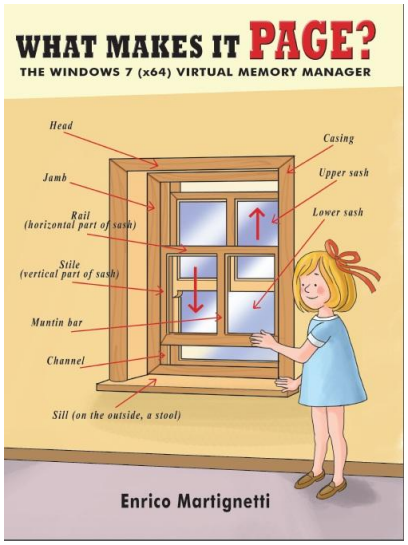



If you are interested in the internals of Windows, check out Enrico Martignetti's book on the Virtual Memory Manager.

Click here to find out more.

# Test Filter Driver

By Enrico Martignetti

Version 1.1

# Introduction

This driver is a generic filter which can be applied on top of any other driver. It does not modify the IRPs it intercepts but passes them to the underlying filtered device. While doing this, it stores the IRPs along with other data into an internal buffer, which can be read by a client program. Thus, it makes possible to explore and log the IRPs targeted at the filtered device.

Together with each IRP, the content of the current I/O stack location is traced as well.

For IRP_MJ_READ and IRP_MJ_WRITE IRPs, the data is also traced. As an example, by installing this filter on top of a disk driver, it is possible to trace all the data buffers written to or read from a disk.

It should be easy to extend this filter by adding code specific to other major function codes, giving it the ability to trace data for other kinds of IRPs besides read and write ones.

A good reading for background information on filter drivers is [1]

I developed this test filter driver mainly because I wanted to practice some Windows device driver programming, then I decided to publish it because its ability to trace IRPs targeted at a given device may be of some use and the code itself may be interesting to someone who, like me, is learning the ropes on this subject.

# Installation

The driver is made up of the single TestFilt.sys file, which can be stored anywhere on a local disk.

Afterwards, we have to create a valid entry into the registry to make it possible for the driver to be loaded. Suppose we name the driver TestFilter and the .sys file is stored as

C:\Apps\TestApp\TestFilt.sys

We'll have to create a key named TestFilter under HKLM\SystemCurrentControlSet\Services and create the following three values into it:

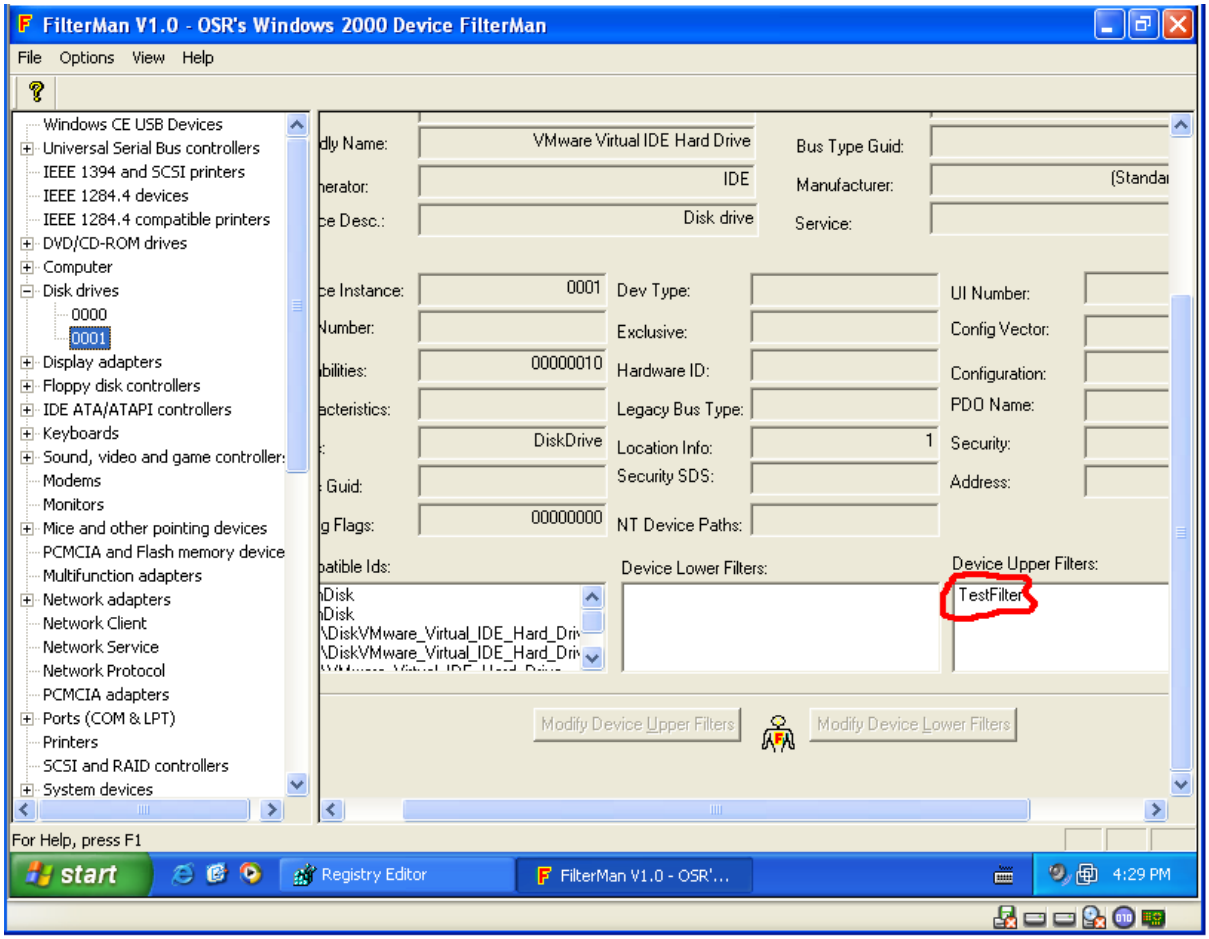
name: ErrorControl
type: REG_DWORD
value: 0x0

name: ImagePath
type: REG_EXPAND_SZ
value: \??\C:\Apps\TestApp\TestFilt.sys

name: Start
type: REG_DWORD
value: 0x3

name: Type
type: REG_DWORD
value: 0x1

Windows must be configured so that it loads our filter and gives him the opportunity to attach itself to the device stack. A great tool to do this is the Filterman utility, available for download at www.osronline.com. Below is a screenshot of Filterman with the filter configured as an upper filter for a disk drive device. The name TestFilter must match the name of the key we created under HKLM…Services.




After rebooting, Windows will load the filter driver, which will attach itself to the disk stack.

The filter sends messages to the debugger while it starts, so it's possible to verify that it has indeed started by checking the debugger console. Here is a sample console log:

```
FLT - DriverEntry - Test filter driver, compiled Aug 10 2008
17:44:49


FLT - Master device created. Device object: 0X82A21810, device
extension: 0x82a218c8

FLT - DriverEntry - Test filter driver succesfully loaded.
```

# Interacting with the Filter

The filter can be controlled by sending I/O control codes to a logical device named *master device*. The minimal client TestClt.exe allows basic control of the filter. Run TestClt /? For a short command line help.

The following command

TestClt /l

Lists the devices currently being filtered (the filter can monitor more devices at the same time – just configure it with FilterMan for any device you care for). Supposing we have installed the filter over a disk device as shown before, we get something similar to this:

```
C:\Apps\TestApp>testclt /l

Test client, compiled Aug  3 2008 22:00:58


Complete data:       yes

Bytes needed:        2240

Device:         FiltAccess1
Text:              VMware Virtual IDE Hard Drive
Device ID:         IDE\DiskVMware_Virtual_IDE_Hard_Drive___________00000001
Instance ID:       31303030303030303030303030303030303030303130
Hardware IDs:
IDE\DiskVMware_Virtual_IDE_Hard_Drive___________00000001, IDE
\VMware_Virtual_IDE_Hard_Drive_________
Compatible IDs:    GenDisk
```

The output shows the device text and various Ids of the filtered device. The name FiltAccess1 on the first line is the name to be used to interact with this particular instance of the filter, through the commands explained below.

When just loaded, the filter is set not to trace any IRP it intercepts and we must send it a command to begin a trace. Suppose we want to trace all the IRP_MJ_WRITE IRPs of our disk. We will enter this command:

```
C:\Apps\TestApp>TestClt /c s 0x4 0x1 FiltAccess1

Test client, compiled Aug  3 2008 22:00:58


Setting flags for MajorFunction = 0x00000004 to 0x00000001
```

The 0x4 is the numerical value of IRP_MJ_WRITE and specifies the major function code for which we are configuring the filter.

The filter has one flag DWORD for each possible major function code which tells it how to behave for that particular code. The /c s command above sets the flag dword to the specified value, which is 0x1 in our example. This value means to turn on the trace for the function code (it is also the only value currently defined - into DrvR3.h).

Finally, FiltAccess1 identifies the filter instance we are controlling.

Upon receving this command, the filter begins tracing and sending messages to the debugger console:

```
FLT - DispatchFilteredWrite - Tracing IRP 0X82AAB608
      MajorFunction = IRP_MJ_WRITE
      MinorFunction = 0
      Filter 0X82A20E40
      target text    VMware Virtual IDE Hard Drive
      target HW ID
IDE\DiskVMware_Virtual_IDE_Hard_Drive_____________00000001
      target inst ID 31303030303030303030303030303030303030303130
      data length = 0x1000
FLT - GetWriteData - Write data address: 0X82880000, length: 0x1000
CB - CBAllocateBlock - Allocating block: start 0X82A0C000, length (not
counting hdr) 0x109c bytes
CB - CBAllocateBlock - Bytes remaining: 0x12f60
CB - CBStore - Storing.
    Current block start:       0X82A0C000
    length (not counting hdr): 0x109c
    store addr:                0X82A0C004
    store length:              0x8
    src addr:                  0XF7D0FFE8
FLT - InterceptIrpAndData - Storing IRP 0X82AAB608 into the buffer
CB - CBStore - Storing.
    Current block start:       0X82A0C000
    length (not counting hdr): 0x109c
    store addr:                0X82A0C00C
    store length:              0x70
    src addr:                  0X82AAB608
FLT - InterceptIrpAndData - Storing current stack location 0X82AAB708 into
the buffer
CB - CBStore - Storing.
    Current block start:       0X82A0C000
    length (not counting hdr): 0x109c
    store addr:                0X82A0C07C
    store length:              0x24
    src addr:                  0X82AAB708
FLT - InterceptIrpAndData - Storing 0x1000 bytes of data at 0X82880000 into
the buffer
CB - CBStore - Storing.
    Current block start:       0X82A0C000
    length (not counting hdr): 0x109c
    store addr:                0X82A0C0A0
    store length:              0x1000
    src addr:                  0X82880000
```

The filter uses an internal buffer to store the traced data, handled in a circular fashion: when the buffer is full, the oldest data are discarded to make room for new ones. The log entries beginning with CB are from the buffer management routines (CircBuffer.cpp).

Once we have captured some data, we can extract it with the client which writes it to the standard output with a command like this one (here we are redirecting the output to the file trace.log):

```
testclt /d x86 wxp FiltAccess1 > trace.log
```

x86 and wxp specify the architecture and OS version on which the filter is running. These information are used by the routines which format the IRP and the I/O stack location to select the correct type declaration, which is platform dependent.

The resulting trace.log file contains something similar to this:

```
Test client, compiled Aug 10 2008 18:43:15

Device succesfully opened


--------------------------------------

Traced IRP

Data Length: 0x1000

IRP:

MdlAddress =                                    0X82A38530
Flags =                                         0x43
AssociatedIrp.MasterIrp =                       00000000
AssociatedIrp.SystemBuffer =                    00000000
IoStatus.Status =                               0
IoStatus.Information =                          0
RequestorMode =                                 0
PendingReturned =                               0
Cancel =                                        0
CancelIrql =                                    0
CancelRoutine =                                 00000000
UserBuffer =                                    0X828A7000
Tail.Overlay.Thread =                           0X82BC8DA8


I/O Stack location:

MajorFunction =                                 IRP_MJ_WRITE
MinorFunction =                                 0
Flags =                                         0xc
Control =                                       0xe0
Parameters.Write.Length =                       0x1000
Parameters.Write.Key =                          0
```

```
Parameters.Write.ByteOffset =                    0x3fa01000
DeviceObject =                                   0X82A20E40
FileObject =                                     0X82A20E40

0x00000000 - 52 43 52 44 28 00 09 00 e6 27 11 01 00 00 00 00 -- RCRD(....'......
0x00000010 - 01 00 00 00 01 00 01 00 30 0f 00 00 00 00 00 00 -- ........0.......
0x00000020 - db 27 11 01 00 00 00 00 47 f4 00 00 00 00 00 00 -- .'......G.......
[…]
```

Turning off the trace is as simple as clearing the flag for the function code:

```
C:\Apps\TestApp>TestClt /c s 0x4 0x0 FiltAccess1

Test client, compiled Aug 10 2008 18:43:15


Setting flags for MajorFunction = 0x00000004 to 0000000000
```

The "pseudo function code" 0xff sets the flags for all the function codes at once, thus, for instance:

```
C:\Apps\TestApp>TestClt /c s 0xff 0x1 FiltAccess1
```

will turn on the trace for all kinds of IRPs.

For major function codes different from IRP_MJ_READ and IRP_MJ_WRITE, turning on the trace has the effect of tracing the IRP and current I/O stack location, without any additional data.

# Source Code and Building

The source code is organized into a Visual C++ 2008 Express solution.

However, it's not mandatory to have VS installed to edit the code and build the driver. The solution projects can just be regarded as subfolders of the overall source tree and all the building is done with the WDK build utility or the SDK nmake tool. VS can be used as a text editor and to access the various files through the Solution Explorer window, but this is not necessary.

The source files have been compiled with the following products:

- Microsoft Windows Driver Kit 7.1.0.7600 for the driver sources.
- Windows Software Development Kit (SDK) for Windows 7 and .NET Framework 3.5 Service Pack 1 for the client sources.

The solution is made up of the projects described below.

## *CommonFiles*

Contains some files common to several projects.

## *Driver Project*

Contains the driver source code (DrvMain.cpp, CircBuffer.cpp).

The various folder ending with "build" contain the SOURCE and MAKEFILE to build the filter for a specific platform.

To build the driver, run build -gcwZ in the WDK build command prompt for the platform the subdirectory is for.

## *KrnlUtils Project*

Contains a library of utility functions for working with kernel mode data structures in user mode programs. More details about these functions can be found later in the section *KrnlUtils: IRP and I/O stack formatting*.

The library can be built from a Windows SDK command prompt by executing nmake from the project directory:

| | |
|---|---|
| nmake | updates the library incrementally |
| nmake rebuild | deletes the last build and rebuilds everything |

The library has been built for the following Windows versions: Windows XP x86, Windows Vista x64, Windows vista x86, Windows 7 x64, Windows 7 x86.

## *TestClt Project*

Contains the source code for the minimal client.

The executable can be built from a Windows SDK command prompt by executing nmake from the project directory:

| | |
|---|---|
| nmake | updates the executable incrementally |
| nmake rebuild | deletes the last build and rebuilds everything |

The executable has been built for the following Windows versions: Windows XP x86, Windows Vista x64, Windows vista x86, Windows 7 x64, Windows 7 x86. The various versions are available under the Executables\TestClt folder of the filter zip file. There different directories correspond to builds for different Windows versions:

| | |
|---|---|
| 5.01\x86 | Windows XP x86 |
| 6.0\AMD64 | Windows Vista x64 |
| 6.0\x86 | Windows Vista x86 |
| 6.1\AMD64 | Windows 7 x64 |
| 6.1\x86 | Windows 7 x86 |

The makefiles use the APPVER and PROCESSOR_ARCHITECTURE environment variables to build the output path for the executable.

# Filter Structure

The filter code is entirely contained into DrvMain.cpp, while the circular buffer management routines are separated into CircBuffer.cpp

## *Devices Created by the Filter*

The filter creates a single master device and then, for each device it filters:

- One filter device (FD)
- One filter access device (FAD)

The FD is the one which attaches itself to the stack of the filtered one. The FAD for a particular FD is created to allow interaction with it, by means of IRPs the FAD processes. Each FAD has access to the device extension of the associated FD, where the control flags and intercepted data are stored, therefore the FAD can control the FD and extract the data.

FADs are kept in a linked list and the master device has a pointer to the list head, so that it is able to return a list of symbolic links for the FADs as well as information on the filtered device for each FAD (what we get with the /l command of the minimal client).

## *Main Dispatch Routine*

The driver main dispatch routine is named MainDispatch and its job is to determine for which kind of device an IRP is: the main device an FD or an FAD. It looks into the device extension where the member hdr.DevType, common to all three kinds of devices, stores a value from an enumerated type which identifies the device category.

Afterwards it just calls the main dispatch routine for the relevant type: FltMainDispatch, FAMainDispatch, MAMainDispatch.

FltMain Dispatch always forwards the IRP to the next lower driver, so that the filter is transparent. It also traces the IRP data, if tracing is active for the particular major code.

MainDispatch is set as the dispatch routine for all possible major function codes, therefore all IRPs targeted at the filter are forwarded to the underlying stack.

## *IRP and Data Tracing*

The filter code stores the traced data into its internal circular buffer. The buffer management routines are in a separate module: CircBuffer.cpp. The routines are general purpose ones and are declared into CircBuffer.h

The tracing code is invoked into FltMainDispatch. There are two functions specific for read and write IRPs, which also catpure the read or written data and a generic function (GenericTrace()), used for all the other IRPs, which just captures the IRP and the current stack location.

When a read is performed on a FAD, it is processed by FAMainDispatch() and gets the oldest IRP stored into the internal buffer, if any, or is pended and completed when data are available.

# KrnlUtils: IRP and I/O stack formatting

The KrnlUtils project contains a set of utility routines used by the minimal client, among which there are functions to display the content of an IRP or an I/O stack location in readable format.

I have not been able to find data declarations for user mode programs for the two data types IRP and IO_STACK_LOCATION, so I had to build my own ones. The problem here is to correctly take into account packing and alignment for all the data types involved, however the declaration I extracted seems to be correct. For the curious ones, the next section explaines the method I used.

These kernel mode data types are declared in KrnlDefs.h, surrounded by #if directives. There are, in fact, five declarations: for XP on x86, for Vista on x86/x64 and for Windows 7 on x86/x64. The #if directives control which one is expanded.

The PlatfDepCompile.cpp source contains routines which use these data types but does not include KrnlDefs.h.

Instead, one source per platform (Win7X64.cpp, Win7X86.cpp, WlhX64.cpp, WlhX86.cpp, WxpX86.cpp) includes PlatfDepCompile and KrnlDefs.h and defines the macros which result in the data types for the specific platform being expanded. Thus, these three source files result in object modules which can correctly format IRPs and such for a given platform. The three platform specific sources define platform specific functions with unique names to format the data (FormatIrpWlhX64, FormatIrpWlhX86, etc.). These functions are called by the one into KrnlUtils.cpp like FormatIrp, which take platform identifiers as input parameters and call the specific formatting function.

Since the functions in KrnlUtils.cpp are not compiled for a particular platform, they cannot refer to kernel data types such IRP. Therefore, the input pointers to the structures to be formatted are declared as PCHAR.

On the other hand, the functions in KrnlUtils allow to specify *at run time* the platform from which a given data block has been retrieved and to format it correctly.

The following section explains the method used to extract the data types declarations.

## *IRP and IO_STACK_LOCATION Declarations for User Mode*

The problem was to code declarations for these two data types for all the platforms to be supported.

Doing it by hand would mean correctly interpreting all the included WDK headers, which depend on several /D defines expanded at compilation time by the build utility and use several macros which hide directives to control, among other things, memory alignment and packing. Furthermore, all the subtypes used by IRP and IO_STACK_LOCATION need to be declared.

Instead of doing this by hand, I compiled a kernel mode module with all the standard tools (WDK build env, build utility) and I added the /P compiler option to the compiler parameters through the USER_C_FLAGS macro. This option causes the compiler to output a .i file which is the output of the C preprocessor (the build then fails, because the /P option causes the compiler not to create the .obj file).

This is extremely useful, because all preprocessor directive are resolved in this file, which means that:

- All the included files are expanded into this unique file
- All the conditional directives (#if, …) have been resolved based upon the correct set of defines and only the code blocks that had to be expanded are present
- All the macros have been expanded

I then copied the declarations from the .i file into my own .h (KrnlDefs.h) manually copying all the required subtypes.

There was still one issue left: packing could be modified anywhere inside the code by means of #pragma pack directives, therefore I had to ensure my declarations reflected that.

To be sure of this, I used the #pragma pack(show) directive, which prints on the console the current packing. I renamed my .i file into .cpp and inserted show directives before each of the structures or unions I wanted to declare (IRP, IO_STACK_LOCATION and the types they use), then I compiled the resulting file, getting the correct packing for each data type, which I replicated into KrnlDefs.h with my own #pragma pack directives.

KrnlDefs.h contains several declarations for IRP, IO_STACK_LOCATION and their associated subtypes, one for each of the supported platform (XP on x86, Vista on x86 and x64, Windows 7 on x86 and x64). They are enclosed in #if directives which control which one is expanded.

# Revision History

## *1.0*

First release.

## *1.1*

Fixed bug in DriverEntry (DrvMain.cpp). `pFilterDev->Flags` is now set to the flags from the device object returned by *IoAttachDeviceToDeviceStack*. In previous versions, it was set to the flags from the target device (`pTargDev`). This was incorrect, athough it worked with some devices (e.g. disks). This bug caused a blue screen when the filter was layered on top of a serial port.

Changed all the DbgPrint calls into DbgPrintEx ones, with DPFLTR_IHVDRIVER_ID and DPFLTR_ERROR_LEVEL, so that debug messages from the driver are printed in the debugger even when the message filtering of Windows Vista and later versions is in effect.

Added IRP and IO stack location formatting for: Windows 7 x64, Windows 7 x86.

# References

[1] Fun With Filters - Win2K/WDM Device Filter Drivers; The NT Insider, Vol 7, Issue 1, Jan-Feb 2000; available at www.osronline.com