The Simplest Device Drivers


Source code: KmdKit\examples\simple\Beeper


Source code: KmdKit\examples\simple\DateTime

3.1 How to compile and link the kernel-mode device driver

I always place driver's source code into a batch file. Such file is a mixture of *.bat and *.asm files, but has "bat" extension.

```
;@echo off
;goto make


.386                    ; driver's code start

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ; the rest of the driver's code ;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

end DriverEntry           ; driver's code end


:make

set drv=drvname

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

del %drv%.obj

echo.
pause
```

If you run such "self-compiling" file the following will occur. First two commands are commented out, thus they ignored by masm compiler, but accepted by command processor, that in turn ignores semicolon symbol. The control jumps to :make label where some options for the compiler and linker are specified. All instructions following the end directive is ignored by the compiler. Thus all lines between goto make command and :make label are ignored by the command processor but accepted by the compiler. And all that is outside (including goto make command and :make label) is ignored by the compiler but accepted by the command processor. This method is extremely convenient, since the source code itself keeps all the info about how it should be compiled and linked. Also you can simply add some extra processing if you need. I use this method for all my drivers. Since the control programs usually don't require anything special you can compile it as you like.

```
set drv=drvname
```

We define an environment variable, which will be the substitution for the driver's file name.

The used linker options means:

Command Key	Description
/driver	- tells the linker it should create a Windows NT kernel-mode driver; The most important effect of this option is the addition of a new section called "INIT". Here goes the part of ".idata" section, which contains a list of IMAGE_IMPORT_DESCRIPTOR structures and the names of imported functions and modules. This "INIT" section is marked as discardable in order to NT image loader discard it after locating the imported function addresses.
/base:0x10000	- sets the driver base address equal to 10000h. We have already discussed about it in the preceding part;
/align:32	- the system memory is a precious resource. Therefore, the device driver's files have more "fine" section alignment;
/out:%dvr%.sys	- the linker creates *.exe files by default. Or builds a *.dll if the /DLL option is specified. We should force the linker to create *.sys file.
/subsystem:native	- In the PE header of every executable file there is a field that tells the image loader which subsystem this module requires: Win32, POSIX or OS/2. It's necessary to place a driver's image into the appropriate environment. When we compile *.exe or *.dll file, we usually indicate under this option that the executable file requires a Win32 subsystem. The kernel-mode drivers don't require any subsystem at all, since they run in the native environment.

3.2 The simplest possible kernel-mode device driver

3.2.1 Simplest driver source code

Here is the source code of the simplest possible kernel-mode device driver.

```
;@echo off
;goto make

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
; simplest - Simplest possible kernel-mode driver
;
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.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

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

.code

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

DriverEntry proc pDriverObject:PDRIVER_OBJECT, pusRegistryPath:PUNICODE_STRING

    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret

DriverEntry endp

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

end DriverEntry

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                               B U I L D I N G   D R I V E R
```

```
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:make

set drv=simplest

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

del %drv%.obj

echo.
pause
```

3.2.2 DriverEntry Routine

Like any other executable module each driver must have the entry point, which is called when the driver is loaded into the memory. The driver's entry point is the DriverEntry routine. This name is conventionally given to the main entry point of a kernel-mode device driver. You may rename it anything you like. The DriverEntry routine initializes driver-wide data structures. The prototype for DriverEntry routine is defined as follows:

```
DriverEntry proto DriverObject:PDRIVER_OBJECT, RegistryPath:PUNICODE_STRING
```

Unfortunately well-known "hungarian notation" by Charles Simonyi is not used in DDK. I will use it everywhere if possible. Therefore, I have added the prefixes to the DriverObject and the RegistryPath.

The data types of PDRIVER_OBJECT and PUNICODE_STRING are defined in \include\w2k\ntddk.inc and \include\w2k\ntdef.inc respectively.

```
PDRIVER_OBJECT  typedef PTR DRIVER_OBJECT
PUNICODE_STRING typedef PTR UNICODE_STRING
```

When the I/O Manager calls the DriverEntry routine it passes two pointers to it:

Parameter	Description
pDriverObject	- a pointer to a barely initialized driver object that represents the driver. Windows NT is an object-oriented operating system. So, the drivers are represented as objects. By the loading of the driver into the memory the system creates driver object which represents the given driver. The driver object is nothing more then DRIVER_OBJECT structure (defined in \include\w2k\ntddk.inc). The pDriverObject pointer gives the driver an access to that structure. But we don't need to touch it this time.
pusRegistryPath	- a pointer to a counted Unicode string that specifies a path to the driver's registry subkey. We have discussed about the driver's registry subkey in the previous part. The driver can use this pointer to store or retrieve some driver-specific information. If a driver will need to use the path after its DriverEntry routine has completed, the driver should save a copy of the unicode string, not the pointer itself since it has no meaning outside the DriverEntry routine.

Counted Unicode String is also the structure of type UNICODE_STRING. Unlike the user-mode code, the kernel-mode code operates with the strings in UNICODE_STRING format. It's defined in \include\w2k\ntdef.inc like this:

```
UNICODE_STRING STRUCT
    _Length        WORD    ?
    MaximumLength  WORD    ?
    Buffer         PWSTR   ?
UNICODE_STRING ENDS
```

Parameter	Description
_Length	- The length of the string in bytes (not characters), not counting the terminating null character (I had to change an original Length name, since it is a masm reserved word);
MaximumLength	- The length in bytes (not characters) of the buffer pointed by Buffer member;
Buffer	- Pointer to the Unicode-string itself. Don't expect it as always zero-terminated. It does not sometimes!

The main advantage of this format is its clear determination of both the current string length, and its maximum possible length. It allows avoid some additional calculations.

The above-described driver (\src\Article2-3\simplest\simplest.sys) is the simplest one. The only thing it does allows to load itself. Since it can't do anything more it returns an error code STATUS_DEVICE_CONFIGURATION_ERROR (see \include\w2k \ntstatus.inc for complete list of possible error codes). If you return STATUS_SUCCESS the driver will remain in the memory,

and you can't unload it, since we have not defined the DriverUnload routine responsible for this.

You can register and load any driver with the KmdManager utility.

3.3 Beeper device driver

3.3.1 Beeper driver source code

Now let's examine the beeper driver. Last time we have written its control program.

```
;@echo off
;goto make

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;  beeper - Kernel Mode Drive
;  Makes beep thorough computer speaker
;
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.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\hal.inc

includelib \masm32\lib\w2k\hal.lib

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                     E Q U A T E S
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

TIMER_FREQUENCY        equ 1193167                   ; 1,193,167 Hz
OCTAVE                 equ 2                         ; octave multiplier

PITCH_C                equ 523                       ; C        -  523,25 Hz
PITCH_Cs               equ 554                       ; C#       -  554,37 Hz
PITCH_D                equ 587                       ; D        -  587,33 Hz
PITCH_Ds               equ 622                       ; D#       -  622,25 Hz
PITCH_E                equ 659                       ; E        -  659,25 Hz
PITCH_F                equ 698                       ; F        -  698,46 Hz
PITCH_Fs               equ 740                       ; F#       -  739,99 Hz
PITCH_G                equ 784                       ; G        -  783,99 Hz
PITCH_Gs               equ 831                       ; G#       -  830,61 Hz
PITCH_A                equ 880                       ; A        -  880,00 Hz
PITCH_As               equ 988                       ; B        -  987,77 Hz
PITCH_H                equ 1047                      ; H        - 1046,50 Hz

; We are going to play c-major chord

TONE_1                 equ TIMER_FREQUENCY/(PITCH_C*OCTAVE)
TONE_2                 equ TIMER_FREQUENCY/(PITCH_E*OCTAVE)
TONE_3                 equ (PITCH_G*OCTAVE)           ; for HalMakeBeep

DELAY                  equ 1800000h                   ; for my ~800mHz box

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                         M A C R O S
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DO_DELAY MACRO
    mov eax, DELAY
    .while eax
        dec eax
    .endw
ENDM

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

.code

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
;                                         MakeBeep1
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

MakeBeep1 proc dwPitch:DWORD

    ; Direct hardware access

    cli

    mov al, 10110110y
    out 43h, al

    mov eax, dwPitch
    out 42h, al

    mov al, ah
    out 42h, al

    ; Turn speaker ON

    in al, 61h
    or  al, 11y
    out 61h, al

    sti

    DO_DELAY

    cli

    ; Turn speaker OFF

    in al, 61h
    and al, 11111100y
    out 61h, al

    sti

    ret

MakeBeep1 endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                         MakeBeep2
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

MakeBeep2 proc dwPitch:DWORD

    ; Hardware access using WRITE_PORT_UCHAR and READ_PORT_UCHAR
    ; functions from hal.dll

    cli

    invoke WRITE_PORT_UCHAR, 43h, 10110110y

    mov eax, dwPitch
    invoke WRITE_PORT_UCHAR, 42h, al
    mov eax, dwPitch
    invoke WRITE_PORT_UCHAR, 42h, ah

    ; Turn speaker ON

    invoke READ_PORT_UCHAR, 61h
    or  al, 11y
    invoke WRITE_PORT_UCHAR, 61h, al

    sti

    DO_DELAY

    cli

    ; Turn speaker OFF

    invoke READ_PORT_UCHAR, 61h
    and al, 11111100y
    invoke WRITE_PORT_UCHAR, 61h, al

    sti

    ret

MakeBeep2 endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                        DriverEntry
```

```
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DriverEntry proc pDriverObject:PDRIVER_OBJECT, pusRegistryPath:PUNICODE_STRING

    invoke MakeBeep1, TONE_1
    invoke MakeBeep2, TONE_2

    ; Hardware access using hal.dll HalMakeBeep function

    invoke HalMakeBeep, TONE_3
    DO_DELAY
    invoke HalMakeBeep, 0

    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret

DriverEntry endp

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

end DriverEntry

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

:make

set drv=beeper

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

del %drv%.obj

echo.
pause
```

This driver is intended to beep c-major arpeggio using the motherboard speaker. For this purpose the driver uses IN and OUT CPU instructions, accessing the appropriate I/O ports. It is well-known that the access to the I/O ports is guarded by Windows NT as an important system resource. An attempt to execute IN or OUT instruction from user-mode results in termination of the process. But actually there is a way to bypass this limitation, i.e. to allow the user-mode to access the I/O ports directly. We'll talk about it a bit later.

3.3.2 Controlling the system timer

There are three timers inside the computer. These are known as timers 0, 1 and 2 and they reside in the *Programmable Interval Timer* (PIT). Timer 2 is used to control sound generation. The frequency at which timer oscillates is determined by an initial count value. The timer counts down from this value to zero, and when it reaches zero, the timer oscillates. The counter is then re-set to the predetermined initial count value and the process starts again. The counting down process is controlled by the main system oscillator, which runs at a frequency of 1,193,180 Hz. This value is fixed across the entire range of PC families. Every time it oscillates, the system timer counts down once. To vary the frequency at which the timer oscillates, we just need to give it a new initial count value. To calculate the frequency at which the speaker will sound we have to use this formula: 1193180/. You can find more detailed information searching the Web.

There is one subtlety here, which I misunderstand a bit. The QueryPerformanceFrequency from kernel32.dll really returns the value of 1193180. But in hal.dll's HalMakeBeep function I have found a little bit different value equal to 1193167. I'll use this value. Probably it's some time compensation. I don't know. Anyway it doesn't prevent us from beeping the speaker.

We play the first sound (do) of the c-major chord using MakeBeep1 routine.

```
    mov al, 10110110y
    out 43h, al
```

First we have to set the timer's control register. We should load a binary value 10110110 into port 43h to achieve this goal.

```
    mov eax, dwPitch
    out 42h, al

    mov al, ah
    out 42h, al
```

Then, in two consecutive statements, we load the low byte and high byte of the new initial count value into port 42h.

```
    in al, 61h
    or  al, 11y
    out 61h, al
```

Now we turn the speaker on by setting bits 0 and 1 of the value on port 61h. Now the speaker is producing the sound.

```
DO_DELAY MACRO
    mov eax, DELAY
    .while eax
        dec eax
    .endw
ENDM
```

We let the speaker sound for some time, using DO_DELAY macro. Yes - it is primitive, but is rather effective.

```
    in al, 61h
    and al, 11111100y
    out 61h, al
```

To turn the speaker off we need to clear bits 1 and 2 of the value on port 61h. Don't forget that the timer is a global system resource. Therefore we disable the maskable hardware interrupts by clearing the interrupt flag. It will be much more difficult on the multi-processor machine.

We play the second (mi) sound of the c-major chord using MakeBeep2 routine. It differs only by using WRITE_PORT_UCHAR and READ_PORT_UCHAR functions from hal.dll instead of in/out. HAL hides hardware-dependent details such as I/O interfaces (as in our case) and other, making it machine-independent.

The third (sol) sound of the c-major chord we play with the HalMakeBeep from hal.dll. As a parameter it is necessary not to use the initial count value, but the frequency value itself instead.

In the beginning of the beeper.bat file you will find all twelve key notes. I used only three of them. Others are left for your future synthesizer ;-). To turn the speaker off it is necessary to call alMakeBeep once again, passing 0 as an argument.

The beeper driver returns an error code to the system and is removed from the memory. I repeat. It is necessary to return an error code only to cause the system removes the driver from the memory. When we'll reach full-function drivers, we'll have to return STATUS_SUCCESS.

3.3.3 Starting the driver automatically

The scp.exe installs driver beeper.sys to be started on demand. Last time we have discussed different start types of the drivers. Now we try to force the system to start our driver automatically. It can be done in many ways. The simplest one is to comment the call to DeleteService out, change SERVICE_DEMAND_START to SERVICE_AUTO_START and SERVICE_ERROR_IGNORE to SERVICE_ERROR_NORMAL, then recompile scp.asm and execute it. After scp.exe exits the registry will contain the brand new service entry. Now you can completely forget about it. During the next system boot the driver beeper.sys will remind you about itself. In the Event Log you will find the entry about driver's startup failure. Select from the Start menu *Programs/ Administrative Tools/Event Viewer*, select System Log, and double-click on an Event Log entry to see it. You will see something like this:

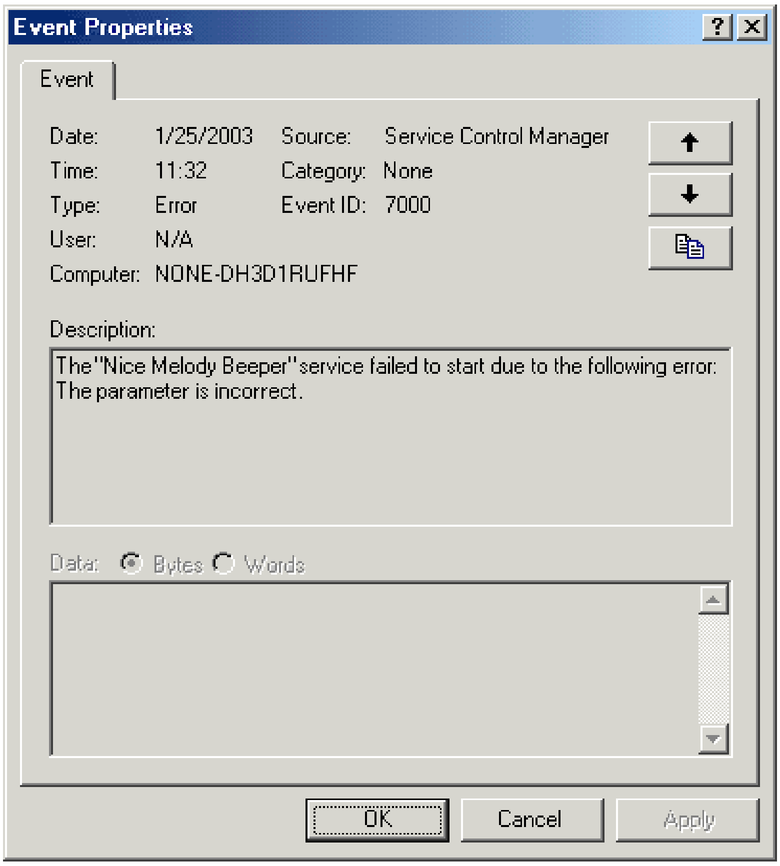

Figure 3-1. Failure Event Log entry

Don't forget to remove the registry entry otherwise you will hear that nice melody every time your system boots up.

3.4 Service Control Program for giveio driver

3.4.1 Giveio driver's SCP source code

Now let's take a look at another SCP to control giveio.sys driver.

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;  DateTime - Service Control Program for giveio 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\Macros\Strings.mac

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                         M A C R O S
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

CMOS MACRO by:REQ
    mov al, by
    out 70h, al
    in al, 71h
```

```
    mov ah, al
    shr al, 4
    add al, '0'

    and ah, 0Fh
    add ah, '0'
    stosw
ENDM

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

.code

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                             DateTime
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DateTime proc uses edi

local acDate[16]:CHAR
local acTime[16]:CHAR
local acOut[64]:CHAR

    ; See Ralf Brown's Interrupt List for details

    ;:::::::::::::::::: Set data format  ::::::::::::::::::

    mov al, 0Bh                 ; status register B
    out 70h, al
    in al, 71h

    push eax                    ; save old data format
    and al, 11111011y           ; Bit 2: Data Mode - 0: BCD, 1: Binary
    or al, 010y                 ; Bit 1: 24/12 hour selection - 1 enables 24 hour mode
    out 71h, al

    ;:::::::::::::::::::: Lets' fetch current date  ::::::::::::::::::::

    lea edi, acDate

    CMOS 07h                    ; date of month
    mov al, '.'
    stosb

    CMOS 08h                    ; month
    mov al, '.'
    stosb

    CMOS 32h                    ; two most significant digit od year
    CMOS 09h                    ; two least significant digit od year

    xor eax, eax                ; terminate string with zero
    stosb

    ;:::::::::::::::::::: Lets' fetch current time ::::::::::::::::::::

    lea edi, acTime

    CMOS 04h                    ; hours
    mov al, ':'
    stosb

    CMOS 02h                    ; minutes
    mov al, ':'
    stosb

    CMOS 0h                     ; seconds

    xor eax, eax                ; terminate string with zero
    stosb

    ;::::::::::::: restore old data format ::::::::::::

    mov al, 0Bh
    out 70h, al
    pop eax
    out 71h, al

    ;:::::::::::::::::: Show current date and time :::::::::::::::

    invoke wsprintf, addr acOut, $CTA0("Date:\t%s\nTime:\t%s"), addr acDate, addr acTime
    invoke MessageBox, NULL, addr acOut, $CTA0("Current Date and Time"), MB_OK

    ret
```

```
DateTime endp

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

start proc

local fOK:BOOL
local hSCManager:HANDLE
local hService:HANDLE
local acDriverPath[MAX_PATH]:CHAR

local hKey:HANDLE
local dwProcessId:DWORD

    and fOK, 0          ; assume an error

    ; Open the SCM database

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

        push eax
        invoke GetFullPathName, $CTA0("giveio.sys"), sizeof acDriverPath, addr acDriverPath, esp
        pop eax

        ; Register driver in SCM active database

        invoke CreateService, hSCManager, $CTA0("giveio"), $CTA0("Current Date and Time fetcher."), \
                SERVICE_START + DELETE, SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START, \
                SERVICE_ERROR_IGNORE, addr acDriverPath, NULL, NULL, NULL, NULL, NULL

        .if eax != NULL
            mov hService, eax

            invoke RegOpenKeyEx, HKEY_LOCAL_MACHINE, \
                                $CTA0("SYSTEM\\CurrentControlSet\\Services\\giveio"), \
                                0, KEY_CREATE_SUB_KEY + KEY_SET_VALUE, addr hKey

            .if eax == ERROR_SUCCESS

                ; Add current process ID into the registry

                invoke GetCurrentProcessId
                mov dwProcessId, eax
                invoke RegSetValueEx, hKey, $CTA0("ProcessId", szProcessId), NULL, REG_DWORD, \
                                        addr dwProcessId, sizeof DWORD

                .if eax == ERROR_SUCCESS

                    ; Start driver

                    invoke StartService, hService, 0, NULL
                    inc fOK                 ; Set OK flag
                    invoke RegDeleteValue, hKey, addr szProcessId
                .else
                    invoke MessageBox, NULL, $CTA0("Can't add Process ID into registry."), \
                                        NULL, MB_ICONSTOP
                .endif

                invoke RegCloseKey, hKey

            .else
                invoke MessageBox, NULL, $CTA0("Can't open registry."), NULL, MB_ICONSTOP
            .endif

            ; Remove driver from SCM database

            invoke DeleteService, hService
            invoke CloseServiceHandle, hService
        .else
            invoke MessageBox, NULL, $CTA0("Can't register driver."), NULL, MB_ICONSTOP
        .endif
        invoke CloseServiceHandle, hSCManager
    .else
        invoke MessageBox, NULL, $CTA0("Can't connect to Service Control Manager."), \
                                NULL, MB_ICONSTOP
    .endif

    ; If OK display current date and time to the user

    .if fOK
```

```
        invoke DateTime
    .endif

    invoke ExitProcess, 0

start endp

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

end start
```

3.4.2 Using the registry for passing some info to the driver

There is nothing new here except a few points.

```
            invoke RegOpenKeyEx, HKEY_LOCAL_MACHINE, \
                                $CTA0("SYSTEM\\CurrentControlSet\\Services\\giveio"), \
                                0, KEY_CREATE_SUB_KEY + KEY_SET_VALUE, addr hKey

            .if eax == ERROR_SUCCESS
                invoke GetCurrentProcessId
                mov dwProcessId, eax
                invoke RegSetValueEx, hKey, $CTA0("ProcessId", szProcessId), NULL, REG_DWORD, \
                                        addr dwProcessId, sizeof DWORD

                .if eax == ERROR_SUCCESS
                    invoke StartService, hService, 0, NULL
```

Before starting the driver we create additional ProcessId value under driver registry subkey. It contains current process identifier, which is the identifier of SCP itself. Please notice how I use $CTA0 macro here. I did specify a label szProcessId, which the text "ProcessId" will be marked with. This lets us to reference this text later on. My text macros are flexible enough, by the way.

If the new registry value was added successfully, we can start the driver. What this additional registry value is for you'll find out a bit later.

```
                    inc fOK
                    invoke RegDeleteValue, hKey, addr szProcessId
                .else
                    invoke MessageBox, NULL, $CTA0("Can't add Process ID into registry."), \
                                        NULL, MB_ICONSTOP
                .endif

                invoke RegCloseKey, hKey
```

Having returned from the StartService we consider that the driver has done its work and set fOK flag. The call to the RegDeleteValue is not necessary here, because all driver registry subkeys will be removed by the subsequent call to the DeleteService. But it's a good programming practice to clean up explicitly.

```
    .if fOK
        invoke DateTime
    .endif
```

Having removed the driver entry from the SCM database we close all opened handles and, if fOK flag is set, call DateTime function.

3.4.3 Accessing the CMOS

In a computer motherboard there is a microchip that is used to store some system configuration information, such as disk drive parameters, memory configuration, and the date-time. This microchip is often referred to as "the CMOS" (CMOS is an acronym stands for Complementary Metal Oxide Semiconductor). The microchip is battery-powered and has the real-time clock (RTC). We can obtain its data accessing 70h and 71h I/O ports. See "Ralf Brown's Interrupt List" for details (http://www-2.cs.cmu.edu/afs/cs/user/ralf/pub/WWW/files.html).

```
    mov al, 0Bh                 ; status register B
    out 70h, al
```

```
    in al, 71h

    push eax                    ; save old data format
    and al, 11111011y           ; Bit 2: Data Mode - 0: BCD, 1: Binary
    or al, 010y                 ; Bit 1: 24/12 hour selection - 1 enables 24 hour mode
    out 71h, al
```

Firstly we set a convenient data format using the status register B.

Using CMOS macro we can obtain all the information we need from CMOS and format it at the same time.

```
    invoke wsprintf, addr acOut, $CTA0("Date:\t%s\nTime:\t%s"), addr acDate, addr acTime
    invoke MessageBox, NULL, addr acOut, $CTA0("Current Date and Time"), MB_OK
```

Then we expose all retrieved data. And you should see something like this:

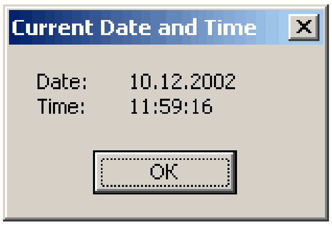

Figure 3-2. The output of the DateTime.exe

Most strange thing here is that we have accessed the SMOS memory without the system stops us. As I have already mentioned above, the access to I/O ports is protected under Windows NT. Executing IN or OUT instruction in user-mode will cause process termination. But we have touched them. How it can be? Well, it becomes possible due to the giveio driver.

3.5 Giveio device driver

3.5.1 Giveio driver source code

```
;@echo off
;goto make

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;  giveio - Kernel Mode Driver
;
;  Demonstrate direct port I/O access from a user mode
;
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.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

includelib \masm32\lib\w2k\ntoskrnl.lib

include \masm32\Macros\Strings.mac

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                     E Q U A T E S
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

IOPM_SIZE equ 2000h      ; sizeof I/O permission map

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

.code

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
;                                         DriverEntry
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DriverEntry proc pDriverObject:PDRIVER_OBJECT, pusRegistryPath:PUNICODE_STRING

local status:NTSTATUS
local oa:OBJECT_ATTRIBUTES
local hKey:HANDLE
local kvpi:KEY_VALUE_PARTIAL_INFORMATION
local pIopm:PVOID
local pProcess:LPVOID

    invoke DbgPrint, $CTA0("giveio: Entering DriverEntry")

    mov status, STATUS_DEVICE_CONFIGURATION_ERROR

    lea ecx, oa
    InitializeObjectAttributes ecx, pusRegistryPath, 0, NULL, NULL

    invoke ZwOpenKey, addr hKey, KEY_READ, ecx
    .if eax == STATUS_SUCCESS

        push eax
        invoke ZwQueryValueKey, hKey, $CCOUNTED_UNICODE_STRING("ProcessId", 4), \
                               KeyValuePartialInformation, addr kvpi, sizeof kvpi, esp
        pop ecx

        .if ( eax != STATUS_OBJECT_NAME_NOT_FOUND ) && ( ecx != 0 )

            invoke DbgPrint, $CTA0("giveio: Process ID: %X"), \
                                dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR [kvpi]).Data

            ; Allocate a buffer for the I/O permission map

            invoke MmAllocateNonCachedMemory, IOPM_SIZE
            .if eax != NULL
                mov pIopm, eax

                lea ecx, kvpi
                invoke PsLookupProcessByProcessId, \
                        dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR [ecx]).Data, addr pProcess
                .if eax == STATUS_SUCCESS

                    invoke DbgPrint, $CTA0("giveio: PTR KPROCESS: %08X"), pProcess

                    invoke Ke386QueryIoAccessMap, 0, pIopm
                    .if al != 0

                        ; I/O access for 70h port

                        mov ecx, pIopm
                        add ecx, 70h / 8
                        mov eax, [ecx]
                        btr eax, 70h MOD 8
                        mov [ecx], eax

                       ; I/O access for 71h port

                        mov ecx, pIopm
                        add ecx, 71h / 8
                        mov eax, [ecx]
                        btr eax, 71h MOD 8
                        mov [ecx], eax

                        invoke Ke386SetIoAccessMap, 1, pIopm
                        .if al != 0
                            invoke Ke386IoSetAccessProcess, pProcess, 1
                            .if al != 0
                                invoke DbgPrint, $CTA0("giveio: I/O permission is successfully given")
                            .else
                                invoke DbgPrint, $CTA0("giveio: I/O permission is failed")
                                mov status, STATUS_IO_PRIVILEGE_FAILED
                            .endif
                        .else
                            mov status, STATUS_IO_PRIVILEGE_FAILED
                        .endif
                    .else
                        mov status, STATUS_IO_PRIVILEGE_FAILED
                    .endif
                    invoke ObDereferenceObject, pProcess
                .else
                    mov status, STATUS_OBJECT_TYPE_MISMATCH
                .endif
                invoke MmFreeNonCachedMemory, pIopm, IOPM_SIZE
            .else
```

```
                invoke DbgPrint, $CTA0("giveio: Call to MmAllocateNonCachedMemory failed")
                mov status, STATUS_INSUFFICIENT_RESOURCES
            .endif
        .endif
        invoke ZwClose, hKey
    .endif

    invoke DbgPrint, $CTA0("giveio: Leaving DriverEntry")

    mov eax, status
    ret

DriverEntry endp

;//////////////////////////////////////////////////////////////////////////////////////////////
;
;//////////////////////////////////////////////////////////////////////////////////////////////

end DriverEntry

;//////////////////////////////////////////////////////////////////////////////////////////////
;
;//////////////////////////////////////////////////////////////////////////////////////////////

:make

set drv=giveio

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

del %drv%.obj

echo.
pause
```

The driver's code is based on well-known example (giveio) by Dale Roberts. I have decided it will be appropriate to mention here.

3.5.2 I/O permission bit map

Our driver changes the I/O permission bit map (IOPM) that allows the process free access to the I/O ports. Refer to this doc for details http://www.intel.com/design/intarch/techinfo/pentium/PDF/inout.pdf .

Each process has its own I/O permission bit map, thus access to the individual I/O ports can be granted to the individual process. Each bit in the I/O permission bit map corresponds to the byte I/O port. If this bit is set, the access to the corresponding port is forbidden, if it is clear the process may access this I/O port. Since the I/O address space consists of 64K individually addressable 8-bit I/O ports, the maximum IOPM size is 2000h bytes.

> The purpose of the TSS is to save the state of the processor during task or context switches. For performance reasons, Windows NT does not use this architectural feature and maintains one base TSS that all processes share. This means that IOPM is also shared. So any changes to it are not private for particular process but are system-wide.

There are some undocumented functions in the ntoskrnl.exe to manipulate with the IOPM: Ke386QueryIoAccessMap and Ke386SetIoAccessMap.

```
Ke386QueryIoAccessMap proto stdcall dwFlag:DWORD, pIopm:PVOID
```

Ke386QueryIoAccessMap copies current IOPM by the size of 2000h bytes from TSS to the memory buffer pointed to by pIopm parameter.

Parameter	Description
dwFlag	0 - Fill memory buffer with 0FFh values (all bits are set - access denied); 1 - Copy current IOPM from TSS to the memory buffer.
pIopm	Points to the memory buffer to receive current IOPM. The buffer size must be not less than 2000h bytes.

If the function succeeds it returns nonzero value in al (not eax) register. If the function fails, the al (not eax) register is clear.

```
Ke386SetIoAccessMap proto stdcall dwFlag:DWORD, pIopm:PVOID
```

Ke386SetIoAccessMap copies specified IOPM by the size of 2000h from the memory buffer pointed to by pIopm parameter to TSS.

Parameter	Description
dwFlag	It can be only 1 - permits copying. Any other value causes the function to return an error.
pIopm	Points to the memory buffer containing IOPM. The buffer size must not be less than 2000h bytes.

If the function succeeds it returns nonzero value in al (not eax) register. If the function fails, the al (not eax) register is clear.

After the IOPM has been copied to the TSS, the IOPM offset pointer must be adjusted to point to new IOPM. This is done by using Ke386IoSetAccessProcess - one more very useful and also completely undocumented function from the ntoskrnl.exe.

```
Ke386IoSetAccessProcess proto stdcall pProcess:PTR KPROCESS, dwFlag:DWORD
```

Ke386IoSetAccessProcess permits/forbids using IOPM for the process.

Parameter	Description
pProcess	Points to the KPROCESS structure (I'll tell you later about it).
dwFlag	0 - Denies access to the I/O ports, setting offset to the IOPM abroad of TSS segment; 1 - Allows access to the I/O ports, setting offset to the IOPM in limits TSS segment (88h).

If the function succeeds it returns nonzero value in al (not eax) register. If the function fails, the al (not eax) register is clear.

By the way, almost all functions from ntoskrnl are prefixed. By this prefix you can determine to which system major executive component the function belongs to.

To denote internal functions used a variation of the prefix - either the first letter of the prefix followed by an i (for internal) or the full prefix followed by a p (for private) or f (fastcall). For example, Ke represents kernel functions, Psp refers to internal process support functions, Mm represents the Memory Manager functions and so on.

The first parameter to Ke386IoSetAccessProcess function is a pointer to the process object, which is KPROCESS structure (defined in \include\w2k\w2kundoc.inc. I have specially prefixed file name with "w2k", since undocumented structures differ across Windows NT versions. So, to use this include file in the driver intended for XP is not the good idea). Ke386IoSetAccessProcess sets the IopmOffset member of KPROCESS structure to the appropriate value.

3.5.3 Reading info from the registry

Since we have to call Ke386IoSetAccessProcess, we need the pointer to the process object. It can be obtained by many different ways. I have chosen the most simple - using the process identifier. For this reason in the DateTime.exe we get the current process identifier and we put it into the registry. In this case we use the registry for passing parameter between the user-mode code and the kernel-mode device driver. Since the DriverEntry routine runs in the System process context, there is no way to find out, what exactly process actually has started the driver.

The second parameter - pusRegistryPath - to the DriverEntry routine is a pointer to the driver registry subkey. And we use it to get process identifier from the registry.

Now let's see how all that works.

```
    lea ecx, oa
    InitializeObjectAttributes ecx, pusRegistryPath, 0, NULL, NULL
```

We have to initialize OBJECT_ATTRIBUTES structure (\include\w2k\ntdef.inc) before we can call ZwOpenKey. I've used InitializeObjectAttributes macro for this, but you'd better do it manually since InitializeObjectAttributes macro is not always behaves as expected. And you can do it like this:

```
lea ecx, oa
xor eax, eax
assume ecx:ptr OBJECT_ATTRIBUTES
mov [ecx].dwLength, sizeof OBJECT_ATTRIBUTES
mov [ecx].RootDirectory, eax                    ; NULL
push pusRegistryPath
pop [ecx].ObjectName
mov [ecx].Attributes, eax                       ; 0
mov [ecx].SecurityDescriptor, eax               ; NULL
mov [ecx].SecurityQualityOfService, eax         ; NULL
assume ecx:nothing
```

ZwOpenKey returns registry key handle in hKey. Second parameter specifies the access rights required to the key. And you should remember that ecx register contains the pointer to the initialized object attributes of the key being opened.

```
    invoke ZwOpenKey, addr hKey, KEY_READ, ecx
    .if eax == STATUS_SUCCESS

        push eax
        invoke ZwQueryValueKey, hKey, $CCOUNTED_UNICODE_STRING("ProcessId", 4), \
                                KeyValuePartialInformation, addr kvpi, sizeof kvpi, esp
        pop ecx
```

ZwQueryValueKey returns the entry value for an open registry key. And we use it to get process identifier from the registry. Second parameter points to the name of the value entry for which the data is requested. I've used $CCOUNTED_UNICODE_STRING macro to define UNICODE_STRING structure (4 bites aligned) and the unicode-string itself. If you don't like macros, you can use the common way:

```
usz dw 'U', 'n', 'i', 'c', 'o', 'd', 'e', ' ', 's', 't', 'r', 'i', 'n', 'g', 0
us UNICODE_STRING {sizeof usz - 2, sizeof usz, offset usz}
```

But I never liked this way, so I wrote the following macros: COUNTED_UNICODE_STRING, $COUNTED_UNICODE_STRING, CCOUNTED_UNICODE_STRING, $CCOUNTED_UNICODE_STRING (\Macros\Strings.mac).

Third parameter specifies the type of information requested. KeyValuePartialInformation is a symbolic constant (defined in \include\w2k\ntddk.inc). The fourth and fifth parameters are the pointer to KEY_VALUE_PARTIAL_INFORMATION structure and its size respectively. In the Data member of this structure we'll get our process identifier. The last parameter is the pointer to the number of bytes returned. We should also reserve the place for it on the stack before calling ZwQueryValueKey.

3.5.4 Give user-mode process access to the I/O ports

```
        .if ( eax != STATUS_OBJECT_NAME_NOT_FOUND ) && ( ecx != 0 )
            invoke MmAllocateNonCachedMemory, IOPM_SIZE
            .if eax != NULL
                mov pIopm, eax
```

If ZwQueryValueKey successfully returns, we allocate a virtual address range of noncached and cpu cache-aligned memory for IOPM by calling the MmAllocateNonCachedMemory.

```
                lea ecx, kvpi
                invoke PsLookupProcessByProcessId, \
                       dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR [ecx]).Data, addr pProcess
                .if eax == STATUS_SUCCESS
                    invoke Ke386QueryIoAccessMap, 0, pIopm
```

By passing the process identifier to the PsLookupProcessByProcessId we get in pProcess the pointer to our process object. Ke386QueryIoAccessMap copies IOPM in the memory buffer.

```
                    .if al != 0

                        mov ecx, pIopm
                        add ecx, 70h / 8
                        mov eax, [ecx]
                        btr eax, 70h MOD 8
                        mov [ecx], eax

                        mov ecx, pIopm
                        add ecx, 71h / 8
                        mov eax, [ecx]
                        btr eax, 71h MOD 8
                        mov [ecx], eax

                        invoke Ke386SetIoAccessMap, 1, pIopm
                        .if al != 0
                            invoke Ke386IoSetAccessProcess, pProcess, 1
                            .if al != 0
                            .else
                                mov status, STATUS_IO_PRIVILEGE_FAILED
                            .endif
                        .else
```

```
                        mov status, STATUS_IO_PRIVILEGE_FAILED
                    .endif
                .else
                    mov status, STATUS_IO_PRIVILEGE_FAILED
                .endif
```

Now we'll clear the bits corresponding to 70h and 71h I/O ports, write modified IOPM back and call Ke386IoSetAccessProcess to allow I/O access.

```
                invoke ObDereferenceObject, pProcess
            .else
                mov status, STATUS_OBJECT_TYPE_MISMATCH
            .endif
```

The previous call to PsLookupProcessByProcessId had incremented a reference count for the process object. The Object Manager increments a reference count for an object each time it gives out a pointer to it; when kernel-mode components have finished using the pointer, they call the Object Manager to decrement the object's reference count. The system also increments the reference count when it increments the handle count (gives someone a handle to the object), and likewise decrements the reference count when the handle count decrements (someone closes some handle), because a handle is also a reference to the object that must be tracked. So even after an object's open handle counter reaches 0, the object's reference count might remain positive, indicating that the operating system is still using the object. Sooner or later the reference count also drops to 0. When this happens the Object Manager deletes the object from the memory.

We decrement the process object's reference count by calling ObDereferenceObject. And it returns to its previous state.

```
            invoke MmFreeNonCachedMemory, pIopm, IOPM_SIZE
        .else
            invoke DbgPrint, $CTA0("giveio: Call to MmAllocateNonCachedMemory failed")
            mov status, STATUS_INSUFFICIENT_RESOURCES
        .endif
    .endif
    invoke ZwClose, hKey
.endif
```

Calling MmFreeNonCachedMemory we release memory buffer, and close registry handle by calling ZwClose.

The work is done - the driver is not needed any more. Since it returns an error code, the system removes it from the memory. But now, the user-mode process has direct access to two I/O ports.

In this example we have accessed the CMOS memory just for instance. We could beep with the system speaker, as in the previous driver beeper.sys. I leave it to you. But remember you must not use the privileged instructions like cli and sti. And you must not call functions from the hal.dll also, since they are in the kernel-mode address space. The only thing you can do is to give yourselves an access to all 65535 I/O ports using this code-snippet:

```
invoke MmAllocateNonCachedMemory, IOPM_SIZE
.if eax != NULL
    mov pIopm, eax
    invoke RtlZeroMemory, pIopm, IOPM_SIZE
    lea ecx, kvpi
    invoke PsLookupProcessByProcessId, \
            dword ptr (KEY_VALUE_PARTIAL_INFORMATION PTR [ecx]).Data, addr pProcess
    .if eax == STATUS_SUCCESS
        invoke Ke386SetIoAccessMap, 1, pIopm
        .if al != 0
            invoke Ke386IoSetAccessProcess, pProcess, 1
        .endif
        invoke ObDereferenceObject, pProcess
    .endif
    invoke MmFreeNonCachedMemory, pIopm, IOPM_SIZE
.else
    mov status, STATUS_INSUFFICIENT_RESOURCES
.endif
```

Always keep in mind that playing the system speaker and reading the CMOS memory is harmless enough. But accessing some other I/O ports can be potentially dangerous, basically since you can't synchronize it in the user-mode.

3.6 A couple words about driver debugging

Now we can talk about the driver debugging in more details. As I've already mentioned, you should better use SoftICE as a debugger.

To force a breakpoint in the driver's code we have to execute a CPU breakpoint instruction. You achieve it by placing an "int 3" somewhere in your driver's code. The "int 3" raises an exception that is handled by the kernel debugger like SoftICE. But before you use it make sure you have enabled INT 3 handling. Use the I3HERE command to specify that any interrupt 3 instruction pops up SoftICE. Check *SoftICE Command Reference* for details. And always bear in mind that not handled breakpoint exception in kernel will cause a bug check resulting in the BSOD! So, don't forget to type "i3here on" before starting the driver. In latter SoftICE versions the int 3 handling is set by default for the kernel-mode addresses.

I repeatedly called the DbgPrint function in the giveio driver's code. This function causes a string to be printed onto the debugger command window. SoftICE is perfectly understands it. You can also use DebugView by Mark Russinovich (www.sysinternals.com) to monitor debug output.
