Entwurf und Implementierung einer Multitasking-Umgebung für den Arduino Due

Bachelorarbeit

InfB-BA/Inf Abschlussmodul B.Sc. Informatik

Autor: Enno Köster 5953573
Erstgutachter: Norman Hendrich
Zweitgutachter: Kai Jander
Zeitraum: Beginn: 16.12.2015, Abgabe: 3.6.2015
Universität Hamburg

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Bachelor of Science Informatik

Juni 2015

Inhaltsverzeichnis

1 Einleitung

Der Name Arduino steht für die einfache, günstige, plattformübergreifende Entwicklung von eingebetteten Systemen auf Basis von offener Hardware und Software.

Bei Systemen mit geringer Leistung und sehr beschränkten Ressourcen wie den AVR-Prozessor basierten alten Arduinos werden meist nur wenige oder eine einzige Aufgabe bearbeitet. Wie in allen anderen Bereichen wird auch die Leistung eingebetteter Systeme stetig größer. Der Arduino Due erhöht im Vergleich zu seinen Vorgängermodellen die Taktrate des Prozessors um das sechfache und die Größe des Arbeitsspeichers um das 48-fache. Durch diese Leistungssteigerung ist der Gedanke naheliegend ein System für eine Vielzahl von Aufgaben zu nutzen.

Mehrere Aufgaben ganz einfach in einem Kontrollfluss (Task) zu erledigen macht den Quelltext recht schnell unübersichtlich, insbesondere wenn häufig zwischen den Aufgaben gewechselt werden muss. Die Entwicklung wird fehleranfällig und kompliziert. Dem kann man entgegenwirken, indem man logisch trennbare Teile in verschiedene Tasks aufteilt. Die einzelnen Tasks müssen sich den Prozessor teilen. Damit sie ihre Funktion erfüllen können, müssen sie abwechselnd auf dem Prozessor ausgeführt werden. Um dies zu gewährleisten, muss es ein System, genannt Scheduler, geben, dass die Verteilung der Rechenzeit kontrolliert.

Diese Arbeit zielt auf ein möglichst einfach zu nutzendes System im Geiste der Arduinophilosophie ab. Es soll mit wenigen Befehlen die Implementation mehrerer nebeneinander ausgeführter Funktionen ermöglichen, sodass auch Einsteiger schnell und ohne viel Vorwissen anfangen können das System zu nutzen. Die Hauptzielplattform ist der Arduino Due. Die Unterstützung älterer Arduino Systeme besitzt allerdings ebenfalls einen hohen Stellenwert, weil sie die Lösung einer größeren Zahl von Entwicklern zugänglich macht.

Mit der Ausführung mehrerer Funktionen nebeneinander ist es allerdings nicht getan. Durch den Einsatz mehrerer Tasks ergeben sich zusätzliche Anforderungen an die Arduino-Bibliothek, welche Anpassungen notwendig machen. Die Funktionalitäten der Bibliothek und des Multitaskingsystems überschneiden sich. Es soll anhand einiger Beispiele aufgezeigt werden, wie diese Konflikte gelöst werden können.

Auch auf den Nutzer des Multitaskingsystems kommen zusätzliche Anforderungen zu. Zum Beispiel können Abhängigkeiten zwischen den verschiedenen Teilen seines Programms bestehen, die eine Kommunikation und/oder Synchronisation erfordern. Diese Probleme muss der Nutzer selbst lösen. Das Multitaskingsystem kann ihm dabei allerdings mit einigen zusätzlichen Funktionalitäten unter die Arme greifen. Neben der Hauptaufgabe des Schedulings soll das System nach Möglichkeit auch für die Kommunikation und Synchronisation Funktionen bereithalten.

2 Grundlagen

2.1 Begriffsdefinitionen

In der Literatur sind viele der in dieser Arbeit genutzten Grundbegriffe verschieden definiert oder aber gleiche Begriffe meinen unterschiedliche Dinge. Dieses kurze Kapitel soll Missverständnissen vorbeugen, indem die wichtigsten Ausdrücke festgelegt werden. Ihre Definitionen basieren auf denen aus "Moderne Betriebssysteme"[Tan09] von Andrew S. Tanenbaum.

Prozesse, Threads und Tasks

Ein Prozess ist die Instanz eines Programms in Ausführung, inklusive des aktuellen Wertes des Befehlszählers, der Registerinhalte und der Belegungen der Variablen. Konzeptionell besitzt jeder Prozess seine eigene virtuelle CPU. Jeder Prozess hat seinen eigenen Speicherbereich, der vor Zugriffen durch andere Prozesse geschützt ist. Er kann drei Zustände haben, rechnend(die Befehle werden in diesem Moment auf der CPU ausgeführt), rechenbereit(kurzzeitig gestoppt, um einen anderen Prozess rechnen zu lassen) und blockiert(nicht lauffähig bis ein bestimmtes externes Ereignis eintritt). [Tan09]

In traditionellen Betriebssystemen hat jeder Prozess einen eigenen Adressraum und einen einzigen Ausführungsfaden(*thread of control*). [Tan09] Threads sind also einem Prozess untergeordnet. Sie besitzen einen eigenen Befehlszähler und Registerinhalte. Sie haben Zugriff auf auf den kompletten Speicherbereich des Prozesses. Die Belegungen von Variablen sind abhängig von der Art ihrer Allokation zwischen den Threads geteilt oder unabhängig voneinander. Mehr dazu unter *Stack und Heap*.

Exkurs Arduino: Die Prozessoren älterer Arduino-Boards besitzen keine hardwareseitige Unterstützung für Speicherschutz, um die Trennung der Speicherbereiche verschiedener Prozesse zu erzwingen. Erst der Arduino Due hat eine "Memory-Protection-Unit". Entsprechend ist die softwareseitige Unterstützung des Prozesskonzepts ebenfalls in den meisten Systemen non-existent, in Zukunft aber vielleicht realisierbar. Im folgenden soll deswegen nur noch die Rede von Tasks sein, wenn Aussagen sowohl auf Prozesse als auch auf Threads zutreffen, und angenommen werden, dass damit immer Threads mit geteiltem Speicher gemeint sind. Falls für das Thema wichtige Unterschiede bestehen, werden diese erwähnt.

Stack und Heap

Der Speicher wird üblicherweise in drei Teile aufgeteilt, den Stack, den Heap und einen Bereich für globale Variablen. Wenn der Arduino regulär mit einem einzelnen Task programmiert wird, ist die Unterscheidung der Abschnitte größtenteils unerheblich, da der Ort und die Größe des Stacks und des globalen Bereichs vorher bekannt sind. Sobald allerdings mehrere Tasks laufen sollen, muss die Speicherverwaltung sich auch um die dynamisch erzeugten Stacks kümmern.

Für jeden Task wird bei seinem Start eine gewisse feste Menge Speicher für seinen Stack voralloziert. Der Stack enthält Rücksprungadressen für Funktionen und alle lokalen Variablen in Funktionen, die "normal" (z.B. int a=0;) deklariert werden. Zusätzlich kann ein Task z.b. über die Funktion *malloc()* in C oder den *operator new* in C++ dynamisch zusätzlichen Speicher allozieren. In den globalen Bereich fallen zum einen Variablen, die außerhalb aller Funktionen deklariert sind, und zum anderen als *static* deklarierte Variablen, die zwar innerhalb einer Funktion deklariert werden, aber trotzdem global sind.

Multitasking

Multitasking bedeutet die Ausführung mehrerer Tasks auf einem Prozessor. Für eine fehlerfreie Funktion muss klar definiert sein, wie ein laufender Task die Kontrolle abgibt, wie ein anderer sie wieder aufnimmt und welcher der gestarteten Tasks als nächstes an der Reihe ist. Die Bestimmung der Reihenfolge ist Aufgabe des Schedulers, der weiter unten beschrieben wird.

Ein Wechsel der Kontrolle von einem Task zum anderen wird Kontextwechsel genannt. Als Kontext wird hier der Inhalt der Register betrachtet. Es wird zunächst der Kontext des abgebenden Tasks im Speicher gesichert. Dann bestimmt der Scheduler, welcher Task als nächstes ausgeführt werden soll. Danach werden die zu einem früheren Zeitpunkt gesicherten Kontextinformationen des folgenden Tasks wiederhergestellt.

Interrupts

Interrupts sind, wie der Name schon suggeriert, Unterbrechungen des normalen Kontrollflusses. Diese können von verschiedenen Ereignissen ausgelöst werden.

Das wichtigste für diese Arbeit ist der Ablauf eines Timers, denn darauf basieren die später beschriebenen preemptiven Scheduler. Fehler im derzeit ausgeführten Programm wie zum Beispiel unzulässige Speicherzugriffe oder Berechnungsfehler können ebenfalls Interrupts auslösen. Außerdem können im Bereich der eingebetten Systeme besonders wichtige äußere Ereignisse, wie Flankenwechsel an digitalen Pins des Prozessors, den normalen Ablauf unterbrechen.

Wenn ein Interrupt ausgelöst wird, arbeitet der Prozessor eine vorher für dieses Ereignis festgelegte Prozedur ab. Die Adresse dieser Prozedur ist in einer Tabelle abgelegt, die "Interrupt Vector" genannt wird.

2.2 Scheduler

Der Scheduler ist das zentrale Element des Systems. Er bestimmt, welcher Task als nächstes ausgeführt wird.

Scheduler lassen sich anhand zwei verschiedener Merkmale voneinander unterscheiden und in Kategorien einordnen. Das erste ist der Zeitpunkt eines Kontextwechsels, welcher entweder vom gerade ausgeführten Task selbst gewählt oder vom System vorgegeben werden kann. Das zweite Merkmal ist die Reihenfolge und Häufigkeit, in der die Tasks ausgeführt werden.

2.2.1 Zeitpunkt eines Kontextwechsels

Kooperative Scheduler

Dieser Schedulertyp zeichnet sich dadurch aus, dass die Kontrolle vom aktuell laufenden Task selbstständig an den Scheduler zurückgegeben wird. Ein Kontextwechsel passiert nur, wenn der Task eine in den meisten Systemen "yield()"(englisch: hergeben) genannte Funktion aufruft, welche den Task unterbricht und den Scheduler aufruft.

Kooperative haben gegenüber preemptiven Schedulern einige kleine Vorteile. Diese werden jedoch durch einen signifikanten Nachteil erkauft.

Ein Vorteil dieses Ansatzes ist, dass im Gegensatz zu dem später erklärten preemptiven Ansatz keine hardwareseitige Unterstützung für Timer-Interrupts benötigt wird. Alle Arduino Systeme haben diese Unterstützung. Daher ist dieser Punkt höchstens in dem Fall von Interesse, dass man eine Portierung des Systems auf andere Prozessoren plant, welche keine Timer-Hardware besitzen.

Außerdem spricht im Falle von sehr wenigen Kontextwechseln der geringe Verlust an Rechenzeit für das Scheduling für kooperative Scheduler. Dies fällt allerdings nur bei sehr niedrig getakteten Prozessoren ins Gewicht.

Ein weiterer Pluspunkt ist, dass der Programmierer selbst bestimmt, wann ein Task unterbrochen wird. Wenn der Zeitpunkt bestimmter Vorgänge innerhalb einer Sequenz besonders wichtig ist, um zum Beispiel die Spezifikationen eines Protokolls exakt einhalten zu können, dann ist dies ganz einfach dadaurch zu gewährleisten, dass man im betreffenden Task die Kontrolle nicht abgibt bevor die komplette Sequenz durchlaufen ist. Beim preemptiven Ansatz muss man zusätzliche Maßnahmen ergreifen, um dies zu gewährleisten.

Der große Nachteil des kooperativen Ansatzes ist allerdings genau dieses mögliche Verhalten eines Tasks, den Prozessor für längere Zeit belegen zu können. Dies kann durch einen Programmierfehler auch unabsichtlich zu beliebig langen Wartezeiten führen. Ein Fehler in einem Task kann das ganze System gefährden. Der Programmierer muss selbst dafür Sorge tragen, dass keiner seiner Tasks den Prozessor zu lange belegt und damit eventuell andere Tasks in ihrer Funktion behindert.

Preemptive Scheduler

Dieser Schedulertyp wartet nicht wie kooperative Scheduler darauf, dass ein Task freiwillig den Prozessor freigibt. Dies ist zwar generell möglich, aber nicht notwendig. Ein preemptiver Scheduler erzwingt die Unterbrechung der Ausführung eines Tasks nach einem festen Intervall. Ein Timer löst periodisch einen Interrupt aus, woraufhin der Scheduler seine Arbeit aufnimmt.

Der große Vorteil preemptiver Scheduler im Vergleich zu ihren kooperativen Pendants ist, dass sie ihren großen Nachteil korrigieren. Ein Task kann nicht alle anderen blockieren, weil ihm nach Ablauf seines Intervalls garantiert die Kontrolle über den Prozessor entzogen wird.

Erkauft wird das durch die entsprechenden kleinen Nachteile.

Wenn der Scheduler zum Beispiel auf ein Intervall von einer Millisekunde eingestellt ist, dann wird 1000 mal pro Sekunde ein Kontextwechsel vorgenommen, ganz egal, ob das aus Sicht der beteiligten Tasks notwendig ist oder nicht.

Wenn man einen Ablauf im Programm hat, dessen Elemente zwingend zueinander zeitnah abgearbeitet werden müssen, dann muss dieser Teil als sogenannter “kritischer Bereich” markiert werden, um eine Unterbrechung durch den Scheduler zu unterbinden.

2.2.2 Reihenfolge der Ausführung

Round Robin

Die Grundidee dieser Variante ist sehr einfach. Der Prozessor, soll den Tasks zu gleichen Teilen zur Verfügung gestellt werden. In der Frage , wie “gleich” definiert sein soll, unterscheiden sich die Implementationen.

Eine Möglichkeit ist, dass man einfach die Tasks der Reihe nach einmal ausführen lässt. Beim kooperativen Ansatz, würde das bedeuten, dass Tasks, die erst nach längerer Zeit die Kontrolle zurückgeben, gleich oft laufen aber mehr Zeit bekommen würden. Beim preemptiven Ansatz tritt dieses Problem nicht auf, da jeder Task spätestens nach Ablauf des vom Scheduler festgelegten Intervalls unterbrochen wird.

Der Ansatz, das Ausführungsrecht reihum zu verteilen, kann noch etwas verfeinert werden, indem man es ermöglicht den Tasks eine Priorität zuzuordnen. Je nach System können diese verschieden genutzt werden. In Desktop-Systemen werden höher priorisierte Tasks einfach häufiger ausgeführt. In Echtzeitbetriebssystemen hat die höhere Priorität üblicherweise immer Vorrang. Mehrere niedrig priorisierte Tasks wechseln sich nur solange untereinander ab, bis ein Task höherer Priorität lauffähig wird. Dieser belegt den Prozessor dann solange, bis er mit seiner Arbeit fertig ist. [Tha12]

Deadline

Bei dieser Schedulerart bestimmt die am nächsten liegende Deadline der Tasks, welcher von ihnen als nächstes ausgeführt werden soll. Die verschiedenen Implementationen unterscheiden sich darin, wie diese die Deadline bestimmen. Üblicherweise wird diese bei Erstellung des Tasks als Parameter angegeben.

2.3 Notwendigkeiten beim Einsatz mehrerer Tasks

Durch den Einsatz mehrerer Tasks, ergeben sich je nach ihren Aufgaben verschiedene Probleme. Dieser Abschnitt erklärt kurz die gängigen Methoden, mit denen ihnen begegnet wird.

2.3.1 Interprozess-Kommunikation

Wenn mehrere Tasks auf dem selben eingebetteten System ausgeführt werden, ist die Wahrscheinlichkeit, dass ihre Aufgaben voneinander abhängen recht hoch. Um korrekt arbeiten zu können, brauchen sie einen Weg um miteinander kommunizieren und sich synchronisieren zu können.

Eine Möglichkeit dies zu realisieren ist über eine Variable, die beschrieben wird bevor eine Ressource genutzt wird, sodass andere Tasks anhand ihres Inhalts sehen können, ob die Ressource frei ist. Dies setzt voraus, dass die beteiligten Tasks Zugriff darauf haben. Prozesse haben normalerweise komplett getrennte Speicherbereiche. Diese Restriktion kann unter manchen Systemen durch spezielle geteilte Speicherbereiche("Shared Memory") umgangen werden. Im Falle von Threads kann das durch den gemeinsam genutzten Hauptspeicher direkt passieren.

Die einfachste Umsetzung der Kommunikation über diesen geteilten Speicher wäre wie in Listing 2.1 zu verfahren. Einer globalen Variablen *belegt* wird ein Wert zugewiesen, um den anderen Tasks zu signalisieren, dass man eine Ressource belegt hat.

Listing 2.1: Einfache Shared Memory Kommunikation

```
if(belegt==false){
   belegt=true;
   nutzeRessource();
}else{
   wait();
}
```

Diese klare und einfache Herangehensweise funktioniert leider nicht richtig, wenn ein Task direkt nach der Überprüfung der Bedingung, also vor dem setzen der globalen Variable, unterbrochen wird. In diesem Fall kann ein anderer Task genau das gleiche machen und die Ressource belegen. Wenn dieser wiederum unterbrochen wird bevor er die Ressource wieder freigegeben hat, dann kann der erste Task, der ja schon überprüft hat, ob die Ressource frei ist, anfangen sie zu nutzen, obwohl sie ja nun belegt ist.

Dieses Problem kann man dadurch Lösen, dass der entsprechende Task vorher in eine sogenannte "kritische Region" eintritt und sie später wieder verlässt. Das verhindert die Unterbrechung durch andere Threads und damit das Fehlverhalten. Eine solche Funktion, die von mehreren Threads zur gleichen Zeit aufgerufen werden können ohne ihr Verhalten zu ändern, nennt man threadsichere Funktion. Der Vorteil für den Nutzer ist, dass er bei der Programmierung nicht unterscheiden muss, ob eine Funktion auch von anderen genutzt wird oder nicht.

2.3.2 Threadsicherheit

Es ist auf verschiedene Arten möglich, die Threadsicherheit zu gewährleisten. Eine ist das Schreiben von "reinen" Funktionen wie in Listing 2.2.

Listing 2.2: Beispiel einer "reinen Funktion"

```
int add(int a, int b){
  return a+b;
}
```

Für "reine" Funktionen muss folgendes gelten: Sie muss für gleiche Parameter auch immer das gleiche zurückgeben und sie darf keine Seiteneffekte haben. Das heißt, ihr Ergebnis darf nicht von externen Informationen abhängen. Darunter können zum Beispiel globale Variablen, static deklarierte Variablen, oder per Pointer an andere Tasks weitergegebene Variablen fallen.

Eine andere Art die Threadsicherheit zu gewährleisten ist, den Teil der Funktion als kritischen Bereich zu markieren, in dem sich nie mehr als ein Task befinden darf. Diesen Teil zu identifizieren ist aber oft eine schwere Aufgabe. Da die Korrektheit wichtiger ist als die Geschwindigkeit, ist es ratsam diesen Bereich eher etwas zu groß also zu klein zu fassen. Dies hat allerdings zur Folge, dass ein Task, der in diesem Bereich ist, alle anderen Tasks blockiert, bis er ihn wieder verlässt. Ob dies ein Problem darstellt, hängt natürlich von den Aufgaben der anderen Tasks und der Auslastung des Systems ab. Wenn der Task im kritischen Bereich der einzige ist, der eine knappe zeitliche Vorgabe einzuhalten hat, dann leidet nur die Performance der anderen Tasks darunter. Wenn allerdings auch andere Tasks eine Deadline einzuhalten haben, dann kann die Blockade durch den Task im kritischen Bereich, ihre korrekte Funktion beeinträchtigen.

Aufgrund dieser potentiellen Probleme ist sinnvoll den Einsatz von kritischen Bereichen auf ein Minimum zu reduzieren. Einige der später behandelten Systeme bieten einfacher zu nutzende Konstrukte als das simple betreten eines kritischen Bereichs, die dem Programmierer helfen, diese Minimierung zu erreichen.

3 Übersicht bestehender Projekte

Im Internet sind viele Systeme zu finden, die zum Ziel haben auf schwacher Hardware wie den Arduino Mikrocontrollern mehrere Funktionen nebeneinander auszuführen. Die Implementierungen unterscheiden sich sehr stark in ihrem Funktionsumfang, ihrer Aktualität und Lebensdauer, ihrer Kompatibilität mit Hardware und Bibliotheken und ihrer verfügbaren Dokumentation.

Einige Systeme sind lediglich ein bequemes Frontend für die Timer-Funktionen der Arduino-Hardware. Sie Arbeiten intern nicht mit getrennten Tasks. Andere beinhalten einen vollständigen Scheduler, in den meisten Fällen mit dem Round Robin Algorithmus, und eine durch den Einsatz mehrerer Tasks notwendige Speicherverwaltung. Ein paar Systeme bringen neben den Grundfunktionen eines Betriebssystems auch noch weitere Bibliotheken mit, um zusätzliche Funktionen bereitzustellen.

Im Punkt Aktualität und Lebensdauer gibt es ebenfalls große Unterschiede. Viele Projekte sind erst in den letzten drei Jahren entstanden. Die meisten jüngeren Projekte waren allerdings auch recht kurzlebig. An der Commit-Historie ihrer Repositorys war zu erkennen, dass an diesen nur für ein halbes bis ganzes Jahr gearbeitet wurde und dann die Entwicklung zum Stillstand kam. Ältere Systemen wie FreeRTOS oder ChibiOS hingegen werden nach wie vor aktiv weiterentwickelt.

Bei der Kompatibilität gilt im Allgemeinen, dass alle jungen Projekte nur sehr spezifische Hardware unterstützen, wie z.B. nur bestimmte AVR-basierte Arduino-Controller. Diese Einschränkungen ergeben sich dadurch, dass in einem Projekt ausschließlich für diese Controller ein Assembler-Makro zur Sicherung der Registerinhalte geschrieben wurde. Außerdem können fest kodierte Vorgaben im Quelltext im Konflikt mit Vorgaben einer Bibliothek stehen, weil zum Beispiel beide den gleichen Hardware-Timer für ihre Aufgabe nutzen wollen. Ältere Systeme bieten oft mehr Funktionen und Konfigurationsmöglichkeiten, um Inkompatibilitäten entgegenzuwirken.

Auch die Dokumentation ist sehr unterschiedlich und reicht von ein paar Kommentarzeilen im Code bis hin zu einer vollständigen sauberen API-Dokumentation. Manche Projekte ergänzen dies mit Beispielen für den einfacheren Einstieg. Diese unterscheiden sich aber ebenfalls sowohl in Qualität also auch Quantität sehr.

Im Folgenden soll zunächst kurz auf die einfachen Timing-Bibliotheken und danach auf die vollständigen Scheduler eingegangen werden.

3.1 Timing Bibliotheken

Die einfacheren Lösungen kann man als "Timing-Bibliotheken" bezeichen. Sie ermöglichen nicht das Ausführen mehrerer nebeneinander laufender Threads sondern rufen nur zu bestimmten Zeitpunkten bestimmte Funktionen auf. Ein Nutzungsbeispiel für diese Bibliotheken ist das Steuern von LEDs. Drei Funktionen werden auf ein Wiederholungsintervall von $500ms$, $1000ms$ und $1300ms$ eingestellt. Die Bibliothek ruft nun die Funktionen zu den entsprechenden Zeitpunkten auf. Wenn gerade keine der Funktionen auszuführen ist, wartet die Bibliothek.

Bis zu diesem Punkt sind die verschiedenen Implementationen gleich. Manche von ihnen verfeinern das einfache Grundprinzip etwas, indem sie es ermöglichen einen kleinen zeitlichen Offset oder eine Priorität für die Funktionen festzulegen. Dies dient dazu Konflikte, die im oben beschriebenen Beispiel zwangsläufig resultieren, auf eine deterministische Art zu lösen. Anhand des obigen Beispiels lassen sich die potentiell auftretenden Probleme leicht veranschaulichen:

Die erste Funktion mit $500ms$ Intervall soll nach $1000ms$ wieder ausgeführt werden. Die zweite Funktion soll ebenfalls nach $1000ms$ ausgeführt werden. An diesem Punkt entsteht ein Konflikt und es muss entschieden werden, welche der beiden Funktionen als erstes ausgeführt wird.

Zwei einfache Lösungen sind, die Reihenfolge der Ausführung von der Reihenfolge der Definition der Intervalle abhängig zu machen oder den Nutzer selbst bei der Definition Prioritäten festlegen zu lassen. Beides würde allerdings bedeuten, dass der Startzeitpunkt für die zweite Funktion von der Ausführungsdauer der ersten abhängig ist. In manchen Anwendungsfällen ist dies vielleicht unerwünscht.

Die Dritte Lösung versucht dem Problem der ersten beiden entgegenzuwirken, indem jeder Funktion ein Offset hinzugefügt wird. Verschieden festgelegte Offsets sorgen dafür, dass der Konflikt beim Start gar nicht erst entstehen kann. Zu jedem Intervall wird noch ein für jedes Intervall unterschiedlicher zeitlicher Versatz hinzugefügt. Statt den Zeitpunkt der Ausführung der ersten Funktion als $n * 500ms$ zu berechnen wird z.B. $n * 500ms + 200\mu s$ berechnet. Auf alle Aufrufe angewendet können die Startzeitpunkte nicht mehr zusammenfallen. Unter der Voraussetzung, dass die zuerst gestartete Funktion bis zum Startzeitpunkt der zweiten ihre Aufgabe erledigt hat, ist mit dem Offset gewährleistet, dass die Abstände zwischen den Ausführungen der Funktionen immer exakt gleich ist. Wenn jedoch eine Funktion länger braucht als bis zum Startzeitpunkt der nächsten Funktion, dann wird das System wieder nichtdeterministisch.

Als Basis für komplexere Projekte sind diese Bibliotheken ungeeignet, aber für einfache Programme vielleicht genau das richtige, daher sollen sie an dieser Stelle nicht ganz unerwähnt bleiben. Im Prinzip sind sie eine einfache Form eines Deadline-Schedulers mit zwei Nachteilen. Sie sind nicht in der Lage, die Lücken zwischen den einzelnen aufgerufenen Funktionen mit der Ausführung anderer Funktionen zu füllen. Und laufende Funktionen können, z.B. nach einem Interrupt, nicht zu Gunsten anderer Funktionen unterbrochen werden. Wenn dies kein Hindernis darstellt, sind diese Bibliotheken ein akzeptabler Ersatz für einen vollwertigen Deadline-Scheduler. Im Anhang ist eine Liste mit Namen und Internet-Adressen zu finden.

3.2 Vollständige Scheduler

3.2.1 Übersicht

Es gibt eine Vielzahl von verfügbaren Systemen. Sie alle lückenlos zu erfassen ist nur schwer möglich. Hier soll nur eine grobe Übersicht der Systeme gegeben werden. Tabelle 3.1 zeigt die im Internet gefundenen Systeme, die auf kleine eingebettete Systeme ausgerichtet sind. Als letzte Aktivität wurde entweder das letzte Release oder, wenn zugänglich, der letzte Commit im Sourcecode-Repository angenommen. Die Lebenszeit ist die aufgerundete Differenz zwischen erstem und letztem Release bzw. Commit.

Name	Lizenz	Letzte Aktivität	Lebenszeit
arduOS	LGPL	2014-09-26	1
avr-os	Apache	2013-05-27	1
ArDOS	LGPL	2013-10-31	1
Erika Enterprise	GPL[1]	2015-04-22	7
BeRTOS	GPL[1]	2014-11-11	7+[2]
DuinOS	GPL	2013-04-03	3
Nut/OS	BSD kompatible	2015-04-17	12
ScmRTOS	MIT	2015-04-24	9
TinyOS	BSD	2015-04-19	9
ChibiOS	GPL[1]	2015-04-24	8
FreeRTOS	GPL[1]	2015-03-24	12
NilRTOS	GPL	2014-08-16	1
os47 v1.01	MIT	2014-09-05	1

[1] mit Linking-Exception; Kommerzielle Lizenz auch möglich
[2] genaues Alter mangels Informationen unbekannt

Tabelle 3.1: Scheduler-Übersicht

Jedes dieser Systeme käme prinzipiell in Frage als Basis für ein Multitaskingsystem, dass mit der Arduino-Bibliothek zusammen funktioniert. Allerdings haben einige von ihnen, wie am Anfang des Kapitels schon erwähnt, größere Vor- oder Nachteile.

Die erste Gruppe hat den Nachteil, dass an ihnen scheinbar nicht mehr aktiv weiterentwickelt wird. Dadurch ist es notwendig alle zukünftigen Anforderungen selbst zu erfüllen und Fehlerkorrekturen selbst vorzunehmen. Da es andere Systeme mit mehr Aktivität gibt, muss man sich diese zusätzliche Last nicht auferlegen.

Die zweite Gruppe bilden die Projekte die zwar noch aktiv sind, aber aus anderen Gründen weniger attraktiv sind als die darauf folgenden der dritten Gruppe. Einige von ihnen haben sehr spezifische Zielsetzungen, die mit der eher allgemeinen Ausrichtung der Arduino-Plattform in Konflikt stehen könnten. Andere haben eine weniger gute Hardwareunterstützung und/oder Dokumentation.

Die Systeme der dritten Gruppe haben den zusätzlichen Vorteil schon fertig als in der Arduino-IDE verwendbare Bibliothek verpackt worden zu sein. Dadurch sind sie sofort ohne zusätzlichen Portierungsaufwand Einsatzbereit.

3.2.2 Einzelbetrachtungen

Erika Enterprise

Erika Enterprise ist ein System aus dem Automotive-Bereich. Es ist das nach eigener Aussage erste kostenlose offene Echtzeitbetriebssystem mit einer OSEK/VDX-Zertifizierung. [Srl14b] Es liefert nicht nur eine eigene Bibliothek sondern verpackt diese gleich in eine komplette Eclipse-IDE mit diversen zusätzlichen Features. Die Dokumentation ist online als PDF verfügbar. Offiziell unterstützt dieses System nur den Arduino Uno. [Srl14a]

Für das Ziel dieser Arbeit ist das System ungeeignet, weil das System als Bibliothek für die Arduino-IDE verpackt werden soll und den Arduino Due unterstützen soll. Im Fall von Erika Enterprise wäre dies mit sehr hohem Aufwand verbunden.

BeRTOS

BeRTOS bringt wie Erika Enterprise eine eigene IDE mit. Allerdings sind hier die IDE und das System in der Ordnerstruktur einfacher voneinander zu trennen, da das Projekt eine relativ gute Dokumentation hat. Außerdem sind einige zusätzliche Bibliotheken für externe Hardware und grafische Oberflächen im Paket enthalten. Diese enthält ebenfalls eine Anleitung zur Portierung des Systems auf andere Mirkrocontroller, sodass das Fehlen einer Unterstützung für die meisten Arduino-Boards ausgebessert werden könnte. [Srl11]

DuinOS

DuinOS ist die Kombination der Arduino-Bibliothek mit FreeRTOS. Es erfüllt also im Prinzip die Anforderungen dieser Arbeit. Allerdings wurde FreeRTOS nicht als Bibliothek für die Arduino-IDE verpackt, sodass man es mit einer *include*-Direktive einbinden kann, sondern direkt mit der Arduino-Bibliothek verwoben. Wenn Wert auf aktuelle Versionen gelegt wird, muss jedes Update der IDE oder des Betriebssystems direkt von den Entwicklern in DuinOS integriert werden. Tun sie dies nicht, hat man den Nachteil einer veralteten Komponente. Außerdem wurde FreeRTOS so eingebunden, dass in der IDE jedes Board doppelt auftaucht, einmal mit und einmal ohne DuinOS. All dies macht den Einsatz des Systems mühsamer und komplizierter als es sein muss. Darüber hinaus wird DuinOS nur noch inklusive der angepassten IDE verteilt.

Nut/OS

Nut/OS legt besonderen Wert auf die Kommunikation über TCP/IP und unterstützt DHCP, DNS und HTTP. Es hat nur einen kooperativen Scheduler. Eine offizielle Unterstützung für Arduino-Boards gibt es nicht, wäre allerdings durch eine Anleitung zum Portieren des Systems auf andere Controller machbar. Das Projekt hat ein Wiki, dass in englischer Sprache recht umfangreich ist. Es hat auch einen deutschen Teil, welcher sehr viel kleiner ausfällt und viele Punkte unbeantwortet lässt. Allerdings ist es das einzige der hier gelisteten Projekte, dass überhaupt eine deutsche Dokumentation hat. [eG10]

ScmRTOS

ScmRTOS ist eines der wenigen Systeme, die in C++ geschrieben sind. Über die Zielsetzung des Projekts sagt die Webseite des Projekts leider nichts aus. Eine offizielle Unterstützung der Arduino-Boards gibt es nicht, ähnliche Boards werden aber unterstützt. Statt einer detaillierten Portierungsanleitung wie bei vielen anderen Projekten gibt es hier nur eine Auflistung der einzelnen Bestandteile eines Ports. Die Portierung dieses Systems wäre wahrscheinlich etwas aufwendiger im Vergleich zu den anderen Systemen. Die Dokumentation ist in Form eines PDFs vorhanden, welches alle Themen abdeckt. [sT12]

ChibiOS

ChibiOS hat einen besonderen Fokus auf Effizienz. Nach eigenen Angaben hat es die besten Kontextwechselzeiten in seiner Klasse. [Sir15] Das System hat neben den Grundfunktionen eines Echtzeitbetriebssystems auch einen Hardware Abstraction Layer, der Unterstützung für verschiedene externe Hardware bereitstellt. Es hat eine Online-Dokumentation, die sämtliche Komponenten des Systems detailliert erklärt. Das projekteigene Wiki enthält Artikel zu vielen Themen rund um die Benutzung des Systems und auch über allgemeines wie die Grundkonzepte von Echtzeitbetriebssystemen oder das Optimieren von Programmen für eingebettete Systeme. Eine offizielle Unterstützung für die Arduino-Boards gibt es nicht, aber es existiert eine Portierungsanleitung und ähnliche Boards werden unterstützt. Allerdings ist die Portierung gar nicht notwendig.

Dieses System ist schon fertig als Bibliothek für die Arduino-IDE von Bill Greiman verpackt worden und steht zum Download bereit. Die enthaltene Version 2.6.5 ist zwar nicht mehr ganz aktuell, sollte aber bei Bedarf unkompliziert durch die derzeit aktuelle Version zu ersetzen sein. Laut der Github-Seite ist es sowohl zu den alten Arduino-Boards als auch zum Due kompatibel. [Gre14a]

FreeRTOS

FreeRTOS zeichnet sich durch eine extrem breite Hardwareunterstützung und eine Vielzahl von Beispiel-Applikationen aus. Die Dokumentation ist sehr reichhaltig und detailliert. Im Vergleich mit den anderen Projekten hat FreeRTOS genausten und umfangreichsten Anleitungen für Einsteiger. Außerdem spricht die sehr weite Verbreitung des Systems für dessen Einsatz. Nach einer Umfrage von EE Times hat FreeRTOS einen Marktanteil von 12%. [PC13] Durch diese weite Verbreitung ist auch das Angebot an Material zum Thema FreeRTOS im Internet sehr viel höher als bei den anderen hier gelisteten Systemen.

Wie ChibiOS steht dieses System schon als Bibliothek für die Arduino-IDE verpackt zur Verfügung und ist sowohl mit alten Boards als auch mit dem Due kompatibel. Die verpackte Version 8.0.1 schon ein Jahr alt, aber ein Upgrade sollte auch hier nicht schwer sein. [Gre15]

NilRTOS

NilRTOS ist ein Experiment des Entwicklers von ChibiOS, Giovanni Di Sirio, mit dem Ziel herauszufinden wie klein man ein Echtzeitbetriebssystem machen kann ohne essenzielle Funktionalität aufzugeben. Es ist eine Untermenge von ChibiOS und API-kompatibel zu ihm. Der Speicherbedarf im ROM ist mit rund 700 Bytes in etwa ein Zehntel dessen, was ChibiOS für sich beansprucht. Die Dokumentation fällt im Gegensatz zu der von ChibiOS sehr kurz aus. [Sir]

Auch dieses System steht schon fertig als Bibliothek für die Arduino-IDE zur Verfügung. Im Bezug auf die Hardwareunterstützung werden nur die alten AVR-Controller basierten Arduino-Boards erwähnt. Aufgrund der experimentellen Natur des Projekts und der geringen Aktivität ist unklar, ob dieses System in Zukunft weiter gepflegt wird. [Gre14b]

os47

os47 ist ein in C++ für den Arduino Uno geschriebenes System, dass auch auf anderen AVR-Prozessor basierten Problemlos laufen sollte. Für einen Betrieb auf dem Arduino Due müsste man es auf dessen ARM-Architektur portieren. Das Projekt hat zwar keine Anleitung dafür, ist aber extrem gut kommentiert, sodass der Aufwand dafür sich trotz allem in Grenzen hält. Der Funktionsumfang des Systems ist aufgrund des jungen Alters nicht so groß wie bei den anderen Systemen. [Dem14]

Leider ist seit der letzten Aktivität schon fast ein Jahr vergangen, sodass eine Weiterentwicklung durch den ursprünglichen Autor unwahrscheinlich ist. Für einen kleinen Einblick in den Aufbau eines minimalen Echtzeitbetriebssystems ist dieses Projekt aber dennoch gut geeignet.

3.2.3 Auswahl

Das in dieser Arbeit geplante System soll besonders für Einsteiger gut geeignet sein. Das heißt, dass es wenig Vorkenntnisse erfordern darf, unkompliziert einzusetzen ist und eine möglichst gute Dokumentation haben muss. Es soll in der Arduino-IDE verwendbar sein und möglichst keine zusätzlichen externen Abhängigkeiten einführen. Außerdem soll es neben dem Arduino Due möglichst viele andere Arduino-Boards unterstützen.

FreeRTOS ist bereits als Arduino-IDE kompatible Bibliothek verpackt erhältlich und hat eine breite Hardwareunterstützung. Die Dokumentation ist vorbildlich und es gibt eine Vielzahl von Beispielen, die den Einstieg erleichtern. Auf ChibiOS treffen diese Punkte generell auch zu. Es ist ebenfalls direkt in der Arduino-IDE einsatzbereit und hat eine gute Dokumentation. Aber die mitgelieferten Beispiele sind sehr viel komplexer und damit schwerer zu verstehen als diejenigen, die FreeRTOS beiligen. Außerdem ist das Finden von Informationen im Internet im Zusammenhang mit FreeRTOS einfacher, weil es weiter verbreitet ist.

FreeRTOS ist damit am Ende die bessere Wahl für die Anforderungen.

4 Integration mit der Arduino-Bibliothek

4.1 Probleme

Dieser Abschnitt geht auf Probleme ein, die bei der Nutzung eines Multitaskingsystems zusammen mit der Arduino-Bibliothek entstehen können. Die Funktionen der Arduino-Bibliothek sind nicht auf die gleichzeitige Nutzung durch mehrere Tasks ausgelegt. In manchen Fällen kann dies zu Problemen führen.

Wenn Funktionen geteilte Ressourcen nutzen, kann ein gleichzeitiger Aufruf zu fehlerhaftem Verhalten führen. Ein Beispiel dafür ist die *malloc*-Funktion, die der Speicherallokation dient. Wenn zwei Tasks sie gleichzeitig aufrufen, führt dies im schlimmsten Fall dazu, dass an beide der gleiche Speicherbereich herausgegeben wird, sodass die beiden Tasks gegenseitig ihre Daten überschreiben würden. Ein weiteres Beispiel ist die Funktion zum Senden über die serielle Schnittstelle. Wenn zwei Funktionen diese gleichzeitig nutzen, würden die Nachrichten der beiden vermischt werden. Mehr dazu in den Einzelbetrachtungen in den Abschnitten 4.3.1 und 4.3.3.

Funktionen mit strikten zeitlichen Anforderungen können selbst ohne gleichzeitigen Aufruf fehlerhaftes Verhalten verursachen, wenn ein anderer Thread ihre Ausführung für eine kurze Zeit unterbricht. Es konnte kein Beispiel für diese Problematik gefunden werden. Es wäre allerdings denkbar, dass in einer der zahllosen externen Bibliotheken ein solcher Fall auftritt, wenn z.B. ein Kommunikationsprotokoll nach einer bestimmten Zeit eine Antwort erwartet.

Mehr oder weniger ein Spezialfall der oben beschriebenen Probleme sind Funktionalitäten, die sowohl in der Arduino-Bibliothek als auch im Multitaskingsystem enthalten sind. Dabei handelt es sich im Allgemeinen um Funktionen aus zwei Bereichen.

Der erste ist die Zeiteinteilung, welche die zentrale Aufgabe des Multitaskingsystems ist. Ein Beispiel dafür ist die *delay*-Funktion, welche später in Abschnitt 4.3.2 genauer betrachtet wird. Diese Funktion dient dem Warten um eine bestimmte Zeit. Sie kann effizienter implementiert werden, indem man die zusätzlichen Randbedingungen durch den Einsatz mehrerer Tasks berücksichtigt.

Der zweite Bereich ist die Speicherverwaltung, welche das System ebenfalls übernehmen muss, weil die mitgelieferten Funktionen des AVR-GCC, den die Arduino-IDE nutzt, nicht für den gleichzeitigen Aufruf aus mehreren Tasks ausgelegt ist. Ein Beispiel dafür ist das weiter oben schon angesprochene *malloc*.

4.2 Lösungsmöglichkeiten

Man kann den Integrationsproblemen auf mehrere Arten begegnen, welche sich darin unterscheiden, an welcher Stelle man die Probleme angeht und wer für die Lösung verantwortlich ist. Keine der Optionen ist allerdings ein Allheilmittel. Je nach Funktion ist eine besser geeignet als die anderen.

4.2.1 Arduino-Bibliothek erweitern

Eine Variante ist die Behandlung der besonderen Anforderungen beim Einsatz mehrerer Threads in der Arduino-Bibliothek. Da die Bibliothek auch weiterhin auf den alten Arduinos und ohne Threads sinnvoll Einsetzbar bleiben soll, muss aber darauf geachtet werden, dass der Ressourcenverbrauch nicht, oder zumindest nur sehr begrenzt, anwächst. Das gilt für die benötigten Takte zur Abarbeitung, für den Arbeitsspeicher und auch für den persistenten Speicher, in dem das Programm liegt. Eine Möglichkeit dies zu gewährleisten wäre die Bereitstellung von zwei Funktionsvarianten. Wenn ein Multitaskingsystem genutzt wird, kommt die angepasste threadsichere Variante zum Einsatz, sonst die einfache.

Auf lange Sicht ist der Weg die Bibliothek zu erweitern für viele Funktionen wahrscheinlich die beste Lösung. Ein Fall davon ist die später in 4.3.3 betrachtete *Serial*-Klasse. Allerdings ist der Umfang der Arduino-Bibliothek groß genug, dass eine Überarbeitung der gesamten Bibliothek einen großen Zeitraum in Anspruch nehmen würde.

4.2.2 Arduino-Funktionen ersetzen

Eine weitere Möglichkeit den Problemen zu begegnen ist das Überschreiben von Funktionen der Arduino-Bibliothek mit angepassten Varianten durch das Multitaskingsystem. Dies kann auf zwei Arten geschehen.

Multitaskingsystem ersetzt Funktionen

Um die Ersetzung von Seiten des Multitaskingsystems zu realisieren, muss man in der Arduino-Bibliothek alle Funktionen, für die ein Überschreiben ermöglicht werden sollen, als schwach deklarieren. Dies teilt dem Compiler mit, dass bei einem Namenskonflikt, also zwei Funktionen mit gleichem Namen, kein Fehler ausgegeben werden soll, sondern dass stattdessen die schwache Funktion ignoriert werden soll.

Beim von der Arduino-IDE genutzen AVR-GCC geht dies durch das Hinzufügen von *__attribute__((weak))* vor dem Funktionsnamen oder nach der Parameterliste der Funktion oder durch das Hinzufügen von *#pragma weak <Funktionsname>* in der Zeile vor der Funktionsdefinition. Ein Beispiel dafür kann man in Listing 4.6 sehen. die *yield*-Funktion ist als schwach deklariert.

Arduino-Bibliothek ersetzt Funktionen

Eine Alternative zur Deklarartion von schwachen Funktionen, die auch jetzt schon in manchen Teilen der Arduino-Bibliothek zu finden ist, ist in Listing 4.1 zu sehen. In der Bibliothek für ein WiFi-Shield wird eine Funktion *delay_ms* definiert, die je nach genutztem System verschiedene Wartefunktionen aufruft. Wenn FreeRTOS genutzt wird, ruft sie *vTaskDelay* auf. Diese Funktion teilt gibt die Kontrolle an den Scheduler von FreeRTOS ab und teilt ihm mit, dass dieser Task die nächsten *delay* Millisekunden nicht aktiviert werden soll. Für den Fall, dass NutOS verwendet wird, geschieht in etwa das gleiche mit er entsprechenden Funktion von NutOS. Wenn keines der beiden Systeme verwendet wird, dann wird stattdessen die Bibliothekseigene *cpu_delay_ms*-Funktion aufgerufen, die wiederum, ähnlich der Wartefunktion aus Listing 4.7 per Busy-Waiting die gewünschte Zeit verstreichen lässt.

Listing 4.1: Ausschnitt der delay.c(wifiHD) der Arduino-Bibliothek

```
void delay_ms(unsigned long delay)
{
#if (defined FREERTOS_USED)
  vTaskDelay( (portTickType)TASK_DELAY_MS(delay) );
#elif (defined NUTOS_USED)
  NutSleep(delay);
#else
  cpu_delay_ms(delay, s_fcpu_hz);
#endif

}
```

Diese Herangehensweise hat den Nachteil, dass die entsprechenden Funktionen der verschiedenen Multitaskingsysteme alle nach dem oben zu sehenden Muster in der Arduino-Bibliothek eingetragen werden müssen, damit sie verwendet werden können. Jedes System müsste sich für die Aufnahme in die Liste an die Arduino-Entwickler wenden. Die Entwickler des Multitaskingsystems wären von denen der Arduino-Bibliothek abhängig. Die Liste würde langsam aber stetig anwachsen.

Solche Änderungen auf der anderen Seite in allen Multitaskingsystemen vorzunehmen erscheint allerdings auch nicht sinnvoll. Dies würde nur den die Positionen umdrehen und den Entwicklern des Systems auferlegen eine solche Liste zu führen.

Auf beiden Seiten würde der Code durch solche Anpassungen unübersichtlich werden und die eigentliche Funktionalität verbergen. Deswegen sollten die Probleme wenn möglich in ein externes Paket ausgelagert werden, damit beide Seiten sich auf ihre Kernaufgaben konzentrieren können.

Erstellung einer Zwischenschicht

Der Mittelweg zwischen der Ersetzung nur auf der einen oder der anderen Seite, und eine wahrscheinlich für alle sauberere Lösung, wäre die Erstellung einer Zwischenschicht, welche die Brücke zwischen Arduino-Bibliothek und Multitaskingsystem schlägt. So könnte man die Quelltexte der beiden Komponenten weitestgehend unangetastet lassen und die notwendigen Modifikationen in die Zwischenschicht auslagern. Wie dies aussehen könnte zeigt Abschnitt 4.3.1, der sich mit der *malloc*-Funktion beschäftigt.

4.2.3 Nutzer verantwortlich machen

Die dritte Alternative ist das Abschieben der Verantwortung auf den Programmierer, der basierend auf dem System und der Bibliothek versucht ein eigenes Programm zu schreiben. Man könnte in der Dokumentation der Arduino-Bibliothek ganz einfach den gefährdeten Funktionen einen Hinweis hinzufügen, der den Nutzer darauf aufmerksam macht, dass sie nicht threadsicher ist. Dies erfordert auf Seiten des Programmierers ein tiefes Verständnis aller Abläufe in den Funktionen, um mit dem System und der Bibliothek zu arbeiten. Er muss die Nebeneffekte in seinem Gesamtsystem abschätzen können und anhand dessen entscheiden, ob beziehungsweise welche Maßnahmen zu ergreifen sind.

So zu verfahren würde im krassen Widerspruch zur Arduino-Philosophie der Einfachheit stehen. Im allgemeinen kann diese Lösung deshalb nur eine temporäre Hilfsmaßnahme sein, um die Zeit zu überbrücken, bis eine saubere Lösung außerhalb des Verantwortlichkeitsbereichs des Nutzers des Systems gefunden wurde. Dies setzt selbstverständlich voraus, dass es eine solche gibt.

Es gibt allerdings Fälle, in denen es schwer bis gar nicht möglich ist, dem Nutzer die Arbeit abzunehmen, weil die auftretenden Probleme nicht unbedingt ein Fehler der Funktion sind. Ein Beispiel dafür ist das Lesen von der seriellen Schnittstelle. Wenn zwei Tasks gleichzeitig von ihr lesen, dann bekommen beide nur jeweils einen Teil der ankommenden Daten. Auf Hardware- oder Betriebssystemseite ist es aber nicht möglich zu wissen, welche Teile der Daten einem bestimmten Task zugeordnet werden müssen. Der Programmierer muss hier selbst sorge dafür tragen, dass sein System sich korrekt verhält und dass die Eingaben über die serielle Schnittstelle richtig zugeordnet werden.

4.3 Einzelbetrachtungen

4.3.1 malloc()-Funktion

Die Arduino-IDE nutzt für die alten Boards den AVR-GCC und für den Due einen angepassten gcc. Beide haben liefern eine *libc*-Implementation mit, welche auch die Funktion *malloc* enthält. Die *malloc*-Funktion dient der dynamischen Allokation von Speicher. Eine "Freelist" genannte Liste enthält die Startadressen der freien Speicherstellen im Arbeitsspeicher. Listing 4.2 zeigt einen Ausschnitt aus der *malloc*-Funktion der *avr-libc*. In diesem sieht man den Teil der Funktion, der über der Liste iteriert, um für eine Speicheranforderung eine möglichst geeignete Stelle zu finden.

Listing 4.2: Ausschnitt aus malloc.c (avr-libc)

```
  for (s = 0, fp1 = __flp, fp2 = 0;
  fp1;
  fp2 = fp1, fp1 = fp1->nx) {
    if (fp1->sz < len)
    continue;
->  if (fp1->sz == len) {
      /*
      * Found it.  Disconnect the chunk from the
      * freelist, and return it.
      */
      if (fp2)
      fp2->nx = fp1->nx;
      else
      __flp = fp1->nx;
      return &(fp1->nx);
    }
    else {
      if (s == 0 || fp1->sz < s) {
        /* this is the smallest chunk found so far */
        s = fp1->sz;
        sfp1 = fp1;
        sfp2 = fp2;
      }
    }
->}
```

An diesem Ausschnitt kann man erkennen, dass die Implementation nicht threadsicher ist. Es gibt zwei Zeilen, hervorgehoben durch einen Pfeil, an denen die Unterbrechung eines Tasks ein fehlerhaftes Verhalten hervorrufen kann.

Der Ablauf des Fehlers ist wie folgt:

- Task 1 ruft *malloc* auf und findet eine geeignete freie Stelle im Speicher.

  Task 1 befindet sich in diesem Moment an der ersten oder der zweiten markierten Zeile, je nach dem, ob die Stelle exakt so groß ist wie gefordert oder ob sie größer ist und in zwei Teile aufgeteilt werden muss.

- Task 1 wird unterbrochen.

- Task 2 ruft *malloc* auf und findet die selbe Stelle im Speicher wie Task 1.

- Beide Tasks werden die gefundene Stelle im Folgenden für sich beanspruchen.

Bei genauerer Betrachtung des Rests der Funktion könnte man möglicherweise noch mehr Fehlerquellen im Zusammenhang mit gleichzeitigen Aufrufen finden. Dieses Beispiel soll jedoch genügen, um das Problem aufzuzeigen.

FreeRTOS stellt fünf Implementationen zur Speicherallokation zur Verfügung, die jeweils auf bestimmte Anwendungsfälle angepasst sind. Um das System zu betreiben, muss man sich für eine von ihnen entscheiden, denn FreeRTOS nutzt ausschließlich diese interne Funktion zur Verwaltung seines Speichers und setzt voraus, dass sie definiert ist. [Ltdb] Im von Bill Greiman bereitgestellten Paket *FreeRTOS-Arduino* wird die Variante genutzt, die auf eine schon definierte *malloc*-Funktion zurückgreift, diese aber gegen gleichzeitigen Aufruf zu schützen versucht. Listing 4.3 zeigt die Funktion.

Listing 4.3: Ausschnitt aus heap_3.c (FreeRTOS)

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;

    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif

    return pvReturn;
}
```

Vor dem Aufruf von *malloc* werden hier alle Tasks pausiert. Nach dem Aufruf werden sie wieder freigegeben. Wenn alle Tasks *pvPortMalloc* nutzen statt direkt *malloc* aufzurufen, dann ist hiermit das Problem gelöst, denn es wird zwischen dem Pausieren und der Freigabe kein anderer Task ausgeführt. Ein einzelner Task, der den direkten Aufruf nutzt, kann diese Lösung allerdings aushebeln. Wenn der Ablauf wie oben bei dem einfachen *malloc* ist und nur Task 2 *pvPortMalloc* benutzt, dann kann der Konflikt wieder auftreten.

Im eigenen Code kann man eventuell noch überschauen, wo Speicher alloziert wird und die entsprechenden Aufrufe ersetzen. Problematischer wird es allerdings, wenn eine genutzte Bibliothek Speicher alloziert. Man müsste jede genutzte Bibliothek überprüfen und gegebenenfalls die *malloc*-Aufrufe ersetzen. Das hätte aber zur Folge, dass die Bibliothek nur noch in Kombination mit FreeRTOS nutzbar wäre. Wenn man ein einfaches Projekt hat, bei dem der Einsatz eines Betriebssystems nicht gewünscht ist, müsste man die Anpassungen wieder entfernen. Das gleiche Problem entstünde auch, wenn man ein Projekt hätte, bei dem ein anderes System als FreeRTOS eingesetzt werden soll.

Auf lange Sicht kann das manuelle Ersetzen der *malloc*-Aufrufe also keine Lösung sein. Besser wäre es, wenn man wie in Kapitel 4.2.2 beschrieben die Funktion als schwach deklarieren würde und in einer Zwischenschicht die Zuordnung des *malloc*-Namens zur Funktion *pvPortMalloc*. Listing 4.4 zeigt die kleine notwendige Änderung, um die Funktion schwach zu deklarieren und Listing 4.5 zeigt die Zuordnung der spezialisierten *malloc*-Implementation zum allgemeinen Namen.

Listing 4.4: *malloc*-Deklaration in der *libc*

```
Original: void * malloc(size_t len) {...}

    Neu: void * malloc(size_t len) __attribute__ ((weak)) {...}
```

Listing 4.5: Zuordnung der alternativen *malloc*-Implementation

```
void * malloc(size_t len) __attribute__ ((alias("pvPortMalloc")));
```

Bei dieser Herangehensweise wäre es beliebigen anderen Systemen möglich das gleiche zu tun, ohne dass von Seiten der Arduino-Entwickler ein weiteres Eingreifen nötig wäre.

4.3.2 delay()- und yield()-Funktionen

Listing 4.6 zeigt die schwach deklarierte Funktion *yield*. Diese Funktion dient der Abgabe der Kontrolle eines Threads an einen anderen und ist an dieser Stelle vorerst leer definiert. Ein externer Scheduler kann nun diese leere Funktion durch eine eigene ersetzen.

Listing 4.6: Ausschnitt aus hooks.c der Arduino-Bibliothek

```
/**
 * Empty yield() hook.
 *
 * This function is intended to be used by library writers to build
 * libraries or sketches that supports cooperative threads.
 *
 * Its defined as a weak symbol and it can be redefined to implement a
 * real cooperative scheduler.
 */
static void __empty() {
    // Empty
}
void yield(void) __attribute__ ((weak, alias("__empty")));
```

Welchen Zweck die Ersetzung hat, kann man in Listing 4.7 sehen. Die *delay*-Funktion der Arduino-Bibliothek wartet *ms* Millisekunden. Mit der leeren *yield*-Funktion läuft diese Schleife immer wieder in schneller Folge ab bis die entsprechende Zeit abgelaufen ist. Dieses Verfahren ist auch als "Busy-Waiting" bekannt. Mit einer überschriebenen *yield*-Funktion von einem Scheduler ist es möglich an dieser Stelle anderen Tasks die Kontrolle zu übergeben, während der aktuelle Task wartet. Dadurch wird die Zeit statt mit häufigen Abfragen der Bedingungen von *while* und *if* mit der Ausführung anderer Tasks, und damit sinnvoller, gefüllt.

Listing 4.7: Ausschnitt aus wiring.c der Arduino-Bibliothek

```
void delay(unsigned long ms){
    uint16_t start = (uint16_t)micros();
    while (ms > 0) {
        yield();
        if (((uint16_t)micros() - start) >= 1000) {
            ms--;
            start += 1000;
        }
    }
}
```

Die Grundidee *yield* in *delay* zu nutzen ist sehr gut, denn so können bestehende Funktionen mit Wartezeiten mit einem einfachen Scheduler unkompliziert nebeneinander ausgeführt werden. Das MultipleBlink-Tutorial von arduino.cc [ard] ist ein schönes Beispiel, wie dies angewendet werden kann. Die Implementation der *delay*-Funktion ist allerdings schlecht, da sie nur unter bestimmten Bedingungen korrekt funktioniert.

In jedem Durchlauf der *while*-Schleife wird am Anfang die Kontrolle abgegeben. Wenn später die Ausführung fortgesetzt wird, wird einmal überprüft, ob mehr als $1000\mu s$ vergangen sind. Falls ja, wird der *ms*-Zähler dekrementiert.

Diese Logik setzt voraus, dass bei einem Aufruf mit $100ms$ die Kontrolle mindestens 100 mal innerhalb der zu wartenden Zeit wiedererlangt wird. Wenn dies nicht der Fall ist, dann kann der Zähler noch nicht bei Null angekommen sein. Wenn der Task zum Beispiel in der Zeit nur 50 mal ausgeführt wurde, dann ist der Zähler erst auf 50 reduziert worden. Das hat zur Folge, dass mehr als die angeforderte Zeit gewartet wird. Wie lange dies genau ist, hängt einzig vom Verhalten der anderen Tasks ab. Dieses Problem entsteht, weil nur überprüft wird, ob mehr als $1000\mu s$ vergangen sind, nicht aber, wieviel mehr. Das Fehlverhalten kann einfach korrigiert werden, indem man das *if*-Statement durch ein *while*-Statement ersetzt. In dem Fall würde der Zähler immer so oft dekrementiert, bis die Differenz kleiner als 1000 ist. Danach entspräche der Zähler wieder dem zu diesem Zeitpunkt passenden Wert. Erst dann würde die Kontrolle wieder abgegeben werden.

Neben diesem Fehler in der Logik der Funktion ist allerdings auch die Tatsache ein Nachteil, dass sie nicht als schwach deklariert ist und damit nicht ersetzt werden kann wie *yield*. In dieser Form ist für die Überprüfung, ob die Zeit abgelaufen ist, sehr viel Aufwand notwendig. Es muss einen Kontextwechsel in den Task geben, dann die Bedingung geprüft werden und danach wieder aus ihm heraus gewechselt werden. Mit der aktuellen Fassung der Funktion muss dies im oben skizzierten Beispiel mindestens 100 mal passieren. Selbst mit der empfohlenen Anpassung könnte dies noch beliebig oft passieren, wenn die anderen Tasks oft die Kontrolle zurückgeben.

Deswegen wäre es weit aus effizienter die Zeiterfassung in den Scheduler zu verlagern. Dieser könnte die gleichen Berechnungen und Überprüfungen ohne die Kontextwechsel davor und danach anstellen.

4.3.3 Serial-Klasse

Das in vielen Beispielen der Arduino-Bibliothek genutzte *Serial*-Objekt ist vom Typ *HardwareSerial*. Es dient der einfachen Benutzung der seriellen Schnittstelle.

Listing 4.8 zeigt die Definition der Klasse in gekürzter Form. Hier ist zu sehen, dass alle direkt in der Klasse definierten Methoden ihren Parameter an die *write*-Methode für den Typ *uint8_t* weitergeben. Die Varianten für die Behandlung von Strings werden aus der *Print*-Klasse übernommen. Listing *lst:print-write* zeigt ihre Implementation. Die erste Methode reicht ihren Parameter, einen String, und die Länge des Strings weiter an die zweite Methode. Die zweite Methode ruft wiederum für jedes Zeichen des Strings die *write*-Methode für den Typ *uint8_t* auf. Die Methode für *uint8_t*-Werte wird also für alle Sendeoperationen verwendet.

Listing 4.8: Gekürzter Ausschnitt aus HardwareSerial.h der Arduino-Bibliothek

```
class HardwareSerial : public Stream
{     ....
    virtual size_t write(uint8_t);
    inline size_t write(unsigned long n) { return write((uint8_t)n); }
    inline size_t write(long n) { return write((uint8_t)n); }
    inline size_t write(unsigned int n) { return write((uint8_t)n); }
    inline size_t write(int n) { return write((uint8_t)n); }
    using Print::write; // pull in write(str) and write(buf, size) from Print
    ....
};
```

Listing 4.9: Ausschnitte aus Print.h und Print.cpp der Arduino-Bibliothek

```
size_t write(const char *str) {
  if (str == NULL) return 0;
  return write((const uint8_t *)str, strlen(str));
}

size_t Print::write(const uint8_t *buffer, size_t size){
  size_t n = 0;
  while (size--) {
    n += write(*buffer++);
  }
  return n;
}
```

Die in Listing 4.9 gezeigte Methode *Print::write(const uint8_t *buffer, size_t size)* ist nicht threadsicher. Dies kann man leicht an einem Beispiel erkennen. Ein Task, der sie nutzt, versendet die Zeichen des übergebenen Strings einzeln der Reihe nach. Wenn er dabei von einem anderen Task unterbrochen wird und dieser seinerseits Daten über die serielle Schnittstelle sendet, dann werden die Daten der beiden Tasks vermischt. Der Empfänger erhält zunächst den ersten Teil der Nachricht des ersten Tasks, dann einen Teil oder die komplette Nachricht des zweiten Tasks und am Ende den Rest der Nachricht des ersten Tasks. Damit ist mindestens die erste Nachricht zerstört. Es ist also nicht gesichert, dass eine Nachricht korrekt gesendet wird, wenn mehr als ein Task auf die serielle Schnittstelle zugreift.

Die in Listing 4.10 gezeigte *write*-Methode ist ebenfalls nicht threadsicher. Es kann an mehreren Stellen zu Fehlverhalten kommen.

Die erste Bedingung prüft, ob sofort gesendet werden kann. Wenn dies der Fall ist, dann wird der zu sendende Wert in das Datenregister geschrieben und anschließend ein Bit gesetzt, dass dem Mikrocontroller singalisiert, dass er ein Zeichen senden soll. Unterbricht ein anderer Task den Ablauf genau nach dem füllen des Datenregisters und sendet seinerseits etwas über die Schnittstelle, dann überschreibt er dabei den Wert des ersten Tasks. Wenn danach der erste Task seine Tätigkeit fortsetzt und das Sendebit setzt, dann wird statt seines eigenen Wertes der Wert seines Vorgängers gesetzt. Seine Information geht also verloren.

Listing 4.10: Ausschnitt aus HardwareSerial.cpp der Arduino-Bibliothek

```
size_t HardwareSerial::write(uint8_t c)
{
  // If the buffer and the data register is empty, just write the byte
  // to the data register and be done. This shortcut helps
  // significantly improve the effective datarate at high (>
  // 500kbit/s) bitrates, where interrupt overhead becomes a slowdown.
  if (_tx_buffer_head == _tx_buffer_tail && bit_is_set(*_ucsra, UDRE0)) {
    *_udr = c;
    sbi(*_ucsra, TXC0);
    return 1;
  }
  tx_buffer_index_t i = (_tx_buffer_head + 1) \% SERIAL_TX_BUFFER_SIZE;

  // If the output buffer is full, there's nothing for it other than to
  // wait for the interrupt handler to empty it a bit
  while (i == _tx_buffer_tail) {
    if (bit_is_clear(SREG, SREG_I)) {
      // Interrupts are disabled, so we'll have to poll the data
      // register empty flag ourselves. If it is set, pretend an
      // interrupt has happened and call the handler to free up
      // space for us.
      if(bit_is_set(*_ucsra, UDRE0))
    _tx_udr_empty_irq();
    } else {
      // nop, the interrupt handler will free up space for us
    }
  }

  _tx_buffer[_tx_buffer_head] = c;
  _tx_buffer_head = i;

  sbi(*_ucsrb, UDRIE0);
  _written = true;

  return 1;
}
```

5 Fazit

Die von Bill Greiman als Bibliothek für die Arduino-IDE verpackte Version von FreeRTOS entspricht schon von sich aus in etwa den Anforderungen dieser Arbeit. FreeRTOS-Arduino ist sofort im Lehrbetrieb und im Hobby-Bereich einsetzbar. Die Grundfunktionen stehen bereit und das System ist mit allen Arduino-Boards kompatibel. Es gelten dabei allerdings einige Einschränkungen, die berücksichtigt werden müssen, weil manche Funktionen fehlerhaft oder ungenau arbeiten können. Die Arduino-Bibliothek muss noch an einigen Stellen überarbeitet werden, um die problemlose Nutzung zu gewährleisten.

Die fehlende Threadsicherheit der Speicherallokation durch die *malloc*-Funktion bedeutet, dass bei der Benutzung dieser Funktion immer die Gefahr eines Fehlverhaltens besteht. An dieser Stelle sind die Arduino-Entwickler gefordert ihre mitgelieferten Versionen der *libc* anzupassen, sodass externe Bibliotheken unkompliziert eine sichere Variante der *malloc*-Funktion bereitstellen können.

Gleiches gilt für die *delay*-Funktion, deren korrekte Funktion vom Verhalten aller aktiven Tasks abhängig ist. Statt externen Bibliotheken ein einhängen innerhalb der Funktion zu ermöglichen sollten sie ein komplettes Ersetzen der Funktion zulassen. Dadurch könnten die Bibliotheken optimierte Versionen anbieten, die je nach Anwendungsfall entweder die Performance erhöhen oder den Stromverbrauch verringern könnten.

Wenn diese beiden Forderungen erfüllt sind, bleibt nur noch die Überprüfung der gesamten Arduino-Bibliothek auf Threadsicherheit und die Überarbeitung der dabei gefundenen gefährdeten Funktionen. Damit wären alle notwendigen Schritte für einen produktiven Einsatz ohne Einschränkungen getan.

Neben diesen notwendigen Korrekturen gibt es noch einige kleinere Verbesserungen, die ebenfalls in Angriff genommen werden können. Diese sind nicht zwingend notwendig, würden aber die Qualität des Systems erhöhen.

Die erste und einfachste Maßnahme ist ein Upgrade der in FreeRTOS-Arduino enthaltenen Version des Betriebssystems von 8.0.1 auf 8.2.1. Dies würde ein neues Feature mit sich bringen, welches ein schnelleres Aufwecken von Tasks verspricht. [Ltda]

Die zweite Maßnahme ist das das Zusammenfassen der im Moment getrennten Zweige von FreeRTOS-Arduino für AVR-Prozessoren und ARM-Prozessoren. Die Fusion der beiden Teile würde die Benutzung noch ein wenig einfacher machen. Beim Importieren der Bibliothek muss nicht mehr darauf geachtet werden, welcher Prozessor eingesetzt wird. Und bei einem Prozessorwechsel muss nicht mehr die Bibliothek gewechselt werden.

6 Anhang

Timing-Bibliotheken

- DueTimer
  https://github.com/ivanseidel/DueTimer
- Scheduler Library for Arduino
  http://www.codeproject.com/Tips/731031/Scheduler-Library-for-Arduino
- RELIABLE TASK SCHEDULING WITH ARDUINO / AVR
  https://chrisbarlow.wordpress.com/2012/09/13/reliable-task-scheduling-with-arduino-avr/
- A very simple Arduino task manager
  http://bleaklow.com/2010/07/20/a_very_simple_arduino_task_manager.html
- arduOS
  http://playground.arduino.cc/Code/ArduOS

Vollständige Scheduler

Name	URL
arduOS	http://playground.arduino.cc/Code/ArduOS
avr-os	https://github.com/chrismoos/avr-os
ArDOS	https://bitbucket.org/ctank/ardos-ide/wiki/Home
DuinOS	https://github.com/DuinOS/DuinOS
ScmRTOS	http://scmrtos.sourceforge.net/ScmRTOS
Nut/OS	http://www.ethernut.de
BeRTOS	http://dev.bertos.org/
TinyOS	http://www.tinyos.net/
ARTE/ERIKA	http://erika.tuxfamily.org/drupal/
ChibiOS	http://www.chibios.org
FreeRTOS	http://www.freertos.org
NilRTOS	http://chibios.sourceforge.net/docs3/nil/index.html
os47 v1.01	http://www.rtos47.com

Literaturverzeichnis

[ard] arduino.cc. Multiple blink tutorial. http://www.arduino.cc/en/Tutorial/MultipleBlinks.

[Dem14] Yves Demirdjian. os47 homepage. http://www.rtos47.com/, September 2014.

[eG10] egnite GmbH. Deutschsprachiges nutwiki. http://www.ethernut.de/nutwiki/Deutschsprachiges_NutWiki, September 2010.

[Gre14a] Bill Greiman. Chibios-arduino. https://github.com/greiman/ChibiOS-Arduino, August 2014.

[Gre14b] Bill Greiman. Nilrtos-arduino. https://github.com/greiman/NilRTOS-Arduino, August 2014.

[Gre15] Bill Greiman. Freertos-arduino. https://github.com/greiman/FreeRTOS-Arduino, Januar 2015.

[Ltda] Real Time Engineers Ltd. Freertos task notifications. http://www.freertos.org/RTOS-task-notifications.html.

[Ltdb] Real Time Engineers Ltd. Memory management options for the freertos [...] kernel. http://www.freertos.org/a00111.html.

[PC13] EE Times Peter Clarke. Android, freertos top ee times' 2013 embedded survey. http://www.eetimes.com/document.asp?doc_id=1263083, Februar 2013.

[Sir] Giovanni Di Sirio. Nilrtos dokumentation. http://chibios.sourceforge.net/docs3/nil/index.html.

[Sir15] Giovanni Di Sirio. Chibios/rt homepage. http://www.chibios.org/dokuwiki/doku.php, April 2015.

[Srl11] Develer Srl. Portingguide. http://dev.bertos.org/wiki/PortingGuide, Juni 2011.

[Srl14a] Evidence Srl. Erika enterprise manual for arduino uno board. http://erika.tuxfamily.org/wiki/index.php?title=Arduino, Mai 2014.

[Srl14b] Evidence Srl. Osek vdx certifications. http://erika.tuxfamily.org/wiki/index.php?title=OSEK_VDX_certifications, Februar 2014.

[sT12] scmRTOS Team. Scmrtos homepage. http://scmrtos.sourceforge.net/ScmRTOS, Dezember 2012.

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Pearson/Prentice Hall, Upper Saddle River, NJ, 2009.

[Tha12] Le Trung Thang. Comparing real-time scheduling on the linux kernel and an rtos. http://www.embedded.com/design/operating-systems/4371651/Comparing-the-real-time-scheduling-policies-of-the-Linux-kernel-and-an-R April 2012.

Erklärung (§ 14 Abs. 8 B.Sc. MIN PO)

„Ich versichere, dass ich die Bachelorarbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht."

Ort, Datum	Unterschrift

Erklärung für die Bibliothek

Ich erkläre mein Einverständnis, dass die von mir erstellte Abschlussarbeit in die Bibliothek der Uni Hamburg München eingestellt wird. Ich wurde darauf hingewiesen, dass die Hochschule in keiner Weise für die missbräuchliche Verwendung von Inhalten durch Dritte infolge der Lektüre der Arbeit haftet. Insbesondere ist mir bewusst, dass ich für die Anmeldung von Patenten, Warenzeichen oder Geschmacksmustern selbst verantwortlich bin und daraus Ansprüche selbst verfolgen muss.

Erklärung der Studierenden/des Studierenden

Ort, Datum	Unterschrift

Einverständniserklärung des/der betreuenden Hochschullehrers/-lehrerin bzw. Lehrbeauftragten

Ort, Datum	Unterschrift