Systemnahe Programmierung

Technische Informatik II

Roman Wetenkamp

9. Dezember 2021

Inhaltsverzeichnis

Vorwort

Liebe Mitstudierende,

Systemnah Programmieren – Klingt komplizierter als es ist. Vielleicht habt ihr euch schon einmal einen Elektronik-Adventskalender oder -baukasten schenken lassen, um damit herumzuspielen, einen Feuchtigkeitssensor für Blumenerde zu entwickeln oder die intelligente Wäscheklammer? In diesem Modul widmen wir uns genau mit diesem Gebiet, dem hardwarenahen Herumtüfteln an Mikrocomputern.

In diesem Skript notieren wir alles Relevante aus der Vorlesung, ergänzen es um ein paar Aufgaben und Anmerkungen und bereiten uns so auf die Klausur vor. Dieses Skript basiert auf der Vorlesung von Joachim Wagner an der DHBW Mannheim im Studiengang Informatik – Cyber Security, die Passgenauigkeit für andere Dozierende oder Jahrgänge kann ich nicht beurteilen.

Viel Erfolg!

Roman Wetenkamp
Mannheim, den 9. Dezember 2021

Fehlerfinden Mein Dank gilt folgenden Personen, die Fehler gefunden, korrigiert und so dieses Skript verbessert haben:

- Daniel Riebel

Warnung Das Studium an einer Dualen Hochschule unterscheidet sich von dem Studium an Universitäten oder regulären Fachhochschulen insbesondere dadurch, dass aufgrund der Dualität von Theorie und Praxis meist nur die Hälfte der Zeit zur Vermittlung des Stoffes zur Verfügung steht (wenn dann auch intensiver). Daher gehen Sie bitte nicht davon aus, dass Sie dieses Skript ausreichend auf Klausuren in regulären Vollzeitstudiengängen vorbereitet!

Hinweis Dieses Dokument ist kein Vorlesungsmaterial, hat nicht den Anspruch auf Vollständigkeit und enthält mit Sicherheit Fehler. Desweiteren ist es noch lange nicht vollendet (es ist infrage zustellen, ob es das je sein wird), und doch möchte ich Sie ermutigen, beizutragen! Jegliche Fehler, Probleme oder Anmerkungen können Sie mir gerne über das dazugehörige GitHub-Repository unter der URL `https://github.com/RWetenkamp/sysprog` zukommen lassen. Danke!

Teil I.
Einfache Ein-/Ausgabe-Prozesse

1. Einführung

Systemnah zu programmieren bedeutet zuallererst, viele Ebenen der Abstraktion, die Betriebssysteme oder Virtualisierungen hinter sich zu lassen und zurückzukehren zu dem grundlegendsten Konzept der Computertechnik, dem Bit.

Definition 1. *Ein Bit ist die kleinste Speichereinheit eines Computersystems. Der Wert eines Bits ist entweder **0** für **aus** oder **1** für **ein**. Sprechen wir von einem 8-Bit-Register, so bezeichnen wir damit acht einzeln adressierbare Werte, die jeweils auf 0 oder 1 gesetzt werden können.*

$$8\ \textit{Bit} \mathrel{\widehat{=}} 1\ \textit{Byte}$$

Sie werden in Ihren Programmen verschiedene Bitoperationen implementieren müssen, da die zugrundliegende Hardware zu großen Teilen aus Registern besteht, die sie auf diese Weise ansprechen können.

Bezeichnung	Symbol	C-Operatorsymbol	a	b	Ergebnis
Konjunktion	$\wedge$	`&`	0	0	0
			0	1	0
			1	0	0
			1	1	1
Disjunktion	$\vee$	`|`	0	0	0
			0	1	1
			1	0	1
			1	1	1
XOR	$\oplus$	`^`	0	0	0
			0	1	1
			1	0	1
			1	1	0

Tabelle 1: Übersicht über binäre Bitoperationen

Bezeichnung	Symbol	C-Operatorsymbol	a	Ergebnis
Negation	$\neg$		0	1
			1	0
Linksverschiebung		`<<`	0000	`1 << 1`: 0001, `1 << n`: 00...1...0
Rechtsverschiebung		`>>`	1000	`1 >> 1`: 0100, `1 << n`: 00...1...0

Tabelle 2: Übersicht über unäre Bitoperationen

Im folgenden Kapitel betrachten wir die verwendete Hardware und beginnen damit, die einzelnen Bestandteile und ihre Aufgaben zu erläutern. Wir werden jeweils anhand von

Beispielprogrammen zeigen, wie die einzelnen Komponenten angesprochen und programmiert werden können. Die Programmiersprache hierfür ist C, in der Vorlesung wurde das „Microchip Studio“ als IDE verwendet. Sie können es hier beziehen: Microchip Studio

2. Hardware

Grundlegend ist Ihnen der Aufbau eines Computersystems sicherlich vertraut.

- Die **CPU** (Central Processing Unit), also der Prozessor, ist die zentrale Komponente, die die Befehle ausführt.
- Während der Ausführung speichert die CPU Programmdaten im Arbeitsspeicher, dem **RAM** (Random Accessible Memory). Dieser Speicher wird nur zur Laufzeit genutzt und ist daher nicht persistent.
- Ebenso gibt es einen Massenspeicher, eine Festplatte, auf der Daten persistent gespeichert werden und bei Bedarf gelesen/geschrieben werden.

Diese Grundstruktur gleicht im Wesentlichen dem Von-Neumann-Prinzip und ist sowohl auf klassische PCs anwendbar als auch auf Microcontroller, denen wir uns im Rahmen dieser Vorlesung widmen wollen.

Bemerkung 1. *Ein **Microcontroller** ist ein Minicomputer auf einem Chip. Er enthält einen Microprozessor und Speicher.*

Ein solcher Microcontroller findet sich häufig auf Entwicklungsboards, wie z. B. der Arduino-Produktfamilie, die bereits aufgelötete Ports für weitere Komponenten enthalten. Derartige Boards gibt es zuhauf im Markt. Häufig sind die Unterschiede gering, da Arduino-Boards unter einer Open-Source-Lizenz stehen und somit von anderen ohne Weiteres kopiert werden dürfen.

Im Rahmen dieser Vorlesung arbeiten wir mit Arduino-Microcontrollern oder zu Arduino kompatiblen Äquivalenten. Die von uns verwendeten Boards enthalten allesamt folgenden Microprozessor:

ATMEL ATmega 328P

Die Kenntnis des Microcontrollers ist unabdingbar für den weiteren Verlauf der Vorlesung. Die Microcontroller Arduino Uno und Arduino Nano enthalten genannten Prozessor. Wir werden als Referenz jeweils den Arduino Uno verwenden.

Aufbau Dieser Microcontroller enthält einige relevante Komponenten, die über einen Bus mit der CPU verbunden sind. Wir entnehmen diese dem folgenden Blockschaltbild.

- **Flash** – Der Flash-Speicher ist neben SRAM und EEPROM einer der drei Speichertypen. Der Flash-Speicher ist unveränderlich und kann lediglich von außen gebrannt werden. Hier wird das aktuelle Skript / Programm gespeichert.
- **SRAM** – Hierbei handelt es sich um einen winzig kleinen Arbeitsspeicher. Dieser Speicher ist nicht persistent und wird von der CPU während der Laufzeit verwaltet und verwendet.

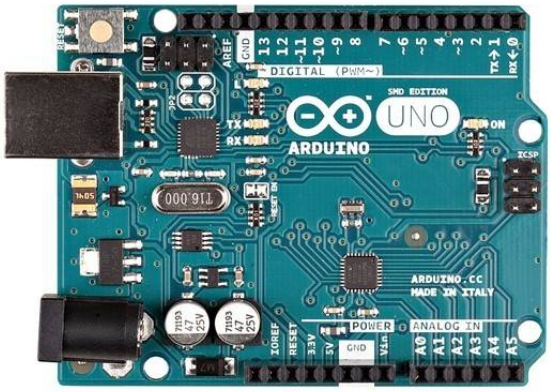
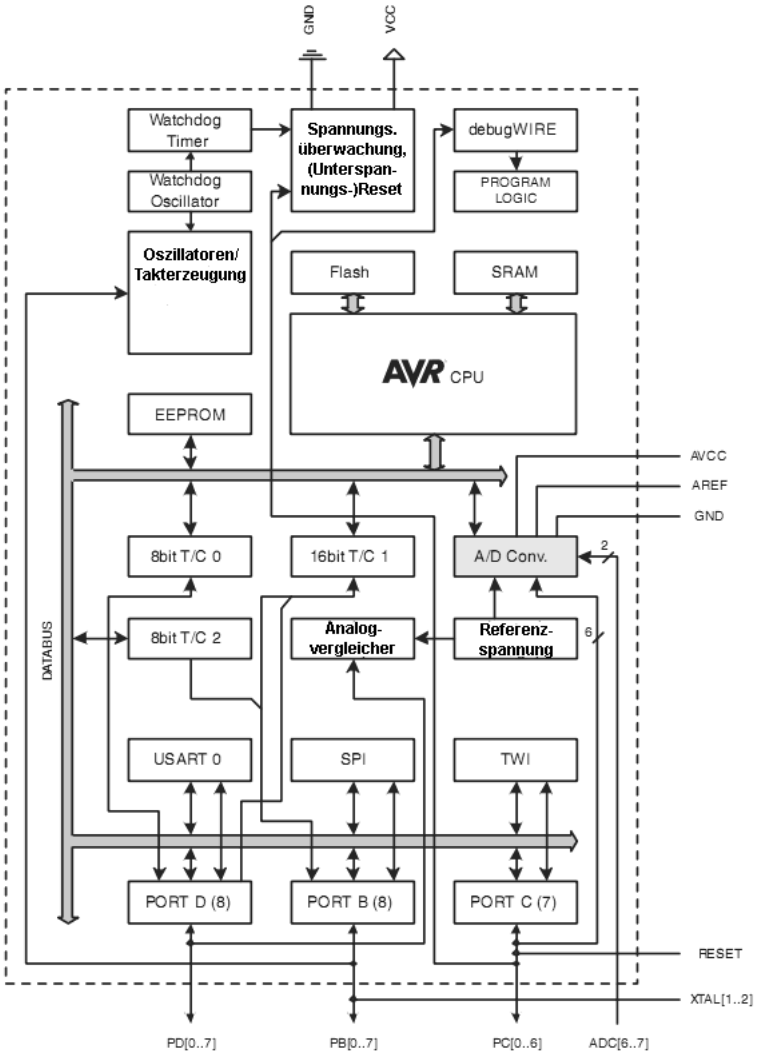


Abbildung 1: Arduino Uno mit ATMEL ATmega328P



- **EEPROM** – Dieser Speicher ist persistent und fungiert als „Festplatte“.
- **Watchdog** – Sollte der Chip nach einer gewissen Zeit kein Lebenssignal mehr sen-

den, setzt der Watchdog den Microcontroller automatisch zurück.

- **A/D Converter** – Die Prozessoren der ATmega-Reihe können analoge Signale in digitale umwandeln. Dafür ist dieser Baustein zuständig.
- **8/16bit T/C** – Diese Bauteile sind Timer/Counter, die zur Prozesssteuerung genutzt werden können. Wir widmen uns diesen Bauteilen später.
- **USART** – Eine serielle Schnittstelle für diversen Datenverkehr ist der USART-Baustein.
- **TWI** – Die TWI-Schnittstelle kann für die Verknüpfung mehrerer Microcontroller untereinander genutzt werden.

Im Allgemeinen wird die Kommunikation mit jeder Hardware/CPU über I/O-Ports abgewickelt.

Bemerkung 2. *Gelegentlich kann es zu Verwirrung kommen, da sowohl internen I/O-Ports des Microcontrollers als auch die Anschlusspins für elektronische Bauteile als **Ports** bezeichnet werden. Im Kontext dieses Kurses bezeichnen wir letztere als „Beinchen“.*

System Der hier gezeigte Controller folgt der **RISC**-Architektur. RISC steht für „Reduced Instruction Set Controller“. Es hat sich gezeigt, dass mit einem im Vergleich zu Assembler reduzierten Befehlssatz ähnlich effizient, jedoch viel leichter dekodierbar und in der Regel schneller, gearbeitet werden kann. Der Microcontroller enthält von sich aus kein Betriebssystem. Wollen wir ein Programm ausführen, so müssen wir dieses speziell für diesen Microprozessor in einer hardwarenahen Programmiersprache wie Java oder C implementieren.

3. Grundlegendes zur Programmierung

Wir werden in dieser Vorlesung – und damit auch in diesem Skript – die Programmiersprache C verwenden. Um die benötigten Bibliotheken nutzen zu können, müssen die passenden Treiber installiert werden. Am einfachsten lässt sich dies durch die Arduino IDE und das zuvor erwähnte Microchip Studio realisieren. Das **Flashen**, also das Aufspielen der Programme auf das Board, erfolgt über USB.

Programmaufbau Wie Sie es von C gewohnt sein werden, unterscheiden wir **Programmzeilen** und **Präprozessordirektiven**. Die Befehle, die ausgeführt werden sollen, müssen Bestandteil der `main()`-Funktion des Programms sein.

Bemerkung 3. *Jedes Programm, dass Sie flashen wollen, muss eine Endlosschleife enthalten! Andernfalls wird die Operation lediglich ein einziges Mal ausgeführt.*

Angewandt bedeutet dies:

```
#define F_CPU 16000000UL
#include <avr/io.h>

int main() {
        // Initialisierungen
        while(1) {
```

```
            // Hier koennen Sie Ihren Code platzieren
        }
}
```

Sie erkennen die Definition der Konstante F_CPU in der ersten Zeile. Hier legen wir fest, dass der Prozessor mit einem Takt von 16 MHz arbeitet.

3.1. Register und Ports

Unser Ziel ist es, verschiedene elektronische Bauteile an den Ports des Microcontrollers anzuschließen und mit diesem interagieren zu lassen. Dafür veranschaulichen wir uns zunächst die Pinbelegung des Boardes. Von Relevanz für das nun folgende programma-

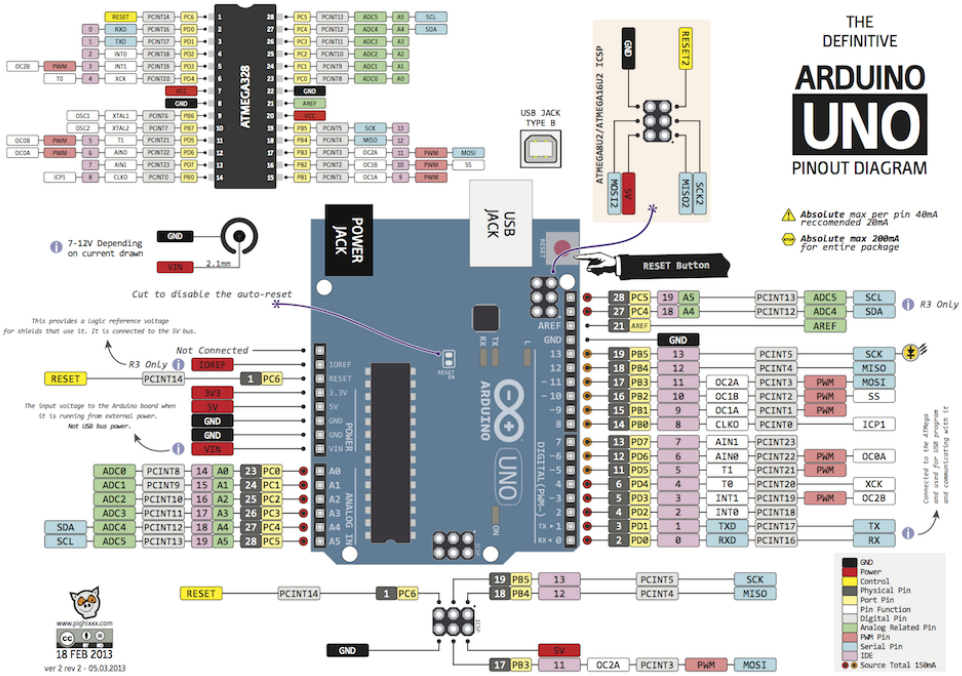


Abbildung 2: Pinbelegungen des Arduino Unos

tische Ansprechen der Pins sind die hier gelb hinterlegten „Port Pin“-Belegungen, also beispielsweise der Port **PB5** an Pin 19 bzw. Onboard-IDE 13. Wir erkennen, dass es drei mal acht dieser Ports gibt, die Register PB (für Port B), PC und PD. Diese Ports nutzen wir – wie angesprochen – um zusätzliche Bauteile anzusprechen.

Für jeden dieser 24 Pins müssen wir festlegen, ob es sich um eine Input- oder Output-Port handelt. Diese Informationen halten wir im **DDR (Data direction register)** fest.

Programmzeilen

```
DDRB |= (1 << DDB5);                                    // (1)

DDRD |= (1 << DDD3) | (1 << DDD4) | (1 << DDD5);        // (2)

DDRD &= ~(1 << DDD2);                                   // (3)
```

(1) Im Data Direction Register B verschieben wir eine 1 an die Stelle des fünften Bits, um Pin B5 als Ausgang zu definieren. Wir verwenden die Disjunktion, um andere, möglicherweise bestehende Eingaben, nicht zu überschreiben. Alternativ dazu ließe sich das Register selbstredend auch durch eine Angebe der Art `0b00100000` initialisieren, falls man beabsichtigt, nur das Bit 5 als Ausgang zu definieren.

(2) In dieser Zeile gehen wir ähnlich vor und initialisieren das DDR D. Nun soll jedoch nicht bloß ein einzelnes Bit initialisiert, sondern drei gleichzeitig, die wir gegenseitig disjunktiv verknüpfen.

(3) Nun bleibt noch, einen Pin wie hier als Eingang zu definieren. Da sich Nullen binär nicht schieben lassen, arbeiten wir hier mit der Negation der 1-Shift-Operation.

Im nächsten Kapitel betrachten wir die Möglichkeiten des Einlesens genauer, insbesondere behandeln wir dort Interrupts.

Mit der Definition des DDRs für die entsprechenden Bits ist die Hälfte getan. Wollen wir nun die Systemspannung von 5 V an einen spezifizierten Port anlegen, gehen wir analog vor:

```
PORTB |= (1 << PORTB5);          // einschalten
PORTB &= ~(1 << PORTB5);         // ausschalten
```

3.2. Eingangssignale

In vielen Schaltungen verwenden wir Taster und Schalter, um auf Benutzereingaben zu reagieren. Diese Bauelemente beeinflussen die Ströme, in dem eine Spannung erst bei Tastendruck anliegt oder dann nicht mehr. Der Microcontroller liest diese Spannungen nun aus und interpretiert sie entsprechend mit 0/1 (an/aus). Für dieses Verfahren gibt es zwei Varianten:

- **Polling** – Hierbei wertet das Programm intervallbasiert die anliegende Spannung am Pin des Bauelements aus. Dieses Vorgehen ist ressourcenintensiv und daher nach Möglichkeit zu vermeiden.
- **Interrupts** – Effizienter hingegen ist es, wenn das Programm durch einen vordefinierten Interrupt unterbrochen wird. Dieses Verfahren wird im nächsten Kapitel ausführlicher behandelt.

Eine allgemeine Schwierigkeit beim Einlesen der Daten liegt aufgrund des **Prellens** vor. So verläuft der Spannungswechsel zwischen 0 und 5 V längst nicht so sauber, wie es wünschenswert ist. Stattdessen kann es bis zu 10 Sekunden dauern, bis ein klares Signal vorliegt, da elektromagnetische Spannungen und Wellen die gemessene Spannung verändern. Um diesem Phänomen zu begegnen, werden sogenannte **Pull-Up-Widerstände** eingesetzt, die dafür sorgen, dass die Spannung im nicht gedrückten Zustand „hochgehoben“ wird und im Falle des Tastendrucks spürbar abfällt. Der Widerstand hat hierbei häufig den Wert 4.7 kΩ.

4. Programmbeispiele

An dieser Stelle will ich die relevantesten Programme aus der Vorlesung abbilden, um sie reproduzierbar zu machen.

Abbildung 3: Prellen eines Tasters

4.1. Blinken

Ein Standard-Beispiel, um zu testen, ob ein Arduino-Board funktionsfähig ist, ist das **Blink**-Beispiel. Wir wollen die Onboard-LED des Boards in einem vordefinierten Intervall zum Blinken bringen.

Schaltung Für dieses Beispiel müssen keine weiteren Komponenten an das Board angeschlossen werden. Wir verwenden lediglich die bereits fest verlötete Onboard-LED.

Programm

```
/*
 * Blinken
 *
 */

#define F_CPU 16000000UL          // Prozessortaktkonstante
#include <avr/io.h>               // Systembibliothek
#include <util/delay.h>           // Systembibliothek

int main(void)
{
        DDRB |= (1 << DDB5);              // initialisiert B5 als Ausgangspin
        PORTB |= (1 << PORTB5);           // legt Spannung an Port B5 an

        while (1)
        {
                PORTB &= ~(1 << PORTB5);          // entzieht B5 die Spannung
                _delay_ms(500);                   // Wartet 500 Millisekunden
                PORTB |= (1 << PORTB5);           // setzt B5 unter Spannung
                _delay_ms(500);
        }
}
```

4.2. Tastendruck auslesen

– folgt –

Aufgaben

Teil II.
Interrupts

5. Interrupts