VxWorks - Aufbau und Programmierung

Dominik Meyer <dmey@informatik.uni-kiel.de>
AG Echtzeitsysteme / Eingebettete Systeme
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Zusammenfassung

In dieser Ausarbeitung stelle ich das Programmieren und das Entwickeln von Software unter Echtzeitbetriebssystemen vor und gehe dabei auf Features, Entwicklungsumgebungen und allgemeines Arbeiten ein. Das Hauptaugenmerk lege ich dabei auf das Echtzeitbetriebssystem VxWorks der Firma WindRiver.

1. EINFÜHRUNG

1.1 Allgemeines

1.2 Was ist VxWorks

Im Allgemeinen wird mit VxWorks ein Echtzeitbetriebssystem (RTOS) der Firma Wind River (`www.windriver.com`) bezeichnet. VxWorks ist aber eigentlich ein Toolkit, aus dem sich ein RTOS erzeugen läßt. Das Toolkit enthält eine Entwicklungsumgebung für die Desktop-Betriebssystem Windows, Unix und Linux sowie einen Betriebssystem-Kern,den WIND Kernel, ein einfaches Ein- und Ausgabe-System und einen vollständigen TCP/IP-Stack. Weitere Module sind von Wind River und anderen Firmen erhältlich und erweitern VxWorks um viele Möglichkeiten. Wie alle modernen Betriebssysteme ist VxWorks in Schichten aufgebaut (siehe Abbildung 1). Die unterste Schichte bildet dabei die Hardwarenahe- oder Treiberschicht. Nur in dieser Schicht interagiert das Betriebssystem direkt mir der Hardware. Diese werde ich aber nicht weiter betrachten, da diese nur für System-Programmierer aber nicht für Applikation-Programmierer interessant ist. Die Treiberschicht legt auch fest, auf welchen Prozessor Architekturen das Betriebssystem lauffähig ist. Für VxWorks sind das ARM, MIPS, PowerPc, SuperH, Intel Pentium und Intel XScale. Über der Treiberschicht befindet sich der WIND Kernel, als das Herz des Betriebssystems. Bei VxWorks besteht der Kern aus einem Microkernel, der den Scheduler,die System-Uhr und die Semaphoren-Verwaltung zur Verfügung stellt. Robust definierte Schnittstellen ermöglichen es dem Rest des Betriebssystems auf die Dienste des Microkernels zuzugreifen. Im 2. Abschnitt dieser Ausarbeitung wird der Microkernel etwas genauer betrachtet und die für einen Programmierer wichtigen Funktionen beschrieben. Als Schale um den Microkernel sind andere wichtige Betriebssystem-Dienste angeordnet (siehe Abbildung 1):

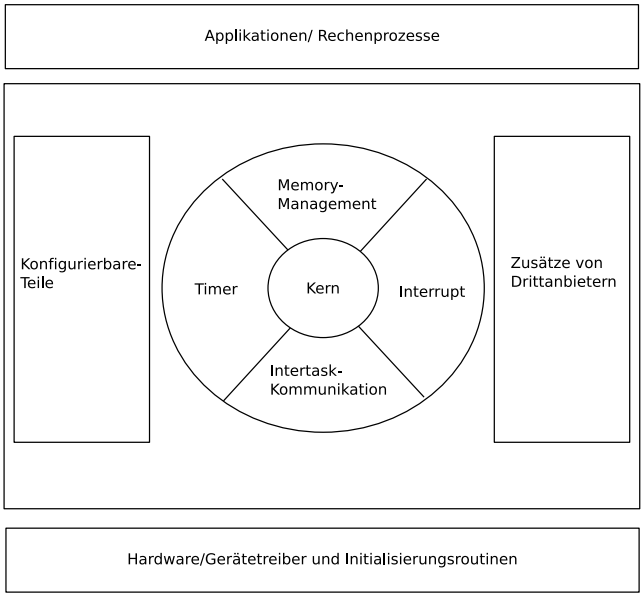

Abbildung 1: Kernel Struktur

1. Speicher-Management
2. Timer
3. Interrupt-Verwaltung
4. Intertaskkommunikation

Auf das Speicher-Management und die Intertaskkommunikation wird in in den Abschnitten 3 nitt 4 eingegangen und einen kurzer Einblick in die Funktionsweisen dieser Dienste gegeben.
Für ein Echtzeitbetriebssystem ist die Abwicklung von Interrupts sehr wichtig. Deshalb wird in Abschnitt 2.4 kurz auf diese Eingegangen. Weiter ist in Abbildung 1 zu erkennen, wie die Anwendungsebene von den Systemkomponenten abgekoppelt ist. Innerhalb der Systemebene gibt es zusätzlich zu den um den Kern angeordneten Betriebssystem Diensten, noch weitere, die von Dritt-Anbietern zur Verfügung gestellt werden. Diese stellen dann Funktionen zur Verfügung, die bei speziellen Anwendungen benötigt werden. Außerdem ist die bereits erwähnte Abkopplung der geräte- und architekturspezifischen Betriebssystem-Komponenten zu erkennen. Dies ermöglicht eine Einfache Anpassung von VxWorks auf eine andere Prozessor-Architektur und andere Hardwarekomponenten.
Anschnitt 6 befaßt sich mit dem Software-Entwurf unter

VxWorks. Dabei wird die Entwicklungsumgebung *Tornado* kurz vorgestellt, die beim Kauf von VxWorks mitgeliefert wird. Zusätzlich werden einige Beispiele zur Programmierung unter VxWorks gezeigt.

2. KERNEL

In diesem Abschnitt befasse ich mich mit dem WIND Kernel. Um auf Scheduling und Semaphoren näher eingehen zu können, stelle ich als erstes den Task, die kleinste organisatorische Einheit, von VxWorks vor. In einem späteren Abschnitt gehe ich dann auf POSIX Threads und Scheduling ein.

2.1 Task

Definition eines Tasks:
Ein Task ist eine organisatorische Einheit, die aus ausführbaren Programmcode, eigenem Speicher und Variablen besteht und ist gekennzeichnet durch eine Priorität und einen Taskzustand[7]

Genau dies trifft auch auf VxWorks zu. Die Tasks bilden in VxWorks die Einheiten, die vom Scheduler mit Systemresourcen versorgt werden können. Sie bilden die Einheiten, die in den Desktop-Betriebssystemen die Prozesse sind. Sie ähneln aber in ihrem Aufbau eher den Threads. Ein Task enthält folgende Dinge:

1. den Programmcounter (PC) des Prozessors für diesen Task vor dem letzten Task-Wechsel
2. den Inhalt der CPU Register für diesen Task vor dem letzten Task-Wechsel
3. die Speicher-Adresse des Stacks
4. die Festlegung der Ein- und Ausgabegeräte für die standard Ein- und Ausgabe und für die Fehler-Ausgabe
5. einen Verzögerungstimer
6. einen Zähler für die Zeit, die in einem Time-Slice bereits verbraucht ist
7. einige Kernel-Kontrollstrukturen
8. Signal-Handler
9. einige Werte zum Debuggen und für PerformanceMessungen

VxWorks arbeitet, wie die meisten RTOS, mit einem *flat memory model*, daß bedeutet, daß alle Tasks in einem großen Speicherbereich laufen. Sie können also auf den Speicherbereich der anderen Tasks zugreifen. Auf diesen Punkt wird im Abschnitt 4 über Intertaskkommunikation noch eingegangen. Genaueres über das Speicher-Management und wie Speicherbereiche geschützt werden können wird in Abschnitt 3 erklärt.
Task nehmen immer einen bestimmten Zustand im System an. Dieser Zustand hängt davon ab, was der Scheduler gerade für diesen Prozess vorgesehen hat. In VxWorks kann eine Task folgende Zustände annehmen:

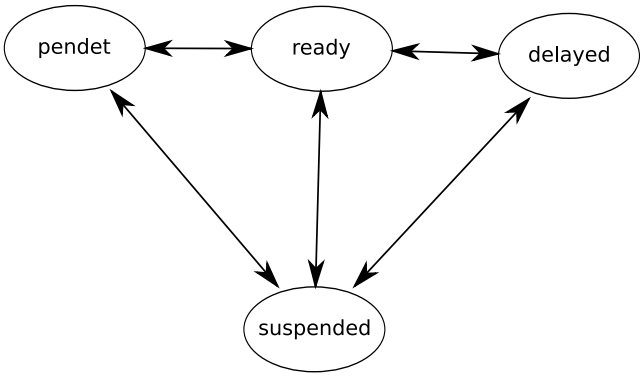

Abbildung 2: Taskzustände in VxWorks

- der Task wartet nur auf die CPU (READY)
- der Task wartet auf eine andere Ressource als die CPU (PEND)
- der Task schläft für einen Zeitraum (DELAY)
- der Task steht nicht zur Ausführen bereit, dieser State dient zu Debug Zwecken (SUSPEND)
- der Task wartet auf eine Ressource außer der CPU und besitzt einen Timeout für das Warten (PEND + T)

Die möglichen Taskstatewechsel sind in Abbildung 2 abgebildet. Im nächsten Abschnitt wird der Scheduler von VxWorks behandel. Dort werden sich diese Taskstates dann wiederfinden.

2.2 Scheduler

Da VxWorks ein Multitasking Betriebssystem ist, kommt der Kernel um einen Scheduler nicht herum. Der Scheduler ist das Herz jedes Multitasking-Betriebssystems. Er teilt jedem Task des Systems Ressourcen zu und legt fest wie viel Rechenzeit ein Prozess an einem Stück bekommt.
VxWorks unterstützt dabei zwei Scheduler-Algorithmen:

Priority Preemptive Scheduling Mit dieser Strategie vergibt der Benutzer, bei der Entwicklung des Systems, Taskprioritäten, die die Wichtigkeit der Aufgabe der Tasks widerspiegeln. Dem Entwickler stehen dabei Prioritäten von 1 bis 255 zur Verfügung, wobei 1 die höchste Priorität beschreibt und 255 die niedrigste. Während der Laufzeit des Systems wählt der Scheduler immer den Prozess mit der höchsten Priorität aus und gibt diesem Rechenzeit. Ein Taskwechsel zu einem anderen Task findet nur statt, wenn der aktive Prozess auf eine Ressource oder ein Ereignis wartet. Dann wird dem Prozess, mit der nächsthöheren Priorität, Rechenzeit zugewiesen. Wird ein Prozess mit einer höheren Priorität durch das Freiwerden einer Ressource aufgeweckt und hat der aktive Prozess eine niedrigere Priorität, wird ein Taskwechsel ausgeführt und der vorher wartende Prozess bekommt Rechenzeit und Zugriff auf die freien Ressourcen.

Round Robin Mit dieser Strategie wird jedem Prozess gleichmäßig viel Rechenzeit der CPU zur Verfügung gestellt. Alle Tasks besitzen die gleiche Priorität und bekommen immer eine feste Anzahl von CPU Zyklen zugewiesen.

Zusätzlich zu diesen beiden Scheduling-Algorithmen lassen sich in VxWorks daraus auch Misch-Systeme bilden. Dabei werden Gruppen von Tasks gebildet. Jede dieser Gruppe bekommt nach der Wichtigkeit dieser Gruppe ihre Prioritäten zugewiesen. Da nun die Tasks einer Gruppe die selbe Priorität besitzen, werden diese Tasks mit Round-Robin gescheduled und die Auswahl der Gruppen geschieht über Priority-Preemptive Scheduling.
Um in das gesamte System Dynamik zu bringen, damit auf Veränderungen der Umwelt Anpassungen vorgenommen werden könnten, gibt es verschiedene Möglichkeiten den Scheduler zu beeinflussen.

1. Bei Round-Robin Scheduling läßt sich die Zeit, die einem Task zur Verfügung stehen verändern. Dadurch erreicht man unter Umständen eine bessere Reaktionsfähigkeit des Systems, da die Tasks nun häufiger an der Reihe sind.

2. Bei Priority-Preemptive Scheduling lassen sich zur Laufzeit die Prioritäten von Tasks verändern. Wenn sich in der realen Welt die Wichtigkeit einer Berechnung verändert, kann diesem auch in VxWorks Rechnung getragen werden.

3. Zusätzlich zu diesem beiden Methoden, kann ein Task den Scheduler noch abschalten. Dadurch ist er der einzige Task der Zugriff auf die CPU erhält. Wird dieser Prozess durch das nicht Vorhandensein einer Ressource geblockt, wird der Task mit der höchsten Priorität ausgewählt, um Rechenzeit zu erhalten. Durch das An- und Abschalten des Scheduler läßt sich zum Beispiel *mutual exclusion* implementieren. Zu beachten ist nur, daß weiterhin Interrupts, den Task unterbrechen können.

2.3 Semaphoren

Wie jedes andere Betriebssystem stellt VxWorks auch Semaphoren zur Task-Synchronisation zur Verfügung. Folgende Semaphoren-Typen sind implementiert:

Binary Semaphore
Dies sind die schnellsten Semaphoren. Sie lassen sich für die verschiedensten Dinge einsetzen. Optimiert sind sie für *Mutual Exclusion* und Synchronisation von Tasks. Diese Art von Semaphoren ist eigentlich in jedem heute eingesetztem Betriebssystem zu finden.

Counting Semaphore
Diese Semaphoren sind ähnlich wie die binären Semaphoren. Zusätzlich zählen sie mit, wie oft eine Semaphore genommen worden ist. Sie sind optimiert um eine Ressource zu schützen, von der es eine bestimmte Anzahl gibt. Auch diese Semaphore ist heute oft in den gängigen Betriebssystemen zu finden.

Mutual Exclusion Semaphore
Diese Semaphore sind normalerweise nicht in anderen Betriebssystemen zu finden. Dort werden häufig Binäre-Semaphoren dafür verwendet, mutual exclusion zu implementieren. In VxWorks sind diese Semaphore auch benäre, besitzen aber folgende Zusatz-Eigenschaften:

- sie können nur für mutual exclusion verwendet werden
- eine Semaphore kann nur von dem Task, der sie genommen hat wieder zurückgegeben werden
- es kann nicht von einer *interrupt service routine* (ISR) verwendet werden
- das globale Freigeben aller auf diese Semaphore wartenden Tasks ist nicht möglich

Bei dem Einsatz von Semaphoren ergibt sich dann ein Problem, wenn ein Task aus dem System entfernt werden soll, aber dieser Task zu der Zeit noch eine Semaphore hält. Dies kann zum Deadlock der anderen Tasks führen und so das gesamte System zum Stillstand bringen. Um dies zu verhindern kann in VxWorks beim Erzeugen einer Semaphore ein Flag gesetzt werden, das dem Betriebssystem mitteilt, daß der Task, welcher eine Semaphore hält nicht gelöscht werden kann.

2.4 Interrupt Verwaltung

Interrupts müssen in Echtzeitbertriebssystemen sehr schnell abgehandelt werden. Damit das Betriebssystem nicht in einen Deadlock übergehen kann, müssen bestimmte Richtlinien bei der Programmierung von Interrupt-Handlern beachtet werden. In VxWorks werden diese Handler *Interrupt Service Routines*(ISR) genannt. Um schnelle Reaktionszeiten der ISR zu gewährleisten laufen diese in VxWorks in einem speziellen Kontext, außerhalb aller Tasks-Kontexte. Deshalb ist beim Aufruf einer ISR keine Task-Wechsel nötig. VxWorks stellt für alle gängigen Interrupts eigene ISR zur Verfügung. Bestehen aber eigene Anforderungen an einen Interrupt, so können diese in selbstgeschriebenen ISR erfüllt werden. Dabei sind aber einige wichtige Dinge zu beachten:

- Alle ISR benutzen denselben Interrupt Stack. Dieser Stack wurde bereits beim Start des Betriebsystems initialisiert. Dieser Stack muß groß genug sein, um den schlimmsten anzunehmenden Fall von verschachtelten ISR bedienen zu können. Außerdem sollte in den ISR's darauf geachtet werden, daß der Stack nicht unnötig belegt wird.

- Da eine ISR nicht in einem normalen Task-Kontext läuft bestehen bestimmte Einschränkung in dem zur Verfügung stehenden Funktionsumfang. Deshalb dürfen Funktionen, die die ISR zum Blocken bringt, nicht verwendet werden. Dazu zählen bestimmt IO-Funktionen, Semaphoren und Mutexes. Deshalb dürfen auch die Funktionen zum allozieren von Speicher und zum freigeben von Speicher verwendet werden, da diese zu den blocking-Funktionen zählen. Eine ISR darf auch nicht direkt über die Treiber mit der Hardware kommunizieren. Auch eine Benutzung eines mathematischen Coprozessors ist unter VxWorks innerhalb einer ISR nicht

möglich, da der Interrupt Treiber nicht die Register des Coprozessors sichert. Durch eine Benutzung würden dann Ergebnisse, auf die der Task, der unterbrochen wurde, verfälschen. Muß eine ISR unbedingt Funktionen des Coprozessors nutzen, muß der Programmierer dafür sorgen, daß die Register gespeichert und nach der Benutzung wieder zurückgesetzt werden.

- Zur Kommunikation zwischen ISR's und normalen Tasks können die in Abschnitt 4 vorgestellten Methoden verwendet werden.

3. MEMORY MANAGEMENT

Jedes Betriebssystem stellt auch Speicherverwaltungsroutinen zur Verfügung. Über die Speicherverwaltung können Tasks und Betriebssystem Speicher für Daten und Puffer anfordern. In den heutigen Desktop-Betriebssystem ist es üblich die Speicherverwaltung auf einem *Virtual Memory Interface*(VMI) zu implementieren. VMI bedeutet, daß der physikalische Speicher nicht direkt angesprochen wird, sondern daß es eine Abstraktionsschicht gibt, die es ermöglicht Auslagerungsdateien oder -partitionen zu verwenden. Diese Schicht sorgt dafür, daß zur Zeit nicht benötigter Speicher aus dem physischen Speicher in die Auslagerungsdateien oder -partitionen verschoben wird und daß der physische Speicher durch schnelleren Cache gepuffert werden kann. Viele RTOS verwenden dieses Prinzip nicht, da es dort meist keine große Unterstützung für Festspeicher-Medien gibt, auf die Speicher ausgelagert werden könnte. WindRiver hat das VMI Konzept aber aufgegriffen und in VxWorks implementiert. VxWorks stellt einmal eine Basisunterstützung von VMI zur Verfügung und zusätzlich in einem Modul eine erweitere Unterstützung.

3.1 Basic VMI

Diese Speicherverwaltungs- Routinen stehen immer unter VxWorks zur Verfügung. Sie sind direkt im Basis-Code des Betriebssystems enthalten. Die Basisunterstützung unter VxWorks sieht so aus, daß bei der Initialisierung des System, beim Booten, der Kernel einen Virtuellen Speicherbereich erzeugt. Dieser virtuelle Speicherbereich ist eine eins zu eins Abbildung vom physikalischem Speicher. In diesem virtuellem Speicherbereich werden alle später gestarteten Tasks ausgeführt. Die Tasks können untereinander auf den gesamten Speicherbereich dieser Virtuellen Einheit zugreifen und so auch den Programmcode von anderen Tasks verändern. Es gibt zum einen den gewollten Zugriff auf einen gemeinsamen Speicherbereich und zum anderen den ungewollten. Auf ersteren wird in Abschnitt 4 genauer eingegangen. Der ungewollte Fall kann aber mit diesen Basis-Funktionen nicht verhindert werden. Der Umfang der Basisfunktionalität hängt auch davon ab, ob die Hardware eine *Memory Management Uni*(MMU) zur Verfügung stellt. Mithilfe einer MMU kann das VMI Speicherbereiche, die z.B. von Geräten mit *Direkt Memory Access*(DMA) zugegriffen wird, automatisch vom Caching ausnehmen. Ist keine MMU vorhanden, muß der Programmierer selbst für die Cache-Updates sorgen, oder das Caching komplett deaktivieren. Durch das Deaktivieren des Caches wird die Gesamt-Performance des Betriebssystems aber schlechter.

3.2 VxVMI Modul

Dieses Modul kann zusätzlich zum Betriebssystem erworben werden. Es kann aber nur bei Vorhandensein einer MMU eingesetzt werden. Die Basis-Funktionen, wie eben beschrieben, bleiben erhalten. VxVMI benötigt eine MMU um zu verhindern, daß bestimmte Bereiche des Speichers überschrieben werden. Beim Start von VxWorks werden alle Text-Segmente, die den Programmcode des Betriebssystems enthalten, schreibgeschützt. Beim Start von weiteren Tasks werden deren Text-Segmente ebenfalls automatisch schreibgeschützt. Dadurch ist sichergestellt, daß andere Tasks, nicht den Programmcode von bereits laufenden Tasks verändern können. Zusätzlich wird beim Start von VxWorks auch die sich im Speicher befindende *Interrupt Vector Tabelle*(IVT), die die Sprungadressen enthält, die bei dem Auftreten von Interrupts angesprungen werden, auf „nur lesend" gesetzt. Durch diese Maßnahmen wird die Zuverlässigkeit bedeutend erhöht. Zusätzlich zu diesen Erweiterungen bietet VxVMI die Möglichkeit, ganze Speicherbereiche vom Caching aus zunehmen oder sie ebenfalls, wie die Text-Segmente, gegen Schreibzugriffe zu schützen. Eine weitere, sehr wichtige Erweiterung ist *Private Virtual Memory*(PVM). PVM bietet die Möglichkeit, daß einzelne Tasks mehrere virtuelle Speicherbereiche erzeugen können, die sie nach belieben in den Auslagerungsspeicher verschieben können und wieder in den Speicher laden können. Jeder dieser virtuellen Speicherbereiche hat für die globalen Objekte des Systems, die gleichen virtuellen Speicheradressen, damit die Tasks weiterhin auf System-Objekte wie die Semaphoren zugreifen können. Mit den Basis-Funktionen ist es also möglich ganz normal zu arbeiten. Besteht aber in dem zu entwickelnden System das Bedürfnis nach mehr Zuverlässigkeit und Sicherheit, kommt man um die Nutzung des VxVMI Moduls zur Absicherung der Speicherbereiche nicht herum.

4. INTERTASKKOMMUNIKATION

Wie bereits erwähnt wurde ist VxWorks ein Multitasking-Betriebssystem. Deshalb ist es sehr wichtig, daß die parallel ausgeführten Tasks mit einander Kommunizieren können, um Aktionen und Zugriffe zu Synchronisieren. Dafür stehen in VxWorks mehrere Möglichkeiten zur Verfügung:

1. Shared Memory, zum Austausch simpler Daten
2. Message Queues, zum Austausch von Nachrichten innerhalb einer CPU
3. Pipes, zum Austausch von Nachrichten innerhalb einer CPU
4. Sockets und *Remote Procedure Calls*(RPC), zur Kommunikation über ein Netzwerk
5. Signale, für Fehler- und Exception-Handling
6. VxMessagePassing(VxMP) und VxFusion zum Austausch von Nachrichten über mehrere CPUs hinweg

Im folgenden wird nur auf die Verfahren eingegangen, die mit der Basis Version von VxWorks möglich sind und innerhalb einer CPU stattfinden. Das Thema „Sockets und RPC" ist zu umfassend, als daß es hier behandelt werden könnte.

4.1 Shared Memory

Dies ist die einfachste Art unter VxWorks Daten auszutauschen. Wie im Abschnitt 3 und 2.1 erwähnt wurde, besitzt VxWorks ein flat memory model und alle Tasks können auf den Speicherbereich der anderen Tasks zugreifen. Durch die-

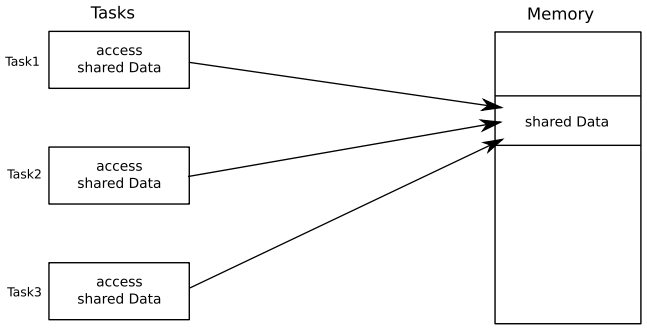

Abbildung 3: Shared Data Structure taken from VxWorks Programmers Guide

se Methode können globale Variablen, lineare Puffer, Ring-Puffer und linked Lists und Zeiger direkt von den verschiedenen Task-Kontexten referenziert werden. Wenn man diese Methode wählt, muß aber darauf geachtet werden, daß auf den gemeinsamen Speicherbereich nicht gleichzeitig geschrieben wird oder gleichzeitig geschrieben und gelesen wird. Um dies zu gewährleisten können die Semaphoren aus Abschnitt 2.3 eingesetzt werden.

4.2 Message Queues

Heutige Echtzeit-Applikationen werden aus unabhängigen Tasks zusammengesetzt, die aber zum Erfüllen ihrer Aufgabe eng zusammenarbeiten. Zur Synchronisation solcher Tasks wurden bereits Semaphoren vorgestellt. Zusätzlich zur Synchronisation müssen aber oft auch kurze Nachrichten ausgetauscht werden. Dies ist zum Beispiel bei verteiltem Rechnern notwendig, um Aufgaben zu verteilen und die Ergebnisse der Berechnungen wieder zusammenzubringen. Für solche Aufgaben eignen sich Message Queues. Message Queues bilden Kanäle zwischen zwei oder auch mehreren Tasks. Unter VxWorks arbeiten diese Kanäle nach dem FIFO Prinzip. Zusätzlich dazu gibt es die Möglichkeit eine Nachricht als URGENT zu markieren. Diese Nachricht wird dann am Kopf der Message Queue eingereiht und nicht am Ende. Zusätzlich zu den Prioritäten kann beim Versenden oder Empfangen auch ein Timeout angegeben werden. Beim Versandt bedeutet der Timeout, wie viele Ticks der Tasks auf das Freiwerden eines Platzes in der Queue wartet. Beim Empfang gibt der Timeout die Anzahl der CPU Ticks an, die der Task warten soll, wenn initial keine Nachricht in der Queue war. Der Timeout kann auch so gewählt werden, daß der Task gar nicht wartet oder daß er für immer wartet. Wird eine Bidirektionale Verbindung zwischen Tasks gewünscht wird dieses, wie in Abbildung 4 gezeigt, mit zwei Message Queues implementiert. Es können auch mehrere Tasks gleichzeitig auf die gleiche Message Queue lesend oder schreibend zugreifen. Dabei geht das Empfangen der Nachricht nach dem Prinzip „wer zuerst kommt bekommt die Nachricht". Es wird also nicht automatische sichergestellt, daß alles Tasks die verschickte Nachricht bekommen. Ist so ein Verhalten gewünscht, muß die Kommunikation über eine Server/Client

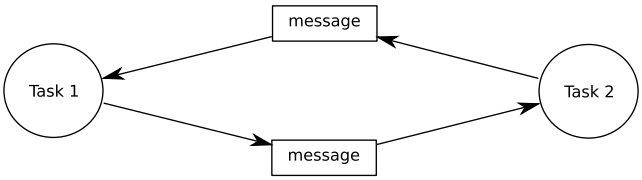

Abbildung 4: Full Duplex Message Queue taken from VxWorks Programmers Guide

Struktur erfolgen. Dabei dient ein Task als Server. Dieser empfängt von allen Clients über eine Message Queue Nachrichten. Diese Nachrichten werden dann über je eine Message Queue zu jedem einzelnen Client weiter verschickt. Siehe dazu Abbildung 5.

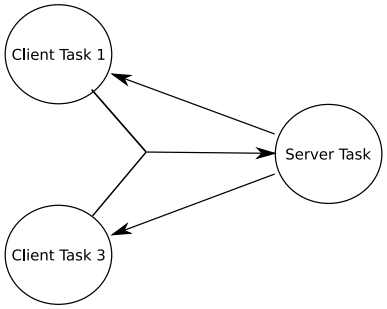

Abbildung 5: Server Client Struktur

4.3 Pipes

Pipes bieten eine Alternative zu Message Queues um Nachrichten zwischen Tasks auszutauschen. Pipes werden mit Gerätenamen gekoppelt und die einzelnen Tasks können mit den standard IO-Befehlen auf diese Pipes zugreifen. Sie sind den Message Queues sehr ähnlich. Der Unterschied besteht nur in den Zugriffs-Methoden. Bei Pipes gibt es keine Prioritäten, sie sind normale FIFO Queues. Der Schreibzugriff auf eine Pipe ist non-blocking, daß heißt, der Task schreibt in die Pipe und kehrt dann in den laufenden Task zurück. Der Lesezugriff auf eine Pipe ist blocking. Der Task wartet solange auf eine Nachricht in der Pipe bis eine ankommt. Kommt keine, so kann der Task für immer warten. Ähnlich wie in Abbildung 5 läßt sich auch eine Server/Client Struktur mit Pipes aufbauen.

4.4 Signale

Zusätzlich zu den bisher beschriebenen Methode zur Intertaskkommunikation gibt es noch dem Signal-Mechanismus. Signale können von jedem Task und jeder ISR an jeden anderen Task gesendet werden. Jeder Task kann nun einen Signal-Handler initialisieren. Beim Ankommen eines Signals wird der Task suspended und der angegebene Singal-Handler wird im Kontext des Tasks ausgeführt. Dadurch hat der Signal-Handler auf den Speicher des Tasks Zugriff. Beim Programmieren von Signal-Handlern muß darauf geachtet werden, daß dieser nicht in einen blocking Zustand kommt. Passiert dies kommt es zum Stillstand des gesamten Task und unter Umständen zum Deadlock des gesamten Systems.

Signal-Handler sollten nur wenige Ticks laufen und so wenige Operationen wie möglich durchführen, damit der Task seine normale Funktionsweise wieder aufnehmen kann.
Mit Signalen lassen sich am besten Laufzeitfehler von anderen Tasks melden oder das kurze signalisieren, daß eine bestimmte Aufgabe erfüllt wurde. Letzteres könnte z.B. das Auslesen eines Sensors sein. Sobald dieser Ausgelesen wurde und das Ergebnis im Speicher steht, kann mit einem Signal ein anderer Task benachrichtigt werden, daß dieser dieses Ergebnis verwenden kann.

5. POSIX

Zur Verbesserung der Kompatibilität unterstützt VxWorks zusätzlich noch den POSIX Standard 1003.1b. Dieser definiert bestimmte Kernel-Funktionen für Echtzeitbetriebssysteme. In diesem Abschnitt wird ein kleiner Überblick über die Implementierten POSIX Funktionen geben und Vergleiche zwischen einigen ausgewählten POSIX Erweiterungen und den dazugehörigen Erweiterungen des WIND Kernels.

5.1 POSIX Erweiterungen

POSIX Clocks und Timer

POSIX Memory locking

POSIX Threads

POSIX Scheduling Interface

POSIX Semaphoren

POSIX Mutexes und Condition Variables

POSIX Message Queues

POSIX Queued Signals

6. PROGRAMMIERUNG

6.1 Tornado Entwicklungs Umgebung

Beim Kauf von VxWorks liefert Wind River auch eine Entwicklungsumgebung mit. Diese Tornado genannte Umgebung bietet alle Funktionen, die nötig sind um Echtzeit-Applikationen zu entwickeln und sie in einem Simulator oder später auf der Zielarchitektur zu testen und zu debuggen. In Abbildung 6 ist der Aufbau dieser Umgebung gezeigt. Mit dieser Entwicklungsumgebung ist es möglich ein komplettes Echtzeitsystem zu konfigurieren und die nötigen Applikationen in das System einzuspielen. Wie man auf der Abbildung erkennen kann, bildet der Target-Server die Verbindung zwischen den Entwicklungswerkzeugen und der Zielarchitektur oder einem Simulator dieser.

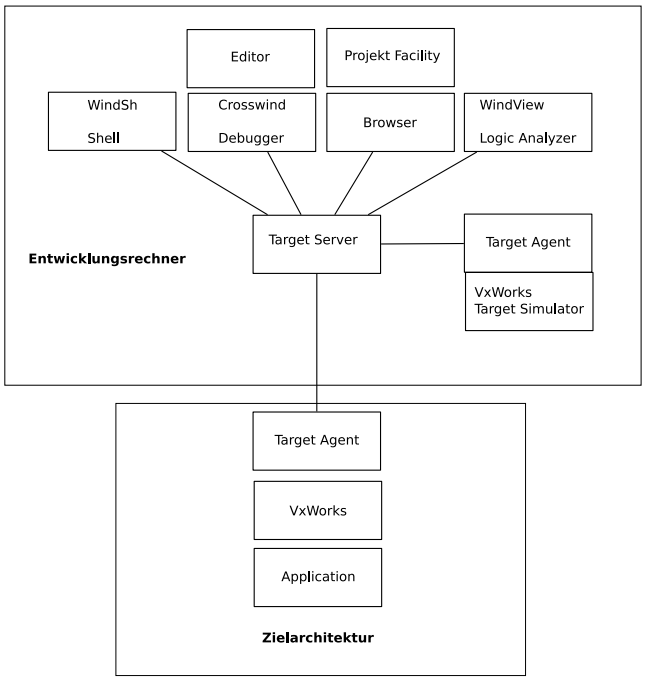

Abbildung 6: Entwicklungsumgebung Tornado

7. SCHLUSSFOLGERUNG

8. LITERATURVERZEICHNIS

[1] *VxWorks / Tornado II FAQ*. `http://www.xs4all.nl/~borkhuis/vxworks/vxworks.html`.

[2] *VxWorks 5.x*. Techn. Ber., Wind River.

[3] *VxWorks Programmer's Guide*.

[4] *VxWorks 6*. Techn. Ber., Wind River, 2000.

[5] Bala, N.: *Real-time Operating Systems (RTOS) modelling*.

[6] Collier, J.: *An Overview Tutorial of the VxWorks® Real-Time Operating System*. `http://cross-comp.com/pages/embedded/index.php#VxWorks\%20Programming`.

[7] Witzak, M. P.: *Echtzeit Betriebsysteme - Eine Einführung in Architektur und Programmierung*. Franzis', 2000.