Betriebssysteme für IoT-Geräte

Denis Küchemann
Seminar Hot Topics in Communication Systems and Internet of Things
Communciation and Networked Systems (ComSys)
Institute of Computer Science

***Zusammenfassung*—Dieser Artikel befasst sich mit Betriebssystemen in der Welt des Internet der Dinge. Es wird die Beschaffenheit von IoT-Geräten analysiert und was ein Betriebssystem leisten muss, um diese Geräte effizient und sicher zu steuern. Abschließend werden die Betriebssysteme TinyOS, Contiki, RIOT und Nano-RK miteinander verglichen.**

I. Einleitung

Die Vision des Internet der Dinge (Internet of Things, IoT) ist die Vernetzung eindeutig identifizierbarer und adressierbarer Gegenstände, insbesondere die von Alltagsgegenständen. Die vernetzten (mit dem Internet verbundenen) Geräte (Systeme) sollen Informationen über ihre Umwelt sammeln und mit dieser kommunizieren. Es bedarf hierzu besonderer Betriebssysteme (OSs), die den Anforderungen des Systems gerecht werden können. Dazu gehören speziell bei drahtlosen Systemen ein effektives Energie-Management und die Einbindung von leistungsstarken Sicherheitsmechanismen. Ebenso ist die Zurverfügungstellung von IoT-, sowie Standard-Protokollen wie IPv4 und IPv6 für die Vernetzung erforderlich. Durch die physische Beschaffenheit sehr kleiner Systeme wird es außerdem wichtig, die Speicherallozierung für diese Systeme mit einem Speicher in Kilobyte-Größenordnung effizient zu gestalten. In diesem Zusammenhang werden die Begriffe embedded OS und RTOS (Real-Time-Operating-System) wichtig. Jedes OS bietet spezielle Vorzüge gegenüber anderen OSs. Welche weiteren Features ein OS robuster oder leistungsstärker machen, hängt nicht nur vom jeweiligen System, sondern auch von persönlichen Präferenzen ab. Der aktuelle Bedarf an IoT-OSs zeigt sich durch das massive Bestreben der Industrie, auch den erweiterten häuslichen, privaten Bereich mit vernetzter Elektronik auszustatten. Auch die Globalplayers Apple und Google entwickeln und verbessern Systeme oder kaufen deren Entwickler auf. So kommen z.B. aktuell Mobiltelefone mit Near-Field-Communication-Chips (NFC) auf den Markt, um z.B. an Kassensystemen vor Ort zu bezahlen oder es werden Wohnungen mit selbstlernenden Raumthermostaten und Rauchmeldern (Smart-Home) ausgestattet. Cisco konstatiert, dass bereits im Jahr 2015 15 Milliarden Systeme miteinander vernetzt sein könnten [1] und schätzt vorsichtig, dass 2020 50 Milliarden vernetzte Systeme existieren [2].

Im folgenden Kapitel II werden zuerst die Beschaffenheit eingebetteter Systeme analysiert und die dadurch bedingten Anforderungen an ein IoT-OS eingeführt. Anschließend werden in Kapitel III verschiedene Arten von OSs und deren Eigenschaften vorgestellt. In Kapitel IV werden einige IoT-OS vorgestellt und miteinander verglichen. Abschließend wird im Resümee zusammengefasst, welche Themen in dieser Arbeit behandelt wurden und ein Ausblick auf zukünftige Entwicklungsmöglichkeiten von IoT-OSs gegeben.

II. Hauptmerkmale von Systemen und OSs

A. Gerätebeschaffenheit

Eingebettete Systeme[1] basieren auf Mikrocontrollern (MCUs), in denen Speicher, Prozessoren und Peripherie verbaut sind. Die Prozessoren in den MCUs können dabei teilweise eine 4-Bit-Architektur haben und z.B. eine Taktung von nur 1MHz, der Speicher nur 2kB groß sein [3]. MCUs mit diesen Eigenschaften werden in physisch sehr kleinen Systemen verbaut. Populäre Beispiele für leistungsstärkere Systeme sind Mobiltelefone/Smartphones und Laptops oder Tablets. Ein System muss aber nicht drahtlos, d.h. über eine Antenne verbunden sein - das System kann ebenso über ein Kabel vernetzt sein. Ein System kann auch bei drahtloser Vernetzung an ein Stromnetz (via Kabel) angeschlossen sein. Üblicherweise haben Systeme/MCUs ohne eine derartige Stromanbindung einen sehr niedrigen Stromverbrauch, da sie derzeit noch Batterien verbaut haben, die nicht die Fähigkeit besitzen, sich selbst aufzuladen. Nach einer gewissen Zeit müssen diese Batterien ausgetauscht oder erneut aufgeladen werden. Derzeit werden häufig Alkali-Mangan-Batterien oder Lithium-Ionen-Akkumulatoren als Energiequelle benutzt. [4]. Der RFID-Chip[2], der oft in der Automobilindustrie Anwendung findet, revolutionierte die Idee des IoT: Der RFID-Chip stellt die Schnittstelle zwischen dem damit ausgestatteten Objekt und dem Internet dar. Die Vernetzung und eindeutige Identifizierung des Objekts ist also gegeben. Ein typischer RFID-Chip besitzt allerdings nicht immer eine CPU. Mindestens müssen aber ein integrierter Schaltkreis für eine Vielzahl von Funktionen und eine Antenne für die Vernetzung verbaut sein. [5]

B. Anforderungen an das Betriebssystem

1) Energie-Management: Auch wenn ein System einen sehr niedrigen Stromverbrauch hat, muss ein OS ein effizientes Energie-Management besitzen und auf Stromknappheit reagieren können. Dennoch bieten viele open-source-RTOS kein optimales Energie-Management [4]. In Kapitel III-B6 wird genauer analysiert, wie Energie mithilfe des OS eingespart werden kann.

[1] Kombination von Hard- und Software mit dedizierter Funktion

[2] Der Radio-Frequency-Identification-Chip ist ein Sender-Empfänger-System, das es Herstellern ermöglicht, ihre Produkte zu orten.

2) Sicherheitsmaßnahmen: Insbesondere bei drahtlosen Systemen ist die Sicherheit wie immer ein großes und wichtiges Thema. Sicherheitsrisiken lassen sich zuerst bei OSs mit frei zugänglichem Quellcode (free OSs) feststellen [4]. Einige free OSs kompilieren z.B. Anwendungen und Kernel zusammen. Dabei wird ein gemeinsamer Adressraum verwendet, was ein Sicherheitsrisiko darstellen kann [6]. In Kapitel III-B4 werden einige Methoden vorgestellt, die die Sicherheit eines OS erhöhen.

3) Speicherallozierung: Ein OS muss in Bezug auf den Speicher skalieren können. Durch die Systembeschaffenheit bedingt ist es nötig, Speicher von wenigen Kilobyte allozieren und auslesen zu können. Dabei können einige OSs bereits Speicher mit unter einem kB allozieren und ROM unter 4kB auslesen. Für die Effizienz der Speicherallozierung ist es wichtig, zu analysieren, ob ein OS statische oder dynamische Speicherallozierung unterstützt. Kapitel III-B5 untersucht die Unterschiede dynamischer und statischer Allozierung und deren Verwendung in OSs für eingebettete Systeme.

4) Protokolle: Für IoT-Systeme wurden neue Protokolle entwickelt. Insbesondere für drahtlose Systeme mussten neue Standards gesetzt werden, da ältere Protokolle nicht für low-power-Netzwerke optimiert sind.

- IEEE 802.15.4 ist ein Übertragungsprotokoll der Bitübertragungs- und Mac-Schicht, dass für WPANs (Wireless Personal Area Networks) und speziell für WSNs (Wireless Sensor Networks) entwickelt wurde [7]. IEEE 802.11 (WLAN) und IEEE 802.15.1 (Bluetooth) waren als Alternativen nicht in der Lage, eine lange Batterielebensdauer zu garantieren. Das Protokoll zielt außerdem darauf ab, zukünftige Drahtlostechnologien in den Punkten Sicherheit, Komfort, Zuverlässigkeit, Produktivität und Produktionskosten zu verbessern. Einige der Haupteigenschaften sind Datenraten von 250, 40 oder 20 kB/s, Unterstützung von Echtzeitsystemen, dynamische Systemadressierung, CSMA-CA[3]-Kanalzugriff und ein vollständiges Handshake-Protokoll [8].
- IPv4 und IPv6 sind Standardprotokolle der Vermittlungsschicht. IPv4 benutzt 32Bit-Adressen. Für die Anzahl an vernetzten Geräten ist die Anzahl der Adressen lange nicht mehr ausreichend. Mit IPv6 wurde die Anzahl Adressen auf 2^{128} erhöht. Für einen möglichst geringen Energieverbrauch wird bei drahtlos vernetzten Systemen 6LoWPAN (IPv6 over Low Power Wireless Personal Area Networks) (auch IPv6 over IEEE 802.15.4 genannt [9]) benutzt. Dazu werden IPv6 Pakete so komprimiert, dass der IEEE 802.15.4-Standard eingehalten werden kann: Die MTU[4] von IPv6 beträgt 1280 Bytes; IEEE 802.15.4 sieht lediglich eine MTU von 127 Bytes vor. Die IEEE 802.15.4 Arbeitsgruppe beschreibt nur eine Kompression für UDP-Header [10].
- RPL (Routing Protocol for Low power and lossy networks) ist ein Routing-Protokoll, dass für drahtlose Low power and Lossy Networks(LLNs) nach IEEE 802.15.4-Standard optimiert ist. Typische Eigenschaften eines LLNs sind hoher Datenverlust, Netzinstabilitäten und geringe Datenraten. RPL bietet für diese Eigenschaften eine kostenoptimierte Wegwahl für die Vermittlung von Nachrichten, die in RFC6550 spezifiziert sind [11].
- Während das Transport(schicht)protokoll TCP insbesondere für LLNs nicht geeignet erscheint, wird das verbindungslose Transportprotokoll UDP bevorzugt. [12] Vorteilhaft für low-power-Netze ist nicht nur der kleinere Header von nur 7 Bytes. Durch das Fehlen von Handshake und Verbindung, Paketsortierung oder Fehlerbehebung ist die Systembelastung niedrig und eine höhere Geschwindigkeit gegeben. Dennoch unterstützen einige IoT-OSs TCP vollständig für stärkere Systeme mit dauerhafter Stromnetzanbindung.
- CoAP(Constrained Application Protocol) ist ein Protokoll der Anwendungsschicht und wurde für den Nachrichtenaustausch eingebetteter Systeme (M2M, machine to machine) entworfen. Dadurch, dass HTTP TCP als Transportprotokoll benutzt, ist HTTP nicht für LLNs geeignet. CoAP kann zur Übertragung UDP benutzen. Für eine sichere Datenübertragung wird zusätzlich das Transport-Verschlüsselungsprotokoll DTLS (Datagram Transport Layer Security) benutzt [13].

[3]Carrier Sense Multiple Access/Collision Avoidance: Prinzip, um mehrmalige Funkschnittstellen-Belegung und Sende-Kollisionen zu vermeiden

[4]Maximum Transmission Unit: maximale Paketgröße

III. Arten von Betriebssystemen

A. Kernel-Arten

Im monolithischen Kernel werden alle Dienste und Treiber im Kernel-Modus verarbeitet. Durch den gemeinsam benutzten Adressraum kann ein Fehler allerdings das gesamte System abstürzen lassen. Für eine Erhöhung von Stabilität und Sicherheit versucht der Mikro-Kernel, sämtliche Dienste und Treiber im User-Modus zu verarbeiten. Durch die Entwicklung des L4-Mikro-Kernels können Mikro-Kernel heute auch mit der Geschwindigkeit von monolithischen Kerneln mithalten [14]. Nano-Kernel, Pico-Kernel, Femto-Kernel etc. sind Bezeichnungen für Mikro-Kernel, die mehr durch Marketing als durch ihre tatsächliche Größe entstanden sind. Tatsächlich wurde durch die Verwirrung der Bezeichnungen und die Entwicklung des L4-Kernels letztendlich eine klare Definition für den Mikro-Kernel gefunden: Der Mikro-Kernel ist minimal und beinhaltet nur Code, der im Kernel ausgeführt werden muss. Durch diese Definition sind kleinere Kernel nicht möglich [15]. Der Hybrid-Kernel (Makro-Kernel) macht sich die jeweiligen Vorteile von monolithischem und Mikro-Kernel zu Nutze. Er unterscheidet sich insofern von einem monolithischen Kernel, als einige Module aus dem User-Modus des monolithischen Kernels im Kernel-Modus ausgeführt werden. Welche Module dies genau sind, wird nicht genau spezifiziert. Sie sind von der Implementierung des OS abhängig. Einige Kernel besitzen eine HAL(Hardware-Abstraction-Layer), die Programmen durch Virtualisierung keine direkte Speicheradressierung erlaubt. Der Exo-Kernel bietet diese Möglichkeit durch eine minimalistische Hardware-Abstraktion und kann damit Programme beschleunigen. Dazu wird vom Exo-Kernel festgestellt, ob Speicher verfügbar ist und dieser vom Programm benutzt werden darf [16].

B. RTOSs

Gängige Multitasking-OSs (z.B. Desktop-OSs) können der Echtzeitanforderung nicht standhalten. Die in Millisekunden gemessene Echtzeit kann je nach Zweck variieren. Für Audio- und Videoübertragungen sind 250ms tolerierbar. In der industriellen Fertigungstechnik hingegen müssen Werte unter 10ms eingehalten werden, um noch als Echtzeit zu gelten [17]. Milliarden von intelligenten, vernetzten Systemen sollen in Zukunft Entscheidungen treffen und Probleme lösen. Für viele Systeme bzw. Subnetzwerke im IoT ist es wichtig, in Echtzeit zu operieren. Das IoT selbst operiert nicht zwingend in Echtzeit, beinhaltet aber durchaus Echtzeitnetzwerke.
Ob ein RTOS auf einem System verwendet wird, hängt vom jeweiligen System und dessen Anwendung ab. Bei größeren Programmen, die mehr Funktionen, mehr Kommunikationsschnittstellen und eine größere Anzahl an Interrupt-Quellen haben, ist der Einsatz eines RTOS wahrscheinlich, insbesondere bei Programmgrößen ab 1MB. Umgekehrt werden RTOS nicht nötigerweise bei Programmen, die kleiner als 64kB sind verwendet. Die Verwendung von RTOS von Drittanbietern auf einem MCU löst unter den MCU-Herstellern Debatten aus, ob man die 100%ige Kontrolle über die Hardware in die Hände des Drittanbieter-RTOS legt. [18] Bei der Verwendung von Drittanbieter-Software wird nicht die maximal mögliche Effizienz erreicht, man spart aber Produktionskosten ein. Vorteilhaft an der Benutzung eines RTOS auf einem MCU sind die effizientere Verwaltung der CPU-Ressourcen und die Bereitstellung von Debug- und Analysetools für komplexere Programme. Durch standardisierte Stacks und vorgetestete und -integrierte Treiber ist es möglich, Programme wiederzuverwenden. Das preemptive Multitask-Design eines RTOS erlaubt die Einbindung neuer Funktionen in Programme, ohne den Echtzeitstatus anderer Funktionen zu verletzen. [18] Für Entwickler ist es einfacher, mit Threads zu programmieren, als sich einzeln auf Prozessorallokation zu fokussieren; allerdings kann eine unsachgemäße Benutzung einen extremen CPU-Overhead verursachen und viel Speicher benötigen. Ebenso muss darauf geachtet werden, hohe Prioritäten nur an Threads zu vergeben, die unbedingt ausgeführt werden müssen. [19]
Unter RTOS erfolgt eine Einteilung in soft-RTOS und hard-RTOS:

- Das soft-RTOS kann nicht garantieren, dass eine vorgegebene (Interrupt-)Latenz unterschritten wird. Latenzen werden üblicherweise eingehalten, aber eine Garantie kann höchstens im Durchschnitt erfolgen. Dies ist z.B. bei Videoübertragungen mit durchschnittlich 25 fps der Fall.
- Das hard-RTOS hält die Anforderung deterministisch ein, d.h. in allen Fällen muss die Latenz unterschritten werden. Dies ist z.B. in Flugzeugkontrollsystemen der Fall.

Bei hard-RTOSs bedeutet eine verspätete Antwort eine falsche Antwort, während bei soft-RTOSs eine verspätete Antwort nur eine geringe Qualität bedeutet [20].

1) Latenzminimierung: Um die Interrupt-Latenz gering zu halten, gibt es zwei Möglichkeiten [21].

- Für den Fall, dass ein Interrupt ein Interrupt unterbricht (verschachtelter/nested-Interrupt), werden Interrupts nur wieder zugelassen, wenn der Prozessor bereits genügend Daten auf den Stack geschrieben hat. Wurde der nested-Interrupt behandelt, wird der ursprüngliche Interrupt wieder bearbeitet.
- Besteht für Interrupts eine Priorisierung, sollen Interrupts mit geringer Priorität maskiert werden bzw. soll diesen nicht viel Zeit zugeteilt werden.

2) Scheduling: Ein RTOS hat gegenüber einem Desktop-OS erweiterte Scheduling-Algorithmen. Sie bestimmen, wann und wie lange eine Applikation den Prozessor beanspruchen darf. Üblicherweise unterstützen moderne OSs (z.B. Linux, Mac OS ab Version 9, Windows ab Windows 95) neben kooperativem auch preemptives Scheduling. [22]

- Beim kooperativen Scheduling bekommt ein Prozess sämtliche Ressourcen zur Verfügung gestellt, die er benötigt und das OS wartet, bis der Prozess diese Ressourcen durch die Fertigstellung seiner Aufgabe wieder freigibt.
- Das preemptive Scheduling kann dem Prozess bereits vor der Fertigstellung seiner Aufgabe die Ressourcen wieder entziehen. Ein Prozess bekommt beim preemptiven Verfahren bereits vorher determinierte Zeitscheiben zugeteilt. Der Scheduler kann so besser auf Deadlines eingehen und es wird vermieden, dass ein einzelner Prozess Zeit und Ressourcen für sich alleine beanspruchen kann.

In einem preemptiven, prioritätsbasierten Scheduler verarbeitet die CPU den Prozess mit der höchsten Priorität. Dazu wechselst der Kernel den Kontext, solange der derzeitig verarbeitete Prozess eine niedrigere Priorität hat. Der Round-Robin-Algorithmus verteilt unter bereiten Prozessen gleicher Priorität fair CPU-Leistung über Zeitscheiben. Das in RTOSs weit verbreitete FPS (Fixed-Priority-Scheduling) ist ein System zur Verarbeitung der Prozesse, die die höchste Priorität haben. Dabei ist die Priorität fix. Die Priorität wird anhand der zeitlichen Voraussetzungen festgelegt. EDF (Earliest-Deadline-First) hingegen ist ein dynamischer Scheduling-Algorithmus. Der nächste ausgeführte Prozess ist der mit der kürzesten bzw. nächsten Deadline. Während die relativen Deadlines bekannt sind, werden die absoluten Deadlines zur Laufzeit berechnet. Sollte das System überlastet sein, können Deadlines und Prioritäten nicht mehr ausreichend sein. Im VBS (Value-Based-Scheduling) werden den Prozessen Werte zugeordnet. Dieser Wert beschreibt die Rückgabe bei vorzeitiger Verarbeitung des Prozesses, d.h. vor seiner Deadline. Ziel ist es, bei verschiedenen Werten die Summe aller Werte zu maximieren. Verpasste Deadlines sind dabei zweitrangig. Die Forschung beschäftigt sich hingegen mit gleichen Werten und der Einhaltung der Deadlines [23]. Der oft eingesetzte Algorithmus RMS (Rate-Monotonic-Scheduling) ist ein preemptiver Algorithmus. Er vergibt Prioritäten umgekehrt proportional zu den Deadlines der Prozesse [20].

3) Modularität: RTOSs wurden lange Zeit als Softwarepakete ausgeliefert, die sofort benutzbar sind, ähnlich den Windows-OSs. Durch die hohe Anzahl und die Verschiedenheit der Systeme wird es zunehmend wichtiger, RTOSs modular auszuliefern, ähnlich den Linux-Distributionen. Nicht jedes System benötigt dieselben Module. Für Effizienz und

Zuverlässigkeit eines RTOS können nicht benötigte Module entfernt werden [24].

4) Sicherheitsmaßnahmen: In militärischen oder Kommunikations-Anwendungen ist es keine Option, dem *fail-first patch-later*-Ansatz zu folgen. Durch die Vernetzung von Straßenverkehr und Finanzdienstleistungen wird spätestens klar, dass ein OS auf Anhieb sicherstellen muss, dass es durch unerlaubtes Eindringen in die vernetzten Systeme nicht möglich ist, Menschenleben und Eigentum zu gefährden. MILS (Multiple Independent Levels of Security/Safety) ist eine Architektur für hochsichere Systeme. Mathematische Verifizierung wird durch Reduzierung auf folgende vier Kernkonzepte möglich [25]:

- Informationsflusssicherung: Der Informationsaustausch wird lediglich zwischen Komponenten gleicher Sicherheitsstufe oder vertrauenswürdigen Sicherheitsmonitoren erlaubt.
- Datenisolation: Die Datenseparation findet durch den Kernel statt.
- periodische Verarbeitung: Verarbeitung von klassifizierten und nicht klassifizierten Informationen zu unterschiedlichen Zeiten.
- Schadensbegrenzung: Ein Fehlschlag soll nur lokal erkannt und behandelt werden. Dieser darf nicht das gesamte System beeinflussen.

Durch die Wahl der Kernel-Architektur können Sicherheit und Stabilität immens erhöht werden. Üblicherweise werden Mikro-Kernel in RTOSs verwendet. Dennoch werden auch andere Kernel-Arten eingesetzt, sogar monolithische Kernel [26]. Die MILS-Architektur gliedert sich in drei Schichten. Der (i) Seperation-Kernel ist eine (bis zu 5000 Zeilen Code) kleine, mathematisch verifizierte und vertrauenswürdige Software, die Zeit und Raum separiert. Die (ii) Middleware ist z.B. ein PCS[5], das MILS in Hinblick auf Separation zwischen Kernel und Anwendung erweitert. Middleware und Kernel müssen die „NEAT“-Eigenschaften haben [25]:

- **N**on-bypassable: Sicherheitsfunktionen dürfen nicht umgangen werden können.
- **E**valuable: Sicherheitsfunktionen müssen klein und einfach gestaltet sein, um mathematisch verifiziert und evaluiert werden zu können.
- **A**lways Invoked: Die Sicherheitsfunktionen müssen immer aktiviert sein.
- **T**amperproof: Die Sicherheitsfunktionen dürfen nicht durch schädlichen Code verändert werden.

Mit Anwendung des MILS-Konzepts in heutigen RTOS ist es also möglich, dass in der vernetzten Küche zuhause der gleiche Sicherheitsmechanismus wie in einem militärischen Flugzeug, das Atombomben abwerfen kann, installiert ist [27].
IEEE 802.15.4 gibt zwar den Verschlüsselungsalgithmus vor, es ist jedoch keine Aussage darüber getroffen worden, wie die Schlüssel verwaltet und welche Authentifizierungsrichtlinien angewendet werden müssen. ZigBee ist ein höherschichtiger Standard, der auf IEEE 802.15(.4) aufbaut, das Schlüssel-Management übernimmt und Sicherheitsrichtlinien einführt. ZigBee fügt zwei Extra Sicherheitsschichten über 802.15.4 ein. Für die Netzwerk- und Programmsicherheitsschicht werden Richtlinien, aufbauend auf dem AES 128-Verschlüsselungsalgorithmus[6], implementiert. Die ZigBee-Sicherheitsmechanismen sehen drei Arten von Schlüsseln vor. Der Master-Key ist in jedem Knoten vorinstalliert und soll die Link-Keys geheim halten. Jedes System-Paar besitzt je einen einzigartigen Link-Key. Der Link-Key wird zur Sicherung der Programmschicht benutzt und soll den Datenaustausch zwischen Systemen schützen. Der einzigartige Network-Key wird vom Trust-Center (z.B. ein dediziertes System) generiert. Dieser wird in regelmäßigen Intervallen neu generiert - der alte Schlüssel wird dann verwendet, um den neuen Schlüssel im Netzwerk zu verteilen. Der Network-Key wird für Management und Kommunikationskontrolle des Netzwerkes benutzt. ZigBee stellt zwei Arten von Sicherheitsrichtlinen zur Verfügung. Im Residential-Mode teilt das Trust-Center lediglich den Network-Key. Diese Richtlinie ist ideal für drahtlose Netzwerke mit beschränkten Ressourcen. Im Commercial-Mode hingegen werden Network-Key und sämtliche Link-Keys vom Trust-Center geteilt und haben einen hohen zusätzlichen Speicheraufwand. [28]

5) Speicherallozierung: Bei Desktop-OSs geht man von unbegrenzt, d.h. immer ausreichend Speicher aus. Bei eingebetteten Systemen ist davon auszugehen, dass nur sehr wenig Speicher zur Verfügung steht und dieser durch etwaige Sicherheitsmechanismen und implementierte Protokolle noch geringer wird. Ein RTOS muss deswegen die Fähigkeit besitzen, wenig Speicher effizient zu allozieren und freizugeben. Da ein System auf unbestimmte Zeit einsatzfähig sein muss und es zu keinen Ausfällen kommen sollte, dürfen sich keinerlei Speicherlecks bilden. Die dynamische Speicherallozierung, die Speicher zur Laufzeit und nicht zur Kompilierzeit alloziert, ist daher keine Option für ein RTOS. Durch Allozierung zur Laufzeit ist die Zeit, die zum Allozieren benötigt wird, unvorhersehbar. Ein weiterer Nachteil ist die Speicherfragmentierung, die dynamische Allozierung als Allozierungsart ausschließt, sofern keine Sicherungsmechanismen implementiert wurden [29]: Es ist möglich, dass Programmen erlaubt wird, eine dynamische Allozierung zu benutzen [26]. Viele RTOS bieten Dienste, die dynamische Allozierung durch Immunität vor Speicherfragmentierung ermöglichen. Unterstützend kann eine gute Fehlerbehandlung implementiert werden, die bei Speicherüberläufen verhindert, dass ein anderer Stack zerstört wird [29].

6) Energiemanagement: Ein RTOS kann auf verschiedene Arten Energie einsparen. Grundsätzlich soll sich ein MCU (und damit Scheduler und CPU) die meiste Zeit im Zustand Schlafmodus (Sleep-Mode) befinden. Ein RTOS implementiert den Sleep-Mode; allerdings ist stark vom System abhängig, wie der Sleep-Modus genutzt und angepasst werden sollte. Der Sleep-Mode beinhaltet je nach RTOS verschiedene Energie-Modi, die sich im Stromverbrauch und den verfügbaren Modulen unterscheiden. Üblicherweise werden folgende Standardmodi von einem RTOS implementiert, wobei

[5]Partitioning Communications System - Sicherheitsarchitektur, die auf Informationsflussrichtlinien basiert

[6]Advanced Encryption Standard - ein hochsicherer Verschlüsselungsstandard

sich allerdings systemoptimierte Modi zwischen diesen Standardmodi befinden können und einzelne Kriterien der Modi abweichend sein könnten. Als Beispiel für die Dimensionen der Energieeinsparung sollen die Modi anhand des 8Bit-AVR-MCU ATmega32 und des 32Bit-MCU EFM32 evaluiert werden [30].

(i) Der Active- oder Run-Modus ist der einzige Modus, in dem aktiv Applikationen ausgeführt werden. Dabei ist der Active-Modus kein Sleep-Modus, weil er den meisten Strom verbraucht und die CPU aktiv tätig ist. (ii) Die CPU ist im Idle-Modus gestoppt, da hier keine Applikationen ausgeführt werden. Der Idle-Modus ist die erste Stufe der Energiesparmaßnahmen. Der ATmega32 spart ca. 70% Energie im Idle-Modus. Bei einer Stromaufnahme von 3,3V pro MHz werden im Active-Modus 1,1mA und im Idle-Mode lediglich 0,3mA verbraucht [31]. (iii) Der Stand-By-Modus ist der nächste Basismodus unter dem Idle-Modus. Der Unterschied zum Idle-Modus ist, dass viele Module nicht mehr verfügbar sind. Z.B. werden ADC[7](kann nicht mehr als Weckruf benutzt werden) und CPU-Timer abgeschaltet - der Oszillator ist jedoch noch aktiv. Der Code befindet sich im Gegensatz zum Idle-Modus nicht mehr im Speicher [32]. Der Stand-By-Modus verbraucht im ATmega32 35μA. (iv) Der Shutdown-Modus (Deep-Sleep-Modus, Power-Down-Modus) spart die meiste Energie. Dabei sind CPU, Speicher und sämtliche Peripherie abgeschaltet. Sämtliche Oszillatoren sind ebenfalls abgeschaltet. Dadurch, dass sich der Hauptoszillator erst stabil einschwingen muss, benötigt ein Weckruf wesentlich länger. Je nach Implementierung reichen Interrupts oder Resets als Weckruf. Tatsächlich heruntergefahrene Systeme müssen erneut angeschaltet werden. Dabei variiert auch der Stromverbrauch. Nicht gänzlich heruntergefahrene Systeme benötigen ca. 0,3μA. Im echten Shutdown-Modus werden davon ca. 10% verbraucht. Aus dem Shutdown-Modus benötigt der EFM32 160μs. Bei einem Verbrauch von 0,4μA im Shutdown-Modus verbraucht der EFM32 innerhalb von 7 Jahren ca. 25% eines AA-Batterie-Paars. [33] Im Vergleich verbraucht ein typischer Wireless-Sensor-Knoten 25mA und erschöpft das Paar AA-Batterien nach ca. 5 Tagen [34]. Der Sleep-Modus sollte so oft wie möglich in der tiefstmöglichen Stufe benutzt werden.
Ein RTOS kann weiterhin die vorhandene Hardware unterstützen, indem einige einfache Regeln befolgt werden. Statt die CPU oft mathematische Funktionen rechnen zu lassen, können stattdessen Wertetabellen benutzt werden. Ein Polling (Schalterzustände abfragen) sollte vermieden und durch die Benutzung von Interrupts ersetzt werden. Größere Unterprogramme benötigen lange für eine Ausführung. Stattdessen sollten entsprechend kürzere Makros (Unterprogrammvariante) verwendet werden. [35] Sollten Lichtquellen am System angebracht sein und das Betriebssystem Einfluss auf deren Helligkeitslevel haben, kann weiter Energie eingespart werden, indem deren Helligkeitslevel auf ein minimales Niveau gesenkt wird.

[7]Analog-Digital-Converter, wandelt analoge Eingangssignale in digitale Daten um.

C. gängige OSs

Typische OSs, die als Desktop-OS, Server-OS oder mobile-OS Anwendung finden, sind Schwergewichte. Windows OSs werden als Komplettpaket ausgeliefert und bieten kaum Anpassungsmöglichkeiten, da es proprietäre OSs sind. Der Fokus der Windows 9x- oder Windows NT-Produktlinien liegt in der Verwendung auf Desktopsystemen. Die Mac OS-Produktlinie ist ebenfalls proprietär und besaß ursprünglich die Einschränkung der Benutzung auf Apple-Desktop-Systemen. Andere unixoide OSs wie Linux können durch Modularisierung stark entschlackt werden. Die Konfigurationsmöglichkeiten von Linux sind breit gefächert.
Die Server-OSs finden ebenfalls kaum Anwendung auf MCUs. Die OSs beinhalten Software, die auf die Bereitstellung von Diensten ausgelegt ist. Im Gegenzug könnten typische Desktop-OS-Features wie z.B. Sound-Ausgabe oder grafische Benutzeroberfläche entfernt worden sein.
Die mobilen Ableger Windows mobile, iOS und Android haben ähnliche Merkmale wie ihre Desktop-Pendants. Der Anwendungsfokus begrenzt sich allerdings auf mobile Systeme, insbesondere Smartphones.
Keines dieser OSs, von denen die Desktop-OSs typischerweise einen monolithischen Kernel besitzen, erfüllt die RTOS-Bedingungen, durch die hohe Anzahl an Modulen und die Fokussierung auf Benutzerfreundlichkeit, Design und Anwendungsbreite. Ebenso können diese OSs von leistungsstarken Systemen ausgehen. Es steht zumeist viel Speicher, eine schnelle CPU und eine dauerhafte Anbindung an das Stromnetz oder Akkus mit viel Fassungsvermögen zur Verfügung. Protokolle müssen daher nicht in den Funktionen beschränkt werden und höhere Latenzen sind nicht kritisch. Der Speicher kann ohne größere Probleme auch dynamisch alloziert werden. Die Sicherheit wird garantiert durch zusätzlich installierte Programme und umfangreichere Protokolle wie die der IEEE 802.11-Familie (WLAN). Zwar ist das Energie-Management auf mobile Systeme angepasst, diese benötigen aber meist den aktiven Modus für die Bereitstellung der gewünschten Dienste und können innerhalb von Tagen oder Stunden keine Energiereserven mehr besitzen.

IV. Vergleich bekannter OSs

A. Warum nicht einfach ein Linux benutzen?

Da die Mac OS- und Windows 9x/Windows NT-Produktlinien aufgrund ihrer immensen Einschränkungen nicht als IoT-OSs benutzt werden können, könnte man annehmen, dass Linux ein geeigneter Kandidat für ein IoT-OS ist. Ein Linux-Software-Paket kann durch die Modularisierung auf ein Minimum beschränkt werden. Module, die für eine Benutzung mit eingebetteten System nötig sind, könnten hinzugefügt werden. Dennoch ist Linux ein extrem großes OS und ist nicht in der Lage, 8Bit- oder 16Bit-MCUs zu steuern. Selbst einige 32Bit-MCUs haben nicht genug Speicher für den Linux-Kernel, der einen minimalen Memory-Footprint von ca. 1MB besitzt. [36]

B. IoT-OS = RTOS?

Einige IoT-OSs sind auch RTOSs. Viele unterstützen die Echtzeitanforderung allerdings nur partiell (hardwareabhängig) oder agieren als soft-RTOS. Die Anwendungsmöglichkeiten für RTOSs nehmen jedoch zu und es ist zu erwarten, dass der Anteil an benutzten RTOSs auf MCUs zunehmen wird.

C. IoT-OSs auf eingebetteten Systemen

Im folgenden werden einige bekannte IoT-OS im Detail vorgestellt.

1) TinyOS: TinyOS ist eines der IoT-OSs, welches keinerlei Echtzeitunterstützung anbietet und einen monolithischen Kernel besitzt. Jedoch besitzt TinyOS einen sehr geringen Memory-Footprint von nur 400 Byte RAM und unter 4kB ROM. Somit kann TinyOS auf extrem kleinen Systemen installiert werden. Das Speicher-Management in TinyOS erfolgt statisch. Seit Version 2.1 wird das einfach zu benutzende Multithreading-Programmiermodell unterstützt, welches mit dem effizienteren Event-Driven-Programmiermodell gekoppelt ist. Ältere Versionen benutzen einen FIFO-Scheduler, während in neueren Versionen auch ein EDF-Scheduler unterstützt wird. Durch Optimierung auf kleine Systeme existieren kaum Sicherheitsmechanismen. Erst seit Version 2.1 wurde ein Speicherschutz implementiert, der durch den resource-to-resource-compiler Deputy Typen- und Speichersicherheit für C-Code garantiert. Mit TKN15.4 wurde für Version 2 von TinyOS eine Implementierung von IEEE 802.15.4 vorgenommen [37]. Version 2.1.1 integrierte das Protokoll 6LoWPAN in TinyOS. Zusätzlich zu TinyRPL wird das ad-hoc-Netzwerk-routing-Protokoll TYMO implementiert, welches auf DYMO (Dynamic MANET On-demand Routing, MANET = Mobile Ad-hoc Networking) basiert, jedoch eine abgeänderte Paketstruktur besitzt. Als Programmiersprache für Anwendungsentwicklung wird der low-level C-Dialekt nesC verwendet. Durch die Annahme, dass zu jedem Zeitpunkt immer nur ein Programm ausgeführt wird, wird ein Single-Level-Dateisystem benutzt. Weitere zusätzliche Features von TinyOS sind Kommunikationssicherheit, Datenbank- und Simulationsunterstützung. [6]

2) Contiki: Contiki ist ein IoT-OS mit einem modularen Event-Driven-Kernel, der optional preemptives Multithreading unterstützt. Für das Multithreading werden Protothreads benutzt, die einen kleinen (2 Byte) Speicher-Overhead haben. Der Memory-Footprint von Contiki ist größer als bei TinyOS. Dennoch ist auch dieser mit unter 2kB vergleichsweise gering. Allerdings liegt der ROM-Footprint bei ca 30kB. Da in Contiki Speicher dynamisch alloziert wird, wird ein Managed-Memory-Allocator verwendet, um Speicherfragmentierung zu verhindern. Allerdings gibt es keinen Speicherschutz zwischen verschiedenen Programmen. Der Scheduler von Contiki verarbeitet synchrone und asynchrone Events; dabei werden synchrone Events sofort an den Prozess weitergeleitet, während asynchrone Events in eine Warteschlange eingereiht und später an den Zielprozess weitergeleitet werden. Da Events bis zur Fertigstellung ausgeführt werden und dem Interrupt-Handler nicht erlaubt ist, neue Events zu versenden, existiert keine Prozesssynchronisation in Contiki. Contiki unterstützt verschiedene Netzwerkmechanismen: Für 8Bit-MCUs wird mit uIP eine lightweight-TCP/IP-Implementation unterstützt. Der IPv6-Stack unterstützt ebenfalls RPL unter dem Namen ContikiRPL. Da Contiki nicht multicast-fähig ist, existieren keine Implementationen von IGMP (Internet Group Management Protocol) oder MLD (Multicast Listener Discovery). Contiki bietet eine partielle Unterstützung für Echtzeitanfragen [38]. Der RIME-Stack ist ein alternativer Netzwerk-Stack, der Programmen erlaubt, eigene Routing-Protokolle zu verwenden, wenn der IPv6-Overhead zu groß wird. Programme für Contiki werden in der Programmiersprache C geschrieben. Das Coffee-Dateisystem für Flash-basierte Sensoren stellt eine effiziente Programmierschnittstelle mit geringem Memory-Footprint und Performance-Overhead bereit. Bisher scheint keine Implementation von Kommunikationssicherheitsrichtlinien vorgenommen worden zu sein. Allerdings existiert der Vorschlag, Sicherheitsmaßnahmen mit dem Namen ContikiSec zu implementieren. [6]

3) RIOT: RIOT ist ein multithreading-fähiges (hard-)RTOS, dass einen Fokus in der Entwicklungsunterstützung hat. Trotz der hard-real-time-Eigenschaft ist RIOT ein robustes RTOS, durch die Verwendung einer Mikro-Kernel-Architektur. Durch die modulare Konfiguration von RIOT ist ein Memory-Footprint von nur wenigen 100 Bytes RAM und 5kB ROM möglich. Threads haben lediglich einen Overhead von je ca. 25 Bytes. Der Speicher kann in RIOT dynamisch und statisch alloziert werden. Auch RIOT bietet keinen Speicherschutz [39]. Der prioritätsbasierte RIOT-Scheduler ist tickless; statt mehrfach pro Tick aufzuwachen und festzustellen, ob ein Timer abgelaufen ist, wird gewartet bis der nächste Timer ausläuft: Der Scheduler wird, sobald keine Tasks mehr abgearbeitet werden müssen, das System in den Idle-Modus versetzen. Dessen einzige Aufgabe ist es, festzustellen, in welchen tieferen Schlafzustand das System hineinversetzt werden kann, um Energie zu sparen. Das System kann im Idle-Modus nur durch externe oder Kernel-Interrupts geweckt werden. RIOT unterstützt die Protokolle 6LoWPAN, CoAP, RPL und IEEE 802.15.4 für Systeme mit beschränkten Ressourcen. Ebenso werden TCP, UDP und IPv6 voll unterstützt. Dadurch, dass RIOT in C geschrieben wurde, kann neben C auch C++ als Programmiersprache benutzt werden. Standard-Tools wie gcc, gdc und valgrind werden mitgeliefert. RIOT ermöglicht somit eine starke Vereinfachung der Programmentwicklung für das breite Spektrum der verschiedenen Systeme gegenüber anderen (IoT-)OSs. In Zukunft soll RIOT voll POSIX-konform sein. [40] [41] Die Entwicklung von ParSec für event-driven WSNs soll auch in RIOT die Kommunikationssicherheit garantieren. [42]

4) Nano-RK: Nano-RK ist ein speziell für WSNs entwickeltes preemptives Multithreading-RTOS. Der Memory-Footprint liegt bei 2kB RAM und 18kB ROM. Die Speicherallozierung erfolgt statisch, bietet jedoch keinen Speicherschutz. Es existieren zwei verschiedene Prioritäts-Scheduling-Ebenen in Nano-RK (Netzwerk-Level und Prozess-Level). Auf dem Prozess-Level wird der jeweils höchstpriorisierte Thread ausgeführt. Als Scheduling-Algorithmus wird RMS verwendet; die Priorität ist höher, wenn der Thread eine kürzere Periode hat. Darüber hinaus wird Rate-Harmonized-Scheduling implementiert. Durch Gruppierung der Ausführung von Threads

sollen CPU-Idle-Zeiten eliminiert werden und Energie gespart werden. Für Systeme mit beschränkter Energiezufuhr wurde in Nano-RK RT-Link eingeführt. Das Ziel von RT-Link ist es, die Lebensdauer von batteriebetriebenen Sensornetzwerken zu erhöhen. Dabei kann das über einen TDMA[8]-Link-Layer eingebundene RT-Link eine kollisionsfreie Echtzeitkommunikation garantieren. Dabei ist das dem IEEE 802.15.4 ähnliche RT-Link für Operationen zwischen synchronisierten Multi-Hop-Netzwerken entwickelt worden. Nano-RK unterstützt außerdem eine Basis-Implementation von 6LoWPAN und RPL, jedoch kein CoAP [43]. Programme für Nano-RK werden in der Programmiersprache C geschrieben. [6]

D. Vergleich

Während RIOT die Echtzeitfähigkeit als eines seiner Hauptziele angibt, unterstützen auch Nano-RK und Contiki (partiell) Echtzeit. TinyOS verfügt hingegen nicht explizit über die Fähigkeit, in Echtzeit zu antworten. Zwar wurde in neueren TinyOS-Versionen zusätzlich zum FIFO-Scheduler der EDF-Scheduler implementiert; dieser ist jedoch immer noch unzureichend für eine verlässliche Echtzeitunterstützung. Das Echtzeitmodul rtime von Contiki ermöglicht hingegen Basisfunktionen für Echtzeit-Scheduling [38] [44]. Nano-RK bietet die Echtzeitunterstützung über die Implementierung der Scheduler RMS und RHS. Durch Implementierung des tickless-Schedulers wird Echtzeit in RIOT ermöglicht. RIOT und Nano-RK sind demnach gegenüber Contiki und TinyOS die bessere Wahl für Echtzeitanwendungen.
RIOT und Nano-RK sind pure Multithreading-OSs. Contiki unterstützt Protothreads, während es ebenso ein event-driven-Programmiermodell besitzt. TinyOS kombiniert seit Version 2.1 wie Contiki die Vorteile des effizienten Event-basierten Programmiermodells mit Multithreading via einer Implementierung von TOSThreads. Die nachträgliche Implementierung von Multithreading soll dabei sehr einfach sein. Es existiert eine TOSThread-API für C und nesC, welche die Programmiersprache für TinyOS ist. [45] Contiki, Nano-RK und Riot unterstützen pures C, während RIOT auch die Möglichkeit bietet, in C++ zu programmieren. RIOT erwähnt eine einfache Anwendungsentwicklung ebenso als eines der Hauptziele. Anhang A [39] zeigt, wie das „Hello World“-Programm in TinyOS, Contiki und RIOT programmiert werden kann. Dabei fällt auf, dass unter RIOT in weitläufig üblichem C-Code programmiert werden kann und dadurch tatsächlich eine einfachere Anwendungsentwicklung ermöglicht wird.
TinyOS und Nano-RK benutzen einen monolithischen Kernel. Die Kernel dieser OSs können z.B. durch Fehler eines Treibers einen kompletten Systemabsturz erzeugen. RIOT vermeidet dies durch die Benutzung eines Mikro-Kernels. Der modulare Kernel von Contiki kann Module bei Bedarf laden. Der modulare Kernel kann allerdings beim Laden eines fehlerhaften Moduls ebenso einen Systemabsturz verursachen. TinyOS bietet als einziges verglichenes OS einen Speicherschutz (Deputy). Trotz möglicher Bildung von Speicherlecks ermöglicht Contiki nur dynamische Speicherallozierung. RIOT ermöglicht hingegen statische und dynamische Allozierung von Speicher. TinyOS und Nano-RK vermeiden entstehende Speicherlecks; lediglich statische Speicherallozierung wird unterstützt. Kommunikationssicherheit wird in TinyOS und Contiki durch die Implementierungen von TinySec bzw. ContikiSec gewährleistet. Nano-RK bietet hingegen keine zusätzliche Kommunikationssicherheit. In RIOT soll ParSec für die Kommunikationssicherheit implementiert werden. [46] Im Sicherheits- und Stabilitätsvergleich überzeugt TinyOS. Trotz Benutzung eines monolithischen Kernels bindet TinyOS gegenüber den anderen OSs sämtliche erwähnte Mechanismen ein und schützt mit statischer Speicherallozierung vor Speicherlecks. Die Sicherheitsarchitektur MILS ist in keinem der OSs vorhanden.
Die Tabellen I und II vergleichen die erwähnten Features der OSs in Listenform.

Das Protokoll IEEE 802.15.4, welches ebenfalls Sicherheit und Stabilität über das OS hinaus im Netzwerk erweitern kann, ist lediglich in Contiki nicht implementiert. Nano-RK und TinyOS besitzen eigene bzw. ähnliche Implementierungen. 6LoWPAN wird jedoch von allen erwähnten OSs unterstützt. Contiki implementiert 6LoWPAN unter dem Namen SICSlowPAN, TinyOS benennt seine Implementierungen 6LoWPANCLi und BLIP (b6lowPAN). Ebenso ist RPL in jedem der genannten OSs vorhanden. Contiki und TinyOS nennen ihre Implementierungen ContikiRPL bzw. TinyRPL. Als Alternative zu HTTP ist CoAP nur in Nano-RK nicht vorhanden. Nano-RK besitzt zwar eine ZigBee-Implementierung, bietet aber ansonsten kaum zusätzliche Protokolleinbindungen. Contiki hingegen besitzt diese Implementierung nicht, unterstützt allerdings eine Vielzahl anderer Protokolle. Darunter IPv4, HTTP, TCP, uIP(v6) und RIME. TCP und ZigBee werden auch von RIOT und partiell von TinyOS unterstützt. TinyOS unterstützt darüber hinaus noch TYMO. Die umfangreichste Unterstützung verschiedener Protokolle bietet Contiki. Da kein ZigBee und IEEE 802.15.4 implementiert sind, könnten aber wichtige Sicherheitsmechanismen, die in den anderen verglichenen OSs vorhanden sind, fehlen. Nano-RK hat einen sehr kleinen Protokoll-Stack; der Protokoll-Stack von RIOT und TinyOS ist umfangreicher und scheint solider als die von Nano-RK und Contiki zu sein.
Tabelle III listet, welche Protokolle im Einzelnen von den OSs unterstützt werden.

V. Resümee

In diesem Artikel wurde analysiert, was RTOS bzw. IoT-OSs sind. Typische Stromsparmechanismen (Sleep- und Idle-Modi) und der Bedarf neuer Protokolle wie 6LoWPAN, RPL und CoAP, sowie die Bereitstellung von effektiven Sicherheitsmechanismen (ZigBee, MILS), Schedulingalgorithmen (EDF, FPS, RMS, VBS) und effizientem Speicher-Management (Zweck statischer und dynamischer Allozierung) wurden untersucht. Die IoT-OSs TinyOS, Contiki und die IoT-RTOSs RIOT und Nano-RK wurden miteinander verglichen.

VI. Ausblick

Durch den Aufschwung des IoT ist einiges zu erwarten: Nicht nur die (Weiter)entwicklung von open-source und proprietären IoT-OSs. Z.B. die Erforschung von selbstaufladenden

[8]Time Division Multiple Access: Multiplexverfahren für Nachrichtenübertragungen

Akkus könnte Systeme so verändern, dass sich der Fokus eines IoT-OS stetig aktuellen Anforderungen anpasst. Bisher ist das IoT noch in den Kinderschuhen, wird sich aber schnell etablieren können wie das Internet selbst oder die Entwicklung von Handys und Smartphones. Die Schätzung, dass in den Jahren 2015 bis 2020 35 Milliarden zusätzliche IoT-Systeme das tägliche Leben und die Industrie maßgeblich verändern, wirft aber auch kritische Fragen auf. Wie verwenden Globalplayers wie Google, Apple und Microsoft ihre bereits immense Marktmacht, wenn sämtliche Lebensbereiche weltweit vernetzt sind? Microsoft updatet sein 15 Jahre altes Windows Embedded (Windows CE) auf IoT-Standards und es stellt sich die Frage, wie proprietäre OSs in einer komplett vernetzten Welt den Menschen kontrollieren können statt umgekehrt. Die allgemeine Sicherheitsfrage stellt sich aber auch bei opensource OSs. Welchen OSs kann man bei sensiblen Daten vertrauen? Welchen OSs kann man ruhigen Gewissens die Kontrolle über vernetzte medizinische Systeme anvertrauen? Diese IoT-Systeme werden schnell nicht nur an einem Menschen, sondern auch in einem Menschen Anwendung finden. Der Herzschrittmacher steht stellvertretend für diese These. Die häufigere Implementation von z.B. MILS scheint spätestens dann angebracht.

ANHANG A
PROGRAMMIERUNG FÜR IOT-OSS

TinyOS:

```
#include <stdio.h>
#include <stdlib.h>

module HelloworldM{
        provides{
                interface Hello;
        }
}
implementation{
        command void Hello.sayhello() {
                printf("Hello world!");
        }
}
```

```
interface Hello{
        command void sayhello();
}
```

Contiki:

```
#include "contiki.h"
#include <stdio.h>

PROCESS(hello_world_process,
        "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);

PROCESS_THREAD(hello_world_process,
                ev, data){
        PROCESS_BEGIN();
        printf("Hello world!\n");
        PROCESS_END();
}
```

RIOT:

```
#include <stdio.h>

int main(void){
        printf("Hello World!\n");
        return 0;
}
```

ANHANG B
TABELLARISCHER VERGLEICH

LITERATUR

[1] G. Gross, "Cisco Predicts 15 Billion Network Devices in 2015," www.pcworld.com/article/229170/article.html, Juni 2011, abgerufen am 02.01.2015.

[2] J. Dorrier, "Is Cisco's Forecast of 50 Billion Internet-Connected Things by 2020 Too Conservative?" http://singularityhub.com/2013/07/30/is-ciscos-forecast-of-50-billion-internet-connected-things-by-2020-too-conservative/, Juli 2013, abgerufen am 02.01.2015.

[3] S. Heath, *Embedded Systems Design, Second Edition*, 2nd ed. Newnes, Dezember 2002, pp. 11–12.

[4] J. Paqet, "New Operating Systems for the Internet of Things," www.californiaconsultants.org/download.cfm/attachment/CNSV-1405-Jordan.pdf, Mai 2014, abgerufen am 23.11.2014.

[5] H. Lehpamer, *RFID Design Principles*, 2nd ed. Artech House Publishers, Februar 2012, abgerufen am 02.01.2015.

[6] M. O. Farooq, T. Kunz, "Operating Systems for Wireless Sensor Networks: A Survey," *Selected Papers from FGIT 2010*, 2011.

[7] J. M. Tjensvolf, "Comparison of the IEEE 802.11, 802.15.1, 802.15.4 and 802.15.6 wireless standards," http://janmagnet.files.wordpress.com/2008/07/comparison-ieee-802-standards.pdf, September 2007, abgerufen am 02.01.2015.

[8] J. A. Gutierrez, "IEEE Std. 802.15.4 Enabling Pervasive Wireless Sensor Networks," www.cs.berkeley.edu/~prabal/teaching/cs294-11-f05/slides/day21.pdf, 2005, abgerufen am 23.11.2014.

[9] G. Montenegro, N. Kushalnagar, J. Hui, D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," ttp://www.ietf.org/rfc/rfc4944.txt, September 2007, abgerufen am 23.11.2014.

[10] A. Ludovici, A. Calveras und J. Casademont, "Forwarding Techniques for IP Fragmented Packets in a Real 6LoWPAN Network," http://www.mdpi.com/1424-8220/11/1/992, 2011, abgerufen am 23.11.2014.

[11] T. Winter, P. Thubert, A. Brandt, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, R. Alexander, J. Hui, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," http://www.ietf.org/rfc/rfc6550.txt, März 2012, abgerufen am 23.11.2014.

[12] Micriμm Embedded Software, "Part 3: Internet Usage and Protocols," http://micrium.com/iot/internet-protocols/, 2013, abgerufen am 02.01.2015.

[13] R. Trapickin, "Constrained Application Protocol (CoAP): Einführung und Überblick," www.net.in.tum.de/fileadmin/TUM/NET/NET-2013-08-1/NET-2013-08-1_16.pdf, 2013, abgerufen am 23.11.2014.

[14] B. Roch, "Monolithic kernel vs. Microkernel," http://davepowell.org/media/teaching/operating-systems/handouts/COMP354-Lec02-KernelComparisons.pdf, abgerufen am 02.01.2015.

[15] G. Heiser, "Microkernel, nanokernel - what's the difference?" www.ok-labs.com/blog/entry/microkernel-nanokernel-whats-the-difference/, Februar 2008, abgerufen am 23.11.2014.

[16] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," www.cs.utexas.edu/~dahlin/Classes/439/ref/exokernel.pdf, 1995, abgerufen am 23.11.2014.

[17] J. T. Benra, W. A. Halang, *Softwareentwicklung für Echtzeitsysteme*, 1st ed. Springer Berlin Heidelberg, August 2009, pp. 99–100.

[18] N. Lethaby, "Why Use a Real-Time Operating System in MCU Applications," www.ti.com/lit/wp/spry238/spry238.pdf, abgerufen am 23.11.2014.

[19] B. Lamie, "Why use an RTOS?" www.iar.com/Global/Resources/Viewpoints/Why_use_an_RTOS.pdf, abgerufen am 30.11.2014.

[20] M. Esponda, "Scheduling in Echtzeitbetriebssystemen," www.inf.fu-berlin.de/lehre/WS11/OS/slides/OS_V9_Scheduling_Teil_3.pdf, 2011, abgerufen am 23.11.2014.

[21] A. N. Sloss, "Interrupt Handling," http://web.calstatela.edu/faculty/cliu/EE446/ARM/ARM%20Interrupt.pdf, April 2011, abgerufen am 07.01.2015.

[22] T. B. Skaali, "Scheduling of Real- Time processes, strategies and analysis," www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/forelesninger-rt/2011-7_Scheduling_of_Real-Time_processes.pdf, 2011, abgerufen am 23.11.2014.

[23] J. R. Haritsa, M. J. Carey, M. Livny, "Value-Based Scheduling in Real-Time Database Systems," www.vldb.org/journal/VLDBJ2/P117.pdf, Mai 1991, abgerufen am 23.11.2014.

Tabelle I
TABELLARISCHER VERGLEICH VON IOT-OSS I

OS	Echtzeitunterstützung	Architektur	Scheduler	Speicherallozierung	Speicherschutz
TinyOS	keine	monolithisch	FIFO (, EDF)	statisch	Deputy
Contiki	partiell	modular	direkte Ausführung und Einreihung in Warteschlange bei asynchronen Events	dynamisch	keiner
RIOT	hard	Mikro-Kernel	tickless: reduziert Overhead durch Minimierung von Thread-Switching	statisch, dynamisch	keiner
Nano-RK	hard	monolithisch	RMS, RHS	statisch	keiner

Tabelle II
TABELLARISCHER VERGLEICH VON IOT-OSS II

OS	Programmiermodell	min. RAM	min. ROM	Kommunikationssicherheit	Programmiersprache
TinyOS	Event-Driven (, TOSThreads)	400B	4kB	TinySec	nesC
Contiki	Event-Driven, Protothreads	2kB	30kB	ContikiSec	C
RIOT	Multithreading	1,5kB	5kB	(ParSec)	C, C++
Nano-RK	Multithreading	2kB	16kB	keine	C

Tabelle III
PROTOKOLLUNTERSTÜTZUNG DER OSS

OS	IEEE 802.15.4	6LoWPAN	RPL	CoAP	zusätzliche Protokolle
TinyOS	TKN15.4	(6LoWPANCLi), BLIP	TinyRPL	ja	IPv6, Tymo, ICMP, UDP (, TCP, ZigBee)
Contiki	nein	SICSlowPAN	ContikiRPL	ja	IPv4, IPv6, ICMP, UDP, TCP, HTTP, uIP(v6), RIME
RIOT	ja	ja	ja	ja	IPv4, IPv6, ICMP, UDP, TCP, ZigBee
Nano-RK	RT-Link	ja	ja	nein	IPv6, ICMP, UDP, ZigBee

[24] C. Mathas, "Real-Time Operating Systems (RTOS): What's Really at the Core of the Internet of Things?" www.em.avnet.com/en-us/design/technical-articles/Pages/Articles/Real-Time-Operating-Systems-RTOS-Whats-Really-at-the-Core-of-the-Internet-of-Things.aspx, 2014, abgerufen am 23.11.2014.

[25] R. W. Beckwith, W. M. Vanfleet, L. MacLaren, "High Assurance Security/Safety for Deeply Embedded, Real-time Systems," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.4412&rep=rep1&type=pdf, 2004, abgerufen am 02.01.2015.

[26] E. Baccelli, O. Hahm, M. Güneş, M. Wählisch, T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," www.riot-os.org/docs/riot-infocom2013-abstract.pdf, 2013, abgerufen am 23.11.2014.

[27] EE Times-Webseite, "Getting a lock on RTOS security," www.eetimes.com/document.asp?doc_id=1304191, 2007, abgerufen am 23.11.2014.

[28] D. Gascón, "Security in 802.15.4 and ZigBee networks," http://sensor-networks.org/index.php?page=0903503549, Februar 2009, abgerufen am 02.01.2015.

[29] C. Walls, "Dynamic Memory Allocation and Fragmentation in C and C++," www.design-reuse.com/articles/25090/dynamic-memory-allocation-fragmentation-c.html, abgerufen am 23.11.2014.

[30] Atmel-Webseite, "Atmel ATmega32(L) datasheet," www.atmel.com/images/doc2503.pdf, Februar 2011, abgerufen am 23.11.2014.

[31] Microcontroller.net, "Sleep Mode," www.mikrocontroller.net/articles/Sleep_Mode, abgerufen am 23.11.2014.

[32] S. Ethier, "Implementing Power Management on the Biscayne SH7760 Reference Platform Using the QNX® Neutrino® RTOS," www.qnx.com/developers/articles/article_296_2.html, abgerufen am 23.11.2014.

[33] R. L. Larsen, "Advanced Sleep-Mode Techniques for Enhanced Battery Life in Real-Time Environments," www.digikey.com/en/articles/techzone/2011/dec/advanced-sleep-mode-techniques-for-enhanced-battery-life-in-real-time-environments, Juli 2007, abgerufen am 23.11.2014.

[34] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," http://compilers.cs.ucla.edu/emsoft05/BhattiCarlsonDaiDengRoseShethShuckerGruenwaldTorgersonHan05.pdf, August 2005, abgerufen am 02.01.2015.

[35] Microcontroller.net, "Ultra Low Power," www.mikrocontroller.net/articles/Ultra_low_power, abgerufen am 23.11.2014.

[36] Micriμm Embedded Software, "Part 5: The Internet of Things and the RTOS," www.micrium.com/iot/iot-rtos/, 2013, abgerufen am 02.01.2015.

[37] J.-H. Hauer, "TKN15.4: An IEEE 802.15.4 MAC Implementation for TinyOS 2," www.tkn.tu-berlin.de/fileadmin/fg112/Papers/TKN154.pdf, 2009, abgerufen am 30.11.2014.

[38] D. Willmann, "Contiki - A Memory-Efficient Operating System for Embedded Smart Objects," https://totalueberwachung.de/blog/files/seminar-

contiki.pdf, Januar 2009, abgerufen am 29.11.2014.
[39] O. Hahm, “The friendly Operating System for the Internet of Things,” www.riot-os.org/files/RIOT.pptx, 2013, abgerufen am 02.01.2015.
[40] Riot Webpräsenz, “RIOT - The friendly Operating System for the Internet of Things,” www.riot-os.org, Januar 2013, abgerufen am 29.11.2014.
[41] E. Baccelli, O. Hahm, M. Wählisch, M. Güneş, T. Schmidt, “RIOT: One OS to Rule Them All in the IoT,” https://hal.inria.fr/hal-00768685/PDF/RR-8176.pdf, Januar 2013, abgerufen am 29.11.2014.
[42] N. Dziengel, N. Schmittberger, J. Schiller, M. Güneş, “Secure Communications for Event-Driven Wireless Sensor Networks,” http://www.des-testbed.net/system/files/dziengelsna2011.pdf, abgerufen am 07.01.2015.
[43] Nano-RK Homepage, “6LoWPAN,” www.nanork.org/projects/nanork/wiki/6LoWPAN, Mai 2010, abgerufen am 30.11.2014.
[44] Contiki-Dokumentation, “Real-time task scheduling,” http://contiki.sourceforge.net/docs/2.6/a01673.html, abgerufen am 04.01.2015.
[45] TinyOS-Dokumentation, “TOSThreads Tutorial,” http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSThreads_Tutorial, November 2009, abgerufen am 04.01.2015.
[46] RIOT GitHub-Repository, “Secure Micro-Mesh Routing (MMR) Protocol,” https://github.com/RIOT-OS/RIOT/pull/233, abgerufen am 07.01.2015.