generated class GUID.

I have created another project in the same solution named *Booster2* that has the same code, but with the needed changes in the INF so that we get a new device class.

Here are the changes in the INF:

```
[Version]
Signature="$WINDOWS NT$"
Class=BoosterDevice
ClassGuid={AE4151AF-8C29-41C3-BB16-0B3115733333}
Provider=%ManufacturerName%
CatalogFile=Booster2.cat
DriverVer= ; TODO: set DriverVer in stampinf property pages
PnpLockdown=1

; define new device class
[ClassInstall32]
```

AddReg=DevClass_AddReg

[DevClass_AddReg]

HKR,,,BoosterDevice
HKR,,SilentInstall,,1

The GUID in *ClassGuid* was generated with the *Create GUID* tool. Once the generated driver is installed, the new device in *Device Manager* is shown similar to figure 14-8.

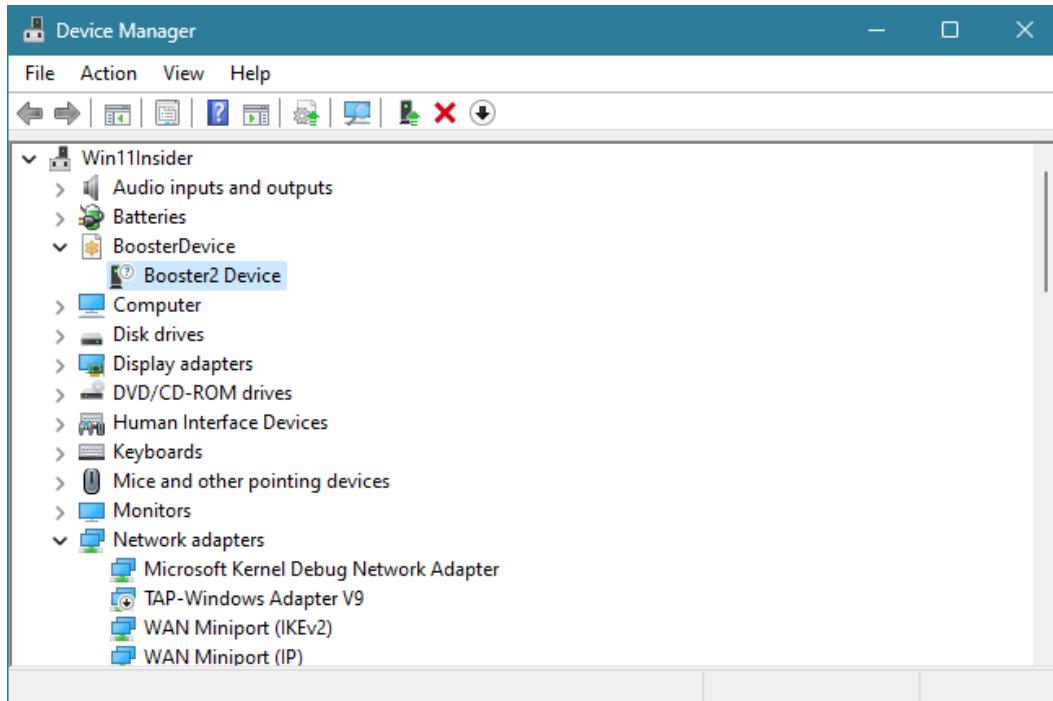


Figure 14-8: A *Booster* device with its own device class

Does the client need to change? Not necessarily. It's still valuable to look for the `GUID_Booster` device interfaces. However, it also makes sense to add the device class `GUID` to the `BoosterCommon.h` header file. Then the search function has the flexibility to search by device class instead of (or in addition to) the device interface.



1. Add the device class `GUID` to `BoosterCommon.h`.
2. Change the search function to look for the device class instead of the device interface.
3. Implement the *Zero* driver from earlier modules using KMDF.

Summary

KMDF provides a higher level of abstraction over WDM. Its real power is clearly visible when writing drivers for hardware-based devices, but as we have seen in this chapter it has some niceties we can take advantage of to simplify coding.

Chapter 15: Miscellaneous Topics

In this last chapter of the book, we'll take a look at various topics that didn't fit well in previous chapters.

In this chapter:

- Driver Signing
 - Driver Verifier
 - Filter Drivers
 - Device Monitor
 - Driver Hooking
 - Kernel Libraries
-

Driver Signing

Kernel drivers are the only official mechanism to get code into the Windows kernel. As such, kernel drivers can cause a system crash or another form of system instability. The Windows kernel does not have any distinction between “more important” drivers and “less important” drivers. Microsoft naturally would like Windows to be stable, with no system crashes or instabilities. Starting from Windows Vista, on 64 bit systems, Microsoft requires drivers to be signed using a proper certificate acquired from a certificate authority (CA). Without signing, the driver will not load.

Does a signed driver guarantee quality? Does it guarantee the system will not crash? No. It only guarantees the driver files have not changed since leaving the publisher of the driver and that the publisher itself is authentic. It's not a silver bullet against driver bugs, but it does give some sense of confidence in the driver.

For a hardware-based driver, Microsoft requires these to pass the *Windows Hardware Quality Lab* (WHQL) tests, containing rigorous tests for stability and driver functionality. If the driver passes these tests, it receives a Microsoft stamp of quality, which the driver publisher can advertise as a sign of quality and trust. Another consequence of passing WHQL is making the driver available through Windows Update, which is important for some publishers.

Starting with Windows 10 version 1607 (“Anniversary update”), for systems that were freshly installed (not upgraded from an earlier version) with *secure boot* on - Microsoft requires drivers to be signed by Microsoft as well as by the publisher. This is true for all types of drivers, not just related to hardware. Microsoft provides a web portal where drivers can be uploaded (must already be signed by the publisher), tested in some ways by Microsoft and finally signed by Microsoft and returned back to the publisher. It may take some time for Microsoft to return the signed driver the first time the driver is uploaded, but later iterations are fairly fast (several hours).

The driver that needs to be uploaded includes the binaries only. The source code is not required.

Figure 15-1 shows an example driver image file from *Nvidia* that is signed by both *Nvidia* and Microsoft on a Windows 10 19H1 system.

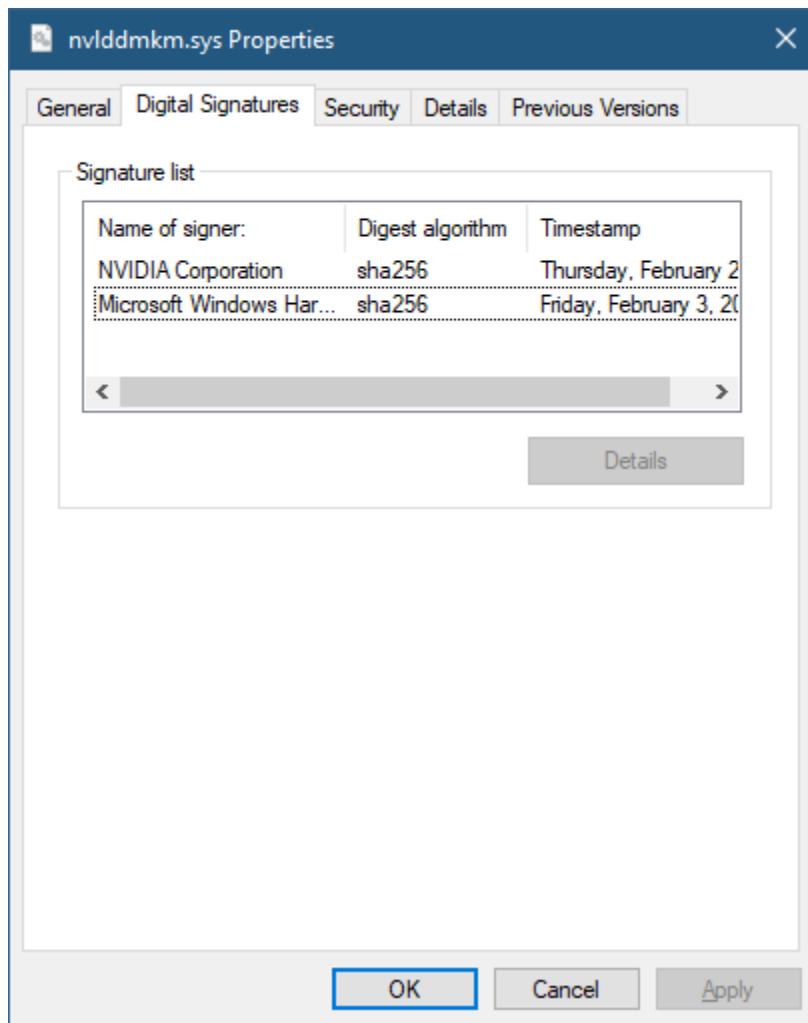


Figure 15-1: Driver signed by vendor and by Microsoft

The first step in driver signing is obtaining a proper certificate from a certificate authority (such as Verisign, Globalsign, DigiCert, Symantec, and others) for at least kernel code signing. The CA will validate the identity of the requesting company, and if all is well, will issue a certificate. The downloaded certificate can be installed in the machine's certificate store. Since the certificate must be

kept secret and not leak, it is typically installed on a dedicated build machine and the driver signing process is done as part of the build process.

The actual signing operation is done with the *SignTool.exe* tool, part of the Windows SDK. You can use Visual Studio to sign a driver if the certificate is installed in a certificate store on the local machine. Figure 15-2 shows the signing properties in Visual Studio.

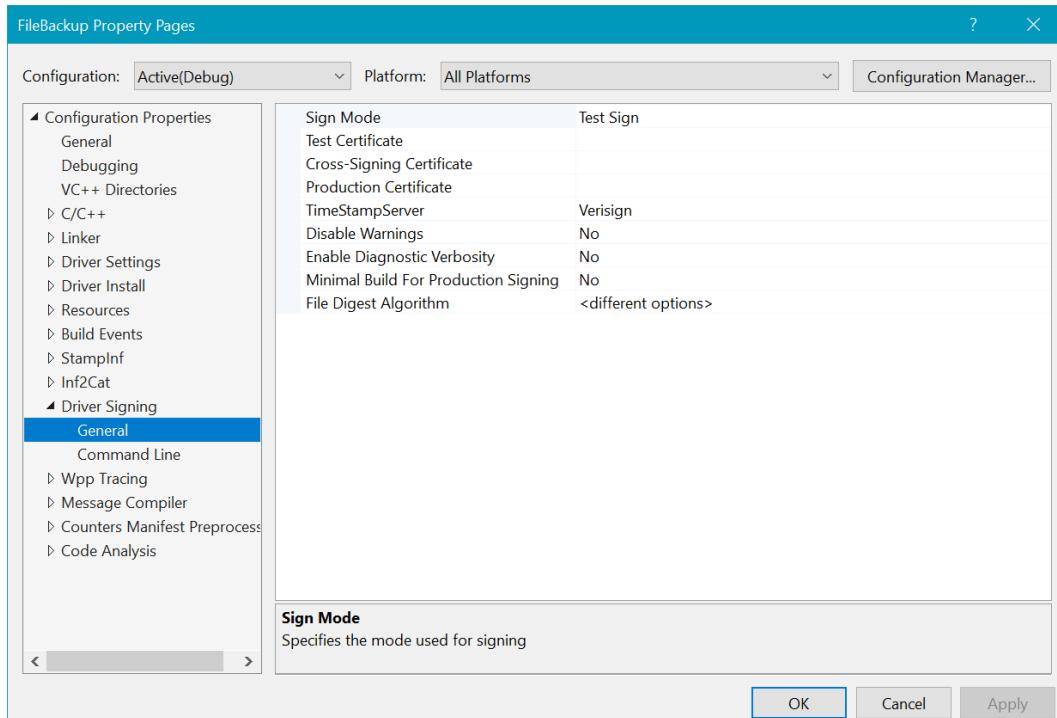


Figure 15-2: Driver signing page in Visual Studio

Visual Studio provides two types of signing: Test sign and production sign. With test signing, a test certificate (a locally-generated certificate that is not trusted globally) is typically used. This allows testing the driver on systems configured with test signing enabled, as we've done throughout this book. Production signing is about using a real certificate to sign the driver for production use.

Test certificates can be generated at will using Visual Studio when selecting a certificate, as shown in Figure 15-3.

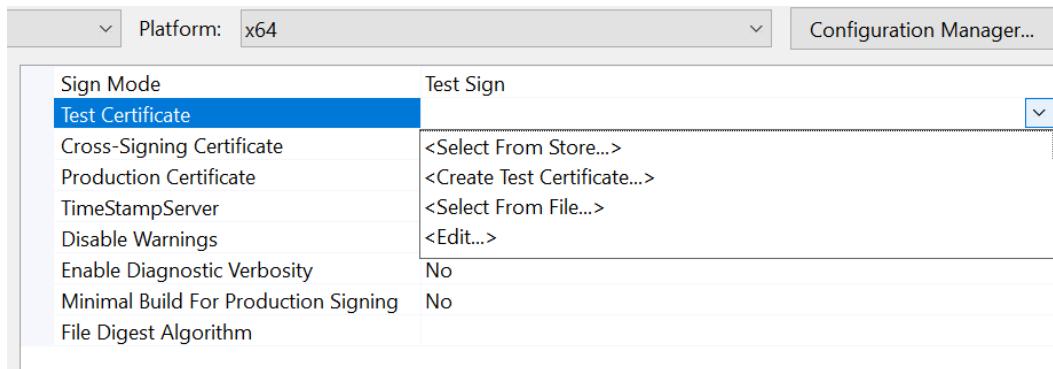


Figure 15-3: Selecting a certificate type in Visual Studio

Figure 15-4 shows an example of production signing a release build of a driver in Visual Studio. Note that the digest algorithm should be *SHA256* rather than the older, less secure, *SHA1*.

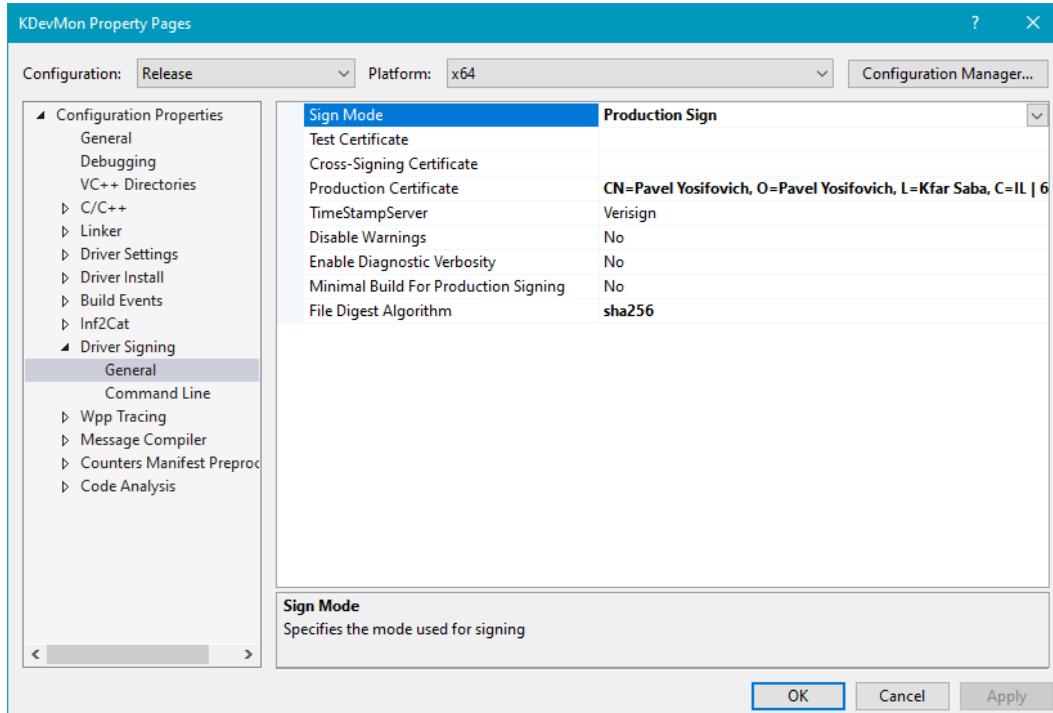


Figure 15-4: Production signing a driver in Visual Studio

Dealing with the various procedures for registering and signing drivers is beyond the scope of this book. Things got more complicated in recent years due to new Microsoft rules and procedures. Consult the official documentation available [here⁴](#).

⁴<https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later>

Driver Verifier

Driver Verifier is a built-in tool that existed in Windows since Windows 2000. Its purpose is to help identify driver bugs and bad coding practices. For example, suppose your driver causes a BSOD in some way, but the driver's code is not on any call stacks in the crash dump file. This typically means that your driver did something which was not fatal at the time, such as writing beyond one of its allocated buffers, where that memory was unfortunately allocated to another driver or the kernel. At that point, there is no crash. However, sometime later that driver or the kernel will use that overflowed data and most likely cause a system crash. There is no easy way to associate the crash with the offending driver. The driver verifier offers an option to allocate memory for the driver in its own "special" pool, where pages at higher and lower addresses are inaccessible, and so will cause an immediate crash upon a buffer overflow or underflow, making it easy to identify the problematic driver.

Driver verifier has a GUI and a command line interface, and can work with any driver - it does not require any source code. The easiest way to start with the verifier is to open it by typing *Verifier* in the *Run* dialog or searching for *Verifier* when clicking the *Start* button. Either way, the verifier presents its initial user interface shown in Figure 15-5.

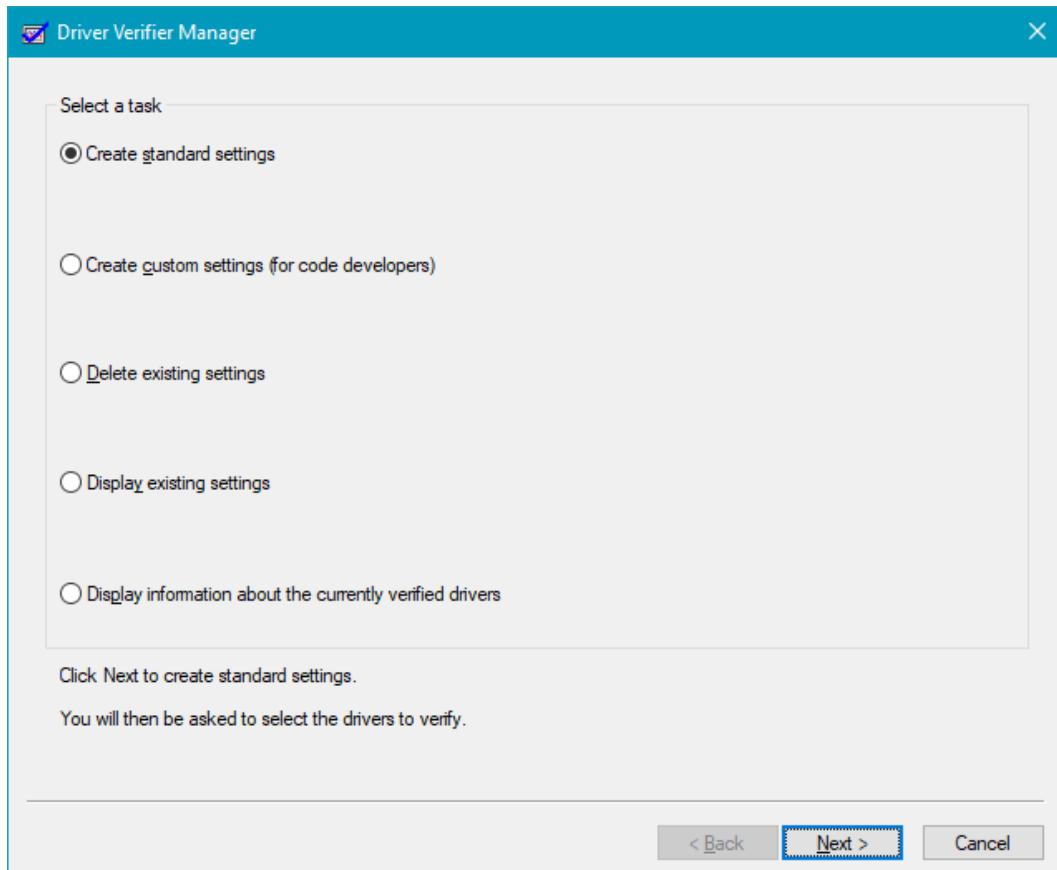


Figure 15-5: Driver Verifier initial window

There are two things that need to be selected: the type of checks to do by the verifier, and the drivers that should be checked. The first page of the wizard is about the checks themselves. The options available on this page are as follows:

- *Create standard settings* selects a predefined set of checks to be performed. We'll see the complete list of available checks in the second page, each with a flag of *Standard* or *Additional*. All those marked *Standard* are selected by this option automatically.
- *Create custom settings* allows fine grained selection of checks by listing all the available checks, shown in Figure 15-6.
- *Delete existing settings* deletes all existing verifier settings.
- *Display existing settings* shows the current configured checks and the drivers for which this checks apply.
- *Display information about the currently verified drivers* shows the collected information for the drivers running under the verifier in an earlier session.

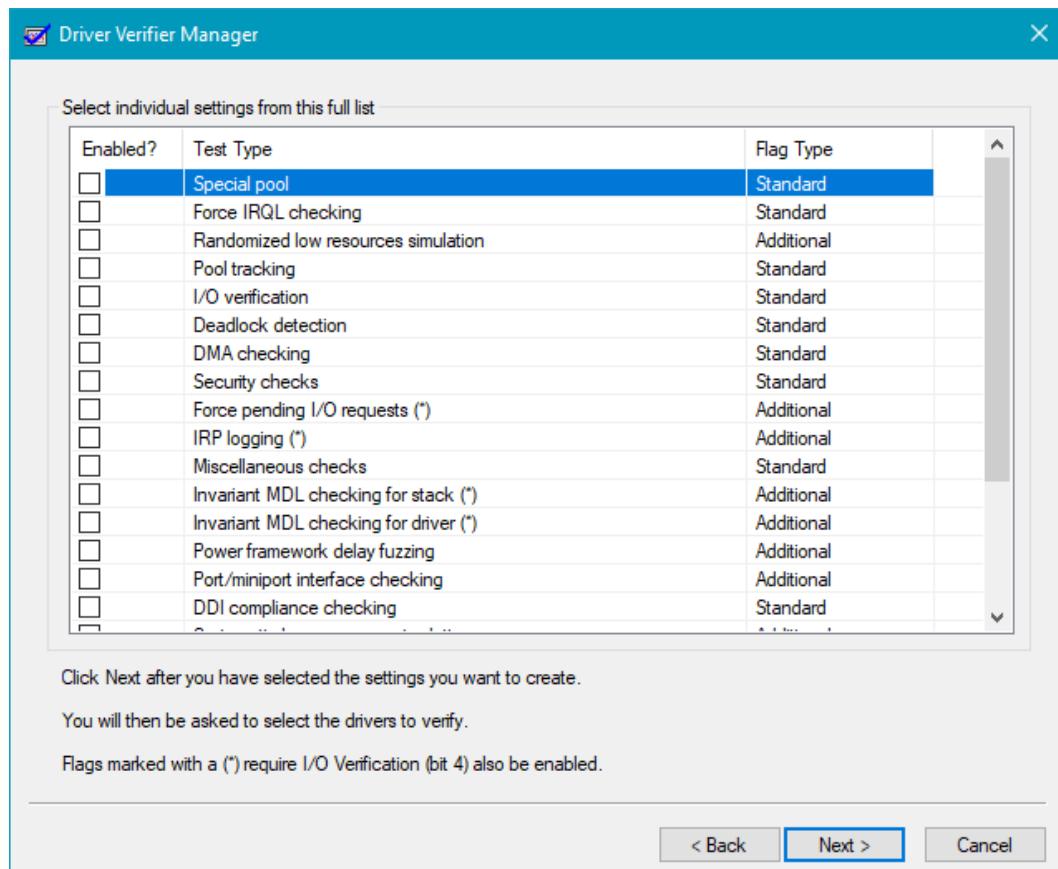


Figure 15-6: Driver Verifier selection of settings

Selecting *Create custom settings* shows the available list of verifier settings, a list that has grown considerably since the early days of Driver Verifier. The flag *Standard* flag indicates this setting is part of the Standard settings that can be selected in the first page of the wizard. Once the settings have been selected, the Verifier shows the next step for selecting the drivers to execute with these settings, shown in Figure 15-7.

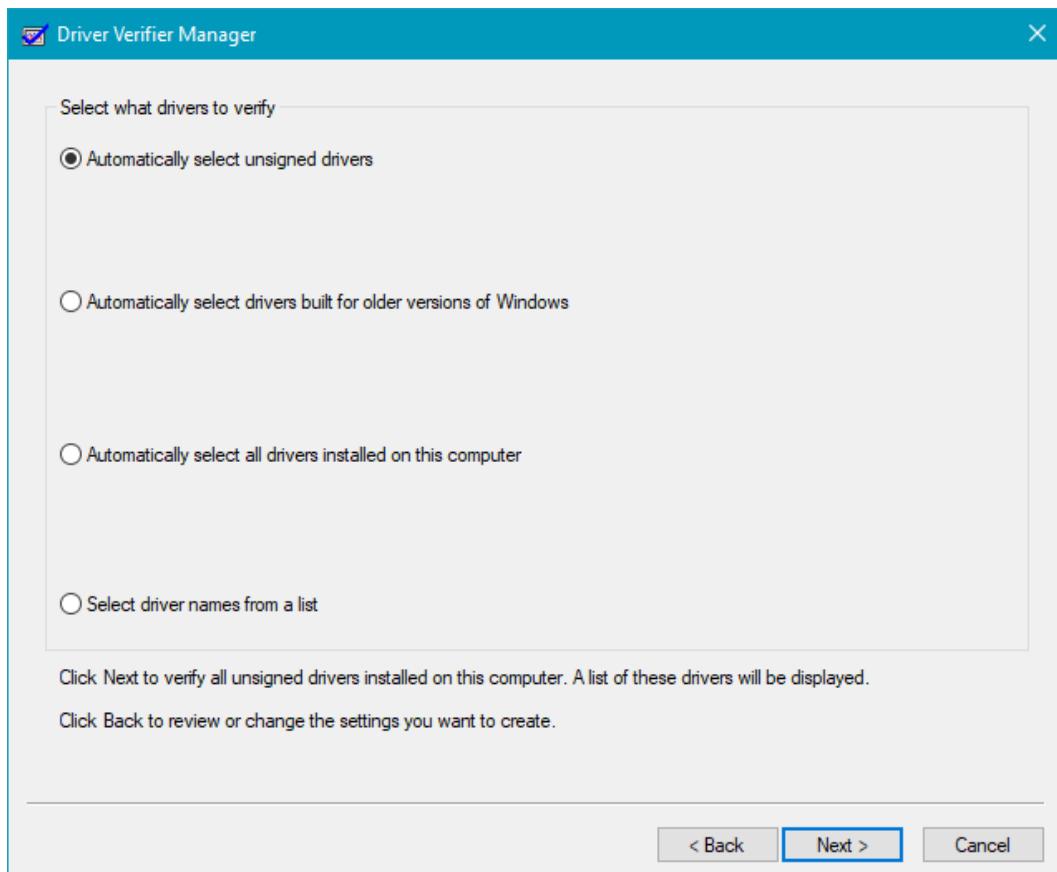


Figure 15-7: Driver Verifier initial driver selection

Here are the possible options:

- *Automatically select unsigned drivers* is mostly relevant for 32 bit systems as 64 bit systems must have signed drivers (unless in test signing mode). Clicking *Next* will list such drivers. Most systems would not have any.
- *Automatically select drivers built for older versions of Windows* is a legacy setting for NT 4 hardware based drivers. Mostly uninteresting for modern systems.
- *Automatically select all drivers installed on the computer* is a catch all option that selects all drivers. This theoretically could be useful if you are presented with a system that crashes but no one has any clue as to offending driver. However, this setting is not recommended, as it slows down the machine (verifier has its costs), because verifier intercepts various operations (based on the previous settings) and typically causes more memory to be used. So it's better in such a scenario to select the first (say) 15 drivers, see if the verifier catches the bad driver, and if not select the next 15 drivers, and so on.
- Select driver names from a list* is the best option to use, where Verifier presents a list of drivers currently executing on the system, as shown in Figure 15-8. If the driver in question is not

currently running, clicking *Add currently not loaded driver(s) to the list...* allows navigating to the relevant SYS file(s).

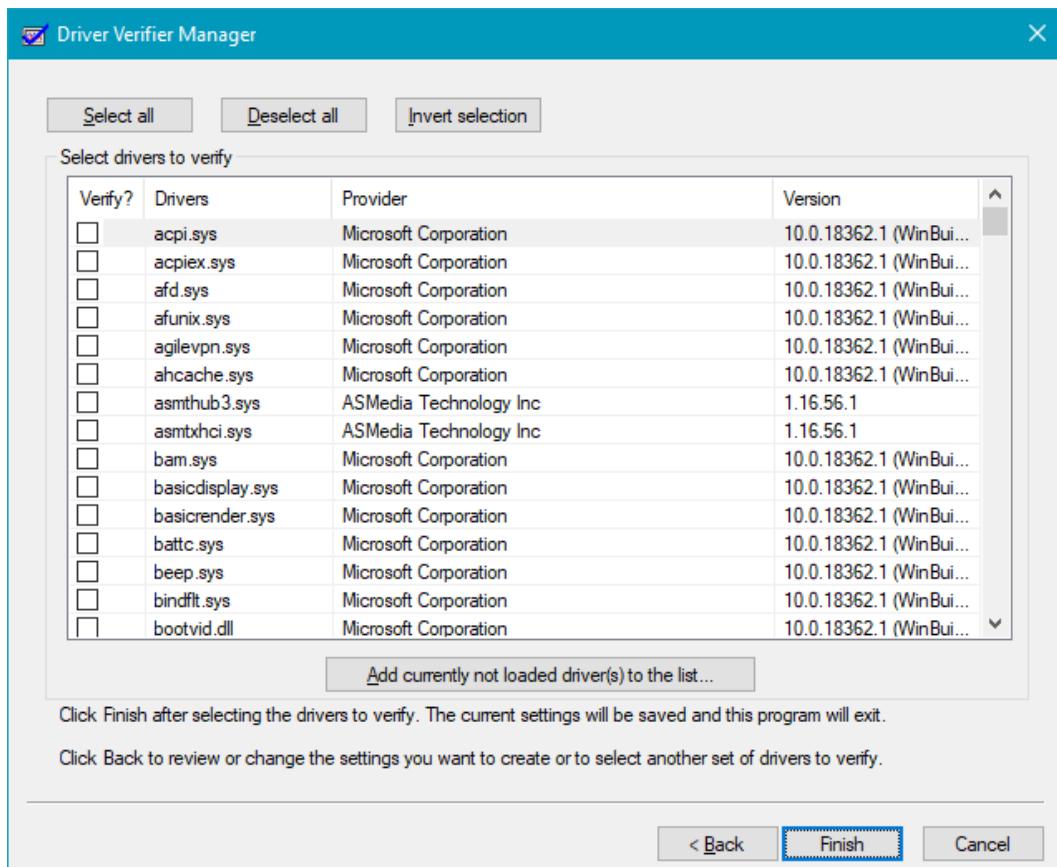


Figure 15-8: Driver Verifier specific driver selection

Finally, clicking *Finish* changes the settings permanent until revoked, and the system typically needs to be restarted so that verifier can initialize itself and hook drivers, especially if these are currently executing.

Example Driver Verifier Sessions

Let's start with a simple example involving the *NotMyFault* tool from *Sysinternals*. As discussed in chapter 6, this tool can be used to crash the system in various ways. Figure 15-9 shows *NotMyFault* main UI. Some of the options to crash the system will do so immediately, with the driver *MyFault.sys* appearing on the call stack of the crashing thread. This is an easy crash to diagnose. However, the option *Buffer overflow* may or may not crash the system immediately. If the system crashes somewhat later, than it's unlikely to find *MyFault.sys* on the call stack.



Make sure you run *NotMyFault64.exe* on a 64-bit system.

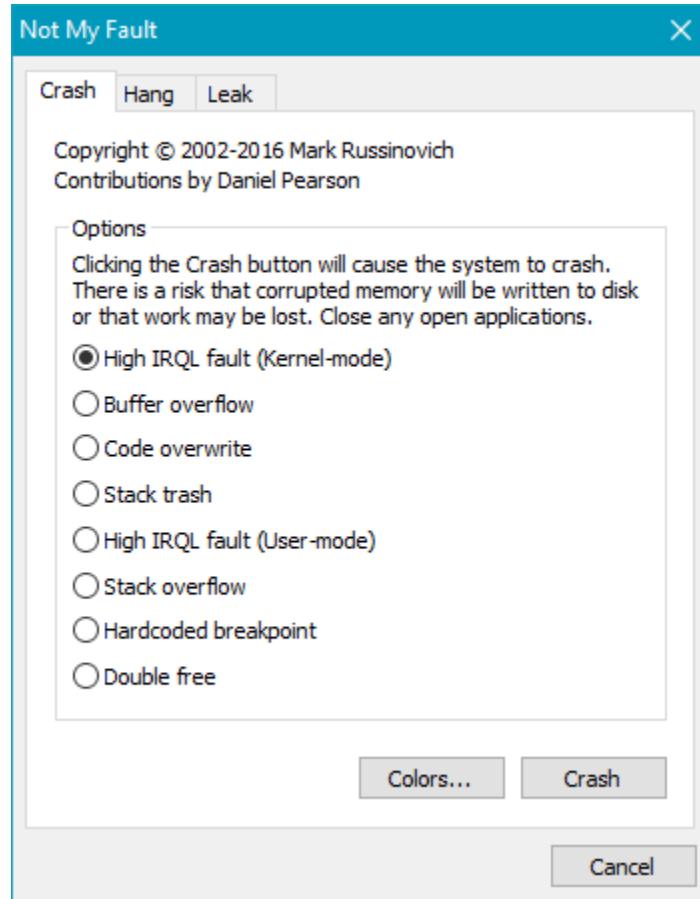


Figure 15-9: NotMyFault main UI

Let's try this (in a virtual machine). It may take several clicks on *Crash* to actually crash the system. Figure 15-10 shows the result on a Windows 7 VM after some clicks on *Crash* and several seconds passing by. Note the BSOD code (BAD_POOL_HEADER). A good guess would be the buffer overflow wrote over some of the metadata of a pool allocation.

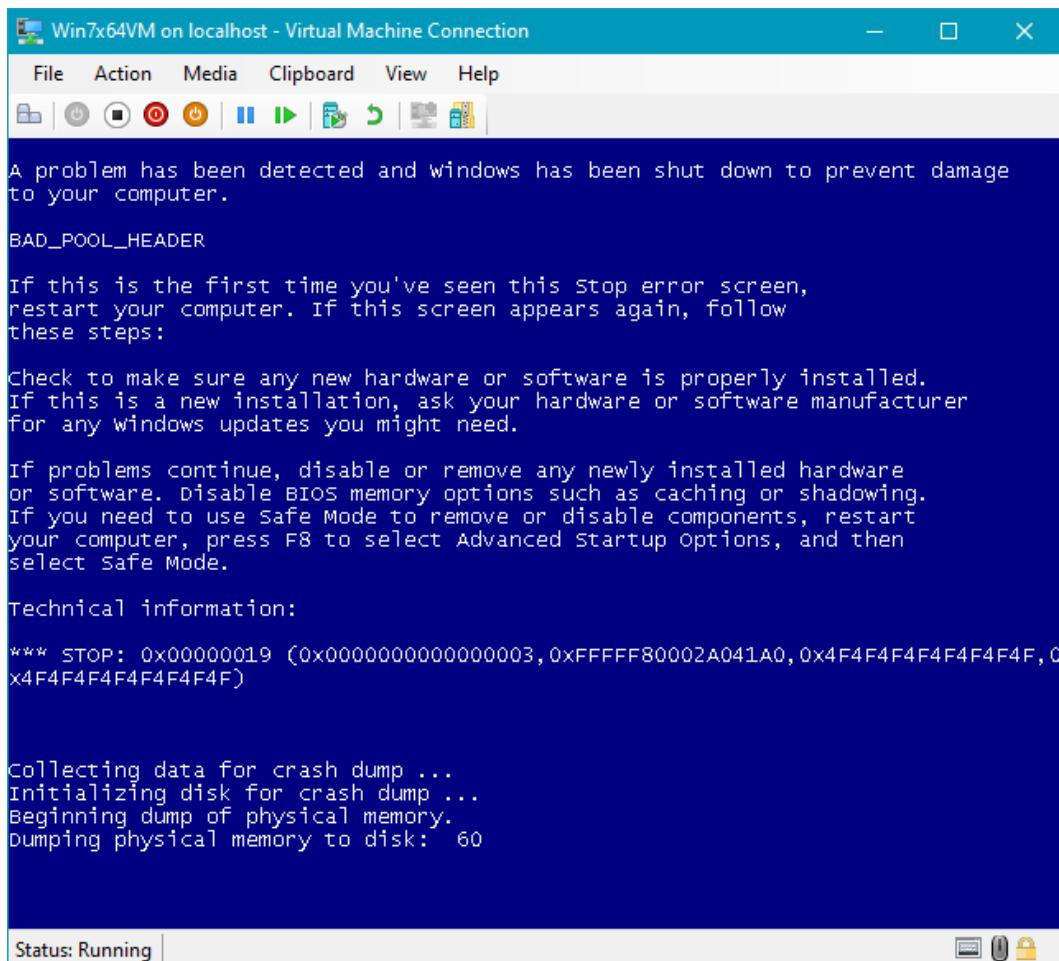


Figure 15-10: NotMyFault causing BSOD on Windows 7 with Buffer overflow

Loading the resulting dump file and looking at the call stack shows this:

```
1: kd> k
# Child-SP          RetAddr         Call Site
00 fffff880`054be828 fffff800`029e4263 nt!KeBugCheckEx
01 fffff880`054be830 fffff800`02bd969f nt!ExFreePoolWithTag+0x1023
02 fffff880`054be920 fffff800`02b0669b nt!ObpAllocateObject+0x12f
03 fffff880`054be990 fffff800`02c2f012 nt!ObCreateObject+0xdb
04 fffff880`054bea00 fffff800`02b1a7b2 nt!PspAllocateThread+0x1b2
05 fffff880`054bec20 fffff800`02b20d95 nt!PspCreateThread+0x1d2
06 fffff880`054beea0 fffff800`028aaad3 nt!NtCreateThreadEx+0x25d
07 fffff880`054bf5f0 fffff800`028a02b0 nt!KiSystemServiceCopyEnd+0x13
08 fffff880`054bf7f8 fffff800`02b29a60 nt!KiServiceLinkage
```

```
09 fffff880`054bf800 fffff800`0286ac1a nt!RtlpCreateUserThreadEx+0x138
0a fffff880`054bf920 fffff800`0285c1c0 nt!ExpWorkerFactoryCreateThread+0x92
0b fffff880`054bf9e0 fffff800`02857dd0 nt!ExpWorkerFactoryCheckCreate+0x180
0c fffff880`054bfa60 fffff800`028aaad3 nt!NtReleaseWorkerFactoryWorker+0x1a0
0d fffff880`054bfae0 00000000`76e1ac3a nt!KiSystemServiceCopyEnd+0x13
```

Clearly, *MyFault.sys* is nowhere to be found. *analyze -v*, by the way is no wiser and concludes that the module **nt** is the culprit.

Now let's try the same experiment with Driver Verifier. Choose standard settings and navigate to the *System32\Drivers* to locate *MyFault.sys* (if it's not currently running). Restart the system, run *NotMyFault* again, select *Buffer overflow* and click *Crash*. You will notice that the system crashes immediately, with a BSOD similar to the one shown in Figure 15-11.

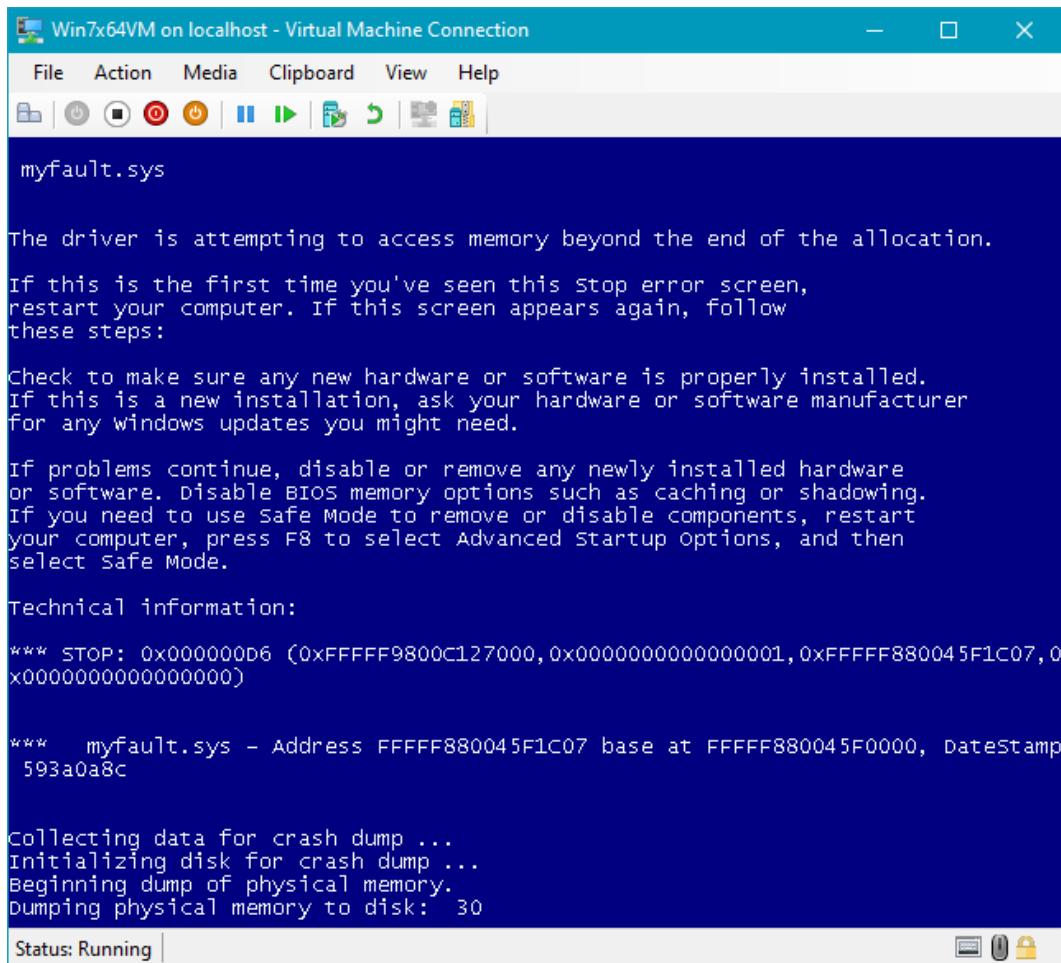


Figure 15-11: NotMyFault BSOD on Windows 7 with Buffer overflow and Verifier active

The BSOD itself is immediately telling. The dump file confirms it with the following call stack:

```
0: kd> k
# Child-SP          RetAddr           Call Site
00 ffffff880`0651c378 ffffff800`029ba462 nt!KeBugCheckEx
01 ffffff880`0651c380 ffffff800`028ecb96 nt!MmAccessFault+0x2322
02 ffffff880`0651c4d0 ffffff880`045f1c07 nt!KiPageFault+0x356
03 ffffff880`0651c660 ffffff880`045f1f88 myfault+0x1c07
04 ffffff880`0651c7b0 ffffff800`02d63d56 myfault+0x1f88
05 ffffff880`0651c7f0 ffffff800`02b43c7a nt!IovCallDriver+0x566
06 ffffff880`0651c850 ffffff800`02d06eb1 nt!IopSynchronousServiceTail+0xfa
07 ffffff880`0651c8c0 ffffff800`02b98296 nt!IopXxxControlFile+0xc51
08 ffffff880`0651ca00 ffffff800`028eead3 nt!NtDeviceIoControlFile+0x56
09 ffffff880`0651ca70 00000000`777e98fa nt!KiSystemServiceCopyEnd+0x13
```

We have no symbols for *MyFault.sys*, but clearly it's the culprit.

Filter Drivers

The Windows driver model is device-centric as we've seen already in chapter 7. Devices can be layered on top of each other, resulting in the highest layer device getting first crack at an incoming IRP. This same model is used for file system drivers, which we leveraged in chapter 12 with the help of the Filter Manager, which is specialized for file system filters. However, the filtering model is generic and can be utilized for other types of devices. In this section we'll take a closer look at the general model of device filtering, which we'll be able to apply to a broad range of devices, some of which are related to hardware devices while others are not.

The kernel API provides several functions that allow one device to be layered on top of another device. The simplest is probably *IoAttachDevice* which accepts a device object to attach and a target named device object to attach to. Here is its prototype:

```
NTSTATUS IoAttachDevice (
    PDEVICE_OBJECT SourceDevice,
    _In_  PUNICODE_STRING TargetDevice,
    _Out_ PDEVICE_OBJECT *AttachedDevice);
```

The output of the function (besides the status) is another device object to which the *SourceDevice* was actually attached to. This is required since attaching to a named device which is not at the top of its device stack succeeds, but the source device is actually attached on top of the topmost device, which may be another filter. It's important, therefore, to get the real device that the source device attached itself to, as that device should be the target of requests if the driver wishes to propagate them down the device stack. This is illustrated in Figure 15-12.

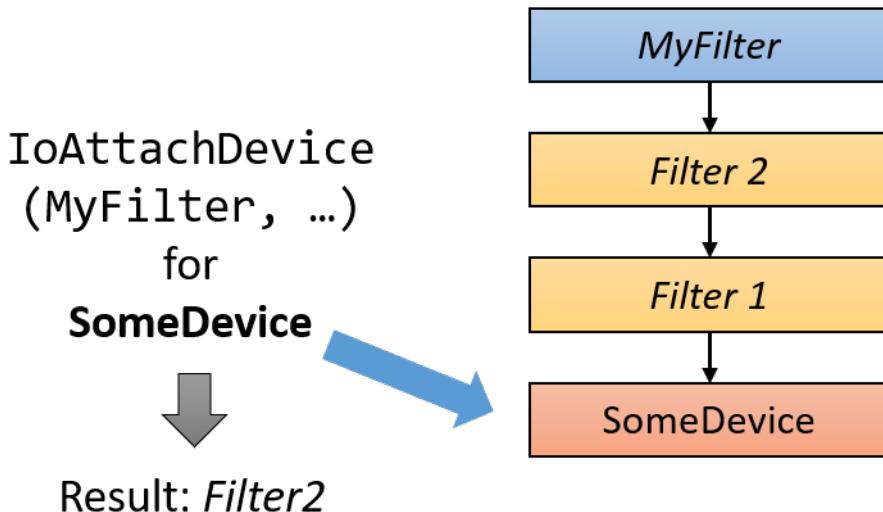


Figure 15-12: Attaching to a named device

Unfortunately, attaching to a device object requires some more work. As discussed in chapter 7, a device can ask the I/O manager to help with accessing a user's buffer with Buffered I/O or Direct I/O (for `IRP_MJ_READ` and `IRP_MJ_WRITE` requests) by setting the appropriate flags in the `Flags` member of the `DEVICE_OBJECT`. In a layering scenario there are multiple devices, so which device is the one that determines how the I/O manager should help with I/O buffers? It turns out it's always the topmost device. This means that our new filter device should copy the value of `DO_BUFFERED_IO` and `DO_DIRECT_IO` flags from the device it actually layered on top of. The default for a device just created with `IoCreateDevice` has neither of these flags set, so if the new device fails to copy these bits, it most likely will cause the target device to malfunction and even crash, as it would not expect its selected buffering method not being respected.

There are a few other settings that need to be copied from the attached device to make sure the new filter looks the same to the I/O system. We'll see these settings later when we build a complete example of a filter.

What is this device name that `IoAttachDevice` requires? This is a named device object within the Object Manager's namespace, viewable with the `WinObj` tools we've used before. Most of the named device objects are located in the `\Device\` directory, but some are located elsewhere. For example, if we were to attach a filter device object to *Process Explorer*'s device object, the name would be `\Device\ProcExp152` (the name is case insensitive).

Other functions for attaching to another device object include `IoAttachDeviceToDeviceStack` and `IoAttachDeviceToDeviceStackSafe`, both accepting another device object to attach to rather than a name of a device. These functions are mostly useful when building filters registered for hardware-based device drivers, where the target device object is provided as part of device node building (partially described in chapter 7 as well). Both return the actual layered device object, just as `IoAttachDevice` does. The `Safe` function returns a proper NTSTATUS, while the former returns NULL on failure. Other than that, these functions are identical.

Generally, kernel code can obtain a pointer to a named device object with `IoGetDeviceObjectPointer`

that returns a device object and a file object open for that device based on a device name. Here is the prototype:

```
NTSTATUS IoGetDeviceObjectPointer (
    _In_ PUNICODE_STRING ObjectName,
    _In_ ACCESS_MASK DesiredAccess,
    _Out_ PFILE_OBJECT *FileObject,
    _Out_ PDEVICE_OBJECT *DeviceObject);
```

The desired access is typically FILE_READ_DATA or any other that is valid for file objects. The returned file object's reference is incremented, so the driver needs to be careful to decrement that reference eventually (ObDereferenceObject) so the file object does not leak. The returned device object can be used as an argument to IoAttachDeviceToDeviceStack(Safe).

Filter Driver Implementation

A filter driver needs to attach a device object over a target device for which filtering is required. We'll discuss later when attachment should occur, but for now let's assume the call to one of the "attach" functions is made at some point. Since the new device object will now become the topmost device in the device stack, any request the driver does not support will bounce back to the client with an "unsupported operation" error. This means that the filter's DriverEntry must register for all major function codes if it wants to make sure the underlying device object continues to function normally. Here is one way to set this up:

```
for (int i = 0; i < ARRSIZE(DriverObject->MajorFunction); i++)
    DriverObject->MajorFunction[i] = HandleFilterFunction;
```

The above code snippet sets all major function codes pointing to the same function. The HandleFilterFunction function must, at the very least, call the lower layered driver using the device object obtained from one of the "attach" functions. Of course, being a filter, the driver will want to do additional work or different work for requests it's interested in, but all the requests it does not care about must be forwarded to the lower layer device, or else that device will not function properly.

This "forward and forget" operation is very common in filters. Let's see how to implement this functionality. The actual call that transfers an IRP to another device is IoCallDriver. However, before calling it the current driver must prepare the next I/O stack location for the lower driver's use. Remember that initially, the I/O manager only initializes the first I/O stack location. It's up to every layer to initialize the next I/O stack location before using IoCallDriver to pass the IRP down the device stack.

The driver can call IoGetNextIrpStackLocation to get a pointer to the next layer's IO_STACK_LOCATION and go ahead and initialize it. In most cases, however, the driver just wants to present to the lower layer the same information it received itself. One function that can help with that is IoCopyCurrentIrpStackLocationToNext, which is pretty self explanatory. This function, however, does **not** just blindly copy the I/O stack location like so:

```
auto current = IoGetCurrentIrpStackLocation(Irp);  
auto next = IoCopyCurrentIrpStackLocationToNext(Irp);  
*next = *current;
```

Why? The reason is subtle, and has to do with the completion routine. Recall from chapter 7 that a driver can set up a completion routine to be notified once an IRP is completed by a lower layer driver (`IoSetCompletionRoutine/Ex`). The completion pointer (and a driver-defined context argument) are stored in the `next` I/O stack location, and that's why a blind copy would duplicate the higher-level completion routine (if any), which is not what we want. This is exactly what `IoCopyCurrentIrpStackLocationToNext` avoids.

But there is actually a better way if the driver does not need a completion routine and just wants to use “forward and forget”, without paying the price of copying the I/O stack location data. This is accomplished by skipping the I/O stack location in such a way so that the next lower layer driver sees the same I/O stack location as this one:

```
IoSkipCurrentIrpStackLocation(Irp);  
status = IoCallDriver(LowerDeviceObject, Irp);
```

`IoSkipCurrentIrpStackLocation` simply decrements the internal IRP’s I/O stack location’s pointer, and `IoCallDriver` increments it, essentially making the lower driver see the same I/O stack location as this layer, without any copying going on; this is the preferred way of propagating the IRP down if the driver does not wish to make changes to the request and it does not require a completion routine.



Technically, the I/O stack location is incremented by `IoSkipCurrentIrpStackLocation` and decremented back by `IoCallDriver`, as the I/O stack locations are used from the bottom of memory going up.

Attaching Filters

When does a driver call one of the attach functions? The ideal time is when the underlying device (the attach target) is being created; that is, the device node is in the process of being built. This is common in filters for hardware-based device drivers, where filters can be registered in the named values `UpperFilters` and `LowerFilters` we saw in chapter 7. For these filters, the proper location for actually creating the new device object and attaching it to an existing device stack is in a callback set with the `AddDevice` member accessible from the driver object like so:

```
DriverObject->DriverExtension->AddDevice = FilterAddDevice;
```

We’ve briefly discussed that in chapter 14 when looking at driver initialization with KMDF.

This *AddDevice* callback is invoked when a new hardware device belonging to the driver has been identified by the Plug & Play system. This routine has the following prototype:

```
NTSTATUS AddDeviceRoutine (
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PDEVICE_OBJECT PhysicalDeviceObject);
```

The I/O system provides the driver with the device object at the bottom of the device stack (*PhysicalDeviceObject* or PDO) to be used in a call to *IoAttachDeviceToDeviceStack(Safe)*. This PDO is one reason why *DriverEntry* is not a suitable location to make an attach call - at this point the PDO is not yet provided. Furthermore, a second device of the same type may be added into the system (such as a second USB camera), in which case *DriverEntry* is not going to be called at all; only the *AddDevice* routine will.

Here is an example for implementing an *AddDevice* routine for a filter driver (error handling omitted):

```
struct DeviceExtension {
    PDEVICE_OBJECT LowerDeviceObject;
};

NTSTATUS FilterAddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT PDO) {
    PDEVICE_OBJECT DeviceObject;
    auto status = IoCreateDevice(DriverObject, sizeof(DeviceExtension), nullptr,
        FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);

    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
    status = IoAttachDeviceToDeviceStackSafe(
        DeviceObject,           // device to attach
        PDO,                   // target device
        &ext->LowerDeviceObject); // actual device object

    //
    // copy some info from the attached device
    //
    DeviceObject->DeviceType = ext->LowerDeviceObject->DeviceType;
    DeviceObject->Flags |= ext->LowerDeviceObject->Flags &
        (DO_BUFFERED_IO | DO_DIRECT_IO);

    //
    // important for hardware-based devices
    //
```

```

DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
DeviceObject->Flags |= DO_POWER_PAGABLE;

return status;
}

```

A few important points on the code above:

- The device object is created without a name. A name is not needed, because the target device is named and is the real target for IRPs, so no need to provide our own name. The filter is going to be invoked regardless.
- In the `IoCreateDevice` call we specify a non-zero size for the second argument, asking the I/O manager to allocate an extra buffer (`sizeof(DeviceExtension)`) along with the actual `DEVICE_OBJECT`. Up until now we used global variables to manage state for a device because we had just one. However, a filter driver may create multiple device objects and attach to multiple device stacks, making it harder to correlate device objects with some state. The device extension mechanism makes it easy to get to a device-specific state given the device object itself. In the above code we capture the lower device object as our state, but this structure can be extended to include more information as needed.
- We copy some information from the lower device object, so that our filter appears to the I/O system as the target device itself. Specifically, we copy the device type and the buffering method flags. Copying the buffering method flags is critical, as the buffering method is determined by the uppermost device - our filter as it may turn out.
- Finally, we remove the `DO_DEVICE_INITIALIZING` flag (set by the I/O system initially) to indicate to the Plug & Play manager that the device is ready for work. The `DO_POWER_PAGABLE` flag indicates Power IRPs should arrive in `IRQL < DISPATCH_LEVEL`, and is in fact mandatory.

Given the above code, here is a “forward and forget” implementation that uses the lower device as described in the previous section:

```

NTSTATUS FilterGenericDispatch(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(ext->LowerDeviceObject, Irp);
}

```

Attaching Filters at Arbitrary Time

The previous section looked at attaching a filter device in the `AddDevice` callback, called by the plug & Play manager while the device node is being built. For non-hardware based drivers, we may or may not have registry settings to use for filters. In the latter case, no `AddDevice` callback is ever invoked.

For these more general cases, the filter driver can attach filter devices theoretically at any time, by creating a device object (`IoCreateDevice`) and then using one of the “attach” functions. This means

the target device already exists, it's already working, and at some point it gets a filter. The driver must make sure this slight "interruption" does not have any adverse effect on the target device. Most of the operations shown in the previous sections are relevant here as well, such as copying some flags from the lower device. However, some extra care must be taken to make sure the target device's operations are not disrupted.

Using `IoAttachDevice`, the following code creates a device object and attaches it over another named device object (error handling omitted):

```
//  
// use hard-coded name for illustration purposes  
//  
UNICODE_STRING targetName = RTL_CONSTANT_STRING(L"\Device\SomeDeviceName");  
  
PDEVICE_OBJECT DeviceObject;  
auto status = IoCreateDevice(DriverObject, 0, nullptr,  
    FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);  
  
PDEVICE_OBJECT LowerDeviceObject;  
status = IoAttachDevice(DeviceObject, &targetName, &LowerDeviceObject);  
  
//  
// copy information  
//  
DeviceObject->Flags |= LowerDeviceObject->Flags &  
    (DO_BUFFERED_IO | DO_DIRECT_IO);  
  
DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;  
DeviceObject->Flags |= DO_POWER_PAGABLE;  
DeviceObject->DeviceType = LowerDeviceObject->DeviceType;
```

Astute readers may notice that the above code has an inherent race condition. Can you spot it?

This is essentially the same code used in the `AddDevice` callback in the previous section. But in that code there was no race condition. This is because the target device was not yet active - the device node was being built, device by device, from the bottom to the top. The device was not yet in a position to receive requests.

Contrast that with the above code - the target device is working and could be very busy, when suddenly a filter appears. The I/O system makes sure there is no issue while performing the actual attach operation, but once the call to `IoAttachDevice` returns (and in fact even before that), requests continue to come in. Suppose that a read operation comes in just after `IoAttachDevice` returns but before the buffering method flags are set - the I/O manager will see the flags as zero (neither I/O) since it only looks at the topmost device, which is now our filter! So if the target device uses Direct I/O (for example), the I/O manager will not lock the user's buffer, will not create an MDL, etc. This could lead to a system crash if the target driver always assumes that `Irp->MdlAddress` (for example) is non-NULL.

The window of opportunity for failure is very small, but it's better to play it safe.

How can we solve this race condition? We must prepare our new device object fully before actually attaching. We can do that by calling `IoGetDeviceObjectPointer` to get the target device object, copy the required information to our own device (at this time still not attached), and only then call `IoAttachDeviceToDeviceStack(Safe)`. We'll see a complete example later in this chapter.



Write the appropriate code to use `IoGetDeviceObjectPointer` as described above.

Filter Cleanup

Once a filter is attached, it must be detached at some point. Calling `IoDetachDevice` with the lower device object pointer performs this operation. Notice the lower device object is the argument, **not** the filter's own device object. Finally, `IoDeleteDevice` for the filter's device object should be called, just as we did in all our drivers so far.

The question is when should this cleanup code be called? if the driver is unloaded explicitly, then the normal unload routine should perform these cleanup operations. However, some complication arises in filters for hardware-based drivers. These drivers may need to unload because of a Plug & Play event, such as a user yanking out a device out of the system. A hardware based driver receives an `IRP_MJ_PNP` request with a minor `IRP_MN_REMOVE_DEVICE` indicating the hardware itself is gone, so the entire device node is not needed and it will be torn down. It's the responsibility of the driver to handle this PnP request properly, detach from the device node and delete the device.

This means that for hardware-based filters, a simple "forward and forget" for `IRP_MJ_PNP` will not suffice. Special treatment is needed for `IRP_MN_REMOVE_DEVICE`. Here is some example code:

```
NTSTATUS FilterDispatchPnp(PDEVICE_OBJECT fido, PIRP Irp) {
    auto ext = (DeviceExtension*)fido->DeviceExtension;
    auto stack = IoGetCurrentIrpStackLocation(Irp);

    UCHAR minor = stack->MinorFunction;
    IoSkipCurrentIrpStackLocation(Irp);
    auto status = IoCallDriver(ext->LowerDeviceObject, Irp);
    if (minor == IRP_MN_REMOVE_DEVICE) {
        IoDetachDevice(LowerDeviceObject);
        IoDeleteDevice(fido);
    }
    return status;
}
```

More on Hardware-Based Filter Drivers

Filters for hardware-based driver have some further complications. The `FilterDispatchPnp` function shown in the previous section has a race condition in it. The problem is that while some IRP is being handled, a remove device request might come in (handled on another CPU, for instance). This will cause `IoDeleteDevice` calls in drivers that are part of the device node while a filter is preparing to send the other request down the device stack. A more detailed explanation of this race condition is beyond the scope of this book, but regardless, we need an air-tight solution.

The solution is an object provided by the I/O system called a remove lock, represented by the `IO_REMOVE_LOCK` structure. Essentially, this structure manages a reference count of the number of outstanding IRPs currently being handled and an event that is signaled when the I/O count is zero and a remove operation is in progress. Using an `IO_REMOVE_LOCK` can be summarized as follows:

1. The driver allocates the structure as part of a device extension or a global variable and initializes it once with `IoInitializeRemoveLock`.
2. For every IRP, the driver acquires the remove lock with `IoAcquireRemoveLock` before passing it down to a lower device. If the call fails (`STATUS_DELETE_PENDING`) it means a remove operation is in progress and the driver should return immediately.
3. Once a lower driver is done with the IRP, release the remove lock (`IoReleaseRemoveLock`).
4. When handling `IRP_MN_REMOVE_DEVICE` call `IoReleaseRemoveLockAndWait` before detaching and deleting the device. The call will succeed once all other IRPs are no longer being processed.

With these steps in mind, the generic dispatch passing requests down must be changed as follows (assuming the remove lock was already initialized):

```
struct DeviceExtension {
    IO_REMOVE_LOCK RemoveLock;
    PDEVICE_OBJECT LowerDeviceObject;
};

NTSTATUS FilterGenericDispatch(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

    //
    // second argument is unused in release builds of Windows
    //
    auto status = IoAcquireRemoveLock(&ext->RemoveLock, Irp);
    if(!NT_SUCCESS(status)) {    // STATUS_DELETE_PENDING
        Irp->IoStatus.Status = status;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
        return status;
    }
    IoSkipCurrentIrpStackLocation(Irp);
```

```

status = IoCallDriver(ext->LowerDeviceObject, Irp);

IoReleaseRemoveLock(&ext->RemoveLock, Irp);
return status;
}

```

The IRP_MJ_PNP handler must be modified to use the remove lock properly:

```

NTSTATUS FilterDispatchPnp(PDEVICE_OBJECT fido, PIRP Irp) {
    auto ext = (DeviceExtension*)fido->DeviceExtension;
    auto status = IoAcquireRemoveLock(&ext->RemoveLock, Irp);
    if (!NT_SUCCESS(status)) { // STATUS_DELETE_PENDING
        Irp->IoStatus.Status = status;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
        return status;
    }

    auto stack = IoGetCurrentIrpStackLocation(Irp);
    UCHAR minor = stack->MinorFunction;

    IoSkipCurrentIrpStackLocation(Irp);
    auto status = IoCallDriver(ext->LowerDeviceObject, Irp);
    ifi 2% (minor == IRP_MN_REMOVE_DEVICE) {
        // wait if needed
        IoReleaseRemoveLockAndWait(&ext->RemoveLock, Irp);

        IoDetachDevice(ext->LowerDeviceObject);
        IoDeleteDevice(fido);
    }
    else {
        IoReleaseRemoveLock(&ext->RemoveLock, Irp);
    }
    return status;
}

```

Device Monitor

With the information presented thus far it is possible to build a generic driver that can attach to device objects as filters to other devices. This allows for intercepting requests to (almost) any device we're interested in. A companion user-mode client will allow adding and removing devices to filter.

We'll create a new *Empty WDM driver* project named *KDevMon* as we've done numerous times. The driver should be able to attach to multiple devices, and on top of that expose its own *Control*

Device Object (CDO) to handle user-mode client configuration requests. The CDO will be created in `DriverEntry` as usual, but attachments will be managed separately, controlled by requests from a user-mode client.

To manage all the devices currently being filtered, we'll create a helper class named `DevMonManager`. Its primary purpose is to add and remove devices to filter. Each device will be represented by the following structure:

```
struct MonitoredDevice {
    UNICODE_STRING DeviceName;
    PDEVICE_OBJECT DeviceObject;
    PDEVICE_OBJECT LowerDeviceObject;
};
```

For each device, we need to keep the filter device object (the one created by this driver), the lower device object to which it's attached and the device name. The name will be needed for detach purposes. The `DevMonManager` class holds a fixed array of `MonitoredDevice` structures, a fast mutex to protect the array and some helper functions. Here are the main ingredients in `DevMonManager`:

```
const int MaxMonitoredDevices = 32;

class DevMonManager {
public:
    void Init(PDRIVER_OBJECT DriverObject);
    NTSTATUS AddDevice(PCWSTR name);
    int FindDevice(PCWSTR name);
    bool RemoveDevice(PCWSTR name);
    void RemoveAllDevices();
    MonitoredDevice& GetDevice(int index);

    PDEVICE_OBJECT CDO;

private:
    bool RemoveDevice(int index);

private:
    MonitoredDevice Devices[MaxMonitoredDevices];
    int MonitoredDeviceCount;
    FastMutex Lock;
    PDRIVER_OBJECT DriverObject;
};
```

Adding a Device to Filter

The most interesting function is `DevMonManager::AddDevice` which does the attaching. Let's take it step by step.

```
NTSTATUS DevMonManager::AddDevice(PCWSTR name) {
```

First, we have to acquire the mutex in case more than one add/remove/find operation is taking place at the same time. Next, we can make some quick checks to see if all our array slots are taken and that the device in question is not already being filtered:

```
Locker locker(Lock);
if (MonitoredDeviceCount == MaxMonitoredDevices)
    return STATUS_BUFFER_TOO_SMALL;

if (FindDevice(name) >= 0)
    return STATUS_SUCCESS;
```

Now it's time to look for a free array index where we can store information on the new filter being created:

```
for (int i = 0; i < MaxMonitoredDevices; i++) {
    if (Devices[i].DeviceObject != nullptr)
        continue;
```

A free slot is indicated by a NULL device object pointer inside the `MonitoredDevice` structure. Next, we'll try and get a pointer to the device object that we wish to filter with `IoGetDeviceObjectPointer`:

```
UNICODE_STRING targetName;
RtlInitUnicodeString(&targetName, name);

PFILE_OBJECT FileObject;
PDEVICE_OBJECT LowerDeviceObject = nullptr;
auto status = IoGetDeviceObjectPointer(&targetName, FILE_READ_DATA,
    &FileObject, &LowerDeviceObject);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to get device object pointer (%ws) (0x%8X)\n",
        name, status));
    return status;
}
```

The result of `IoGetDeviceObjectPointer` is in fact the topmost device object, which is not necessarily the device object we were targeting. This is fine, since any attach operation will actually

attach to the top of the device stack anyway. The function can fail, of course, most likely because a device with that specific name does not exist.

The next step is to create the new filter device object and initialize it, partly based on the device object pointer we just acquired. At the same time, we need to fill the `MonitoredDevice` structure with the proper data. For each created device we want to have a device extension that stores the lower device object, so we can get to it easily at IRP handling time. For this, we define a device extension structure called simply `DeviceExtension` that can hold this pointer (in the `DevMonManager.h` file):

```
struct DeviceExtension {
    PDEVICE_OBJECT LowerDeviceObject;
};
```

Back to `DevMonManager::AddDevice` - let's create the filter device object:

```
PDEVICE_OBJECT DeviceObject = nullptr;
WCHAR* buffer = nullptr;

do {
    status = IoCreateDevice(DriverObject, sizeof(DeviceExtension), nullptr,
                           FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status))
        break;
```

`IoCreateDevice` is called with the size of the device extension to be allocated in addition to the `DEVICE_OBJECT` structure itself. The device extension is stored in the `DeviceExtension` field in the `DEVICE_OBJECT`, so it's always available when needed. Figure 15-13 shows the effect of calling `IoCreateDevice`.

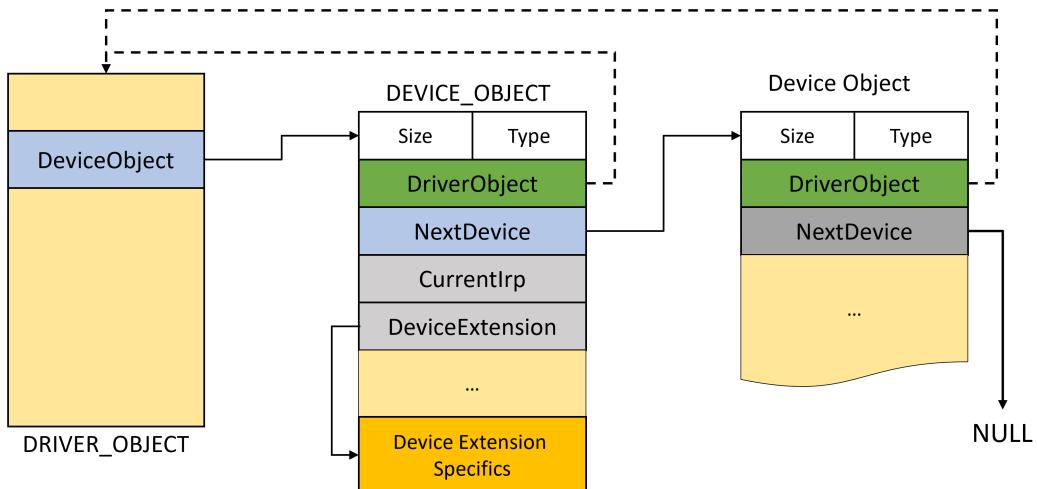


Figure 15-13: The effects of `IoCreateDevice`

Now we can continue with device initialization and the `MonitoredDevice` structure:

```
//  
// allocate buffer to copy device name  
//  
buffer = (WCHAR*)ExAllocatePool2(POOL_FLAG_PAGED, targetName.Length,  
    DRIVER_TAG);  
if (!buffer) {  
    status = STATUS_INSUFFICIENT_RESOURCES;  
    break;  
}  
  
auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;  
  
DeviceObject->Flags |= LowerDeviceObject->Flags &  
    (DO_BUFFERED_IO | DO_DIRECT_IO);  
DeviceObject->DeviceType = LowerDeviceObject->DeviceType;  
  
Devices[i].DeviceName.Buffer = buffer;  
Devices[i].DeviceName.MaximumLength = targetName.Length;  
RtlCopyUnicodeString(&Devices[i].DeviceName, &targetName);  
Devices[i].DeviceObject = DeviceObject;
```

Technically, we could have used LowerDeviceObject->DeviceType instead of FILE_DEVICE_UNKNOWN in the call to IoCreateDevice and save the trouble of copying the DeviceType field explicitly.

At this point the new device object is ready, all that's left is to attach it and finish some more initializations:

```
status = IoAttachDeviceToDeviceStackSafe(  
    DeviceObject, // filter device object  
    LowerDeviceObject, // target device object  
    &ext->LowerDeviceObject); // result  
if (!NT_SUCCESS(status))  
    break;  
  
Devices[i].LowerDeviceObject = ext->LowerDeviceObject;  
//  
// hardware based devices require this  
//  
DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
```

```

DeviceObject->Flags |= DO_POWER_PAGABLE;

MonitoredDeviceCount++;
} while (false);
}

```

The device is attached, with the resulting pointer saved immediately to the device extension. This is important, as the process of attaching itself generates at least two IRPs - IRP_MJ_CREATE and IRP_MJ_CLEANUP and so the driver must be prepared to handle these. As we shall soon see, this handling requires the lower device object to be available in the device extension.

All that's left now is to clean up:

```

if (!NT_SUCCESS(status)) {
    if (buffer)
        ExFreePool(buffer);
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    Devices[i].DeviceObject = nullptr;
}

if (LowerDeviceObject) {
    // dereference - not needed anymore
    ObDereferenceObject(FileObject);
}

return status;
}
}

```

Dereferencing the file object is important; it was obtained by `IoGetDeviceObjectPointer`. Failure to do so is a kernel leak. Note that we do not need (in fact we must not) dereference the device object returned from `IoGetDeviceObjectPointer` - it will be dereferenced automatically when the file object's reference drops to zero.

Here is the full `AddDevice` method for easy reference:

```

NTSTATUS DevMonManager::AddDevice(PCWSTR name) {
    Locker locker(Lock);
    if (MonitoredDeviceCount == MaxMonitoredDevices)
        return STATUS_BUFFER_TOO_SMALL;

    if (FindDevice(name) >= 0)
        return STATUS_SUCCESS;

    for (int i = 0; i < MaxMonitoredDevices; i++) {
        if (Devices[i].DeviceObject != nullptr)

```

```
    continue;
UNICODE_STRING targetName;
RtlInitUnicodeString(&targetName, name);

PFILE_OBJECT FileObject;
PDEVICE_OBJECT LowerDeviceObject = nullptr;
auto status = IoGetDeviceObjectPointer(&targetName, FILE_READ_DATA,
    &FileObject, &LowerDeviceObject);
if (!NT_SUCCESS(status)) {
    KdPrint(("Failed to get device object pointer (%ws) (0x%8X)\n",
        name, status));
    return status;
}

PDEVICE_OBJECT DeviceObject = nullptr;
WCHAR* buffer = nullptr;

do {
    status = IoCreateDevice(DriverObject, sizeof(DeviceExtension),
        nullptr, FILE_DEVICE_UNKNOWN, 0, FALSE, &DeviceObject);
    if (!NT_SUCCESS(status))
        break;

    //
    // allocate buffer to copy device name
    //
    buffer = (WCHAR*)ExAllocatePool2(PPOOL_FLAG_PAGED,
        targetName.Length, DRIVER_TAG);
    if (!buffer) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        break;
    }

    auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;

    DeviceObject->Flags |= LowerDeviceObject->Flags &
        (DO_BUFFERED_IO | DO_DIRECT_IO);

    DeviceObject->DeviceType = LowerDeviceObject->DeviceType;
    Devices[i].DeviceName.Buffer = buffer;
    Devices[i].DeviceName.MaximumLength = targetName.Length;
    RtlCopyUnicodeString(&Devices[i].DeviceName, &targetName);
    Devices[i].DeviceObject = DeviceObject;
}
```

```
status = IoAttachDeviceToDeviceStackSafe(
    DeviceObject,           // filter device object
    LowerDeviceObject,      // target device object
    &ext->LowerDeviceObject); // result
    if (!NT_SUCCESS(status))
        break;

Devices[i].LowerDeviceObject = ext->LowerDeviceObject;
// hardware based devices require this
DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
DeviceObject->Flags |= DO_POWER_PAGABLE;

MonitoredDeviceCount++;
} while (false);

if (!NT_SUCCESS(status)) {
    if (buffer)
        ExFreePool(buffer);
    if (DeviceObject)
        IoDeleteDevice(DeviceObject);
    Devices[i].DeviceObject = nullptr;
}
if (LowerDeviceObject) {
    // dereference - not needed anymore
    ObDereferenceObject(FileObject);
}
return status;
}
// should never get here
NT_ASSERT(false);
return STATUS_UNSUCCESSFUL;
}
```

Removing a Filter Device

Removing a device from filtering is fairly straightforward - reversing what AddDevice did:

```
bool DevMonManager::RemoveDevice(PCWSTR name) {
    Locker locker(Lock);
    int index = FindDevice(name);
    if (index < 0)
        return false;

    return RemoveDevice(index);
}

bool DevMonManager::RemoveDevice(int index) {
    auto& device = Devices[index];
    if (device.DeviceObject == nullptr)
        return false;

    ExFreePool(device.DeviceName.Buffer);
    IoDetachDevice(device.LowerDeviceObject);
    IoDeleteDevice(device.DeviceObject);
    device.DeviceObject = nullptr;

    MonitoredDeviceCount--;
    return true;
}
```

The important parts are detaching the device and deleting it. `FindDevice` is a simple helper to locate a device by name in the array. It returns the index of the device in the array, or -1 if the device is not found:

```
int DevMonManager::FindDevice(PCWSTR name) {
    UNICODE_STRING uname;
    RtlInitUnicodeString(&uname, name);
    for (int i = 0; i < MaxMonitoredDevices; i++) {
        auto& device = Devices[i];
        if (device.DeviceObject &&
            RtlEqualUnicodeString(&device.DeviceName, &uname, TRUE)) {
            return i;
        }
    }
    return -1;
}
```

The only trick here is to make sure the fast mutex is acquired before calling this function.

Initialization and Unload

The `DriverEntry` routine is fairly standard, creating a CDO that would allow adding and removing filters. There are some differences, however. Most notably, the driver must support all major function codes, as the driver now serves a dual purpose: on the one hand, it provides configuration functionality to add and remove devices when calling the CDO, and on the other hand the major function codes will be called by clients of the filtered devices themselves.

We start `DriverEntry` by creating the CDO and exposing it through a symbolic link as we've seen numerous times:

```
DevMonManager g_Data;

extern "C" NTSTATUS
DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING) {
    UNICODE_STRING devName = RTL_CONSTANT_STRING(L"\Device\KDevMon");
    PDEVICE_OBJECT DeviceObject;

    auto status = IoCreateDevice(DriverObject, 0, &devName,
        FILE_DEVICE_UNKNOWN, 0, TRUE, &DeviceObject);
    if (!NT_SUCCESS(status))
        return status;

    UNICODE_STRING linkName = RTL_CONSTANT_STRING(L"\?\?\Device\KDevMon");
    status = IoCreateSymbolicLink(&linkName, &devName);
    if (!NT_SUCCESS(status)) {
        IoDeleteDevice(DeviceObject);
        return status;
    }
    DriverObject->DriverUnload = DevMonUnload;
```

Nothing new in this piece of code. Next we must initialize all dispatch routines so that all major functions are supported:

```
for (auto& func : DriverObject->MajorFunction)
    func = HandleFilterFunction;

// equivalent to:
// for (int i = 0; i < ARRAYSIZE(DriverObject->MajorFunction); i++)
//     DriverObject->MajorFunction[i] = HandleFilterFunction;
```

We've seen similar code earlier in this chapter. The above code uses a C++ reference to change all major functions to point to `HandleFilterFunction`, which we'll meet very soon. Finally, we need to save the returned device object for convenience in the global `g_Data` (`DevMonManager`) object and initialize it:

```

g_Data.CDO = DeviceObject;
g_Data.Init(DriverObject);

return status;
}

```

The `Init` method just initializes the fast mutex and saves the driver object pointer for later use with `IoCreateDevice` (which we covered in the previous section).

We will not be using a remove lock in this driver to simplify the code. The reader is encouraged to add support for a remove lock as described earlier in this chapter.

Before we dive into that generic dispatch routine, let's take a closer look at the unload routine. When the driver is unloaded, we need to delete the symbolic link and the CDO as usual, but we also must detach from all currently active filters. Here is the code:

```

void DevMonUnload(PDRIVER_OBJECT DriverObject) {
    UNREFERENCED_PARAMETER(DriverObject);
    UNICODE_STRING linkName = RTL_CONSTANT_STRING(L"\?\?\KDevMon");
    IoDeleteSymbolicLink(&linkName);
    NT_ASSERT(g_Data.CDO);
    IoDeleteDevice(g_Data.CDO);

    g_Data.RemoveAllDevices();
}

```

The key piece here is the call to `DevMonManager::RemoveAllDevices`. This function is fairly straightforward, leaning on `DevMonManager::RemoveDevice` for the heavy lifting:

```

void DevMonManager::RemoveAllDevices() {
    Locker locker(Lock);
    for (int i = 0; i < MaxMonitoredDevices; i++)
        RemoveDevice(i);
}

```

Handling Requests

The `HandleFilterFunction` dispatch routine is the most important piece of the puzzle. It will be called for all major functions, targeted to one of the filter devices or the CDO. The routine must make that distinction, and this is exactly why we saved the CDO pointer earlier. Our CDO supports create, close and `DeviceIoControl`. Here is the initial code:

```
NTSTATUS HandleFilterFunction(PDEVICE_OBJECT DeviceObject, PIRP Irp) {
    if (DeviceObject == g_Data.CDO) {
        switch (IoGetCurrentIrpStackLocation(Irp)->MajorFunction) {
            case IRP_MJ_CREATE:
            case IRP_MJ_CLOSE:
                return CompleteRequest(Irp);

            case IRP_MJ_DEVICE_CONTROL:
                return DevMonDeviceControl(DeviceObject, Irp);
        }
        return CompleteRequest(Irp, STATUS_INVALID_DEVICE_REQUEST);
    }
}
```

If the target device is our CDO, we switch on the major function itself. For create and close we simply complete the IRP successfully by calling a helper function we met in chapter 7:

```
NTSTATUS CompleteRequest(PIRP Irp,
    NTSTATUS status = STATUS_SUCCESS,
    ULONG_PTR information = 0);

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

For IRP_MJ_DEVICE_CONTROL, we call DevMonDeviceControl, which should implement our control codes for adding and removing filters. For all other major functions, we just complete the IRP with an error indicating “unsupported operation”.

If the device object is not the CDO, then it must be one of our filters. This is where the driver can do anything with the request: log it, examine it, change it - anything it wants. For our driver we’ll just send to the debugger output some pieces of information regarding the request and then send it down to the device underneath the filter.

First, we’ll extract our device extension to gain access to the lower device:

```
auto ext = (DeviceExtension*)DeviceObject->DeviceExtension;
```

Next, we’ll get the thread that issued the request by digging deep into the IRP and then get the thread and process IDs of the caller:

```
auto thread = Irp->Tail.Overlay.Thread;
HANDLE tid = nullptr, pid = nullptr;
if (thread) {
    tid = PsGetThreadId(thread);
    pid = PsGetThreadProcessId(thread);
}
```

In most cases, the current thread is the same one that made the initial request, but it doesn't have to be - it's possible that a higher-layer filter received the request, did not propagate it immediately for whatever reason, and later propagated it from a different thread.

Now it's time to output the thread and process IDs and the type of operation requested:

```
auto stack = IoGetCurrentIrpStackLocation(Irp);

DbgPrint("Intercepted driver: %wZ: PID: %d, TID: %d, MJ=%d (%s)\n",
    &ext->LowerDeviceObject->DriverObject->DriverName,
    HandleToUlong(pid), HandleToUlong(tid),
    stack->MajorFunction, MajorFunctionToString(stack->MajorFunction));
```

The `MajorFunctionToString` helper function just returns a string representation of a major function code. For example, for `IRP_MJ_READ` it returns "IRP_MJ_READ".

At this point the driver can further examine the request. If `IRP_MJ_DEVICE_CONTROL` was received, it can look at the control code and the input buffer. If it's `IRP_MJ_WRITE`, it can look at the user's buffer, and so on.

This driver can be extended to capture these requests and store them in some list (as we did in chapters 8 and 9, for example), and then allow a user mode client to query for this information. This is left as an exercise for the reader.

Finally, since we don't want to hurt the operation of the target device, we'll pass the request along unchanged:

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

The `DevMonDeviceControl` function mentioned earlier is the driver's handler for `IRP_MJ_DEVICE_CONTROL`. This is used to add or remove devices from filtering dynamically. The defined control codes are as follows (in `KDevMonCommon.h`):

```
#define DEVMON_DEVICE 0x8004

#define IOCTL_DEVMON_ADD_DEVICE           \
    CTL_CODE(DEVMON_DEVICE, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_REMOVE_DEVICE        \
    CTL_CODE(DEVMON_DEVICE, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_REMOVE_ALL          \
    CTL_CODE(DEVMON_DEVICE, 0x802, METHOD_NEITHER, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_START_MONITOR       \
    CTL_CODE(DEVMON_DEVICE, 0x803, METHOD_NEITHER, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_STOP_MONITOR        \
    CTL_CODE(DEVMON_DEVICE, 0x804, METHOD_NEITHER, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_ADD_DRIVER          \
    CTL_CODE(DEVMON_DEVICE, 0x805, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DEVMON_REMOVE_DRIVER       \
    CTL_CODE(DEVMON_DEVICE, 0x806, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

The handling code should be fairly easy to understand by now:

```
NTSTATUS DevMonDeviceControl(PDEVICE_OBJECT, PIRP Irp) {
    auto stack = IoGetCurrentIrpStackLocation(Irp);
    auto status = STATUS_INVALID_DEVICE_REQUEST;
    auto code = stack->Parameters.DeviceIoControl.IoControlCode;

    switch (code) {
        case IOCTL_DEVMON_ADD_DEVICE:
        case IOCTL_DEVMON_REMOVE_DEVICE:
        {
            auto buffer = (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
            auto len = stack->Parameters.DeviceIoControl.InputBufferLength;
            if (buffer == nullptr || len < 2 || len > 512) {
                status = STATUS_INVALID_BUFFER_SIZE;
                break;
            }

            buffer[len / sizeof(WCHAR) - 1] = L'\0';
            if (code == IOCTL_DEVMON_ADD_DEVICE)
                status = g_Data.AddDevice(buffer);
            else {
                auto removed = g_Data.RemoveDevice(buffer);
                status = removed ? STATUS_SUCCESS : STATUS_NOT_FOUND;
            }
            break;
        }
    }
}
```

```
    }

    case IOCTL_DEVMON_REMOVE_ALL:
    {
        g_Data.RemoveAllDevices();
        status = STATUS_SUCCESS;
        break;
    }
}

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

Testing the Driver

The user mode console application is again fairly standard, accepting a few commands for adding and removing devices. Here are some examples for issuing commands:

```
devmon add \device\procexp152
devmon remove \device\procexp152
devmon clear
```

Here is the main function of the user mode client (very little error handling):

```
int wmain(int argc, wchar_t* argv[]) {
    if (argc < 2)
        return Usage();

    auto const cmd = argv[1];

    HANDLE hDevice = CreateFile(L"\\\\.\\\\kdevmon",
        GENERIC_READ | GENERIC_WRITE, 0,
        nullptr, OPEN_EXISTING, 0, nullptr);
    if (hDevice == INVALID_HANDLE_VALUE)
        return Error("Failed to open device");

    DWORD bytes;
    if (_wcsicmp(cmd, L"add") == 0) {
        if (!DeviceIoControl(hDevice, IOCTL_DEVMON_ADD_DEVICE, argv[2],
            DWORD(::wcslen(argv[2]) + 1) * sizeof(WCHAR), nullptr, 0,
            &bytes, nullptr))
            return Error("Failed to add device");
        printf("Add device %ws successful.\n", argv[2]);
    }
}
```

```

        return 0;
    }
    else if (_wcsicmp(cmd, L"remove") == 0) {
        if (!DeviceIoControl(hDevice, IOCTL_DEVMON_REMOVE_DEVICE, argv[2],
            DWORD(::wcslen(argv[2]) + 1) * sizeof(WCHAR), nullptr, 0,
            &bytes, nullptr))
            return Error("Failed in remove device");
        printf("Remove device %ws successful.\n", argv[2]);
        return 0;
    }
    else if (_wcsicmp(cmd, L"clear") == 0) {
        if (!DeviceIoControl(hDevice, IOCTL_DEVMON_REMOVE_ALL,
            nullptr, 0, nullptr, 0, &bytes, nullptr))
            return Error("Failed in remove all devices");
        printf("Removed all devices successful.\n");
    }
    else {
        printf("Unknown command.\n");
        return Usage();
    }

    return 0;
}

```

We've seen this kind of code many times before.

The driver can be installed like so:

```
sc create devmon type= kernel binpath= c:\book\kdevmon.sys
```

And started with:

```
sc start devmon
```

As a first example, we'll launch *Process Explorer* (must be running elevated so its driver can be installed if needed), and filter requests coming to it:

```
devmon add \device\procexp152
```

Remember that *WinObj* shows a device named *ProcExp152* in the *Device* directory of the object manager namespace. We can launch *DbgView* from *SysInternals* elevated, and configure it to log kernel output. Here is some example output:

```

1 0.00000000 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_CONTROL)
2 0.00016690 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_CONTROL)
3 0.00041660 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_CONTROL)
4 0.00058020 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_CONTROL)
5 0.00071720 driver: \Driver\PROCEXP152: PID: 5432, TID: 8820, MJ=14 (IRP_MJ_DEVICE_CONTROL)

```

It should be no surprise to find out the process ID of *Process Explorer* on that machine is 5432 (and it has a thread with ID 8820). Clearly, *Process Explorer* sends to its driver requests on a timely basis, and it's always IRP_MJ_DEVICE_CONTROL.

The devices that we can filter can be viewed with *WinObj*, mostly in the *Device* directory, shown in Figure 15-14.

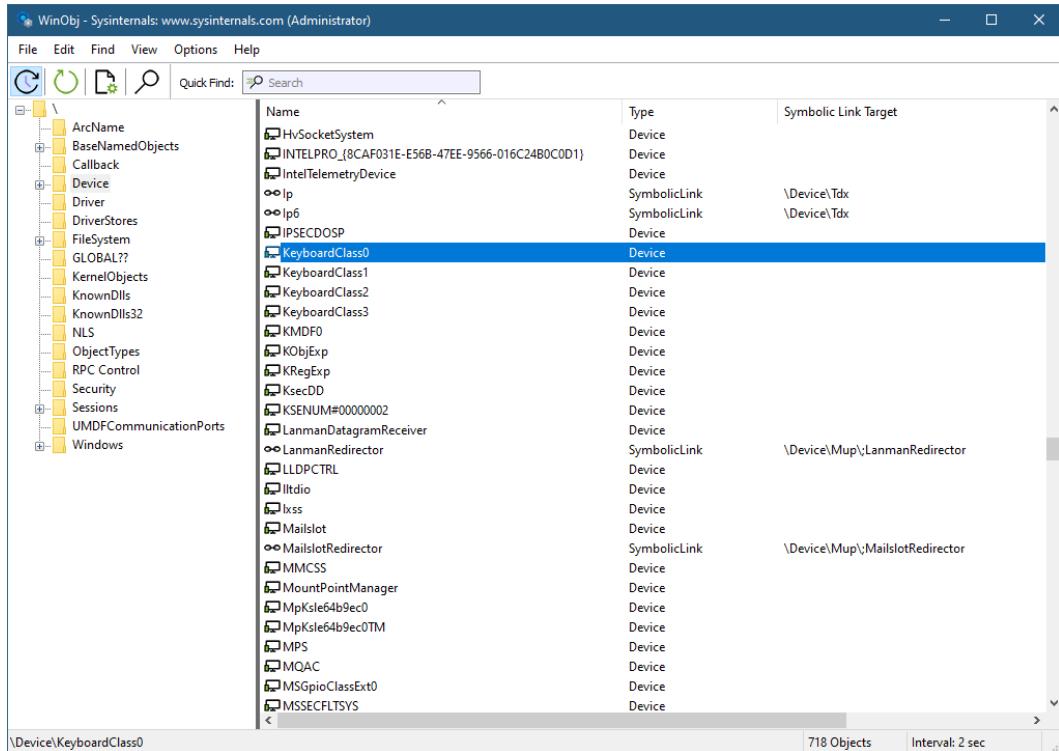


Figure 15-14: Device directory in *WinObj*

Let's filter on *keyboardclass0*, which is managed by the keyboard class driver:

```
devmon add \device\keyboardclass0
```

Now press some keys. You'll see that for every key pressed you get a line of output. Here is some of it:

```
1 11:31:18 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
2 11:31:18 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
3 11:31:19 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
4 11:31:19 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
5 11:31:20 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
6 11:31:20 driver: \Driver\kbdclass: PID: 612, TID: 740, MJ=3 (IRP_MJ_READ)
```

What is this process 612? This is an instance of *Csrss.exe* running in the user's session. One of *Csrss*' duties is to get data from input devices. Notice it's a read operation, which means some response buffer is expected from the keyboard class driver. But how can we get it? We'll get to that in the next section.

You can try out other devices. Some may fail to attach (typically those that are open for exclusive access), and some are not suited for this kind of filtering, especially file system drivers.

Here is an example with the *Multiple UNC Provider* device (MUP):

```
devmon add \device\mup
```

Navigate to some network folder and you'll see lots of activity similar to what you see here:

```
001 11:46:19 driver: \FileSystem\FltMgr: PID: 4, TID: 6236, MJ=2 (IRP_MJ_CLOSE)
002 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=0 (IRP_MJ_CREATE)
003 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=13 (IRP_MJ_FILE_SYSTEM_CONTROL)
004 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=18 (IRP_MJ_CLEANUP)
005 11:46:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 5600, MJ=2 (IRP_MJ_CLOSE)
006 11:47:00 driver: \FileSystem\FltMgr: PID: 7212, TID: 4464, MJ=0 (IRP_MJ_CREATE)
007 11:47:00 driver: \FileSystem\FltMgr: PID: 7212, TID: 4464, MJ=13 (IRP_MJ_FILE_SYSTEM_CONTROL)
...
054 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=13 (IRP_MJ_FILE_SYSTEM_CONTROL)
055 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=18 (IRP_MJ_CLEANUP)
```

```
056 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=2 (IRP_MJ_CLO\SE)
057 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 8272, MJ=5 (IRP_MJ_QUE\RY_INFORMATION)
...
094 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=0 (IRP_MJ_CRE\ATE)
095 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=0 (IRP_MJ_CRE\ATE)
096 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=5 (IRP_MJ_QUE\RY_INFORMATION)
097 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=18 (IRP_MJ_CL\EANUP)
098 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=5 (IRP_MJ_QUE\RY_INFORMATION)
099 11:47:25 driver: \FileSystem\FltMgr: PID: 6164, TID: 6620, MJ=2 (IRP_MJ_CLO\SE)
100 11:47:25 driver: \FileSystem\FltMgr: PID: 7212, TID: 7288, MJ=12 (IRP_MJ_DI\RECTORY_CONTROL)
...
...
```

Notice the layering is on top of the Filter Manager we met in chapter 10. Also notice that multiple processes are involved (both are *Explorer.exe* instances). The MUP device is a volume for the Remote file system. This type of device is best filtered with a file system mini-filter.



Feel free to experiment!

Results of Requests

The generic dispatch handler we have for the *DevMon* driver only sees requests coming in. These can be examined, but an interesting question remains - how can we get the results of the request? Some driver down the device stack is going to call `IoCompleteRequest`. If the driver is interested in the results, it must set up an I/O completion routine.

As discussed in chapter 7, completion routines are invoked in reverse order of registration when `IoCompleteRequest` is called. Each layer in the device stack (except the lowest one) can set up a completion routine to be called as part of request completion. At this time, the driver can inspect the IRP's status, examine output buffers, etc.

Setting up a completion routine is done with `IoSetCompletionRoutine` or (better) `IoSetCompletionRoutineEx`. Here is the latter's prototype:

```
NTSTATUS IoSetCompletionRoutineEx (
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_ PIRP Irp,
    _In_ PIO_COMPLETION_ROUTINE CompletionRoutine,
    _In_opt_ PVOID Context,      // driver defined
    _In_ BOOLEAN InvokeOnSuccess,
    _In_ BOOLEAN InvokeOnError,
    _In_ BOOLEAN InvokeOnCancel);
```

Most of the parameters are pretty self-explanatory. The last three parameters indicate for which IRP completion status to invoke the completion routine:

- If *InvokeOnSuccess* is TRUE, the completion routine is called if the IRP's status **passes** the NT_-SUCCESS macro.
- If *InvokeOnError* is TRUE, the completion routine is called if the IRP's status **fails** the NT_-SUCCESS macro.
- If *InvokeOnCancel* is TRUE, the completion routine is called if the IRP's status is STATUS_-CANCELLED, which means the request has been canceled.

The completion routine itself must have the following prototype:

```
NTSTATUS CompletionRoutine (
    _In_ PDEVICE_OBJECT DeviceObject,
    _In_ PIRP Irp,
    _In_opt_ PVOID Context);
```

The completion routine is called by an arbitrary thread (the one that called `IoCompleteRequest`) at IRQL DISPATCH_LEVEL (2). This means all the rules for IRQL 2 must be followed.

What can the completion routine do? It can examine the IRP's status and buffers, and can call `IoGetCurrentIrpStackLocation` to get more information from the IO_STACK_LOCATION. It must **not** call `IoCompleteRequest`, because this already happened (this is the reason we are in the completion routine in the first place).

What about the return status? There are actually only two options here: STATUS_MORE_PROCESSING_REQUIRED and everything else. Returning that special status tells the I/O manager to stop propagating the IRP up the device stack and “undo” the fact the IRP was completed. The driver claims ownership of the IRP and must eventually call `IoCompleteRequest` again (this is not an error). This option is mostly for hardware-based drivers and will not be discussed further in this book.

Any other status returned from the completion routine continues propagation of the IRP up the device stack, possibly calling other completion routines for upper layer drivers. In this case, the driver must mark the IRP as pending if the lower device marked it as one:

```
if (Irp->PendingReturned)
    IoMarkIrpPending(Irp); // sets SL_PENDING_RETURNED in irpStackLoc->Control
```

This is necessary because the I/O manager does the following after the completion routine returns:

```
Irp->PendingReturned = irpStackLoc->Control & SL_PENDING_RETURNED;
```



The exact reasons for all these intricacies are beyond the scope of this book. The best source of information on these topics is Walter Oney's excellent book, "Programming the Windows Driver Model", second edition (MS Press, 2003). Although the book is old (covering Windows XP), and it's about hardware device drivers only, it's still quite relevant and has some great information.



Implement an I/O completion routine for the *DevMon* driver.

Driver Hooking

Using filter drivers described in this chapter and in chapter 10 provides a lot of power to a driver developer: the ability to intercept requests to almost any device. In this section I'd like to mention another technique, that although not "official", may be quite useful in certain cases.

This driver hooking technique is based on the idea of replacing dispatch routine pointers of running drivers. This automatically provides "filtering" for all devices managed by that driver. The hooking driver will save the old function pointers and then replace the major function array in the driver object with its own functions. Now any request coming to a device under control of the hooked driver will invoke the hooking driver's dispatch routines. There is no extra device objects or any attaching going on here.



Some drivers are protected by *PatchGuard* against these kinds of hooks. A canonical example is the NTFS file system driver - on Windows 8 and later - cannot be hooked in that way. If it is, the system will crash after a few minutes.



PatchGuard (officially called *Kernel Patch Protection*) is a kernel mechanism that hashes various data structures that are considered important, and if any change is detected - will crash the system. A classic example is the *System Service Dispatch Table* (SSDT) which points to system services (system calls).

Drivers have names and thus are part of the Object Manager's namespace, residing in the *Driver* directory, shown with *WinObj* in Figure 15-15 (must run elevated to see the contents of the *Driver* directory).

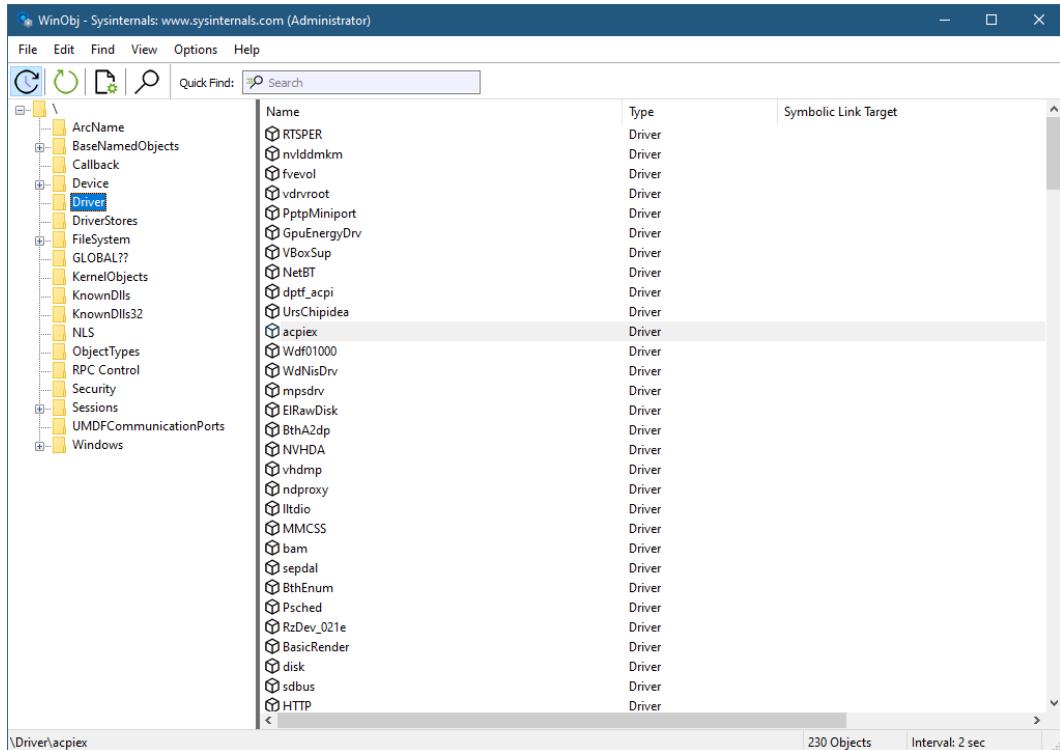


Figure 15-15: The *Driver* directory in *WinObj*

To hook a driver, we need to locate the driver object pointer (DRIVER_OBJECT), and to do that we can use an undocumented, but exported, function that can locate any object given its name:

```
NTSTATUS ObReferenceObjectByName (
    _In_ PUNICODE_STRING ObjectPath,
    _In_ ULONG Attributes,
    _In_opt_ PACCESS_STATE PassedAccessState,
    _In_opt_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_TYPE ObjectType,
    _In_ KPROCESSOR_MODE AccessMode,
    _Inout_opt_ PVOID ParseContext,
    _Out_ PVOID *Object);
```

Here is an example of calling *ObReferenceObjectByName* to locate the *kbdclass* driver:

```

UNICODE_STRING name;
RtlInitUnicodeString(&name, L"\\driver\\kbdclass");

PDRIVER_OBJECT driver;
auto status = ObReferenceObjectByName(&name, OBJ_CASE_INSENSITIVE,
    nullptr, 0, *IoDriverObjectType, KernelMode,
    nullptr, (PVOID*)&driver);
if(NT_SUCCESS(status)) {
    // manipulate driver
    ObDereferenceObject(driver);    // eventually
}

```

The hooking driver can now replace the major function pointers, the unload routine, the add device routine, etc. Any such replacement should always save the previous function pointers for unhooking when desired and for forwarding the request to the real driver. Since this replacement must be done atomically, it's best to use `InterlockedExchangePointer` to make the exchange atomically.

The following code snippet demonstrates this technique:

```

for (int j = 0; j <= IRP_MJ_MAXIMUM_FUNCTION; j++) {
    InterlockedExchangePointer((PVOID*)&driver->MajorFunction[j],
        MyHookDispatch);
}
InterlockedExchangePointer((PVOID*)&driver->DriverUnload, MyHookUnload);

```

A fairly complete example of this hooking technique can be found in my *DriverMon* project on Github at <https://github.com/zodiacon/DriverMon>.



Implement a driver that hooks other drivers using this technique. Create a user-mode client that can hook a specified driver on the command line.

Kernel Libraries

In the course of writing drivers, we developed some classes and helper functions that can be used in multiple drivers. It makes sense, though, to package them in a single library that we can then reference instead of copying source files from project to project.

The project templates provided with the WDK don't explicitly provide a static library for drivers, but it's fairly easy to make one. The way to do this is to create a normal driver project (based on *WDM*

Empty Driver for example), and then just change the project type to a static library as shown in Figure 15-16.

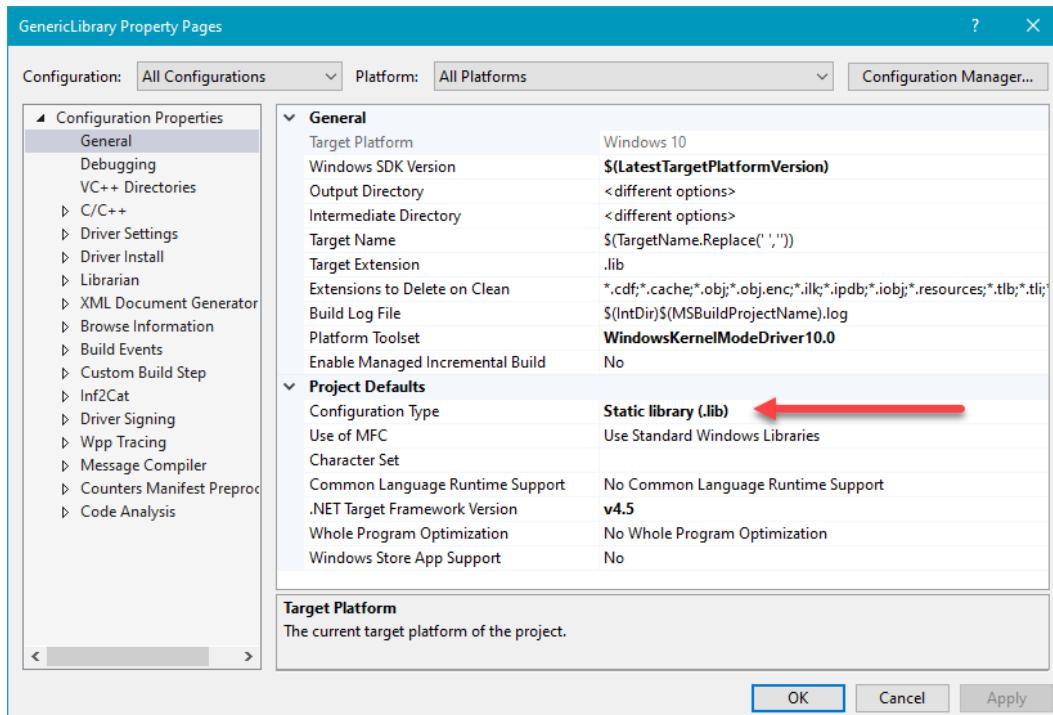


Figure 15-16: Configuring a kernel static library

A driver project that wants to link to this library just needs to add a reference with Visual Studio by right-clicking the *References* node in *Solution Explorer*, choosing *Add Reference...* and checking the library project. Figure 15-17 shows the references node of an example driver after adding the reference.

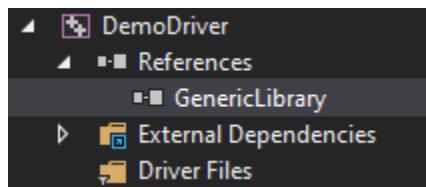


Figure 15-17: Referencing a library

Summary

Kernel programming is a vast topic, some parts of which we covered in this book. Obviously, there is more. Most kernel driver topics are documented in the WDK, and if you followed the book you should have a much easier time reading that documentation.

I wish you happy kernel programming!

Appendix: The Kernel Template Library

The Kernel Template Library (KTL) is a set of types and functions to help write kernel drivers in a safe and less error-prone way. Many of these classes have been used throughout the book. This appendix summarizes the provided classes at the time of writing.

The KTL is a work in progress. Interested readers are welcome to contribute by providing pull requests and raising issues. The KTL can be found at
<https://github.com/zodiacon/windowskernelprogrammingbook2e/tree/master/ktl>

Standard Library

The `std.h` file adds support for move semantics with a `std::move` function that behaves like its user-mode counterpart. This allows adding move semantics to kernel types.

Synchronization

Several wrappers are provided to deal with thread and processor synchronization. All have an `Init` method, as well as `Lock` and `Unlock`.

- `FastMutex` - wraps a `FAST_MUTEX` structure.
- `Mutex` - wraps a `KMUTEX` structure.
- `SpinLock` - wraps a `KSPIN_LOCK`.
- `ExecutiveResource` - wraps an `ERESOURCE`. Also has `LockShared` method to acquire the shared lock.
- `Locker<>` class template - provides RAII locking over any one of the above.
- `SharedLocker<>` used with `ExecutiveResource` when the shared lock is needed.

Memory

The `new` and `delete` operator are overloaded, with an enumeration that makes it less likely to get an error with the pool flags (`memory.h` and `Memory.cpp`):

```

enum class PoolType : ULONG64 {
    Paged = POOL_FLAG_PAGED,
    NonPaged = POOL_FLAG_NON_PAGED,
    NonPagedExecute = POOL_FLAG_NON_PAGED_EXECUTE,
    CacheAligned = POOL_FLAG_CACHE_ALIGNED,
    Uninitialized = POOL_FLAG_CACHE_ALIGNED,
    ChargeQuota = POOL_FLAG_USE_QUOTA,
    RaiseOnFailure = POOL_FLAG_RAISE_ON_FAILURE,
    Session = POOL_FLAG_SESSION,
    SpecialPool = POOL_FLAG_SPECIAL_POOL,
};

DEFINE_ENUM_FLAG_OPERATORS(PoolType);

void* __cdecl operator new(size_t size, PoolType pool,
    ULONG tag = DRIVER_TAG);
void* __cdecl operator new[](size_t size, PoolType pool,
    ULONG tag = DRIVER_TAG);

void __cdecl operator delete(void* p, size_t);
void __cdecl operator delete[](void* p, size_t);

```

The `LookasideList` template class is a wrapper around lookaside lists (either paged or non-paged). See `LookasideList.h`.

Strings

The `BasicString<>` template class provides support for a variable-length string, either UTF-16 or ANSI, based on template arguments:

```

template<typename T, PoolType Pool, ULONG Tag = DRIVER_TAG>
class BasicString;

```

Several specialization are defined:

```

template<PoolType Pool, ULONG Tag = DRIVER_TAG>
using WString = BasicString<wchar_t, Pool, Tag>;

template<ULONG Tag = DRIVER_TAG>
using NPWString = BasicString<wchar_t, PoolType::NonPaged, Tag>;
template<ULONG Tag = DRIVER_TAG>
using PWString = BasicString<wchar_t, PoolType::Paged, Tag>;

template<PoolType Pool, ULONG Tag>
using AString = BasicString<char, Pool, Tag>;

template<ULONG Tag = DRIVER_TAG>
using NPAString = BasicString<char, PoolType::NonPaged, Tag>;
template<ULONG Tag = DRIVER_TAG>
using PAString = BasicString<char, PoolType::Paged, Tag>;

```

See *BasicString.h* for more details.

Containers

The `Vector<>` template class abstract a dynamic array of objects that are trivially constructible and copyable, i.e. don't have dynamic memory internally. Examples are integers, and plain structure.

```

template<typename T, PoolType Pool, ULONG Tag = DRIVER_TAG>
class Vector;

```

See *Vector.h* for details.

The `LinkedList<>` template class wraps a LIST_ENTRY based linked-list with synchronization:

```

template<typename T, typename TLock = FastMutex>
struct LinkedList;

```

See the *LinkedList.h* file for details.

File System Mini-Filters

The `FilterFileNameInformation` class provides a RAII wrapper around `PFLT_FILE_NAME_INFORMATION`. See *FileNameInformation.h* for the details.