FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Eingebettete Betriebssysteme in der Automation

Seminararbeit

im Rahmen des Studiums

Technische Informatik

eingereicht von

Peter Hausberger
Matrikelnummer 0925856

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao. Univ. Prof. Dr. Wolfgang Kastner
Mitwirkung: Univ.-Ass. Dipl. Ing. Lukas Krammer

Wien, 21.03.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit, einschließlich Tabellen, Karten und Abbildungen, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kurzfassung

Diese Arbeit gibt einen Überblick über den Einsatz von Eingebetteten Betriebssystemen in der Automation. Während bisherige Automationssysteme meist einem streng zentralen Ansatz folgen, ist heute sowohl in der industriellen als auch in der Gebäudeautomatisierung ein klarer Trend hin zu dezentralen Strukturen erkennbar. Dadurch verlagert sich aber auch ein großer Teil der Komplexität in die unteren Schichten eines Automationssystems, was zusätzliche Anforderungen an die dort eingesetzten Systeme stellt. Neben entsprechender Hardware ist die Verwendung eines geeigneten Betriebssystems von großer Bedeutung. Die direkte Interaktion mit der Umgebung von Eingebetteten Systemen sowie Einschränkungen hinsichtlich Rechenleistung, Speichergröße und Energieverbrauch machen Anpassungen dieser Betriebssysteme in Bereichen wie Architektur, Programmiermodell, Scheduling, Kommunikation oder Echtzeitfähigkeit notwendig. Für diese Anwendungsgebiete existieren eine Reihe von Betriebssystemen. Bedeutende Vertreter werden in dieser Arbeit vorgestellt und hinsichtlich ihrer Stärken und Schwächen untersucht. Als bekannte Betriebssysteme seien TinyOS und Embedded Linux genannt. TinyOS eignet sich besonders für einfache Anwendungen in der Feldebene auf Geräten mit beschränkten Ressourcen wie beispielsweise intelligenten Sensoren oder Aktoren. Embedded Linux ist hingegen für einen Einsatz in komplexeren Anwendungen in der Automationsebene konzipiert.

Abstract

This thesis presents an overview of Embedded Operating Systems in Automation. While automation systems so far usually follow a strictly centralized approach, today both in industrial and building automation a clear trend towards distributed topologies can be seen. However, this also shifts a lot of the complexity in the lower layers of an automation system, which places additional requirements on the systems used. Beside appropriate hardware the use of an proper operating system is very important. The direct interaction with the environment of Embedded Systems and limitations of processing power, memory size and energy consumption make adjustments to these operating systems in areas such as architecture, programming model, scheduling, communication or real time capability necessary. For these application areas there exist a number of operating systems. Important representatives are presented in this thesis and are analysed concerning their strengths and weaknesses. As known operating systems TinyOS and Embedded Linux are mentioned. TinyOS is particularly suitable for simple applications at the field level on devices with limited resources such as intelligent sensors or actuators. In contrast Embedded Linux is suitable for the use in complex applications at the automation level.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Immer komplexer werdende Anwendungen im Bereich der Automation stellen im speziellen auch erhöhte Anforderungen an die dabei verwendete Hardware und Software. Diese Arbeit beschäftigt sich mit dem Einsatz von Eingebetteten Betriebssystemen (engl. Embedded Operating Systems, EOS), die dabei helfen, mit diesen Anforderungen besser zurecht zu kommen. Die Verwendung eines EOS bietet dabei viele Vorteile, wie beispielsweise Flexibilität und eine einfachere Entwicklung und Wartbarkeit der Anwendung. Dies wird durch von EOSs bereitgestellten Mechanismen wie Hardwareabstraktion oder Multitasking, eine effizientere Nutzung von Ressourcen durch bewährte Methoden oder eine einfachere Portierbarkeit auf andere Plattformen ermöglicht. Dem gegenüber stehen allerdings auch einige Nachteile, allen voran ein erhöhter Ressourcenverbrauch, der aber durch die Wahl eines geeigneten EOSs minimiert werden kann.

In den folgenden Abschnitten werden die grundlegenden Begriffe erläutert. Es wird beschrieben was ein Eingebettetes System (engl. Embedded System, ES) ist, was man grundsätzlich unter einem Betriebssystem (engl. Operating System, OS) versteht und welche Anwendung diese beiden Begriffe in der Automation finden.

1.1 Eingebettete Systeme

Der Begriff des Embedded System (ES)s bezieht sich auf die Verwendung von Hardware und Software als eine Einheit, im Gegensatz zu einem Universalrechensystem, wie beispielsweise einem Personal Computer (PC). Eine gute allgemeine Definition ist:

> ***Embedded system.*** *A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In many cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car.* [6]

ESs finden sich überall in unserem Leben, von Smartphones bis hin zu medizinischen Geräten, in Navigationsgeräten, Bankomaten, Druckern oder in der Elektronik von Fahrzeugen, um nur einige Anwendungsfälle zu nennen. Betrachtet man ein Gerät mit integriertem Mikrocontroller, handelt es sich meist um ein ES. Dadurch gibt es auch weit mehr ESs als Universalrechnersysteme. ESs umfassen also einen weiten Bereich von Anwendungen und besitzen daher auch sehr unterschiedliche Anforderungen und Beschränkungen. Einige der wichtigsten davon sind im Folgenden angeführt:

- Systemgröße: kleine bis große Systeme im Sinne von Leistung, das bedeutet auch unterschiedliche Anforderungen an Kosten, Optimierung und Wiederverwendung

- Qualität: lockere bis sehr strenge Anforderungen an die Qualität des Systems in Bezug auf beispielsweise Sicherheit (engl. Safety), Zuverlässigkeit, Zeitverhalten bzw. Echtzeitfähigkeit, Flexibilität oder Gesetzgebung
- Lebensdauer: kurze bis lange Lebensdauern
- Umgebungsbedingungen: unterschiedliche Umgebungsbedingungen, wie z.B. Strahlung, Vibrationen oder Feuchtigkeit
- Anwendungseigenschaften: unterschiedliche Anwendungseigenschaften, wie beispielsweise statische bis dynamische Verteilung der Rechenlast oder langsame bis schnelle Verarbeitungsgeschwindigkeiten
- Berechnungsmodell: unterschiedliche Berechnungsmodelle bzw. Systemdynamik, von diskreten ereignisgesteuerten Systemen bis hin zu kontinuierlichen zeitgesteuerten Systemen

Auch in Hinblick auf die Systemkomponenten unterscheiden sich ESs von Universalrechnersystemen. Zusätzlich zu Prozessor und Speicher befinden sich in einem ESs weitere Komponenten niedriger Komplexität, die in einem typischen PC kaum vorkommen:

- Es gibt unterschiedliche Schnittstellen, die es dem System ermöglichen, die externe Umgebung zu messen und zu steuern.
- Das Human Machine Interface reicht von einfachen Varianten, z.B. einfache Anzeige über Leuchtdioden, bis hin zu sehr komplexen.
- Die Diagnose-Schnittstelle kann dazu verwendet werden, das gesamte gesteuerte System zu prüfen, und nicht nur den Computer an sich.
- Spezielle Hardware, wie beispielsweise Field Programmable Gate Arrays (FPGA), Application Specific Integrated Circuits (ASIC) oder auch analoge Bausteine, wie Analog/Digital (A/D) oder Digital/Analog (D/A) Konverter, können zur Steigerung der Effizienz und Sicherheit (engl. Safety) verwendet werden.
- Die Software hat meist eine fixe Funktion und ist spezifisch für die Anwendung.

Oft sind ESs eng an ihre Umgebung gebunden. Anforderungen an das Zeitverhalten, wie beispielsweise die erforderliche Geschwindigkeit einer Bewegung oder die erforderliche Genauigkeit einer Messung, stellen Bedingungen an die Echtzeitfähigkeit des Systems, das mit seiner Umgebung interagiert. Wenn mehrere Aktionen zeitgleich ausgeführt werden müssen, stellt das eine zusätzliche Anforderung an das Echtzeitverhalten. [2] [4]

1.2 Betriebssysteme

Ein OS ist ein Computerprogramm, das die Basis für Anwendungsprogramme darstellt und deren Abarbeitung unterstützt, kontrolliert und auch steuert. Es agiert dabei als eine Schnittstelle zwischen Applikation und Hardware. Grundsätzlich verfolgt es dabei drei Ziele:

Einfachheit Ein OS soll die Benutzung eines Computers einfacher machen. Während bei traditionellen Desktop-OSs hier vor allem die Verwendung durch den Anwender gemeint ist, steht bei EOSs eine einfache Benutzung durch die jeweilige Applikation im Vordergrund.

Effizienz Ein OS soll es ermöglichen, die Systemressourcen in einer effizienten Art und Weise zu benutzen.

Fähigkeit zur Weiterentwicklung Ein OS soll so konstruiert sein, dass die Entwicklung, der Test und das Einfügen neuer Systemfunktionen ohne Unterbrechung der Services geschehen kann. Dies kann beispielsweise durch einen modularen Aufbau des OSs sichergestellt werden. Dabei sind Aspekte der Wartbarkeit, die Kompatibilität zu anderen OSs oder OS-Versionen sowie die Portierbarkeit von Software auf unterschiedliche Plattformen von besonderer Bedeutung.

Aufgrund der im vorherigen Abschnitt beschriebenen, besonderen Eigenschaften von ESs, sind auch die Anforderungen an die dort eingesetzten OSs besonders und unterscheiden sich deutlich von jenen klassischer OSs. Aus diesem Grund gibt es auch eine Vielzahl an EOSs, die speziell für diese Anwendungsfälle optimiert wurden und im Kapitel 2 genauer behandelt werden. [2] [7]

1.3 Einsatz in der Automation

Eine der Hauptaufgaben der Automation ist die Verarbeitung von Information. Abhängig vom Teilbereich in einem Automatisierungssystem hat diese Information aber unterschiedliche Bedeutung, unterschiedlichen Inhalt und unterschiedliche Eigenschaften. So unterschiedlich wie die verschiedenen Arten von Information sind, so unterschiedlich sind auch die Mittel, um diese zu strukturieren, zu verarbeiten und zu verteilen.

Um mit dieser Komplexität zurechtzukommen, bedient man sich eines hierarchischen Modells, das als Automationspyramide bezeichnet wird. Dabei handelt es sich um ein vereinfachtes Drei-Schichten-Modell, wie in Abbildung 1.1 gezeigt, bestehend aus einer Managementebene, einer Automationsebene und einer Feldebene.

Auf der untersten Ebene, der Feldebene, hat Information nur eine kurzfristige Bedeutung und wird lediglich zur Steuerung und Regelung von einzelnen Prozessen oder Prozessschritten durch Sensoren und Aktoren verwendet.

Die mittlere Schicht, die Automationsebene, bildet eine Brücke zwischen der Feldebene und der Managementebene. Hier finden sich hauptsächlich Speicherprogrammierbare Steuerungen (SPS) und Industrie PCs (IPC). Sie steuern Sensoren und Aktoren der Feldebene. Weiters finden sich in dieser Ebene noch Benutzerschnittstellen, welche Ein-/Ausgabeen (E/A) von Benutzern an die entsprechenden Geräte weiterleiten. Die einzelnen Geräte kommunizieren untereinander über sogenannte Feldbusse. Dadurch kann ein komplexer Gesamtprozess beobachtet und gesteuert werden. Die Informationen, die in dieser Ebene gesammelt werden, werden an die übergeordnete Schicht weitergeleitet.

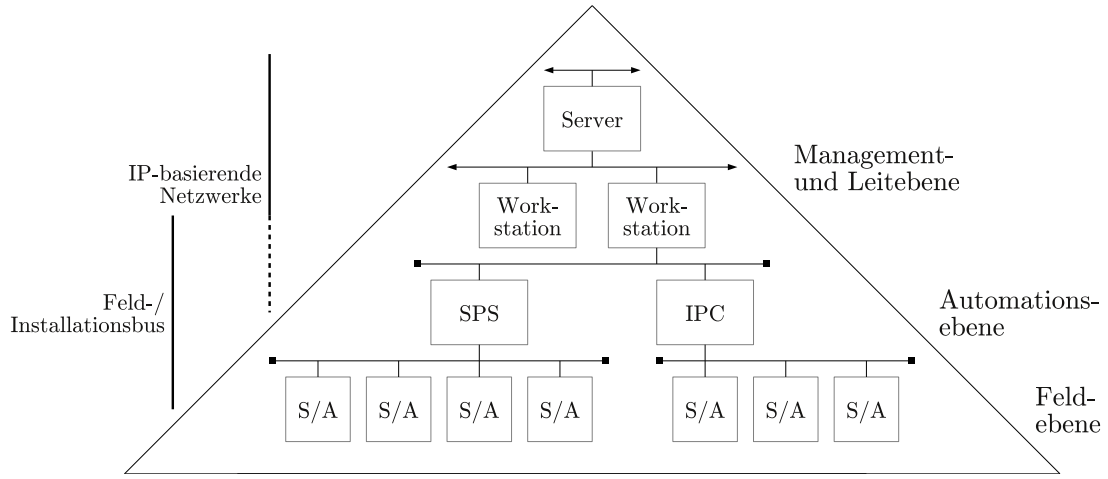

Abbildung 1.1: Automationspyramide [1]

Information auf der höchsten Ebene, auf der Managementebene, wird zur langfristigen Planung und Datenerfassung genutzt. Auf dieser Ebene ist der Grad der Abstraktion sehr hoch. Systeme, wie beispielsweise Manufacturing Execution Systems (MES) und Supervisory Control and Data Acquisition (SCADA) Systems sammeln Information über den technischen Prozess und steuern ihn. Enterprise Resource Planning (ERP) Systems versuchen die in einem Unternehmen vorhandenen Ressourcen möglichst effizient einzusetzen und die Steuerung des Prozesses durch die darunterliegenden Systeme zu optimieren.

Jede Ebene leitet also ihre Daten an die darüber liegende Ebene weiter, und steuert/regelt die darunterliegende Ebene. Zu beachten ist auch, dass die Grenzen zwischen den Ebenen nicht scharf sind, sondern überlappen. Das Modell zeigt aber auch, dass die beiden großen Automationsdomänen, die industrielle und die Gebäudeautomatisierung, unterschiedlich wie sie aus Anwendungssicht sind, viele grundlegende Eigenschaften gemeinsam haben.

Aus diesen Betrachtungen ist erkennbar, dass in einem Automationssystem unterschiedliche Technologien, von der Hardware über die verwendete Netzwerktechnologie bis hin zur eingesetzten Software, verwendet werden und diese auf den unterschiedlichen Ebenen andere Anforderungen haben. Diese Arbeit befasst sich nur mit einem kleinen Teil eines Automationssystems, mit dem Einsatz von EOSs. Die in der Managementebene eingesetzte Software, wie beispielsweise SAP[1] ERP, ist im Grunde eine herkömmliche Anwendungssoftware, die auf herkömmlichen Desktop oder Server OSs basiert. Somit spielt die Managementebene für die weiteren Betrachtungen keine Rolle. EOSs finden hauptsächlich in der Automations- und Feldebene Verwendung. Die Unterschiede zu herkömmlichen OSs und die unterschiedlichen Anforderungen werden in den folgenden Kapiteln im Detail diskutiert. [1]

[1] `http://www.sap.com`

2 Eingebettete Betriebssysteme

Wie bereits in Abschnitt 1.1 beschrieben, stellt die Umgebung von ESs anspruchsvolle Anforderungen an das OS und macht neue Entwurfsstrategien notwendig, welche sich von jenen der klassischen OSs klar unterscheiden. Die Anzahl der OSs, die in ESs verwendet werden, ist weit größer als die Zahl der Systeme, die in Desktop-Computern eingesetzt werden. Aus diesem Blickwinkel betrachtet, sind EOSs von besonderer Wichtigkeit.

Aber ab wann macht die Verwendung eines OS überhaupt Sinn? Eine einfache Anwendung mit einfacher Funktionalität kann ohne weiteres durch ein ES mit einem speziellen Programm oder einem Satz von Programmen ohne zusätzliche Software, wie einem OS, gesteuert werden. Dadurch verringern sich zwar die Systemanforderungen an Rechenleistung und Speicher, aber die Entwicklung und Wartung der Anwendung wird erschwert. Deshalb verwenden komplexere Systeme typischerweise ein OS, da sie auf gewisse Mechanismen wie Hardwareabstraktion oder Multitasking angewiesen sind. Obwohl es grundsätzlich möglich wäre ein klassisches OS, wie beispielsweise Windows[1], für ESs zu verwenden, machen es Beschränkungen in Bereichen wie Speicherplatz und Leistungsverbrauch sowie Echtzeitanforderungen notwendig, speziell entworfene OSs für den Einsatz in ESs zu verwenden. Im Folgenden werden einige besondere Eigenschaften und Entwurfsanforderungen beschrieben. [2]

2.1 Anforderungen

Echtzeitfähigkeit In vielen ESs hängt die Korrektheit einer Berechnung nicht nur vom Wert des Ergebnisses ab, sondern auch vom Zeitpunkt, an dem es erzeugt wird. Meist werden Echtzeitanforderungen von externer Ein-/Ausgabe (E/A) oder von Stabilitätsanforderungen von Regelkreisen vorgegeben. [8]

Reaktionsfähigkeit Software in ESs wird meist als Reaktion auf externe Ereignisse ausgeführt, die nicht periodisch oder zu bekannten Zeitpunkten auftreten, sondern spontan. Dabei muss garantiert werden, dass eine maximale Reaktionszeit nicht überschritten wird. Dies gilt vor allem in Worst Case Bedingungen. Unvorhersagbare Reaktionszeiten können beispielsweise in Regelkreisen die Qualität der Regelung verschlechtern oder Instabilitäten verursachen.

Konfigurierbarkeit Aufgrund der großen Vielfalt von ESs gibt es auch große Unterschiede in den Anforderungen an die Funktionalität von EOSs, sowohl in qualitativer als auch quantitativer Hinsicht. Aus diesem Grund muss ein EOS, das für die Verwendung in unterschiedlichen Systemen bestimmt ist, die Möglichkeit bieten, flexibel konfigurierbar zu sein, so dass nur die Funktionalität, die für eine spezielle Anwendung oder für eine spezielle Hardware gebraucht wird, zur Verfügung steht. Z.B. können

[1] `http://windows.microsoft.com`

Funktionen zum dynamischen Nachladen und Verlinken von Modulen dazu verwendet werden, um nur die wirklich benötigten Software Module in Laufzeit auszuwählen. Eine andere Möglichkeit ist die bedingte Kompilierung (engl. Conditional Compilation). Allerdings stellt die Verifikation ein Problem für den Entwurf mit einer großen Anzahl von abgeleiteten OSs dar.

Flexibilität in der Verwendung von E/A Geräten Es gibt eine Vielzahl an E/A Geräten, die aber nicht von jedem OS unterstützt werden müssen. Beispielsweise können langsame Geräte wie Festplatten, anstatt wie üblich über Kernel-Treiber, über spezielle Tasks integriert werden.

Vereinfachte Schutzmechanismen (engl. Streamlined Protection Mechanisms) ESs werden typischerweise für eine begrenzte klar definierte Funktion entworfen. Nicht getestete Software wird nicht zum System hinzugefügt. Erst wenn die Sofware konfiguriert und getestet wurde, kann angenommen werden, dass sie zuverlässig ist. Dadurch ist es ausreichend, dass ESs nur begrenzte Schutzmechanismen aufweisen, abgesehen von speziellen Sicherheitsmaßnahmen. Z.B. ist es nicht immer notwendig, dass E/A Operationen privilegierte Operationen sind, welche vom OS abgefangen werden. Oft ist es ausreichend oder sogar erwünscht, wenn Benutzertasks direkt E/A Operationen durchführen können. In ähnlicher Weise können auch Speicherschutzmechanismen vereinfacht werden. Diese Varianten können dem Einsatz von OS-Aufrufen vorgezogen werden, welche einen großen Overhead für das Speichern und Wiederherstellen des Task Kontexts verursachen würde. Ein weiteres Beispiel sind Interrupts. Klassische OSs bieten Benutzerprozessen typischerweise nicht die Möglichkeit, Hardware Interrupts direkt zu verwenden. In vielen Fällen, beispielsweise zur Steigerung der Effizienz, ist es jedoch sinnvoll, einen Task direkt durch einen Interrupt zu starten, anstatt durch eine vom OS bereitgestellte Interrupt Service Routine.

Softwarequalitätssicherung Besonders für sicherheitsrelevante Anwendungen ist es notwendig, zu gewährleisten, dass die eingesetzte Software gewisse Qualitätsmerkmale erfüllt. Dafür gibt es unterschiedliche Methoden. Eine Variante ist die Programmverifikation, die sicherstellt, dass ein Programm zu einer Spezifikation konform ist. Da ein Beweis zur formalen Verifikation meist sehr umfangreich und nicht in jedem Fall möglich ist, werden typischerweise nur ausgewählte Softwarekomponenten einer Verifikation unterzogen. Eine weitere Möglichkeit ist das Entwicklen eines OS gemäß einer Norm. Von Bedeutung ist hierbei die IEC 61508, welche die funktionale Sicherheit programmierbarer elektronischer Systeme beschreibt. Die Norm begleitet den Entwicklungsprozess über den gesamten Lebenszyklus des Systems und erstreckt sich über Entwurf, Entwicklung, Inbetriebnahme, Instandhaltung bis hin zur Außerbetriebnahme. [2]

2.2 Eigenschaften

2.2.1 Architektur

Die Architektur eines OSs hat einen entscheidenden Einfluss auf dessen Größe und auf die Art und Weise, wie OSs ihre Services den Applikationen zur Verfügung stellen. Wichtig für

EOSs sind eine geringe Kernelgröße und ein daraus resultierendes kleines Speicherabbild. Die Architektur muss des Weiteren eine einfache Erweiterbarkeit des Kernels und der restlichen OS-Komponenten ermöglichen. Die Architektur muss flexibel sein, d.h. nur Services, die von der Applikation auch wirklich benötigt werden, werden in das System geladen. Grundsätzlich kann zwischen folgenden Architekturtypen unterschieden werden, wobei konkrete OSs oft auch eine Mischung davon aufweisen:

Monolithische Architektur Diese Architektur hat keine wirkliche Struktur. Die vom OS zur Verfügung gestellten Services werden einzeln implementiert und jedes Service stellt eine Schnittstelle für andere Services zur Verfügung. Dadurch können alle Services in einem einzelnen Systemabbild zusammengefasst werden, was zu einem geringen Speicherabbild des OSs führt. Weiters ist die Interaktion zwischen den Modulen sehr einfach und schnell. Nachteilig wirkt sich diese Architektur jedoch auf Zuverlässigkeit und Wartbarkeit aus, da das System komplex und dadurch schwer beherrschbar ist.

Mikrokernel Architektur Eine Alternative ist die Mikrokernel Architektur, in der nur minimale Kernfunktionalität vom Kernel übernommen wird. Dadurch ist auch hier der Speicherverbrauch sehr gering. Zusätzliche Funktionalität wird über sogenannte User-Level-Server, wie beispielsweise Datei-Server oder Speicher-Server angeboten. Wenn einer dieser Dienste abstürzt, ist der Kernel davon nicht betroffen. Dadurch bietet die Mikrokernel Architektur eine höhere Zuverlässigkeit und bessere Eigenschaften in Hinsicht Erweiterbarkeit und Anpassungsfähigkeit. Der Nachteil dieser Architektur ist jedoch, dass OS-Aufrufe aufwändiger sind, da vom User-Mode in den Kernel-Mode gewechselt werden muss. Trotzdem wird diese Architektur für viele ESs verwendet, da die Größe des Kernels gering ist und die Anzahl der OS-Aufrufe in typischen Eingebetteten Anwendungen geringer ist als jene in traditionellen.

Schichtenarchitektur (engl. Layered Architecture) Hier werden die Services in Form von aufeinander aufbauenden Ebenen angeboten. Zu den Vorteilen zählen Zuverlässigkeit und Wartbarkeit, da das System einfach zu verstehen ist.

Virtuelle Maschine In dieser Architektur laufen Applikationen im Kontext einer Virtuellen Maschine ab. Anwendungen sind dadurch einfach auf unterschiedliche Plattformen portierbar. Allerdings weist diese Architektur eine schlechte Performance auf. [2] [3]

2.2.2 Programmiermodell

Das vom OS bereitgestellte Programmiermodell hat entscheidende Auswirkungen auf die Anwendungsentwicklung. In EOSs gibt es grundsätzlich zwei unterschiedliche Programmiermodelle:

Nebenläufige Programmierung (engl. Multithreaded Programming) Viele Entwickler sind mit diesem Programmiermodell am besten vertraut. Es ist jedoch sehr ressourcenintensiv und daher für Geräte mit beschränkten Ressourcen, wie z.B. Sensorknoten, nicht geeignet. Um dieses Modell auf solchen Geräten trotzdem einsetzten zu können, gibt es bereits einige Entwicklungen mit besonderem Augenmerk auf eine schlanke ressourcenschonende Implementierung.

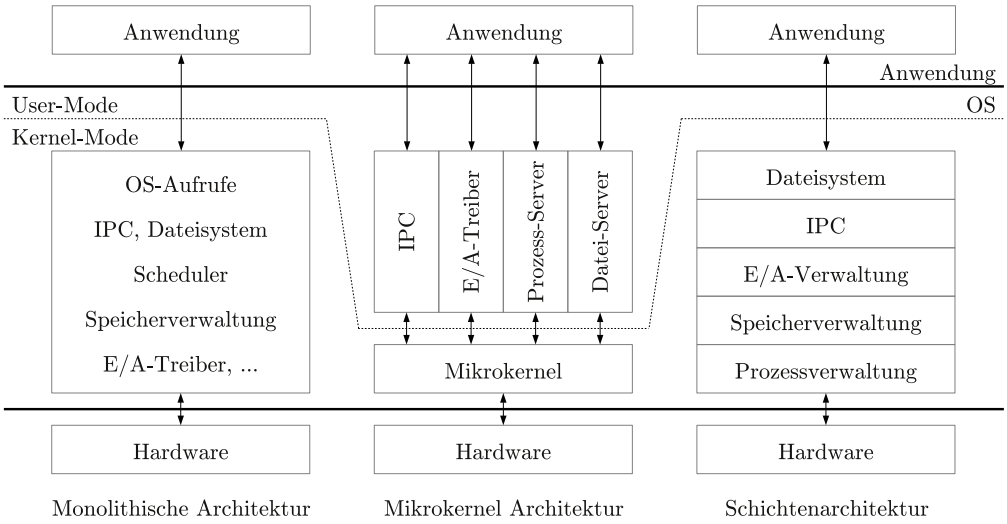

Abbildung 2.1: Betriebssysteme basierend auf verschiedenen Kernel-Architekturen [2]

Ereignisgesteuerte Programmierung Dieses Programmiermodell ist zwar für Geräte mit beschränkten Ressourcen weit besser geeignet, für die meisten traditionellen Anwendungsentwickler jedoch ungewohnt. Für einfache Anwendungen ist dieses Modell meist aber ausreichend. [3]

2.2.3 Ressourcenverwaltung

Zu den Aufgaben eines OS gehören das Allozieren und Teilen von Ressourcen. Dies ist von besonderer Bedeutung, wenn mehrere Programme nebenläufig ausgeführt werden. Wie im vorherigen Abschnitt beschrieben, unterstützen die meisten modernen EOSs Multitasking, weshalb auch Ressourcenteilung notwendig ist. Ein OS verwendet dazu zwei verschiedene Multiplexverfahren. Ressourcenteilung kann in Zeit, z.B. beim Scheduling eines Prozesses, oder in Raum, z.B. bei der Speicherung von Daten, erfolgen. Oft ist ein serialisierter Zugriff auf Ressourcen notwendig, was durch den Einsatz von Synchronisationsmechanismen bewerkstelligt wird. [2] [3]

2.2.4 Scheduling

Das Scheduling der Central Processing Unit (CPU) ist eine spezielle Form der Ressourcenverwaltung und definiert die Reihenfolge der Abarbeitung von Tasks auf der CPU. Typische Ziele eines Schedulers sind minimale Latenz, maximaler Durchsatz, maximale Ressourcenauslastung und Fairness. Die Auswahl eines geeigneten Scheduling Algorithmus hängt dabei vom jeweiligen Anwendungsfall ab. Für Echtzeitanwendungen müssen beispielsweise andere Scheduler eingesetzt werden, als für nicht echtzeitfähige Anwendungen. Da in der Automation je nach konkretem Anwendungsgebiet sowohl echtzeitfähige als auch nichtechtzeitfähige Anwendungen verwendet werden, müssen EOSs entsprechende Algorithmen zur Verfügung stellen, die die Anforderungen erfüllen. Weiters sollte der verwendete Scheduling Algorithmus speicher- und energieeffizient sein. [2] [3]

2.2.5 Speicherverwaltung und -schutz

Auch hier handelt es sich um eine spezielle Form der Ressourcenverwaltung. In traditionellen OSs regelt die Speicherverwaltung die Art und Weise, wie Speicher für verschiedene Prozesse alloziert und dealloziert wird. Die zwei grundlegenden Techniken dazu sind statische und dynamische Speicherallozierung:

Statische Speicherverwaltung ist sehr einfach und ausreichend, wenn nur ein sehr begrenzter Speicherbereich zur Verfügung steht. Gleichzeitig ist dieses Verfahren jedoch auch sehr unflexibel, da keine Speicherallozierung in Laufzeit möglich ist.

Dynamische Speicherverwaltung macht ein System flexibler, da hier Speicher in Laufzeit alloziert und dealloziert werden kann.

Unter Speicherschutz versteht man den Schutz des Speicherbereichs eines Prozesses vor anderen.

In vielen EOSs, wie beispielsweise im in Abschnitt 3.2 beschriebenen TinyOS, war anfänglich kein Speichermanagement verfügbar. Man ging davon aus, dass nur eine einzelne Applikation ausgeführt wird, weshalb kein Speicherschutz notwendig war. Aufgrund immer komplexer werdender Anforderungen unterstützen aber heute die meisten EOSs Multitasking, wodurch auch Speicherverwaltung immer wichtiger wird. [2] [3]

2.2.6 Kommunikation

Man unterscheidet hierbei zwischen Interprozesskommunikation (IPC) innerhalb des Systems und der Kommunikation mit externen Geräten. Ein Automationssystem ist eine verteilte Umgebung, in der die Geräte untereinander innerhalb eines Netzwerks kommunizieren. Über ein Application Programming Interface (API), welches das OS zur Verfügung stellt, kann eine Applikation Daten mit anderen Prozessen oder Geräten im Netzwerk austauschen. In netzwerkbasierender Kommunikation sollte das OS Unterstützung für die Transport-, Vermittlungs- und Sicherungsschicht des Open Systems Interconnection (OSI) Referenzmodells bieten. [2] [3]

2.2.7 Echtzeitunterstützung

Ein Echtzeitsystem muss auf externe Ereignisse in einer garantierten Zeit reagieren. Diese Zeit wird von der Anwendung vorgegeben, nicht vom Computersystem. Der späteste Zeitpunkt, zu dem ein Resultat zur Verfügung stehen muss, wird als Deadline bezeichnet. Wenn das Resultat nach Ablauf einer Deadline immer noch Bedeutung hat, wird sie als weiche Deadline bezeichnet. Wenn hingegen das Verpassen einer Deadline ernsthafte Konsequenzen hat, wird sie als hart bezeichnet. In diesem Sinne wird ein System, das eine harte Deadline einhalten muss, als hartes Echtzeitsystem (engl. Hard Realtime System) bezeichnet. Der Entwurf eines harten Echtzeitsystems unterscheidet sich grundlegend von dem eines weichen Echtzeitsystems (engl. Soft Realtime System). Wenn ein System eine harte echtzeitkritische Anwendung steuern oder beobachten soll, so muss dies auch vom darunterliegenden OS unterstützt werden. Dazu gehört die Implementierung von echtzeitfähigen Scheduling Algorithmen, um die Deadlines von harten Echtzeit-Tasks einzuhalten.

Weiters muss ein OS echtzeitfähige Kommunikationsprotokolle unterstützen. Um die Ende-zu-Ende Verzögerung (engl. End-to-End Delay) so gering (minimale Latenz) und konstant (minimaler Jitter) wie nur möglich zu halten, sollte die Kommunikation auf der Sicherungs-, der Vermittlungs- und der Transportschicht unterstützt werden. Des Weiteren sollte darauf aufbauend eine API angeboten werden, die es der Applikation ermöglicht, einfach und effizient anwendungsspezifische Kommunikationsprotokolle, basierend auf dem vom OS unterstützten Protokollstack, zu entwickeln. Deshalb sollte es den Applikationen möglich sein, die Parameter des Protokollstacks nach deren Ansprüchen anzupassen.

Traditionelle Kommunikationsprotokolle sind für den Einsatz in Echtzeitsystemen nicht geeignet. Beispielsweise ist der Ethernet-Standard, der auf den unteren Schichten des OSI Referenzmodells arbeitet, durch die Verwendung eines nicht deterministischen Medienzugriffsverfahren nicht echtzeitfähig. Mit Industrial Ethernet oder TTEthernet[2] gibt es jedoch Bestrebungen, diesen Standard auch für Echtzeitanwendungen nutzbar zu machen. Beschränkte Ressourcen machen den Einsatz des Transmission Control Protocols (TCP) nahezu unmöglich. Wenn Echtzeit Streams übertragen werden müssen, ist TCP durch seine Fluss- und Staukontrolle nicht geeignet. Eine Alternative dazu wäre das User Datagram Protocol (UDP), das aber keine Rückmeldung über den Status des Netzwerks bietet, was beispielsweise bei der Übertragung von Multimedia Daten notwendig ist. Deshalb sind weder TCP noch UDP ideale Protokolle für diese Anwendungen.

Es sollte hingegen ein Transportschicht Protokoll implementiert werden, das Echtzeitanwendungen unterstützt. Es sollte die Netzwerkbedingungen beobachten und durch eine Staukontrolle den Stau minimieren, damit die Ansprüche an eine Echtzeitkommunikation erfüllt werden können. Des Weiteren sollte ein Routingprotokoll zur Verfügung gestellt werden, welches Routen anhand der Anforderungen der Applikation konstruiert und ein Scheduling Algorithmus auf Medium Access Control (MAC) Ebene implementiert werden, der die Pakete, beispielsweise anhand ihrer Priorität, versendet. [3] [8]

2.3 Entwurf

Grundsätzlich gibt es zwei verschiedene Ansätze, um ein EOS zu entwickeln. Der erste ist die Verwendung eines existierenden klassischen OSs, das an die Anforderungen eines ESs angepasst wird. Der andere Ansatz ist, ein OS speziell für den Einsatz in Eingebetteten Anwendungen zu entwerfen und zu entwickeln. [2]

2.3.1 Anpassung eines klassischen Betriebssystems

Ein existierendes klassisches OS kann für ein ES durch das Hinzufügen von beispielsweise Echtzeitfähigkeit und anderer notwendiger Funktionalität angepasst werden. Dabei kann z.B. Linux[3] oder Windows verwendet werden. Sie sind typischerweise langsamer und weniger berechenbar als spezielle EOSs. Ein Vorteil ist aber, dass das vom klassischen OS abgeleitete EOS auf einem Satz von bekannten Schnittstellen basiert, was die Portabilität erleichtert. Der Nachteil des Einsatzes eines klassischen OSs ist jedoch, dass es nicht für

[2] `http://www.tttech.com/technologies/ttethernet`
[3] `http://www.kernel.org`

Echtzeit und Eingebettete Anwendungen optimiert ist. Aus diesem Grund ist es erforderlich, entsprechende Änderungen und Optimierungen vorzunehmen, um die gewünschte Effizienz zu erreichen. Grundsätzlich sind klassische OSs für den durchschnittlichen, nicht aber für den schlechtesten Fall, optimiert. Z.B. erfolgt die Zuteilung von Ressourcen meist auf Anfrage und es werden die meisten, wenn auch nicht alle, semantischen Informationen einer Applikation verworfen. [2]

2.3.2 Speziell angefertigtes Eingebettetes Betriebssystem

Es gibt eine Vielzahl an EOSs, die von Grund auf für die Verwendung in ESs entworfen worden sind, wie beispielsweise das in Abschnitt 3.2 beschriebene TinyOS. Typische Eigenschaften sind:

- Das OS besitzt eine geringe Größe.
- Ein Echtzeit Scheduling Algorithmus mit schnellem und leichtem Kontextwechsel wird unterstützt
- Auf externe Interrupts wird schnell reagiert, typische Anforderungen sind Reaktionszeiten im Bereich von Mikrosekunden.
- Die Intervalle, in denen Interrupts deaktiviert werden, sind kurz.
- Die Speicherverwaltung unterstützt Partitionen fixer oder variabler Größe und die Möglichkeit Daten und Code im Speicher zu fixieren.
- Um mit Timing Constraints umgehen zu können, garantiert der Kernel eine begrenzte Ausführungszeit für wichtige OS-Aufrufe.
- Es werden spezielle Alarme, Timeouts und Primitiven, um die Ausführung um eine gewisse Zeitdauer zu verzögern bzw. zu unterbrechen und wiederaufzunehmen, angeboten.

Viele dieser Eigenschaften charakterisieren echtzeitfähige EOSs. Für komplexere Systeme ist aber meist eine schnelle Ausführung wichtiger als eine vorhersagbare, was Änderungen im Design, vor allem im Bereich des Schedulers, notwendig macht. [2]

3 Feldebene

3.1 Überblick

Dieses Kapitel beschäftigt sich mit EOSs, die für eine Verwendung in der Feldebene geeignet sind. Wie in Abschnitt 1.3 beschrieben, stellt die Feldebene die unterste Schicht in einem Automationssystem dar und besteht aus Sensoren und Aktoren. Geräte dieser Klasse sind in Bezug auf Rechenleistung, Speichergröße und Energieversorgung stark beschränkt. Der Grund dafür liegt im niedrigen Formfaktor dieser Geräte, der z.B. eine aktive Kühlung nicht zulässt, aber auch im hohen Aufkommen dieser Geräte in einem Automationssystem und dem damit verbundenen finanziellen Druck. Typischerweise haben die verwendeten Mikrocontroller eine Taktfrequenz im Megahertz Bereich, einen Speicher im Kilobyte Bereich und eine oft nur als Batterie ausgeführte Energieversorgung im Bereich von Milliwatt.

Dementsprechend zeichnen sich EOSs in dieser Ebene durch ein kleines Speicherabbild und allgemein durch einen geringen Ressourcenverbrauch aus. Der Schwerpunkt liegt auf Kompaktheit und schneller Ausführung. Sie sind meist applikationsspezifisch, d.h. anders als bei klassischen OSs, die im Grunde die Ausführung mehrerer Applikation kontrollieren, kommt hier nur eine einzige dedizierte Anwendung zur Ausführung, und das OS fungiert im Wesentlichen nur als eine Art Basisbibliothek. Auch fehlen hier oft erweiterte Funktionen, die typischerweise in klassischen OSs vorhanden sind, wie beispielsweise eine Benutzerverwaltung oder eine vollwertige grafische Benutzeroberfläche. Einige bekannte EOSs, die diese Anforderungen erfüllen sind:

FreeRTOS[1] ist ein Open Source Echtzeitbetriebssystem (engl. Real Time OS, RTOS) für Mikrocontroller-Plattformen. Es ist einfach und klein gehalten, der Mikrokernel weist in einer typischen Konfiguration einen statischen Speicherverbrauch von ungefähr fünf Kilobyte auf und ist in der Programmiersprache C geschrieben, wodurch eine gewisse Wartbarkeit und Portierbarkeit, FreeRTOS wird aktuell für 33 Plattformen angeboten, gewährleistet ist. FreeRTOS unterstützt nebenläufige Programmierung mittels Tasks und einem prioritätsbasierenden Round Robin Scheduler, der den einzelnen Tasks für jeweils kurze Zeitschlitze die CPU zuweist. Tasks können über verschiedene Mechanismen, wie beispielsweise Semaphoren oder Software Timer, synchronisiert werden. Weiters werden Methoden zur dynamischen Speicherallozierung oder ein Internet Protocol (IP) Stack angeboten.

Für sicherheitskritische Anwendungen wird das mit FreeRTOS verwandte OS SafeRTOS angeboten. Es bietet einen ähnlichen Funktionsumfang und ähnliche APIs wie FreeRTOS an, ist aber nach der bereits beschriebenen Norm IEC61508 zertifiziert. [9]

Contiki[2] ist ein quelloffenes OS, das speziell für den Einsatz in drahtlosen Sensornetzen konzipiert wurde. Contiki besteht aus einem ereignisgesteuerten Kernel, der in der

[1] `http://www.freertos.org`
[2] `http://www.contiki-os.org`

Programmiersprache C geschrieben ist und dadurch auf zahlreiche Plattformen mit begrenzten Ressourcen portiert wurde, wie beispielsweise auf MSP430[3] oder AVR[4] Mikrocontrollern. Die Größe des Kernels liegt im Kilobyte Bereich, und der dynamische Speicherverbrauch kann typischerweise bis auf wenige Bytes reduziert werden. Nebenläufigkeit wird wahlweise durch einen schlanken kooperativen Scheduler, oder durch einen präemptiven Scheduler mit größerem Ressourcenverbrauch unterstützt. Eine Besonderheit von Contiki gegenüber anderen OSs dieser Klasse ist die Möglichkeit, Anwendungen dynamisch zu laden. Dadurch können Aktualisierungen im laufenden Betrieb durchgeführt werden. Weiters werden verschiedene Netzwerkmechanismen, z.B. ein IP Stack, ein einfaches Dateisystem oder eine graphische Benutzeroberfläche angeboten. [10]

TinyOS[5] ist eines der bekanntesten und auch ältesten EOSs für drahtlose Sensornetze. Anders als die oben beschriebenen OSs ist TinyOS und dessen API in network embedded systems C (nesC), eine Erweiterung der Programmiersprache C, geschrieben. Dies ermöglicht eine sehr abstrakte und trotzdem effiziente Art der Softwareentwicklung. Aus diesem Grund wird im folgenden Abschnitt TinyOS genauer behandelt.

3.2 TinyOS

TinyOS ist ein quelloffenes, flexibles OS, ursprünglich entwickelt für drahtlose Sensor Netzwerke. Es basiert auf einem Komponentenmodell und ist applikationsspezifisch. TinyOS unterstützt nebenläufige Programme mit sehr geringem Speicherverbrauch, das OS hat einen typischen statischen Speicherverbrauch von ca. 400 Bytes. Die TinyOS Bibliothek umfasst Komponenten für Netzwerkprotokolle, verteilte Services, Treiber für Sensoren und Werkzeuge zum Erfassen von Daten. Eine umfangreiche Dokumentation zu TinyOS kann auf der Website des Projekts gefunden werden. Im Folgenden wird TinyOS und der Anwendungsfall in drahtlosen Sensornetzen genauer behandelt. [2] [3]

3.2.1 Architektur

TinyOS weist eine monolithische Struktur auf und verwendet ein Komponentenmodell. Das Komponentenmodell wird dabei bereits auf unterster Ebene unterstützt, nämlich von der verwendeten Programmiersprache nesC, einer Erweiterung von C. Abhängig von den Anforderungen einer Applikation werden verschiedene Komponenten, darunter auch die Anwendungskomponenten, zu einem statischen Image zusammengefasst, welches am Knoten ausgeführt wird.

Eine Komponente ist dabei eine unabhängige Entität, die Berechnungen durchführt und mehrere Interfaces anbieten kann. Eine Komponente besteht aus drei verschiedenen Abstraktionen: Kommandos, Events, und Tasks. Während Kommandos und Events zur Kommunikation zwischen den Komponenten verwendet werden (Inter-Komponenten-Kommunikation), werden Tasks verwendet, um komponenteninterene Nebenläufigkeit auszudrücken. Ein

[3] `http://www.ti.com/msp430`
[4] `http://www.atmel.com/products/microcontrollers/avr`
[5] `http://www.tinyos.net`

Kommando ist eine Anfrage, um ein bestimmtes Service auszuführen, wohingegen ein Event den Abschluss eines Services signalisiert. In TinyOS gibt es keine Trennung von Kernel-Space und User-Space und es wird nur ein einzelner gemeinsam benutzter Stack verwendet, wodurch der Overhead von OS-Aufrufen minimal ist. Der Nachteil ist jedoch, dass eine fehlerhafte oder schädliche Komponente sehr einfach die Funktionalität des OS beeinträchtigen kann. [2] [3]

3.2.2 Programmiermodell

Frühere Versionen von TinyOS unterstützten kein Multitasking, die Applikationsentwicklung folgte damit komplett dem ereignisgesteuerten Modell.

Seit Version 2.1 unterstützt TinyOS auch Multithreading durch die TOS-Threading-Library. Untersuchungen haben jedoch gezeigt, dass, unter Berücksichtigung der beschränkten Ressourcen wie Rechenleistung oder Speicher eines Knoten, in einem ereignisgesteuerten OS eine bessere Nebenläufigkeit erzielt werden kann als in einem Multitasking OS [11]. Trotzdem ist mit präemptiven Threads eine intuitivere Programmierung möglich. Threads in TinyOS bieten die Vorteile eines Multithreading Programmiermodells, verbunden mit der Effizienz eines ereignisgesteuerten Kernels. TOS-Threads sind abwärtskompatibel mit existierendem TinyOS Code. Sie verwenden einen kooperativen Ansatz, d.h. TOS-Threads sind darauf angewiesen, dass die Applikation explizit den Prozessor freigibt. Damit wird die Aufgabe der Nebenläufigkeit zu einem gewissen Grad an den Entwickler selbst übergeben. Applikations-Threads können in TinyOS zwar andere Applikations-Threads unterbrechen, aber nicht Tasks und Interrupt-Handlers. Ein hochpriorer Kernel-Thread übernimmt in TinyOS die Aufgabe des Schedulers. Für die Kommunikation zwischen Applikation und Kernel bietet TinyOS 2.1 Message-Passing an. Wenn die Applikation einen OS-Aufruf macht, wird dieser nicht direkt ausgeführt, sondern nur eine Nachricht in Form eines Tasks gesendet. Danach versucht der Kernel-Thread den aktuell aktiven Thread zu unterbrechen und den OS-Aufruf als Task auszuführen. Dieser Mechanismus stellt sicher, dass nur der Kernel TinyOS-Code direkt ausführt. TOS-Threads allozieren Thread-Control-Blocks mit Speicher für Kontrollstrukturen und einen Stack fixer Größe. [12]

Frühere Versionen von TinyOS gewährleisteten Atomarität durch das Deaktivieren von Interrupts, d.h. dem Mikroprozessor wird mitgeteilt, dass er die Behandlung von externen Ereignissen hinauszögern und erst nachdem die atomare Operation abgeschlossen ist, wenn die Interrupts wieder aktiviert werden, ausführen soll. Diese Technik funktionierte problemlos auf Uniprozessorsystemen. Wenn solche kritische Abschnitte in einem Applikations-Thread auftreten, wirkt sich das globale Deaktivieren von Interrupts jedoch negativ auf die gesamte Systemperformance und Usability aus. Auch dieses Problem wurde in TinyOS 2.1 behandelt. Es bietet nun Möglichkeiten der Synchronisation durch Condition Variables, Semaphoren, und Mutual Exclusion Locks an. Diese Synchronisationsmechanismen wurden auf Hardwareebene, d.h. durch spezielle Instruktionen, wie beispielsweise Test & Set, realisiert. [2] [3]

3.2.3 Scheduling

Frühere Versionen von TinyOS unterstützten nur einen nicht präemptiven First-In-First-Out (FIFO) Scheduling Algorithmus. Der Kern des TinyOS Ausführungsmodells sind Tasks, die bis zu ihrer Fertigstellung nach einem FIFO Prinzip ausgeführt werden, d.h. der Scheduler führt die Tasks in der Reihenfolge der Task-Queue aus. Da TinyOS nur nicht präemptive Scheduler unterstützt, müssen die Tasks auch dementsprechend entworfen werden, d.h. sie sollten kurz sein. Tasks werden also bis zu ihrer Fertigstellung ohne Unterbrechung ausgeführt, dies gilt allerdings nur in Bezug zu anderen Tasks, sie sind aber nicht atomar in Bezug auf asynchrone Interrupthandler und auf Kommandos und Events, die von ihnen aufgerufen werden.

Auch die Nachteile eines FIFO Schedulers müssen berücksichtigt werden. Die Wartezeit eines Tasks hängt also nur von seiner Ankunftszeit in der Task-Queue ab. Das wirkt sich schlecht auf die Echtzeitunterstützung und auf die Fairness des Schedulers aus, wenn beispielsweise viele kurze Tasks hinter einem langen warten.

Um bessere Unterstützung für Echtzeitanwendungen zu bieten, wurde in TinyOS auch Unterstützung für einen Earliest Deadline First (EDF) Scheduling Algorithmus hinzugefügt, der die Tasks anhand der Deadline, zu der ein Task fertig gestellt sein muss, ordnet und damit die Einhaltung dieser Deadline garantiert. Dieser Scheduler ist aber nur in der Lage, brauchbare Pläne zu erstellen, wenn Tasks nicht abhängig von externen Ressourcen sind. Wenn dies der Fall ist, bietet TinyOS keinen stabilen echtzeitfähigen Scheduling Algorithmus an (siehe auch Abschnitt 3.2.7). [2] [3]

3.2.4 Speicherverwaltung und -schutz

TinyOS verwendet eine ausschließlich statische Speicherverwaltung.

Auf Mikrocontrollern sind die Ressourcen beschränkt und hardwaremäßiger Speicherschutz ist meist nicht verfügbar. Weiters machen diese Ressourcenbeschränkungen den Einsatz von unsicheren systemnahen Programmiersprachen notwendig, wie im Fall von TinyOS nesC. In TinyOS 2.1 wurde jedoch auch ein softwaremäßiger Speicherschutz berücksichtigt. Die Ziele sind das Abfangen von Zugriffsfehlern durch Zeiger und Arrays und brauchbare Fehlerdiagnostik und Fehlerbehandlungsstrategien. In TinyOS wird dies durch den sogenannten *Deputy* realisiert. Der *Deputy* ist Teil der Safe TinyOS Toolchain und ist ein Precompiler, welcher Typ- und Speicherschutz für C-Code gewährleistet. Code, der mit dem *Deputy* kompiliert wurde, basiert auf einer Mischung von statischen und dynamischen Überprüfungen. Safe TinyOS ist abwärtskompatibel mit früheren Versionen von TinyOS. Die Safe TinyOS Toolchain fügt Überprüfungen in den Applikationscode ein, um Sicherheit in Laufzeit zu gewährleisten. Wenn eine Überprüfung eine Sicherheitsverletzung wie z.B. eine Speicherzugriffsverletzung erkennt, werden entsprechende Aktionen, ebenfalls durch Safe TinyOS eingefügte Codeabschnitte, ausgeführt. [3] [13]

3.2.5 Externe Kommunikation in drahtlosen Sensornetzwerken

Da TinyOS ursprünglich für drahtlose Sensornetze konzipiert wurde, wurde speziell Wert auf eine effiziente drahtlose Kommunikation gelegt. Fortschritte im Bereich von integrierten Mikrosystemen (engl. Micro Electro Mechanical Systems) führten zur Entwicklung von miniaturisierten und billigen Sensoren. Diese sind nicht nur in der Lage, eine physikalische Größe zu messen, sondern auch Berechnungen damit durchzuführen und diese Daten zu übertragen, wobei vor allem ein Trend zur digitalen drahtlosen Kommunikation erkennbar ist. Ein solcher Knoten besteht dabei im Grunde aus einem Mikrocontroller, einem Transceiver und einem Analog/Digital-Konverter, und wird meist als System on Chip (SoC) umgesetzt. Abbildung 3.1 zeigt das Blockschaltbild eines typischen Knotens. Die kritischsten Ressourcen sind dabei die Energieversorgung, meist durch eine Batterie zur Verfügung gestellt, ein limitierter Speicherbereich, meist nur in der Größenordnung weniger Kilobyte, und begrenzte Rechenleistung. Der Mikrocontroller arbeitet mit einer niedrigen Taktfrequenz im Vergleich zu traditionellen Recheneinheiten.

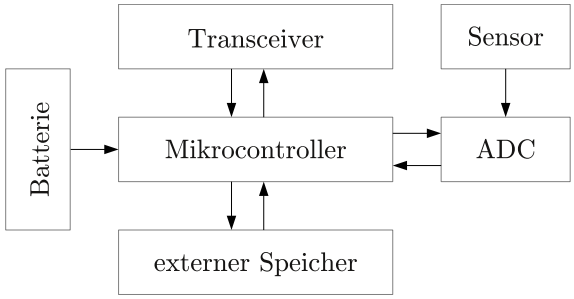

Abbildung 3.1: Blockschaltbild eines drahtlosen Sensorknotens [3]

Drahtlose Sensornetze sind dynamisch, da beispielsweise aufgrund von harten Umgebungsbedingungen oder aufgrund des Leerwerdens der Batterie die Knoten ausfallen können. Oft ist es auch nicht möglich, den Knoten nach der Inbetriebnahme auszutauschen, was weitere Anforderungen an die Lebensdauer stellt. Eine dichte Aufstellung der Sensoren und eine verteilte Abarbeitung der Messdaten, z.B. über Multi-Hop-Kommunikation zwischen den Knoten, sind Anforderungen für die Qualität, Fehlertoleranz und Zuverlässigkeit eines solchen Netzwerkes. Diese Eigenschaften stellen auch zusätzlich Anforderungen an das EOS. Es fungiert dabei als Ressourcenmanager für komplexe Systeme. In einem typischen System sind diese Ressourcen Prozessor, Speicher, Timer, Geräte oder Netzwerkschnittstellen. Die Aufgabe des OSs ist es, die Vergabe dieser Ressourcen an die Tasks in geordneter und kontrollierter Weise zu kontrollieren. Drahtlose Sensornetze sind heute von großer Bedeutung. Die Anwendungsgebiete sind dabei vielfältig, und wachsen ständig.

Frühere Versionen von TinyOS unterstützten zwei verschiedene Multi-Hop-Protokolle:

Dissemination Das Dissemination Protokoll sendet Daten zuverlässig zu jedem Knoten im Netzwerk (engl. Realiable Broadcast). Es wird beispielsweise von Administratoren zum Konfigurieren von Abfragen und zum Programmieren des Netzwerks verwendet. Das Dissemination Protokoll implementiert die zwei Schnittstellen DisseminationValue

und DisseminationUpdate. Nach dem Erzeuger-Verbraucher-Muster (eng. Producer-Consumer-Pattern) ruft der Erzeuger DisseminationUpdate auf. Das Kommando DisseminationUpdate.change soll immer dann aufgerufen werden, wenn der Erzeuger einen neuen Wert verteilen möchte. Auf der anderen Seite wird das DisseminationValue Interface vom Verbraucher verwendet. Das Event DisseminationValue.changed signalisiert einen geänderten Wert.

TYMO TYMO ist die Implementierung des Dynamic Mobile Ad-Hoc Network (MANET) On-Demand (DYMO) Protokolls, ein Routing Protokoll für mobile Ad-Hoc-Netzwerke. In TYMO wurde jedoch das Format der Pakete geändert, und es wurde aufbauend auf den aktiven Messaging-Stack implementiert.

Darüber hinaus bietet TinyOS Version 2.1 noch Unterstützung für viele weitere Protokolle, wie beispielsweise 6LoWPAN[6], eine IPv6 Vermittlungsschicht innerhalb eines Sensornetzwerks. Auf der Sicherungsschicht werden ein Single-Hop Time Division Multiple Access (TDMA) Protokoll, ein Hybrid-Protokoll aus TDMA und Carrier Sense Multiple Access (CSMA) und eine optionale Implementierung eines zu IEEE 802.15.4 konformen Protokolls unterstützt. [3]

3.2.6 Ressourcenteilung

TinyOS verwendet zwei Mechanismen, um geteilte Ressourcen zu verwalten:

Virtualisierung Eine virtualisierte Ressource erscheint als eine unabhängige Instanz, d.h. die Applikation verwendet sie unabhängig von anderen Applikationen.

Completion Events Ressourcen, die nicht virtualisiert werden können, werden durch Completion Events behandelt. Ein Beispiel hierfür ist der Kommunikationsstack von TinyOS, welcher von mehreren Threads gemeinsam benutzt wird, aber nur schlecht virtualisiert werden kann. Der Kommunikationsstack kann zu einem Zeitpunkt nur ein Paket versenden, Sendeoperationen von anderen Threads schlagen während dieser Zeit fehl. Solche geteilte Ressourcen werden durch Completion Events behandelt, die die wartenden Threads über die Fertigstellung eines bestimmten Tasks informieren. [2] [3]

3.2.7 Echtzeitunterstützung

TinyOS bietet keine explizite Unterstützung für Echtzeitanwendungen. Wie bereits in Abschnitt 3.2.3 beschrieben, laufen Tasks in TinyOS ununterbrochen bis zu ihrer Fertigstellungen in einem FIFO Prinzip. Aus diesem Grund ist TinyOS in dessen ursprünglicher Form keine gute Wahl für Echtzeitanwendugen. Wie ebenfalls bereits beschrieben, beinhalten neuere Versionen von TinyOS jedoch Unterstützung für einen EDF Scheduling Algorithmus.

TinyOS bietet keine speziellen Implementierungen für Netzwerkprotokolle auf Sicherungs-, Vermittlungs- oder Transportschicht, die die erhöhten Anforderungen für Echtzeitanwendungen erfüllen. Auf der Sicherungsschicht unterstützt TinyOS jedoch TDMA, welches an die Anforderungen der Applikation angepasst werden kann. [3]

[6] `http://www.6lowpan.net`

3.2.8 Sonstige Eigenschaften

Unterstützte Plattformen TinyOS unterstützt aktuell viele Plattformen, darunter beispielsweise Mica oder Telos[7], um nur einige zu nennen. Es wird Unterstützung für AVR, MSP430 und XScale[8] Mikroprozessoren angeboten.

Dateisystem TinyOS unterstützt ein einfaches Single-Level-Dateisystem. Der Grund dafür, dass nur eine Schicht unterstützt wird, ist die Annahme, dass nur eine einzelne Applikation am Knoten ausgeführt wird. Da auch der Speicherbereich begrenzt ist, ist ein Single-Level-Dateisystem für diese Anforderungen ausreichend. [3]

Datenbank Die Bearbeitung von Daten ist eine der Hauptaufgaben einer typischen Anwendung. Aus diesem Grund bietet TinyOS eine Datenbank in Form von TinyDB an. [14]

Simulation TinyOS bietet Simulationsunterstützung in Form von TOSSIM[9] an. Auch der gesamte Simulationscode ist in nesC geschrieben und kann dadurch ebenfalls direkt am Knoten getestet werden. [3]

[7] `http://docs.tinyos.net/tinywiki/index.php/Platform_Hardware`
[8] `http://www.intel.com/design/intelxscale`
[9] `http://docs.tinyos.net/tinywiki/index.php/TOSSIM`

4 Automationsebene

4.1 Überblick

Dieses Kapitel behandelt EOSs, die für den Einsatz in der Automationsebene geeignet sind. Wie in Abschnitt 1.3 beschrieben, stellt die Automationsebene die mittlere Schicht in einem Automationssystem dar. Wie auch in der Feldebene spielt auch hier der schonende Verbrauch von Ressourcen, im speziellen von Rechenleistung, Speicher und Energie, eine wichtige Rolle. Noch mehr im Vordergrund steht jedoch eine gewisse Flexibilität, auf die in der Feldebene noch bewusst verzichtet wurde. Weiters sind erweiterte Funktionen wie verschiedene Fehlerdiagnose- und Fehlerbehandlungsmöglichkeiten und die Unterstützung einer grafischen Benutzeroberfläche von Bedeutung. Da die Hardware der in dieser Schicht verwendeten Industrie-PCs jener von handelsüblichen Computern ähnlich ist, werden oft klassische OSs, wie Windows oder Linux, verwendet. Spezielle Anforderungen hinsichtlich Speicherverbrauch, Echtzeitfähigkeit oder Zuverlässigkeit machen aber Anpassungen dieser Systeme notwendig. Weitere Anforderungen werden an die Netzwerkfähigkeit gestellt. Geräte der Automationsebene müssen sowohl mit der Managementebene kommunizieren können, weshalb vor allem IP-basierende Netzwerktechnologien unterstützt werden müssen, als auch mit der Feldebene, was eine Unterstützung von Feldbussen notwendig macht. Einige bekannte EOSs in dieser Ebene sind:

Windows Embedded[1] ist eine Familie von EOSs von Microsoft. Dazu zählen vier verschiedene Kategorien von OSs für ESs, wodurch ein weiter Markt abgedeckt werden kann.

In der Automation von Bedeutung ist Windows Embedded Compact. Es ist ein modulares echtzeitfähiges OS mit einem angepassten Kernel, dessen Speicherverbrauch weniger als ein Megabyte beträgt und deshalb besonders für kleine Systeme geeignet ist. Beispielsweise basiert die Windows Phone OS-Familie darauf. Es ist für viele moderne Plattformen verfügbar, darunter ARM[2], MIPS[3] und x86.

Weiters ist Windows Embedded Standard eines der heute am häufigsten eingesetzten OSs in der Automation. Z.B. setzt mit Siemens[4] einer der wichtigsten Hersteller in der industriellen Automation fast ausschließlich auf Windows Embedded Standard. Es basiert auf herkömmlichen Windows Desktop OSs, ist jedoch modular aufgebaut. Im System sind dadurch nur jene Komponenten enthalten, welche auch wirklich gebraucht werden, was den Speicherverbrauch verringert und unerwünschte Nebeneffekte durch unbenutzte Module minimiert. Im Unterschied zu Windows Embedded Compact steht unter Windows Embedded Standard die gesamte Windows API zur Verfügung und es werden alle Applikationen und Gerätetreiber unterstützt, die für das entsprechende

[1] `http://www.microsoft.com/windowsembedded`
[2] `http://www.arm.com`
[3] `http://www.mips.com`
[4] `http://www.siemens.com`

Desktop OS entwickelt wurden. Windows Embedded Standard ist jedoch nicht für harte Echtzeitanwendungen geeignet. 1996 wurde mit Windows NT 4.0 Embedded die erste Version dieser Serie veröffentlicht. Darauf folgte 2001 Windows XP Embedded, die komponentenbasierte Version von Windows XP Professional, die auch heute noch in vielen ESs Verwendung findet. Die aktuelle Version ist Windows Embedded Standard 7, basierend auf Windows 7. [15]

VxWorks[5] ist ein von Wind River Systems entwickeltes proprietäres RTOS für ESs, das hauptsächlich in harten Echtzeitumgebungen Anwendung findet. Bekannte Projekte, die VxWorks verwenden, sind beispielsweise die US-amerikanischen Raumsonden Spirit und Opportunity, oder der Boeing 787 Dreamliner. Der verwendete Mikrokernel bietet eine sehr gute Unterstützung für Multitasking. Tasks haben eine fixe Priorität und werden über einen präemptiven echtzeitfähigen Round Robin Scheduler zugeteilt. Diese Echtzeit-Prozesse sind gegenüber anderen User-Mode-Tasks und gegenüber dem Kernel durch Speicherschutzmechanismen isoliert. Tasks können über umfangreiche IPC Mechanismen miteinander kommunizieren und synchronisiert werden. Weiters bietet VxWorks noch Unterstützung für verschiedene Dateisysteme und Netzwerktechnologien. VxWorks wurde auf die meisten modernen Plattformen, die heute in ESs Anwendung finden, portiert, darunter beispielsweise ARM, PowerPC oder x86. [16]

QNX[6] ist ein proprietäres unixoides RTOS. Es basiert auf einer Mikrokernel Architektur, in der der Kernel nur wenige Hauptaufgaben wie Scheduling oder IPC übernimmt. Alle anderen OS-Funktionen, wie z.B. Prozess- oder Speicherverwaltung, sind in Form von User-Mode Prozessen realisiert. Diese User-Mode Prozesse laufen in einem geschützten Speicherbereich, wodurch die Robustheit des Gesamtsystems erhöht wird, da ein fehlerhafter oder schädlicher Prozess einen anderen nicht beeinflussen kann. Weiters können nicht benötigte Funktionen einfach deaktiviert werden, ohne das OS selbst zu ändern, sondern nur indem der entsprechende Prozess nicht ausgeführt wird. QNX wurde auf viele Plattformen portiert und läuft heute auf praktisch jeder modernen CPU, die im Eingebetteten Markt Anwendung findet, darunter beispielsweise PowerPC, x86, MIPS oder ARM. QNX findet breite Anwendung z.B. in industriellen Steuerungssystemen oder in der Fahrzeugelektronik. Darüber hinaus basiert das mobile OS BlackBerry[7] 10 auf QNX. [17]

Embedded Linux ist ganz allgemein die Bezeichnung für EOSs, die auf dem offenen Linux Kernel[8] basieren. Linux befindet sich in vielen Märkten auf dem Vormarsch. Im Server-Markt oder im mobilen Bereich ist es bereits eine feste Größe, aber auch in Eingebetteten Anwendungen findet es bereits häufig Verwendung. Die Quelloffenheit, und die damit verbundene Flexibilität in der Anpassung des OSs, ist nur einer von vielen Gründen hierfür. Embedded Linux besitzt auch ein großes Potential für den Einsatz in der Automation. Aus diesem Grund wird dieses OSs im folgenden Abschnitt im Detail behandelt. [4]

[5] `http://www.windriver.com/products/vxworks`
[6] `http://www.qnx.com`
[7] `http://www.blackberry.com/`
[8] `http://www.kernel.org`

4.2 Embedded Linux

Als Embedded Linux wird typischerweise ein Gesamtsystem bezeichnet, bestehend aus OS-Kern und Systemwerkzeugen, das speziell für Eingebettete Anwendungen zugeschnitten ist. Oft wird hierfür auch der Begriff Distribution verwendet.

Als Linux wird dabei im Grunde nur der Kernel bezeichnet. Wie bereits beschrieben, unterstützt der Kernel einige grundlegende Eigenschaften, auf die die Applikationen aufbauen. Dazu gehören beispielsweise Hardware- und Geräteverwaltung, oder eine Reihe von Abstraktionen, wie Virtueller Speicher, Tasks, Sockets oder Dateien. Obwohl der Begriff Embedded auch oft in Bezug auf den Kernel verwendet wird, gibt es keine spezielle Form des Linux Kernels für Eingebettete Anwendungen. Stattdessen ist es möglich, den offiziellen Linux Kernel durch das Aktivieren und Deaktivieren vieler optionaler Funktionen für die beabsichtigte Verwendung und für eine breite Auswahl von Anwendungen und Geräten, wie beispielsweise Workstations, Server oder eben Eingebettete Geräte, zu konfigurieren. Somit kann die Unterstützung für bestimmte Eigenschaften, die in größeren Linux Systemen üblicherweise vorhanden sind, wie beispielsweise Swapping oder der Support für Terabyte an Hauptspeicher, wegfallen.

Eine Embedded Linux Distribution enthält zudem noch Systemwerkzeuge, die ebenfalls für Eingebettete Anwendungen zugeschnitten sind. Diese Werkzeuge und Programmpakete, die auf dem Kernel aufbauen, sind meist ebenfalls Open Source, aber nicht speziell nur unter Linux verwendbar, sondern können in den meisten Fällen auch auf anderen unixoiden OSs eingesetzt werden. Dazu zählen eine Reihe von System- und Entwicklungs-Tools, wie beispielsweise ein Cross-Compiler oder ein Debugger.

Embedded Linux, oder Linux im Allgemeinen, umfasst ein weites Themengebiet, welches den Umfang dieser Arbeit sprengen würde. Deshalb befassen sich die folgenden Abschnitte nur mit einigen Aspekten, die für den Einsatz in der Automation wesentlich sind. [4] [18]

4.2.1 Architektur und Programmiermodell

Ein Linux System besteht im Allgemeinen aus einer Vielzahl an Komponenten, die miteinander interagieren. Abbildung 4.1 zeigt die grundsätzliche Struktur eines Linux Systems. Die Abbildung ist eine starke Abstraktion des Kernels und der anderen Komponenten, aber ausreichend für die folgenden Betrachtungen. Linux Systeme sind in der Regel sehr ähnlich aufgebaut, womit die strukturellen Unterschiede zwischen einem Embedded Linux System und einem Server oder Desktop System auf diesem Abstraktionsniveau sehr gering sind.

Direkt über der Hardware befindet sich der Linux Kernel. Der Kernel ist der Kern des OS. Er ist monolithisch aufgebaut und ist in der Programmiersprache C geschrieben. Die Hauptaufgabe des Kernels ist es, die Hardware zu verwalten und der Benutzeranwendung Services auf einer hohen Abstraktionsebene anzubieten. Wie andere unixoide Kernel auch, ist Linux für die Ansteuerung und Verwaltung von E/A-Geräten, für das Scheduling von Prozessen und für die Speicherverwaltung verantwortlich. Applikationen, die die vom Kernel angebotene API verwenden, können dabei mit wenig bis gar keinen Änderungen auf unterschiedliche unterstützte Plattformen portiert werden.

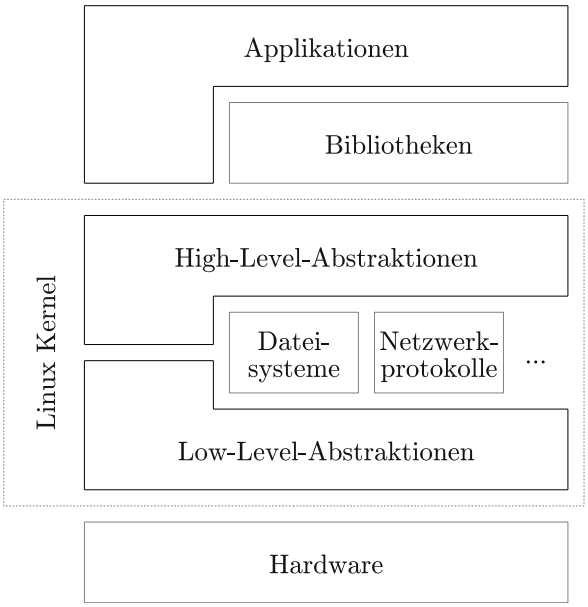

Abbildung 4.1: Architektur eines Linux Systems [4]

Im Kernel gibt es im Grunde zwei Abstraktionen, die die von der Applikation geforderte Funktionalität anbieten. Die Low-Level-Schnittstellen sind speziell auf die entsprechende Hardwarekonfiguration zugeschnitten und bieten Möglichkeiten, die Hardware-Ressourcen direkt über geräteunabhängige APIs anzusprechen. Beispielsweise werden Speicherzugriffe auf ARM und x86 System anders durchgeführt, sind aber über eine gemeinsame API für Komponenten höherer Ebenen erreichbar. Typischerweise behandeln diese Low-Level-Services CPU-spezifische Operationen, architekturspezifische Speicheroperation und grundlegende Zugriffe auf Geräte.

Darüber bieten Komponenten höherer Ebenen die allen Unix-Systemen gemeinsamen Abstraktionen an, wie z.B. Prozesse, Dateien, Sockets und Signale. Da die Low-Level-API bereits plattformunabhängig ist, kann die Implementierung dieser Komponenten unabhängig von der darunterliegenden Architektur geschehen. Es gibt jedoch einige Ausnahmen, bei denen der Kernel auf höheren Ebenen Spezialbehandlungen oder unterschiedliche Funktionen für verschiedene Architekturen verwenden muss.

Zwischen diesen Schichten werden sogenannte Interpretation-Components benötigt, um strukturierte Daten von und zu den Geräten zu verstehen und um mit diesen Geräten interagieren zu können. Dateisystemtypen oder Netzwerkprotokolle sind primäre Beispiele hierfür.

Während der normalen Ausführung benötigt der Kernel zumindest ein entsprechend strukturiertes Dateisystem, das Root-Dateisystem. Von diesem lädt der Kernel die erste Applikation, die auf dem System ausgeführt werden soll. Auch zukünftige Operationen wie beispielsweise das Laden von Modulen basieren auf diesem Dateisystem. Es kann sich entweder auf einem wirklichen Speichergerät befinden oder aber beim Start des Systems in den Random Access Memory (RAM) geladen und von dort ausgeführt werden.

Man würde annehmen, dass sich oberhalb des Kernels unmittelbar die Applikation befindet. Die vom Kernel angebotenen Services sind aber meist nicht dafür geeignet, um direkt von Benutzeranwendungen verwendet zu werden. Stattdessen verwenden Applikationen Bibliotheken, um mit dem Kernel zu interagieren, die die vom Kernel bereitgestellten Services weiter abstrahieren und eine geläufigere API anbieten. Die bekannteste dieser Bibliotheken ist die GNU C Library (glibc)[9]. Da diese Bibliothek sehr groß ist, verwenden EOSs stattdessen oft leichtgewichtigere Alternativen, wie beispielsweise die uClibc[10]. Weiters gibt es noch Universalbibliotheken wie boost[11] oder Bibliotheken für Graphical User Interfaces (GUI) wie Qt[12] oder GTK[13]. Sie bieten erweiterte Funktionalität und auch eine gewisse Plattformunabhängigkeit über verschiedene OSs hinweg an.

Bibliotheken werden typischerweise dynamisch in die Applikation eingebunden (engl. Dynamic Linking). D.h. sie sind nicht Teil der Binärdatei der Applikation, sondern werden erst beim Programmstart in den Hauptspeicherbereich der Applikation geladen. Dadurch können viele Anwendungen auf die gleiche Instanz einer Bibliothek zugreifen, und nicht jede Anwendung muss eine Kopie davon enthalten. Beispielsweise wird die glibc nur einmal beim Start des OSs in den Speicher geladen, und dann von allen Applikationen, die die Bibliothek verwenden, geteilt. Daneben gibt es auch noch das statische Einbinden von Bibliotheken (engl. Static Linking), dabei werden Bibliotheken direkt in die ausführbare Binärdatei der Applikation mitaufgenommen. In ESs wird diese Variante oft bevorzugt, beispielsweise wenn nur Teile einer Bibliothek von nur einer Applikation verwendet werden. Dadurch kann verhindert werden, dass die gesamte Bibliothek in den Speicher des Systems geladen werden muss. Weiters ergibt sich durch statisches Einbinden eine deutliche Leistungssteigerung. [4]

4.2.2 Scheduling

Da der Linux Kernel in verschiedenen Anwendungsgebieten, wie beispielsweise Servern, Desktops oder eben ESs, mit teils unterschiedlichen Lastverteilungen Einsatz findet, sind die Anforderungen an den Linux Scheduler hoch. Anfänglich verwendete der Linux Kernel sehr einfache Scheduler, welche aber mit der steigenden Anzahl von Prozessen und Prozessoren nicht zurechtkamen. Der Scheduler in Linux 2.4 war beispielsweise nicht für den Einsatz in Symmetric Multiprocessing (SMP) Systems optimiert, was die Effizienz des Schedulers negativ beeinflusste. Weiters wurde ein nicht präemptiver Ansatz verwendet, was wiederum schlechte Auswirkungen auf die Fairness hatte. Der in Linux 2.6 erstmals verwendete O(1)-Scheduler sollte vieler dieser Probleme beheben. Wie der Name schon sagt, war dieser Scheduler in der Lage, eine Scheduling Entscheidung in konstanter Zeit, also unabhängig von der Anzahl der Tasks, zu treffen. Zur Entscheidungsfindung wurden zahlreiche Heuristiken verwendet, beispielsweise um zu bestimmen, ob ein Task E/A-gebunden oder CPU-gebunden ist. Dies machte den Scheduler aber komplex und schwer zu warten.

[9] `http://www.gnu.org/software/libc`
[10] `http://www.uclibc.org`
[11] `http://www.boost.org`
[12] `http://www.qt-project.org`
[13] `http://www.gtk.org`

Aus diesem Grund kommt seit Version 2.6.23 (Okt. 2007) der sogenannte Completely Fair Scheduler (CFS) zum Einsatz. Die Idee hinter dem CFS ist es, Fairness in Form von Prozessorzeit, die den Tasks zugewiesen wird, zu gewährleisten. Wenn die Prozessorzeit der Tasks nicht im Gleichgewicht ist, d.h. wenn einem oder mehreren Tasks nicht eine faire Menge an Prozessorzeit zugeteilt wurde, dann soll jenen Tasks, die nicht im Gleichgewicht sind, Zeit zum Ausführen gegeben werden. Um das Gleichgewicht zu bestimmen, verwaltet der Scheduler die Prozessorzeit, die einem Task zugeteilt wurde, was als Virtual Runtime bezeichnet wird. Je kleiner diese Virtual Runtime eines Tasks ist, desto höher ist dessen Bedarf an Prozessorzeit.

Anstatt aber die ausführbaren Tasks in einer Queue zu verwalten, wie das bei den vorherigen Schedulern üblich war, verwendet der CFS dazu einen Rot-Schwarz-Baum, der nach der Virtual Runtime sortiert ist. Ein Rot-Schwarz-Baum ist ein höhenbalancierter binärer Suchbaum. Er ist immer balanciert, d.h. die Anzahl der Knoten auf dem längsten Pfad von der Wurzel zu einem Blatt ist nie mehr als doppelt so hoch wie jene auf dem kürzesten Pfad. Dadurch sind Operationen in einem Baum, wie beispielsweise suchen oder einfügen, in O(log n) möglich, wobei n die Anzahl der Knoten ist.

Durch eine nach der Virtual Runtime sortierte Speicherung der Tasks in einem Rot-Schwarz-Baum befinden sich Tasks, die den Prozessor dringend brauchen (niedrige Virtual Runtime), auf der linken Seite des Baums, und Tasks, die den Prozessor weniger dringend brauchen, auf der rechten Seite. Der Scheduler wählt dann den Task ganz links aus, und weist ihm als nächstes den Prozessor zu. Nachdem der Task für eine variable Zeitdauer ausgeführt wurde, wird diese Zeit zur Virtual Runtime hinzugefügt, und der Task wieder neu in den Baum eingefügt. Auf diese Weise werden Tasks auf der linken Seite des Baums abgearbeitet und der Inhalt des Baums wandert von links nach rechts. Dadurch wird ein Gleichgewicht in der Ausführung aller lauffähigen Tasks gewährleistet.

Prioritäten werden nicht direkt verwendet, sondern nur über unterschiedliche Abklingfaktoren für die erlaubte Ausführungszeit eines Tasks angewendet. Der CFS umfasst weiters noch das Sleeper-Fairness Konzept. Damit wird sichergestellt, dass Tasks, die gerade nicht ausgeführt werden (da sie z.B. gerade auf E/A Geräte warten), einen vergleichbaren Anteil am Prozessor erhalten, wenn sie ihn dann eventuell brauchen. Der CFS bietet noch weit mehr Funktionalität, wie beispielsweise Scheduling von Gruppen, auf die hier aber nicht weiter eingegangen wird. [2] [18] [19]

4.2.3 Externe Kommunikation

Viele ESs werden heute in universellen Netzwerken, damit sind vor allem IP-basierende Rechnernetze gemeint, eingesetzt. Obwohl diese Geräte in ihren Ressourcen weit mehr beschränkt sind, wird dennoch von ihnen erwartet, die gleiche Funktionalität wie typische Computersysteme zu unterstützen. Da Linux unter anderem breite Verwendung in Mainstream-Servern findet, bietet es eine gute Unterstützung für universelle Netzwerke:

Ethernet (IEEE 802.3) ist heute der meist verbreitete und auch effizienteste Netzwerktyp. Die Übertragungsgeschwindigkeit wurde im Laufe der Zeit immer erhöht. Dadurch und auch durch die Anforderung der Netzwerkfähigkeit von ESs sind viele

Eingebettete Plattformen mit Ethernet-Hardware, wie z.B. einer Network Interface Card (NIC), ausgerüstet. Linux unterstützt eine viele 10 Megabit (IEEE 802.3i) und 100 Megabit (IEEE 802.3u), und auch einige Gigabit (IEEE 802.3ab) Ethernet Geräte.

Wi-Fi (IEEE 802.11) beschreibt eine drahtlose Datenübertragung im 2.4 Gigahertz (802.11b) und 5 Gigahertz (802.11a) Band. Heute ist es das drahtlose Pendant zu Ethernet, im Sinne von weiter Verbreitung und Mainstream Support, und ist dementsprechend von ähnlicher Bedeutung für die Netzwerkfähigkeit. Linux bietet eine umfassende Unterstützung für IEEE 802.11b Hardware. Eine komplette Liste aller aktuell unterstützten Geräte und dazugehörigen Treiber ist unter [18] zu finden. Da die meisten Operationen von IEEE 802.11 gleich sind wie jene von Ethernet, braucht der Kernel keine zusätzlichen Subsysteme, um sie zu unterstützen. Es können auch weitestgehend dieselben Tools verwendet werden, mit Ausnahme der Besonderheiten von IEEE 802.11, wie beispielsweise das Setzen des Service Set Identifier (SSID) und der Schlüssel für Kryptographie und das Monitoring von Signalstärke und Verbindungsqualität.

TCP/IP und UDP/IP Auf höheren Schichten (Vermittlungs- und Transportschicht) bietet Linux Unterstützung für TCP/IP und UDP/IP in Form von Sockets. Ein Socket ermöglicht dabei eine Kommunikation zwischen einem Client Prozess und einem Server Prozess, und kann verbindungsorientiert (engl. Stream Sockets, TCP) oder verbindungslos (engl. Datagram Sockets, UDP) sein. Ein Socket kann also als ein Endpunkt in einer Kommunikation angesehen werden. Die beiden Endpunkte können dabei auf dem gleichen Computer liegen (Interprozesskommunikation) oder auf unterschiedlichen (Netzwerkkommunikation). Ein Client Socket verwendet eine Adresse, im Fall von IP die IP-Adresse, um sich zu einem Server Socket auf einem anderen Prozess oder Computer zu verbinden und Daten mit ihm auszutauschen. Der Datenaustausch kann dabei bidirektional erfolgen. Der Benutzerprozess interagiert also nicht direkt mit dem Ethernet oder Wi-Fi Gerätetreiber, sondern über eine abstrakte Socket-Schnittstelle mit den Funktionen send und recv. Über das Socket-Modul wird das Transportschicht-Modul (TCP und UDP) und das IP-Modul, und schließlich erst der Gerätetreiber für die NIC angesprochen. [2]

Industrielle Steuerungstechnik und Automatisierung sind, wie andere rechnergestützte Anwendungen auch, sehr auf Netzwerkfähigkeit angewiesen. Universelle Netzwerktechniken wie Ethernet oder Token Ring sind für den Einsatz in harschen und anspruchsvollen Umgebungen von industriellen Applikationen oft nicht geeignet. Standard Ethernet ist beispielsweise zu verwundbar für Electromagnetic Interference (EMI) oder Radio Frequency Interference (RFI), um in typischen industriellen Umgebungen eingesetzt werden zu können. Weiters werden höhere Ansprüche an ein deterministisches Zeitverhalten gestellt. Für diese Anforderungen wurden im Laufe der Zeit spezielle industrielle Netzwerke, als Feldbusse bekannt, entwickelt. Um sich an diese Anforderungen anzupassen, wird versucht, beispielsweise den Verdrahtungsaufwand zu verringern, die Modularität zu verbessern oder erweiterte Möglichkeiten der Diagnose und Fehlerbehebung anzubieten.

Linux bietet aktuell Support für viele bekannte industrielle Netzwerke, wie beispielsweise das Controller Area Network (CAN), ARCnet[14] oder Modbus[15]. Natürlich gibt es neben den genannten noch unzählige andere industrielle Netzwerke. Für einige davon gibt es aktuell noch gar keinen Support in Linux, wie beispielsweise für das Actuator-Sensor-Interface (AS-i), für ControlNet oder für SERCOS[16], um nur einige bekannte zu nennen. Für andere gibt es zwar Ansätze, um von wirklichem Linux-Support sprechen zu können ist aber noch weitere Arbeit notwendig. Beispiele hierfür sind Interbus[17] oder LonWorks, für die aktuell nur experimentelle Treiber verfügbar sind. [4]

4.2.4 Echtzeitunterstützung

Es gibt eine Reihe von Aspekten, die den Einsatz von Linux in harten Echtzeit Anwendungen bisher verhindert haben:

Scheduling Der Einsatz von dynamischen Prioritäten eignet sich zwar gut für Time-Sharing-Systeme, kann aber verhindern, dass einem wichtigen Prozess die CPU zugewiesen wird, wenn er sie braucht.

Speicherverwaltung Die komplexe Speicherverwaltung in Linux wirkt sich negativ auf das Echtzeitverhalten aus. Paging kann unerwartete Verzögerungen verursachen, beispielsweise wenn die Seite im Speicher gerade gesperrt ist. Auch ein mögliches Remapping der Page-Table-Entries in der Memory-Management-Unit (MMU) kann einen schnellen Kontextwechsel verhindern.

Ungenaue/grobe Synchronisationsmechanismen Aufgrund des nicht präemptiven Kernels kann das System im Fall von langen Kernel Operationen nicht sofort auf Ereignisse reagieren.

Der aktuelle Kernel ab Version 2.6 bietet bereits einige Lösungen für die oben genannten Probleme. Beispielsweise wurden, wie in Abschnitt 4.2.2 beschrieben, Verbesserungen im Scheduler gemacht. Auch wurde der Kernel präemptiv gemacht, atomare Kernelsegmente wurden verkleinert und durch Spinlocks geschützt, anstatt Interrupts zu deaktivieren. Auch Swapping kann deaktiviert werden. Für weiche Echtzeitaufgaben, wie beispielsweise Datenerfassung, ist ein aktuelles Standard Linux System daher bereits gut geeignet, und kann daher für viele Anwendungen, welche gelegentliche minimale Verzögerungen in der Reaktion tolerieren, eingesetzt werden.

Linux kann aber nicht als hartes RTOS bezeichnet werden. Trotzdem wurde Linux bereits in einigen harten Echtzeit Anwendungen eingesetzt [20]. Dies wurde hauptsächlich dadurch erzielt, dass Interrupts völlig deaktiviert wurden, wenn Echtzeitverhalten gefordert war. Durch das Deaktivieren von Interrupt stehen aber viele OS-Dienste nicht mehr zur Verfügung. Deshalb kann diese Lösung auch nur für spezielle Anwendungen eingesetzt werden, aber nicht generell, z.B. wenn die Dauer, in der Echtzeit gefordert ist, nur sehr kurz ist.

[14] `http://www.arcnet.com`
[15] `http://www.modbus.org`
[16] `http://www.sercos.com`
[17] `http://www.interbus.com`

Zwei Open Source Linux Erweiterungen, Xenomai[18] und RTAI[19], verfolgen einen anderen Ansatz. Sie erweitern Linux um die Möglichkeit, einen Satz von Echtzeit Tasks zu definieren, denen die CPU in einer deterministischen garantierten Zeitdauer zur Verfügung gestellt wird, wenn sie diese brauchen. RTAI und Xenomai stammen vom gleichen Projekt ab und basieren auf denselben Grundsätzen. Sie sind kein Ersatz für Linux, sondern eine zusätzliche Komponente in einem Embedded Linux System, die mit dem Kernel zusammenarbeitet und für das Scheduling von Echtzeit Tasks verantwortlich ist. Für die restliche Funktionalität ist aber weiterhin Linux zuständig. RTAI und Xenomai werden beide aktiv entwickelt und stellen eine alternative zu proprietären OSs, wie z.B. VxWorks, dar.

Um mit dem Kernel kooperieren zu können, muss die darunterliegende Hardware vom Kernel und Xenomai/RTAI geteilt werden. Diese Aufgabe übernimmt der ADEOS Nanokernel[20]. Hardware Interrupts werden normalerweise von ADEOS behandelt, welcher diese dann an die entsprechenden Komponenten, also RTAI/Xenomai oder dem Linux Kernel, weiterleitet. Die Komponenten werden von ADEOS logisch in Form einer Pipeline organisiert. Die Komponente am Kopf der Pipeline, in diesem Fall RTAI/Xenomai, empfängt die Benachrichtigung eines Interrupts als erstes und kann dann entscheiden, ob ADEOS den Interrupt zu den unteren Komponenten in der Pipeline, in diesem Fall Linux, weiterleiten soll. Dadurch sind die Antwortzeiten unabhängig vom Linux Kernel. Abbildung 4.2 zeigt die Struktur eines Xenomai Systems.

RTAI ist hingegen leicht anders organisiert, wie in Abbildung 4.3 ersichtlich. Anstatt ADEOS alle Interrupts behandeln zu lassen, fängt RTAI jene Interrupts, an denen es besonders interessiert ist, z.B. Interrupts die das Echtzeitscheduling beeinflussen, direkt ab. Der Grund für diesen Ansatz ist Effizienz, da der Overhead, der durch die Behandlung des Interrupts in ADEOS entsteht, wegfällt.

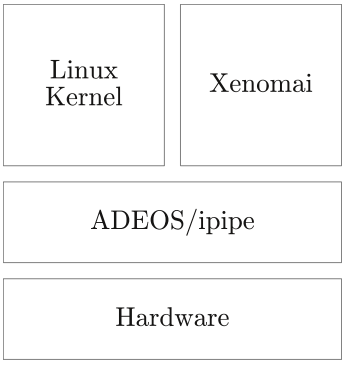

Abbildung 4.2: Struktur eines Xenomai Systems [5]

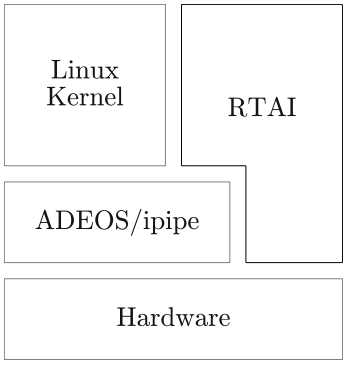

Abbildung 4.3: Struktur eines RTAI Systems [5]

Echtzeit Tasks können sich entweder im primären Modus, in dem sie hart echtzeitfähig sind, oder im nicht deterministischen sekundären Modus, in dem sie herkömmliche Sys-

[18] `http://www.xenomai.org`
[19] `https://www.rtai.org`
[20] `http://home.gna.org/adeos`

temaufrufe an den Linux-Kernel absetzen können, befinden. Ein Wechsel in den primären bzw. sekundären Modus erfolgt automatisch, wenn ein API-Aufruf an Xenomai bzw. Linux durchgeführt wird. Dadurch ist eine einfache Trennung von echtzeitfähigen und nicht echtzeitfähigen Anteilen, wie beispielsweise Initialisierung, Visualisierung oder Debugging, möglich. Weiters bietet Xenomai Ersatz und Ergänzungen von Diensten an, die in ihrer Linux Variante nicht deterministisch und damit nicht echtzeitfähig sind. Dabei handelt es sich vor allem um IPC Mechanismen wie sogenannte Echtzeit Pipes zum Datenaustausch zwischen Echtzeit und Linux Tasks, Semaphoren und Mutual Exclusion Locks, aber auch die Verwaltung von Echtzeit Tasks oder die dynamische Speicherverwaltung. [5] [21]

4.2.5 Unterstützte Plattformen

Linux stellt einige grundlegende Anforderungen an die darunterliegende Hardware. Beispielsweise benötigt es zumindest eine 32 Bit CPU mit einer Memory Management Unit (MMU). Weiters muss genügend Arbeits- und Hauptspeicher zur Verfügung stehen. Schließlich muss der Kernel in der Lage sein, ein Root-Dateisystem zu laden und darauf zuzugreifen.

Linux gehört heute zu einem der am häufigsten portierten Systeme. Die Unterstützung durch eine große Open Source Community und durch eine Reihe namhafter Hersteller ist sicher ein wesentlicher Grund dafür. Der offizielle Kernel unterstützt aktuell 24 Architekturen, darunter ARM, x86, MIPS oder PowerPC, um nur einige zu nennen, die speziell in ESs Anwendung finden. [4] [18]

4.2.6 Distributionen

Obwohl es grundsätzlich möglich ist, ein Embedded Linux System selbst zusammenzustellen, also die in Abschnitt 4.2.1 beschriebenen Komponenten wie Kernel, Betriebssoftware und Entwicklungswerkzeuge nach den eigenen Anforderungen auszuwählen und zu konfigurieren, erfordert dies viel Aufwand und Kenntnisse in diesem Bereich. Gerade wenn es um die Minimierung der Produkteinführungszeit (Time-to-Market) geht, ist dies sehr wichtig. Daher haben sich einige Hersteller zum Ziel gesetzt, kommerzielle oder auch freie Embedded Linux Distribution zu vertreiben. Einige bekannte, die sich für den Einsatz in der Automation eignen, sind im Folgenden angeführt:

MontaVista Linux[21] MontaVista ist eines der führenden Unternehmen am Embedded Linux Markt. Unter dem Namen MontaVista Linux bietet es eine Vielzahl von Produkten und Services für industrielle Anwendungen an. MontaVista beteiligt sich auch aktiv an der Kernelentwicklung, dazu gehören beispielsweise Verbesserungen im Scheduler und Echtzeiterweiterungen am Linux Kernel. Aus diesem Grund war MontaVista auch eines der ersten Unternehmen, das eine kommerzielle Distribution mit dem Linux Kernel 2.6, und die damit einhergehenden Verbesserungen im Echtzeitverhalten, wie unter 4.2.2 beschrieben, angeboten hat.

LynuxWorks Embedded Linux[22] LynuxWorks, früher unter dem Namen Lynx Real Time Systems bekannt, ist ein weiterer traditioneller Hersteller von EOSs. Ursprünglich

[21] `http://www.mvista.com/monta_vista_linux6.php`
[22] `http://www.lynuxworks.com/embedded-linux/embedded-linux.php`

entwickelte das Unternehmen das proprietäre RTOS LynxOS. Obwohl Lynx Real Time Systems damit bereits ein etabliertes EOS im Angebot hatte, setzte es schon früh auf den Einsatz von Open Source und Linux im Eingebetteten Markt. Um diese Entscheidung zu bekräftigen, änderte es daraufhin seinen Namen zu LynuxWorks und bietet seither die Distribution LynuxWorks Embedded Linux an. Sie basiert ebenfalls auf dem Linux Kernel 2.6, und ist für den Einsatz in ESs optimiert. Weiters bietet LynuxWorks noch eine Reihe von Entwicklungswerkzeugen, wie beispielsweise Erweiterungen für verschiedene Integrated Development Environments (IDE) und Debugging Tools. Im Gegensatz zu MontaVista, das ihre Embedded Linux Distribution auch als RTOS vertreibt, verfolgt LynuxWorks hier einen zweispurigen Ansatz. Anwendungen mit harten Echtzeitanforderungen sollen weiterhin auf LynxOS setzen, währenddessen für nicht echtzeitfähige Anwendungen Embedded Linux angeboten wird. LynxOS ist kompatibel zu Linux, so sind Linux Binärdateien auf LynxOS lauffähig. Ein ähnlichen Weg wie LynuxWorks verfolgt hier auch der Hersteller des unter 4.1 beschriebenen RTOSs VxWorks, WindRiver Systems.

Emdebian[23] Eine noch sehr junge Embedded Linux Distribution, die sich aber in der Open Source Community bereits breiter Beliebtheit erfreut, ist Emdebian. Beispielsweise findet es aktuell als offiziell unterstütztes OSs für den Minicomputer Rasperry Pi[24] Anwendung. Es basiert auf Debian, eine der am häufigsten verwendeten Linux Distributionen im Desktop- und Server-Bereich, und ist wie dieses ebenfalls ein Open Source Projekt. Im Vergleich zu Debian bietet Emdebian eine bessere Konfigurierbarkeit und eine feinere Kontrolle über die Auswahl der Pakete und deren Größe und Abhängigkeiten, und ist damit für den Einsatz in Eingebetteten Geräten mit beschränkten Ressourcen besser geeignet. Wie auch Debian verwendet Emdebian aber ebenfalls nur einen Standard Linux Kernel und ist deshalb nur für weiche Echtzeitaufgaben geeignet. Weiters wird eine Toolchain angeboten, die Cross-Compiling unterstützt. Emdebian wird aktuell aktiv weiterentwickelt. Das Projekt selbst bezeichnet den eigenen Zustand momentan als Work-In-Progress. Gerade auch aus diesem Grund kann es in Bezug auf Stabilität und Hersteller-Support aktuell nicht mit den oben genannten kommerziellen Distributionen gleichgesetzt werden. [4] [22]

[23] `http://www.emdebian.org`
[24] `http://www.raspberrypi.org`

5 Schlussbetrachtungen

In dieser Arbeit wurden Eingebettete Betriebssysteme in der Automation anhand ihrer Anforderungen und Eigenschaften, wie Architektur, Scheduling oder Echtzeitunterstützung, diskutiert. Weiters wurden einige konkrete EOSs vorgestellt und deren spezielle Eigenschaften beschrieben. Tabelle 5.1 zeigt eine Gegenüberstellung der beiden in dieser Arbeit genauer behandelten OSs. Diese sind jeweils Vertreter für eine Kategorie von EOSs. TinyOS eignet sich besonders für einfache Anwendungen in der Feldebene auf Geräten mit beschränkten Ressourcen wie beispielsweise intelligenten Sensoren oder Aktoren. Embedded Linux ist hingegen für einen Einsatz in komplexeren Anwendungen in der Automationsebene auf beispielsweise IPCs konzipiert. Da das Einsatzgebiet von EOSs, gerade jenes von Embedded Linux, sehr vielfältig ist, ist zu bemerken, dass diese Gegenüberstellung nur anhand einer typischen Verwendung in einem Automationssystem erfolgte.

Die in dieser Arbeit vorgestellten EOSs sind nur eine kleine Auswahl aus einer Vielzahl von sehr unterschiedlichen Systemen. Daraus ein für die eigenen Anforderungen passendes auszuwählen, ist eine schwierige Aufgabe. Anders als beispielsweise bei Desktops oder im mobilen Segment, wo aktuell das auf Linux basierende OS Android[1] klarer Marktführer ist, gibt es im Eingebetteten Bereich noch keine deutliche Vormachtstellung eines bestimmten OSs. Durch die Popularität von Linux kann sich diese Situation jedoch in naher Zukunft auch hier ändern. Laut [4] setzen zurzeit viele verschiedene Hersteller auf die Entwicklung von Embedded Linux Projekten, die jedoch bisher nicht öffentlich bekannt gemacht bzw. sogar geheim gehalten werden.

Aufgrund der steigenden Komplexität der Anwendungen im Bereich der Automation wird die Notwendigkeit einer soliden Software-Basis durch EOSs in Zukunft immer wichtiger. Durch immer leistungsfähigere Hardware übernimmt die Automationsebene mehr und mehr auch Aufgaben der Management- und Leitebene. Noch stärker ist dieser Trend auf der Feldebene erkennbar. Während viele bisherige Automationssysteme meist einem streng zentralen Ansatz folgen, ist heute eine klare Entwicklung hin zu einer dezentralen Struktur erkennbar, in der die beteiligten Sensoren und Aktoren eigenständig miteinander kooperieren. Dadurch übernimmt auch die Feldebene Aufgaben der Automationsebene, was zu einer Verschiebung der Grenzen der Automationspyramide führt. Schlagwörter wie Internet of Things oder Ubiquitous Computing bestätigen diesen Trend. Aber gerade aus diesem Grund werden die Anforderungen an die eingesetzten OSs immer höher. Dazu gehören beispielsweise die Unterstützung von mehreren parallelen Applikationen, ein Multitasking Programmiermodell oder eine dynamische Speicherverwaltung. Auch auf die Beachtung von wesentlichen Sicherheitsaspekten muss in Zukunft ein besonderes Augenmerk gelegt werden. Bei der Umsetzung dieser Funktionen muss aber weiterhin auf eine schonende Verwendung der Ressourcen geachtet werden, was eine große Herausforderung darstellt. Dies gilt besonders für Low-Level OSs in der Feldebene.

[1] `http://www.android.com`

	TinyOS	**Embedded Linux**
Architektur	applikationsspezifische komponentenbasierte Architektur mit monolithischem Kernel	modulares OS mit monolithischem Kernel
Applikations-verwaltung	eine einzige dedizierte Applikation, in Laufzeit keine Trennung zwischen OS und Applikation	mehrere dynamisch ladbare Applikationen, strikte Trennung zwischen Kernel- und User-Space
Programmier-modell	ereignisgesteuerte Programmierung in der Programmiersprache nesC	nebenläufige Programmierung in unterschiedlichen Programmiersprachen von C bis Java
Scheduling	nicht präemptiver FIFO Scheduler	unterschiedliche Scheduling-Klassen, standardmäßig präemptiver CFS
Speicher-verwaltung	statische Speicherverwaltung	komplexe virtuelle Speicherverwaltung mit dynamischer Allozierung
Echtzeit-unterstützung	keine explizite Echtzeitunterstützung	native Unterstützung für weiche Echtzeit, Unterstützung harter Echtzeit durch Erweiterungen wie z.B. Xenomai
OS-Overhead	durch spezielle Architektur gering	durch aufwändige aber leistungsfähige OS-Mechanismen relativ groß
Plattform	energiesparende Mikrocontroller mit Taktfrequenzen im Megahertz Bereich, RAM und ROM im Kilobyte Bereich	IPCs mit leistungsfähigen 32/64 Bit Prozessoren, hunderte Megabyte RAM und mehrere Gigabyte Sekundärspeicher

Tabelle 5.1: Gegenüberstellung von TinyOS und Embedded Linux

Literaturverzeichnis

[1] T. Sauter, S. Soucek, W. Kastner und D. Dietrich: *The Evolution of Factory and Building Automation.* Industrial Electronics Magazine, IEEE, 2011.

[2] W. Stallings: *Operating systems: internals and design principles.* Prentice Hall, 2008.

[3] M. O. Farooq und T. Kunz: *Operating systems for wireless sensor networks: a survey.* Sensors, 2011.

[4] K. Yaghmour, J. Masters, G. Ben-Yossef und P. Gerum: *Building embedded Linux systems.* O'Reilly Media, Incorporated, 2008.

[5] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa und C. Taliercio: *Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application.* In: *Real-Time Conference, 2007 15th IEEE-NPSS*, Seiten 1–5. IEEE, 2007.

[6] M. Barr: *Embedded systems glossary. Retrieved on.* Netrino Technical Library, Seiten 4–21, 2007.

[7] A. S. Tanenbaum und A. S. Woodhull: *Operating systems: design and implementation*, Band 68. Prentice Hall, 1997.

[8] H. Kopetz: *Real-time systems: design principles for distributed embedded applications*, Band 25. Springer, 2011.

[9] Real Time Engineers Ltd.: *Offizielle FreeRTOS Website.* `http://www.freertos.org`, abgerufen am 5. Februar 2013.

[10] A. Dunkels, B. Gronvall und T. Voigt: *Contiki-a lightweight and flexible operating system for tiny networked sensors.* In: *29th Annual IEEE International Conference on Local Computer Networks*, Seiten 455–462. IEEE, 2004.

[11] Kevin K., C. J. M. Liang, J. Paek, R. Musaloiu-E., P. Levis, A. Terzis und R. Govindan: *TOSThreads: thread-safe and non-invasive preemption in TinyOS.* In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, Seiten 127–140. ACM, 2009.

[12] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*: *Tinyos: An operating system for sensor networks.* Ambient intelligence, 35, 2005.

[13] N. Cooprider, W. Archer, E. Eide, D. Gay und J. Regehr: *Efficient memory safety for TinyOS.* In: *Proceedings of the 5th international conference on Embedded networked sensor systems*, Seiten 205–218. ACM, 2007.

[14] S. R. Madden, M. J. Franklin, J. M. Hellerstein und W. Hong: *TinyDB: An acquisitional query processing system for sensor networks.* ACM Transactions on Database Systems (TODS), 2005.

[15] Microsoft: *Offizielle Windows Embedded Website.* `http://www.microsoft.com/windowsembedded`, abgerufen am 10. Februar 2013.

[16] Windriver Systems: *Offizielle VxWorks Website.* `http://www.windriver.com/products/vxworks`, abgerufen am 10. Februar 2013.

[17] QNX Software Systems Limited: *Offizielle QNX Website.* `http://www.qnx.com`, abgerufen am 20. März 2013.

[18] The Linux Kernel Organization, Inc.: *The Linux Kernel Archives.* `http://www.kernel.org`, abgerufen am 25. Februar 2013.

[19] M. T. Jones: *Inside the Linux 2.6 completely fair scheduler.* IBM DeveloperWorks, Tech. Rep., 2009.

[20] B. G. Penaflor, J. R. Ferron, M. L. Walker, D. A. Piglowski und R. D. Johnson: *Real-time control of DIII–D plasma discharges using a Linux alpha computing cluster.* Fusion engineering and design, 2001.

[21] S. Smolorz: *Echtzeit-Linux mit Xenomai.* Elektronik, März, 2007.

[22] The Emdebian Debian Project: *Offizielle Emdebian Website.* `http://www.emdebian.org`, abgerufen am 26. Februar 2013.

Akronyme und Abkürzungen

EOS Embedded Operating System

ES Embedded System

OS Operating System

PC Personal Computer

FPGA Field Programmable Gate Array

ASIC Application Specific Integrated Circuit

A/D Analog/Digital

D/A Digital/Analog

SPS Speicherprogrammierbare Steuerung

IPC Industrie PC

MES Manufacturing Execution System

SCADA Supervisory Control and Data Acquisition

ERP Enterprise Resource Planning

E/A Ein-/Ausgabe

CPU Central Processing Unit

IPC Interprozesskommunikation

API Application Programming Interface

OSI Open Systems Interconnection

TCP Transmission Control Protocol

UDP User Datagram Protocol

MAC Medium Access Control

RTOS Real Time OS

IP Internet Protocol

nesC network embedded systems C

SoC System on Chip

FIFO First-In-First-Out

EDF Earliest Deadline First

DYMO Dynamic MANET On-Demand

MANET Mobile Ad-Hoc Network

TDMA Time Division Multiple Access

CSMA Carrier Sense Multiple Access

glibc GNU C Library

GUI Graphical User Interface

RAM Random Access Memory

SMP Symmetric Multiprocessing

CFS Completely Fair Scheduler

NIC Network Interface Card

SSID Service Set Identifier

EMI Electromagnetic Interference

RFI Radio Frequency Interference

CAN Controller Area Network

AS-i Actuator-Sensor-Interface

MMU Memory-Management-Unit

IDE Integrated Development Environment

ROM Read Only Memory