THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Interrupt-Techniken und ihre Probleme aus Softwaresicht

Hauptseminar Software für technische Systeme
Institut für Technik und Informatik

Sommersemester 2011

Ingo Schöndube

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Dokument beschäftigt sich mit Interrupt-Techniken und ihren Problemen aus Softwaresicht. Entstanden ist es im Rahmen des Hauptseminars 'Software für Embedded Systems' am Institut für Technik und Infromatik der TH Mittelhessen. Ziel der Ausarbeitung ist es, ein grundlegendes Verständnis für die Notwendigkeit und die Vorteile beim Einsatz von Interrupt-Techniken zu vermitteln. Zudem werden die hierbei auftretenden Problematiken identifiziert und mögliche Lösungsansätze aufgezeigt. Diese Ausarbeitung betrachtet die Thematik konzeptionell und berücksichtigt dabei keine konkreten Realisierungen.

Die Zielgruppe sind fortgeschrittene Studenten der Informatik oder vergleichbare Studiengänge, die sich mit dem Thema Mikrocontrollertechnik auseinandersetzen. Daher werden Grundkenntnisse in Programmierung sowie Softwareentwicklung vorrausgesetzt.

Als eine passende Einleitung in die Thematik erachte ich es als sinvoll, zunächst Techniken ohne Interrupts vorzustellen und aus deren resultirenden Problemen eine Motivation für den Einsatz von Interrupt-Techniken zu schaffen. Daraus ergibt sich eine thematische Überleitung in das Kapitel der Interrupts. Hier beginne ich mit einer Begriffsdefinition und betrachte anschließend die Konzepte auf der Hardware- bzw. Softwareseite. Danach zeige ich die Probleme bei der Verwendung von Interrupts aus Softwaresicht auf. Abschließend werden die wichtigsten Punkte bewertet und Empfehlungen für weiterführende Literatur gegeben.

Kapitel 2

Interrupts

2.1 Techniken ohne Interrupts

Motivation

Um die Notwendigkeit eines Mechanismus, der sich hard- und softwareseitig um asynchronen und nebenläufige Unterbrechungen kümmert zu verdeutlichen, ist es sinnvoll zunächst Methoden zu diskutieren, welche ohne einen solchen Hardwaremechanismus arbeiten. Anhand dieser Implementationen werden relativ schnell die Vor- bzw. Nachteile solcher Algorithmen offensichtlich, sodass eine thematische Überleitung zu den Interrupt-Techniken und deren Notwendigkeit folgt.

Die vorgestellten Beispiele sind in ihrer Natur eher trivial, verdeutlichen dennoch sehr anschaulich und nachvollziehbar die zugrundeliegende Problematik. Zudem sind dies Probleme, die sowohl im Embedded Software als auch im 'normalen' Software Bereich häufig auftreten und von den Entwicklern effizient, gekapselt und standarisiert zu lösen sind.

2.1.1 Busy-Waiting

Eine sehr häufig auftretende Situation ist das Abfragen einer externen Informationsquelle, z.B. Eingabegeräte wie Tastatur oder Maus. Die Behandlung dieser asynchronen und zeitlich nicht vorhersehbaren Ereignisse kann durch *Busy-Waiting* gelöst werden.

Beim Busy-Waiting wird der Zustand eines Gerätes permanent abgefragt. Dies wird auch *Polling* genannt. Abbildung 2.1 verdeutlicht dies anhand eines Zustandsdiagramms.

Der Vorteil dieser Methode ist die Möglichkeit eine einfache Implementierung zu nutzen.

Hier übernimmt die Bedingungsüberprüfung der While-Schleife die Aufgabe des Abfragens der Ressource, auf welche gepollt werden soll. Ein solcher Programmkode ist transparent. Die semantische Bedeutung dieses Abschnittes ist

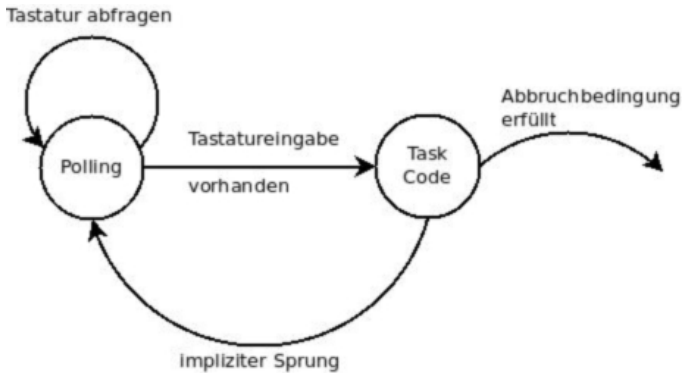

Abbildung 2.1: Busy-Waiting

leicht ersichtlich und birgt daher eine geringe Fehleranfälligkeit.

Leider weist diese Methode erhebliche Nachteile auf. Es ist nicht möglich mehrere Ressourcen simultan abzufragen. Somit ist diese Fixierung auf eine Ressource als nicht aussreichend zu betrachten. Zwar könnte die Bedingungsprüfung erweitetert werden, allerdings würde dies das Softwaredesign negativ beeinflussen in folge von zusammenführen nicht zusammengehöriger Ressourcenabfragen. Zudem führt eine komplexe Verknüpfung von Bedingungen durch boolsche Operatoren schnell zu mangelnder Übersichtlichkeit und damit zu Fehlern. Auch die Nutzung der CPU ist bei einer solchen Implementation sehr schlecht, da die CPU mit der permanenten Bedingungsprüfung beschäftigt ist und somit für sinvolle Instruktionen nicht zur Verfügung steht. Durch die Tatsache, dass die CPU dauerhauft am schalten ist, wird sie den Anforderungen verbrauchsoptimierter Systeme nicht gerecht. Im Zeitalter der Green-IT sowie der mobilen Endgeräte ist dies eine nicht mehr akzeptable Lösung des Problems. Ein hoher Stromverbrauch stellt einen erhöhten Kostenfaktor dar, zudem wird hierdurch die CO_2 Emmission gesteigert, was wiederumm schlecht für die Umwelt ist. Eine lange Akkulaufzeit bei mobilen Geräten ist eine wichtige Grundvorraussetzung zur Nutzerakzeptanz und somit zur Marktetablierung.

2.1.2 Round-Robin

Das Round-Robin Verfahren gleicht einige Nachteile des Busy-Waiting Verfahrens aus, indem es mehrere Ressourcen in einem gestaffelten Intervall abfragt. Diese Staffelung wird meist durch einen zuvor definierten Timeout realisiert. Hierzu wird der Zeitwert der Bedingungsprüfung hinzugefügt. Die Anzahl der abzufragenden Ressourcen spiegelt dabei die Zahl der aneinander gereihten While-Schleifen wieder. Folgende Abbildung 2.2 zeigt den schematischen Aufbau des Round-Robin Verfahrens.

Der Aufwandszuwachs dieser Methodik ist verhältnissmässig gering. Er verhält sich linear zur Anzahl hinzuzufügender Ressourcen. Dabei bleibt das Verfahren

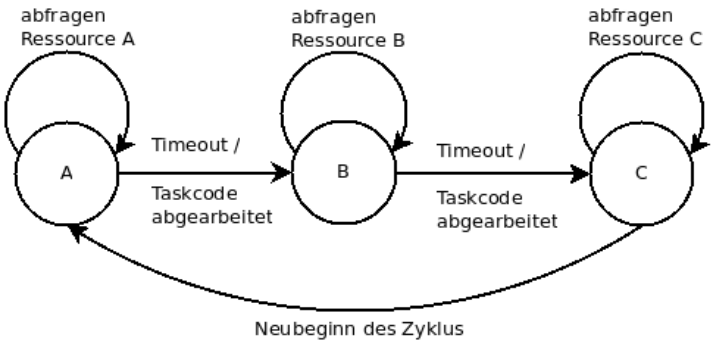

Abbildung 2.2: Round-Robin Verfahren

im Gegensatz zum Busy-Waiting auch bei steigender Komplexität den Designansprüchen treu. Zudem wahrt sie die Übersichtlichkeit durch modularisierte Trennung der Abfrageblöcke. Ein weiterer Vorteil stellt die deterministische Berechenbarkeit des Gesamtzyklusses dar. Dabei bildet sich die Ausführungszeit einer Ressource durch Addition der Wartezeit mit der Ausführungszeit des zugehörigen Block-Codes. Um nun die Gesamtzykluszeit zu erhalten werden die von den Ressourcen benötigten Zeiten addiert. Dies genügt Systemen mit weichen Echtzeitanforderungen.

Allerdings weist auch diese Methodik einige gravierende Nachteile auf. So ist es möglich, dass beim pollen auf eine bestimmte Ressource die Anfrage einer anderen Ressource verloren geht. Eine Pufferung von Anfragen ist aus softwarestrukturellen Gründen nicht realisierbar. Dieses Szenario ensteht, wenn eine Ressource eine kürzere Anfragezeit besitzt als die gesamte Zyklusdauer. Hier tritt das *Response-Problem* auf, das System reagiert nicht ausreichend schnell auf ein Ereignis. Auch eine Priorisierung von Anfragen ist nicht oder nur mangelhaft möglich. Die Beherrschbarkeit bei steigender Komplexität und Ressourcenanzahl sinkt rapide. Hinzufügen jeder weiteren Ressource kann den balancierten Zykluss zerstören.

2.2 Interrupt

Die vorgestellten Softwarelösungen decken die Anforderungen moderner Softwareimplementierungen nicht ab, so dass Mikroporzessoren bzw. 'Computer' relativ früh mit einem Interruptmechanissmus, oder kurz *Interrupt*, ausgestattet wurden. Ein solcher Mechanismus bietet unter anderem asynchrones Reagieren auf externe Events, ohne dass CPU-Zeit mit endlosen Abfragen vergeudet wird. Auch die bereits angesprochene Priorisierung und Pufferung externer und interner Events ist durch eine solche hard- und softwareseitige Unterstützung ohne weiteres möglich. Zu bedenken ist jedoch, dass eine solche Lösung die Komplexität der Hardware steigert und somit mehr Platz auf dem Chip erfordert. Weiterhin verbraucht der Microcontroller bzw. der Microprozessor hierdurch mehr Energie. Allerdings wer-

den diese Nachteile gerne in kauf genommen da sie alle erforderlichen Mittel bereitstellen und somit der Nutzen dieser Lösung überwiegt. *Interrupts* sind somit ein sehr mächtiges Werkzeug für die Kommunikation mit der restlichen Peripherie.

2.2.1 Was sind Interrupts?

Interrupts sind ins deutsche übersetzt *'Unterbrechungen'*. Diese simple Übersetzung beschreibt relativ gut was Interrupts sind. Der Terminus "Unterbrechung" bezieht sich in diesem Kontext auf die Unterbrechung des 'Normalen'-Programmflusses zu einem beliebigen nicht vorhersehbaren Zeitpunkt. Dies wird durch einen speziell und ausschliesslich für diese Aufgabe entworfenen Hardwaremechanismus umgesetzt. Je nach Grösse bzw. Komplexität des Mikroprozessors bietet eben jener eine gewisse Anzahl an Features. Grosse CPUs beherbergen einen Interrupt-Controller, der eine weite Bandbreite an Hardwareunterstützung anbietet. Als Beispiel für eine solch komplexe CPU sei hier die x86 Architektur genannt.

Da es sich um einen Hard- und Softwaremechanismuss handelt, ist es zunächst sinnvoll den hardwareseitigen Aspekt zu betrachten, um so eine Grundlage für das notwendige Verständniss der Software zu etablieren. Die *Interrupt-Techniken* auf Seite der Software werden anschliessend in einer feineren Granularität betrachtet. Diese herangehensweise zeigt spätestens bei der detalierten Betrachtung des *Interruptablaufes* die aus dieser Technik resultierenden Probleme auf.

2.2.2 Hardwarerealisierung

Die Realisierung der Kommunikation mit der Aussenwelt über Interrupts erfolgt meist durch spezielle Baugruppen des Mikroprozessors. Eine solche Baugruppe ist z.B. ein eingangs Pin des Chips, wie in Abbildung 2.3[1] schematisch dargestellt. Dies ist die externe Sicht auf den Microchip und seine Ein- und Ausgangspins sowie die mit ihm verbunden Baugruppen, welcher im Regelfall auf einer Platine verbaut sind. Um eine Kommunikation bewerkstelligen zu können wird ein Periferiechip oder ein Taster mit einem Interrupt fähigen Pin-In des Chips verdrahtet, so dass die Änderung des Pegels, ausgelöst durch den *Service-Request* an diesem Pin, einen Interrupt des Mikrocontrollers bewirkt. Dieser Vorgang wird als *Interrupt Request* oder auch *Interrup Service Request* bezeichnet.

Solche Anfragen werden jedoch nicht auschliesslich über einen externen Anschluss gestellt, auch im Chip untergebrachte Baugruppen, wie z.B. ein Analog-Digital-Wandler oder in einem Mikrocontroller integierter Hauptspeicher, können *Interrupts* auslösen.

Die interne Realisierung ist um einiges komplexer und variiert erheblich von CPU zu CPU, daher wird sie hier konzeptionell beschrieben, um eine möglichst allgemeine Sicht darzustellen.

[1] in Anlehnung an [1, SIMON07]

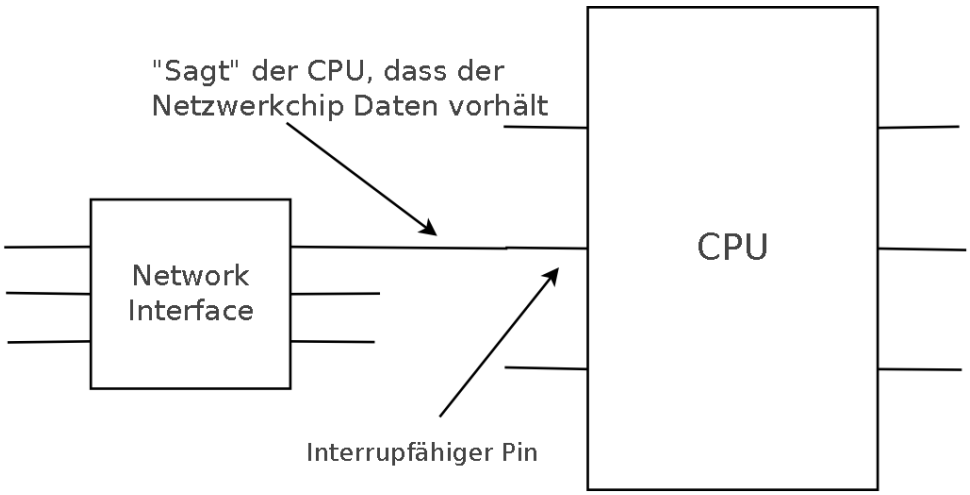

Abbildung 2.3: Externe Sicht auf die CPU

Liegt nun an einem Pin-In des Mikrocontrollers ein *Interrupt Request* an, löst dies einen Programmzählersprung, aus dem Hauptprogramm in ein Nebenprogramm, aus. Zusätzlich wird in einem speziellen Register der CPU, dem Interruptflagregister, ein Flag gesetzt, um zu signalisieren, dass eine Anfrage stattgefunden hat. Das Nebenprogramm, auch als *Interrupt Service Rountine*[2] oder kurz *ISR* bezeichnet, wird nun abgearbeitet. Nachdem die Anfrage behandelt wurde, springt der Programmzähler wieder auf die urspsrüngliche Position im Hauptprogramm und setzt die Ausführung dort fort. Damit dieser Ablauf korrekt funktioniert ist es notwendig, dass der aktuelle Programmzählerstand des Hauptprogramms gespeichert wird, um es nach Beendigung der *Interrupt Service Rountine* an der richtigen Stelle fortzusetzen. In der Regel wird der Programmzählerstand in einem Register der CPU oder auf dem Stack abgelegt. Damit die CPU bei einem *Interrupt Service Request* in die korrekte *Interrupt Service Routine* springt ist es zudem notwendig, dass die Startadressen der *ISRs* hinterlegt sind.

2.2.3 Interrupt Vector

Diese Hinterlegung der *Interrupt Service Routinen* wird durch den *Interrupt Vector* bewerkstelligt. Hinter diesem Terminus verbirgt sich eine Tabelle, die bei einer festgelegten Adresse im Hauptspeicher oder in einem gesonderten Modul hinterlegt wird. Jede Zeile bzw. Adresse steht dabei für einen bestimmten *Interrupt Service Request*, dem durch eine Eintragung eine *ISR* zugewiesen werden kann. Die Einträge in dieser Tabelle variieren von Mikrocontroller zu Mikrocontroller erheblich. Eine Information jedoch muss gespeichert werden, entweder die Startadresse oder ein Pointer auf die *ISR*.

[2]Genauer wird in Kapitel 3. auf die *ISR* eingegangen. Zunächst ist es ausreichend sie als Nebenprogramm bzw. aufgerufene Prozedur zu betrachten.

Abbildung 2.4: Interrupt 1 ist höher priorisiert und wird vollständig abgearbeitet bis der gepufferte Interrupt 2 zur Abarbeitung kommt

Heutzutage bietet fast jeder Mikrocontroller die Möglichkeit *Interrupt Requests* zu priorisieren. Hierzu wird im *Interrupt Vector* eine zusätzliche Priorisierungsinformation kodiert, welche bei einem Request ausgewertet wird. Sollte nun bereits eine *Interrupt Service Routine* einer höher priorisierten Anfrage abgearbeitet werden, wird zwar ein Flag im Interruptregister der CPU gesetzt, die *ISR* aber nicht unmittelbar ausgeführt. Abbildung 2.4 verdeutlicht diesen Mechanismus anhand eines zeitlichen Verlaufs. Dabei wird *Interrupt 2* zu einem späteren Zeitpunkt ausgeführt als der höher priorisierte *Interrupt 1*. Diese Funktionalität bietet wieterhin die Möglichkeit Interrupts zu verschachteln. Im Englischen als *Interrupt Nesting* bezeichnet. Dabei kann ein Interrupt mit einer niedrigen Priorität von einem Interrupt mit einer höheren Priorität unterbrochen werden. Als Beispiel sei hier die Bremse in einem Auto genannt. Obwohl gerade andere Dinge vom Bremssteuergerät bearbeitet werden, ist es wichtig, das ein Druck auf das Bremspedal vor allen anderen Berechnungen bearbeitet wird.

Ein weiterer wichtiger Aspekt der Priorisierung ist, dass Abschalten bestimmter Interrupts. Der Interruptmechanismus besitzt meist in einem Statusregister der CPU eine ”minimales“ Prioritätslevel. Durch eine niedrig priorisieren der abzuschaltenden Interrupts werden diese vom Mechanusmus nicht mehr berücksichtigt. Wie wir sehen werden ist diese Funktionalität sehr wichtig um *Interrupt Service Routinen* kritische Aufgaben bearbeiten zu lassen, ohne dass sie unterbrochen werden.

Damit eine Prozedur als *Interrupt Service Routine* im *Interrupt Vector* eingetragen wird, gibt es je nach verwandter Programmiersprache und deren Abstraktionslevel, verschiedene Herangehensweisen. Wird der Mikrocontroller in Assembler programmiert, ist es notwendig den Linker mit einem Kommando anzuweisen an der richtigen Stelle im *Interrupt Vector* einen Eintrag zu erstellen, der auf ein selbstgeschriebenes Unterprogramm verweist. Wird C bzw. C++ verwendet, ist es üblich, dass der Compiler bestimmte ’Pseudo‘- Schlüsselwörter akzeptiert, die ihn veranlassen selbständig für das entsprechenden Unterprogramm eine Eintragung zu tätigen. Ein solches Schlüsselwort kann z.B.
’ `_interrupt` ‘ sein, welches vor die als *ISR* zu verwendenden Prozedur geschrieben wird.

2.2.4 Interrupt Ablauf

Nun ist es sinnvoll den Ablauf eines Interrupts in Gänze darzustellen, da alle wichtigen Begrifflichkeiten bekannt sind. Die Abbildung[3] 2.5 stellt den typischen Ablauf eines solchen Vorgangs schematisch dar.

Das aktuell laufende Hauptprogramm wird durch einen Interrupt **(1)** unterbrochen und der Kontext gesichert. Dabei werden alle wichtigen Informationen und Variablen, die das Hauptprogramm benötigt, zwischengespeichert. Dazu zählt das Prozessorstatuswort, sämtliche Register und der Programmzähler. Hier nach erfolgt der Sprung in den *Interrupt Vector* **(2)**, aus welchem die Startadresse der *ISR* ausgelesen wird. Nun wird die *ISR* ausgeführt **(3)**. Diese allokiert für gewöhnlich einen neuen Stackframe, in dem sie ihre Berechnungen durchführen wird. Bei vielen Mikrocontrollern erfolgt ein impliziter *Interrupt disable*, so dass die *ISR* nicht durch weitere Interrupts unterbrochen werden kann. Nach dem die *ISR* abgearbeitet wurde erfolgt der Rücksprung in das unterbrochene Hauptprogramm **(4)**. Hier für wird der Kontext wieder hergestellt, so dass sich der Kontextwechsel für das Hauptprogramm vollkommen transparent darstellt. Zudem werden die Interrupts, durch Setzten des entsprechenden Statusworts, wieder *enabled*.

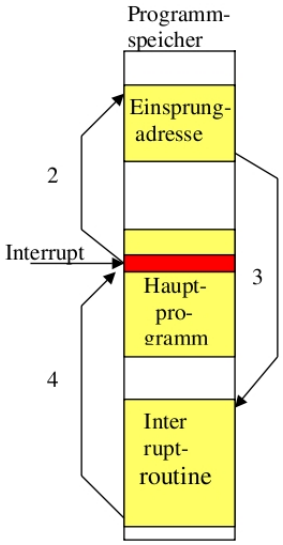

Abbildung 2.5: Ablauf eines Interrupts

Dieser Vorgang kann bei verschiedenen Mikrocontroller im Detail sehr unterschiedlich aussehen und benötigt ein genaues Studium des Handbuches. Die beschriebenen Punkte jedoch sind das Mindestmaß an notwendigen Schritten die für einen Wechsel in und aus der *ISR* benötigt werden. Die Details der Sicherung des Kontextes werden im folgenden Kapitel genauer diskutiert, sie bieten durchaus Optimierungspotential.

[3] entnommen von [5,http://www.boundscheck.com/knowledge-base/embedded/how-interrupts-work/669/]

Kapitel 3

Interrupt Service Routine

Einleitung

Nachdem im vorherigen Kapitel häufig von der *Interrupt Service Routine* gesprochen wurde, dient dieses Kaptiel der detaillierten Beschreibung. Weiterhin sind Kriterien genannt, die eine gute und schnelle *ISR* ausmachen. Zusätzlich werden bereits hier einige Probleme bei dem Design und der Realisierung einer *ISR* angesprochen.

3.1 Was sind Interrupt Service Routinen?

Die *Interrupt Service Routine* ist im Grunde genommen 'normaler' Programmkode, der sich nur in wenigen Details von dem des Hauptprogramms unterscheidet. In der Regel werden *ISR* wie Prozeduren kodiert, die keinen Rückgabewert besitzen und mit einem 'Pseudo'-Schlüsselwort wie `_interrupt` gekennzeichnet sind. Innerhalb dieser Prozedur lässt man den Mikrocontroller gezielt auf das Ereigniss, welche die Ausführung der *ISR* auslöst, reagieren. Zwar ist es möglich auch andere Arbeiten in einer *ISR* zu bewältigen, dies wiederspricht in den meisten Fällen jedoch der Idee, dass ein Ereigniss hier gekapselt behandelt werden soll.

Der Gestaltung und Nutzung von Funktionen sind innerhalb einer solchen Routine im Regelfall keine Grenzen gesetzt. So ist es z.B. möglich, aber nur bedingt empfehlenswert, während einer *ISR* auf ein LCD-Display zu schreiben.

Ein wichtiger Aspekt, der unbedingt beachtet werden sollte, ist die bereits erwähnte Sicherung des Hauptprogrammkontextes. Die Unterbrechung muss für das Hauptprogramm unbemerkt bleiben, ansonsten könnte dies zu Dateninkonsistenzen führen und das Programm stürzt ab. Eine solches Szenario tritt ein wenn z.B. eine *ISR* die Register der CPU nicht sichert sie aber verändert. Das Hauptprogramm kann diese Veränderung nicht feststellen und arbeitet nun mit den fehlerhaft modifizierten Daten weiter, die Folge sind wahrscheinlich unsinnige Berechnungen oder gar ein Absturz.

3.2 ISR Softwaredesign

3.2.1 Kontextwechsel

Damit solche schwerwiegenden Fehler, wie sie im vorherigen Abschnitt beschrieben wurden, vermieden werden, ist es üblich, die *ISR* den Kontext sichern zu lassen. Zur Sicherung des Kontexts werden die Register und das Prozessorstatuswort auf den Stack gelegt. Wird in einer Hochsprache programmiert, erzeugt der Compiler bei gekennzeichneten *ISRs* automatisch die nötigen Instruktionen zur Sicherung und Wiederherstellung des Kontextes. Je nach verwendeter Hardware und Compiler, kann diese Sicherung sehr unterschiedlich ablaufen. Es existieren auch Hardwareimplementationen, die selbsttätig den Kontext wechseln und somit keine Software für die Sicherung benötigen. Erfolgt die Programmierung des Mikrocontrollers in Assembler, ist es meist notwendig die nötigen Instruktionen selbst zu kodieren. Bei dieser, meist zeitraubenden, Tätigkeit, ist es jedoch möglich genau abzuwägen, ob und wieviel zu sichern ist. Benötigt z.B. die *ISR* für ihre Bearbeitung keine Register, ist es nicht notwendig diese zu sichern, da die Konsistenz der Daten gewahrt bleibt.

Ist nun der Kontext erfolgreich gewechselt, wird bei Stackmaschinen ein neuer Stackframe allokiert. Innerhalb dieses Stackframes führt nun die *ISR* ihre Berechnungen aus. Sollte ein Unterprogrammaufruf stattfinden oder die *ISR* von einer weiteren *ISR* unterbrochen werden, setzt diese ihrerseist einen neuen Stackframe oben auf den Stack. Nach Abschluss der Berechnung werden diese Stackframes wieder freigegeben. Bei Register basierenden Maschinen verläuft dieser Ablauf konzeptionell analog, bringt aber die Schwierigkeiten einer korrekten Registerallokation mit sich.

3.2.2 Optimierung

Wie bereist diskutiert, ist es meist sinvoll die *Interrupt Service Routine* 'schlank' zu halten. Mit 'schlank' ist hier die Menge und die Dauer der Aktionen gemeint, die in ihr kodiert sind. Eine gute Devise bei der Konzeption einer *ISR* ist folgender Satz: *So wenig Instruktionen wie möglich, so viele wie nötig.* Der Sinn einer solchen Routine ist in der Regel das schnelle und asynchrone Reagieren auf Ereignisse, eine lang andauernde Berechnung durchzuführen widerspricht dem. Allerdings möchte ich die Verwendung länger dauernder Operationen nicht gänzlich ausschließen, da es im Einzelfall durchaus sinvoll sein kann. Die Notwendigkeit eine *ISR* zeitlich so zu optimieren, dass sie adäquat auf Ereignisse reagiert, ohne das Hauptprogramm zu blockieren, kann soweit gehen, dass man ein Oszilloskop verwendet, um den zeitlichen Ablauf im Mikrosekundenbereich zu messen. Bemerkt man dabei, dass die *ISR* zu lange Laufzeiten aufweist, ist es notwendig ihren Programmfluss weiter zu optimieren. Auch wenn heutzutage häufig in einer Hochsprache wie C, C++ oder gar Java programmiert wird, ist es in diesem Fall notwendig, die wichtigsten Instruktionen auf Assemblerebene zu kodieren. Einige langzeit er-

fahrene Programmierer gehen sogar soweit zubehaupten, dass *Interrupt Service Routinen* ausschließlich in Assambler programmiert werden sollten. Dies erachte ich allerdings in den meisten Fällen für nicht notwendig, da heutige Compiler einen sehr hohen Grad an Optimierung aufweisen. Diese Optimierung geht bei sehr modernen Compilern so weit, dass das Kodieren der *ISR* in Assembler keinen messbaren Geschwindigkeitsvorteil bringt.

3.2.3 Kommunikation und Datenaustausch

Auch wenn eine *ISR* die Ereignissbehandlung gut innerhalb einer eigenen Prozedur kapselt, ist es meist notwendig die Grenzen dieser Kapselung aufzulösen und mit dem Hauptprogramm Daten auszutauschen.

Dieser Datenaustausch kann auf unterschiedlichste Weise geschehen. Die häufigste und auch eine der einfachsten Varianten ist das Anlegen einer statischen Variable, welche dann von der *ISR* und dem Hauptprogramm gelesen und geschrieben wird. Auch ist es möglich Registerinhalte so zu modifizieren, dass sie im Hauptprogramm zur Verfügung stehen. Diese Methode birgt allerdings einige Risiken. Eine Nichtbeachtung von *Receconditions* kann zu einem Programmabsturz führen, da wichtige Daten überschrieben werden. Eine noch heiklere Methode ist der direkte Stackeingriff. Dabei greift die *ISR* in den Stack des Hauptprogramms. Allerdings ist dies nur machbar, wenn man sich zu jedem beliebigen Zeitpunkt im Klaren darüber ist, welche Daten und Strukturen auf dem Stack liegen. Selbst der kleinste Berechnungsfehler des Stackabstandes oder ein unbeachteter Prozeduraufruf im Hauptprogramm verursacht ein Desaster, da Daten an eine unvorhersehbare Stelle geschrieben werden. Der eleganteste Weg einen Datentransfer zu bewerkstelligen ist sicherlich das Hantieren mit Pointern auf selbst erstellte Datenstrukturen. Hierbei hat man mehr Sicherheit, wie die Modifikationen der *ISR* aussehen werden.

Wie wir sehen sind bei einem solchen Datentransfer die immer wiederkehrenden Probleme der Datenkonsistenz und Nebenläufigkeit zu bewältigen. Da ein Interrupt zu einem nicht vorhersehbaren beliebigen Zeitpunkt eintreten kann, läuft das Hauptprogramm quasi mit der *ISR* parallel. Diese Nebenläufigkeit muss behandelt werden. Probleme der Interrupt-Technik leiten somit eine thematische Überleitung in des nächste Kapitel ein.

Kapitel 4

Probleme aus Software Sicht

Einleitung

Dieses Kapitel beschäftigt sich mit den Problemen, die bei der Verwendung von Interrupt-Techniken aus Softwaresicht entstehen. Einige von ihnen werden diskutiert sowie Lösungsansätze vorgestellt und bewertet. Um diese zu bewerkstelligen verwende ich ein Beispiel aus dem Buch "An embedded software primer"[1]. Das Beispiel stellt die Problematiken gut dar, ohne dass unnötiger Balast hinzu kommt. Verwendet wird hierführ das (fiktive) Monitoringprogramm eines Kernreaktors, welches permanent zwei Temperatursensoren abfragt und vergleicht. Ist die Temperatur der Sensoren nicht gleich, wird ein Unfall angenommen und ein Alarm ausgelöst. Das untenstehende Listing[2] 4.1 verdeutlicht den Programmaufbau.

Listing 4.1: Monitorinprogramm

```
1  static int iTemp[2];
2
3  void _interrupt vReadTemps (void)
4  {
5      iTemp[0] = // read value from hardware
6      iTemp[1] = // read value from hardware
7  }
8
9
10 int main(void)
11 {
12         ...
13         iTemp0 = lTemp[0];
14         iTemp1 = lTemp[1];
15         if(iTemp0 != iTemp1)
```

[1] [1, David E. Simon: An embeddes software primer: Addison-Wesley, 2007]
[2] in Anlehnung an [1, SIMON07]

```
16                      // Set off howling alarm
17                  ...
18          }
```

4.1 Shared-Date Problem

4.1.1 Beschreibung

Bei oben aufgeführtem Listing 4.1 entsteht ein schwer zu findender Fehler, der nur sporadisch und zufällig auftritt.

Die *Interrupt Service Routine* übernimmt hier das Auslesen und evtl. die Analog-Digital-Wandlung der Sensorenwerte. Sie ist, wie auch der Programmkode der `main`-Prozedur, sehr übersichtlich und leicht zu verstehen. Der fatale Fehler tritt auf, wenn das Hauptprogramm zwischen Zeile `13` und `14` unterbrochen wird. Da nun die *ISR* das globale Array mit neuen Werten beschreibt, sind die Werte der Variablen `iTemp0` und `iTemp1` nicht mehr konsistent. Der Wert in der Variable `iTemp0` spiegelt einen 'alten' Zustand wieder, während die Variable `iTemp1` bereits einen 'neuen' Zustand enthält. Da sie in Zeile `15` miteinander verglichen werden, und troz gleicher Temperatur an den Sensoren nun unterschiedliche Werte in den Variablen gespeichert sind, wird fälschlicherweise ein Alarm ausgelöst.

Die Problematik wird als *Shared-Data Problem* bezeichnet. Die gemeinsam genutzten Daten, über die die *ISR* mit dem Hauptprogramm kommuniziert, sind nicht gegen nebenläufige Unterbrechungen geschützt. Eine solche nebenläufige Unterbrechung kann auch an anderer Stelle auftreten, z.B. in Zeile `15`. Der Vergleich wird nicht atomar ausgeführt, sondern durch die Kompilation auf viele Assembleranweisungen abgebildet, so dass beim Schieben des Wertes der Variable `iTemp0` eine Unterbrechung stattfinden kann, welche den Wert der zweiten Variable `iTemp1` verändert. Das Resultat ist ein fehlerhaftes Auslösen des Alarms, da die erste Variable einen älteren Temperaturwert speichert als die Zweite.

4.1.2 Analyse

Anhand der Disskusion lässt sich das *Shard-Data Problem* auf die bereits bekannten zwei Kernproblematiken abstrahieren. Zum einen ist dies die Gewährleistung des konsistenten Zugriffes auf gemeinsam genutzte Ressourcen, zum anderen das ausschließen von unsynchronisierten nebenläufigen Unterbrechung. Meist entsteht diese Problematik anhand der Tatsache, dass bei der Abbildung von Hochsprachenkode auf Maschineninstruktionen Anweisungen nicht atomar abbildbar sind. Aus dieser Erkenntnis folgt unmittelbar der erste Ansatz zur Lösung der Problematik im nächsten Abschnitt.

4.1.3 Lösungsansätze

Interrupts ausschalten

Der simple und nahe liegende Lösungsansatz, während kritischer Abschnitte im Hauptprogrammkode die Interrupts temporär zu unterbinden, führt zunächst zum Erfolg. Der Programmfluss kann bei der Speicherung und dem Vergleich der beiden Variablen nicht mehr unterbrochen werden, so dass diese niemals in einen inkonsistenten Zustand geraten können. Um dies zu realisieren wird in Zeile `12` ein `interrup_disable();` respektive in Zeile `17` ein `interrupt_enable();` hinzugefügt.

Die Vorteile dieser Methode sind, wie bei den Eingangsbeispielen, die einfache Realisierung. Jeder interruptfähige Mikrocontroller bietet die Möglichkeit Interrupts ein- bzw. auszuschalten. Die Transparenz und der geringe Aufwand der Implementierung sprechen zusätzlich für diese Lösung.

Jedoch sind dieser Implementation, wie es bei den Simplen meist der Fall ist, einige Nachteile inhärent. Die Funktionalität ist durch das Abschalten der Interrupts eingeschränkt. So ist je nach Hardware die Reaktion auf Ereignisse, die während der abgeschalteten Interrupts stattfinden, verlangsamt oder gar nicht realisierbar. Hierbei kommt das bereits bekannte *Response-Problem* zum tragen, vgl. Interruptpriorisierung 2.2.3 sowie Abbildung 2.4. Gravierender jedoch ist die Tatsache, dass nicht sorgfältige Implementation bzw. unachtsam getätigte Prozeduraufrufe zu versteckten Bugs führen. So mag es notwendig sein, dass innerhalb eines kritischen Abschnittes, eine Prozedur, welche ihrer seits einen kritischen Abschnitt enthält, aufgerufen wird. Da die gerufene Prozedur die Interrupts aller Wahrscheinlichkeit nach wieder einschalten wird, sind die Interrupts innerhalb des kritischen Abschnittes des Hauptprogramms wieder möglich. Die Protektion vor nebenläufigen Unterbrechungen fällt somit weg und öffnet Raum für einen Absturz durch einen Datenfehler. Die Problematik lässt sich mit sorgfältig betrachteten Prozeduraufrufen und Prozeduren die *reentrant* sind lösen, wird aber bei steigender Komplexität äusserst unübersichtlich.

Optimistisches Lesen

Ein weiteres Verfahren, welches das *Shared-Data Problem* löst, ist das optimistische Lesen, hierbei wird eine Variable mehrfach verglichen. Ist sie bei diesen Vergleichen identisch, wird angenommen, das Datum sei konsistent. In Pseudokode kann dies so beschrieben werden:

Listing 4.2: Optimistisches Lesen

```
1 lReturn = lTemperature;
2 while(lReturn != lTemperature)
3     lRerturn = lTemperature;
```

Auch dieses Verfahren ist einfach zu implementieren und transparent. Es bietet zudem den großen Vorteil, dass Interrupts nicht ausgeschaltet werden müssen.

Somit erfolgt keine Erhöhung der Reaktionslatenz und ein Abarbeitungsstau noch wartender Interrupts wird vermieden. Diese Methode ist sehr performant, wenn wenige Unterbrechungen zu erwarten sind, da zwei- oder dreimaliges Vergleichen der Variable sehr schnell abgearbeitet werden.

Nachteile bei diesem Ansatz sind die mangelnde Performanz bei häufig zu erwartenden Unterbrechungen, die nicht optimale Laufzeit durch das *Polling* auf eine Variable und die Tatsache, dass ein moderner Compiler höchstwahrscheinlich die *Polling*-Schleife wegoptimieren wird. Gänzlicher Verzicht auf Optimierung löst letzteres Problem, ist aber keine wirkliche Alternative, da in vielen anderen Abschnitten des Programms nicht optimaler Kode generiert wird.

Pufferung von Variablen

Der letzte Ansatz, den ich hier vorstellen möchte, ist das Puffern von Variablen. Wie der Name dieses Verfahrens suggeriert, wird für eine Variable eine zweite, der Puffer, angelegt. Der Puffer wird von der *ISR* beschrieben, während das Hauptprogramm auf die Variable zugreift. Ein Flag, meist eine boolsche Variable, überwacht, dass das Hauptptogramm und die *ISR* niemals die gleiche Variable schreiben bzw. lesen. Hat das Hauptprogramm den Lesevorgang auf diese Variable beendet, wird die Zuordnung vertauscht. Abbildung 4.1 verdeutlicht dies anhand einer schematischen Darstellung.

Abbildung 4.1: Pufferung einer gemeinsam genutzten Ressource

Diese Lösung bietet ebenfalls den Vorteil, dass Interrupts nicht ausgeschaltet werden müssen und dass sie, bei einer kleinen Anzahl an Variablen, die so abgesichert werden, übersichtlich und leicht umzusetzen ist.

Jedoch ist es möglich, dass eine Änderung in der Puffervariable von der *ISR* bereits geschrieben wurde, das Hauptprogramm dies aber zunächst nicht bemerkt, da es noch die andere Variable liest. Unter Umständen braucht bei einem solchen Szenario das System zwei Durchläufe des Hauptprogramms, bis es eine Änderung berücksichtigen kann. Weiterhin ist nachteilig zu bemerken, dass durch die zusätzlichen Variablen ein gewisser Overhead geschaffen wird, der sich negativ auf die Performance des Systms auswirken kann. Bei steigender Komplexität wird diese Variante ebenfalls sehr schnell unübersichtlich und schlecht wartbar. Dies geht

soweit, dass das Vertauschen einer einzigen Anweisung, das Setzten des Flags an der falschen Stelle, das System bereits gefährdet.

Bei dieser Variante ist es noch erwähnenswert, dass die gepufferte Variable durch einen Ring-Puffer ersetzt werden kann. Diese Variation kapselt den Puffer in einer bekannten Datenstruktur, gleicht leider aber keinen Nachteil aus.

4.2 Ausblick

Für das *Shared-Data Problem* gibt es durchaus eine große Anzahl an Lösungsansätzen, die ihrerseits mit individuellen Vor- bzw. Nachteilen behaftet sind. Die Bekanntesten von ihnen wurden hier diskutiert,in der Hoffnung, dass der Leser nun befähigt ist, die korrekte Variante für seine Problemstellung auswählen zu können. Zudem wurde eine Sensibilität für die oftmals trivial anmutenden Probleme geschaffen.

Als Ausblick möchte ich hier allerdings einige bekannte und häufig angewandte Softwarepatterns nennen, welche dieses und andere Probleme lösen.

Im Bereich der Embedded Software ist es bei kleinen Systemen, wie z.B. einem Voltmeter, die nicht über einen größeren Mikrocontroller verfügen, durchaus üblich das *Round-Robin* Verfahren anzuwenden. Die Vorteile der betrachteten Implementation überwiegen hier die zumeist in diesem Kontext vernachlässigbaren Nachteile.

Als Erweiterung wird das *Round-Robin* Verfahren zusätzlich mit einen Interruptmechanismuss ausgestattet. So kann schnelles Reagieren auf einige Ereignisse realisiert werden, wohingegen der Rest der Implementation übersichtlich und transparent bleibt.

Ein weitaus ausgeklügelteres Verfahren ist das *Function-Queue-Scheduling*. Dieses Verfahren reiht die Ereignisse nach ihrer Priorisierung in einer Warteschlange an und arbeitet sie ab. Es ist jedoch mit einem gewissen Implementationsaufwand verbunden und sollte nur angewendet werden, wenn die zuvor genannten Lösungen nicht mehr anwendbar sind.

Als letzte, sehr mächtige, aber auch Overhead behaftete Lösung möchte ich hier die Verwendung eines *Real Time Operating Systems* oder kurz *RTOS* nennen. Ein solches *RTOS* bietet eine vielzahl Features, die die genannten Probleme kapseln und somit lösen. Zudem ist es durch ein solches System möglich weitere Programme auf der CPU auszuführen, da es über ein Task-Scheduling sowie Speichermanagement verfügt.

Die Auswahl der jeweiligen Lösung bleibt dabei dem Programmierer und Softwareengineer überlassen, der für ein gegebenes Problem immer die einfachste und schlankeste Lösung wählen sollte. Ein Versteifen auf einen Lösungsansatz ist eher kontraproduktiv und führt zu nicht optimalen Ergebnissen. Im Buch "An embedded software primer"[3] wird häufig erwähnt, dass es meist einfach sei ein *RTOS*

[3] [1, SIMON07]

zu verwenden. Diese quasi Empfehlung ist allerdings mit Vorsicht zu genießen, da ein solches System oft einen ungewollten Overhead mit sich bringt.

Kapitel 5

Fazit

Sicher gibt es eine große Anzahl weiterer Probleme und Fallstricke bei der Arbeit mit Interrupts und Interrupt Service Routinen, die hier keine Erwähnung fanden. Dies ist allerdings der nahezu übermächtigen Menge an Vielfältigkeiten der Probleme geschuldet. Zumeist birgt jede Hardware ihre speziellen Eigenheiten, die es durch sorgfältiges Studium des Handbuches sowie gut vorbereiteten Tests zu erforschen gilt. Erst nach einer solchen Einarbeitungsphase ist absehbar, welche der Lösungsansätze auf dem Mikrocontroller bzw. dem System realisierbar sind. Dabei gilt es folgende Devise zu beachten: *Differenzierte Auswahl der Lösung nach Problemstellung*. Meist gibt es 'die richtige' Lösung nicht, sondern eine korrekte Lösung muss von Fall zu Fall individuell durchdacht und getestet werden.

Tendenziell nimmt dies durch die Tatsache der Standarisierung von Mikrocontrollen respektive deren Designs ab, ist dennoch spätestens bei der Implementation einer eigenen Treiberbibliothek wieder von Nöten. Als eine solche Standardisierung sei hier die Cortex Mikrocontroller Familie und der mit ihr verbundene CMSIS genannt. Bei diesem Framework ist es für den Programmierer in der Regel nicht mehr notwendig, spezielle Kenntnisse über die Hardware zu besitzen. Es werden lediglich Libary Funktionen aufgerufen, die die Hardware korrekt konfigurieren und ansteuern.

Abschließend ist zu erwähnen, dass die Kernpunkte, die in dieser Arbeit dargelegt wurden, ungeachtet des Abstraktionslevels, sich meist zu einer kompakten Themengruppe zusammen fassen lassen. So ist die Datenkonsistenz sowie die Beachtung und Auflösung von Nebenläufigkeit und deren Synchronisation eine wichtige Thematik. In nahezu allen Bereichen der Informatik spielt sie eine Rolle, die gravierensten Folgen treten meiner Ansicht nach allerdings bei der hardwarenahen Programmierung auf, da hier meist das gesamte System abstürzt und nicht nur ein Teilbereich. Daraus resultierend sind einige der hier erörterten Lösungsansätze ebenfalls in anderen Bereichen anwendbar.

Anhang A

Quellenverzeichniss

1. Simon, David E.: An Embedded Software Primer: Addison-Wesley, 2007
2. Dr. Wüst, Klaus: Mikroprozessortechnik: Grundlagen, Architektur, Schaltungstechnik und Betrieb von Mikroprozessoren: Vieweg+Teubner, 2010
3. Geisse, Hellwig: Skript Betriebssysteme, 2010
4. Labrosse, Jean J.: Embedded Software: Newnes 2008
5. http://www.boundscheck.com/knowledge-base/embedded/how-interrupts-work/669/

Abbildungsverzeichnis