Echtzeitfähiges Design für kleine Mikrocontrollersysteme

Markus Köchy[1], Christian Siemers[1] und Harald Richter[2]
[1]Fachhochschule Nordhausen, Weinberghof 4, D-99734 Nordhausen
(koechy|siemers)@fh-nordhausen.de
[2]Technische Universität Clausthal, Institut für Informatik, Julius-Albert-Straße 4, D-38678 Clausthal-Zellerfeld
richter@informatik.tu-clausthal.de

Zusammenfassung. Während der Entwurf von vergleichsweise groß ausgelegten, auf Mikrocontrollern basierenden Rechnersystemen bekannt und Gegenstand vieler Veröffentlichungen ist, gilt dies nur in weit geringerem Maß für kleinere Systeme, deren Funktionalität jedoch nicht minder zeitkritisch sein kann. Für die Unterscheidung klein/groß soll hierbei der Einsatz eines Betriebssystems bzw. der Verzicht darauf ausschlaggebend sein. Dieser Beitrag zeigt Wege und Designverfahren, wie Mikrocontroller-basierte Systeme ohne Echtzeit-Betriebssystem bei harten Zeitanforderungen entwickelt werden können.

Abstract. Microcontroller-based designs of considerable size are rather well-known and part of many publications, but this does not hold for small systems in spite of the fact that these systems are often time-critical, too. The major difference between small and large systems is the use of an operating system or its absence. This paper discusses design paradigms to develop microcontroller-based systems with hard real-time constraints without using an operating system.

1 Einleitung

Die aktuellen Sprachen zur Softwareerstellung sind zwar in der Lage, die logische bzw. arithmetische Funktionalität im Programm korrekt zu beschreiben, jedoch entzieht sich die Zeit einer funktionalen Beschreibungsweise. Dies bedeutet, dass das Programm zwar ”richtig“ funktioniert, d.h. die Algorithmen liefern (bei Annahme der Fehlerfreiheit von Softwareentwickler und Compiler) die korrekten Ergebnisse, ihr zeitliches Verhalten ist jedoch unbestimmt. Echtzeitsysteme, insbesondere in kritischen Bereichen, fordern jedoch die zeitliche Bestimmtheit, da ein zeitliches Fehlverhalten zu schweren Beschädigungen führen oder sogar Menschenleben gefährden kann.

Übliche Auswege aus diesem Dilemma sind die Überdimensionierung eines Mikrocontroller-basierten Systems, die exklusive Bereitstellung von Rechenressourcen für kritische Berechnungen (also die Abkehr von Multitasking-Systemen) oder auch der ausführliche Test. Die ersten beiden Wege führen häufig zu inakzeptablen Mehrkosten. Und auch vor dem guten Gefühl der Sicherheit, in das einen die ausführlichen Tests wiegen, ist nur zu warnen: Bei der Komplexität der heutigen Systeme können bei weitem nicht alle ”Corner-Cases“, d.h. alle Spezialfälle, getestet werden, der Test bleibt immer beschränkt.

Bei Mikrocontrollersystemen mit einem Echtzeit-Betriebssystem wird ein anderer Weg eingeschlagen: Die übliche Designpraxis besteht aus dem sorgfältigen Entwurf der einzelnen Tasks, der Bestimmung der Worst-Case Execution Time (WCET) und dem daraus abgeleiteten (statischen) Scheduling der Tasks. Eines der bekannten Verfahren ist Rate Monotonic [Kle et.al. 1993], bei dem die Prioritäten mit zunehmendem Rechenzeitbedarf abnehmen. Hier lassen sich hinreichende Bedingungen angeben, für die das System immer unter Einhaltung der zeitlichen Bedingungen lauffähig ist.

Wie steht es jedoch um die kleinen Mikrocontrollersysteme, die kein Multitasking-Betriebssystem haben, aber dennoch mehrere Teilaufgaben in Echtzeit bewältigen sollen? Hier existieren keine allgemein bekannten Lehrbuch-Verfahren, wie solche Systeme systematisch entwickelt werden können und die Echtzeitfähigkeit nachgewiesen werden kann. Ziel dieses Beitrags ist es daher, ein Design-Pattern anzugeben, mit dem die gestellte Aufgabe bei Zutreffen einiger Randbedingungen gelöst werden kann.

Das Paper, das Arbeiten im Steering-by-Wire-Forschungsprojekt am Informationstechnischen Zentrum (ITZ) der Technischen Universität Clausthal zusammenfasst, ist im weiteren Verlauf wie folgt gegliedert: Abschnitt 2 zeigt die Randbedingungen dieses Design-Patterns und gibt zugleich eine Klassifizierung der Teilaufgaben an. Diese Klassifizierung ist notwendig, da in den einzelnen Teilklassen unterschiedliche Implementierungsstrategien verfolgt werden müssen. Abschnitt 3 stellt die unterschiedlichen Lösungsansätze dar.

Abschnitt 4 beschreibt das Zusammenfügen der Teilkomponenten sowie die Schritte, die zum Nachweis der Echtzeitfähigkeit unternommen werden müssen. Abschnitt 5 gibt schließlich eine Zusammenfassung der Ergebnisse sowie einen Ausblick.

2 Klassifizierung der Aufgaben

2.1 Randbedingungen für das zu implementierende System

Die Randbedingungen der Aufgabenstellung sowie die des zugrunde gelegten Rechnersystems lassen sich summarisch wie folgt angeben:

- Das Zielsystem basiert auf einem Mikrocontroller, d.h., die Bearbeitung eines Programms erfolgt in einer sequenziellen Bearbeitung der Anweisungen. Insbesondere wird von einem Ein-Controller-System ausgegangen.
- Die WCET der Applikation ist in engen Grenzen zum tatsächlichen Wert bestimmbar. Die Genauigkeit ist hierbei applikationsabhängig definierbar und orientiert sich meist an wirtschaftlichen Gesichtspunkten.
- Das Zielsystem hat kein Betriebssystem, das ein (echtzeitfähiges) Multitasking ermöglicht. Zur Implementierung mehrerer Teilaufgaben wird es dennoch nativ genutzt.

Die Aufzählung der wenigen Punkte macht deutlich, dass es sich hierbei nicht um gravierende Einschränkungen handelt. Vielmehr wird definiert, was ein ”kleines“ System ist. Die Bestimmbarkeit der WCET erweist sich in Mikrocontroller- und Mikroprozessorsystemen, die auf Performance optimiert sind, als äußerst schwierig [Hec et.al. 2003]. Gründe hierfür sind Pipelining und Superskalarität mit den dadurch verursachten Daten- und Kontrollflussabhängigkeiten [Sie 2004] sowie das Verhalten des Cache-Speichers, für das bei einer Laufzeitschätzung nur unter großen Problemen eine Annahme (abgesehen vom unrealistischen Fall ständiger Cache-Misses) getroffen werden kann [Hec et.al. 2003]. All diese Techniken steigern zwar die durchschnittliche Performance, garantieren sie jedoch nicht für jeden einzelnen Anwendungsfall.

Für Mikrocontroller, die weder einen Cache-Speicher nutzen noch superskalar arbeiten, können die Ausführungszeiten unter bestimmten Randbedingungen auch bei Pipelining mit einer Fehlergrenze von ca. 20% recht gut geschätzt werden. Diese Fehlergrenze bedeutet, dass die geschätzte WCET nicht kleiner und maximal 20% größer ist als die tatsächliche maximale Ausführungszeit.

Die zu erfüllende Randbedingung ist dabei, dass datenabhängige Schleifen – hierzu zählen z.B. for- oder while-Schleifen, bei denen die Bedingung zum Abbruch bzw. Erhalt der Schleife von Laufzeit-veränderlichen Daten abhängen – entweder nicht auftreten oder entsprechend genau geschätzt werden können. Die dann verbleibende Quelle für die Unsicherheit in der WCET-Schätzung liegt in den (bedingten) Sprungbefehlen, deren Häufigkeit ca. 20% aller Befehle beträgt und die häufig verschiedene Ausführungszeiten bei Sprung bzw. Nicht-Sprung besitzen. In diesem Fall wird zur Schätzung zumeist die maximale Anzahl der Takte gewählt.

2.2 Klassifizierung der Teilaufgaben

Das in diesem Paper dargestellte Designverfahren beruht darauf, die einzelnen Teilaufgaben zu klassifizieren, ihren gewünschten Eigenschaften nach zu implementieren und das System dann zu integrieren. Die Klassifizierung ist notwendig, da insbesondere im Zeitbereich verschiedene Randbedingungen für die einzelnen Klassen angenommen werden müssen.

Konkret ergeben sich folgende Klassen:

- *Streng zyklisch ablaufende Tasks*: Fester Bestandteil dieser Teilaufgaben sind exakte Zeitabstände, in denen diese Tasks zumindest gestartet werden und generell auch komplett ablaufen müssen, um der Spezifikation zu genügen. Beispiele hierfür sind Messwertaufnahmen oder die Bedienung von asynchronen Schnittstellen zur Datenkommunikation.
- *Ereignis-gesteuerte Tasks*: Der Start einer Task mit dieser Charakterisierung ist an ein externes Ereignis, meist in Form eines Interrupt-Requests, gebunden. Dies bedeutet, dass der Startzeitpunkt nicht zur Compilezeit bestimmbar ist, so dass diese Tasks störend auf den zeitlichen Gesamtablauf wirken können. Typische Vertreter dieser Klasse sind der Empfang von Nachrichten via Netzwerk bzw. die Reaktion darauf oder Schalter in der Applikation, die besondere Zustände signalisieren (etwa ”Not-Aus“).
- *Generelle Tasks mit Zeitbindung*: Die dritte Klasse beschreibt alle Tasks in dem System, die zwar keine scharfen Zeitbedingungen enthalten, im Ganzen jedoch Zeitschranken einhalten müssen. Hiermit sind Tasks beschrieben, die beispielsweise Auswertungen von Messwerten vornehmen. Während die einzelne Auswertung ausnahmsweise über einen Messwertzyklus hinaus dauern darf, muss insgesamt die mittlere Auswertezeit eingehalten werden.

Diese drei Grundklassen zeitabhängiger Teilaufgaben stellen das Grundgerüst zum Systemdesign dar. Die erste Aufgabe des Systemdesigners besteht darin, alle in der Beschreibung vorkommenden Aufgaben in dieses Grundgerüst einzuteilen. Hierbei können natürlich Schwierigkeiten auftreten, die u.a. im folgenden Abschnitt behandelt sind.

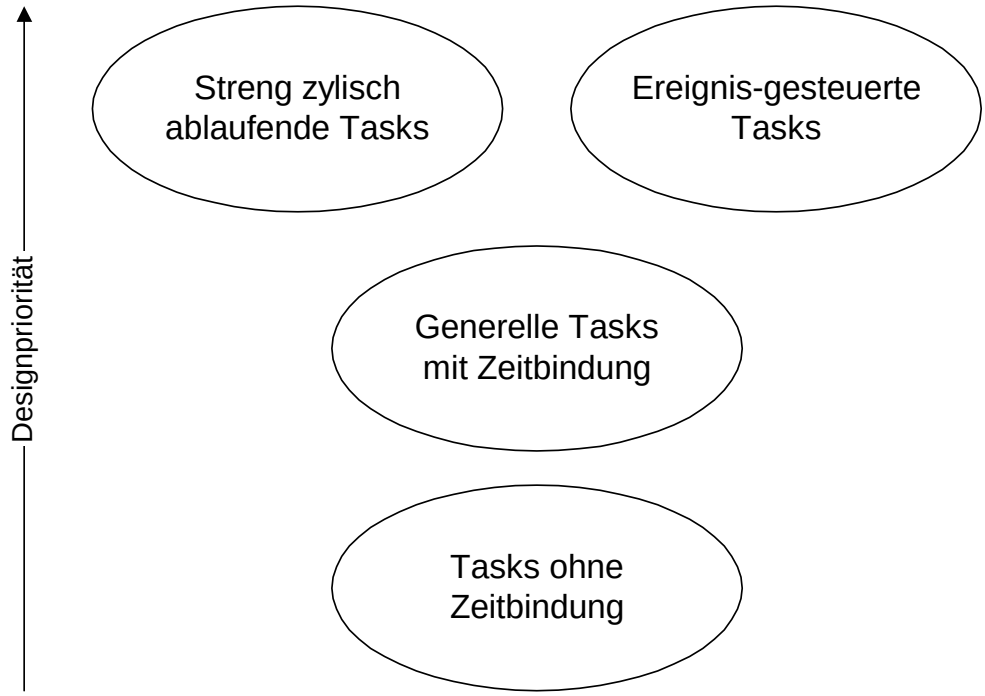

Bild 1 Taskklassen und Designpriorität

Generell gilt, dass eine Teilaufgabe in eine ”höhere Klasse” integriert werden kann. So kann eine Task, die überhaupt keine Zeitbindung besitzt – eine derartige Task dürfte in der Praxis außerordentlich selten vorkommen – natürlich in die Klasse der generellen Tasks mit Zeitbindung sortiert werden. Diese Taskklasse ist in Bild 1 dargestellt, wurde jedoch nicht in die Klassifizierung aufgenommen, da sie irrelevant für das hier dargestellte Designprinzip ist.

Streng zyklisch ablaufende Tasks und Ereignis-gesteuerte Tasks sind in ihrer Designpriorität in etwa gleichzusetzen, siehe Bild 1. In der Praxis kann die Implementierung – siehe Abschnitt 3 – auch sehr ähnlich sein, indem die zyklischen Tasks in Interrupt-Service-Routinen (ISR) mit Timersteuerung und die Ereignis-gesteuerten Tasks in anderen ISRs behandelt werden. Wie noch dargestellt werden

wird, soll die Unterscheidung dennoch aufrecht erhalten bleiben, da zwischen beiden Implementierungen ein fundamentaler Unterschied existiert.

3 Lösungsansätze für die verschiedenen Aufgabenklassen

Der nächste Schritt des Designverfahrens sieht vor, nach einer Klassifizierung der Tasks die Mitglieder der einzelnen Klassen zunächst getrennt voneinander zu implementieren und die maximale Ausführungszeit jeweils zu berechnen. Hierbei wird (in erster Näherung) davon ausgegangen, dass die WCET der einzelnen Teilaufgaben voneinander unabhängig sind. Dies bedeutet insbesondere, dass auf ein blockierendes Warten auf Kommunikation zwischen den Tasks unbedingt verzichtet werden muss, denn dies kann zu großen Problemen bei der Bestimmung der WCET bis hin zur Unmöglichkeit führen. Diese Forderung führt zu einem sicheren Design, da sich Abhängigkeiten etwa in der Form, dass falls Task 1 den maximalen Pfad durchläuft, Task 2 garantiert einen kleineren Pfad als seinen maximalen wählt, nur positiv auf die WCET des Gesamtsystems auswirken können.

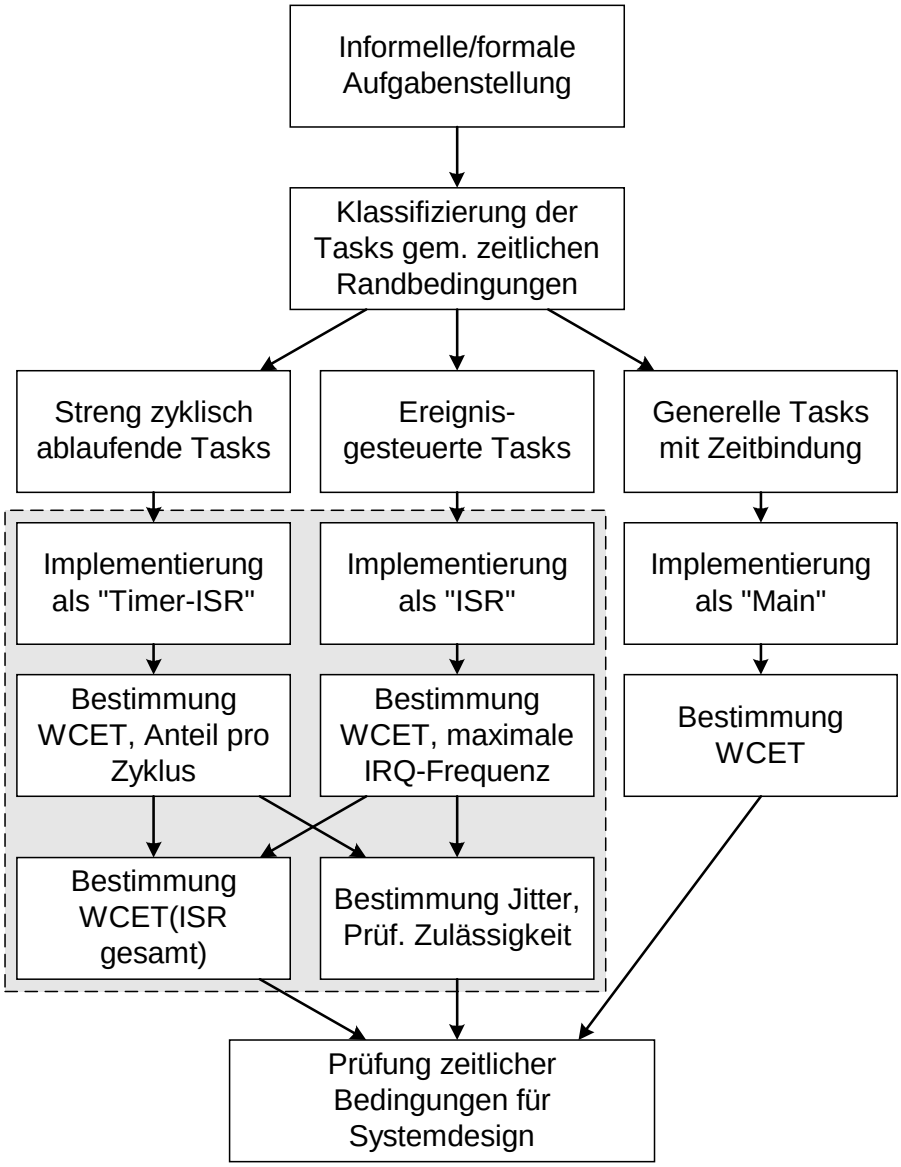

Bild 2 Gesamtablauf Systemdesign

Bild 2 zeigt den gesamten Designprozess ohne Entscheidungen bzw. Rückwirkungen, die aus Übersichtsgründen fortgelassen wurden. Tatsächlich sind in seinem Verlauf einige Abstimmungen und Entscheidungen notwendig, insbesondere in dem grau schattierten Teil der Implementierung zweier ISRs mit gegenseitiger Beeinflussung.

3.1 Streng zyklisch ablaufende Tasks

Die Implementierung von Teilaufgaben, die zyklisch auftreten, ist optimal in einer ISR durchzuführen, die von einem zyklisch arbeitenden Timer aufgerufen wird. Die Probleme, die dabei auftreten, liegen weniger im grundsätzlichen Design als vielmehr darin, mit welcher Frequenz bzw. mit welchem Zeitwert die ISR aufgerufen wird.

Liegt nur eine Aufgabe mit streng zyklischem Verhalten vor, ist die Wahl einfach: Die Zeitkonstante, die beispielsweise zwischen zwei Messungen oder zwei Transmissionen liegt, wird als der Timerwert gewählt. Bei mehreren Aufgaben mit unterschiedlichen Perioden können mehrere Timer eingesetzt werden. Es erweist sich jedoch auch im Sinn der Bestimmung der Lauffähigkeit als optimaler, den größten gemeinsamen Teiler (ggT) der Periodenzeiten als Zeitwert zu wählen und sich auf einen Timer-Interrupt zu beschränken.

Andererseits kann der ggT-Ansatz sehr schnell in ein nicht-lauffähiges System münden. Jeder Aufruf einer ISR erfordert – von Zeiten der Ununterbrechbarkeit des laufenden Programmteils abgesehen – einen zeitlichen Overhead. Als Faustregel sollte man mit 10 – 20 Befehlsausführungszeiten rechnen, die für Interrupt-Latenzzeit, Retten und Restaurieren von Registern und den Rücksprung in das Programm benötigt werden. In einem System, das 1 µs Befehlsausführungszeit hat und alle 200 µs unterbrochen wird, sind das aber bereits 5 – 10 % der gesamten Rechenzeit, die unproduktiv vergehen.

Soweit es irgendwie möglich ist, sollte daher die Periode so gewählt sein, dass der ISR-Overhead klein bleibt (< 5%). Falls die Spezifikationen es zulassen, besteht die ideale Methode für mehrere Aufgaben mit zyklischem Verhalten also darin, die Zykluszeiten aufeinander anzupassen. Messwertaufnahmen können ggf. langsamer oder schneller erfolgen, während die Bedienung von in Software implementierten Kommunikationsschnittstellen meist keine Variation zulässt.

Die Bedienung der seriellen Schnittstelle (RS232) beispielsweise, bei der die Bits der Transmission einzeln in Software zeitgenau gesetzt werden, kostet einiges an Rechenzeit. Als Faustformel können hier ca. 30 – 40 Befehle angenommen werden, was bei den angenommenen Werten (208 µs Zykluszeit entsprechen etwa 4800 bit/s) bis zu 20 % der Rechenzeit ausmacht.

3.2 Ereignis-gesteuerte Tasks

Für Ereignis-gesteuerte Tasks bieten sich grundsätzlich zwei Implementierungsmethoden an: Polling und Interrupt-Service-Routine. Polling, ursprünglich vorgesehen als ständig wiederkehrende Abfrage aus dem normalen Programmverlauf heraus, kann besser als Bestandteil der Timer-ISR (siehe 3.1) implementiert werden. Man spricht dann auch besser von einem (komplett) Zeit-gesteuerten Modell (Time-Triggered).

Folgende Bedingungen sind zu erfüllen, um diesen Ansatz nutzen zu können:

- Die Applikation muss eine Latenzzeit, die mindestens gleich der Zeitperiode des zyklischen Teils ist, akzeptieren können.
- Die Mehrbelastung der Timer-ISR durch die zusätzliche Abfrage muss im gesamten Zeitplan bleiben.

Während die zweite Bedingung in den seltensten Fällen problematisch ist, stößt man bei der ersten Bedingung häufiger an Grenzen. Es ist zwar möglich, die Periode des Timers in Grenzen zu variieren. Dies kann aber aus den in 3.1 genannten Gründen zu nicht-lauffähigen Systemen führen.

Die geringere Belastung des Gesamtsystems durch eine echte Ereignissteuerung – nur bei Auftreten eines Ereignisses wird die entsprechende Routine aufgerufen (Event-ISR) – macht diese so interessant für eine Implementierung in kleinen Systemen. Der Einsparungsgewinn ist allerdings nur virtuell: Soll eine harte Echtzeitfähigkeit erreicht werden, muss für den ”worst case“ geplant werden, also für die maximal auftretende Unterbrechungsfrequenz.

Genau hier setzt die wesentliche Kritik an der Nutzung von Ereignissteuerungen in harten Echtzeitsystemen an, denn die maximal auftretende Aufruffrequenz einer ISR ist a priori nicht planbar

(allein durch defekte Hardware könnten mehr IRQs ausgelöst werden). In [Fal Sie 2003] und [Sie et.al. 2003] wurden Lösungen vorgestellt, die ein deterministisches Verhalten auch bei Nutzung von externen Interruptquellen und einer Ereignissteuerung ermöglichen. Die dort beschriebene Zusatzhardware ist jedoch nicht Standard und somit in kommerziell erhältlichen kleinen Mikrocontrollern nicht vorhanden.

Neben allen diesen Erwägungen gibt es dennoch Fälle, in denen eine Ereignissteuerung vorteilhaft und deterministisch ist:

- Interrupts, die z.B. durch Empfang von Zeichen auf einem Übertragungskanal intern ausgelöst werden, besitzen eine durch die festgelegte Übertragungsgeschwindigkeit bedingte maximale Auftrittsfrequenz. Sie können daher sehr gut in einer Ereignis-gesteuerten ISR behandelt werden.
- Interrupts, deren Auslösung sehr selten ist und deren Ziel nicht das Verbleiben im normalen Betriebszustand, sondern das Erreichen eines sicheren Zustands ist, sind ebenfalls sehr geeignet für Ereignis-gesteuerte Systeme (Beispiel ”Not-Aus“).
- Für Interrupts mit einer sehr geringen Latenzzeit bleibt im Wesentlichen nur die Integration als Ereignis-gesteuerte ISR.

Liegt also einer dieser Gründe vor, sollte bzw. muss der Designer den Weg einer Ereignis-gesteuerten Task bzw. ISR wählen. Hierbei gilt es dann, eine genaue Abstimmung bzw. Priorisierung mit der Timer-ISR zu finden, da beide Tasks konkurrierend sind. Kapitel 4 enthält dazu einige Anmerkungen.

3.3 Generelle Tasks mit Zeitbindung

Die Implementierung von generellen Tasks mit Zeitbindung ist eigentlich unproblematisch, mit einer Ausnahme: Es muss die WCET bestimmt werden, und die Zeiten der Ununterbrechbarkeit, in denen der Interrupt-Eingang abgeschaltet ist, müssen genau mit den Interrupt-Latenzzeiten und dem zulässigen Jitter der übrigen Tasks abgestimmt sein.

Für die Abstimmung sind zwei Fälle zu unterscheiden: Berechnung pro Zykluszeit oder Berechnung pro vielfacher Zykluszeit. In manchen Systemdesigns muss pro Timer-ISR und damit pro Zeitzyklus im Mittel auch eine Aktion im generellen Teil der Applikation erfolgen. Liegt die WCET für einen Zeitzyklus innerhalb der in einem Zeitzyklus noch freien Rechenzeit, besteht für das Systemdesign kein Problem. Anderenfalls kann und muss versucht werden, die WCET über mehrere Zyklen zu berechnen, indem beispielsweise die Werte aus dem zyklischen Teil durch Pufferung zwischengespeichert sind. Führt die Berechnung über mehrere Zyklen im Mittel zu einer kleineren WCET, so kann dies zu einem geeigneten System führen.

Bei anderen Applikationen (Beispiel FFT-Analyse) werden Berechnungen nur über Gruppen von Werten durchgeführt. Falls die Berechnungsgruppe über mehrere Zyklen – etwa durch Messwertaufnahme oder Kommunikation – im Rechner gesammelt wird, sei angenommen, dass auch die Auswertung die gleiche Anzahl der Zyklen in Anspruch nehmen kann. Dies entspricht einem Pipelineprinzip auf Datenblockebene und bedeutet, dass die WCET über alle Zyklen berechnet werden muss.

4 Zusammenfügung der Teile

Das Zusammenfügen der einzelnen Applikationsteile, bestehend aus generellen Tasks, Timer-ISRs und ggf. Event-ISRs, beinhaltet die Organisation der Kommunikation zwischen den einzelnen Teilen sowie die Abstimmung des Zeitverhaltens. Als Kommunikation zwischen diesen Tasks ist ein nicht-blockierendes Semaphoren/Mailbox-System ideal: Semaphore, die seitens einer Task beschrieben und seitens der anderen gelesen und damit wieder gelöscht werden können, zeigen den Kommunikationsbedarf an, während die eigentliche Meldung in einer Mailbox hinterlegt wird.

Blockieren kann durch eine asynchrone Kommunikation vermieden werden: Tasks warten nicht auf den Empfang, eine Quittierung bzw. Antwort auf ihre Transmission, sie senden einfach (via Semaphor/Mailbox). Auch die Abfrage von empfangenen Sendungen erfolgt dann nicht-blockierend.

Die zeitliche Abstimmung der einzelnen Tasks ist wesentlich aufwendiger und muss folgende Überlegungen einschließen:

- Wie beeinflussen ununterbrechbare Teile in der generellen Task bzw. einer ISR die Latenzzeiten der Interrupts? Die Beantwortung dieser Frage ist insbesondere für die zyklischen Tasks mit strenger Zeitbindung wichtig, da man hier davon ausgehen muss, dass Jitter nur in sehr geringem Umfang erlaubt ist.

  Die praktische Ausführung sieht so aus, dass tatsächlich die entsprechenden Befehle im Maschinencode (”SEI“, ”CLI“) gesucht und die WCETs der ununterbrechbaren Zwischenräume bestimmt werden. Diese Zeiten sollten auf das absolute Minimum beschränkt sein. Die Bestimmung muss aber am Maschinencode erfolgen, da meist in einer Hochsprache wie C entwickelt werden wird, deren eingebundene Laufzeitroutinen derartige Abschnitte enthalten können.
- Timer-ISR und Event-ISR stehen in Konkurrenzbeziehung, was die Zuteilung der Rechenzeit betrifft. Grundsätzlich sollte der strengen Zeitbindung der Vorrang gegeben werden, und die Routinen hierfür sind auch Kandidaten für eine Ununterbrechbarkeit. Dies allerdings bedeutet die Erhöhung der Latenzzeit für die Event-ISR, was für den Einzelfall zu prüfen ist.

  Eine Ausnahme bildet der Fall, dass die Event-ISR sehr hoch priorisiert werden muss, weil bei Auftreten ein sicherer Zustand zu erreichen ist. Dieses Ereignis muss sofort behandelt werden, so dass die Timer-ISR in diesem Fall unterbrechbar sein sollte.

Nach dem Zusammenfügen der einzelnen Teile und der Abstimmung der zeitlichen Randbedingungen kann dann die korrekte Funktionsweise des gesamten Systems nachgewiesen werden. Hierzu wird ein Zeitraum betrachtet, in dem ein gesamter Zyklus ablaufen kann. Insbesondere muss die generelle Task die Berechnung beenden können. In diesem Zeitabschnitt darf die Summe der WCETs, multipliziert mit den entsprechenden Auftrittshäufigkeiten, die Gesamtrechenzeit nicht überschreiten.

Für die Latenzzeiten gelten die gesonderten, oben beschriebenen Bedingungen.

5 Zusammenfassung

Die Arbeiten zu diesem Paper wurden und werden im Rahmen des Steering-by-Wire-Forschungsprojekts am Informationstechnischen Zentrum (ITZ) der Technischen Universität Clausthal durchgeführt. Das hier dargestellte Designverfahren zur Herstellung kleiner echtzeitfähiger Systeme beinhaltet eine Anzahl von Schritten, die jeweils durchlaufen werden müssen. Klassifizierung und Realisierung der einzelnen Klassen werden gefolgt von der Bestimmung der einzelnen Worst-Case Execution Times sowie der Betrachtung der zeitlichen Wechselwirkungen.

Dieses Designverfahren bietet damit einen systematischen Zugang zur Realisierung derartiger Systeme.

Literatur

[Fal Sie 2003] Rainer Falsett, Christian Siemers, ”Garantiertes Zeitverhalten mechatronischer Systeme – Planung und Überwachung“. Tagungsband Embedded World 2003, Nürnberg, Februar 2003, S. 971–978.

[Hec et.al. 2003] Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R., 2003. ”The Influence of Processor Architecture on the Design and the Results of WCET Tools”. *Proceedings of the IEEE* 91(7), 1138-1154 (2003)

[Kle et.al 1993] Klein, M.H., et al. A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Boston, MA: Kluwer Academic Publishers, 1993.

[Sie 2004] Siemers, C., 2004. ”Prozessor Technologie“. TecChannel compact Mai 2004.

[Sie et.al. 2003] Christian Siemers, Rainer Falsett, Reinhard Seyer, Klaus Ecker, ”Supporting the Hard Real-Time Requirements of Mechatronic Systems by 2-Level Interrupt Service Management”. Workshop on Parallel and Distributed Real-Time Systems WPDRTS 2003, Nice, France, April 2003. In: Proceedings of the 17th International Parallel & Distributed Processing Symposium IPDPS 2003, Nice, France (in electronic version).