

Low Cost μ C-Trigger

Technischer Report

Johannes Meng (johannes.meng@stud.uni-karlsruhe.de)

Dipl. Inform. Tobias Feldmann (feldmann@ira.uni-karlsruhe.de)

Dr. Ing. Annika Wörner (woerner@ira.uni-karlsruhe.de)

15. Oktober 2008

Zusammenfassung

Der Technische Report beschreibt, wie man mit geringen Kosten ein sehr genaues und komfortables externes Triggergerät mit Hilfe eines Mikrocontrollers (μ C) erstellen kann. Es werden sowohl der Hardwareaufbau auf Basis eines ATmega8- μ C als auch die implementierte Software vorgestellt und erläutert. Mit Hilfe des Triggers können z. B. mehrere Hochgeschwindigkeitskameras synchronisiert werden, um möglichst präzise 3-D-Rekonstruktionen aus hochfrequenten Kamerabildern zu generieren.

1 Einleitung

In Projekten der Arbeitsgruppe *Go Hu.MAn* sollen aus bewegten Kamerabildern mehrerer Kameras 3-D-Informationen extrahiert werden. Damit die 3-D-Rekonstruktion präzise durchgeführt werden kann, ist es notwendig, die verwendeten Kameras dazu zu bringen, ihre Bilder synchron aufzuzeichnen. Diese Synchroni-

sierung wird üblicherweise durch ein externes Triggersignal erreicht. Zu diesem Zweck gibt es auf dem Markt mehrere kommerzielle Triggerboxen zu kaufen. Deren Nachteil sind die recht hohen Preise. Eine Alternative stellen einfache Timer-Bausteine dar. Diese haben allerdings den Nachteil, dass sie häufig nur mit Frequenzen bis zu 30Hz betrieben werden können. Da wir in unseren Arbeiten Kameras mit Bildfrequenzen von 100-200 Bildern pro Sekunde einsetzen, fällt ein einfacher Timer-Baustein als Trigger aus. In diesem Report wird deshalb vorgestellt, wie man mit einem handelsüblichen μ C [Atm07] zu geringen Kosten ein eigenes externes Triggergerät erstellen kann, welches auf Millisekunden genaue Triggerimpulse mit variablen Impulslängen erlaubt.

2 Hardware

Für die Hardware des Triggers wurde auf das μ C-Modul *Crumb8-USB* der Firma *Chip45*

Menge	Beschreibung	Gesamtpreis
1	CrispAVR-USB	41,97 €
1	Crumb8-USB- μ C	20,97 €
1	Crumb8-USB-Connector Kit	1,64 €
2	USB-Kabel A/miniB	3,28 €

Tabelle 1: Teileliste für μ C-Trigger. Preise laut Rechnung (11.02.2008) zzgl. 19% USt.

*GmbH & Co. KG*¹ zurückgegriffen, da deren Module auf dem gängigen Atmel AVR- μ C basieren, darüber hinaus aber auch einige Peripherie, wie serielle Schnittstellen über USB, Status-LED, usw. direkt integrieren. Zusätzlich zum eigentlichen μ C wurde ein Programmieradapter *CrispAVR-USB* [Chi08a] zur Programmierung des μ C angeschafft. Daneben wurden noch Kleinteile, wie das *Connector Kit* und zwei USB-Kabel zur Steuerung und Stromversorgung des μ C und des Programmieradapters gekauft. Die komplette Teileliste ist in Tabelle 1 aufgeführt.

2.1 Hardwareaufbau des μ C-Moduls

Der ATmega8- μ C wurde auf einer fertig bestückten Platine geliefert. Zunächst wurde die Brücke J2 (vgl. Abb. 1) mit Lötzinn geschlossen, um das Modul zukünftig über die USB-Schnittstelle mit Strom zu versorgen, so dass kein separates Netzteil für den Betrieb des Triggers benötigt wird.

Aus praktischen Gründen wurde der μ C mit Hilfe des *Connector Kits* auf einer handelsüblichen Lochrasterplatine mit einem Lochraster von 2.54mm befestigt, um weitere Kabel zum triggern bequem anschließen zu können. Außerdem wurden Pins an die Reset-Kontakte (vgl.

¹<http://www.chip45.com/>

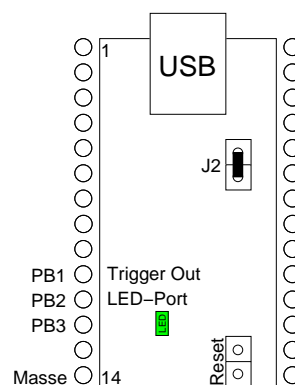


Abbildung 1: Crumb8-USB-Modul mit Belegung.

Abb. 1) gelötet, um den μ C im Falle eines Absturzes zurücksetzen zu können.

Für die Verwendung des Moduls als Trigger wurden die Pins 10, 11, 12 und 14 verwendet (vgl. Abb. 1). Pin 10 (PB1) und 12 (PB3) liefern die Spannung für Triggersignale, Pin 14 liefert die Masse. Über den Port B2 (PB2) kann die LED des μ C-Moduls geschaltet werden. In unserem konkreten Fall wird vorerst nur ein Triggersignal auf Pin 10 gelegt und parallel die LED ein- und ausgeschaltet. Pin 12 kann zukünftig dazu verwendet werden, ein weiteres Triggersignal auszugeben (z. B. für ein Setup mit Hochgeschwindigkeitskameras und parallel eingesetzten Kameras mit einer üblichen Framerate von 25-30 Bildern pro Sekunde).

3 Software

Die Software für μ C-Trigger besteht hauptsächlich aus einer Firmware, die den verwendeten ATmega8-Mikrocontroller ansteuert. Diese Firmware muss verschiedene Dinge leisten:

- Sie soll dafür sorgen, dass in regelmäßigen Abständen ein Trigger-Signal auf einen Ausgang gelegt wird. Alle angeschlossenen Kameras nehmen dann je ein Bild auf.

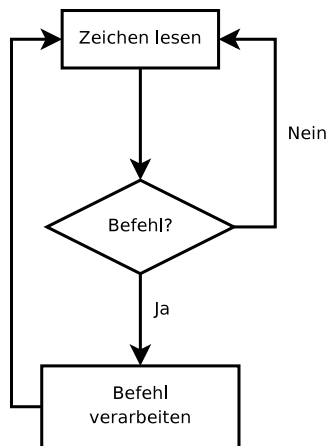


Abbildung 2: Programmfluss

- Parameter wie Impulsintervall und Impulslänge sollen möglichst frei einstellbar sein.
- Dazu ist eine Ansteuerung über die serielle Schnittstelle wünschenswert.
- Impulsdauer und -länge sollen darüber hinaus möglichst exakt umgesetzt werden.

Als Programmiersprache wird C benutzt; die Programme werden mit AVR-GCC übersetzt. Neben dem nahezu komplett von uns geschriebenen Programmteil in der Datei *trigger.c* werden die Funktionen `uart_init`, `uart_receive`, `uart_putc` und `uart_puts` eingesetzt. Diese sind im Wesentlichen aus der Dokumentation des Herstellers Atmel [Atm07] übernommen.

3.1 Konzept

Die Firmware des ATmega8 liest in einer Endlosschleife je ein Zeichen von der UART-Einheit², die wiederum die Eingaben der seriellen Schnittstelle weitergibt (vgl. Abb. 2). Wird das Zeichen als ein Kommando erkannt, so leitet das Programm eine entsprechende Behandlung ein. Ansonsten liest sie das nächste Zeichen ein. Zulässige Befehle sind:

²Universal Asynchronous Receiver Transmitter

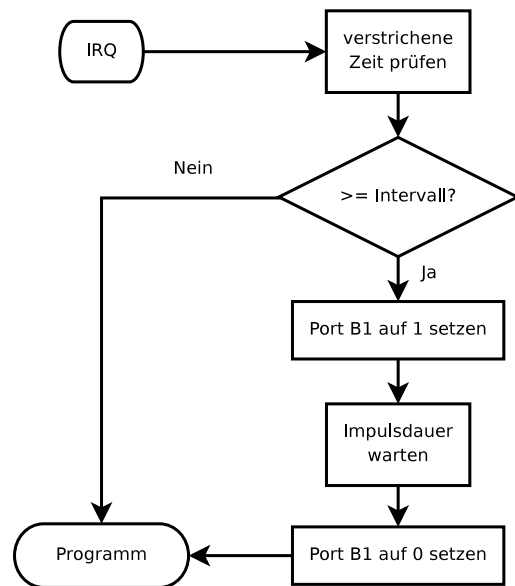


Abbildung 3: Interruptbehandlung

- **h** - Fordert die Ausgabe eines kurzen Hilfetextes über den UART an.
- **v** - Fordert Informationen über die benutzte Programmversion an.
- **i** - Kündigt die Angabe eines neuen Impulsintervalles an.
- **l** - Kündigt die Angabe einer neuen Impulslänge an.
- **p** - Fordert die Ausgabe des momentan gesetzten Impulsintervalles an.
- **q** - Fordert die Ausgabe der momentan gesetzten Impulslänge an.

Im Hintergrund läuft ein Timer des ATmega8. Dieser erzeugt in bestimmten Abständen einen Interrupt Request (IRQ), woraufhin der Prozessor seine Arbeit unterbricht und in die Interrupt Service Routine (ISR) wechselt (vgl. Abb. 3). Dort wird entschieden, ob die dem eingestellten Intervall entsprechende Zeit schon verstrichen ist. Ist dies der Fall, so wird eine logische 1 auf Port B1 erzeugt. Nach der eingestellten Impulsdauer wird der Port wieder auf 0 gesetzt und die

Listing 1: Initialisierung

```
168 // Initialize UART connection;
169 uart_init(F_CPU, BAUD);
170
171 // Set PortB 1 (trigger), 2 (LED) as output, PortD as input.
172 DDRB = (1 << PB1) | (1 << PB2);
173 DDRD = 0;
174
175 // Initialize timer1.
176 timerInit();
177
178 // Enable interrupts.
179 sei();
```

ISR terminiert. Daraufhin setzt der Prozessor seine ursprüngliche Arbeit fort, d. h. er wartet erneut auf Eingaben über die serielle Schnittstelle.

3.2 Durchführung

In den folgenden Abschnitten wird auf die reale Implementierung der Trigger-Software auf dem μ C eingegangen.

Beim Schreiben des Quelltextes wurde dabei besonderen Wert darauf gelegt, die in [Atm03] dargelegten Techniken zur effizienten Programmgestaltung umzusetzen, um ein möglichst effizient arbeitendes Programm zu erhalten.

3.2.1 Initialisierung

Die Initialisierung des Mikrocontrollers wird in Listing 1 abgebildet. Zunächst müssen dem UART die CPU-Geschwindigkeit und die erwünschte Baudrate übergeben werden. Um die korrekte Übertragungsrate von 57600 Baud liefern zu können, läuft der Mikrocontroller mit 14745600 Hz ($256 \times 57600 = 14745600$, die Bau-

drate ist also ein Teiler der Prozessorgeschwindigkeit).

Als nächstes werden die eingesetzten Ports für Eingabe (Port D, dieser wird für die serielle Kommunikation benötigt) bzw. Ausgabe (Port B1, Port B2) konfiguriert. Dazu sind in DDRB die Bits an den Stellen PB1 sowie PB2 auf 1 und das gesamte Register DDRD auf 0 zu setzen.

Nach der Initialisierung des Timers folgt noch der Befehl `sei()`, der alle Interrupts einschaltet.

3.2.2 timerInit()

In Listing 2 wird die Initialisierung des Timers gezeigt. Timer sind bei ATmega8-Controllern Zählregister, die im Prozessortakt (oder mit einem um Faktor³ 2, 8, 64, 256 oder 1024 kleineren Takt) aufwärts und/oder abwärts zählen. Erreichen sie einen gewissen Schwellwert, so kann ein IRQ an den Prozessor gesendet werden, woraufhin eine entsprechende ISR aufgerufen wird. Auf diese Art und Weise erreicht man die regelmäßige Abarbeitung einer Methode.

³Dieser Faktor heißt *Prescale*-Faktor

Listing 2: Initialisierung des Timers

```
250 // Don't use PWM mode.
251 TCCR1A = 0;
252
253 // We want the timer to be reset on overflow (WGM)
254 // and we want the timer to run on full MCU speed, i.e. prescale=1 (CS).
255 TCCR1B = (1 << WGM12) | (1 << CS10);
256
257 // Overflow comparator.
258 // The timer TCNT1 will count 0...OCR1A and then generate an IRQ.
259 // The next clock cycle, TCNT1 is reset to 0.
260 OCR1A = (uint16_t)((uint32_t)(F_CPU / TIMER_RES)) - 1;
261
262 // Enable timer overflow 1A interrupt.
263 TIMSK |= (1 << OCIE1A);
```

Der ATmega8 unterstützt von Haus aus Pulsweitenmodulation (PWM). Hierbei wird ein periodisches Signal auf einem Ausgang erzeugt, wobei die Periode (das Impulsintervall) konstant ist und die Impulsbreite sich ändert. Dies wird eingesetzt, um mit digitalen Signalen analoge zu approximieren. Diese Technik wird z. B. in der Prozedur `blink()` eingesetzt, um die LED auf dem eingesetzten Crumb8-Modul kontinuierlich heller und dunkler werden zu lassen. Da wir auch ein periodisches Signal auf einem Ausgang generieren wollen, scheint die PWM-Funktionalität eine optimale Lösung für unser Problem zu sein. Es fallen jedoch schnell einige Hindernisse auf:

1. PWM arbeitet mit fester Periode. Wir wollen ein (nahezu) frei einstellbares Impulsintervall. Dies ließe sich dadurch lösen, dass man eine Grundperiode von einer Millisekunde nutzt und die Generierung des Impulses auf bestimmte Zählerdurchläufe beschränkt.
2. Wegen unserer Wahl der Prozessorfrequenz von 14745600 Hz erreichen wir niemals eine Zählperiode von genau 1 ms – dazu müsste

die Frequenz ein Vielfaches von 1000 sein. Darum sind die zugelassenen Impulsintervalle, die allesamt Vielfache einer Millisekunde sind, nicht genau zu verwirklichen.

3. Wir lassen auch Impulsbreiten zu, die länger sind als eine Millisekunde. Dies wäre mit der eingebauten PWM-Funktionalität nur kompliziert umsetzbar, wenn wir wie in (1.) eine Zählperiode von einer Millisekunde annehmen.

Durch diese Schwierigkeiten unterscheidet sich der Aufwand kaum von einer selbst entworfenen Lösung, die sich lediglich auf einen Timer, nicht jedoch auf automatische Pulswellenerzeugung verlässt. Hierbei ist wichtig zu bemerken, dass (3.) wesentlich einfacher gelöst werden kann (dies werden wir im Folgenden noch behandeln), die Ungenauigkeit des Timers, beschrieben in (2.), allerdings in gewissem Maße erhalten bleibt.

Wir initialisieren den 16-Bit-Timer `timer1` so, dass der PWM-Modus *nicht* benutzt wird. Außerdem möchten wir, dass das entsprechende Zählregister bei Überlauf automatisch auf 0 zurückgesetzt wird und dass der Timer mit voller

Prozessorgeschwindigkeit läuft, also mit einem Prescale-Faktor von 1.

In TIMSK ist die *Timer Interrupt Mask* definiert, in der das Bit zur Aktivierung eines Interrupts bei einem Überlauf des Timers gesetzt werden soll. Dieses Bit ist über den Namen OCIE1A definiert. Der Befehl `TIMSK |= (1 << OCIE1A)` ist das bitweise ODER der zwei Werte `TIMSK = b1 . . . bn` und `(1 << OCIE1A) = 0 . . . 010 . . . 0`, wobei die 1 an Position OCIE1A steht.

Für alle Ergebnisbits c_i mit $i \neq \text{OCIE1A}$ gilt $c_i = (b_i \vee 0) = b_i$. Diese Bits verändern sich also durch den Befehl nicht. Für das verbleibende Bit ($i = \text{OCIE1A}$) gilt $c_i = (b_i \vee 1) = 1$; es wird auf 1 gesetzt.

Um die mangelnde Genauigkeit des Timers in den Griff zu bekommen, nutzen wir den so genannten *Overflow Comparator*. Wir setzen OCR1A auf einen bestimmten Wert. Erreicht das Zählregister TCNT1 diesen Wert, so wird dies als Überlauf gewertet, d. h. das Register wird auf 0 zurückgesetzt und ein IRQ wird an die CPU gesendet. Der eingestellte Wert entspricht ungefähr der Anzahl von Rechenschritten, die die CPU in einer Millisekunde durchführen kann. Bemerkenswert ist, dass wir für eine hinreichende Genauigkeit TIMER_RES auf 990 (und nicht wie anzunehmen auf 1000) setzen mussten. Diese Zahl wurde im Experiment bestimmt.

Zu guter Letzt muss noch der Interrupt OCIE1A eingeschaltet werden, damit wir einen entsprechenden Überlauf auch tatsächlich bemerken.

3.2.3 Hauptschleife

Bei der Hauptschleife zeigt sich die systemspezifische Optimierung. Es wird eine `for(;;)`-Schleife statt dem üblichen `while(1)` eingesetzt, denn sie produziert kürzeren Maschinencode [Atm03].

In der Schleife wartet das Programm auf ein Byte vom UART und wertet dieses entspre-

chend aus. Die Schleife wird dabei niemals verlassen, denn das Programm soll endlos laufen – es gibt kein Betriebssystem, an das die Kontrolle nach Programmende übergeben werden könnte.

Die Hauptschleife befasst sich offensichtlich nur mit Steuerbefehlen, also solchen, die Parameter wie die Impulslänge ändern oder aber Informationen anfordern. Die Generierung des Impulses an sich wird vollständig vom Timersystem übernommen.

3.2.4 Die Struktur `time`

Zugriffe auf globale Variablen können besser optimiert werden, wenn diese Variablen in einer Struktur zusammengefasst und über einen Zeiger angesprochen werden. Dazu betrachten wir folgende Struktur:

```
typedef struct {
    int a;
    int b;
    int c;
} global;
```

Wird eine Instanz dieser Struktur angelegt, z. B. `global g`, so liegt `g.a` im Speicher an der Adresse `&g`. Direkt im Anschluss liegen `g.b` und `g.c`. Einen Zugriff auf `g.c` kann der Compiler folglich als `*((int*)&global + 2)` ausdrücken, also mit einem Offset auf die Basisadresse `&g`. Dies schafft effizienteren Programmcode, da pro Instanz lediglich eine Basisadresse als Variable gehalten werden muss – die Offsets sind Konstanten.

Wir setzen genau diese Technik ein, um Zugriffe auf die Variablen für Impulsintervall, Impulsbreite und die Anzahl der verstrichenen Millisekunden zu konsolidieren.

Listing 3: Die Interrupt Service Routine

```
268 SIGNAL (SIG_OUTPUT_COMPARE1A)
269 {
270     time* t = &g_t;
271
272     // Stop timer.
273     TCCR1B &= ~(1 << CS10);
274
275     t->elapsed++;
276
277     // We need to generate a peak.
278     if (t->elapsed >= t->interval) {
279         // Set PortB1 to high and turn on LED.
280         sbi(PORTB, PB1);
281         cbi(PORTB, PB2);
282
283         // Wait a little.
284         delay_us(t->peak);
285
286         // Set PortB1 back to low and turn off LED.
287         cbi(PORTB, PB1);
288         sbi(PORTB, PB2);
289
290         // Make up for the wait in our milliseconds counter.
291         uint16_t rem = t->peak % 1000;
292         t->elapsed = (t->peak - rem) / 1000;
293
294         // Reset timer appropriately.
295         TCNT1 = (rem * CYCLES_PER_MICROSEC) + ISR_CYCLES;
296     }
297
298     // Start timer again.
299     TCCR1B |= (1 << CS10);
300 }
```

3.2.5 Die Interrupt Service Routine

Die vollständige Interrupt Service Routine ist in Listing 3 dargestellt. Zunächst wird die Adresse der globalen Zeit-Struktur `g_t` in einer entsprechenden Variable gespeichert. Anschließend wird der Timer angehalten. Dies geschieht, in-

dem der schon erwähnte *Prescale*-Faktor auf Null gesetzt wird und ist aus folgendem Grunde wichtig:

Wird der Timer nicht angehalten, so zählt er beständig im Hintergrund weiter. Läuft er dabei über, so wird ein IRQ erzeugt und direkt

nach Ablauf der aktuellen ISR behandelt – an sich ein erwünschtes Verhalten. Läuft der Timer allerdings mehrfach während der ISR über, so wird trotzdem *nur ein* IRQ erzeugt. Alle anderen Überläufe, die jeweils einer verstrichenen Millisekunde entsprechen, geraten in Vergessenheit, womit die Zeitmessung also sehr ungenau wird.

Die offensichtliche Lösung des Problems ist es, dies nachträglich zu kompensieren, indem die Variablen `g_t.elapsed` und `TCNT1` entsprechend eingestellt werden. Um weitere Komplexität zu vermeiden (Test, ob ein Überlauf stattgefunden hat), wird der Timer ganz abgeschaltet und die volle Dauer der Routine bei der Zeitberechnung mit einbezogen.

Nun wird die Variable `g_t.elapsed` inkrementiert. Dies geschieht bei jedem Aufruf der ISR, was wiederum ungefähr mit einer Frequenz von 1000Hz passiert. In `g_t.elapsed` werden also die verstrichenen Millisekunden gezählt. Erreicht dieser Wert den von `g_t.interval`, so ist die eingestellte Intervalldauer abgelaufen und ein Rechtecksignal muss auf Port B1 erzeugt werden.

Dazu wird der Port zunächst auf 1 gesetzt (die LED wird als visuelle Rückmeldung ebenfalls angeregt). Dann wartet der Mikrocontroller die eingestellte Impulsbreite, woraufhin Port B1 wieder auf 0 zurückgesetzt wird.

Nun folgt, wie schon erwähnt, die Berechnung der verbrauchten Zeit, wobei die Warteschleife im allgemeinen Fall den größten Posten stellt. Die Zählvariable `g_t.elapsed` sowie das Zählregister von `timer1`, `TCNT1`, werden entsprechend eingestellt.

Am Ende jeder ISR muss natürlich der Timer weiterlaufen. Dazu wird der *Prescale*-Faktor wieder auf 1 gesetzt.

3.2.6 Die Verzögerungsprozedur `delay_us()`

Für die Erzeugung der Impulsbreite (s. o.) benötigen wir eine Prozedur, die eine ungefähr mikrosekundengenaue Verzögerung durchführen kann. Atmel stellt solch eine Prozedur zur Verfügung: `_delay_us()`. Der Nachteil dieser Funktion ist, dass sie Fließkommaarithmetik einsetzt, was dazu führt, dass beim Linken eine entsprechende Bibliothek eingebunden wird. Der Speicher des ATmega8 ist allerdings so knapp bemessen, dass er dann unser Programm nicht mehr vollständig aufnehmen kann. Wir müssen also auf den Einsatz von Fließkommatypen verzichten. Darum wurde diese Funktionalität selbst implementiert.

`delay_us` benutzt `delay_loop_2()`, eine Bibliotheksfunktion, die viermal die angegebene Anzahl von *Ticks*, also Rechenschritte, abwartet. Dies wird intern über eine Zählschleife realisiert, die pro Durchlauf vier Instruktionen benötigt – daher auch der Faktor vier. Um die notwendige Anzahl an Ticks zu berechnen, führt `delay_us()` notwendigerweise eine Ganzzahldivision durch. Dies bedeutet natürlich einen Verlust von Genauigkeit, der allerdings im allgemeinen Fall zu vernachlässigen ist. Die Berechnungsvorschrift der Ticks lautet

$$\tau = \mu \cdot \frac{1}{4} \cdot \frac{\nu}{10^6}$$

mit μ Anzahl der Mikrosekunden, ν Prozessorfrequenz und τ Anzahl der Ticks. Dabei ist der Faktor

$$\frac{\nu}{10^6}$$

gleich der Anzahl der Rechenschritte, die die CPU in einer Mikrosekunde (10^{-6} s) durchführen kann, da ν in $\text{Hz} = \text{s}^{-1}$ angegeben ist. Der Faktor $1/4$ erklärt sich durch die Verwendung von `delay_loop_2()` (s. o.).

Listing 4: Die Verzögerungsprozedur.

```

540 void delay_us(uint16_t us) {
541     uint32_t ticks = (uint32_t)us;
542
543     // CAUTION: This is _integer_ division.
544     ticks *= TICK_FACTOR;
545     ticks /= TICK_DIVISOR;
546
547     // Subtract overhead.
548     if (ticks < DELAY_OVERHEAD) {
549         return;
550     }
551     ticks -= DELAY_OVERHEAD;
552
553     _delay_loop_2((uint16_t)ticks);
554 }

```

Um die Nachteile der Ganzzahldivision auszugleichen, stellen wir die Berechnungsreihenfolge explizit um:

$$\tau = \frac{\mu \cdot \nu}{4 \cdot 10^6}$$

Dies führt dazu, dass zunächst μ und ν multipliziert werden, wobei ein großer Wert entsteht, und dann durch $4 \cdot 10^6$ geteilt wird. Ein Zahlenbeispiel macht deutlich, warum dies sinnvoll ist:

$$\tau = 500 \cdot \frac{14745600}{4 \cdot 10^6} = 500 \cdot 3 = 1500$$

Dieses Beispiel zeigt ein Ergebnis, das in gleicher Reihenfolge von einem C-Programm errechnet wurde. Die Division ist offenbar für genaue Berechnungen unbrauchbar, wegen $14.7/4 = 3.675$ liegt ein enormer Rundungsfehler vor. Dieser vergrößert sich massiv durch die anschließende Multiplikation mit der Impulsbreite. Der endgültig berechnete Wert weicht darum stark vom genauen Wert 1843.2 ab. Es fällt außerdem auf, dass der Bruch in der Formel gekürzt werden kann:

$$\frac{14745600}{4 \cdot 10^6} = \frac{36864}{10^4}$$

Dies verursacht keine Genauigkeitsverluste.

Ein zweites Beispiel ändert die Berechnungsreihenfolge und benutzt den gekürzten Bruch:

$$\tau = \frac{500 \cdot 36864}{10^4} = \frac{18432000}{10^4} = 1843$$

Dieser Wert liegt zufriedenstellend nahe am exakt berechneten. Werden die Werte für alle Impulsbreiten im zugelassenen Bereich ($0-17000\mu s$) berechnet, so liegt die mittlere Abweichung vom genauen Wert mit unserem Verfahren bei 0.5 Ticks ($\simeq 1.4\mu s$), mit dem "naiven" Verfahren allerdings bei 5834.7 Ticks – das entspricht ungefähr 1.6 Millisekunden!

Um das Kürzen des Bruches (s. o.) zu begründen sei erwähnt, dass der verwendete Datentyp 32 Bit breit ist. Durch das Kürzen wird erreicht, dass die Variable `ticks` bei der Multiplikation mit $\mu \in \{1, \dots, 17000\}$ nicht überläuft.

Die Konstanten `TICK_FACTOR` und `TICK_DIVISOR` stellen genau den gekürzten Bruch dar – so wird zusätzlich Rechenaufwand gespart.

Die zuletzt beschriebene Optimierung ist natürlich hochgradig abhängig von der benutzten Prozessorfrequenz und muss bei Änderung dieser u. U. entsprechend angepasst werden.

Falls der Aufruf von `delay_us()` selbst schon mehr Zeit verbraucht hat, als gewartet werden soll, bricht die Prozedur nun ab. Ansonsten wird eine entsprechende, experimentell bestimmte Konstante von der Wartezeit abgezogen und die Wartefunktion wird aufgerufen.

Durch die Obergrenze von $17000\mu\text{s}$ für die Impulsbreite wird sichergestellt, dass der resultierende Wert mit 16 Bit dargestellt werden kann:

$$17000 \cdot \frac{14745600}{4 \cdot 10^6} = 62668.8 < 65536 = 2^{16}$$

Dies ist wichtig, da `_delay_loop_2()` nur mit solchen Werten arbeitet.

4 Installation

Für die Installation wird ein PC mit MS WindowsTM oder GNU/LinuxTM benötigt. Unter GNU/LinuxTM müssen verschiedene Kommandozeilen-Werkzeuge benutzt werden. Unter MS WindowsTM erfolgt die Installation recht komfortabel über eine grafische Benutzeroberfläche. Aus Gründen der Einfachheit wird im Folgenden lediglich die Installation unter MS WindowsTM beschrieben. Nach der Installation kann der Trigger mit jedem Betriebssystem angesteuert werden, das ein Terminal-Programm zur Verfügung stellt.

4.1 Benötigte Software

Für die Kompilierung des Trigger-Quelltextes und der anschließenden Installation auf dem μC wird die Software AVR-Studio[®] [Atm08] empfohlen. AVR-Studio[®] unterstützt *out of the box* nur die Sprache Assembler. Damit AVR-Studio[®] auch C-Quelltexte unterstützt,

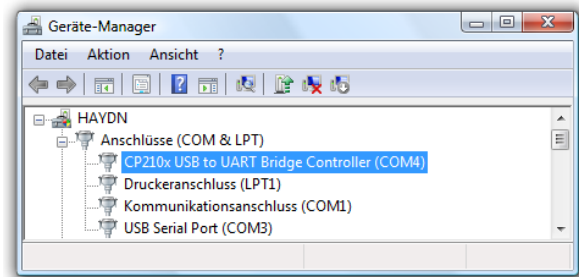


Abbildung 4: Windows-Geräte-Manager mit Programmierer auf COM3 und μC auf COM4.

ist weiterhin der GNU C-Compiler (GCC) nötig. Unter MS WindowsTM lässt sich dieser am leichtesten über das Softwarepaket WinAVRTM [WW08] installieren.

Weiterhin wird der Treiber für das Crumb8-USB-Modul [Chi08b] benötigt und für WindowsTM-Versionen kleiner Vista der Treiber für den USB-Serial-Port des Programmiers [Chi08a].

Zuletzt wird noch ein Terminalprogramm zur Steuerung des μC über die serielle Schnittstelle benötigt. Hierfür wird das freie Programm PuTTY [TDHN08] vorgeschlagen. Die Steuerung des μC wird im Folgenden deshalb auch anhand von PuTTY erläutert.

4.2 Erste Konfiguration des μC -Moduls

Wird das Trigger-Modul das erste Mal per USB angeschlossen, fragt WindowsTM nach den Treibern. Am besten steckt man auch gleich den Programmierer an den Rechner. Damit werden direkt auch die Treiber für diesen installiert. Letzteres funktioniert bei Vista[®] automatisch. Im Geräte-Manager sollten anschließend beide Geräte – wie in Abb. 4 dargestellt – auftauchen.

Danach sollte man den Programmierer mit dem μC -Modul verbinden. Man braucht hierbei

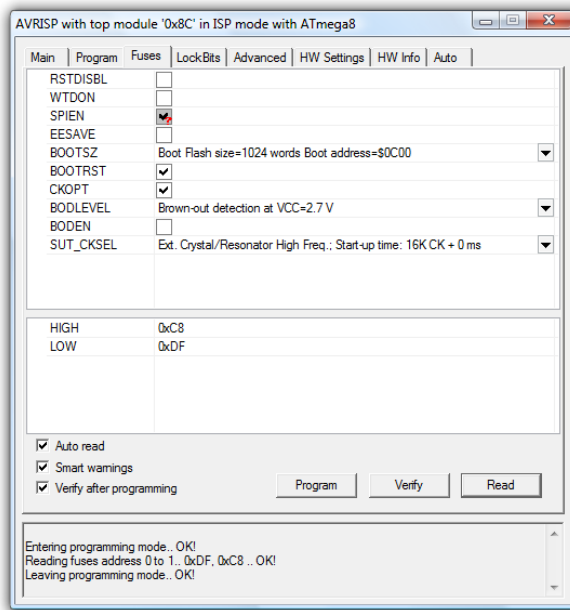


Abbildung 5: Setzen der Fuse-Bits unter AVR-Studio®.

nicht befürchten, den Programmer falsch herum anzuschließen. Der Programmer signalisiert einen falschen Anschluss durch eine blinkende LED. Ist er korrekt angeschlossen, leuchtet die LED konstant.

Öffnet man nun AVR-Studio®, so wird gefragt, ob man ein neues Projekt erstellen oder ein existierendes Projekt laden möchte. Den Dialog kann man ohne Bedenken abbrechen.

Man wählt unter *Tools*→*Program AVR*→*Auto Connect* und wählt als *Device* den *ATmega8*. Im Statusbereich sollten mehrere Meldungen mit *OK* erscheinen.

Als nächstes sind im Tab *Fuses* die Fuses für High und Low folgendermaßen zu setzen (vgl. Abb. 5) [Häm07]:

- High: 0xC8
- Low: 0xDF

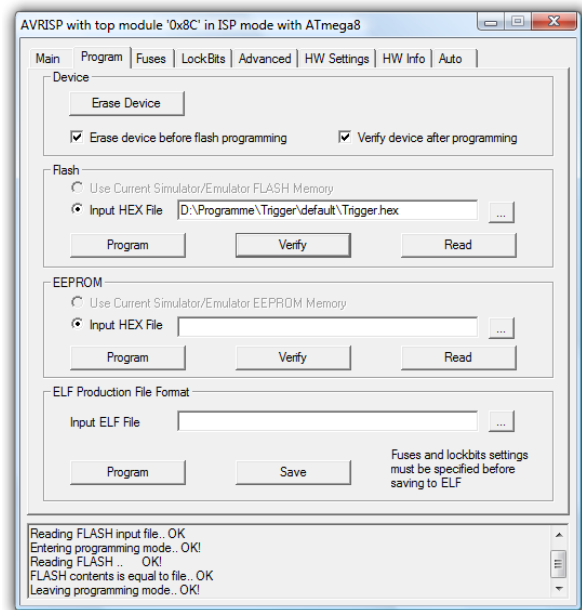


Abbildung 6: Übertragung der Firmware auf μ C.

Diese Einstellungen sind durch *Program* auf den μ C zu übertragen und dann mit *Verify* zu überprüfen. Damit ist die initiale Konfiguration des μ C-Moduls abgeschlossen. Es fehlt nun lediglich noch die Übertragung der Trigger-Firmware auf den μ C.

4.3 Installation der Trigger-Firmware

Auf den Webseiten der Go Hu.MAN⁴ findet sich das Archiv *Trigger1.0.zip*, welche ein komplettes Projekt für AVR-Studio® enthält. Das Archiv ist zu entpacken und die Projektdatei mit AVR-Studio® zu öffnen. Im Projekt ist eine fertige HEX-Datei als Firmware vorhanden. Diese kann unter *Tools*→*Program AVR*→*Auto Connect* auf der Registerkarte *Program* auf den μ C geschrieben werden (vgl. Abb. 6). Der

⁴<http://hu-man.ira.uka.de>

Schreibvorgang sollte mit *Verify* überprüft werden.

Sollen Veränderungen am Sourcecode vorgenommen werden, muss anschließend ein *Build* durchgeführt werden und die neu erzeugte HEX-Datei, wie oben beschrieben, erneut übertragen werden.

4.4 Steuerung des Triggers über Konsole

Ist das μ C-Modul ordnungsgemäß eingerichtet und an den Rechner per USB angeschlossen, sollte es im Gerätemanager – wie in Abbildung 4 dargestellt – auftauchen. Zu beachten ist, dass das μ C-Modul auch auf anderen Ports als COM4 liegen kann. Die folgenden Schritte sind daher ggf. an andere Ports anzupassen.

Nachdem der COM-Port des μ C-Moduls (hier COM4) bestimmt wurde, kann PuTTY gestartet werden. Es öffnet sich der PuTTY-Konfigurationsdialog (vgl. Abb. 7) mit den Session-Einstellungen. Hier wählt man als *Connection Type* die Einstellung *Serial* und trägt als *Serial Line* den COM-Port des μ C und als *Speed* 57600 ein.

Anschließend wechselt man in die Terminal-Einstellungen (vgl. Abb. 8). Für die korrekte Darstellung von Zeilenumbrüchen aktiviert man hier die Checkbox *Implicit CR in every LF*.

Danach wird über die Schaltfläche *Open* die Verbindung zum μ C hergestellt. Man landet vor einem leeren Terminalfenster. Drückt man die Taste *v*, meldet sich der Trigger mit seiner Versionsnummer. Mit der Taste *h* erreicht man die Hilfe, die weitere mögliche Befehle zur Einstellung von Triggerimpulsfrequenz und -impulslänge auflistet (vgl. Abb. 9).

Die eingestellte Frequenz und Länge wird auf den Ports B1 und B2 geschaltet (vgl. Abb. 1), d. h. das Signal wird sowohl auf Pin 10 ausgegeben als auch die Status-LED entsprechend als optische Rückgabe geschaltet.

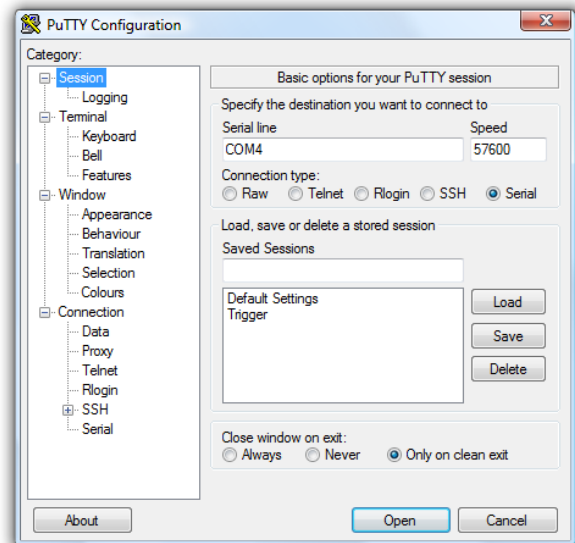


Abbildung 7: Unter *Session* das Feld *Serial* aktivieren, dann *Serial Line* und *Speed* festlegen.

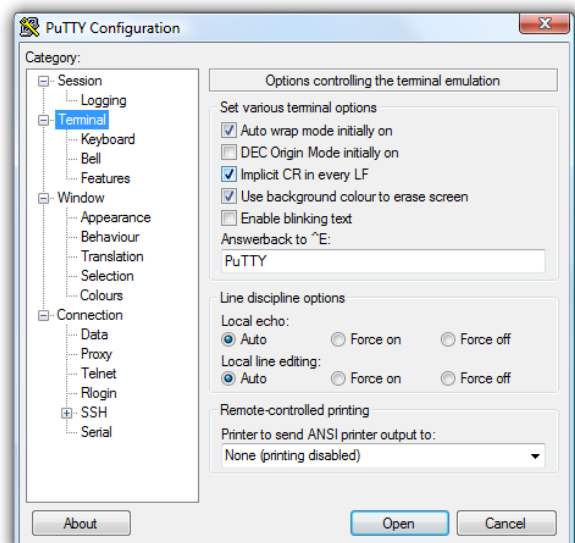
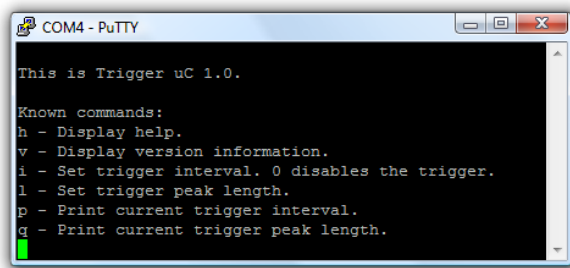


Abbildung 8: Unter *Terminal* Aktivierung des Felds *Implicit CR in every LF*.



```

COM4 - PuTTY
This is Trigger uC 1.0.

Known commands:
h - Display help.
v - Display version information.
i - Set trigger interval. 0 disables the trigger.
l - Set trigger peak length.
p - Print current trigger interval.
q - Print current trigger peak length.

```

Abbildung 9: Steuerung des μC über das Terminalprogramm PuTTY.

Angenommen, man möchte 25 Bilder/Sekunde mit einer getriggerten Kamera aufnehmen. Mit der Taste `i` ist das Triggerintervall auf 40ms zu setzen. Damit man die LED gut erkennen kann, setzt man die Signallänge mit der Taste `l` willkürlich auf $15000\mu\text{s}$ (15ms). Es ist eine schnell blinkende LED auf dem μC -Modul zu sehen.

5 Evaluation

Die Funktionsweise des μC -Triggers wurde anhand eines 100MHz-Oszilloskops vom Typ *Tektronix 2235* für verschiedene Frequenzen zwischen 1-200Hz und Signallängen von 1-17000 μs evaluiert. Innerhalb dieser Bereiche konnte die erwartete Funktionalität bezüglich der gelieferten Signale am Ausgang des μC -Triggers festgestellt werden.

6 Danksagung

Wir danken der *Arbeitsgruppe Aktives Sehen* der Universität Koblenz-Landau für die Beratung und die Einführung in die μC -Programmierung.

Diese Forschungsarbeit der Arbeitsgruppe *Group on Human Motion Analysis* wurde durch das Ministerium für Wissenschaft,

Forschung und Kunst, Baden-Württemberg unterstützt.

Literatur

- [Atm03] ATMEL CORPORATION: *AVR035: Efficient C Coding for AVR*. <http://www.chip45.com/download/doc1497.pdf>, Abruf: 10.03.2008, 2003.
- [Atm07] ATMEL CORPORATION: *ATmega8(L) Datasheet, revision S*. <http://www.chip45.com/download/doc2486.pdf>, Abruf: 10.03.2008, 2007.
- [Atm08] ATMEL CORPORATION: *AVR Studio 4*[®]. http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725, Abruf: 16.04.2008, 2008.
- [Chi08a] CHIP45: *CrispAVR-USB*. <http://www.chip45.com/index.pl?page=CrispAVR-USB&lang=de>, Abruf: 16.04.2008, 2008.
- [Chi08b] CHIP45: *Crumb8-USB*. <http://www.chip45.com/index.pl?page=Crumb8-USB&lang=de>, Abruf: 16.04.2008, 2008.
- [Häm07] HÄMMERLING, MARK: *AVR Fuse Calculator*. <http://palmavr.sourceforge.net/cgi-bin/fc.cgi>, Abruf: 17.04.2008, 2007.
- [Sch08] SCHWARZ, ANDREAS: *Mikrocontroller.net*. <http://www.mikrocontroller.net/>, Abruf: 17.04.2008, 2008.
- [TDHN08] TATHAM, SIMON, OWEN DUNN, BEN HARRIS und JACOB NEVINS: *PuTTY: A Free Telnet/SSH Client*. <http://www.chiark.greenend.org.uk/~sgtatham/putty/>, Abruf: 16.04.2008, 2008.
- [WW08] WEDDINGTON, ERIC und JOERG WUNSCH: *WinAVRTM*. <http://sourceforge.net/projects/winavr>, Abruf: 16.04.2008, 2008.