

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，在驱动内会自动触发 IRP_MJ_READ 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类处理。

首先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; // 用来取得要删除设备对象
    UNICODE_STRING SymLinkName; // 局部变量symLinkName
    pDev = pDriver->DeviceObject;
    IoDeleteDevice(pDev); // 调用IoDeleteDevice用于
    删除设备
    RtlInitUnicodeString(&SymLinkName, L"\\??\\LySharkDriver"); // 初始化字符串将
    symLinkName定义成需要删除的符号链接名称
    IoDeleteSymbolicLink(&SymLinkName); // 调用
    IoDeleteSymbolicLink删除符号链接
    DbgPrint("驱动卸载完毕...");
}

// 创建设备连接
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
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;
    }
}

```

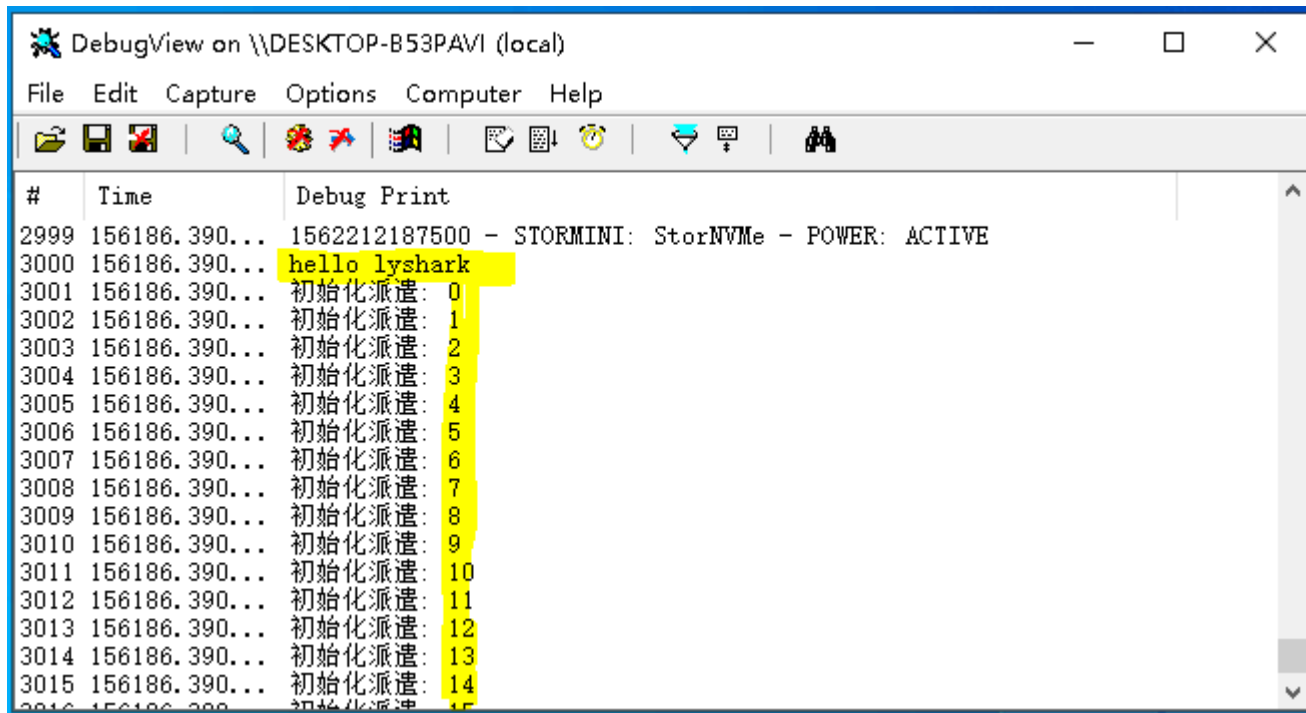
```

    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
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; // 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

    // 增加派遣处理
    pDriver->MajorFunction[IRP_MJ_READ] = DispatchRead; // 读取派遣函数
    pDriver->MajorFunction[IRP_MJ_WRITE] = DispatchWrite; // 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：

DebugView on \\DESKTOP-B53PAVI (local)

File Edit Capture Options Computer Help

Dispatch Table

#	Time	Debug Print
479	503.50704956	17172500000 - STORMINI: StorNVMe - POWE...
480	504.50765991	17182500000 - STORMINI: StorNVMe - POWE...
481	505.91723633	17196562500 - STORMINI: StorNVMe - POWE...
482	505.91748047	17196562500 - STORMINI: StorNVMe - POWE...
483	505.91760254	17196562500 - STORMINI: StorNVMe - POWE...
484	506.13018799	派遣函数 IRP_MJ_CREATE 执行
485	506.13018799	读取长度: 5
486	506.14086914	传入长度: 13 传入数据: hello lyshark
487	506.92373657	17206718750 - STORMINI: StorNVMe - POWE...
488	509.12179565	17228593750 - STORMINI: StorNVMe - POWE...
489	509.12179565	17228593750 - STORMINI: StorNVMe - POWE...
490	509.6	C:\Users\lyshark\Desktop\BYLyShark.exe
491	509.6	读取字节: h
492	509.6	读取字节: e
493	520.7	读取字节: l
		读取字节: l
		读取字节: o
		请按任意键继续. . .