Technische Universität Berlin
Fakulät IV
Institut für Energie und Automatisierungstechnik
Fachgebiet Elektronische Mess- und Diagnosetechnik

Praktikum Messdatenverarbeitung
Betreuer: José-Luis Bote-Garcia
SS2018

Versuchsprotokoll 1

Auslesen eines ADU auf einem Mikrocontroller

Serdar Gareayaghi (374183)
Ongun Türkcüoglu (371690)
Onur Akdemir (375959)
Marjan Chowdhury (344675)

24. Mai 2018

Inhaltsverzeichnis

1 Einleitung

Der Ziel dieses Praktikum ist, Messdaten mit Hilfe einer Mikrocontroller aufzunehmen. Der verwendete Mikrocontroller ist ein Atmega1281 mit einem integrierten Analog-Digital-Wandler. Für die Aufnahme der Messdaten muss der Analog-Digital-Wandler initialisiert und nach jeder Analog-Digital-Wandlung muss ein Interrupt ausgelöst. In dieser Interrupt-Routine werden umgewandelte Daten in den Sende-Puffer auf dem Mikrocontroller geschrieben. Der Sende-Puffer wird in den nächsten Praktikum für die Bearbeitung der Daten in Matlab benutzt.

2 Praktikumsaufgaben

2.1 Aufgabe 1

```
1  #define F_CPU 7372800UL
2
3  #include <inttypes.h>
4  #include <util/delay.h>
5  #include <avr/io.h>
6  #include <avr/signature.h>
7  #include <avr/interrupt.h>
8
9
10 void controlLEDs(uint16_t value){
11   if(value >= 0 && value <= 127){
12     PORTC |= (1<<PC1) | (1 << PC4) | (1 << PC5);
13   }
14
15   else if(value >= 128 && value <= 511){
16     PORTC &=  ~((1<<PC1) | (1 << PC4) | (1 << PC5));
17     PORTC |= (1 << PC1) | (1 << PC4);
18     // PC5 red, turn on red
19   }
20
21   else if(value >= 512 && value <= 768){
22     PORTC &=  ~((1<<PC1) | (1 << PC4) | (1 << PC5));
23     PORTC |= (1 << PC1);
24     // PC1 green
25     // PC5 red, turn on red and green
```

```
26    }
27
28    else if(value >= 769 && value <= 1023){
29      PORTC &=  ~((1<<PC1) | (1 << PC4) | (1 << PC5));
30      // PC4 orange
31    }
32
33  }
34
35
36  ISR(ADC_vect){
37    uint16_t resultADC = ADC; // read ADC
38    controlLEDs(resultADC);
39    //ADCSRA |= (1 << ADSC);
40  }
41
42
43  void init(void){
44    // initialize ADC and IO
45    ADMUX |= (0 <<  MUX4) | (0 <<  MUX3) | (0 <<  MUX2) | (0 <<
        MUX1) | (0 <<  MUX0); // Single-ended input ADC0
46    ADMUX |= (1 << REFS1) | (1 << REFS0); // set internal
        reference for ADC, 2.56V
47    ADCSRA |= (1 << ADEN); // enable ADC
48    ADCSRA |= (1 << ADPS2) | (0 << ADPS1) | (1 << ADPS0); //
        ADC-prescaler set to 1/32
49    ADCSRA |= (1 << ADIE); // enable ADC interrupt
50    ADCSRA |= (1 << ADATE); // Auto-trigger enabled;
51    ADCSRB |= (0 << ADTS2) | (0 << ADTS1) | (0 << ADTS0); //
        free-running mode enabled
52  }
53
54  int main(void){
55
56    init(); // run func init()
57    sei();  // enable global interrupts
58    ADCSRA |= (1 << ADSC); // start ADC conversation
59
60    DDRC |= (1<<PC1) | (1 << PC4) | (1 << PC5);
61
62      while(1){
63      }
```

```
64 }
```

Listing 1: Code für die Vorbereitungsaufgabe für die Ansteuerung der LEDs mittels ADU

Der Listing 1 zeigt den Code für die Initializierung und die Ansteuerung der LEDs. Dabei wird der eingelesene ADU-Wert in die Funktion controlLEDs() als Parameter übergeben und die LEDs werden nach diesem ADU-Wert an- bzw. ausgeschaltet.

2.2 Aufgabe 2: adcInit()

Für die Implementierung des Analog-Digital-Umsetzers muss erstmal eine Funktion geschrieben werden, die zur Initializierung des ADUs dient. Dabei soll die Funktion folgende Anforderungen implementieren:

- Als Spannungsreferenz soll die interne 2.56V Spannung genutzt werden.
- Der Eingang ist der Kanal ADC0 im Single-Ended-Modus.
- Der ADU-Takt soll ein $\frac{1}{32}$ des CPU-Takts betragen.
- Auto-Trigger soll als der Triggermodus eingestellt werden, und die Umsetzung soll durch einen Compare-Match von Timer1 gestartet werden.

Der Code, der die Anforderungen implementieren ist im Listing 2 dargestellt.

```
1  void adcInit() {
2    cli();
3    ADMUX |= (1 << REFS1) | (1 << REFS0); // 2.56V reference
       voltage for ADC
4    ADCSRB |= (0 << MUX0) | (0 << MUX1) | (0 << MUX2) | (0 <<
       MUX3) | (0 << MUX4); // select ADC0
5    ADCSRA |= (1 << ADATE); // Auto-trigger mdoe enabled
6    ADCSRA |= (1 << ADPS2) | (1 << ADPS0); // ADC-prescaler set to
       1/32
7    ADCSRB |= (1 << ADTS2) | (1 << ADTS0); // Timer/Counter1
       Compare-Match set
8    ADCSRA |= (1 << ADIE); // enable ADC
9    ADCSRA |= (1 << ADEN); // enable ADC-Interrupt
10   //ADCSRA |= (1 << ADSC); // start ADC-conversion
11   sei();
12
13 }
```

Listing 2: adcInit()-Funktion aus adc.c

Dabei ist es zu beachten, dass manche Zuweisungen der Register redundant sind (zum Beispiel, Zeile 4, 'select ADC0'). Meistens sind Register im Atmega1281 mit null initiliasiert, und deswegen muss an sich nicht erneut eingestellt werden. Trotzdem, solche redundante Zuweisungen entsprechen auch hohe Übersichtlichkeit.

Es ist auch wichtig zu erwähnen, dass in der Zeile 2 Global-Interrupt-Flag gelöscht ist, und wiederum in der Zeile 11 gesetzt ist. Damit ist es sicher gestellt, dass während der Initializierungsphase kein Interrupts ausgelöst werden können.

2.3 Aufgabe 3: adcStart()

In der Aufgabe 3 muss eine Funktion adcStart() geschrieben werden, welche die Clock-Rate von Timer1 und den Timer-Modus einstellen soll. Außerdem soll die Funktion die Aufnahme einer einstellbaren Anzahl von Samples bei einer wählbaren Sampling-Rate realisieren. Der Code für die Funktion adcStart() ist im Listing 3.

```
1  void adcStart(uint16_t sampleRateCode, uint32_t sampleCount,
     trigger_t triggerMode, int16_t triggerLevel) {
2    cli();
3    uint32_t sampleCountGlobal = sampleCount; // global Var for
```

```
    sampleCount
4   OCR1A = sampleRateCode; // set Compare-Match to var
    sampleRateCode
5   TCNT1 = 0;          // start Timer at 0
6   TIMSK1 = (1 << OCIE1A) | (1 << OCIE1B); // enable
    Timer-Compare-Match Interrupt
7   TCCR1A = 0; // Output-Compare disable
8   TCCR1B |= (1 << CS10); // Timer prescaler set to CPU-clock
9   TCCR1B |= (1 << WGM12); // CTC-Modus set
10  counter = 0;
11  sei();
12
13 }
```

Listing 3: adcStart()-Funktion aus adc.c

Die Variablen 'sampleRateCode' und 'counter' sind am Anfang des Codes als global Variable initialisiert, weil diese Variablen anschließend in der Interrupt-Routine gebraucht werden.

```
1 uint32_t counter = 0;
2 uint32_t sampleCountGlobal = 0;
```

Listing 4: Globale Variable für die adcStart()-Funktion

2.4 Aufgabe 4: Interrupt-Routine

Die Interrupt-Routine in adc.c wird jedesmal angerufen, wenn eine Analog-Digital-Umsetzung durchläuft. Die Zeile 3 inkrementiert die globale Variable 'counter' um 1, und die Zeilen 5 bis 6 speichern den digitalen Wert in eine Variable ein bzw. den Offset von 512 Bits realisieren. Dabei ist es wichtig, dass der Register 'ADC' aus den einzelnen Registern 'ADCL' und 'ADCH' besteht und erleichtert die Aufnahme des umgesetzten Wertes, welcher eine Länge von 10 Bit hat.

Die Zeilen 9 - 14 realisieren die If-Bedingung, für wenn die Aufnahme der gewünschen Anzahl des Samples erfolgreich abgeschlossen ist. Anschließend wird der Timer1 ausgeschaltet und die globale Variablen 'counter' und 'sampleCountGlobal' werden wieder null gesetzt. Damit ist es möglich, eine neue Umsetzung anzufangen. Der Code ist im Listing 5 dargestellt.

```
1  ISR(ADC_vect) {
2
3    counter++;
4
5    uint16_t valADC = ADC;    // alternatively read result from ADC
6    uint16_t valADC_offset = valADC - 512;
7    filterWrite2Buf(valADC_offset);
8
9    if(counter == sampleCountGlobal){
10     ADCSRA &= ~(1 << ADIE); // ADC-Interrupt turn off
11     TCCR1B &= ~((1 << CS10) | (1 << CS11) | (1 << CS12)); //
           turn off Timer
12     counter = 0;
13     sampleCountGlobal = 0;  // both set to 0
14   }
15
16 }
```

Listing 5: Interrupt-Routine

2.5 Aufgabe 5: adcIsRunning()

Die relativ übersichtliche Funktion adcIsRunning() liefert eine 1, wenn der ADU weitere Samples aufnehmen muss und liefert eine 0, wenn die gewünschte Anzahl an Samples erreicht sind.

```
1  uint8_t adcIsRunning() {
2    if (counter == sampleCountGlobal) {
3      return 0;
4    }
5    else {
6    return 1;
7    }
8  }
```

Listing 6: adcIsRunning()-Funktion aus der adc.c

Der gesamte Code kann mit dem folgenden Link angeschaut werden, bzw. es wird mit dem Protokoll mitgeliefert: `https://github.com/onguntoglu/messdatenverarbeitung/blob/master/pr1/praktikum_end/adc.c`

3 Abgabeaufgaben

Die folgende Fragen sind mit Hilfe des Datenblatts für den Mikrocontroller Atmega1281 geantwortet.

3.1 Aufgabe 1

Der AVR-Mikrocontroller verwendet die Methode der sukzessiven Approximation. Dabei representiert der minimale Wert die Masse (GND) und der maximale Wert entspricht die Spannung am AREF minus 1 LSB. Durch binäre Suche der Quantisierungsniveaus wird das analoge Signal zu einem digitalen Signal umgewandelt.

3.2 Aufgabe 2

Der ADU von dem AVR-Mikrocontroller lässt sich im allgemeinen mit 6 Registern einstellen:

- ADMUX (ADC Multiplexer Selection Register): beinhaltet die Register für die Referenzspannungseinstellungen, Left-Adjust-Einstellung, Analog Channel and Gain Selection Bits
- ADCSRB (ADC Control and Status Register B): Register für die Einstellungen der Eingängen des ADUs und Auto-Trigger-Source
- ADCSRA (ADC Control and Status Register A): Register für ADC Enable, ADC Start Conversion, ADC Auto Trigger Enable, ADC Interrupt Flag, ADC Interrupt Enable, ADC Prescaler Select Bits
- ADCL and ADCH (The ADC Data Register): Die Register, in dem die Ergebnisse der Umsetzungen gespeichert sind.
- DIDR0 (Digital Input Disable Register 0)
- DIDR2 (Digital Input Disable Register 2)

3.3 Aufgabe 3

Die Abtastrate wird durch den Register ADPS2, ADPS1 und ADPS0 (vom ADCSRA) eingestellt. Die erlaubte Abtastraten sind Faktoren vom CPU-Takt und es kann folgende Abtastrate

eingestellt werden: $\frac{1}{2}F_{CPU}$, $\frac{1}{4}F_{CPU}$, $\frac{1}{8}F_{CPU}$, $\frac{1}{16}F_{CPU}$, $\frac{1}{32}F_{CPU}$, $\frac{1}{64}F_{CPU}$ und $\frac{1}{128}F_{CPU}$. Daraus folgt, dass die maximale Abtastrate für den ADU ist die Hälfte des CPU-Takts.

3.4 Aufgabe 4

Die Spannungsquellen für die Referenzspannung für den ADU ist durch Einsetzen der Bits REFS1 und REFS0 vom Register ADMUX einstellbar. Dabei ist es wichtig, dass wenn eine externe Referenzspannung verwendet wird, soll die gewünsche Referenzspannung an dem AREF-Pin liegen. Folgende Refenrenzspannungen sind erlaubt:

- Spannung am AREF, interne Referenzspannung ausgeschaltet.
- AVCC mit externer Kapazität am AREF
- interne Referenzspannung 1.1V, externe Kapazität am AREF
- interne Referenzspannung 2.56V, externe Kapazität am AREF

3.5 Aufgabe 5

Der kommentierte Code ist mit dem Protokoll abgegeben, bzw. kann unter dem Link `https://github.com/onguntoglu/messdatenverarbeitung/blob/master/pr1/praktikum_end/adc.c` angeschaut werden.

4 Auswertung und Zusammenfassung

In diesem Versuch wurde der Analog-Digital-Umsetzer von dem AVR-Mikrocontroller mit dem oben genannten Anforderungen (siehe 2.2) initializiert. Anschließend ist die Aufnahme von Samples mit der Funktion adcStart() und mit der Interrupt-Routine realisiert.