# 1 Unterbrechungsbehandlung

Die Ausführung und Planung von *Unterbrechungen (Interrupts)* ist eine fundamentale Aufgabe des Betriebssystems. Die hierzu verwendeten Konzepte haben großen Einfluss auf andere Subsysteme des Betriebssystems. [...]

## 1.1 Domänenanalyse

### 1.1.1 Domänenabgrenzung

Die hier untersuchte Domäne *Unterbrechungsbehandlung (IRQ-Handling)* beschränkt sich auf die *Verteilung* und *Bearbeitung* von sogenannten *Hardware-Interrupts (IRQs)*, also durch externe Geräte (wie ein E/A-Gerät oder ein Zeitgeber) signalisierte Unterbrechungen, im Kontext eines *Einprozessorsystems*. Unterbrechnungsbehandlung wird damit bewusst getrennt von der *Ausnahmebehandlung* untersucht. Dieses ist insofern ungewöhnlich, das beide Felder in den meisten Hardwarearchitekturen und (damit auch) Betriebssystemimplementierungen eng miteinander verzahnt sind. Nichtsdestotrotz handelt es sich jedoch um unterschiedliche Domänen.

### 1.1.2 Grundlegende Feststellungen

Die Behandlung einer Unterbrechungsanforderung (IRQ) kann von sowohl von der Hardwareseite (technisch) als auch von der Betriebssystemseite (konzeptionell) betrachtet werden:

- Aus technischer Sicht ist Unterbrechungsverarbeitung zunächst eine reine Hardwareangelegenheit. Erst mit der Ausführung des IRQ-Handlers kann das Betriebssystem die Kontrolle zurück erhalten. Die Priorisierung und Verteilung mehrerer simultaner IRQs (nested Interrupts) ist in den meisten Fällen ebenfalls durch die Hardware vorgegeben. Somit ist es in erster Instanz immer die Hardware, welche die (Hardware-) IRQs verteilt – im Unterschied zu z.B. Threads, die vom Scheduler des Betriebssystems verteilt werden.

- Das konzeptionelle Modell, d.h. die vom BS für Treiberentwickler angebotenen Abstraktionen, kann durchaus von der durch die Hardware vorgegebenen Struktur abweichen. Aus Effizienzgründen wird es jedoch oft hardwarenah implementiert. Im einfachsten Fall wird nur das Modell der Hardware „durchgereicht“ (OSEK). Im Extremfall wird von den Eigenschaften der Hardware vollständig abstrahiert und die Hardware-IRQs werden durch das BS auf normale Threads mit beliebiger Priorität abgebildet (L4Ka). Die meisten Betriebssysteme liegen irgendwo dazwischen, indem sie die Unterbrechungsbehandlung in eine frühe Phase, hier *Prolog* genannt, und eine späte Phase, hier als *Epilog* bezeichnet, aufteilen. Die Verteilung und Aktivierung der Prologe ist üblicherweise durch die Hardware gesteuert, während die Epiloge durch das BS verzögert verteilt und aktiviert werden.

Die Abstraktionen des Betriebssystems sind hier nach folgenden Dimensionen kategorisiert:

**handler modell** beschreibt, wie (d.h. mit welchen BS-Abstraktionen) ein Treiberentwickler Programmcode zur Ausführung bei einer Unterbrechung registriert. Hier gibt es zwei Modelle:

**pro-/epilog** Viele Betriebssysteme teilen die Bearbeitung einer Unterbrechungsanforderung in zwei Phasen auf. Die erste Phase abstrahiert nur geringfügig von der Hardware. Sie ist hochprior, da sie z.B. der von der Hardware vorgegebenen Verteilung unterliegt. Die eigentliche (funktionale) Bearbeitung der Unterbrechung erfolgt jedoch in einer zweiten, konzeptionell geringerprioren Phase unter der Kontrolle des Betriebssystems. Für diese zwei Phasen gibt es die unterschiedlichsten Bezeichnungen: Im klassischen UNIX wurden sie als *Bottom-Half (BH)* und *Top-Half (TH)* des Kerns bezeichnet. Diese Bezeichnungen werden auch in Linux verwendet, allerdings mit eher umgedrehter Bedeutung (siehe Abbildung 1.1). In Windows NT heißen sie *Interrupt-Level* und *DPC/Dispatch-Level*, in Pure *Prolog* und *Epilog*. Letztere Bezeichnungen spiegeln wohl am besten die Bedeutung dieser Phasen wieder und werden deshalb hier verwendet.

**IRQ-threads** Der zweite häufig anzutreffende Ansatz ist die Abbildung der Unterbrechungsanforderung auf Threads. Das Eintreffen einer IRQ führt zum „Wecken“ eines (üblicherweise hochprioren) Threads, der dann über den Scheduler aktiviert wird und den Code zur Behandlung der Unterbrechung ausführt. Die Priorisierung und Verteilung von IRQs ist somit in die normale Threadverteilung integriert.

**IRQ scheduling** beschreibt, wann und nach welchen Kriterien die Verteilung des registrierten Handlers erfolgt. Beim Prolog/Epilog-Modell bezieht sich dieses auf die Aktivierung des Prologs, beim Thread-Modell auf die Aktivierung des entsprechenden Threads.

**execution model** beschreibt die Ablaufumgebung und ihre Randbedingungen für Interruptcode. Dieses umfasst die Menge der zur Verfügung stehenden Systemaufrufe, die Frage wann und wodurch eine bestimmte Unterbrechung wieder freigegeben wird oder unter welchen Umständen der Scheduler des Betriebssystems aufgerufen wird. Beim Prolog/Epilog-Modell können sich diese Randbedingungen für die verschiedenen Phasen unterscheiden.

### 1.1.3 Unterbrechungsbehandlung in verschiedenen Systemen

#### 1.1.3.1 Linux

Linux verwendet ein Prolog/Epilog-Modell. Die Prologe heißen *Top-Halves*, die Epiloge *Bottom-Halves*, oder, ab Kernversion 1.4 auch *Tasklets*[1]. Die Prologabarbeitung ist nicht priorisiert, Unterbrechungsanforderungen werden daher in LCFS-Reihenfolge abgearbeitet. Prologe aktivieren die zugehörigen Epiloge explizit[2], die Abarbeitung der Epiloge erfolgt in jedem Fall verzögert an wohldefinierten Stellen, wie z.B. nach Abarbeitung sämtlicher Prologe oder nach Verdrängung eines Threads durch den Scheduler. Sowohl in der Prolog- als auch Epilogphase dürfen alle nicht-blockierende Systemaufrufe verwendet werden. Sämtliche dort verwendeten Datenstrukturen müssen folglich durch IRQ-Sperren geschützt werden. Die Abarbeitung des Interruptcodes erfolgt synchron und im Kontext des unterbrochenen Prozesses.

#### 1.1.3.2 Solaris

Solaris bildet IRQs auf hochpriore Threads ab (IRQ-Threads). Ein IRQ-Thread läuft im Kern. Seine Priorität ist abhängig von der Unterbrechungsebene (IRQL) des HW-IRQ, jedoch in jedem Fall höher als die aller normalen Threads. Dank dieser Bedingung lassen sich IRQ-Threads zunächst rein hardwarebasiert verteilen ohne den Scheduler des BS zu verwenden. Technisch ist ein IRQ-Thread daher zunächst nur ein Pseudothread, der synchron zum unterbrochenen Thread abgearbeitet wird. Erst wenn ein verwendeter Systemaufruf blockieren muss (z.B. weil ein anderer Thread eine benötigte Ressource gesperrt hält), wird der IRQ-Thread zu einem echten Thread und damit zu einer Entität der normalen Threadverteilung. Durch das in Solaris implementierte Pioritätsvererbungsprotokoll erbt der ressourcebesitzende Thread in diesem Fall die Priorität des IRQ-Threads. Der prozessorinterne IRQL bleibt bis zur vollständigen Abarbeitung der Unterbrechungsbehandlung auf seinem Niveau.

[1] Verglichen zu den älteren Bottom-Halves sind die neueren Tasklets flexibler in der Handhabung und auf SMP-Systemen parallel ausführbar, was hier jedoch ohne Belang ist.

[2] Linux verwendet dazu sogenannte *Software-IRQs*, die konzeptionell einem in Software implementierten AST entsprechen.

Die Unterteilung des Kerns in eine „untere Hälfte“ (*Bottom-Half*) und eine „obere Hälfte“ (*Top-Half*) geht auf das klassische UNIX-Modell zurück. Bei UNIX ist die *Bottom-Half* der Teil des Kerns, in dem die Unterbrechungsbearbeitung stattfindet (IRQ-Handler aller Treiber), während die *Top-Half* den Code des Kerns beinhaltet, der von den Benutzerprozessen betreten und ausgeführt wird. Die beiden Hälften sind formal strikt getrennt, die Interaktion zwischen beiden Hälften findet z.B. über Softwareunterbrechungen statt. Die Bezeichnung der Hälften als „obere“ und „untere“ entspricht dem Gedanken eines Schichtenmodells, bei dem die Hardware zu unterst liegt, darauf aufsetzend in zunehmenden Abstraktionsstufen die verschiedenen Schichten des Betriebssystems, deren oberste schließlich die Schnittstelle zu den Benutzerprogrammen darstellt.
Linux, als Beispiel für ein modernes UNIX, hat die Bezeichnungen Top-Half und Bottom-Half übernommen, verwendet sie jedoch unglücklicherweise auf *völlig andere Weise*. Unter Linux bezeichnen sie die beiden Ausführungshälften *eines* Treibers, weshalb man (bezogen auf das Gesamtsystem) auch genauer von mehreren *Top-Halves* und *Bottom-Halves* sprechen muss. Verwirrend ist dabei vor allem, dass bei Linux die Top-Halves „unten“ stehen, sie bezeichnen hier den Teil des Treibers, der als IRQ-Handler von der Hardware aktiviert wird. Die Top-Half eines Treibers erledigt die zeitkritischen und wichtigen Aufgaben; die eigentliche Bearbeitung wird an die zugehörige Bottom-Half delegiert. Alle Bottom-Halves werden dann zu einem späteren Zeitpunkt durch den Kern aktiviert und sind jederzeit durch neue IRQs bzw. Top-Halves unterbrechbar. Sie stehen damit bezogen auf das Schichtmodell weiter oben, jedoch immer noch unterhalb der normalen Prozessausführung. Somit bilden die Menge aller Linux-Top-Halves und Linux-Bottom-Halves *zusammen* eine Einheit, die in UNIX als *die* Bottom-Half bezeichnet würde.

Abbildung 1.1: Richtungsfragen: Wo ist oben, wo ist unten?

#### 1.1.3.3 OSEK/VDX

OSEK unterscheidet zwei verschiedene Typen von *Interrupt Service Routine (ISR)*. Der Typ 1 (*ISR-1*) ist sehr hardwarenah, ihm steht nur ein stark eingeschränkter Satz an Systemfunktionen zur IRQ-Manipulation zur Verfügung. Dem Typ 2 (*ISR-2*) ist es erlaubt nahezu alle nichtblockierenden Systemfunktionen zu verwenden. Eine optionale Erweiterung (*erweiterte ISR-2*) ermöglicht zudem, Resourcen gemeinsam zwischen Threads und IRQs zu verwenden. Für die Priorisierung von Ressourcenzugriffen kommt unter OSEK generell das *Priority Ceiling Protocol (PCP)* zum Einsatz, falls ISR-2 daran beteiligt sind, so erhalten sie virtuelle Thread-Prioritäten, die über allen anderen Thread-Prioritäten aber unterhalb der Ceiling-Priorität liegen. Nach Beendigung einer ISR-1 wird der unterbrochene Kontrollfluss fortgesetzt. Nach der Rückkehr aus allen ISRs wird, falls die zuletzt aktive ISR eine ISR-2 war, vom System implizt der Scheduler aufgerufen. Die Verteilung der ISR selber ist hardwareabhängig und nicht spezifiziert.

ISR-1 und ISR-2 unterliegen sehr unterschiedlichen Einschränkungen hinsichtlich ihrer möglichen Interaktionen mit dem System. Eine ISR-1 wird hier als ein Prolog ohne nachfolgenden Epilog aufgefasst werden, eine ISR-2 entsprechend als Epilog ohne benutzerdefinierten Prolog. Frühere Versionen von OSEK sahen noch einen Typ 3 vor, der als ISR-1 beginnt und dann ex-

plizit seinen Kontext auf ISR-2 erweitern konnte[3].

#### 1.1.3.4 Pure

Pure verwendet ein einfaches Prolog/Epilog-Modell. Prologe werden synchron zum aktuellen Kontrollfluss abgearbeitet, die Verteilung der Prologe erfolgt durch die Hardware. Im Prologkontext sind nur wenige nicht-blockierende Systemaufrufe erlaubt. Die meisten internen Datenstrukturen sind durch einen globalen Lock (*Guard*) geschützt, der im Epilogkontext implizit gesperrt ist. Damit sind im Epilogkontext alle nicht-blockierenden Systemaufrufe erlaubt. Epiloge werden synchron zum Prolog abgearbeitet, falls der Guard verfügbar ist. Falls der Guard nicht verfügbar ist, wird ihre Abarbeitung bis zur Freigabe durch den besitzenden Thread oder Interrupt verzögert (asynchron).

#### 1.1.3.5 L4Ka

L4Ka verfolgt das Thread-Modell am konsequentesten, indem es jede IRQ umsetzt auf einen IPC zu einem wartenden IRQ-Thread in einem (nicht-privilegierten) Treiberprozess. Im Anschluss daran erfolgt die Aktivierung des Schedulers, welcher dann den geweckten IRQ-Thread, so er die höchste Priorität hat, zur Ausführung bringt. Die Verteilung der IRQs ist somit vollständig in die normale Threadverteilung integriert. Dem IRQ-Thread obliegt auch sämtliche Interaktion mit dem Gerät, z.B. das Sichern des Hardwarezustands. Die entsprechenden Speicher-/Registerbereiche müssen dazu Teil seines Addressraums sein. Indem sich der IRQ-Thread erneut für den betreffenden IRQ registriert, wird die Bearbeitung der IRQ quittiert und damit für die nächste Unterbrechung freigegeben.

### 1.1.4 Analysemodell

## 1.2 Entwurf eines konfigurierbare Modells für die Unterbrechungsbehandlung

### 1.2.1 Zielbestimmung

Das Hauptziel des hier beschriebene Modells für eine konfigurierbare Unterbrechungsbehandlung ist die Bereitstellung eines einheitlichen Treibermodells mit dem ein- und derselbe Trei-

[3] Der Typ 3 hat eine bewegte Geschichte hinter sich. Von OSEK/VDX 1.0 bis zum aktuellen OSEK/VDX2.2.1 hat wurde die Unterstützung für ISR-3 zunächst als optional (1.0), dann als zwingend (2.0r1), wieder als optional (2.1r1) und schließlich als gestrichen (2.2.1) beschrieben.

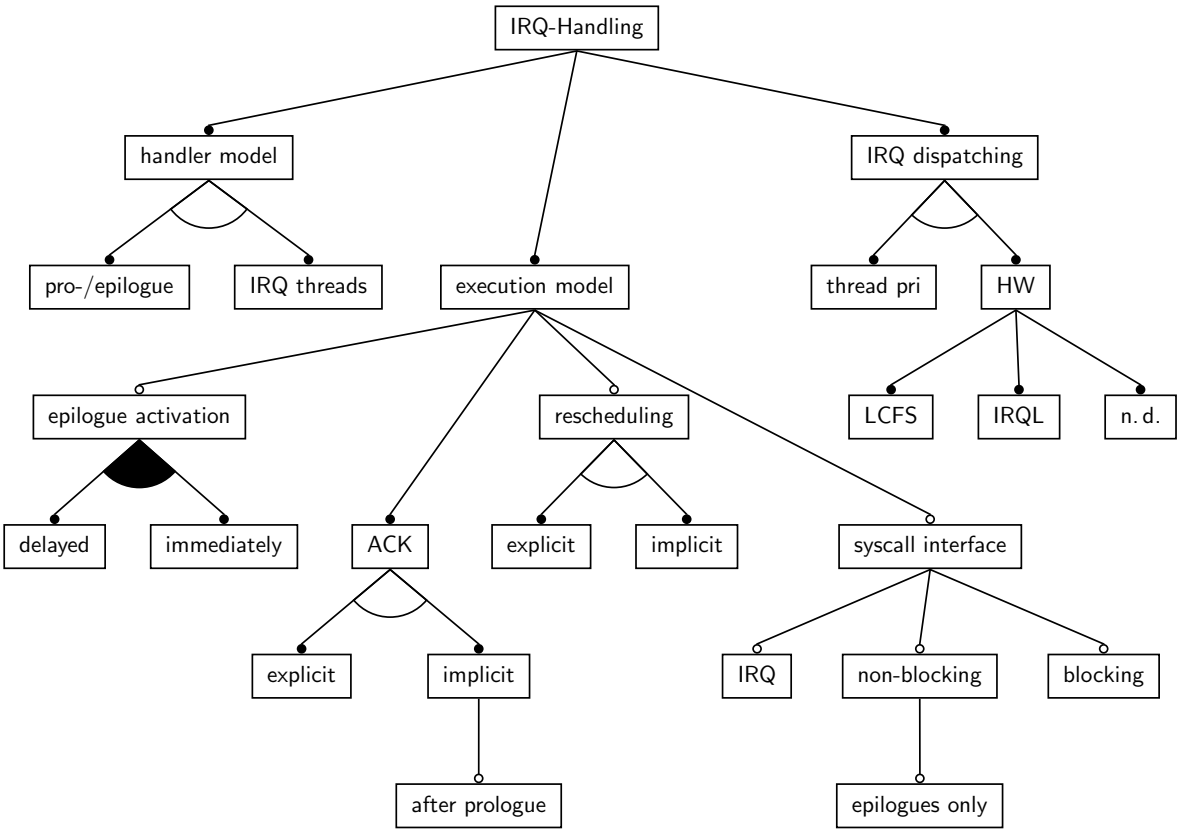


Abbildung 1.2: Domänenmodell Unterbrechungsbehandlung

| Feature | Description | Linux | Solaris | Win NT | OSEK | Pure | L4Ka |
|---|---|---|---|---|---|---|---|
| **1. handler model** | | | | | | | |
| **1.1. pro-/epilogue** | IRQ processing by prologues and delayed epilogues | x | | x | x | x | |
| **1.2. IRQ-threads** | IRQs are mapped to thread activations | | x | | | | x |
| **2. dispatching** | | | | | | | |
| **2.1. thread pri** | handler interrupts active control flow, iff pri(IRQ-Thread) > pri(running thread) | | | | | | x |
| **2.2. HW** | | | | | | | |
| **2.2.1. LCFS** | handler interrupts active control flow immediately | x | | | | | |
| **2.2.2. IRQL** | handler interrupts active control flow, iff pri(IRQ) > IRQL (HW priorities) | | x | x | | | |
| **2.2.3. n.d.** | IRQ scheduling is undefined (depends on actual hardware) | | | | x | x | |
| **3. execution model** | | | | | | | |
| **3.1. epilogue activation** | | | | | | | |
| **3.1.1. delayed** | asynchronous execution of epilog (AST) | x | | x | | x | |
| **3.1.2. immediately** | synchronous execution of epilog (direct) | | | | x | X | |
| **3.2. ACK** | | | | | | | |
| **3.2.1. explicit** | IRQ-line is re-enabled explicitly | | | | | | |
| **3.2.2. implicit** | IRQ-line is re-enabled implicitly (after handler termination) | | x | | x | | x |
| **3.2.2.1. after prologue** | IRQ-line is re-enabled implicitly before entering the epilogue | x | | x | | x | |
| **3.3. rescheduling** | | | | | | | |
| **3.3.1. explicit** | rescheduling after handler termination has to be requested explicitly | | | | | x | |
| **3.3.2. implicit** | rescheduling takes place after handler termination | x | x | x | x (ISR2) | | x |
| **3.4. syscall interface** | | | | | | | |
| **3.4.1. IRQ** | Few low-level syscalls only (e.g. only enable/disable IRQs) | x | x | X | x (ISR1) | x | x |
| **3.4.2. non-blocking calls** | (almost) all non-blocking syscalls | x | x | x | x (ISR2) | x | x |
| **3.4.2.1. epilogues only** | non-blocking syscalls permitted in epilogue phase only | | | | x | x | |
| **3.4.3. blocking calls** | (almost) all blocking syscalls | | x | | | | x |

Tabelle 1.1: Merkmalszuordnung der untersuchten Systeme

Als *nicht-funktionale Eigenschaften* einer Software bezeichnet man diejenigen Eigenschaften, die in keinen unmittelbaren Einfluss auf die eigentliche (Haupt-)aufgabe oder Funktion der Software haben. So ist z.B. die Funktion eines Sortieralgorithmus dadurch spezifiziert, dass er eine Menge von Daten in eine wohldefinierte Ordnung bringt. Laufzeit und Speicherbedarf des Algorithmus sind hingegen nicht-funktionale Eigenschaften; sie haben zwar möglicherweise erheblichen Einfluss auf die praktische Verwendbarkeit, nicht jedoch auf die Korrektheit (Erfüllung der Spezifikation) des Algorithmus. Nicht-funktionale Eigenschaften sind im allgemeinen „schwer fassbar“, weil holistisch und emergent. Sie ergeben sich oft erst aus dem Zusammenspiel aller Einzelteile eines Systems. Dementsprechend schwierig ist es, sie konfigurierbar zu gestalten.
Die *Architektur* eines Betriebssystems beschreibt ebenfalls eine Menge nicht-funktionaler Eigenschaften. Ob das System nun grob- oder feingranular synchronisiert, monolithisch oder modular, prodedur- oder prozessorientiert aufgebaut ist hat keinen direkten Einfluss auf die Funktionalität der Anwendung. Letztlich stellt ein Betriebssystem eine virtuelle Machine dar, eine „black-box“, die nach außen hin eine bestimmte Spezifikation erfüllt, unabhängig von dessen Architektur und der Implementierung im Inneren. Ist die Architektur eines Systems damit beliebig? Aus funktionaler Sicht ist sie das tatsächlich, nicht jedoch bezüglich der nicht-funktionale Eigenschaften. In der Praxis sind gerade die Anforderungen an nicht-funktionale Eigenschaften, gepaart mit einem hohen Maß an Erfahrung, die Triebfeder für die Entscheidung zugunsten einer bestimmten Architektur. Monolithische, prozedurorientierte Kerne sind nun mal erfahrungsgemäß genügsamer bezüglich der Systemressourcen, prozessorientierte Mikrokerne bieten erfahrungsgemäß ein höheres Maß an Schutz usw.
Letztlich geht es bei dem Wunsch nach einer konfigurierbaren Architektur also darum, nicht-funktionale Eigenschaften *indirekt* eben doch beeinflussbar und modifizierbar zu gestalten.

Abbildung 1.3: Nicht-funktionale Eigenschaften und Architektur

bercode in verschiedenen Architekturen eingesetzt werden kann. Es geht dabei um die klare Trennung der *funktionalen Anteile* eines Gerätetreibers (bzw. dessen IRQ-Handler), die seine eigentliche Aufgabe beschreiben, von den *nicht-funktionalen Anteilen*, die durch Hardware und Architektur determiniert sind. Durch diese Trennung sollte es letztlich möglich sein, die architekturbedingten Anteile *konfigurierbar* zu gestalten.

### 1.2.2 Vorüberlegungen

Betrachtet man das Modell der Unterbrechungsbehandlung in existierenden Systemen (Abbildung 1.2, Tabelle 1.1), so wird deutlich, dass die Menge der architekturbedingten (nicht-funktionalen) Merkmale deutlich überwiegen. Weder das verwendete *handler model*, die Art des *dispatching*, noch der überwiegende Teile des *execution model* haben einen unmittelbaren Einfluss auf die mögliche und tatsächliche *Funktionalität* eines IRQ-Handler. Einzig die unter *syscall interface* grob beschriebene Menge der zur Verfügung stehenden BS-Abstraktionen hat einen direkten Einfluss auf die erreichbare Funktionalität. Sie bestimmt die möglichen Interaktionen mit dem System, syntaktisch ausgedrückt über die Menge der verwendbaren Systemaufrufe.

**Feststellung 1:** Die Menge der verfügbaren Systemaufrufe ist ein geeignetes Maß für die maximal mögliche Funktionalität eines Unterbrechungsbehandlers.

**Feststellung 2:** In der Domäne *Unterbrechungsbehandlung* ist der überwiegende Teil der identifizierten Merkmale von nicht-funktionalem Charakter.

Indirekt kann jedoch ein Zusammenhang zwischen nicht-funktionalen Merkmalen und der Menge der zur Verfügung stehenden Systemaufrufe vermutet werden. So fällt auf, dass die auf Threads basierenden Modelle von Solaris und L4 dem Treiberentwickler die meisten Freiheiten bieten (hier sind sogar blockierende Interaktionen mit dem System erlaubt), während die Systeme mit näher an der Hardware aufgezogenen Modellen deutlich weniger erlauben. Die tatsächlichen Auswirkungen derartiger Einschränkungen ergeben sich jedoch nicht alleine aus dem IRQ-Modell, sondern aus der *Kombination der Architektureigenschaften der verschiedenen Systemkomponenten*. So ist es beispielsweise völlig unerheblich, ob ein mit dem Scheduler interagierender Treiber blockierende Funktionen aufrufen darf oder nicht, wenn die Funktionen des Schedulers nicht blockieren.

**Feststellung 3:** Ein bestimmtes Modell für Unterbrechungsbehandlung ist nicht per se funktional mächtiger als ein anderes. Die funktionale Mächtigkeit ergiebt sich erst aus dem Gesamtsystems.

Das einheitliche Treibermodell soll deshalb kein „Minimalmodell“ sein, sondern vielmehr keinerlei funktionale Einschränkungen für den Treiberentwickler mit sich bringen. Vielmehr soll es möglich sein, anhand der Konfiguration des Gesamtsystems und der tatsächlich verwendeten Systemaufrufe zu entscheiden, welche Architektureigenschaften die Unterbrechungsbehandlung erfüllen muss.