Das RTOS Symobi: Erfüllung der Anforderungen in eingebetteten Systemen

Robert Dörfel
Technische Universität München
Fakultät für Informatik
doerfel@in.tum.de

Michael Haunreiter
Miray Software AG
m.haunreiter@miray.de

Uwe Baumgarten
Technische Universität München
Fakultät für Informatik
baumgaru@in.tum.de

ABSTRACT

In eingebetteten Systemen gibt es viele Anforderungen, die das Betriebssystem erfüllen muss. Diese reichen von stark begrenzten Hardwareressourcen, Zuverlässigkeit, Echtzeitfähigkeit bis hin zur Portabilität des Betriebssystems. Die Herausforderung bei der Entwicklung eines solchen Betriebssystems besteht vor allem darin, diese sich teilweise widersprechenden Eigenschaften in einem Gesamtsystem optimal zu vereinen. Das embeddable RTOS Symobi bietet mit seiner Architektur einen modernen Ansatz, mit dem die genannten Anforderungen vereint und gleichzeitig jede einzelne weitgehend erfüllt werden kann.

1. EINLEITUNG

Betriebssysteme für eingebettete Systeme stehen vor dem Konflikt, dass sie einerseits maximale Leistung mit den meist ohnehin knapp bemessenen Hardwareressourcen erzielen müssen, andererseits aber auch möglichst vielseitig einsetzbar sein sollen. Sie sollten daher bei der Erfüllung der an sie gestellten Anforderungen eine möglichst gute Balance zwischen Abstraktion und Spezialisierung bieten. Zu den wichtigsten Anforderungen an ein Betriebssystem für eingebettete Systeme gehört die *Zuverlässigkeit*, weil sie in Umgebungen eingesetzt werden, in denen ein Ausfall der Hardware oder ein Absturz des Betriebssystems in der Regel zu größeren Schäden führen kann als bei einem Desktopsystem. Beispielsweise führt ein Ausfall einer Fertigungssteuerung in der Fabrik zu hohen wirtschaftlichen Schäden. Eingebettete Systeme werden häufig in der Steuerung von technischen Prozessen verwendet. Dort spielt auch die *Echtzeitfähigkeit* eine große Rolle, da das System rechtzeitig auf Sensorwerte reagieren muss, um entsprechend die Aktuatoren zu steuern. Durch die große Anzahl der verschiedenen Hardware-Plattformen sollte das Betriebssystem in eingebetteten Systemen sehr anpassungsfähig sein. So ist es sehr wertvoll, wenn sich das Betriebssystem durch eine schnelle und leichte *Portierbarkeit* auszeichnet, damit es auf den unterschiedlichen in diesem Bereich eingesetzten Prozessoren läuft [3]. Nicht nur die Prozessor-Plattform unterscheidet sich in eingebetteten Systemen sondern auch die Größe des Arbeitsspeichers und die Anzahl und Art der angeschlossenen Geräte. Somit kann ein Betriebssystem mit guter *Skalierbarkeit* leicht an die jeweilige Hardware angepasst werden. Dabei sollte sich das Betriebssystem nicht nur zur Compile- und Installationszeit gut skalieren lassen, sondern auch dynamisch zur Laufzeit. Durch die Ressourcenknappheit, die in eingebetteten Systemen vorherrscht, muss das Betriebssystem in seiner Minimalkonfiguration sehr schlank sein. Trotzdem wird erwartet, dass das Betriebssystem je nach Anforderung und Ausstattung des eingebetteten Systems wachsen kann. Eingebettete Systeme sind zunehmend auch kommunikationsorientiert. Dies fordert vom Betriebssystem möglichst transparente *Kommunikationsmechanismen* für Anwendungen und Dienste. Neben der geforderten Zuverlässigkeit steht in eingebetteten Systemen die *Effizienz* des Betriebssystems im Vordergrund. Da in eingebetteten Systemen die Rechenleistung der Prozessoren häufig niedriger als bei Desktop- oder Serversystemen ist, muss das Betriebssystem möglichst effizient arbeiten, um die für sich selbst benötigte Prozessorlast und den Speicherplatz gering zu halten. Viele der eingebetteten Systeme werden mit Batterie in platzsparenden Gehäusen betrieben. Durch *Energiemanagement* seitens des Betriebssystems kann die Leistung des eingebetteten Systems an momentane Anforderungen angepasst werden. So kann Energie gespart und Wärmeentwicklung vermieden werden, was zu einer längeren Akkulaufzeit führt und unerwünschte oder gar schädliche Wärmeentwicklung im Gehäuse verhindert.

Das RTOS Symobi[1] berücksichtigt bereits in seinem Architekturansatz die genannten Anforderungen. Ziel von Symobi ist es, diese in einem Betriebssystem zu vereinen und gleichzeitig jede einzelne so weitgehend zu erfüllen, dass das Ergebnis mit dem einseitig spezialisierter Betriebssysteme vergleichbar ist. Im nächsten Abschnitt wird zunächst die Architektur von Symobi beschrieben und auf die Besonderheiten des Systems eingegangen. Anschließend wird in Abschnitt 3 vorgestellt, wie Symobi die genannten Anforderungen an ein Betriebssystem für eingebettete Systeme erfüllt.

GI/ITG KuVS Fachgespräch: Systemsoftware und Energiebewusste Systeme, 11. Oktober 2007, Universität Karlsruhe

[1]Das embeddable RTOS Symobi wurde von der Firma Miray Software von Grund auf neu entwickelt. Seit 2002 besteht dabei eine enge Zusammenarbeit mit der Arbeitsgruppe MVS unter Leitung von Prof. Dr. Baumgarten am Lehrstuhl I13 der TU München. Die Zusammenarbeit konzentriert sich besonders auf den Einsatz von Symobi auf mobilen embedded Systemen und hat unter anderem die Unterstützung zum Einsatz von Symobi auf diversen ARM-basierten embedded Systemen (PXA25x, PXA270, IXP425) geführt.

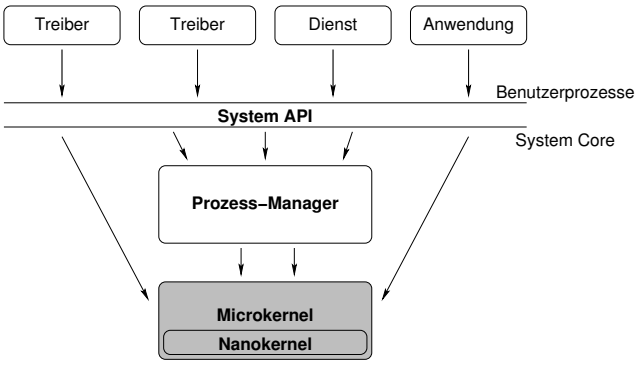

Abbildung 1: Die Microkernel-Architektur von Symobi. Die Pfeile stellen die Funktionsaufrufrichtung dar.

Anhand eines Beispiels zur Benutzerinteraktion in Symobi wird die modulare und skalierbare Architektur von Symobi verdeutlicht. In Abschnitt 3.7 wird beschrieben, wie neben dem im Microkernel vorhandenen allgemeinen auch ein hardwarespezifisches, plattformabhängiges Energiemanagement in Symobi integriert werden kann. Abschließend wird zusammengefasst, wie Symobi die Anforderungen in eingebetteten Systemen mit seinen Konzepten löst.

2. ARCHITEKTUR

Symobi baut auf einer Microkernel-Architektur auf, die in Abbildung 1 dargestellt ist. Basis des Betriebssystems ist der abgeschlossene Microkernel, auf dem die restlichen Komponenten des Betriebssystems aufbauen. Der Prozess-Manager, der zusammen mit dem Microkernel den so genannten *System Core* bildet, sorgt für die Verwaltung der Prozesse. Treiber und Betriebssystemdienste - Aufrufe des System Core werden nicht als Dienste, sondern als Systemaufrufe bezeichnet - sind nicht im Microkernel enthalten und laufen wie Anwendungen als eigenständige Prozesse im nicht-privilegierten Prozessormodus. Über das in C++ implementierte, objektorientierte System API erhalten die Benutzerprozesse Zugriff auf den System Core. Nur der in Abbildung 1 grau hinterlegte Microkernel läuft im privilegierten Prozessormodus. Selbst der Prozess-Manager, der Teil des System Cores ist, wird im nicht-privilegierten Modus ausgeführt.

Da die Treiber, Dienste und Anwendungen in eigenen Prozessen laufen, die getrennte, zugriffsgeschützte Adressräume haben, steht für den Nachrichtenaustausch ein IPC-Mechanismus zur Verfügung, auf den in Abschnitt 2.4 eingegangen wird.

2.1 System Core

Die Basis von Symobi bildet der System Core, der aufgeteilt ist in den privilegierten Microkernel und den nicht-privilegierten Prozess-Manager. Über diese wird eine Trennung zwischen Mechanismus und Strategie realisiert. Zu den Aufgaben des System Core gehören Scheduling, Synchronistionsoperationen, I/O-Management, Speichermanagement und die IPC. Während der Microkernel dem Prozess-Manager und den anderen Benutzerprozessen primitive Operationen zur Verfügung stellt, enthält der Prozess-Manager die nötigen Strategien für komplexere Betriebssystemoperationen und den IPC Mechanismus. Treiber, Dienste und Anwendungen sind in normalen Benutzerprozessen organisiert. Über das System API können diese auf die Funktionen des System Core zugreifen.

Zum Funktionsumfang des Microkernels gehören Scheduling, Thread-Synchronisation, Reservierung und Freigabe von Systemressourcen und Interrupts.

Der Microkernel enthält einen so genannten Nanokernel, der die prozessorspezifischen Low-Level-Operationen implementiert und so dem übrigen Microkernel eine Abstraktion der prozessornahen Hardware zur Verfügung stellt. Die Schnittstelle zwischen dem Nanokernel und dem Rest des Microkernels ist für alle Prozessorplattformen identisch. Bestimmte Prozessoreigenschaften können falls notwendig vom Nanokernel durch Softwareimplementierungen ersetzt werden, damit unter allen Plattformen die einheitliche Schnittstelle vorhanden ist. Nur im Nanokernel sind Codeanteile in Assembler enthalten. Der übrige Microkernel nutzt diese Schnittstelle und ist selbst ausschließlich in C geschrieben.

Der Prozess-Manager enthält hauptsächlich die Strategien für Ressourcenverwaltung und IPC. Bei der Vergabe von Systemressourcen entscheidet der Prozess-Manager wie die Ressourcen verteilt werden und reserviert diese entsprechend über die Primitiven des Microkernels. Dieser Mechanismus wird anhand des Speichermanagements veranschaulicht. Der Microkernel verwaltet den Speicher, der über seine Funktionen reserviert und freigegeben werden kann. Wenn ein Benutzerprozess Speicher reserviert, entscheidet der Prozess-Manager, welcher Speicherbereich verwendet wird und reserviert diesen über die Funktionen des Microkernels.

Der Prozess-Manager enthält überwiegend C++ Programmcode, wobei aus Gründen der Performance auf C++-Sprachmerkmale verzichtet wird, die sich negativ auf das Laufzeitverhalten auswirken können (z. B. Exceptions).

2.2 Treiber und Dienste

Der System Core enthält keine Gerätetreiber oder Betriebssystemdienste, wie z. B. einen Netzwerk- oder Filesystemdienst. Diese sind alle außerhalb des System Core in eigenen Benutzerprozessen realisiert. Zugriff auf die Hardware erhalten die Treiber über das System API.

Bevor der Treiber auf Systemressourcen (z. B. Interrupt, I/O-Port) zugreifen kann, muss er diese über das System API reservieren. Bei einer Reservierung entscheidet der Prozess-Manager, ob der Treiber Zugriff auf die entsprechende Ressource erhält. Falls die Ressource bereits belegt bzw. nicht als *shared* reserviert ist oder der Treiber nicht die dafür nötigen Rechte besitzt, wird der Zugriff vom Prozess-Manager abgelehnt. Sobald dem Treiber die Reservierung genehmigt wurde, kann er ebenfalls über Funktionen des System API darauf zugreifen (z. B. auf Interrupt warten, I/O-Ports lesen/schreiben).

Treiber können ihre Funktionalität auf drei Arten anderen Prozessen zur Verfügung stellen. Üblicherweise ist der Treiber als Bibliothek in einen Dienstprozess eingebunden, auf

den andere Prozesse per IPC zugreifen. Eine weitere Möglichkeit ist die direkte Integration in einen Anwendungsprozess, der diesen Treiber exklusiv verwendet (z. B. Sicherheitsarchitektur, Spezialtreiber). Für kritische Systemumgebungen ist auch die Implementierung eines Treibers als einzelner Prozess vorgesehen, der in diesem Fall seine Funktionen direkt per IPC zur Verfügung stellt. Dienste, die einen solchen Treiber verwenden, bleiben von dessen möglichen Fehlfunktionen unbeeinträchtigt.

2.3 Anwendungen

Anwendungen sind in Symobi Prozesse, die nicht als Dienste fungieren sondern in der Regel als Clients andere Dienste nutzen. Für die Dienste, die Anwendungen zur Verfügung stehen, gibt es unter Symobi verschiedene Möglichkeiten. Durch seinen klaren modularen Aufbau ist Symobi frei erweiterbar. Treiber und Dienste können unabhängig erstellt, verbessert, erweitert und ersetzt werden. Für diese freie Gestaltbarkeit des Betriebssystems wird der Begriff *Open-System-Architecture* eingeführt. Damit können neben den bestehenden auch eigene Treiber und Dienste implementiert werden. Diese sind frei gestaltbar, so dass auch komplette Subsysteme hinzugefügt werden können. Ein Beipiel hierfür ist die im Rahmen einer Diplomarbeit umgesetzte Java-VM für Symobi [2].

Die Anwendungen werden selbst in eingebetteten Systemen immer umfangreicher und komplexer. Um die objektorientierte Entwicklung unter Symobi zu erleichtern, bietet das System API eine auf Reference Counting basierende Garbage Collection an, die echtzeitfähig und weitgehend transparent ist, d. h. einzig bei der Deklaration von Variablen ist eine abweichende Syntax erforderlich.

2.4 Interprozesskommunikation

Da die Prozesse in Symobi getrennte, geschützte Adressräume besitzen, werden Daten zwischen Prozessen per IPC ausgetauscht. Durch die Trennung der Treiber und Dienste in eigene Prozesse ist eine effiziente IPC nötig. Als grundlegender IPC Mechanismus wird das Client/Server-Modell verwendet. Prozesse, die einen oder mehrere Dienste anbieten, werden als Server während Prozesse, die Dienste nutzen, als Clients bezeichnet werden. In einem Server öffnet jeder Dienst mit Hilfe des System API mindestens einen Kanal (Channel). Dieser ist entsprechend dem Client/Server-Modell synchron und bidirektional. Jeder Kanal besitzt eine systemweit eindeutige ID, über die auch Threads aus anderen Prozessen eine Verbindung (Connection) zu diesem Kanal aufbauen können. Der Dienst kann jedoch entscheiden, ob er eine Verbindung mit einem Client eingehen will oder nicht. Die Entscheidung darüber kann dynamisch und situationsabhängig zur Laufzeit erfolgen. Sobald ein Client eine Verbindung zum Server erhalten hat, kann er über Nachrichten Anfragen an den jeweiligen Dienst senden. Während der Dienst die Anfrage bearbeitet, blockiert der Sende-Thread im Client. Sobald der Dienst das Ergebnis als Nachricht über die selbe Verbindung zurückgesendet hat, wird der Sende-Thread wieder geweckt und kann das Ergebnis verarbeiten.

Der IPC-Mechanismus in Symobi lässt es zu, dass ein Prozess gleichzeitig Client und Server ist. Dabei bietet der Prozess über einen oder mehrere Kanäle einen oder mehrere Dienste an und enthält zusätzlich Threads, die als Clients

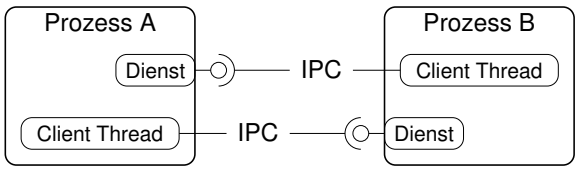

Abbildung 2: Die IPC unter Symobi. Die Kreise an den Prozessen sind Kanäle, an die sich Client Threads per IPC anbinden können.

Verbindungen zu anderen Servern haben. Es ist sogar möglich, dass zwei Prozesse jeweils einen Dienst anbieten, zu dem sie sich gegenseitig verbinden. Diese Konfiguration ist in Abbildung 2 dargestellt, wobei die Kreise die Kanäle der Prozesse sind, an die sich Clients anbinden können. Dabei können mehrere Threads, auch aus unterschiedlichen Prozessen, gleichzeitig mit einem Kanal verbunden sein.

Zusätzlich bietet Symobi *Managed Direct Memory* (MDM) an, das unter anderem auch zusammen mit der IPC für den Datenaustausch zwischen Prozessen genutzt werden kann. Das MDM folgt dem Prinzip eines Zero-Copy-Mechanismus. Allerdings bietet das MDM von Symobi im Gegensatz zu anderen Implementierungen zusätzlich einen umfassenden Zugriffsschutz. Per MDM können insbesondere große Datenmengen zwischen Prozessen noch effizienter ausgetauscht werden als über IPC-Nachrichten. Der IPC-Mechanismus wird trotzdem für die Synchronisation benötigt, so dass MDM kein alternativer Kommunikationsmechanismus ist, sondern nur eine Alternative für den reinen Datenaustausch zwischen Prozessen.

3. ANFORDERUNGEN

Nachdem bis hierher die Architektur von Symobi vorgestellt wurde, erläutert dieses Kapitel, wie Symobi die Anforderungen an ein Betriebssystem für eingebettete Systeme erfüllt. Dabei sind viele der Ziele konträr. So verliert beispielsweise ein zuverlässiger Microkernel an Effizienz [4]. Die Architektur von Symobi ist darauf ausgerichtet, eine optimierte Verbindung aller dieser Ziele zu erreichen.

Durch die im System Core umgesetzten Strategien und spezielle Mechanismen maximiert Symobi die Leistung einzelner Systemeigenschaften ohne andere, teilweise konträre Eigenschaften zu stark zu beeinträchtigen. Dabei steht die Zuverlässigkeit an erster Stelle, noch vor der Effizienz.

Im nachfolgendem werden die Ziele Zuverlässigkeit, Echtzeitfähigkeit, Portierbarkeit und Skalierbarkeit betrachtet. Wie ein transparenter Kommunikationsmechanismus und Benutzerinteraktion in Symobi realisiert werden, ist in den Kapiteln 3.5 und 3.6 beschrieben. Abschließend wird die Möglichkeit diskutiert, wie ein Energiemanagement in Symobi implementiert werden kann.

3.1 Zuverlässigkeit

Einer der wichtigsten Anforderungen an eingebettete Systeme ist deren Zuverlässigkeit und Robustheit, da sie in Umgebungen eingesetzt werden, in denen ein Fehlverhalten

zu körperlichem, materiellem oder wirtschaftlichem Schaden führt [1]. Dafür ist ein zuverlässiges Betriebssystem notwendig. In Symobi ist bereits durch den Einsatz einer Microkernel-Architektur eine höhere Zuverlässigkeit gewährleistet [4]. Die Architektur des Microkernels in Symobi enthält anerkannte Prinzipien, die die Zuverlässigkeit eines Microkernels erhöhen [5]. Durch die Auslagerung des Prozess-Managers aus dem Microkernel ist das Prinzip der Trennung des Mechanismus von der Strategie implementiert. Ebenfalls erhalten die Komponenten in Symobi die jeweils niedrigste Privilegstufe, die sie für die Erfüllung ihrer Aufgaben benötigen, was als ein weiteres Prinzip bekannt ist [5]. Dies ist in Symobi dadurch umgesetzt, dass alle Treiber und Dienste in eigenen Benutzerprozessen laufen und nur der Microkernel im privilegierten Modus des Prozessors arbeitet.

Weil keine Treiber und Dienste im Microkernel enthalten sind, führen fehlerhafte Gerätetreiber nicht zum Absturz des gesamten Systems, sondern nur zum Absturz des jeweiligen Prozesses. Falls ein Treiber abgestürzt ist, kann der jeweilige Prozess aus dem System entfernt und der Treiber in einem neuen Prozess wieder gestartet werden. Da in einem monolithischen Betriebssystem die häufigste Ursache für einen Systemabsturz fehlerhafte Treiber sind [6], wird dadurch in Symobi eine wesentliche Fehlerquelle ausgeschaltet.

In Symobi resultiert die grundlegende Zuverlässigkeit des Betriebssystems auch aus einer geringen absoluten Größe des Microkernels. Mit nur ca. 10-15 Tausend Codezeilen, die im privilegierten Modus des Prozessors ausgeführt werden enthält der Microkernel schon rein statistisch weniger mögliche Fehler als größere Kernel [7]. Außerdem ermöglichen der geringe Codeumfang und die Tatsache, dass der Code im Microkernel unveränderlich ist, prinzipiell auch eine manuelle Codeinspektion.

3.2 Echtzeitfähigkeit

Da eingebettete Systeme häufig zur Steuerung technischer Prozesse eingesetzt werden, muss das dafür verwendete Betriebssystem echtzeitfähig sein. Der Microkernel in Symobi ist ein hart-echtzeitfähiger, reaktiver Microkernel ohne eigene Threads, wofür unter anderem folgende Eigenschaften relevant sind.

Das auf Prioritäten basierende Echtzeit-Scheduling des Microkernel sorgt dafür, dass keine echtzeitkritischen Threads durch andere Threads unterbrochen werden. Dafür werden die 32 Prioritäten in zwei Gruppen eingeteilt. Die niedrigen Prioritäten sind für nicht echtzeitkritische Aufgaben vorgesehen, die nach dem Round Robin Verfahren unter Einbeziehung ihrer Priorität gescheduled werden. Dabei werden sie nach Ablauf ihrer Zeitscheibe unterbrochen, so dass auch Threads mit niedrigerer Priorität aktiv werden können. Die zweite Gruppe mit den höheren Prioritäten sind für echtzeitkritische Threads reserviert. Threads mit diesen Prioritäten können nur von Threads mit einer höheren Priorität unterbrochen werden. Dabei ist der Programmierer selbst dafür zuständig, die Prioritäten in den von ihm erstellten Treibern, Diensten und Anwendungen so zu vergeben, dass die Deadlines zeitkritischer Threads eingehalten werden. Wenn bespielsweise zwei Threads die höchste Priorität erhalten, kann dennoch immer nur einer davon tatsächlich in Echtzeit agieren.

Die Garbage Collection, die das System API anbietet, ist ebenfalls echtzeitfähig. Durch den Mechanismus des Reference Countings wird kein zusätzlicher, asynchron ablaufender Garbage Collection Thread benötigt, dessen Einfluss auf das Echtzeitverhalten nicht vorhersagbar wäre. Die Operationen des Reference-Counting-Mechanismus werden augenblicklich, inline und synchron im jeweiligen Thread ausgeführt, so dass das Echtzeitverhalten unverändert bleibt.

3.3 Portierbarkeit

In Bereich eingebetteter Systeme finden viele verschiedene Prozessorplattformen Verwendung. Ein Betriebssystem für eingebettete Systeme sollte deshalb sehr leicht portierbar sein, damit es auf verschiedenen Plattformen lauffähig ist. Zur Zeit läuft Symobi auf der x86, ARM/XScale und PowerPC Plattform.

Die einfache Protierbarkeit des Microkernels wird durch den enthaltenen Nanokernel erreicht. Da nur der Nanokernel, der die Abstraktion der prozessornahen Hardware bildet, durch seine enthaltenen Assembler-Routinen bei einer Portierung des System Cores angepasst werden muss, ist der System Core leicht portierbar. Der restliche Microkernel und der Prozess-Manager sind in C bzw. C++ geschrieben und bauen auf den Nanokernel bzw. dem Microkernel auf. Damit ist bei einem für die Zielplattform vorhandenen C/C++-Compiler der Portierungsaufwand für den restlichen Microkernel und den Prozess-Manager minimal.

Allerdings muss der Nanokernel auf der neuen Plattform die Schnittstelle zwischen Nanokernel und Microkernel umsetzen. Dennoch ist der Gesamtaufwand für eine Portierung wesentlich niedriger als bei einer Portierung des gesamten Microkernels.

Die Portabilität der Treiber in den Benutzerprozessen ist je nach Programmierung des Treibers unterschiedlich. Wenn sie keinen Assembler Code sondern nur reinen C/C++-Code enthalten, bringt die Portierung auf die neue Hardware keine Schwierigkeiten mit sich, da der Treiber mit einem vorhandenen C/C++-Compiler für die neue Plattform ohne Codeänderungen erzeugt werden kann. Allerdings müssen möglicherweise andere Systemressourcen wie Interrupts oder I/O-Ports reserviert werden. Dafür sind geringfügige Anpassungen im Quelltext notwendig.

Für Dienste, die nur auf anderen Treibern und dem System API aufbauen, sind keine weiteren Anpassungen an eine neue Plattform notwendig. Sie müssen nur mit einem entsprechenden Compiler für die neue Plattform übersetzt werden.

3.4 Skalierbarkeit

Da eingebettete Systeme im Allgemeinen sehr unterschiedliche Größen haben können, sollte ein dafür konzipiertes Betriebssystem sehr gut skalierbar sein, um die entsprechenden Anforderungen zu erfüllen. Die Systeme variieren in der Rechenleistung des Prozessors, in der Größe des Arbeits- und des persistenen Speichers und in der Anzahl und Art der angeschlossenen Geräte.

Symobi ist durch seinen Microkernel, der nur ca. 45 KB umfasst, und dem Prozess-Manager, der eine Größe von

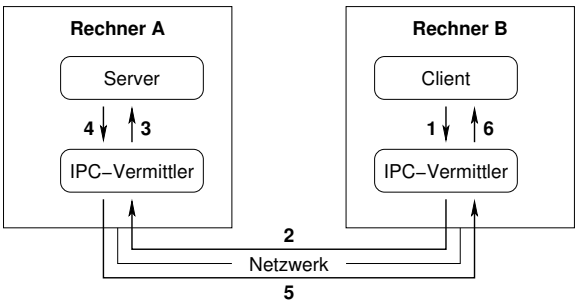

Abbildung 3: Für die Kommunikation über Netzwerk ist der IPC-Vermittler notwendig, der die IPC Nachrichten über das zugrunde liegende Netz schickt.

ca. 55 KB hat, mit einer Gesamtgröße von ca. 100 KB auf der x86 Plattform sehr schlank. So läuft der System Core selbst bereits auf kleinsten eingebetteten Systemen. Da die Treiber und Dienste außerhalb des System Core laufen, können diese je nach Anforderung des Systems hinzugefügt werden. Der gesamte Speicherbedarf von Symobi richtet sich dann nach Größe und Anzahl der gestarteten Treiber, Dienste und Anwendungen.

Darüber hinaus ist Symobi auch zur Laufzeit sehr gut skalierbar. So können Treiber oder Dienste zu jedem Zeitpunkt gestartet oder beendet werden. In kleinen Systemen können somit die knappen Ressourcen je nach Anfrage und Auslastung unterschiedlich reserviert werden. So kann ein Dienst zusammen mit einem möglicherweise notwendigen Treiber erst gestartet werden, wenn dessen Funktionalität gebraucht wird. Wenn dieser nicht mehr benötigt wird, kann er wieder mit dem Treiber beendet werden, um die reservierten Systemressourcen freizugeben.

3.5 Externe Kommunikation

In eingebetteten Systemen spielt die Kommunikation nach außen eine zunehmende Rolle. So findet vor allem in mobilen Endgeräten eine Kommunikation untereinander statt. Für den Informationsaustausch sind diese Endgeräte häufig auch mit einer größeren Infrastruktur oder Basisstation verbunden.

Der IPC-Mechanismus unter Symobi erlaubt prinzipiell auch eine Kommunikation über Rechnergrenzen hinweg. Da sich ein Client mit Hilfe einer ID zu einem Kanal verbindet, kann sich der Server nicht nur auf dem lokalen Knoten, sondern auch auf einem entfernten Knoten im Netzwerk befinden. Dies ist für den Client transparent, da er das System API in beiden Fällen auf die gleiche Weise verwendet. Aus Sicht des Servers erstellen Clients Verbindungen zu dessen Kanälen, über die die Clients ihre Nachrichten schicken. Ob die Verbindung lokal oder über das Netzwerk aufgebaut wird, bleibt dem Server dabei verborgen. Damit können Client und Server mit dem IPC-Mechanismus unter Symobi über ein Netzwerk kommunizieren, ohne dass eine Programmänderung im Client oder Server notwendig ist.

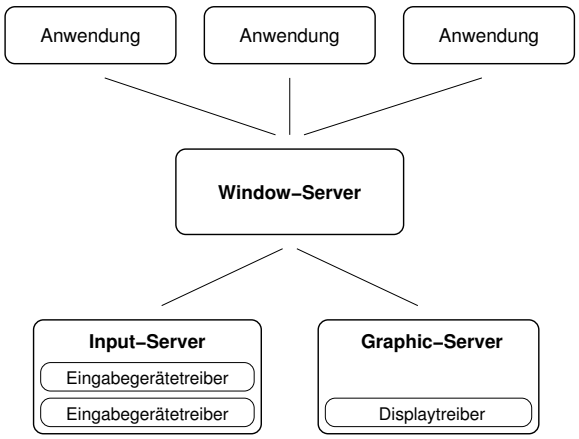

Abbildung 4: Der Input- und Graphic-Server stellt die grundlegenden Operationen für die Benutzerein- und Ausgabe in der Standardkonfiguration bereit. Auf diesen baut der Window Server auf, der den Anwendungen ein graphisches Fenstersystem liefert.

Allerdings muss dafür ein zusätzlicher Dienst, der sogenannte IPC-Vermittler, in das System eingebracht werden, der für das Auffinden der Dienste im Netzwerk und das Versenden der Nachrichten über das Netzwerk zuständig ist [4]. Dieser Zusammenhang ist in Abbildung 3 illustriert. Der IPC-Vermittler ist wie jeder andere Dienst ein Benutzerprozess. Er bietet lokal im Rechner als Server alle Dienste an, die er auf anderen Knoten im Netzwerk findet. Zusätzlich stellt er einen Namensdienst bereit, über den die Clients Kanal-IDs von Diensten im Netzwerk erfragen können. Anschließend verbinden diese sich lokal über IPC mit dem IPC-Vermittler, der die Anfrage über das zugrunde liegende Netz an den IPC-Vermittler des entfernten Knotens schickt. Dieser wandelt die Anfrage wieder in eine lokale IPC-Nachricht um, und übergibt diese dem Server. Die Antwort des Servers wird über den selben Weg zurück an den Client gesandt.

In welchem Netzwerk, die beiden Rechner verbunden sind, ist für den Client- und Serverprozess ebenfalls transparent. Nur der IPC-Vermittler muss für die oben beschriebene Kommunikation wissen, welches Netz verfügbar ist. Wenn mehrere Pfade in unterschiedlichen Netzen für die Verbindung von Client und Server zur Verfügung stehen, ist es auch möglich, das jeweils am besten geeignete zu verwenden. So ist es beispielsweise in mobilen System mit drahtloser Kommunikation häufig der Fall, dass verschiedene Kommunikationsarten mit unterschiedlichen Bandbreiten vorhanden sind. So ist es denkbar, dass in einem mobilen Endgerät die Kommunikation über WLAN, Bluetooth, GSM oder UMTS möglich ist. Auch bei einem Verbindungsabbruch in einem Netz kann der IPC-Vermittler die Kommunikation in einem anderen Netz, in dem sich Server- und Client-Knoten befinden, fortführen.

3.6 Benutzerinteraktion

In eingebetteten Systemen findet meist Interaktion mit dem Benutzer statt, wofür es unterschiedliche Ein- und Ausgabe-

geräte gibt. Um den Anwendungen eine einheitliche Schnittstelle für die Benutzerein- und Ausgabe trotz der Vielfalt der Geräte zu bieten, nutzt Symobi das Konzept der Client/Server-Architektur. So stellt Symobi einen Input-Server für die Benutzereingaben und einen Graphic-Server für eine graphische Oberfläche bereit. Die beiden Server greifen über die entsprechenden Treiber, die als Bibliotheksfunktionen in die Server eingebunden werden, auf die Ein- bzw. Ausgabegeräte zu. Auf den Input- und den Graphic-Server baut der Window-Server auf, der den Anwendungen ein grafisches Fenstersystem in einer einheitlichen Schnittstelle mit Maus- und Tastatureingaben anbietet. Die Abbildung 4 stellt die Realisierung der Benutzerinteraktion in Symobi schematisch dar.

Um die vielen unterschiedlichen Ein- und Ausgabegeräte, die es für eingebettete Systeme gibt, zu unterstützen, bietet die Open-System-Architecture von Symobi eine gute Grundlage. Es müssen lediglich die Treiber für die Ein- und Ausgabegeräte angepasst werden, wobei der Window-Server bei einer Änderung der Ein- oder Ausgabegeräte nicht verändert werden muss, da er seine Informationen aus dem Input- und Graphic-Server bezieht.

3.7 Energiemanagement

Eingebettete Systeme müssen in Umgebungen arbeiten, in denen nicht nur wenig Speicher und wenig Prozessorleistung vorhanden ist, sondern auch Energie nur in begrenzten Maße zur Verfügung steht. Deshalb müssen eingebettete Systeme sehr energiebewusst arbeiten, was von Betriebssystemseite koordiniert werden sollte. In diesem Bereich bietet Symobi durch seine modulare und skalierbare Architektur die Möglichkeit, Komponenten für das Energiemanagement problemlos hinzuzufügen. Das System API hat Funktionen, um die Prozessorauslastung abzufragen. Mit diesen Informationen können Gerätetreiber die Hardware entsprechend steuern. So ist es nicht nur möglich die Hardware für das Energiemanagement anzusteuern, sondern jeder Treiber könnte z. B. die Leistungsaufnahme des von ihm angesteuerten Geräts nach den Informationen über die Auslastung des Systems autonom regeln.

Der Microkernel selbst arbeitet ebenfalls energiebewusst. So hält er den Prozessor mit einem entsprechenden Befehl an, wenn alle Prozesse im System auf Ereignisse warten. Durch das zeitgesteuerte Schedulingverfahren wird der Prozessor mit einem Interrupt wieder geweckt, sobald ein Prozess rechenbereit wird.

4. ZUSAMMENFASSUNG

In eingebetteten Systemen sind unterschiedliche Anforderungen an ein Betriebssystem vorhanden, die zum Teil konträr sind. Das embeddable RTOS Symobi vereinigt und erfüllt mit seinem Architektur-Konzept die Anforderungen hinsichtlich Zuverlässigkeit, Robustheit, Echtzeitfähigkeit, Portabilität und Skalierbarkeit. Auch zunehmend bedeutende Eigenschaften wie transparente Kommunikationsmechanismen und Energiemanagement werden von Symobi berücksichtigt. So wird mit dem Microkernel, der keine Treiber und Dienste enthält, eine zuverlässige Basis geschaffen. Zusammen mit dem Prozess-Manager entsteht daraus ein hochgradig skalierbares RTOS, dessen schlanker System Core den knappen Speicherressourcen in eingebetteten Systemen Rechnung trägt. Durch das Hinzufügen weiterer Treiber und Dienste kann Symobi für spezielle Anwendungen bzw. bestimmte Hardware-Plattformen angepasst werden. Seine Echtzeitfähigkeit erreicht Symobi durch einen reaktiven Microkernel mit einem auf Prioritäten basierende Echtzeitscheduling. Die IPC in Symobi sorgt für eine transparente Kommunikation, bei der Client und Server davon unabhängig sind, ob sich der Kommunikationspartner lokal oder auf einem entfernten Knoten im Netzwerk befindet. Im Bereich des Energiemanagements, dessen Rolle in eingebetteten Systemen in Zukunft noch weiter zunehmen wird, bietet Symobi ebenfalls mit seiner modularen Architektur eine solide Grundlage, die jetzt noch unbekannten Energiefunktionen der zukünftigen Hardware flexible und ohne die Anpassung des Microkernels in das Gesamtsystem zu integrieren.

5. LITERATUR

[1] Calvez, J. P.: *Embedded real-time systems*. John Wiley & Sons, Inc., New York, NY, USA, 1993. Translator-Alan Wyche and Translator-Charles Edmundson.

[2] Dörfel, R.: *Konzeption und Implementierung einer Java VM für μnOS*. Diplomarbeit, Technische Universität München, 2006.

[3] Kopetz, H.: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[4] Liedtke, J.: *On micro-kernel construction*. SIGOPS Oper. Syst. Rev., 29(5):237–250, 1995.

[5] Shapiro, J. und N. Hardy: *EROS: a principle-driven operating system from the ground up*. Software, IEEE, 19(1):26–33, 2002.

[6] Swift, M. M., B. N. Bershad und H. M. Levy: *Improving the reliability of commodity operating systems*. In: *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, S. 207–222, New York, NY, USA, 2003. ACM Press.

[7] Tanenbaum, A. S., J. N. Herder und H. Bos: *Can We Make Operating Systems Reliable and Secure?*. Computer, 39(5):44 – 51, Mai 2006.