2.Aufgabe

2.1)

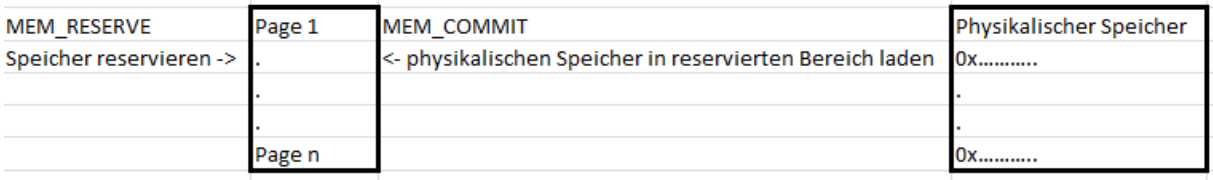
Der physikalische Adressraum ist der Adressraum des realen Arbeitsspeichers. Der virtuelle Adressraum beschreibt einen gedachten, nicht real vorhandenen Arbeitsspeicher, der als virtueller Speicher bezeichnet wird. Der virtuelle Adressraum ist im Normalfall größer als der physikalische Adressraum. Der virtuelle Speicher wird auf der Platte abgebildet. Dabei werden nicht alle Teile des Programms und analog nicht der gesamte Datenbereich benötigt. Es reicht also, nur die benötigten Programm- und Datenbereichs-Teile im Arbeitsspeicher zu halten. Bei Bedarf können die zerlegten Programme und Daten in den Speicher geladen werden, wenn sie von der CPU benötigt werden. Zur Abarbeitung ist es dann erforderlich, die virtuellen Adressen in physikalische Adressen zu transformieren.

2.2)

Die Funktion VirtualAlloc reserviert Speicher auf Page-Level, also der kleinsten Speichereinheit die von der CPU allokiert oder freigegeben wird. Die Größe einer Page ist abhängig von der CPU, sie kann entweder 1024 oder 4096 Bytes groß sein. Die Speicherallokierung erfolgt in 2 Schritten. Zuerst wird ein virtueller Speicher reserviert. Die Reservierung benötigt keinen Arbeitsspeicher. Es wird nur verhindert, dass ein Teil des virtuellen Adressraums für andere Zwecke benutzt wird. Nach dieser Reservierung können Teile des Bereichs oder der gesamte Bereich "committed" werden, wodurch physikalischer Speicher in den reservierten Bereich "gemappt" wird. Also wird die VirtualAlloc-Funktion sowohl für die Reservierung als auch dem "committen" des Speichers verwendet.

MEM_RESERVE reserviert den virtuellen Adressspeicher, der später "committed" wird. MEM_COMMIT allokiert den Speicher, der vom Programm schließlich benutzt wird. Auf reservierte Pages kann nicht zugegriffen werden, bis ein anderer Aufruf von VirtualAlloc mit dem Parameter MEM_COMMIT aufgerufen wird.

Skizze Aufruf Virtual Alloc:



2.3)

In Windows CE 5.0 bezieht sich der Kernel Mode auf den Zugriffsbereich für einen Thread. Ist ein Thread im Kernel Mode, kann auf den Kernel-Adressraum zugegriffen werden. Dabei kann mithilfe eines Aufrufes von SetKMode jederzeit in den Kernel Mode wechseln bzw. diesen verlassen. Die meisten APIs sind dabei nicht im Kernel implementiert (nk.exe), also geht ein Thread normalerweise nicht in den Kernel Mode wenn eine API aufgerufen wird, es sei denn, es wird eine API aufgerufen, die in nk.exe implementiert wurde. Dann wird temporär in den Kernel Mode gewechselt.

In Windows CE 6.0 ist die Implementierung gleich wie bei 5.0, jedoch wird die SetKMode API nicht mehr unterstützt. Threads können nicht mehr zu einem beliebigen Zeitpunkt in den Kernel Mode wechseln oder den Kernel Mode verlassen. Es ist also nicht einfach so möglich, auf physikalischen Speicher zuzugreifen. Jedoch geht ein Thread nun standardmäßig in den Kernel Mode, da die meisten Module der System-APIs in Windows CE 6.0 in den Kernel-Prozess geladen werden.

Der Unterschied zwischen den beiden Versionen liegt also darin, dass in CE 6.0 ein Thread nur in den Kernel Mode wechselt, wenn Code innerhalb des Kernel-Prozesses ausgeführt wird. Wenn eine User-Applikation einen Funktionspointer an Kernel Mode Code übergibt, und dieser den Funktionspointer direkt aufruft, ist der Thread auch dann noch im Kernel Mode, während eigentlich User Code ausgeführt wird. Dies ist sehr unsicher, da der Code des User Modes alle Kerneladressen und Kernel-APIs erreichen kann. Daher soll beim Schreiben von Treibern immer die CEDDK-Funktion (CeDriverPerformCallback) aufgerufen werden. Dadurch geht der Thread wieder ordnungsgemäß in den User-Prozess zurück, damit die hier aufgerufenen Funktionen nicht auf irgendwelche Kernel-spezifischen Adressen bzw. APIs erreichen kann.

5.Aufgabe

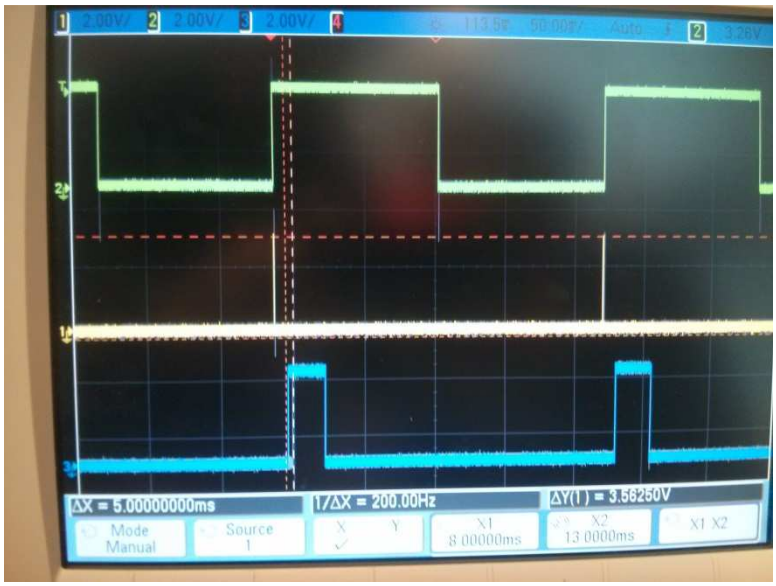
5.1)

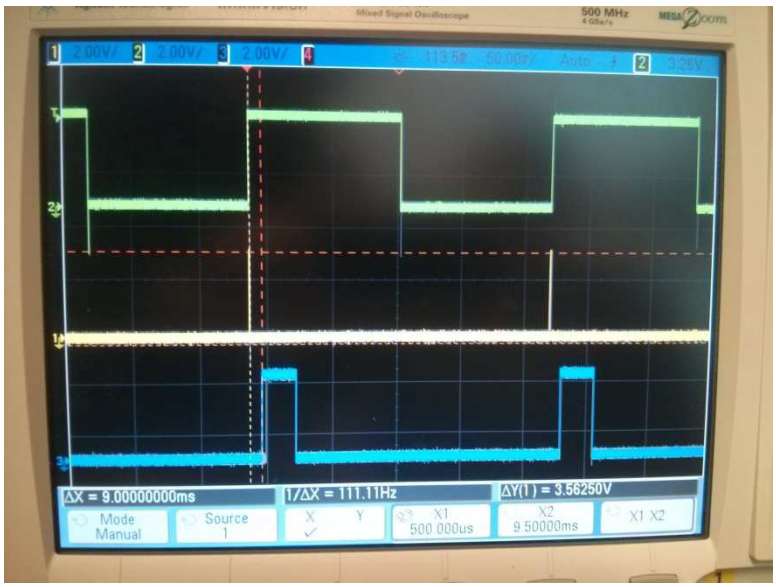
IST: Latency - 9ms

Jitter – 5ms

ISR: Latency – 0ms

Jitter – 0ms



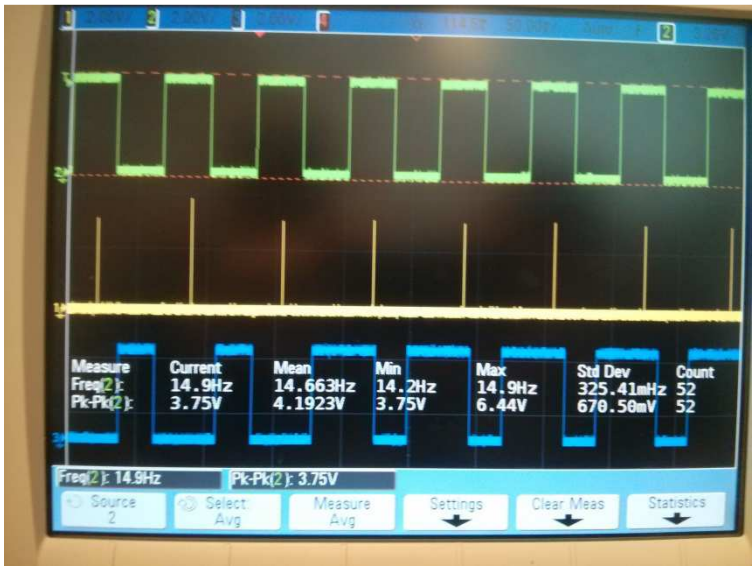


5.2)

Bei einer normalen Priorität liegt die maximale Frequenz bei ca 9.5 Hz. Bei einer Priorität von 10 liegt sie bei ca 15 Hz.

Normale Priorität:

Priorität 10:



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"
#include "eos_defines.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
ISR_INFO isr_info;
bool KillFlag = false;

//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;
}

DWORD WINAPI ISTFunction(LPVOID lpParam)
{
        isr_info.pGPIORegs->gafr2_l &= ~(1 << 31 | 1 << 30); // set alternate function 0 for
GPIO79
        isr_info.pGPIORegs->gpdr2   |= (1 << 15);  //set GPIO79 as output

        RETAILMSG(RETAIL_ON, (TEXT("ISTFunction start \r\n")));
        if (lpParam != 0)
        {
                HANDLE hEvent = (HANDLE) lpParam;
                if (hEvent != 0)
```

```
		{
			while (!KillFlag)
			{
				WaitForSingleObject(hEvent, INFINITE);

				if (!KillFlag)
				{
					RETAILMSG(RETAIL_ON, (TEXT("ISTFunction setLed 1 \r\n")));
					isr_info.pGPIORegs->gpsr2 = (1 << 15); // reset GPIO79
					RETAILMSG(RETAIL_ON, (TEXT("ISTFunction clearLed 1
\r\n")));
					isr_info.pGPIORegs->gpcr2 = (1 << 15); // set GPIO79
				}
			}
		}
	}
	RETAILMSG(RETAIL_ON, (TEXT("ISTFunction end \r\n")));
	return 0;
}

//====================================================================
// LED_Init - Driver initialization function
//
DWORD LED_Init(LPCSTR pContext, DWORD dwBusContext){
	PDRVCONTEXT pDrv;
	RETAILMSG(RETAIL_ON, (TEXT("Test debug output without context \r\n")));
	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));
	}

	DWORD dwGpio = 15;
	DWORD dwIrq = 0;
	DWORD dwEdge = GPIO_EDGE_RISING;

	if (!KernelIoControl(IOCTL_HAL_GPIO2IRQ, &dwGpio, sizeof(DWORD), &dwIrq, sizeof(DWORD),
NULL))
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. IOCTL_HAL_GPIO2IRQ\r\n"),
pContext));
		return 0;
	}

	if (!KernelIoControl(IOCTL_HAL_IRQEDGE, &dwIrq, sizeof(BYTE), &dwEdge, sizeof(BYTE),
NULL))
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. IOCTL_HAL_IRQEDGE\r\n"),
pContext));
		return 0;
	}

	if (!KernelIoControl(IOCTL_HAL_REQUEST_SYSINTR, &dwIrq, sizeof(DWORD), &(pDrv-
>dwSysIntr), sizeof(DWORD), NULL))
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure.
IOCTL_HAL_REQUEST_SYSINTR\r\n"), pContext));
		return 0;
	}
```

```
	pDrv->hEvent = CreateEvent(NULL, false, false, TEXT("ISTEvent"));
	if (!(pDrv->hEvent))
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. Create Event failed.\r\n"),
pContext));
		return 0;
	}

	if (!InterruptInitialize(pDrv->dwSysIntr, pDrv->hEvent, NULL, 0))
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. InterruptInitialize
failed.\r\n"), pContext));
		return 0;
	}

	pDrv->hIsrHandle = LoadIntChainHandler(TEXT("\\Program Files\\Drivers\\ISRDll.dll"),
TEXT("ISRHandler"), dwIrq);
	if (!pDrv->hIsrHandle)
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. LoadIntChainHandler
failed.\r\n"), pContext));
		return 0;
	}

	isr_info.SysIntr = pDrv->dwSysIntr;

	isr_info.pGPIORegs = (GPIOREG*)CreateStaticMapping(GPIO_BASE >> 8, sizeof(GPIOREG));
	if (!isr_info.pGPIORegs)
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. CreateStaticMapping
failed.\r\n"), pContext));
		return 0;
	}

	if (!KernelLibIoControl(pDrv->hIsrHandle, IOCTL_ISR_INFO, &isr_info, sizeof(isr_info),
NULL, 0, NULL))
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. IOCTL_HAL_RELEASE_SYSINTR
failed.\r\n"), pContext));
		return 0;
	}

	pDrv->hIntThread = CreateThread(NULL, 0, ISTFunction, pDrv->hEvent, 0, NULL);
	if (!pDrv->hIntThread)
	{
		RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init failure. CreateThread failed.\r\n"),
pContext));
		return 0;
	}


	RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Init-- pDrv: %x\r\n"), pDrv));

	return (DWORD)pDrv;

FREECHAINHANDLER:
	FreeIntChainHandler(pDrv->hIsrHandle);
INTERRUPTDISABLE:
	InterruptDisable(pDrv->dwSysIntr);
KERNELIOCONTROL:
	KernelIoControl(IOCTL_HAL_RELEASE_SYSINTR, &pDrv->dwSysIntr, sizeof(DWORD), NULL, 0,
NULL);
CLOSEHANDLE:
	CloseHandle(pDrv->hEvent);
DELETESTATICMAPPING:
	DeleteStaticMapping((LPVOID)isr_info.pGPIORegs, sizeof(GPIOREG));
}

BOOL LED_Deinit(DWORD dwContext)
{
```

```
	PDRVCONTEXT pDrv = (PDRVCONTEXT)dwContext;

	RETAILMSG(RETAIL_ON, (DTAG TEXT("LED_Deinit++ dwContext: %x\r\n"), dwContext));

	// Verify that the context handle is valid
	if(pDrv && (pDrv->dwSize != sizeof(DRVCONTEXT))){
		return 0;
	}

	// terminate thread
	KillFlag = true;
	SetEvent(pDrv->hEvent);

	FreeIntChainHandler(pDrv->hIsrHandle);
	InterruptDisable(pDrv->dwSysIntr);
	KernelIoControl(IOCTL_HAL_RELEASE_SYSINTR, &pDrv->dwSysIntr, sizeof(DWORD), NULL, 0,
NULL);
	CloseHandle(pDrv->hEvent);
	DeleteStaticMapping((LPVOID)isr_info.pGPIORegs, sizeof(GPIOREG));

	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;
	}

	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;
}
```

ISRDll.cpp:

```
//
// Copyright (c) Microsoft Corporation.  All rights reserved.
//
//
// Use of this source code is subject to the terms of the Microsoft end-user
// license agreement (EULA) under which you licensed this SOFTWARE PRODUCT.
// If you did not accept the terms of the EULA, you are not authorized to use
// this source code. For a copy of the EULA, please see the LICENSE.RTF on your
// install media.
//
// isrdll.c
//
// Installable Interrupt Service Routine.
//
//
#include <windows.h>
#include "gpio.h"
#include "isr.h"

// NOTE: It's very importat that this dll doesn't use any Windows APIs or
//       any other reference to external dlls. (it will not load otherwise)

#define SYSINTR_NOP         0
#define SYSINTR_CHAIN       3


DWORD g_dwSysIntr;
GPIOREG* g_pGPIORegs;
DWORD g_dwValue;


// Globals

/*
 @doc INTERNAL
 @func    BOOL | DllEntry | Process attach/detach api.
 *
 @rdesc The return is a BOOL, representing success (TRUE) or failure (FALSE).
 */
BOOL __stdcall DllEntry(HINSTANCE hinstDll,         // @parm Instance pointer.
                                            DWORD dwReason,             // @parm Reason routine
is called.
                                            LPVOID lpReserved)           // @parm system
parameter.
{
     if (dwReason == DLL_PROCESS_ATTACH) {
     }

     if (dwReason == DLL_PROCESS_DETACH) {
     }

     return TRUE;
}


// this can be used for ISR dll that have multile instances (like giisr.dll)
DWORD CreateInstance(void){
    return 0;
}

void DestroyInstance(DWORD InstanceIndex){
}



BOOL IOControl(DWORD   InstanceIndex,
                        DWORD   IoControlCode,
                        LPVOID  pInBuf,
                        DWORD   InBufSize,
```

```
                        LPVOID  pOutBuf,
                        DWORD   OutBufSize,
                        LPDWORD pBytesReturned){

        if (pBytesReturned) {
         *pBytesReturned = 0;
    }

    switch (IoControlCode){
              case IOCTL_ISR_INFO:
                     // Copy instance information
                     if ((InBufSize != sizeof(ISR_INFO)) || !pInBuf) {
                            // Invalid size of output buffer or input buffer pointer
                            return FALSE;
                     }
                     g_dwSysIntr=((ISR_INFO*)pInBuf)->SysIntr;
                     g_pGPIORegs=((ISR_INFO*)pInBuf)->pGPIORegs;

                     // Do some initializations:
                     g_dwValue=0;
                     g_pGPIORegs->gafr1_l &= ~(1 << 9 | 1 << 8); // set alternate function 0
for GPIO36
                     g_pGPIORegs->gpdr1   |= (1 << 4);  //set GPIO36 as output

                     RETAILMSG(1, (TEXT("Inst. ISR DLL: ") TEXT("IOCTL: got SYSINTR: %u , GPIO:
%u \r\n"),g_dwSysIntr, g_pGPIORegs));

                     break;


        case IOCTL_GET_VALUE:
         // Copy instance information
         if ((OutBufSize != sizeof(DWORD)) || !pOutBuf) {
             // Invalid size of output buffer or input buffer pointer
             return FALSE;
         }

         // The compiler may generate a memcpy call for a structure assignment,
         // and we're not linking with the CRT, so use our own copy routine.
         *((DWORD*)pOutBuf) = g_dwValue;
                if(pBytesReturned) *pBytesReturned=sizeof(DWORD);

         break;

    default:
         // Invalid IOCTL
         return FALSE;
    }
    return TRUE;
}



DWORD ISRHandler(DWORD InstanceIndex){
        RETAILMSG(1, (TEXT("ISRHandler setLed \r\n")));
        g_pGPIORegs->gpsr1 = (1 << 4); // set GPIO36 -> LED1 on

        g_dwValue++;

        //RETAILMSG(1, (TEXT("ISRHandler clearLed \r\n")));
        g_pGPIORegs->gpcr1 = (1 << 4); // reset GPIO36 -> LED1 off

        // if this isr is used to handle a shared interrupt, here we should check if this
interrupt is really intended for us.
        // If it's for us we retrun g_dwSysIntr, if it's not for us we should retrun
SYSINTR_CHAIN

    return g_dwSysIntr;
}
```