**1.Aufgabe**

**Bedeutung der einzelnen Stream Device Funktionen:**

**XXX\_Init** initialisiert ein Device und allokiert die nötigen globalen Ressourcen im Device driver. Dabei wird ein Pointer zu einem string, der den Registry-Pfad enthält, als Parameter übergeben. Wenn der User ein Device verwendet, wird diese Initialisierungsfunktion zuallererst vom Device Manager aufgerufen.

**XXX\_PreDeInit** markiert eine Device-Instanz als ungülitg und weckt schlafende Threads.

**XXX\_Deinit** wird vom Device Manager aufgerufen wenn ein Device entfernt wird. In dieser Funktion werden alle Ressourcen, die vom Treiber verwendet wurden, wieder freigegeben.

**XXX\_Open** öffnet ein Device zum Lesen, Schreiben oder beidem. Hier sollten außerdem die benötigten Ressourcen zum Lesen und Schreiben allokiert werden.

**XXX\_PreClose** benachrichtigt den Driver, geschlossene Handles als ungültig zu markieren und schlafende Threads zu wecken. Der Device Manager ruft zuerst diese Funktion auf und wartet mit dem Aufruf von **XXX\_Close**, bis das keine externen Threads mehr in der Treiber-DLL mehr laufen die das zu schließende Handle benutzen. Gleiches gilt für **XXX\_PreDeinit** und **XXX\_Deinit**.

**XXX\_Close** schließt ein Device, welche das Handle als Parameter erhält, das in der XXX\_Open-Funktion aufgerufen wurde.

**XXX\_Read** liest Daten vom Device aus. Diese werden in einem Buffer gespeichert, dessen Länge mindestens dem dritten Parameter Count entsprechen soll.

**XXX\_Write** schreibt Daten zum Device. Diese Daten werden mit einem Buffer übergeben und mit Count wird angegeben, wieviele Bytes vom Buffer in zum Device geschrieben werden.

**XXX\_Seek** verschiebt den Zeiger auf die Dateien in einem Device.

**XXX\_IOControl** sendet einen Befehl an das Device, um eine bestimmte Operation zu specifizieren. Die spezifische Operation wird mit dem Parameter dwCode festgelegt.

**XXX\_PowerDown** schaltet Devices ab. Diese Funktion ist nur brauchbar, wenn ein Device über Software abgeschaltet werden kann. Sie wird im Kernel Mode ausgeführt und ist deshalb nicht unterbrechbar.

**XXX\_PowerUp** stellt die Stromzufuhr wieder her und wird ebenfalls im Kernel Mode ausgeführt.

**2.Aufgabe**

**Ledswitch.h:**

#ifndef LEDSWITCH\_H\_

#define LEDSWITCH\_H\_

#include "gpio.h"

bool initLeds(GPIOREG \* pGpioReg);

void setLeds(GPIOREG \* pGpioReg, char data);

/\* ToDo: implement missing functions as defined !!! \*/

bool initSwitches(GPIOREG \* pGpioReg);

char readSwitches(GPIOREG \* pGpioReg);

bool initPushButtons(GPIOREG \* pGpioReg);

char readPushButtons(GPIOREG \* pGpioReg);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#endif

**Ledswitch.cpp:**

#include "stdafx.h"

#include <windows.h>

#include "ledswitch.h"

bool initLeds(GPIOREG \* pGpioReg){

// check if pointer is valid

if(!pGpioReg){

return false;

}

// set port direction and function

pGpioReg->gafr2\_l &= ~(1 << 31 | 1 << 30); // set alternate function 0 for GPIO79

pGpioReg->gafr1\_l &= ~(1 << 9 | 1 << 8); // set alternate function 0 for GPIO36

pGpioReg->gafr1\_l &= ~(1 << 11 | 1 << 10); // set alternate function 0 for GPIO37

pGpioReg->gafr1\_l &= ~(1 << 7 | 1 << 6); // set alternate function 0 for GPIO35

pGpioReg->gpdr1 |= (1 << 3); //set GPIO35 as output

pGpioReg->gpdr1 |= (1 << 4); //set GPIO36 as output

pGpioReg->gpdr1 |= (1 << 5); //set GPIO37 as output

pGpioReg->gpdr2 |= (1 << 15); //set GPIO79 as output

// reset leds

setLeds(pGpioReg, 0x00);

return true;

}

void setLeds(GPIOREG \* pGpioReg, char data){

// mask several leds

if(data & 0x01){

pGpioReg->gpsr1 = (1 << 3); // reset GPIO35

}else{

pGpioReg->gpcr1 = (1 << 3); // set GPIO35

}

if(data & 0x02){

pGpioReg->gpsr1 = (1 << 5); // reset GPIO37

}else{

pGpioReg->gpcr1 = (1 << 5); // set GPIO37

}

if(data & 0x04){

pGpioReg->gpsr1 = (1 << 4); // reset GPIO36

}else{

pGpioReg->gpcr1 = (1 << 4); // set GPIO36

}

if(data & 0x08){

pGpioReg->gpsr2 = (1 << 15); // reset GPIO79

}else{

pGpioReg->gpcr2 = (1 << 15); // set GPIO79

}

}

bool initPushButtons(GPIOREG \* pGpioReg){

// check if pointer is valid

if(!pGpioReg){

return false;

}

pGpioReg->gafr0\_u &= ~(1 << 19 | 1 << 18); //set alternate function 0 for GPIO25 - push button 1

pGpioReg->gafr0\_u &= ~(1 << 21 | 1 << 20); //set alternate function 0 for GPIO26 - push button 2

pGpioReg->gafr0\_u &= ~(1 << 15 | 1 << 14); //set alternate function 0 for GPIO23 - push button 3

pGpioReg->gafr0\_u &= ~(1 << 17 | 1 << 16); //set alternate function 0 for GPIO24 - push button 4

pGpioReg->gpdr0 &= ~ (1 << 25); //set GPIO25 as input

pGpioReg->gpdr0 &= ~ (1 << 26); //set GPIO26 as input

pGpioReg->gpdr0 &= ~ (1 << 23); //set GPIO23 as input

pGpioReg->gpdr0 &= ~ (1 << 24); //set GPIO24 as input

return true;

}

char readPushButtons(GPIOREG \* pGpioReg){

char ret = 0;

if(pGpioReg->gplr0 & (1 << 25)){ // read GPIO25

ret |= 0x01;

}

if(pGpioReg->gplr0 & (1 << 26)){ // read GPIO26

ret |= 0x02;

}

if(pGpioReg->gplr0 & (1 << 23)){ // read GPIO23

ret |= 0x04;

}

if(pGpioReg->gplr0 & (1 << 24)){ // read GPIO24

ret |= 0x08;

}

return ret;

}

bool initSwitches(GPIOREG \* pGpioReg) {

// check if pointer is valid

if(!pGpioReg){

return false;

}

// set the alternate functions to function 0

pGpioReg->gafr0\_l &= ~(1<<31 | 1<<30); // GPIO15

pGpioReg->gafr2\_u &= ~(1<<1 | 1<<0 ); // GPIO80

pGpioReg->gpdr0 &= ~(1<<15); // set GPIO15 as input

pGpioReg->gpdr2 &= ~(1<<16); // set GPIO80 as input

return true;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

char readSwitches(GPIOREG \* pGpioReg) {

unsigned char ret = 0;

if (pGpioReg->gplr0 & (1<<15)) // GPIO15

ret |= 0x01;

if (pGpioReg->gplr2 & (1<<16)) // GPIO80

ret |= 0x02;

if (pGpioReg->gplr1 & (1<<20)) // GPIO52

ret |= 0x04;

if (pGpioReg->gplr0 & (1<<19)) // GPIO19

ret |= 0x08;

return ret;

}

**Mapreg.h:**

#ifndef \_\_MAPREG\_H\_\_

#define \_\_MAPREG\_H\_\_

#define PAGE\_SIZE 4096

/\* ToDo: implement missing functions as defined !!! \*/

void\* MapRegister(DWORD pa);

void UnMapRegister(void\* pRegs);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#endif

**Mapreg.cpp:**

#include <windows.h>

#include <commctrl.h>

#include "MapReg.h"

#define RETAIL\_ON TRUE

extern "C"{

BOOL VirtualCopy(LPVOID lpvDest, LPVOID lpvSrc, DWORD cbSize, DWORD fdwProtect);

}

void\* MapRegister(DWORD pa)

{

// allocate memory first and map it if allocation succeeds

LPVOID addr = VirtualAlloc(0, PAGE\_SIZE, MEM\_RESERVE, PAGE\_NOACCESS);

if (addr != NULL) {

bool ret = VirtualCopy(addr, (LPVOID) (pa>>8), PAGE\_SIZE, PAGE\_READWRITE | PAGE\_NOCACHE | PAGE\_PHYSICAL);

if (ret) {

RETAILMSG(RETAIL\_ON, (TEXT("VirtualCopy returned: %d\r\n"), ret));

return addr;

}

else {

return NULL;

}

}

else {

return NULL;

}

}

void UnMapRegister(void\* pRegs)

{

// free memory here

VirtualFree(pRegs, PAGE\_SIZE, MEM\_DECOMMIT | MEM\_RELEASE);

}

**Led\_driver.h**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Header file of generic stream device driver for windows ce 6.0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Declare the external entry points here. Use declspec so we don't need

// a .def file. Bracketed with extern C to avoid mangling in C++

#ifdef \_\_cplusplus

extern "C"{

#endif //\_\_cplusplus

\_\_declspec(dllexport) DWORD LED\_Init(LPCSTR pContext, DWORD dwBusContext);

\_\_declspec(dllexport) DWORD LED\_Open(DWORD dwContext, DWORD dwAccess, DWORD dwShare);

/\* ToDo: implement missing functions as defined !!! \*/

\_\_declspec(dllexport) BOOL LED\_PreDeinit(DWORD dwContext);

\_\_declspec(dllexport) BOOL LED\_Deinit(DWORD dwContext);

\_\_declspec(dllexport) BOOL LED\_PreClose(DWORD dwOpen);

\_\_declspec(dllexport) BOOL LED\_Close(DWORD dwOpen);

\_\_declspec(dllexport) DWORD LED\_Read(DWORD dwOpen, LPVOID pBuffer, DWORD dwCount);

\_\_declspec(dllexport) DWORD LED\_Write(DWORD dwOpen, LPVOID pBuffer, DWORD dwCount);

\_\_declspec(dllexport) DWORD LED\_Seek(DWORD dwOpen, long lDelta, WORD wType);

\_\_declspec(dllexport) DWORD LED\_IOControl(DWORD dwOpen, DWORD dwCode, PBYTE pIn, DWORD dwIn,

PBYTE pOut, DWORD dwOut, DWORD \*pdwBytesWritten);

\_\_declspec(dllexport) void LED\_PowerDown(DWORD dwContext);

\_\_declspec(dllexport) void LED\_PowerUp(DWORD dwContext);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifdef \_\_cplusplus

} // extern "C"

#endif //\_\_cplusplus

**Led\_driver.cpp:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Source file of generic stream device driver for windows ce 6.0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "stdafx.h"

#include <windows.h> // for all that windows stuff

#include <commctrl.h>

#include "led\_driver.h" // local program includes

#include "gpio.h"

#include "mapreg.h"

#include "ledswitch.h"

// trun on retail messages

#define RETAIL\_ON TRUE

// Used as a prefix string for all debug zone messages.

#define DTAG TEXT("LEDDrv: ")

// globals

HINSTANCE hInst; // dll instance handle

//driver instance structure

typedef struct{

DWORD dwSize;

int nNumOpens;

GPIOREG \* pGpioRegs;

}DRVCONTEXT, \*PDRVCONTEXT;

//=======================================================================

//DLLMain - DLL initialization entry point

//

BOOL APIENTRY DllMain( HANDLE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

hInst = (HINSTANCE)hModule;

switch(ul\_reason\_for\_call){

case DLL\_PROCESS\_ATTACH:

RETAILMSG(RETAIL\_ON, (TEXT("LedDriver DLL\_PROCESS\_ATTACH \r\n")));

// improve performance by passing on thread attach calls

DisableThreadLibraryCalls(hInst);

break;

case DLL\_PROCESS\_DETACH:

RETAILMSG(RETAIL\_ON, (TEXT("LedDriver DLL\_PROCESS\_DETACH\r\n")));

break;

}

return TRUE;

}

//=======================================================================

// LED\_Init - Driver initialization function

//

DWORD LED\_Init(LPCSTR pContext, DWORD dwBusContext){

PDRVCONTEXT pDrv;

RETAILMSG(RETAIL\_ON, (TEXT("LED\_Init++ dwContext: %x\r\n"), pContext));

// Allocate a driver instance structure - required if we want to manage

// more instances

pDrv = (PDRVCONTEXT)LocalAlloc(LPTR, sizeof(DRVCONTEXT));

if(pDrv){

// initialize structure

memset((PBYTE) pDrv, 0, sizeof(DRVCONTEXT));

pDrv->dwSize = sizeof(DRVCONTEXT);

// read registry to determine the size of the disk

// GetConfigData((DWORD)pContext);

}else{

RETAILMSG(RETAIL\_ON, (DTAG TEXT("LED\_Init failure. Out of memory\r\n"), pContext));

}

RETAILMSG(RETAIL\_ON, (DTAG TEXT("LED\_Init-- pDrv: %x\r\n"), pDrv));

return (DWORD)pDrv;

}

BOOL LED\_Deinit(DWORD dwContext)

{

return true;

}

//=======================================================================

// LED\_Open - Called when driver opened

// Use dwAccess and dwShare flags to manage access rights

//

DWORD LED\_Open(DWORD dwContext, DWORD dwAccess, DWORD dwShare){

PDRVCONTEXT pDrv = (PDRVCONTEXT)dwContext;

RETAILMSG(RETAIL\_ON, (DTAG TEXT("LED\_Open++ dwContext: %x\r\n"), dwContext));

// Verify that the context handle is valid

if(pDrv && (pDrv->dwSize != sizeof(DRVCONTEXT))){

return 0;

}

GPIOREG\* pGPIORegs = (GPIOREG\*)MapRegister(GPIO\_BASE);

if (pGPIORegs)

{

pDrv->pGpioRegs = pGPIORegs;

}

else

{

return 0;

}

initLeds(pGPIORegs);

initPushButtons(pGPIORegs);

initSwitches(pGPIORegs);

// Count the number of opens

InterlockedIncrement((long \*)&pDrv->nNumOpens);

RETAILMSG(RETAIL\_ON, (TEXT("LED\_Open-- \r\n")));

return (DWORD)pDrv;

}

BOOL LED\_Close(DWORD dwOpen)

{

PDRVCONTEXT pDrv = (PDRVCONTEXT)dwOpen;

RETAILMSG(RETAIL\_ON, (DTAG TEXT("LED\_Close++ dwContext: %x\r\n"), dwOpen));

// Verify that the context handle is valid

if(pDrv && (pDrv->dwSize != sizeof(DRVCONTEXT))){

return false;

}

UnMapRegister((void\*)pDrv->pGpioRegs);

// Count the number of opens

InterlockedDecrement((long \*)&pDrv->nNumOpens);

RETAILMSG(RETAIL\_ON, (TEXT("LED\_Close-- \r\n")));

return true;

}

DWORD LED\_Read(DWORD dwOpen, LPVOID pBuffer, DWORD dwCount)

{

PDRVCONTEXT pDrv = (PDRVCONTEXT)dwOpen;

RETAILMSG(RETAIL\_ON, (DTAG TEXT("LED\_Read++ dwContext: %x\r\n"), dwOpen));

// Verify that the context handle is valid

if(pDrv && (pDrv->dwSize != sizeof(DRVCONTEXT))){

return 0;

}

char\* result = (char\*) pBuffer;

\*result = readSwitches(pDrv->pGpioRegs);

\*result = (readPushButtons(pDrv->pGpioRegs) << 4);

// Count the number of opens

InterlockedDecrement((long \*)&pDrv->nNumOpens);

RETAILMSG(RETAIL\_ON, (TEXT("LED\_Read-- \r\n")));

return 1;

}

DWORD LED\_Write(DWORD dwOpen, LPVOID pBuffer, DWORD dwCount)

{

PDRVCONTEXT pDrv = (PDRVCONTEXT)dwOpen;

RETAILMSG(RETAIL\_ON, (DTAG TEXT("LED\_Write++ dwContext: %x\r\n"), dwOpen));

// Verify that the context handle is valid

if(pDrv && (pDrv->dwSize != sizeof(DRVCONTEXT))){

return 0;

}

char\* input = (char\*) pBuffer;

setLeds(pDrv->pGpioRegs, \*input);

// Count the number of opens

InterlockedDecrement((long \*)&pDrv->nNumOpens);

RETAILMSG(RETAIL\_ON, (TEXT("LED\_Write-- \r\n")));

return 1;

}

Der physikalische Speicher wird in der LED\_Open-Funktion gemappt. Danach kann mit den beiden Funktionen LED\_Read und LED\_Write von den GPIO-Registern gelesen bzw. auf diese geschrieben werden, da der Speicher für diese allokiert in LED\_Open allokiert wird.

Beim Testtreiber werden zuerst alle Leds getestet. Danach wird in einer Schleife immer der Wert der Buttons und Switches ausgegeben. Diese Werte werden in der Konsole auf ihre Gültigkeit überprüft.

**Led\_driver.cpp:**

#include "stdafx.h"

#include <windows.h>

#include <commctrl.h>

#include <iostream>

#include "RegEdit.h"

#include "DevDrv.h"

int \_tmain(int argc, \_TCHAR\* argv[])

{

// struct for registry entry

RegEntry EosRegEntries[] = {

{ TEXT("Dll"), // key DLL: specify the name of the dll which implements the driver

REG\_SZ, // type is a zero-terminated unicode string

0,

TEXT("\\Program Files\\test\_driver\\LedDriver.dll") }, // value: name of dll

{ TEXT("Prefix"), // key Prefix: specify the three-letter name of the driver

REG\_SZ, // type is a zero-terminated unicode string

0,

TEXT("LED") }, // value: name of dll

{ TEXT("Order"), // key Order: specify load order. drivers with lower numbers will be loaded before

REG\_DWORD, // type is a 4-byte binary value

4,

NULL },

{ NULL, 0, 0, NULL } // terminating entry

};

HANDLE hDev = INVALID\_HANDLE\_VALUE;

DWORD ret = AddEntryToRegistry(HKEY\_LOCAL\_MACHINE, EOSDRIVER\_REG\_KEY, EosRegEntries);

ret = LoadDriver(&hDev, EOSDRIVER\_REG\_KEY);

if (!ret){

HANDLE hDrv = CreateFileW(TEXT("LED1:"),

GENERIC\_READ | GENERIC\_WRITE,

0,

NULL,

OPEN\_EXISTING,

FILE\_ATTRIBUTE\_NORMAL,

NULL);

DWORD err = GetLastError();

char Buffer[] = "e";

DWORD NrOfBytesWritten;

DWORD NrOfBytesWrite = (DWORD)sizeof(Buffer); // driver uses only the first byte

DWORD NrOfBytesRead = 0;

bool read = false;

bool written = false;

char chr = 0xFF;

// test leds

Buffer[0] = 0x00;

written = WriteFile(hDrv, Buffer, NrOfBytesWrite, &NrOfBytesWritten, NULL);

Buffer[0] = 0x01;

written = WriteFile(hDrv, Buffer, NrOfBytesWrite, &NrOfBytesWritten, NULL);

Buffer[0] = 0x02;

written = WriteFile(hDrv, Buffer, NrOfBytesWrite, &NrOfBytesWritten, NULL);

Buffer[0] = 0x04;

written = WriteFile(hDrv, Buffer, NrOfBytesWrite, &NrOfBytesWritten, NULL);

Buffer[0] = 0x08;

written = WriteFile(hDrv, Buffer, NrOfBytesWrite, &NrOfBytesWritten, NULL);

Buffer[0] = 0x0F;

written = WriteFile(hDrv, Buffer, NrOfBytesWrite, &NrOfBytesWritten, NULL);

do{

// test buttons and switches

// read one byte from device driver

read = ReadFile(hDrv, Buffer, 1, &NrOfBytesRead, 0);

err = GetLastError();

// check for errors

if (!read && (err != 0)){

CloseHandle(hDrv);

UnloadDriver(&hDev);

CloseHandle(hDev);

return -1;

}

// print read values

std::cout << "Value of switches: " << (Buffer[0] & 0x0F) << std::endl;

std::cout << "Value of buttons: " << ((Buffer[0] & 0xF0) >> 4) << std::endl;

// End program if button T4 was pressed

} while (true);

// close the handle (CreateFile)

CloseHandle(hDrv);

// we need no further CloseHandle because hDev is still invalid after

// unloading the driver

UnloadDriver(&hDev);

}

return 0;

}