The I/O Subsystem

4.1 I/O Manager
4.2 The control program for the VirtToPhys driver
- 4.2.1 Control program source code
- 4.2.2 Device object
- 4.2.3 Driver object
- 4.2.4 Symbolic link object
- 4.2.5 File object
- 4.2.6 Communicating with the device
- 4.2.7 I/O Control Codes
- 4.2.8 Data exchange
- 4.2.9 Cleanup

Source code: KmdKit\examples\simple\VirtToPhys

I/O Manager

Unlike the user-mode, where we can call functions from some dll directly, simply using its address, in the kernel-mode such scenario would be extremely dangerous in the view of the system stability. Therefore, the system provides the intermediary to communicate with the kernel-mode. Such intermediary is an *I/O Manager*, which is one of the *I/O subsystem*'s component. The I/O Manager connects applications and system components with devices, and defines the infrastructure that supports device drivers.

Very simplified scheme of how the I/O Manager interacts with the user-mode applications and the device drivers is given on figure 4-1.

Figure 4-1. Simplified I/O subsystem architecture

From the above figure follows that absolutely all calls from the user-mode applications to the devices and, therefore, to the device drivers are under control of the I/O Manager.

The user-mode code is forced to order the I/O operation to the device. Only and exactly the device. The driver must create some device (or devices) to control. In our case this device is virtual one. Of course, creating the device doesn't mean some new real device will be created. It just means some new object will be created in the memory (namely device object) representing a physical or logical device on the system and describing its characteristics.

Creating the device the driver tells the I/O Manager: "Here is the device for me to control. If you will receive some I/O request to this device, send it to me, and I'll take care about the rest." The driver only knows how to handle I/O requests to its device (s). The only responsibility of the I/O Manager is to create and direct the I/O request to the appropriate device driver. And the user-mode code does not (and should not) know at all, which driver services a particular device(s).

4.2 The control program for the VirtToPhys driver

4.2.1 Control program source code

Strictly speaking this code combines the service control program responsible for registration and starting the driver, and the client program to communicate with the device.

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
; VirtToPhys.asm - Driver Control Program for VirtToPhys driver
;
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.386
.model flat, stdcall
option casemap:none

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

include \masm32\include\windows.inc

include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\advapi32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\advapi32.lib

include \masm32\include\winioctl.inc

include \masm32\Macros\Strings.mac

include common.inc

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

.code

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                      BigNumToString
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

BigNumToString proc uNum:UINT, pszBuf:LPSTR

; This function accepts a number and converts it to a
; string, inserting commas where appropriate.

local acNum[32]:CHAR
local nf:NUMBERFMT

    invoke wsprintf, addr acNum, $CTA0("%u"), uNum

    and nf.NumDigits, 0
    and nf.LeadingZero, FALSE
    mov nf.Grouping, 3
```

```
    mov nf.lpDecimalSep, $CTA0(".")
    mov nf.lpThousandSep, $CTA0(" ")
    and nf.NegativeOrder, 0
    invoke GetNumberFormat, LOCALE_USER_DEFAULT, 0, addr acNum, addr nf, pszBuf, 32

    ret

BigNumToString endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                              start
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

start proc uses esi edi

local hSCManager:HANDLE
local hService:HANDLE
local acModulePath[MAX_PATH]:CHAR
local _ss:SERVICE_STATUS
local hDevice:HANDLE

local adwInBuffer[NUM_DATA_ENTRY]:DWORD
local adwOutBuffer[NUM_DATA_ENTRY]:DWORD
local dwBytesReturned:DWORD

local acBuffer[256+64]:CHAR
local acThis[64]:CHAR
local acKernel[64]:CHAR
local acUser[64]:CHAR
local acAdvapi[64]:CHAR

local acNumber[32]:CHAR

    invoke OpenSCManager, NULL, NULL, SC_MANAGER_ALL_ACCESS
    .if eax != NULL
        mov hSCManager, eax

        push eax
        invoke GetFullPathName, $CTA0("VirtToPhys.sys"), \
                                sizeof acModulePath, addr acModulePath, esp
        pop eax

        invoke CreateService, hSCManager, $CTA0("VirtToPhys"), \
                              $CTA0("Virtual To Physical Address Converter"), \
                              SERVICE_START + SERVICE_STOP + DELETE, SERVICE_KERNEL_DRIVER, \
                              SERVICE_DEMAND_START, SERVICE_ERROR_IGNORE, addr acModulePath, \
                              NULL, NULL, NULL, NULL, NULL

        .if eax != NULL
            mov hService, eax

            ; Driver's DriverEntry procedure will be called
            invoke StartService, hService, 0, NULL
            .if eax != 0

                ; Driver will receive I/O request packet (IRP) of type IRP_MJ_CREATE
                invoke CreateFile, $CTA0("\\\\.\\slVirtToPhys"), GENERIC_READ + GENERIC_WRITE, \
                                0, NULL, OPEN_EXISTING, 0, NULL

                .if eax != INVALID_HANDLE_VALUE
                    mov hDevice, eax

                    lea esi, adwInBuffer
                    assume esi:ptr DWORD
                    invoke GetModuleHandle, NULL
                    mov [esi][0*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("kernel32.dll", szKernel32)
                    mov [esi][1*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("user32.dll", szUser32)
                    mov [esi][2*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("advapi32.dll", szAdvapi32)
                    mov [esi][3*(sizeof DWORD)], eax

                    lea edi, adwOutBuffer
                    assume edi:ptr DWORD
                    ; Driver will receive IRP of type IRP_MJ_DEVICE_CONTROL
                    invoke DeviceIoControl, hDevice, IOCTL_GET_PHYS_ADDRESS, \
                                            esi, sizeof adwInBuffer, \
                                            edi, sizeof adwOutBuffer, \
                                            addr dwBytesReturned, NULL

                    .if ( eax != 0 ) && ( dwBytesReturned != 0 )

                        invoke GetModuleFileName, [esi][0*(sizeof DWORD)], \
```

```
                                    addr acModulePath, sizeof acModulePath

                    lea ecx, acModulePath[eax-5]
                    .repeat
                        dec ecx
                        mov al, [ecx]
                    .until al == '\'
                    inc ecx
                    push ecx

                    CTA0 "%s \t%08Xh\t%08Xh   ( %s )\n", szFmtMod

                    invoke BigNumToString, [edi][0*(sizeof DWORD)], addr acNumber
                    pop ecx
                    invoke wsprintf, addr acThis, addr szFmtMod, ecx, \
                                     [esi][0*(sizeof DWORD)], \
                                     [edi][0*(sizeof DWORD)], addr acNumber

                    invoke BigNumToString, [edi][1*(sizeof DWORD)], addr acNumber
                    invoke wsprintf, addr acKernel, addr szFmtMod, addr szKernel32, \
                                     [esi][1*(sizeof DWORD)], \
                                     [edi][1*(sizeof DWORD)], addr acNumber

                    invoke BigNumToString, [edi][2*(sizeof DWORD)], addr acNumber
                    invoke wsprintf, addr acUser, addr szFmtMod, addr szUser32, \
                                     [esi][2*(sizeof DWORD)], \
                                     [edi][2*(sizeof DWORD)], addr acNumber

                    invoke BigNumToString, [edi][3*(sizeof DWORD)], addr acNumber
                    invoke wsprintf, addr acAdvapi, addr szFmtMod, addr szAdvapi32, \
                                     [esi][3*(sizeof DWORD)], \
                                     [edi][3*(sizeof DWORD)], addr acNumber

                    invoke wsprintf, addr acBuffer, \
                                     $CTA0("Module:\t\tVirtual:\t\tPhysical:\n\n%s\n%s%s%s"), \
                                     addr acThis, addr acKernel, addr acUser, addr acAdvapi

                    assume esi:nothing
                    assume edi:nothing
                    invoke MessageBox, NULL, addr acBuffer, $CTA0("Modules Base Address"), \
                                       MB_OK + MB_ICONINFORMATION
                .else
                    invoke MessageBox, NULL, $CTA0("Can't send control code to device."), NULL, \
                                       MB_OK + MB_ICONSTOP
                .endif
                ; Driver will receive IRP of type IRP_MJ_CLOSE
                invoke CloseHandle, hDevice
            .else
                invoke MessageBox, NULL, $CTA0("Device is not present."), NULL, MB_ICONSTOP
            .endif
            ; DriverUnload proc in our driver will be called
            invoke ControlService, hService, SERVICE_CONTROL_STOP, addr _ss
        .else
            invoke MessageBox, NULL, $CTA0("Can't start driver."), NULL, MB_OK + MB_ICONSTOP
        .endif
        invoke DeleteService, hService
        invoke CloseServiceHandle, hService
    .else
        invoke MessageBox, NULL, $CTA0("Can't register driver."), NULL, MB_OK + MB_ICONSTOP
    .endif
    invoke CloseServiceHandle, hSCManager
.else
    invoke MessageBox, NULL, $CTA0("Can't connect to Service Control Manager."), NULL, \
                       MB_OK + MB_ICONSTOP
.endif

invoke ExitProcess, 0

start endp

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

end start
```

Not considering the code that prepares the input data sending to the device and the code responsible for formatting and displaying the output from the device, there is a few new stuff here - only three calls: CreateFile, DeviceIoControl and CloseHandle. All these functions accept the device (I repeat, not the driver) handle as an argument.

4.2.2 Device object

After loading, VirtToPhys driver creates the named device "devVirtToPhys" (The "dev" prefix is not necessary, but I have added it with purpose - I'll tell you why below).

The device name is placed in the *Object Manager namespace*. Object Manager is the system component responsible for creating, deleting, protecting, and tracking objects. By convention, device objects are placed in the \Device directory, inaccessible by the applications using the Win32 API.

To traverse the namespace maintained by the Object Manager use my Windows Object Explorer (WinObjEx) (http://www.wasm.ru/) or Object Viewer by Mark Russinovich (http://www.sysinternals.com/).

To view objects created by VirtToPhys on your computer, simply run VirtToPhys.exe, but do not close a dialog window.

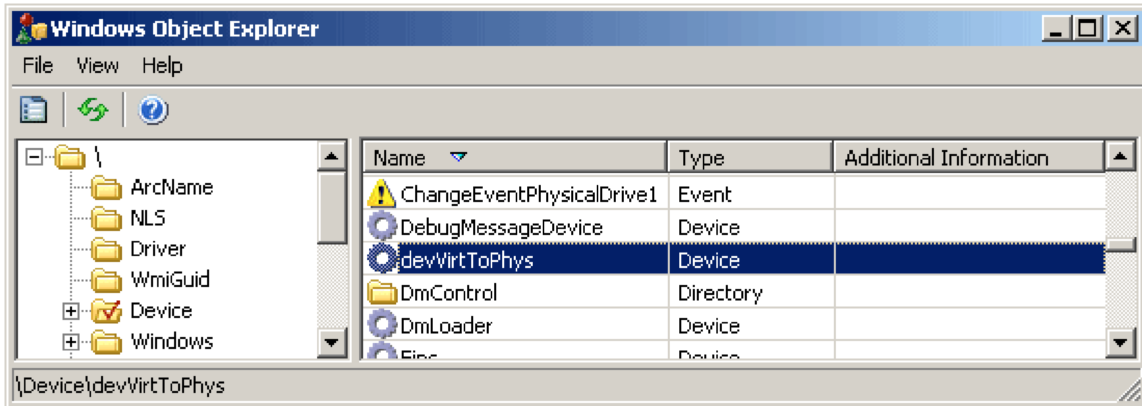

Figure 4-2. devVirtToPhys device object in the object manager namespace

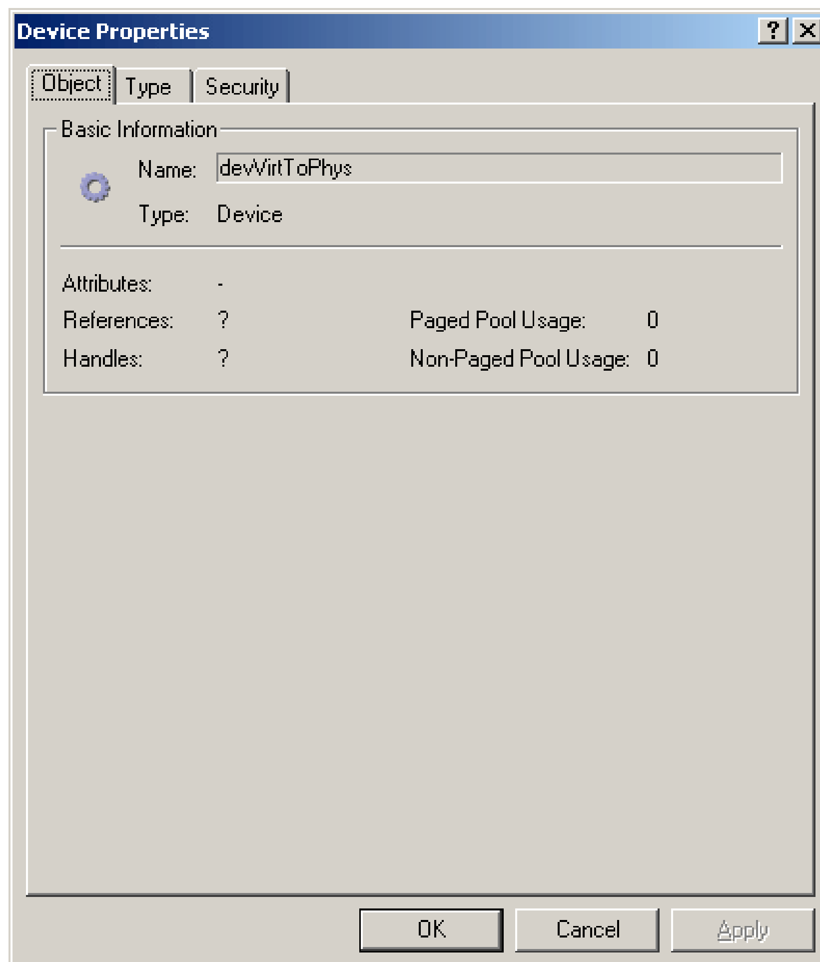

Figure 4-3. devVirtToPhys device object properties

4.2.3 Driver object

VirtToPhys driver object (I did not use any prefixes in the name) is placed in the \Driver directory.

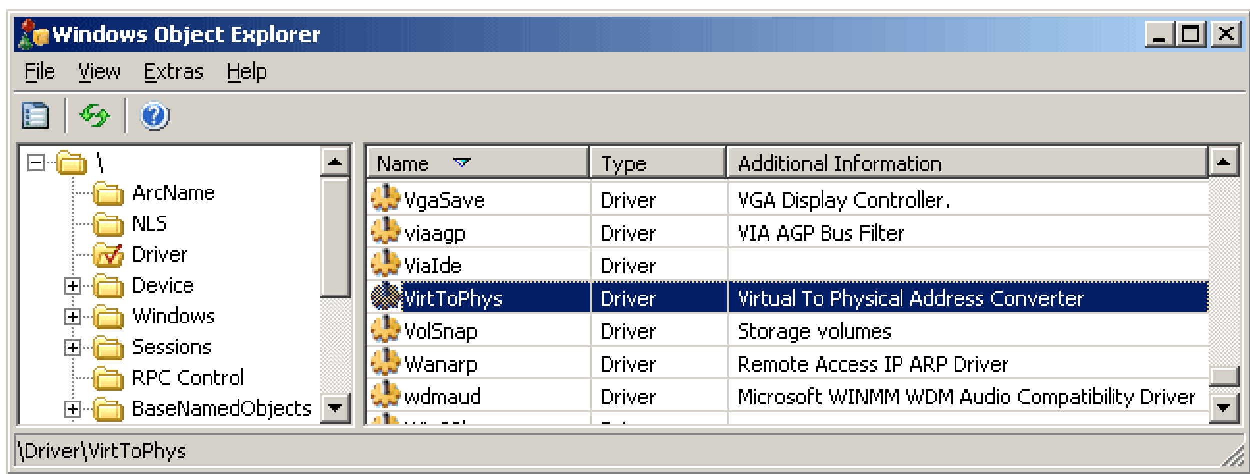

Figure 4-4. VirtToPhys driver object in the object manager namespace

4.2.4 Symbolic link object

Internal device names can't be used in Win32 applications (all directories, except for "\BaseNamedObjects" and "\??", are invisible to user programs) - instead, the device name must appear in a special directory in the Object Manager's namespace, "\??". This directory contains a *symbolic links* to the real, internal device names. Device drivers are responsible for creating links in this directory so their devices will be accessible to Win32 applications.

So, our driver needs to make it possible for user-mode code to open the device object, thus it have to create a symbolic link in the "\??" directory which points to the device object in the "\Device" directory. Thereafter, when the caller wants to have the device handle, the I/O Manager can find the device object directly.

By the way, you can examine or even change these links from user-mode with the Win32 QueryDosDevice and DefineDosDevice functions.

Having opened "\??" directory you will see, that it teems with symbolic links. Prior to Windows NT 4, this directory was named \DosDevices; it was renamed to "\??" for performance reasons - that name places first in the alphabetical order.

For backward compatibility in the Object Manager's namespace root directory there is the "\DosDevices" link to the "\??" directory.

The driver VirtToPhys creates the symbolic link "slVirtToPhys" to the device "devVirtToPhys" in the "\??" directory, which value is the string "\Device\devVirtToPhys". Here I've used the prefix "dev".

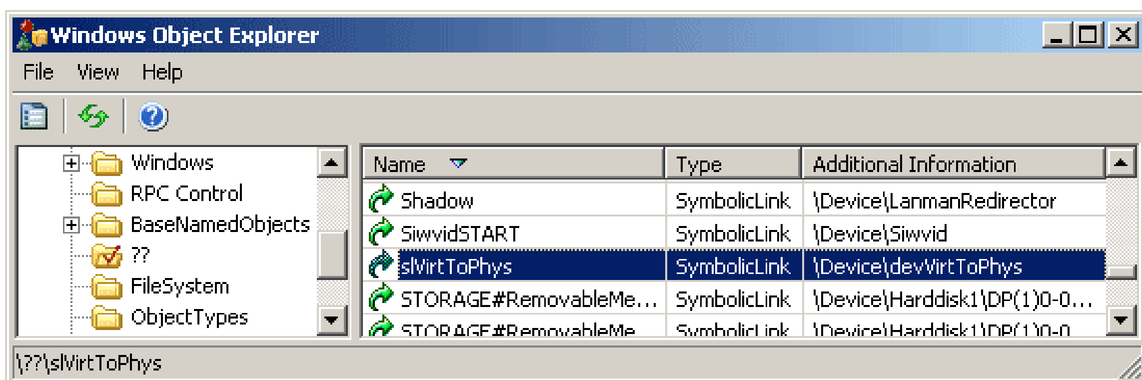

Figure 4-5. slVirtToPhys symbolic link object in the object manager namespace

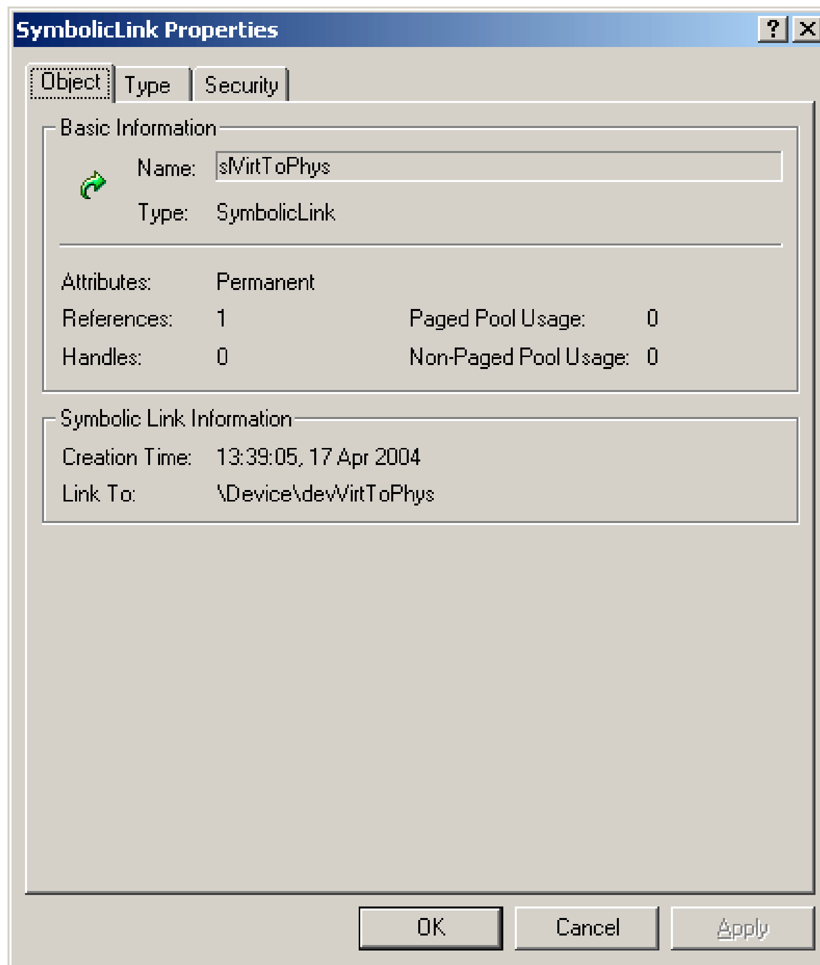

Figure 4-6. slVirtToPhys symbolic link object properties

I've added the prefixes only to distinguish the different kinds of objects. The point is to show it's not necessary for the device name and the symbolic link name to be (though, usually it's the case) coincided with the name of the driver. The important thing here is that the symbolic link name should specify the valid device name. And one more important point - there can not be two objects with the same name in a single object directory, just as there are no two files with identical name in the same file system directory.

Thus, upon exit from the StartService function we have three new objects: the driver "\Driver\VirtToPhys", the device "\Device\devVirtToPhys" and the symbolic link to the device " \??\slVirtToPhys".

If you still remember, in the second part of this doc, I have promised to tell what "\??", preceding the driver's file path like "\??\C:\masm32\..." is. So, "\??\C:" is a symbolic link to the internal device named "\Device\HarddiskVolume1", or the first volume on the first hard drive in the system.

4.2.5 File object

Let's get back to our source code. After the driver is started, we want to call it somehow. To accomplish this, we only need to open a file handle to the driver calling CreateFile:

The description of the CreateFile takes a a lot of space in the documentation. But only the small piece of that info is concerned about the device drivers.

```
CreateFile proto stdcall     lpFileName:LPCSTR,             dwDesiredAccess:DWORD, \
                             dwShareMode:DWORD,             lpSecurityAttributes:LPVOID, \
                             dwCreationDistribution:DWORD,  dwFlagsAndAttributes:DWORD, \
                             hTemplateFile:HANDLE
```

Despite its name, this function creates or opens existing (many Create* functions work this way) object, but not just a file. Microsoft definitely should name it CreateObject. The device can appear as an object.

Parameter	Description	
lpFileName	Points to a null-terminated string that specifies the name of the device to open. The symbolic link name pointing to the device object, to be exact.	
dwDesiredAccess	Specifies the type of access to the device. We will need two values:	
	GENERIC_READ	Specifies read access. Data can be read from the device;

	GENERIC_WRITE	Specifies write access. Data can be written to the device.
	These flags can be combined together.	
dwShareMode	Set of bit flags that specifies how the device can be shared. Three values can be useful to us:	
	0	The device cannot be shared. Subsequent open operations on the device will fail, until the handle is closed. Though the documentation stands this, I could not achieve it using 0.
	If you need to share the device, use the following values:	
	FILE_SHARE_READ	Subsequent open operations on the device will succeed only if read access is requested;
	FILE_SHARE_WRITE	Subsequent open operations on the device will succeed only if write access is requested.
lpSecurityAttributes	Pointer to a SECURITY_ATTRIBUTES. Since any special protection is not necessary for us and we don't need returned handle to be inherited by child processes we simply specify NULL here.	
dwCreationDistribution	Specifies the action to take on files that exist, and which action to take when files do not exist. For devices, this parameter must be always OPEN_EXISTING.	
dwFlagsAndAttributes	Specifies the attributes and flags. This parameter will be always equal 0.	
hTemplateFile	Specifies a handle to a template file. For devices, this parameter must be always NULL.	

If CreateFile successfully creates or opens the specified device, a handle to a device is returned; otherwise, INVALID_HANDLE_VALUE is returned.

Most Windows functions that return a handle return NULL when they are unsuccessful. CreateFile, however, returns INVALID_HANDLE_ VALUE defined as -1.

We call CreateFile as follows.

```
        invoke CreateFile, $CTA0("\\\\.\\slVirtToPhys"), GENERIC_READ + GENERIC_WRITE, \
                            0, NULL, OPEN_EXISTING, 0, NULL
```

I hope everything is clear with the last five parameters. The second parameter is a combination of flags GENERIC_READ + GENERIC_WRITE, since we intend both to send the data to the device and to receive the results of its work.

Let's examine the first parameter. It is a pointer to the symbolic link name, in the form "\\.\slVirtToPhys". The "\\.\" is a Win32-defined alias for the local computer. The CreateFile is the wrap around the other function NtCreateFile (realized in \%SystemRoot%\System32\ntdll.dll), which in turn accesses the corresponding system service (don't confuse to Win32 service processes).

> System service is an entry point into the kernel from environment subsystems. A system service dispatch is triggered as a result of executing an int 2Eh (Windows NT/W2K) or sysenter (Windows XP/2003) instruction on x86 processors. Executing these instruction results in a trap that causes the executing thread to transition into kernel-mode and enter the system service dispatcher.

NtCreateFile substitutes an alias for the local computer "\\.\" with the "\??" (thus "\\.\slVirtToPhys" turns to "\??\slVirtToPhys") and calls the kernel's ObOpenObjectByName function. Through the symbolic link ObOpenObjectByName finds the "\Device\devVirtToPhys" object and returns the pointer to it (thus the symbolic link visible from the user-mode code is used by the Object Manager for compilation in the internal device name). Using this pointer NtCreateFile creates the new file object representing the device and returns its handle.

> The operating system abstracts all I/O requests as operations on a virtual file, hiding the fact that the target of an I/O operation might not be a file-structured device. The driver converts the requests from requests made to a virtual file to hardware-specific requests. This abstraction generalizes an application's interface to devices. All data that is read or written is regarded as a simple stream of bytes directed to these virtual files.

Before CreateFile returns, the I/O Manager creates IRP the type of IRP_MJ_CREATE and sends it to the driver for processing. The driver-defined routine that is responsible for processing this type of IRP will execute in the same thread context as the initiator of the I/O requests (the caller of the CreateFile) at IRQL = PASSIVE_LEVEL. If that driver-defined routine successfully returns, the Object Manager creates a handle for the file object in the process's handle table and the handle propagates back through the calling chain, finally reaching the application as a return parameter from CreateFile.

The newly created file object is the executive object and does not get into the Object Manager namespace. You can use Process Explorer utility by Mark Russinovich (http://www.sysinternals.com) to explore such objects.

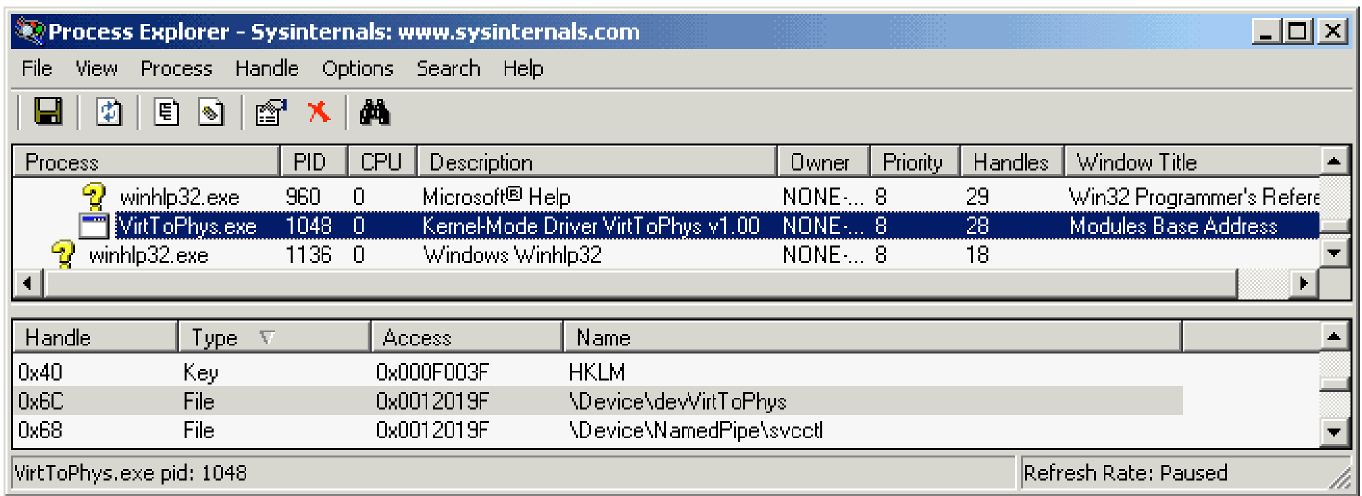

Figure 4-7. File object

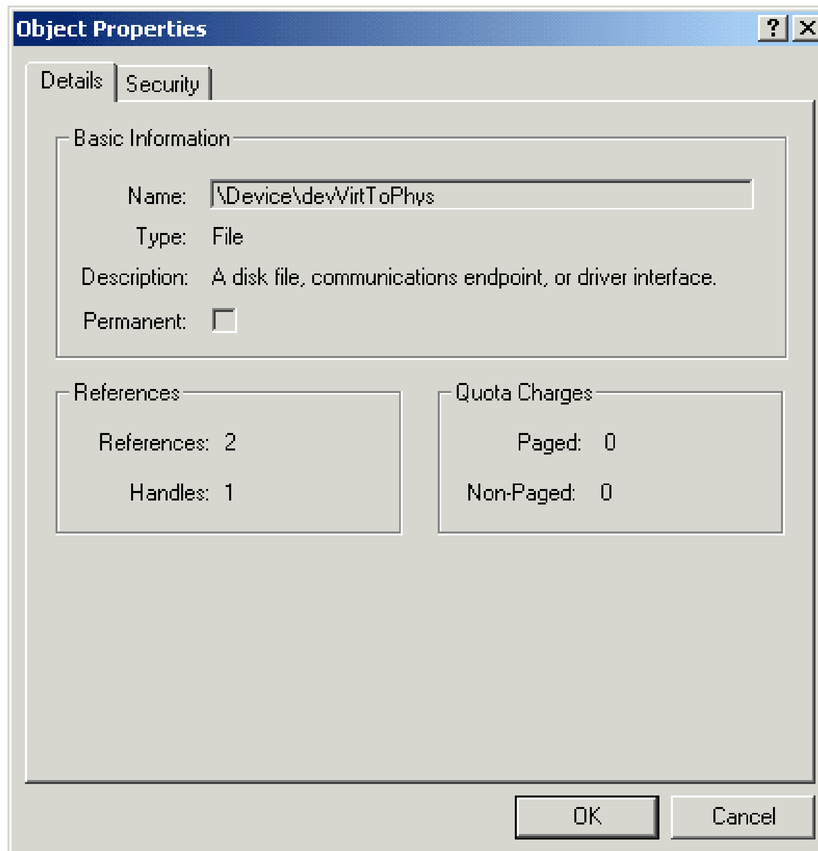

Figure 4-8. File object properties

Let me summarize. So, "\\.\slVirtToPhys" turns to the symbolic link "\??\slVirtToPhys" and finally used to find the appropriate device "\Device\devVirtToPhys". From the device object DEVICE_OBJECT is fetched out which driver is responsible for managing this device. Then I/O Manager sends IRP_MJ_CREATE request directly to this driver. This way the driver knows that some code tries to get the access to its device. If the driver wants to grant the access it returns success. Now the Object Manager creates a handle for the virtual file object representing the device and returns it to the user-mode code.

> Handles and symbolic links serve as indirect pointers to system resources; this indirection keeps application programs from fiddling directly with system data structures.

4.2.6 Communicating with the device

```
            .if eax != INVALID_HANDLE_VALUE
                mov hDevice, eax
```

If CreateFile returned valid device handle we save it in hDevice variable. Now we are able to communicate with the device calling ReadFile, WriteFile, and DeviceIoControl. The DeviceIoControl is the universal function to communicate with the devices. Here is its prototype:

```
DeviceIoControl proto stdcall hDevice:HANDLE,         dwIoControlCode:DWORD, \
                              lpInBuffer:LPVOID,      nInBufferSize:DWORD, \
                              lpOutBuffer:LPVOID,     nOutBufferSize:DWORD, \
                              lpBytesReturned:LPVOID, lpOverlapped:LPVOID
```

The DeviceIoControl accepts even more parameters than CreateFile, but it's simple enough.

Parameter	Description
hDevice	Handle to the device;
dwIoControlCode	-Control code that indicates what control operation to perform; We'll discuss how to define these codes a bit further on.
lpInBuffer	Pointer to a buffer that contains the data required to perform the operation. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not require input data;
nInBufferSize	Specifies the size, in bytes, of the buffer pointed to by lpInBuffer;
lpOutBuffer	Pointer to a buffer that receives the operation's output data. This parameter can be NULL if the dwIoControlCode parameter specifies an operation that does not produce output data;
nOutBufferSize	Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer;
lpBytesReturned	Pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by lpOutBuffer;
lpOverlapped	Pointer to an OVERLAPPED structure. This structure is required to help control an asynchronous operation. As we only want to call our driver synchronously (the DeviceIoControl will not return until the appropriate driver's routine will complete), we pass NULL.

4.2.7 I/O Control Codes

The device driver can be considered as a package of kernel-mode functions. I/O Control Code defines which function will be called. The dwIoControlCode argument to DeviceIoControl is used for this purpose. It indicates the control operation we want to perform and how it should be performed.

The control code is a 32-bit numeric constant that can be defined using the CTL_CODE macro that's part of both the winioctl.inc and the ntddk.inc include files.

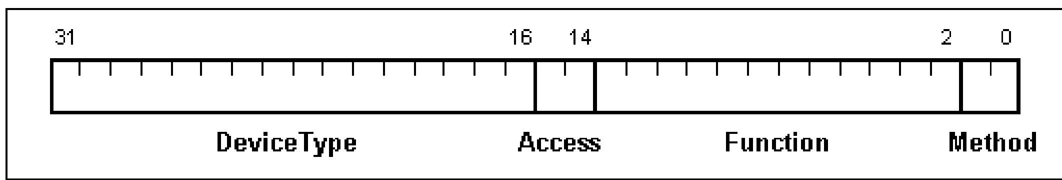

Figure 4-9. I/O Control Code Layout

<table>
<tr><th>Bit field</th><th colspan="2">Description</th></tr>
<tr><td>DeviceType</td><td colspan="2">The device type (16 bits) indicates the type of the device that implements this control operation.
Values in the range 0 - 7FFFh are reserved by Microsoft. Values in the range 8000h - 0FFFFh are available for developers of new kinds of kernel-mode drivers.
In \include\w2k\ntddk.inc you can find a set of FILE_DEVICE_XXX symbolic constants which values are from the range reserved by Microsoft. We will use FILE_DEVICE_UNKNOWN. However you can define another FILE_DEVICE_XXX.</td></tr>
<tr><td rowspan="5">Access</td><td colspan="2">The access code (2 bits) indicates the access rights an application needs to its device handle to issue this control operation.
As this field is only two bits long, we have four possibilities here:</td></tr>
<tr><td>FILE_ANY_ACCESS (0)</td><td>Maximum access rights. The driver will carry out the requested operation for any caller that has a handle for its device.</td></tr>
<tr><td>FILE_READ_ACCESS (1)</td><td>Read access rights. With this required access, the device driver transfers data from the device to memory buffer.</td></tr>
<tr><td>FILE_WRITE_ACCESS (2)</td><td>Write access rights. With this required access, the device driver transfers data from memory buffer to its device.</td></tr>
<tr><td>FILE_READ_ACCESS or FILE_WRITE_ACCESS (3)</td><td>Both read and write access rights. With this required access, the device driver transfers data between memory buffer and the device.</td></tr>
<tr><td>Function</td><td colspan="2">The function code (12 bits) indicates precisely which control operation this code describes.
It can take any value in the range 800h - 0FFFh for private I/O control codes. Values in the range 0 - 7FFh are reserved by Microsoft for public I/O control codes.</td></tr>
<tr><td rowspan="2">Method</td><td colspan="2">The buffering method (2 bits) indicates how the I/O Manager will handle the input and output buffers supplied by the application.
This field is two bits long, so four values can be used as one of the following system-defined constants:</td></tr>
<tr><td>METHOD_BUFFERED (0)</td><td>buffered I/O;</td></tr>
</table>

	METHOD_IN_DIRECT (1) METHOD_OUT_DIRECT (2)	direct I/O;
	METHOD_NEITHER (3)	neither I/O.

We'll talk about buffer management in more details later. Now the important thing is the buffered method is most safe, since the system takes care about buffers handling, resulting in an overhead of the memory copy operation. But drivers commonly use buffered I/O when callers transfer requests smaller than one page (4 KB), because the copy operation of small buffers matches the overhead of the memory lock performed by direct I/O. And we use buffered method in VirtToPhys driver.

You can form I/O control code manually, but it's much more convenient to use a macro CTL_CODE, that offers a mechanism to generate IOCTL values. Here it is:

```
CTL_CODE MACRO DeviceType:=<0>, Function:=<0>, Method:=<0>, Access:=<0>
    EXITM %(((DeviceType) SHL 16) OR ((Access) SHL 14) OR ((Function) SHL 2) OR (Method))
ENDM
```

As I already have said the CTL_CODE macro is defined both in the winioctl.inc, which included in the source code of the service control program, and in the ntddk.inc, included in the driver's source code.

Since we use NUM_DATA_ENTRY, DATA_SIZE constants and IOCTL_GET_PHYS_ADDRESS I/O control code both in the service control program and in the driver, they placed in separate include file common.inc. Thus all changes in this file will be mirrored in the both source codes.

```
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)
```

4.2.8 Data exchange

Now let's return to the driver's source code.

```
                    lea esi, adwInBuffer
                    assume esi:ptr DWORD
                    invoke GetModuleHandle, NULL
                    mov [esi][0*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("kernel32.dll", szKernel32)
                    mov [esi][1*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("user32.dll", szUser32)
                    mov [esi][2*(sizeof DWORD)], eax
                    invoke GetModuleHandle, $CTA0("advapi32.dll", szAdvapi32)
                    mov [esi][3*(sizeof DWORD)], eax
```

Here we fill adwInBuffer buffer with the virtual addresses to be converted.

```
                    lea edi, adwOutBuffer
                    assume edi:ptr DWORD
                    invoke DeviceIoControl, hDevice, IOCTL_GET_PHYS_ADDRESS, \
                                            esi, sizeof adwInBuffer, \
                                            edi, sizeof adwOutBuffer, \
                                            addr dwBytesReturned, NULL
```

And by calling DeviceIoControl we pass the buffer to the driver, that has to convert each virtual address to physical one.

```
                    .if ( eax != 0 ) && ( dwBytesReturned != 0 )

                        invoke GetModuleFileName, [esi][0*(sizeof DWORD)], \
                                                  addr acModulePath, sizeof acModulePath

                        lea ecx, acModulePath[eax-5]
                        .repeat
                            dec ecx
                            mov al, [ecx]
                        .until al == '\'
                        inc ecx
                        push ecx
```

```
            CTA0 "%s \t%08Xh\t%08Xh   ( %s )\n", szFmtMod

            invoke BigNumToString, [edi][0*(sizeof DWORD)], addr acNumber
            pop ecx
            invoke wsprintf, addr acThis, addr szFmtMod, ecx, \
                             [esi][0*(sizeof DWORD)], \
                             [edi][0*(sizeof DWORD)], addr acNumber

            invoke BigNumToString, [edi][1*(sizeof DWORD)], addr acNumber
            invoke wsprintf, addr acKernel, addr szFmtMod, addr szKernel32, \
                             [esi][1*(sizeof DWORD)], \
                             [edi][1*(sizeof DWORD)], addr acNumber

            invoke BigNumToString, [edi][2*(sizeof DWORD)], addr acNumber
            invoke wsprintf, addr acUser, addr szFmtMod, addr szUser32, \
                             [esi][2*(sizeof DWORD)], \
                             [edi][2*(sizeof DWORD)], addr acNumber

            invoke BigNumToString, [edi][3*(sizeof DWORD)], addr acNumber
            invoke wsprintf, addr acAdvapi, addr szFmtMod, addr szAdvapi32, \
                             [esi][3*(sizeof DWORD)], \
                             [edi][3*(sizeof DWORD)], addr acNumber

            invoke wsprintf, addr acBuffer, \
                             $CTA0("Module:\t\tVirtual:\t\tPhysical:\n\n%s\n%s%s%s"), \
                             addr acThis, addr acKernel, addr acUser, addr acAdvapi

            assume esi:nothing
            assume edi:nothing
            invoke MessageBox, NULL, addr acBuffer, $CTA0("Modules Base Address"), \
                               MB_OK + MB_ICONINFORMATION
        .else
            invoke MessageBox, NULL, $CTA0("Can't send control code to device."), NULL, \
                               MB_OK + MB_ICONSTOP
        .endif
```

If DeviceIoControl successfully returns, dwBytesReturned is equal to number of bytes we have in adwOutBuffer buffer filled by the driver. Now our task is simple. We have to format derived info and show it to our user. I'm sure you smart enough to understand it by yourself what is going on here. The escape sequences used in $CTA0 are the common one (see \Macros \Strings.mac for details).

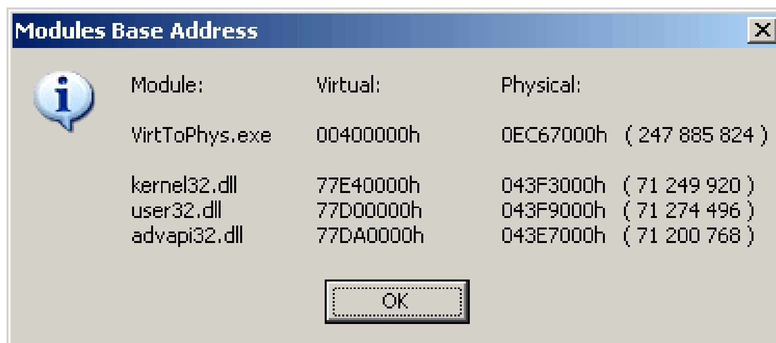

Figure 4-10. The output of VirtToPhys.exe

4.2.9 Cleanup

```
        invoke CloseHandle, hDevice
```

Now we only have to close opened device handle. At this point I/O Manager sends two IRPs to device driver. Firstly it is IRP_MJ_CLEANUP, telling the driver that its device handle is about to close. And then IRP_MJ_CLOSE, telling the driver that its device handle have been closed. By the way, you can prevent closing your device handle by returning error code from the routine responsible for handling IRP_MJ_CLEANUP request. The driver-defined routines responsible for processing these types of IRP will execute in the same thread context as the initiator of the I/O requests (the caller of the CloseHandle) at IRQL = PASSIVE_LEVEL.

We'll talk in the next part about how the driver handles IRP.

To make the drivers work under previous builds of Windows NT you need to change "\??" to "\DosDevices" and recompile the driver, since, as I have already mentioned, prior to Windows NT 4 "\??" directory was named "\DosDevices".

Copyright © 2002-2004 Four-F, four-f@mail.ru