FH-OÖ Hagenberg/ESD
Embedded Operating Systems, WS 2014
Florian Eibensteiner, Rainer Findenig, Josef Langer © 2014

8. Übung: Windows CE 6.0: Driver Development, Part Two

OBERÖSTERREICH

Name(n): Punkte:

1 Windows CE: Interrupthandling

Wie aus der Vorlesung bereits bekannt sein sollte, teilt sich die Interruptverarbeitung in die zwei Schritte Interrupt Service Routine (ISR) und Interrupt Service Thread (IST) auf [PB08].

Jedem Interrupt Request (IRQ) wird eine ISR zugewiesen, wobei eine ISR mehrere Quellen haben kann. Tritt nun ein Interrupt auf, ruft der Kernel die entsprechende ISR auf. Die ISR liefert als Rückgabewert einen System Interrupt (SYSINTR) an den Kernel zurück, der das mit dem SYSINTR assizoierte Event feuert, was in weiterer Folge dazu führt, dass der Scheduler den entsprechenden IST aktiviert.

Daraus läßt sich ableiten, dass die ISR lediglich dazu dient, den entsprechenden SYSINTR zu maskieren und zurück zu geben. Man kann natürlich auch noch andere zeitkritische Aufgaben in der ISR implementiern. Allerdings muss beachtet werden, dass während des Ausführens der ISR alle andern IRQs, die gleiche bzw. niedrigere Priorität haben, blockiert werden und daher nicht abgearbeitet werden können.

Eine ISR wird in einer DLL implementiert, welche zur Laufzeit mithilfe der Funktion *LoadIntChainHandler* geladen werden kann. Für einen IRQ können mehrere ISRs registriert werden, die vom Kernel nach dem FIFO-Prinzip aufgerufen werden wenn der entsprechende IRQ auftritt. Liefert eine ISR den Wert SYSINTR_CHAIN zurück (ISR verarbeitet Interrupt nicht), ruft der Kernel die nächste ISR aus der FIFO auf, solange bis eine ISR einen gültigen SYSINTR zurückgibt. Die restlichen ISRs in der FIFO werden in diesem Fall nicht mehr aufgerufen. Eine ISR kann jederzeit mit der Funktion *FreeIntChainHandler* entladen werden. Allerdings wird lediglich die ISR aus der Interrupt-Chain entfernt, die DLL bleibt jedoch geladen bis ein Neustart des Systems erfolgt. Daher sind Änderungen der DLL immer mit einem Neustart des Systems verbunden.

Der IST ist ein gewöhnlicher System-Thread, dessen Priorität hoch genug sein sollte, damit die von der Quelle (IRQ) geforderte Funktionalität in entsprechender Zeit abgearbeitet werden kann. Der IST wird üblicherweise im Treiber implementiert und muss folgende Funktionalität beinhalten [PB08]:

- warten auf das Event, welches mit dem SYSINTR assoziiert ist,
- und den Kernel benachrichtigen, dass die Abarbeitung des Interrupts abgeschlossen ist (*InterruptDone*).

Erklären Sie in eigenen Worten den Ablauf des Interrupt-Handlings (siehe Folie 41 u. 42 aus der
□ VL3: Windows CE).

2 Interrupt für Schalter 1

Erweitern Sie ihren Treiber aus letzten Übung um ein Interrupt-Handling für Schalter 1 (GPIO 15). Jedesmal wenn eine steigende Flanke an GPIO 15 auftritt soll ein Interrupt ausgelöst werden der einen möglichst kurzen Puls an LED 2 (GPIO 36) ausgibt. Erstellen Sie eine DLL, welche die geforderte Funktionalität implementiert und orientieren Sie sich dabei an der Vorlage im Moodle (*isrdll.c*).

Die ISR-DLL implementiert folgnde Entry-Points:

- IOControl: Diese Funktion ermöglicht die Kommunikation und Konfiguration der ISR. In unserm Fall wird die Funktion dazu genutzt, um den SYSINTR und den Pointer auf die GPIOs einzustellen.
- CreateInstance, DestroyInstance: Diese Funktionen werden benötigt um mehrere Instanzen der ISR zu verwalten. In unserem Fall werden diese Funktionen nicht benötigt, allerdings müssen sie exportiert werden damit der Ladevorgang der DLL erfolgreich ist.
- ISRHandler: Diese Funktion implementiert die eigentliche ISR.

□ Vervollständigen Sie die DLL damit die obige Spezifikation erfüllt wird.

3 Interrupt Service Thread

Der Hauptteil des Interrupt-Handlings soll aus oben genannten Gründen in der IST erledigt werden. Implementieren Sie einen Interrupt Service Thread, welcher auf das Event der ISR wartet und jedesmal einen Puls and LED 1 (GPIO 79) ausgibt wenn sie getriggert wird. Wie oben erwähnt, muss die IST dem Kernel mitteilen, dass die Abarbeitung des Interrupts abgeschlossen ist. Beachten Sie, dass der Thread beendet werden muss wenn der Treiber entladen wird. Überlegen Sie sich einen geeigneten Mechanismus und beenden Sie den Thread an einer geeigneten Stelle (*PreClose*, *Close*,
□ *PreDeinit*, *Deinit*). Begründen Sie ihre Entscheidung.

4 Erweitern des Treibers

Um beide Teile, ISR und IST, in den Treiber einbinden zu können, muss die Initialisierung des Treibers angepasst bzw. erweitert werden. Folgende Schritte müssen gemacht werden:

- Ermitteln der IRQ Nummer für den GPIO Pin.

```
KernelIoControl(IOCTL_HAL_GPIO2IRQ, &dwGpio, sizeof(DWORD),
                &dwIrq, sizeof(DWORD), NULL);
```

- Definition der Flankensensitivität. (Hier werden beide Buffer für die *Eingabe* verwendet!)
```
KernelIoControl(IOCTL_HAL_IRQEDGE, &dwIrq, sizeof(BYTE),
                &dwEdge, sizeof(BYTE), NULL);
```
- Ermitteln der SYSINTR Nummer welche dem IRQ entspricht.
```
KernelIoControl(IOCTL_HAL_REQUEST_SYSINTR, &dwIrq, sizeof(DWORD),
                &dwSysIntr, sizeof(DWORD), NULL);
```
- Erstellen des Events, auf das der IST wartet (*CreateEvent*).
- Registrieren des Events für den ermittelten SYSINTR beim Kernel und aktivieren des Interrupts (*InterruptInitialize*).
- Laden der DLL und einhängen der ISR in die Interrupt-Chain (*LoadIntChainHandler*).
- Die ISR kann nur auf statisch gemappten Speicher zugreifen, daher muss der Speicher für den Zugriff auf die GPIOs in der ISR, statisch gemappt werden. (*CreateStaticMapping*).
- Konfigurieren der ISR durch setzen des SYSINTR und einem Pointer (auf den statisch gemappten Speicher) auf die GPIOs.
```
KernelLibIoControl(hISR, IOCTL_ISR_INFO, &isr_info, sizeof(isr_info),
                   NULL, 0, NULL);
```
- Erstellen eines Threads (*CreateThread*).

Die Abfolge der Konfigurationsschritte ist ebenfalls wichtig und sollte nach obiger Auflistung erfolgen.

Wenn der Treiber beendet wird, müssen neben den bisherigen Schritten folgende Punkte beachtet werden:

- Beenden des IST.
- Enladen der ISR (*FreeIntChainHandler*).
- Deaktivieren des Interrupts (*InterruptDisable*).
- Freigabe des allokierten SYSINTR.
```
KernelIoControl(IOCTL_HAL_RELEASE_SYSINTR, &dwSysIntr, sizeof(DWORD),
                NULL, 0, NULL);
```
- Schließen des Events, auf welches der IST sensitiv war (*CloseHandle*).
- Freigabe des statisch gemappten Speicherbereichs (*DeleteStaticMapping*).

Beim Entladen des Treibers bzw. bei der Deinitialisierung muss die Reihenfolge der einzelnen Schritte ebenfalls berücksichtigt werden.

Die benötigten IOCTL-Codes und Funktionen sind in der Datei *eos_defines.h* enthalten. Zusätzliche Informationen sind in der MSDN unter [win] enhthalten.

Beachten Sie bei der Implementierung der Initialisierung, dass sobald ein Fehler auftritt, alle zuvor gemachten Schritte wieder rückgängig gemacht werden müssen. Anderenfalls könnte Ihre Implementierung zu einem instabilen System und nicht reproduzierbarem Verhalten führen.

5 Real-Time-Verhalten von Windows CE

Nun soll das Echtzeitverhalten von Windows CE 6.0 untersucht werden. Benutzen Sie den Aufbau bzw. das Messverfahren wie es in Abb.1 dargestellt ist.

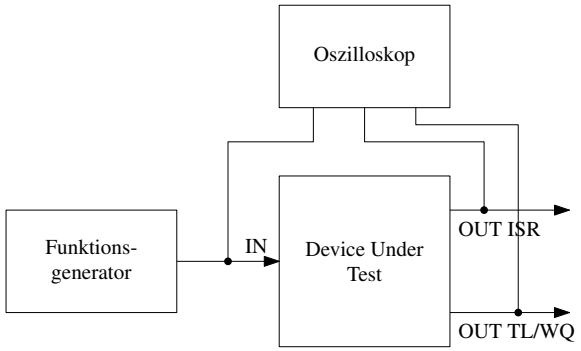

Figure 1: Testaufbau für Aufgabe 1.

Die Anschlussbelegung ist in nachstehender Tabelle zusammengefasst.

Name	**GPIO**	**X2**
`OUT_IST`	79	15
`OUT_ISR`	36	14
`IN`	15	16
	`GND`	20

Verwenden Sie einen Funktionsgenerator und messen Sie mit dem Oszilloskop den Jitter und die Latency der ISR und des IST, beginnen Sie dabei bei einer sehr niedrigen Frequenz und erhöhen Sie diese dann solange, bis das System an seine Grenzen stößt (IST kann nicht mehr ausgeführt werden bevor nächste IRQ auftritt).

Da der IST ein normaler Thread ist, wrikt sich dessen Priorität auf dessen Latency und Jitter aus. Erhöhen Sie daher die Priorität des IST auf 10 (`CeSetThreadPriority()`) und wiederholen Sie die Messungen. Welche Unterschiede (im Verhalten der *ISR* und der *IST*) können Sie beobachten,
□ und wie können Sie diese erklären?

Wie hoch ist die maximale Eingangsfrequenz die verarbeitet werden kann? Belegen Sie das Ergebnis
□ mit einer Messung.

References

[PB08] Stanislav Pavlov and Pavel Belevsky. *Windows Embedded CE 6.0 Fundamentals*. Microsoft Press Corp., 2008.

[win] Kernel Functions – MSDN Library – Shared Windows Mobile 6 and Windwos Embedded CE 6.0 Library. Website. Jan 2009, <http://msdn.microsoft.com/en-us/library/ee482951.aspx>.