FAKULTÄT FÜR !NFORMATIK

Hardware Abstraction Layer für serielle Kommunikation und Software-Timer in embedded Devices

BAKKALAUREATSARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Jörg Rohringer
Matrikelnummer 0325451

an der
Fakultät für Informatik der Technischen Universität Wien
Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Mag.rer.soc.oec. Wolfgang Granzer

Abstract

Nowadays microcontrollers are part of nearly every electronic device. This document shall show a possibility to use functionality of different microcontrollers, without much knowledge about the internal procedures. This is an interesting point, because on the one hand a developer, who has to implement several functions on many different microcontrollers, needs much time to get used to the different types, if he implements every function from scratch. On the other hand, a company that accomplishes such project with different kinds of microcontrollers is interested on low costs.

The aim of this work is to design an interface, which should ease the usage of the SPI-protocol, the UART, and a software timer. This interface is implemented on 3 different microcontrollers from different vendors.

The document is split into 4 parts. The first part illustrates the functionality of the SPI-protocol, the UART, and the I^2C-protocol. The I^2C-protocol is only explained for the sake of completeness, but not implemented. The second part deals with the operating mode of a hardware and a software timer.
In the following part, the functionality of the APIs is described. The last part deals with an analysis of the execution time of the software timer.

Zusammenfassung

Microcontroller sind in der heutigen Zeit allgegenwärtig. Es sollen in dieser Arbeit Möglichkeiten vorgestellt werden, die es ermöglichen, bestimmte Komponenten eines Microcontrollers vereinfacht bedienen zu können. Dies ist deshalb interessant, weil in größeren Projekten oft verschiedene Microcontroller zum Einsatz kommen, welche alle unterschiedlich zu programmieren sind. Das bedeutet einerseits einen großen Zeitaufwand für Entwickler, um sich einzuarbeiten, und damit verbunden andererseits erhöhte Kosten für das Unternehmen in dem das Projekt durchgeführt wird.

Das Ziel dieser Arbeit ist ein Interface zu bieten, welches die Verwendung des SPI-Protokolls, des UARTs und eines Software Timers vereinfachen soll. Dieses Interface wird auf 3 verschiedenen Microcontrollern unterschiedlicher Hersteller implementiert.

Diese Arbeit gliedert sich in 4 Teile. Der erste Teil erklärt die Funktionsweise des UARTs, des SPI-Protokolls und des I^2C-Protokolls. Das I^2C-Protokoll wird zwar theoretisch der Vollständigkeit halber beschrieben, aber nicht implementiert. Der zweite Teil beschäftigt sich mit der Funktionsweise eines Software-Timers, basierend auf einem Hardware-Timer. Danach folgt die Beschreibung der implementierten APIs. Der letzte Teil beschäftigt sich mit der Analyse der Ausführungszeit der Software-Timer.

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung und Projektfeld

In der heutigen Zeit werden in vielen elektronischen Geräten Microcontroller eingesetzt, welche es ihnen ermöglichen, immer komplexere Aufgaben zu meistern. Vielfach werden Microcontroller unterschiedlicher Typen verwendet, von denen jeder Stärken und Schwächen aufweist. Oftmals ist auch die Auswahl schwierig aufgrund der Vielfalt der am Markt verfügbaren Microcontroller.
Jeder Microcontroller hat seinen eigenen Befehlssatz, mit dem er programmiert werden muss, und unterschiedliches Verhalten bei der Ausführung von Interruptroutinen oder dergleichen. Das Problem ist daher, dass sich ein Softwareentwickler für jeden Microcontroller neu einarbeiten muss und damit ein großer Zeitaufwand einhergeht, der zusätzliche Kosten verursacht.
Ziel dieser Arbeit ist es daher, eine Möglichkeit zu schaffen, welche die Eigenheiten des Microcontroller abstrahiert und den Zeitaufwand für die Softwareentwicklung reduziert. Dies soll damit erreicht werden, indem für bestimmte Komponenten eines Microcontrollers eine API (**A**pplication **P**rogramming **I**nterface) geschaffen wird, welche eine hardware-unabhängige Schicht implementiert. Über ein Interface soll es möglich sein, diese Komponenten verwenden zu können, ohne sich allzu sehr mit den inneren Vorgängen beschäftigen zu müssen. Das kann helfen, Softwareentwicklungen zu beschleunigen und Kosten zu senken. Dieses Konzept soll durch Abbildung 1 verdeutlicht werden. Diese API soll im Bezug auf den Speicherplatz klein und schlank sein.
In der vorliegenden Arbeit wurde:

- eine UART-API (**U**niversal **A**synchronous serial **R**eceiver and **T**ransmitter API),
- eine SPI-API (**S**erial **P**eripheral **I**nterface API),
- und eine Timer-API

entwickelt und implementiert. Der Vollständigkeit halber wird noch das I^2C- oder TWI-Kommunikationprotokoll erklärt, da es neben dem UART- und SPI-Protokoll eines der am weitesten verbreiteten Kommunikationsprotokolle darstellt. Dieses wird allerdings bei der Implementierung nicht berücksichtigt.

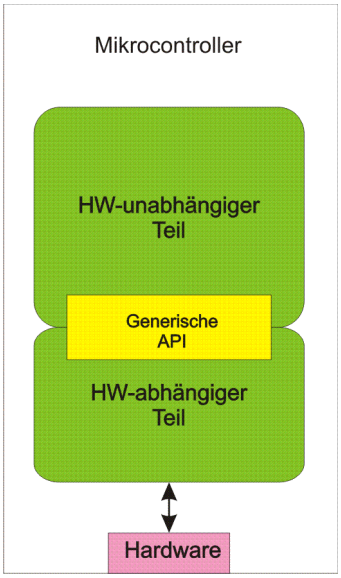


Abbildung 1: Grundsätzlicher Aufbau der APIs

2 Funktionsweise Kommunikationsprotokolle

2.1 UART-Protokoll

Die Entwicklung des UARTs ist geschichtlich gesehen sehr eng verbunden mit der Standardisierung des Datenkommunikationsprotokolls RS-232. Dabei handelt es sich um ein serielles Übertragungsverfahren zwischen 2, oder mehreren, Kommunikationspartnern. Mit diesem Protokoll können, sowohl asynchron, als auch synchron, Daten in Form von 5 bis 9-bit langen Worten übertragen werden. Bei der asynchronen Übertragung, welche meistens verwendet wird, benötigt man streng genommen nur 2 physikalische Leitungen zwischen dem Sender und dem Empfänger: eine Sendeleitung und eine Empfangsleitung. Dem eigentlichen Datenwort müssen bei der Übertragung noch Informationen in Form von Bits hinzugefügt werden, damit der Kommunikationspartner die Daten richtig empfangen kann. Bei der synchronen Datenübertragung sind diese Informationen nicht notwendig, da man zusätzlich zu den Datenleitungen noch eine Taktleitung mitführt.
Der Vorteil dieses Protokolls liegt darin, dass nur zwei bzw. drei Leitungsadern von Nöten sind, um zu kommunizieren, und dass die maximale Leitunglänge sehr groß ist. Nachteilig wirkt sich allerdings aus, dass alle Kommunikationsparameter bei den kommunizierenden Modulen vorher explizit konfiguriert und parametriert werden müssen.

2.1.1 Einstellungsmöglichkeiten

Vor der Übertragung mit dem UART, müssen einige Parameter bei den Kommunikationsmodulen konfiguriert werden:

- Die **Bitrate** ist die Übertragungsgeschwindigkeit. Wird zum Beispiel eine Bitrate von 19200bps(**b**its **p**er **s**econd API) ausgewählt, dann werden in 1 Sekunde 19200 Zeichen (Bits) übertragen.
- Die **Anzahl zu übertragenden Datenbits**, welche zwischen 5 und 9 Bits betragen kann.
- Die **Paritätsinformationen** geben Auskunft darüber, ob ein Datenwort einen Fehler aufweist. Dabei wird dem Datenwort ein Bit angehängt. Es kann bei dieser Fehlererkennung allerdings nur ein fehlerhaftes Bit sicher erkannt werden. Sind mehrere Bits fehlerhaft ist eine Fehlererkennung nicht mehr möglich. Es kann zwischen drei verschiedenen Paritäten gewählt werden. Wie diese Paritätsbits dargestellt werden, ist in Abbildung 2 zu sehen.
  - **Keine Paritätsinformation:** Hier wird dem Datenwort kein Paritätsbit hinzugefügt.
  - **Gerade Paritätsinformation:** Es werden die Bits des Datenwortes mittels einer XOR-Verknüpfung verbunden. Das Ergebnis ist das anzuhängende Paritätsbit.
  - **Ungerade Paritätsinformation:** Es werden ebenfalls die Bits des Datenwortes mit einer XOR-Verknüpfung verbunden. Das Ergebnis wird allerdings noch invertiert, bevor es an das Datenwort angefügt wird.
- Die **Stoppbits**, wobei eines oder zwei ausgewählt werden können. Diese werden nach der Paritätsinformation angehängt.

2.1.2 Kommunikationablauf

Die Kommunikation zwischen UART-Modulen kann auf mehrere Arten stattfinden:

- **Asynchrone Datenübertragung zwischen 2 Knoten:** Es werden hier jeweils einzelne Datenworte mit einer Größe von 5 bis 9 Bits übertragen. Diese Konfiguration ist in Abbildung 3 zu sehen. Dabei müssen

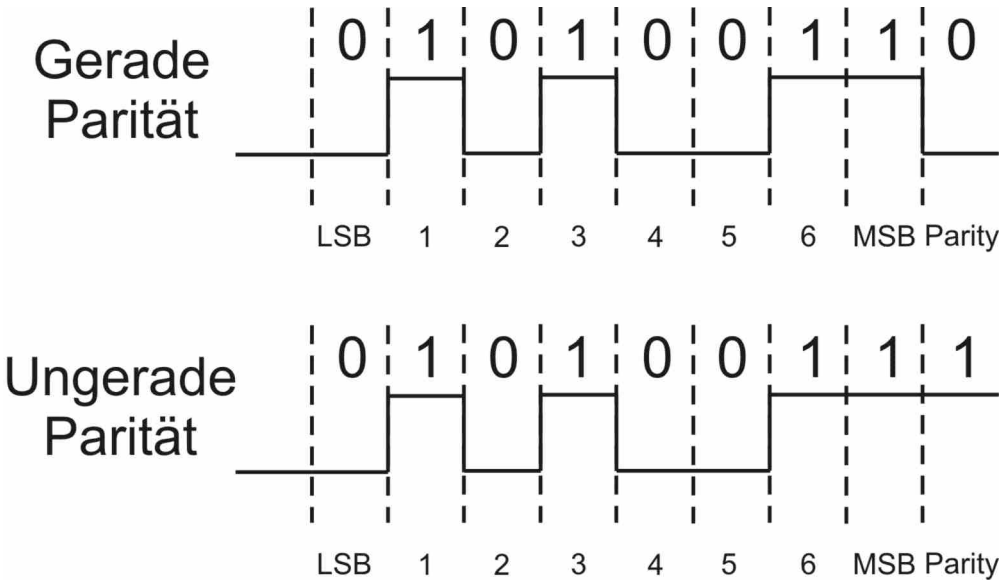


Abbildung 2: Die verschiedenen Möglichkeiten der Paritätsbits

vor der Übertragung die Kommunikationsparameter festgelegt werden, um eine klare Verständigung zwischen den Knoten zu gewährleisten. Einerseits muss die Übertragungsgeschwindigkeit, andererseits das verwendete Paritätsbit zur Fehlererkennung, sowie die Anzahl der verwendeten Stoppbits festgelegt werden. Danach kann die eigentliche Datenübertragung beginnen. Hierbei werden die Bits nacheinander, beginnend mit

- einem Startbit,
- danach das eigentliche Datenwort, wobei das LSB (**L**east **S**ignificant **B**it) zuerst gesendet wird,
- dann das Paritätsbit
- und die Stoppbits übertragen.

Diese Bits werden nach dem NRZ (**N**on **R**eturn to **Z**ero) Verfahren kodiert. Näheres dazu ist im Abschnitt über den physikalischen Kommunikationsablauf zu finden.

- **Asynchrone Datenübertragung zwischen mehreren Knoten:** Wird dieser Modus ausgewählt, dann verhält sich die Kommunikation wie bei einem Bussystem. Dabei müssen sich alle kommunizierenden Module im selben Modus befinden. Das Mastermodul sendet ein

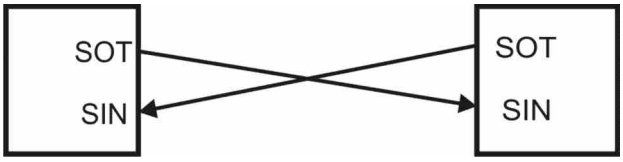


Abbildung 3: Schaltbild zur asynchronen Datenübertragung zwischen 2 Knoten

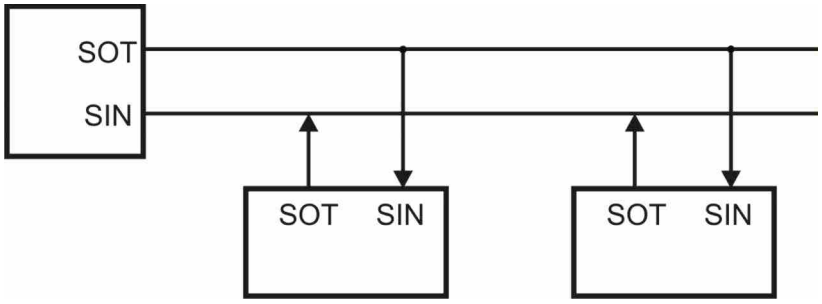


Abbildung 4: Master-Slave Komunikation mit dem UART

9-Bit Adresswort, wobei das 9-te Bit signalisiert, dass es sich um eine Adressinformation handelt. Alle Slavemodule erhalten dieses Datenwort und lesen es aus. Das Modul, welches adressiert wurde, kann danach die folgenden Datenwörter empfangen. Die Entscheidung, ob ein Slave adressiert wurde oder nicht, muss vom Programmierer getroffen werden. Dieser, beziehungsweise dessen Programmcode, muss die empfangene Adresse mit der Adresse des Slaves vergleichen und gegebenenfalls den Empfang aktivieren. Jene Module, deren Adresse nicht mit der gesendeten übereinstimmen, ignorieren die folgenden Datenwörter. In Abbildung 4 ist solch eine Konfiguration zu sehen. In dieser Abbildung sind die Pins SIN die Pins der Komponenten, auf denen ein Datenwort empfangen wird, und die Pins SOT die Pins auf denen ein Datenwort gesendet wird.
Diese Art der Kommunikation wird zwar theoretisch beschrieben, ist jedoch in der Implementierung nicht enthalten.

- **Synchrone Datenübertragung zwischen 2 Knoten:** Bei dieser Übertragungsart mittels UART werden Datenworte mit Hilfe eines Taktsignals synchron übertragen. Dazu ist eine zusätzliche Taktleitung zu den vorhandenen Sende- und Empfangsleitungen von Nöten. Es entfallen bei dieser Kommunikationsvariante die Start- und Stopp-Bits vollständig, da diese bei den vorher besprochenen Übertragungen

für den Empfänger nötig waren, um sich zu synchronisieren. Es ist auch möglich Paritätsbits zu verwenden, jedoch unterstützen nicht alle Microcontroller die Verwendung eines Paritätsbits bei der synchronen Datenübertragung. Diese Übertragungsvariante ist in Abbildung 5 zu sehen.

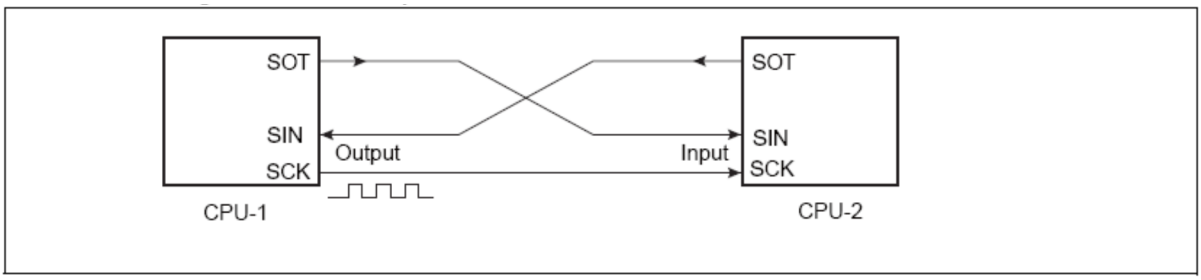


Abbildung 5: Schaltbild zur synchronen Datenübertragung [MSP430]

2.1.3 Physikalischer Kommunikationsablauf

Physikalisch gesehen werden vom UART einzelne Bits seriell auf einer Leitung gesendet, und auf einer zweiten empfangen. Die Daten werden dabei nach dem NRZ Verfahren codiert. Dabei werden die logischen Zustände "0" und "1" als unterschiedliche Signalpegel auf der Übertragungsleitung angelegt. Abbildung 6 soll diesen Aspekt verdeutlichen. Welche Werte diese Signalpegel aufweisen, hängt vom verwendeten UART ab.

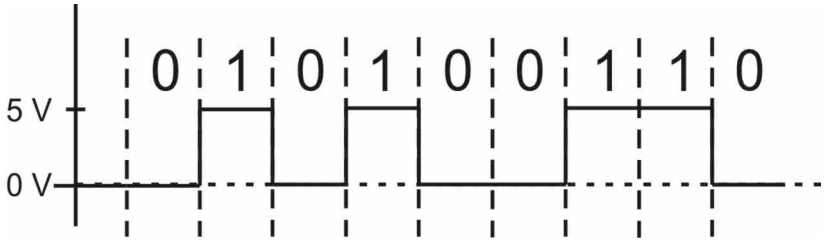


Abbildung 6: Bitfolge in NRZ-Kodierung

Abbildung 7 zeigt die am häufigsten verwendeten Datenformate.

2.1.4 Verwendungsmöglichkeiten

Grundsätzlich ist der UART eine Schnittstelle zu Peripheriegeräten wie beispielsweise USB-Chips oder dem TPUART (hierbei handelt es sich um ein

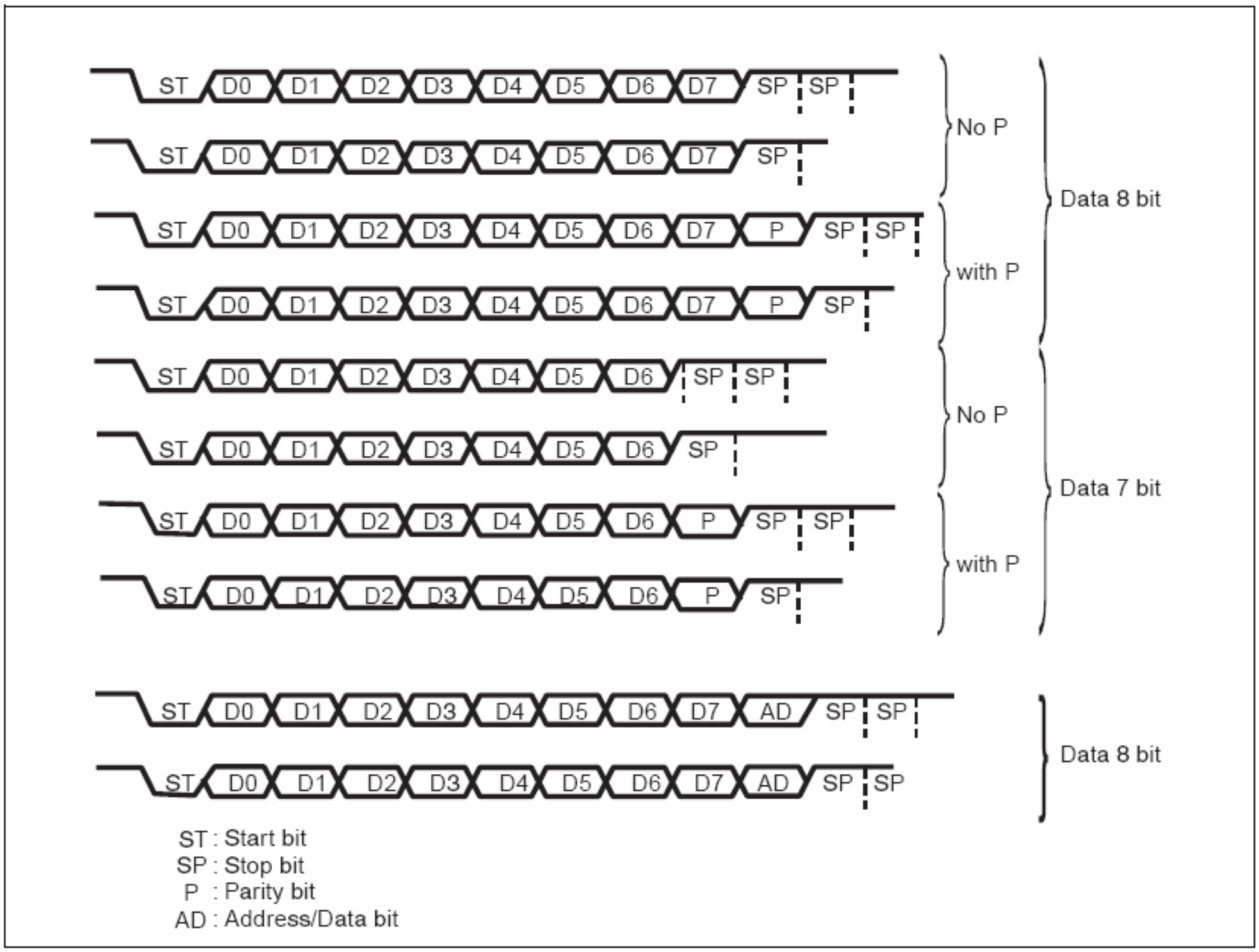


Abbildung 7: Framekodierung des UARTS [MSP430]

Schnittstellenmodul zur Kommunikation mit KNX/EIB(**E**uropean **I**nstallation **B**us), welcher in der Gebäudeautomatisierung zum Einsatz kommt). Außerdem kommt dem UART eine große Bedeutung in Bezug auf Debugging zu. Bei der Programmierung von Microcontrollern, welche auch in vielen Fällen UARTs besitzen, ist diese Art der Kommunikation sehr hilfreich zum Auffinden von semantischen Fehlern, indem man Informationsnachrichten an den Computer versendet. Die meisten Computer besitzen UARTs und können mithilfe von Konsolen dem Nutzer empfangene Daten in Form von lesbaren Zeichen ausgeben. Diese Verwendungsart bezeichnet man auch als HMI (**H**uman **M**achine **I**nterface), da der Nutzer nicht nur Nachrichten empfangen, sondern auch senden kann.

2.2 SPI-Protokoll

Das SPI (**S**erial **P**eripheral **I**nterface) bezeichnet ein Bussystem, welches ursprünglich von Motorola entwickelt wurde. Da es kein offizieller Standard ist, unterscheiden sich die Implementationen auf diversen Microcontrollern teils erheblich. Deswegen beschreibt diese Arbeit das SPI-Protokoll aufgrund der SPI-Implementierung der verwendeten Microcontroller, und [SPI].

In einem Verbund aus SPI kommunizierenden Modulen wird grundsätzlich zwischen Master und Slave unterschieden, wobei es mehrere Master und mehrere Slaves geben kann. Da bei diesem Protokoll die Kommunikation zwischen Master und Slave gleichzeitig erfolgt (full duplex), sind mindestens 3 Leitungen (MISO (**M**aster **I**n-**S**lave **O**ut) oder SOMI (**S**lave **O**ut-**M**aster **I**n), MOSI (**M**aster **O**ut-**S**lave **I**n) oder SIMO (**S**lave **I**n-**M**aster **O**ut), und SCK (**S**erial **C**loc **K**)) notwendig. Es ist jedoch auch eine 4-Pin Konfiguration möglich. Hierbei wird noch eine zusätzliche Signalleitung benötigt, nämlich die SS (**S**lave **S**elect) Leitung. In der Literatur wird bei dieser Leitung auch von STE (**S**lave **T**ransmit **E**nable) oder CS (**C**able **S**elect) gesprochen. Bei SPI handelt es sich um ein synchrones Datenübertragungsverfahren, es werden also nur die Datenworte, ohne Synchronisationsbits und ohne Paritätsbits, übertragen. Es wird daher eine separate Taktleitung (SCK) benötigt. Diese Taktleitung wird vom Master betrieben.

2.2.1 Kommunikationsablauf

Um die Übertragung zu starten, ist es notwendig das zu übertragende Datenwort in das dafür vorgesehene Ausgangsdatenregister des Masters zu schreiben. Abhängig davon ob der 3-Pin oder 4-Pin Mode bei der Initialisierung des Microcontrollers aktiviert wurde, muss beim 4-Pin Mode noch der SS Signalisierungspin aktiviert werden. Beim 3-Pin Mode entfällt diese Aktivierung. Daraufhin wird ein Taktsignal auf der SCK-Leitung generiert, womit der Master seine Daten auf dem MOSI-Pin anlegt. Da es sich bei dieser Kommunikation um eine Full-Duplex Übertragung handelt, überträgt der Empfänger daraufhin gleichzeitig, synchron mit dem SCK-Signal, auch seine Daten auf dem MISO-Pin.

Wann das nächste Bit des Datenwortes bei der Sendeleitung angelegt, und das zu empfangende auf der Empfangsleitung abgetastet wird, hängt davon ab, welcher Modus dafür gewählt wurde. Durch diesen Modus wird die Taktpolarität (CKPL) und die Taktphase (CKPH) eingestellt. Die Taktpolarität gibt an, ob das Taktsignal in der Ruhephase logisch “0“ oder logisch “1“ ist. Die Taktphase beschreibt, wann das nächste Datenbit ausgegeben wird. Ist CKPH logisch “0“ findet die Datenübernahme bei jeder ungeraden Flanke

des Taktsignals statt, ist CKPH logisch “1“ werden die Daten bei der geraden Flanke übernommen. Tabelle 1 zeigt die möglichen Modi. In Abbildung 8 sind die verschiedenen Modi für ein besseres Verständnis zu sehen.

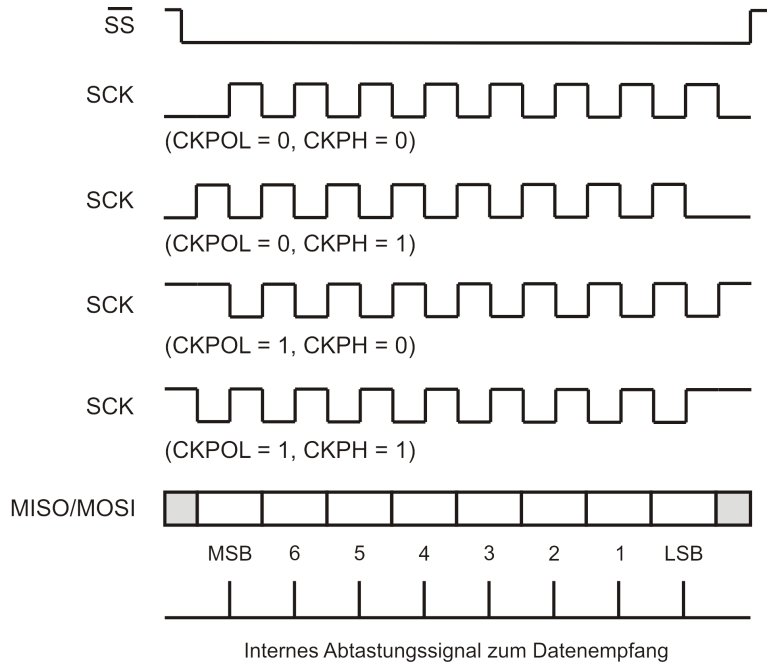


Abbildung 8: Alle SPI-Modi im Überblick [MSP430]

Modus	CKPL	CKPH
0	0	0
1	0	1
2	1	0
3	1	1

Tabelle 1: Mögliche SPI-Modi

3-Pin SPI-Mode Dieser SPI-Mode verwendet nur die Datenleitungen MISO und MOSI, und die Taktleitung SCK. Abbildung 9 zeigt die Beschaltung eines SPI Master-Slave-Netzwerks im 3-Pin Modus. Diese Beschaltung eignet sich nur zur Kommunikation *eines* Masters mit *einem* Slave, da es beim SPI-Protokoll keine explizite Adressierung gibt. Sollen mehrere Slaves oder Master verwendet werden, eignet sich nur der 4-Pin Modus dafür.

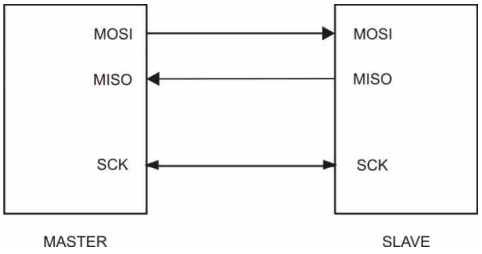


Abbildung 9: 3-Pin Konfiguration

4-Pin SPI-Mode Bei dieser Konfiguration kommt zu den, im vorherigen Abschnitt, genannten Leitungen noch eine hinzu, nämlich die SlaveSelect(SS)- oder CableSelect(CS)-leitung. Abbildung 10 zeigt solch eine Konfiguration. Dabei wird die Adressierung der einzelnen Slaves von den zu ihnen führenden SS-Leitungen übernommen. Soll ein Slave adressiert werden, so wird die SS-Leitung des betreffenden Slaves, welche im Ruhezustand logisch "1" ist, auf logisch "0" gesetzt. Beim verwendeten Modus ist es auch möglich mehrere Master nebeneinander zu betreiben und diese kommunizieren zu lassen. Diese Einstellung muss jedoch vorher bei allen im Netzwerk befindlichen Masterbausteinen getroffen werden.

Abbildung 10: 4-Pin Konfiguration

2.2.2 Verwendungsmöglichkeiten

SPI wird hauptsächlich dazu verwendet, Microcontroller untereinander, aber auch Peripheriegeräte, wie zum Beispiel Speicherbausteine, anzubinden. Zu den bekanntesten Peripheriebausteinen zählt sicher die MMC-Speicherkarte, welche kompatibel zur SD-Speicherkarte ist. Um SPI zum Verbinden von Peripheriebausteinen verwenden zu können, sollten sich alle Bausteine auf der selben Platine befinden, da es bei physikalischer Trennung zu Störungen kommen kann. SPI verfügt nämlich über keinen Störschutz gegenüber externen Signalen. Der Unterschied zum UART im synchronen Modus besteht darin, dass bei einem SPI-Master und einem SPI-Slave keine aufwändigen Adressierungsroutinen implementiert werden müssen. Es reicht einem Master im 4-Pin-Mode eine logische "0" auf der SS-Leitung auszugeben, um einen Slave zu adressieren. In allen anderen Belangen sind sich SPI und UART sehr ähnlich.

2.3 I²C-Protokoll

Der I²C-Bus wurde von Philips Semiconductor entwickelt. Er beschreibt ein Bussystem, in dem es keine a priori festgelegten Master oder Slaves gibt. Diese werden bei jeder Kommunikationsrunde neu ausgewählt. Für die Kommunikation werden zwei Leitungen benötigt, eine SDA-Leitung (**S**erial **DA**ta) und eine SCL-Leitung (**S**erial **CL**ock). Jede Buskomponente hat eine eindeutige Adresse, über die sie adressiert werden kann. Die Komponenten müssen vor der Datenübertragung die SCL-Leitung initialisieren (siehe Kapitel 2.3.1), um sich zu synchronisieren. Hat danach eine Komponente Kontrolle über die Busleitung erlangt, kann sie einen Slave adressieren, um ihm Daten zu senden oder Daten von ihm zu empfangen. Dies wird durch das R/W-Bit, welches im Anschluss an die Adresse gesendet wird, signalisiert. Logisch "1" bedeutet, dass der Master Daten empfangen will, logisch "0" dass der Master Daten sendet. Anschließend wird das eigentliche Datenwort gesendet, wobei das MSB (**M**ost **S**ignifikant **B**it) zuerst gesendet. Nach jedem gesendeten Byte muss der Slave ein Bestätigungsbit senden. Empfängt der Master Daten, muss er jeweils ein Bestätigungsbit für jedes Byte senden. Dieses Bestätigungsbit ist eine logische "0" nach dem letzten übertragenen Bit. In Abbildung 11 ist solch ein Datentransfer zu sehen.

Dieses Bussystem hat in der ursprünglichen Konfiguration die Möglichkeit, Daten mit einer Geschwindigkeit von bis zu 100kbits/s zu übertragen. In der aktuellen Spezifikation befinden sich auch weitere Varianten, mit ähnlichem Übertragungsmuster, aber höheren Geschwindigkeiten. Das ist einerseits der Fast-Mode, mit einer Übertragungsgeschwindigkeit von bis

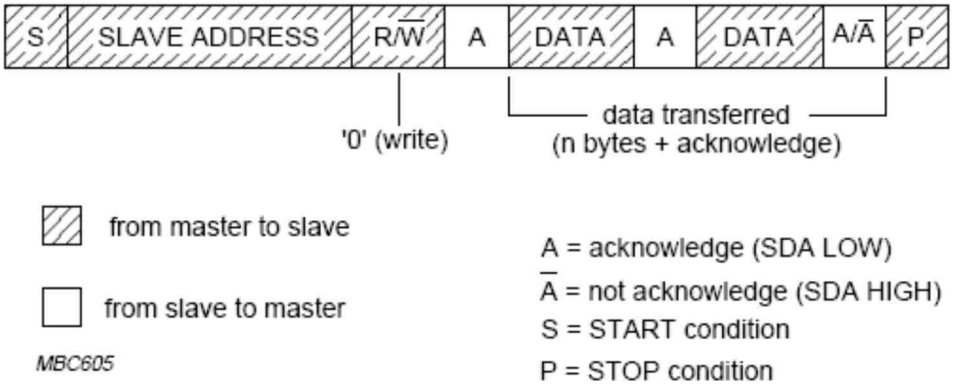


Abbildung 11: Datentransfer mit I²C [I²C-Bus]

zu 400kbit/s, und andererseits der Hs-mode (**H**igh-**s**peed Mode), mit einer Übertragungsgeschwindigkeit von bis zu 3,4Mbits/s. Komponenten, die diese hohen Übertragungsstandards beherrschen, sind jedoch alle abwärtskompatibel.

2.3.1 Kommunikationablauf

Synchronisation

Als erstes müssen sich die Buskomponenten synchronisieren. Dabei wird die Taktleitung von einer Komponente von High auf Low gezogen (Low ist gegenüber High dominant), die anderen Komponenten erkennen diesen Pegelwechsel und setzen ihre Taktleitung ebenfalls auf Low. Jede Komponente hat eine vordefinierte Zeitdauer, wie lange sie die Taktleitung auf Low hält. Haben alle Komponenten ihre Low Periode beendet, befindet sich die Taktleitung wieder auf High. Die erste Komponente, welche ihre High Periode abgeschlossen hat, veranlasst die nächste Pegeländerung auf Low. Damit entspricht dem Taktsignal die Low Periode der Komponente mit der längsten Low Periode, und die High Periode der Komponente mit der kürzesten High Periode. Abbildung 12 soll diesen Zusammenhang verdeutlichen.
Dieses Synchronisierungsverfahren bietet auch eine Möglichkeit, langsamere I²C-Komponenten mit schnelleren kommunizieren zu lassen. Ist zum Beispiel eine Komponente zwar in der Lage schneller Daten zu empfangen, diese aber nur langsam zu speichern oder zu verarbeiten, kann die Taktleitung auf Low gezogen werden, bis die Verarbeitung beendet ist, um die schnellere Komponente zu bremsen.

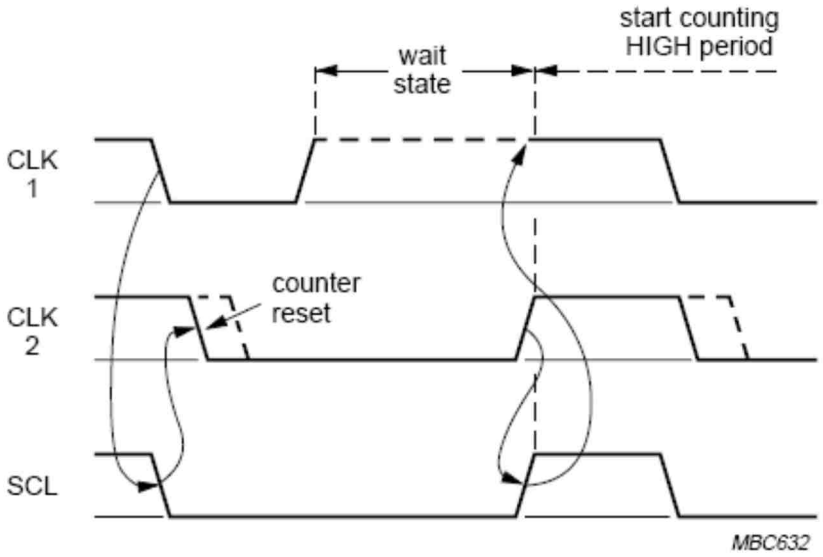


Abbildung 12: Taktsynchronisierungsvorgang [I²C-Bus]

Arbitration
Bei der Arbitrierung wird unter den Komponenenten entschieden, wer senden darf. Dabei zieht der Sender die Datenleitung (SDA-Leitung) auf Low, während das Taktsignal High ist, da in diesem Protokoll übertragene Daten nur als gültig betrachtet werden, wenn die Taktleitung High ist. Das entspricht dem Startbit. Danach wird die Adresse des Slaves gesendet, welcher angesprochen werden soll. Senden mehrere Master gleichzeitig, verliert derjenige, der als erstes von der Datenleitung eine logische “0“ einliest, obwohl er eine logische “1“ senden wollte (auch hier ist logisch “0“ dominant gegenüber logisch “1“). Sendet ein Master die Adresse einer anderen Komponente, die auch als Master senden wollte, so muss diese sofort nach dem misslungenen Sendeversuch in den Slave Modus wechseln, um die übertragenen Daten empfangen, beziehungsweise senden zu können. Verliert ein Master die Arbitration, in der er nicht adressiert wurde, kann er bis zum Ende des Bytes, in dem er verloren hat, das Taktsignal weiter übertragen. Die Arbitrierungsphase kann auch über die Sendung der Adresse hinausgehen, wenn Daten gesendet werden. Dies ist beispielsweise der Fall wenn 2 Master ein und den selben Slave adressieren wollten. Nachdem die Daten erfolgreich übertragen wurden, kann der Master ein weiteres Startbit senden, sonst schließt er die Transmission mit einem Stoppbit ab. Das Stoppbit entspricht einer Flanke von Low auf High, während die SCL-Leitung High ist. Abbildung 13 zeigt

die Arbitrierungsphase zwischen zwei Mastern.
Handelt es sich bei den konkurrierenden Komponenten um Hs-Mode Master, so wird die Arbitrierungsphase auf jeden Fall im ersten übertragenen Byte abgeschlossen, da im Hs-Mode im ersten Byte die Adresse des Senders angegeben wird, und diese eindeutig ist.

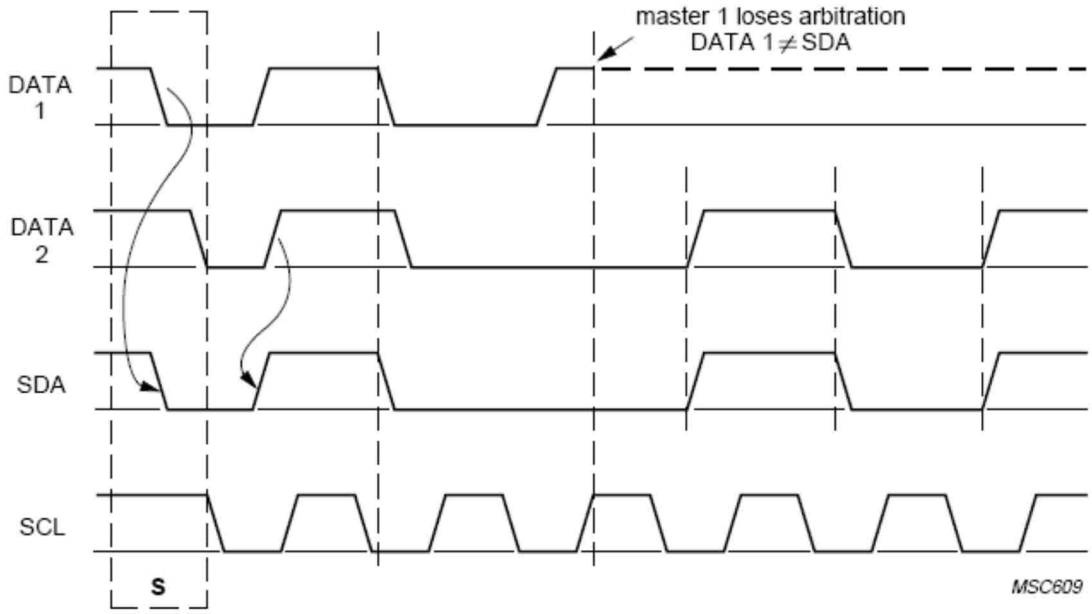


Abbildung 13: Arbitrierungsvorgang [I^{2}C-Bus]

Acknowledge
Nach jedem gesendeten Byte hat der Empfänger die Möglichkeit ein Acknowledge (ACK) Bit zu senden. Dazu setzt der Master nach der Übertragung die SDA-Leitung auf logisch "1". Der Slave kann daraufhin die SDA-Leitung wieder auf logisch "0" (ACK) setzen, um dem Master zu signalisieren, dass die Daten erfolgreich empfangen wurden, oder er sendet ebenfalls eine logische "1" (NACK - **N**ot**ACK**nowledge) falls der Empfang nicht erfolgreich war. Der Master kann nun ein Stoppbit senden, um die Übertragung zu beenden, oder ein weiteres Startbit, um den Übertragungsvorgang zu wiederholen. Die Übertragung eines NACK kann mehrere Gründe haben:

1. Kein Slave, der an dem Bus angeschlossen ist, hat die Adresse, an welche der Master senden wollte.

2. Der Slave ist gerade beschäftigt und kann die Übertragung nicht annehmen (wenn er beispielsweise gerade zeitkritische Aufgaben durchführt).

3. Der Slave erhält bei der Übertragung unklare Daten oder Befehle.

4. Der Slave kann nicht mehr Daten empfangen (beispielsweise nicht genügend Speicherplatz).

5. Ein Master, der von einem Slave Daten empfangen hat, signalisiert damit das Ende der Übertragung.

Adressierung

Nach dem Senden des Startbits wird die Slaveadresse übertragen. Es gibt zwei Arten von Adressen:

7-Bit Adressen: Hierbei können bis zu 112 Geräte miteinander verbunden werden. Dabei entsprechen die ersten sieben Bits der Adresse. Das 8te Bit, das LSB, beschreibt die Richtung des Datentransfers. Ist dieses Bit 0, so signalisiert der Master, dass er Daten senden will, ist es 1, muss der Slave Daten senden. Möchte der Master nach der erfolgreichen Übertragung weitere Daten von dem selben oder einem anderen Slave empfangen oder senden, kann er das, indem er anstatt eines Stoppbits ein weiteres Startbit (repeated Start condition) sendet.
Die Slaveadresse kann aus einem fixen, zum Zeitpunkt der Herstellung festgelegten, und einem programmierbaren Teil bestehen. Dies hat den Vorteil, dass bei mehreren gleichen Bausteinen, für jede Komponente eine eigene individuelle Adresse vergeben werden kann. Es gibt allerdings auch Adressen die nicht für die 7-Bit Adressierung von Buskomponenten herangezogen werden dürfen. In Tabelle 2 sind diese Adressen aufgelistet.

10-Bit Adressen: Bei dieser Adressierungsart stehen insgesamt 1024 Adressen zur Verfügung. Die 10-Bit-Slave Adresse wurde eingeführt um einen größeren Adressraum zu ermöglich. Dabei werden die ersten beiden Bytes nach dem Startbit, als Adresse herangezogen. Das erste Byte enthält dabei die Information, dass es sich bei der Adresse um eine 10-Bit Adresse handelt und das 9te und 10te bit dieser Adresse. Alle Slaves vergleichen diese 2 Bits der Adresse und senden daraufhin ein Acknowledge.
Ist das R/$\bar{\text{W}}$-Bit des ersten Byte 0, dann möchte der Master etwas zu einem

Slave Adresse	R/$\bar{W}$-Bit	Bedeutung
0000 000	0	General Call Address
0000 000	1	Start Byte
0000 001	X	CBUS-Adresse
0000 010	X	reserviert für andere Busformate
0000 011	X	reserviert für zukünftige Zwecke
0000 1XX	X	Hs-Modus Master Code
1111 1XX	X	reserviert für zukünftige Zwecke
1111 0XX	X	10-Bit Adressierung

Tabelle 2: Reservierte Slaveadressen

Slave senden, und das zweite Byte enthält die restliche Adresse des Empfängers. Nach dieser Adresse sendet nur noch ein Slave ein Acknowledge. Dieser Slave bleibt auch vom Master adressiert, bis ein Stoppbit empfangen wird.
Möchte der Master hingegen von einem Slave Daten empfangen, muss folgendermaßen vorgegangen werden. Anfangs sendet der Master die beiden Bytes mit der Slaveadresse, wobei er das R/$\bar{W}$-Bit auf 0 setzt. Jetzt ist ein Slave fix adressiert. Danach kann der Master ein neues Startbit setzen. Nun folgt nocheinmal das erste Byte der 10-Bit Adresse, jedoch wird das R/$\bar{W}$-Bit nun auf 1 gesetzt. Der Slave erinnert sich, dass er vorher adressiert war. Das zweite Byte entspricht dann nicht dem 2. Byte der Slaveadresse, sondern enthält bereits die Daten des Slaves. In Abbildung 14 und 15 sind die beiden Adressierungsvarianten zu sehen.

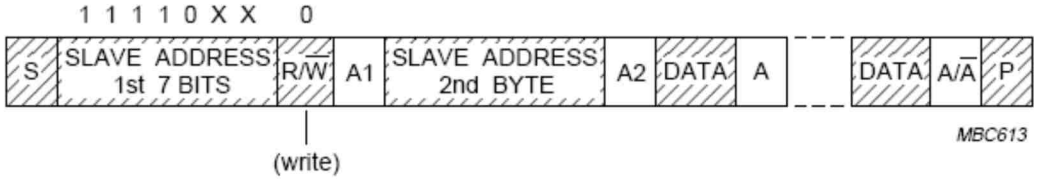


Abbildung 14: Datenübertragung von Master zu Slave [I^{2}C-Bus]

General Call Address: Der General Call Address kommt eine besondere Bedeutung zu. Sie spricht grundsätzlich alle Komponenten an, welche

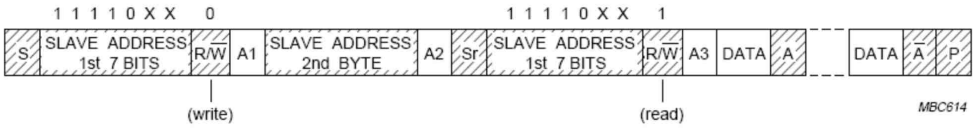


Abbildung 15: Datenempfang des Masters vom Slave [I^{2}C-Bus]

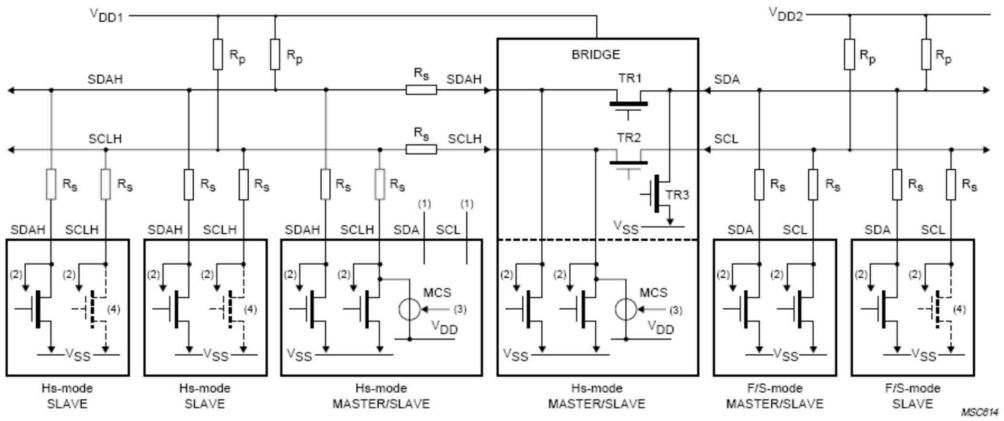
allerdings dafür ausgelegt sein müssen. Ist im nächsten gesendeten Byte das R/$\bar{\text{W}}$-Bit 0, dann setzen diese, dafür ausgelegten Komponenten, ihre individuelle Adresse, auf eine vorgegebene, individuelle Adresse. Ist das R/$\bar{\text{W}}$-Bit 1, dann handelt es sich bei dem Sender um einen Hardware Master. Diese Master können keine individuellen Slaves ansprechen, und übertragen im zweiten Byte ihre eigene Adresse, um sie bekannt zu machen. Diese Adresse wird von einer dafür vorgesehenen Komponente empfangen, welche dann die Datenübertragung von und zu dem Master regelt.

Fast-Mode und High-Speed Mode Kommunikation: In den beiden schnellen Übertragungsmodi sind, außer im Hinblick auf ihre höheren Übertragungsgeschwindigkeiten, alle Eigenschaften in Bezug auf Frame Formate und Protokollablauf gleich. Daher sind Komponenten, die diese Übertragungsarten untersützen, abwärtskompatibel mit älteren, langsameren Komponenten. Jedoch sollten einzelne ältere Komponenten, welche nicht aufwärtskompatibel sind, nicht in solchen Hochgeschwindigkeitsnetzwerken verwendet werden, da es bei diesen Komponenten zu unvorhersehbaren Zuständen kommen kann. Umgekehrt gilt diese Beschränkung nicht.

Fastmode: In diesem Modus können Module an der Kommunikation teilnehmen, welche eine Mindestvoraussetzung erfüllen. Diese besteht darin, dass sie mit einer Geschwindigkeit von 400kbit/s an der Synchronisationsphase teilnehmen können. Für die eigentliche Kommunikation können dann die Übertragungsgeschwindigkeiten reduziert werden, siehe dazu Abschnitt 2.3.1.

Hs-Mode: Module die diesen Modus unterstützen, besitzen neben der SDA und SCL Leitung noch zwei weitere, nämlich eine SDAH und einer SCLH Leitung, welche für die Hs-Mode Datentransfers genutzt werden. Die Synchronisation und Arbitrierung findet im Standard-Mode statt, also im langsamsten Übertragungsmodus. Während der Arbitrierungsphase sendet jeder Hs-Master seine einzigartige Adresse. Diese hat, wie aus Tabelle 2 zu

entnehmen ist, das Format „0000 0XXX“, wobei hier auch das R/$\bar{\text{W}}$-Bit zur Adresse zählt. Das „X“ beschreibt einen Platzhalter für die 3-Bit Adresse. Durch die Beschaffenheit der Adresse, ist es nur möglich 8 Master gleichzeitig zu betreiben. Nach der Übertragung wird kein Bestätigungsbit gesendet, weil die Adresse einzigartig ist, und keine 2 Komponenten dieselbe Adresse haben können. Dann wird in den Hs-Mode umgeschaltet. Dabei verwenden die Hs-kompatiblen Komponenten die vorher erwähnten SDAH und SCLH Leitungen. Diese sind durch ihre Beschaffenheit in der Lage Daten mit einer höheren Geschwindigkeit zu übertragen. Die Datenübertragung selbst funktioniert dann wie im Standardmode. Der Master adressiert einen Slave und sendet oder empfängt Daten. Danach kann er die Übertragung mit einem Stoppbit beenden oder mit einem neuerlich gesendeten Startbit weiter übertragen. Nach einem Stoppbit kehren die Hs-Module wieder in den Standardmode zurück.
Eine weitere Möglichkeit Standard-Mode oder Fast-Mode Module mit Hs-Modulen zu verbinden, besteht darin, die SDA und SCL Leitungen der langsameren Komponenten, mit den SDAH und SCLH Leitungen der schnelleren Module zu verbinden. Dafür wird jedoch eine Interconnection Bridge benötigt, welche bei einer Hochgeschwindigkeitsübertragung die langsamen Module abtrennt. Solch eine Konfiguration ist in Abbildung 16 zu sehen



(1) Bridge not used. SDA and SCL may have an alternative function.
(2) To input filter.
(3) Only the active master can enable its current-source pull-up circuit.
(4) Dotted transistors are optional open-drain outputs which can stretch the serial clock signal SCL or SCLH.

Abbildung 16: Konfiguration eines Hs-Komponenten und F/S-Kompenten Netzes [I^{2}C-Bus]

2.3.2 Verwendungsmöglichkeiten

Die Verwendungsmöglichkeiten sind mannigfaltig. Dieses Protokoll kann dazu verwendet werden, um unter Microcontrollern zu kommunizieren. Außerdem gibt es viele I^2C-kompatible Komponenten, wie Speicherbausteine, LCD-Treiber und Datenkonvertierungsschaltkreise. Zudem können damit Gerätekomponenten auch intern kommunizieren, beispielweise das Sendersuchmodul und Signalprozessoren in Radio- und Videosystemen. In Abbildung 17 sind zwei Beispiele für die mögliche Kommunikationsmodule innerhalb der Geräte zu sehen. Eine weitere Verwendungsmöglichkeit ist auch bei diversen Chipkarten zur Zeiterfassung gegeben, welche über das I^2C kommunizieren. Ein großer Vorteil, den das Protokoll auch bietet, ist die geringe Störanfälligkeit. Nähere Informationen zu diesem Protokoll sind in [I^2C-Bus] zu finden.

3 Funktionsweise Hardware-Software Timer

Timer oder Counter bietet jeder in dieser Arbeit verwendete Microcontroller an. Sie werden unter anderem dazu benötigt, um regelmäßig wiederkehrende Operationen auszuführen. Dies kann das Senden einer Nachtricht mit den vorher beschriebenen Protokollen sein, aber auch beispielsweise die Generierung eines pulsweitenmodulierten Signals, etwa zum Steueren der Geschwindigkeit eines Motors.

3.1 Funktionsweise eines Microcontroller Hardware-Timer

Ein Timer/Counter ist bei einem Microcontroller ein eigenes Modul. Er funktioniert auf Basis von Registern. Dabei wird in gewissen zeitlichen Abständen ein Zählregister, je nach Konfiguration, um eins inkrementiert oder dekrementiert. In Abbildung 18 ist ein Timer des ATMega16 zu sehen, um die Zusammenhänge zu verdeutlichen. Um einen Timer auszulösen, sind folgende Varianten möglich:

- Das Zählregister des Timers läuft über (bei einem 8-Bit Register tritt dieser Überlauf bei der Inkrementation von 255 um 1 auf).
- Es wird der Wert 0 im Zählregister erreicht (bei Dekrementation von 1 um 1).
- Es wird ein bestimmter, vorher definierter Wert durch Inkrementation oder Dekrementation erreicht.

Wird ein Timer ausgelöst, setzt die Timerlogik diverse Bits in auslesbaren Kontrollregistern. Diese Bits können dann

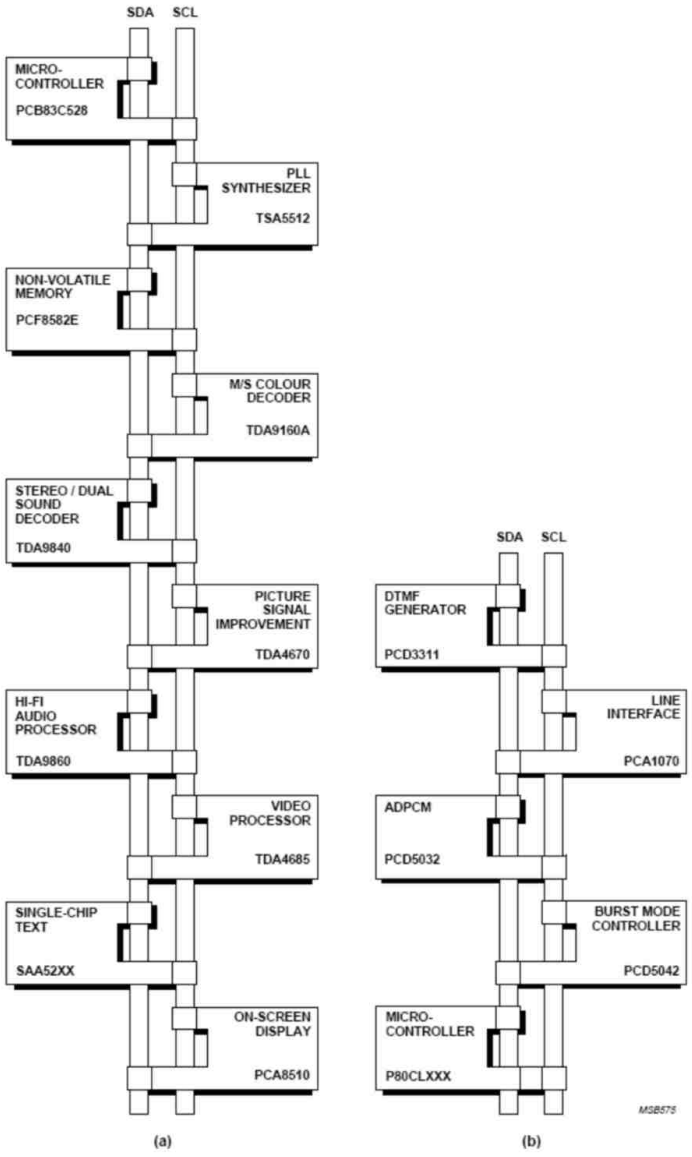


Abbildung 17: 2 Beispiele für I^{2}C Netze: (a) ein hochintegriertes TV-Gerät (b) eine schnurlose DECT Basisstation [I^{2}C-Bus]

- manuell ausgelesen werden,
- Interrupts generieren, welche daraufhin die momentane Arbeit der CPU

unterbrechen, um in eine ISR (**I**nterrupt **S**ervice **R**outine) zu springen, um dort Aktionen zu setzen,

- oder ein Signal auf einem Ausgangspin steuern.

Wann das Zählregister verändert wird, hängt damit zusammen, mit welchem Takt das Timermodul betrieben wird. Dies kann

- einerseits über das Taktsignal des Microcontrollers selbst erfolgen,
- andererseits über einen externen Taktgenerator.

Meistens kann man die Timermodule auch dahingehend konfigurieren, dass das Zählerregister nicht bei jedem eingehenden Takt verändert wird. Dazu wird ein Prescaler verwendet. Dieser teilt das Taktsignal und erreicht somit, dass mehrere Takte vergehen können, bevor das Zählregister verändert wird.

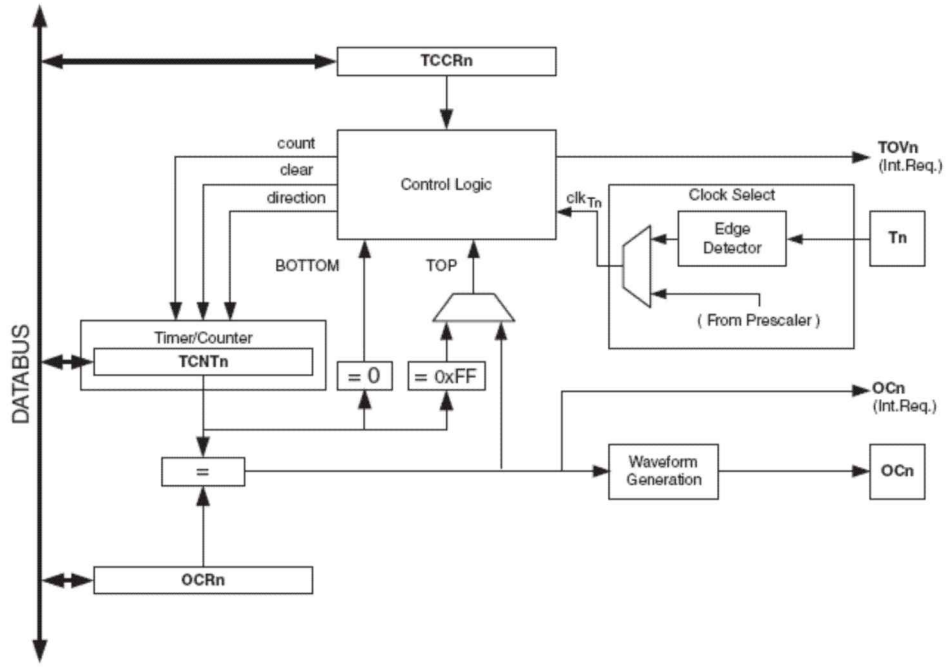


Abbildung 18: Funktionales Modul eines Hardware Timers [ATMega16]

3.2 Nutzen einer Timer API für Software-Timer

Ein Hardwaretimer ist oft hilfreich, wenn es darum geht, Zeiten zwischen Ereignissen zu messen, und Aufgaben in bestimmten Zeitperioden wiederholt auszuführen oder Ausgangssignale zu generieren. Ein großer Nachteil

dieser Timer ist allerdings, dass sie nur in begrenzter Anzahl in einem Microcontroller zu finden sind. Ein nachträgliches Hinzufügen von Timern ist nicht möglich. Für solche Zwecke wurde ein Timer-API (**A**pplication **P**rogramming **I**nterface) für Software Timer geschrieben.

Ein Software Timer basiert auf einem Hardware-Timer. Dieser Software-Timer hat den Vorteil, dass er ähnliche Möglichkeiten bietet wie Hardware-Timer, aber mehr oder weniger in unbegrenzter Menge (abhängig natürlich von dem auf dem Microcontroller vorhandenen Speicherplatz) zur Verfügung steht. Einziger Nachteil ist die Geschwindigkeit, mit der Software-Timer arbeiten. Sie werden während der Hardware-Timer ISR abgearbeitet. Für diese Abarbeitung werden unterschiedlichste Befehle benötigt, die auch mehrere Taktzyklen zur Ausführung benötigen. Daher ist es nicht möglich, Software-Timer zu programmieren, welche dieselbe relative Genauigkeit (bezogen auf die Periodendauer) bieten wie Hardware-Timer. Für viele Aufgaben ist diese Unschärfe allerdings ausreichend, da diese ohnehin in größeren Zeitintervallen ausgeführt werden.

3.3 Funktionsweise eines Microcontroller Software-Timer

Um einen Software-Timer zu verwenden, sollte es eine Registrierungsroutine geben. Um Ressourcen zu sparen, sollte erst bei der erstmaligen Verwendung eines Software-Timer der Hardware-Timer initialisiert werden. Ebenso sollte er deaktiviert werden, wenn kein Software-Timer mehr aktiv ist. Außerdem müssen in der Registrierungsfunktion konfigurierbar sein:

- Die Periode: Das ist der Stand des Zählregisters, von dem heruntergezählt wird. Ist das Zählregister bei 0 angelangt, wird der Timer Callback ausgeführt. Zusammen mit der Periode des verwendeten HW-Timer ergibt sich das Zeitintervall für den SW-Timer.

- Die Zykluszeit: Ein Timer kann zyklisch sein oder nicht. Ist er nicht zyklisch, wird er nach einmaliger Ausführung des Callbacks wieder deaktiviert, sonst nicht.

- Die Priorität: Je höher diese ist, desto früher wird der Timerinterrupt bearbeitet, wenn zur gleichen Zeit mehrere Timer auf die Bearbeitung warten.

Um einen zyklischen Timer deaktivieren zu können, benötigt man Information, um ihn eindeutig identifizieren zu können. Diese eindeutige ID wird von der Registrierungsfunktion zurückgegeben.

4 Beschreibung der API für die Kommunikationsprotokolle und den SW-Timer

4.1 UART API

Bevor der UART verwendet werden kann, muss er initialisiert werden. Dazu müssen im Vorfeld drei Interrupt-Handler deklariert werden. Das sind:

- *void (*uart_mini_tx_handler_t)(byte port);* Eine Routine die aufgerufen wird, wenn ein Sendevorgang erfolgreich abgeschlossen wurde.

  Parameter:

  - *byte port* gibt an, welcher UART den Sendevorgang abgeschlossen hat.

- *void (*uart_mini_rx_handler_t)(byte port, byte data);* Wird aufgerufen, falls ein Datenwort empfangen wurde.

  Parameter:

  - *byte port* gibt den UART an, der Daten empfangen hat.
  - *byte data* enthält die korrekt empfangenen Daten.

- *void (*uart_mini_rx_err_handler_t)(byte port, byte error_code);* Eine Empfangsfehlerroutine, welche bei falsch empfangenen Datenbytes aufgerufen wird. Falsch empfangen heißt in diesem Zusammenhang, dass generell ein Übertragungsfehler stattgefunden hat, sei es

  - ein Paritätsfehler,
  - eine falsche Übertragungsgeschwindigkeit,
  - eine falsche Anzahl an Stoppbits,
  - oder eine fehlerhafte Anzahl an Datenbits in einem Datenwort.

  Außerdem wird diese Routine aufgerufen, wenn kein Datenbyte empfangen werden sollte (Empfang wurde nicht mit *uart_rx* vorbereitet).

  Parameter:

  - *byte port* gibt den UART an, der einen Fehler meldet.
  - *byte error_code* zeigt, welcher Fehler aufgetreten ist.

Dann können folgende Funktionen verwendet werden:

- *uart_ init(byte spi, byte port, byte word_ size, byte stop_ bits, uint16_ t baudrate, byte parity, uart_ mini_ rx_ handler_ t rhandler, uart_ mini_ tx_ handler_ t thandler, uart_ mini_ rx_ err_ handler_ t ehandler)*; Initialisiert den UART.

  Parameter:

  - *byte spi*: Mit diesem Parameter wird angegeben, ob ein UART oder eine SPI-Schnittstelle initialisiert wird. Bei Verwendung des UART ist der Wert dieses Parameters immer „*SPI_ OFF*“.
  - *byte port*: Hier wird der UART angegeben, der initialisiert werden soll. Will man beispielsweise UART0 verwenden, ist dieser Parameter „*UART0*“[1].
  - *byte word_ size*: Um die Wortlänge zu wählen, kann dieser Wert mit „*DATA_ 7*“, für eine 7-bit langes Datenwort, oder „*DATA_ 8*“, für ein 8-bit langes Datenwort belegt werden.
  - *byte stop_ bits*: Hier wird die Anzahl der Stopp Bits angegeben. „*STOP_ 1*“ steht für ein Stopp Bit, „*STOP_ 2*“ für zwei.
  - *uint16_ t baudrate*: Für die Baudrate stehen einige vordefinierte Werte zur Verfügung. Diese müssen auf Sender und Empfänger gleich sein.
    - UART_BPS_1200: 1200 Bits pro Sekunde
    - UART_BPS_2400: 2400 Bits pro Sekunde
    - UART_BPS_4800: 4800 Bits pro Sekunde
    - UART_BPS_9600: 9600 Bits pro Sekunde
    - UART_BPS_19200: 19200 Bits pro Sekunde
  - *byte parity*: Hier wird die Parität angegeben. Diese kann die Werte
    - „*UART_ PARITY_ NONE*“ für kein Paritätsbit,
    - „*UART_ PARITY_ EVEN*“ für gerade Parität und
    - „*UART_ PARITY_ ODD*“ für ungerade Parität haben.
  - *uart_ mini_ rx_ handler_ t rhandler*: Hier wird der zuvor definierte Callback angegeben, der bei erfolgreichem Empfang eines Datenwortes aufgerufen wird.

[1] Dieser abstrakte Parameter wird von der Implementierung verwendet, um die korrekten Pins des jeweiligen UARTs zu verwenden

- *uart_mini_tx_handler_t thandler*: Es handelt sich hierbei um den Callback, der ausgeführt wird, wenn ein Datenwort erfolgreich gesendet wurde.
- *uart_mini_rx_err_handler_t ehandler*: Wurde ein Datenwort falsch empfangen, wird dieser zuvor definierte Callback aufgerufen.

Returnwerte:

- RET_SUCCESS: Routine erfolgreich ausgeführt.
- RET_UART_BAUD: Unzulässige Baudrate.
- RET_UART_DATA: Falsche Anzahl von Datenbytes.
- RET_UART_PARITY: Unzulässige Parität.
- RET_UART_UNKNOWN: Nicht vorhandene UART-Schnittstelle.
- RET_UART_STOP: Unzulässige Anzahl an Stoppbits.

- *byte uart_tx(byte port, byte data, BOOL blocking);* Sendet ein Datenwort.
  Parameter:
  - *byte port* gibt hier wiederum den UART an, über den gesendet werden soll.
  - *byte data* enthält das zu sendende Datenwort.
  - *BOOL blocking* gibt an, ob es sich bei der Transmission um eine blockierende oder nicht blockierende Funktion handelt. Blockierend heißt, dass das Programm mit der Weiterverarbeitung wartet, bis die Übertragung abgeschlossen ist. Bei der nicht blockierenden Übertragung läuft das Anwendungsprogramm weiter, während das Datenwort übertragen wird. Die Signalisierung, dass das Byte übertragen wurde, erfolgt im nicht blockierenden Fall über Handler.

  Returnwerte:
  - RET_SUCCESS: Routine erfolgreich ausgeführt.
  - RET_UART_TX_IN_PROGRESS: Es ist gerade eine Übertragung im Gange.
  - RET_UART_UNKNOWN: UART ist nicht vorhanden

- *byte uart_rx(byte port, byte *data);* Empfängt ein Datenwort. Durch den Aufruf dieser Funktion, wird der Empfang eines Datenwortes gestartet. Danach läuft das Anwendungsprogramm weiter, bis tatsächlich Daten empfangen werden. Nach dem Empfang, sind die Daten auf dem Speicherplatz vorhanden, auf welchen durch **data* verwiesen wird. Die Signalisierung für den Empfang erfolgt über Handler.

  Parameter:

  - *byte port* gibt den UART an, auf dem die Daten empfangen werden sollen.
  - *byte *data* ist ein Pointer, der auf die Adresse zeigt, in die die empfangenen Daten gespeichert werden sollen.

  Returnwerte:

  - RET_SUCCESS: Routine erfolgreich ausgeführt.
  - RET_UART_RX_IN_PROGRESS: Es ist gerade ein Empfang im Gange.
  - RET_UART_UNKNOWN: UART unbekannt

4.2 SPI API

Die Initialisierung der SPI-Schnittstelle erfolgt ähnlich zu der des UARTs. Es wird ebenfalls die Funktion *uart_init* verwendet. Es müssen zuvor ebenfalls die Interrupthandler deklariert werden. Dabei handelt es sich um dieselben Callbacks, wie bei der Verwendung des UARTs.

Das sind:

- *void (*uart_mini_tx_handler_t)(byte port);* Eine Routine die aufgerufen wird, wenn ein Sendevorgang erfolgreich abgeschlossen wurde.

  Parameter:

  - *byte port* gibt an, welche SPI-Schnittstelle den Sendevorgang abgeschlossen hat.

- *void (*uart_mini_rx_handler_t)(byte port, byte data);* Eine Empfangsroutine, die ebenfalls vom Typ void sein muss.

  Parameter:

  - *byte port* gibt die SPI-Schnittstelle an, die Daten empfangen hat.
  - *byte data* enthält die korrekt empfangenen Daten.

- *void (*uart_mini_rx_err_handler_t)(byte port, byte error_code);* Eine Empfangsfehlerroutine, welche bei falsch empfangenen Datenbytes aufgerufen wird. Falsch empfangen heißt in diesem Zusammenhang, dass generell ein Übertragungsfehler stattgefunden hat, sei es

  - ein Paritätsfehler,
  - eine falsche Übertragungsgeschwindigkeit,
  - oder eine fehlerhafte Anzahl an Datenbits in einem Datenwort.

  Außerdem wird diese Routine aufgerufen, wenn kein Datenbyte empfangen werden sollte (Empfang wurde nicht mit *uart_rx* vorbereitet).

  Parameter:

  - *byte port* gibt den UART an, der einen Fehler meldet.
  - *byte error_code* zeigt, welcher Fehler aufgetreten ist.

Danach können folgende Funktionen verwendet werden:

- *byte mc_is_spi(byte port, byte mode, BOOL answer, byte slave_select_pin);* Es müssen hier die Parameter für die Verwendung der SPI API bekanntgegeben werden. Diese Funktion muss vor *uart_init* aufgerufen werden.

  Parameter:

  - *byte port*: Hier erfolgt die Auswahl des „UARTs“, der verwendet werden soll. Nachdem nicht alle verwendeten Microcontroller ihre UARTs auch als SPI-Schnittstellen verwenden können, sei hier auf die Abschnitte zu den spezifischen Microcontrollerimplementierungen verwiesen. In diesen werden die möglichen Parameterwerte genannt.
  - *byte mode*: Es wird hier der SPI-Modus ausgewählt. Zu der Information, welcher Parameter verwendet werden kann, sei ebenfalls auf die Microcontrollerspezifischen Teile verwiesen.
  - *BOOL answer*: Dieser Parameter legt fest, ob der Microcontroller als Slave oder als Master verwendet werden soll. Ist answer *TRUE*, agiert der Microcontroller als Master, sonst als Slave.

– *byte slave_select_pin*: Hier wird der Pin ausgewählt, der im 4-Pin Mode verwendet werden soll, um als Slave-Select Leitung zu fungieren. Die wählbaren Pins sind im Microcontroller-spezifischen Teil genannt.

Returnwerte:

– RET_SUCCESS: Routine erfolgreich ausgeführt.
– RET_SS_PIN_NOT_USABLE: Es wurde ein nicht verwendbarer Pin als Slave-Select Pin ausgewählt.
– RET_UART_UNKNOWN: Ausgewählter UART nicht vorhanden.

- *uart_init(byte spi, byte port, byte word_size, byte stop_bits, uint16_t baudrate, byte parity, uart_mini_rx_handler_t rhandler, uart_mini_tx_handler_t thandler, uart_mini_rx_err_handler_t ehandler)*; Diese Funktion verhält sich ähnlich wie bei der Initialisierung eines UART's. In Bezug auf die Parameter gibt es jedoch einige Unterschiede.

  Parameter:

  – *byte spi*: Dieser Parameter wird verwendet, um die SPI-Funktionalität des Microcontrollers zu aktivieren. Zwei Modi sind möglich, „*SPI_3*“ für den 3-Pin Mode, und „*SPI_4*“ für den 4-Pin Mode.
  – *byte port*: Hier erfolgt die Auswahl des „UARTs“, der verwendet werden soll. Nachdem nicht alle verwendeten Microcontroller ihre UARTs auch als SPI-Schnittstellen verwenden können, sei hier auf die Abschnitte zu den spezifischen Microcontrollerimplementierungen verwiesen. In diesen werden die möglichen Parameterwerte genannnt.
  – *byte word_size*: Da es bei der hier implementierten SPI-Art nur 8-Bit Worte gibt, wird der hier eingegebene Wert einfach ignoriert.
  – *byte stop_bits*: Bei diesem Wert verhält es sich genauso, wie mit der *word_size*. Er wird ignoriert, weil durch die Synchronisation keine Stoppbits benötigt werden.
  – *uint16_t baudrate*: Hier wird die Baudrate angegeben. Welche Werte angegeben werden können, sind dem implementierungsspezifischen Teil zu entnehmen.
  – *byte parity*: Wird ebenfalls ignoriert.

- *uart_mini_rx_handler_t rhandler*: Hier wird der zuvor definierte Callback zum Empfang eines Datenwortes angegeben.
- *uart_mini_tx_handler_t thandler*: Es handelt sich hierbei um einen Callback, der ausgeführt wird, wenn ein Datenwort erfolgreich gesendet wurde.
- *uart_mini_rx_err_handler_t ehandler*: Wurde ein Datenwort falsch empfangen, wird dieser zuvor definierte Callback aufgerufen.

Returnwerte:

- RET_SUCCESS: Routine erfolgreich ausgeführt.
- RET_UART_BAUD: Es wurde eine unerlaubte Baudrate verwendet.
- WRONG_SPI_MODE: Der angegebene Modus in Bezug auf Phase und Polarität des Taktsignals existiert nicht.
- SPI_PIN_MODE_UNKNOWN: Es wurde ein anderer Wert als „*SPI_3*“ und „*SPI_4*“ für den Funktionsparameter *spi* gewählt.
- RET_UART_UNKNOWN: Angegebene SPI-Schnittstelle existiert nicht.

- *byte uart_tx(byte port, byte data, BOOL blocking);* Sendet ein Datenwort.

Parameter:

- *byte port* gibt hier wiederum das SPI-Inteface an, über das gesendet werden soll.
- *byte data* enthält das zu sendende Datenwort.
- *BOOL blocking* gibt an, ob es sich bei der Transmission um eine blockierende oder nicht blockierende Funktion handelt. Blockierend heißt, dass das Programm mit der Weiterverarbeitung wartet, bis die Übertragung abgeschlossen ist. Bei der nicht blockierenden Übertragung läuft das Anwendungsprogramm weiter, während das Datenwort übertragen wird. Die Signalisierung, dass das Byte übertragen wurde, erfolgt im nicht blockierenden Fall über Handler.

Returnwerte:

- RET_SUCCESS: Routine erfolgreich ausgeführt.

  - RET_UART_TX_IN_PROGRESS: Es ist gerade eine Übertragung im Gange.
  - RET_UART_UNKNOWN: SPI-Schnittstelle ist nicht vorhanden.

- *byte uart_rx(byte port, byte *data);* Empfängt ein Datenwort. Durch den Aufruf dieser Funktion, wird der Empfang eines Datenwortes gestartet. Danach läuft das Anwendungsprogramm weiter, bis tatsächlich Daten empfangen werden. Nach dem Empfang, sind die Daten auf dem Speicherplatz vorhanden, auf welchen durch **data* verwiesen wird. Die Signalisierung für den Empfang erfolgt über Handler.

  Parameter:

  - *byte port* gibt das SPI-Interface an, auf dem die Daten empfangen werden sollen.
  - *byte *data* ist ein Pointer, der auf die Adresse zeigt, in die die empfangenen Daten gespeichert werden sollen.

  Returnwerte:

  - RET_SUCCESS: Routine erfolgreich ausgeführt.
  - RET_UART_RX_IN_PROGRESS: Es ist gerade ein Empfang im Gange.
  - RET_UART_UNKNOWN: Angegebene SPI-Schnittstelle existiert nicht.

- *byte uart_spi_enable(byte port)* wird dazu verwendet die SS-Leitung für den auszuwählenden Slave auf logisch „0“ zu setzen und damit dem Empfänger mitzuteilen, dass Daten übertragen werden sollen.

  Parameter:

  - *byte port* gibt an, mit welcher SPI-Schnittstelle gesendet werden soll.

  Returnwerte:

  - RET_SUCCESS: Routine erfolgreich ausgeführt.
  - RET_UART_NOT_SPI: Der angegebene Port ist keine SPI-Schnittstelle.
  - RET_UART_UNKNOWN: Angegebene SPI-Schnittstelle existiert nicht.

- *byte uart_spi_disable(byte port)* wird dazu verwendet die SS-Leitung für den auszuwählenden Slave auf logisch „1“ zu setzen und damit dem Empfänger mitzuteilen, dass die Datenübertragung abgeschlossen ist.

  Parameter:

  - *byte port* gibt an, mit welcher SPI-Schnittstelle gesendet werden soll.

  Returnwerte:

  - RET_SUCCESS: Routine erfolgreich ausgeführt.
  - RET_UART_NOT_SPI: Der Angegebene Port ist keine SPI-Schnittstelle.
  - RET_UART_UNKNOWN: Angegebene SPI-Schnittstelle existiert nicht.

4.3 Timer API

Für die Verwendung eines Software Timers muss vor der Initialisierung sichergestellt sein, dass ein Software Timer-Callback deklariert wurde.

- void (*timer_handler_t)(byte id); wird aufgerufen, wenn ein Software Timer abgelaufen ist.

  Parameter:

  - *byte id* enthält die Timer Identifikationsnummer zur Unterscheidung, falls mehrere Timer denselben Callback verwenden.

Dann können folgende Funktionen verwendet werden.

- *byte register_timer(word period, BOOL cyclic, byte priority, timer_handler_t thandler);* Initialisiert den Timer, bei dem ein Takt 500μs dauert. Bei jedem neuen Timer, der registriert wird, werden alle nun aktiven Timer der Priorität nach sortiert. Diese Sortierung wird von einem Bubble-Sort Algorithmus übernommen.
  Um zu gewährleisten, dass alle abgelaufenen Timer ausgeführt und, falls es sich um nicht zyklischen Timer handelt, deaktiviert werden, ist eine Statemachine implementiert worden. Diese beinhaltet 3 Zustände *Idle*, *Executing* und *ToDelete*. In Abbildung 19 wird eine derartige Statemachine dargestellt. Während der Timer zählt, befindet er sich im Zustand Idle. Wird er ausgeführt, wechselt er in den Zustand

Executing, wobei der Callback ausgeführt wird. Handelt es sich um einen nicht zyklischen Timer, so wird nach dem Executing State in den ToDelete State gewechselt. Natürlich kann auch ein zyklischer Timer, wenn er nicht mehr verwendet wird, gelöscht werden. Unter der Annahme, die Hardware-Timer ISR ist nicht preemptiv, sollte die Entfernung des Timers allerdings erst beim nächsten Aufruf der Hardware-Timer ISR stattfinden, da es sonst zu Problemen mit dem Aufruf der anderen Software-Timer Callbacks kommen könnte. Das Problem soll durch das folgende Szenario veranschaulicht werden:
Die Informationen für die Software-Timer ist in einem Array gespeichert (unter anderem auch ein Pointer, welcher auf die Position des Callbacks zeigt). Es soll nun ein Timer entfernt werden, dessen Informationen am Anfang des Arrays gespeichert sind. Durch die Entfernung eines Timers werden die verbliebenen Timer neu geordnet. Wird der Timer zum falschen Zeitpunkt entfernt, kann es vorkommen, dass ein Callback-Pointer eines anderen Timers nicht mehr auf die richtige Position zeigt.
Durch eine Entfernung eines Software-Timers erst beim nächsten Aufruf der Hardware-Timer ISR, kann dieser undefinierte Zustand vermieden werden.

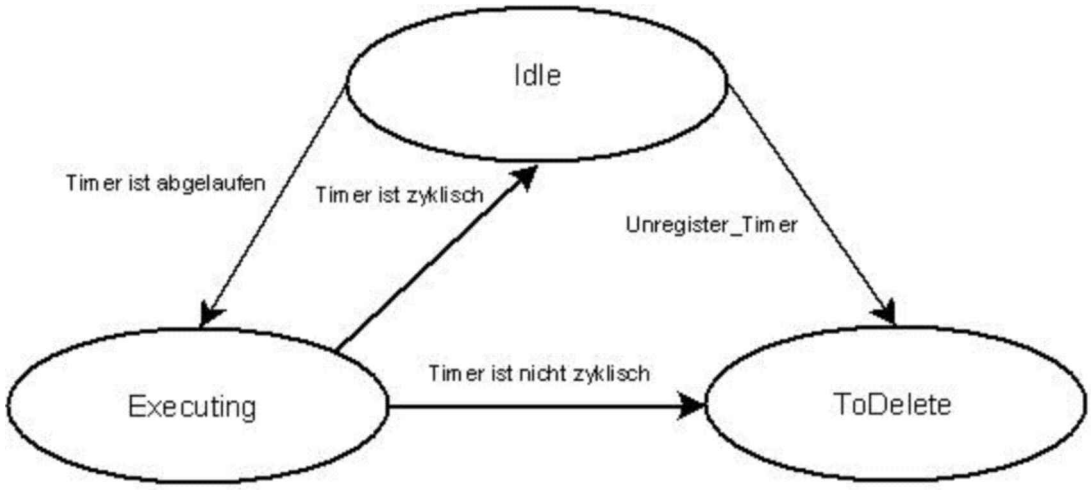


Abbildung 19: Zustandsmodell eines Software Timers

Parameter:

– *period*: Es wird hier die Anzahl der Takte angegeben, die vergehen

müssen, bevor der Software-Timer-Callback aufgerufen wird. Das Maximum von *period* ist 65535. 1 entspricht hier 500μs.

- *cyclic*: Dieser Parameter gibt an, ob ein Timer nur einmal gestartet wird oder nach jedem Aufruf des Callbacks erneut. Es sind nur die Werte *TRUE* und *FALSE* erlaubt.
- *priority*: Je niedriger der Wert dieses Parameters ist, desto höher ist die Priorität des Timers. Der Wert 0 ist die höchste Priorität. Der höchste verwendbare Wert ist 127.
- *thandler*: Hier wird der zuvor definierte Callback angegeben.

Returnwerte:

- Timeridentifikationsnummer: Diese Nummer identifiziert den Timer eindeutig. Sie wird nur übertragen, wenn kein Fehler aufgetreten ist.
- TIMER_UNAVAILABLE: Es wurden zu viele zyklische Timer registriert. Es können maximal 8 Timer gleichzeitig verarbeitet werden.
- PERIOD_NULL_IMPOSSIBLE: Es ist nicht möglich einen Timer zu registrieren, der die Periode „0“ hat.
- PRIORITY_VALUE_TOO_HIGH: Es wurde ein höherer Wert als 127 für die *priority* angegeben.
- TIMERHANDLER_NOT_SPECIFIED: Es wurde kein Timer-Callback angegeben.

- *byte register_fast_timer(word period, BOOL cyclic, byte priority, timer_handler_t thandler);* Initialisiert den Timer, bei dem ein Takt 100μs dauert.

  Parameter:

  - *period*: Es wird hier die Anzahl der Takte angegeben, die vergehen müssen, bevor der Software-Timer-Callback aufgerufen wird. Das Maximum von *period* ist 65535. 1 entspricht hier 100μs.
  - *cyclic*: Dieser Parameter gibt an, ob ein Timer nur einmal gestartet wird oder nach jedem Aufruf des Callbacks erneut. Es sind nur die Werte *TRUE* und *FALSE* erlaubt.
  - *priority*: Je niedriger der Wert dieses Parameters ist, desto höher ist die Priorität des Timers. Der Wert „0“ ist die höchste Priorität. Der höchste verwendbare Wert ist „127“.

  - *thandler*: Hier wird der zuvor definierte Callback angegeben.

  Returnwerte:

  - Timeridentifikationsnummer: Diese Nummer identifiziert den Timer eindeutig. Sie wird nur übertragen, wenn kein Fehler aufgetreten ist.
  - TIMER_UNAVAILABLE: Es wurden zu viele zyklische Timer registriert. Es können maximal 8 Timer gleichzeitig verarbeitet werden.
  - PERIOD_NULL_IMPOSSIBLE: Es ist nicht möglich einen Timer zu registrieren, der die Periode „0“ hat.
  - PRIORITY_VALUE_TOO_HIGH: Es wurde ein höherer Wert als 127 für die *priority* angegeben.
  - TIMERHANDLER_NOT_SPECIFIED: Es wurde kein Timer-Callback angegeben.

- *void unregister_timer(byte id);* entfernt einen zyklischen 500μs-Timer.

  Parameter:

  - *byte id*: Dies ist die ID des Timers, mit deren Hilfe der zu entfernende Timer identifiziert wird.

  Returnwerte:

  - RET_SUCCESS: Routine erfolgreich ausgeführt.
  - TIMER_UNKNOWN: Es wurde eine falsche Identifikationsnummer angegeben.
  - TIMER_IS_NOT_CYCLIC: Da ein nichtzyklischer Timer automatisch nach einem Durchlauf deregistriert wird, kann er nicht mit Hilfe der Funktion *unregister_timer* entfernt werden.

- *void unregister_fast_timer(byte id);* entfernt einen zyklischen 100μs-Timer.

  Parameter:

  - *byte id*: Dies ist die ID des Timers, mit deren Hilfe der zu entfernende Timer identifiziert wird.

Returnwerte:

- RET_SUCCESS: Routine erfolgreich ausgeführt.
- TIMER_UNKNOWN: Es wurde eine falsche Identifikationsnummer angegeben.
- TIMER_IS_NOT_CYCLIC: Da ein nichtzyklischer Timer automatisch nach einem Durchlauf deregistriert wird, kann er nicht mit Hilfe der Funktion *unregister_fast_timer* entfernt werden.

5 Implementierung

Dieses Kapitel beschreibt, wie die in der Aufgabenstellung besprochenen Kommunikationsprotokolle und der SW-Timer auf den unterschiedlichen Microcontrollern implementiert wurden. Eine I²C-API wurde nicht implementiert, daher enthält diese Implementierungsübersicht auch keine Informationen darüber.

5.1 Verwendete Microcontroller

5.1.1 ATMEL ATMega16

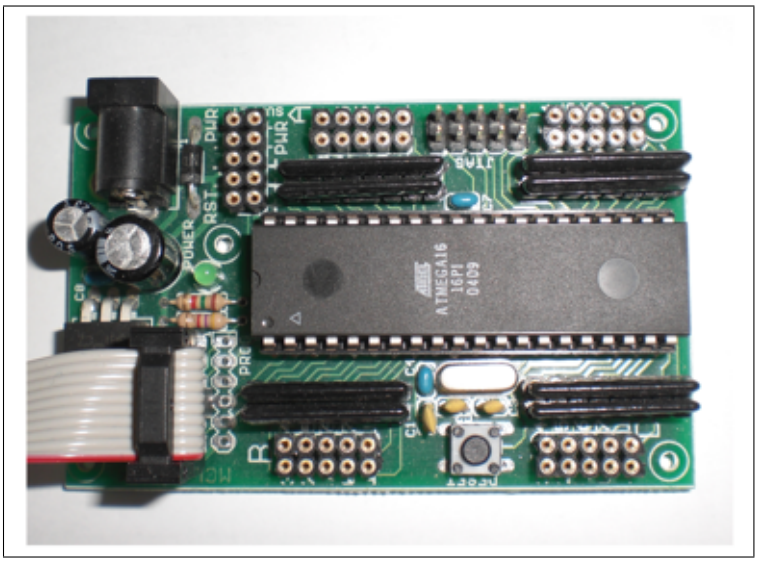


Abbildung 20: ATMEL ATMega16

Der ATMega16 Microcontroller (Abbildung 20) der Atmel Corporation, ist ein energiesparender 8-Bit Microcontroller. Die ALU (**A**rithmetic

Logic **U**nit) basiert auf einer RISC-Architektur (**R**educed **I**nstruction **S**et **C**omputing). Der ATMega16 verfügt außerdem über eine Harvard-Architektur. Das bedeutet, der Microcontroller verwendet für den Programmcode und die Daten getrennte Speicher. Für den Programmcode wird ein 16kB großer reprogrammierbarer Flashspeicher eingesetzt. Zur Datenspeicherung verwendet der Microcontroller 1kB SRAM und 512 Byte EEPROM. Getaktet ist dieser Microcontroller, der in dieser Arbeit verwendet wird, mit einem externen Quarz, der mit 16MHz schwingt.

Der ATMega16 verfügt über folgende Kommunikationsschnittstellen:

- 1 UART (**U**niversal **S**ynchronous and **A**synchronous serial **R**eceiver and **T**ransmitter)
- 1 SPI (**S**erial **P**eripheral **I**nterface)
- 1 TWI [1] (**T**wo-**W**ire Serial **I**nterface)

Diese Schnittstellen können unabhängig voneinander verwendet werden. Weiterführende Informationen sind unter [ATMega16] zu finden.

5.1.2 TI MSP430

Die MSP430 Microcontrollerfamilie (Abbildung 21) der Firma Texas Instruments verfügt, wie der ATMega16, über eine CPU (**C**entral **P**rocessing **U**nit), basierend auf einer RISC-Architektur. Allerdings handelt es sich bei diesen Microcontrollern um einen 16-bit Microcontroller mit einer „Von-Neumann"-Speicherarchitektur. Diese Architektur zeichnet sich dadurch aus, dass sie für Programme und Daten einen gemeinsamen Adressbereich verwendet. Innerhalb dieser Familie sind verschiedene Microcontroller mit unterschiedlichen Konfigurationen zu finden. Der hier verwendete MSP430F149 verfügt über 60kB Programmspeicher, sowie 2kB an RAM (**R**andom **A**ccess **M**emory). Dieser Microcontroller ist extern mit 8MHz getaktet.

Der MSP430F149 verfügt über folgende, für diese Arbeit relevanten Kommunikationsschnittstellen:

- 2 UART-Schnittstellen
- 2 SPIs
- 1 TWI

[1] TWI ist ein anderer Name für das I^2C-Kommunikationsprotokoll

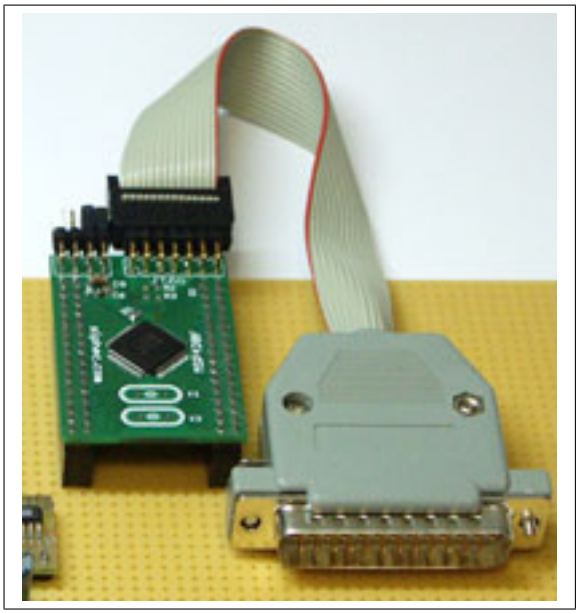


Abbildung 21: MSP430F149 [MSP430-HP]

Diese Schnittstellen können nicht unabhängig voneinander verwendet werden, weil sich je ein UART und ein SPI die Pins am Gehäuse teilen. Als Informationsgrundlage dient hier [MSP430].

5.1.3 KNXcalibur

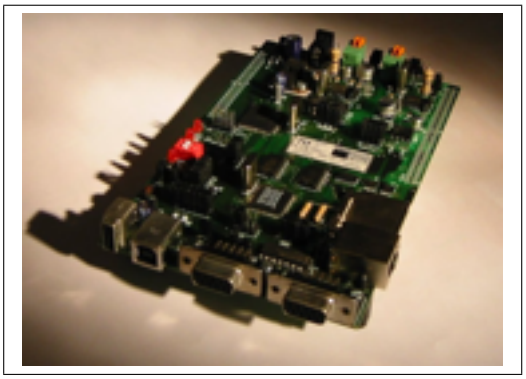


Abbildung 22: KNXcalibur [PRA05]

KNXcalibur (Abbildung 22) ist eine Plattform, welche am Institut für Rechnergestützte Automation an der TU Wien entwickelt wurde [PRA05]. Herzstück dieser Plattform ist ein MB90F334 Microcontroller der Firma Fu-

jitsu Siemens. Dieser Microcontroller hat eine 16-bit CPU, welcher eine eigens von Fujitsu Siemens entwickelte Architektur namens F2MC-16 CPU-CORE ARCHITECTURE zugrunde liegt. An Speicher bietet der MB90F334 384kB Programmspeicher und 24kB RAM. KNXcalibur arbeitet mit einer Taktung von 24MHz.

KNXcalibur verfügt über folgende Schnittstellen:

- 4 UART-Schnittstellen
- 4 SPIs
- und 3 I^2C-Schnittstellen (**I**nter-**I**ntegrated **C**ircuit)

Je ein SPI und ein UART verwenden dieselben Pins, die I^2Cs haben eigenständige Pins, können also unabhängig von den anderen Schnittstellen verwendet werden.

5.2 Microcontroller-abhängigie Implementierungsübersicht

Diese Übersicht beschreibt die Funktionen, welche für die Benutzung der Kommunikations- und Timer-API zur Verfügung stehen, und wie diese zu verwenden sind. Diese sind für alle verwendeten Microcontroller gleich.

5.2.1 ATMEL ATMega16 Implementierungsvariante

Bei der Implementierung des ATMega16 werden folgende, in Tabelle 3 gezeigten Pins verwendet:

Pinnummer	Name	Bedeutung
05	SS	SlaveSelect Leitung für SPI
06	MOSI	MasterOut SlaveIn für Datentransfer vom Master zum Slave
07	MISO	MasterIn SlaveOut für Datentransfer vom Slave zum Master
08	SCK	Taktsignal für SPI
14	RXD	Empfangsleitung des UARTs
15	TXD	Sendeleitung des UARTS

Tabelle 3: Verwendete Pins des ATMega16

Für die Verwendung der UART- und SPI-Schnittstelle existieren für den Parameter *port* in der Initialisierungsfunktion *uart_init* die Werte

- UART0 für den UART und
- UART1 für das SPI.

Bei Verwendung der SPI-Schnittstelle muss ein Pin angegeben werden, der als Slave-Select Leitung dienen soll. Es kann jedoch nicht jeder Pin des ATMega16 dafür verwendet werden. Wird beispielsweise der UART verwendet, so dürfen die Pins 14 und 15 nicht verwendet werden. Dies gilt natürlich auch für die Pins, die von der SPI-Schnittstelle genutzt werden. Folgende, in Tabelle 4 beschriebenen Pins dürfen ebenfalls nicht verwendet werden.

Pinnummer	Name	Bedeutung
09	RESET	Startet das Anwendungsprogramm neu
10	VCC	Stromversorgung
11	GND	Masse
12	XTAL2	Ausgang des internen Oszillators
13	XTAL1	Eingang für einen externen Oszillator
30	AVCC	Stromversorgung für A/D-Converter
31	GND	Masse
32	AREF	Referenzspannung für A/D-Converter

Tabelle 4: Nicht verwendbare Pins des ATMega16

Weiters hat der ATMega16 noch einige Besonderheiten, die bei Verwendung der SPI-Schnittstelle beachtet werden müssen. Wird das SPI im 3-Pin Slave Mode verwendet, so muss die Slave-Select Leitung manuell auf Masse gesetzt werden, da keine in Hardware ausgeführte Lösung angeboten wird. Außerdem gibt es, wie im vorherigen Abschnitt erwähnt, spezielle Baudratenwerte für die SPI-Schnittstelle:

- UART_BPS_4000000: 4000 kBits pro Sekunde
- UART_BPS_2000000: 2000 kBits pro Sekunde
- UART_BPS_1000000: 1000 kBits pro Sekunde
- UART_BPS_500000: 500 kBits pro Sekunde
- UART_BPS_250000: 250 kBits pro Sekunde
- UART_BPS_125000: 125 kBits pro Sekunde

Wird der Slavebetrieb genutzt, so sollte das Taktsignal in einer Art gewählt werden, dass eine High Periode zumindest 2 Taktzyklen und eine Low Periode ebenfalls zumindest 2 Taktzyklen des ATMega16 dauert. Andernfalls funktioniert die Abtastung des Taktsignals nicht richtig.

5.2.2 Texas Instruments MSP430F149 Implementierungsvariante

Bei der Verwendung der Kommunikationsbibliothek für den MSP430, werden die Pins aus Tabelle 5 verwendet.

Pinnummer	Name	Bedeutung
28	STE0	SlaveSelect Leitung für SPI von UART0
29	SIMO0	MasterOut SlaveIn für Datentransfer vom Master zum Slave (SPI von UART0)
30	SOMI0	MasterIn SlaveOut für Datentransfer vom Slave zum Master (SPI von UART0)
31	UCLK0	Taktsignal für SPI von UART0
32	TXD0	Sendeleitung des UART0
33	RXD0	Empfangsleitung des UART0
34	TXD1	Sendeleitung des UART1
35	RXD1	Empfangsleitung des UART1
44	STE1	SlaveSelect Leitung für SPI von UART1
45	SIMO1	MasterOut SlaveIn für Datentransfer vom Master zum Slave (SPI von UART1)
46	SOMI1	MasterIn SlaveOut für Datentransfer vom Slave zum Master (SPI von UART1)
47	UCLK1	Taktsignal für SPI von UART1

Tabelle 5: Verwendete Pins des MSP430F149

Wird das SPI des MSP430 genutzt, so muss man sich im Vorhinein entscheiden, welcher UART als SPI-Schnittstelle verwendet wird. Dies geschieht mit Hilfe der *uart_ init*-Funktion. Ein Kommunikationsmodul kann entweder als UART oder SPI verwendet werden, beide Funktionen sind für ein Modul nicht möglich. Außerdem gibt es, wie beim ATMega16, einige Pins, die nicht für die Slave-Select Leitung der SPI-Schnittstelle verwendet werden dürfen. Bei Verwendung der SPI-Funktionalität von UART0 und UART1, dürfen die TXD- und RXD-Leitungen des verwendeten UARTs nicht verwendet werden. Ebenso dürfen, falls beispielsweise UART0 als SPI, und UART1 zur UART-Kommunikation verwendet werden, die Pins von UART1 nicht verwendet

werden. In Tabelle 6 werden noch andere Pins genannt die nicht benutzt werden dürfen.

Pinnummer	Name	Bedeutung
01	DV_{CC}	Spannungsversorgung
07	V_{REF+}	Positive Referenzspannung für A/D-Converter
08	XIN	Eingang für externen Oszillator
09	XOUT	Ausgang von externem Oszillator
10	Ve_{REF+}	Eingang für positive Referenzspannung für A/D-Converter
11	V_{REF-}/Ve_{REF-}	Negative Referenzspannung für A/D-Converter
52	XT2OUT	Ausgang von externem Oszillator
53	XT2IN	Eingang für externen Oszillator
54	TDO/TDI	Test Signal Ausgang oder Programmdaten Eingang
55	TDI/TCLK	Test Signal Eingang oder Test Takt Eingang
56,57	TMS,TCK	Für die Programmierung notwendig
58	RST	Startet das Anwendungsprogramm neu
62	AV_{SS}	Masse für A/D-Converter
63	DV_{SS}	Masse
64	AV_{CC}	Spannungsversorgung für A/D-Converter

Tabelle 6: Nicht verwendbare Pins des MSP430F149

Die Baudraten der SPI-Schnittstelle entsprechen denen der im allgemeinen Implementierungsteil genannten UART-Baudraten. Im Slavebetrieb des MSP430 ist außerdem zu beachten, dass der Taktgenerator des Senders maximal mit der halben Frequenz des Empfängers senden darf.
Eine weitere Besonderheit des MSP430 ist, dass die Taktrate mit der die Peripheriemodule des Microcontrollers arbeiten, händisch per Software eingestellt werden muss. Die anderen beiden Microcontroller sind einfacher handzuhaben, da diese Einstellung entfällt.

5.2.3 KNXcalibur Implementierungsvariante

Bei der Verwendung von KNXcalibur, werden die in Tabelle 7 beschriebenen Pins verwendet.

Wird das SPI des KNXcalibur genutzt, gelten dieselben Einschränkungen wie für den MSP430. Auch bei KNXcalibur dürfen einige Pins nicht für die Slave-Select Leitung verwendet werden (siehe Tabelle 8). Bei Verwendung der SPI-Funktionalität von UART0 bis UART3 dürfen die TXD- und RXD-Leitungen des verwendeten UARTs nicht verwendet werden. Ebenso dürfen,

Pinnummer	Name	Bedeutung
11	SIN0	Empfangsleitung von SPI0 und UART0
12	SOT0	Sendeleitung von SPI0 und UART0
17	SCK0	Taktleitung von SPI0
18	SIN1	Empfangsleitung von SPI1 und UART1
19	SOT1	Sendeleitung von SPI1 und UART1
20	SCK1	Taktleitung von SPI1
29	SIN2	Empfangsleitung von SPI2 und UART2
30	SOT2	Sendeleitung von SPI2 und UART2
31	SCK2	Taktleitung von SPI2
32	SIN3	Empfangsleitung von SPI3 und UART3
33	SOT3	Sendeleitung von SPI3 und UART3
34	SCK3	Taktleitung von SPI3

Tabelle 7: Verwendete Pins des KNXcalibur

falls beispielsweise UART0 als SPI, und UART1 zur UART-Kommunikation verwendet werden, die Pins von UART1 nicht verwendet werden.

Das KNXcalibur besitzt standardmäßig keine SPI-Schnittstelle. Für die Verwendung werden daher die synchrone Übertragungsfunktionalität der UARTs zweckentfremdet. Aufgrund dessen gibt es einige Einschränkungen bei den Funktionen der SPI-Schnittstelle. Es werden bei dieser Implementierung nur 2 der möglichen 4 Modi, welche in dem Abschnitt über das SPI-Protokoll (siehe Tabelle 1) zu finden sind, angeboten, nämlich Modus 0 und 3. Außerdem funktioniert KNXcalibur im Slavemodus nur im 3-Pin Betrieb, da keine Slave-Select Pins für die einzelnen SPI-Schnittstellen angeboten werden. Informationen dazu sind in [KNXcalAPP] zu finden.

Die Baudraten des SPI entsprechen denen, die auch für die UARTs zur Verfügung stehen.

5.3 Beschreibung des Sourcecodes

5.3.1 Ordnerstruktur

Die Ordnerstruktur sieht folgendermaßen aus:

Pinnummer	Name	Bedeutung
13,14	X0A, X1A	32KHz Oszillator Ausgang
15	V_{CC}	Spannungsversorgung
16	V_{SS}	Masse
36	AV_{CC}	Spannungsversorgung für A/D-Converter
37	AVRH	Referenzspannungseingang für A/D-Converter
38	AV_{SS}	Masse für A/D-Converter
47	V_{SS}	Masse
71	UTEST	USB Test Pin
72	V_{SS}	Masse
73	DVM	D-Pin für USB
74	DVP	D+Pin für USB
75	V_{CC}	Spannungsversorgung
76	V_{SS}	Masse
77	HVM	D-Pin für USB Mini-HOST
78	HVP	D+Pin für USB Mini-HOST
79	V_{CC}	Spannungsversorgung
80	HCON	Anschluss für Pull-Up Widerstand
87-89	MD0-MD2	Operationsmode Select Pin
90	RST	RESET Pin
105	V_{CC}	Spannungsversorgung
106	V_{SS}	Masse
107,108	X0, X1	Anschlüsse für externen Oszillator

Tabelle 8: Nicht verwendbare Pins des KNXcalibur

```
  /UART_SPI_TIMER_API/
|
|--- /common/
|     |--- config/ (Headerdatei für allgemeine Parameter)
|     |--- h/ (Microcontroller-Unabhängige API Header-Dateien)
|--- /AVR/
|     |--- config/ (Headerdatei für spez. AVR-Parameter)
|     |--- h/ (Implementierungsspezifische AVR-Headerdateien)
|     |---src/ (Implementierungsspezifische AVR-Sourcedateien)
|     |---demo/ (Beispielprogramm für AVR)
|--- /MSP430/
|     |--- config/ (Headerdatei für spez. MSP-Parameter)
|     |--- h/ (Implementierungsspezifische MSP-Headerdateien)
```

```
|     |---src/ (Implementierungsspezifische MSP-Sourcedateien)
|     |---demo/ (Beispielprogramm für MSP)
|--- /KNXcalibur/
|     |--- config/ (Headerdatei für spez. KNXcalibur-Parameter)
|     |--- h/ (Implementierungsspezifische KNXcalibur-Headerdateien)
|     |---src/ (Implementierungsspezifische KNXcalibur-Sourcedateien)
|     |---demo/ (Beispielprogramm für KNXcalibur)
```

5.3.2 Config Dateien

/UART_SPI_TIMER_API/common/config/common_config.h:
Es werden hier allgemeine Parameter für alle verwendeten Microcontroller, wie beispielsweise Bitoperationen beschrieben, und die Konfigurationsdateien der verwendeten Microcontroller eingebunden.

/UART_SPI_TIMER_API/AVR/config/avr_config.h:
Enthält Konstanten für die Verwendung des ATMega16.

/UART_SPI_TIMER_API/MSP430/config/msp430_config.h:
Hier werden die Konstanten für die Verwendung des MSP430 beschrieben.

/UART_SPI_TIMER_API/KNXcalibur/config/mb90330_config.h:
Enthält die für die Verwendung des KNXcalibur benötigten Konstanten.

5.3.3 API Headerfiles

/UART_SPI_TIMER_API/common/h/uart.h:
Beinhaltet die für die Verwendung von UART- und SPI-Schnittstellen notwendigen Konstanten und Funktionsdeklarationen.

/UART_SPI_TIMER_API/common/h/timer.h:
Enthält Funktionsdeklarationen und Konstanten für die Verwendung von Softwaretimern.

5.3.4 API-Implementierungen

/UART_SPI_TIMER_API/AVR/src/timer.c:
/UART_SPI_TIMER_API/MSP430/src/timer.c:
/UART_SPI_TIMER_API/KNXcalibur/src/timer.c:
In dieser Datei sind die Funktionen für den Softwaretimer realisiert.

/UART_SPI_TIMER_API/AVR/src/uart.c:
/UART_SPI_TIMER_API/MSP430/src/uart.c:
/UART_SPI_TIMER_API/KNXcalibur/src/uart.c:
Hier sind die Funktionen für die UART- und SPI-Schnittstellen realisiert.

/UART_SPI_TIMER_API/AVR/src/demo/demo.c:
/UART_SPI_TIMER_API/MSP430/src/demo/demo.c:
/UART_SPI_TIMER_API/KNXcalibur/src/demo/demo.c:
Diese Datei enthält ein Beispielprogramm für die Verwendung von UART und Softwaretimer.

5.3.5 Benutzerinformationen zur Verwendung der API's

ATMega16 Die API für den ATMega16 wurde unter Linux mit dem avr-gcc Compiler erstellt. Dieser war in der Version 4.2.1 vorhanden. Für den Download wurde das Programm uisp in der Version 20050207 verwendet. Informationen zu Installation und Konfiguration sind in [AVR-GCC] zu finden.

Um das Beispielprogramm testen zu können, ist im Ordner `demo/` zusätzlich zum Programm *demo.c* noch ein `Makefile` enthalten, das unter Linux mit dem Befehl `make clean all install` das Beispielprogramm kompiliert und auf den Microcontroller lädt.

MSP430 Bei der Erstellung der API für den MSP430 wurde der msp-gcc Compiler unter Linux verwendet. Dieser war in der Version 3.2.3 vorhanden. Für den Download wurde das Programm msp430-jtag in der Version 2.3 verwendet. Nähere Informationen zu Installation und Konfiguration sind in [MSP-GCC] zu finden.

Um das Beispielprogramm für den MSP430 testen zu können ist im Ordner `demo/`, ebenso wie im vorherigen Abschnitt beschrieben, zusätzlich zum

Programm *demo.c* noch ein `Makefile` enthalten, das unter Linux mit dem Befehl `make clean all install` das Beispielprogramm kompiliert und auf den Microcontroller lädt.

KNXcalibur Im Fall des KNXcalibur wurde die Entwicklungsumgebung Softune Workbench in der Version V30L23 unter Windows verwendet. Zum Download auf KNXcalibur wurde der Fujitsu Flash Microcontroller Programmer verwendet, welcher in Version V01L23 zur Verfügung stand. Informationen zur Softune Workbench und zum Flash MCU Programmer sind in [KNX2] und [KNX3] zu finden. Informationen zum Download eines Programmes auf KNXcalibur sind in [PRA05] zu finden.

Das Beispielprogramm liegt bereits in kompilierter Form vor (`demo.mhx`). Es muss mithilfe des Flash Programmers auf KNXcalibur geladen werden.

6 Analyse

Dieser Abschnitt beschäftigt sich mit den Ausführungszeiten der Hardware Timer ISRs. Dies ist von großer Bedeutung im Bezug auf die maximale Taktrate des Software Timers. Je länger die Ausführungszeit der Hardware Timer ISR ist, desto größer muss auch die Dauer eines Taktes des Software Timers sein, um bei oftmaligem Aufruf eines Software Callbacks auch die Befehle darin ausführen zu können, beziehungsweise wartende Callbacks nicht zu lange zu behindern.

Die Analyse wurde folgendermaßen durchgeführt:

6.1 Hardware-Timer-ISR Ausführungszeit

Es wurde die Zeit gemessen, die der Microcontroller benötigt, um in die Hardware-Timer-ISR zu gelangen. Dazu wurde ein Programm geschrieben, welches das Hardware Timer-Intervall derart kurz wählt, dass nach der Ausführung der ISR ohne Unterbrechung in die nächste ISR gesprungen wird, weil am Ende der ISR das Interrupt Flag des Hardware Timers sofort wieder aktiviert wird, ohne einen anderen Programmbefehl ausführen zu können. Sobald in die ISR gesprungen wurde, wird ein Pin von logisch „0“ auf logisch „1“ gesetzt, beim nächsten Sprung in die ISR wird der Pin wieder umgeschaltet, etc. Dann wurden die Werte aller verwendeten Microcontroller verglichen.

Als erstes wurde der ATMega16 begutachtet. Er arbeitet mit 16MHz, dass heißt, ein Taktzyklus dauert 62,5 Nanosekunden. In Abbildung 23 ist die Dauer des Hardware-Timer-ISR Aufrufes zu sehen. Sie beträgt 2,1288μs.

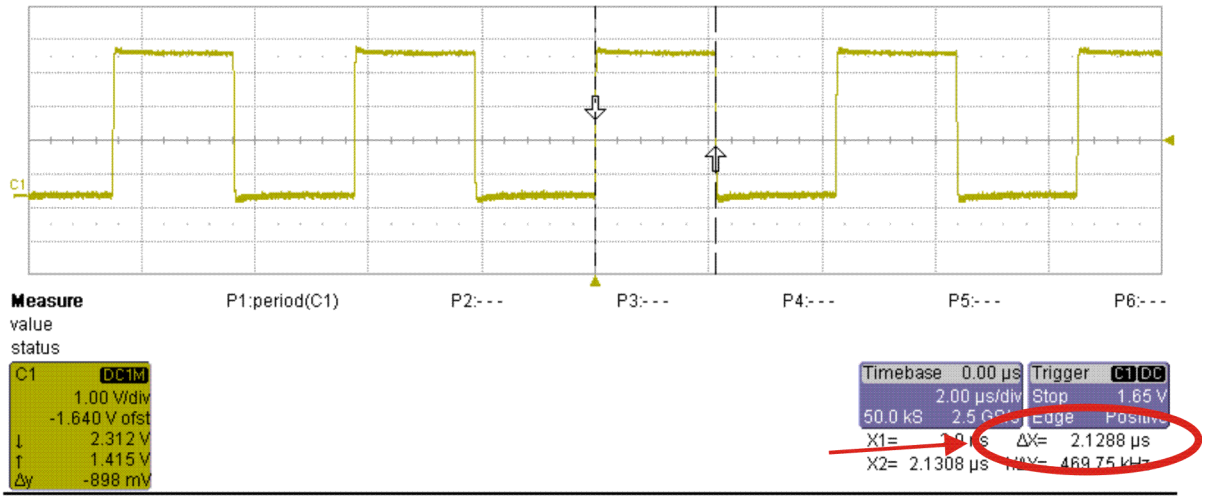


Abbildung 23: Messung von ATMega16 Timeraufruf

Die Ausführungszeit des MSP430 ist in Abbildung 24 zu sehen. Der MSP430 arbeitet mit 8MHz - ein Taktzyklus dauert 125ns - und benötigt für den Aufruf 1,8752μs.

Die Zeit, die KNXcalibur zum Aufruf des Timer benötigt, beträgt 1,1644μs und ist in Abbildung 25 dargestellt. Ein Taktzyklus dauert 41,67ns bei einer Taktrate von 24MHz.

Der Vergleich der Microcontroller auf Abbildung 26 zeigt, dass der MSP430 trotz halber Taktrate, eine um 13% schnellere Aufrufzeit für die Hardware-Timer-ISR besitzt als der ATMega16. Dies liegt wahrscheinlich daran, dass der MSP 16-Bit Register besitzt, und damit die Statusregister schneller zwischenspeichern kann. KNXcalibur ist in der Ausführung 82% schneller als der ATMega16 und um 61% schneller als der MSP430. Dass der KNXcalibur der schnellste ist, verwundert nicht aufgrund der höheren Taktrate.

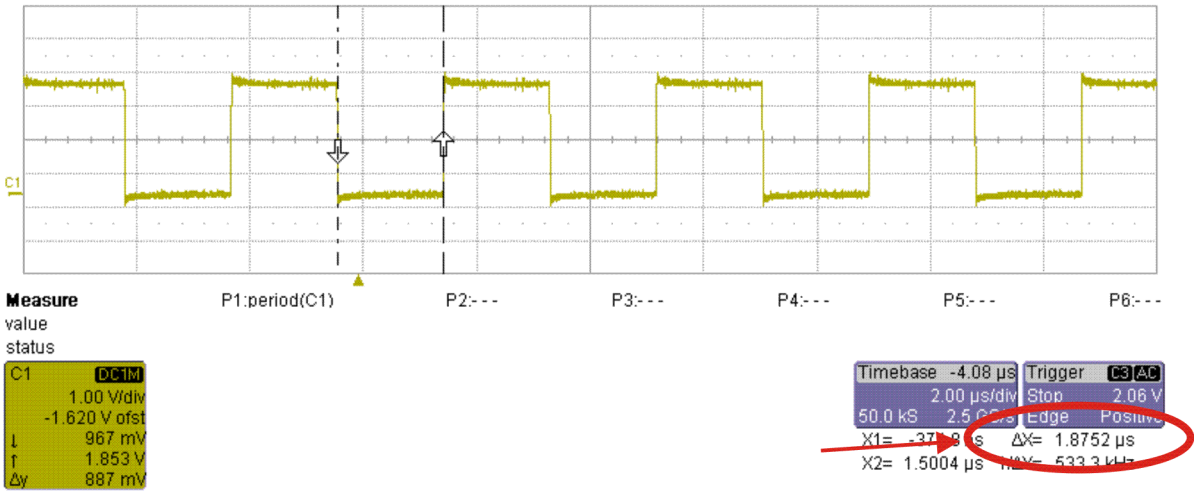


Abbildung 24: Messung von MSP430 Timeraufruf

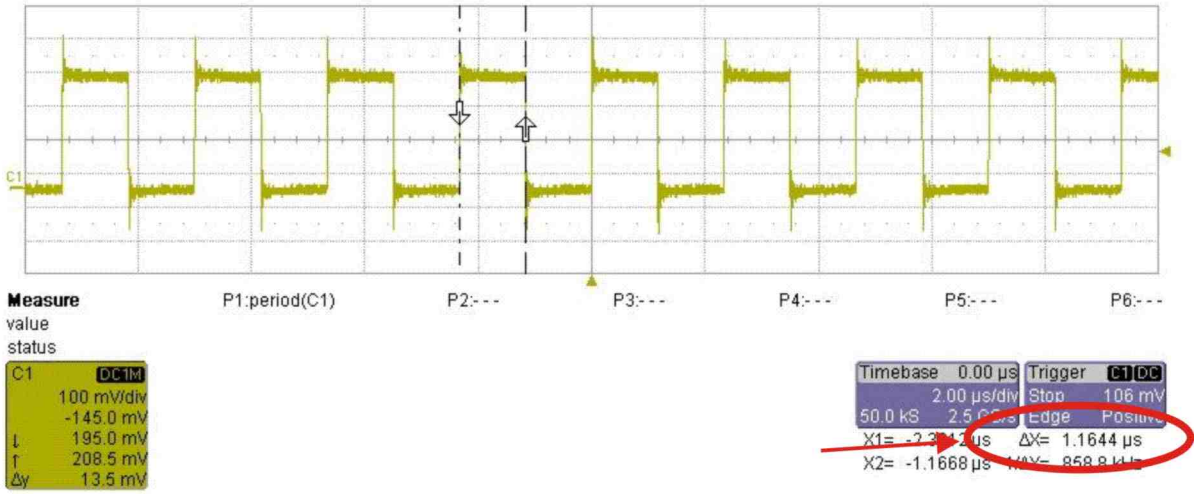


Abbildung 25: Messung von KNXcalibur Timeraufruf

6.2 Ausführungszeit der nicht optimierten Hardware-Timer-ISR

Als nächstes wurde die Hardware-Timer-ISR-Laufzeit untersucht. Am Anfang und Ende der ISR wurde ein Pin von logisch „0“ auf logisch „1“ und

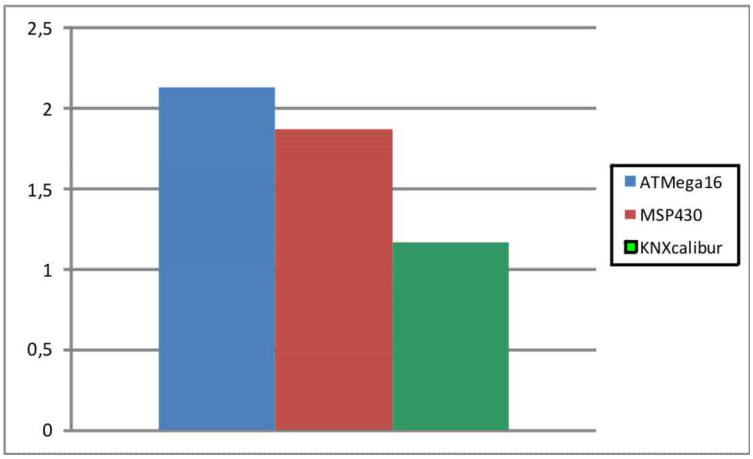


Abbildung 26: Vergleich der Timeraufrufzeiten

zurück geschaltet, um herauszufinden wie lange der Microcontroller für die Bearbeitung der Befehle der ISR benötigt. Dann wurden diese Werte wieder verglichen.

Der ATMega16 benötigt zur Ausführung der nicht optimierten ISR 46,2576μs (Abbildung 27).

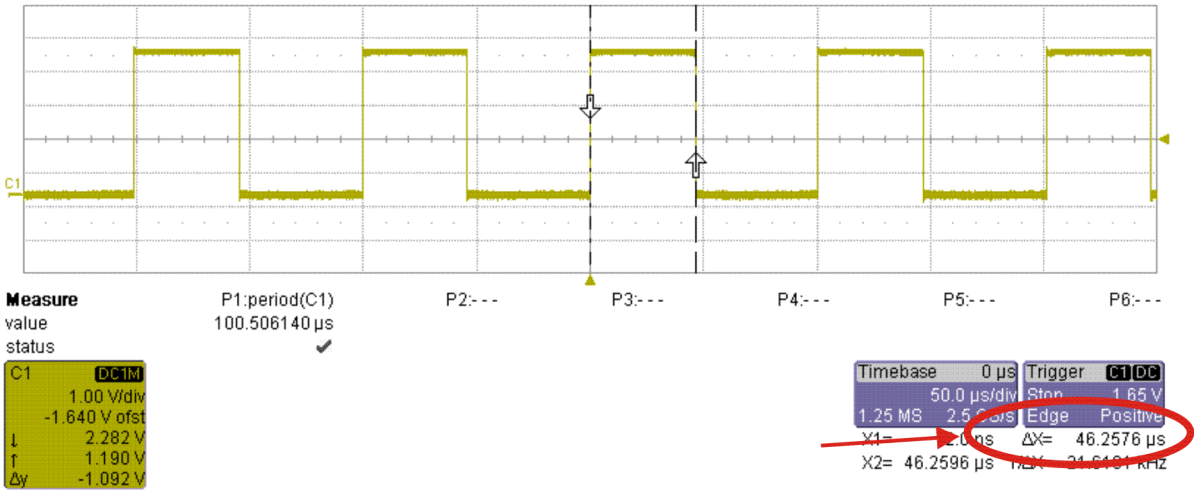


Abbildung 27: Ausführungszeit der nichtoptimierten TimerISR des ATMega16

Die Ausführungszeit des MSP430 für die Timer-ISR beträgt 58,9936μs (Abbildung 28).

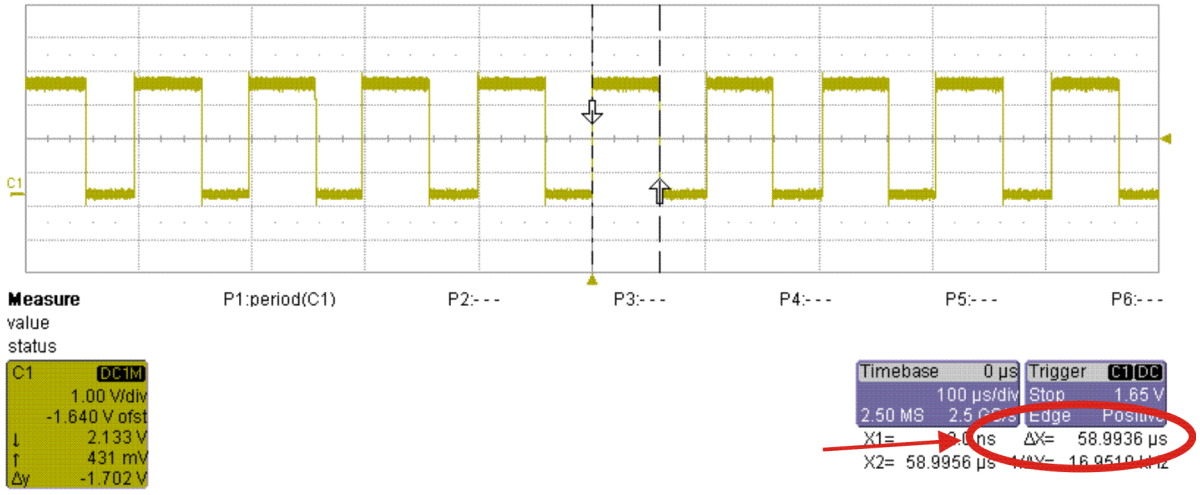


Abbildung 28: Ausführungszeit der nichtoptimierten Timer-ISR des MSP430

KNXcalibur benötigt zur Ausführung der nicht optimierten Timer-ISR 54,7890μs (Abbildung 29).

Der Vergleich in Abbildung 30 zeigt, dass der AVR bei doppelter Takrate, nur um 28% schneller ist, was wiederum wahrscheinlich auf einen besser optimierten Befehlssatz, bzw. Compiler zurückzuführen ist. Außerdem ist der ATMega16 in der Ausführung interessanter Weise auch um 18% schneller als KNXcalibur, was wahrscheinlich auf die Von-Neumann-Architektur des KNXcalibur liegt, der die Programmbefehle und Daten aus dem selben Speicher laden muss, und dadurch mehr Zeit benötigt als der ATMega16, welcher seine Programmbefehle und Daten aus unterschiedlichen Speichern bezieht, bedingt durch die Harvard-Architektur der CPU.

6.3 Ausführungszeit der optimierten Hardware-Timer-ISR

Dann wurde die Laufzeit der Hardware-Timer-ISR optimiert. Dies wurde erreicht, indem der Auswahlmechanismus, der entscheidet welcher Software-Timer-Callback als nächste ausgeführt wird, direkt in die Hardware-Timer-

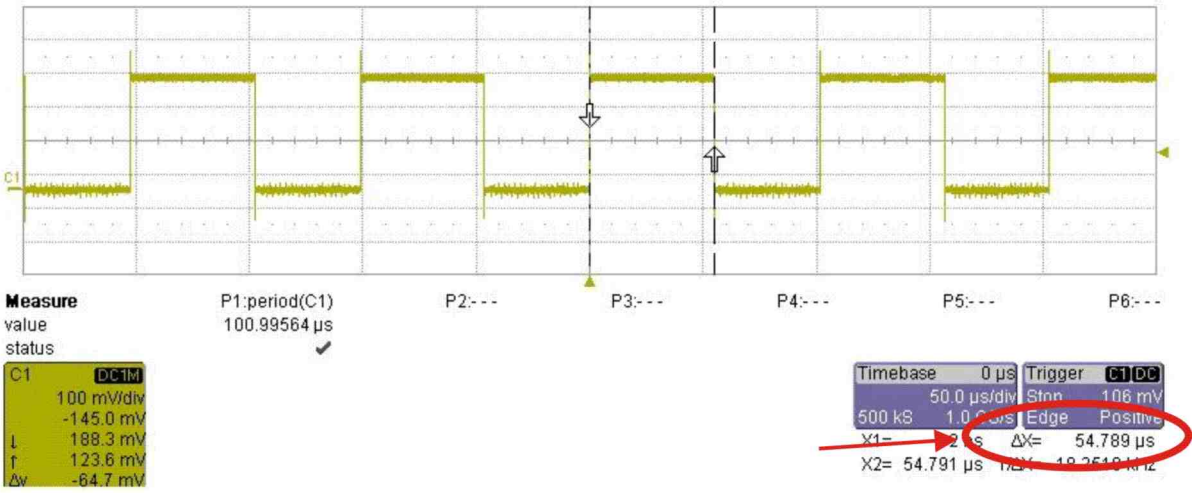


Abbildung 29: Ausführungszeit der nichtoptimierten Timer-ISR des KNXcalibur

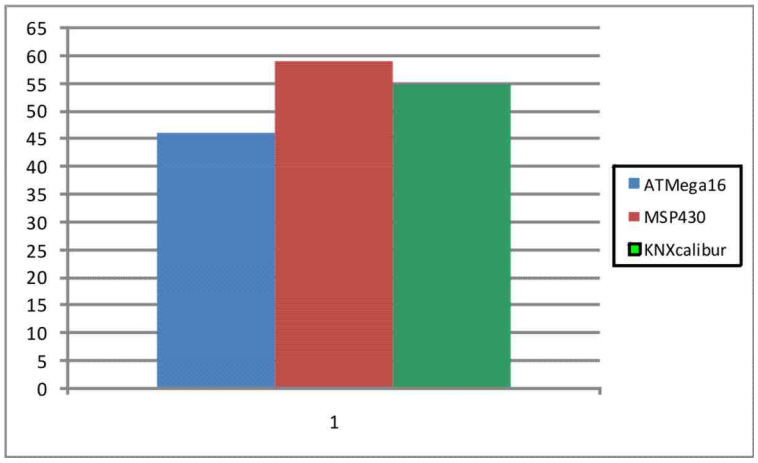


Abbildung 30: Vergleich der Timerausführungszeiten

ISR integriert wurde, und nicht als eigenständige Routine in der ISR aufgerufen wurde.

Der ATMega16 erreichte eine Zeit von 31,5072μs(Abbildung 31) bei der Ausführung der optimierten ISR, was einer Laufzeitoptimierung von 47% entspricht. Dies zeigt, dass ein Prozeduraufruf ein sehr kostspieliges Unterfangen

in Bezug auf die Laufzeit darstellt.

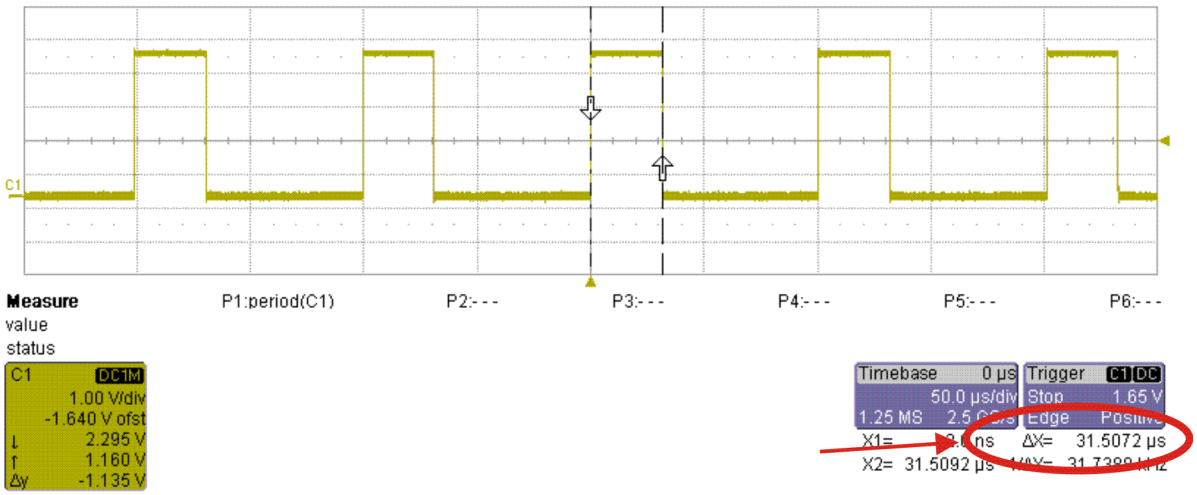


Abbildung 31: Ausführungszeit der optimierten AVR-ISR

Beim MSP430 wurde eine Zeit von 44,4952μs erreicht, also eine Optimierung der Laufzeit von 33%(Abbildung 32).

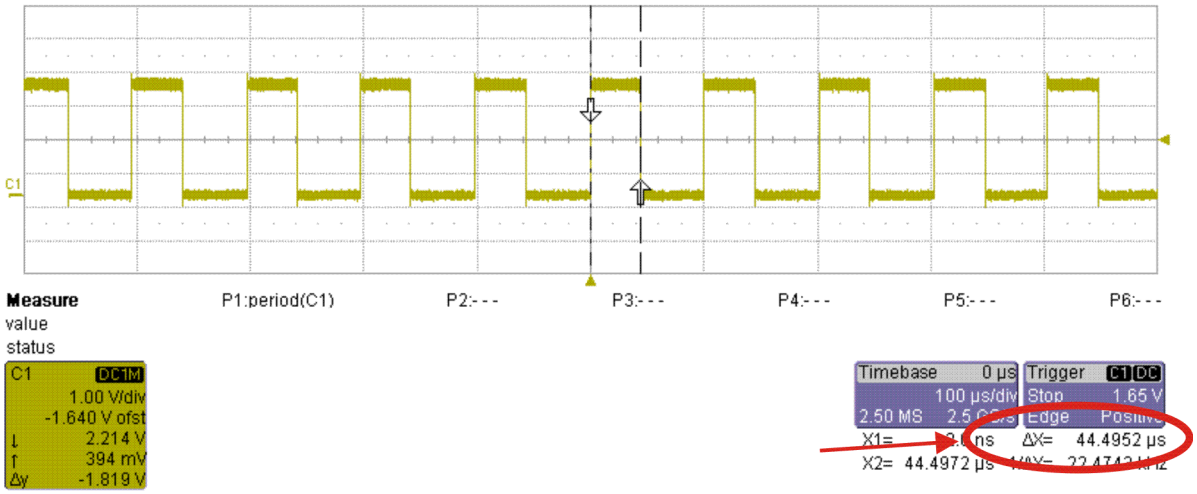


Abbildung 32: Ausführungszeit der optimierten MSP-ISR

Beim MSP430 wurde eine Zeit von 40,5380μs erreicht, also eine Optimierung der Laufzeit von 35%(Abbildung 33).

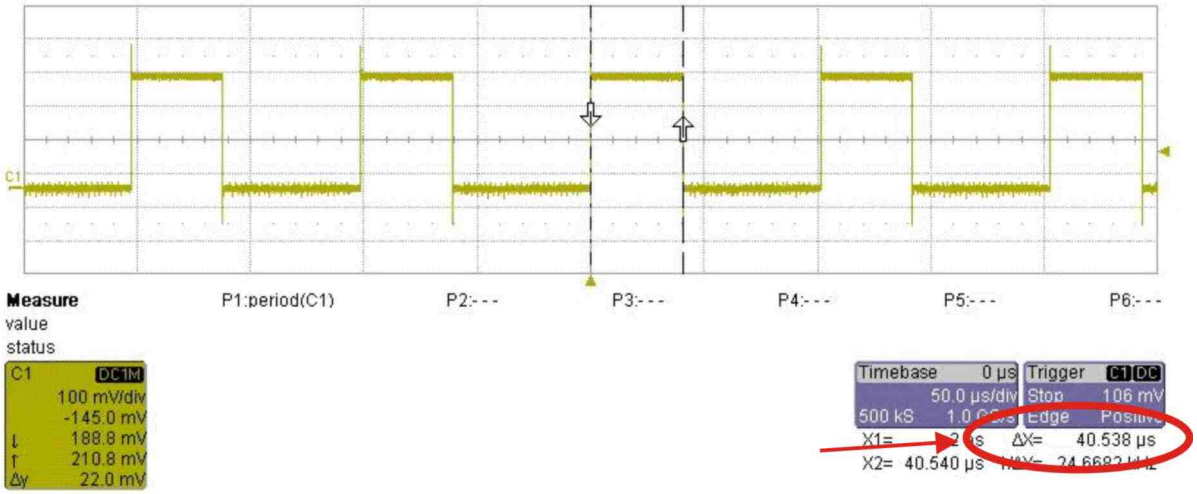


Abbildung 33: Ausführungszeit der optimierten KNXcalibur-ISR

Interessant ist, dass die Optimierung der Hardware-Timer-ISR beim ATMega16 eine um 14% schnellere Ausführungszeit brachte als beim MSP430, und eine um 12% schnellere Ausführungszeit als KNXcalibur. Die geringere Verbesserung der Ausführungszeiten im Vergleich zum ATMega16 kommt wahrscheinlich daher, dass die ohnehin schon besser optimierten Befehlssätze bzw. Compiler keine derartige Beschleunigung der Ausführung im Vergleich zum ATMega16 zulässt. Diese Zusammenhang soll durch Abbildung 34 verdeutlicht werden.

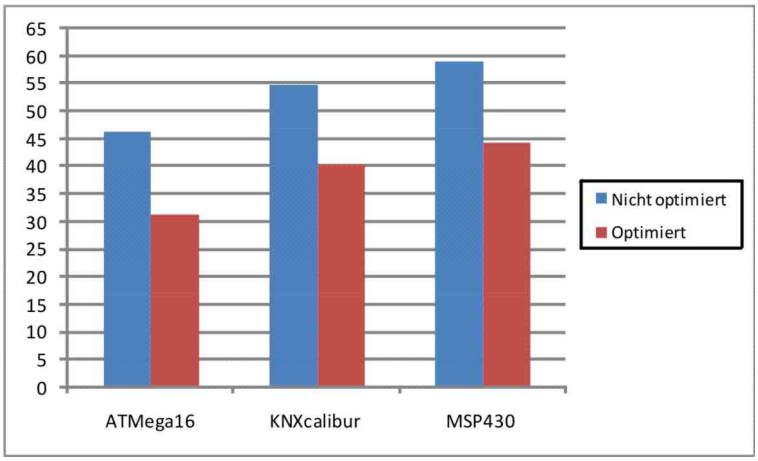


Abbildung 34: Vergleich der optimierten Timerausführungszeiten

Literatur

[ATMega16] Datenblatt des ATMEL ATMega16. Download unter http://www.atmel.com/.

[MSP430] Datenblatt des MSP430f149. Zu Beziegen unter http://focus.ti.com/.

[PRA05] Fritz Praus. A versatile networked embedded platform for KNX/EIB. Diplomarbeit, Technische Universität Wien, Wien, 2005.

[SPI] Application Note AN991 von *freescale semiconductor* zur Verwendung des SPI-Protokolls.

[I^{2}C-Bus] Spezifikation des I^{2}C Datenübertragungsprotokolls. Zu Beziehen unter http://www.nxp.com/.

[AVR-GCC] http://www.mikrocontroller.net/articles/AVR_und_Linux

[MSP-GCC] http://mspgcc.sourceforge.net/

[KNX2] http://mcu.emea.fujitsu.com/mcu_tool/detail/SWB_(F2MC-16)_V3.htm

[KNX3] http://mcu.emea.fujitsu.com/mcu_tool/detail/FLASH_PROGRAMMER_16LX.htm

[KNXcalAPP] Application note MCU-AN-300002-E-V12 von Fujitsu Microelectronics Europe zur Verwendung des SPI-Protokolls unter KNXcalibur.

[MSP430-HP] http://www.siphec.com/HOWTO/MSP430%20HOWTO/