Echtzeitbetriebssysteme

OSEK

Hintergrund

- Gemeinschaftsprojekt der deutschen Automobilindustrie (u.a. BMW, DaimlerChrysler, VW, Opel, Bosch, Siemens)
- OSEK: **O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug
- Ziel: Definition einer Standard-API für Echtzeitbetriebssysteme
- Standard ist frei verfügbar (http://www.osek-vdx.org), aber keine freien Implementierungen.
- Es existieren ebenso Ansätze für ein zeitgesteuertes Betriebs-system (OSEKTime), sowie eine fehlertolerante Kommunikations-schicht.

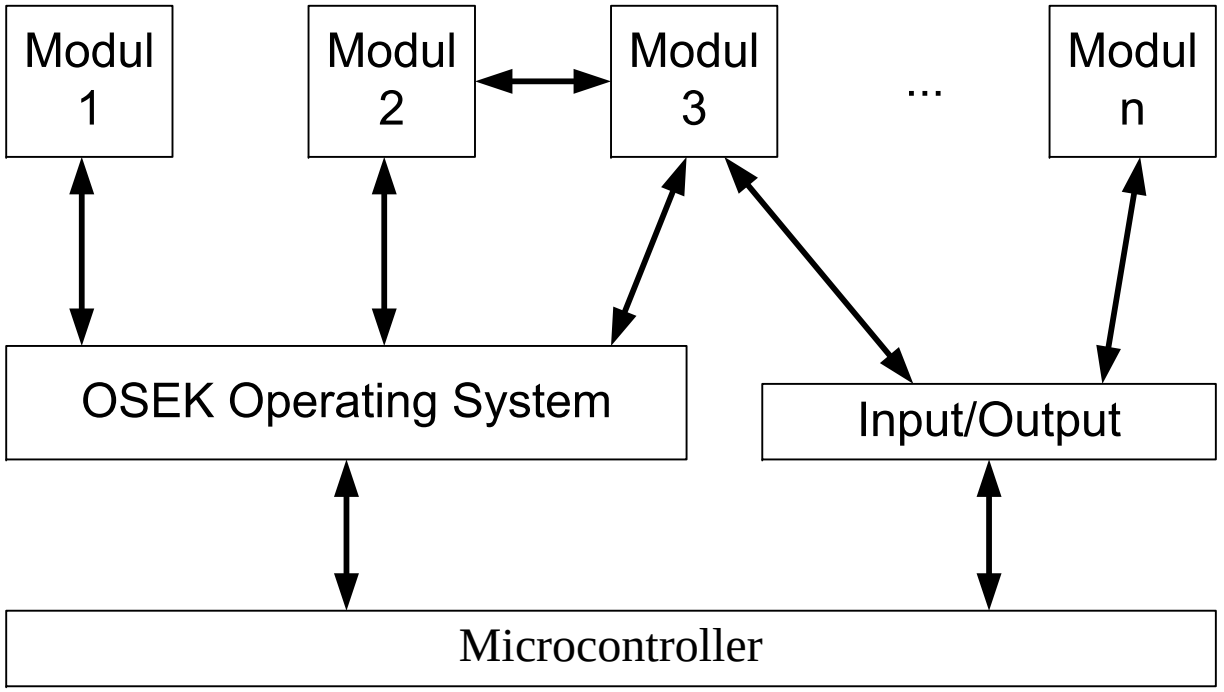
Anforderungen

- Designrichtlinien bei der Entwicklung von OSEK:
  - harte Echtzeitanforderungen
  - hohe Sicherheitsanforderungen an Anwendungen
  - hohe Anforderungen an die Leistungsfähigkeit
  - typische: verteilte Systeme mit unterschiedlicher Hardware (v.a. Prozessoren)

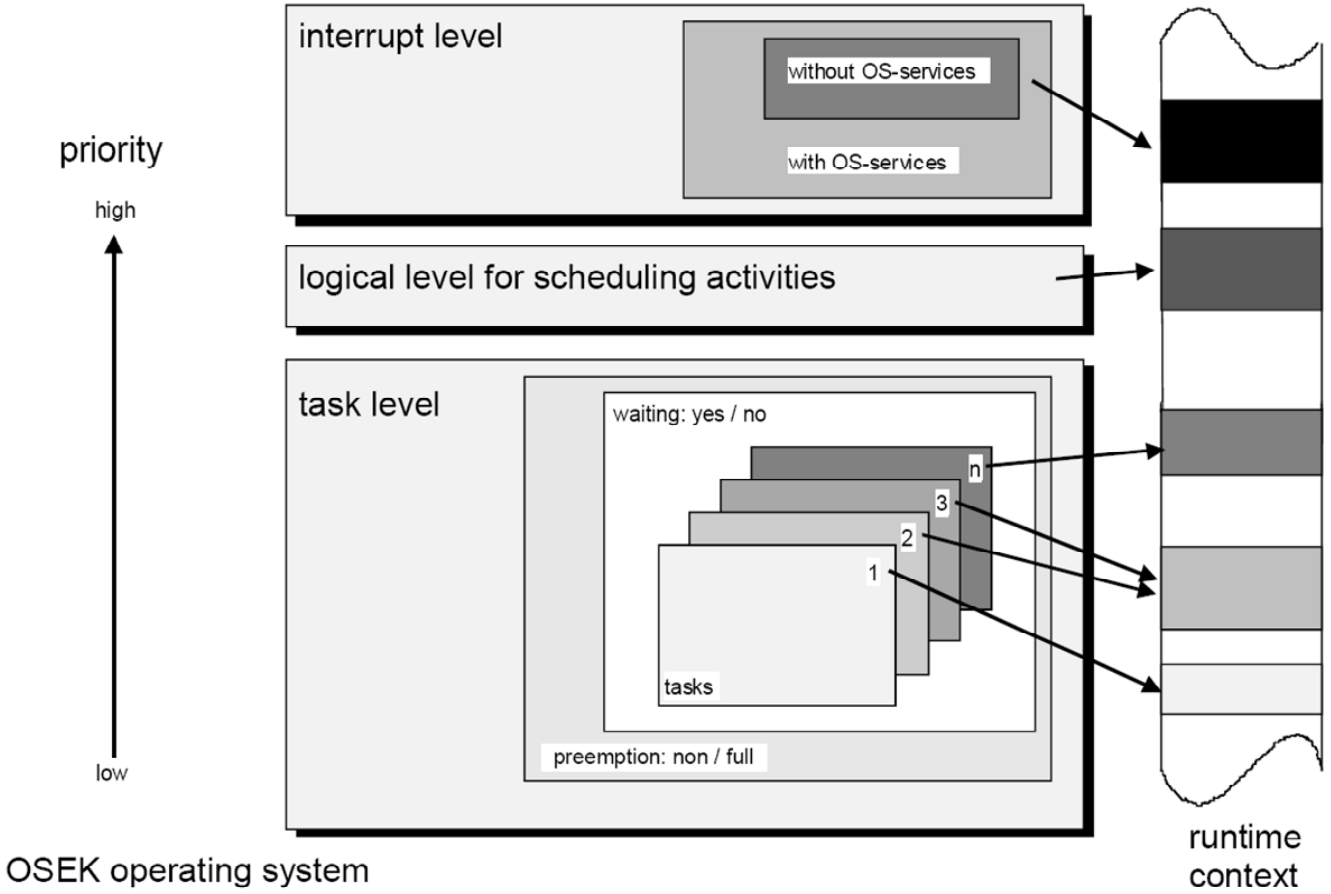
  $\Rightarrow$ typische Anforderungen von Echtzeitsystemen
- Weitere Ziele:
  - Skalierbarkeit
  - einfache Konfigurierbarkeit des Betriebssystems
  - Portabilität der Software
  - Statisch allokiertes Betriebssystem

OSEK Architektur

- Die Schnittstelle zwischen den einzelnen Anwendungsmodulen ist zur Erhöhung der Portierbarkeit standardisiert. Die Ein- und Ausgabe ist ausgelagert und wird nicht näher spezifiziert.



Ausführungsebenen in OSEK



Scheduling und Prozesse in OSEK

- Scheduling:
  - ausschließlich Scheduling mit statischen Prioritäten.
- Prozesse:
  - OSEK unterscheidet zwei verschiedene Arten von Prozessen:
    1. Basisprozesse
    2. Erweiterte Prozesse: haben die Möglichkeit über einen Aufruf der Betriebssystemfunktion waitEvent() auf externe asynchrone Ereignisse zu warten und reagieren.
  - Der Entwickler kann festlegen, ob ein Prozess unterbrechbar oder nicht unterbrechbar ist.
  - Es existieren somit vier Prozesszustände in OSEK: running, ready, waiting, suspended.

Betriebssystemklassen

- Der OSEK-Standard unterscheidet vier unterschiedliche Klassen von Betriebsystemen. Die Klassifizierung erfolgt dabei nach der Unterstützung:
  1. von mehrmaligen Prozessaktivierungen (einmalig oder mehrfach erlaubt)
  2. von Prozesstypen (nur Basisprozesse oder auch erweiterte Prozesse)
  3. mehreren Prozessen der selben Priorität
- Klassen:
  - BCC1: nur einmalig aktivierte Basisprozesse unterschiedlicher Priorität werden unterstützt.
  - BCC2: wie BCC1, allerdings Unterstützung von mehrmalig aufgerufenen Basisprozessen, sowie mehreren Basisprozessen gleicher Priorität.
  - ECC1: wie BCC1, allerdings auch Unterstützung von erweiterten Prozessen
  - ECC2: wie ECC1, allerdings Unterstützung von mehrmalig aufgerufenen Prozessen, sowie mehreren Prozessen gleicher Priorität.
- Die Implementierung unterscheidet sich vor allem in Bezug auf den Scheduler.

Unterbrechungsbehandlung

- In OSEK wird zwischen zwei Arten von Unterbrechungsbehandlern unterschieden:
  - ISR Kategorie 1: Der Behandler benutzt keine Betriebssystemfunktionen.
    - typischerweise die schnellsten und höchstpriorisierten Unterbrechungen.
    - Im Anschluss der Behandlung wird der unterbrochene Prozess fortgesetzt.
  - ISR Kategorie 2: Die Behandlungsroutine wird durch das Betriebssystem unterstützt, dadurch sind Aufrufe von Betriebssystemfunktionen erlaubt.
    - Falls ein Prozess unterbrochen wurde, wählt der Scheduler nach Beendigung der ISR den nächsten auszuführenden Prozess.

Prioritätsinversion

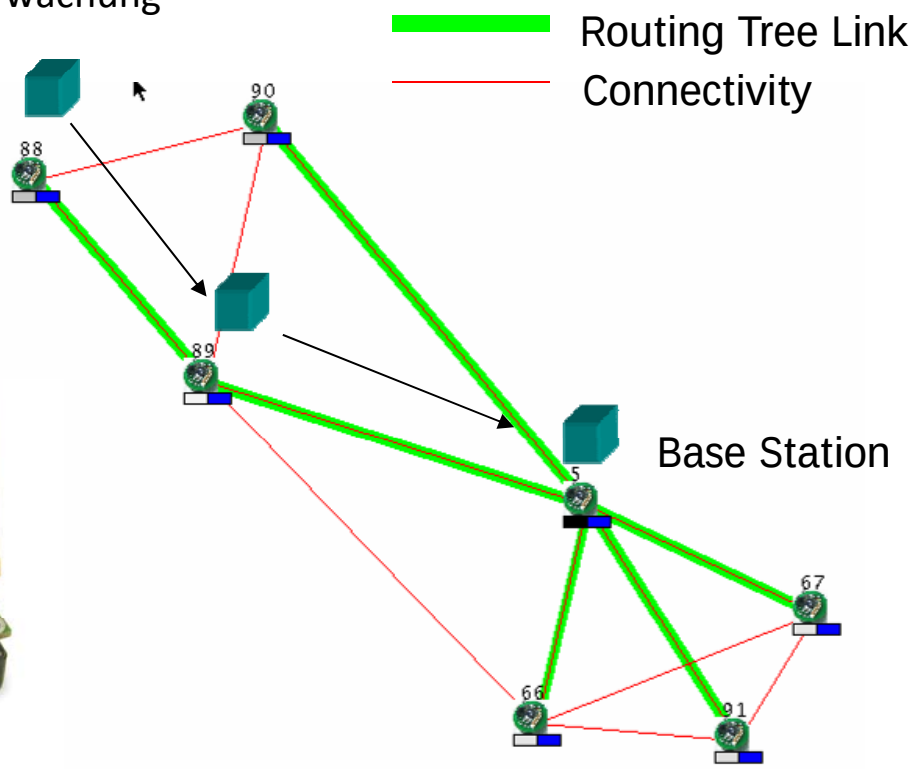
- Zur Vermeidung von Prioritätsinversion und Verklemmungen schreibt OSEK ein Immediate Priority Ceiling Protokoll vor:
  - Jeder Ressource wird eine Grenze (Maximum der Priorität der Prozesse, die die Ressource verwenden) zugewiesen.
  - Falls ein Prozess eine Ressource anfordert, wird die aktuelle Priorität des Prozesses auf die entsprechende Grenze angehoben.
  - Bei Freigabe fällt der Prozess auf die ursprüngliche Priorität zurück.

Echtzeitbetriebssysteme

TinyOS

Einsatzgebiet: AdHoc-Sensornetzwerke

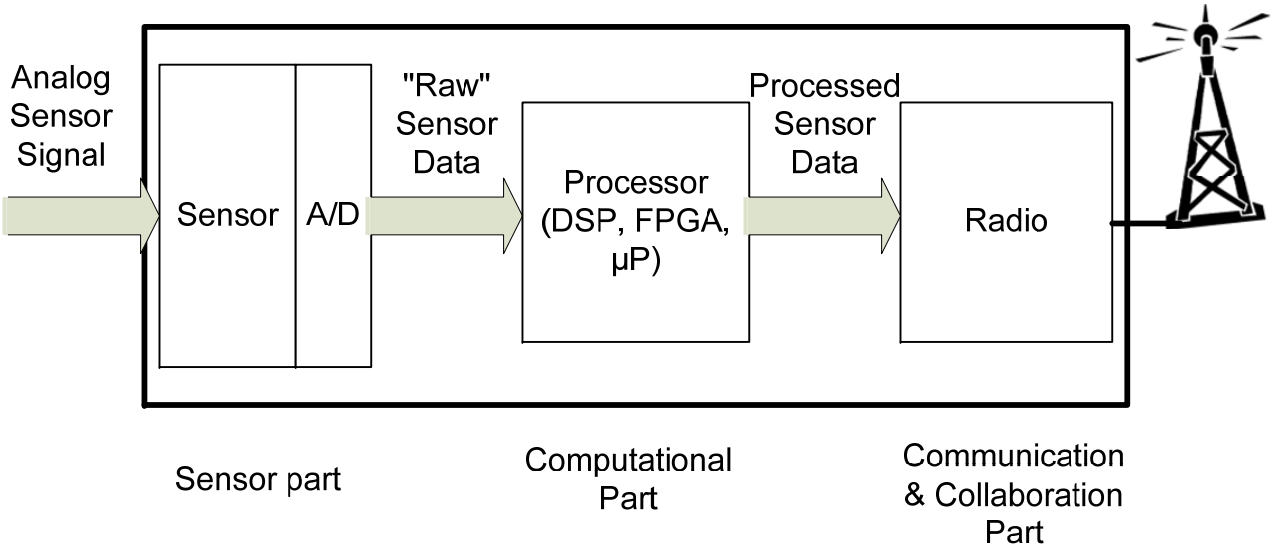
- Begriff Smart-Dust: Viele kleine Sensoren überwachen die Umgebung
- Ziele: robuste und flächendeckende Überwachung
- Probleme:
  - eingeschränkte Lebensdauer (Batterie)
  - eingeschränkter Speicherplatz
  - geringe Kommunikationsbandbreite
  - geringe Rechenleistung



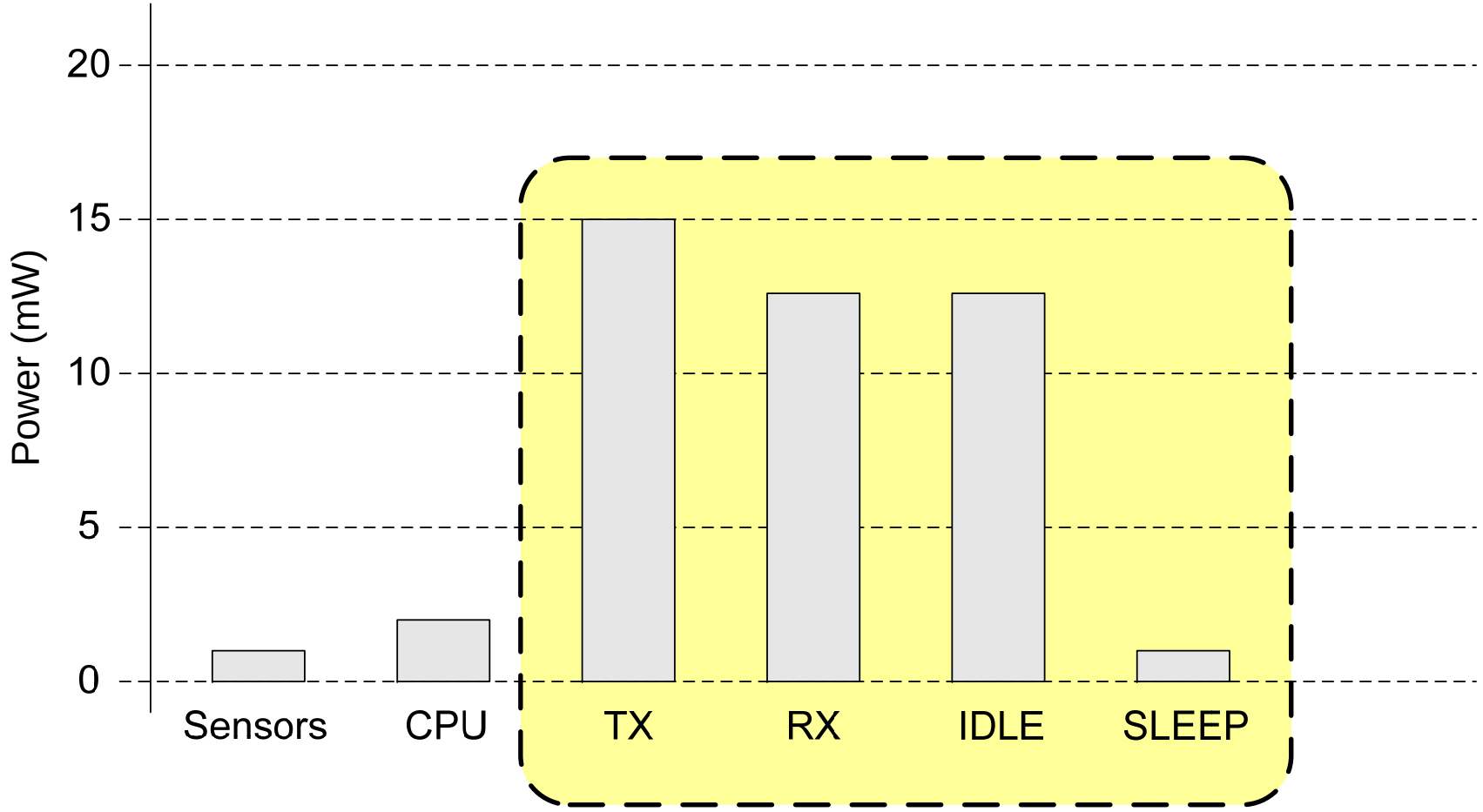
Quelle: http://tinyos.millennium.berkeley.edu

Hardware

- CPU: 4MHz, 8Bit, 512 Byte Ram
- Flash-Speicher: 128 kByte
- Funkmodul: 2,4 GHz, 250 kbps
- Diverse Sensormodule: z.B. Digital/Analog, Licht, Feuchtigkeit, Druck

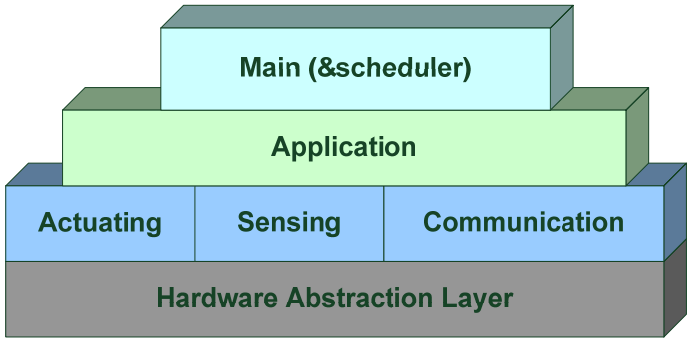


Stromverbrauch

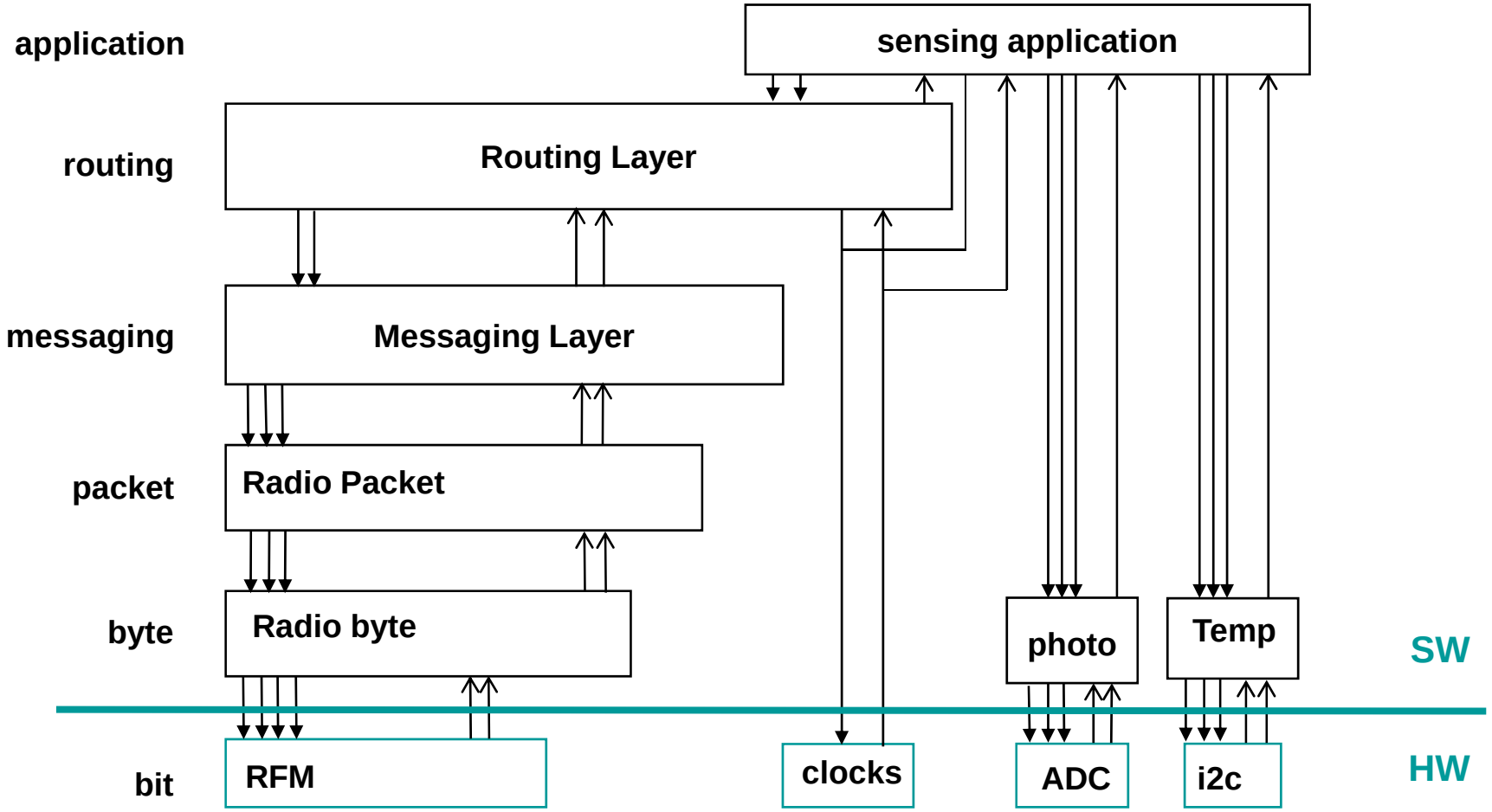


TinyOS

- TinyOS ist kein wirkliches Betriebssystem im traditionellen Sinn, eher ein anwendungs-spezifisches Betriebssystem
  - keine Trennung der Anwendung vom OS ⇒ Bei Änderung der Anwendung muss komplettes Betriebssystem neu geladen werden.
  - kein Kernel, keine Prozesse, keine Speicherverwaltung
  - Es existiert nur ein Stack (single shared stack)
- Ereignisbasiertes Ausführungsmodell
- Nebenläufigkeitskonzept:
  - Aufgaben können in unterschiedlichen Kontext ausgeführt werden:
  - Vordergrund: Unterbrechungsereignisse
  - Hintergrund: Tasks
  - Prozesse können durch Ereignisse, nicht jedoch durch andere Prozesse unterbrochen werden.
  - Scheduling für Tasks: Fifo
- Implementierung erfolgt in NesC (Erweiterung von C)
- Statische Speicherallokation



TinyOS - Architektur



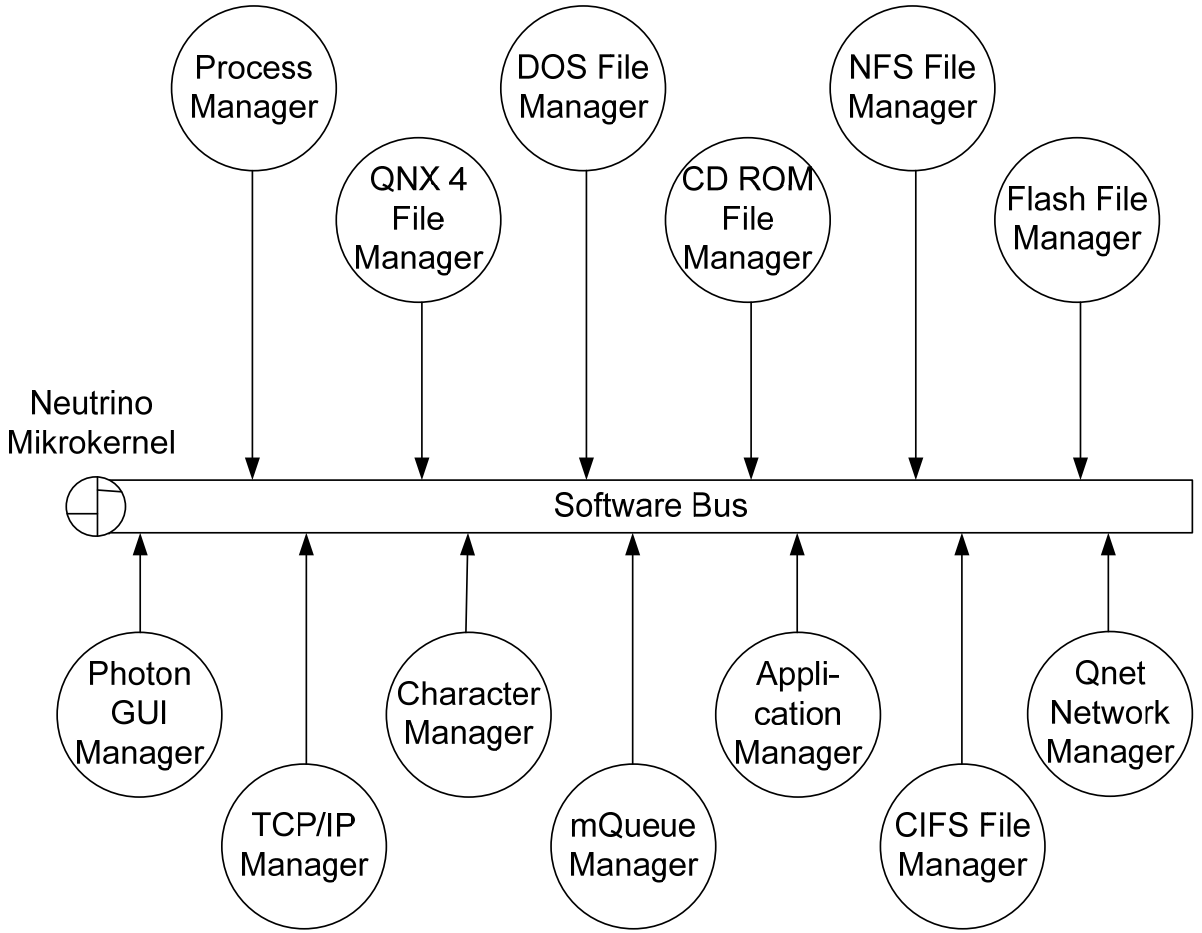
Echtzeitbetriebssysteme

QNX

Einführung

- Geschichte:
  - 1980 entwickeln Gordon Bell und Dan Dodge ein eigenes Echtzeitbetriebssystem mit Mikrokernel.
  - QNX orientiert sich nicht an Desktopsystemen und breitet sich sehr schnell auf dem Markt der eingebetteten Systeme aus.
  - Ende der 90er wird der Kernel noch einmal komplett umgeschrieben, um den POSIX-Standard zu erfüllen. ⇒ Ergebnis: QNX Neutrino.
- Besonderheiten von QNX
  - stark skalierbar, extrem kleiner Kernel (bei Version 4.24 ca.11kB)
  - Grundlegendes Konzept: Kommunikation erfolgt durch Nachrichten

QNX Architektur



Neutrino Microkernel

- Der Mikrokernel in QNX enthält nur die notwendigsten Elemente eines Betriebssystems:
  - Umsetzung der wichtigsten POSIX Elemente
    - POSIX Threads
    - POSIX Signale
    - POSIX Thread Synchronisation
    - POSIX Scheduling
    - POSIX Timer
  - Funktionalität für Nachrichten
- Eine ausführliche Beschreibung findet sich unter

  http://www.qnx.com/developers/docs/momentics621_docs/neutrino/sys_arch/kernel.html

Prozessmanager

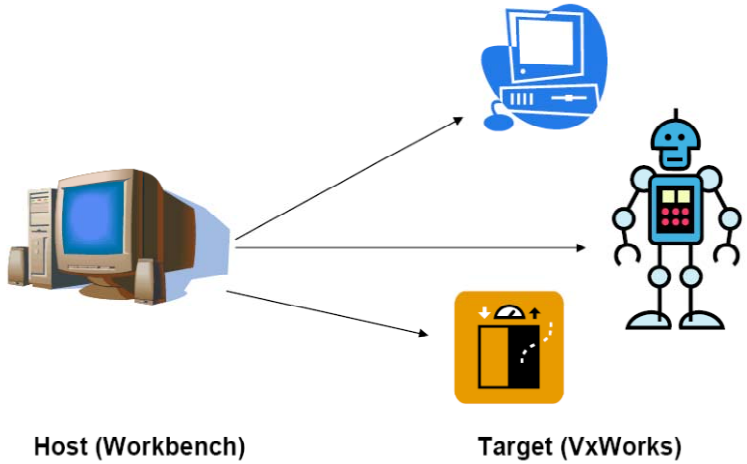
- Als wichtigster Prozess läuft in QNX der Prozessmanager.
- Die Aufgaben sind:
  - Prozessmanagement:
    - Erzeugen und Löschen von Prozessen
    - Verwaltung von Prozesseigenschaften
  - Speichermanagement:
    - Bereitstellung von Speicherschutzmechanismen,
    - von gemeinsamen Bibliotheken
    - und POSIX Primitiven zu Shared Memory
  - Pfadnamenmanagement
- Zur Kommunikation zwischen und zur Synchronisation von Prozessen bietet QNX Funktionalitäten zum Nachrichtenaustausch an.

Echtzeitbetriebssysteme

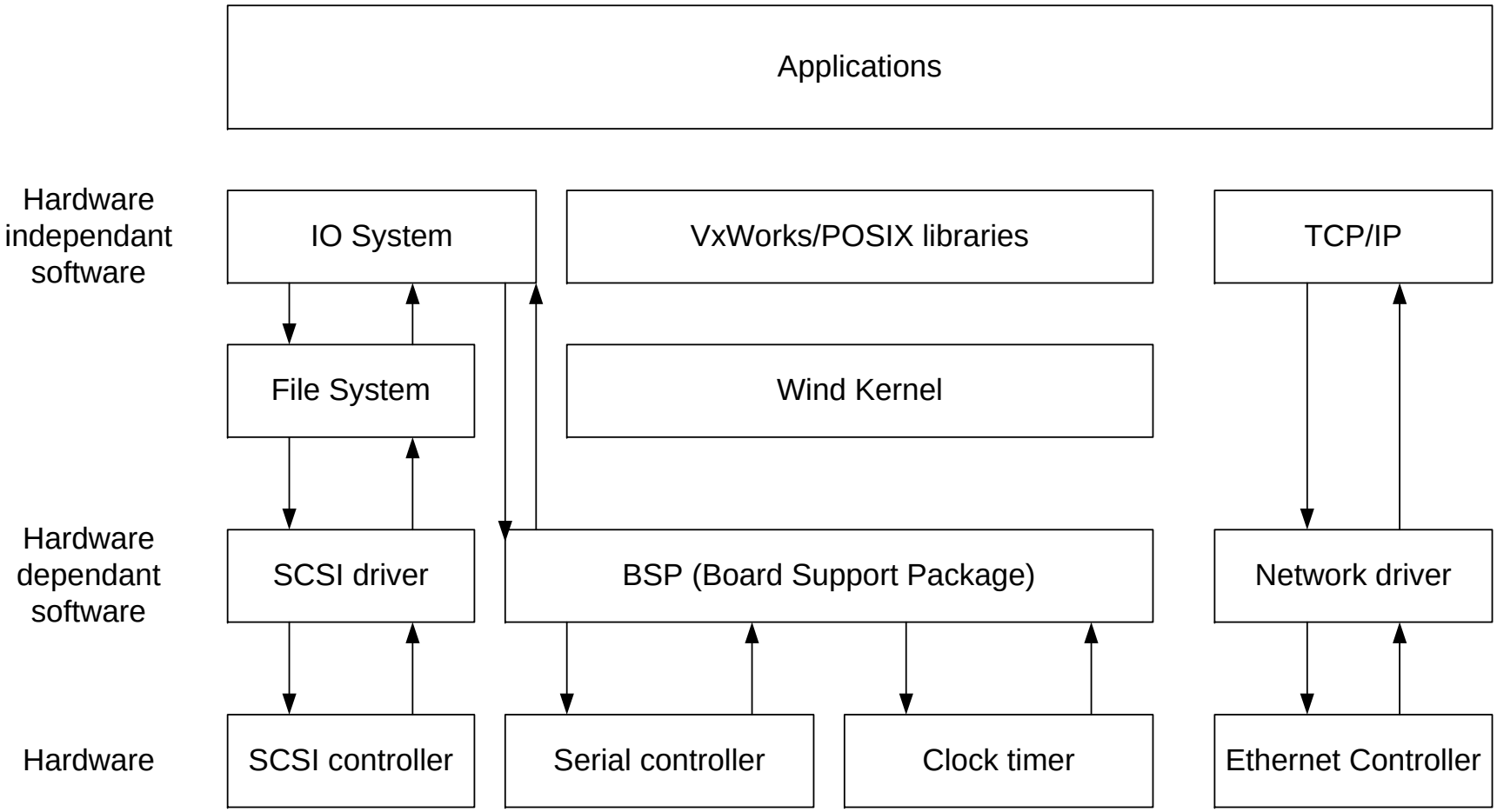
VxWorks

Eigenschaften

- Host-Target-Entwicklungssystem
- Eigene Entwicklungsumgebung Workbench mit Simulationsumgebung und integriertem Debugger basierend auf E
- Zielplattformen der Workbench 2.0: V
- Auf der Targetshell wird auch ein Inter die Shell eingegeben werden
- Kernel kann angepasst werden, allerdi werden
- Marktführer im Bereich der Echtzeitbe



Architektur



Prozessmanagement

- **Schedulingverfahren**: Es werden nur die beiden Verfahren FIFO und RoundRobin angeboten. Ein Verfahren für periodische Prozesse ist nicht verfügbar.
- **Prioritäten**: Die Prioritäten reichen von 0 (höchste Priorität) bis 255.
- **Uhrenauflösung**: Die Uhrenauflösung kann auf eine maximale Rate von ca. 30 KHz (abhängig von Hardware) gesetzt werden.
- **Prozessanzahl**: Die Anzahl der Prozesse ist nicht beschränkt (aber natürlich abhängig vom Speicherplatz)
- **API**: VxWorks bietet zum Management von Prozessen eigene Funktionen, sowie POSIX-Funktionen an.

Interprozesskommunikation und Speichermanagement

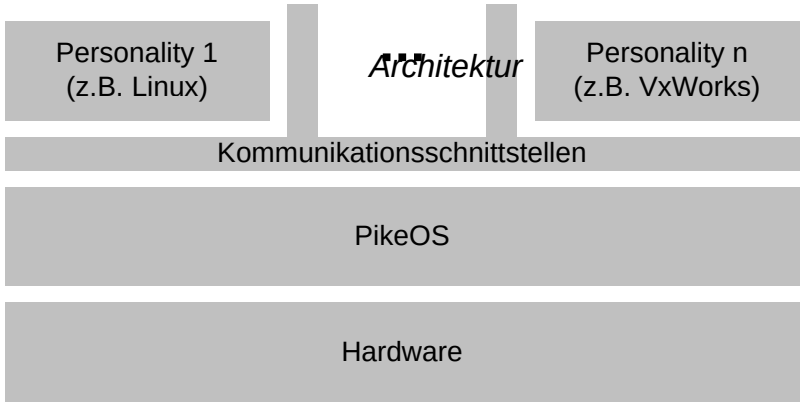
- Zur Interprozesskommunikation werden folgende Konzepte unterstützt:
  - Semaphor
  - Mutex (mit Prioritätsvererbung)
  - Nachrichtenwartschlangen
  - Signale
- Seit Version 6.0 wird zudem Speichermanagement angeboten:
  - Der Entwickler kann Benutzerprozesse mit eigenem Speicherraum entwickeln.
  - Bisher: nur Threads im Kernel möglich.

Echtzeitbetriebssysteme

PikeOS

PikeOS: Betriebssystem mit Paravirtualisierung

- Idee: Virtualisierung der Hardware – jede Partition (Personality) verhält sich als hätte sie eine eigene CPU zur Verfügung
- Mehrere Betriebssysteme können auf der gleichen CPU nebenläufig ausgeführt werden.
- Die Speicherbereiche, sowie CPU-Zeiten der einzelnen Partitionen werden statisch während der Implementierung festgelegt.
- Durch die Partitionierung ergeben sich diverse Vorteile:
  - Bei einer Zertifizierung muss nur der sicherheitskritische Teil des Gesamtsystems zertifiziert werden.
  - Reduzierung der Steuergeräte durch Zusammenführung der Funktionalitäten mehrerer Steuergeräte
  - Echtzeitkomponenten können einfacher von nicht-kritischen Komponenten getrennt werden – Nachweis der Fristeneinhaltung wird einfacher



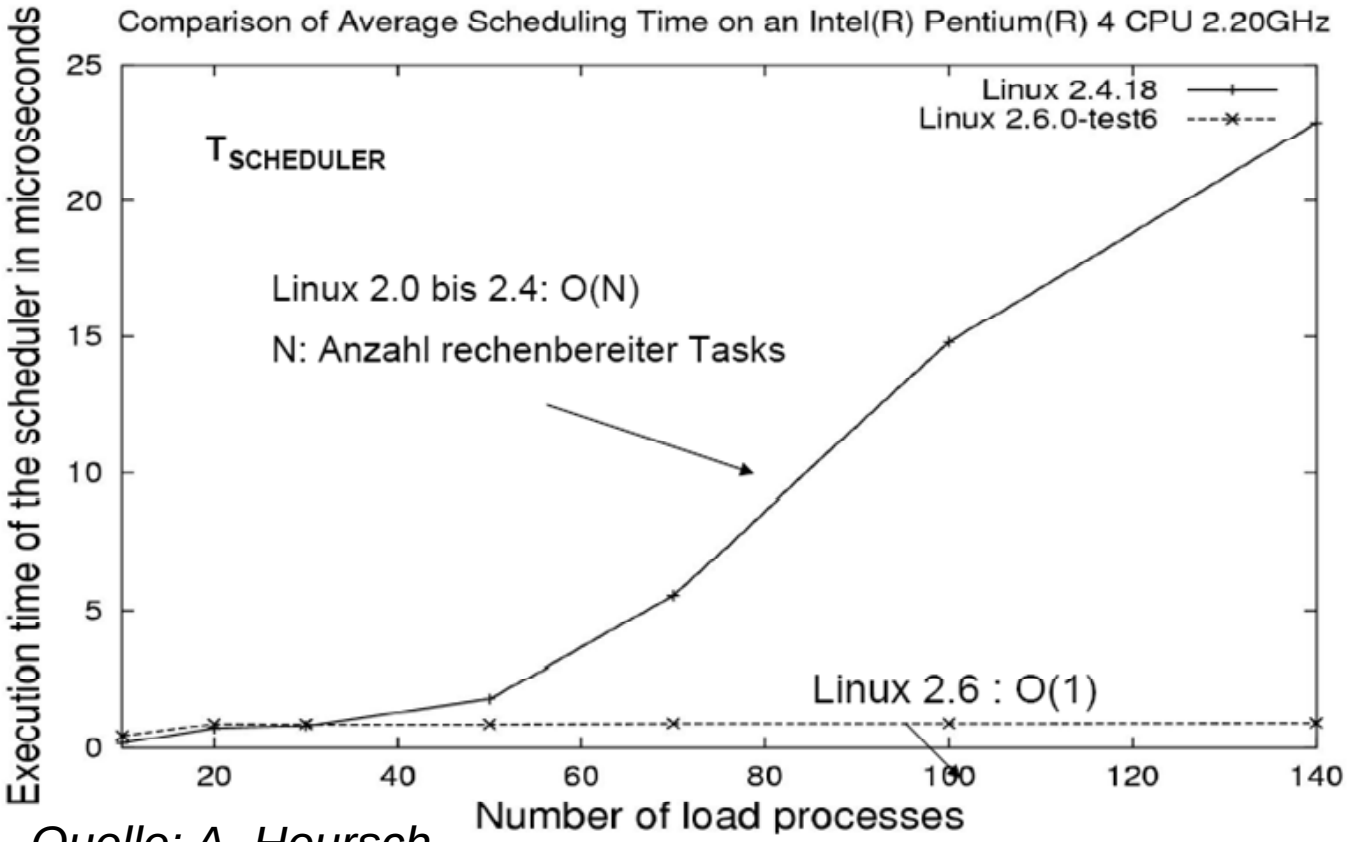
Echtzeitbetriebssysteme

Linux Kernel 2.6

Bestandsaufnahme

- Für die Verwendung von Linux Kernel 2.6 in Echtzeitsystemen spricht:
  - die Existenz eines echtzeitfähigen Schedulingverfahrens (prioritätenbasiertes Scheduling mit FIFO oder RoundRobin bei Prozessen gleicher Priorität)
  - die auf 1 ms herabgesetzte Zeitauflösung der Uhr (von 10ms in Kernel 2.4)
- Gegen die Verwendung spricht:
  - die Ununterbrechbarkeit des Kernels.

Vergleich Schedulerlaufzeiten Kernel 2.4/2.6



Quelle: A. Heursch

Unterbrechbarkeit des Kernels

- Im Kernel ist der **Preemptible Kernel Patch** als Konfigurationsoption enthalten $\Rightarrow$ Erlaubt die Unterbrechung des Kernels.
- **Problem**: Existenz einer Reihe von kritischen Bereichen, die zu langen Verzögerungszeiten führen.
- **Low Latency Patches** helfen bei der Optimierung, aber harte Echtzeitanforderungen können nicht erfüllt werden.
- Weitere Ansätze: z.B. Verwendung von binären Semaphoren (Mutex) anstelle von generellen Unterbrechungssperren, Verhinderung von Prioritätsinversion durch geeignete Patches, siehe Paper von A. Heursch

Speichermanagement

- Linux unterstützt Virtual Memory
- Die Verwendung von Virtual Memory führt zu zufälligen und nicht vorhersagbaren Verzögerung, falls sich eine benötigte Seite nicht im Hauptspeicher befindet.
  $\Rightarrow$ Die Verwendung von Virtual Memory in Echtzeitanwendungen ist nicht sinnvoll.
- Vorgehen: Zur Vermeidung bietet Linux die Funktionen `mlock()` und `mlockall()` zum **Pinning** an.
- Pinning bezeichnet die Verhinderung des Auslagerns eines bestimmten Speicherbereichs oder des kompletten Speichers eines Prozesses.

Uhrenauflösung

- Die in Linux Kernel 2.6 vorgesehene Uhrenauflösung von 1ms ist häufig nicht ausreichend.
- Problemlösung: Verwendung des **High Resolution Timer Patch (hrtimers)**
  - Durch Verwendung des Patches kann die Auflösung verbessert werden.
  - Der Patch erlaubt z.B. die Erzeugung einer Unterbrechung in 3,5 Mikrosekunden von jetzt an.
  - Einschränkung: Zeitliche Angabe muss schon vorab bekannt sein $\Rightarrow$ keine Zeitmessung möglich
  - Gründe für die hrtimers-Lösung findet man unter: http://www.kernel.org/git/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/hrtimers.txt

Echtzeitbetriebssysteme

RTLinux/RTAI

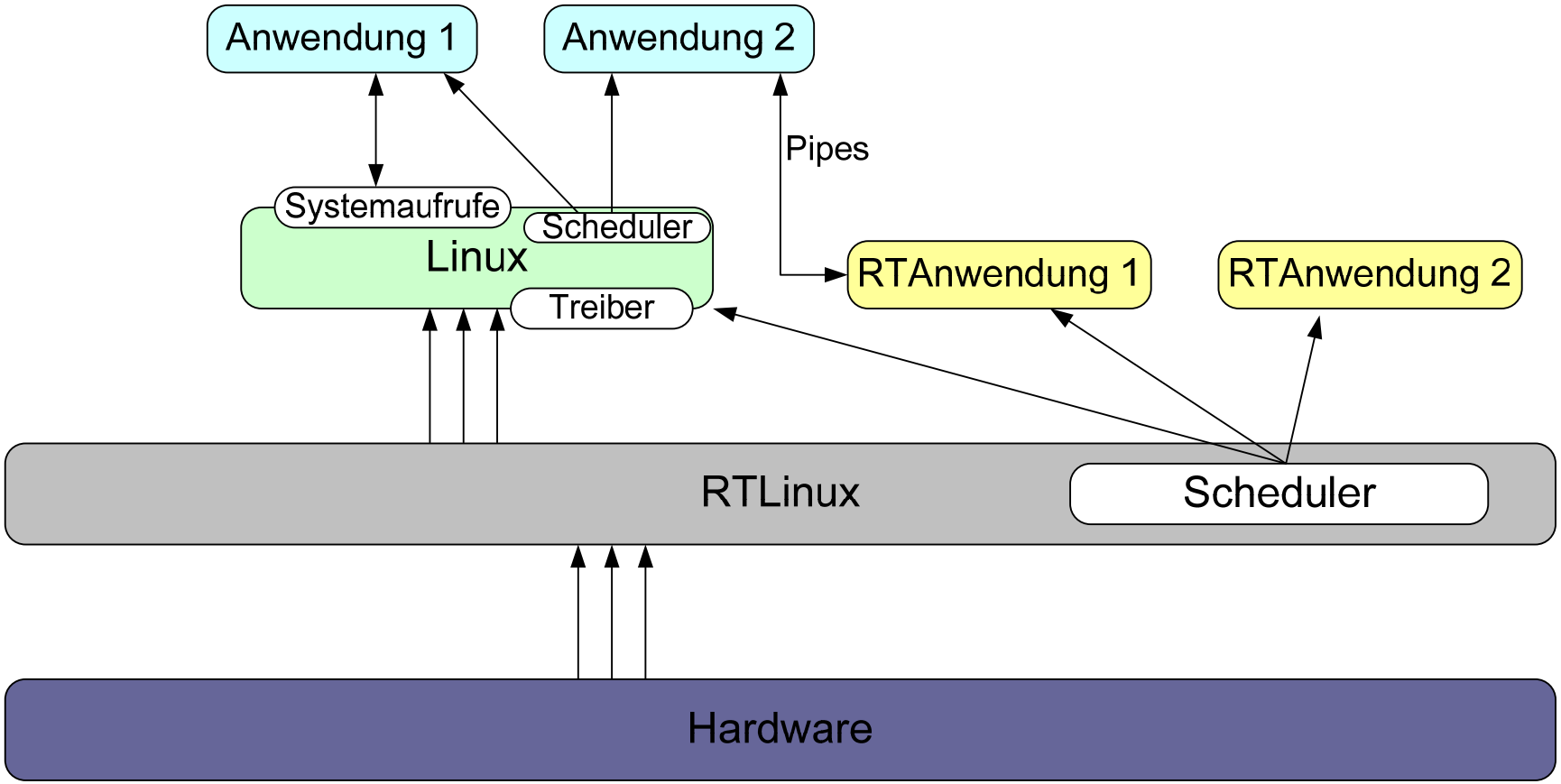
Motivation

- Aus diversen Gründen ist die Verwendung von Linux in Echtzeitsystemen erstrebenswert:
  - Linux ist weitverbreitet
  - Treiber sind sehr schnell verfügbar
  - Es existieren viele Entwicklungswerkzeuge ⇒ die Entwickler müssen nicht für ein neues System geschult werden.
  - Häufig müssen nur geringe Teile des Codes echtzeitfähig ausgeführt werden.
- **Probleme**:
  - grobgranulare Synchronisation
  - trotz Patches oft zu lange Latenzzeiten
  - Hochpriorisierte Prozesse können durch andere Prozesse mit niedrigerer Priorität blockiert werden, Grund: Hardwareoptimierungsstrategien (z.B. Speichermanagement)
- **Ansatz**: Modifikation von Linux, so dass auch harte Echtzeitanforderungen erfüllt werden.

Ansatz

- Anstelle von Patches wird eine neue Schicht zwischen Hardware und Linux-Kernel eingefügt:
  - Volle Kontrolle der Schicht über Unterbrechungen
  - Virtualisierung von Unterbrechungen (Barabanov, Yodaiken, 1996): Unterbrechungen werden in Nachrichten umgewandelt, die zielgerichtet zugestellt werden.
  - Virtualisierung der Uhr
  - Anbieten von Funktionen zum virtuellen Einschalten und Ausschalten von Unterbrechungen
  - Das Linux-System wird als Prozess mit niedrigster Priorität ausgeführt.

RTLinux Architektur



Unterschiede RTAI/RTLinux

- RTLinux verändert Linux-Kernel-Methoden für den Echtzeiteingriff ⇒ Kernel-Versions-Änderungen haben große Auswirkungen.
- RTAI fügt Hardware Abstraction Layer (HAL) zwischen Hardware und Kernel ein. Hierzu sind nur ca. 20 Zeilen Code am Originalkern zu ändern. HAL selbst umfasst kaum mehr als 50 Zeilen ⇒ Transparenz.
- RTAI ist frei, RTLinux in freier (Privat, Ausbildung) und kommerzieller Version.
- Beide Ansätze verwenden ladbare Kernel Module für Echtzeitprozesse.
- RTAI (mit Variante LXRT) erlaubt auch die Ausführung von echtzeitkritischen Prozessen im User-Space, Vorteil ist beispielsweise der Speicherschutz

Echtzeitbetriebssysteme

Windows CE & Windows Embedded

Eigenschaften

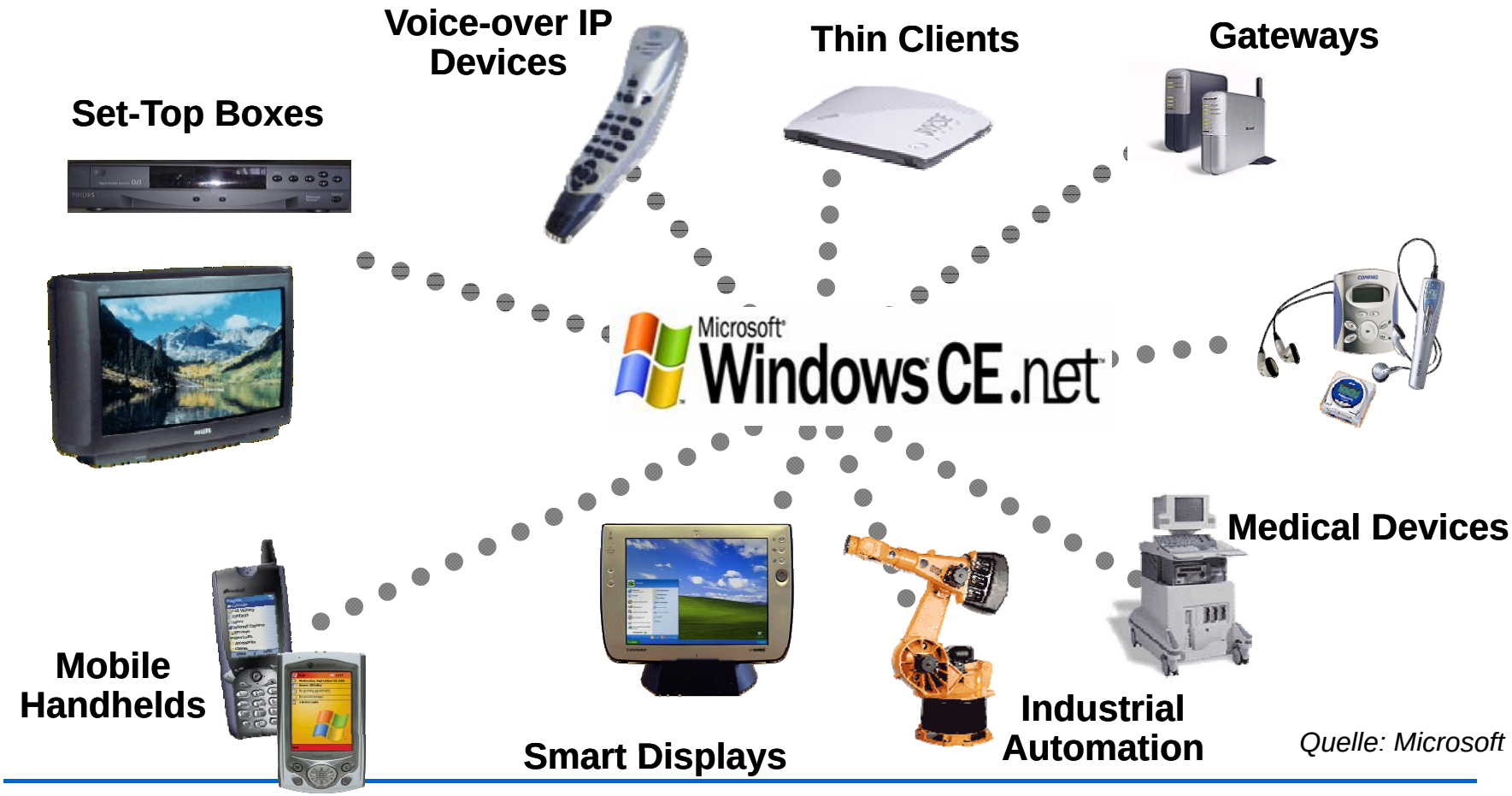
- Windows CE
  - 32-bit, Echtzeitbetriebssystem
  - Unterstützung von Multitasking
  - Stark modularer Aufbau
  - Skalierbar entsprechend der gewünschten Funktionalität
- Windows Embedded
  - „Skalierbares Windows XP“
  - Komponenten von XP können entfernt werden um den benötigten Speicherplatz zu minimieren

Windows CE und Embedded im Vergleich

Windows XP Embedded		Windows CE.net
x86 processors	Processor Support	Multiple processors / power management
Full Win32 API compatibility	Win32 API Compatibility	Requires additional effort
Basic images from 8MB (“Hello World”)	Footprint	Basic images from 350 KB
With 3rd party extensions	Real-time	Native

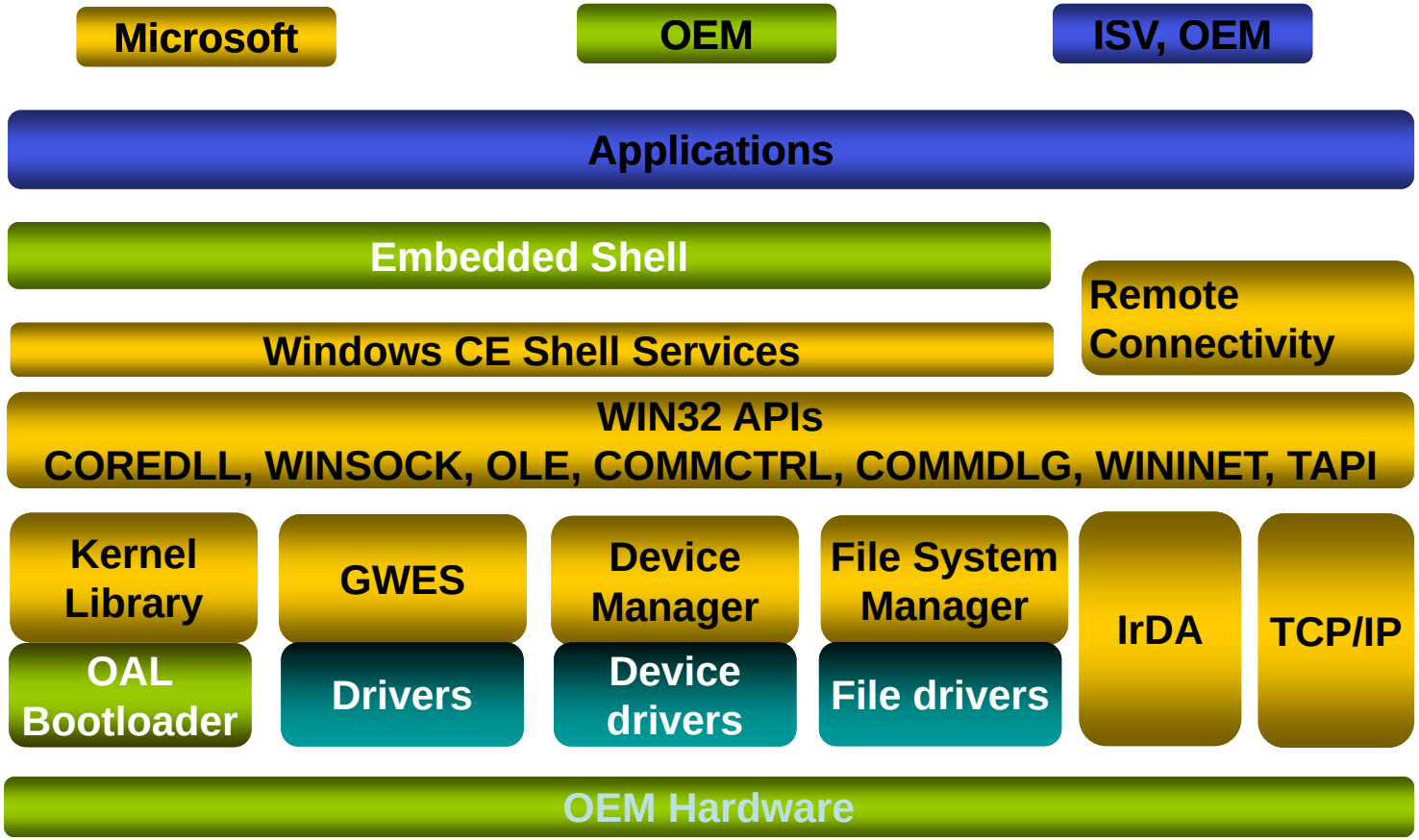
Quelle: Microsoft

Einsatzbereiche



Quelle: Microsoft

Windows CE Architektur



Quelle: Microsoft

Funktionen des Betriebssystemkerns

- Kernel, Speicherverwaltung
  - Shared heap
  - Unterstützung von Watchdogs
  - 64 Systeminterrupts
- Geräteunterstützung
  - Unterstützung diverser Massenspeicher, z.B. USB, Flash,..
- Browser
- Multimedia
  - Diverse Graphiktreiber
  - umfassende Codecunterstützung
- Kryptographie-Funktionen

Echtzeitunterstützung

- Unterstützung verschachtelter Interrupts
- 256 Prioritätslevel
- Thread quantum level control
- Speicherschutz (Pinning) zur Umgehung von Virtual Memory
- Eingebaute Leistungsüberwachungswerkzeuge
- Niedrige ISR/IST Latenz
  - ISR/IST Latenz von 2.8/26.4 Mikrosekunden auf Intel 100MHz Board

Echtzeitbetriebssysteme

Zusammenfassung

Zusammenfassung

- Es gibt kein typisches Echtzeitbetriebssystem da je nach Einsatzbereich die Anforderungen sehr unterschiedlich sind.
- Der minimale Speicherbedarf reicht von wenigen Kilobyte (TinyOS, QNX) bis hin zu mehreren Megabyte (Windows CE / XP Embedded).
- Die Betriebssysteme sind typischerweise skalierbar. Zur Änderung des Leistungsumfangs von Betriebssystemen muss das System entweder neu kompiliert werden (VxWorks) oder neue Prozesse müssen nachgeladen werden (QNX).
- Die Echtzeitfähigkeit von Standardbetriebssysteme kann durch Erweiterungen erreicht werden (RTLinux/RTAI).
- Die Schedulingverfahren und die IPC-Mechanismen orientieren sich stark an den in POSIX vorgeschlagenen Standards.
- Das Problem der Prioritätsinversion wird zumeist durch Prioritätsvererbung gelöst.

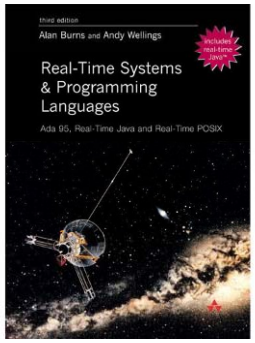
Kapitel 8

Programmiersprachen für Echtzeitsysteme

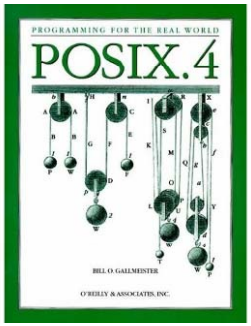
Inhalt

- Motivation
  - Anforderungen von Echtzeitsystemen
  - Geschichte
- PEARL
- Ada
- Real-Time Java
- Zusammenfassung

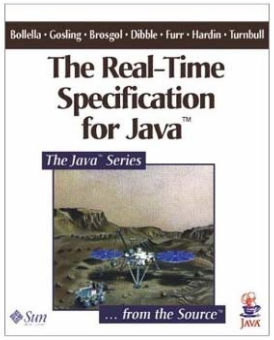
Literatur



A. Burns, A. Wellings: Real-Time Systems & Programming Languages, 2001



B. Gallmeister: POSIX.4 Programming for the Real World, 1995



G. Bollella: The Real-Time Specification for Java, 2000

Paper:

- N. Wirth: Embedded Systems and Real-time Programming, EMSOFT 2001
- Ascher Opler: Requirements for Real-Time Languages, Communications of ACM 1966

Programmiersprachen für Echtzeitsysteme

Anforderungen

Anforderungen

- Die Anforderungen an Programmiersprachen für Echtzeitsysteme fallen in verschiedene Bereiche:
  - Unterstützung bei der Beherrschung komplexer und nebenläufiger Systeme
  - Möglichkeit zur Spezifikation zeitlicher Anforderungen
  - Unterstützung der hardwarenahen Programmierung
  - Erfüllung hoher Sicherheitsanforderungen (Fehlersicherheit)
  - Möglichkeiten zum Umgang mit verteilten Systemen

Beherrschung komplexer nebenläufiger Systeme

- Anforderungen an Programmiersprachen
  - Konstrukte zur Aufteilung der Anwendung in kleinere, weniger komplexe Subsysteme
  - Unterstützung von Nebenläufigkeit (Prozesse, Threads)
  - Daten- und Methodenkapselung in Modulen zur Erleichterung der Wiederverwendbarkeit
  - Eignung für unabhängiges Implementieren, Übersetzen und Testen von Modulen durch verschiedene Personen

Einhalten zeitlicher Anforderungen

- Projektierbares Zeitverhalten
  - Möglichkeit zur Definition von Prioritäten
  - wenig (kein) Overhead durch Laufzeitsystem (z.B. Virtual Machine)
- Bereitstellung umfangreicher Zeitdienste
- Zeitüberwachung aller Wartezustände
- Möglichkeit zur Aktivierung von Prozessen
  - sofort
  - zu bestimmten Zeitpunkten
  - in bestimmten Zeitabständen
  - bei bestimmten Ereignissen

Unterstützung hardwarenaher Programmierung

- Ansprechen von Speicheradressen, z.B. „memory mapped I/O“
- Unterbrechungs- und Ausnahmebehandlung
- Unterstützung vielseitiger Peripherie
- Definition virtueller Geräteklassen mit einheitlichen Schnittstellen
- einheitliches Konzept für Standard- und Prozesse- Ein-/Ausgabe

Erfüllung hoher Sicherheitsanforderungen

- Lesbarkeit, Übersichtlichkeit, Einfachheit durch wenige Konzepte
- Modularisierung und strenge Typüberprüfung als Voraussetzung zur frühen Fehlererkennung durch Übersetzer, Binder und Laufzeitsystem
- Überprüfbare Schnittstellen (-beschreibungen) der Module
- Verifizierbarkeit von Systemen

Sicherheit fängt schon im Kleinen an

- Lexikalische Konventionen können Fehler verhindern.
- Negatives Beispiel: FORTRAN
  - In FORTRAN werden Leerzeichen bei Namen ignoriert.
  - Variablen müssen in FORTRAN nicht explizit definiert werden

- Problem in Mariner 1:
  Aus einer Schleife

```
DO 5 K = 1,3
```

  wird durch versehentliche Verwendung eines Punktes

```
DO5K=1.3
```

  eine Zuweisung an eine nicht deklarierte Variable.
  $\Rightarrow$ Zerstörung der Rakete, Schaden 18,5 Millionen $

Anforderungen durch verteilte Systeme:

- Notwendigkeit vielseitiger Protokolle zur Kommunikation (Feldbus, LAN)
- Unterstützung von Synchronisation auch in verteilten Systemen
- Möglichkeit zur Ausführung von Operationen auf Daten anderer Rechner
- Konfigurationsmöglichkeit zur Zuordnung von Programmen/Modulen zu Rechnern
- Möglichkeit zur automatischen Neukonfigurierung in Fehlersituationen

Programmiersprachen für Echtzeitsysteme

Geschichte

Geschichte: 1960-1970

- 1960-1970
  - Verwendung von Assemblerprogrammen, da der Speicher sehr teuer ist
  - Programme sind optimiert ⇒ jedes Bit wird genutzt
- ab ca. 1966
  - erster Einsatz von höheren Sprachen, z.B.
    - CORAL und RTL/2
    - ALGOL 60
    - FORTRAN IV
  - Prozeduraufrufe für Echtzeitdienste des Betriebssystems
  - Probleme:
    - viel Wissen über Betriebssystem notwendig
    - wenig portabel
    - keine semantische Prüfung der Parameter durch den Übersetzer, da keine speziellen Datentypen für Prozesse, Uhren oder Semaphoren existierten ⇒ schwierige Fehlersuche

Geschichte: 1970-1980

- Existenz erster Echtzeitsprachen (nationale bzw. internationale Normen):
  - PEARL (Deutschland): Process and Experiment Automation Realtime Language
  - HAL/S (USA)
  - PROCOL (Japan)
  - RT-FORTRAN
  - RT-BASIC
- Neue Datentypen (z.B. task, duration, sema, interrupt) mit zugehörigen Operationen sind in die Sprache integriert
- Einführung einheitlicher Anweisungen vor Ein-/Ausgabe und die Beschreibung von Datenwegen

Geschichte: 1970-1980

- Vorteil:
  - Benutzerfreundliche Sprachelemente
  - Prüfung der Semantik der Parameter bei Betriebssystemaufrufen durch Übersetzer möglich
  - Weitgehende Portabilität
- Nachteil: geeignete Betriebssysteme sind nicht vorhanden Möglichkeiten
  1. Entwicklung eines eigenen Betriebssystems ⇒ hohe Entwicklungskosten
  2. Anpassung eines vorhandenen Standardbetriebssystems ⇒ Gefahr der Existenz überflüssiger Teile im Betriebssystem, eingeschränkte Portabilität

Geschichte ab 1978

- universelle sichere hohe Sprachkonzepte für alle Anwendungsbereiche
  - Standardisierung, inbesondere durch Department of Defense (DOD): Ada
  - Datentypen (z.B. task, duration, interrupt) oder systemabhängige Parameter werden in sprachlich sauberer Weise mittels Module /Packages eingebunden
- Beispiele:
  - Ada83,Ada95
  - CHILL
  - PEARL, PEARL 90, Mehrrechner.PEARL

Geschichte heute:

- Trend hin zu universellen Sprachen (z.B. C,C++ oder Java) mit Bibliotheksprozeduren für Echtzeitdienste angereichert (z.B. POSIX)
- herstellerspezifische Speziallösungen für eingeschränkte Anwendungsbereiche, z.B.
  - Prüfsysteme
  - Standardregelungsaufgaben
  - Förderungstechnik
  - Visualisierung (Leitstand)
  - Telefonanlagen
- Beispiele:
  - SPS-Programmierung (Speicherprogrammierbare Steuerung)
  - ATLAS (Abbreviated Test Language for All Systems) für Prüfsysteme (v.a. Flugzeugelektronik)
  - ESTEREL

Programmiersprachen für Echtzeitsysteme

PEARL

Daten

- Process and Experiment Automation Real-Time Language
- DIN 66253
- Ziele:
  - Portabilität
  - Sicherheit
  - sichere und weitgehend rechnerunabhängige Programmierung
- lauffähig z.B. unter UNIX, OS/2, Linux
- Versionen: BASIC PEARL (1981), Full PEARL (1982), Mehrrechner PEARL (1988), PEARL 90 (1998)
- http://www.irt.uni-hannover.de/pearl/

Eigenschaften

- strenge Typisierung
- modulbasiert
- unterstützt (prioritätenbasiertes) Multitasking
- E/A-Operationen werden von eigentlicher Programmausführung separiert
- Synchronisationsdienste: Semaphore, Bolt-Variablen
- Zugriff auf Unterbrechungen
- erleichterte Zeitverwaltung

Grundstruktur

```
/*Hello World*/
MODULE Hello;
   SYSTEM;
        termout: STDOUT;

   PROBLEM;
        DECLARE x FLOAT;
        T: TASK MAIN;
        x := 3.14;                    !PI
        PUT 'Hello' TO termout;
   END;
MODEND;
```

Erläuterung

- **Modularität**: Anwendungen können in einzelne Module aufgeteilt werden (`MODULE`, `MODEND`).
- Aufspaltung in System- und Problemteil:
  - **Systemteil** (`System;`): Definition von virtuellen Geräten für alle physischen Geräte, die das Modul benutzt. Der Systemteil muss auf den entsprechenden Computer angepasst sein
    $\Rightarrow$ Hardwareabhängigkeit
  - **Problemteil** (`PROBLEM;`): eigentlicher, portabler Programmcode
- Sonstige Notationen typisch für prozedurale Sprachen:
  - Kommentare `!`,`/*…*/`
  - Semikolon zur Terminierung von Anweisungen

Datentypen

- Variablen werden durch DECLARE deklariert und mittels INIT initialisiert.
- Durch das Schlüsselwort INV werden Konstanten gekennzeichnet.
- Die temporalen Variablen bieten eine Genauigkeit von einer Millisekunde
- Die Genauigkeit der Datentypen kann angegeben werden
- Zeiger auf Datentypen werden unterstützt

Schlüsselwort	Bedeutung	Beispiel
FIXED	Ganzzahlige Variable	-2
FLOAT	Gleitkommazahl	0.23E-3
CLOCK	Zeitpunkt	10:44:23.142
DURATION	Zeitdauer	2 HRS 31 MIN 2.346 SEC
CHAR	Folge von Bytes	'Hallo'
BIT	Folge von Bits	'1101'B1

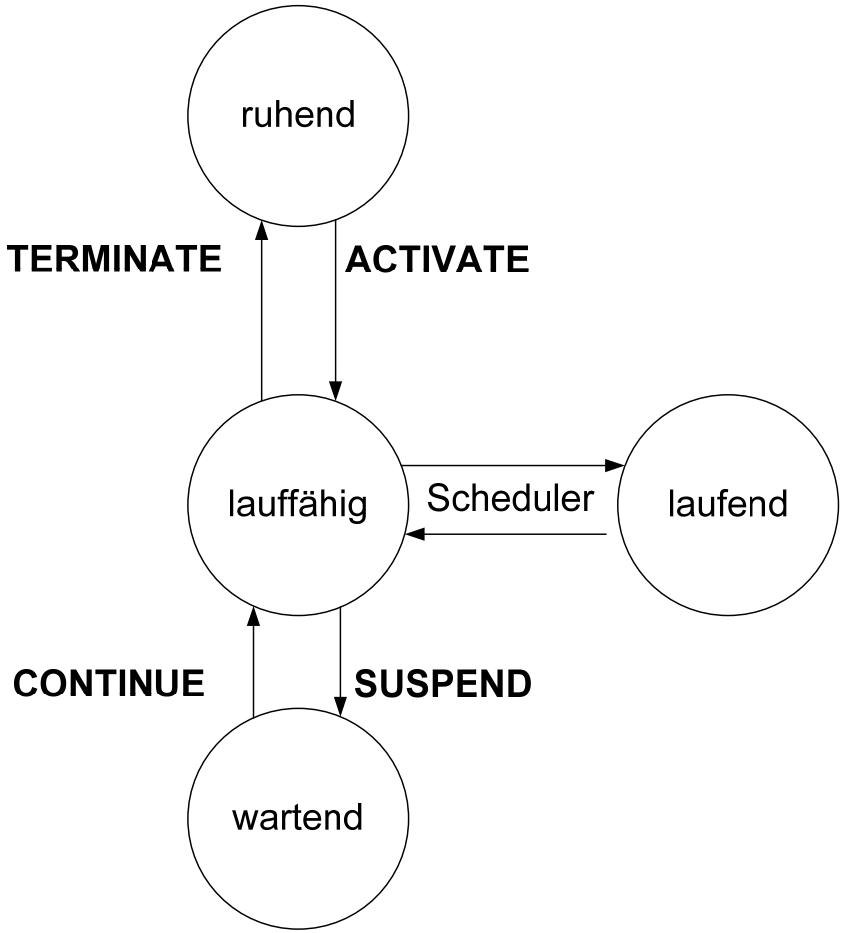
Prozessmodell

- Initial sind alle Prozesse bis auf MAIN ruhend
- Zustandswechsel sind unter Angabe einer exakten Zeit möglich:

```
AFTER 5 SEC ACTIVATE Task1;

AT 10:15:0 ALL 2 MIN
     UNTIL 11:45:0
     ACTIVATE Student;
```

- Scheduling präemptives, prioritätenbasiertes Schedulingverfahren mit Zeitscheiben (Round-Robin)
- Zuweisung der Prioritäten durch den Benutzer
- Zeitscheibenlänge abhängig vom Betriebssystem



Prozess-Synchronisation

- Zur Synchronisation bietet PEARL Semaphore und Bolt-Variablen:
  - Semaphore (Datentyp: SEMA):
    - Deklaration wie bei einer Variablen
    - Operationen REQUEST und RELEASE zum Anfordern und Freigeben des Semaphores
    - Mittels der Operation TRY kann versucht werden den Semaphore nicht blockierend zu lesen
    - Es werden keine Möglichkeiten zur Vermeidung von Prioritätsinversion geboten
  - Bolt-Variablen (Datentyp: BOLT):
    - Bolt-Variablen besitzen wie Semaphoren die Zustände belegt und frei und zusätzlich einen 3. Zustand: Belegung nicht möglich
    - RESERVE und FREE funktionieren analog zu Semaphore-Operationen REQUEST bzw. RELEASE
    - exklusive Zugriffe mit RESERVE haben Vorrang von (nicht exklusiven) Zugriffen mit ENTER (Freigabe mit LEAVE)
    - Eine elegante Formulierung des Leser-Schreiber-Problems ist damit möglich

Lösung: Leser-Schreiber-Problem

```
PROBLEM;
    DECLARE content CHAR(255); ! Speicher: 255 Bytes
         DCL key BOLT;     ! Default: frei

   LESER1: TASK;
        DECLARE local1 CHAR(255);
        ENTER key; local1=content; LEAVE key;
   END;

   LESER2: TASK;
        DECLARE local2 CHAR(255);
        ENTER key; local2=content; LEAVE key;
   END;

   SCHREIBER1: TASK;
        DECLARE newcontent1 CHAR(255);
        RESERVE key; content=newcontent1; FREE  key;
   END;

   SCHREIBER2: TASK;
        DECLARE newcontent2 CHAR(255);
        RESERVE key; content=newcontent2; FREE  key;
   END;
```

Unterbrechungen

- Es können nur Prozesse, die durch WHEN eingeplant sind, aktiviert oder fortgesetzt werden.
- Es ist möglich Unterbrechungen durch DISABLE/ENABLE zu sperren bzw. freizugeben.
- Beispiel: Student 2 weckt Student 1 beim Eintreffen der Unterbrechung auf.

```
MODULE Vorlesung:
   System;
        alarm: IR*2;

   PROBLEM;
        SPECIFY alarm INTERRUPT;

   student2: TASK PRIORITY 20
        WHEN alarm ACTIVATE
   student1;
        DISABLE alarm;
        ...
        ENABLE alarm;
   END;
MODEND;
```

Programmiersprachen für Echtzeitsysteme

Ada

Einleitung

- 1970 von Jean Ichbiah (Firma Honeywell Bull) entworfen
- Durch das Department of Defense gefördert
- Mitglied der Pascal Familie
- Häufige Verwendungen für Systeme mit hohen Anforderungen an die Sicherheit.
- Bis 1997 mussten alle Systeme im Rahmen von DOD-Projekten mit einem Anteil von mehr als 30% neuen Code in ADA implementiert werden.
- Versionen: Ada 83, Ada 95
- Freie Compiler sind verfügbar: z.B. http://www.adahome.com
- http://www.ada-deutschland.de/