Aufgabe 1

Testausgabe:

```
Python 2.5.4 (r254:67916, Dec 23 2008, 15:10:54) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ## working on region in file c:/Users/Treiber/AppData/Local/Temp/python-3908go2.py...
May 19 2015 14:08:29^@
ReadI2C adress ack: 0
ReadI2C register ack: 0
ReadI2C adress 2 ack: 0
ReadI2C value ack: 1
('Seconds: ', 72)
ReadI2C adress ack: 0
ReadI2C register ack: 0
ReadI2C adress 2 ack: 0
ReadI2C value ack: 1
('Seconds: ', 5)
Sleep for 4 seconds
ReadI2C adress ack: 0
ReadI2C register ack: 0
ReadI2C adress 2 ack: 0
ReadI2C value ack: 1
('Seconds: ', 9)
```

Sourcecode:

wdm1-test.py:

```
# Test file for Wdm1

import win32file, win32api, sys, serial
sys.path += ["DeviceDriverAccess/Release"]

import time

from DeviceDriverAccess import GetDeviceViaInterface

from struct import *

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

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
```

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

class HWDevice:
    def __init__(self,guid):
        self.guid = guid
        self.drvHnd = None
        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()
        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)

    def CloseDrv(self):
        """
        Close the handle to device driver
        """
        if self.drvHnd is not None:
            win32file.CloseHandle(self.drvHnd)
            self.drvHnd = None

    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):

        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

        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(result-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

d = HWDevice(WDM1_GUID)

dateTime = d.DeviceIoControl(GET_BUILDTIME,"")
print dateTime

ser = serial.Serial(0)

print ("Seconds: ", d.ReadI2C(RTC_BASEADDRESS, 2)[0])

d.WriteI2C(RTC_BASEADDRESS, 2, [0x5])

print("Seconds: ", d.ReadI2C(RTC_BASEADDRESS, 2)[0])
print("Sleep for 4 seconds")
time.sleep(4)
print("Seconds: ", d.ReadI2C(RTC_BASEADDRESS, 2)[0])

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

Dispatch.cpp:

```
...
PUCHAR COM1_BASEADRESS = ((PUCHAR)(ULONG_PTR)0x3F8);
...

NTSTATUS Wdm1DeviceControl(IN PDEVICE_OBJECT fdo,
	IN PIRP Irp)
{
	PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);
	NTSTATUS status = STATUS_SUCCESS;
	ULONG BytesTxd = 0;

	ULONG ControlCode = IrpStack->Parameters.DeviceIoControl.IoControlCode;
	ULONG InputLength = IrpStack->Parameters.DeviceIoControl.InputBufferLength;
	ULONG OutputLength = IrpStack->Parameters.DeviceIoControl.OutputBufferLength;

	DebugPrint("DeviceIoControl: Control code %x InputLength %d OutputLength %d",
		ControlCode, InputLength, OutputLength);

	// Get access to the shared buffer
	KIRQL irql;
	KeAcquireSpinLock(&BufferLock, &irql);
	switch (ControlCode)
	{
		///////	Zero Buffer
	case IOCTL_WDM1_ZERO_BUFFER:
		// Zero the buffer
		if (Buffer != NULL && BufferSize > 0)
			RtlZeroMemory(Buffer, BufferSize);
		break;

		///////	Remove Buffer
	case IOCTL_WDM1_REMOVE_BUFFER:
		if (Buffer != NULL)
		{
			ExFreePool(Buffer);
			Buffer = NULL;
			BufferSize = 0;
		}
		break;

		///////	Get Buffer Size as ULONG
	case IOCTL_WDM1_GET_BUFFER_SIZE:
		if (OutputLength < sizeof(ULONG))
			status = STATUS_INVALID_PARAMETER;
		else
		{
			BytesTxd = sizeof(ULONG);
			RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer, &BufferSize,
sizeof(ULONG));
		}
		break;

		///////	Get Buffer
	case IOCTL_WDM1_GET_BUFFER:
		if (OutputLength > BufferSize)
			status = STATUS_INVALID_PARAMETER;
		else
		{
			BytesTxd = OutputLength;
			RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer, Buffer, BytesTxd);
		}
		break;
```

```
		///////		Get DateTime
	case IOCTL_WDM1_GET_BUILDTIME:
	{
		if (OutputLength < dateTimeSize){
			status = STATUS_INVALID_PARAMETER;
		}
		else {
			memset(dateTimeBuffer, 0, dateTimeSize);
			strcpy(dateTimeBuffer, __DATE__);
			strcat(dateTimeBuffer, " ");
			strcat(dateTimeBuffer, __TIME__);
			DebugPrint("DateTime: %s", dateTimeBuffer);
			BytesTxd = dateTimeSize;
			RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer, dateTimeBuffer,
dateTimeSize);
		}
	}
	break;

	case IOCTL_WDM1_READ_DATABIT:
	{
		if (OutputLength < 1)
		{
			status = STATUS_INVALID_PARAMETER;
		}
		else
		{
			UCHAR dataBit = READ_PORT_UCHAR (SER_MSR(COM1_BASEADRESS));

			if (dataBit & 0x10)
			{
				dataBit = 1;
			}
			else
			{
				dataBit = 0;
			}

			BytesTxd = 1;
			RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer, &dataBit, BytesTxd);

		}


	}
	break;

	case IOCTL_WDM1_READ_CLOCKBIT:
	{
		if (OutputLength < 1)
		{
			status = STATUS_INVALID_PARAMETER;
		}
		else
		{
			UCHAR clockBit = READ_PORT_UCHAR (SER_MSR(COM1_BASEADRESS));

			if (clockBit & 0x20)
			{
				clockBit = 1;
			}
```

```
                else
                {
                        clockBit = 0;
                }

                BytesTxd = 1;
                RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer, &clockBit, BytesTxd);

        }
}
break;

case IOCTL_WDM1_WRITE_DATABIT:
{
        if (InputLength < 1)
        {
                status = STATUS_INVALID_PARAMETER;
        }
        else
        {
                UCHAR dataBit = 0;
                RtlCopyMemory(&dataBit, Irp->AssociatedIrp.SystemBuffer, 1);
                BytesTxd = 1;

                UCHAR byte = READ_PORT_UCHAR (SER_MCR(COM1_BASEADRESS));
                if (dataBit)
                {
                        byte = byte | 0x01;
                }
                else
                {
                        byte = byte & 0xFE;
                }

                WRITE_PORT_UCHAR (SER_MCR(COM1_BASEADRESS), byte);
        }
}
break;

case IOCTL_WDM1_WRITE_CLOCKBIT:
{
        if (InputLength < 1)
        {
                status = STATUS_INVALID_PARAMETER;
        }
        else
        {
                UCHAR clockBit = 0;
                RtlCopyMemory(&clockBit, Irp->AssociatedIrp.SystemBuffer, 1);
                BytesTxd = 1;

                UCHAR byte = READ_PORT_UCHAR (SER_MCR(COM1_BASEADRESS));
                if (clockBit)
                {
                        byte = byte | 0x02;
                }
                else
                {
                        byte = byte & 0xFD;
                }

                WRITE_PORT_UCHAR (SER_MCR(COM1_BASEADRESS), byte);
        }
}
break;
```

```
		///////		Invalid request
	default:
		status = STATUS_INVALID_DEVICE_REQUEST;
	}
	// Release shared buffer
	KeReleaseSpinLock(&BufferLock, irql);

	DebugPrint("DeviceIoControl: %d bytes written", (int)BytesTxd);

	// Complete IRP
	return CompleteIrp(Irp, status, BytesTxd);
}
```

Dispatch.cpp:

```
...

#define IOCTL_WDM1_READ_DATABIT CTL_CODE(	\
                    FILE_DEVICE_UNKNOWN,		\
                    0x807,                                  \
                    METHOD_BUFFERED,                        \
                    FILE_ANY_ACCESS)

#define IOCTL_WDM1_READ_CLOCKBIT CTL_CODE(	\
                    FILE_DEVICE_UNKNOWN,		\
                    0x808,                                  \
                    METHOD_BUFFERED,                        \
                    FILE_ANY_ACCESS)

#define IOCTL_WDM1_WRITE_DATABIT CTL_CODE(	\
                    FILE_DEVICE_UNKNOWN,		\
                    0x809,                                  \
                    METHOD_BUFFERED,                        \
                    FILE_ANY_ACCESS)

#define IOCTL_WDM1_WRITE_CLOCKBIT CTL_CODE(	\
                    FILE_DEVICE_UNKNOWN,		\
                    0x810,                                  \
                    METHOD_BUFFERED,                        \
                    FILE_ANY_ACCESS)
```