Aufgabe 1

Fragen:

a) Bei der Portierung des Treibers machte die Berechnung der IOCTL Codes Probleme.

b) Man könnte die Funktionalität des Treibers in ein eigenes Modul packen und in beiden Treibern verwenden. Im Python Programm könnte man von der HWDevice Klasse die Grundfunktionalitäten in der Basisklasse lassen und die Funktionen die sich auf ein bestimmtes Betriebssystem beziehen in einer abgeleiteten Klasse implementieren.

Testausgabe:

Python:

```
14:80:0
Sleep for 4 seconds
14:80:4
```

Treiber:

```
[ 8489.706113] ioctl was called num: -2147195904, param: 0
[ 8489.706115] read databit was called
[ 8489.706120] read DataBit: 0
[ 8489.706133] ioctl was called num: -2147195901, param: 3215241996
[ 8489.706136] write clockbit was called
[ 8489.706138] clockbit to write 1
[ 8489.706157] ioctl was called num: -2147195902, param: 3215241996
[ 8489.706160] write databit was called
[ 8489.706162] dataBit to write 0
```

Sourcecode:

testdriver.py:

```
# Test file for Wdm1


import sys, serial, array
from time import sleep
my_platform = "";

if sys.platform == "win32":
    my_platform = "win32"
    import win32file, win32api
    sys.path += ["DeviceDriverAccess/Release"]
    from DeviceDriverAccess import GetDeviceViaInterface

    # Constants for Wdm1
    WDM1_GUID = pack("LHHBBBBBBBB", 0x1ef8a96b, 0x6c26, 0x42a4, 0xb9, 0x19, 0x82, 0x50,
0x93, 0x13, 0xbc, 0x5b)

    def CTL_CODE(DeviceType, Function, Method, Access):
        return (DeviceType << 16) | (Access << 14) | (Function << 2) | Method

elif sys.platform == "linux2":
    my_platform = "linux2"

    from fcntl import ioctl

    IOCTL_READ_DATABIT = -2147195904
    IOCTL_READ_CLOCKBIT = -2147195903
    IOCTL_WRITE_DATABIT = -2147195902
    IOCTL_WRITE_CLOCKBIT = -2147195901


import time
from struct import *

FILE_DEVICE_UNKNOWN = 0x00000022
METHOD_BUFFERED = 0
METHOD_IN_DIRECT = 1
METHOD_OUT_DIRECT = 2
METHOD_NEITHER = 3
FILE_ANY_ACCESS = 0

ZERO_BUFFER = 0x801
REMOVE_BUFFER = 0x802
GET_BUFFER_SIZE = 0x803
GET_BUFFER = 0x804
UNRECOGNISED = 0x805
GET_BUILDTIME = 0x806
READ_DATABIT = 0x807
READ_CLOCKBIT = 0x808
WRITE_DATABIT = 0x809
WRITE_CLOCKBIT = 0x810

RTC_BASEADDRESS = 0x51

DEVICE_FILENAME = "/home/user/TPG2/char_dev"
```

```
class HWDevice:
    def __init__(self):
        if my_platform == "win32":
            self.guid = WDM1_GUID
        self.drvHnd = None
        self.fd = -1
        self.OpenDrv()
        self.data = 0
        self.clock = 0

    def OpenDrv(self):
        """
        Open a handle to the device driver. If the driver is already open,
        close it first an reopen it.
        """
        self.CloseDrv()

        # Windows Open --------
        if my_platform == "win32":
            try:
                name = GetDeviceViaInterface(self.guid)
            except:
                raise IOError (1, "Wdm1 Device not found")

            desiredAccess = win32file.GENERIC_READ | win32file.GENERIC_WRITE
            self.drvHnd = win32file.CreateFile(name,
                                               desiredAccess,
                                               win32file.FILE_SHARE_WRITE,
                                               None,
                                               win32file.OPEN_EXISTING,
                                               0,
                                               0)
        # Linux open ---------
        elif my_platform == "linux2":
           self.fd = open(DEVICE_FILENAME, "r+")
           print self.fd
           if (self.fd < 0):
               print "cant open device"
               return



    def CloseDrv(self):
        """
        Close the handle to device driver
        """
        if my_platform == "win32" and self.drvHnd is not None:
            win32file.CloseHandle(self.drvHnd)
            self.drvHnd = None
        elif my_platform == "linux2" and self.fd >= 0:
            self.fd.close()

    def Write(self, string):
        win32file.WriteFile(self.drvHnd, string, None)

    def Read(self, numofbytes=1):
        hr, result = win32file.ReadFile(self.drvHnd, numofbytes, None)
        return result

    def SetFilePointer(self, distance):
        win32file.SetFilePointer(self.drvHnd, distance, win32file.FILE_BEGIN)
```

```
    def DeviceIoControl(self, function, input):

        if my_platform == "win32":
            IOCTL_USB_GET_DEVICE_DESCRIPTOR = CTL_CODE(FILE_DEVICE_UNKNOWN, function,
METHOD_BUFFERED, FILE_ANY_ACCESS)

            try:
                result = win32file.DeviceIoControl(self.drvHnd,
IOCTL_USB_GET_DEVICE_DESCRIPTOR, input, 512)
            except win32file.error, e:
                print "problem with driver or stack over/underflow"
                print "Unexpected error:", e
                result = 0
        elif my_platform == "linux2":
            if function == READ_DATABIT:
                result = ioctl(d.fd, IOCTL_READ_DATABIT, 0);
            elif function == READ_CLOCKBIT:
                result = ioctl(d.fd, IOCTL_READ_CLOCKBIT, 0);
            elif function == WRITE_DATABIT:
                result = ioctl(d.fd, IOCTL_WRITE_DATABIT, input);
            elif function == WRITE_CLOCKBIT:
                result = ioctl(d.fd, IOCTL_WRITE_CLOCKBIT, input);

            #print "set msg"
            #print ioctl(d.fd, IOCTL_WRITE_DATABIT, pack("b",1));
            #print fcntl.fcntl(d.fd, IOCTL_SET_MSG, "test msg")

            #print "get msg"
            #buf = array.array('h', [0])
            #print fcntl.fcntl(d.fd, IOCTL_GET_MSG)

            #result = 0
            #self.fd.write("asdfasf TestString ...")
            #result = self.fd.read()

        return result


    def ReadI2C(self, adress, register, numOfBytes=1):
        self.SendStartI2C()

        ack = self.WriteByteI2C(adress << 1)
        #print "ReadI2C adress ack: %d" % ack

        ack = self.WriteByteI2C(register)
        #print "ReadI2C register ack: %d" % ack

        self.SendStartI2C()

        ack = self.WriteByteI2C((adress << 1) + 1)
        #print "ReadI2C adress 2 ack: %d" % ack

        byteList = []
        for x in range(0, numOfBytes):
            ack = 0

            if x == numOfBytes-1:
                ack = 1
            else:
                ack = 0

            byteList.append(self.ReadByteI2C(ack));
```

```
            #print "ReadI2C value ack: %d" % ack

        self.SendStopI2C()

        return byteList

    def WriteI2C(self, adress, register, data):
        self.SendStartI2C()

        ack = self.WriteByteI2C(adress << 1)
        #print "WriteI2C adress ack: %d" % ack

        ack = self.WriteByteI2C(register)
        #print "WriteI2C register ack: %d" % ack

        for dataByte in data:
            ack = self.WriteByteI2C(dataByte)
            #print "WriteI2C dataByte ack: %d" % ack

        self.SendStopI2C()

    def ReadByteI2C(self, ack):
        bit7 = self.ReadBitI2C()
        bit6 = self.ReadBitI2C()
        bit5 = self.ReadBitI2C()
        bit4 = self.ReadBitI2C()
        bit3 = self.ReadBitI2C()
        bit2 = self.ReadBitI2C()
        bit1 = self.ReadBitI2C()
        bit0 = self.ReadBitI2C()

        byte = (bit7 << 7) + (bit6 << 6) + (bit5 << 5) + (bit4 << 4) + (bit3 << 3) +
(bit2 << 2) + (bit1 << 1) + bit0

        #send ack
        self.WriteBitI2C(ack)

        return byte

    def WriteByteI2C(self, byte):
        self.WriteBitI2C(byte&128 != 0)
        self.WriteBitI2C(byte&64 != 0)
        self.WriteBitI2C(byte&32 != 0)
        self.WriteBitI2C(byte&16 != 0)
        self.WriteBitI2C(byte&8 != 0)
        self.WriteBitI2C(byte&4 != 0)
        self.WriteBitI2C(byte&2 != 0)
        self.WriteBitI2C(byte&1 != 0)

        #send ack
        ack = self.ReadBitI2C()

        return ack

    def ReadBitI2C(self):
        self.WriteClock(True)
        bit = self.ReadData()
        self.WriteClock(False)
        return bit
```

```
def WriteBitI2C(self, value):
    bit = self.WriteData(value)
    self.WriteClock(True)
    self.WriteClock(False)

def SendStartI2C(self):
    self.WriteData(True)

    self.WriteClock(True)
    self.WriteData(False)
    self.WriteClock(False)
    #print "-----"

def SendStopI2C(self):
    #print "-----"
    self.WriteData(False)

    self.WriteClock(True)
    self.WriteData(True)
    #self.WriteClock(False)

def ReadClock(self):
    clockBit = self.DeviceIoControl(READ_CLOCKBIT ,"")
    result = unpack("b",clockBit)[0]
    return abs(result-1)
    #print "Read clock: %d" % self.clock
    #return self.clock

def WriteClock(self, isOn):
    if isOn == False:
        self.DeviceIoControl(WRITE_CLOCKBIT ,pack("b",1))
        #self.clock = 1
    else:
        self.DeviceIoControl(WRITE_CLOCKBIT ,pack("b",0))
        #self.clock = 0

    #print "Write clock %d" % self.clock

def ReadData(self):
    dataBit = self.DeviceIoControl(READ_DATABIT ,"")
    #result = unpack("b",dataBit)[0]
    return abs(dataBit-1)
    #print "Read data: %d" % self.data
    #return self.data

def WriteData(self, isOn):
    if isOn == False:
        self.DeviceIoControl(WRITE_DATABIT ,pack("b",1))
        #self.data = 1
    else:
        self.DeviceIoControl(WRITE_DATABIT ,pack("b",0))
        #self.data = 0

    #print "Write data %d" % self.data
```

```
    def GetTime(self):
        #return "%d:%d:%d" % d.ReadI2C(RTC_BASEADDRESS, 4)[0].decode("utf-8"),
d.ReadI2C(RTC_BASEADDRESS, 3)[0].decode("utf-8"), d.ReadI2C(RTC_BASEADDRESS,
2)[0].decode("utf-8")
        string = "%d:%d:%d" % (d.ReadI2C(RTC_BASEADDRESS, 4)[0],
d.ReadI2C(RTC_BASEADDRESS, 3)[0], d.ReadI2C(RTC_BASEADDRESS, 2)[0])
        return string


d = HWDevice()

ser = serial.Serial(0)

#for i in range(1,100):
   # print d.DeviceIoControl(READ_DATABIT, 0)

   # print "write data 0"
 #   d.DeviceIoControl(WRITE_DATABIT ,pack("b",1))
  #  print d.DeviceIoControl(WRITE_CLOCKBIT ,pack("b",1))
   # sleep(1)
    #print d.DeviceIoControl(READ_CLOCKBIT, 0)

    #print d.DeviceIoControl(WRITE_DATABIT ,pack("b",0))
    #print "write clock 1"
    #d.DeviceIoControl(WRITE_CLOCKBIT ,pack("b",0))
    #sleep(1)

#print d.DeviceIoControl(READ_DATABIT, 0)


print d.GetTime()
print("Sleep for 4 seconds")
time.sleep(4);
print d.GetTime()


ser.close()
d.CloseDrv()
```

chardev.h:

```
#ifndef CHARDEV_H
#define CHARDEV_H

#include <linux/ioctl.h>

/*
 * The major device number. We can't rely on dynamic
 * registration any more, because ioctls need to know
 * it.
 */
#define MAJOR_NUM 100

/*
 * Set the message of the device driver
 */
#define IOCTL_WDM1_READ_DATABIT _IOR(MAJOR_NUM, 0, char *)

#define IOCTL_WDM1_READ_CLOCKBIT _IOR(MAJOR_NUM, 1, char *)

#define IOCTL_WDM1_WRITE_DATABIT _IOR(MAJOR_NUM, 2, char *)

#define IOCTL_WDM1_WRITE_CLOCKBIT _IOR(MAJOR_NUM, 3, char *)


#define DEVICE_FILE_NAME "char_dev"

#endif
```

chardev.c:

```
#include <linux/kernel.h>  /* We're doing kernel work */
#include <linux/module.h>  /* Specifically, a module */
#include <linux/fs.h>
#include <asm/uaccess.h>   /* for get_user and put_user */

#include "chardev.h"
#define SUCCESS 0
#define FAILURE -1
#define DEVICE_NAME "char_dev"
#define BUF_LEN 80


#define SER_MCR(x) ((x)+4)
#define SR_MCR_DTR 0x01
#define SR_MCR_RTS 0x02
#define SER_MSR(x) ((x)+6)
#define SR_MSR_CTS 0x10
#define SR_MSR_DSR 0x20

unsigned COM1_BASEADRESS = 0x3F8;


static int Device_Open = 0;

static char Message[BUF_LEN];

static char *Message_Ptr;
```

```
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
       printk(KERN_INFO "device_open(%p)\n", file);
#endif

       /*
        * We don't want to talk to two processes at the same time
        */
       if (Device_Open)
              return -EBUSY;

       Device_Open++;
       /*
        * Initialize the message
        */
       Message_Ptr = Message;
       try_module_get(THIS_MODULE);
       return SUCCESS;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
       printk(KERN_INFO "device_release(%p,%p)\n", inode, file);
#endif

       /*
        * We're now ready for our next caller
        */
       Device_Open--;

       module_put(THIS_MODULE);
       return SUCCESS;
}

static ssize_t device_read(struct file *file,   /* see include/linux/fs.h   */
                           char __user * buffer,   /* buffer to be
                                                    * filled with data */
                           size_t length,    /* length of the buffer     */
                           loff_t * offset)
{
       /*
        * Number of bytes actually written to the buffer
        */
       int bytes_read = 0;

#ifdef DEBUG
       printk(KERN_INFO "device_read(%p,%p,%d)\n", file, buffer, length);
#endif

       /*
        * If we're at the end of the message, return 0
        * (which signifies end of file)
        */
       if (*Message_Ptr == 0)
              return 0;
```

```
	while (length && *Message_Ptr) {

		/*
		 * Because the buffer is in the user data segment,
		 * not the kernel data segment, assignment wouldn't
		 * work. Instead, we have to use put_user which
		 * copies data from the kernel data segment to the
		 * user data segment.
		 */
		put_user(*(Message_Ptr++), buffer++);
		length--;
		bytes_read++;
	}

#ifdef DEBUG
	printk(KERN_INFO "Read %d bytes, %d left\n", bytes_read, length);
#endif

	/*
	 * Read functions are supposed to return the number
	 * of bytes actually inserted into the buffer
	 */
	return bytes_read;
}

/*
 * This function is called when somebody tries to
 * write into our device file.
 */
static ssize_t
device_write(struct file *file,
	     const char __user * buffer, size_t length, loff_t * offset)
{
	int i;

#ifdef DEBUG
	printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, length);
#endif

	for (i = 0; i < length && i < BUF_LEN; i++)
		get_user(Message[i], buffer + i);

	Message_Ptr = Message;

	/*
	 * Again, return the number of input characters used
	 */
	return i;
}

long device_ioctl(   /* removed inode */
		  struct file *file,  /* ditto */
		  unsigned int ioctl_num,   /* number and param for ioctl */
		  unsigned long ioctl_param)
{
	//int i;
	//char *temp;
	//char ch;

    unsigned char dataBit = 0;
    unsigned char clockBit = 0;
    unsigned char mcr_byte = 0;
    char* clockBitToWrite = 0;
    char* dataBitToWrite = 0;
```

```
printk(KERN_INFO "ioctl was called num: %d, param: %lu\n", ioctl_num, ioctl_param);

   /*
    * Switch according to the ioctl called
    */
   switch (ioctl_num) {
   case IOCTL_WDM1_READ_DATABIT:
    printk(KERN_INFO "read databit was called\n");

    dataBit = inb(SER_MSR(COM1_BASEADRESS));
          if (dataBit & 0x10)
          {
                dataBit = 1;
          }
          else
          {
                dataBit = 0;
          }
    printk(KERN_INFO "read DataBit: %d\n", dataBit);
          return dataBit;
          break;

   case IOCTL_WDM1_READ_CLOCKBIT:
    printk(KERN_INFO "read clockbit was called\n");

          clockBit = inb(SER_MSR(COM1_BASEADRESS));
          if (clockBit & 0x20)
          {
                clockBit = 1;
          }
          else
          {
                clockBit = 0;
          }
          printk(KERN_INFO "read clockBit: %d\n", clockBit);
    return clockBit;
          break;

   case IOCTL_WDM1_WRITE_DATABIT:
       printk(KERN_INFO "write databit was called\n");

          if (ioctl_param != 0){
                dataBitToWrite = (char*)ioctl_param;
                printk(KERN_INFO "dataBit to write %d\n", dataBitToWrite[0]);
                mcr_byte = inb(SER_MCR(COM1_BASEADRESS));
        printk(KERN_INFO "mcr_byte: %x",mcr_byte);
        if (dataBitToWrite[0])
                {
                      mcr_byte = mcr_byte | 0x01;
                }
                else
                {
                      mcr_byte = mcr_byte & 0xFE;
                }
                outb(mcr_byte, SER_MCR(COM1_BASEADRESS));
          }
          else
          {
                printk(KERN_INFO "cant cast ioctl_param\n");
                return FAILURE;
          }
          break;
```

```
    case IOCTL_WDM1_WRITE_CLOCKBIT:
              printk(KERN_INFO "write clockbit was called\n");

              if (ioctl_param != 0){
                     clockBitToWrite = (char*)ioctl_param;
                     printk(KERN_INFO "clockbit to write %d\n", clockBitToWrite[0]);
            mcr_byte = inb(SER_MCR(COM1_BASEADRESS));
                     if (clockBitToWrite[0])
                     {
                            mcr_byte = mcr_byte | 0x02;
                     }
                     else
                     {
                            mcr_byte = mcr_byte & 0xFD;
                     }
                     outb(mcr_byte, SER_MCR(COM1_BASEADRESS));
              }
              else
              {
                     printk(KERN_INFO "cant cast ioctl_param\n");
                     return FAILURE;
              }
         break;
       }
       return SUCCESS;
}

struct file_operations Fops = {
       .read = device_read,
       .write = device_write,
       .unlocked_ioctl = device_ioctl,
       .open = device_open,
       .release = device_release, /* a.k.a. close */
};

int init_module()
{
       int ret_val;
       /*
        * Register the character device (atleast try)
        */
       ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

       /*
        * Negative values signify an error
        */
       if (ret_val < 0) {
              printk(KERN_ALERT "%s failed with %d\n",
                     "Sorry, registering the character device ", ret_val);
              return ret_val;
       }

       printk(KERN_INFO "%s The major device number is %d.\n",
              "Registeration is a success", MAJOR_NUM);
       printk(KERN_INFO "If you want to talk to the device driver,\n");
       printk(KERN_INFO "you'll have to create a device file. \n");
       printk(KERN_INFO "We suggest you use:\n");
       printk(KERN_INFO "mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
       printk(KERN_INFO "The device file name is important, because\n");
       printk(KERN_INFO "the ioctl program assumes that's the\n");
       printk(KERN_INFO "file you'll use.\n");
```

```
    printk(KERN_INFO "IOCTL_WDM1_READ_DATABIT %d.\n", IOCTL_WDM1_READ_DATABIT);
    printk(KERN_INFO "IOCTL_WDM1_READ_CLOCKBIT %d.\n", IOCTL_WDM1_READ_CLOCKBIT);
    printk(KERN_INFO "IOCTL_WDM1_WRITE_DATABIT %d.\n", IOCTL_WDM1_WRITE_DATABIT);
    printk(KERN_INFO "IOCTL_WDM1_WRITE_CLOCKBIT %d.\n", IOCTL_WDM1_WRITE_CLOCKBIT);

       return 0;
}

/*
 * Cleanup - unregister the appropriate file from /proc
 */
void cleanup_module()
{
  //int ret;

       /*
        * Unregister the device
        */
       unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

       /*
        * If there's an error, report it
        */
       //if (ret < 0)
       //     printk(KERN_ALERT "Error: unregister_chrdev: %d\n", ret);
}
```