Contents

1 Einleitung

1.1 Einführung in das Themenfeld

Kommunikation zwischen Mikrokontrollern unterscheidet sich grundsätzlich in 2 Kategorien. Serielle Kommunikation und parallele Kommunikation. In den meisten Mikrokontrollern, Sensoren und weiteren Peripherals hat sich die serielle Kommunikation für die meisten Anwendungen als Standard durchgesetzt. Der hauptsächliche Grund dafür ist, dass mit der seriellen Kommunikation an Pins und Adern der Verbindung gespart werden kann und somit Kosten und Formfaktor stark reduziert werden können.

Aus diesem Grund haben viele Mikrokontroller bereits Hardwarekomponenten, die gängige serielle Protokolle vollständig in Hardware implementiert anbieten, oder zu mindestens Hardwarekomponenten bereitstellen, die speziell für diese konfiguriert und genutzt werden können.

1.2 Motivation

Diese Arbeit konzentriert sich auf die Entwicklung mit der ATtiny25/45/85 Mikrokontrollerserie von Atmel[2].

Der ATtiny hat einige Vorteile für die Entwicklung von Prototypen. Zum ersten ist er kostengünstig (ca 1 Euro / Chip), zum zweiten relativ gut mit öffentlichen Libraries unterstützt und dank einer aktiven Community sowie Einbindung in das Arduino [10] Ökosystem einfach zu programmieren und zu flashen.

Der Nachteil von der ATtiny-Serie sind seine begrenzten Kommunikationsinterfaces und Hardwareressourcen. Da der ATtiny nicht über ein integriertes UART Modul verfügt, gibt es eine Software-Implementierung, genannt SofwareSerial[4], die eine UART Kommunikation mit dem ATtiny ermöglicht. Das Problem dabei ist, dass diese mit keinerlei Unterstützung der Hardware arbeitet und daher viel mehr Zyklen des Prozessors nutzt als eigentlich nötig. Außerdem leidet die maximale Baudrate (Kommunikationsgeschwindigkeit) unter der Sofwareimplementierung. Da die Implementierung auch stark Interruptgesteuert ist kann diese zusätzlich den Fluss der Programmausführung stark stören.

2 Konzept

2.1 UART Protokoll

Es gibt mehrere serielle Kommunikationsprotokolle, die für bestimmte Zwecke optimiert sind. Eine Übersicht der meist verbreiteten zeigt `Tabelle 2.1`.

	Pins	Typ	Architektur	Distanz	Reihenschaltung
UART	2	asynchron/full&half duplex	kein Master	∼5m	unterstützt
SPI	4	synchron/full-duplex	Master-Slave	<5m	teilweise
I2C	2	synchron/half-duplex	Master-Slave	∼1m	nicht unterstützt

Table 2.1: Serielle Kommunikationsprotokolle

Die Möglichkeit, UART Geräte in Reihe zu schalten, sodass das Signal bei jedem Gerät erneut verstärkt wird, sowie die geringe Anzahl an Pins und die relativ hohe Maximaldistanz zwischen den Kommunikationspartnern machen UART zu dem klaren Favoriten für dieses Projekt.

2.2 Funktionalität und Limitationen des USI Moduls

Das ATtiny Universal Serial Interface [2] ist ein Interface, dass Hardwarekomponenten enthält die serielle Kommunikation unterstützen können. Die Protokollimplementierung ist dabei dem Nutzer überlassen. Das bedeutet, das USI Modul stellt lediglich einige Hardwareregister, Interrupts und Konfigurationsoptionen bereit, die benutzt werden können um half-duplex Protokolle wie I2C und UART zu implementieren, `Figure 2.1`.

Die genauen Hardwarefunktionalitäten, die dafür zur Verfügung gestellt werden sind:

- 1-byte Datenregister (USIDR)
- 1-byte Bufferregister (USIBR)
- Statusregister (USISR)
- Kontrollregister (USICR)
- 4-Bit Counter (0-15) mit verschiedenen Taktquellen (extern, Timer etc..)

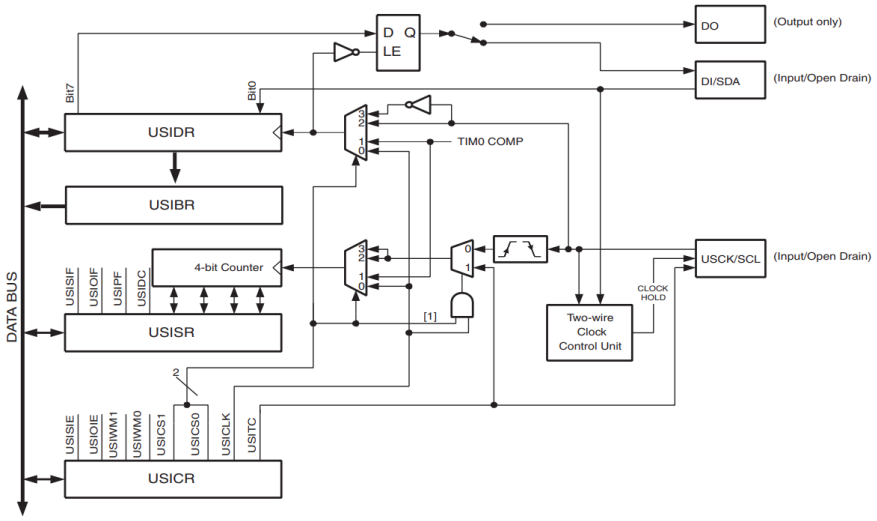

Figure 2.1: USI Blockdiagramm

- USI Overflow interrupt (USI_OVF)
- USI Start Condition Interrupt (USI_START)

Nicht USI-gebundene ATtiny Peripherals die für die hier beschriebene Implementierung genutzt werden zeigt `Tabelle` 2.2.

Timer/Counter0	warten bis zur Mitte des Startbits (empfangen) & USI Timer clocking (senden)
Timer/Counter1	Byte Empfangszeit messen & Sendepause
Output Compare Register & Interrupts	Timer konfigurieren und auf Timer reagieren
Pinchange Interrupt PCINT0	Startbit des Dateninputpins erkennen

Table 2.2: Genutzte ATtiny Peripherals

2.3 Verwandte Arbeiten

- Atmel Appnote AVR307[3] ist eine Application Note von Atmel die beschreibt, wie ein einfaches UART für die strukturell verwandten Mikrokontroller ATtiny26, ATtiny2313 und ATmega169 implementiert werden kann.
- SoftwareSerial [4] ist eine Software Implementierung des UART Protokolls, die keine der USI Hardware Peripherals benutzt.

- MarkOsborne USISerialSend.ino[9] ist ein Skript, dass das Senden von einzelnen Bytes mithilfe des USI-Moduls auf dem ATtiny ermöglicht. In dieser Implementierung entstehen jedoch die weiter unten beschriebenen Spikes, die die Kommunikation stören können. Außerdem ist das Füllen des USI Datenregisters nicht optimiert.
- MarkOsborne USISerial.ino[8] ist ein Skript, dass das Empfangen von UART Nachrichten auf dem ATtiny mit dem USI Modul ermöglicht.

2.4 Kommunikationsprotokoll und resultierende Charakteristik

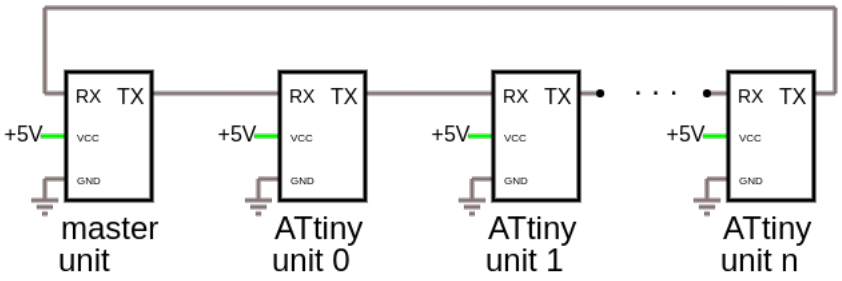

Figure 2.2: Kommunikationsarchitektur

Das implementierte Kommunikationsprotokoll ist ein single-master, half-duplex UART Protokoll für in Reihe geschaltete ATtinys, `Figure 2.2`. Da die einzelnen Mikrokontroller miteinander in Serie verbunden werden, kann es prinzipiell zu Kollisionen kommen. Dieser Fall tritt auf, wenn eine Unit an ihre Nachfolgende in der Kette eine Nachricht schickt, obwohl diese die vorherige Nachricht noch nicht vollständig an wiederum ihre Nachfolgende weitergeleitet hat, siehe `Figure 2.3`. Aufgrund der Tatsache, dass das ATtiny USI Modul nur ein Register für aus- und eingehende Daten hat, kann die entstehende Kollision ohne weiteres nicht behoben werden.

Um diesen Fall zu vermeiden, gibt es eine vom Nutzer der Software definierte "Sendepause", die jede Unit einhalten muss, nachdem sie eine Nachricht gesendet hat um der nachfolgenden Unit die Zeit zu geben, um die Nachricht vollständig zu verarbeiten und weiterzuleiten, siehe `Figure 2.4`. Sendebefehle an die Library, die zu dieser Zeit getätigt werden, werden nicht ausgeführt, sondern benachrichtigen den Nutzer mit einem Rückgabewert, dass diese Aktion zurzeit ungültig ist.

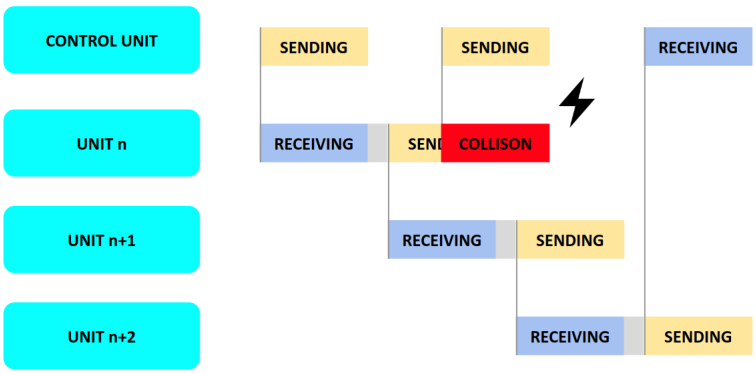

Figure 2.3: Kollision von Paketen

Die Sendepause muss dabei vom Nutzer definiert sein, da sie nicht nur von der Sendezeit, sondern auch von der Verarbeitungszeit der Nachricht abhängt. Wenn also eine Unit lange mit den Daten der Nachricht rechnen muss, bevor diese weitergesendet werden kann, dann muss die Sendepause entsprechend angepasst werden.

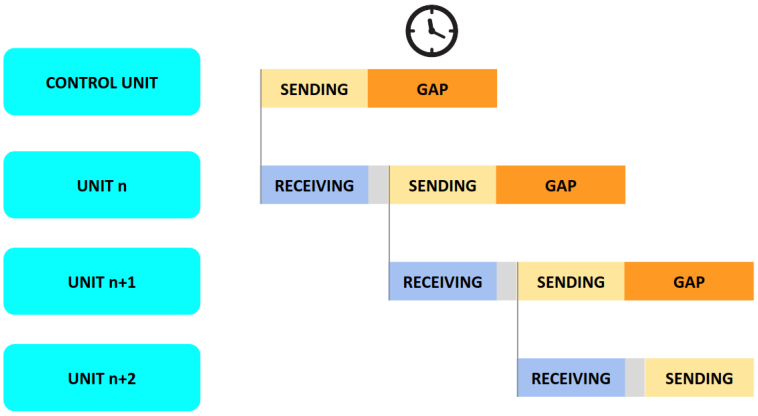

Figure 2.4: Kollisionsvermeidung

3 Implementierung

3.1 Projektstruktur

Die hier beschriebene Implementierung ist open-source auf Github zugängig [1]. Die Orderstruktur folgt dem Vorbild von Platformio[11], welches das Tool ist, mit dem der Code für die Mikrokontroller gebaut und geflasht wurde. Die platformio.ini Datei zeigt die möglichen build targets und ihre jeweilige Konfiguration.

Der /src/ Ordner enhält die Datein für

- eine UART Echo Implementierung
- ein Test Skript zum Prüfen und feinjustieren der Taktfrequenz der ATtinys
- ein Sourcefile für die angepasste FIFO Implementation
- die Sourcefiles für die ATtinys als UART Units
- das Sourcefile für die Masterunit (einen Arduino Nano)

Der /include/ Order enthält zugehörige Header-Dateien.

3.2 Implementierungsdetails

3.2.1 Statusdiagramm

Die Implementierung basiert auf 4 verschiedenen Stati, in denen sich die Library befinden kann. Im Anfangszustand befindet sich jede Unit im "READY" Status, von da aus können sowohl ein Startbit der vorherigen Unit als auch einen Sendebefehl durch den Nutzer der Library einen Statusübergang erzeugen. Die weiteren Stati und Übergänge zeigt `Figure` 3.1.

3.2.2 USI Sende-Buffer

Eine der Herausforderung für die Implementierung des UART Protokoll mithilfe des USI Moduls ist das Aufteilen einer Nachricht in byte-große Segmente und das rechtzeitige Füllen des USI Datenregister mit diesen Segmenten. Da Das USI Datenregister nur 1 Byte groß ist, und das UART Protokoll sowohl 1 Startbit, als auch mindestens 1 Stopbit und in manchen Fällen sogar zusätzliche Paritätsbits pro payload Byte benötigt, muss das USI Register mehr als 1 mal gefüllt werden, um ein

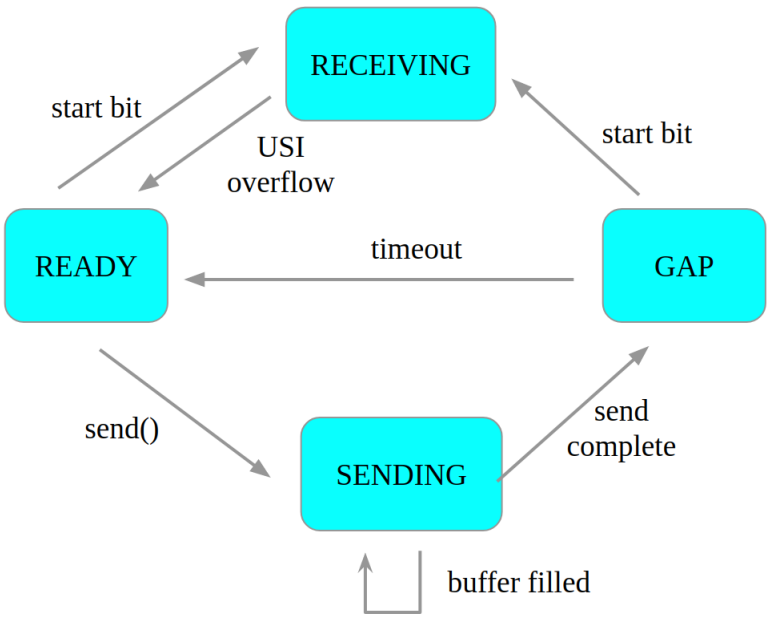

Figure 3.1: Statusdiagramm

Byte zu senden. Um das USI so effizient wie möglich zu nutzen (d.h immer wenn möglich den Buffer ganz zu füllen) muss also in der Standardkonfiguration (`1 startbit, 1 stopbit, keine Paritätsbits`) zwischen 5 Fällen unterschieden werden, siehe `Figure 3.2`.

USIDR							
START	1	2	3	4	5	6	7
8	STOP	START	1	2	3	4	5
6	7	8	STOP	START	1	2	3
4	5	6	7	8	STOP	START	1
2	3	4	5	6	7	8	STOP

Figure 3.2: USI Data Register

Diese Segmentierung von Bytes bedeutet, dass der USI buffer neu gefüllt werden muss während ein Byte übertragen wird. Wenn der gesamte USI buffer geleert wurde, führt dies zu einem *USI_OVF* Interrupt der sofort von der Implementierung behandelt wird, um den Buffer neu zu füllen. Da jedoch trotz Optimierung durch Doppelbuffering (der jeweils nächste USI Buffer steht schon bereit wenn

der Interrupt ausgelöst wird) einige wenige CPU Zyklen zwischen dem Feuern des Interrupts und dem Füllen des Buffers vergehen, ist der Datenoutputpin von dem USI Modul für kurze Zeit ungesteuert. Dies führt zu Spikes, die nur wenige Nanosekunden lang sind, jedoch von dem Nachfolgenden Gerät als Flanke interpretiert werden können und somit die Kommunikation stören, `Figure` 3.3.

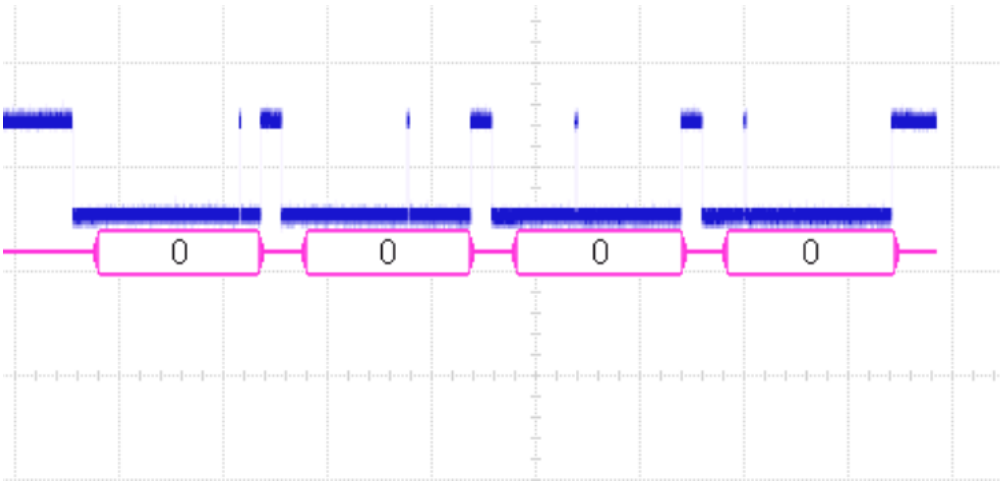

Figure 3.3: Statusdiagramm

Um diese Spikes zu verhindern, werden `Timer1` und die korrespondierenden Compare-interrupts genutzt. Bei Beginn einer Byteübertragung wird *Timer0* konfiguriert, um das USI Modul zu steuern, das heißt, dass *Timer0* auf die Länge eines Bits gesetzt wird. Zur gleichen Zeit wird *Timer1* mit einer höheren Taktfrequenz konfiguriert um parallel mitzulaufen. Da der Zeitpunkt des *USI_OVF* interrupts bekannt ist, wird `Timer1` genutzt um kurz vor Auftreten dieses Interrupts einen eigenen Interrupt zu feuern. In der ISR wird dann das USI Modul kurzzeitig von dem Data Output Pin getrennt. Sobald dann der echte *USI_OVF* Interrupt feuert, füllen wir den USI Outputbuffer und verbinden das USI Modul wieder mit dem Data Output Pin. Das bewirkt, dass das der Data Output Pin zu keinem Zeitpunkt unerwünschtes Verhalten zeigt.

4 Anwendungsbeispiel

4.1 Konzept zur synchronen Steuerung von Lichteffekten

Das Anwendungsbeispiel, das dem gesamten Projekt zugrunde liegt, ist eine Lichtinstallation mit vielen einzelnen Units (bis zu 80). Die Units steuern jeweils eine LED, die Licht in einen Acrylstab streut. Für diese Installation sind die einzelnen Units bis zu 5 Meter auseinander platziert. Die Anforderung ist, dass trotz der Daisy-Chain Architektur Lichteffekte programmiert werden können, die synchron über mehrere Units stattfinden können. D.h, der Kommunikationsweg für einen gegebenen Befehl muss berücksichtigt werden. Warum das relevant ist, zeigt die folgene Rechnung:

Beispielrechnung für die Latenz einer Nachricht an 1) ein Gerät, 2) alle Geräte:
Clock Speed: 8MHz
Baud Rate: 115200
Latenz (worst case), Nachricht an einen:

$$\frac{1}{115200} x8x6x80 = 0.03$$

Latenz (worst case), Nachricht an alle:

$$\frac{1}{115200} x8x6x3240 = \ 1.35$$

Wenn also alle Units gleichzeitig aufblinken sollen, würden Sie dafür 1.35 Sekunden brauchen. Es bedarf also einer anderen Lösung.

4.2 Details zum Zeitstempel-Protokoll auf Anwendungsebene

Im Folgenden wird das Kommunikationsprotokoll auf Anwendungsebene für das oben genannte Anwendungsbeispiel beschrieben. Jeder ATtiny startet als unbekannte Unit ohne ID und wird durch einen Protokoll-Reset konfiguriert. ID's werden auf Grundlage der Rheinfolge der Units vergeben. Die Datenpakete sind immer 6 Byte groß. Im ersten Schritt sendet die Masterunit eine Nachricht "R00000". Die erste Unit nimmt die ID 0 an und sendet das Packet "R10000" weiter u.s.w. Gleichzeitig misst jede der Units wie viele eigene CPU Zyklen sie braucht, um die

Nachricht zu empfangen. Wenn die letzte Unit die Nachricht an die Masterunit gesendet hat, haben alle Units eine ID und eine Zeitmessung für das Empfangen eines Bytes und die Masterunit kennt die Anzahl der Units in der Kette.

Im nächsten Schritt sendet die Masterunit Pakete an alle Units mit dem Muster "Tnoooo", wobei n die ID der Unit ist. Die Unit, dessen ID genannt ist, nutzt die leeren Bytes der Nachricht um den vorher gemessenen Zeitwert zu kodieren und schickt die Nachricht weiter. Alle anderen Units leiten die Nachricht weiter bis sie wieder von der Masterunit empfangen wird.

Am Ende dieses Schritts kennt die Masterunit die Zeitmessungswerte aller Units, und kann somit eine lokale Zeitbasis für jede Unit errechnen. Dies kann hilfreich sein, da die ATtinys leicht unterschiedliche Taktraten der Piezokristalle haben können.

Die lokale Zeitbasis der einzelnen Units kann nun genutzt werden, um Nachrichten synchronisiert zu schicken, sodass sie zu einem definierten Zeitpunkt bei verschiedenen Units ankommen, siehe `Figure 4.1`.

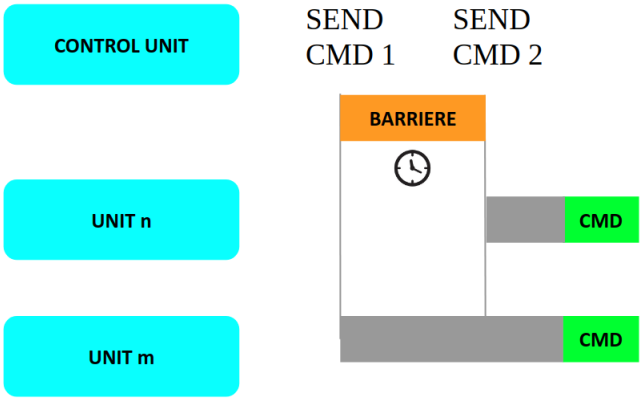

Figure 4.1: Zeitversetzte Steuerung

References

[1] Github Repository des Projekts,
https://github.com/boeckhoff/AttinyUSI

[2] Atmel ATtiny Datasheet,
http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2586-AVR-8-bit-Microcontroller-ATtiny25-ATtiny45-ATtiny85_Datasheet.pdf

[3] Atmel Appnote AVR307,
https://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en591877

[4] Arduino SoftwareSerial library,
https://github.com/arduino/ArduinoCore-avr/tree/master/libraries/SoftwareSerial

[5] AVR Instruction Set Manual,
http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf

[6] USI Modul,
https://www.mikrocontroller.net/articles/USI

[7] Basics of UART communication,
https://www.elprocus.com/basics-of-uart-communication-block-diagram-applications/

[8] USI Uart Serial Receive
http://becomingmaker.com/usi-serial-uart-ATtiny85/

[9] USI Uart Serial Send
http://becomingmaker.com/usi-serial-send-ATtiny/

[10] Arduino website,
https://www.arduino.cc/

[11] Platformio website,
https://platformio.org/

[12] I2C Bus Electrical Specifications,
http://www.mosaic-industries.com/embedded-systems/sbc-single-board-computers/freescale-hcs12-9s12-c-language/instrument-control/i2c-bus-specifications