在笔者上一篇文章`《驱动开发：应用DeviceIoContro开发模板》`简单为大家介绍了如何使用`DeviceIoContro`模板快速创建一个驱动开发通信案例，但是该案例过于简单也无法独立加载运行，本章将继续延申这个知识点，通过封装一套标准通用模板来实现驱动通信中的常用传递方式，这其中包括了如何传递字符串，传递整数，传递数组，传递结构体等方法。可以说如果你能掌握本章模板精讲的内容基本上市面上的功能都可以使用本方法进行通信。

首先定义驱动功能号和名字，提供接口给应用程序调用。

```
#define IOCTL_IO_Msg            CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_TEST           CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_Array          CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_STRUCT         CTL_CODE(FILE_DEVICE_UNKNOWN, 0x803,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_String         CTL_CODE(FILE_DEVICE_UNKNOWN, 0x804,
METHOD_BUFFERED, FILE_ANY_ACCESS)
```

IOCTL_IO_TEST 传递整数： 派遣例程用于传递整数类型，首先客户端通过`DriveControl.IoControl`将数据通过变量`input`传入到内核中，并等待返回，如果返回了结果则`outpot`里存储的就是返回结果，`ref_len`则是返回长度。

看看客户端如何接收这个数据的传递。

```
// 传入x参数,返回到y中,返回长度为z
DWORD input = 100, output = 0, ref_len = 0;
DriveControl.IoControl(IOCTL_IO_TEST, &input, sizeof(input), &output,
sizeof(output), &ref_len);

std::cout << "传入参数: " << input << std::endl;
std::cout << "输出参数: " << output << std::endl;
std::cout << "参数长度: " << ref_len << std::endl;
```

对于驱动中我们不需要做任何操作只需要通过`memcpy(&dw, pIoBuffer, sizeof(DWORD))`得到缓冲区内的数据，对该数据`dw++`递增，最后通过`memcpy(pIoBuffer, &dw, sizeof(DWORD))`再将数据写回到应用层。

```
case IOCTL_IO_TEST:
{
  DWORD dw = 0;

  // 得到输入参数
  memcpy(&dw, pIoBuffer, sizeof(DWORD));

  // 对输入参数进行处理
  dw++;

  // 设置输出参数
  memcpy(pIoBuffer, &dw, sizeof(DWORD));
```

```
    // 返回通信状态
    status = STATUS_SUCCESS;
    break;
}
```

IOCTL_IO_Array 传递数组： 派遣例程用于传递数组类型，首先定义数组`MyArray`将数组`首地址`以及`数组长度`传递到内核中，内核收到首地址以及长度后通过`uInSize / sizeof(int)`得到每一个元素的长度，最后循环输出元素即可。

```
// --------------------------------------------------------------------------
// 应用层
// --------------------------------------------------------------------------
// 传入数组
int MyArray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
int array_count = 10 * sizeof(int);

DriveControl.IoControl(IOCTL_IO_Array, &MyArray, array_count, 0, 0, 0);

// --------------------------------------------------------------------------
// 内核层
// --------------------------------------------------------------------------
// 输出一个数组
case IOCTL_IO_Array:
{
    int *ArrayPtr = (int *)pIoBuffer;
    int count = uInSize / sizeof(int);

    for (int x = 0; x < count; x++)
    {
        DbgPrint("计数器: %d \n", ArrayPtr[x]);
    }

    status = STATUS_SUCCESS;
    break;
}
```

IOCTL_IO_STRUCT 传递结构： 结构体的传输与数组类似，仅仅只是在接收到数据后对其进行一定的转换即可，应用层只需要`DriveControl.IoControl`发送`send_ptr`指针并等待回传recv_ptr`即可，最后将得到的结果直接输出，而内核层收到数据后仅仅也只需要`(MyData *)pIoBuffer`将数据转为一个指针即可操作这片区域，当操作完成时以同样的方式`memcpy(pIoBuffer, &send_data, sizeof(MyData))`返回一个结构体给应用层。

```
// --------------------------------------------------------------------------
// 应用层
// --------------------------------------------------------------------------
// 传入一个结构体,返回结构体
MyData send_ptr, recv_ptr;
DWORD dwSize = 0;

memset(send_ptr.szUname, 0, 1024);
memset(recv_ptr.szUname, 0, 1024);

send_ptr.uuid = 1001;
strcpy(send_ptr.szUname, "lyshark");

// 派遣命令
DriveControl.IoControl(IOCTL_IO_STRUCT, &send_ptr, sizeof(send_ptr), &recv_ptr,
sizeof(recv_ptr), &dwSize);
// DeviceIoControl(hDevice, IOCTL_IO_STRUCT, &send_ptr, sizeof(send_ptr),
(LPVOID)&recv_ptr, sizeof(recv_ptr), &dwSize, 0);

std::cout << "内核返回数据: " << recv_ptr.uuid << std::endl;
std::cout << "内核返回数据: " << recv_ptr.szUname << std::endl;
std::cout << "内核返回长度: " << dwSize << std::endl;

// --------------------------------------------------------------------------
// 内核层
// --------------------------------------------------------------------------
// 测试传递结构体
case IOCTL_IO_STRUCT:
{
    MyData recv_data, send_data;

    // 获取到应用层传入的数据
    /*
    MyData *ptr = (MyData *)pIoBuffer;
    DbgPrint("获取序列号: %d \n", ptr->uid);
    DbgPrint("获取名字: %s \n", ptr->szBuf);
    */

    // 获取数据
    memcpy(&recv_data, pIoBuffer, sizeof(MyData));
    DbgPrint("[驱动读入]: UID: %d --> Name: %s \n", recv_data.uuid,
recv_data.szUname);

    // 发送数据
    memset(send_data.szUname, 0, 1024);
    send_data.uuid = 1002;
    strcpy(send_data.szUname, "lyshark");
    memcpy(pIoBuffer, &send_data, sizeof(MyData));

    status = STATUS_SUCCESS;
    break;
}
```

如上就是内核层与应用层的部分代码功能分析，至于传递字符串也很简单大家自己学习下即可掌握，接下来我将完整代码分享出来，大家可以自行测试效果。

驱动程序`WinDDK.sys`完整代码；

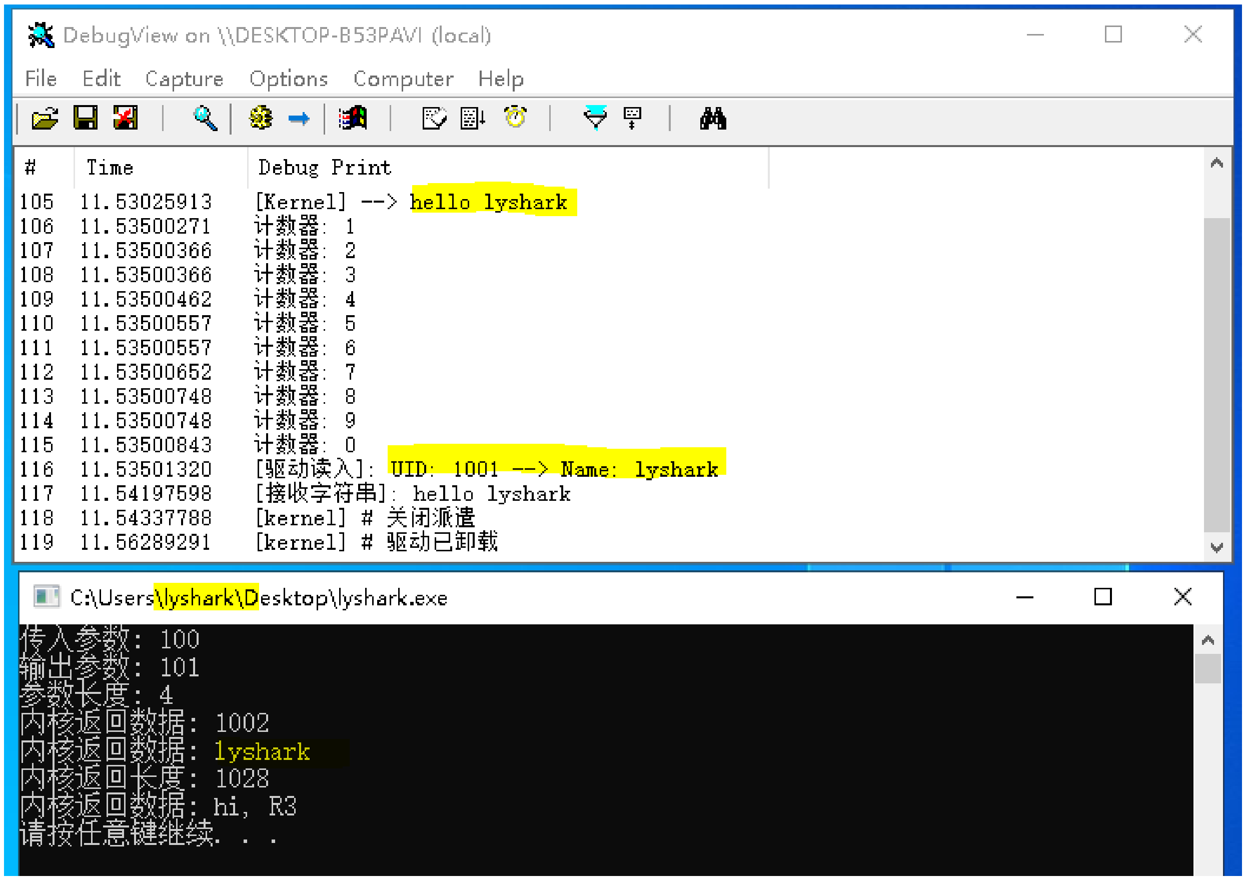
```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#define _CRT_SECURE_NO_WARNINGS
#include <ntifs.h>
#include <windef.h>

// 定义符号链接，一般来说修改为驱动的名字即可
#define DEVICE_NAME        L"\\Device\\WinDDK"
#define LINK_NAME          L"\\DosDevices\\WinDDK"
#define LINK_GLOBAL_NAME   L"\\DosDevices\\Global\\WinDDK"

// 定义驱动功能号和名字，提供接口给应用程序调用
#define IOCTL_IO_Msg            CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, 
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_TEST           CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, 
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_Array          CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, 
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_STRUCT         CTL_CODE(FILE_DEVICE_UNKNOWN, 0x803, 
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_IO_String         CTL_CODE(FILE_DEVICE_UNKNOWN, 0x804, 
METHOD_BUFFERED, FILE_ANY_ACCESS)

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

// 定义传递结构体
typedef struct
{
    int uuid;
    char szUname[1024];
}MyData;

// 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;
}

// 驱动卸载的处理例程
VOID DriverUnload(PDRIVER_OBJECT pDriverObj)
{
    if (pDriverObj->DeviceObject)
    {
        UNICODE_STRING strLink;

        // 删除符号连接和设备
        RtlInitUnicodeString(&strLink, LINK_NAME);
        IoDeleteSymbolicLink(&strLink);
        IoDeleteDevice(pDriverObj->DeviceObject);
        DbgPrint("[kernel] # 驱动已卸载 \n");
    }
}

// IRP_MJ_CREATE 对应的处理例程，一般不用管它
NTSTATUS DispatchCreate(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    DbgPrint("[kernel] # 驱动处理例程载入 \n");
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = 0;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// IRP_MJ_CLOSE 对应的处理例程，一般不用管它
NTSTATUS DispatchClose(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    DbgPrint("[kernel] # 关闭派遣 \n");
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    pIrp->IoStatus.Information = 0;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

// IRP_MJ_DEVICE_CONTROL 对应的处理例程，驱动最重要的函数
NTSTATUS DispatchIoctl(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    NTSTATUS status = STATUS_INVALID_DEVICE_REQUEST;
    PIO_STACK_LOCATION pIrpStack;
    ULONG uIoControlCode;
    PVOID pIoBuffer;
    ULONG uInSize;
    ULONG uOutSize;

    // 获得IRP里的关键数据
    pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

    // 获取控制码
    uIoControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;

    // 输入和输出的缓冲区（DeviceIoControl的InBuffer和OutBuffer都是它）
    pIoBuffer = pIrp->AssociatedIrp.SystemBuffer;

    // EXE发送传入数据的BUFFER长度（DeviceIoControl的nInBufferSize）
    uInSize = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;

    // EXE接收传出数据的BUFFER长度（DeviceIoControl的nOutBufferSize）
    uOutSize = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;

    // 对不同控制信号的处理流程
    switch (uIoControlCode)
    {
        // 输出一段话
    case IOCTL_IO_Msg:
    {
        DbgPrint("[Kernel] --> hello lyshark \n");
        status = STATUS_SUCCESS;
        break;
    }

    // 测试单一参数传递
    case IOCTL_IO_TEST:
    {
        DWORD dw = 0;

        // 得到输入参数
        memcpy(&dw, pIoBuffer, sizeof(DWORD));

        // 对输入参数进行处理
        dw++;

        // 设置输出参数
        memcpy(pIoBuffer, &dw, sizeof(DWORD));

        // 返回通信状态
        status = STATUS_SUCCESS;
        break;
    }

    // 输出一个数组
    case IOCTL_IO_Array:
    {
        int *ArrayPtr = (int *)pIoBuffer;
        int count = uInSize / sizeof(int);

        for (int x = 0; x < count; x++)
        {
            DbgPrint("计数器: %d \n", ArrayPtr[x]);
        }

        status = STATUS_SUCCESS;
        break;
    }

    // 测试传递结构体
    case IOCTL_IO_STRUCT:
    {
        MyData recv_data, send_data;

        // 获取到应用层传入的数据
        /*
        MyData *ptr = (MyData *)pIoBuffer;
        DbgPrint("获取序列号: %d \n", ptr->uid);
        DbgPrint("获取名字: %s \n", ptr->szBuf);
        */

        // 获取数据
        memcpy(&recv_data, pIoBuffer, sizeof(MyData));
        DbgPrint("[驱动读入]: UID: %d --> Name: %s \n", recv_data.uuid,
recv_data.szUname);

        // 发送数据
        memset(send_data.szUname, 0, 1024);
        send_data.uuid = 1002;
        strcpy(send_data.szUname, "lyshark");
        memcpy(pIoBuffer, &send_data, sizeof(MyData));

        status = STATUS_SUCCESS;
        break;
    }
    // 测试传递字符串
    case IOCTL_IO_String:
    {
        char szString[256] = { 0 };
        char szSendPtr[256] = { 0 };

        // 接收字符串
        memcpy(szString, pIoBuffer, sizeof(szString));
        DbgPrint("[接收字符串]: %s \n", szString);

        // 发送字符串
        strcpy(szSendPtr, "hi, R3");
        memcpy(pIoBuffer, &szSendPtr, sizeof(szSendPtr));

        status = STATUS_SUCCESS;
        break;
    }
    }

    // 设定DeviceIoControl的*lpBytesReturned的值（如果通信失败则返回0长度）
    if (status == STATUS_SUCCESS)
    {
        pIrp->IoStatus.Information = uOutSize;
    }
    else
    {
        pIrp->IoStatus.Information = 0;
    }

    // 设定DeviceIoControl的返回值是成功还是失败
    pIrp->IoStatus.Status = status;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return status;
}
```

```
// 驱动的初始化工作
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegistryString)
{
    NTSTATUS status = STATUS_SUCCESS;
    UNICODE_STRING ustrLinkName;
    UNICODE_STRING ustrDevName;
    PDEVICE_OBJECT pDevObj;

    // 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);
        pDriverObj->MajorFunction[i] = DefaultDispatch;
    }

    // 设置分发函数和卸载例程
    pDriverObj->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
    pDriverObj->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;
    pDriverObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchIoctl;
    pDriverObj->DriverUnload = DriverUnload;

    // 创建一个设备
    RtlInitUnicodeString(&ustrDevName, DEVICE_NAME);

    // FILE_DEVICE_EXTENSION 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。
    status = IoCreateDevice(pDriverObj, sizeof(FILE_DEVICE_EXTENSION), &ustrDevName, FILE_DEVICE_UNKNOWN, 0, FALSE, &pDevObj);
    if (!NT_SUCCESS(status))
    {
        return status;
    }

    // 判断支持的WDM版本，其实这个已经不需要了，纯属WIN9X和WINNT并存时代的残留物
    if (IoIsWdmVersionAvailable(1, 0x10))
    {
        RtlInitUnicodeString(&ustrLinkName, LINK_GLOBAL_NAME);
    }
    else
    {
        RtlInitUnicodeString(&ustrLinkName, LINK_NAME);
    }

    // 创建符号连接
    status = IoCreateSymbolicLink(&ustrLinkName, &ustrDevName);
    if (!NT_SUCCESS(status))
    {
        DbgPrint("创建符号链接失败 \n");
        IoDeleteDevice(pDevObj);
        return status;
    }
    DbgPrint("[ Hello LyShark.com ] # 驱动初始化完毕 \n");

    // 返回加载驱动的状态（如果返回失败，驱动讲被清除出内核空间）
    return STATUS_SUCCESS;
}
```

应用层客户端程序`lyshark.exe`完整代码；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

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

#pragma comment(lib,"user32.lib")
#pragma comment(lib,"advapi32.lib")

// 定义驱动功能号和名字，提供接口给应用程序调用
#define IOCTL_IO_Msg            0x800
#define IOCTL_IO_TEST           0x801
#define IOCTL_IO_Array          0x802
#define IOCTL_IO_STRUCT         0x803
#define IOCTL_IO_String         0x804

class cDrvCtrl
{
public:
    cDrvCtrl()
    {
        m_pSysPath = NULL;
        m_pServiceName = NULL;
        m_pDisplayName = NULL;
        m_hSCManager = NULL;
        m_hService = NULL;
        m_hDriver = INVALID_HANDLE_VALUE;
    }
    ~cDrvCtrl()
    {
        CloseServiceHandle(m_hService);
        CloseServiceHandle(m_hSCManager);
        CloseHandle(m_hDriver);
    }

    // 安装驱动
    BOOL Install(PCHAR pSysPath, PCHAR pServiceName, PCHAR pDisplayName)
    {
        m_pSysPath = pSysPath;
        m_pServiceName = pServiceName;
        m_pDisplayName = pDisplayName;
        m_hSCManager = OpenSCManagerA(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        if (NULL == m_hSCManager)
        {
            m_dwLastError = GetLastError();
            return FALSE;
        }
        m_hService = CreateServiceA(m_hSCManager, m_pServiceName, m_pDisplayName,
            SERVICE_ALL_ACCESS, SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START,
SERVICE_ERROR_NORMAL,
            m_pSysPath, NULL, NULL, NULL, NULL, NULL);
        if (NULL == m_hService)
        {
            m_dwLastError = GetLastError();
            if (ERROR_SERVICE_EXISTS == m_dwLastError)
            {
                m_hService = OpenServiceA(m_hSCManager, m_pServiceName,
SERVICE_ALL_ACCESS);
                if (NULL == m_hService)
                {
                    CloseServiceHandle(m_hSCManager);
                    return FALSE;
                }
            }
            else
            {
                CloseServiceHandle(m_hSCManager);
                return FALSE;
            }
        }
        return TRUE;
    }

    // 启动驱动
    BOOL Start()
    {
        if (!StartServiceA(m_hService, NULL, NULL))
        {
            m_dwLastError = GetLastError();
            return FALSE;
        }
        return TRUE;
    }

    // 关闭驱动
    BOOL Stop()
    {
        SERVICE_STATUS ss;
        GetSvcHandle(m_pServiceName);
        if (!ControlService(m_hService, SERVICE_CONTROL_STOP, &ss))
        {
            m_dwLastError = GetLastError();
            return FALSE;
        }
        return TRUE;
    }

    // 移除驱动
    BOOL Remove()
    {
        GetSvcHandle(m_pServiceName);
        if (!DeleteService(m_hService))
        {
            m_dwLastError = GetLastError();
            return FALSE;
        }
        return TRUE;
    }

    // 打开驱动
    BOOL Open(PCHAR pLinkName)
    {
        if (m_hDriver != INVALID_HANDLE_VALUE)
            return TRUE;
        m_hDriver = CreateFileA(pLinkName, GENERIC_READ | GENERIC_WRITE, 0, 0,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
        if (m_hDriver != INVALID_HANDLE_VALUE)
            return TRUE;
        else
            return FALSE;
    }

    // 发送控制信号
    BOOL IoControl(DWORD dwIoCode, PVOID InBuff, DWORD InBuffLen, PVOID OutBuff,
DWORD OutBuffLen, DWORD *RealRetBytes)
    {
        DWORD dw;
        BOOL b = DeviceIoControl(m_hDriver, CTL_CODE_GEN(dwIoCode), InBuff,
InBuffLen, OutBuff, OutBuffLen, &dw, NULL);
        if (RealRetBytes)
            *RealRetBytes = dw;
        return b;
    }
private:

    // 获取服务句柄
    BOOL GetSvcHandle(PCHAR pServiceName)
    {
        m_pServiceName = pServiceName;
        m_hSCManager = OpenSCManagerA(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        if (NULL == m_hSCManager)
        {
            m_dwLastError = GetLastError();
            return FALSE;
        }
        m_hService = OpenServiceA(m_hSCManager, m_pServiceName, SERVICE_ALL_ACCESS);
        if (NULL == m_hService)
        {
            CloseServiceHandle(m_hSCManager);
            return FALSE;
        }
        else
        {
            return TRUE;
        }
    }

    // 获取控制信号对应字符串
    DWORD CTL_CODE_GEN(DWORD lngFunction)
    {
        return (FILE_DEVICE_UNKNOWN * 65536) | (FILE_ANY_ACCESS * 16384) |
(lngFunction * 4) | METHOD_BUFFERED;
    }

public:
    DWORD m_dwLastError;
    PCHAR m_pSysPath;
    PCHAR m_pServiceName;
    PCHAR m_pDisplayName;
    HANDLE m_hDriver;
    SC_HANDLE m_hSCManager;
    SC_HANDLE m_hService;
};

void GetAppPath(char *szCurFile)
{
    GetModuleFileNameA(0, szCurFile, MAX_PATH);
    for (SIZE_T i = strlen(szCurFile) - 1; i >= 0; i--)
    {
        if (szCurFile[i] == '\\')
        {
            szCurFile[i + 1] = '\0';
            break;
        }
    }
}

// 定义传递结构体
typedef struct
{
    int uuid;
    char szUname[1024];
}MyData;

int main(int argc, char *argv[])
{
    cDrvCtrl DriveControl;

    // 设置驱动名称
    char szSysFile[MAX_PATH] = { 0 };
    char szSvcLnkName[] = "WinDDK";;
    GetAppPath(szSysFile);
    strcat(szSysFile, "WinDDK.sys");

    // 安装并启动驱动
    DriveControl.Install(szSysFile, szSvcLnkName, szSvcLnkName);
    DriveControl.Start();

    // 打开驱动的符号链接
    DriveControl.Open("\\\\.\\WinDDK");

    // 无参数输出
    DriveControl.IoControl(IOCTL_IO_Msg, 0, 0, 0, 0, 0);

    // 传入x参数,返回到y中,返回长度为z
    DWORD input = 100, output = 0, ref_len = 0;
    DriveControl.IoControl(IOCTL_IO_TEST, &input, sizeof(input), &output,
sizeof(output), &ref_len);

    std::cout << "传入参数: " << input << std::endl;
    std::cout << "输出参数: " << output << std::endl;
    std::cout << "参数长度: " << ref_len << std::endl;

    // 传入数组
    int MyArray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    int array_count = 10 * sizeof(int);

    DriveControl.IoControl(IOCTL_IO_Array, &MyArray, array_count, 0, 0, 0);

    // 传入一个结构体,返回结构体
    MyData send_ptr, recv_ptr;
    DWORD dwSize = 0;

    memset(send_ptr.szUname, 0, 1024);
    memset(recv_ptr.szUname, 0, 1024);

    send_ptr.uuid = 1001;
    strcpy(send_ptr.szUname, "lyshark");

    // 派遣命令
    DriveControl.IoControl(IOCTL_IO_STRUCT, &send_ptr, sizeof(send_ptr), &recv_ptr,
sizeof(recv_ptr), &dwSize);
    // DeviceIoControl(hDevice, IOCTL_IO_STRUCT, &send_ptr, sizeof(send_ptr),
(LPVOID)&recv_ptr, sizeof(recv_ptr), &dwSize, 0);

    std::cout << "内核返回数据: " << recv_ptr.uuid << std::endl;
    std::cout << "内核返回数据: " << recv_ptr.szUname << std::endl;
    std::cout << "内核返回长度: " << dwSize << std::endl;
    // 传入一个字符串,返回一个字符串
    char szString[256] = { 0 };
    char szRecvPtr[256] = { 0 };

    // 派遣命令
    strcpy(szString, "hello lyshark");
    DriveControl.IoControl(IOCTL_IO_String, &szString, sizeof(szString), &szRecvPtr,
sizeof(szRecvPtr), &dwSize);
    std::cout << "内核返回数据: " << szRecvPtr << std::endl;


    // 关闭符号链接句柄
    CloseHandle(DriveControl.m_hDriver);

    // 停止并卸载驱动
    DriveControl.Stop();
    DriveControl.Remove();

    system("pause");
    return 0;
}
```

手动编译这两个程序，将驱动签名后以管理员身份运行`lyshark.exe`客户端，此时屏幕中即可看到滚动输出效果，如此一来就实现了循环传递参数的目的。


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help
C:\Users\lyshark\Desktop\lyshark.exe

```
#    Time          Debug Print
105  11.53025913   [Kernel] --> hello lyshark
106  11.53500271   计数器: 1
107  11.53500366   计数器: 2
108  11.53500366   计数器: 3
109  11.53500462   计数器: 4
110  11.53500557   计数器: 5
111  11.53500557   计数器: 6
112  11.53500652   计数器: 7
113  11.53500748   计数器: 8
114  11.53500748   计数器: 9
115  11.53500843   计数器: 0
116  11.53501320   [驱动读入]: UID: 1001 --> Name: lyshark
117  11.54197598   [接收字符串]: hello lyshark
118  11.54337788   [kernel] # 关闭派遣
119  11.56289291   [kernel] # 驱动已卸载
```

```
传入参数: 100
输出参数: 101
参数长度: 4
内核返回数据: 1002
内核返回数据: lyshark
内核返回长度: 1028
内核返回数据: hi, R3
请按任意键继续. . .
```