Betriebssysteme für Wireless Sensor Network Motes

Markus Dauberschmidt
Betreuer: Christian Sauter
Seminar Sensorknoten: Betrieb, Netze und Anwendungen SS2011
Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitektur
Fakultät für Informatik, Technische Universität München
Email: daubersc@in.tum.de

KURZFASSUNG

In dieser Arbeit werden drei Betriebssysteme näher vorgestellt, die vor allem auf „Wireless Sensor Network Motes“ (WSN Mote), also auf Rechenknoten eines Sensor-Netzwerkes eingesetzt werden. Die vorliegende Arbeit geht hier auf die Besonderheiten und Unterschiede dieser Betriebssysteme ein und arbeitet die für WSNs wichtigen Aspekte heraus.

Schlüsselworte

Wireless sensor networks (WSN), Mikrocontroller, TinyOS, Contiki, cocoOS

1. EINLEITUNG

Diese Arbeit befasst sich mit Betriebssystemen für Wireless Sensor Network (WSN) Motes. Im Abschnitt „Wireless Sensor Networks“ wird beschrieben, was ein WSN bzw. ein „Mote“ ist und wofür WSNs eingesetzt werden können.

Im Abschnitt „Betriebssysteme für WSN-Motes“ wird diskutiert, warum überhaupt ein Betriebssystem auf einem WSN Mote sinnvoll ist, und welche Funktionen es bieten sollte.

In den Abschnitten „TinyOS“, „Contiki“ und „cocoOS“ werden die drei genannten Betriebssysteme näher vorgestellt. Dabei werden die jeweiligen Besonderheiten dargelegt, sowie die Aspekte, die besonders für WSNs von Belang sind.

Diese Arbeit schliesst mit einer Zusammenfassung der Besonderheiten der verschiedenen Betriebssysteme.

2. WIRELESS SENSOR NETWORKS

Das Forschungsgebiet der Wireless Sensor Networks, kurz WSNs, hat in den letzten Jahren immens an Bedeutung gewonnen. Das primäre Einsatzgebiet eines WSN ist es, Daten an verschiedenen geograpischen Punkten des Netzes zu erheben, ggf. weiterzuverarbeiten und dann konsolidiert an einen „Data-Sink“ bzw. Root-Knoten des Netzwerkes weiterzureichen [2]. Ein Knoten in einem solchen Netz werden durch „Motes“ gebildet (siehe Abb. 1), kleinen autarken System, bestehend aus einem Mikrocontroller, einem oder mehreren Sensoren, einer i.d.R. autarken Stromversorgung aus einer Batterie, Solarzelle oder einer „Energy Harvesting“ Einheit und einer Möglichkeit zur Datenübertragung und Kommunikation, i.d.R. einem Funksender. Seltener wird auch ein GSM, WLAN oder Ethernet-Modul zur Kommunikation verwendet. Ist es nicht möglich, die Daten weiter zu senden, so sollte der Mote in der Lage sein, die Messergebnisse lokal aufzuzeichnen und später erneut versuchen diese zu übertragen. Was ein Mote messen und übertragen soll, ist so individuell wie vielseitig. Es folgen drei Beispiele für den Einsatz eines WSNs.

Abbildung 1: Ein MICAZ Mote (Quelle: [18])

- In einem Industriekomplex wird eine Fertigungsstraße durch Sensorknoten überwacht. Die Knoten sitzen an neuralgischen Punkten der Anlage und senden regelmäßig Messdaten über Temperatur, Druck und Luftfeuchtigkeit der verschiedenen Fertigungsstationen an eine zentrale Messstation, die diese Daten aufbereitet und der Leitwarte zur Verfügung stellt. Die Sensoren können auch an unzugänglichen Stellen angebracht sein, an denen eine Verkabelung zu aufwändig wäre oder möglicherweise die Fertigungsprozesse stören könnte.

- In einem waldbrandgefährdeten Gebiet werden im Abstand von mehreren hundert Metern Motes an Baumstämmen befestigt. Diese messen die Luftfeuchtigkeit, Windgeschwindigkeit und Umgebungstemperatur und übertragen diese Daten zusammen mit ihrer eigenen Position an eine zentrale Station. Sollte durch große Trockenheit ein Feuer ausbrechen, schlagen die Sensoren Alarm. Mit Hilfe der Sensordaten kann sofort eine Übersicht über den Brandherd in der Zentrale erstellt werden, sowie die Ausbreitungsrichtung also auch die Geschwindigkeit live verfolgt werden (siehe [8]).

- In einem Naturpark in Afrika werden Zebras mit Motes versehen, die regelmäßig ihre aktuelle Position über einen GPS Sensor ermitteln und speichern. Die Daten werden hierbei über Wochen oder gar Monate gesammelt. Ein Austausch der gespeicherten Informationen erfolgt, sobald sich zwei mit Sensoren versehene Tiere

doi: 10.2313/NET-2011-07-1_04

nahe genug kommen. Dieses Projekt wurde unter dem Namen „ZebraNet“ [9] bekannt.

2.1 Mikrocontroller

Ein Mote beinhaltet in der Regel ein Rechensystem auf Basis eines Mikrocontrollers. Ein Mikrocontroller ist eine Recheneinheit, vergleichbar mit der CPU in einem herkömmlichen IT-System, nur ungleich einfacher. Im Allgemeinen sind es, im Gegensatz zu CPUs, keine „All-Purpose“-Recheneinheiten, sondern sind auf das Wesentliche reduziert, um möglichst stromsparend ihre eigentliche Aufgabe zu erledigen. Diese liegt zumeist im Steuer- und Regelbereich und darauf sind Mikrocontroller optimiert. Mikrocontroller finden sich heute überall in eingebetteten Systemen. Egal, ob in der Waschmaschine, in der Kaffeemaschine, im Handy oder der Fernbedienung für den Fernseher: überall arbeiten ein oder mehrere Mikrocontroller im Inneren.

Für die Programmierung von Mikrocontrollern kommen verschiedene Sprachen zum Einsatz: Für sehr echtzeitkritische Aufgaben kann die Programmierung hardwarenah in Assembler erfolgen, in der Regel wird jedoch C, selten auch C++ verwendet. Auch existieren bereits erste Lösungen, die mit den Sprachen des Microsoft .NET Micro Frameworks programmiert werden können ([21]). Auch wenn die Programmierung nicht in Assembler erfolgt, benötigt der Programmierer dennoch genaue Kenntnisse über die verwendete Mikrocontroller-Plattform, deren Möglichkeiten und Verhalten.

Die Kommunikation mit der Umwelt erfolgt über serielle Schnittstellen, USB, Funk, Netzwerkinterfaces oder entsprechende Anzeigeperipherie wie LEDs oder LCDs. Entweder ist die Unterstützung dieser Technologien von den Herstellern in dem jeweiligen Mikrocontroller von Haus aus vorgesehen, oder die entsprechenden Peripheriegeräte werden über einen eigenen Bus (z.B. SPI, i2C) angesprochen.

3. BETRIEBSSYSTEME FÜR WSN-MOTES

Die Knoten eines WSNs besitzen in der Regel einen oder mehrere Sensoren und sind durch Mikrocontroller bzw. daran angeschlossene Peripherie realisiert. Während die ersten WSNs nur statisch Daten sammelten und zu einem zentralen Gateway weiterleiteten, der die Daten dann agreggiert zur Weiterverarbeitung zur Verfügung stellte, erlauben es moderne WSNs auch, die einzelnen Knoten zu steuern, das Routing innerhalb des Netzwerkes zu ändern oder die Knoten zur Laufzeit mit einer neuen Steuersoftware zu versehen. Da WSN aus hunderten von Knoten bestehen können, kommt der Koordination der einzelnen Knoten und dem kontrollierten Abfragen bzw. Einsammeln von Daten eine zentrale Bedeutung zu.

Für die Kommunikation mit der Außenwelt gibt es eine Reihe von Protokollen und Standards. So sind aktuell z.B. IPv6 (implementiert in dem Protokoll „6LoWPAN“ [20]), oder Funk-Technologien wie 802.15.4 oder „ZigBee“ ([1]) ein Gebiet der aktiven Forschung.

Die oben genannten Aspekte machen deutlich, dass für eine effiziente Umsetzung einer Projekt- oder Produktidee sehr viel Breitenwissen rund um das Umfeld, in dem man sich bewegt, vorhanden sein muss. Es ist nicht ausreichend, nur die eigentliche Anwendung zu entwickeln, sondern man muss sich auch mit den Besonderheiten befassen, die die Entwicklung für WSNs nach sich zieht.

Daher ist es wenig verwunderlich, dass in den letzten Jahren, bedingt durch die immer mächtigeren Controller-Generationen, auch erste Betriebssysteme für Mikrocontroller entwickelt worden sind, die dem Entwickler einen Teil dieser Arbeiten abnehmen.

Bei den folgenden Punkten handelt es sich um wesentliche Anforderungen, die an Betriebssysteme im Bereich Wireless Sensor Networks gestellt werden (vgl. auch [16]):

- Das Betriebssystem sollte den Umstand berücksichtigen, dass nur sehr begrenzte Energieressourcen zur Verfügung stehen, Stromsparmodi des Controllers verwenden und nach dem Ausführen des Codes so schnell wie möglich wieder in einen Ruhezustand zu wechseln.
- Es sollte von der zugrundeliegenden Hardwareplattform abstrahieren (HAL=Hardware Abstraction Layer). Code, der für das Betriebssystem auf Plattform A geschrieben wurde, sollte nach Austauschen der HAL auch auf Plattform B verwendbar sein.
- Die Unterstützung für Netzwerkprotokolle sollte direkt im Betriebssystem vorhanden sein oder über Erweiterungen zur Verfügung stehen. Idealerweise unterstützt es Standards wie ZigBee oder TCP/IP und bietet die Nutzung für Anwenderprogramme über entsprechende Schnittstellen (zum Kernel) an.
- Das Betriebssystem sollte möglichst „light-weight“ sein, d.h. es sollte so viele Ressourcen wie möglich dem Anwenderprogramm zur Verfügung halten und selbst möglichst wenig Speicher verbrauchen.
- Mechanismen zur Synchronisation des Zugriffs auf geteilte Ressourcen sollten vom Betriebssystem bereitgestellt werden (z.B. Semaphore, Locks).
- Wichtigere Programmteile sollten gegenüber unwichtigeren Programmteilen bei der Abarbeitung bevorzugt werden (Task-Scheduling, unterschiedliche Schedulingverfahren). Idealerweise unterstützt die eingesetzte Zielplattform bereits hardware-seitig die Prioritätenvergabe, so dass diese nicht in Software nachgebildet werden muss.

In den folgenden drei Abschnitten werden die drei Betriebssysteme TinyOS, Contiki und cocoOS vorgestellt, welche explizit für den Einsatz auf Mikrocontrollern entwickelt wurden und Unterstützung für gängige Anforderungen im Bereich der Wireless Sensor Networks mitbringen.

3.1 TinyOS

TinyOS [30] ist das älteste der drei betrachteten Betriebssysteme. Es ist ein Open Source Betriebssystem und wurde an der Berkeley University von Dr. D. Culler für die dort genutzten WSNs mit Berkeley Motes entwickelt. Inzwischen unterstützt es aber auch etliche weiter Motes wie die Modelle epic, mulle und shimmer2. Es ist zu berücksichtigen,

doi: 10.2313/NET-2011-07-1_04

dass TinyOS standardmäßig *nur* auf den Plattformen eingesetzt werden kann, für die entsprechende Build-Umgebungen existieren! Es ist nicht ohne Weiteres möglich, TinyOS auf einem selbstgebauten Mote einzusetzen, selbst wenn dieser den gleichen Mikrocontroller verwendet, wie ein unterstützter Mote. [13] beschreibt, wie eine Portierung auf eine neue Hardware durchgeführt werden kann. Anwendungen können zunächst auch für den mitgelieferten TOSSIM Emulator compiliert und der Ablauf simuliert werden, bevor die Software produktiv auf die Motes aufgespielt wird.

Es bietet native Unterstützung für 6LoWPAN, dem IPv6 Layer für WSNs.

Die erste Version wurde im Jahr 2006 veröffentlicht. TinyOS ist aktuell in der Version 2.1.1 verfügbar. Auf der TinyOS Homepage steht ein VMWare Image zum Download bereit, in dem alle notwendigen Entwicklungstools bereits vorhanden und vorkonfiguriert sind. Möchte man die Entwicklungsumgebung direkt in ein existierendes System integrieren, so sind hierfür Anleitungen für die Plattformen Windows, Linux und MacOS vorhanden.

TinyOS war ursprünglich in einem C geschrieben, welches jedoch nicht besonders gut für das ereignisgesteuerte Programmiermodell geeignet war [11], das bei TinyOS zum Einsatz kommt. Aus der Notwendigkeit heraus, dieses Programmiermodell bestmöglich zu unterstützten, wurde eine neue Sprache mit dem Namen „nesC“ geschaffen, welche besser für die Programmierung von Sensorknoten geeignet ist [6]. TinyOS wurde daraufhin mit Release der Version 2 nochmals komplett in nesC neu programmiert.

nesC ist eine Sprache, in der die Sprachelemente nicht von Grund auf neu entwickelt wurden, sondern es handelt sich dabei um einen C-Dialekt. Das heisst, dass viele Dinge aus C übernommen wurden und nur spezielle Funktionen für die Besonderheiten in TinyOS (siehe unten) hinzugekommen sind. Man hat sich dagegen entschieden, diese Konzepte rein über Makros in C umzusetzen, da diese nur bedingt durch Compiler optimiert werden können [6]. Da es für nesC aber eigens entwickelte Compiler bzw. Tool-Chains gibt, können zum einen schon zur Compilezeit detaillierte Code-Überprüfungen vorgenommen werden, zum anderen auch viele Optimierungen realisiert werden [6].

TinyOS beinhaltet im Lieferungszustand bereits eine große Anzahl fertig nutzbarer Komponenten (wie z.B. Timer, Scheduler) und bietet auch Simulator-Tools und etliche Beispielanwendungen, die die verschiedenen Konzepte verdeutlichen.

Das Betriebssystem ist dafür gedacht, leichtgewichtige Anwendungen ablaufen zu lassen, wie sie für WSNs üblich sind. Das heisst, es geht um das Erfassen von Daten, ggf. noch deren Aufbereitung und die Weiterleitung der Daten.

TinyOS unterstützt die TI MSP430 (16 Bit) Mikrocontroller Familie[27], sowie die Atmel ATmega128 (8 Bit) [24] und den eher weniger bekannten Intel px28ax Controller. Darüber hinaus werden die gängigen Funkchips wie die Chipcon CC1000 (ISM Band, 433/868 MHz) und CC2420 (2,4 GHz) und Atmel AT86RF212 (868 MHz) und AT86RF230 (2,4 GHz) unterstützt. Wird der Funkchip CC2420 von Texas Instruments eingesetzt, kann TinyOS hier auch die in die Hardware intergrierten AES-Verschlüsselungsfunktionen nutzen, um den Funkverkehr abzusichern [23].

Der von TinyOS bereitgestellte Arbiter schaltet Funkübertragungs-Hardware immer nur für kurze Zeitspannen an, um z.B. neue Pakete zu empfangen, und lässt die Funkgruppen nie dauerhaft aktiviert. Daraus resultiert eine drastische Stromeinsparung, ohne für die Aussenwelt erkennbar zu machen, dass das Gerät nicht dauerhaft auf Empfang ist [29]. Allerdings wird dieser Vorteil mit einer höheren Latenz erkauft.

Zur Zeit-Synchronisation der Motes im Netzwerk verwendet TinyOS das *Flooding Time Synchronization Protocol* [12], welches Synchronisation bis zu Abweichungen im Millisekundenbereich ermöglicht [26].

Das für den Versand von Daten verwendete Protokoll „Collection Tree Protocol“ (CTP) ist sehr unempfindlich gegenüber Störungen und Ausfällen der Infrastruktur, so dass es als sehr robustes Routingprotokoll gilt [7].

Das Aufspielen von neuer Firmware auf Motes zur Laufzeit ist mit Hilfe des *Deluge* Protokolls möglich. Hierbei wird ein Komplettimage zu dem Mote übertragen, der mit einer neuen Software versehen werden soll. Der selektive Austausch nur einiger Komponenten eines Knotens, der unter TinyOS läuft, ist nicht möglich, da es keine Trennung zwischen Betriebssystem und Anwenderprogramm gibt.

Für die Datenverteilung zu den Knoten unterstützt TinyOS die Protokolle Drip, DIP und DHV. Bei Neuentwicklungen sollte dem modernen Protokoll DHV der Vorzug gegeben werden, da dieses die Verteilung am effizientesten gestaltet [25].

Programme in TinyOS bestehen aus *Components*, welche vom Typ *Module* oder *Configuration* sein können.

Eine *Component* in TinyOS besteht wiederum aus zwei Teilen: einem *interface* Teil und einem *implementation* Teil. Komponenten stellen selbst Interfaces bereit und nutzen externe Interfaces.

In *Modules* wird die Implementierung eines Interfaces vorgenommen, während in *Configurations* die Verbindung zwischen Komponenten und deren Interfaces definiert wird. Jede nesC Anwendung besteht aus wenigstens einer *Configuration* Komponente, die die einzelnen Teile der Anwendung beschreibt.

Die einzelnen *Components* werden dann vom TinyOS Scheduler ereignisgesteuert stanardmäßig nach dem FIFO-Prinzip aufgerufen. Die zentrale Motivation dieses Ansatzes ist, das System so schnell wie möglich wieder in den Schlafmodus zu schicken, um dadurch Energie zu sparen. Der Schedulingmechanismus lässt sich jedoch auch austauschen [14], um z.B. einen prioritäts-basierten Scheduler zu verwenden. Idealerweise bietet die unterstützte Mikrocontroller-Plattform hardware-seitig auch direkt unterschiedliche Prioritäten für Interrupts an. Diese Anforderung ist sowohl bei den MSP430 als auch den Atmel AVR Mikrocontrollern erfüllt, allerdings

doi: 10.2313/NET-2011-07-1_04

werden nur Hardware-Interrupts wie Timer, ADC, UART, ermöglicht. Diese Prioritäten sind auch fest vom Mikrocontroller vorgegeben und können nicht vom Entwickler geändert werden. Die Verwendung von vom Entwickler definierten „Software-Interrupts“ ist nicht möglich. Ein „Vectored Interrupt Controller“, bei dem den einzelnen Interrupts definierte Prioritäten zugewiesen werden können, existiert beispielsweise beim ARM Cortex M3 Mikrocontroller. Mit dem „Egs“ Mote existiert eine Plattform auf ARM Cortex M3 Basis, für die TinyOS bereits portiert wurde ([10]).

Das TinyOS Entwicklungssystem besteht aus Kommandozeilen-Tools, mit denen die Programme compiliert werden und auf die verschiedenen Motes übertragen werden. TinyOS bringt hierfür ein eigenes Buildsystem mit.

Im Folgenden werden diese Konzepte anhand der Beispielanwendung „BlinkC“ vorgestellt. Diese Anwendung lässt eine LED eines Motes mit einer festen Frequenz blinken und verdeutlicht die Nutzung der Timer, die das Betriebssystem abstrahiert zur Verfügung stellt. Die Anwendung besteht aus einer *Configuration* Komponente und einer *Module* Komponente.

Zunächst zur *Configuration* Komponente

```
1  configuration BlinkAppC {
   }
3
   implementation {
5    components MainC, BlinkC, LedsC;
     components new TimerMilliC() as Timer0;
7
     BlinkC -> MainC.Boot;
9    BlinkC.Timer0 -> Timer0;
     BlinkC.Leds -> LedsC;
11 }
```

Zeile 1-2 definiert diese Komponente als *configuration*. Die ersten zwei Zeilen im Abschnitt „implementation“ definieren die zu nutzenden Interfaces anderer Komponenten, namentlich `MainC`, `BlinkC` und `LedsC`. MainC und LedsC sind Dateien des TinyOS Betriebssystems, BlinkC ist unsere eigene Komponente. Desweiteren wird ein Timer mit den Namen Timer0 im System bekannt gemacht. Die letzten drei Zeilen der Datei verbinden die Elemente unserer Komponente BlinkC mit den anderen angeführten Komponenten.

Die Datei BlinkC, implementiert als *Module* die eigentliche Anwendung und ist wie folgt aufgebaut:

```
1  #include "Timer.h"

3  module BlinkC @safe() {
     uses interface Timer<TMilli> as Timer0;
5    uses interface Leds;
     uses interface Boot;
7  }

9  implementation {
     event void Boot.booted() {
11     call Timer0.startPeriodic( 250 );
     }
13   event void Timer0.fired() {
       call Leds.led0Toggle();
15   }
   }
```

Das Schlüsselwort „module“ zu Beginn der Datei deklariert diese Komponente als *Module*. Durch den Zusatz „@safe“ wird der Compiler angewiesen, zusätzliche Checks in den Programmcode einzubauen. Dadurch ist es möglich, z.B. Zugriffe ausserhalb der definierten Grenzen eines Arrays zu erkennen und das Programm kontrolliert abstürzen zu lassen, anstatt dass es zum ungewollten Überschreiben von Speicherbereichen kommt. Die „uses“ Schlüsselwörter in den folgenden 3 Zeilen besagen, dass diese Komponente die genannten 3 Komponenten nutzt. Die Semantik ist vergleichbar zu dem „implements“ Schlüsselwort in Java. Dadurch, dass unsere Komponente diese Interfaces nutzen will, muss sie zeitgleich auch alle geforderten Events implementieren. In unserem Beispiel ist dies das „fired()“ Event der genutzten Timer-Komponente. Der Rest des Programms ist selbsterklärend und illustriert die einfache Art wie in TinyOS ein Timer konfiguriert und eine LED umgeschaltet werden kann, ohne konkrete Kenntnis über die zugrundeliegende Hardware zu haben. Der Programmierer spricht lediglich „Led0“ an, muss aber nicht wissen, an welchem Port des Mikrocontrollers diese hängt. Die tatsächliche Zuordnung der Alias-Namen zur Hardware geschieht in den Header-Dateien von TinyOS für die jeweilige Zielplattform. Das „call“ Schlüsselwort ist nur notwendig, wenn Funktionen von Interfaces aufgerufen werden sollen. Der Aufruf von Funktionen innerhalb der Komponente ist, wie bei C üblich, ohne weiteren Aufwand direkt durch Angabe des Funktionsnamens möglich.

Ein Blick in die TinyOS System-Dateien bestätigt den erwarteten Aufbau:

Beispiel: Timer.nc

```
   tos/lib/timer/Timer.nc:
2  interface Timer {
     // basic interface
4    command void startPeriodic( uint32_t dt );
     command void startOneShot( uint32_t dt );
6    command void stop();
     event void fired();
8    ...
   }
```

Die drei „command“ Definitionen sorgen dafür, dass die Methoden `startPeriodic`, `startOneShot` und `stop` auf dem Interface ausgeführt werden können. Um das Interface auch nutzen zu können, muss ein Handler für das Event „fired“ implementiert werden, was in der Datei BlinkC.nc geschehen ist.

Das letzte Beispiel zeigte bereits ein Programm, welches asynchrone Aufrufe nutzte. Natürlich können in TinyOS auch synchrone Aufrufe verwendet werden, aber davon wird abgeraten, da synchroner Code immer ohne Unterbrechung ausgeführt wird und dies mitunter dazu führt, dass das System nicht mehr zeitnah reagieren kann.

Stattdessen macht man sich in TinyOS sogenannte „Split-

doi: 10.2313/NET-2011-07-1_04

Phase“ Operationen zu nutze, wo das Absetzen eines Befehls von dessen Erledigungsmeldung entkoppelt ist.

Um diese asynchronen Arbeiten auszuführen, hat TinyOS das Konzept der „Tasks“ vorgesehen. Tasks werden nicht sofort ausgeführt, sondern erst, wenn das System dafür Zeit hat.

Ein Task wird in TinyOS durch das Schlüsselwort „task“ definiert. Tasks haben einen „void“ Rückgabewert und nehmen keine Parameter entgegen. Dies erinnert an die Art und Weise, wie Threads in Java definiert werden. Auch hier gibt es eine `run`-Methode die implementiert werden muss, die aber keinen Rückgabewert gibt und keine Argumente nimmt.

Unser Beispiel kann wie folgt umgebaut werden:

```
task void computeTask() {
  uint32_t i;
  for (i = 0; i < 400001; i++) {}
}

event void Timer0.fired() {
  call Leds.led0Toggle();
  post computeTask();
}
```

Um einen Task zu starten wird der Aufruf „`post taskname()`“ verwendet.

Für das Task-Management verwendet TinyOS intern eine FIFO-Queue. Jeder Task wird durch ein Status-Byte auf dem Stack beschrieben. Die Länge ist auf 255 Tasks begrenzt. Wird ein Task aus der Queue gestartet, so läuft dieser ohne Unterbrechung bis zum Ende. Aus diesem Grund sollten Tasks nicht zu aufwändig sein, sondern sich wiederum ggf. in Subtasks splitten, falls größere Berechnungen durchgeführt werden müssen. Die Verwendung einer FIFO-Queue bedingt, dass alle Tasks gleichberechtigt sind und ihre Ausführungsreihenfolge ausschließlich von der Position innerhalb der Queue abhängt. Es ist nicht möglich, Tasks mit Prioritäten zu versehen und dadurch eine bevorzugte Abarbeitung zu erzwingen. Der Scheduler ist allerdings selbst eine TinyOS Komponente und kann als solche ersetzt bzw. angepasst werden um z.B. eine andere Scheduling Policy zu ermöglichen. Das Einhalten gewisser Vorgaben, wie beispielsweise „Fairness“ zur Verhinderung des „Verhungerns“ eines Tasks, sind vom Scheduler selbst sicherzustellen.

3.1.1 Persistente Speicherung von Daten

TinyOS abstrahiert auch das Einbinden von persistentem Speicher. In der Regel besitzen die verschiedenen Motes zusätzlichen On-board Speicher in Form von Flash-Speicher oder auch ein EEPROM. TinyOS nutzt plattform-spezifische Implementierungen der Komponenten `ConfigStorageC`, `LogStorageC` und `BlockStorageC`, die den Zugriff auf diese Komponenten abstrahieren [28, 3].

Der zur Verfügung stehende Flashspeicher kann von TinyOS in einzelne (logische) Volumes separiert werden, ähnlich den Partitionen einer Festplatte.

Das Konzept von Dateien ist TinyOS nicht bekannt. Es gibt kein Dateisystem im üblichen Sinne. Logdaten werden je nach Implementierung entweder sequentiell als Abfolge von sog. Records geschrieben oder über einen Ringbuffer persistiert. Beide Varianten werden von TinyOS in Form der Komponente „LogRead/LogWrite“ abstrahiert.

3.1.2 Ressourcenarbitrierung

Ressourcen in TinyOS können einer von drei Kategorien angehören: *dedicated*, *virtualized* oder *shared*. Die Kategorie bestimmt die Art und Weise, wie auf diese Ressource (etwa das Funksystem) zugegriffen werden kann. Eine `dedicated` Ressource steht der nutzenden Komponente exklusiv zur Verfügung, während `virtualized` Ressourcen die gleichzeitige Nutzung durch mehrere Komponenten ermöglichen: Jeder Client nutzt die Ressource, als würde sie ihm exklusiv zur Verfügung stehen, tatsächlich wird der Zugriff aller Komponenten durch das TinyOS gemultiplexed. Virtualisierte Ressourcen stellen den Komponenten keine Möglichkeit zur Verfügung, den Powerstate der Ressource zu kontrollieren, während `dedicated` Komponenten dies anbieten. `Shared` Ressourcen wiederum setzen auf `dedicated` Ressourcen auf, wobei der Zugriff über einen Arbiter reglementiert wird. Fragen mehrere Komponenten eine Ressource an, so ist es Aufgabe des Arbiters, diese Ressource entsprechend den aufrufenden Komponenten zur Verfügung zu stellen. Sobald ein Client eine Ressource im Zugriff hat, wird vorausgesetzt, dass er diese selbst wieder nach Gebrauch zurückgibt („Cooperative behaviour“). Ein Arbiter kann selbst keine Ressourcen entziehen - nur zuteilen. Der Arbiter kümmert sich mit Hilfe eines PowerManagers auch um die Stromversorgung der geteilten Ressourcen.

Neben den Power-State-Änderungen des Powermanagers versucht TinyOS natürlich auch die Stromspar-Funktionen der zugrundeliegenden Hardware zu nutzen. Jedesmal, wenn die Task-Queue leer ist, wird der bestmögliche Strom-Spar-Zustand ermittelt und der Controller in diesen Zustand versetzt, bis er durch ein externes Ereignis (z.B. das Empfangen eines Pakets über die Funkgruppe) wieder geweckt wird.

3.1.3 Datensammlung und -Verteilung

TinyOS beinhaltet im Lieferumfang bereits *Collector* und *Disseminator* Komponenten, die den Anwendungsentwickler dabei unterstützen, Daten im WSN zu verteilen (disseminate), bzw. einzusammeln (collect). Der Aufbau eines Routing-Trees geschieht dabei vollautomatisch im Hintergrund.

Die Auswahl des Dissemination-Verfahrens (TinyOS unterstützt die Verfahren Drip, DIP und DHV, die sich hinsichtlich ihrer Effizienz unterscheiden), geschieht erst auf Ebene des Makefiles. Rein vom Code sind zur Nutzung alle drei Verfahren (fast) identisch. Weitere Verfahren die TinyOS anbietet, sind etwa das Routingprotocol DYMO, mit dem on-demand Routen zwischen den Motes ermittelt werden können.

Um große Daten zu übertragen - das schließt auch Update-Images für die Motes selbst ein - kann das *Deluge T2* Protokoll verwendet werden. Damit ist es möglich, eine neue Firmware Version an einer zentralen Stelle in das Netzwerk einzuspielen und diese mit Hilfe des Protokolls auf alle Motes verteilen zu lassen [17]. Mit Deluge T2 ist es allerdings nur möglich, den gesamten Mote zu aktualisieren.

doi: 10.2313/NET-2011-07-1_04

Mit *blib* gibt es auch eine 6lowpan/IPv6 Implementierung in TinyOS. Der TCP Stack ist jedoch noch immer im experimentellen Stadium und es werden noch nicht alle Mote Typen unterstützt. Mit „nwprog“ gibt es auch hier eine Möglichkeit, die Motes über ein IP Netzwerk neu zu programmieren (via Deluge).

3.1.4 Energiesparen

TinyOS optimiert das Abschalten zur Zeit nicht genutzter Hardware-Komponenten selbständig. Für das Funkmodul steht die Funktionsart *LPL* (Low Power Listening) zur Verfügung, welches die Funkgruppe nur für die absolut notwendige Zeit aktiviert, um zu prüfen, ob ein Paket übertragen wird, welches für den aktuellen Mote bestimmt ist. Welcher Powersave-Mode des Mikrocontrollers aktuell am Besten geeignet ist, ermittelt TinyOS anhand der Status- und Controlregister, welche chip-spezifisch ausgewertet werden, einem „dirty bit“, welches die Notwendigkeit einer neuen Power-Save-Berechnung signalisiert und einem „Power-State override“. Die Berechnungen finden innerhalb der TinyOS core scheduling loop statt. Finden Ereignisse statt, die zu einer Veränderung des Powersave-Modes führen können, wird dies durch das Dirtybit signalisiert und der Zustand daraufhin reevaluiert.

3.1.5 TOSThreads

Ab TinyOS 2.1 können nun auch präemptive Threads mithilfe der TOSThreads Bibliothek erstellt werden [22]. Der Kernel selbst läuft als hoch-priorer Thread. Eine Besonderheit ist, dass die TOSThreads Library es gestattet, die Anwendung in reinem C zu schreiben und kein nesC verwenden zu müssen. Dies ist möglich, da TOSThreads einen Wrapper über die Split-Phase-Operationen legt. Die Anwendung wird dennoch effizient in TinyOS ausgeführt. Threads bieten `start()`, `stop()`, `pause()`, `resume()`, `sleep()`, `run()` und `join()` Funktionen, wie sie aus z.B. POSIX kompatiblen OS bekannt sind. Desweiteren werden auch die folgenden Synchronisations-Primitive unterstützt: Mutex, Semaphore, Barrier, Condition variable, Blocking reference counter.

3.2 Contiki

Contiki sieht sich in direkter Konkurrenz zu TinyOS und will vieles besser machen. Als ein Vorteil wird von den Entwicklern angeführt, dass es bei der Nutzung von Contiki nicht notwendig ist, einen neuen C-Dialekt (nesC) wie bei TinyOS zu lernen. An der Contiki Entwicklung sind u.a Firmen wie SAP, Cisco, Atmel, sowie die TU München beteiligt.

Wie bei TinyOS gibt es auch hier ein VM Image, das alles enthält, was man für die Entwicklung benötigt. Desweiteren gibt es einen Simulator namens Cooja, der genutzt werden kann, wenn keine echten Motes verfügbar sind, und der z.B. auch graphisch die Kommunikationsbeziehungen zwischen Teilnehmern eines WSNs darstellen kann. Die Hardware, die von Contiki unterstützt wird, ist die gleiche, die auch von TinyOS unterstützt wird. Wie bereits im Abschnitt über TinyOS erwähnt, kann Contiki aber nur auf Geräten eingesetzt werden, für die ein entsprechender Port existiert. Contiki wurde bereits für eine ganze Reihe von Systemen portiert und ist keinesfalls nur auf Mikrocontroller beschränkt. So gibt es Ports für den C64, Apple II, Atari ST, Gameboy, Playstation und viele andere Klassiker der Computerära der 80er und 90er Jahre.

3.2.1 Programmiermodell

Als Ablaufmodell kommt bei Contiki ebenso wie bei TinyOS ein eventbasiertes Modell zum Einsatz, da dieses den Memory-Overhead für echte Multi-Threading Systeme vermeidet. Bei Event-basierten Systemen ist es nicht notwendig, Stackspeicher für jeden Thread zu reservieren, was den eingeschränkten Ressourcen von Mikrocontrollern zu Gute kommt.

Ein großer Vorteil gegenüber TinyOS ist bei Contiki die Möglichkeit, Applikationen zur Laufzeit zu laden bzw. zu entladen. Contiki bietet auch Features, die nur von größeren Betriebssystemen bekannt sind, wie Interprozesskommunikation (IPC) via „Message Passing“ und Multithreading innerhalb einer Anwendung. Realisiert wird dies in Contiki über sogenannte „Protothreads“. Hierbei handelt es sich um eine sehr leichtgewichtige Thread-Implementierung, die für die Datenspeicherung über den Context-Wechsel hinweg keinen Stack, sondern globale Variablen nutzt. Pro Thread fallen dafür nur 2 Byte Speicher für die Verwaltung an (vgl. [5]).

Hauptmotivation für die Entwicklung der Protothreads war die Tatsache, dass eventbasierte Programmierung voraussetzt, dass die Anwendungen wie Zustandsmaschinen aufgebaut sind. Es kann aber, je nach Anwendung, alles andere als trivial sein, die gewünschte Logik korrekt als Zustandsmaschine abzubilden und erfordert oft ein Umdenken der Entwickler. Da solche Zustandsmaschinen oftmals ohne eingehende Vorabplanung umgesetzt werden, kommt es hierbei zu höheren Aufwänden beim Testen und Debuggen. Protothreads vereinfachen die Umsetzung, da sie nach außen hin keine Zustandsmaschinen zu nutzen scheinen. Tatsächlich wird intern der Kontrollfluss jedoch auf Schritte einer Zustandsmaschine abgebildet. Protothreads-Programme sind zumeist auch kompakter als entsprechende Statemachine-Implementierungen. Detaillierte Informationen zur Implementierung von Protothreads findet sich in [5].

Ein Protothread wird über das Makro „`PT_THREAD`“ definiert und enthält den Code zwischen den beiden Markern „`PT_BEGIN`“ und „`PT_END`“. Zentrales Element ist der Befehl „`PT_WAIT_UNTIL`“, welches eine Bedingung übergeben bekommt und den Thread so lange blockiert, bis die Auswertung der Bedingung zu „true“ erfolgt. Protothreads können *nur* an diesen Stellen blockieren und die Kontrolle abgeben, andere Möglichkeiten gibt es nicht. Dies erleichtert aber auch gleichzeitig dem Programmierer die Fehlersuche, da die Punkte im Code, an denen blockiert wird, sofort erkennbar sind.

Um dieses Konzept zu verdeutlichen, wird im folgenden Codeausschnitt das im Kapitel „TinyOS“ vorgestellte Beispielprogramm mit Protothreads für Contiki umgeschrieben.

```
1 #include "pt.h"

3 struct pt pt;
  struct timer timer;
5
  PT_THREAD(example(struct pt *pt))
7 {
    PT_BEGIN(pt);
9
```


doi: 10.2313/NET-2011-07-1_04

```
    while(1) {
11      timer_start(&timer);
        PT_WAIT_UNTIL(pt,
13          timer_expired(&timer));
        call_toggle_led();
15
    }
17  PT_END(pt);
}
```

3.2.2 Persistente Speicherung von Daten

Für das persistente Schreiben von Daten bietet Contiki das „Coffee“ Filesystem, welches Funktionen zum Lesen, Schreiben, Positionieren (Seek), Anhängen und Löschen von Dateien bietet. Ebenso werden auch Verzeichnisse unterstützt. Im Gegensatz zu TinyOS hat der Entwickler hier die Möglichkeit, „echte“ Dateien und Verzeichnisse zu erzeugen. Contiki abstrahiert die Zugriffe hierbei von der eigentlichen Ressource, d.h. prinzipiell kann als Schreibziel jedes Medium verwendet werden, für das ein entsprechender Treiber in Contiki existiert (Netzwerkshare, SD-Karte, ...)

3.2.3 Netzwerkstack

Contiki kommt mit zwei fertigen Kommunikations-Stacks: „uIP“ und „Rime“. Bei uIP handelt es sich um eine vollwertige Implementierung eines TCP/IP Stacks, welcher speziell für die begrenzten Betriebsmittel eines Mikrocontrollers entwickelt wurde. uIP bietet eine API an, die an die Posix Sockets angelehnt ist und u.a. Funktionen zum DNS Lookup bietet, sowie die Protokolle IP, IPv6, UDP, TCP, ARP und 6lowpan unterstützt. „Rime“ ist der Versuch, ein einheitliches Protokoll zu implementieren, welches von der zugrundeliegenden Hardware abstrahiert und dem Anwendungsentwickler die Mühe abnimmt, für jede Funktechnologie, jedes Funk-Hardwaremodul oder jedes Übertragungsprotokoll seine Anwendung erneut anpassen zu müssen.

Als Media Access Control (MAC) kommt das CSMA/CD ([15]) Verfahren zum Einsatz. Um den geringen Ressourcen von Motes Rechnung zu tragen, können verschiedene „Radio Duty Cycling Layer“ (RDCs) ausgewählt werden, die zur Compilezeit gebunden werden. Implementierungen, die die Funkgruppe des Mote nur alle paar hundert Millisekunden aktivieren, um zu prüfen, ob neue Daten vorliegen, verbrauchen gegenüber einem Mote, dessen Funkgruppe permanent aktiviert ist, natürlich weniger Strom. Erkauft wird dies dann mit einer größeren Latenz, bis der Mote auf eine Nachricht reagiert (vgl. auch Abschnitt TinyOS).

3.2.4 Speicher-Management

Contiki nutzt einen Speichermanager, der genutzt werden kann, um fragmentierungsfrei dynamisch Speicher zu allokieren. Dieser verschiebt freigegebene bzw. genutzte Speicherbereiche bei jeder Änderung, so dass sich möglichst große zusammenhängende Flächen ergeben und einer Fragmentierung entgegen gewirkt wird.

Dies wiederum setzt allerdings voraus, dass Anwendungen nie direkt auf den Speicher zugreifen, sondern über einen Umweg über den Contiki Speichermanager, damit ein Zugriff auf einen vormals allokierten Speicherbereich nicht in einem inzwischen anderweitig genutzten Bereich endet.

3.2.5 Laden und Entladen von Anwendungen

Contiki nutzt ein DLL (Dynamic linking and loading) Laufzeitsystem, um Applikationen dynamisch zu laden und zu entladen. Da WSN-Knoten häufig im Produktiveinsatz an unzugänglichen Stellen platziert werden (vgl. die verschiedenen Szenarien in Abschnitt 1), wird die Fähigkeit benötigt, Teile des Systems zur Laufzeit durch „over-the-air“ Programmierung zu aktualisieren, da ein Abbau des Sensors und der Anschluss an eine Programmierstation in der Regel zu aufwändig ist. Software-Updates können hier einerseits Fehlerkorrekturen an den vorhandenen Anwendungen sein, es können aber auch zusätzliche Anwendungen eingespielt werden. Contiki setzt für die Binaries das ELF Objektformat ein, welches z.B. im Linux Umfeld weit verbreitet ist.

Im Gegensatz zu TinyOS geht Contiki bei den „over-the-air“ Updates einen anderen Weg und erlaubt das selektive Updaten bzw. Hinzufügen von Applikationen. Dies ist sinnvoll, denn das Ersetzen des kompletten Firmware-Images ist nur selten notwendig und wirkt sich negativ auf den Stromverbrauch und die Dauer des Updates aus.

Ein Full-System-Replacement ist relativ einfach durchzuführen, da keine Vorverarbeitung des Images auf dem Zielknoten stattfinden muss. Alle im Code enthaltenen Adressen sind absolut und das Image wird einfach in den Flashspeicher eingespielt und überschreibt den Speicher komplett. Der Ansatz in Contiki, ein dynamisches Linken von Modulen einzuführen, war relativ neu, da der mit dem Linken verbundene Berechnungsaufwand bis zu diesem Zeitpunkt als zu aufwändig und damit für WSN Motes als unwirtschaftlich angesehen wurde. Beim dynamischen Linken werden alle in dem Modul deklarierten Funktionsaufrufe und Symbole (z.B. aus dem Kernel) aufgelöst. Erst, wenn alle Referenzen erfolgreich verknüpft worden sind, ist das Modul nutzbar. Funktionsaufrufe müssen dabei nicht nur Aufrufe aus anderen Modulen oder dem Kernel sein, sondern sind auch die Aufrufe im eigenen Modul: Da das Modul dynamisch in den Speicher geladen wird, sind alle im Code verwendeten Adressen Offsets oder Alias-Adressen, die erst beim Laden auf die physikalischen Speicheradressen gesetzt werden. Diesen Vorgang nennt man „Relocation“.

Dies ist das gleiche Verhalten, wie es DLLs in der Windows-Welt oder SO (Shared Objects) in Linux Betriebssystemen aufweisen.

Objektdateien im ELF Format beinhalten unter anderem den eigentlichen Programm-Code, Symboltabellen und Relokierungstabellen, also alle Informationen, die für ein dynamisches Laden des Programmstückes notwendig sind. Ein großer Nachteil des ELF-Formats ist jedoch, dass es den Anspruch hat, sowohl unter 32 Bit als auch 64 Bit Betriebssystemen ohne Änderungen genutzt werden zu können. Daher werden auch grundsätzlich nur 64 Bit kompatible Datentypen verwendet, was bedeutet, dass die ELF-Dateien mehr Speicherplatz in Anspruch nehmen, als auf einem Mote nötig wäre, was wiederum zu längeren Übertragungszeiten und größerem Speicherplatzbedarf führt. Daher hat man das CELF Format („Compressed ELF“) eingeführt, welches diese Kompatibilität wieder entfernt, und native 8 und 16 Bit Datentypen unterstützt. Die CELF Datei ist im Schnitt nur noch halb so groß wie die normale ELF Datei und wird

doi: 10.2313/NET-2011-07-1_04

automatisch während des Compiler-Vorgangs erzeugt.

Ein Contiki System besteht aus dem Core Bereich und dem Bereich für die ladbaren Module. Der Core Bereich ist im Wesentlichen das, was man den Kernel nennt. Er bietet Datenstrukturen und die C-Bibliotheksfunktionen an. Ein ladbares Modul kann die Funktionen des Kernels aufrufen und darüber auch mit anderen Modulen Daten austauschen. Diese Teilung macht es möglich, nur die ladbare Applikation auszutauschen und den Kernel unangetastet zu lassen. Darüber hinaus ist es möglich auch den Core zur Laufzeit auszutauschen, jedoch wird von diesem Feature eher selten Gebrauch gemacht.

3.3 cocoOS

Das dritte Betriebssystem, welches im Rahmen dieser Ausarbeitung betrachtet wird, ist cocoOS [19]. cocoOS erfährt aktuell die meisten Änderungen, ist aber auch ein recht neues Projekt, welches erst 2010 gestartet wurde.

Im Vergleich zu TinyOS und Contiki fällt sofort auf, dass cocoOS noch in den Kinderschuhen steckt. Bislang sind lediglich OS Primitive wie Tasks, Events, Messages und Semaphore realisiert. Aktualisierungen der Firmware über Funk, Strom-Spar-Modi, IP stacks oder ähnliches sind für dieses Betriebssystem noch Fremdwörter.

cocoOS ist derzeit nur für die AVR Plattform verfügbar, ein Port für MSP430 ist jedoch ebenfalls vorgesehen.

Wie auch bei TinyOS und Contiki OS wird bei cocoOS das Prinzip eines kooperatives Multitasking eingesetzt. Die „Task procedures“ werden stets komplett ausgeführt. Es ist keine Präemption vorgesehen.

Die bisher vorhandenen Primitive sind sehr einfach zu nutzen.

3.3.1 Tasks

Bis zu 16 Tasks können im System vorhanden sein. Mittels der API Funktion `task_create` kann ein neuer Task dem System hinzugefügt werden. Beim Erzeugen eines Tasks wird ein Pointer auf die Taskprozedur übergeben, es kann eine Priorität vergeben werden und eine MessageQueue für den entsprechenden Task.

Die Taskroutine selbst wird durch zwei Makros eingekleidet, ähnlich wie bei ContikiOS:

```
void task(void) {
    task_open();
    ...
    task_close();
}
```

Auf äußere Ereignisse kann mittels des Aufrufes `event_wait` gewartet werden. Das Task blockiert bis zum Eintreffen des Events und ein anderer Task wird vom cocoOS Scheduler als aktiver Task ausgewählt.

Ein Beispiel für einen Task, welcher endlos ausgeführt wird und auf ein Ereignis von der seriellen Schnittstelle wartet, ist im folgenden Listing gegeben:

```
void task(void) {
    uint8_t data;
    task_open();
    for (;;) {
        event_wait(rxEvt);
        uart_get(&data);
        ...
    }
    task_close();
}
```

3.3.2 Events

Im obigen Beispiel kam bereits ein Aufruf von `event_wait` vor. Mittels dieses API Calls können Tasks auf das Eintreffen eines Ereignisses warten. Zunächst muss jedoch ein Event mittels der Funktion `event_create` erzeugt worden sein. Im Code selbst kann dann mittels `event_signal` oder, falls man sich in einer Interrupt Service Routine (ISR) befindet, mit `event_ISR_signal` das entsprechende Event signalisiert werden. Das Warten auf mehrere Events (Synchronisationsprimitiv „Barriere“) ist ebenfalls möglich und zwar über die Funktion `event_wait_multiple`.

Unser bereits bekanntes „Blinkprogramm“ könnte in cocoOS wie folgt implementiert werden:

```
Evt_t timerExpiredEvent;

void main(void) {
    system_init();
    os_init();
    timerExpiredEvent = event_create();
    task_create(task,1,NULL,0,0);
    setup_timer();
    clock_start();
    os_start();
    return 0;
}
void task(void) {
    uint8_t data;
    task_open();
    for (;;) {
        event_wait(timerExpiredEvent);
        toggle_led();
        ...
    }
    task_close();
}
```

3.3.3 Messages

Sollen mehr als einfache Signale verschickt werden, können „Messages“ verwendet werden. Messages sind C structs beliebigen Inhalts, welche zwischen Tasks versendet werden können und in einer Message-Queue gepuffert werden (siehe API Call `task_create` weiter oben). Auch das periodische Versenden von Nachrichten ist möglich.

Der Zugriff auf die Message-Queue erfolgt über `msg_q_get` und `msg_q_give`, welche intern als Mutex die Message-Queue für den exklusiven Zugriff locken.

doi: 10.2313/NET-2011-07-1_04

Über `msg_post` bzw. `msg_receive` erfolgt dann der Versand bzw. Empfang der Nachrichten.

Ein Beispiel:

```
static void task1(void) {
    static Msg_t msg;
    msg.signal = 10MS_SIG;
    task_open();
    for (;;) {
        task_wait( 10 );
        msg_q_get( task2 );
        msg_post( task2, msg );
        msg_q_give( task2 );
    }
    task_close();
}
```

3.3.4 Semaphore

Über `sem_bin_create` bzw. `sem_counting_create` kann ein binäres Semaphor (mutex) oder ein zählendes Semaphor erzeugt werden. Mittels `sem_wait` und `sem_signal` werden die Operationen „P“ („Probeer“) bzw. „V“ („Vrijgeven“) umgesetzt (Nähere Informationen zu Semaphoren unter [4]).

Damit sind die Möglichkeiten von cocoOS bereits erschöpft. cocoOS bietet kein Modulkonzept, bringt keine Treiber für etwaige Funkmodule mit und bietet auch keine Netzwerk-Protokolle. Damit entfällt auch die Möglichkeit Teile des Systems zur Laufzeit auszutauschen. Der Funktionsumfang von cocoOS ist somit wesentlich geringer als der der anderen beiden vorgestellten Betriebssysteme TinyOS und Contiki.

4. ZUSAMMENFASSUNG

In der vorliegenden Arbeit wurden drei Betriebssysteme für Wireless Sensor Netzwerkknoten vorgestellt: Tiny OS, Contiki und cocoOS. TinyOS, das älteste der drei vorgestellten Betriebssysteme, zeichnet sich besonders durch den stabilen Einsatz aus. Contiki bietet durch den Einsatz von Protothreads neue Möglichkeiten, die so bei TinyOS bislang nicht möglich waren und erst durch TOSThreads nachträglich eingebaut wurden. Es wurde das dynamische Laden von Anwendungen unter Contiki vorgestellt, ein Aspekt, der gerade im WSN Umfeld sehr positiv anzusehen ist. Sowohl TinyOS als auch Contiki OS verbergen erfolgreich die Komplexität der Hardware vor dem Programmierer und erlauben die Entwicklung von Anwendungen, die auf Motes unterschiedlicher Hersteller laufen. cocoOS hingegen ist ein sehr neues Projekt, welches aktuell erst einige wenige Datenstrukturen und Basis-Funktionen eines Betriebssytems bietet, und sich um Dinge wie eine HAL (Hardware Abstraction Layer) bislang nicht gekümmert hat. Durch das Studium des noch überschaubaren cocoOS Sourcecodes ist es jedoch noch gut möglich, ein Verständnis für diese grundlegenden Betriebssystem-Objekte zu entwickeln, ein Aspekt der bei TinyOS und Contiki durch die inzwischen erreichte Komplexität der Projekte mit deutlich mehr Aufwand verbunden ist.

Die meiste Dokumentation zum Betriebssystem ist aktuell bei TinyOS vorhanden. Hier werden dem Entwickler in mehreren Tutorials die grundlegenden Konzepte des Betriebssystems nahegebracht. Die Dokumentation in Contiki ist als oberflächlich zu bezeichnen. Entwickler, die genauere Informationen benötigen, sind darauf angewiesen, das Verhalten in den Source-Dateien von Contiki nachzuvollziehen. cocoOS besitzt von allen drei Betriebssystemen die kürzeste Dokumentation, was aber in Anbetracht der einfachen Datenstrukturen, und dem geringen Funktionsumfang nachvollziehbar ist.

Betriebssysteme auf Mikrocontrollern befinden sich auf einer wesentlich niedrigeren Evolutionsebene als andere bekannte Betriebssysteme aus dem Desktop- und Server-Umfeld. Dies ist in erster Linie den stark beschränkten Hardware-Ressourcen geschuldet, die für die Entwicklung von Programmen auf den Controllern zur Verfügung stehen. Zudem handelt es sich um ein relativ spezielles Einsatzgebiet, welches von großen Unternehmen bislang nicht als Markt betrachtet wird. Durch die vielen Aktivitäten im Bereich der Wireless Sensor Networks ist jedoch davon auszugehen, dass alle drei vorgestellten Betriebssystemen stetig weiterentwickeln werden.

5. FAZIT

Alle drei vorgestellten Betriebssysteme ermöglichen die Entwicklung von Anwendungen und entlasten den Entwickler davon, wichtige Komponenten wie Ereignisbehandlung und Netzwerk-Konnektivität selbst implementieren zu müssen. Sowohl TinyOS als auch Contiki bieten eine breite Unterstützung für vielfältige Netzwerkprotokolle. Eine generelle Empfehlung für eines der Betriebssysteme ist schwer zu geben, da sie sich in bestimmten Punkten (z.B. Programmiermodell, Sprachsyntax, Netzwerk-Schichten, Filesystem) teils deutlich unterscheiden.

6. LITERATUR

[1] ZigBee Alliance. ZigBee Alliance. `http://www.zigbee.org`, 2011.

[2] Dargie, W. and Poellabauer, C. *Fundamentals of wireless sensor networks: theory and practice*. John Wiley and Sons, 2010.

[3] Jonathan Hui David Gay. TEP 103: Permanent Data Storage. `http://www.tinyos.net/tinyos-2.x/doc/html/tep103.html`.

[4] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[5] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[6] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[7] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, and Philip Levis. CTP: Robust and efficient collection through control and data plane integration. February 2008.

doi: 10.2313/NET-2011-07-1_04

[8] Mohamed Hefeeda and Majid Bagheri. Wireless sensor networks for early detection of forest fires. 2007.

[9] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGPLAN Not.*, 37:96–107, October 2002.

[10] Jeonggil Ko, Qiang Wang, Thomas Schmid, Wanja Hofer, Prabal Dutta, and Andreas Terzis. Egs: A Cortex M3-based Mote Platform.

[11] Philip Levis. TinyOS Programming Manual. `http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf`, 2006.

[12] Miklos Maroti, Branislav Kusy, Simon, and Akos Ledeczi. The flooding time synchronization protocol. pages 39–49. ACM Press, 2004.

[13] Martin Leopold. Creating a new platform for TinyOS 2.x. `http://www.tinyos.net/tinyos-2.1.0/doc/html/tep131.html`, 2007.

[14] Philip Levis, Cory Sharp. Schedulers and Tasks. `http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html`, 2011.

[15] Tom Sheldon. CSMA/CD explained. `http://www.linktionary.com/c/csma.html`, 2001.

[16] Tanenbaum, Andrew S. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[17] TinyOS Project. Deluge T2 - Tutorial. `http://docs.tinyos.net/index.php/Deluge_T2`, 2010.

[18] TUM, LS Knoll. Multifunk. `http://www6.in.tum.de/pub/Main/ResearchMultifunk/micaz.PNG`, 2011.

[19] Various. cocoOS. `http://www.cocoos.net`.

[20] Various. IPv6 over Low power WPAN (6lowpan). `http://datatracker.ietf.org/wg/6lowpan/charter/`.

[21] Various. Netduino. `http://netduino.com`.

[22] Various. TOSThreads Tutorial. `http://docs.tinyos.net/index.php/TOSThreads_Tutorial`, 2009.

[23] Various. CC2420 Security Tutorial. `http://docs.tinyos.net/tinywiki/index.php/CC2420_Security_Tutorial`, 2010.

[24] Various. ATmega128 Product page. `http://www.atmel.com/dyn/products/product_card.asp?part_id=2018`, 2011.

[25] Various. Dissemination. `http://docs.tinyos.net/tinywiki/index.php/Dissemination`, 2011.

[26] Various. FAQ - TinyOS Documentaiton Wiki. `http://docs.tinyos.net/tinywiki/index.php/FAQ`, 2011.

[27] Various. MSP430G2xx 16 bit Microcontroller. `http://focus.ti.com/paramsearch/docs/parametricsearch.tsp?familyId=1937&sectionId=95&tabId=2662&family=mcu`, 2011.

[28] Various. Storage Tutorial. `http://docs.tinyos.net/index.php/Storage`, 2011.

[29] Various. TinyOS - Resource Arbitration and Power Management. `http://docs.tinyos.net/index.php/Resource_Arbitration_and_Power_Management`, 2011.

[30] Various. TinyOS Home Page. `http://www.tinyos.net`, 2011.

doi: 10.2313/NET-2011-07-1_04