**Implementierung**

# Echtzeitbetriebssysteme

**Architektur eines RTOS am Beispiel Windows CE**

23.03.2010 | Autor: Rudi Swiontek und Ralf Ebert*

**Echtzeitbetriebssysteme, kurz RTOS, besitzen neben harter Echtzeitfähigkeit einen Systemzeitgeber (Systemtimer), haben ein deterministisches (vorhersagbares) Verhalten und werden durch Ereignisse gesteuert. Diese können durch die Hardware (I/O-Controller) ausgelöst werden (Interrupts) oder durch CPU-Ereignisse (Exceptions, Division durch 0). Beispielhaft werden diese Eigenschaften an Hand von Windows Embedded CE 6.0 erklärt.**

*Die Autoren: Dipl.-Inf. Rudi Swiontek (rudi.swiontek@hilf.de) ist im Bereich Schulung und Entwicklung bei der Firma HILF! tätig. Als erfahrener Produktspezialist ist er u.a. für die Windows-Embedded-Betriebssysteme zuständig. Ralf Ebert (ohne Foto) (ralf.ebert@silica.com) ist Produktspezialist beim Microsoft-Embedded-Distributor Avnet Silica und dort unter anderem für Design-In-Unterstützung und Beratung zuständig.

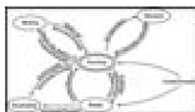
Ein Echtzeit-Embedded-System besteht aus vier Komponenten: der Hardware, dem BSP, dem RTOS und der Anwendersoftware. Das BSP realisiert die Anpassung des Betriebssystems an die zu Grunde liegende Hardware. Diese vier Komponenten müssen im Zusammenspiel das deterministische Verhalten garantieren. Hier gilt je härter die zeitlichen Anforderungen sind, desto schneller muss die eingesetzte Hardware sein. Für jedes verwendete API wird eine minimale und maximale Verzögerungszeit (Latency) garantiert, egal wie viele Ereignisse gerade bearbeitet werden müssen.

Für jede Anwendung kann ein kleiner Prototyp realisiert werden um zu zeigen, dass die Anwendung sich mit dem verwendeten Betriebssystem realisieren lässt. Dazu werden die maximalen Verzögerungszeiten zusammengezählt (worst-case definition). Ist die maximale Zeit aller verwendeten APIs kürzer als das benötigte Zeitfenster, dann wird immer garantiert rechtzeitig zu reagieren und alle Ereignisse zu verarbeiten (die Firma: www.dedicated-systems.com testet RTOS-Systeme und zeigt

Meßmethoden auf, wie man die verschiedenen Verzögerungszeiten ermitteln kann).

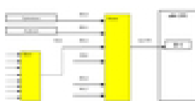
**Klassischer Aufbau eines RTOS**

Echtzeitbetriebssysteme besitzen einen Kernel. Es handelt sich im Prinzip um einen hardwareunabhängigen effizienten Scheduler mit präzise definierten Schnittstellen, der die höchsten Rechte (Privileged-Level) im System hat. Das Echtzeitbetriebssystem bildet eine Schale um den Kernel, die von Applikationsprogrammen nicht durchdrungen wird. Der Kernel beinhaltet die Systemuhr (Retriggerable Timer), den Scheduler (Threadmanager) und eine Verwaltung der Synchronisationsobjekte. Der Scheduler steuert den zeitlichen Ablaufen der Threads. Wird ein neuer Thread erzeugt, d.h. dem Kern bekannt gemacht (CreateThread), laufen verschieden Aktionen gleichzeitig ab.



Eine Thread ist eine organisatorische Einheit, die aus ausführbarem Programmcode, eigenem Speicher und Variablen besteht und ist gekennzeichnet durch eine Priorität und einen Taskzustand. Der Thread wird mit seiner Priorität in eine Kernelliste eingetragen. Der Kern verwaltet mehrere Listen und für den richtigen zeitlichen Ablauf wird eine Ready Queue angelegt (Bild 3), die die einzelnen ablaufbereiten Threads beinhaltet. Der Scheduler prüft anhand der Ready Liste welcher Thread die CPU zugeteilt bekommt. In prioritätsgesteuerten Systemen bekommt derjenige Thread die CPU zugeteilt, der ablaufbereit ist und die höchste Priorität besitzt.

**So bearbeitet das RTOS Interrupts**



Alle Interrupt-Programme sind im BSP (Board Support Packed) integriert, Teil des RTOS und besitzen die höchste Privilegierungs-Stufe im System. Interrupts sind Benachrichtigungen, die von der Hardware oder Software generiert werden, um die CPU zu informieren, dass ein Ereignis aufgetreten ist, das sofort beachtet werden muss (z.B. Systemtimer oder Keyboard Interrupt).

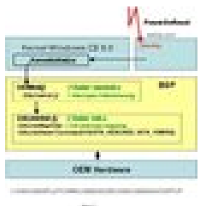
Beim Auftreten eines Interrupts hält die CPU die Ausführung der aktuellen Anwendung (Thread) an, springt zu einem Handler im Kernel, um auf das Ereignis zu reagieren, und setzt die Ausführung des ursprünglichen Threads fort, nachdem der Interrupt verarbeitet wurde. Jeder Interruptquelle wird eine Priorität zugeordnet. So hat der

Systemtimer die höchste Interrupt-Priorität und der Keyboard-Interrupt die zweithöchste Priorität.

**Nested versus Shared Interrupts**

Echtzeitbetriebssysteme unterstützen sowohl „Nested-Interrupts“ als auch „Shared-Interrupts“. Nested-Interrupt bedeutet, dass ein Interrupt mit einer höheren Priorität einen Interrupt mit einer niedrigeren Priorität jederzeit unterbrechen kann. (z.B. Systemtimer-Interrupt unterbricht einen Keyboard-Interrupt). Shared Interrupt bedeutet, dass sich mehrere PCI-Geräte einen Interrupt (IRQ10) teilen müssen. Je nach Interruptquelle kann ein Interrupt als Pegel- oder Flanken-getriggerter Interrupt programmiert werden.

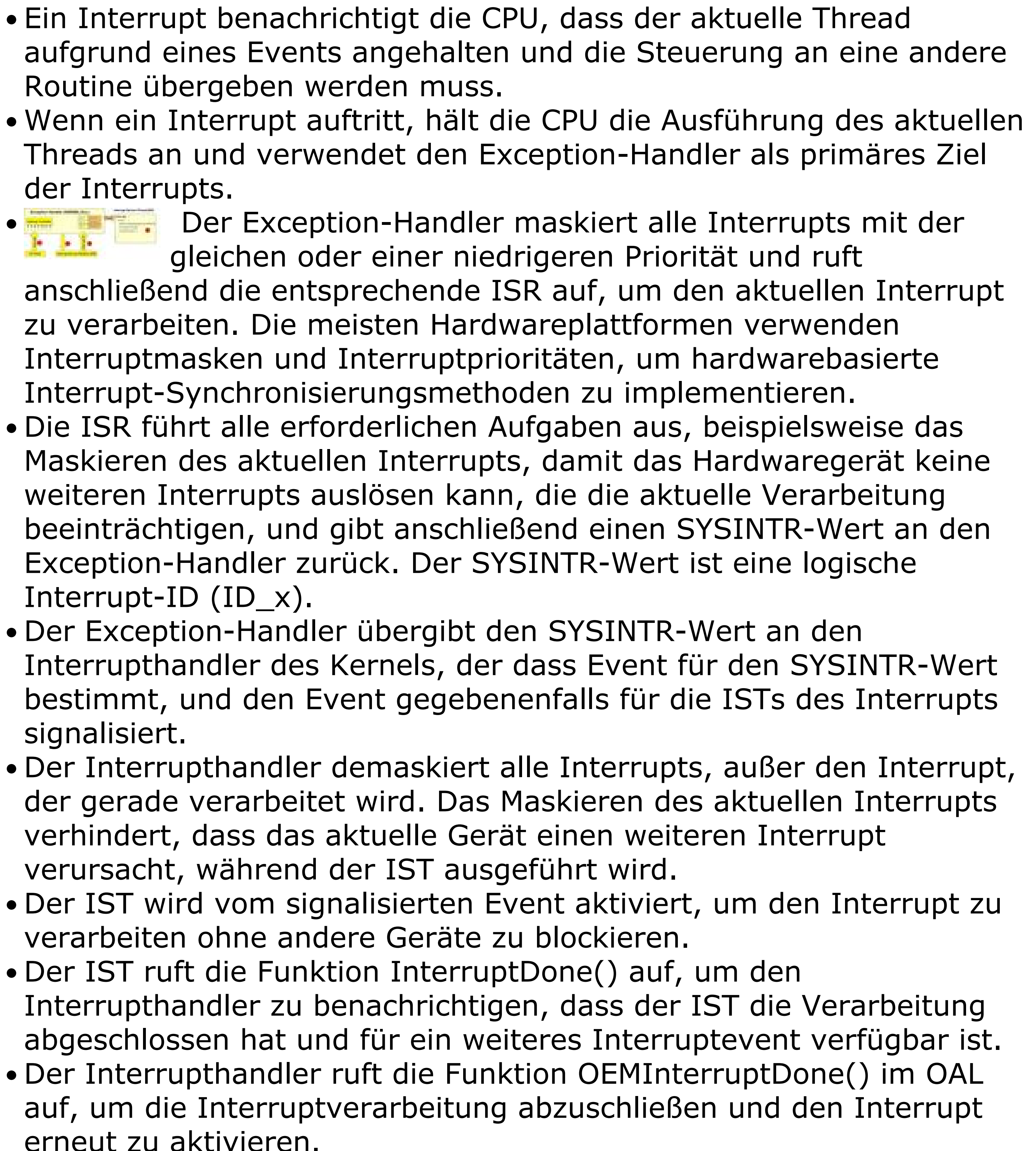
**Windows Embedded CE 6.0: Architektur für die Interruptverarbeitung**



Windows Embedded CE 6.0 ist ein portables Betriebssystem, das aufgrund der flexiblen Interruptbehandlungsarchitektur unterschiedliche CPU-Typen (ARM, x86, MIPS und SHx) mit verschiedenen Interrupts unterstützt. Die Architektur für die Interruptverarbeitung nutzt die Interrupt-Synchronisierungsfunktionen im OEM Adaptation Layer (OAL) und die Thread-Synchronisierungsfunktionen von Windows Embedded CE, um die Interruptverarbeitung in ISRs (Interrupt Service Routine) und ISTs (Interrupt Service Thread) aufzuteilen.

Die Interruptverarbeitung in Windows Embedded CE 6.0 basiert auf folgenden Konzepten:

- Nach einem PowerOnReset wird das Assemblerprogramm _StartUp() ausgeführt, das die CPU und den Speicher initialisiert. Anschließend erfolgt der Aufruf _KernelInitialize() um Windows CE zu initialisieren.
- Der Kernel ruft die Funktion OEMInit() im OAL auf, um alle verfügbaren im Kernel integrierten ISRs basierend auf den IRQ-Werten mit den entsprechenden Hardwareinterrupts zu registrieren. Die IRQ-Werte identifizieren die Quelle des Interrupts in den Registern der Prozessor-Interruptcontroller. Auch wird darin mit der Funktion OALIntrMapInit() eine Zuordungstabelle angelegt, in der jedem HW-Interrupt eine ID zugeordnet wird.
- Die Gerätetreiber rufen die Funktion LoadIntChainHandler() auf, um die in den ISR DLLs implementierten ISRs dynamisch zu installieren. LoadIntChainHandler lädt die ISR DLL in den Kernelspeicher und registriert die angegebene ISR mit dem IRQ-Wert in der

Interrupt-Verteilertabelle des Kernels.
- Ein Interrupt benachrichtigt die CPU, dass der aktuelle Thread aufgrund eines Events angehalten und die Steuerung an eine andere Routine übergeben werden muss.
- Wenn ein Interrupt auftritt, hält die CPU die Ausführung des aktuellen Threads an und verwendet den Exception-Handler als primäres Ziel der Interrupts.
-  Der Exception-Handler maskiert alle Interrupts mit der gleichen oder einer niedrigeren Priorität und ruft anschließend die entsprechende ISR auf, um den aktuellen Interrupt zu verarbeiten. Die meisten Hardwareplattformen verwenden Interruptmasken und Interruptprioritäten, um hardwarebasierte Interrupt-Synchronisierungsmethoden zu implementieren.
- Die ISR führt alle erforderlichen Aufgaben aus, beispielsweise das Maskieren des aktuellen Interrupts, damit das Hardwaregerät keine weiteren Interrupts auslösen kann, die die aktuelle Verarbeitung beeinträchtigen, und gibt anschließend einen SYSINTR-Wert an den Exception-Handler zurück. Der SYSINTR-Wert ist eine logische Interrupt-ID (ID_x).
- Der Exception-Handler übergibt den SYSINTR-Wert an den Interrupthandler des Kernels, der dass Event für den SYSINTR-Wert bestimmt, und den Event gegebenenfalls für die ISTs des Interrupts signalisiert.
- Der Interrupthandler demaskiert alle Interrupts, außer den Interrupt, der gerade verarbeitet wird. Das Maskieren des aktuellen Interrupts verhindert, dass das aktuelle Gerät einen weiteren Interrupt verursacht, während der IST ausgeführt wird.
- Der IST wird vom signalisierten Event aktiviert, um den Interrupt zu verarbeiten ohne andere Geräte zu blockieren.
- Der IST ruft die Funktion InterruptDone() auf, um den Interrupthandler zu benachrichtigen, dass der IST die Verarbeitung abgeschlossen hat und für ein weiteres Interruptevent verfügbar ist.
- Der Interrupthandler ruft die Funktion OEMInterruptDone() im OAL auf, um die Interruptverarbeitung abzuschließen und den Interrupt erneut zu aktivieren.

**Interrupt Service Routines (ISR)**

Die ISR bestimmt die Interruptquelle, maskiert oder demaskiert den Interrupt auf dem Gerät und gibt einen SYSINTR-Wert für den Interrupt zurück. Die ISR gibt SYSINTR_NOP zurück, um anzuzeigen, dass keine weitere Verarbeitung erforderlich ist. Der Kernel signalisiert

folglich das Event für einen IST nicht, um den Interrupt zu verarbeiten. Wenn der Gerätetreiber jedoch einen IST verwendet, um den Interrupt zu verarbeiten, übergibt die ISR die logische Interrupt-ID an den Kernel.

Der Kernel bestimmt und signalisiert das Interruptevent und der IST, der nach dem Aufruf der Funktion WaitForSingleObject fortgesetzt wird, führt die Verarbeitungsanweisungen in einer Schleife aus. Die Latenz zwischen der ISR und dem IST hängt von der Priorität des Threads und den anderen ausgeführten Threads ab. ISTs werden normalerweise mit einer höheren Priorität ausgeführt.

**Interrupt Service Threads (IST)**

Ein IST ist ein normaler Thread, der bei einem Interrupt zusätzliche Verarbeitungsvorgänge ausführt, nachdem die ISR beendet wurde. Die IST-Funktion umfasst eine Schleife und einen WaitForSingleObject()-Aufruf, um den Thread unbegrenzt zu blockieren, bis der Kernel das angegebene IST-Event mit PulseEvent() signalisiert. Bevor man das IST-Event verwenden kann, muss man die Funktion InterruptInitialize() mit dem SYSINTR-Wert und einem Eventhandle als Parameter aufrufen, damit der CEKernel das Event ankündigen kann, wenn eine ISR den SYSINTR-Wert zurückgibt.

Nachdem der IST die IRQ-Verarbeitung abgeschlossen hat, muss er die Funktion InterruptDone() aufrufen, um das System darüber zu informieren, dass der Interrupt verarbeitet wurde, der IST den nächsten IRQ verarbeiten kann und der Interrupt mit der Funktion OEMInterruptDone() erneut aktiviert werden kann.

**Statische Interruptzuordnungen**

Damit die ISR einen korrekten SYSINTR-Rückgabewert bestimmen kann, muss eine Zuordnung zwischen dem IRQ und der SYSINTR vorhanden sein, die in der OAL hart-codiert sein kann. Um den IRQs auf einem Zielgerät statische SYSINTR-Werte zuzuweisen, ruft man während der Systeminitialisierung die Funktion OALIntrStaticTranslate() auf.

Statische SYSINTR-Werte und Zuordnungen sind jedoch keine übliche Methode zum Zuweisen von IRQs mit SYSINTRs, da dies schwierig ist und OAL-Codeänderungen erfordert, um die benutzerdefinierte Interruptverarbeitung zu implementieren. Statische SYSINTR-Werte werden normalerweise nur für die wichtigsten Hardwarekomponenten eines Zielgeräts verwendet, wenn kein expliziter Gerätetreiber

vorhanden ist und die ISR im OAL implementiert ist.

**Dynamische Interruptzuordnungen**

Es müssen die SYSINTR-Werte im OAL nicht hart-codiert werden, wenn man die Funktion KernelIoControl() in den Gerätetreibern mit dem IOCTL-Code IOCTL_HAL_REQUEST_SYSINTR aufruft, um die IRQ/SYSINTR-Zuordnungen zu registrieren. Der Aufruf endet mit der Funktion OALIntrRequestSysIntr, die einen neuen SYSINTR für den angegebenen IRQ dynamisch zuweist und die IRQ- und SYSINTR-Zuordnungen anschließend in den Interrupt-Zuordnungsarrays des Kernels registriert. Das Suchen eines verfügbaren SYSINTR-Werts bis zu SYSINTR_MAXIMUM ist flexibler als statische SYSINTR-Zuweisungen, da diese Methode keine OAL-Änderungen erfordert, wenn Sie neue Treiber zum BSP hinzufügen.

**Kommunikation zwischen einer ISR und IST**

Da die ISR und der IST zu verschiedenen Zeitpunkten und in verschiedenen Kontexten ausgeführt werden, müssen die physischen und virtuellen Zuordnungen beachtet werden, wenn eine ISR Daten an den IST übergibt. Eine Methode zum Übertragen der Daten ist das Reservieren eines physischen Speicherbereichs in einer .bib-Datei.

Die Datei Config.bib enthält mehrere Treiber-Beispiele. Die ISR kann die Funktion OALPAtoVA aufrufen, um die physische Adresse des reservierten Speicherbereichs in eine virtuelle Adresse umzuwandeln. Da die ISR im Kernelmodus ausgeführt wird, kann man auf den reservierten Speicher zugreifen, um Daten vom Peripheriegerät zu puffern.

Der IST ruft die Funktion MmMapIoSpace außerhalb des Kernels auf, um den physischen Speicher einer prozesspezifischen virtuellen Adresse zuzuordnen. MmMapIoSpace verwendet die Funktionen VirtualAlloc und VirtualCopy, um den physischen Speicher auf virtuellen Speicher abzubilden. VirtualAlloc und VirtualCopy können jedoch direkt aufgerufen werden, wenn mehr Kontrolle über den Adresszuordnungsprozess nötig ist.

**Installierbare ISRs**

Um die Flexibilität und Anpassbarkeit sicherzustellen, unterstützt Windows Embedded CE installierbare ISRs (IISR), um Gerätetreiber bei Bedarf in den Kernelbereich zu laden, beispielsweise wenn Plug&Play-Peripheriegeräte angeschlossen werden. Installierbare ISRs sind außerdem eine Lösung für Prozess-Interrupts, wenn mehrere

Hardwaregeräte den gleichen Interrupt verwenden.

Die ISR-Architektur hängt von DLLs ab, die den Code für die installierbare ISR enthalten und die Einsprungspunkte exportieren. Diese Funktionen sind ISRHandler, CreateInstance, DestroyInstance und IOControl. Beispiel unter „%_WINCEROOT%\PUBLIC\COMMON\OAK\DRIVERS\GIISR\giisr.c".

**Registrieren einer IISR**

Die Funktion LoadIntChainHandler() erwartet drei Parameter, die anzugeben sind, um eine installierbare ISR zu laden und zu registrieren. Der erste Parameter (lpDateiname) gibt den Dateinamen der zu ladenden ISR DLL an. Der zweite Parameter (lpFunktionsName) gibt den Namen der Interrupthandlerfunktion an, und der dritte Parameter (bIRQ) definiert den IRQ, für den die installierbare ISR registriert werden soll. Als Antwort auf eine getrennte Hardwarekomponente kann ein Gerätetreiber eine installierbare ISR über die Funktion FreeIntChainHandler() entladen.

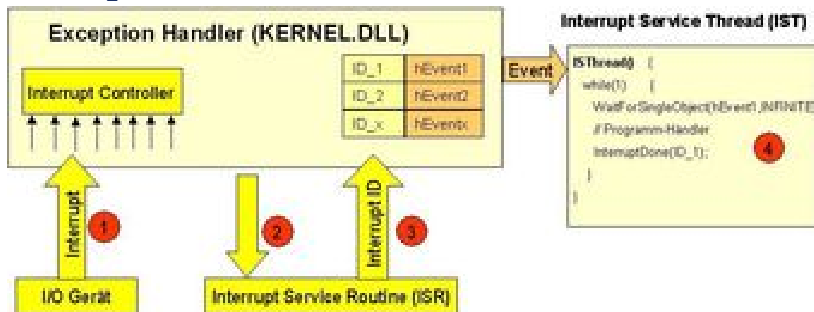
**Externe Abhängigkeiten und installierbare ISRs**

Zu beachten ist, dass LoadIntChainHandler() die ISR DLLs in den Kernelbereich lädt. Das heißt, dass die installierbare ISR keine Betriebsystem-APIs einer höheren Ebene aufrufen und andere DLLs weder importieren noch implizit linken kann. Wenn die DLL explizit oder implizit mit anderen DLLs gelinkt ist oder die C-Laufzeitbibliothek verwendet, kann die DLL nicht geladen werden. Die installierbare ISR muss vollständig unabhängig sein.

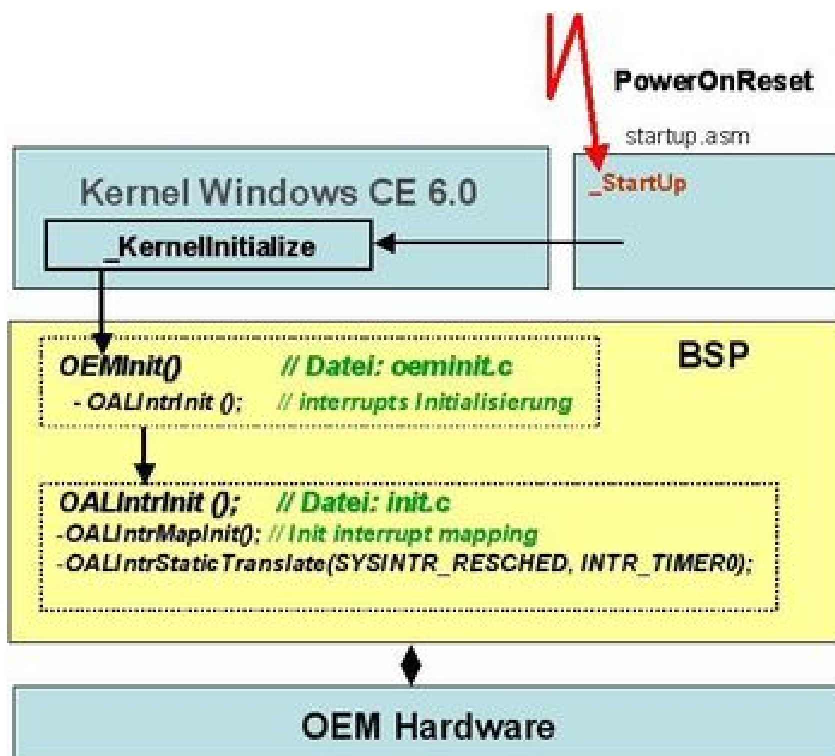
Redakteur: Martina Hafner

Die Beiträge auf dieser Website sind urheberrechtlich geschützt. Bei Fragen zu den Nutzungsrechten wenden Sie sich bitte an manuela.maure
manuela.maurer@vogel.de oder Tel.: 0931-418-2888.
Dieses PDF wurde Ihnen bereitgestellt von
http://www.elektronikpraxis.vogel.de

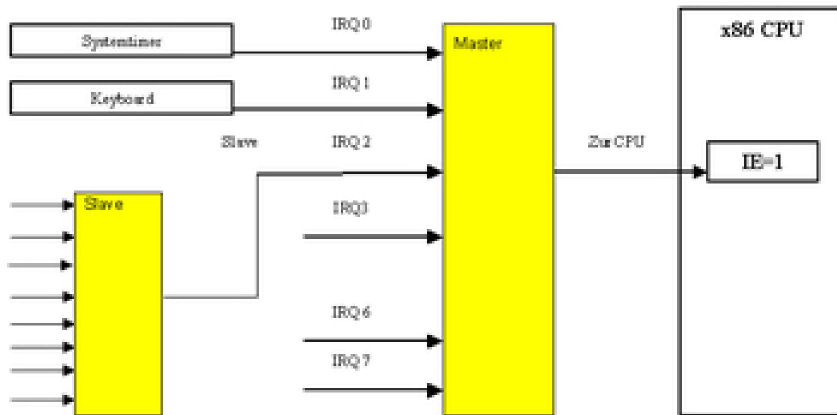
**Bildergalerie**




Wenn ein Interrupt auftritt, hält die CPU aktuelle Threads an und verwendet den Exception-Handler als primäres Ziel des Interrupts.




Schematische Darstellung der Interrupt-Verarbeitung in Windows CE

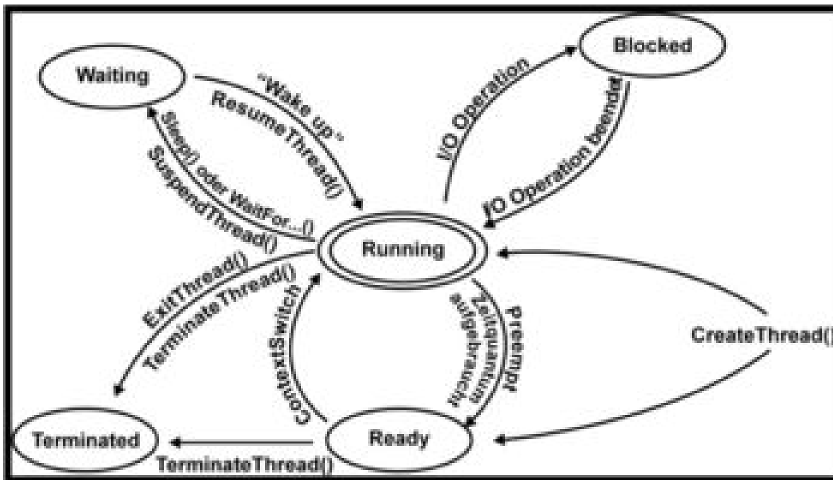



Interrupt-Controller Master und Slave.

**Bild 1**

Ein Echtzeit-System besteht aus den vier Komponenten Hardware, BSP, RTOS und Applikation.

---




Ein Thread ist eine organisatorische Einheit aus Programmcode, Speicher und Variablen und gekennzeichnet durch Priorität und Taskzustand. Bild: mögliche Taskzustände.