Einführung in die Echtzeitbetriebssysteme

Hauptseminararbeit in dem Studiengang B.Sc. Informatik

von

Maximilian von Piechowski

Technische Hochschule Mittelhessen

Inhaltsverzeichnis

1 Was versteht man unter Echtzeitbetriebssysteme?

Ein Echtzeit-Betriebssystem, auch RTOS (Real Time Operating System) genannt, hat neben den Funktionen eines universallen Betriebssystem zusätzliche Zeitfunktionen zur Einhaltung von Zeitbedingungen und Vorhersehbarkeit des Prozessverhaltens.
(vgl. http://de.wikipedia.org/wiki/Echtzeitbetriebssystem, 21.6.2011, 17:40)

Beim Start eines Desktop Computers versucht das Betriebssystem so schnell wie möglich Kontrolle über den Computer zu übernehmen. In einem eingebettetem System wird dagegen im Normalfall zuerst die eigene Applikation gestartet.

Aus Performance Gründen wird bei einem Echtzeitsystem auf bestimmte Sicherheitsfunktionen verzichtet. So kann es passieren, dass bei einem Speicherzugriffsfehler das ganze System abstürzt und das komplette System neugestartet werden muss. Um Speicher zu sparen werden nur Dienste des RTOS mit einbezogen, die für das eingebettete System benötigt werden.

Weitere Unterschiede gibt es im Aufbau eines solchen Betriebssystem. In einem Echtzeitbetriebssystem kommt häufig die Mikrokernarchitektur zum Einsatz. Diese Architektur erleichtert die Skalierbarkeit und die Vorhersehbarkeit von Zugriffe auf Systemfunktionen. Der Nachteil dieser Architektur ist ihre Effizienz, die durch häufige Wechsel zwischen den sogenannten User- und Kernel-Mode sinkt.

Realzeitverarbeitung wird laut DIN 44300 wie folgt beschrieben:

“Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen“

1.1 Eigenschaften von Echtzeitbetriebssystemen

Vier wichtige Eigenschaften kennzeichnen ein Echtzeitbetriebssystem.

Rechtzeitigkeit

Die Geschwindigkeit der Ausführung eines Prozesses hat nicht Vorrang, sondern der Zeitpunkt seiner Ausführung und die festgelegte Zeitspanne, in der er ausgeführt wird. Korrekte Zeitpunkte werden von der Umwelt vorgegeben. So muss z.B. der Airbag eines Autos in einer bestimmten Zeitspanne nach Eintritt des Unfallereignisses (der von außen vorgegebene Zeitpunkt der Ausführung) ausgelöst werden, damit er seine Aufgabe erfüllt.

Vorhersehbarkeit

Die Vorhersehbarkeit der Arbeit eines Systems ist eine Ergänzung der Rechtzeitigkeitsforderung, da die Rechtzeitigkeit nur garantiert ist, wenn das System vorhersehbar arbeitet, d.h. erlaubte Funktionen dürfen in ihrer Ausführungszeit nicht sehr variabel sein. Hier spielt der Scheduler (vgl. 2.1.) eine wichtige Rolle.

Gleichzeitigkeit

Gleichzeitigkeit bedeutet, dass das System grundsätzlich die Verarbeitung paralleler Prozesse und verteilter Systeme unterstützt. So können nicht nur bei Mehrkernprozessoren die Vorteile von paralleler Verarbeitung genutzt werden.

Verlässlichkeit

Wie bereits in der DIN-Norm beschrieben, sollte ein Betriebssystem immer „betriebsbereit“ sein. Das System auf dem die Software läuft sollte robust und fehlertolerant sein. Es sollte im besonderen sichergestellt werden, dass Fehlbenutzung der Soft- oder Hardware nicht zur Abstürzen führt. Da Fehler nie ganz ausgeschlossen werden können, muss auch mit eventuellen Ausfällen rechnet werden. Die Minimierung der Fehler, im besonderen in einer Software, steht daher an erster Stelle. (vgl. Juliane T. Benra (2009), Seite 3 f.)

1.2 Zielsetzung bei der Programmierung eines Echtzeitbetriebssystem

Bei der Programmierung eines RTOS sollten bestimmte Eigenschaften des Projekts bzw. des Betriebssystems verfolgt werden. Ein Echtzeitbetriebssystem sollte so kompakt wie möglich gehalten werden, damit dieses auch auf kleineren Speichermedien Platz findet. Ein System sollte so skalierbar sein, dass Funktionen, die ein Programmierer nicht braucht, ausgeschaltet oder hinzufügt werden können. Die zuvor erwähnten Echtzeiteigenschaften sollten von dem System unterstützt werden. Unter anderem durch Interrupts und einem entsprechenden Scheduler. Die Qualitätssicherung ist ein weiterer wichtiger Teil der Entwicklung. Der Code sollte ausreichend dokumentiert und getestet werden, wenn eine neue Version einer Software erstellt wird. Außerdem sollte der Entwickler Kontakt zu den Anwendern suchen, um Feedback über sein Programm zu bekommen und um besser auf die Wünsche der Anwender eingehen zu können. (vgl. Juliane T. Benra, Seite 70 und 71)

1.3 Unterschiede der Echtzeitbetriebssysteme

Vor dem Einsatz eines Echtzeitbetriebssystems sollte zunächst geprüft werden, in welchen Anwendungsbereichen das System verwendet werden soll. So werden Systeme unter anderem als „weich“ oder „hart“ bezeichnet. Sogenannte „harte“ Systeme sind z.B. Systeme, die einen Airbag steuern oder in einem Atomkraftwerk zum Einsatz kommen. Sie zeichnen sich dadurch aus, dass die nicht Einhaltung von vorgegebenen Zeiten, schädlichen Folgen für den Menschen oder die

Umwelt nach sich ziehen kann. Dagegen haben „weiche“ Systeme, die z.B. in einer Fertigungsstraße verwendet werden, bei Fehlfunktionen nicht diese gravierenden Folgen. Dem Unternehmen entsteht durch eine Verzögerungen meist „nur“ ein finanzieller Schaden. (vgl. Juliane T. Benra (2009), Seite 3)

Bei der Wahl des Betriebssystems sollte der Programmierer darauf achten, ob diese die Hardware-Architektur unterstützt.

Der Einsatz von kommerziellen Echtzeitbetriebssystemen sollte geprüft werden, da häufig auch ein freies Betriebssystem die Unterstützung bietet, die ein Programmierer braucht und somit Kosten eingespart werden können. Ein kommerzielles Betriebssystem hat den Vorteil, dass der Support für die Software meist besser ist.

1.4 Einsatz eines Echtzeitbetriebssystems

Bevor man sich entscheidet ein Echtzeitbetriebssystem zu verwenden, stellt sich die Frage, ob in ein solches System nur Vorteile hat. Zum einen kann die Einarbeitungszeit in ein solches System länger dauern, als die eigentliche Entwicklung eines Programms, ohne ein solches System.

Der Einsatz ist dann zu empfehlen, wenn das Programm, das entwickelt werden soll, eine gewisse Größe erreichen wird und damit auch eine höhere Komplexität. Die Komplexität kann durch den Einsatz von einem RTOS sinken.

Außerdem können Code-Teile für andere Hardware-Architekturen wiederverwendet werden, ohne diese im großen Umfang neu strukturieren zu müssen.

2 Prozesse und Prozesszustände

Bei den Prozessen in einem Echtzeitbetriebssystem unterscheidet man zwischen Anwenderprozessen und überwachenden Prozessen. Anwenderprozesse sind solche, die vom Entwickler, der das RTOS benutzt, erzeugt werden. Dienste, die bei einem Betriebssystem im Hintergrund laufen, wie zum Beispiel der Scheduler, werden überwachende Prozesse genannt.

Ein Prozess kann synchron oder asynchron zugeteilt werden. Bei der synchronen Zuteilung steht vor der Ausführung des Programms fest, wann und wie lange welcher Prozess den Prozessor in Anspruch nimmt. Damit ist die Ausführung strikt sequentiell. Um auf Einflüsse aus der Umwelt reagieren zu können wird die asynchrone Zuteilung verwendet. Bei der asynchronen Zuteilung wird zur Laufzeit entschieden, welcher Prozess an die Reihe kommt. In den heutigen Systemen kommt keine synchrone Zuteilung zur Anwendung, stattdessen wird die asynchrone Zuteilung verwendet, damit auf Anforderungen von außen reagiert werden kann.

In einem RTOS werden ein oder mehrere Prozesse erzeugt, wenn eine Funktion des RTOS

aufgerufen wird. In den meisten Echtzeitbetriebssystemen ist ein Prozess eine einfache Subroutine. Die Prozesse befinden sich immer in einer der Zustände, „laufend“, „bereit“ oder „blockiert“. (vgl. Juliane T. Benra (2009), Seite 74, 75, 85)

2.1 Der Scheduler

Der Scheduler verfolgt die Zustände der Prozesse und entscheidet, welcher Prozesse in den Zustand „laufend“ versetzt wird. Dabei gibt es Unterschiede zu den Schedulern in Windows und Unix. Der Scheduler entscheidet anhand der Prioritäten, welche zuvor an die Prozesse vergeben werden. Wenn ein Prozess im Zustand „bereit“ ist und einer höhere Priorität als der laufende hat, wird dieser sofort verdrängt. Wie mit Prozessen mit gleicher Priorität umgegangen wird, hängt vom jeweiligen RTOS. Eine Möglichkeit ist, Prozesse mit gleicher Priorität zu verbieten. Eine andere Möglichkeit besteht darin, einen Prozess zunächst auszuführen bis dieser „blockiert“ und somit der andere Prozessauf den Prozessor zugreifen kann.

Prozesse die eine hohe Priorität haben und im Zustand „laufend“ sind, blockieren so lange den Prozessor bis diese ihn wieder freigeben. Das RTOS geht davon aus, dass Prioritäten entsprechend ihrer Relevanz vergeben werden und dass sie mit Bedacht gewählt werden.

Ein Prozess kann sich immer nur selbst blockieren und kann nicht von außerhalb blockiert werden. Weder der Scheduler oder andere Prozesse können einen Prozess blockieren. Das hat den Nachteil, dass ein Prozess erst ausgeführt werden muss, damit dieser feststellen kann, dass nichts mehr zu tun ist. Ist ein Prozess blockiert, wird dieser nie den Prozessor bekommen. Außer andere Prozesse oder Interrupts signalisieren, dass das Ereignis eingetreten ist, auf das gewartet wird.

Damit der Scheduler weiß, bei welchen Ereignis er welchen Prozess das Signal geben muss, stellt das RTOS einige Funktionen bereit, damit die Prozesse dem Scheduler mitteilen können auf welches Ereignis sie warten. Wenn alle Prozesse blockiert sind, durchläuft der Scheduler eine Schleife und wartet auf ein Ereignis. Der Programmierer muss dafür sorgen, dass ein solches Ereignis dann auch eintritt. (vgl. David E. Simon (2009), Seite 140 und 141)

Die Verdrängung eines Prozesses wird auf zwei verschiedene Arten geregelt.

none-preemptive RTOS

Bei einem none-preemptive RTOS darf der Prozess mit der kleineren Priorität seine Arbeit zu Ende führen bis er blockiert. Dann übernimmt der Prozess mit der höheren Priorität.

preemptive RTOS

Bei einem preemptive RTOS wird der ausführende Prozess sofort von dem Prozess mit der

höheren Priorität verdrängt. (vgl. David E. Simon(2009), Seite 142 und 143)

Initialisierung

Damit Prozesse unabhängig voneinander geschrieben werden können und der Programmierer sich nicht darum kümmern muss, welcher Prozess an der Reihe ist, muss dem Betriebssystem beim Start mitgeteilt werden, welche Prozesse mit welcher Priorität gestartet werden sollen. Dies geschieht nach der Initialisierung des RTOS, aber noch vor dem Start des RTOS.

2.2 Schedule-Verfahren

Unter den Echtzeitbetriebssystemen gibt es verschieden Verfahren des schedulings.

Earliest deadline first scheduling

Bei dem EDFS handelt es sich um ein Verfahren, bei dem zur Laufzeit berechnet wird welcher Prozess am nächsten an seiner „Deadline“(Ende der Ausführung) ist und dieser wird dann ausgeführt.

Rate-monotonic scheduling

Bei dem Rate-monotonic scheduling wird in einer statischen Klasse festgehalten, welche Priorität ein Prozess hat. Die Priorität wird daran berechne,t wie lange die Ausführung eines Prozesszyklus dauert. Umso kürzer dieser ist, umso höher ist die Priorität des Prozesses.

Round-Robin

Die Prozesses erhalten eine Reihenfolge und jeder Prozess bekommt eine Zeitspanne zur Ausführung, die der Programmierer festlegt. Jeder Prozess wird dann Reihum in der angegebenen Reihenfolge und Zeit ausgeführt.

Fixed priority preemptive scheduling

Jedem Prozess wird eine feste Priorität gegeben. Wenn ein Prozess mit höherer Priorität in den Zustand „bereit“ kommt, wird der laufende Prozess sofort verdrängt. Dieses Verfahren findet häufige Anwendung in Echtzeitbetriebssystemen

Event scheduling

Bei dem Event scheduling werden Verdrängungen durch Events gesteuert. So kann ein Prozess entweder durch ein Event gestoppt oder gestartet werden. Ähnlich wie das Event scheduling funktioniert auch das zeitbasierte scheduling, mit dem Unterschied, dass die Ausführung durch einen bestimmten Zeitpunkt gestartet oder gestoppt wird.(vgl. http://en.wikipedia.org/wiki/Real-time_operating_system, 21.6.2011,17:40)

3 Prozesse und Daten

Jeder Prozess hat seinen eigenen privaten Kontext. Der Kontext beinhaltet die Registerwerte, den Programmzähler und den Stack. Andere Daten, darunter globale, statische und initialisierte, werden von allen anderen Prozessen geteilt. So verhalten sich in einem RTOS die Prozesse ähnlich, wie Threads in einem Desktop-Betriebssystem. (vgl. . David E. Simon(2009), Seite 144 und 145)

3.1 Probleme bei gemeinsam genutzten Daten

Versuchen zwei Prozesse auf die gleiche Variable zeitgleich zuzugreifen, treten Fehler auf. Dies passiert auch bei vermeintlich atomaren Befehlen. Ein einfaches Inkrement einer Zahl besteht aus drei Befehlen „MOVE R1, i“, „ADD R1, 1“ und „MOVE i, R1“, dabei kann es zwischen jedem Befehl zur einer Unterbrechung kommen. Ob eine Funktion threadsicher ist, hängt unter anderem vom Mikroprozessor und dem Compiler ab.

Um dies zu verhindern, sollte eine Funktion verwendet werden, die „reentrant“ ist. Die Funktion sollte Variablen auf eine atomaren Weg benutzen, außer sie befinden sich auf dem Stack oder ist eine private Variable des Prozesses. In dieser Funktion sollten keine Funktionen aufgerufen werden, die nicht auch „reentrant“ sind. Die Funktion sollte außerdem die Hardware nur atomar ansprechen.

Beispiel: Atomarer Aufruf

I++;	
MOVE R1,i	INC(i)
ADD R1, 1	
MOVE i,R1	

Das Beispiel zeigt, wie ein vermeintlich atomarer Befehl aussehen kann und sich nicht sagen lässt, ob es sich um einen atomaren Befehl handelt, da dies auch vom Prozessor und vom Compiler abhängt.

In der linken Spalte ist zu erkennen, dass das Inkrement von i++ drei Schritte Bedarf.

1. Es wird der Wert der in i steht in den Register R1 geschoben.

2. Der Inhalt von R1 wird mit 1 addiert und in dem Register R1 gespeichert.

3. Der Inhalt des Registers wird wieder in i geschoben.

Zwischen jedem dieser Aufrufe kann eine ungewollte Unterbrechung stattfinden. Anders in der rechten Spalte. Der Prozessor braucht nur einen Maschinenbefehl, um ein Inkrement auszuführen.

(vgl. David E. Simon(2009) Seite 147 f.)

3.2 Überblick über C Speicherarten

Static

Die Variable befindet sich in einer festen Speicherstelle und kann von mehreren Prozessen genutzt werden. Sie kann nicht über andere C-Dateien angesprochen werden. Wenn die Variable in einer bestimmten Funktion angelegt wurde, kann diese auch nur über diese Funktion verändert werden.

Public

Die Variable kann von anderen Prozessen bzw. Funktionen aus anderen C-Dateien verändert werden.

Initialized

Ob eine Variable initialisierte ist, ändert nicht die „Sichtbarkeit“ für andere Prozesse oder C-Dateien.

Pointer

Zeiger befinden sich in einem festen Speicherbereich und können damit auch gemeinsam von anderen Prozessen genutzt werden.

Local

Lokale Variablen liegen auf dem Stack.

Parameters

Parameter liegen i.d.R. auf dem Stack, außer es handelt sich um Zeiger, dann kann es sein, dass mehrere Prozesse darauf zugreifen. (vgl. David E. Simon(2009) Seite 150 und 151)

4 Semaphoren und gemeinsam genutzte Daten

Um Probleme mit gemeinsamen genutzten Daten zu umgehen, kann der Einsatz von Semaphoren Abhilfe schaffen. Semaphoren schützen einen bestimmten Bereich davor, dass ein anderer Prozess gleichzeitig auf diesen zugreifen kann. Betritt ein Prozess diesen Bereich, werden andere Prozesse, die auch den Bereich betreten wollen, blockiert. Erst wenn der Prozess der den kritischen Bereich betreten hat und den Semaphor hält diesen wieder freigegeben hat, kann einer der warteten Prozesse den Bereich betreten. Dabei gibt es verschiedene Arten von Semaphoren. Die einfachste Form ist der binäre Semaphor. Dabei kann der Semaphor immer nur von einem Prozess gehalten werden. (vgl. . David E. Simon(2009) Seite 153 und 154)

4.1 Initialisierung

Bei der Initialisierung ist darauf zu achten, dass der Semaphor vor dem Start des RTOS initialisiert wird. Wenn z.B. die Initialisierung erst in einem der Prozesse stattfindet, die mit dem Semaphor arbeiten, kann es zu unvorhergesehenen Fehler kommen, da nicht fest steht, welcher Prozess den Prozessor als erstes bekommt. (vgl. . David E. Simon(2009) Seite 158 und 159)

4.2 Reentrancy und Semaphoren

Um sicher zu gehen, dass der Bereich, der geschützt werden soll auch wirklich geschützt wird, sollte bei wechselseitigen Ausschluss in der Funktion selbst, die Befehle der Semaphoren ausführen werden und nicht außerhalb der Funktion. Ansonsten könnte leicht vergessen werden, den Semaphore zu benutzen. (vgl. David E. Simon(2009), Seite 160)

4.3 Mehrere Semaphoren im Einsatz

In einem RTOS können beliebig viele Semaphoren eingesetzt werden. Alle Semaphoren sind unabhängig voneinander. Wenn ein Semaphor belegt ist, kann ein anderer frei sein. Der Einsatz von mehreren verschiedenen Semaphoren hat den Vorteil, dass ein Prozess nicht zwangsläufig blockiert wird, obwohl er eine andere Variable verändern will(für jede Variable einen anderen Semaphor). Belegt z.B. Prozess P1 Semaphor S1 um Variable A zu ändern, und will jetzt Prozess P2 Variable B ändern, der auch durch Semaphor S1 geschützt ist, kann das nicht passieren solange P1 S1 belegt. Das Wissen darüber, welcher Semaphor zu welchen Daten gehört, liegt beim Programmierer haben. Das RTOS hat keine Funktionen um dies zu prüfen. (vgl. Seite 161 und 162, An Embedded Software Primer)

4.4 Probleme bei der Nutzung von Semaphoren

Bei der Nutzung von Semaphoren können verschiedene Probleme auftreten.

- Es wird der Befehl zur Sperrung nicht ausgeführt.
- Die Sperre wird nicht wieder aufgehoben, sodass alle weiteren Prozesse, die den Bereich betreten wollen für immer blockiert werden.
- Durch den Einsatz von mehreren Semaphoren in dem Programm, kann es dazu kommen, dass man den falschen Semaphor benutzt
- Ein Semaphor wird zu lange gehalten, sodass andere Prozesse, die darauf warten nicht mehr ihre Echtzeiteigenschaften erfüllen.
- Ein Prozess mit hoher Priorität wird blockiert und kann nicht ausgeführt werden, da ein

Prozess mit kleiner Priorität den Semaphor hält und ein Prozess mit mittlerer Priorität den kleineren Prozess verdrängt, sodass der Prozess mit kleiner Priorität, den Semaphor nicht freigegeben kann. (vgl. David E. Simon(2009), Seite 164 f.)

4.5 Semaphoren Varianten

Neben den zuvor besprochenen binären Semaphoren gibt es weitere Arten.

- Der zählende Semaphor. Er lässt eine bestimmte Anzahl von Prozessen in den kritischen Bereich.
- Der Resource Semaphor. Er kann nur von den Prozessen wieder freigeben werden, die ihn belegt haben.
- Der Mutex. Er kann unter anderem auch mit dem Prioritäten-Problem, wie oben beschrieben, umgehen. Es gibt Betriebssysteme, die ein Semaphor Mutex nennen, nicht dessen Eigenschaft besitzt.

Bei der Vorgehensweise, welcher Prozess zuerst einen Bereich betreten darf, nachdem dieser blockiert war, gibt es Unterschiede. Entweder wird der Prozess ausgeführt, der am längsten wartet oder der mit der höheren Priorität. Die Handhabung liegt am RTOS. Bei einigen RTOSs kann die Vorgehensweise festlegt werden. (vgl. David E. Simon (2009), Seite 167)

4.6 Wege zum Schutz von gemeinsam genutzten Daten

Es gibt drei verschiedene Wege die Daten zu schützen.

Der erste Weg ist das Ausschalten der Interrupts. Dies ist eine sehr effiziente Lösung, sollte aber nur eingesetzt werden, wenn der zu schützende Bereich sehr kurz ist. Zusätzlich kann kein Prozesswechsel stattfinden, da dieser durch Interrupts gesteuert wird.

Der zweite Weg ist das Benutzen von Semaphoren. Hier wird wie vorher schon beschrieben, nicht der Wechsel der Prozesse verhindert, sondern es wird mit Sperren gearbeitet, die es möglich machen, dass mehrere Prozesse den gleichen Code zur gleichen Zeit ausführen können.

Der dritte Weg ist das Ausschalten der Prozesswechsel. Hierbei wird verhindert, dass ein Prozess bei seiner Ausführung unterbrochen wird. Interrupts können aber weiterhin Kontrolle über den Prozessor übernehmen. (vgl. David E. Simon(2009), Seite 167 und 168)

5 Weitere Bestandteile eines Echtzeitbetriebssystems

Ein RTOS kann noch weitere Dienste zur Verfügung stellen, um den Programmierer zu unterstützen. Die Kommunikation kann nicht nur über Semaphoren geschehen, sondern es gibt weitere Funktionen wie Queues, Mailboxes und Pipes. Zusätzlich gibt es Zeitfunktion und Events.

5.1 Kommunikation

Queues

Genau wie bei den Semaphoren muss eine Initialisierung der Queues stattfinden, bevor diese benutzt werden können. Der Speicher muss explizit vom Programmierer allokiert werden. Jede Queue wird über eine ID identifiziert.

Wenn das Lesen aus der Schlange oder das Schreiben in die Schlange scheitert, wird ein Fehlercode erzeugt. Der Programmierer muss dann entsprechend auf den Fehlercode reagieren.

In anderen RTOS gibt es auch die Möglichkeit, dass der Prozess blockiert solange die Schlange leer oder voll ist und der Befehl ausgeführt wird, wenn das entsprechende Ereignis eingetreten ist.

Mailboxes

Eine Mailbox verhält sich ähnlich wie eine Queue, hat aber noch zusätzliche Funktionen, die z.B. überprüfen, ob die Mailbox leer ist oder die die Mailbox zerstören können. Die Anzahl der Funktionen hängt vom jeweiligen Betriebssystem ab.

Pipes

Einige RTOS, die Pipes unterstützen, haben die Möglichkeit eine variierende Größe einer Nachrichten zu senden. Das hat den Vorteil, dass nicht beim Start des Programmes feststehen muss wie groß eine Nachricht sein muss oder kann. Dabei werden die Standard C-Funktionen fread und fwrite benutzt. (vgl. David E. Simon(2009, Seite 173 f.)

5.2 Zeitfunktionen

Ein Echtzeitbetriebssystem kann verschiedene Zeitfunktionen zur Verfügung stellen.

Abschaltung von Hardware

Eine Zeitfunktion, die zum Einsatz kommt, dient zur Abschaltung von Hardware. So kann nach einer bestimmten Zeit, in der die Hardware nicht genutzt wird, diese abgeschaltet werden. Z.B. kann ein Barcode-Reader 5 Sekunden nach der Benutzung ausgeschaltet werden, um Strom zu sparen.

Delay

Eine weitere Zeitfunktion ist das „Delay“. Es arbeitet ähnlich wie ein Sleep und lässt den Prozess eine bestimmte Zeit warten. Dabei ist zu beachten, dass die Zeit, die der Prozess warten soll., nicht genau zu bestimmen ist, da jeder Tick eine bestimmte Zeit benötigt und damit die vorgegebene Wartezeit nicht punktgenau erreicht werden muss. Die Genauigkeit hängt somit von der Länge eines Ticks ab.

Warten auf Ereignisse

Prozesse können auf eine Benachrichtigung von bestimmten Diensten von Semaphoren, Queues und Mailboxen eine bestimmte Zeit lang warten. Sollte das Ereignis nicht in der Zeit eingetreten sein, wird das Programm fortgesetzt.

Ausführung eines Prozesses

Ein Prozess kann nach einer bestimmten Zeit gestartet oder beendet werden. Dabei wird der Code direkt im Interrupt oder in einem neuen Prozess ausgeführt.
(vgl. David E. Simon(2009), Seite 186 f.)

6 Beispiel RTOS: FreeRTOS

Bei dem FreeRTOS handelt es sich um eine skalierbares Echtzeitbetriebssystem für kleine eingebettete Systeme.

6.1 Eckdaten des FreeRTOS

-Das FreeRTOS unterstützt 27 Architekturen unter anderem ARM7 ARM9 und den Cortex 3M.

-Die Größe des Kernels liegt zwischen 4 und 9 Kbyte.

-Es bietet Routinen zur Kommunikation und Synchronisation, darunter Semaphoren und Queues.

-Das System unterstützt Sub-Routinen und Prozesse und Mutexe zum Vererben der Prioritäten.

6.2 Programmbestandteile

Ein FreeRTOS Programm besteht aus fünf Hauptfunktionen

Die Main ist wie auch in anderen Betriebssystemen der Einstiegspunkt in das Programm und wird als erstes ausgeführt. Die Funktion vApplicationMallocFailedHook wird ausgeführt, wenn es beim Allokieren des Speichers zu einem Fehler kommt. VapplicationStackOverflowHook wird gestartet, wenn es zu einem Überlauf des Stacks kommt. vApplicationIdleHook kann über ein Flag aktiviert werden und wird ausgeführt, wenn der Prozess gerade untätig ist. VapplicationTickHook wird bei jedem Tick ausgeführt, dazu muss aber schon wie bei der IdleHook zuvor eine Flag gesetzt werden.

7 Literaturverzeichnis

http://en.wikipedia.org/wiki/Real-time_operating_system, 21.6.2011,17:50

http://en.wikipedia.org/wiki/Real-time_operating_system

www.freertos.org

Juliane T. Benra und Wolfgang A. Halang: Software-Entwicklung für Echtzeitsysteme, Springer Verlag 2009

David E. Simon, An Embedded Software Primer, Addison-Wesley 2009

http://de.wikipedia.org/wiki/Echtzeitbetriebssystem, 21.6.2011, 17:40