驱动框架、内核驱动与R3通信

简介

- 硬件驱动的抽象

  `系统 -> 驱动 -> 硬件` 抽象为 `系统 -> 驱动 -> 文件`，驱动功能（如：打开、关闭、读写、控制、电源）等向系统注册函数指针，即注册派遣函数

  设备与文件间的抽象

- 驱动对象(1)绑定设备对象(n)

驱动对象DRIVER_OBJECT

```
typedef struct _DRIVER_OBJECT {
  CSHORT              Type;
  CSHORT              Size;
  PDEVICE_OBJECT      DeviceObject;       // 设备对象，是个链表
  ULONG               Flags;
  PVOID               DriverStart;        // 本驱动模块在内存中的位置
  ULONG               DriverSize;         // 本驱动模块的大小
  PVOID               DriverSection;      // 对应_LDR_DATA_TABLE_ENTRY结构体的地址，
是所有内核模块的链表
  PDRIVER_EXTENSION   DriverExtension;
  UNICODE_STRING      DriverName;         // 本驱动的名字
  PUNICODE_STRING     HardwareDatabase;
  PFAST_IO_DISPATCH   FastIoDispatch;
  PDRIVER_INITIALIZE  DriverInit;
  PDRIVER_STARTIO     DriverStartIo;
  PDRIVER_UNLOAD      DriverUnload;       // 卸载函数
  PDRIVER_DISPATCH    MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];     // 派遣函数，
28项
} DRIVER_OBJECT, *PDRIVER_OBJECT;

#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
```

```c
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                IRP_MJ_PNP      // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

设备对象DEVICE_OBJECT

```c
typedef struct _DEVICE_OBJECT {
  CSHORT                         Type;
  USHORT                         Size;
  LONG                           ReferenceCount;
  struct _DRIVER_OBJECT          *DriverObject;       // 回指驱动对象
  struct _DEVICE_OBJECT          *NextDevice;
  struct _DEVICE_OBJECT          *AttachedDevice;
  struct _IRP                    *CurrentIrp;
  PIO_TIMER                      Timer;
  ULONG                          Flags;
  ULONG                          Characteristics;
  __volatile PVPB                Vpb;
  PVOID                          DeviceExtension;
  DEVICE_TYPE                    DeviceType;
  CCHAR                          StackSize;
  union {
    LIST_ENTRY         ListEntry;
    WAIT_CONTEXT_BLOCK Wcb;
  } Queue;
  ULONG                          AlignmentRequirement;
  KDEVICE_QUEUE                  DeviceQueue;
  KDPC                           Dpc;
  ULONG                          ActiveThreadCount;
  PSECURITY_DESCRIPTOR           SecurityDescriptor;
  KEVENT                         DeviceLock;
  USHORT                         SectorSize;
  USHORT                         Spare1;
  struct _DEVOBJ_EXTENSION *DeviceObjectExtension;
  PVOID                          Reserved;
} DEVICE_OBJECT, *PDEVICE_OBJECT;
```

框架

`KdBreakPoint`和`KdPrint`宏利用了条件编译，只有在Debug下才会被调用

其中`KdPrint`需要自行对参数加括号，例如：`KdPrint(("Hello Kernel! - Install"));`

主体

1. 注册卸载函数，在卸载函数中完成对设备的删除

   `DriverObject->DriverUnload = 卸载函数`

   卸载函数声明为：

```
typedef
VOID
DRIVER_UNLOAD (
    _In_ struct _DRIVER_OBJECT *DriverObject
    );

typedef DRIVER_UNLOAD *PDRIVER_UNLOAD;
```

2. 注册派遣函数

   `DriverObject->MajorFunction[xxx] = 派遣函数`

   派遣函数声明为：

```
typedef
NTSTATUS
DRIVER_DISPATCH (
    _In_ struct _DEVICE_OBJECT *DeviceObject,
    _Inout_ struct _IRP *Irp
    );

typedef DRIVER_DISPATCH *PDRIVER_DISPATCH;
```

3. 设置IO模式

```
DriverObject->Flags = DO_BUFFERED_IO;   // 缓冲区，在R0中开辟空间，复制缓冲区内容到此处
DriverObject->Flags = DO_DIRECT_IO;     // 直接，映射内存到R3的物理地址上，调用API MmGetSystem
```

   常用缓冲区模式，通过`Irp->AssociatedIrp.SystemBuffer`来获取内核申请的缓冲区

4. 创建设备对象

   `IoCreateDevice`函数，指定内核设备`FILE_DEVICE_UNKNOWN`

```
NTSTATUS IoCreateDevice(
  PDRIVER_OBJECT  DriverObject,            // 驱动对象的指针
  ULONG           DeviceExtensionSize,     // 设备扩展大小
  PUNICODE_STRING DeviceName,              // Nt设备名
  DEVICE_TYPE     DeviceType,              // 设备类型，FILE_DEVICE_UNKNOWN为内核设备
  ULONG           DeviceCharacteristics,   // 指定一个或多个系统定义的常量，它们一起提供有关驱动程序设备的附加信息
  BOOLEAN         Exclusive,               // 指定设备对象是否表示独占设备，大多数情况是FALSE
  PDEVICE_OBJECT  *DeviceObject            // 传出参数，PDEVICE_OBJECT的指针
);
```

Nt设备名格式：`\Device\DeviceName`，参考：https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/nt-device-names

5. 如果R3需要交互，则为设备对象创建一个符号链接

`IoCreateSymbolicLink`函数创建符号链接

```
NTSTATUS IoCreateSymbolicLink(
  PUNICODE_STRING SymbolicLinkName,       // 符号链接名
  PUNICODE_STRING DeviceName            // 设备名
);
```

符号链接名格式：`\DosDevices\DosDeviceName`或者`\??\DosDeviceName`，参考：https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-ms-dos-device-names

`IoDeleteSymbolicLink`函数删除符号链接，需要在卸载函数中完成对符号链接的删除

```
NTSTATUS IoDeleteSymbolicLink(
  PUNICODE_STRING SymbolicLinkName    // 符号链接名
);
```

关于内核API的返回值

基本上内核函数的返回值都为`NTSTATUS`类型，属于宏定义

返回值说明参考：https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55

派遣函数

```c
typedef
NTSTATUS
DRIVER_DISPATCH (
    _In_ struct _DEVICE_OBJECT *DeviceObject,   // 设备对象
    _Inout_ struct _IRP *Irp                    // IO请求包
    );

typedef DRIVER_DISPATCH *PDRIVER_DISPATCH;
```

_IRP参数

IO请求包（IO Request Packet），属于派遣函数的第二个参数

- 完成请求：`IoCompleteRequest`

  需要填写`Irp`参数信息：`Irp->IoStatus.Infomation`操作的字节数、`Irp->IoStatus.Status`操作的状态

- 设置异步回调：`IoSetCompletionRoutine`

注意

每个派遣函数在返回之前都要设置IO完成状态，不然R3那边会阻塞（在同步的情况下）

实例

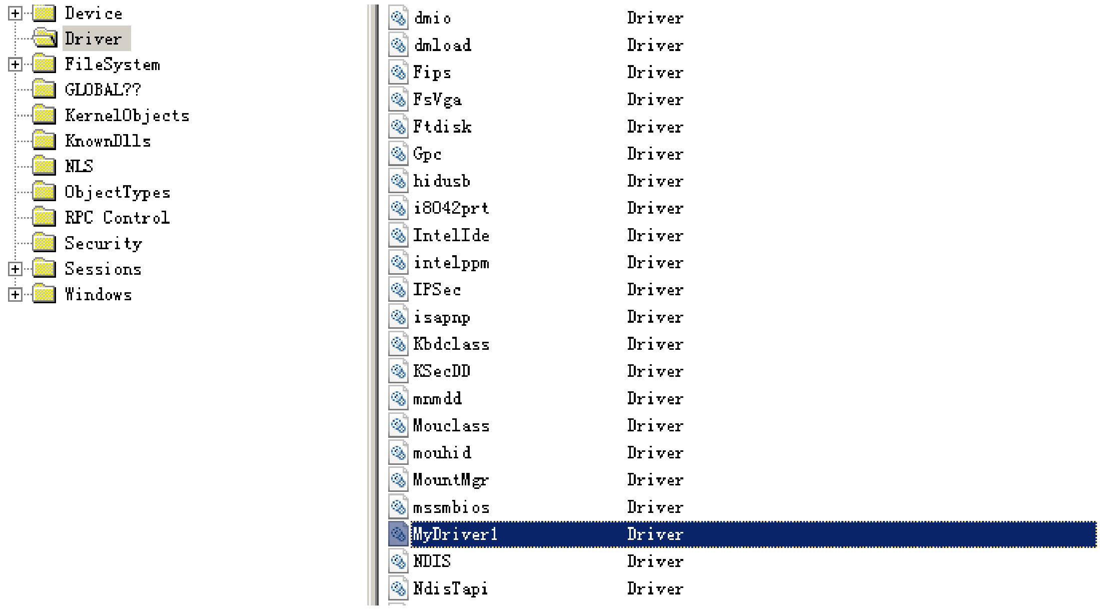
```c
#include <Ntddk.h>

#define MY_DEVICE_NAME L"\\Device\\MyDevice-dsh"
#define MY_SYMBOL_NAME L"\\DosDevices\\MyDevice-dsh"

#define MY_CODE(function) CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800 + (function),
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define CODE_FUN1 MY_CODE(0)
#define CODE_FUN2 MY_CODE(1)


// 入口函数
DRIVER_INITIALIZE DriverEntry;

// 卸载函数
DRIVER_UNLOAD DriverUnload;

// 创建
NTSTATUS
MyDispatchCreate(
    _In_ struct _DEVICE_OBJECT *DeviceObject,
    _Inout_ struct _IRP *Irp
);

// 读
NTSTATUS
MyDispatchRead(
    _In_ struct _DEVICE_OBJECT *DeviceObject,
```

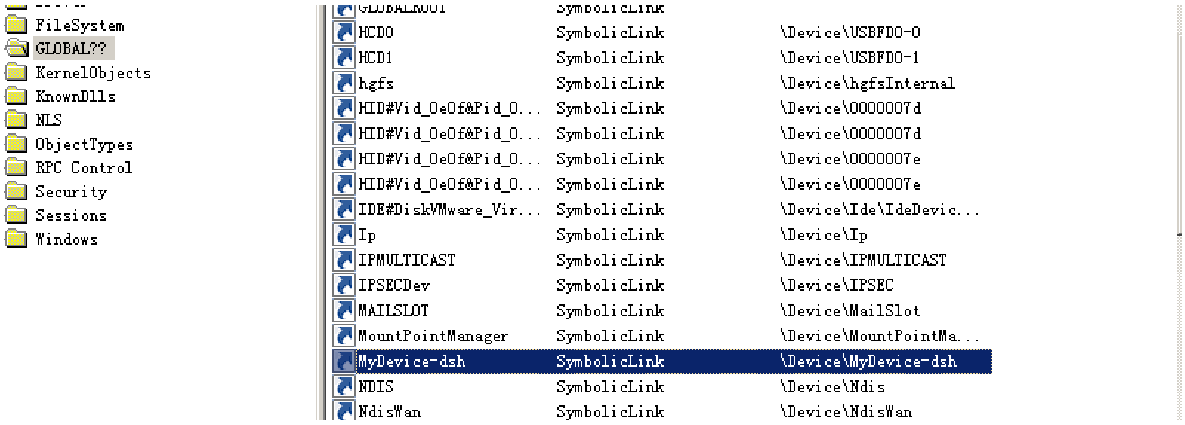
```
    _Inout_ struct _IRP *Irp
);

// 卸载驱动
void DriverUnload(struct _DRIVER_OBJECT* DriverObject)
{
    // 删除设备
    IoDeleteDevice(DriverObject->DeviceObject);     // 这是个链表，多个设备对象时就需
要遍历了

    // 删除符号链接，如果有的话
    UNICODE_STRING symbo_name;
    RtlInitUnicodeString(&symbo_name, MY_SYMBOL_NAME);
    IoDeleteSymbolicLink(&symbo_name);

    KdPrint(("Hello Kernel! - UnInstall"));
}


// 入口函数
_Use_decl_annotations_
NTSTATUS
DriverEntry(struct _DRIVER_OBJECT* DriverObject, PUNICODE_STRING  RegistryPath)
{
    KdBreakPoint();
    DbgPrint(("Hello Kernel! - Install\n"));

    // 注册卸载函数
    DriverObject->DriverUnload = DriverUnload;

    // 注册派遣函数
    DriverObject->MajorFunction[IRP_MJ_CREATE] = MyDispatchCreate;
 // 创建
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = MyDispatchControl;
// 控制
    DriverObject->MajorFunction[IRP_MJ_CLOSE] = MyDispatchClose;
 // 关闭

    // 设置IO模式
    DriverObject->Flags = DO_BUFFERED_IO;   // 复制缓冲区，在R0中开辟空间，复制缓冲区
内容到此处


    NTSTATUS status;    // 返回状态

    // 创建设备对象，一般为IO管理器管理，内核设备FILE_DEVICE_UNKNOWN
    UNICODE_STRING driver_name;
    RtlInitUnicodeString(&driver_name, MY_DEVICE_NAME);
    PDEVICE_OBJECT device_object_ptr = NULL;
    status = IoCreateDevice(DriverObject, 0, &driver_name, FILE_DEVICE_UNKNOWN,
FILE_DEVICE_SECURE_OPEN, FALSE, &device_object_ptr);

    if(!NT_SUCCESS(status)) {
        KdPrint(("IoCreateDevice Error: %p\n", status));
        return status;
    }

    // 创建符号连接
```

```
    UNICODE_STRING symbo_name;
    RtlInitUnicodeString(&symbo_name, MY_SYMBOL_NAME);
    status = IoCreateSymbolicLink(&symbo_name, &driver_name);

    if (!NT_SUCCESS(status)) {
        KdPrint(("IoCreateSymbolicLink Error: %p\n", status));
        return status;
    }

    return STATUS_SUCCESS;
}

// 创建
NTSTATUS
MyDispatchCreate(_In_ struct _DEVICE_OBJECT *DeviceObject, _Inout_ struct _IRP
*Irp)
{
    KdPrint(("Dispatch: [MyDispatchCreate]"));

    // 完成请求
    Irp->IoStatus.Information = 0;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return STATUS_SUCCESS;
}

// 读
NTSTATUS
MyDispatchRead(_In_ struct _DEVICE_OBJECT *DeviceObject, _Inout_ struct _IRP
*Irp)
{
    KdPrint(("Dispatch: [MyDispatchRead]"));

    NTSTATUS Status = STATUS_SUCCESS;
    ULONG_PTR Information = 0;

    PIO_STACK_LOCATION stack_location = IoGetCurrentIrpStackLocation(Irp);
           // 获取Irp堆栈
    ULONG input_length = stack_location-
>Parameters.DeviceIoControl.InputBufferLength;      // 获取输入缓冲区长度
    ULONG output_length = stack_location-
>Parameters.DeviceIoControl.OutputBufferLength;    // 获取输出缓冲区长度
    ULONG control_code = stack_location-
>Parameters.DeviceIoControl.IoControlCode;          // 控制码
    PVOID systembuff = Irp->AssociatedIrp.SystemBuffer;
            // 内核缓冲区

    switch (control_code) {
        case CODE_FUN1:
            // ...
            break;
        case CODE_FUN2:
            // ...
            break;
    }

    // 完成请求
```

```
    Irp->IoStatus.Information = Information;
    Irp->IoStatus.Status = Status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return STATUS_SUCCESS;
}
```

可以在winobj工具中查看设备对象和符号链接





与R3交互

通过R3的API与驱动的符号链接做交互

通过`CreateFile`来打开符号链接，需要指定文件名为`"\\\\.\\xxx"`

例如：

```
::CreateFile(TEXT("\\\\.\\MyDevice-dsh"), GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);
```

拿到句柄之后就可以通过`ReadFile`、`WriteFile`、`CloseHandle`、`DeviceIoControl`等API对驱动进行交互

DeviceIoControl

`DeviceIoControl` 的声明如下：

```
BOOL DeviceIoControl(
  HANDLE       hDevice,              // 设备句柄，由CreateFile打开的
  DWORD        dwIoControlCode,      // 操作码
  LPVOID       lpInBuffer,           // 输入缓冲区
  DWORD        nInBufferSize,        // 输入缓冲区大小
  LPVOID       lpOutBuffer,          // 输出缓冲区
  DWORD        nOutBufferSize,       // 输出缓冲区大小
  LPDWORD      lpBytesReturned,      // 返回的数据长度
  LPOVERLAPPED lpOverlapped          // 重叠IO，可s为NULL
);
```

操作码使用宏 `CTL_CODE` 来定义，例如：

```
//          设备类型          0-0x7ff保留    以什么形式            权限
CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)

// 一般会自己定义宏
#define MY_CODE(function) CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800 + (function),
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define CODE_READ_MEM MY_CODE(0)     // 定义操作码
```

如果选择了 `METHOD_BUFFERED`，输入输出缓冲区指向的是一个，就是在内核中申请的 `SystemBuff`，即驱动将输出结果直接放入 `SystemBuff`

R0和R3程序需要定义相同的宏定义，R3程序要包含头文件 `#include <winioctl.h>`，此头文件必须在 `windows.h` 之后

驱动的IRP堆栈

驱动可分层，每层的参数信息都保存在IRP堆栈上，驱动可调用 `IoAttachDevice` 注册下一层驱动，`IoCopyCurrentIrpStackLocationToNext` 将IRP信息传入IRP栈中供下个驱动使用、`IoCallDriver` 传递给下一层驱动

驱动访问R3

每次派遣回调函数来的时候，PID并不确定，不能直接对R3的地址进行操作

以缓冲区模式获取R3的输入输出

以控制为例，获取R3的输入输出

```
PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(Irp);           //获取IRP堆栈
ULONG nInLenth = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;   //输入缓冲区大小
ULONG nOutLenth = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength; //输入缓冲区大小
ULONG nControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;   //控制码
PVOID pBuff = Irp->AssociatedIrp.SystemBuffer;      //缓冲区，输入输出使用的SystemBuff是同一个
```

当使用缓冲区模式时，`Irp->UserBuffer`为空，系统不给使用

同步问题

在内核空间中，对内核空间的地址访问时需要考虑同步问题

比如调用内核同步的API：`KeEnterCriticalRegion`进临界区、`KeLeaveCriticalRegion`出临界区