在前几篇文章中给大家具体解释了驱动与应用层之间正向通信的一些经典案例，本章将继续学习驱动通信，不过这次我们学习的是通过运用 `Async` 异步模式实现的反向通信，反向通信机制在开发中时常被用到，例如一个杀毒软件如果监控到有异常进程运行或有异常注册表被改写后，该驱动需要主动的通知应用层进程让其知道，这就需要用到驱动反向通信的相关知识点，如下将循序渐进的实现一个反向通信案例。

在开始学习Async反向通信之前先来研究一个Sync正向通信案例，不论是正向反向通信其在通信模式上与 `《通过ReadFile与内核层通信》` 所介绍的通信模式基本一致，都是通过 `ReadFile` 触发驱动中的 `IRP_MJ_READ` 读取派遣，唯一的区别是在传输数据时使用了 `MmGetSystemAddressForMdl` 方式，它将给定 `MDL` 描述的物理页面映射到系统空间，并调用 `RtlCopyMemory()` 将全局字符串复制到这个空间内，这样客户端就可以循环读取内核传出的数据。

我们来看驱动端代码是如何实现的这个功能，代码并没有什么特殊的无法理解的点，只是需要注意我们在驱动入口调用 `IoCreateDevice()` 时传入了第二个参数 `FILE_DEVICE_EXTENSION`，该参数的作用是，创建设备时，指定设备扩展内存的大小，传一个值进去，就会给设备分配一块非页面内存。

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

// 保存一段非分页内存,用于给全局变量使用
#define FILE_DEVICE_EXTENSION 4096

// 定义全局字符串
static int global_count = 0;
static char global_char[5][128] = { 0 };

// 驱动绑定默认派遣函数
NTSTATUS _DefaultDispatch(PDEVICE_OBJECT _pDeviceObject, PIRP _pIrp)
{
    _pIrp->IoStatus.Status = STATUS_NOT_SUPPORTED;
    _pIrp->IoStatus.Information = 0;
    IoCompleteRequest(_pIrp, IO_NO_INCREMENT);
    return _pIrp->IoStatus.Status;
}

// 驱动创建后触发
NTSTATUS _SyncCreateCloseDispatch(PDEVICE_OBJECT _pDevcieObject, PIRP _pIrp)
{
    _pIrp->IoStatus.Status = STATUS_SUCCESS;
    _pIrp->IoStatus.Information = 0;
    IoCompleteRequest(_pIrp, IO_NO_INCREMENT);
    return _pIrp->IoStatus.Status;
}

// 应用层读数据后触发
NTSTATUS _SyncReadDispatch(PDEVICE_OBJECT _pDeviceObject, PIRP _pIrp)
{
    NTSTATUS status = STATUS_SUCCESS;
    PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(_pIrp);
    PVOID pBuffer = NULL;
    ULONG uBufferLen = 0;

    do
    {
```

```
        // 读写请求使用的是直接I/O方式
        pBuffer = MmGetSystemAddressForMdl(_pIrp->MdlAddress);
        if (pBuffer == NULL)
        {
            status = STATUS_UNSUCCESSFUL;
            break;
        }
        uBufferLen = pIrpStack->Parameters.Read.Length;
        DbgPrint("读字节长度: %d \n", uBufferLen);

        // 最大支持20字节读请求
        uBufferLen = uBufferLen >= 20 ? 20 : uBufferLen;

        // 输出五次字符串
        if (global_count < 5)
        {
            RtlCopyMemory(pBuffer, global_char[global_count], uBufferLen);
            global_count = global_count + 1;
        }

    } while (FALSE);

    // 填写返回状态及返回大小
    _pIrp->IoStatus.Status = status;
    _pIrp->IoStatus.Information = uBufferLen;

    // 完成IRP
    IoCompleteRequest(_pIrp, IO_NO_INCREMENT);
    return status;
}

// 卸载驱动
VOID _UnloadDispatch(PDRIVER_OBJECT _pDriverObject)
{
    // 删除创建的设备
    UNICODE_STRING  Win32DeviceName;
    RtlInitUnicodeString(&Win32DeviceName, L"\\DosDevices\\LySharkSync");
    IoDeleteDevice(_pDriverObject->DeviceObject);
}

// 驱动入口
NTSTATUS DriverEntry(PDRIVER_OBJECT _pDriverObject, PUNICODE_STRING _pRegistryPath)
{
    UNICODE_STRING DeviceName, Win32DeivceName;
    PDEVICE_OBJECT pDeviceObject = NULL;
    NTSTATUS status;
    HANDLE hThread;
    OBJECT_ATTRIBUTES ObjectAttributes;

    // 设置符号名
    RtlInitUnicodeString(&DeviceName, L"\\Device\\LySharkSync");
    RtlInitUnicodeString(&Win32DeivceName, L"\\DosDevices\\LySharkSync");
```

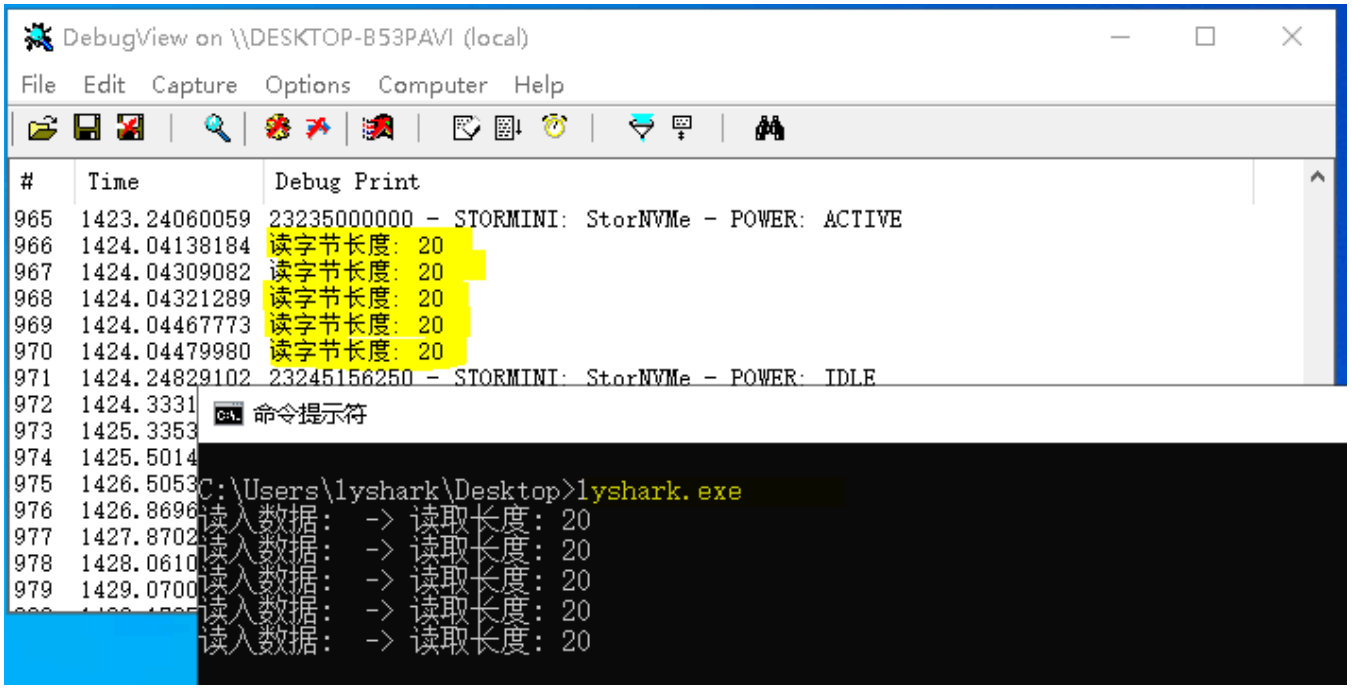
```
    // 循环初始化IRP函数
    for (ULONG i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        _pDriverObject->MajorFunction[i] = _DefaultDispatch;
    }

    // 再次覆盖派遣函数
    _pDriverObject->MajorFunction[IRP_MJ_CREATE] = _SyncCreateCloseDispatch;
    _pDriverObject->MajorFunction[IRP_MJ_CLOSE] = _SyncCreateCloseDispatch;
    _pDriverObject->MajorFunction[IRP_MJ_READ] = _SyncReadDispatch;
    _pDriverObject->DriverUnload = _UnloadDispatch;

    // 分配一个自定义扩展 大小为sizeof（DEVEXT）
    // By: LyShark.com
    status = IoCreateDevice(_pDriverObject, sizeof(FILE_DEVICE_EXTENSION), &DeviceName,
FILE_DEVICE_UNKNOWN, 0, FALSE, &pDeviceObject);
    if (!NT_SUCCESS(status))
        return status;

    if (!pDeviceObject)
        return STATUS_UNEXPECTED_IO_ERROR;

    // 为全局变量赋值
    strcpy(global_char[0], "hi,lyshark A");
    strcpy(global_char[1], "hi,lyshark B");
    strcpy(global_char[2], "hi,lyshark C");
    strcpy(global_char[3], "hi,lyshark D");
    strcpy(global_char[4], "hi,lyshark E");

    // 指定读写方式为 直接I/O MDL模式
    pDeviceObject->Flags |= DO_DIRECT_IO;

    // 数据传输时地址校验大小
    pDeviceObject->AlignmentRequirement = FILE_WORD_ALIGNMENT;
    status = IoCreateSymbolicLink(&Win32DeivceName, &DeviceName);

    pDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
    return STATUS_SUCCESS;
}
```

对于应用层来说并没有什么特别的，同样调用`ReadFile`读取内核中的参数，同样for循环读取五次，代码如下：

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

int main(int argc, char *argv[])
{
    HANDLE hFile;
    char Buffer[10] = { 0 };
    DWORD dwRet = 0;
    BOOL bRet;
```

```
    hFile = CreateFileA("\\\\.\\LySharkSync", GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
        return 0;

    for (int x = 0; x < 5; x++)
    {
        bRet = ReadFile(hFile, Buffer, 20, &dwRet, NULL);
        if (!bRet)
        {
            CloseHandle(hFile);
            return 0;
        }
        printf("读入数据: %s -> 读取长度: %d \n", Buffer, dwRet);
    }
    return 0;
}
```

这段代码运行效果如下:



与同步模式不同，`异步模式`虽然同样使用`ReadFile`实现通信，但在通信中引入了`Event`事件通知机制，这也是异步与同步最大的区别所在，用户层可以分别创建多个Event事件，等待内核依次做出相应并最终一并返回。

首先驱动内定义了`_DeviceExtension`自定义接口，该接口用于保存此次事件所对应的`Irp`以及其所对应的DPC时间等。

异步分发函数`_AsyncReadDispatch`同样是被`IRP_MJ_READ`派遣函数触发的，触发后其内部会首先`IoGetCurrentIrpStackLocation`得到当前IRP的堆栈信息，然后设置`IoMarkIrpPending()`并最终将该IRP通过`InsertTailList()`插入到IRP链表内等待被处理。

- IoMarkIrpPending
    - 用于标记指定的IRP，标志着某个驱动的分发例程（分发函数）因需要被其他的驱动程序进一步处理最终返回STATUS_PENDING状态。

函数`_CustomDpc`则是定时器内部要执行的具体操作，在`DriverEntry`驱动入口处做了如下初始化，初始化了链表，并初始化了一个定时器，最后启动这个定时器每隔1秒都会执行一次`_CustomDpc`如果我们的IRP链表内`IsListEmpty()`检测存在数据，则会主动拷贝内存`RtlCopyMemory`并推送到应用层。

```
// 初始化IRP链表
InitializeListHead(&pDevExt->IrpList);
// 初始化定时器
KeInitializeTimer(&(pDevExt->timer));
// 初始化DPC pDevExt是传给_CustomDpc函数的参数
KeInitializeDpc(&pDevExt->dpc, (PKDEFERRED_ROUTINE)_CustomDpc, pDevExt);

// 设置定时时间位1s
pDevExt->liDueTime = RtlConvertLongToLargeInteger(-10000000);
// 启动定时器
KeSetTimer(&pDevExt->timer, pDevExt->liDueTime, &pDevExt->dpc);
```

驱动层完成代码如下所示：

```
#include <ntddk.h>

// 自定义接口扩展
typedef struct _DeviceExtension
{
    LIST_ENTRY IrpList;
    KTIMER timer;
    LARGE_INTEGER liDueTime;
    KDPC dpc;
}DEV_EXT, *PDEV_EXT;

// 默认派遣函数
NTSTATUS _DefaultDispatch(PDEVICE_OBJECT _pDeviceObject, PIRP _pIrp)
{
    _pIrp->IoStatus.Status = STATUS_NOT_SUPPORTED;
    _pIrp->IoStatus.Information = 0;
    IoCompleteRequest(_pIrp, IO_NO_INCREMENT);
    return _pIrp->IoStatus.Status;
}

// 创建派遣函数
NTSTATUS _AsyncCreateCloseDispatch(PDEVICE_OBJECT _pDevcieObject, PIRP _pIrp)
{
    _pIrp->IoStatus.Status = STATUS_SUCCESS;
    _pIrp->IoStatus.Information = 0;
    IoCompleteRequest(_pIrp, IO_NO_INCREMENT);
    return  _pIrp->IoStatus.Status;
}

// 读取派遣函数
NTSTATUS _AsyncReadDispatch(PDEVICE_OBJECT _pDeviceObject, PIRP _pIrp)
{
    NTSTATUS status;
    PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(_pIrp);
```

```
    PDEV_EXT pDevExt = (PDEV_EXT)_pDeviceObject->DeviceExtension;

    IoMarkIrpPending(_pIrp);

    // 将IRP插入自定义链表中插入的是ListEntry
    InsertTailList(&pDevExt->IrpList, &_pIrp->Tail.Overlay.ListEntry);

    // 返回pending 主要返回给I/O管理器的值必须和IRP的Pending标志位一致
    // By: LyShark.com
    // 即调用iomarkirppending和返回值要一致
    return STATUS_PENDING;
}

// DPC线程
VOID _CustomDpc(PKDPC Dpc, PVOID DeferredContext, PVOID SystemArgument1, PVOID
SystemArgument2)
{
    PIRP pIrp;
    PDEV_EXT pDevExt = (PDEV_EXT)DeferredContext;
    PVOID pBuffer = NULL;
    ULONG uBufferLen = 0;
    PIO_STACK_LOCATION pIrpStack = NULL;

    do
    {
        if (!pDevExt)
        {
            break;
        }

        // 检查尾端IRP链表是否为空 为空则跳出
        if (IsListEmpty(&pDevExt->IrpList))
        {
            break;
        }

        // 从IRP链表中取出一个IRP并完成该IRP 取出的是ListEntry的地址
        PLIST_ENTRY pListEntry = (PLIST_ENTRY)RemoveHeadList(&pDevExt->IrpList);
        if (!pListEntry)
            break;

        pIrp = (PIRP)CONTAINING_RECORD(pListEntry, IRP, Tail.Overlay.ListEntry);
        pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

        DbgPrint("当前DPC Irp: 0x%x\n", pIrp);

        // 驱动程序的读写方式位直接I/O
        pBuffer = MmGetSystemAddressForMdl(pIrp->MdlAddress);
        if (pBuffer == NULL)
        {
            pIrp->IoStatus.Status = STATUS_UNSUCCESSFUL;
            pIrp->IoStatus.Information = 0;
            IoCompleteRequest(pIrp, IO_NO_INCREMENT);
```

```
            break;
        }
        uBufferLen = pIrpStack->Parameters.Read.Length;
        DbgPrint("读取DPC长度: %d\n", uBufferLen);

        // 支持5字节以下的读请求
        uBufferLen = uBufferLen > 13 ? 13 : uBufferLen;

        // 复制请求内容
        RtlCopyMemory(pBuffer, "hello lyshark", uBufferLen);

        pIrp->IoStatus.Status = STATUS_SUCCESS;
        pIrp->IoStatus.Information = uBufferLen;

        // 完成该IRP
        IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    } while (FALSE);

    // 重新设置定时器
    KeSetTimer(&pDevExt->timer, pDevExt->liDueTime, &pDevExt->dpc);
}

// 卸载驱动
VOID _UnloadDispatch(PDRIVER_OBJECT _pDriverObject)
{
    UNICODE_STRING Win32DeviceName;
    PDEV_EXT pDevExt = (PDEV_EXT)_pDriverObject->DeviceObject->DeviceExtension;

    RtlInitUnicodeString(&Win32DeviceName, L"\\DosDevices\\LySharkAsync");

    // 删除定时器
    // LyShark
    KeCancelTimer(&pDevExt->timer);
    // 删除创建的设备
    IoDeleteDevice(_pDriverObject->DeviceObject);
}

// 驱动入口
NTSTATUS DriverEntry(PDRIVER_OBJECT _pDriverObject, PUNICODE_STRING _pRegistryPath)
{
    UNICODE_STRING DeviceName, Win32DeivceName;
    PDEVICE_OBJECT pDeviceObject = NULL;
    NTSTATUS status;
    PDEV_EXT pDevExt = NULL;
    HANDLE hThread;
    OBJECT_ATTRIBUTES ObjectAttributes;
    CLIENT_ID CID;

    RtlInitUnicodeString(&DeviceName, L"\\Device\\LySharkAsync");
    RtlInitUnicodeString(&Win32DeivceName, L"\\DosDevices\\LySharkAsync");

    for (ULONG i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++)
```

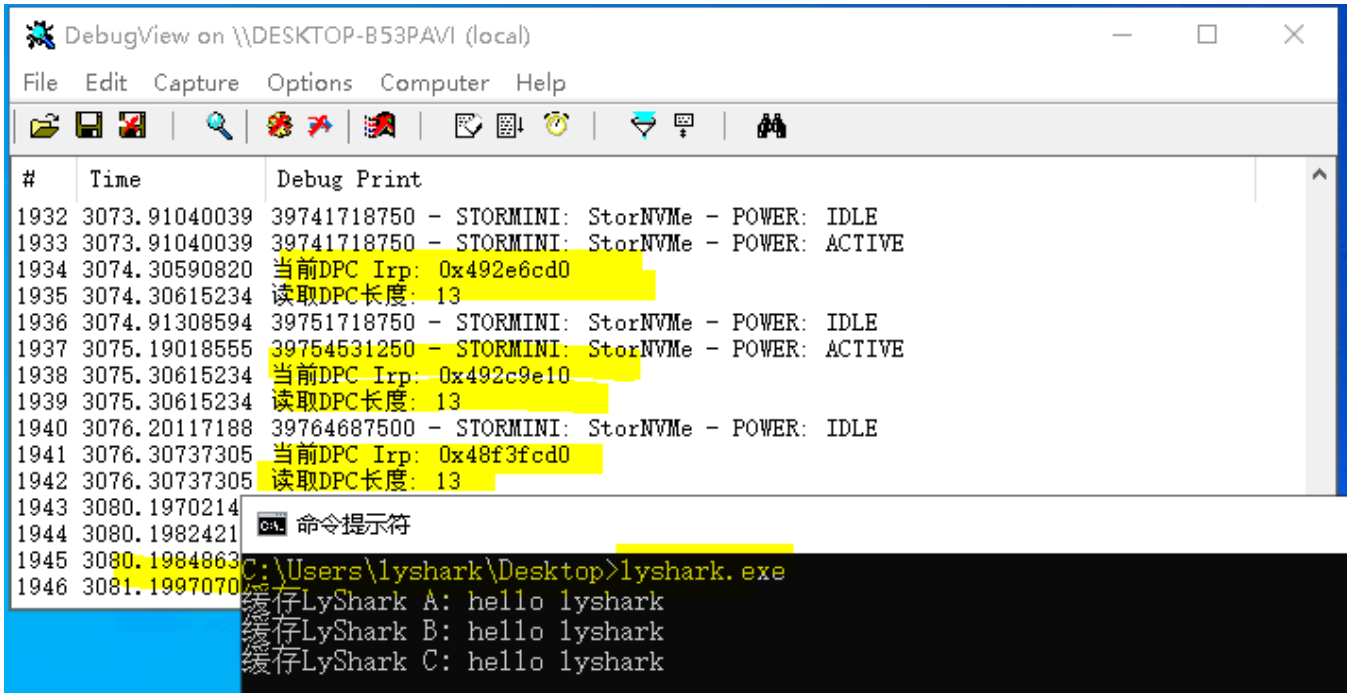
```
    {
        _pDriverObject->MajorFunction[i] = _DefaultDispatch;
    }

    _pDriverObject->MajorFunction[IRP_MJ_CREATE] = _AsyncCreateCloseDispatch;
    _pDriverObject->MajorFunction[IRP_MJ_CLOSE] = _AsyncCreateCloseDispatch;
    _pDriverObject->MajorFunction[IRP_MJ_READ] = _AsyncReadDispatch;
    _pDriverObject->DriverUnload = _UnloadDispatch;

    // 分配自定义扩展
    status = IoCreateDevice(_pDriverObject, sizeof(DEV_EXT), &DeviceName,
FILE_DEVICE_UNKNOWN, 0, FALSE, &pDeviceObject);
    if (!NT_SUCCESS(status))
        return status;
    if (!pDeviceObject)
        return STATUS_UNEXPECTED_IO_ERROR;

    pDeviceObject->Flags |= DO_DIRECT_IO;
    pDeviceObject->AlignmentRequirement = FILE_WORD_ALIGNMENT;
    status = IoCreateSymbolicLink(&Win32DeivceName, &DeviceName);

    pDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
    pDevExt = (PDEV_EXT)pDeviceObject->DeviceExtension;

    // 初始化IRP链表
    InitializeListHead(&pDevExt->IrpList);
    // 初始化定时器
    KeInitializeTimer(&(pDevExt->timer));
    // 初始化DPC pDevExt是传给_CustomDpc函数的参数
    KeInitializeDpc(&pDevExt->dpc, (PKDEFERRED_ROUTINE)_CustomDpc, pDevExt);

    // 设置定时时间位1s
    pDevExt->liDueTime = RtlConvertLongToLargeInteger(-10000000);
    // 启动定时器
    KeSetTimer(&pDevExt->timer, pDevExt->liDueTime, &pDevExt->dpc);

    return STATUS_SUCCESS;
}
```

驱动层说完了，接下来是应用层，对于应用层来说，需要使用`CreateEvent`打开通知事件，或者叫做事件对象，然后当有通知时，则直接使用`ReadFile`读取对应的缓冲区，当所有读取全部结束`WaitForMultipleObjects`等待结束即输出结果。

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

int main(int argc, char *argv[])
{
    HANDLE hFile;
    char Buffer[3][32] = { 0 };
    DWORD dwRet[3] = { 0 };
    OVERLAPPED ol[3] = { 0 };
```

```
    HANDLE hEvent[3] = { 0 };

    // By:LyShark
    hFile = CreateFileA("\\\\.\\LySharkAsync", GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
        NULL);
    if (INVALID_HANDLE_VALUE == hFile)
        return  0;

    // event用来通知请求完成
    hEvent[0] = CreateEvent(NULL, TRUE, FALSE, NULL);
    ol[0].hEvent = hEvent[0];

    hEvent[1] = CreateEvent(NULL, TRUE, FALSE, NULL);
    ol[1].hEvent = hEvent[1];

    hEvent[2] = CreateEvent(NULL, TRUE, FALSE, NULL);
    ol[2].hEvent = hEvent[2];

    // 读取事件内容到缓存
    ReadFile(hFile, Buffer[0], 13, &dwRet[0], &ol[0]);
    ReadFile(hFile, Buffer[1], 13, &dwRet[1], &ol[1]);
    ReadFile(hFile, Buffer[2], 13, &dwRet[2], &ol[2]);

    // 等待三个事件执行完毕
    WaitForMultipleObjects(3, hEvent, TRUE, INFINITE);

    // 输出结果
    printf("缓存LyShark A: %s \n", Buffer[0]);
    printf("缓存LyShark B: %s \n", Buffer[1]);
    printf("缓存LyShark C: %s \n", Buffer[2]);

    CloseHandle(hFile);
    return  0;
}
```

这段代码最终运行效果如下：


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help
Time Debug Print
1932 3073.91040039 39741718750 - STORMINI: StorNVMe - POWER: IDLE
1933 3073.91040039 39741718750 - STORMINI: StorNVMe - POWER: ACTIVE
1934 3074.30590820 当前DPC Irp: 0x492e6cd0
1935 3074.30615234 读取DPC长度: 13
1936 3074.91308594 39751718750 - STORMINI: StorNVMe - POWER: IDLE
1937 3075.19018555 39754531250 - STORMINI: StorNVMe - POWER: ACTIVE
1938 3075.30615234 当前DPC Irp: 0x492c9e10
1939 3075.30615234 读取DPC长度: 13
1940 3076.20117188 39764687500 - STORMINI: StorNVMe - POWER: IDLE
1941 3076.30737305 当前DPC Irp: 0x48f3fcd0
1942 3076.30737305 读取DPC长度: 13
1943 3080.1970214
1944 3080.1982421
1945 3080.1984863
1946 3081.1997070
命令提示符
C:\Users\lyshark\Desktop>lyshark.exe
缓存LyShark A: hello lyshark
缓存LyShark B: hello lyshark
缓存LyShark C: hello lyshark