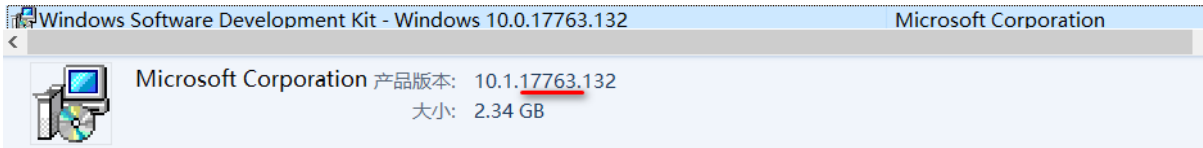


# 1 驱动开发

## 1.1 环境配置

### 1.1.1 开发环境

在开发驱动程序之前，我们需要配置好开发环境，首先安装好VS IDE（这里自己选择版本），其次因为我们需要开发驱动程序所以需要安装WDK（WDK下载地址：<https://docs.microsoft.com/zh-cn/windows-hardware/drivers/other-wdk-downloads>），在我们安装WDK时候需要注意其版本应与SDK的版本一致。我们可以通过控制面板-程序卸载，找到当前VS IDE安装的SDK版本，如下所示，我的系统上SDK的版本是17763：



接着你需要在WDK的下载地址中选择对应系统的安装包，由于我当前是Windows 10 21H2版本，但是页面中并没有对应的版本选择，所以我把所有Windows10的WDK安装包都下载下来了：

Windows 版本	WDK 和相关下载
Windows 11 版本 21H2	<a href="#">Windows 11 版本 21H2 WDK</a>
Windows Server 2022	<a href="#">适用于 Windows Server 2022 的 WDK</a>
Windows 10 版本 2004	<a href="#">适用于 Windows 10 版本 2004 的 WDK</a>
Windows 10 版本 2004	适用于 Windows 10 版本 2004 (10.19041.1) 的 WDK* 请参阅下方的“说明”
Windows 10 版本 1903	<a href="#">适用于 Windows 10 版本 1903 的 WDK</a>
Windows 10 版本 1809	<a href="#">适用于 Windows 10 版本 1809 的 WDK</a>
Windows 10 版本 1803	<a href="#">适用于 Windows 10 版本 1803 的 WDK</a>
Windows 10 版本 1709	<a href="#">适用于 Windows 10 版本 1709 的 WDK</a>
Windows 10 版本 1703	<a href="#">适用于 Windows 10 版本 1703 的 WDK</a>
Windows 10 版本 1607	<a href="#">适用于 Windows 10 版本 1607 的 WDK</a>
Windows 8.1 更新	WDK 8.1 更新 (仅英语版) - 暂时不可用 WDK 8.1 更新测试包 (仅英语版) - 暂时不可用 <a href="#">WDK 8.1 示例</a>
Windows 8	<a href="#">WDK 8</a> (仅英语) <a href="#">WDK 8 可再发行组件</a> (仅英语) <a href="#">WDK 8 示例</a>
Windows 7	<a href="#">WDK 7.1.0</a>

安装包依次打开，就找到了与SDK对应版本的WDK安装包，接着按步骤安装即可：

Windows Driver Kit - Windows 10.0.17763.1

— □ ×

## Specify Location

- ☒ Install the Windows Driver Kit - Windows 10.0.17763.1 to this computer

Install Path:

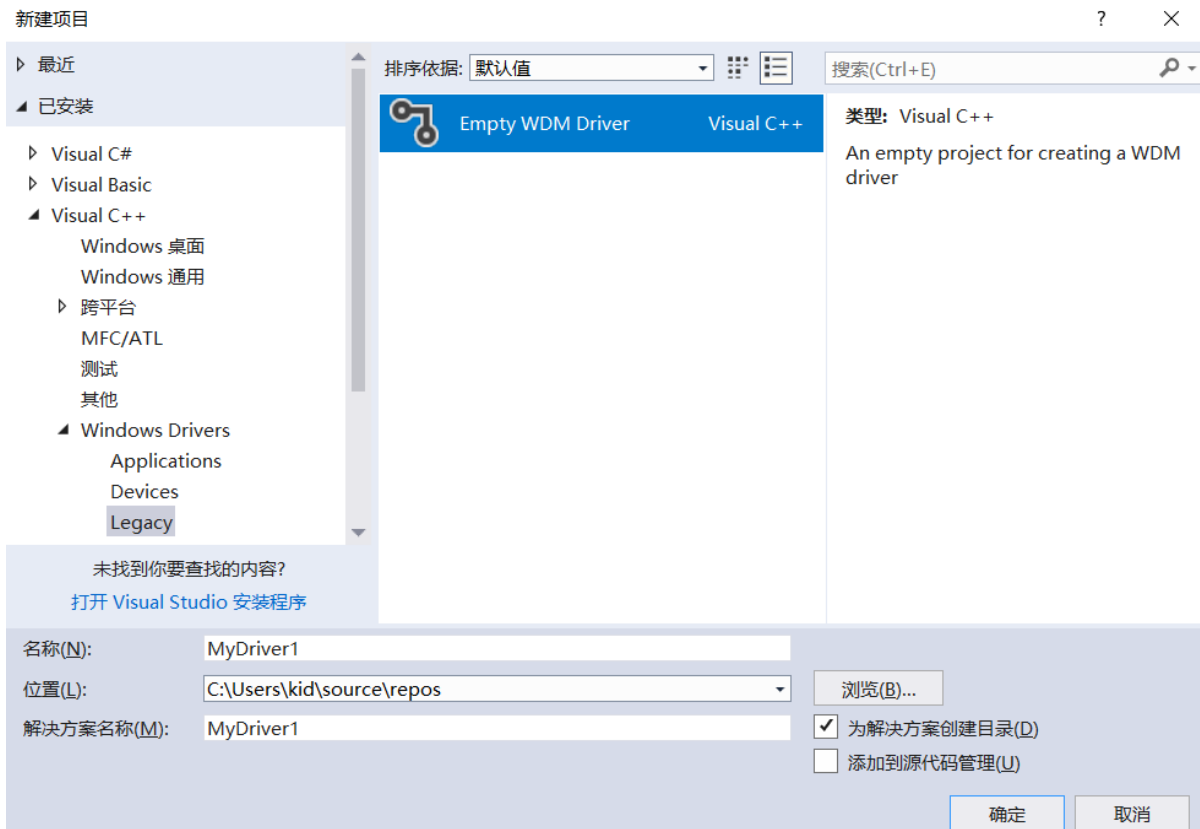
C:\Program Files (x86)\Windows Kits\10\

Browse...

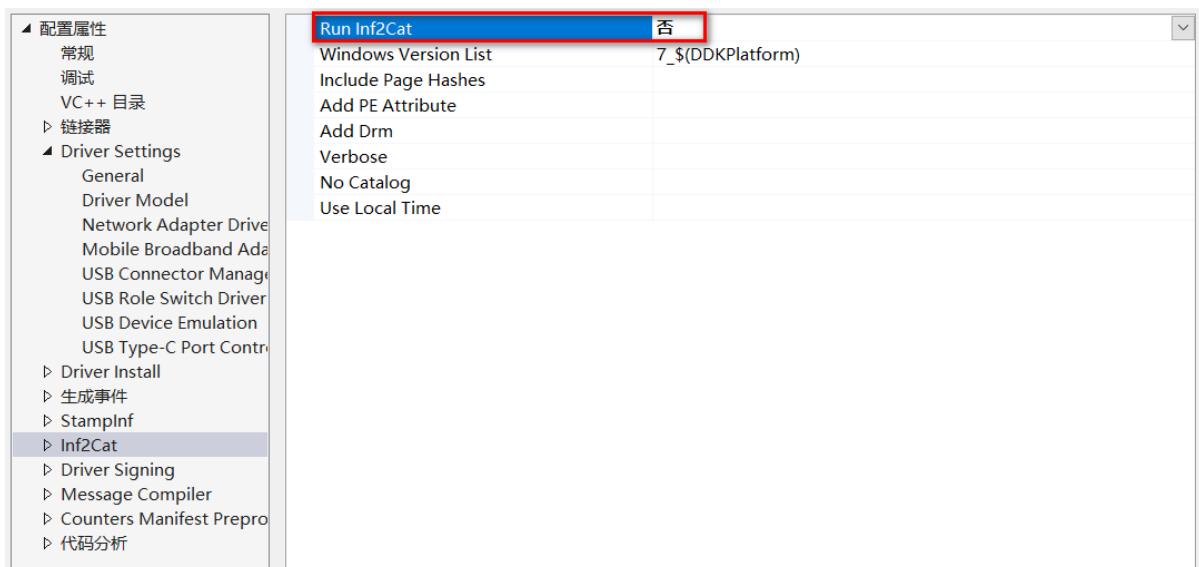
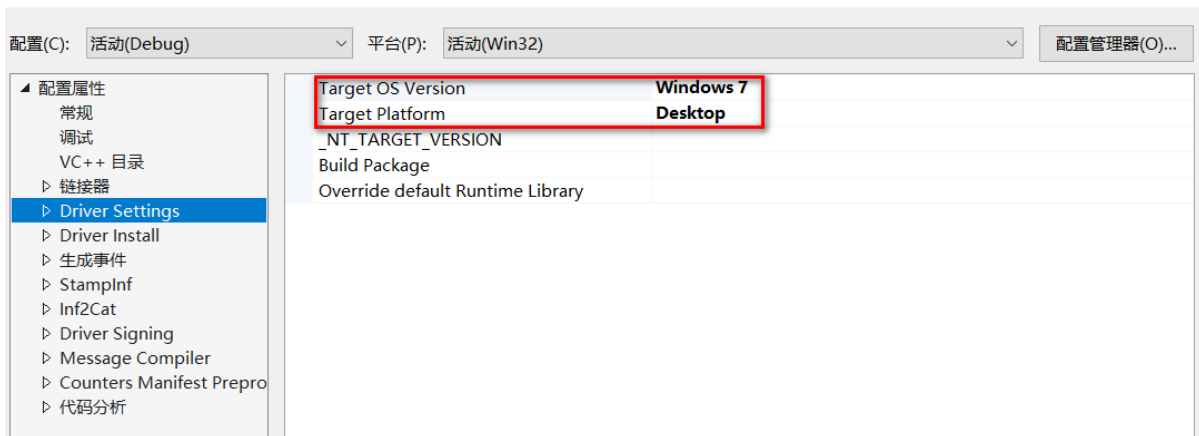
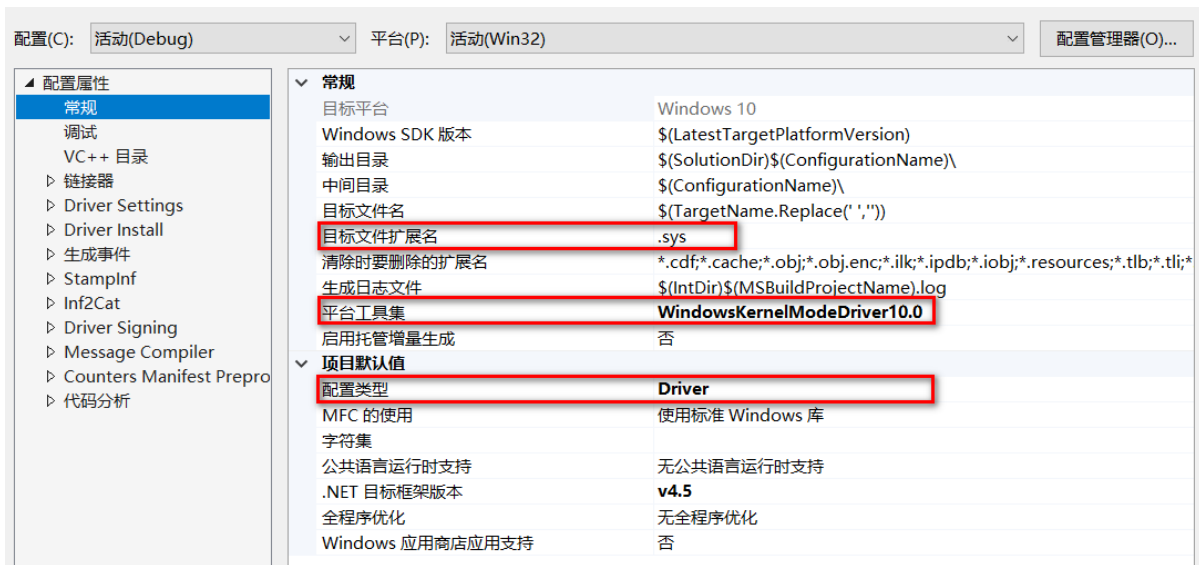
\* Windows Kit common installation path used

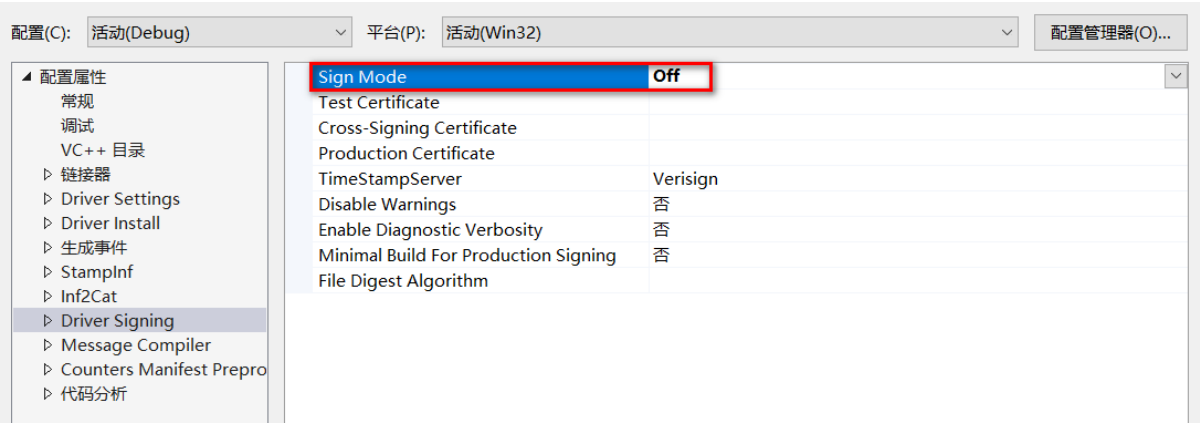
### 1.1.2 项目配置

安装好开发环境之后我们打开VS2017，新建项目并选择VC++下的Windows Drivers，创建Empty WDM Driver项目：



创建完项目之后，进入到项目属性，按如下图所示进行配置：



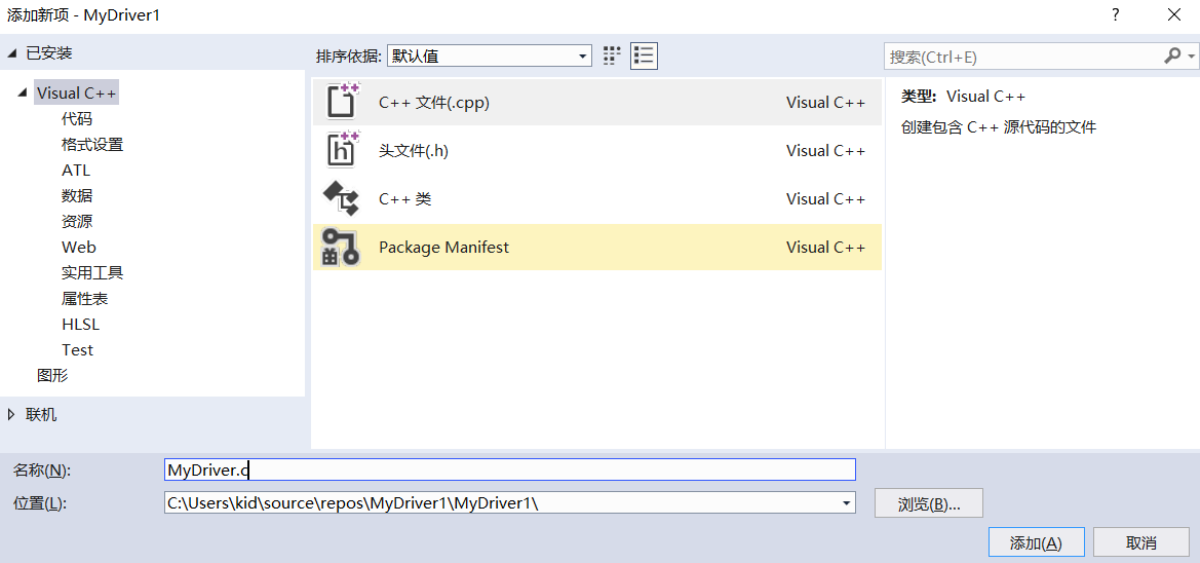


至此，我们所有的配置工作就搞定了。

## 1.2 Hello Driver

### 1.2.1 驱动程序代码

当我们配置好驱动开发环境及项目之后，可以创建代码文件，但需要注意的是我们在学习阶段时候建议代码文件使用C语言，这样编译器就不会进行太多的优化，也便于我们调试：



接着我们将项目代码的警告配置修改一下：

配置属性	附加包含目录	\$(IntDir);%(AdditionalIncludeDirectories)
常规	其他 #using 指令	
调试	调试信息格式	程序数据库 (/Zi)
VC++ 目录	支持仅我的代码调试	否
C/C++	公共语言运行时支持	
链接器	使用 Windows 运行时扩展	
Driver Settings	取消显示启动版权标志	是 (/nologo)
Driver Install	警告等级	等级 3 (/W3)
生成事件	将警告视为错误	否 (/WX-)
StampInf	警告版本	
Inf2Cat	诊断格式	传统型 (/diagnostics:classic)
Driver Signing	SDL 检查	
Wpp Tracing	多处理器编译	
Message Compiler		
Counters Manifest Prepro		
代码分析		

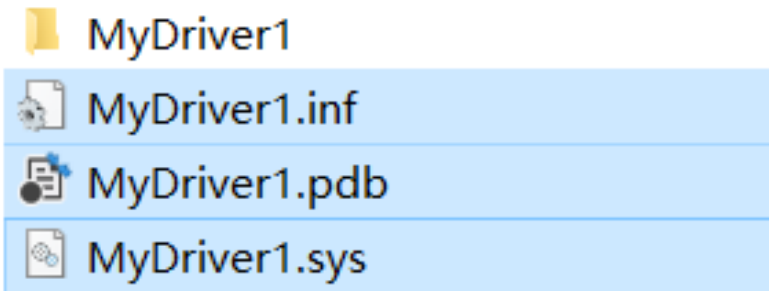
然后我们开始写自己的第一个驱动程序代码，最基本的格式就是包含ntddk.h头文件，以及写好驱动程序入口函数DriverEntry：

```

1  #include <ntddk.h> // 必须要包含的头文件
2
3  // 自定义的驱动程序卸载函数
4  VOID DriverUnload(PDRIVER_OBJECT DriverObject)
5  {
6      DbgPrint("Bye. \n");
7  }
8
9  // 驱动程序入口函数
10 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
    RegistryPath)
11 {
12     DbgPrint("Hello Driver. \n");
13
14     // 设置一个卸载函数，当驱动停止时触发
15     DriverObject->DriverUnload = DriverUnload;
16     return STATUS_SUCCESS;
17 }
```

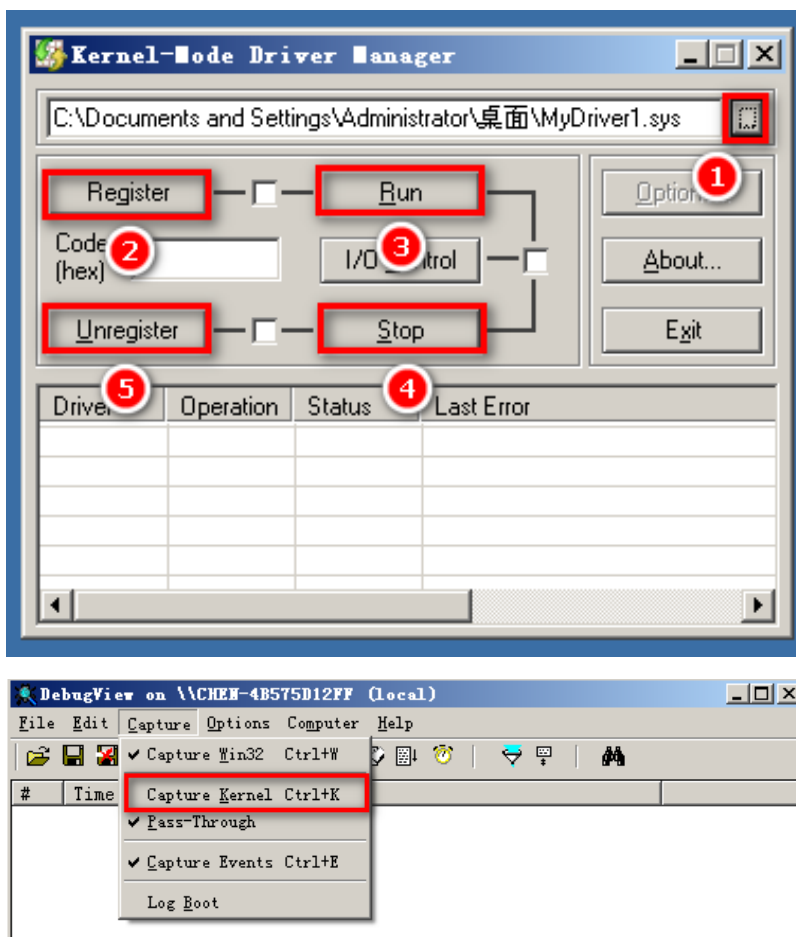
最后我们编译代码，在项目的Debug目录下找到.sys文件，这就是我们编译出来的驱动程序：

名称

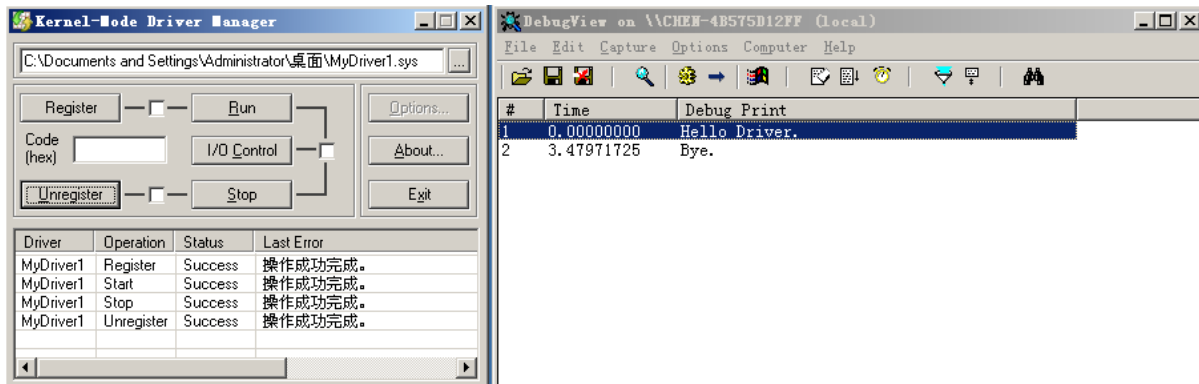


### 1.2.2 使用驱动程序

将编译好的驱动程序、KdmManager、DebugView拖到XP虚拟机中（在虚拟机中去调试驱动比实体机中方便），使用KdmManager加载驱动，按下图所示步骤依次进行，于此同时也要打开DebugView，同时开启监听内核的选项：



当我们依次进行注册、运行、停止、卸载，可以清晰的在DebugView中观察到我们驱动程序所输出的内容，并且我们也知道了函数执行的流程：当点击Run按钮时进入驱动程序入口函数，当点击Stop按钮时进入自定义的驱动程序卸载函数：



## 1.3 调试驱动程序

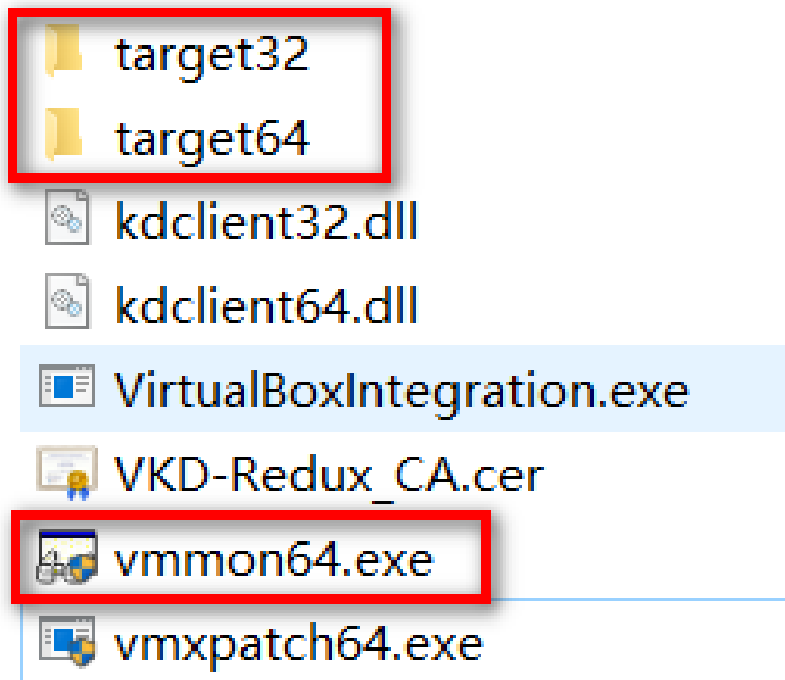
### 1.3.1 调试环境

调试驱动程序不像应用程序一样简单（直接在OD之类的调试工具中下断点），要想调试驱动程序就需要使用双机调试，在虚拟机中运行驱动程序，在实体机上使用0环调试器（也就是WinDbg）进行调试。

之前我们进行双机调试配置时，需要手动修改文件然后重启，这个步骤相对来说比较繁琐，我们可以通过VirtualKD（开源项目地址：<https://github.com/4d61726b/VirtualKD-Redux>）程序来简化整个步骤。

如下图所示就整个程序的目录结构，target32/64目录下的文件就是根据虚拟机操作系统的类型选择并复制进虚拟机的，vmmon64.exe就是主程序，直接实体机打开：

名称



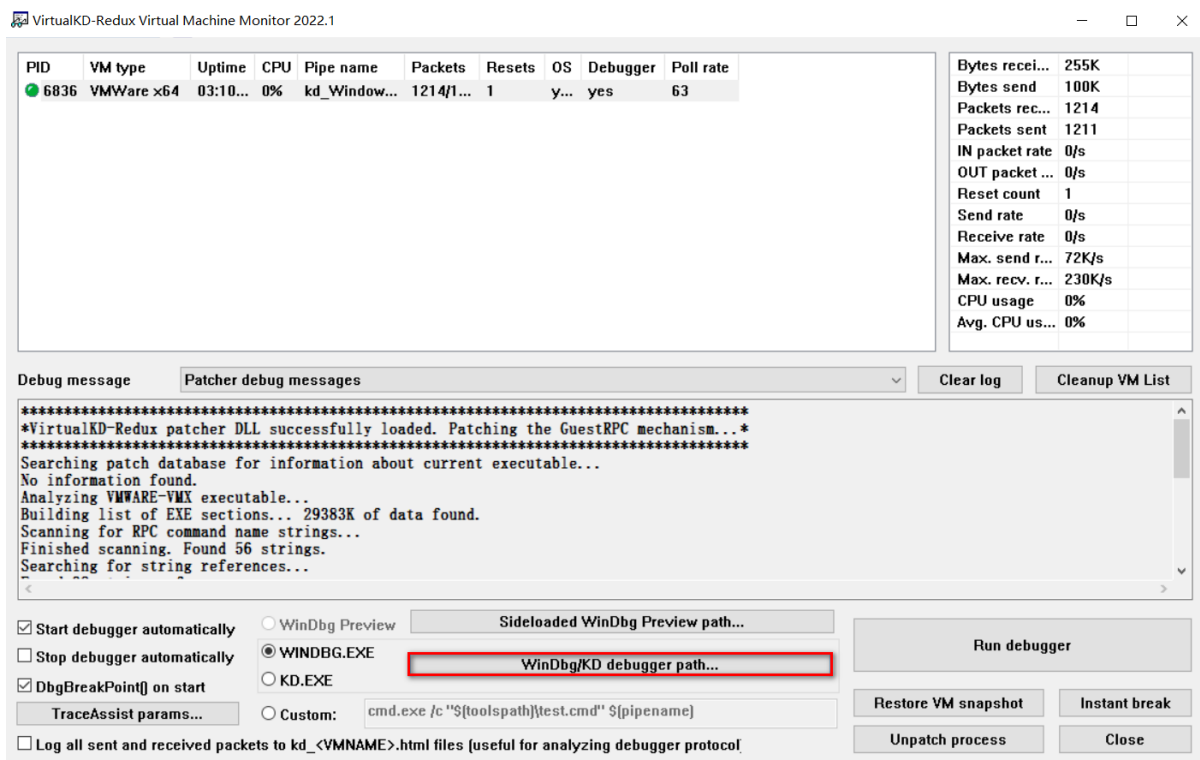
我的虚拟机系统是Windows XP 32位，所以我将target32目录放入虚拟机中，并且按如下步骤操作：

双击运行vminstall.exe-点击Install按钮-点击“是”-系统重启-选择VKD-Redux启动





于此同时打开vmmon64.exe，选择好对应的Windbg程序路径（一般情况下程序会寻找默认Windbg程序路径）：



最后进入调试模式，管道连接上会自动打开Windbg程序：

Kernel 'com:pipe,reset=0,reconnect,port=\\.\pipe\kd\_Windows\_XP\_Professional.vmarevm' - WinDbg:10.0.17763.1 AMD64

File Edit View Debug Window Help

Command

```

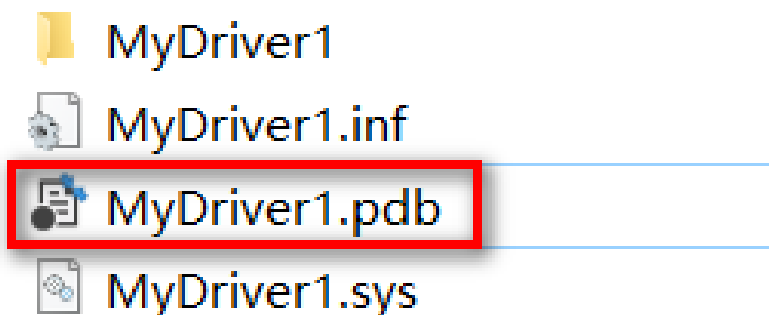
*****
* You are seeing this message because you pressed either
*   CTRL+C (if you run console kernel debugger) or,
*   CTRL+BREAK (if you run GUI kernel debugger),
* on your debugger machine's keyboard.
*
*   THIS IS NOT A BUG OR A SYSTEM CRASH
*
* If you did not intend to break into the debugger, press the "g" key, then
* press the "Enter" key now. This message might immediately reappear. If it
* does, press "g" and "Enter" again.
*****
nt!RtlpBreakWithStatusInstruction:
8052b5d8 cc          int     3
kd> g
ERROR: DavReadRegistryValues/RegQueryValueExW(4). WStatus = 5
ERROR: DavReadRegistryValues/RegQueryValueExW(5). WStatus = 5
ERROR: DavReadRegistryValues/RegQueryValueExW(6). WStatus = 5
watchdog!WdUpdateRecoveryState: Recovery enabled.
Break instruction exception - code 80000003 (first chance)
*****
* You are seeing this message because you pressed either
*   CTRL+C (if you run console kernel debugger) or,
*   CTRL+BREAK (if you run GUI kernel debugger),
* on your debugger machine's keyboard.
*
*   THIS IS NOT A BUG OR A SYSTEM CRASH
*
* If you did not intend to break into the debugger, press the "g" key, then
* press the "Enter" key now. This message might immediately reappear. If it
* does, press "g" and "Enter" again.
*****

```

### 1.3.2 PDB文件

无论是OD还是Windbg，都需要符号文件的加持才能在逆向时获得到更多的信息，例如你用OD去调试程序时经常可以看到一些Windows API是以函数名称的形式存在于反汇编里的，这就是符号文件起的作用。

PDB文件也就是符号文件，我们在编译程序时候，只要不取消调试信息的输出，默认情况下是在编译输出的目录中找到所编译程序的PDB文件的，例如我们上文中编译的驱动程序的输出目录下就有对应的PDB文件。



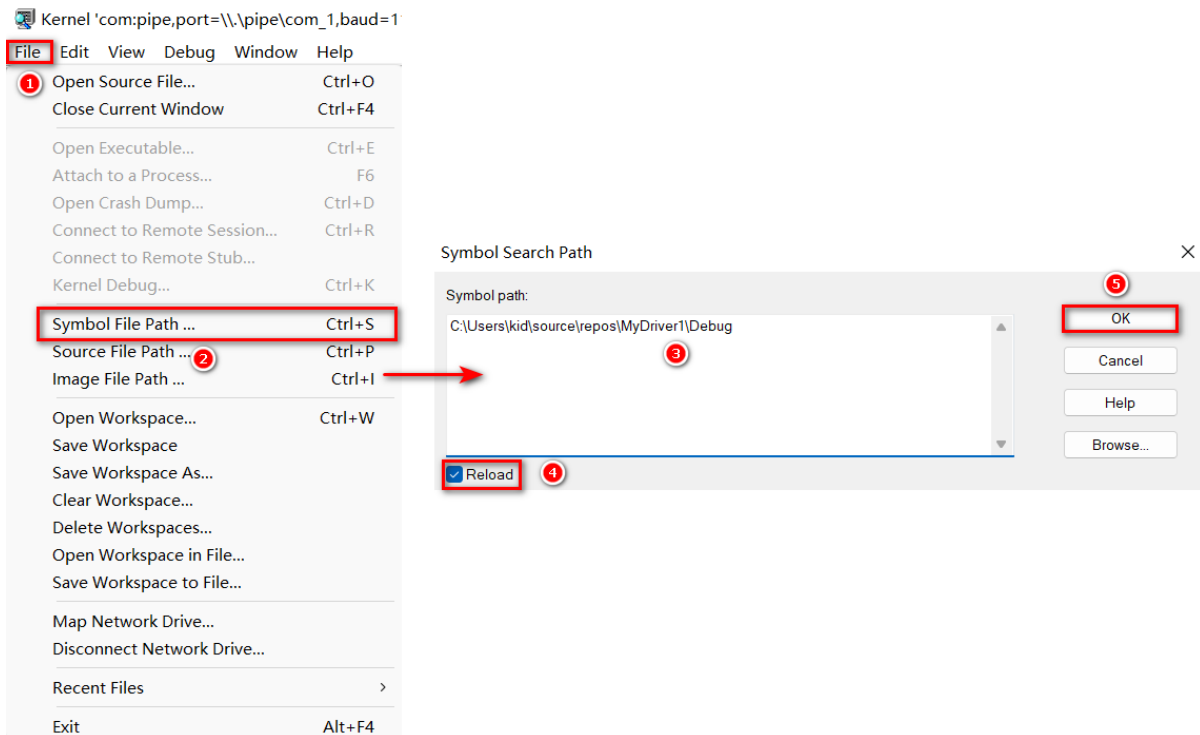
我们要想在Windbg中调试驱动程序就需要这个PDB文件，但是在这之前我们需要先在驱动代码内写上内联汇编，用于断点：

// 驱动程序入口函数

```
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    _asm
    {
        int 3
        mov eax, eax
    }
    DbgPrint("Hello Driver. \n");

    // 设置一个卸载函数, 当驱动卸载时触发
    DriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

编译之后将文件放入虚拟机, 以及在Windbg重载符号文件, 按如下图操作填入PDB文件所在路径即可:



接着在虚拟机中用KdmManager加载并运行驱动程序, 运行时 (KdmManager软件中点击Run按钮) 就会在Windbg中断下来, 如下图在Windbg中新建了一个窗口 (左边的窗口) 展示当前驱动程序断点的位置, 并且都是以源代码形式展示给我们的, 这种效果与我们在调试应用程序时用VS是一样的:

```
c:\users\kid\source\repos\mydriver1\mydriver1\mydriver.c
#include "ntddk.h" // 必须要包含的头文件

// 自定义的驱动程序卸载函数
VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
    DbgPrint("Bye. \n");
}

// 驱动程序入口函数
NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    _asm
    {
        int 3
        mov eax, eax
    }
    DbgPrint("Hello Driver. \n");
    // 设置一个卸载函数, 当驱动卸载时触发
    DriverObject->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

```
*****
* If you did not intend to break into the debugger, press the "g" key, then *
* press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again. *
*****
nt!RtlpBreakWithStatusInstruction:
8052b5d8 cc          int     3
kd> g
ERROR: DavReadRegistryValues/RegQueryValueExW(4). WStatus = 5
ERROR: DavReadRegistryValues/RegQueryValueExW(5). WStatus = 5
ERROR: DavReadRegistryValues/RegQueryValueExW(6). WStatus = 5
watchdog!WdUpdateRecoveryState: Recovery enabled.
Break instruction exception - code 80000003 (first chance)
*****
* You are seeing this message because you pressed either *
* CTRL+C (if you run console kernel debugger) or, *
* CTRL+BREAK (if you run GUI kernel debugger), *
* on your debugger machine's keyboard. *
*****
THIS IS NOT A BUG OR A SYSTEM CRASH
*****
* If you did not intend to break into the debugger, press the "g" key, then *
* press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again. *
*****
nt!RtlpBreakWithStatusInstruction:
8052b5d8 cc          int     3
0: kd> g
Break instruction exception - code 80000003 (first chance)
MyDriver!DriverEntry+0x3:
bab99003 cc          int     3
*****
```

1.4 基础内容

1.4.1 内核API的使用

在应用层编程时我们可以通过包含Windows.h这个头文件来使用Windows提供的API，但是在内核编程时我们不能使用应用层的API，而要使用内核专用的API，所以我们需要包含的头文件就变成了ntddk.h（需要安装好WDK）。

```
1 #include <ntddk.h> // 必须要包含的头文件
2
3 #include <windows.h> // 无法包含
4
5
```

无法打开源文件 "windows.h"

与应用程序开发一样，我们想要去了解某个内核API的信息，也需要查阅文档，老版本的WDK安装之后会自带文档信息，较新一点的WDK不会自带，我们需要去官网查看：<https://docs.microsoft.com/zh-cn/windows-hardware/drivers/ddi/>



1.4.2 未导出函数的使用

WDK文档里只包含了导出的函数信息，对于未导出的函数是查阅不到相关资料的（我们也可以将其称之为未文档化函数）。如果我们想使用未导出的函数，需要定义一个函数指针，并且为函数指针提供正确的函数地址就可以使用了。未导出函数的地址可以通过特征码搜索、解析内核的PDB文件来找到。

### 1.4.3 基本数据类型的使用

在内核编程的时候，必须要遵守WDK的编码习惯，例如无符号类型不要在类型前加上关键词unsigned，而是要遵循WDK自己的类型：

WDK的写法	表达的意思
ULONG	unsigned long
UCHAR	unsigned char
UINT	unsigned int
VOID	void
PULONG	unsigned long *
PUCHAR	unsigned char *
PUNIT	unsigned int *
PVOID	void *

如果你按后者去写，在不同的平台上移植代码可能会导致数据宽度不同，要修改到相同宽度就需要修改代码，而按WDK的写法就可以减少这种不必要的情况。

### 1.4.4 返回值

大部分内核函数的返回值都是NTSTATUS类型，它本质是一个宏，里面包含的类型有很多，如下三个就是常见的返回值：

宏名称	实际值	含义
STATUS_SUCCESS	0x00000000	成功
STATUS_INVALID_PARAMETER	0xC000000D	参数无效
STATUS_BUFFER_OVERFLOW	0x80000005	缓冲区长度不够

当你调用的内核函数返回结果不是STATUS\_SUCCESS，那就说明函数在执行时遇到了问题，具体的问题可以根据返回值在ntstatus.h头文件中去寻找：

```
MyDriver.c
ntstatus.h
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\shared\ntstatus.h

163 //
164 // MessageText:
165 //
166 // STATUS_SUCCESS
167 //
168 #define STATUS_SUCCESS ((NTSTATUS)0x00000000L) // ntsubauth
169
170 //
171 // MessageId: STATUS_WAIT_1
172 //
173 // MessageText:
174 //
175 // STATUS_WAIT_1
176 //
177 #define STATUS_WAIT_1 ((NTSTATUS)0x00000001L)
178
179 //
180 // MessageId: STATUS_WAIT_2
```

通过宏的名称也能知道个大概，如果仍然不知道问题的含义，可以在微软的WDK文档中去搜索相关宏名称。

1.4.5 内核中的异常处理

在内核中一个小小的错误就可能导致蓝屏，例如我们去读写一个无效的内存地址。为了让自己的内核程序更加健壮，在编写内核程序时要使用到异常处理。

在Windows下提供了结构化异常处理机制，编译器普遍都支持，如下就是代码使用方法：

```
1  __try
2  {
3      // 填入可能要出错的代码
4  }
5  __except (filter_value)
6  {
7      // 填入出错后要执行的代码
8  }
```

如上示例中的filter\_value，就是当内核程序出现异常时决定程序如何执行的，一般有这三种情况：

宏名称	实际值	含义
EXCEPTION_EXECUTE_HANDLER	1	进入except代码块执行
EXCEPTION_CONTINUE_SEARCH	0	不处理异常，由上一层调用函数处理
EXCEPTION_CONTINUE_EXECUTION	-1	继续执行错误处的代码

### 1.4.6 常用的内核内存函数

编写内核程序是无法避免去操作内存的，对内存的使用主要就是申请、设置、拷贝、释放，常用的内核函数如下（应用层与内核层的对比）：

应用	内核
malloc	ExAllocatePool
memset	RtlFillMemory
memcpy	RtlMoveMemory
free	ExFreePool

### 1.4.7 内核字符串种类

在内核中我们的字符串种类变多了，有CHAR、WCHAR、ANSI\_STRING、UNICODE\_STRING。前两个就是char和wchar\_t在内核的写法，但是一般不建议使用这种写法而使用后两个（后两个即前两个的升级版），因为在内核中使用内存是要非常小心的，如果使用前两个类型字符串是没有控制长度的，即字符串到0x0或两个0x0就截止，但如果读写字符串时没有遵循这个规则读取越界了就会导致蓝屏。

ANSI\_STRING字符串的定义如下：

```

1  typedef struct _STRING {
2      USHORT Length;
3      USHORT MaximumLength;
4      #ifdef MIDL_PASS
5          [size_is(MaximumLength), length_is(Length) ]
6      #endif // MIDL_PASS
7      _Field_size_bytes_part_opt_(MaximumLength, Length) PCHAR Buffer;
8  } STRING;
9
10 typedef STRING ANSI_STRING;
```

UNICODE\_STRING字符串的定义如下：

```

1  typedef struct _UNICODE_STRING {
2      USHORT Length;
3      USHORT MaximumLength;
4      #ifdef MIDL_PASS
5          [size_is(MaximumLength / 2), length_is((Length) / 2) ] USHORT *
6      Buffer;
7      #else // MIDL_PASS
8          _Field_size_bytes_part_opt_(MaximumLength, Length) PWCH Buffer;
9      #endif // MIDL_PASS
```

```
9 } UNICODE_STRING;
```

我们从这两个结构体的定义就可以看到他们的结构是一样的，我们去读写字符串时就可以根据Buffer+Length这两个成员精确的读取字符串，避免出现读取越界导致蓝屏的问题。

1.4.8 内核字符串常用函数

字符串的操作就是创建、复制、比较、转换，但由于编码问题在内核中也有不同的函数表达：

ANSI_STRING字符串	UNICODE_STRING字符串	含义
RtlInitAnsiString	RtlInitUnicodeString	创建字符串
RtlCopyString	RtlCopyUnicodeString	字符串复制
RtlCompareString	RtlCompareUnicodeString	字符串比较
RtlAnsiStringToUnicodeString	RtlUnicodeStringToAnsiString	编码转换

1.5 内核空间与模块

1.5.1 内核空间

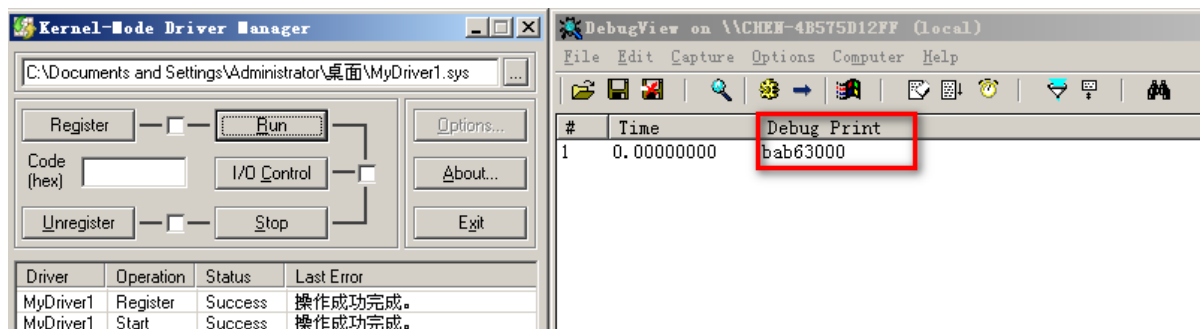
在之前的学习中我们反复提到一个进程有4GB大小的内存空间，低2G是程序自己的，高2G是共享的，也就是内核空间。现在我们已经知道一个简单的驱动程序从开发到运行的流程，可以做一个实验来论证“高2G是共享的”。

我们可以先编写一个驱动程序，主要作用是定义一个全局变量，然后运行时输出其地址：

```
9  ULONG a = 0x12345678;
10
11  // 驱动程序入口函数
12  NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject,
13  PUNICODE_STRING RegistryPath)
14  {
15      DbgPrint("%x \n", &a);
16      // 设置一个卸载函数，当驱动卸载时触发
17      DriverObject->DriverUnload = DriverUnload;
18      return STATUS_SUCCESS;
19  }
```



接着我们在虚拟机中运行该驱动程序获取全局变量的地址：



得到地址为0xBAB63000，然后在Windbg中使用!process 0 0随便找个程序，进入它的内存空间，查看该内存空间中的地址0xBAB63000内容：

```
Failed to get VadRoot
PROCESS 898125f8 SessionId: 0 Cid: 0598 Peb: 7ffd5000 ParentCid: 07d8
DirBase: 0a3c0360 ObjectTable: e197e670 HandleCount: 59.
Image: Dbgview.exe
```

```
0: kd> .process 898125f8
Implicit process is now 898125f8
WARNING: .cache forcedecodeuser is not enabled
0: kd> dd BAB63000
bab63000 12345678 2067c305 df983cfa 00000000
bab63010 00000000 00000000 00000000 00000000
bab63020 00000000 00000000 00000000 00000000
bab63030 00000000 00000000 00000000 00000000
bab63040 00000000 00000000 00000000 00000000
bab63050 00000000 00000000 00000000 00000000
bab63060 00000000 00000000 00000000 00000000
bab63070 00000000 00000000 00000000 00000000
```

最终结果如上所示，我们看到在Dbgview.exe进程的内存空间中，其地址0xBAB63000内容存储着我们驱动程序定义的内容，也就证实了4GB空间中高2G是共享的，这高2G的空间就是的内核空间。

## 1.5.2 内核模块

由于硬件的种类很多，导致系统内核无法完全兼容所有的迎接，所以微软提供了接口，让开发人员按规定格式编写驱动程序来支持自己的硬件；这些驱动程序每一个我们都可以当作是一个模块，也就可以称之为内核模块，它们遵循PE结构，并且可以加载到内核中。

我们之前编写驱动程序的时候必须要写好两个函数，一个是入口函数，一个是卸载函数。在入口函数中，有两个形参，分别是PDRIVER\_OBJECT、PUNICODE\_STRING。

我们来看一下PDRIVER\_OBJECT，它是一个结构体，我们称之为驱动对象，可以通过VS或者Windbg去查看这个结构体的成员：

```

0: kd> dt _DRIVER_OBJECT
nt!_DRIVER_OBJECT
+0x000 Type           : Int2B
+0x002 Size           : Int2B
+0x004 DeviceObject   : Ptr32 _DEVICE_OBJECT
+0x008 Flags          : Uint4B
+0x00c DriverStart    : Ptr32 Void
+0x010 DriverSize     : Uint4B
+0x014 DriverSection  : Ptr32 Void
+0x018 DriverExtension : Ptr32 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING
+0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
+0x028 FastIoDispatch : Ptr32 _FAST_IO_DISPATCH
+0x02c DriverInit     : Ptr32 long
+0x030 DriverStartIo  : Ptr32 void
+0x034 DriverUnload   : Ptr32 void
+0x038 MajorFunction  : [28] Ptr32 long

```

这个结构体里主要包含驱动程序的一些基础信息，具体的可以看如下代码注释：

```

1  typedef struct _DRIVER_OBJECT {
2      CSHORT Type;
3      CSHORT Size;
4      PDEVICE_OBJECT DeviceObject;
5      ULONG Flags;
6      PVOID DriverStart; // 结构体对应的驱动程序在内核空间的位置
7      ULONG DriverSize; // 结构体对应的驱动程序的大小
8      PVOID DriverSection; // 指针，指向_LDR_DATA_TABLE_ENTRY结构体
9      PDRIVER_EXTENSION DriverExtension;
10     UNICODE_STRING DriverName; // 结构体对应的驱动程序的名字
11     PUNICODE_STRING HardwareDatabase;
12     PFAST_IO_DISPATCH FastIoDispatch;
13     PDRIVER_INITIALIZE DriverInit;
14     PDRIVER_STARTIO DriverStartIo;
15     PDRIVER_UNLOAD DriverUnload; // 定义驱动程序卸载函数的地址
16     PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
17
18 } DRIVER_OBJECT;
19 typedef struct _DRIVER_OBJECT *PDRIVER_OBJECT;

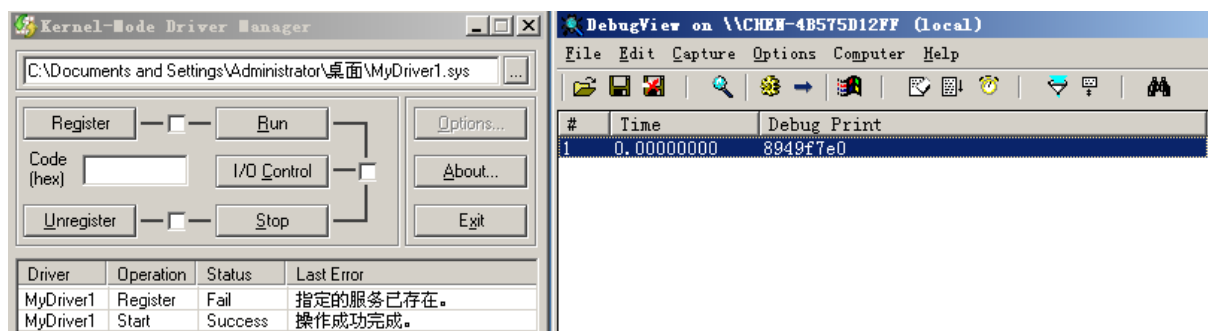
```

我们可以编写一个驱动程序输出对应驱动对象的地址来看下这几个成员：

```

1  NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
   RegistryPath)
2  {
3
4      DbgPrint("%x \n", DriverObject);
5      // 设置一个卸载函数，当驱动卸载时触发
6      DriverObject->DriverUnload = DriverUnload;
7      return STATUS_SUCCESS;
8  }

```



在Windbg中使用如下格式命令可以很直观的看到结构体中每个成员对应的内容：

1dt \_DRIVER\_OBJECT 驱动对象地址

```
0: kd> dt _DRIVER_OBJECT 8949f7e0
nt!_DRIVER_OBJECT
+0x000 Type           : 0n4
+0x002 Size           : 0n168
+0x004 DeviceObject   : (null)
+0x008 Flags          : 0x12
+0x00c DriverStart    : 0xbac88000 Void
+0x010 DriverSize     : 0x6000
+0x014 DriverSection  : 0x896d3730 Void
+0x018 DriverExtension : 0x8949f888 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING "\Driver\MyDriver1"
+0x024 HardwareDatabase : 0x8067c258 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xbac8c000 long MyDriver1!GsDriverEntry+0
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : 0xbac89030 void MyDriver1!DriverUnload+0
+0x038 MajorFunction  : [28] 0x804f5282 long nt!IopInvalidDeviceRequest+0
```

我们也可以利用DriverSection指向的结构体获取其他内核模块的信息，因为它指向的是\_LDR\_DATA\_TABLE\_ENTRY结构体，在该结构体内有一个成员InLoadOrderLinks是双链表，它记录着前一个和下一个内核模块的\_LDR\_DATA\_TABLE\_ENTRY结构体地址：

```

0: kd> dt _LDR_DATA_TABLE_ENTRY 0x896d3730
nt!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x896c1ba8 - 0x897112e8 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0xffffffff - 0xffffffff ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x630069 - 0x0 ]
+0x018 DllBase : 0xbac88000 Void
+0x01c EntryPoint : 0xbac8c000 Void
+0x020 SizeOfImage : 0x6000
+0x024 FullDllName : _UNICODE_STRING "\\??\\C:\\Documents and Settings\\Administrator\\桌面\\MyDriver1.sys"
+0x02c BaseDllName : _UNICODE_STRING "MyDriver1.sys"
+0x034 Flags : 0x9104000
+0x038 LoadCount : 1
+0x03a TlsIndex : 0x49
+0x03c HashLinks : _LIST_ENTRY [ 0xffffffff - 0x10087 ]
+0x03c SectionPointer : 0xffffffff Void
+0x040 CheckSum : 0x10087
+0x044 TimeDateStamp : 0xffffffffe
+0x044 LoadedImports : 0xffffffffe Void
+0x048 EntryPointActivationContext : (null)
+0x04c PatchInformation : 0x0079004d Void
0: kd> dt _LDR_DATA_TABLE_ENTRY 0x896c1ba8
nt!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x8055d700 - 0x896d3730 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0xb1159750 - 0x1 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x018 DllBase : 0xb1154000 Void
+0x01c EntryPoint : 0xb117be85 Void
+0x020 SizeOfImage : 0x2a000
+0x024 FullDllName : _UNICODE_STRING "\\SystemRoot\\system32\\drivers\\kmixer.sys"
+0x02c BaseDllName : _UNICODE_STRING "kmixer.svs"

```

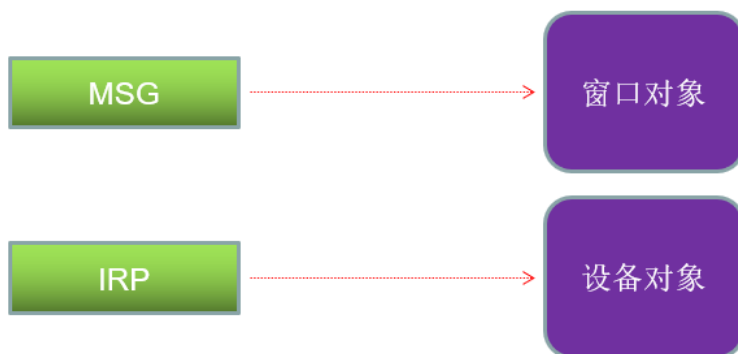
## 1.6 应用与驱动通信

本章我们主要了解在正常的开发中R3应用是如何与R0驱动进行通信的，在这之前我们需要先学习一些新的概念来补全前置内容。

### 1.6.1 设备对象

我们在开发窗口应用程序时，消息被封装成一个结构体MSG，在开发内核驱动程序时，消息被封装成另外一个结构体IRP（I/O Request Package，输入或输出请求包）。

在窗口应用程序中，能够接收MSG消息的只能是窗口对象，在内核驱动程序中，能够接受IRP消息的只能是设备对象。



## 1.6.2 通信实现

### 创建设备对象

正常情况下，一个设备对象是对应一个设备的，如：鼠标、键盘。但是设备对象也可以是一个抽象的概念，不对应到具体某个硬件，也就是我们可以使用如下代码去创建一个设备对象。

```

1  // 创建设备名称
2  UNICODE_STRING DeviceName;
3  RtlInitUnicodeString(&DeviceName, L"\\Device\\MyDevice"); // 设备名称
4  PDEVICE_OBJECT pDeviceObj = NULL;
5  // 创建设备对象
6  NTSTATUS ioCreateDevRet = IoCreateDevice(
7      DriverObject, // 调用方驱动程序对象
8      0,
9      &DeviceName, // 设备名称
10     FILE_DEVICE_UNKNOWN, // 设备类型，当前不与某个具体设备挂钩，所以类型为UNKNOWN
11     FILE_DEVICE_SECURE_OPEN, // 设备属性，大多数驱动程序仅指定
        FILE_DEVICE_SECURE_OPEN 属性，这可确保将相同的安全设置应用到设备的命名空间中的任何
        打开的请求
12     FALSE, // 设备对象是否表示独占设备，如果启用了对设备的独占访问，则一次只能打开设备
        的一个句柄
13     &pDeviceObj // 创建的设备对象，指向接收指向新创建的 DEVICE_OBJECT 结构的指针的
        变量的指针
14 );
15 // 判断IoCreateDevice函数是否执行成功
16 if (ioCreateDevRet != STATUS_SUCCESS)
17 {
18     DbgPrint("IoCreateDevice Error, code: %s\\n", ioCreateDevRet);
19     return ioCreateDevRet;
20 }

```

### 数据交换配置

创建好设备对象之后，就需要设置0环和3环交换数据的方式，有以下三种：

1. 缓冲区读写（DO\_BUFFERED\_IO），操作系统将应用程序提供缓冲区的数据直接复制到内核模式下的地址中；
2. 直接读写（DO\_DIRECT\_IO），操作系统会将用户模式下的缓冲区锁住，然后操作系统将这段缓冲区在内核模式地址再次映射一遍，这样用户模式的缓冲区和内核模式的缓冲区指向的是同一区域的物理内存，缺点就是要单独占用物理页面；
3. 其他读写，在调用IoCreateDevice函数创建设备后不设置交换数据模式即默认为其他方式读写，在使用其他方式读写设备时，派遣函数直接读写应用程序提供的缓冲区地址。在驱动程序中，**直接操作应用程序的缓冲区地址是很危险的（不建议使用）**，只有驱动程序与应用程序运行在相同线程上下文的情况下，才能使用这种方式。

我们可以通过设备对象的Flags成员设置交换数据的方式：

```
1 // 设置交换数据方式
2 pDeviceObj->Flags |= DO_BUFFERED_IO; // 缓冲区读写
```

设备符号链接

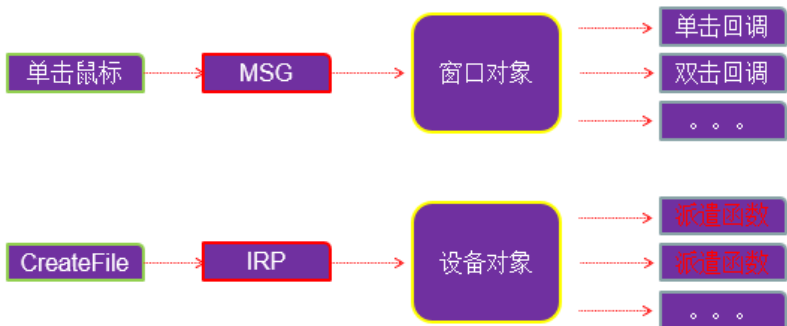
在上述代码中我们创建的设备名称是给0环使用的，如果要在3环访问到设备就需要通过符号链接，我们可以理解为它就是一个设备的别名，如果不这样设置则在3环是无法访问到设备的。我们可以通过IoCreateSymbolicLink函数来创建符号链接，它与设备名称一样，传递的名称需要以UNICODE编码。

```
1 // 创建符号链接
2 UNICODE_STRING SymbolicLinkName;
3 RtlInitUnicodeString(&SymbolicLinkName, L"\\??\\MyTestDriver");
4 IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
```

需要注意的是，在内核模式下符号链接的名称是以“\\??”开头，但是在用户模式下则是以“\\.\\”开头，所以在3环代码中符号链接名称应写为“\\.\\MyTestDriver”。（记得在实际代码中转义特殊符号）

派遣函数

接下来就是编写IRP类型对应的派遣函数了，在这之前我们回顾一下窗口程序的消息流程，当你单击鼠标会产生MSG消息，该消息发送给窗口对象，在窗口对象内会根据消息序号找到对应的处理函数进行处理。同理，在驱动程序中，当你在3环使用了某个函数就会产生IRP消息，该消息发送给设备对象，在设备对象内会根据消息类型选择对应的派遣函数处理。



常见的IRP类型与其对应的3环下的函数及作用如下所示：

IRP类型	来源函数	函数作用
IRP_MJ_CREATE	CreateFile	打开设备
IRP_MJ_READ	ReadFile	从设备中读取数据
IRP_MJ_WRITE	WriteFile	从设备中写入数据
IRP_MJ_CLOSE	CloseHandle	关闭设备

IRP类型	来源函数	函数作用
IRP_MJ_DEVICE_CONTROL	DeviceIoControl	设备控制，比读取、写入操作更加灵活
IRP_MJ_POWER	X	在操作系统处理电源消息时产生该类型
IRP_MJ_SHUTDOWN	X	关闭系统前会产生该类型

在驱动对象中有一个成员MajorFunction，它是一个具有28个成员的数组，对应着就是28种IRP类型：

```

#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
#define IRP_MJ_LOCK_CONTROL     0x11
#define IRP_MJ_CLEANUP          0x12
#define IRP_MJ_CREATE_MAILSLOT  0x13
#define IRP_MJ_QUERY_SECURITY   0x14

```

...

所以我们想注册某个IRP类型对应的派遣函数时候可以使用如下格式：

1	DriverObject->MajorFunction[IRP类型] = 派遣函数名;
---	---

派遣函数也是有一个固定格式的：



```

1  NTSTATUS MyDispatchFunction(PDEVICE_OBJECT pDevObj, PIRP pIrp)
2  {
3      // 业务代码
4
5      // 设置返回状态
6      pIrp->IoStatus.Status = STATUS_SUCCESS; // Get'Last()得到的就是这个值
7      pIrp->IoStatus.Information = 0; // 返回给3环多少数据, 没有则填0
8      IoCompleteRequest(pIrp, IO_NO_INCREMENT);
9      return STATUS_SUCCESS;
10 }

```

我们可以先定义好CreateFile和CloseHandle的派遣函数（实际测试发现是必须要定义，否则无法打开设备）：

```

1  // 注册派遣函数
2  DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateDispatchFunc;
3  DriverObject->MajorFunction[IRP_MJ_CLOSE] = CloseDispatchFunc;
4
5  NTSTATUS CreateDispatchFunc(PDEVICE_OBJECT pDevObj, PIRP pIrp)
6  {
7      DbgPrint("CreateDispatchFunc ... \n");
8      // 设置返回状态
9      pIrp->IoStatus.Status = STATUS_SUCCESS;
10     pIrp->IoStatus.Information = 0; // 返回给3环多少数据, 没有则填0
11     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
12     return STATUS_SUCCESS;
13 }
14
15 NTSTATUS CloseDispatchFunc(PDEVICE_OBJECT pDevObj, PIRP pIrp)
16 {
17     DbgPrint("CloseDispatchFunc ... \n");
18     // 设置返回状态
19     pIrp->IoStatus.Status = STATUS_SUCCESS;
20     pIrp->IoStatus.Information = 0; // 返回给3环多少数据, 没有则填0
21     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
22     return STATUS_SUCCESS;
23 }

```

## 灵活通信

为了更灵活的通信，我们可以使用DeviceIoControl函数，该函数在应用层使用，可以用来与内核层传输数据。我的应用层代码如下，根据代码注释我们知道该函数的参数作用，其中操作码我们也可以理解为一个密码，内核层也应定义操作码，根据操作码来执行不同的指令：

```

1  #include <windows.h>
2  #include <winioctl.h>
3  #include <stdio.h>
4
5  HANDLE g_hDevice;
6

```



```

7  int main(int argc, char* argv[])
8  {
9      // 通过符号链接打开设备
10     g_hDevice = ::CreateFile("\\\\.\\MyTestDriver", GENERIC_WRITE |
11     GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
12     DWORD errCode = ::GetLastError();
13     if (errCode != 0)
14     {
15         printf("Error: %d \n", errCode);
16         exit(0);
17     }
18     // 与驱动进行通信
19     DWORD dwInBuffer = 0x11112222;
20     DWORD tcOutBuffer = 0;
21     DWORD dwLength;
22     ::DeviceIoControl(
23         g_hDevice, // 设备句柄
24         // 操作码
25         CTL_CODE(
26             FILE_DEVICE_UNKNOWN, // 设备类型
27             0x800, // 不可以使用0x0-0x7FF这个范围, 你可以使用0x800-0xFFF
28             METHOD_BUFFERED, // 数据交换的方式
29             FILE_ANY_ACCESS // 访问模式
30         ),
31         &dwInBuffer, // 输入缓冲区地址
32         0x10, // 输入缓冲区长度
33         &tcOutBuffer, // 输出缓冲区地址
34         0x10, // 输出缓冲区长度
35         &dwLength, // 实际返回长度
36         NULL // 指向OVERLAPPED, 此处为NULL
37     );
38     printf("%x \n", tcOutBuffer);
39     CloseHandle(g_hDevice);
40     return 0;
41 }

```

内核层代码如下，在派遣函数DeviceDispatchFunc中，我们通过参数plrp获取IRP数据，并根据操作码进入读取数据、写入数据指令：

```

1  #include <ntddk.h> // 必须要包含的头文件
2
3  // 操作码定义
4  #define OPER CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED,
5  FILE_ANY_ACCESS)
6
7  // 自定义的驱动程序卸载函数
8  VOID DriverUnload(PDRIVER_OBJECT DriverObject)
9  {
10     DbgPrint("Bye. \n");

```

```

10 }
11
12 NTSTATUS CreateDispatchFunc(PDEVICE_OBJECT pDevObj, PIRP pIrp)
13 {
14     DbgPrint("CreateDispatchFunc ... \n");
15     // 设置返回状态
16     pIrp->IoStatus.Status = STATUS_SUCCESS;
17     pIrp->IoStatus.Information = 0; // 返回给3环多少数据, 没有则填0
18     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
19     return STATUS_SUCCESS;
20 }
21
22 NTSTATUS CloseDispatchFunc(PDEVICE_OBJECT pDevObj, PIRP pIrp)
23 {
24     DbgPrint("CloseDispatchFunc ... \n");
25     // 设置返回状态
26     pIrp->IoStatus.Status = STATUS_SUCCESS;
27     pIrp->IoStatus.Information = 0; // 返回给3环多少数据, 没有则填0
28     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
29     return STATUS_SUCCESS;
30 }
31
32 NTSTATUS DeviceDispatchFunc(PDEVICE_OBJECT pDevObj, PIRP pIrp)
33 {
34     DbgPrint("DeviceDispatchFunc ... \n");
35
36     ULONG uWrite;
37     ULONG uRead;
38     NTSTATUS retStatus;
39
40     // 获取IRP数据
41     PIO_STACK_LOCATION pIrpStack = IoGetCurrentIrpStackLocation(pIrp);
42     // 获取操作码
43     ULONG uIoControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;
44     // 获取输入、输出数据的长度
45     // ULONG uDataInLength = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;
46     // ULONG uDataOutLength = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;
47     // 获取缓冲区地址 (输入、输出的缓冲区都是一个)
48     PVOID pIoBuffer = pIrp->AssociatedIrp.SystemBuffer;
49
50     // 判断操作码
51     switch (uIoControlCode)
52     {
53     case OPER:
54         // DbgPrint("Size: %d \n", uDataOutLength);
55         // 从缓冲区读取数据
56         memcpy(&uRead, pIoBuffer, 4);
57         DbgPrint("uRead: %x \n", uRead);
58         // 写入数据至缓冲区
59         uWrite = 0x22334455;

```

```

60         memcpy(pIoBuffer, &uWrite, 4);
61         pIrp->IoStatus.Information = 4; // 返回给3环多少数据, 没有则填0
62         retStatus = STATUS_SUCCESS;
63         break;
64     }
65     // 设置返回状态
66     pIrp->IoStatus.Status = retStatus == STATUS_SUCCESS ? retStatus :
STATUS_SUCCESS;
67     IoCompleteRequest(pIrp, IO_NO_INCREMENT);
68     return STATUS_SUCCESS;
69 }
70
71 // 驱动程序入口函数
72 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
73 {
74     // 创建设备名称
75     UNICODE_STRING DeviceName;
76     RtlInitUnicodeString(&DeviceName, L"\\Device\\MyDevice");
77     PDEVICE_OBJECT pDeviceObj = NULL;
78     // 创建设备对象
79     NTSTATUS ioCreateDevRet = IoCreateDevice(
80         DriverObject, // 调用方驱动程序对象
81         0,
82         &DeviceName, // 设备名称
83         FILE_DEVICE_UNKNOWN, // 设备类型, 当前不与某个具体设备挂钩, 所以类型为
UNKNOWN
84         FILE_DEVICE_SECURE_OPEN, // 设备属性, 大多数驱动程序仅指定
FILE_DEVICE_SECURE_OPEN 属性, 这可确保将相同的安全设置应用到设备的命名空间中的任何
打开的请求
85         FALSE, // 设备对象是否表示独占设备, 如果启用了对设备的独占访问, 则一次只能打
开设备的一个句柄
86         &pDeviceObj // 创建的设备对象, 指向接收指向新创建的 DEVICE_OBJECT 结构的
指针的变量的指针
87     );
88
89     // 判断IoCreateDevice函数是否执行成功
90     if (ioCreateDevRet != STATUS_SUCCESS)
91     {
92         DbgPrint("IoCreateDevice Error, code: %s\\n", ioCreateDevRet);
93         return ioCreateDevRet;
94     }
95
96     // 设置数据交换的方式
97     pDeviceObj->Flags |= DO_BUFFERED_IO;
98
99     // 创建符号链接
100    UNICODE_STRING SymbolicLinkName;
101    RtlInitUnicodeString(&SymbolicLinkName, L"\\??\\MyTestDriver");
102    IoCreateSymbolicLink(&SymbolicLinkName, &DeviceName);
103
104    // 注册派遣函数
105    DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateDispatchFunc;

```

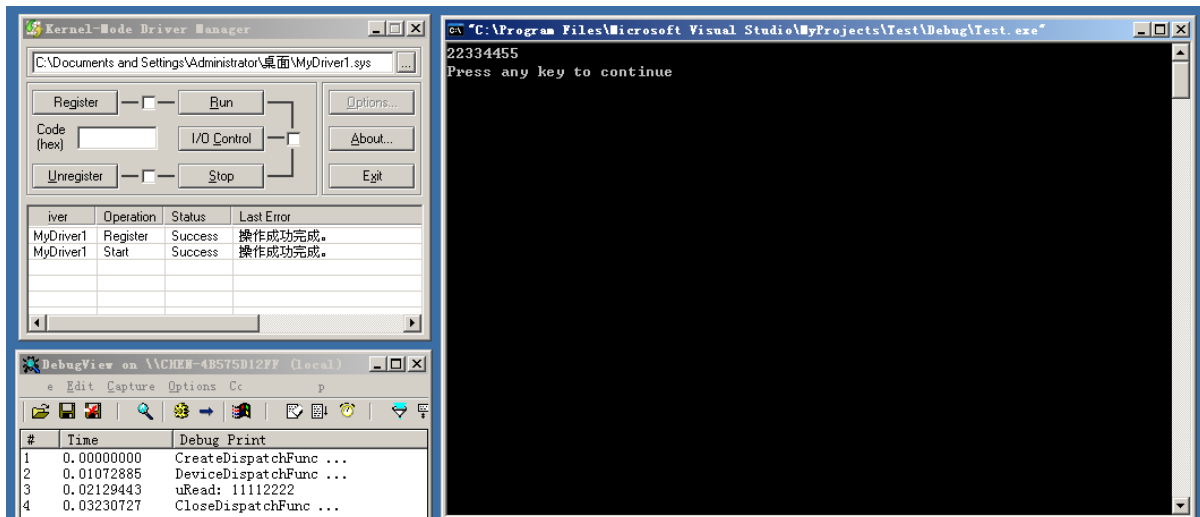
```

106     DriverObject->MajorFunction[IRP_MJ_CLOSE] = CloseDispatchFunc;
107     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DeviceDispatchFunc;

108
109     // 设置一个卸载函数，当驱动卸载时触发
110     DriverObject->DriverUnload = DriverUnload;
111     return STATUS_SUCCESS;
112 }

```

实际运行测试，发现我们可以很顺畅的在3环与0环进行通信：



## 1.7 Hook技术

### 1.7.1 SSDT Hook

SSDT Hook就是根据SSDT（即KeServiceDescriptorTable、KeServiceDescriptorTableShadow）找到系统服务表，并且修改系统服务表的函数地址表的某个函数地址，来达到Hook指定函数的目的。（关于SSDT的内容我们已经在系统调用的学习中了解了）

#### 根据SSDT获取系统服务表

本次实验我们只对第一张系统服务表的函数进行修改，所以我们可以先根据KeServiceDescriptorTable需要获取系统服务表的地址。如下代码所示，我们按成员结构体定义好系统服务表和SSDT，并且导出KeServiceDescriptorTable。

```

1  // 定义系统服务表结构
2  typedef struct _KSYSTEM_SERVICE_TABLE
3  {
4      PULONG ServiceTableBase;
5      PULONG ServiceCounterTableBase;
6      ULONG NumberOfService;
7      PULONG ParamTableBase;

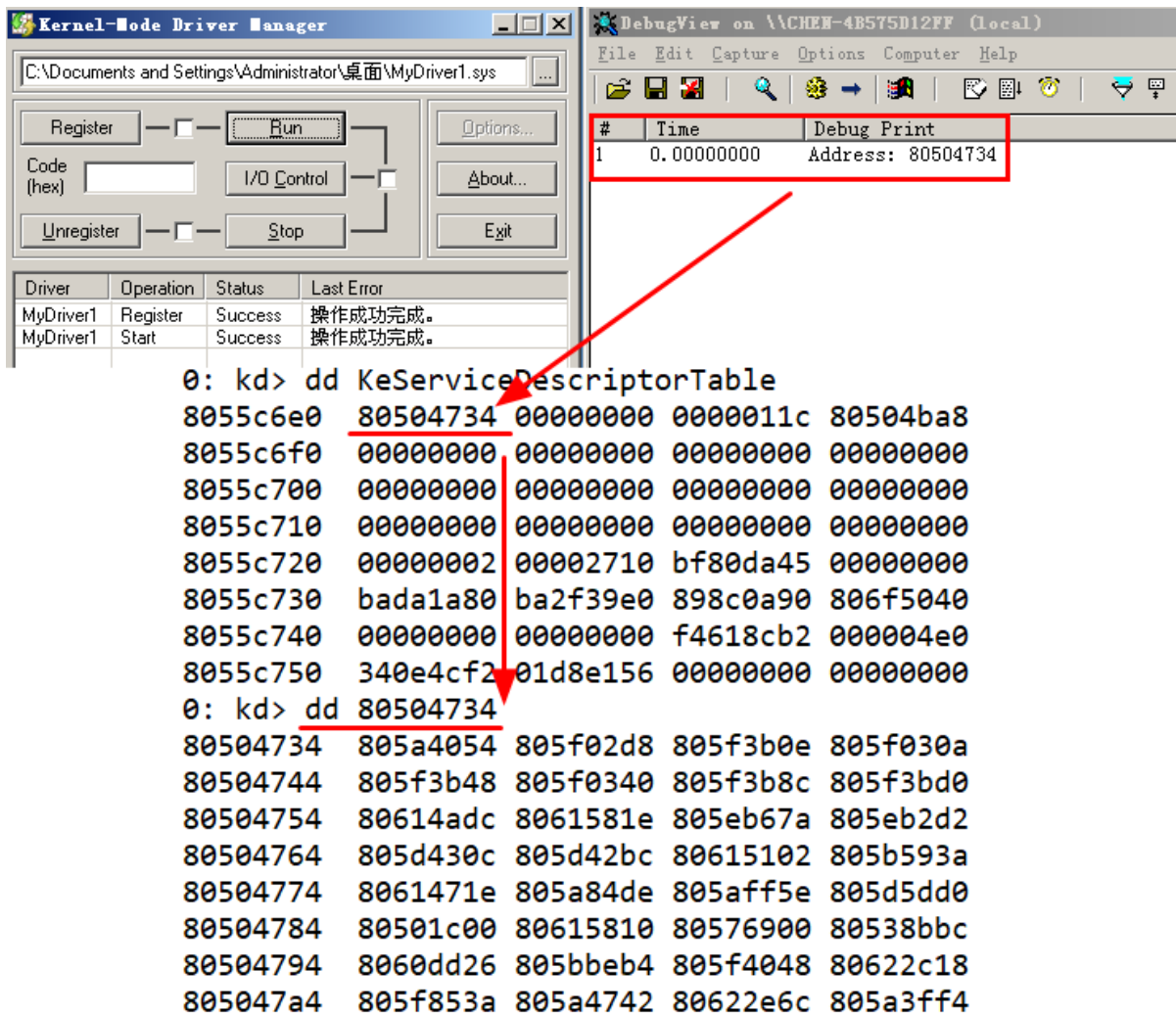
```

```
8 } KSYSTEM_SERVICE_TABLE, *PKSYSTEM_SERVICE_TABLE;  
9  
10 // 定义SSDT结构  
11 typedef struct _KSERVICE_TABLE_DESCRIPTOR  
12 {  
13     KSYSTEM_SERVICE_TABLE KernalFuncTable;  
14     KSYSTEM_SERVICE_TABLE Win32KTable;  
15     KSYSTEM_SERVICE_TABLE NotUsedTableA;  
16     KSYSTEM_SERVICE_TABLE NotUsedTableB;  
17 } KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_DESCRIPTOR;  
18  
19 // 导出由内核模块所导出的SSDT  
20 extern PKSYSTEM_SERVICE_DESCRIPTOR KeServiceDescriptorTable;
```

接着我们可以来驱动入口函数输出函数地址表的地址，来判断我们获取的是否正确：

```
1 DbgPrint("Address: %x\n", KeServiceDescriptorTable->KernalFuncTable.ServiceTableBase);
```

如下图所示，我们获取的函数地址表地址与Windbg展示的是一直的，因此这里我们成功通过SSDT获取了系统服务表：



## 准备要替换的函数

在准备要替换的函数之前，我们要确定替换哪个函数，并且找到其在函数地址表的位置，这里我们选择一个使用的比较多的函数OpenProcess，首先我们在kernel32.dll找到该函数，发现该函数实际是调用ntdll.dll下的NtOpenProcess即ZwOpenProcess，系统调用号为0x7A：

```

.text:7C81E0B9 89 75 F4          mov     [ebp+ObjectAttributes.SecurityQualityOfService], esi
.text:7C81E0BC FF 15 0C 11 80 7C    call    ds:__imp__NtOpenProcess@16 ; NtOpenProcess(x,x,x,x)
; Exported entry 211. NtOpenProcess
; Exported entry 1020. ZwOpenProcess

; _stdcall ZwOpenProcess(x, x, x, x)
public _ZwOpenProcess@16
_ZwOpenProcess@16 proc near
mov     eax, 7Ah ; 'z' ; NtOpenProcess
mov     edx, 7FFE0300h
call    dword ptr [edx]
retn    10h
_ZwOpenProcess@16 endp
  
```

我们可以在函数地址表中找到它，即0x805CAC46：

```
0: kd> dd 80504734+7A*4
8050491c 805cac46 805ecf10 805ecb16 805a9e12
8050492c 80613f36 805c47c6 805caed2 805ecf2e
8050493c 805ecc86 80615e7e 80644128 805c8d34
8050494c 805f72e0 805f2f28 805f3114 805b7da2
8050495c 8060df2e 80576cf0 8061581e 8061581e
8050496c 8053fba6 8060fb00 80610760 80579c3e
8050497c 805bdeda 80579f8c 8060dff6 80576e28
8050498c 80614fba 8057a7f8 805d53ec 805a4e08
```

接着我们可以在Hook前先保存这个地址，这样当卸载驱动时便于还原：

```
1 // 用于保存被Hook的函数地址
2 ULONG OldFuncAddr;
3
4 // 保存被Hook的函数地址
5 OldFuncAddr = KeServiceDescriptorTable->
  >KrnFuncTable.ServiceTableBase[0x7A];
```

接着我们按开发文档上ZwOpenProcess的格式自定义一个函数，并且定义好NtOpenProcess的函数指针在自定义的函数内最终调用原函数：

```
1 // 声明函数指针
2 typedef NTSTATUS(*NTOpenProcess)(
3     PHANDLE ProcessHandle,
4     ACCESS_MASK DesiredAccess,
5     POBJECT_ATTRIBUTES ObjectAttributes,
6     PCLIENT_ID ClientId
7 );
8
9 // 准备的函数
10 NTSTATUS MyNtOpenProcess(PHANDLE ProcessHandle, ACCESS_MASK DesiredAccess,
11     POBJECT_ATTRIBUTES ObjectAttributes, PCLIENT_ID ClientId)
12 {
13     // 自己的代码
14     DbgPrint("Hooking... \n");
15     // 最终还是走向真正的NtOpenProcess
16     return ((NTOpenProcess)OldFuncAddr)(ProcessHandle, DesiredAccess,
17     ObjectAttributes, ClientId);
18 }
```

这里肯定很多人会被NtXXX和ZwXXX搞得头晕目眩，我们简单了解下这两个命名的区别：在用户角度下也就是我们之前在ntdll.dll中看到的NtOpenProcess和ZwOpenProcess它们本质都是一样的，ZwOpenProcess只是NtOpenProcess的一个别名，它们最终都指向同一个地址；在内核角度下ZwOpenProcess不仅仅是NtOpenProcess的别名，我们可以看如下代码，ZwOpenProcess实际上也是一个系统调用的过程跟用户侧的ZwOpenProcess没什么区别，而内核下的NtOpenProcess则是真正有实现的函数，所以我们在上面的代码声明中都是以NtOpenProcess来命名也是为了规范统一化：

```

0: kd> u nt!ZwOpenProcess
nt!ZwOpenProcess:
80500a28 b87a000000      mov     eax,7Ah
80500a2d 8d542404      lea     edx,[esp+4]
80500a31 9c           pushfd
80500a32 6a08         push    8
80500a34 e8080a0400   call    nt!KiSystemService (80541441)
80500a39 c21000       ret     10h
nt!ZwOpenProcessToken:
80500a3c b87b000000      mov     eax,7Bh
80500a41 8d542404      lea     edx,[esp+4]
0: kd> u nt!NtOpenProcess
nt!NtOpenProcess:
805cac46 68c4000000      push    0C4h
805cac4b 6898b44d80      push    offset nt!ObWatchHandles+0x25c (804db498)
805cac50 e81b0ff7ff      call    nt!_SEH_prolog (8053bb70)
805cac55 33f6          xor     esi,esi
805cac57 8975d4        mov     dword ptr [ebp-2Ch],esi
805cac5a 33c0          xor     eax,eax
805cac5c 8d7dd8        lea     edi,[ebp-28h]
805cac5f ab           stos    dword ptr es:[edi]

```

## 修改函数地址表地址

我们准备好了要替换的函数，接着就是修改函数地址表的地址，**直接修改是不可行的，因为系统服务表所在的物理页是只读的，如果要修改物理页的内容，就需要先修改页属性为可写。**

修改物理页读写属性的方式有两种，第一种方法是根据页目录表、页表基址找到PDE、PTE修改其R/W位值为1，就可以让物理页具有读写属性（关于具体的代码实现可以参考不同分页模式下的MmIsAddressValid函数）：

```

1  // 修改物理页属性
2  NTSTATUS ChangePageAttr(ULONG attrValue)
3  {
4      ULONG RCR4 = 0;
5      // 获得CR4寄存器的值
6      _asm
7      {
8          _emit 0x0F
9          _emit 0x20
10         _emit 0xE0
11         mov RCR4, eax
12     }
13     // 根据CR4寄存器的PAE位判断分页模式
14     if (RCR4 & 0x00000200)
15     {
16         DbgPrint("2-9-9-12 \n");
17         // 获取PDE、PTE，并修改R/W
18         *(ULONG64*)(0xC0600000 + (((OldFuncAddr >> 21) << 3) & 0x3FF8)) |=
attrValue;

```



```

19         *(ULONG64*)(0xC0000000 + (((OldFuncAddr >> 12) << 3) & 0x7FFF8))
    |= attrValue;
20     }
21     else
22     {
23         DbgPrint("10-10-12 \n");
24         // 获取PDE、PTE, 并修改R/W
25         *(ULONG*)(0xC0300000 + ((OldFuncAddr >> 20) & 0xFFC)) |=
attrValue;
26         *(ULONG*)(0xC0000000 + ((OldFuncAddr >> 10) & 0x3FFFC)) |=
attrValue;
27     }
28     return STATUS_SUCCESS;
29 }
30
31 // 修改R/W为1
32 ChangePageAttr(0x2);
33 // 修改R/W为0
34 ChangePageAttr(0x0)

```

第二种方法就是通过修改CR0寄存器的WP位，将其设为0就可以关闭写保护无视物理页只读权限进行读写，这样的方法虽然简单但在多核的情况下在代码运行时候有核切换就会存在问题，所以推荐使用的是第一种方法：

```

1  VOID PageProtectOn()
2  {
3      __try
4      {
5          _asm
6          {
7              mov eax, cr0
8              or eax, 10000h
9              mov cr0, eax
10             sti
11         }
12     }
13     __except (1)
14     {
15         DbgPrint("PageProtectOn执行失败!");
16     }
17 }
18
19 VOID PageProtectOff()
20 {
21     __try
22     {
23         _asm
24         {
25             cli
26             mov eax, cr0
27             and eax, not 10000h //and eax, 0xFFEFFFFh
28             mov cr0, eax
29         }

```

```

30     }
31     __except (1)
32     {
33         DbgPrint("PageProtectOff执行失败!");
34     }
35 }

```

接着我们使用下标替换函数地址的代码即可：

```

1  // Hook
2  NTSTATUS HookFuncAddr(ULONG FuncAddr)
3  {
4      ChangePageAttr(0x2);
5      KeServiceDescriptorTable->KrnlfuncTable.ServiceTableBase[0x7A] =
FuncAddr;
6      ChangePageAttr(0x0);
7      return STATUS_SUCCESS;
8  }
9
10 HookFuncAddr((ULONG)MyNtOpenProcess);

```

## 完整代码

最终将所有代码调优一下细节即可，完整代码如下：

```

1  #include <ntddk.h> // 必须要包含的头文件
2
3  // 定义系统服务表结构
4  typedef struct _KSYSTEM_SERVICE_TABLE
5  {
6      PULONG ServiceTableBase;
7      PULONG ServiceCounterTableBase;
8      ULONG NumberOfService;
9      PULONG ParamTableBase;
10 } KSYSTEM_SERVICE_TABLE, *PKSYSTEM_SERVICE_TABLE;
11
12 // 定义SSDT结构
13 typedef struct _KSERVICE_TABLE_DESCRIPTOR
14 {
15     KSYSTEM_SERVICE_TABLE KrnlFuncTable;
16     KSYSTEM_SERVICE_TABLE Wind32KTable;
17     KSYSTEM_SERVICE_TABLE NotUsedTableA;
18     KSYSTEM_SERVICE_TABLE NotUsedTableB;
19 } KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;
20
21 // 导出由内核模块所导出的SSDT
22 extern PKSYSTEM_SERVICE_DESCRIPTOR KeServiceDescriptorTable;
23
24 // 用于保存被Hook的函数地址
25 ULONG OldFuncAddr;

```

```

26
27 // 声明函数指针
28 typedef NTSTATUS(*NTOPENPROCESS)(
29     PHANDLE ProcessHandle,
30     ACCESS_MASK DesiredAccess,
31     POBJECT_ATTRIBUTES ObjectAttributes,
32     PCLIENT_ID ClientId
33 );
34
35 // 准备的函数
36 NTSTATUS MyNtOpenProcess(PHANDLE ProcessHandle, ACCESS_MASK DesiredAccess,
37     POBJECT_ATTRIBUTES ObjectAttributes, PCLIENT_ID ClientId)
38 {
39     // 自己的代码
40     DbgPrint("Hooking... \n");
41     // 最终还是走向真正的OpenProcess
42     return ((NTOPENPROCESS)OldFuncAddr)(ProcessHandle, DesiredAccess,
43     ObjectAttributes, ClientId);
44 }
45
46 // 修改物理页属性
47 NTSTATUS ChangePageAttr(ULONG attrValue)
48 {
49     ULONG RCR4 = 0;
50     // 获得CR4寄存器的值
51     _asm
52     {
53         _emit 0x0F
54         _emit 0x20
55         _emit 0xE0
56         mov RCR4, eax
57     }
58     // 根据CR4寄存器的PAE位判断分页模式
59     if (RCR4 & 0x000000200)
60     {
61         DbgPrint("2-9-9-12 \n");
62         // 获取PDE、PTE, 并修改R/W
63         *(ULONG64*)(0xC0600000 + (((OldFuncAddr >> 21) << 3) & 0x3FF8)) |=
64         attrValue;
65         *(ULONG64*)(0xC0000000 + (((OldFuncAddr >> 12) << 3) & 0x7FFF8))
66         |= attrValue;
67     }
68     else
69     {
70         DbgPrint("10-10-12 \n");
71         // 获取PDE、PTE, 并修改R/W
72         *(ULONG*)(0xC0300000 + ((OldFuncAddr >> 20) & 0xFFC)) |=
73         attrValue;
74         *(ULONG*)(0xC0000000 + ((OldFuncAddr >> 10) & 0xFFFFC)) |=
75         attrValue;
76     }
77     return STATUS_SUCCESS;

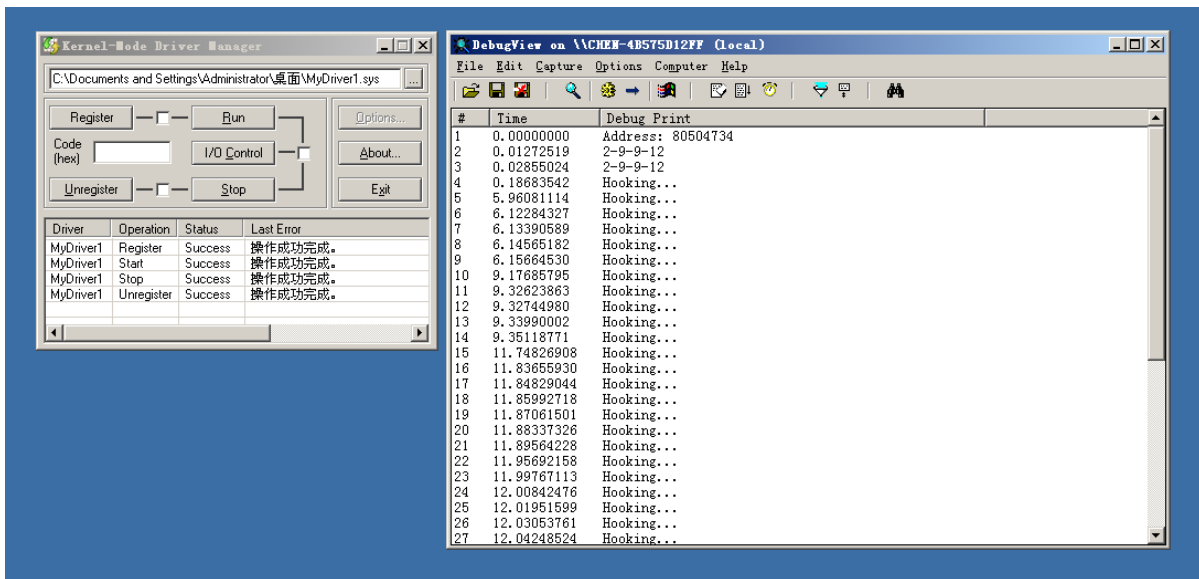
```

```

73     }
74
75     // Hook
76     NTSTATUS HookFuncAddr(ULONG FuncAddr)
77     {
78         ChangePageAttr(0x2);
79         KeServiceDescriptorTable->KrnFuncTable.ServiceTableBase[0x7A] =
FuncAddr;
80         ChangePageAttr(0x0);
81
82         return STATUS_SUCCESS;
83     }
84
85     // 自定义的驱动程序卸载函数
86     VOID DriverUnload(PDRIVER_OBJECT DriverObject)
87     {
88         HookFuncAddr((ULONG)OldFuncAddr);
89         DbgPrint("Bye. \n");
90     }
91
92     // 驱动程序入口函数
93     NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
94     {
95         DbgPrint("Address: %x\n", KeServiceDescriptorTable-
>KrnFuncTable.ServiceTableBase);
96         // 保存被Hook的函数地址
97         OldFuncAddr = KeServiceDescriptorTable-
>KrnFuncTable.ServiceTableBase[0x7A];
98         HookFuncAddr((ULONG)MyNtOpenProcess);
99         // 设置一个卸载函数，当驱动卸载时触发
100        DriverObject->DriverUnload = DriverUnload;
101        return STATUS_SUCCESS;
102    }

```

编译代码，在系统上运行成功实现了Hook：



### 1.7.2 Inline Hook

SSDT Hook的方式是基于系统服务表的，这种方式很容易被发现并且我们只能取Hook系统服务表内的函数，因此我们可以使用Inline Hook的方式来弥补这些缺陷。Inline Hook可以在3环、0环下使用，它的原理就是改变程序的执行流程，例如你可以在某个函数的第一行位置开始替换为JMP指令，跳转到自己的函数地址。除了使用JMP指令改变执行流程，我们还可以使用CALL指令以及PUSH、RET指令的组合。JMP、CALL指令至少占用5个字节，PUSH、RET指令的组合则需要6个字节。

使用Inline Hook时需要注意避免替换全局变量相关的硬编码，并且需要先将通用寄存器压栈保存，然后将被替换的指令加入到自己的指令中以保证程序可以完整执行，最终再JMP回被替换指令所在行的下一行地址。

我们可以使用SSDT Hook的代码来找到要Hook函数的地址，例如此处我想Hook函数NtOpenProcess来监控它的参数，我们不再需要声明函数指针然后写自定义函数，直接写一个自定的函数即可，但也因此我们需要通过其他的方式来获取参数值，我们可以来看一下该函数的实现，似乎并没有用到寄存器，也就表示在这里不是通过寄存器传参，而是通过栈传参，那我们就可以使用EBP寻址来获取参数。

```
0: kd> u NtOpenProcess L10
nt!NtOpenProcess:
805cac46 68c4000000      push    0C4h
805cac4b 6898b44d80      push    offset nt!ObWatchHandles+0x25c (804db498)
805cac50 e81b0ff7ff      call    nt!_SEH_prolog (8053bb70)
805cac55 33f6           xor     esi,esi
805cac57 8975d4         mov     dword ptr [ebp-2Ch],esi
805cac5a 33c0           xor     eax,eax
805cac5c 8d7dd8         lea     edi,[ebp-28h]
805cac5f ab             stos    dword ptr es:[edi]
805cac60 64a124010000    mov     eax,dword ptr fs:[00000124h]
805cac66 8a8040010000    mov     al,byte ptr [eax+140h]
805cac6c 8845cc         mov     byte ptr [ebp-34h],al
805cac6f 84c0           test    al,al
805cac71 0f848f000000    je      nt!NtOpenProcess+0xc0 (805cad06)
805cac77 8975fc         mov     dword ptr [ebp-4],esi
805cac7a a114215680      mov     eax,dword ptr [nt!MmUserProbeAddress (80562114)]
805cac7f 8b4d08         mov     ecx,dword ptr [ebp+8]
```

接着我们需要找到合适的Hook点，也就是我们代码流程切换指令替换的位置，这里我们选择比较方便的 PUSH、RET 指令，也就是要找一个非全局变量相关的位置，如上图标红所示即一个比较不错的 Hook 点，接着我们来编写自定义函数的代码，`__declspec(naked)` 用于表示这是一个裸函数这样编译器就不会为该函数的结尾加入 RET 指令：

```

1  // 准备的函数
2  VOID __declspec(naked) MyNtOpenProcess()
3  {
4      __asm
5      {
6          // 保存寄存器
7          pushad
8          pushfd
9          // 获取传参数
10         mov eax, [ebp + 0x8]
11         mov ProcessHandle, eax
12         mov eax, [ebp + 0xC]
13         mov DesiredAccess, eax
14         mov eax, [ebp + 0x10]
15         mov ObjectAttributes, eax
16         mov eax, [ebp + 0x14]
17         mov ClientId, eax
18     }
19
20     // 调试输出
21     DbgPrint("ProcessHandle: %x, DesiredAccess: %x, ObjectAttributes: %x,
22     ClientId: %x\n", ProcessHandle, DesiredAccess, ObjectAttributes,
23     ClientId);
24
25     __asm
26     {
27         // 还原寄存器
28         popfd
29         popad
30         // 执行被替换的指令，保证运行完整
31         xor eax, eax
32         lea edi, [ebp-0x28]
33         stos dword ptr es:[edi]
34         // 跳转回原函数被替换指令的下一行地址
35         jmp BackAddress
36     }
37 }

```

代码中有很多细节的点，如防止我们的代码中会使用到寄存器先压入原来的寄存器到栈，接着还原，在 JMP 回原函数被替换指令的下一行地址之前我们需要将被替换的指令执行一下，以保证函数运行的完整性。

接着我们需要构造一下修改的指令，即 PUSH、RET 指令，直接使用硬编码的方式即可：

```

1  UCHAR PushRetCmd[6] = { 0 };
2
3  // 硬编码 PUSH 0x12345678

```

```

4  PushRetCmd[0] = 0x68;
5  *(ULONG*)(PushRetCmd + 1) = (ULONG)MyNtOpenProcess;
6  // RET
7  PushRetCmd[5] = 0xC3;

```

再使用RtlMoveMemory来替换函数指令：

```

1  // 修改函数指令
2  NTSTATUS ChangeFuncCmd(UCHAR CmdByte[6])
3  {
4      ChangePageAttr(0x2);
5      RtlMoveMemory((PUCHAR)CmdAddress, CmdByte, 6);
6      ChangePageAttr(0x0);
7
8      return STATUS_SUCCESS;
9  }

```

在驱动入口处我们将该计算的地址，以及指令都填充好即可：

```

1  // 保存被Hook的函数地址
2  FuncAddr = KeServiceDescriptorTable->KrnlfuncTable.ServiceTableBase[0x7A];
3  DbgPrint("FuncAddr: %x \n", FuncAddr);
4  // 计算要替换的函数地址和要JMP的返回地址
5  CmdAddress = FuncAddr + 0x14; // 将要替换的指令地址减去函数首地址得出偏移0x144
6  DbgPrint("CmdAddress: %x \n", CmdAddress);
7  BackAddress = CmdAddress + 0x6; // 替换的指令长度为0x6
8  DbgPrint("BackAddress: %x \n", BackAddress);
9  // 硬编码 PUSH 0x12345678
10 PushRetCmd[0] = 0x68;
11 *(ULONG*)(PushRetCmd + 1) = (ULONG)MyNtOpenProcess;
12 // 硬编码 RET
13 PushRetCmd[5] = 0xC3;
14 // 替换指令
15 ChangeFuncCmd(PushRetCmd);

```

最后别忘记在卸载函数时还原指令以及一些细节上的调整，最终完整代码如下：

```

1  #include <ntddk.h> // 必须要包含的头文件
2
3  // 定义系统服务表结构
4  typedef struct _KSYSTEM_SERVICE_TABLE
5  {
6      PULONG ServiceTableBase;
7      PULONG ServiceCounterTableBase;
8      ULONG NumberOfService;
9      PULONG ParamTableBase;
10 } KSYSTEM_SERVICE_TABLE, *PKSYSTEM_SERVICE_TABLE;
11
12 // 定义SSDT结构
13 typedef struct _KSERVICE_TABLE_DESCRIPTOR

```

```

14 {
15     KSYSTEM_SERVICE_TABLE KrnlFuncTable;
16     KSYSTEM_SERVICE_TABLE Wind32KTable;
17     KSYSTEM_SERVICE_TABLE NotUsedTableA;
18     KSYSTEM_SERVICE_TABLE NotUsedTableB;
19 } KSERVICE_TABLE_DESCRIPTOR, *PKSYSTEM_SERVICE_DESCRIPTOR;
20
21 // 导出由内核模块所导出的SSDT
22 extern PKSYSTEM_SERVICE_DESCRIPTOR KeServiceDescriptorTable;
23
24 // 用于保存被Hook的函数地址
25 ULONG FuncAddr;
26
27 // 用于保存替换指令的地址和返回地址
28 ULONG CmdAddress;
29 ULONG BackAddress;
30
31 // 用于存放修改执行流程的指令
32 UCHAR PushRetCmd[6] = { 0 };
33 // 用于保存被替换的指令
34 UCHAR OriginCmd[6] = { 0x33, 0xC0, 0x8D, 0x7D, 0xD8, 0xAB };
35
36 // 用于获取参数的全局变量
37 PHANDLE ProcessHandle;
38 ACCESS_MASK DesiredAccess;
39 POBJECT_ATTRIBUTES ObjectAttributes;
40 PCLIENT_ID ClientId;
41
42 // 准备的函数
43 VOID __declspec(naked) MyNtOpenProcess()
44 {
45     __asm
46     {
47         // 保存寄存器
48         pushad
49         pushfd
50         // 获取传参数
51         mov eax, [ebp + 0x8]
52         mov ProcessHandle, eax
53         mov eax, [ebp + 0xC]
54         mov DesiredAccess, eax
55         mov eax, [ebp + 0x10]
56         mov ObjectAttributes, eax
57         mov eax, [ebp + 0x14]
58         mov ClientId, eax
59     }
60
61     // 调试输出
62     DbgPrint("ProcessHandle: %x, DesiredAccess: %x, ObjectAttributes: %x,
63 ClientId: %x\n", ProcessHandle, DesiredAccess, ObjectAttributes,
64 ClientId);
65
66     __asm

```



```

65     {
66         // 还原寄存器
67         popfd
68         popad
69         // 执行被替换的指令, 保证运行完整
70         xor eax, eax
71         lea edi, [ebp-0x28]
72         stos dword ptr es:[edi]
73         // 跳转回原函数被替换指令的下一行地址
74         jmp BackAddress
75     }
76 }
77
78 // 修改物理页属性
79 NTSTATUS ChangePageAttr(ULONG attrValue)
80 {
81     ULONG RCR4 = 0;
82     // 获得CR4寄存器的值
83     _asm
84     {
85         _emit 0x0F
86         _emit 0x20
87         _emit 0xE0
88         mov RCR4, eax
89     }
90     // 根据CR4寄存器的PAE位判断分页模式
91     if (RCR4 & 0x00000200)
92     {
93         DbgPrint("2-9-9-12 \n");
94         // 获取PDE、PTE, 并修改R/W
95         *(ULONG64*)(0xC0600000 + (((FuncAddr >> 21) << 3) & 0x3FF8)) |=
attrValue;
96         *(ULONG64*)(0xC0000000 + (((FuncAddr >> 12) << 3) & 0x7FFF8)) |=
attrValue;
97     }
98     else
99     {
100         DbgPrint("10-10-12 \n");
101         // 获取PDE、PTE, 并修改R/W
102         *(ULONG*)(0xC0300000 + ((FuncAddr >> 20) & 0xFFC)) |= attrValue;
103         *(ULONG*)(0xC0000000 + ((FuncAddr >> 10) & 0x3FFFC)) |=
attrValue;
104     }
105
106     return STATUS_SUCCESS;
107 }
108
109 // 修改函数指令
110 NTSTATUS ChangeFuncCmd(CHAR CmdByte[6])
111 {
112     ChangePageAttr(0x2);
113     RtlMoveMemory((PUCHAR)CmdAddress, CmdByte, 6);
114     ChangePageAttr(0x0);

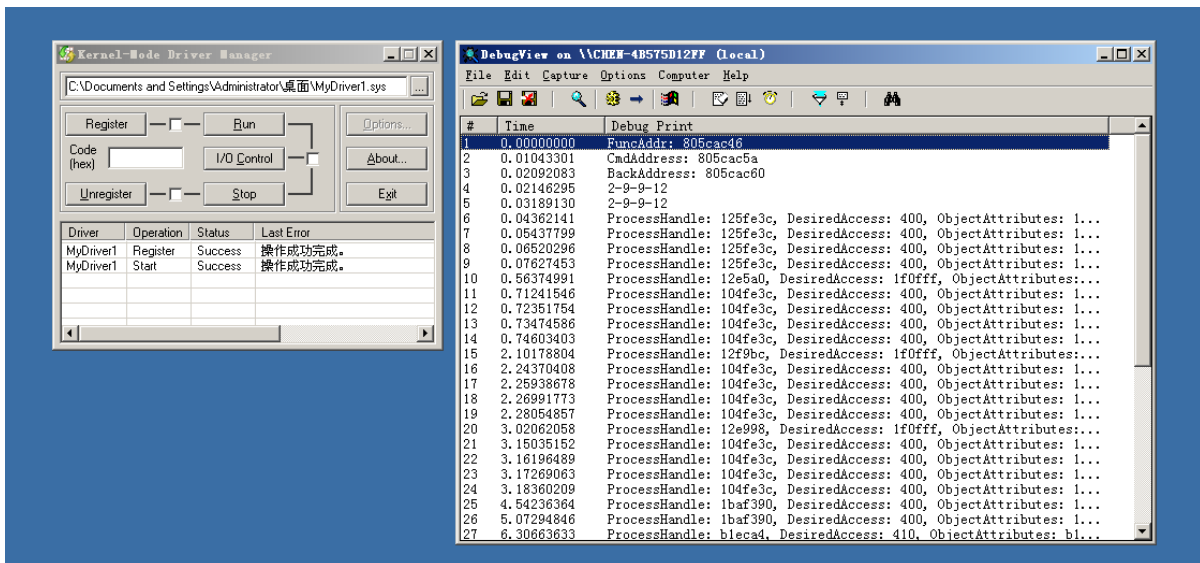
```

```

115
116     return STATUS_SUCCESS;
117 }
118
119 // 自定义的驱动程序卸载函数
120 VOID DriverUnload(PDRIVER_OBJECT DriverObject)
121 {
122     // 还原替换指令
123     ChangeFuncCmd(OriginCmd);
124     DbgPrint("Bye. \n");
125 }
126
127 // 驱动程序入口函数
128 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
129 {
130     // 保存被Hook的函数地址
131     FuncAddr = KeServiceDescriptorTable->
KrnlfuncTable.ServiceTableBase[0x7A];
132     DbgPrint("FuncAddr: %x \n", FuncAddr);
133     // 计算要替换的函数地址和要JMP的返回地址
134     CmdAddress = FuncAddr + 0x14; // 将要替换的指令地址减去函数首地址得出偏移
0x144
135     DbgPrint("CmdAddress: %x \n", CmdAddress);
136     BackAddress = CmdAddress + 0x6; // 替换的指令长度为0x6
137     DbgPrint("BackAddress: %x \n", BackAddress);
138     // 硬编码 PUSH 0x12345678
139     PushRetCmd[0] = 0x68;
140     *(ULONG*)(PushRetCmd + 1) = (ULONG)MyNtOpenProcess;
141     // 硬编码 RET
142     PushRetCmd[5] = 0xC3;
143     // 替换指令
144     ChangeFuncCmd(PushRetCmd);
145
146     // 设置一个卸载函数，当驱动卸载时触发
147     DriverObject->DriverUnload = DriverUnload;
148     return STATUS_SUCCESS;
149 }

```

我们可以加载驱动，来看一下实际效果，如下图所示我们成功进行了Hook：



并且在Windbg中查看nt!NtOpenProcess函数，也可以看到它的指令被我们成功覆盖：

```
0: kd> u nt!NtOpenProcess L40
nt!NtOpenProcess:
805cac46 68c4000000      push    0C4h
805cac4b 6898b44d80      push    offset nt!ObWatchHandles+0x25c (804db498)
805cac50 e81b0ff7ff      call    nt!_SEH_prolog (8053bb70)
805cac55 33f6           xor     esi,esi
805cac57 8975d4         mov     dword ptr [ebp-2Ch],esi
805cac5a 681092c7ba     push    offset MyDriver1!MyNtOpenProcess (bac79210)
805cac5f c3             ret
805cac60 64a124010000   mov     eax,dword ptr fs:[00000124h]
805cac66 8a8040010000   mov     al,byte ptr [eax+140h]
805cac6c 8845cc         mov     byte ptr [ebp-34h],al
```