# MiniFilter 微过滤驱动是相对于 SFilter 传统过滤驱动而言的，传统文件过滤驱动相对来说较为复杂， 且接口不清晰并不符合快速开发的需求，为了解决复杂的开发问题，微过滤驱动就此诞生，微过滤驱动 在编写时更简单，多数 IRP 操作都由过滤管理器 (FilterManager或Fltmgr) 所接管，因为有了兼容层， 所以在开发中不需要考虑底层 IRP 如何派发，更无需要考虑兼容性问题，用户只需要编写对应的回调函数处理请求即可，这极大的提高了文件过滤驱动的开发效率。

接下来将进入正题，讲解微过滤驱动的API定义规范以及具体的使用流程，并最终实现一个简单的过滤功 能，首先你必须在VS上做如下配置，依次打开配置菜单，并增加驱动头文件。

配置属性 > 连接器 > 输入> 附加依赖 > fltMgr.lib

配置属性 > C/C++ > 常规 > 设置 关闭所有警告 (警告视为错误关闭)

未过滤驱动的使用非常容易，在使用之前第一件事就是要向过滤管理器宣告我们的微过滤驱动的存在， 我们以 DriverEntry 入口函数为例，首先在入口处需要使用 FltRegisterFilter 函数注册一个过滤器组件，另外则需要通过 FltStartFiltering 开启过滤功能，而当我们想要关闭时则需要调用

FltUnregisterFilter 注销过滤组件，首先来看一下入口处是如何初始化的；

NTSTATUS DriverEntry(\_In\_ PDRIVER\_OBJECT DriverObject, \_In\_ PUNICODE\_STRING RegistryPath)

{

NTSTATUS status;

DbgPrint("Hello LyShark.com \n");

// FltRegisterFilter 向过滤管理器注册过滤器

// 参数1：本驱动驱动对象

// 参数2：微过滤驱动描述结构

// 参数3：返回注册成功的微过滤驱动句柄

status = FltRegisterFilter(DriverObject, &FilterRegistration, &gFilterHandle);

if (NT\_SUCCESS(status))

{

// 开启过滤

status = FltStartFiltering(gFilterHandle);

DbgPrint("[过滤器] 开启监控.. \n");

if (!NT\_SUCCESS(status))

{

// 如果启动失败,则取消注册并退出FltUnregisterFilter(gFilterHandle); DbgPrint("[过滤器] 取消注册.. \n");

}

}

return status;

}

如上代码中我们最需要关注的是 FltRegisterFilter 函数的第二个参数 FilterRegistration 它用于宣告注册信息，这个结构内包含了过滤器的所有信息，想要注册成功则我们必须更具要求正确的填写FLT\_REGISTRATION 微过滤器注册结构，该结构体的微软定义如下所示；

typedef struct \_FLT\_REGISTRATION { USHORT

USHORT

FLT\_REGISTRATION\_FLAGS

Size; Version;

Flags;



const FLT\_CONTEXT\_REGISTRATION const FLT\_OPERATION\_REGISTRATION PFLT\_FILTER\_UNLOAD\_CALLBACK PFLT\_INSTANCE\_SETUP\_CALLBACK

PFLT\_INSTANCE\_QUERY\_TEARDOWN\_CALLBACK PFLT\_INSTANCE\_TEARDOWN\_CALLBACK PFLT\_INSTANCE\_TEARDOWN\_CALLBACK PFLT\_GENERATE\_FILE\_NAME PFLT\_NORMALIZE\_NAME\_COMPONENT PFLT\_NORMALIZE\_CONTEXT\_CLEANUP PFLT\_TRANSACTION\_NOTIFICATION\_CALLBACK

PFLT\_NORMALIZE\_NAME\_COMPONENT\_EX

ContextRegistration; OperationRegistration;

FilterUnloadCallback;

InstanceSetupCallback; InstanceQueryTeardownCallback; InstanceTeardownStartCallback; InstanceTeardownCompleteCallback; GenerateFileNameCallback; NormalizeNameComponentCallback; NormalizeContextCleanupCallback; TransactionNotificationCallback;

NormalizeNameComponentExCallback;

PFLT\_SECTION\_CONFLICT\_NOTIFICATION\_CALLBACK SectionNotificationCallback;

} FLT\_REGISTRATION, PFLT\_REGISTRATION;

# 当然如上的这些字段并非都要去填充，我们只需要填充自己所需要的部分即可，例如我们代码中只填充 了如下这些必要的部分，其他部分可以省略掉，当使用如下结构体注册时，只要实例发生了变化就会根 据如下配置路由到不同的函数上面做处理。

// 过滤驱动数据结构

CONST FLT\_REGISTRATION FilterRegistration =

{

sizeof(FLT\_REGISTRATION), FLT\_REGISTRATION\_VERSION, 0,

NULL,

Callbacks, Unload, InstanceSetup,

InstanceQueryTeardown, InstanceTeardownStart, InstanceTeardownComplete, NULL,

NULL,

NULL

// 结构大小(默认)

//

//

//

//

//

//

//

//

//

//

//

//

结构版本(默认)

过滤器标志上下文

注册回调函数集

驱动卸载函数

实例安装回调函数实例销毁回调函数实例解除绑定时触发实例解绑完成时触发

GenerateFileName

GenerateDestinationFileName NormalizeNameComponent

};

如上结构中我们最需要注意的是 Callbacks 字段，该字段是操作回调函数集注册，我们对文件的各种操作的回调事件都会被写入到此处，而此处我们只需要增加我们所需要的回调事件即可，以IRP\_MJ\_CREATE 为例，后面紧跟的是 PreOperation 事前回调，以及 PostOperation 事后回调，一般在要进行监控时通常在 PreOperation() 回调中处理，如果时监视则一般在 PostOperation() 中处理。

// 回调函数集

CONST FLT\_OPERATION\_REGISTRATION Callbacks[] =

{

// 创建时触发 PreOperation(之前回调函数) / PostOperation(之后回调函数)

{ IRP\_MJ\_CREATE, 0, PreOperation, PostOperation },

// 读取时触发

{ IRP\_MJ\_READ, 0, PreOperation, PostOperation },

// 写入触发

{ IRP\_MJ\_WRITE, 0, PreOperation, PostOperation },

// 设置时触发

{ IRP\_MJ\_SET\_INFORMATION, 0, PreOperation, PostOperation },

// 结束标志

{ IRP\_MJ\_OPERATION\_END }

};

# 如下完整代码实现了监视当前系统下所有的文件操作，如创建，读取，写入，修改，加载后则会监视系 统下所有的文件操作，当然如果是监控则需要在 PreOperation 事前回调做文章，而如果仅仅只是监视则事前事后都是可以的。

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include <fltKernel.h> #include <dontuse.h> #include <suppress.h>

PFLT\_FILTER gFilterHandle;

// ------------------------------------------------------------------------------

----------

// 声明部分

// ------------------------------------------------------------------------------

----------

DRIVER\_INITIALIZE DriverEntry; NTSTATUS DriverEntry(

\_In\_ PDRIVER\_OBJECT DriverObject,

\_In\_ PUNICODE\_STRING RegistryPath

);

NTSTATUS

InstanceSetup(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ FLT\_INSTANCE\_SETUP\_FLAGS Flags,

\_In\_ DEVICE\_TYPE VolumeDeviceType,

\_In\_ FLT\_FILESYSTEM\_TYPE VolumeFilesystemType

);

VOID

InstanceTeardownStart(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ FLT\_INSTANCE\_TEARDOWN\_FLAGS Flags

);

VOID

InstanceTeardownComplete(

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ FLT\_INSTANCE\_TEARDOWN\_FLAGS Flags

);

NTSTATUS

Unload(

\_In\_ FLT\_FILTER\_UNLOAD\_FLAGS Flags

);

NTSTATUS

InstanceQueryTeardown(



\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_ FLT\_INSTANCE\_QUERY\_TEARDOWN\_FLAGS Flags

);

FLT\_PREOP\_CALLBACK\_STATUS

PreOperation(

\_Inout\_ PFLT\_CALLBACK\_DATA Data,

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_Flt\_CompletionContext\_Outptr\_ PVOID CompletionContext

); FLT\_POSTOP\_CALLBACK\_STATUS

PostOperation(

\_Inout\_ PFLT\_CALLBACK\_DATA Data,

\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,

\_In\_opt\_ PVOID CompletionContext,

\_In\_ FLT\_POST\_OPERATION\_FLAGS Flags

);

// ------------------------------------------------------------------------------

----------

// 回调函数

// ------------------------------------------------------------------------------

----------

// 当实例被安装时触发

NTSTATUS InstanceSetup(\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects, \_In\_ FLT\_INSTANCE\_SETUP\_FLAGS Flags, \_In\_ DEVICE\_TYPE VolumeDeviceType, \_In\_ FLT\_FILESYSTEM\_TYPE VolumeFilesystemType)

{

DbgPrint("[LyShark] 安装 MiniFilter \n"); return STATUS\_SUCCESS;

}

// 当实例被销毁时触发

NTSTATUS InstanceQueryTeardown(\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects, \_In\_ FLT\_INSTANCE\_QUERY\_TEARDOWN\_FLAGS Flags)

{

DbgPrint("[LyShark] 销毁 MiniFilter \n"); return STATUS\_SUCCESS;

}

// 实例解除绑定时触发

VOID InstanceTeardownStart(\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects, \_In\_ FLT\_INSTANCE\_TEARDOWN\_FLAGS Flags)

{

DbgPrint("[LyShark] 解绑 MiniFilter \n"); return STATUS\_SUCCESS;

}

// 实例解绑完成时触发

VOID InstanceTeardownComplete(\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects, \_In\_ FLT\_INSTANCE\_TEARDOWN\_FLAGS Flags)

{

DbgPrint("[LyShark] 解绑完成 MiniFilter \n"); return STATUS\_SUCCESS;

}



// 驱动关闭时卸载监控

NTSTATUS Unload(\_In\_ FLT\_FILTER\_UNLOAD\_FLAGS Flags)

{

DbgPrint("[LyShark] 卸载 MiniFilter \n"); FltUnregisterFilter(gFilterHandle);

return STATUS\_SUCCESS;

}

// 回调函数集

CONST FLT\_OPERATION\_REGISTRATION Callbacks[] =

{

// 创建时触发 PreOperation(之前回调函数) / PostOperation(之后回调函数)

{ IRP\_MJ\_CREATE, 0, PreOperation, PostOperation },

// 读取时触发

{ IRP\_MJ\_READ, 0, PreOperation, PostOperation },

// 写入触发

{ IRP\_MJ\_WRITE, 0, PreOperation, PostOperation },

// 设置时触发

{ IRP\_MJ\_SET\_INFORMATION, 0, PreOperation, PostOperation },

// 结束标志

{ IRP\_MJ\_OPERATION\_END }

};

// 过滤驱动数据结构

CONST FLT\_REGISTRATION FilterRegistration =

{

sizeof(FLT\_REGISTRATION), // 结构大小(默认)

FLT\_REGISTRATION\_VERSION, // 结构版本(默认)

0, // 过滤器标志

NULL, // 上下文

Callbacks, // 注册回调函数集

Unload, // 驱动卸载函数

InstanceSetup, // 实例安装回调函数

InstanceQueryTeardown, // 实例销毁回调函数

InstanceTeardownStart, // 实例解除绑定时触发

InstanceTeardownComplete, // 实例解绑完成时触发

NULL, // GenerateFileName

NULL, // GenerateDestinationFileName

NULL // NormalizeNameComponent

};

// ------------------------------------------------------------------------------

----------

// 功能函数

// ------------------------------------------------------------------------------

----------

// 预操作回调函数(在执行过滤操作之前先执行此处)

FLT\_PREOP\_CALLBACK\_STATUS PreOperation(\_Inout\_ PFLT\_CALLBACK\_DATA Data, \_In\_ PCFLT\_RELATED\_OBJECTS FltObjects, \_Flt\_CompletionContext\_Outptr\_ PVOID

CompletionContext)

{

NTSTATUS status;

// 获取文件路径

UCHAR MajorFunction = Data->Iopb->MajorFunction; PFLT\_FILE\_NAME\_INFORMATION lpNameInfo = NULL;

// 得到文件名相关信息

status = FltGetFileNameInformation(Data, FLT\_FILE\_NAME\_NORMALIZED | FLT\_FILE\_NAME\_QUERY\_DEFAULT, &lpNameInfo);

if (NT\_SUCCESS(status))

{

status = FltParseFileNameInformation(lpNameInfo); if (NT\_SUCCESS(status))

{

// 创建

if (IRP\_MJ\_CREATE == MajorFunction)

{

DbgPrint("[创建文件时] %wZ", &lpNameInfo->Name);

// 拒绝创建

// STATUS\_INSUFFICIENT\_RESOURCES 提示不是有效的资源

// STATUS\_ACCESS\_DISABLED\_NO\_SAFER\_UI\_BY\_POLICY 静默拒绝

// STATUS\_ACCESS\_DENIED 提示访问拒绝

// return STATUS\_ACCESS\_DENIED;

// return FLT\_PREOP\_COMPLETE;

}

// 读取

else if (IRP\_MJ\_READ == MajorFunction)

{

DbgPrint("[读取文件时] %wZ", &lpNameInfo->Name);

// return FLT\_PREOP\_COMPLETE;

}

// 文件写入

else if (IRP\_MJ\_WRITE == MajorFunction)

{

DbgPrint("[写入文件时] %wZ", &lpNameInfo->Name);

// return FLT\_PREOP\_COMPLETE;

}

// 修改文件信息

else if (IRP\_MJ\_SET\_INFORMATION == MajorFunction)

{

DbgPrint("[修改文件] %wZ", &lpNameInfo->Name);

// return FLT\_PREOP\_COMPLETE;

}

}

}

return FLT\_PREOP\_SUCCESS\_WITH\_CALLBACK;

}

// 后操作回调函数 (在执行过滤之后运行此处)

FLT\_POSTOP\_CALLBACK\_STATUS PostOperation(\_Inout\_ PFLT\_CALLBACK\_DATA Data, \_In\_ PCFLT\_RELATED\_OBJECTS FltObjects, \_In\_opt\_ PVOID CompletionContext, \_In\_ FLT\_POST\_OPERATION\_FLAGS Flags)

{

return FLT\_POSTOP\_FINISHED\_PROCESSING;

}

// ------------------------------------------------------------------------------

----------

// 入口函数

// ------------------------------------------------------------------------------

----------

NTSTATUS DriverEntry(\_In\_ PDRIVER\_OBJECT DriverObject, \_In\_ PUNICODE\_STRING RegistryPath)

{

NTSTATUS status;

DbgPrint("Hello LyShark.com \n");

// FltRegisterFilter 向过滤管理器注册过滤器

// 参数1：本驱动驱动对象

// 参数2：微过滤驱动描述结构

// 参数3：返回注册成功的微过滤驱动句柄

status = FltRegisterFilter(DriverObject, &FilterRegistration, &gFilterHandle);

if (NT\_SUCCESS(status))

{

// 开启过滤

status = FltStartFiltering(gFilterHandle);

DbgPrint("[过滤器] 开启监控.. \n");

if (!NT\_SUCCESS(status))

{

// 如果启动失败,则取消注册并退出

FltUnregisterFilter(gFilterHandle);

DbgPrint("[过滤器] 取消注册.. \n");

}

}

return status;

}

过滤驱动的安装方式有多种，可以通过函数注册或者使用INF文件像系统注册驱动，首先以INF为例安 装，通过修改INF中的 ServiceName 以及 DriverName 并将其改为 WinDDK ，将文件保存为

install.inf 鼠标右键选择安装即可。

[Version]

Signature = "$Windows NT$"

Class = "ActivityMonitor" ;指明了驱动的分组,必须指定.

ClassGuid = {b86dff51-a31e-4bac-b3cf-e8cfe75c9fc2} ;GUID 每个分组都有固定的GUID Provider = %Msft% ;变量值 从STRING节中可以看到驱动提供者的名称

DriverVer = 06/16/2007,1.0.0.1 ;版本号

CatalogFile = passthrough.cat ;inf对应的cat 文件 可以不需要

[DefaultInstall] OptionDesc

CopyFiles

= %ServiceDescription%

= MiniFilter.DriverFiles

[DefaultInstall.Services]

|  |  |  |
| --- | --- | --- |
| [DestinationDirs] |  | |
| DefaultDestDir | = 12 | ;告诉我们驱动拷贝到哪里 13代表拷贝到%windir% |
| MiniFilter.DriverFiles | = 12 | ;%windir%\system32\drivers |

AddService = %ServiceName%,,MiniFilter.Service

[DefaultUninstall]

DelFiles = MiniFilter.DriverFiles

[DefaultUninstall.Services]

DelService = %ServiceName%,0x200 ;Ensure service is stopped before deleting

[MiniFilter.Service] ;服务的一些信息DisplayName = %ServiceName%

Description = %ServiceDescription%

ServiceBinary = %12%\%DriverName%.sys ;%windir%\system32\drivers\ Dependencies = "FltMgr" ;服务的依赖

ServiceType = 2 ;SERVICE\_FILE\_SYSTEM\_DRIVER

StartType = 3 ;SERVICE\_DEMAND\_START

ErrorControl = 1 ;SERVICE\_ERROR\_NORMAL LoadOrderGroup = "FSFilter Activity Monitor" ;文件过滤分组

AddReg = MiniFilter.AddRegistry ;文件过滤注册表需要添加的高度值等信息

[MiniFilter.AddRegistry] HKR,,"DebugFlags",0x00010001 ,0x0

HKR,"Instances","DefaultInstance",0x00000000,%DefaultInstance% HKR,"Instances\"%Instance1.Name%,"Altitude",0x00000000,%Instance1.Altitude% HKR,"Instances\"%Instance1.Name%,"Flags",0x00010001,%Instance1.Flags%

[MiniFilter.DriverFiles]

%DriverName%.sys

[SourceDisksFiles] passthrough.sys = 1,,

[SourceDisksNames] 1 = %DiskId1%,,,

[Strings]

Msft = "Microsoft Corporation" ServiceDescription = "WinDDK Mini-Filter Driver" ServiceName = "WinDDK"

DriverName = "WinDDK"

DiskId1 = "WinDDK Device Installation Disk"

DefaultInstance = "WinDDK Instance"

Instance1.Name = "WinDDK Instance" Instance1.Altitude = "370030"

Instance1.Flags = 0x0 ; Allow all attachments

# 第二种安装方式则是通过字写驱动加载工具实现，本人更推荐使用此方式安装，此种方式的原理同样是 向注册表中写出子健，但同时具备有启动与关闭驱动的功能，比INF安装更灵活易于使用，完整代码如下 所示；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include <Windows.h> #include <iostream> #include <winsvc.h> #include <winioctl.h>



// 安装MiniFinter

BOOL InstallDriver(const char lpszDriverName, const char lpszDriverPath, const char lpszAltitude)

{

char szTempStr[MAX\_PATH]; HKEY hKey;

DWORD dwData;

char szDriverImagePath[MAX\_PATH];

if (NULL == lpszDriverName || NULL == lpszDriverPath)

{

return FALSE;

}

// 得到完整的驱动路径

GetFullPathName(lpszDriverPath, MAX\_PATH, szDriverImagePath, NULL);

SC\_HANDLE hServiceMgr = NULL; // SCM管理器的句柄SC\_HANDLE hService = NULL; // NT驱动程序的服务句柄

// 打开服务控制管理器

hServiceMgr = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS); if (hServiceMgr == NULL)

{

// OpenSCManager失败CloseServiceHandle(hServiceMgr); return FALSE;

}

// OpenSCManager成功

// 创建驱动所对应的服务

hService = CreateService(hServiceMgr,

lpszDriverName, // 驱动程序的在注册表中的名字

lpszDriverName, // 注册表驱动程序的DisplayName 值SERVICE\_ALL\_ACCESS, // 加载驱动程序的访问权限SERVICE\_FILE\_SYSTEM\_DRIVER, // 表示加载的服务是文件系统驱动程序SERVICE\_DEMAND\_START, // 注册表驱动程序的Start 值SERVICE\_ERROR\_IGNORE, // 注册表驱动程序的ErrorControl 值szDriverImagePath, // 注册表驱动程序的ImagePath 值"FSFilter Activity Monitor",// 注册表驱动程序的Group 值

NULL,

"FltMgr", // 注册表驱动程序的DependOnService 值

NULL, NULL);

if (hService == NULL)

{

if (GetLastError() == ERROR\_SERVICE\_EXISTS)

{

// 服务创建失败，是由于服务已经创立过

}

else

{

}

}

CloseServiceHandle(hService); // 服 务 句 柄CloseServiceHandle(hServiceMgr); // SCM 句 柄return TRUE;

CloseServiceHandle(hService); // 服 务 句 柄CloseServiceHandle(hServiceMgr); // SCM 句 柄return FALSE;

CloseServiceHandle(hService); // 服务句柄



CloseServiceHandle(hServiceMgr); // SCM句柄

//---------------------------------------------------------------------------

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

// SYSTEM\\CurrentControlSet\\Services\\DriverName\\Instances 子键下的键值项

//---------------------------------------------------------------------------

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

strcpy(szTempStr, "SYSTEM\\CurrentControlSet\\Services\\"); strcat(szTempStr, lpszDriverName);

strcat(szTempStr, "\\Instances");

if (RegCreateKeyEx(HKEY\_LOCAL\_MACHINE, szTempStr, 0, "", TRUE, KEY\_ALL\_ACCESS, NULL, &hKey, (LPDWORD)&dwData) != ERROR\_SUCCESS)

{

return FALSE;

}

// 注册表驱动程序的DefaultInstance 值strcpy(szTempStr, lpszDriverName); strcat(szTempStr, " Instance");

if (RegSetValueEx(hKey, "DefaultInstance", 0, REG\_SZ, (CONST BYTE )szTempStr, (DWORD)strlen(szTempStr)) != ERROR\_SUCCESS)

{

return FALSE;

}

// 刷新注册表RegFlushKey(hKey); RegCloseKey(hKey);

//---------------------------------------------------------------------------

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

// SYSTEM\\CurrentControlSet\\Services\\DriverName\\Instances\\DriverName

Instance 子键下的键值项

//---------------------------------------------------------------------------

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

strcpy(szTempStr, "SYSTEM\\CurrentControlSet\\Services\\"); strcat(szTempStr, lpszDriverName);

strcat(szTempStr, "\\Instances\\"); strcat(szTempStr, lpszDriverName); strcat(szTempStr, " Instance");

if (RegCreateKeyEx(HKEY\_LOCAL\_MACHINE, szTempStr, 0, "", TRUE, KEY\_ALL\_ACCESS, NULL, &hKey, (LPDWORD)&dwData) != ERROR\_SUCCESS)

{

return FALSE;

}

// 注册表驱动程序的Altitude 值

strcpy(szTempStr, lpszAltitude);



if (RegSetValueEx(hKey, "Altitude", 0, REG\_SZ, (CONST BYTE )szTempStr, (DWORD)strlen(szTempStr)) != ERROR\_SUCCESS)

{

return FALSE;

}

// 注册表驱动程序的Flags 值

dwData = 0x0;

if (RegSetValueEx(hKey, "Flags", 0, REG\_DWORD, (CONST BYTE )&dwData, sizeof(DWORD)) != ERROR\_SUCCESS)

{

return FALSE;

}

// 刷新注册表RegFlushKey(hKey); RegCloseKey(hKey); return TRUE;

}

// 启动驱动

BOOL StartDriver(const char lpszDriverName)

{

SC\_HANDLE schManager; SC\_HANDLE schService; SERVICE\_STATUS svcStatus;

if (NULL == lpszDriverName)

{

return FALSE;

}

schManager = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS); if (NULL == schManager)

{

CloseServiceHandle(schManager); return FALSE;

}

schService = OpenService(schManager, lpszDriverName, SERVICE\_ALL\_ACCESS); if (NULL == schService)

{

CloseServiceHandle(schService); CloseServiceHandle(schManager); return FALSE;

}

if (!StartService(schService, 0, NULL))

{

CloseServiceHandle(schService); CloseServiceHandle(schManager);

if (GetLastError() == ERROR\_SERVICE\_ALREADY\_RUNNING)

{

// 服务已经开启

return TRUE;

}

return FALSE;

}



CloseServiceHandle(schService); CloseServiceHandle(schManager);

return TRUE;

}

// 关闭驱动

BOOL StopDriver(const char lpszDriverName)

{

SC\_HANDLE schManager; SC\_HANDLE schService; SERVICE\_STATUS svcStatus; bool bStopped = false;

schManager = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS); if (NULL == schManager)

{

return FALSE;

}

schService = OpenService(schManager, lpszDriverName, SERVICE\_ALL\_ACCESS); if (NULL == schService)

{

CloseServiceHandle(schManager); return FALSE;

}

if (!ControlService(schService, SERVICE\_CONTROL\_STOP, &svcStatus) && (svcStatus.dwCurrentState != SERVICE\_STOPPED))

{

CloseServiceHandle(schService); CloseServiceHandle(schManager); return FALSE;

}

CloseServiceHandle(schService); CloseServiceHandle(schManager);

return TRUE;

}

// 删除驱动

BOOL DeleteDriver(const char lpszDriverName)

{

SC\_HANDLE schManager; SC\_HANDLE schService; SERVICE\_STATUS svcStatus;

schManager = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS); if (NULL == schManager)

{

return FALSE;

}

schService = OpenService(schManager, lpszDriverName, SERVICE\_ALL\_ACCESS); if (NULL == schService)

{

CloseServiceHandle(schManager);



return FALSE;

}

ControlService(schService, SERVICE\_CONTROL\_STOP, &svcStatus); if (!DeleteService(schService))

{

CloseServiceHandle(schService); CloseServiceHandle(schManager); return FALSE;

}

CloseServiceHandle(schService); CloseServiceHandle(schManager);

return TRUE;

}

int main(int argc, char argv[])

{

InstallDriver("minifilter", ".\\WinDDK.sys", "225864");

while (1)

{

char str[20] = "\0";

printf("请输入命令: "); gets(str);

if (strcmp(str, "start") == 0)

{

printf("[ ] 启动驱动 \n"); StartDriver("minifilter");

}

if (strcmp(str, "stop") == 0)

{

printf("[-] 关闭驱动 \n"); StopDriver("minifilter");

}

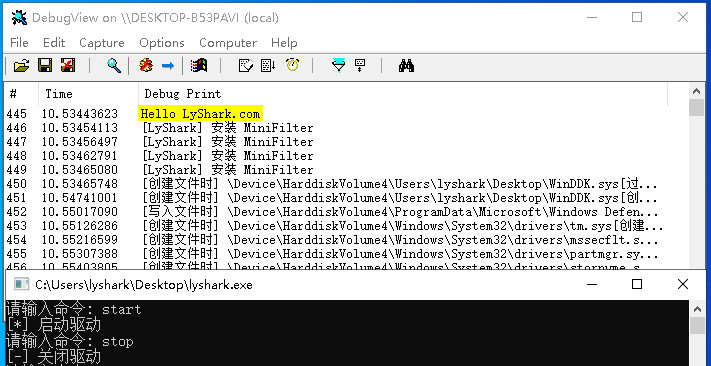
}

return 0;

}

# 至此分别编译驱动程序，以及应用层下的安装程序，并将两者放入到同一目录下，运行客户端程序

lyshark.exe 并输入 start 启动驱动，输入 stop 则是关闭，启动后会看到如下信息；



# 这里简单介绍一下如何摘除微过滤驱动回调函数，其实摘除回调的方法有多种，常用的第一种通过向过 滤驱动中写出一个返回命令让其不被执行从而实现绕过，另一种是找到回调函数并替换为我们自己的回 调，而在自己的回调中什么也不做，这里以第二种方法为例，实现替换的代码可以写成如下案例；



#include <fltKernel.h>

// 设置默认回调

NTSTATUS DriverDefaultHandle(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

NTSTATUS status = STATUS\_SUCCESS;

pIrp->IoStatus.Status = status; pIrp->IoStatus.Information = 0;

IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);

return status;

}

// ------------------------------------------------------------------------

// 消息转向函数

// ------------------------------------------------------------------------

// 自定义回调 消息处理前

FLT\_PREOP\_CALLBACK\_STATUS MyMiniFilterPreOperation(\_Inout\_ PFLT\_CALLBACK\_DATA Data,\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,\_Flt\_CompletionContext\_Outptr\_ PVOID

CompletionContext)

{

UNREFERENCED\_PARAMETER(FltObjects);

UNREFERENCED\_PARAMETER(CompletionContext);

return FLT\_PREOP\_SUCCESS\_WITH\_CALLBACK;

}

// 自定义回调 消息处理后

FLT\_POSTOP\_CALLBACK\_STATUS MyMiniFilterPostOperation(\_Inout\_ PFLT\_CALLBACK\_DATA Data,\_In\_ PCFLT\_RELATED\_OBJECTS FltObjects,\_In\_opt\_ PVOID CompletionContext,\_In\_ FLT\_POST\_OPERATION\_FLAGS Flags)

{

UNREFERENCED\_PARAMETER(Data);

UNREFERENCED\_PARAMETER(FltObjects);



UNREFERENCED\_PARAMETER(CompletionContext); UNREFERENCED\_PARAMETER(Flags);

return FLT\_POSTOP\_FINISHED\_PROCESSING;

}

// 替换回调函数

NTSTATUS RemoveCallback(PFLT\_FILTER pFilter)

{

LONG lOperationsOffset = 0;

PFLT\_OPERATION\_REGISTRATION pFltOperationRegistration = NULL;

// 遍历过滤器Filter

pFltOperationRegistration = (PFLT\_OPERATION\_REGISTRATION)( (PVOID ) ((PUCHAR)pFilter + lOperationsOffset));

try

{

while (IRP\_MJ\_OPERATION\_END != pFltOperationRegistration->MajorFunction)

{

if (IRP\_MJ\_MAXIMUM\_FUNCTION > pFltOperationRegistration-

>MajorFunction)

{

// 替换回调函数

pFltOperationRegistration->PreOperation = MyMiniFilterPreOperation;

pFltOperationRegistration->PostOperation = MyMiniFilterPostOperation;

}

pFltOperationRegistration = (PFLT\_OPERATION\_REGISTRATION) ((PUCHAR)pFltOperationRegistration + sizeof(FLT\_OPERATION\_REGISTRATION));

}

}

except (EXCEPTION\_EXECUTE\_HANDLER)

{

return STATUS\_SUCCESS;

}

return STATUS\_SUCCESS;

}

VOID DriverUnload(PDRIVER\_OBJECT pDriverObject)

{

return STATUS\_SUCCESS;

}

NTSTATUS DriverEntry(PDRIVER\_OBJECT pDriverObject, PUNICODE\_STRING pRegPath)

{

NTSTATUS status = STATUS\_SUCCESS;

pDriverObject->DriverUnload = DriverUnload;

for (ULONG i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++)

{

pDriverObject->MajorFunction[i] = DriverDefaultHandle;

}

return status;

}

