

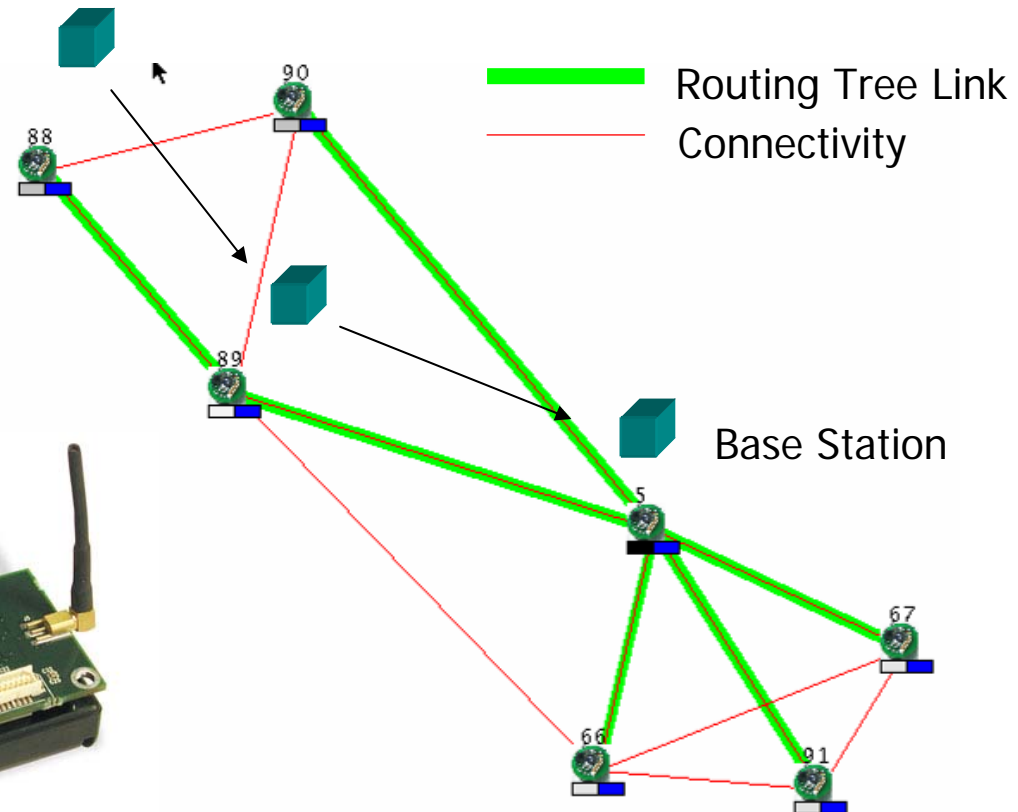


Echtzeitbetriebssysteme

TinyOS

Einsatzgebiet: AdHoc-Sensornetzwerke

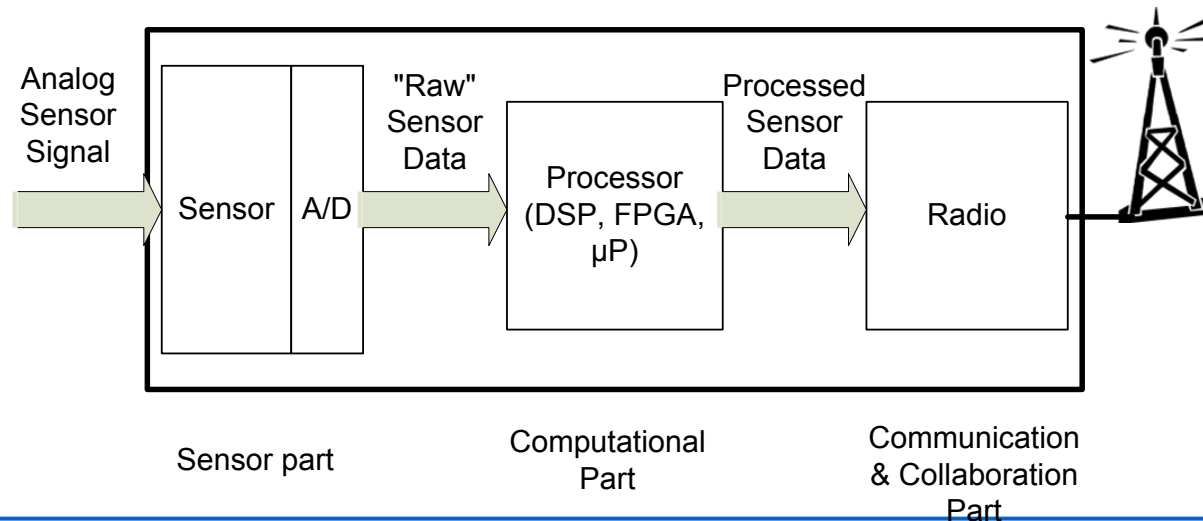
- Begriff Smart-Dust: Viele kleine Sensoren überwachen die Umgebung
- Ziele: robuste und flächendeckende Überwachung
- Probleme:
 - eingeschränkte Lebensdauer (Batterie)
 - eingeschränkter Speicherplatz
 - geringe Kommunikationsbandbreite
 - geringe Rechenleistung



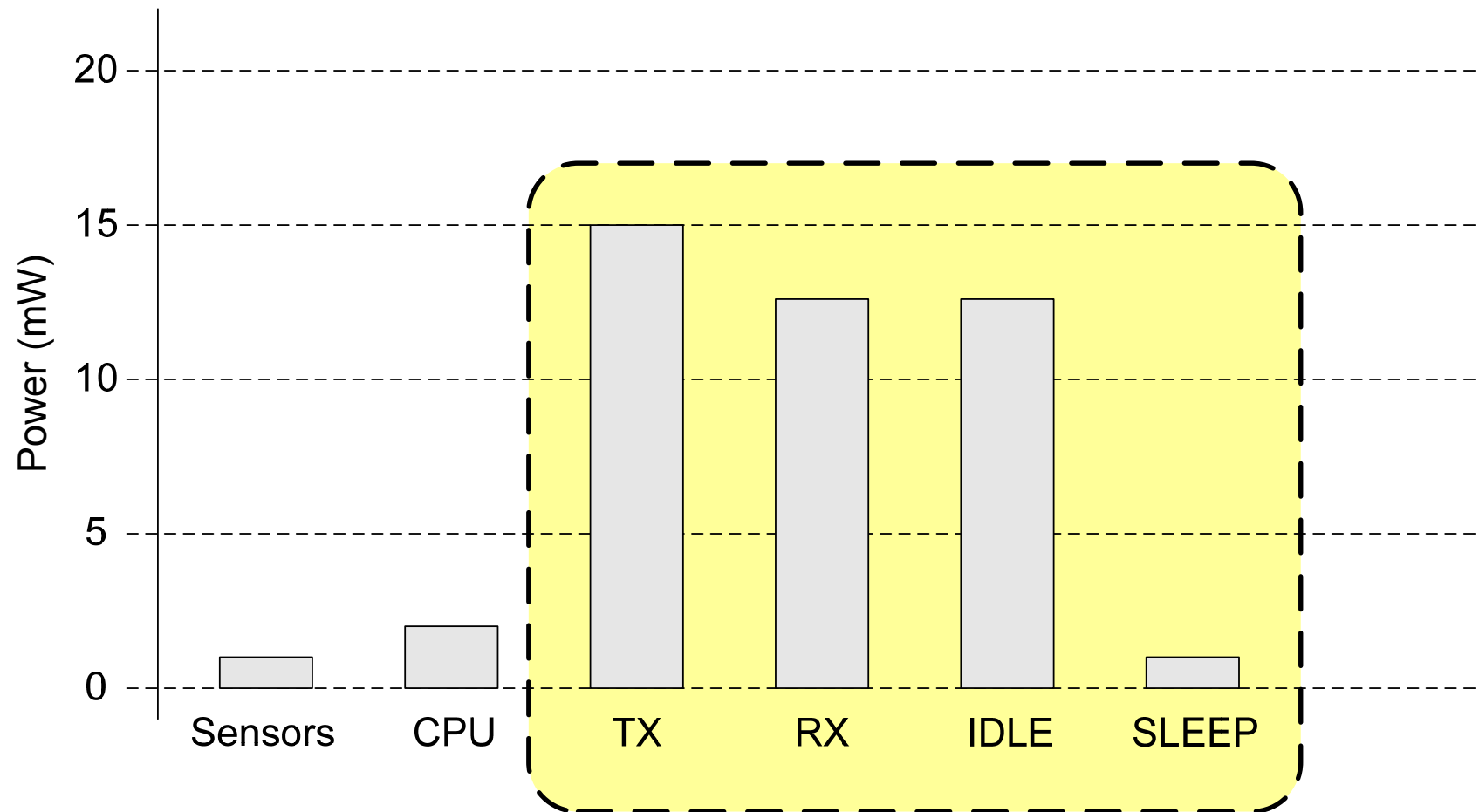
Quelle: <http://tinyos.millennium.berkeley.edu>

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

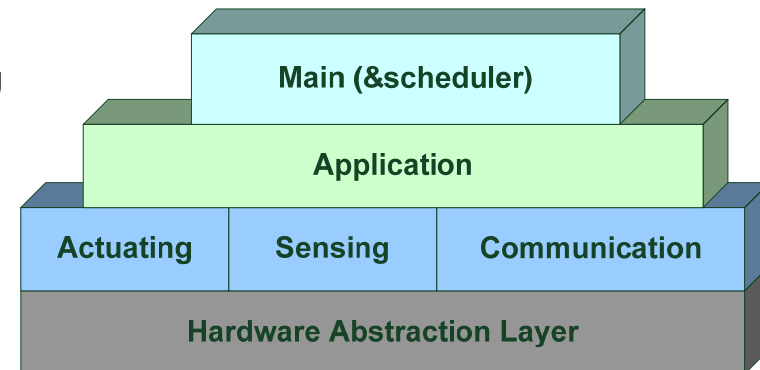


Stromverbrauch



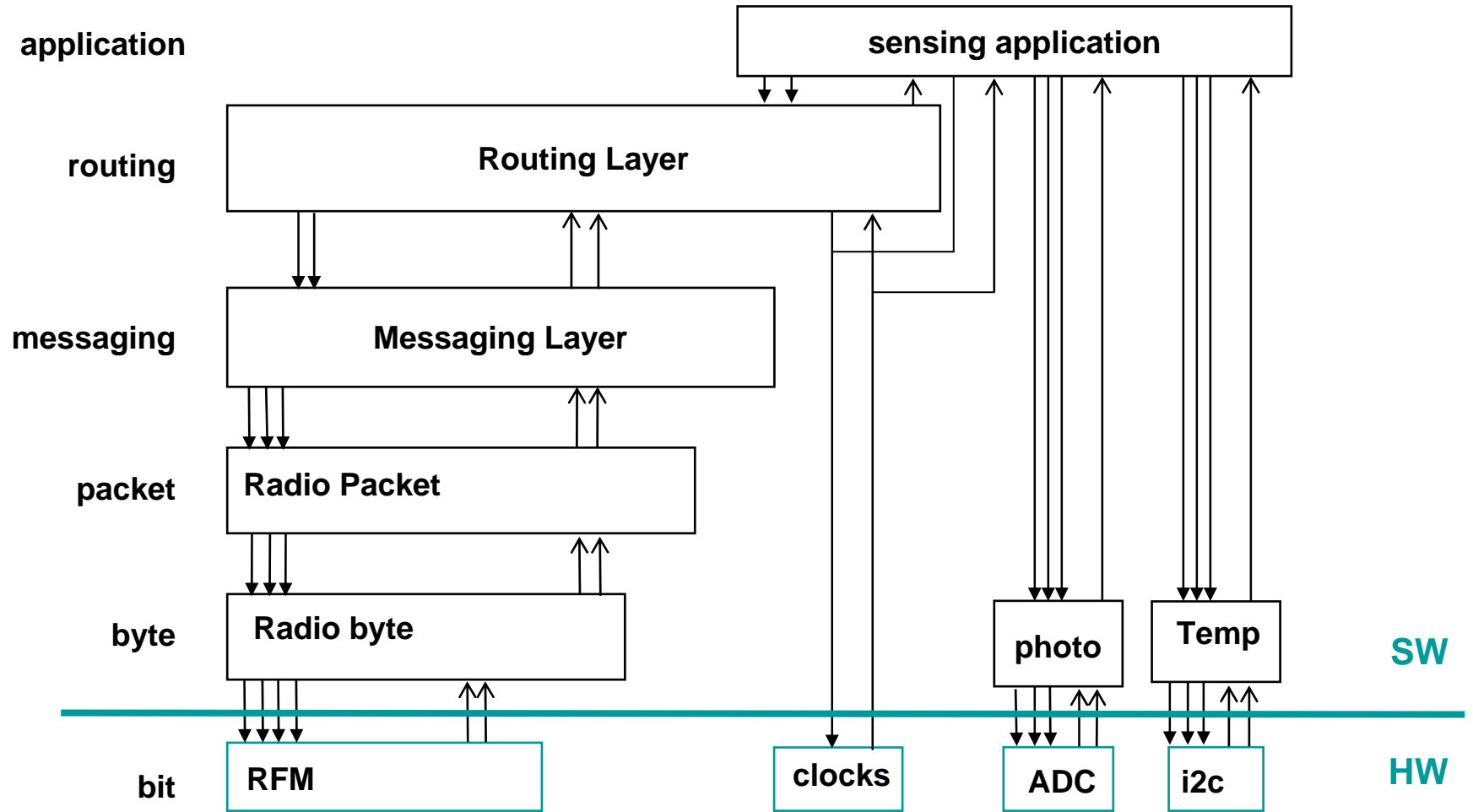
TinyOS

- TinyOS ist kein wirkliches Betriebssystem im traditionellen Sinn, eher ein anwendungsspezifisches Betriebssystem
 - keine Trennung der Anwendung vom OS \Rightarrow 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

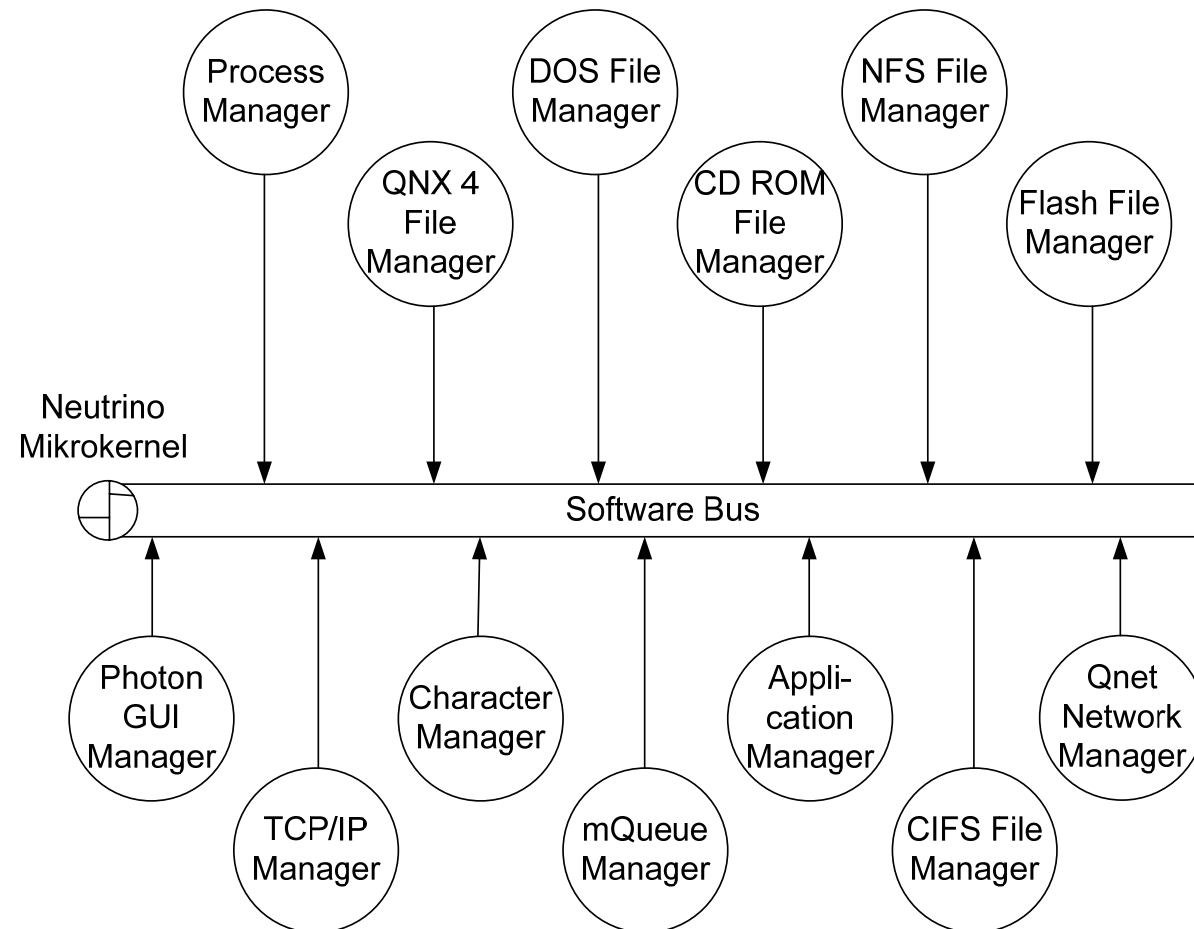
QNX



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.

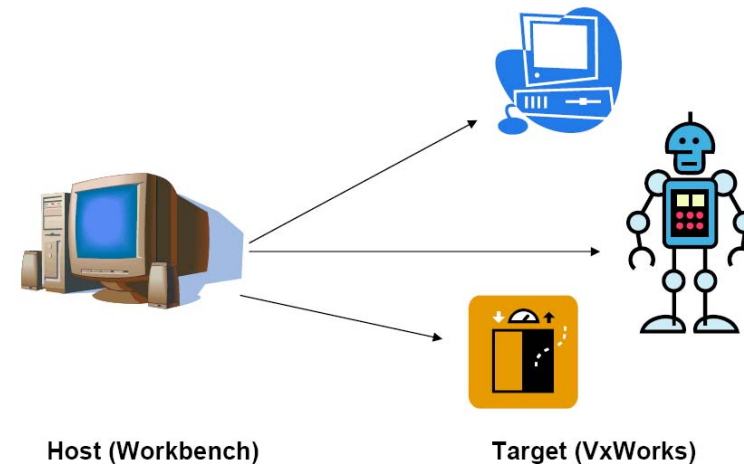


Echtzeitbetriebssysteme

VxWorks

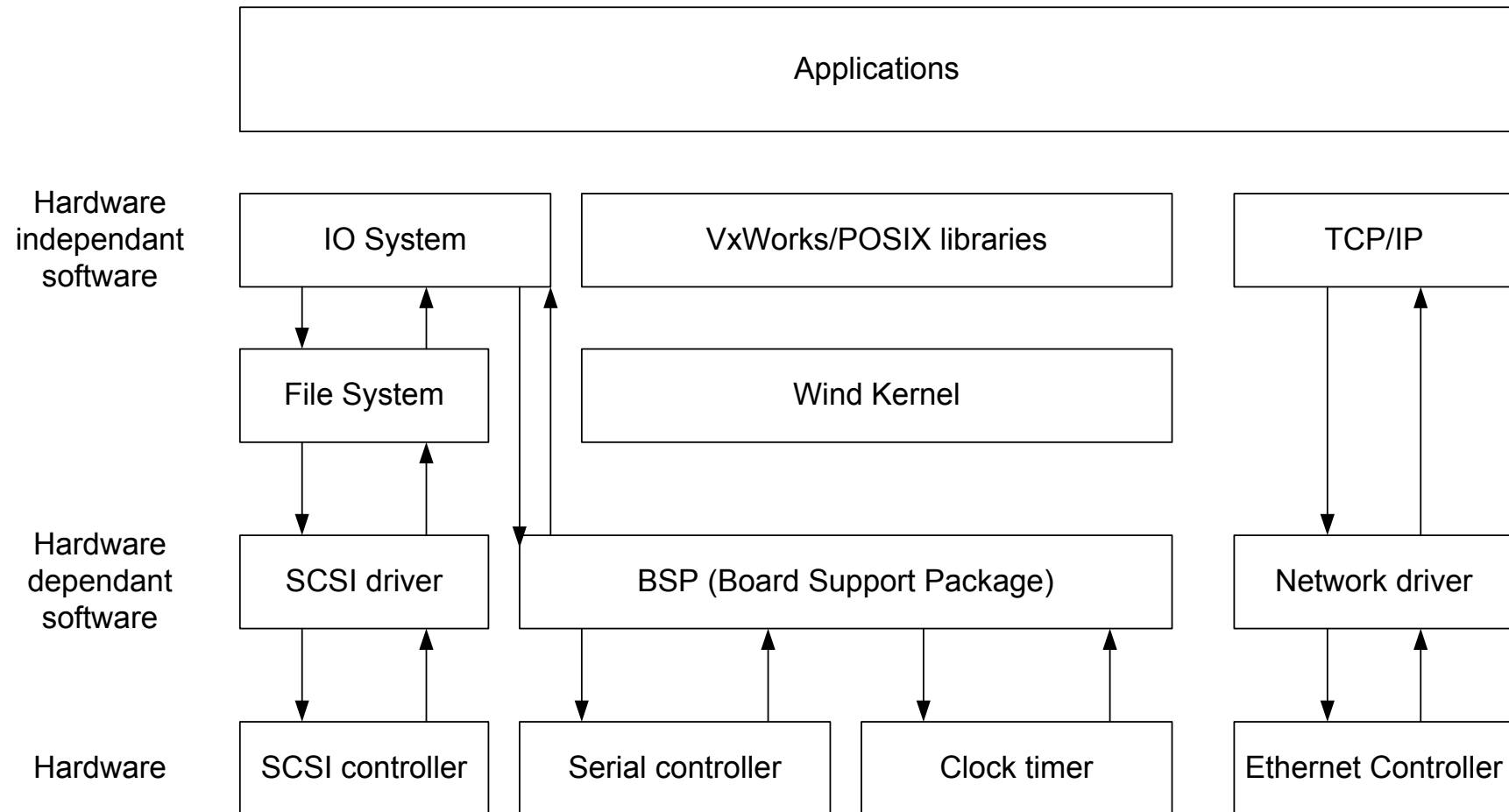
Eigenschaften

- Host-Target-Entwicklungssystem
- Eigene Entwicklungsumgebung
Workbench mit
Simulationsumgebung und
integriertem Debugger basierend
auf Eclipse
- Zielplattformen der Workbench 2.0:
VxWorks, Linux Kernel 2.4/2.6
- Auf der Targetshell wird auch ein
Interpreter ausgeführt \Rightarrow C-Code
kann direkt in die Shell eingegeben
werden
- Kernel kann angepasst werden,
allerdings muss der Kernel dazu
neu kompiliert werden
- Marktführer im Bereich der
Echtzeitbetriebssysteme





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)
 - Nachrichtenwarteschlangen
 - Signale
- Seit Version 6.0 wird zudem Speichermanagement angeboten:
 - Der Entwickler kann Benutzerprozesse mit eigenem Speicherraum entwickeln.
 - Bisher: nur Threads im Kernel möglich.

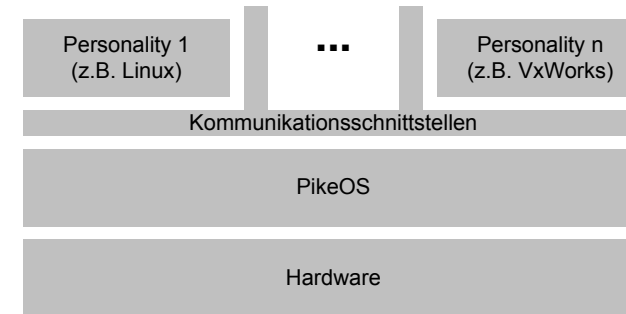


Echtzeitbetriebssysteme

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



Architektur



Echtzeitbetriebssysteme

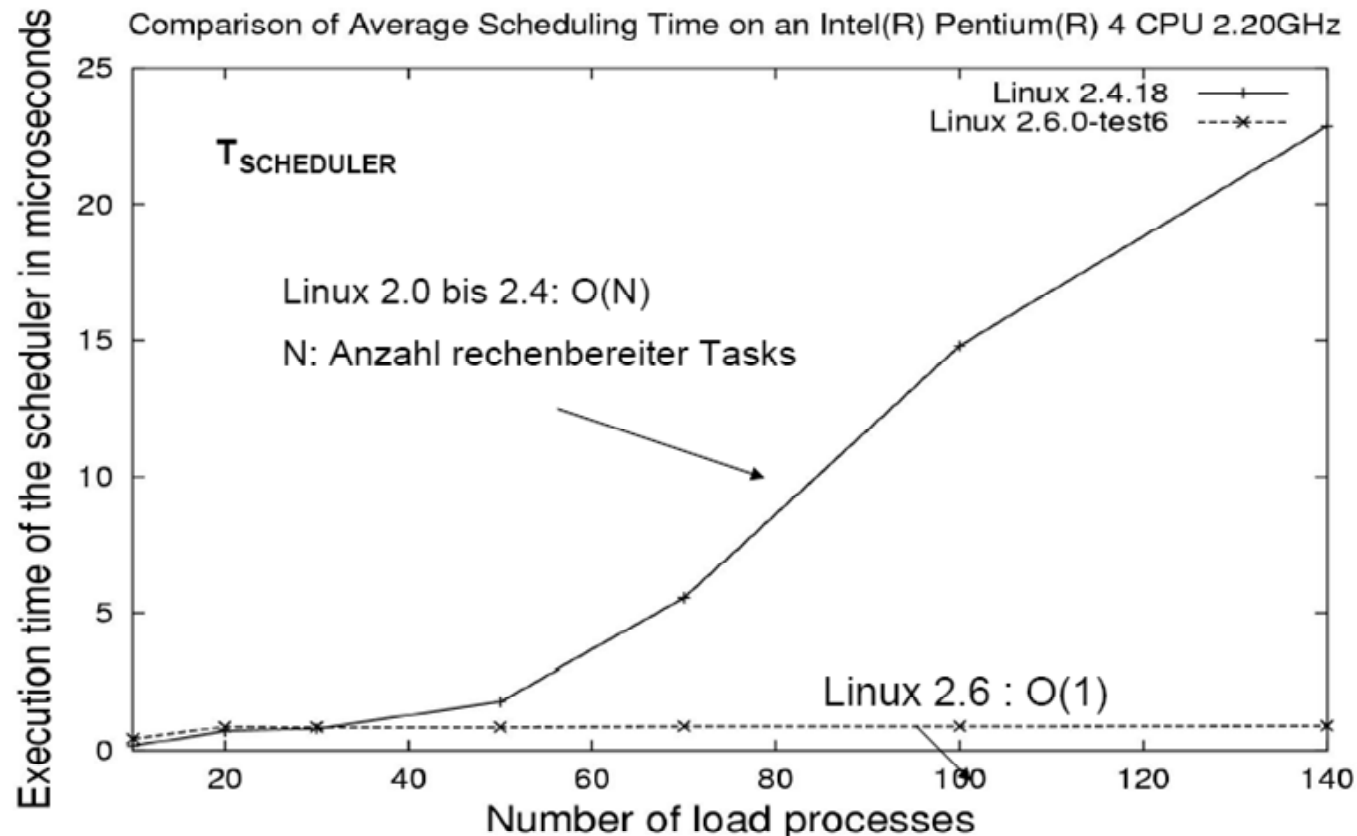
Linux Kernel 2.6



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.
⇒ 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 ⇒ 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>