BERGISCHE UNIVERSITÄT WUPPERTAL
FACHBEREICH C - PHYSIK
ELEKTRONIKPRAKTIKUM

Version 12.19, TEX: 10. November 2020

Versuch EP10

Digitalelektronik
Teil 3:
Mikrocontroller

I. Zielsetzung des Versuches

Nachdem Sie in den letzten beiden Versuchen einfache Digital-ICs und programmierbare Logikbausteine kennengelernt haben, soll es im heutigen Versuch um Mikrocontroller und deren Programmierung gehen.

II. Vorkenntnisse

Grundlagen der Digitaltechnik (Versuch EP8)

Eigenschaften von und Unterschiede zwischen Logikschaltungen (CPLD, FPGA) und Mikrocontrollern.

Grundkenntnisse in C-Programmierung sollten Sie in der Vorlesung „Informatik für Physiker“ erworben haben. Ein Einführung zur C-Programmierung finden Sie auch auf der Internetseite des Elektronikpraktikums.

Die Kenntnisse zum Umgang mit der Programmierumgebung finden Sie in dieser Beschreibung.

III. Theorie zum Versuch

1. Mikrocontroller

1.1. Aufbau

Zentraler Bestandteil jedes Computers ist der Mikroprozessor. Dieser Baustein übernimmt nur die eigentliche Rechenleistung. Die Kommunikation mit der Umgebung, das Abspeichern von Daten und das Programm usw. ist auf anderen Bausteinen untergebracht.

Bei einem **Mikrocontroller** (auch μController, μC, MCU) sind aber alle Funktionen in *einem* Baustein, auf einem Chip vereinigt. Auch der Arbeits- und Programmspeicher ist dort lokalisiert.

Ein Mikrocontroller ist daher praktisch ein Ein-Chip-Computersystem.

Moderne Mikrocontroller haben häufig auch komplexe Peripheriefunktionen wie z.B.

- Universal Serial Bus (USB)
- CAN-Bus (Controller Area Network)
- serielle Schnittstelle (RS232)
- Schnittstellen für andere Peripheriebausteine (SPI- oder IIC- bzw. TWI-Bus)
- Ethernet-Schnittstellen
- Pulsbreitenmodulierte Ausgänge (PWM) zur Erzeugung analoger Spannungen
- Analog-Digital-Wandler (ADC) zur Messung analoger Spannungen
- Schnittstellen für Flüssigkristall-Anzeigen (LCD-Controller und -treiber)

1.2. Unterschiede zwischen programmierbarer Logik und Mikrocontrollern

Programmierbare Logik, insbesondere die FPGAs, die Sie im letzten Versuch kennengelernt haben, arbeitet „in Echtzeit“. Die Eingangssignale werden sofort von den Gattern und Flipflops in der gewünschten Weise verknüpft und an die Ausgänge gegeben. Es können dabei sehr viele Bits gleichzeitig verarbeitet werden.

Daher sind diese Bausteine sehr schnell und können oft Signalfrequenzen von mehreren 100 MHz verarbeiten.

Komplexe Rechenfunktionen sind aber — vor allem bei den einfacheren Bausteinen — nicht oder nur mühsam zu realisieren. Bei den wesentlich komplexeren FPGAs sind auch komplexe Rechnungen möglich, sie sind aber oft teuer.

Mikrocontroller arbeiten nach dem Prinzip eines Computers. Sie arbeiten ein bestimmtes Programm *nacheinander* (sozusagen Zeile für Zeile) ab, nehmen Eingangssignale entgegen, verknüpfen und vergleichen sie, geben sie danach als Ausgangssignale aus oder speichern etwas ab.

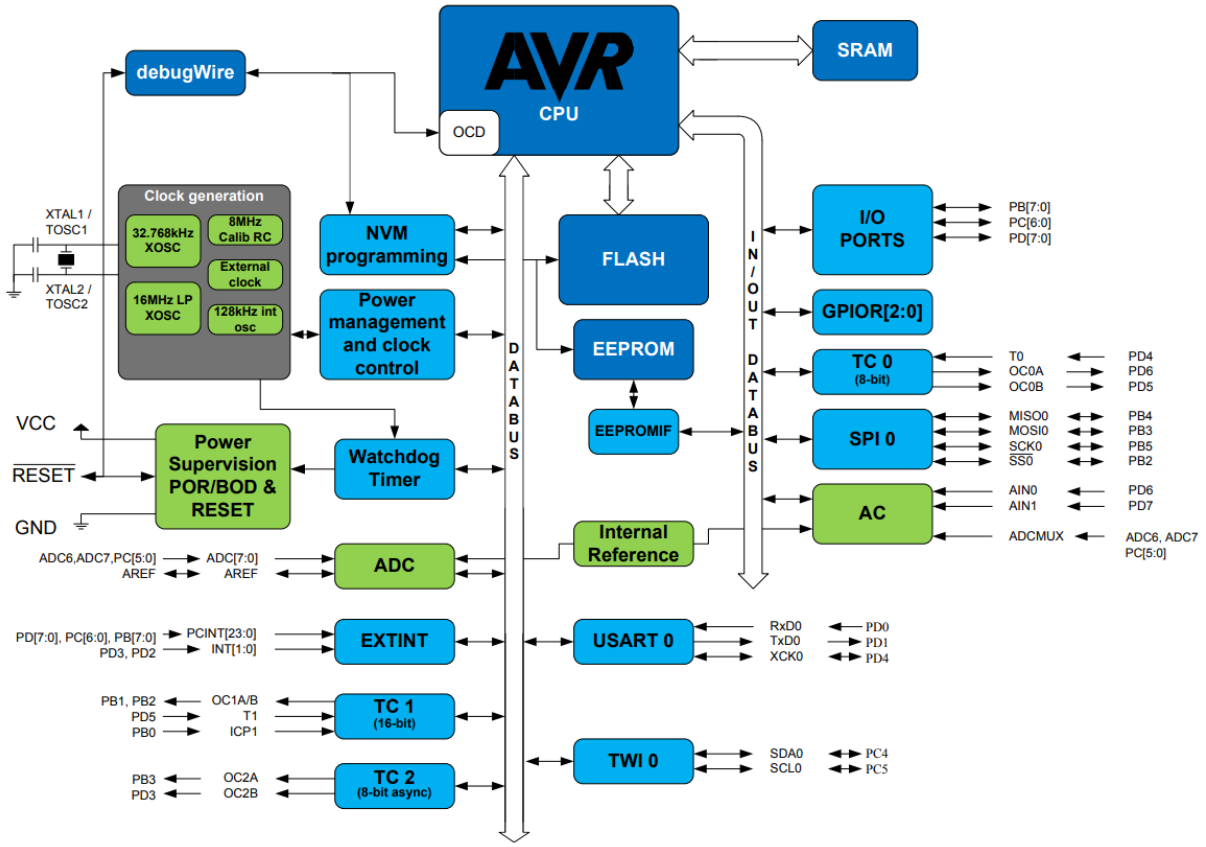
Daher sind diese Bausteine relativ langsam und können nur Signalfrequenzen von einigen MHz verarbeiten. Das liegt auch an den geringen Taktfrequenzen, die auch aus Gründen geringer Stromaufnahme oft nur bei höchstens einigen 10 MHz liegen.

Komplexe Rechenfunktionen sind aber leicht zu realisieren.

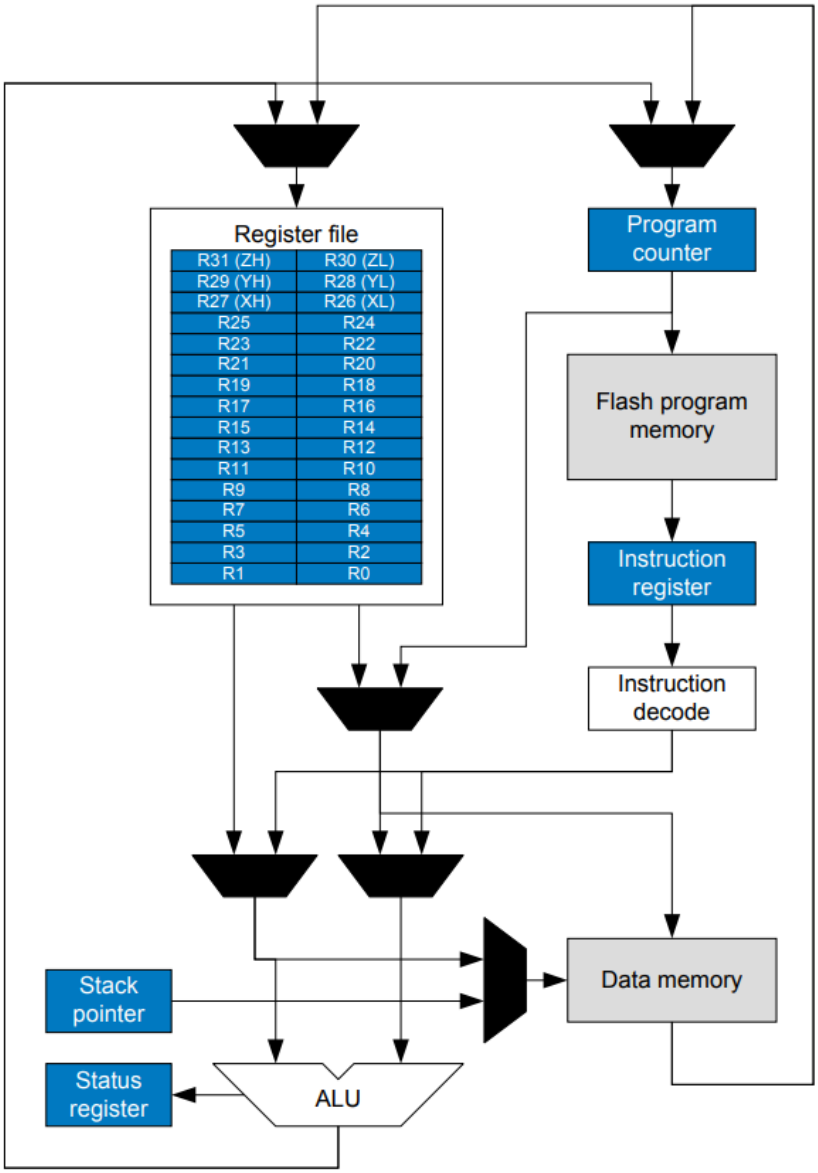
Das Programm kann aus elementaren Programmierbefehlen (d.h. im Assemblercode) aufgebaut werden. Üblich weil wesentlich einfacher und bequemer ist aber die Verwendung einer Hochsprache wie BASIC oder C.

2. Mikrocontroller am Beispiel des AVR

In diesem Praktikumsversuch werden Sie einen Mikrocontroller der AVR-Familie des Herstellers Microchip (genauer den ATMega328P) verwenden. Das Innenleben dieses Mikrocontrollers besteht aus vielen verschiedenen Peripherieeinheiten, wie z. B. Timern, Ein-/Ausgabeeinheiten usw.



Zentraler Bestandteil eines Mikrocontrollers ist der Prozessor (CPU). Der Prozessor lädt Befehle und Daten aus dem (nicht-)flüchtigen Speicher (SRAM, FLASH) und führt diese aus. Der Prozessor ist dabei noch in weitere Einheiten unterteilt.

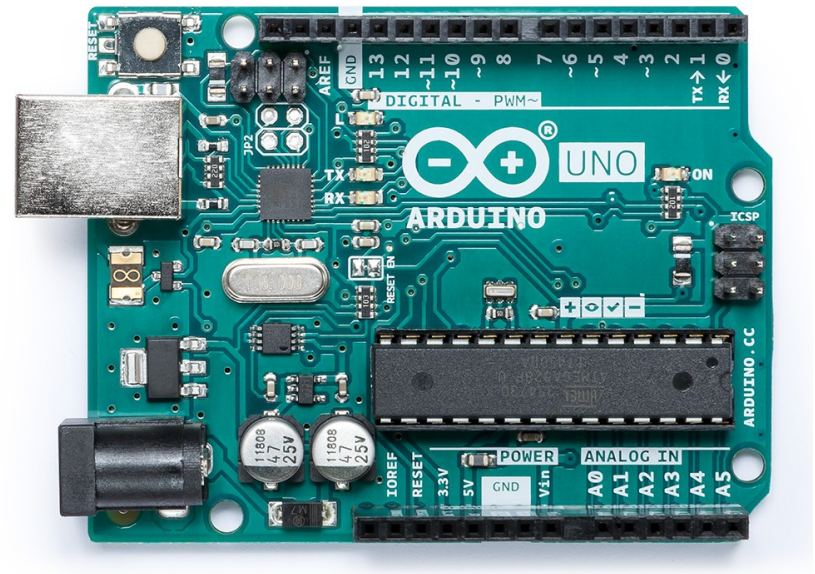


Die Register des Prozessors (engl. register file) sind Speicherelemente, die alle Daten für lokale Berechnungen vorhalten. In diese Register können Werte aus dem Speicher geladen werden und auch in diesen zurück geschrieben werden. Die ALU (arithmetisch-logische Einheit, engl. arithmetic logic unit) ist für alle Berechnungen innerhalb des Prozessors zuständig. Sie verfügt über digitale Schaltungen, um alle möglichen logischen und arithmetischen Operationen (UND, ODER, NOT, Addition, Multiplikation usw.) durchzuführen. Dabei werden immer zum Beispiel 8 Bit verarbeitet, wobei es auch Prozessoren gibt, die wesentlich mehr Bits verarbeiten können.

IV. Hinweise zum Versuchsaufbau

Zur Programmierung eines Mikrocontrollers (ATMega328P) nutzen wir eine kommerzielle Leiterplatte. Diese wurde im Rahmen des Arduino-Projektes entwickelt.

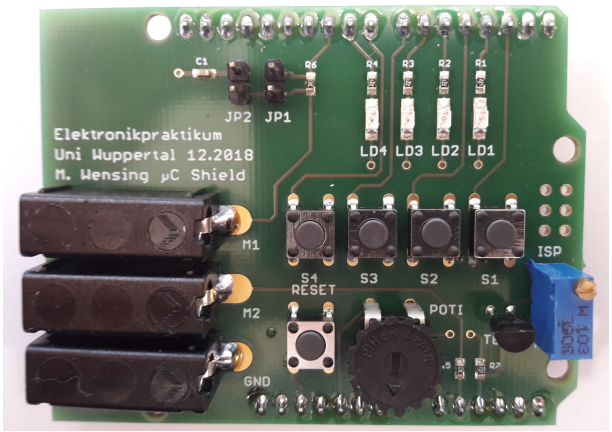
1. Arduino Uno und der EP10-Shield



Das Arduino Uno ist ein Entwicklungsboard aus der Arduino-Familie und bestückt mit einem Microchip ATMega328P Mikrocontroller. Dieser Mikrocontroller bietet Ihnen insgesamt 32 kByte an Programmspeicher und 2 kByte Arbeitspeicher. Über diverse Buchsenleisten sind einige I/O-Pins nach außen geführt. Darunter befinden sich 13 digitale Pins, die sich als Eingang und Ausgang konfigurieren lassen, sowie 6 analoge Eingänge.

Üblicherweise erfolgt die Übertragung eines neuen Programms in den Mikrocontroller über die USB-Schnittstelle. Diese implementiert eine serielle Schnittstelle zum Computer. Zusätzlich ist auf dem Arduino Uno noch eine LED (angeschlossen an Pin D13) vorhanden und ein Taster für das Auslösen eines Reset vorhanden.

Zentrale Philosophie des Arduino-Projekts ist es, dass die Entwicklungsboards nur einen minimalistischen Satz an Peripherie-Komponenten (im Fall des Arduino Uno nur die serielle Schnittstelle) bieten. Weitere Peripherie wird in der Regel über sogenannte „Shields“ zur Verfügung gestellt. Für den Versuch EP10 nutzen wir einen selbst entwickelten Shield mit 4 Tastern, 4 LEDs, einem Temperatursensor sowie einigen Ein-/Ausgängen über Bananenbuchsen.



Name	Pin	Standard pinMode
LD1	D3	OUTPUT
LD2	D5	OUTPUT
LD3	D6	OUTPUT
LD4	D9	OUTPUT
S1	D2	INPUT_PULLUP
S2	D4	INPUT_PULLUP
S3	D7	INPUT_PULLUP
S4	D8	INPUT_PULLUP
M1	D10	INPUT/OUTPUT
M2	A1	INPUT
POTI	A0	INPUT
TEMP	A2	INPUT

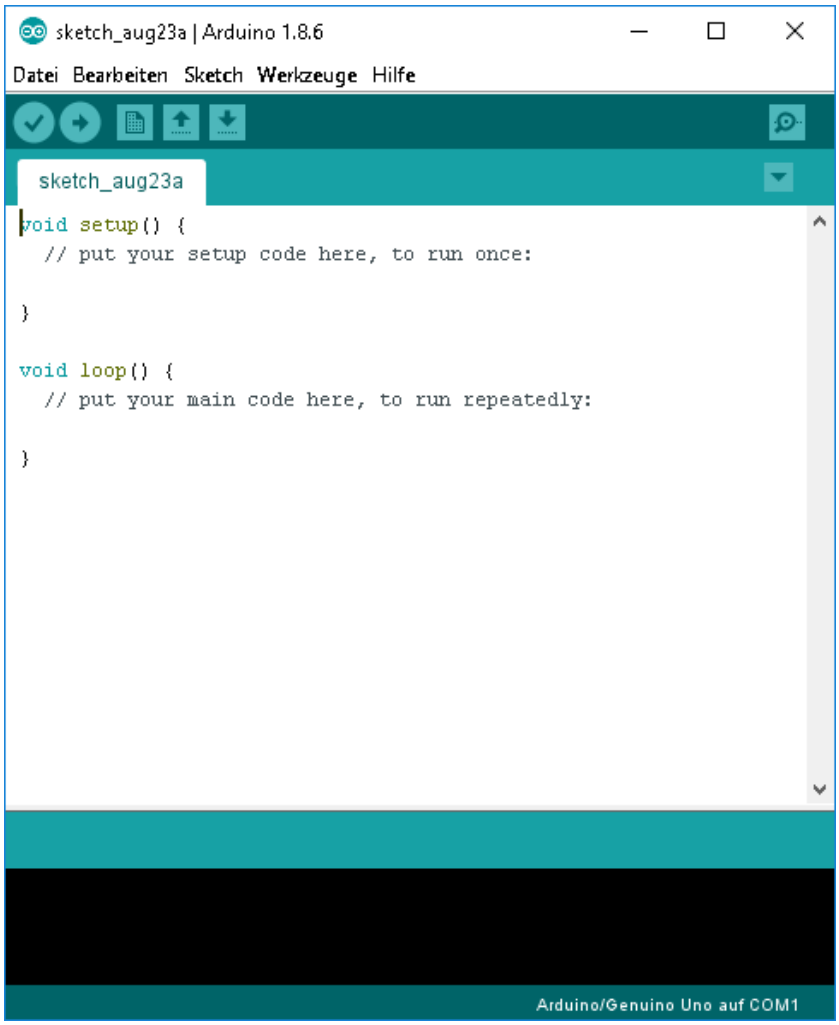
2. Arduino Entwicklungsumgebung

2.1. Öffnen der Entwicklungsumgebung

Entweder: Desktop → Arduino IDE

Oder: Start → Alle Programme → Arduino IDE

Nach dem Start sollten Sie die Benutzeroberfläche der Arduino IDE sehen:



2.2. Anlegen eines Sketches

Die Arduino-Entwicklungsumgebung bezeichnet Projekte unter dem Namen „Sketch“. Diese bestehen aus einer .ino-Datei, die den zentralen Quelltext enthält. Nach dem Anlegen eines Sketch sehen Sie eine leere .ino-Datei:

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

2.3. Aufbau eines Sketches

Wie Sie im vorherigen Abschnitt bereits kennengelernt haben, besteht ein Sketch aus einer zentralen .ino-Datei. Diese Datei hat einen C/C++-ähnlichen Syntax und muss mindestens die beiden Funktionen `setup()` und `loop()` bereitstellen. Während die `setup()`-Funktion einmalig zu Beginn ihres Programms aufgerufen wird und im Wesentlichen der Initialisierung Ihres Mikrocontrollers dient, wird die `loop()`-Funktion periodisch aufgerufen.

2.4. Kompilieren und Hochladen Ihres Sketches

Wenn Sie ihr Programm fertig geschrieben haben, muss es noch kompiliert werden. Beim Kompilieren wird Ihr Arduino-Sketch in Maschinencode übersetzt, der anschließend auf dem Prozessor ausgeführt werden kann. Der

Kompiliervorgang wird mit der Tastenkombination [Strg]+[R] oder durch Klicken des Hakensymbols in der Symbolleiste gestartet. Anschließend wird im unteren Bereich des Fensters in der Konsole Auskunft darüber erteilt, ob es Fehler beim Kompilieren des Sketches gegeben hat und wie viel Programm- und Arbeitsspeicher Ihr Programm benutzt.

Um Ihr kompiliertes Programm auf den Mikrocontroller zu übertragen, wird in der Entwicklungsumgebung die Funktion „Hochladen“ benutzt. Sie ist über die Tastenkombination [Strg] + [U] oder das „Pfeil rechts“-Symbol in der Symbolleiste erreichbar. Stellen Sie jedoch sicher, dass zum Hochladen die richtige Schnittstelle ausgewählt ist. Dazu schauen Sie im Menü „Werkzeuge“ nach, ob unter „Board“ das „Arduino/Genuino Uno“ angewählt ist. Nach dem Hochladen beginnt der Prozessor sofort mit der Ausführung Ihres Programms.

Sollten Sie in Ihrem Programm die serielle Schnittstelle des Mikrocontrollers benutzen, so bietet die Arduino-Umgebung auch ein einfaches Terminal-Programm an. Es kann über „Werkzeuge“ - „Serieller Monitor“ gestartet werden. Stellen Sie sicher, dass die korrekte Baudrate (in der Regel 9600 Baud oder 115200 Baud) eingestellt ist. Falls Sie sich unsicher sind, welche Baudrate korrekt ist, schauen Sie bitte in Ihrem Programm nach dem Aufruf `Serial.begin(x)`. Der Wert von `x` bestimmt die Baudrate.

V. Versuchsdurchführung: Einfache Programme schreiben und auf dem Mikrocontroller testen

1. Hinweise zur Protokollierung

- Geben Sie zu jedem Arbeitsschritt Messwerte und Oszilloskop-Bilder an. Sollten Sie etwas selbst programmiert haben, so geben Sie bitte ebenfalls den Quelltext Ihres Programms an.
- Kommentieren Sie ihren Quellcode bitte ausführlich, sodass Dritte ihn nachvollziehen können.

2. Messung der Arbeitsgeschwindigkeit eines Mikrocontrollers

2.1. Nutzung der Arduino-Funktionen

Kompilieren Sie mit der Arduino-IDE folgenden Sketch. Verbinden Sie nun Kanal 1 des Oszilloskops mit GND und M1. Sie sollten nach dem Start des Programms ein Rechtecksignal auf dem Oszilloskop sehen. Messen Sie die Frequenz des Signals und vergleichen Sie die gemessene Frequenz mit der Quarzfrequenz des Mikrocontrollers (16 MHz).

Hinweis: Stellen Sie sicher, dass JP1 auf dem Shield gesteckt und JP2 nicht gesteckt ist. Diese Jumper dienen zur Konfiguration des Tiefpasses vor der Buchse M1. Für die Messungen zur Arbeitsgeschwindigkeit muss der Tiefpass komplett umgangen werden, was mit dem Setzen von JP1 gewährleistet ist.

```
const int M1 = 10;

void setup() {
  // konfiguriere M1 (D10) als Ausgang
  pinMode(M1, OUTPUT);
}

void loop() {
  // schalte M1 (D10) auf logisch 1 (5V)
  digitalWrite(M1, HIGH);

  // schalte M1 (D10) wieder auf logisch 0 (0V)
  digitalWrite(M1, LOW);
}
```

2.2. Direkte Nutzung der Mikrocontroller-Register

Sie werden im vorherigen Versuchsteil festgestellt haben, dass die gemessene Frequenz weit von der Quarzfrequenz (16 MHz) entfernt ist. Dies liegt daran, dass die Aufrufe von `digitalWrite()` sehr viel Zeit in Anspruch nehmen. Daher wollen wir unser Programm nun so modifizieren, dass wir direkt auf die Register des Mikrocontrollers zugreifen. Für jeden I/O-Port des Mikrocontrollers gibt es die folgenden Register:

- **DDRx**: Data Direction Register (eine 1 hier heißt, dass der jeweilige Pin als Ausgang konfiguriert ist, sonst als Eingang)
- **PORTx**: Port Data Register (falls Eingang: eine 1 aktiviert einen internen Pull-Up-Widerstand, falls Ausgang: setzt den logischen Wert des Ausgangs)
- **PINx**: Liest den logischen Zustand (0/1) des Ports ein.

Kompilieren Sie also nun folgenden Sketch und messen Sie erneut die Frequenz des Ausgangssignals an M1.

```
void setup() {
  // konfiguriere PB3 (D10 = M1) als Ausgang
  DDRB = 0b00000100;
}

void loop() {
  // setze PB2 (D10) auf logisch 1 (5V)
  PORTB = 0b00000100;

  // setze PB2 (D10) auf logisch 0 (0V)
  PORTB = 0b00000000;
}
```

Sie werden feststellen, dass die Frequenz des Ausgangssignals nun deutlich höher ist, als im vorherigen Versuch. Das liegt daran, dass wir einen Großteil der Zeit, die für den Aufrufe von `digitalWrite()` verbraucht wurde eingespart haben. Nachteilig an diesem Programm ist jedoch, dass es deutlich schlechter lesbar ist und nicht mehr so einfach auf andere Hardware portiert werden kann. Außerdem fällt Ihnen sicherlich auf, dass das Tastverhältnis nicht 50 % ist. Das ist damit zu erklären, dass auch der Aufruf der `loop()`-Funktion Rechenzeit benötigt. Betrachten Sie jedoch die Dauer, die das Signal im Zustand logisch 1 verweilt, so ist dies die Dauer, die der Prozessor für einen Taktschritt benötigt.

2.3. Die blinkende LED

Auf dem Arduino-Shield der für das Praktikum entwickelt wurde, befinden sich insgesamt 4 LEDs, die an die Ports D3, D5, D6 und D9 angeschlossen sind. Lassen Sie eine der LEDs mit einer Frequenz von ca. einem Hertz blinken. Dazu können Sie den ersten Sketch der vorherigen Aufgabe als Vorlage benutzen und ihn an passenden Stellen um die Verzögerungsfunktion `delay(ms)`[1] erweitern.

[1] siehe Referenz/Beispiel im Anhang

3. Benutzung der Taster

3.1. Einfache Tasterabfrage

Im vorherigen Abschnitt haben Sie ja bereits gelernt, wie sich mit der Funktion `digitalWrite()` ein digitaler Ausgang setzen lässt. Nun wollen wir ausprobieren, wie sich digitale Signale in einem Programm einlesen lassen. Dazu stellt die Arduino-Umgebung die Funktion `digitalRead()` zur Verfügung. Prüfen Sie die Funktion dieses Programms:

```
const int S1 = 2;
const int LED1 = 3;

void setup() {
  // konfiguriere S1 als Eingang mit Pullup
  pinMode(S1, INPUT_PULLUP);

  // konfiguriere LED1 als Ausgang
  pinMode(LED1, OUTPUT);
}

void loop() {
  // LED1 soll den logischen Wert von S1 widerspiegeln
  digitalWrite(LED1, digitalRead(S1));
}
```

Was fällt Ihnen auf? Bedenken Sie, dass die Taster auf dem Arduino-Shield mit negativer Logik arbeiten. Das heißt, dass ein gedrückter Taster einen logischen 0-Pegel erzeugt, während ein nicht-gedrückter Taster einen logischen 1-Pegel aufweist. Modifizieren Sie das Program so, dass die LED nur leuchtet, wenn der Taster gedrückt ist.

3.2. Taster steuern Speicherfunktionen

Programmieren Sie nun ein Programm, das bei einem Tastendruck den Zustand der LED ändert:

```
const int S1 = 2;
const int LED1 = 3;
int LEDzustand = 0;

void setup() {
  // konfiguriere S1 als Eingang mit Pullup
  pinMode(S1, INPUT_PULLUP);

  // konfiguriere LED1 als Ausgang
  pinMode(LED1, OUTPUT);
}

void loop() {
  if(!digitalRead(S1)) {
    digitalWrite(LED1, LEDzustand);
    LEDzustand = !LEDzustand;
  }
}
```

Was fällt Ihnen auf, wenn Sie versuchen, den Zustand der LED mit dem Taster zu verändern? Modifizieren Sie Ihr Programm so, dass die LED ihren Zustand nicht ändert, wenn der Taster gedrückt bleibt. **Hinweis:** Sie müssen sich den Zustand des Tasters in einer globalen Variablen merken.

Sie werden möglicherweise feststellen, dass der Mikrocontroller Tastendrücke mehrfach erkennt, obwohl Sie den Taster nur einmal gedrückt haben. Dieses Phänomen nennt man Tastenprellen. Statt des sofortigen elektrischen Kontakts erzeugt das Drücken des Tasters kurzzeitig ein mehrfaches Schließen und Öffnen des Kontakts. Dies kann vom Mikrocontroller als mehrere kurz aufeinanderfolgende Tastendrücke interpretiert werden. Sie können dem entgegenwirken, indem Sie nach Erkennung des Tastendrucks eine kurze Zeit warten, bis das Prellen vorbei ist.

4. Verwendung der seriellen Schnittstelle

Der hier verwendete Mikrocontroller bietet eine so genannte serielle Schnittstelle. Mit dieser Schnittstelle und einem Terminalprogramm am PC lässt sich sehr einfach eine einfache Benutzerschnittstelle entwickeln, um Messwerte und Meldungen benutzerfreundlich darzustellen. In diesem Abschnitt wollen wir nun einige Funktionen der seriellen Schnittstelle in Kombination mit Tastern und LEDs verwenden.

4.1. Hallo Welt!

Starten Sie mit folgendem Sketch:

```
void setup() {
  // serielle Schnittstelle konfigurieren
  Serial.begin(9600);

  // einmalig Hallo Welt ausgeben
  Serial.println("Hallo Welt!");
}

void loop() {

}
```

Kompilieren Sie den Sketch und laden ihn auf Ihren Mikrocontroller hoch. Verwenden Sie nun die Funktion „Serieller Monitor“ im Menü „Werkzeuge“, um ein Terminalfenster zu öffnen. Stellen Sie sicher, dass die unten rechts die korrekte Baudrate (für diesen Sketch 9600 Baud) eingestellt ist. Sie sollten dann „Hallo Welt!“ im Terminalfenster sehen.

Erweitern Sie nun Ihr Programm, damit es nach dem „Hallo Welt!“ jede Sekunde einen Zähler hochzählt und über die serielle Schnittstelle dessen Wert ausgibt.

4.2. Verarbeitung von Eingaben

Über die serielle Schnittstelle lassen sich nicht nur bequem lesbare Ausgaben erzeugen, sondern es lässt sich auch der Programmablauf durch Benutzereingaben steuern. Wir wollen zunächst folgendes einfaches Programm betrachten:

```
const int LED1 = 3;

void menu() {
  Serial.println("t: Zustand der LED aendern");
  Serial.println("Auswahl:");
}

void setup() {
  // serielle Schnittstelle konfigurieren
  Serial.begin(9600);

  // LED als Ausgang
  pinMode(LED1, OUTPUT);

  // Menü ausgeben
  menu();
}

void loop() {
  // prüfe, ob Daten empfangen wurden
  if(Serial.available() > 0)
  {
    char empfang = Serial.read();

    // ignoriere Zeilenende
    if(!isControl(empfang))
    {
      // prüfe, welcher Befehl ausgeführt werden soll
      if(empfang == 't')
      {
        digitalWrite(LED1, !digitalRead(LED1));
      }
      else
      {
        Serial.println("Ungueltiger Befehl!");
      }

      // Menü erneut ausgeben
      menu();
    }
  }
}
```

Prüfen Sie, ob das Programm wie erwartet funktioniert. Sie sollten mit dem Befehl „t“ den Zustand der LED verändern können. Erweitern Sie das Programm, indem Sie der if-Abfrage zur Befehlsverarbeitung weitere Abfragen hinzufügen. Implementieren Sie bitte folgende Befehle:

- **1-4**: Auswahl der aktiven LED. Taste 1, wählt LED1, Taste 2 LED2 usw.
- **b**: Die aktive LED soll einige Male schnell blinken.
- **r**: Alle LEDs sollen ausgeschaltet werden.
- **s**: Alle LEDs sollen eingeschaltet werden.

5. Verwendung des internen Analog-Digital-Konverters

Hinweis: Dieser Versuchsteil nutzt bereits einige Erkenntnisse, die Sie im nächsten Versuch EP11 kennenlernen werden. Nehmen Sie diese zunächst als gegeben hin. Sie werden die Grundlagen dazu im nächsten Versuch behandeln.

Unser Arduino Mikrocontroller besitzt auch 6 analoge Eingangskanäle, die es erlauben Spannungen zwischen 0 und 5 Volt messen zu können. Ein analoger Spannungswert wird dabei in einen digitalen Wert mit 10 Bit Auflösung umgewandelt. Das heißt, dass Sie einen digitalen Wert von 0 zurücklesen, wenn sich die Spannung bei 0 V befindet, und einen digitalen Wert von 1023, wenn sich die Spannung bei 5 V befindet. Die Arduino-Entwicklungsumgebung liefert Ihnen mit `analogRead(pin)` bereits vorgefertigte Funktionen zum Einlesen analoger Werte.

5.1. Einlesen eines analogen Messwerts

Auf dem Shield befindet sich ein Potentiometer, das mit POTI bezeichnet ist. Dieses Potentiometer erlaubt es, Spannungen zwischen 0 und 5 Volt auf den ersten ADC-Kanal A0 zu geben. Diese Spannung kann dann eingelesen und auf die serielle Schnittstelle ausgegeben werden:

```
const int POTI = A0;

void setup() {
  // serielle Schnittstelle konfigurieren
  Serial.begin(9600);
}

void loop() {
  // Wert einlesen und ausgeben
  Serial.print("ADC: ");
  Serial.println(analogRead(POTI));

  // etwas warten
  delay(500);
}
```

5.2. Temperaturmessung mit dem LM335

Auf dem EP10-Shield ist ein Temperatursensor vom Typ LM335 verbaut und an A2 angeschlossen. Dieser Temperatursensor funktioniert im Grunde wie eine temperaturabhängige Zenerdiode. Wird der Temperatursensor von einem Strom durchflossen (ca. 1 mA), so gibt der Sensor eine Spannung von 10 mV pro Kelvin aus. Bei Raumtemperatur (25 °C = 298 K) sollte also etwa eine Spannung von 2,98 V zu messen sein. Diese Spannung können Sie mit dem Analog-Digital-Konverter des Arduino erfassen und sich somit ein einfaches Thermometer bauen. Wie muss die Umrechnung zwischen ADC-Wert und Temperatur aussehen? Schreiben Sie einen Sketch, der regelmäßig den Wert des Temperatursensors einliest und die gemessene Temperatur auf der seriellen Schnittstelle ausgibt.

VI. Anhang

1. Wichtige Arduino-Befehle

1.1. digitalRead(pin)

Abfrage des aktuellen Wertes eines digitalen Eingangs.
Parameter:

- **pin**: Pin-Nummer des digitalen Pins, analoge-Pins können über A0..5 auch angegeben werden.

Rückgabe: HIGH oder LOW
Beispiel:

```
int wert = digitalRead(13); // liest den Wert von Pin D13 zurück
```

1.2. digitalWrite(pin, value)

Setzen eines digitalen Ausgangs.
Parameter:

- **pin**: Pin-Nummer (siehe digitalRead())
- **value**: HIGH oder LOW

Rückgabe: nichts
Beispiel:

```
digitalWrite(13, HIGH); // setze D13 auf logisch 1
```

1.3. pinMode(pin, mode)

Konfiguration, ob Ein- oder Ausgang
Parameter:

- **pin**: Pin-Nummer (siehe digitalRead())
- **mode**: INPUT, OUTPUT, INPUT_PULLUP

Rückgabe: nichts
Beispiel:

```
pinMode(13, OUTPUT); // konfiguriere D13 als Ausgang
```

1.4. analogRead(pin)

Einlesen eines analogen Eingangs.
Parameter:

- **pin**: Pin-Nummer (A0..5)

Rückgabe: ADC-Messung (0...1023)
Beispiel:

```
int adcWert = analogRead(A0); // ADC-Messung von A0
```

1.5. delay(ms)

Unterbricht das Programm für eine gewisse Zeitspanne.
Parameter:

- **ms**: Anzahl der Millisekunden, die das Programm warten soll (maximal $2^{32} - 1$ = etwa 80 Tage)

Rückgabe: nichts
Beispiel:

```
delay(500); // warte 500 ms
```

1.6. millis()

Anzahl der Millisekunden seit Programmstart.
Parameter: keine
Rückgabe: Anzahl der Millisekunden seit Programmstart.
Beispiel:

```
unsigned long zeit = millis(); // lese aktuelle Programmzeit in ms
```

1.7. Serial.begin(speed)

Initialisierung der seriellen Schnittstelle
Parameter:

- **speed**: Baudrate (z.B. 9600)

Rückgabe: nichts
Beispiel:

```
Serial.begin(9600); // konfiguriere serielle Schnittstelle mit 9600 Baud
```

1.8. Serial.print(val, format)

Menschenlesbare Textausgabe auf der seriellen Schnittstelle. Gleitkommazahlen werden standardmäßig mit 2 Nachkommastellen (gerundet) dargestellt.
Parameter:

- **val**: Wert formatiert ausgeben.
- **format** (optional): Format (Nachkommastellen, binäre/hexadezimale Darstellung usw., siehe Beispiele)

Rückgabe: Anzahl der geschriebenen Bytes.
Beispiele:

- Serial.print(78) ergibt ”78”
- Serial.print(1.23456) ergibt ”1.23”
- Serial.print(’N’) ergibt ”N”
- Serial.print(”Hello world.”) ergibt ”Hello world.”
- Serial.print(78, BIN) ergibt ”1001110”
- Serial.print(78, OCT) ergibt ”116”

- Serial.print(78, DEC) ergibt ”78”
- Serial.print(78, HEX) ergibt ”4E”
- Serial.print(1.23456, 0) ergibt ”1”
- Serial.print(1.23456, 2) ergibt ”1.23”
- Serial.print(1.23456, 4) ergibt ”1.2346”

1.9. Serial.println(val)

Wie Serial.print() jedoch mit abschließendem Zeilenvorschub.

1.10. Serial.available()

Prüft, ob Daten von der seriellen Schnittstelle gelesen werden können.
Parameter: keine
Rückgabe: Anzahl der Bytes, die gelesen werden können.

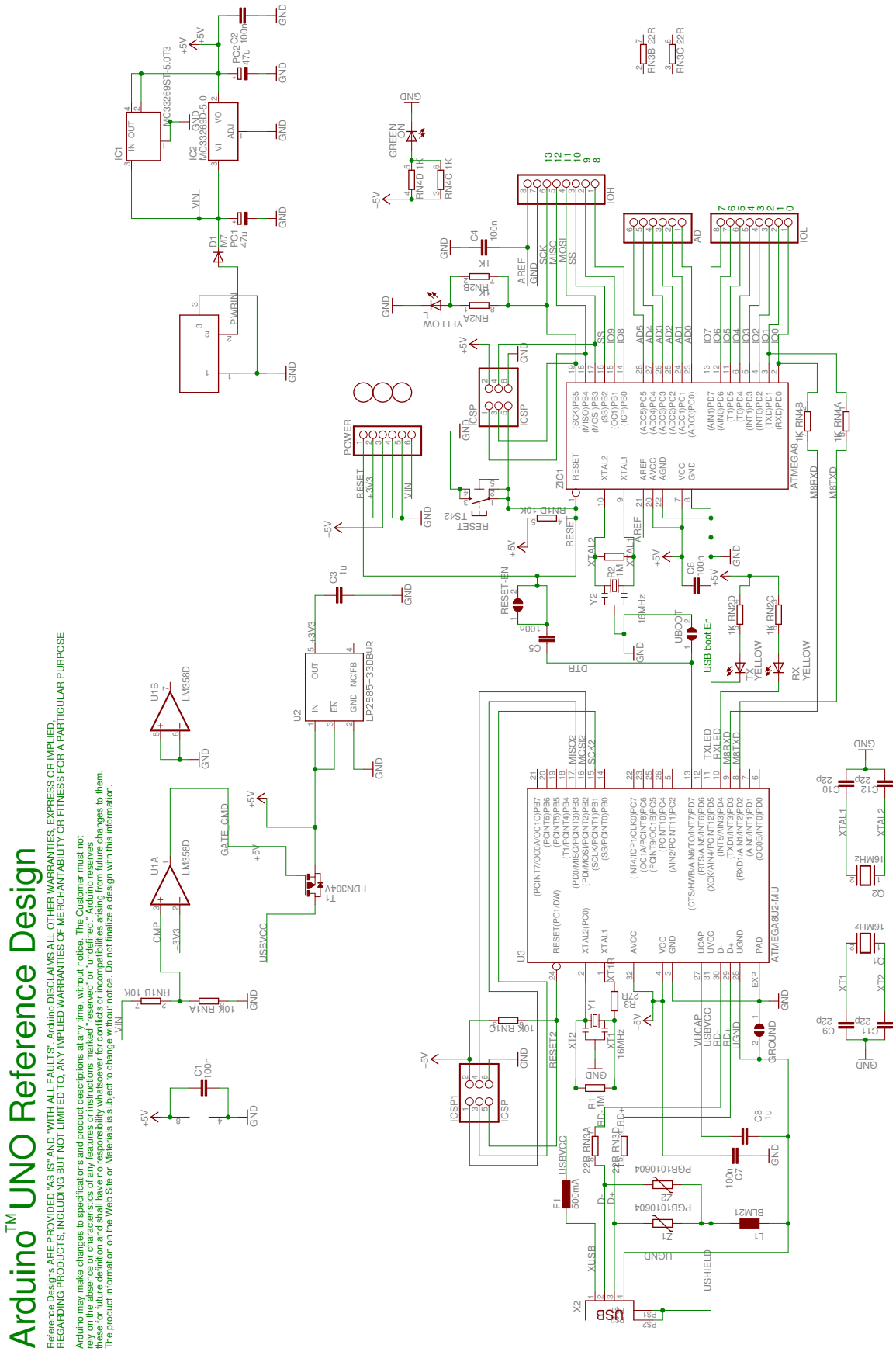
1.11. Serial.read()

Liest ein Byte von der seriellen Schnittstelle.
Parameter: keine
Rückgabe: Datenbyte oder -1, wenn keine Daten im Puffer vorhanden

1.12. weitere Befehle

Eine vollständige Funktionsreferenz der Arduino-Bibliothek finden Sie auf: http://arduino.cc/reference/en

2. Schaltplan des Arduino Uno



3. Schaltplan des Arduino Shields