© The Assembly-Programming-Journal, Vol.2, No.1. (2004)
http://www.Assembly-Journal.com

Kernel Mode Driver Tutorial for MASM32 Programmers Part 5 – The Full Function Driver

Author: Four-F

Abstract

Source code: KmdKit\examples\simple\VirtToPhys

Contents

5.1 The driver's source code

Now it's time to take a look at full-function driver's source code. Here it is:

```
;@echo off
;goto make

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;
; VirtToPhys - Kernel Mode Driver
;
;  Translates virtual addres to physical address
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

.386
.model flat, stdcall
option casemap:none

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                              I N C L U D E   F I L E S
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

include \masm32\include\w2k\ntstatus.inc
include \masm32\include\w2k\ntddk.inc
include \masm32\include\w2k\ntoskrnl.inc
include \masm32\include\w2k\w2kundoc.inc

includelib \masm32\lib\w2k\ntoskrnl.lib

include \masm32\Macros\Strings.mac

include ..\common.inc

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                      C O N S T A N T S
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

.const
CCOUNTED_UNICODE_STRING    "\\Device\\devVirtToPhys",
g_usDeviceName, 4
CCOUNTED_UNICODE_STRING    "\\??\\slVirtToPhys",
g_usSymbolicLinkName, 4

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                            C O D E
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

.code
```

```
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                    GetPhysicalAddress
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

GetPhysicalAddress proc dwAddress:DWORD

    mov eax, dwAddress
    mov ecx, eax

    shr eax, 22
    shl eax, 2

    mov eax, [0C0300000h][eax]

    .if ( eax & (mask pde4kValid) )
        .if !( eax & (mask pde4kLargePage) )
            mov eax, ecx
            shr eax, 10
            and eax, 1111111111111111111100y
            add eax, 0C0000000h
            mov eax, [eax]

            .if eax & (mask pteValid)
                and eax, mask ptePageFrameNumber

                and ecx, 00000000000000000000111111111111y
                add eax, ecx
            .else
                xor eax, eax
            .endif
        .else
            and eax, mask pde4mPageFrameNumber
            and ecx, 00000000001111111111111111111111y
            add eax, ecx
        .endif
    .else
        xor eax, eax
    .endif

    ret

GetPhysicalAddress endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                   DispatchCreateClose
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

DispatchCreateClose proc pDeviceObject:PDEVICE_OBJECT, pIrp:PIRP

    mov eax, pIrp
    assume eax:ptr _IRP
    mov [eax].IoStatus.Status, STATUS_SUCCESS
    and [eax].IoStatus.Information, 0
    assume eax:nothing

    fastcall IofCompleteRequest, pIrp, IO_NO_INCREMENT
    mov eax, STATUS_SUCCESS
    ret

DispatchCreateClose endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                     DispatchControl
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

DispatchControl proc uses esi edi ebx pDeviceObject:PDEVICE_OBJECT,
pIrp:PIRP

local status:NTSTATUS
local dwBytesReturned:DWORD

    and dwBytesReturned, 0

    mov esi, pIrp
    assume esi:ptr _IRP

    IoGetCurrentIrpStackLocation esi
    mov edi, eax
    assume edi:ptr IO_STACK_LOCATION

    .if [edi].Parameters.DeviceIoControl.IoControlCode ==
IOCTL_GET_PHYS_ADDRESS
        .if ( [edi].Parameters.DeviceIoControl.OutputBufferLength
>= DATA_SIZE ) && (
[edi].Parameters.DeviceIoControl.InputBufferLength >= DATA_SIZE )

            mov edi, [esi].AssociatedIrp.SystemBuffer
            assume edi:ptr DWORD

            xor ebx, ebx
            .while ebx < NUM_DATA_ENTRY

                invoke GetPhysicalAddress, [edi][ebx*(sizeof
DWORD)]

                mov [edi][ebx*(sizeof DWORD)], eax
                inc ebx
            .endw

            mov dwBytesReturned, DATA_SIZE
            mov status, STATUS_SUCCESS
        .else
            mov status, STATUS_BUFFER_TOO_SMALL
        .endif
    .else
        mov status, STATUS_INVALID_DEVICE_REQUEST
    .endif

    assume edi:nothing

    push status
    pop [esi].IoStatus.Status

    push dwBytesReturned
    pop [esi].IoStatus.Information

    assume esi:nothing

    fastcall IofCompleteRequest, pIrp, IO_NO_INCREMENT

    mov eax, status
    ret

DispatchControl endp

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                       DriverUnload
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

DriverUnload proc pDriverObject:PDRIVER_OBJECT

        invoke IoDeleteSymbolicLink, addr g_usSymbolicLinkName

        mov eax, pDriverObject
        invoke IoDeleteDevice, (DRIVER_OBJECT PTR
[eax]).DeviceObject

    ret

DriverUnload endp

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                              D I S C A R D A B L E   C O D E
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

.code INIT

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;                                       DriverEntry
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

DriverEntry proc pDriverObject:PDRIVER_OBJECT,
pusRegistryPath:PUNICODE_STRING

local status:NTSTATUS
local pDeviceObject:PVOID

    mov status, STATUS_DEVICE_CONFIGURATION_ERROR

    invoke IoCreateDevice, pDriverObject, 0, addr g_usDeviceName,
FILE_DEVICE_UNKNOWN, \
                                             0, FALSE, addr
pDeviceObject

    .if eax == STATUS_SUCCESS
        invoke IoCreateSymbolicLink, addr g_usSymbolicLinkName,
addr g_usDeviceName
        .if eax == STATUS_SUCCESS
            mov eax, pDriverObject
            assume eax:PTR DRIVER_OBJECT
            mov [eax].MajorFunction[IRP_MJ_CREATE*(sizeof PVOID)],
offset DispatchCreateClose
            mov [eax].MajorFunction[IRP_MJ_CLOSE*(sizeof PVOID)],
offset DispatchCreateClose
            mov [eax].MajorFunction[IRP_MJ_DEVICE_CONTROL*(sizeof
PVOID)],   offset DispatchControl
            mov [eax].DriverUnload,
offset DriverUnload
            assume eax:nothing
            mov status, STATUS_SUCCESS
        .else
            invoke IoDeleteDevice, pDeviceObject
        .endif
    .endif

    mov eax, status
    ret

DriverEntry endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::

end DriverEntry

:make

set drv=VirtToPhys

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:%drv%.sys /subsystem:native /ignore:4078 %drv%.obj rsrc.obj

del %drv%.obj
move %drv%.sys ..

echo.
pause
```

5.2 Driver name and symbolic link name

Let's start with the definition of UNICODE_STRING structures, describing the device and symbolic link names. As I've already mentioned, kernel likes to work with strings in this format.

Inside of absolutely all drivers sources, both assembly and C, I meet the same senseless sequence like this:

```
.const
uszDeviceName          dw "\", "D", "e", "v", "i", "c", "e", "\",
"D", "e", "v", "N", "a", "m", "e", 0
uszSymbolicLinkName    dw "\", "?", "?", "\", "D", "e", "v", "N",
"a", "m", "e", 0

.code
DriverEntry proc . . .
. . .
local usDeviceName:UNICODE_STRING
local usSymbolicLinkName:UNICODE_STRING
. . .
    invoke RtlInitUnicodeString, addr usDeviceName, offset
uszDeviceName
    invoke RtlInitUnicodeString, addr usSymbolicLinkName, offset
uszSymbolicLinkName
```

The purpose of the **RtlInitUnicodeString** is to measure unicode-string and to fill UNICODE_STRING structure in. Since the unicode-strings in this code are defined statically, i.e. never will be changed, it is possible to fill UNICODE_STRING structure in at a link time. It is easier, more visual and saves a few bytes (8 bytes for UNICODE_STRING structure + maximum 3 bytes for alignment against the minimum 14 bytes for **RtlInitUnicodeString** call). That's why I don't like this way. Thus I use CCOUNTED_UNICODE_STRING macro for this. And all above code will turn into two elegant lines.

```
CCOUNTED_UNICODE_STRING "\\Device\\DevName", usDeviceName,       4
CCOUNTED_UNICODE_STRING "\\??\\DevName",      usSymbolicLinkName, 4
```

Having agreed with me you can define your driver and symbolic link names like this:

```
.const
CCOUNTED_UNICODE_STRING    "\\Device\\devVirtToPhys",
g_usDeviceName, 4
CCOUNTED_UNICODE_STRING    "\\??\\slVirtToPhys",
g_usSymbolicLinkName, 4
```

On the earlier releases of Windows NT there is no "\??" directory in the Object Manager namespace. In this case it is necessary to change "\??" to "\DosDevices". In the later Windows releases it will also work, since for backward compatibility there is a symbolic link "\DosDevices" in root directory of the Object Manager namespace, which points to "\??".

5.3 Writing DriverEntry Routine

Every kernel-mode driver has to expose a routine whose name is **DriverEntry** (you can give it any name you like, by the way), which initializes driver-wide data structures. The I/O Manager calls the **DriverEntry** routine when it loads the driver. **DriverEntry** routine runs at IRQL = PASSIVE_LEVEL, which means it can access paged system resources. **DriverEntry** runs in the System process context.

Before we go further, pay attention at this line:

```
.code INIT
```

All code marked like this is directed into the INIT section because it's not needed once the driver finishes initializing. The code inside the INIT section can be discarded as soon as the driver returns from its **DriverEntry**. It's up to the system to decide when to discard it.

It has no big sense for our tiny driver because its sections are 32-bytes aligned (we pass /align:32 key to the linker). Thus all its sections occupy only one page. And it will not be discarded at all even if we mark with "INIT" some lines. Previous Windows NT drivers had large **DriverEntry** routine that had to create device objects, locate resources, configure devices, and so on. In this case using of this feature is lead to significant memory savings. So if your DriverEntry is big enough it is meaningful to place it in a separate section marked as "INIT". I have already told about this section in third part.

```
    mov status, STATUS_DEVICE_CONFIGURATION_ERROR
```

Assume that driver initialization will fail. If it happens, this error code returns to the system, and user-mode call **StartService** will also fail.

5.3.1 Creating Virtual Device

```
    invoke IoCreateDevice, pDriverObject, 0, addr g_usDeviceName,
FILE_DEVICE_UNKNOWN, \
                                             0, FALSE, addr
pDeviceObject
```

Since the main driver purpose is to control some device, physical, virtual, or logical, we firstly have to create such device (virtual one in our case). We achieve this by calling **IoCreateDevice** that is used to create and initialize a device object (see DEVICE_OBJECT structure) for use by the driver. And here is its prototype:

```
IoCreateDevice proto stdcall DriverObject:PDRIVER_OBJECT,
DeviceExtensionSize:DWORD, \
                             DeviceName:PUNICODE_STRING,
DeviceType:DEVICE_TYPE, \
                             DeviceCharacteristics:DWORD,
Exclusive:BOOL, \
                             DeviceObject: PDEVICE_OBJECT
```

Parameter	Description
DriverObject	Points to the driver object (DRIVER_OBJECT structure). Each driver receives a pointer to its driver object as a parameter to its **DriverEntry** routine;
DeviceExtensionSize	Specifies the driver-determined number of bytes to be allocated for the device extension of the device object. The internal structure of the device extension is driver-defined. It has no sense to use it in our simple driver.
DeviceName	Optionally points to a buffer containing a zero-terminated Unicode string that names the device object. The string must be a full path name. Full path here means not a path to some file on HDD, but the path to object as it appears in Object Manager namespace. And this parameter is mandatory for us. We should create the named device, otherwise we can't create the symbolic link, and the user-mode can't get access to our device; The device name must be unique of course.
DeviceType	Specifies one of the system-defined FILE_DEVICE_XXX constants indicating the type of device or the driver-defined value for a new type of device. We use FILE_DEVICE_UNKNOWN.
DeviceCharacteristics	Specifies additional information about the driver's device. We have nothing like this. So it will be 0.
Exclusive	Indicates whether the device object represents an exclusive device. That is, only one handle at a time can send I/O requests to the corresponding device object. Last time I said that I did not manage to get exclusive access to the device using dwShareMode parameter of the **CreateFile**. It can be done with Exclusive. We don't need to own our device exclusively. So we pass FALSE.
DeviceObject	Points to the newly created device object

(DEVICE_OBJECT structure) if the call succeeds.

If next call to **IoCreateSymbolicLink** will fail we must remove our device object from the system. So, we save pointer to device object returned by **IoCreateDevice** in local variable pDeviceObject for future use.

We need this pointer also by unloading the driver. But at that point we can obtain it from the driver object itself.

You can save this pointer in a global variable or somewhere else. But I'm just don't want to create ?data section for this purpose only.

5.3.2 Creating Symbolic Link

```
    .if eax == STATUS_SUCCESS
        invoke IoCreateSymbolicLink, addr g_usSymbolicLinkName,
addr g_usDeviceName
```

If new device was created successfully we have to make it visible to the Win32 subsystem by creating a symbolic link (you have already know what this is). To create a symbolic link we call **IoCreateSymbolicLink**. This function takes an existing device name and a symbolic link name (both passed as UNICODE_STRING data types).

5.3.3 Announcing the Dispatch Routines

```
        .if eax == STATUS_SUCCESS
            mov eax, pDriverObject
            assume eax:PTR DRIVER_OBJECT
            mov [eax].MajorFunction[IRP_MJ_CREATE*(sizeof PVOID)],
offset DispatchCreateClose
            mov [eax].MajorFunction[IRP_MJ_CLOSE*(sizeof PVOID)],
offset DispatchCreateClose
            mov [eax].MajorFunction[IRP_MJ_DEVICE_CONTROL*(sizeof
PVOID)],   offset DispatchControl
```

If the symbolic link was successfully created we are at the next step.

Each driver object contains an array of function pointers to dispatch routines specific to the I/O request. Each driver must set at least one dispatch entry point in this array for the IRP_MJ_XXX requests which the driver handles. Any driver can set as many separate dispatch entry points as the IRP_MJ_XXX codes for the driver to handle. For example, if you want to receive the notice that the system is being shut down, you must "announce" the dispatch routine for such request. And you do that by placing the appropriate dispatch routine address into the IRP_MJ_SHUTDOWN slot of the MajorFunction table of the driver object. If you don't need to process such request you simply do nothing because the I/O Manager fills the entire MajorFunction table of the driver object with the pointers to the system internal routine **IopInvalidDeviceRequest**, which returns an error to the original caller before calling **DriverEntry**.

So, it's your responsibility to provide dispatch routines for each I/O function code you want to process.

We have to handle three types of I/O request packet in our driver. Every kernel-mode driver must support the function code IRP_MJ_CREATE since this code is generated in response to the Win32 **CreateFile** call. Without support for this code, Win32 applications would have no way to obtain a handle to the device. Similarly, the IRP_MJ_CLOSE must also be supported to handle the Win32 **CloseHandle** call. And IRP_MJ_DEVICE_CONTROL allows for extended requests from user-mode clients through the Win32 **DeviceIoControl** call.

Ňčď çŕďđîńŕ	Description
IRP_MJ_CREATE	I/O Manager sends it when a user-mode code has requested a handle for the file object that represents the target device object by calling **CreateFile** function.
IRP_MJ_DEVICE_CONTROL	I/O Manager sends it when a user-mode code has called the **DeviceIoControl** function.
IRP_MJ_CLOSE	I/O Manager sends it when the handle of the file object that represents the target device object has been released by calling **CloseHandle** function.

We handle IRP_MJ_CREATE and IRP_MJ_CLOSE in one **DispatchCreateClose** routine. I'll tell you a bit later about this.

In ntddk.inc, among others, you can find IRP_MJ_XXX codes that can be of interest for us:

```
IRP_MJ_CREATE                equ 0
. . .
IRP_MJ_CLOSE                 equ 2
IRP_MJ_READ                  equ 3
IRP_MJ_WRITE                 equ 4
. . .
IRP_MJ_DEVICE_CONTROL        equ 0Eh
. . .
IRP_MJ_CLEANUP               equ 12h
```

All IRP_MJ_XXX codes are listed in ntddk.inc. Each IRP_MJ_XXX code is the index in MajorFunction array. The preceding code snippet fills three slots of the MajorFunction array.

```
            mov [eax].DriverUnload,
offset DriverUnload
```

The purpose of **DriverUnload** function is to clean up after any global initialization **DriverEntry** might have done. If we want to dynamically unload the driver we have to supply the pointer to unload routine. This routine will be called by the system when the user-mode calls **ControlService** with SERVICE_CONTROL_STOP.

```
            assume eax:nothing
            mov status, STATUS_SUCCESS
```

If we have safely reached this point the driver was successfully initialized. So we report success to the system by returning STATUS_SUCCESS.

5.3.4 Cleanup

```
        .else
            invoke IoDeleteDevice, pDeviceObject
        .endif
    .endif
```

If the call to IoCreateSymbolicLink returns an error, we should release any allocated resources. So, we have to delete device object created by previous call to **IoCreateDevice**. And by calling **IoDeleteDevice** we remove device object from the system. If you have allocated some other resources you also have to return it back to the system of course.

Please always remember you must keep track of the memory you've allocated and any other allocated resources in order to release it when it's no longer needed. You are in the kernel-mode and must do all duty work by yourself. No one else will do that for you!

```
    mov eax, status
    ret
```

We return the current status code to the system. If it's STATUS_SUCCESS, the driver remains in the memory and the I/O Manager will route IRP to it. If it has any other value, the driver is removed.

5.3.5 New objects are here

Thus, after successful **DriverEntry** completion three new objects appear in the system: the driver "\Driver\VirtToPhys", the device "\Device\devVirtToPhys" and the symbolic link "\??\slVirtToPhys" to the device.

- The driver object represents an individual driver in the system.

From this object the I/O Manager obtains the address of each driver's dispatch routine.

- The device object represents a device on the system and describes its characteristics.

  Via this object the I/O Manager obtains the pointer to the driver object that manages this device.

- The file object, representing device object for user-mode.

  Using this object the I/O Manager obtains the pointer to the representing device object.

- The symbolic link that is visible to user-mode.

  The symbolic link is used by the Object Manager.

Figure 5-1 shows the main interrelations between these objects. This scheme will help you to understand a further material more thoroughly.

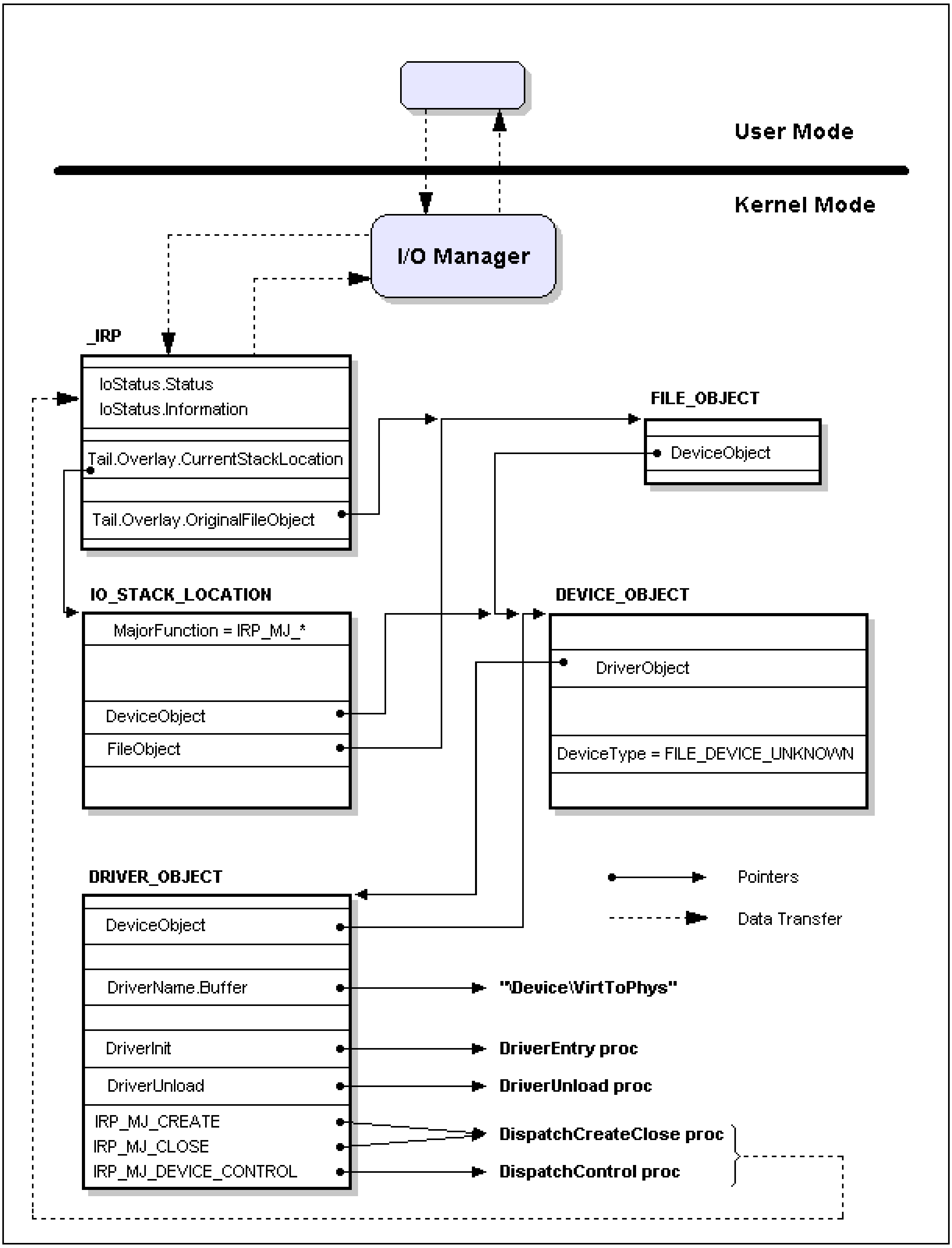


Figure 5-1. The main interrelations between the driver, device and file object.

5.4 I/O Dispatch Routines

The I/O Manager invokes Dispatch routines in response to user-mode or kernel-mode requests. In the case of monolithic or highest-level driver you are guaranteed that **Dispatch routines run in the same thread context as the initiator of the I/O requests. Like the driver's DriverEntry** routine, Dispatch routines run at IRQL = PASSIVE_LEVEL, which means they can access paged system resources.

Each Dispatch routine is declared as follows:

```
DispatchRoutine proto stdcall pDeviceObject:PDEVICE_OBJECT,
pIrp:PIRP
```

Parameter	Description
pDeviceObject	Pointer to the device object (DEVICE_OBJECT structure). If the driver serves some devices, it can determine, who is addressee of the IRP.
pIrp	Pointer to IRP (_IRP structure) describing I/O request. The I/O Manager creates an IRP describing the I/O request and sends its pointer to the device driver in pIrp parameter. It's up to the device driver how to handle this IRP.

Such uniform interface that Dispatch routines present allows the I/O Manager to call any driver without requiring any special knowledge of its structure or internal details.

5.5 Dispatch Routine for IRP_MJ_CREATE and IRP_MJ_CLOSE

Why such different types of IRP are processed by only one Dispatch routine? In our simple driver the only thing we have to do by processing IRP_MJ_CREATE and IRP_MJ_CLOSE requests is to mark IRP as completed.

If separate processing for create and close requests is required you must implement separate **DispatchCreate** and **DispatchClose** routines.

As I have already mentioned IRP_MJ_CREATE is generated in response to the **CreateFile** call. Without support of this code, Win32 applications would have no way to obtain a handle to the device. The mutual close IRP_MJ_CLOSE must also be supported to handle the **CloseHandle** call.

```
DispatchCreateClose proc pDeviceObject:PDEVICE_OBJECT, pIrp:PIRP

    mov eax, pIrp
    assume eax:ptr _IRP
    mov [eax].IoStatus.Status, STATUS_SUCCESS
    and [eax].IoStatus.Information, 0
    assume eax:nothing
```

We fill the I/O status block determining a condition of the IRP.

The Information member of the I/O status block is set to zero indicating the handle to the device can be opened for the create request. Information field has no meaning for the close request. This member may have some other meaning for the other IRP types.

The Status member indicates whether the **CreateFile** or **CloseHandle** call returns without an error. So we fill it with STATUS_SUCCESS.

```
    fastcall IofCompleteRequest, pIrp, IO_NO_INCREMENT

    mov eax, STATUS_SUCCESS
    ret

DispatchCreateClose endp
```

Now we should call **IoCompleteRequest** indicating the driver has completed IRP processing and returns it to the I/O Manager. And the returning by **DispatchCreateClose** STATUS_SUCCESS indicates that the device is ready to accept another I/O requests.

The first parameter of the **oCompleteRequest** tells the I/O Manager which IRP is to be completed. And the second determines a system-defined constant of runtime priority increase of the thread that requested the operation. The driver should compensate any thread that possibly waits for a device operation by giving a priority boost. For example, for sound devices DDK recommends to use IO_SOUND_INCREMENT which is equal to 8.

In our case we simply use IO_NO_INCREMENT equal to zero, which means the runtime priority of current thread remains the same.

IofCompleteRequest is a fastcall-function (notice the 'f' symbol in the prefix). There is also its stdcall counterpart **IoCompleteRequest**. But I use fastcall for the educational purposes.

5.6 Calling conventions

Three calling conventions are used in the Windows NT kernel APIs: __stdcall, __cdecl and __fastcall. Unfortunately, the last one is not supported by masm compiler.

The __fastcall calling convention specifies the first two DWORD arguments are passed in ECX and EDX registers; all other arguments are passed right to left. Called function pops the arguments from the stack.

Fastcall-function's names are mangled (decorated) as follows: @ sign is prefixed to the name, @ sign appended followed by a decimal number that indicates the count of bytes passed to the function as parameters. For example, **IofCompleteRequest** is decorated like this:

@IofCompleteRequest@8

The decorated name above means that it is a fastcall-function, its exported name is **IofCompleteRequest** and it takes two DWORD arguments.

This function is defined in \include\w2k\ntoskrnl.inc as follows (pay no attention on the SYSCALL):

```
EXTERNDEF SYSCALL @IofCompleteRequest@8:PROC
IofCompleteRequest TEXTEQU <@IofCompleteRequest@8>
```

To make it easier to call fastcall-functions I wrote this macro:

```
fastcall MACRO api:REQ, p1, p2, px:VARARG

local arg

    ifnb <px>
        % for arg, @ArgRev( <px> )
            push arg
        endm
    endif

    ifnb <p1>

        ifdifi <p1>, <ecx>
            mov ecx, p1
        endif

        ifnb <p2>
            ifdifi <p2>, <edx>
                mov edx, p2
            endif
        endif
    endif

    call api

ENDM
```

Here is the simplified version of macro. I've placed it in \include\w2k\ntddk.inc. There is no such macro in original ntddk.h, of course.

5.7 Memory buffer management

The I/O Manager performs three types of buffer management:

- ***buffered I/O***;
- ***direct I/O***;
- ***neither I/O***.

Here we'll examine only usage of DeviceIoControl function for I/O processing. The usage of **ReadFile** and **WriteFile** is a bit different. An example of using **ReadFile** to read the device data you will find in \src\NtBuild.

5.7.1 Buffered I/O

Starting I/O operation, the I/O Manager validates all virtual memory pages spanned by the user's buffer are valid. Then it allocates a nonpaged pool buffer of a size sufficient to hold the user's request.

While creating an IRP the I/O Manager copies the user's buffer data into the allocated buffer and passes its address to the driver in AssociatedIrp.SystemBuffer field of _IRP structure. The size of copied data is stored into Parameters.DeviceIoControl.InputBufferLength field of IO_STACK_LOCATION structure (Tail.Overlay.CurrentStackLocation of _IRP points to this structure and the pointer can be fetched with IoGetCurrentIrpStackLocation macro).

The driver handles the IRP and copies output data (if any) into the very same buffer.

When IRP is marked as completed by calling IofCompleteRequest the I/O Manager copies data from the allocated buffer to the user's buffer and then frees the allocated buffer. The amount of data to copy is placed by the driver in IoStatus.Information field of _IRP structure.

As you can see the I/O Manager copies data twice. Thus buffered I/O is used by slower devices that do not generally handle large data transfers as our VirtToPhys device does.

But this method has one big advantage - I/O Manager solves all possible problems with probable memory access errors by itself. We don't need to take care of it.

5.7.2 Direct I/O

This method is used for direct memory access (DMA).

I not examine in details this type of I/O handling, since it does not applicable in a scope of this doc.

When the I/O Manager creates the IRP, it locks the user's buffer into the memory (makes it nonpaged) which is made it accessible to driver code via an address above 80000000h. The I/O Manager stores a description of this memory in the form of a MemoryDescriptorLlist (MDL) and places pointer to it into the _IRP's MdlAddress field. An MDL specifies the physical memory occupied by the buffer. When the I/O Manager has finished IRP usage, it unlocks the buffer

5.7.3 Neither I/O

The I/O Manager doesn't perform any buffer management in any way. It's up to the discretion of the device driver.

The driver gets the user-mode virtual address of the input buffer in the Type3InputBuffer parameter of the stack location, and the user-mode virtual address of the output buffer in the UserBuffer field of the IRP. Neither address is of any use unless you know you're running in the same process context as the user-mode caller. And as the monolithic device driver's writers we know it for sure.

Also we know the monolithic device driver is always called from user-mode at IRQL = PASSIVE_LEVEL. So, no need to take care about the presence of the user buffer in memory. Memory Manager will do all necessary job if the user buffer is swapped out.

Only one more problem remains - the user-mode code can provide us with wrong buffer address or free it somehow (in the case of multithreaded application) while data transfer is in progress.

We have to foresee such situations and to handle it correctly. Thus the usage of Structured Exception Handling (SEH) is necessary (see example \src\Article4-5\NtBuild). But bear in mind, the SEH in the kernel-mode is in its entirety the same as in user-mode, though you can't handle all exceptions this way. For example, the attempt of divide by zero will result in BSOD even with installed SEH-handler!

5.8 Dispatch Routine for IRP_MJ_DEVICE_CONTROL

Once the driver has announced Dispatch routine for IRP_MJ_DEVICE_CONTROL, the I/O Manager starts passing IRPs directly to the driver code when user-mode client calls **DeviceIoControl**.

```
    and dwBytesReturned, 0
```

If any error will occur, the I/O Manager should not copy anything in the user buffer.

```
    mov esi, pIrp
    assume esi:ptr _IRP

    IoGetCurrentIrpStackLocation esi
    mov edi, eax
    assume edi:ptr IO_STACK_LOCATION
```

IoGetCurrentIrpStackLocation macro gives us a pointer to the IRP stack location - the pointer to IO_STACK_LOCATION structure containing some necessary data.

```
    .if [edi].Parameters.DeviceIoControl.IoControlCode ==
IOCTL_GET_PHYS_ADDRESS
```

We must not handle the receipt of an unrecognized I/O control code.

```
NUM_DATA_ENTRY         equ 4
DATA_SIZE              equ (sizeof DWORD) * NUM_DATA_ENTRY
IOCTL_GET_PHYS_ADDRESS equ CTL_CODE(FILE_DEVICE_UNKNOWN, 800h,
METHOD_BUFFERED, FILE_READ_ACCESS + FILE_WRITE_ACCESS)
```

The IOCTL_GET_PHYS_ADDRESS control code we are waiting for is defined in common.inc as well as two constants. This file is included both in the driver and in its client source codes.

```
        .if ( [edi].Parameters.DeviceIoControl.OutputBufferLength
>= DATA_SIZE ) && (
[edi].Parameters.DeviceIoControl.InputBufferLength >= DATA_SIZE )
```

We check the size of user's input and output buffers. If both are less than required we stop processing.

The OutputBufferLength and InputBufferLength fields of IO_STACK_LOCATION structure correspond to nOutBufferSize and nInBufferSize parameters of **DeviceIoControl** function.

```
      mov edi, [esi].AssociatedIrp.SystemBuffer
```

From the IRP stack location we obtain the pointer to the system buffer. This buffer contains now the data user-mode client have sent to us. In our case this data is four virtual addresses our driver have to translate to physical ones.

```
      assume edi:ptr DWORD
```

The compiler should know that edi points to DWORD value. Without this statement we have to use PTR DWORD each time we touch edi.

```
      xor ebx, ebx
      .while ebx < NUM_DATA_ENTRY

          invoke GetPhysicalAddress, [edi][ebx*(sizeof
DWORD)]

          mov [edi][ebx*(sizeof DWORD)], eax
          inc ebx
      .endw
```

We are repeatedly cycling NUM_DATA_ENTRY times through buffer and for each dword (namely virtual address) we meet call **GetPhysicalAddress** whose output (namely physical address) is putted back into the buffer at the same place.

```
      mov dwBytesReturned, DATA_SIZE
      mov status, STATUS_SUCCESS
```

When we reached this point our job is done. So we put number of processed bytes into dwBytesReturned and indicate success in status.

```
    .else
      mov status, STATUS_BUFFER_TOO_SMALL
    .endif
  .else
    mov status, STATUS_INVALID_DEVICE_REQUEST
  .endif
```

If something went wrong status receives an appropriate error code.

```
assume edi:nothing

push status
pop [esi].IoStatus.Status
```

Completing the IRP we place in Status field of the status block a current status value. This status code will be translated to the corresponding Win32 error code. Below is the correspondence for three status codes we used:

Nt Status	Win32 Error
STATUS_SUCCESS	NO_ERROR
STATUS_BUFFER_TOO_SMALL	ERROR_INSUFFICIENT_BUFFER
STATUS_INVALID_DEVICE_REQUEST	ERROR_INVALID_FUNCTION

You can call **RtlNtStatusToDosError** exported by ntdll.dll to translate kernel status to Win32 error. The user-mode applications call **GetLastError** to determine the cause of the failure.

```
push dwBytesReturned
pop [esi].IoStatus.Information
```

The Information field of the status block gets number of bytes I/O Manager has to copy into the user's buffer. The caller of **DeviceIoControl** will receive this value in a variable pointed to by lpBytesReturned.

```
assume esi:nothing

fastcall IofCompleteRequest, pIrp, IO_NO_INCREMENT

mov eax, status
ret
```

Call **IofCompleteRequest** to complete the IRP we have just handled.

And remember we must handle the receipt of even unrecognized I/O control code by setting the I/O status block with an appropriate NTSTATUS value, setting its Information field to zero, and completing the IRP with a PriorityBoost of IO_NO_INCREMENT.

5.9 Memory Address Translation

The kernel-mode code can translate virtual addresses to physical one. And **MmGetPhysicalAddress** kernel function is for this purpose. The **GetPhysicalAddress** function I use here does basically the same (not considering Physical Address Extension mode). Unfortunately I have no opportunity to describe in detail what is going on here. Please refer to "Inside Microsoft Windows 2000" by David Solomon and Mark Russinovich. There you will find the detailed description. Here is my commented code:

```
GetPhysicalAddress proc dwAddress:DWORD

    ; Converts virtual address in dwAddress to corresponding
physical address

    mov eax, dwAddress
    mov ecx, eax

    shr eax, 22                                   ; (Address >> 22) =>
Page Directory Index, PDI
    shl eax, 2                                    ; * sizeof PDE = PDE
offset

    mov eax, [0C0300000h][eax]                    ; [Page Directory
Base + PDE offset]

    .if ( eax & (mask pde4kValid) )               ; .if ( eax & 01y )
        ; PDE is valid
        .if !( eax & (mask pde4kLargePage) )      ; .if ( eax &
010000000y )
            ; small page (4kB)
            mov eax, ecx
            ; (Address >> 12) * sizeof PTE => PTE offset
            shr eax, 10
            and eax, 1111111111111111111100y
            add eax, 0C0000000h                   ; add Page Table
Array Base
            mov eax, [eax]                        ; fetch PTE

            .if eax & (mask pteValid)             ; .if ( eax & 01y )
                ; PTE is valid
                ; mask PFN   (and eax,
11111111111111111111000000000000y)
                and eax, mask ptePageFrameNumber

                ; We actually don't need these two lines
                ; because of module base is always page aligned
                and ecx, 00000000000000000000111111111111y  ; Byte
Index
                add eax, ecx                      ; add byte offset to
physical address
            .else
                xor eax, eax                      ; error
            .endif
        .else
            ; large page (4mB)
            ; mask PFN   (and eax,
11111111110000000000000000000000y)
            and eax, mask pde4mPageFrameNumber
            and ecx, 00000000001111111111111111111111y      ; Byte
Index
            add eax, ecx                      ; add byte offset to
physical address
        .endif
    .else
        xor eax, eax                          ; error
    .endif

    ret

GetPhysicalAddress endp
```

GetPhysicalAddress returns the physical address that corresponds to the given virtual address.

5.10 DriverUnload Routine

DriverUnload routine's work is straightforward, as it must delete each symbolic link and device object that has been created. This routine is called whenever the user-mode code calls **ControlService** with SERVICE_CONTROL_STOP, but only if there are no more opened handles of the device. If at least one open handle exists, device can receive IPR and thus it should remains in memory.

```
        invoke IoDeleteSymbolicLink, addr g_usSymbolicLinkName

        mov eax, pDriverObject
        invoke IoDeleteDevice, (DRIVER_OBJECT PTR
[eax]).DeviceObject
```

The unload routine undoes the work of **DriverEntry**. Namely calling **IoDeleteSymbolicLink** we remove symbolic link from the Object Manager namespace and call to the **IoDeleteDevice** removes the device object itself.

As I have previously mentioned, you are in kernel-mode and must release all allocated resources.

The table 5-1 sums up all you have to know about process context and IRQL of main driver's routines. The info in this table is correct only in the case of monolithic or highest-level driver.

User-mode	Kernel-mode	Process	IRQL

		context	
StartService	DriverEntry	System	PASSIVE_LEVEL
CreateFile	IRP_MJ_CREATE	User-mode caller	PASSIVE_LEVEL
DeviceIoControl	IRP_MJ_DEVICE_CONTROL	User-mode caller	PASSIVE_LEVEL
ReadFile	IRP_MJ_READ	User-mode caller	PASSIVE_LEVEL
WriteFile	IRP_MJ_WRITE	User-mode caller	PASSIVE_LEVEL
CloseHandle	IRP_MJ_CLEANUP, IRP_MJ_CLOSE	User-mode caller	PASSIVE_LEVEL
ControlService,,SERVICE_CONTROL_STOP	DriverUnload	System	PASSIVE_LEVEL

Table 5-1. Correspondence of user-mode functions to the driver's routines

5.11 How to compile

```
:make

set drv=skeleton

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:%drv%.sys /subsystem:native /ignore:4078 %drv%.obj rsrc.obj

del %drv%.obj
move %drv%.sys ..

echo.
pause
```

We have already analyzed all that in the third part. I have added the /ignore:4078 option, since we have two sections with the same name but with the different attributes. Otherwise the linker produces warning:

```
LINK : warning LNK4078: multiple "INIT" sections found with
different attributes (E2000020)
```

5.12 Adding resources

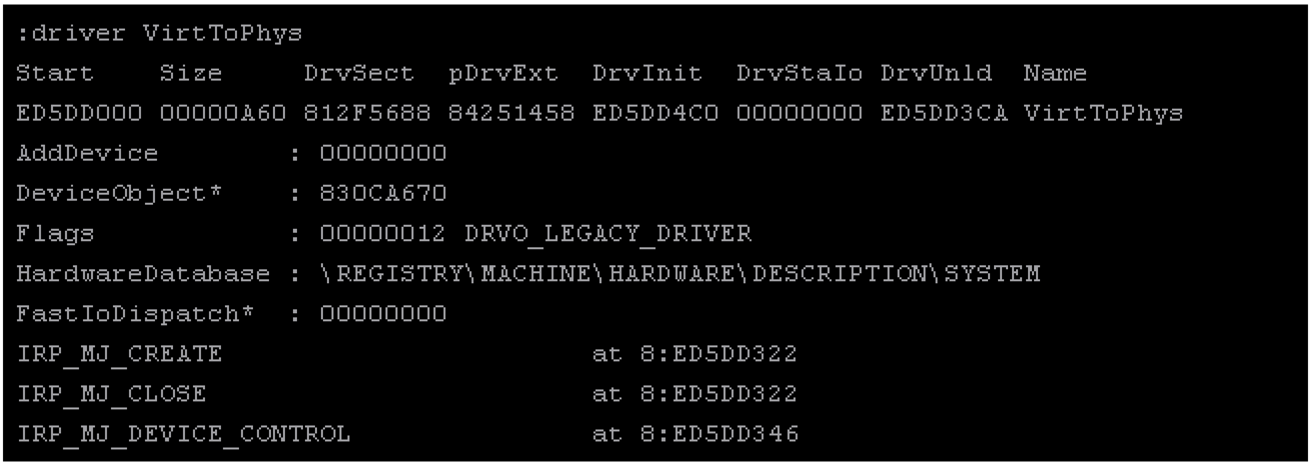
We also put the version resource into the driver image providing the driver's version information. It can be done using a common resource script (see rsrc.rc).

```
VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,0
 PRODUCTVERSION 1,0,0,0
 FILEFLAGSMASK 0x3fL
 FILEFLAGS 0x0L
 FILEOS 0x40004L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904E4"
        BEGIN
            VALUE "Comments", "Written by Four-F\0"
            VALUE "CompanyName", "Four-F Software\0"
            VALUE "FileDescription", "Kernel-Mode Driver VirtToPhys
v1.00\0"
            VALUE "FileVersion", "1, 0, 0, 0\0"
            VALUE "InternalName", "VirtualToPhysical\0"
            VALUE "LegalCopyright", "Copyright © 2003, Four-F\0"
            VALUE "OriginalFilename", "VirtToPhys.sys\0"
            VALUE "ProductName", "Kernel-Mode Driver Virtual To
Physical Address Converter\0"
            VALUE "ProductVersion", "1, 0, 0, 0\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END
```

Nothing special here. It's compiled and linked as usual.

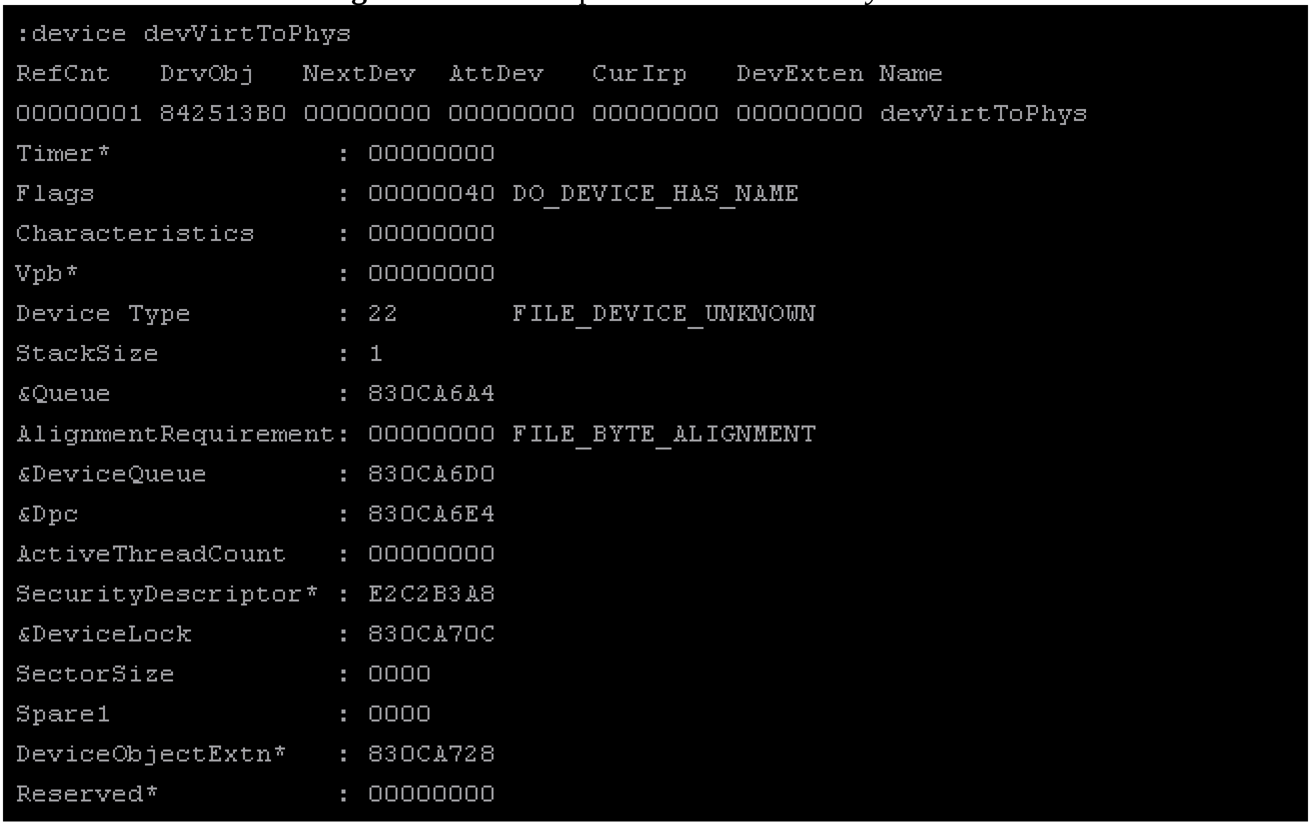
5.13 A little more words about debugging

You can obtain very useful information about the driver and its device with the help of **SoftICE**'s driver and device commands. See ***SoftICE Command Reference*** for details. Here is how it looks on my machine:



```
:driver VirtToPhys
Start    Size     DrvSect  pDrvExt  DrvInit  DrvStaIo DrvUnld  Name
ED5DD000 00000A60 812F5688 84251458 ED5DD4C0 00000000 ED5DD3CA VirtToPhys
AddDevice        : 00000000
DeviceObject*    : 830CA670
Flags            : 00000012 DRVO_LEGACY_DRIVER
HardwareDatabase : \REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM
FastIoDispatch*  : 00000000
IRP_MJ_CREATE                        at 8:ED5DD322
IRP_MJ_CLOSE                         at 8:ED5DD322
IRP_MJ_DEVICE_CONTROL                at 8:ED5DD346
```

Figure 5-2. The output of *driver VirtToPhys*



```
:device devVirtToPhys
RefCnt   DrvObj   NextDev  AttDev   CurIrp   DevExten Name
00000001 842513B0 00000000 00000000 00000000 00000000 devVirtToPhys
Timer*               : 00000000
Flags                : 00000040 DO_DEVICE_HAS_NAME
Characteristics      : 00000000
Vpb*                 : 00000000
Device Type          : 22       FILE_DEVICE_UNKNOWN
StackSize            : 1
&Queue               : 830CA6A4
AlignmentRequirement: 00000000 FILE_BYTE_ALIGNMENT
&DeviceQueue         : 830CA6D0
&Dpc                 : 830CA6E4
ActiveThreadCount    : 00000000
SecurityDescriptor*  : E2C2B3A8
&DeviceLock          : 830CA70C
SectorSize           : 0000
Spare1               : 0000
DeviceObjectExtn*    : 830CA728
Reserved*            : 00000000
```

Figure 5-3. The output of device *device devVirtToPhys*

As you understand, the information **SoftICE** displays, is obtained from DRIVER_OBJECT and DEVICE_OBJECT structures accordingly. Using this info it is possible easily to find these objects in the memory and set breakpoints on its routines.