Systemsoftware

in Sensornetzwerken

Referent: Bernhard Stähli
Betreuer: Oliver Kasten

Übersicht

- Einleitung
  - Aufgaben von Systemsoftware
  - Womit haben wir es zu tun?
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtual Machines in Sensornetzen

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
  - TinyOS Design
  - Komponenten
  - Zusammenbauen von Komponenten
  - Vergleich mit bestehenden Systemen
- Virtuelle Maschinen in Sensornetzen

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen
  - Maté
  - Code Capsules
  - ein einfaches Maté Programm

Übersicht

- Einleitung
  - Aufgaben von Systemsoftware
  - Womit haben wir es zutun?
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen

Definition

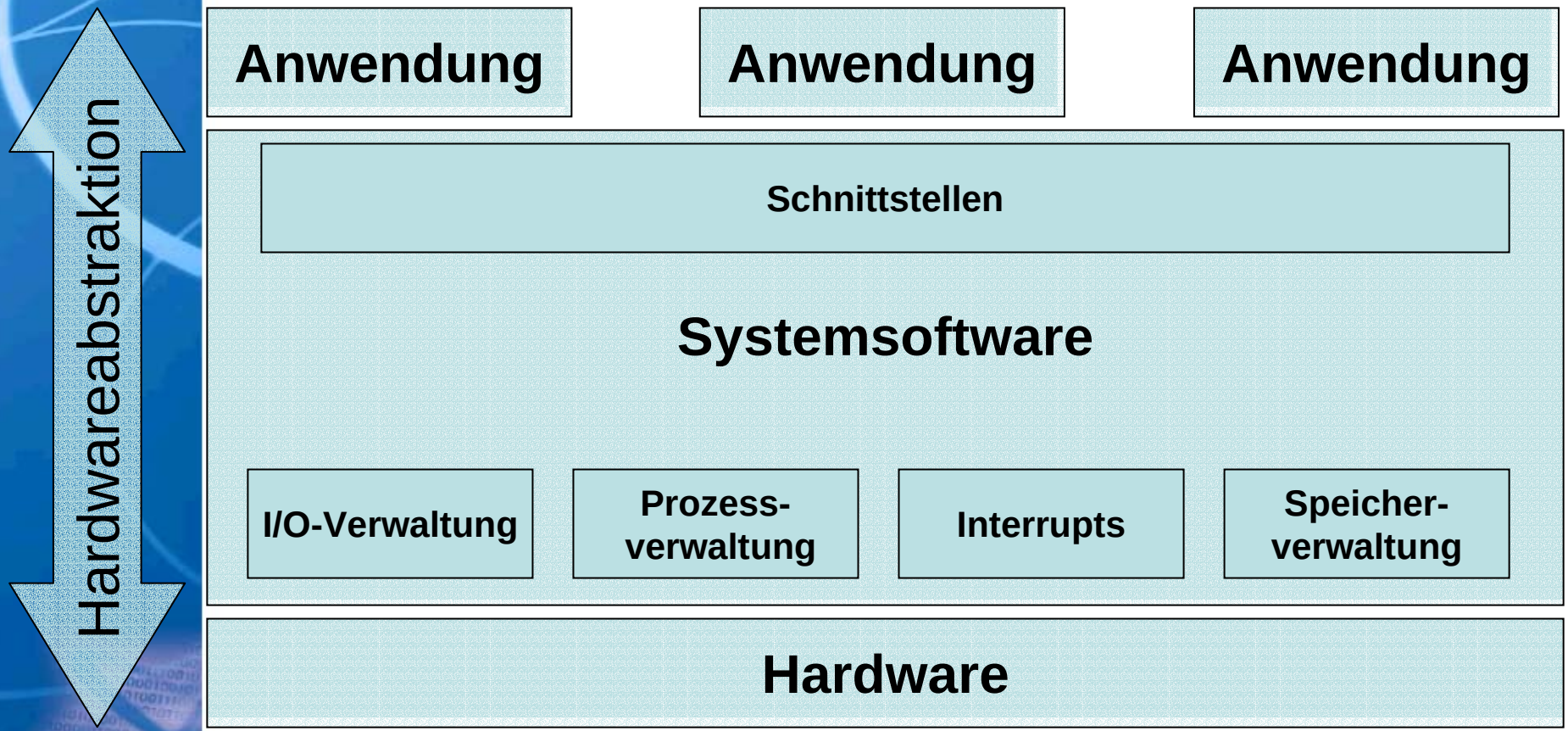
„Unter einem Betriebssystem versteht man all diejenigen Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften dieser Rechen-anlage ***die Basis*** *der möglichen Betriebs-arten des digitalen Rechensystems bilden und die insbesondere die* ***Abwicklung von Programmen*** *steuern und überwachen."*

(DIN 44300)

Aufgaben von Systemsoftware (I)

- verwalten von Ressourcen eines Rechners
  - CPU Zeit
  - Hauptspeicher
  - Festplattenspeicher
  - Kommunikationsmittel
- konkurrierende Prozesse
- Ressourcen fair und effizient verteilen
- Basis für Anwendungsprogramme

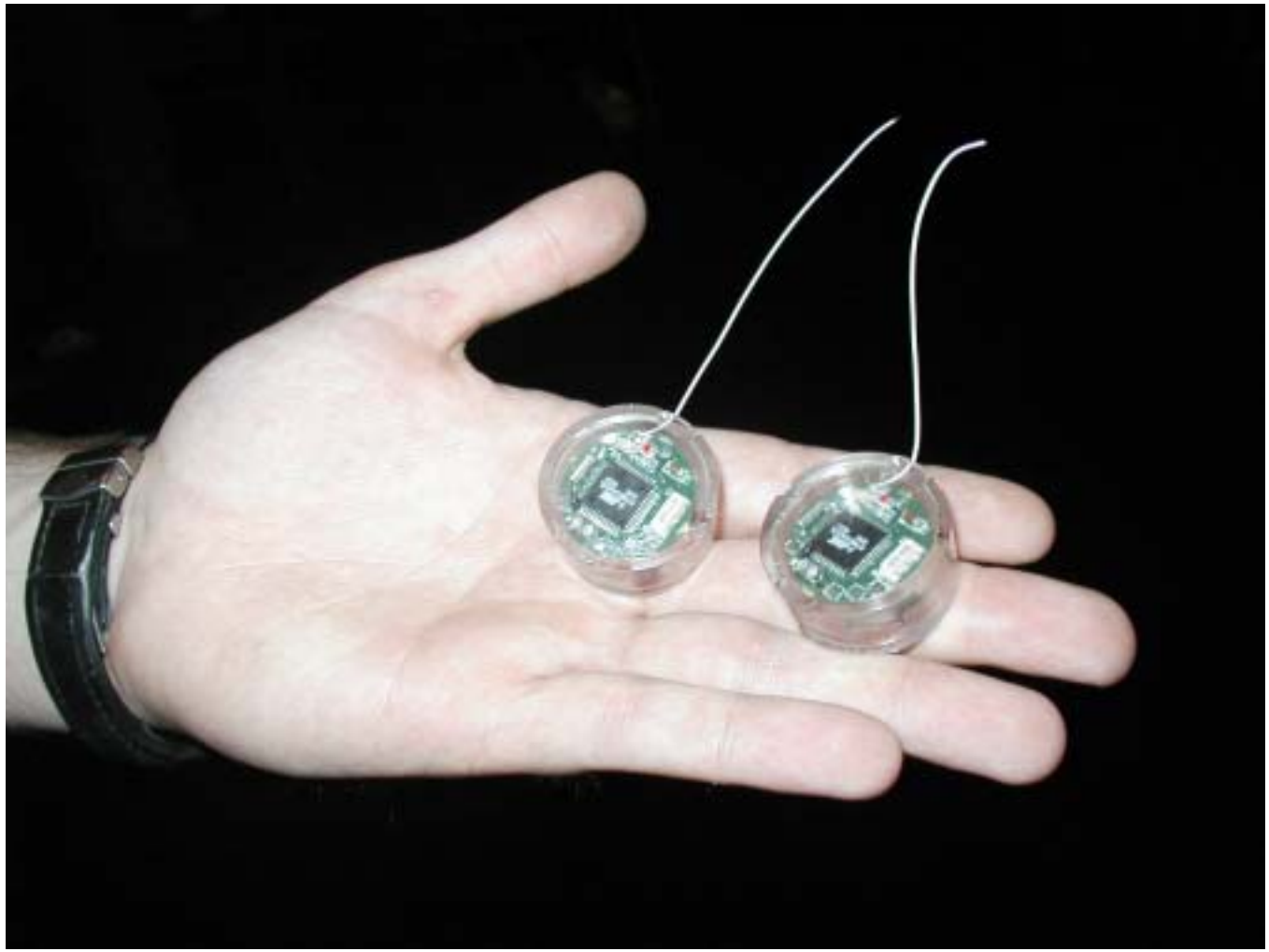
Aufgaben von Systemsoftware (II)



Übersicht

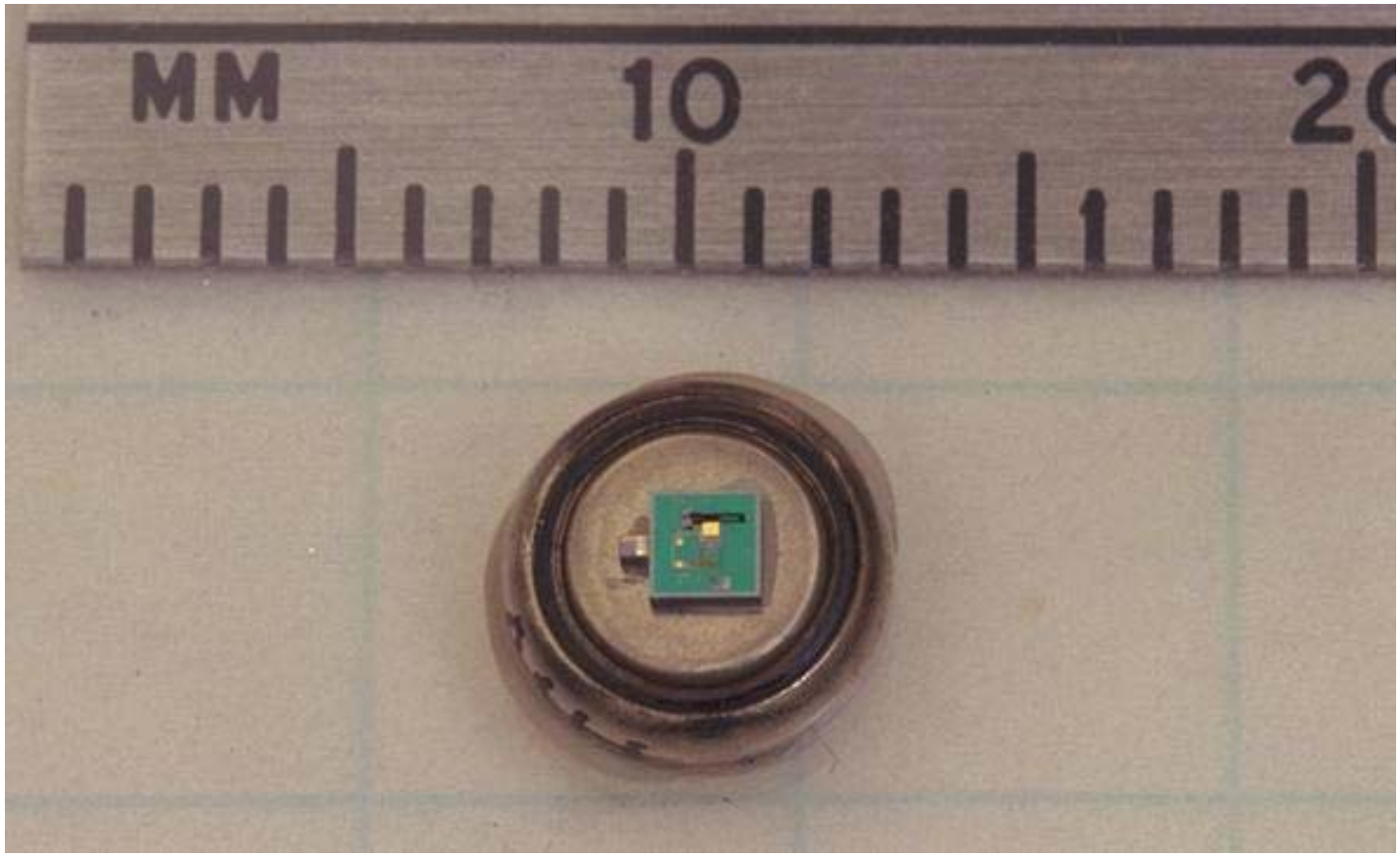
- Einleitung
  - Aufgaben von Systemsoftware
  - Womit haben wir es zu tun?
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen

Womit haben wir es zu tun?



Berkeley Universety of California

Womit haben wir es zu tun?

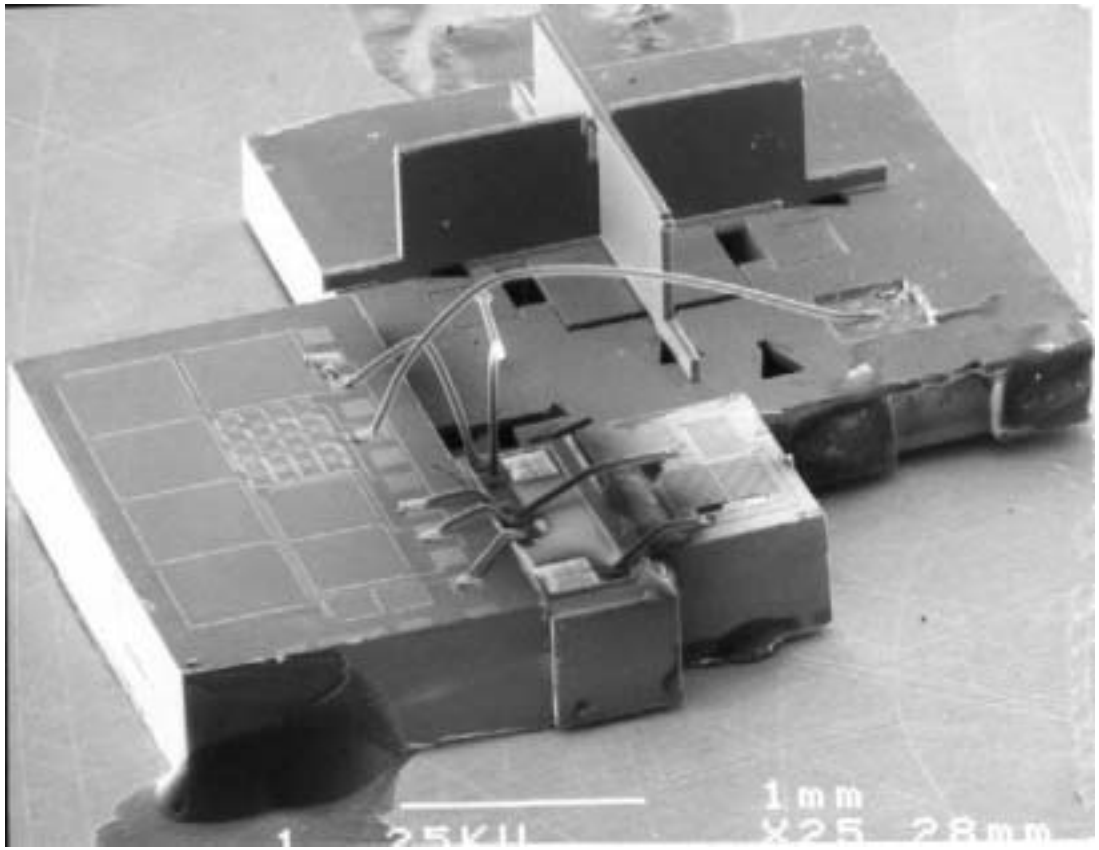


Daft Dust: Bi-direktionale Kommunikation

Womit haben wir es zu tun?

Solargetriebener Mote mit bi-direktionaler Kommunikation.
Enthält Beschleunigungs und Lichtsensor.

Womit haben wir es zu tun?



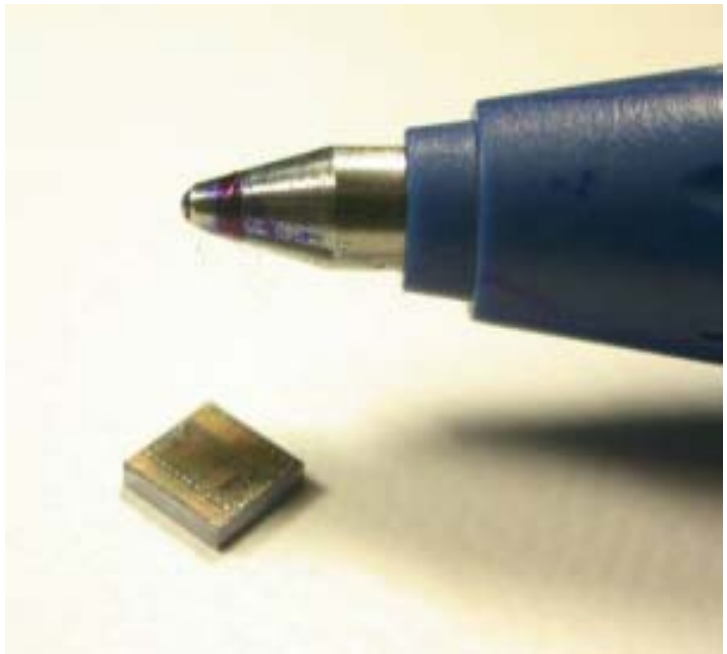
Womit haben wir es zu tun?

Berkeley Universety of California

Specs

Berkeley Universety of California

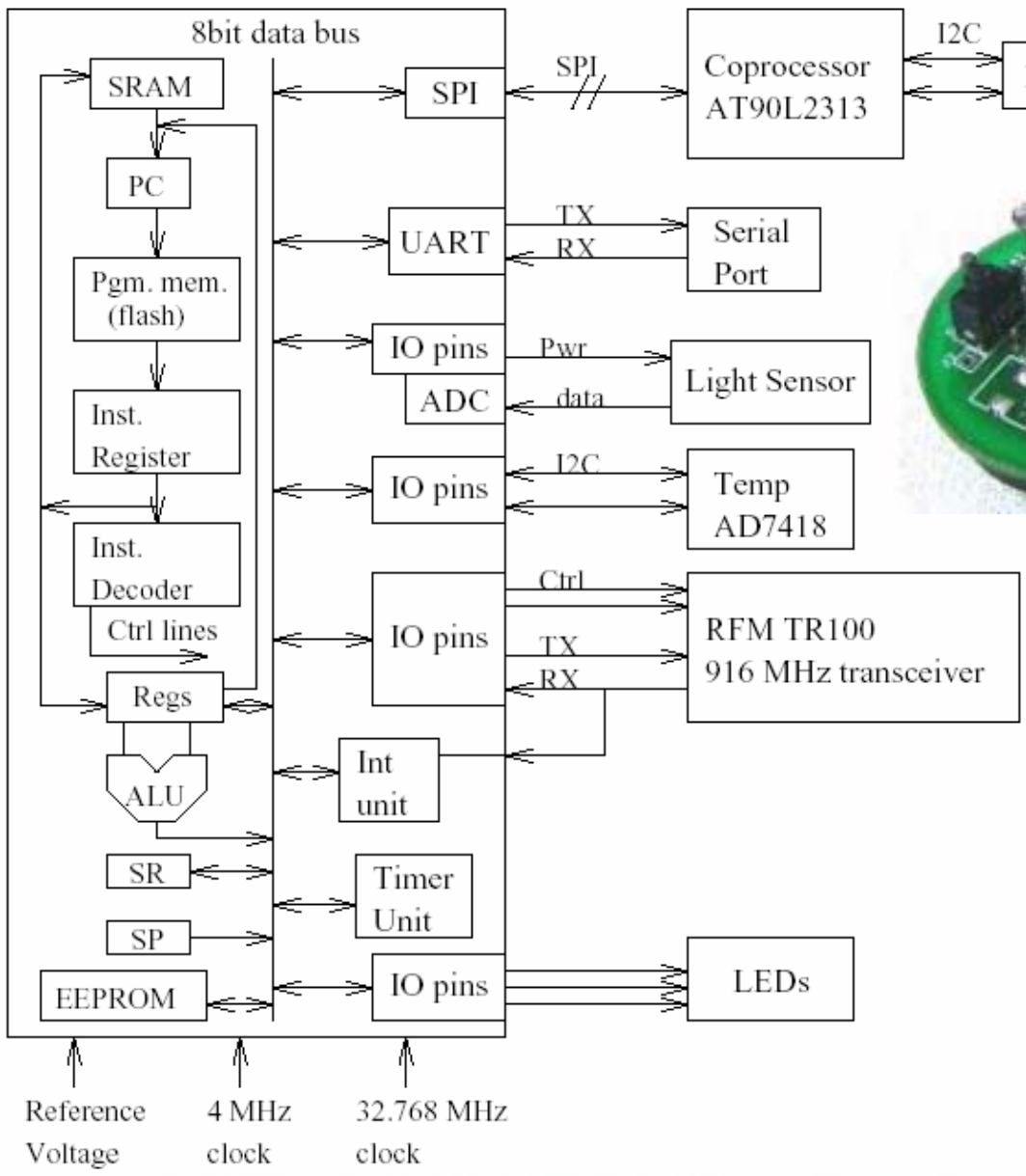
Specs



Berkeley Universety of California

March 6, 2003:
„The first successful test of Spec,
the next generation Mote, is performed.“

Aufbau eines Knotens



Technische Daten

- 4 MHz getakteter Prozessor bei 3 V
- 32 Register
- 8 kB Flashspeicher für den Programmcode
- 512 Bytes Datenspeicher
- 3 Energie-Zustände:
  Idle, Power-Down, Power-Save
- Radio-Kommunikation bei 916.50 MHz
- Bandbreite: 19.2 kbps
- 8 externe Sensoren über einen Bus

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen

Anforderungen an Systemsoftware

- kleiner Speichrbedarf
- Energieeffizienz
- Concurrency
- Vielfalt im Design und Anwendungsbereich
- Fehlertoleranz

Speicherbedarf und Energieeffizienz

- weinig Speicherbedarf
  ⇨ Grössenordnung 3 kB Bytes für das Betriebssystem
  ⇨ Windows XP ~ 1048576 kB

- Rechenkapazität eingeschränkt
  ⇨ der Prozessor hat eine kleine Taktrate (~4 MHz) und wird wenn möglich ausgeschaltet

Concurrency

- viele parallele Datenflüsse
  - ⇨ Kommunikation
    - senden
    - empfangen
  - ⇨ verschiedene Sensormessungen
  - ⇨ Datenverarbeitung
- wegen begrenztem Speicher
  - ⇨ Puffern der Daten ist kaum möglich
    - ⇨ sofort abarbeiten
- Schichtenmodell des Betriebssystems
  - ⇨ Datenflüsse zwischen den Schichten

Vielfalt im Design und Anwendungsbereich

- grosse Anzahl von Hardwareausprägungen
  - ⇨ verschiedene Hersteller
  - ⇨ verschiedene Sensoren
  - ⇨ verschiedene Anwendungsbereiche
- muss modular aufgebaut sein
  - ⇨ kein general-purpose System (Desktop) (Energieeffizienz)

⇨ das gewünschte System wird aus einem Spektrum von Modulen aufgebaut

Fehlertoleranz

- Knoten sind meist unzugänglich
  ⇨ fehlerhafte Programme ⇨ kein „Absturz“
- Ressourcen fehlen für Redundanz
  ⇨ Redundanz knotenübergreifend realisieren
  ⇨ Ausfall einzelner Knoten kann toleriert werden
- OS soll es erlauben, zuverlässige Programme zu schreiben

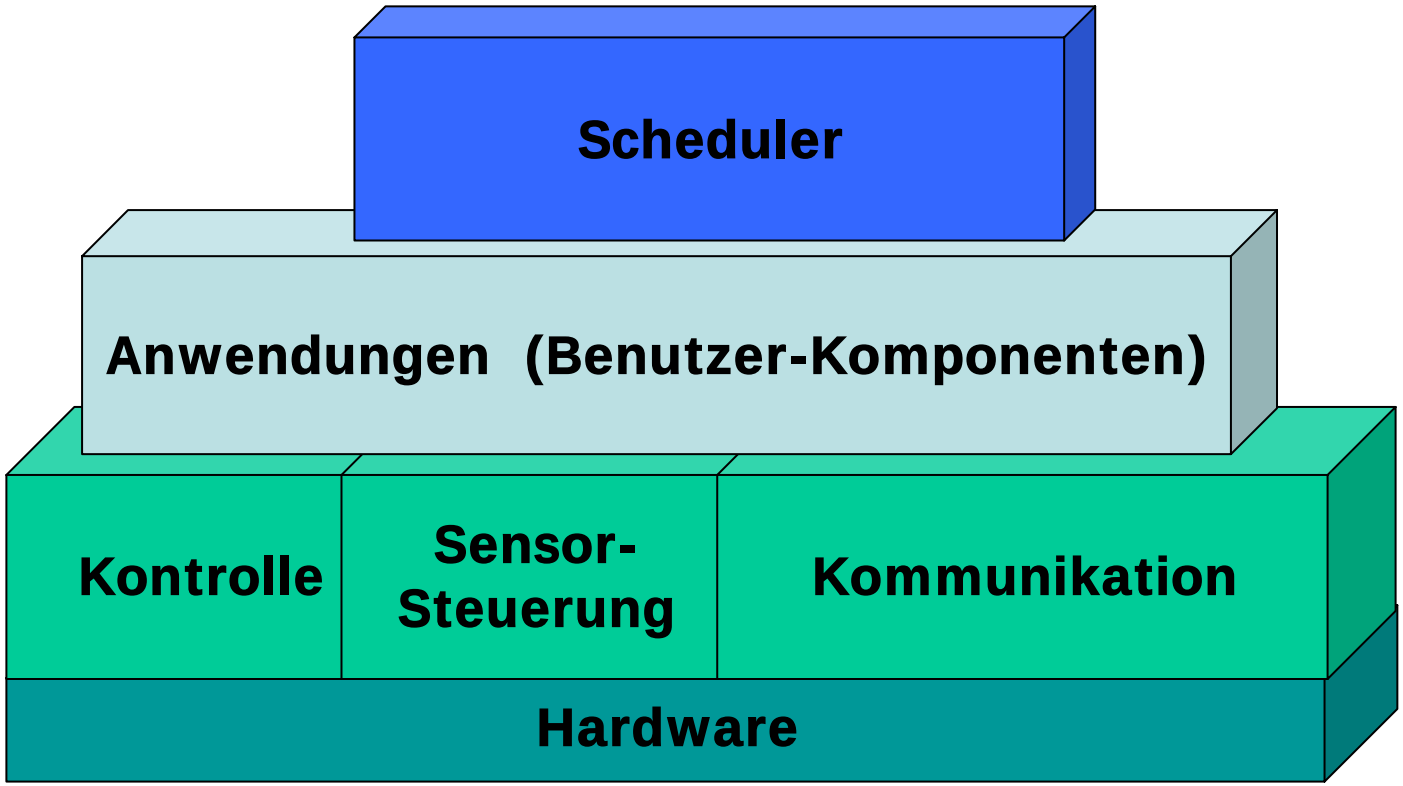
Tiny OS

- entwickelt an der Berkeley University of California
- versucht den Anforderungen von Sensornetze gerecht zu werden
- eventbasiert ⇨Concurrency
- Komponentenbasiert ⇨ Modularität

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
  - TinyOS Design
  - Komponenten
  - Zusammenbauen von Komponenten
  - Evaluation
- Virtual Machines in Sensornetzen

TinyOS Design



Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
  - TinyOS Design
  - Komponenten
  - Zusammenbauen von Komponenten
  - Evaluation
- Virtuelle Maschinen in Sensornetzen

- **System**
  - Scheduler
  - {Komponente}
- **Komponente**
  - {Command Handler}
  - {Event Handler}
  - Frame fixer Grösse
  - {Tasks}

- **System**
  - Scheduler
  - {Komponente}
- **Komponente**
  - {Command Handler}
  - {Event Handler}
  - Frame fixer Grösse
  - {Tasks}

Scheduler

- einfacher FIFO Scheduler
- prioritätsbasierte Scheduler sind in Planung

- **System**
  - Scheduler
  - {Komponente}
- **Komponente**
  - {Command Handler}
  - {Event Handler}
  - Frame fixer Grösse
  - {Tasks}

Command

- nicht-blockierende Anfrage an Komponenten in unteren Schichten
- Command Handler gründet einen Task für die Ausführung
- Rückmeldung wird durch Event signalisiert

- **System**
  - Scheduler
  - {Komponente}
- **Komponente**
  - {Command Handler}
  - {Event Handler}
  - Frame fixer Grösse
  - {Tasks}

Event

- Signalisiert einer oberen Schicht ein Ereignis
- Event-Handler werden aufgerufen, um auf Hardwareevents bzw. Events von unteren Schichten zu reagieren

- **System**
  - Scheduler
  - {Komponente}
- **Komponente**
  - {Command Handler}
  - {Event Handler}
  - Frame fixer Grösse
  - {Tasks}

Frame

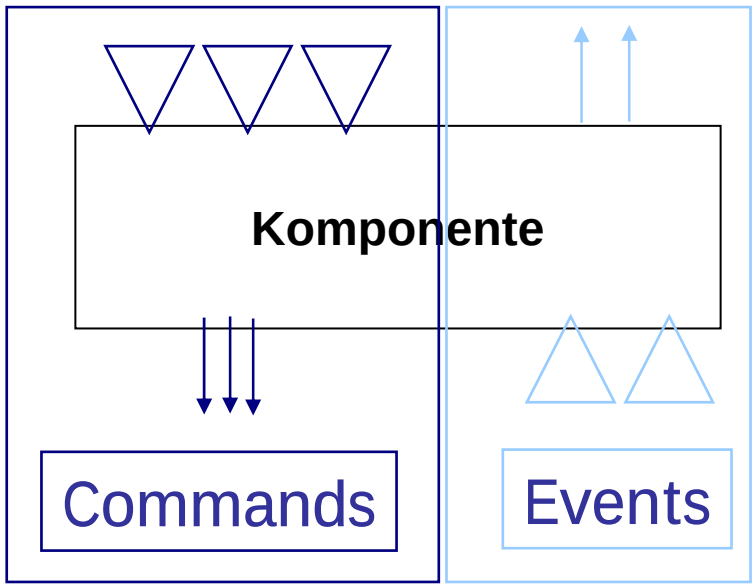
- Speicherbereich fester Grösse
- keine dynamische Allokation

- **System**
  - Scheduler
  - {Komponente}
- **Komponente**
  - {Command Handler}
  - {Event Handler}
  - Frame fixer Grösse
  - {Tasks}

Task

- sind Prozesse die der Scheduler verwaltet
- führen die eigentliche Arbeit aus
- Tasks können quasi parallel laufen
- kommunizieren asynchron durch Events

Komponente



Komponententypen

Hardware Abstraktionen

- bilden Hardware ins Komponentenmodell ab
- **Beispiel**: RFM (radio frequency module)

Synthetische Hardware

- simuliert das Verhalten von komplexer Hardware
- **Beispiel**: Radio Byte Komponente

High Level Software Komponenten

- kontrollieren andere Komponenten
- berechnen das Routing
- Datentransformationen
- **Beispiel**: Messaging Module

Beispiel
Messaging Komponente

- sendet und empfängt Nachrichten
- zerstückelt Nachrichten in Pakete
- liefert Pakete an Paket Schicht
- setzt Pakete zu Nachrichten zusammen

Interface Messaging Komponente

```
//angebotene Commands
char TOS_COMMAND(send_msg)(int addr,int type, char*
   data);
void TOS_COMMAND(powermode)(char mode);
char TOS_COMMAND(init)();
```

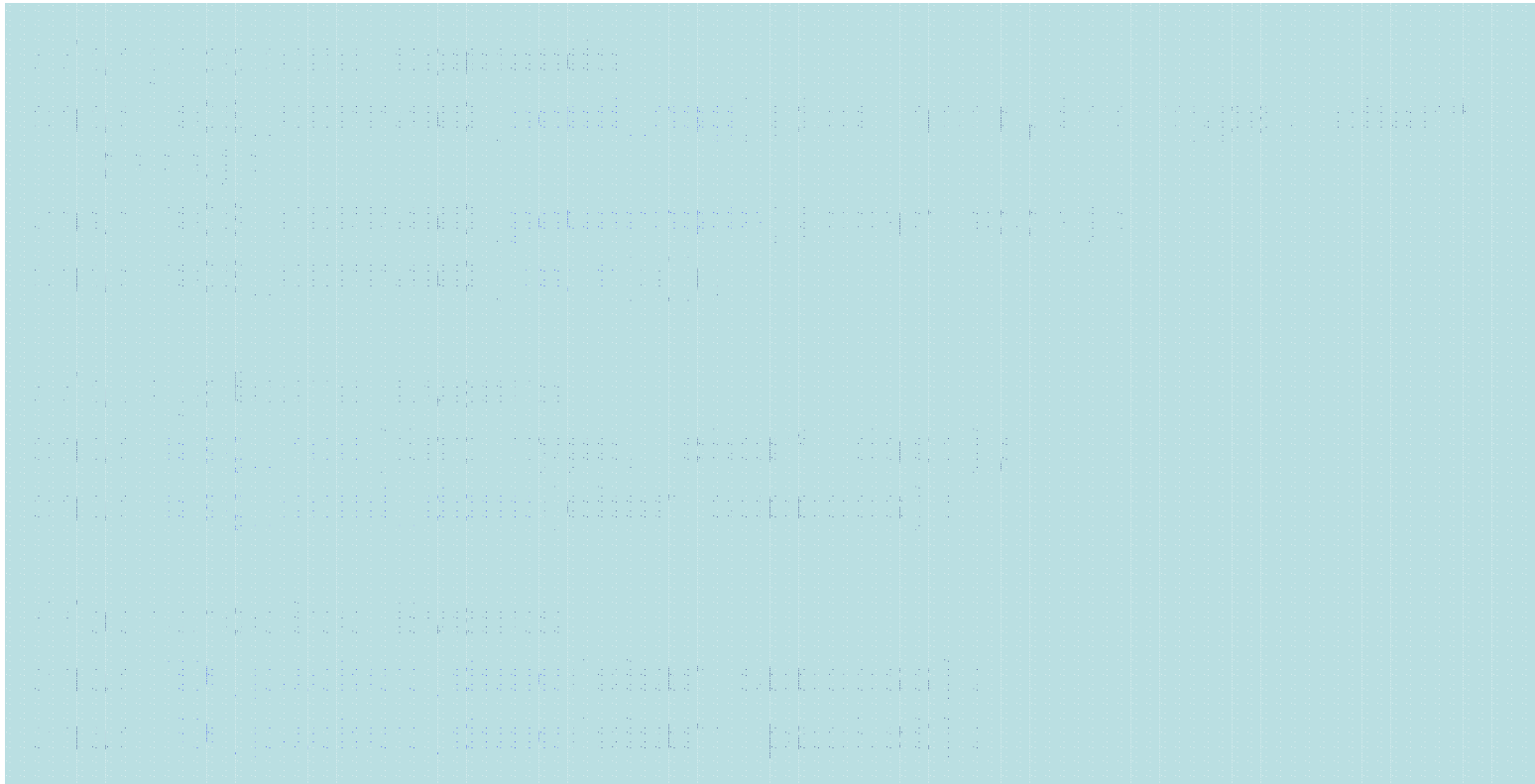
Interface
Messaging Komponente

```
//ausgelöste Events:
char msg_rec(int type, char* data);
char msg_send_done(char success);
```

Interface Messaging Komponente

```
//behandelte Events:
char TX_packet_done(char success);
char RX_packet_done(char* packet);
```

Interface
Messaging Komponente

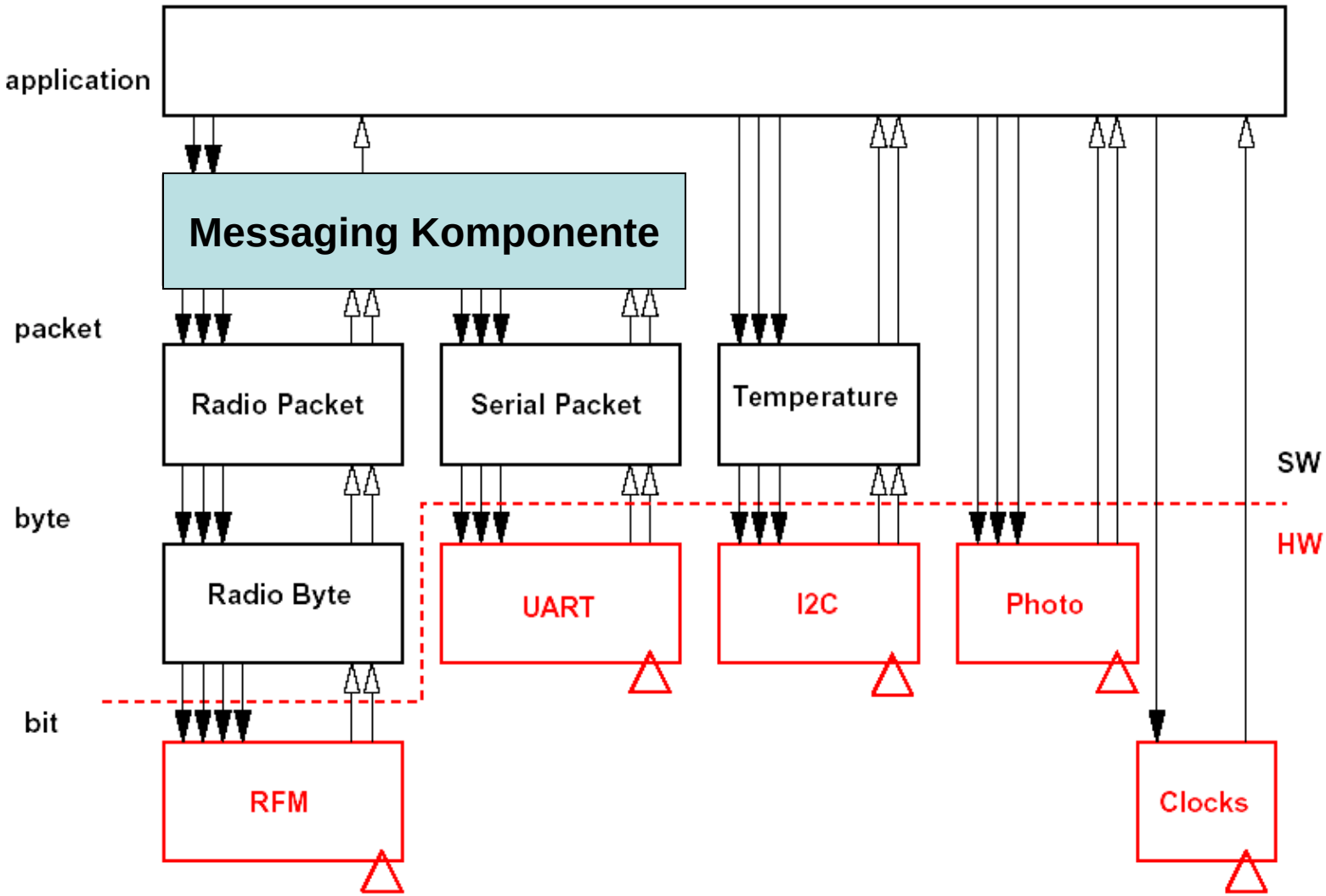


```
//aufgerufene Commands:
char TOS_COMMAND(TX_packet)(char* data);
void TOS_COMMAND(powermode)(char mode);
char TOS_COMMAND(init)();
```

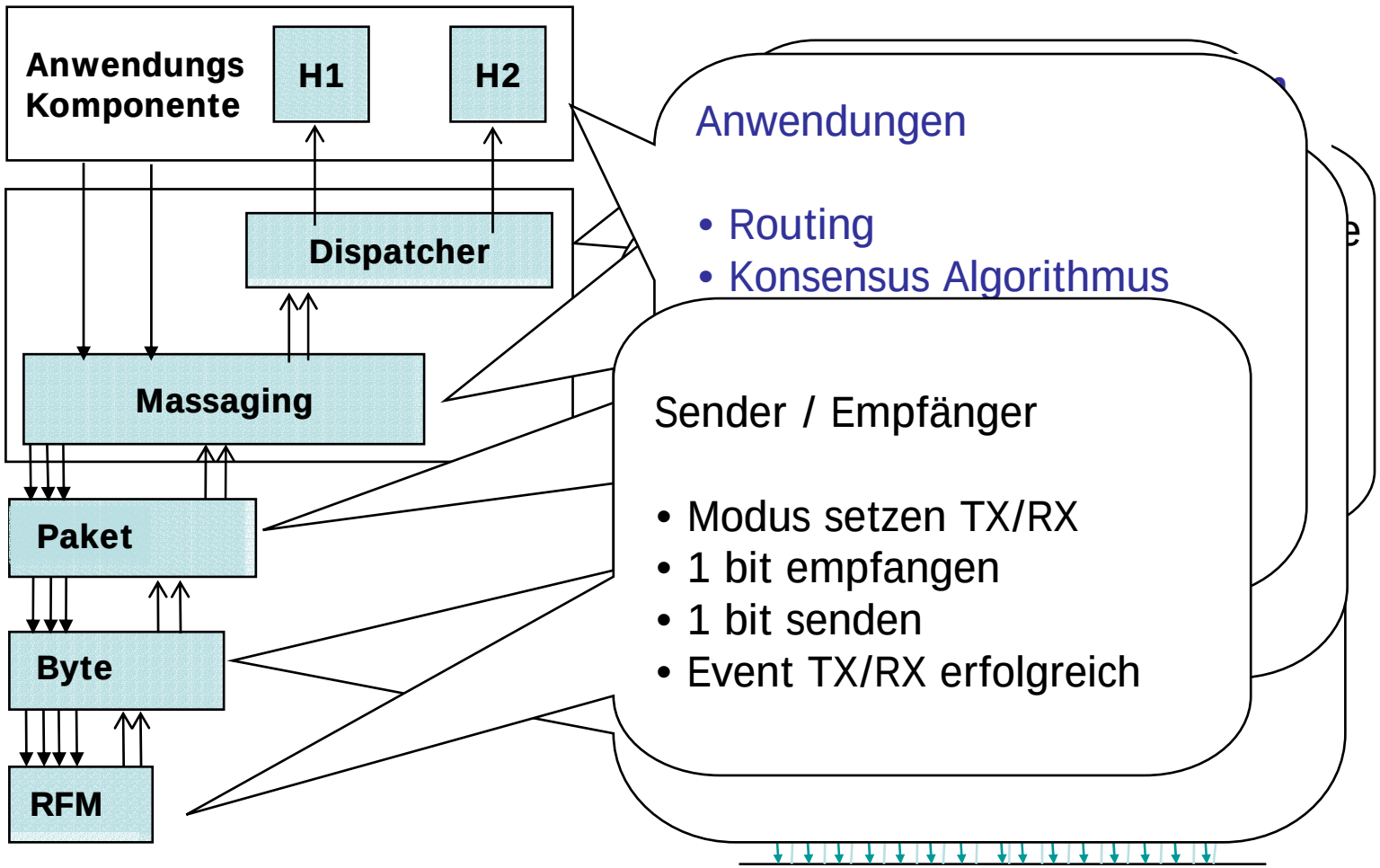
Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
  - TinyOS Design
  - Komponenten
  - Zusammenbauen von Komponenten
  - Evaluation
- Virtual Machines in Sensornetzen

Komponenten



Kommunikationsablauf



Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
  - TinyOS Design
  - Komponenten
  - Zusammenbauen von Komponenten
  - Evaluation
- Virtual Machines in Sensornetzen

Speicherbedarf

Komponente	Code Grösse (Bytes)	Daten Grösse (Bytes)
Multihop Router	88	0
Dispatcher	40	0
Nachrichten Temp.	78	32
Nachrichten Licht	146	8
Messaging Komp.	356	40
Paketschicht	334	40
Radio Byte	810	8
RFM	310	1
Lichtsensor	84	1
Temperatur	64	1
UART	196	1
UART Pakete	314	40
I2C Bus	198	8
Prozessor Init.	172	30
TinyOS Scheduler	178	16
C Runtime	82	0
Total	3450	226

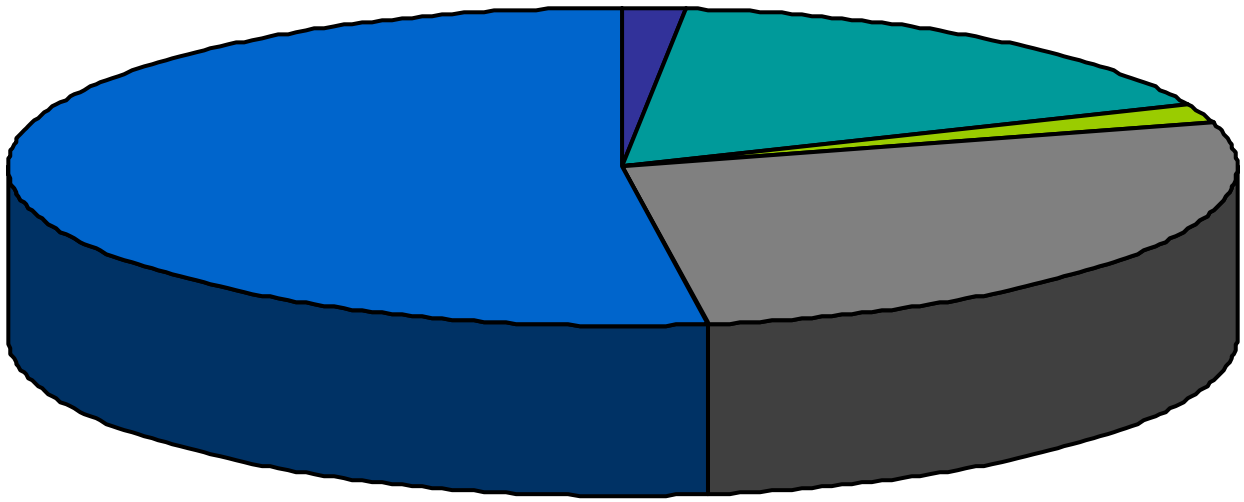
Concurrency

- wichtigster Aspekt bei concurrency-intensiven Anwendungen:
  - ⇨ Zeit für Kontextwechsel (6 Bytecopy)
  - ⇨ Event auslösen (1 Bytecopy)
- Interruptbehandlung ist teuer (9 Bytecopy)
  - ⇨ Abspeichern und Widerherstellen der Register
  - ⇨ Register Windows
    Aufteilung der Register auf Tasks

Parallelität

Komponenten	Paket empfangen	% CPU Bedarf
AM	0.05%	0.02%
Paketschicht	1.12%	0.51%
Radio Handler	26.87%	12.16%
Radio Decodier-Task	5.48%	2.48%
RFM	**66.48%**	30.08%
Empfang	-	-
Idle	-	**54.75%**
Total	**100.00%**	**100.00%**

CPU Bedarf: Senden



AM
Paketschicht
Radio Handler
Radio Codier Task
RFM
Senden
Idle

Parallelität

- Sogar während des Empfangens ist die CPU 50 % idle
- die CPU ist nicht der Flaschenhals im System
- die Bandbreite ist beschränkt
- idle Zeit der CPU sinnvoll nutzen
  ⇨ Concurrency

Nachteile von bestehenden embedded systems OS

- zu leistungsfähige Prozessoren nötig
- Speicherbedarf zu massiv
- zu lange um Kontext zu wechseln

⇨ TinyOS hat deutliche Vorteile in den genannten Bereichen

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen
  - Maté
  - Code Capsules
  - ein einfaches Maté Programm

Wieso eine (neue) VM ?

- unterschiedliche Hardware
- stetig weiterentwickelt
- bestehende VMs benötigen ~160 kB ROM

  ⇨ es stehen nur 8 kB zur Verfügung
- Energieaspekte
- Programme auf den Sensorknoten von aussen installieren

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen
  - Maté
  - Code Capsules
  - ein einfaches Maté Programm

Maté

- entwickelt an der Berkeley Universety of California
- Bytecode Interpreter auf TinyOS
- ist eine TinyOS Komponente

Aufbau von Maté (I)

- Maté Instruktionen abstrahieren die Asynchronität von TinyOS

  ⇨ **send**: warten auf ACK
  ⇨ **sense**: Task schläft, bis Daten anliegen
  ⇨ Pragrammieren ist weniger fehleranfällig

- komplexe Instruktionen
  ⇨ bis zu 20‘000 CPU Zyklen

Aufbau von Maté (II)

- Instruktionen = TinyOS Task
- Subroutinen: Rücksprungadresse auf Returnstack
- Operandenstack
- gemeinsamer Speicherbereich für Interprozesskommunikation

  `gets/puts` (1 Word)

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen
  - Maté
  - Code Capsules
  - ein einfaches Maté Programm

Maté Capsule

- enthält 24 Instruktionen
- passt genau in ein TinyOS Paket
- Versionsnummer ist enthalten

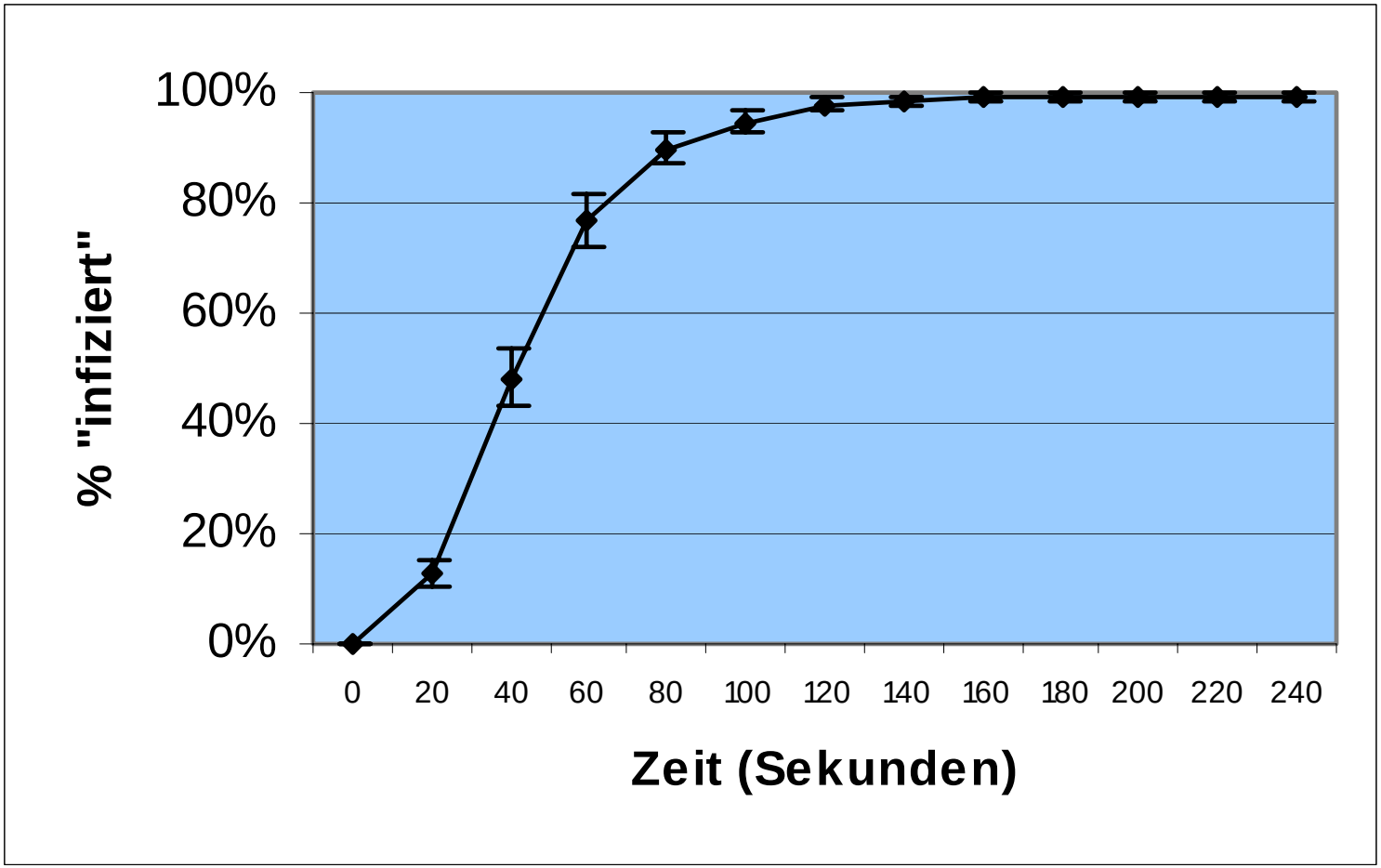
Maté Capsules

- Capsules sind Events zugeordnet
- fehlerhafte Capsules verursachen keinen Systemabsturz
- Subroutine-Capsules
  ⇨ für Programme > 24 Instruktionen

Viraler Code

- Maté installiert neuere Version, die es vom Netz empfängt
  - ⇨ Reaktion auf einen Event wird verändert
- Motes können ihre Capsules weiterleiten (lokaler Broadcast)
  - ⇨ `forw`

Code Ausbreitung



Vergleich Bytecode-Native

Operation	Maté #CPU Zyklen	Native #CPU Zyklen	Verhältnis
Einfach `and`	469	14	**33.5 : 1**
OS Call `rand`	435	45	9.5 : 1
Sensoren `sense`	1342	396	3.4 : 1
Komplex `send`	685+~20'000	~20'000	**1.03 : 1**

Energie Tradeoff
Bsp: Great Duck Island

- einfacher „sense – send“ Loop
- läuft alle 8 Sekunden
- 19 Maté Instruktionen
- 8 kB binären Code
- Energie-Tradeoff:

  ⇨ wenn die **Applikation** < 6 Tage läuft, spart Maté Energie

Übersicht

- Einleitung
- Anforderungen an die Systemsoftware in SN
- Anwendungsbeispiel: TinyOS
- Virtuelle Maschinen in Sensornetzen
  - Maté
  - Code Capsules
  - ein einfaches Maté Programm

Beispielprogramm

- Lichtsensor auslesen
- wenn

  |Sensorwert – letzter Wert| > 32

  ⇨ neuen Wert senden

Besipielprogramm

```
00    pushc 1          # 1 auf Operandenstack pushen
01    sense            # Sensor 1 auslesen (Licht)
02    copy             # Sensorwert auf Stack
03    gets             # zuletzt gesendeter Wert auf Stack
04    inv              # letzten Wert invertieren
05    add              # dif = Sensorwert + (–letzter Wert)
06    pushc 32
07    add              # dif = dif + 32
08    blez 17          # Sensorwert zu klein: springe zu send
09    copy             # Sensorwert auslesen
10    inv              # Sensorwert Wert invertieren
11    gets             # zuletzt gesendeter Wert auf Stack
12    add              # dif = -Sensorwert + letzter Wert
13    pushc 32
14    add              # dif = dif + 32
15    blez 17          # Sensorwert zu gross: springe zu send
16    halt
```

Besipielprogramm

```
17    copy          # Sensorwert auslesen
18    sets          # neuen Wert abspeichern
19    pushm         # Nachricht auf Operandenstack
20    clear         # Nachrichteninhalt löschen
21    add           # Nachrichteninhalt reinkopieren
22    send          # Nachricht versenden
23    halt
```

Referenzen

- [1] Philip Levis and David Culler: Mate: A Tiny Virtual Machine for Sensor Networks
- [2] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister:System Architecture Directions for Networked Sensors
- [3] Phillip Stanley-Marbell and Liviu Iftode: Scylla: A Smart Virtual Machine for Mobile Embedded Systems
- [4] TinyOS Webseite:http://webs.cs.berkeley.edu/tos