# 驱动与应用程序的通信是非常有必要的，内核中执行代码后需要将其动态显示给应用层，但驱动程序与 应用层毕竟不在一个地址空间内，为了实现内核与应用层数据交互则必须有通信的方法，微软为我们提 供了三种通信方式，如下先来介绍通过 ReadFile 系列函数实现的通信模式。

长话短说，不说没用的概念，首先系统中 支持 的 通信模式 可以总结为三种。缓冲区方式读写(DO\_BUFFERED\_IO)

直接方式读写(DO\_DIRECT\_IO) 其他方式读写

而通过 ReadFile,WriteFile 系列函数实现的通信机制则属于 缓冲区通信 模式，在该模式下操作系统会将应用层中的 数据复制 到内核中，此时应用层调用 ReadFile,WriteFile 函数进行读写时，在驱动内会

IRP\_MJ\_READ

# 自动触发

类处理。

与 IRP\_MJ\_WRITE 这两个派遣函数，在派遣函数内则可以对收到的数据进行各

首先需要实现初始化各类派遣函数这么一个案例，如下代码则是通用的一种初始化派遣函数的基本框 架，分别处理了 IRP\_MJ\_CREATE 创建派遣，以及 IRP\_MJ\_CLOSE 关闭的派遣，此外函数

DriverDefaultHandle 的作用时初始化其他派遣用的，也就是将除去 CREATE/CLOSE 这两个派遣之外，其他的全部赋值成初始值的意思，当然不增加此段代码也是无妨，并不影响代码的实际执行。

#include <ntifs.h>

// 卸载驱动执行

VOID UnDriver(PDRIVER\_OBJECT pDriver)

{

PDEVICE\_OBJECT pDev;

备对象

UNICODE\_STRING SymLinkName; symLinkName

pDev = pDriver->DeviceObject;

IoDeleteDevice(pDev);

IoDeleteDevice用于删除设备

RtlInitUnicodeString(&SymLinkName, L"\\??\\LySharkDriver");

将symLinkName定义成需要删除的符号链接名称

IoDeleteSymbolicLink(&SymLinkName);

IoDeleteSymbolicLink删除符号链接

DbgPrint("驱动卸载完毕...");

}

// 用来取得要删除设

// 局部变量

// 调用

// 初始化字符串

// 调用

// 创建设备连接

// LyShark.com

NTSTATUS CreateDriverObject(IN PDRIVER\_OBJECT pDriver)

{

NTSTATUS Status; PDEVICE\_OBJECT pDevObj;

UNICODE\_STRING DriverName;

UNICODE\_STRING SymLinkName;

// 创建设备名称字符串

RtlInitUnicodeString(&DriverName, L"\\Device\\LySharkDriver");

Status = IoCreateDevice(pDriver, 0, &DriverName, FILE\_DEVICE\_UNKNOWN, 0, TRUE, &pDevObj);

// 指定通信方式为缓冲区

pDevObj->Flags |= DO\_BUFFERED\_IO;

// 创建符号链接

RtlInitUnicodeString(&SymLinkName, L"\\??\\LySharkDriver"); Status = IoCreateSymbolicLink(&SymLinkName, &DriverName); return STATUS\_SUCCESS;

}

// 创建回调函数

NTSTATUS DispatchCreate(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

pIrp->IoStatus.Status = STATUS\_SUCCESS; // 返回成功

DbgPrint(" 派 遣 函 数 IRP\_MJ\_CREATE 执 行 \n"); IoCompleteRequest(pIrp, IO\_NO\_INCREMENT); // 指示完成此IRP return STATUS\_SUCCESS; // 返回成功

}

// 关闭回调函数

NTSTATUS DispatchClose(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

pIrp->IoStatus.Status = STATUS\_SUCCESS; // 返回成功

DbgPrint(" 派 遣 函 数 IRP\_MJ\_CLOSE 执 行 \n"); IoCompleteRequest(pIrp, IO\_NO\_INCREMENT); // 指示完成此IRP return STATUS\_SUCCESS; // 返回成功

}

// 默认派遣函数

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;

}

// 入口函数

// By: LyShark

NTSTATUS DriverEntry(PDRIVER\_OBJECT pDriver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("hello lyshark \n");

// 调用创建设备

CreateDriverObject(pDriver);

pDriver->DriverUnload = UnDriver; // 卸载函数pDriver->MajorFunction[IRP\_MJ\_CREATE] = DispatchCreate; // 创建派遣函数pDriver->MajorFunction[IRP\_MJ\_CLOSE] = DispatchClose; // 关闭派遣函数

// 初始化其他派遣

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

{

DbgPrint("初始化派遣: %d \n", i);

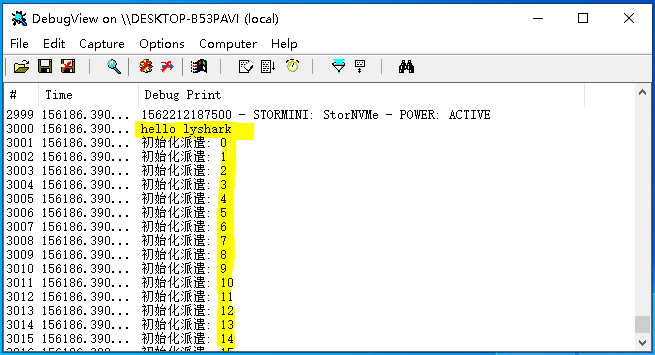
pDriver->MajorFunction[i] = DriverDefaultHandle;

}

DbgPrint("驱动加载完成..."); return STATUS\_SUCCESS;

}

# 代码运行效果如下：



通用框架有了，接下来就是让该驱动支持使用 ReadWrite 的方式实现通信，首先我们需要在

DriverEntry 处增加两个派遣处理函数的初始化。

// 入口函数

// By: LyShark

NTSTATUS DriverEntry(PDRIVER\_OBJECT pDriver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("hello lyshark \n");

// 调用创建设备

CreateDriverObject(pDriver);

// 初始化其他派遣

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

{

DbgPrint("初始化派遣: %d \n", i);

pDriver->MajorFunction[i] = DriverDefaultHandle;

}

pDriver->DriverUnload = UnDriver;

pDriver->MajorFunction[IRP\_MJ\_CREATE] = DispatchCreate; pDriver->MajorFunction[IRP\_MJ\_CLOSE] = DispatchClose;

// 卸载函数

// 创建派遣函数

// 关闭派遣函数

// 增加派遣处理

pDriver->MajorFunction[IRP\_MJ\_READ] = DispatchRead; pDriver->MajorFunction[IRP\_MJ\_WRITE] = DispatchWrite;

// 读取派遣函数

// 写入派遣函数

DbgPrint("驱动加载完成...");

return STATUS\_SUCCESS;

}

接着，我们需要分别实现这两个派遣处理函数，如下 DispatchRead 负责读取时触发，与之对应

DispatchWrite 负责写入触发。

# 引言：

对于读取请求 I/O管理器 分配一个与用户模式的缓冲区大小相同的系统缓冲区 SystemBuffer ，当完成请求时I/O管理器将驱动程序已经提供的数据从系统缓冲区复制到用户缓冲区。

对于写入请求，会分配一个系统缓冲区并将 SystemBuffer 设置为地址，用户缓冲区的内容会被复制到系统缓冲区，但是不设置 UserBuffer 缓冲。

通过 IoGetCurrentIrpStackLocation(pIrp) 接收读写请求长度，偏移等基本参数，

AssociatedIrp.SystemBuffer 则是读写缓冲区， IoStatus.Information 是输出缓冲字节数，

Parameters.Read.Length 是读取写入的字节数。



// 读取回调函数

NTSTATUS DispatchRead(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

NTSTATUS Status = STATUS\_SUCCESS;

PIO\_STACK\_LOCATION Stack = IoGetCurrentIrpStackLocation(pIrp); ULONG ulReadLength = Stack->Parameters.Read.Length;

char szBuf[128] = "hello lyshark";

pIrp->IoStatus.Status = Status;

pIrp->IoStatus.Information = ulReadLength;

DbgPrint("读取长度：%d \n", ulReadLength);

// 取出字符串前5个字节返回给R3层

memcpy(pIrp->AssociatedIrp.SystemBuffer, szBuf, ulReadLength);

IoCompleteRequest(pIrp, IO\_NO\_INCREMENT); return Status;

}

// 接收传入回调函数

// By: LyShark

NTSTATUS DispatchWrite(struct \_DEVICE\_OBJECT DeviceObject, struct \_IRP Irp)

{

NTSTATUS Status = STATUS\_SUCCESS;

PIO\_STACK\_LOCATION Stack = IoGetCurrentIrpStackLocation(Irp); ULONG ulWriteLength = Stack->Parameters.Write.Length;

PVOID ulWriteData = Irp->AssociatedIrp.SystemBuffer;

// 输出传入字符串

DbgPrint("传入长度: %d 传入数据: %s \n", ulWriteLength, ulWriteData);

IoCompleteRequest(Irp, IO\_NO\_INCREMENT); return Status;

}

如上部分都是在讲解驱动层面的读写派遣，应用层还没有介绍，在应用层我们只需要调用 ReadFile 函 数当调用该函数时驱动中会使用 DispatchRead 派遣例程来处理这个请求，同理调用 WriteFile 函数则触发的是 DispatchWrite 派遣例程。

# 我们首先从内核中读出前五个字节并放入缓冲区内，输出该缓冲区内的数据，然后在调用写入，将

hello lyshark 写回到内核里里面，这段代码可以这样来写。



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

int main(int argc, char argv[])

{

HANDLE hDevice = CreateFileA("\\\\.\\LySharkDriver", GENERIC\_READ | GENERIC\_WRITE, 0,

NULL, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, NULL);

if (hDevice == INVALID\_HANDLE\_VALUE)

{

CloseHandle(hDevice); return 0;

}

// 从内核读取数据到本地

char buffer[128] = { 0 }; ULONG length;

// 读入到buffer长度为5

// By:lyshark.com

ReadFile(hDevice, buffer, 5, &length, 0); for (int i = 0; i < (int)length; i++)

{

printf("读取字节: %c", buffer[i]);

}

// 写入数据到内核

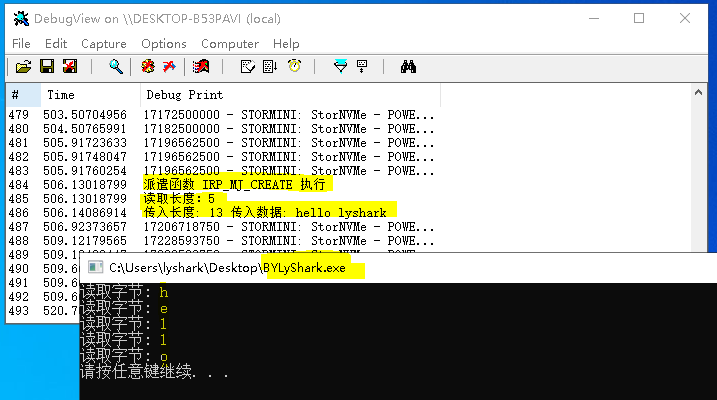
char write\_buffer[128] = "hello lyshark"; ULONG write\_length;

WriteFile(hDevice, write\_buffer, strlen(write\_buffer), &write\_length, 0);

system("pause"); CloseHandle(hDevice); return 0;

}

# 使用驱动工具安装我们的驱动，然后运行该应用层程序，实现通信，效果如下所示：



本书作者： 王瑞 (LyShark)

作者邮箱： m [e@lyshark.com](mailto:e@lyshark.com)

作者博客： h ttps://lyshark.cnblogs.com

团队首页： w ww.lyshark.com