AVR-Tutorial

Inhaltsverzeichnis

[Verbergen]

Aufbau des Tutorials

- Einleitung: Worum geht es überhaupt?
- Benötigte Ausrüstung: Welche Hard- und Software brauche ich, um AVR-Mikrocontroller zu programmieren?
- I/O-Grundlagen: Wie kann ich Taster und LEDs an einen AVR anschließen und benutzen?
- Logik: Verschiedene Grundoperationen und Verknüpfungen
- Arithmetik: Verschiedene Grundoperationen
- Der Stack: Was ist der Stack und wie funktionieren Unterprogrammaufrufe?
- LCD: Ansteuerung eines LC-Displays im 4bit-Modus
- Interrupts: Was sind Interrupts und wie kann ich sie verwenden?
- Vergleiche: Wie werden Entscheidungen getroffen?
- Mehrfachverzweigung: Eine Variable auf mehrere Werte prüfen.
- Der UART: Wie kann ich Daten zwischen einem Mikrocontroller und einem PC austauschen?
- Flash, EEPROM, RAM: Die verschiedenen Speicherarten des AVR und ihre Anwendung.
- Die Timer: in regelmäßigen Zeitabständen Dinge tun.
- Die Timer: Uhr und CTC Modus.
- Der ADC: Die Brücke von der analogen zur digitalen Welt.
- Tasten: Einzelne Tastendrücke und Entprellen.
- PWM: Ein Timer dimmt eine LED.
- Schieberegister: Ausgabeport erweitern.
- SRAM: Wenn die vorhandenen Register nicht mehr reichen.
- 7-Segment Anzeigen: Was ist Multiplexing?
- Der Watchdog und dessen Wirkungsweise.

Falls ihr irgendwelche Fragen habt, stellt diese bitte im Forum!

Was ist ein Mikrocontroller?

Ein Mikrocontroller ist ein Prozessor. Der Unterschied zu PC-Prozessoren besteht darin, dass bei einem Mikrocontroller Speicher, Digital- und Analog-Ein- und -Ausgänge etc. meist auf einem einzigen Chip integriert sind, so dass eine Mikrocontroller-Anwendung oft mit wenigen Bauteilen auskommt.

Mikrocontroller werden als erstes an der Bit-Zahl des internen Datenbusses unterschieden: 4bit, 8bit, 16bit und 32bit. Diese Bit-Zahl kann man als die Länge der Daten interpretieren, die der Controller in einem Befehl verarbeiten kann. Die größte in 8 Bit (= 1 Byte) darstellbare Zahl ist die 255, somit kann ein 8-Bit-Mikrocontroller z.B. in einem Additionsbefehl immer nur Zahlen kleiner-gleich 255 verarbeiten. Zur Bearbeitung von größeren Zahlen werden dann jeweils mehrere Befehle hintereinander benötigt, was natürlich länger dauert.

Ein Mikrocontroller braucht zum Betrieb, wie jeder andere Prozessor auch, einen Takt. Die maximale Taktfrequenz mit der ein Controller betrieben werden kann, reicht von 1 MHz bei alten Controllern bis hin zu über 100 MHz bei teuren 32-Bittern. Diese Taktfrequenz sagt jedoch noch nichts über die tatsächliche Geschwindigkeit eines Prozessors aus. So wird z.B. bei den meisten 8051-Controllern die Frequenz intern durch 12 geteilt, ein mit 24 MHz getakteter 8051 arbeitet also eigentlich nur mit 2 MHz. Benötigt dieser dann für einen Befehl durchschnittlich 2 Taktzyklen, so bleiben "nur" noch 1 Mio. Befehle pro Sekunde übrig - ein AVR, der ungeteilt mit 8MHz arbeitet und für die meisten Befehle nur einen Zyklus braucht, schafft dagegen fast 8 Mio. Befehle pro Sekunde.

Wozu ist ein Mikrocontroller gut?

Hier ein paar Beispiele, für welche Aufgaben Mikrocontroller verwendet werden (können):

- Ladegeräte
- Motorsteuerungen
- Roboter
- Messwerterfassung (z.B. Drehzahlmessung im Auto)
- Temperaturregler
- MP3-Player
- Schaltuhren
- Alarmanlagen
- ...

Welchen Mikrocontroller soll ich verwenden?

Typische Anforderungen an einen Mikrocontroller für Hobbyanwender (einige davon konkurrieren miteinander):

- Gute Beschaffbarkeit und geringer Preis
- Handliche Bauform: Ein Controller mit 20 Pins ist leichter zu handhaben als einer mit 128
- Flash-ROM: Der Controller sollte mindestens 1000 mal neu programmiert werden können
- In-System-Programmierbarkeit (ISP): Man benötigt kein teures Programmiergerät und muss den Controller zur Programmierung nicht aus der Schaltung entfernen
- Kostenlose Software verfügbar: Assembler bekommt man praktisch immer kostenlos

Weitere Entscheidungskriterien sind im Artikel Entscheidung Mikrocontroller zusammengefasst.

Viele dieser Anforderungen werden von den 8-bit-AVR-Controllern von Atmel erfüllt. Deshalb werde ich einen AVR, genauer gesagt den ATmega8, in diesem Tutorial einsetzen.

Und damit kein Missverständnis aufkommt: So etwas wie den "besten" Controller gibt es nicht. Es hängt immer von der Aufgabenstellung ab, welcher Controller **gut** dafür geeignet ist. Natürlich haben sich einige Controller als Standardtypen in der Praxis durchgesetzt, mit denen man in vielen Fällen ein gutes Auslangen hat und die mit ihrer Leistungsfähigkeit einen weiten Bereich abdecken können. Der ATmega8 ist z.B. so einer. Aber daneben gibt es noch viele andere.

Assembler, Basic oder C?

Warum ist dieses Tutorial für Assembler geschrieben, wo es doch einen kostenlosen C-Compiler (WinAVR, AVR-GCC) und einen billigen Basic-Compiler gibt?

Assembler ist für den Einstieg "von der Pike auf" am besten geeignet. Nur wenn man Assembler anwendet, lernt man den Aufbau eines Mikrocontrollers richtig kennen und kann ihn dadurch besser nutzen; außerdem stößt man bei jedem Compiler irgendwann mal auf Probleme, die sich nur oder besser durch das Verwenden von Assemblercode lösen lassen. Und sei es nur, dass man das vom Compiler generierte Assemblerlisting studiert, um zu entscheiden, ob und wie man eine bestimmte Sequenz im C-Code umschreiben soll, um dem Compiler das Optimieren zu ermöglichen/erleichtern.

Allerdings muss auch erwähnt werden, dass das Programmieren in Assembler besonders fehleranfällig ist und dass es damit besonders lange dauert, bis das Programm erste Ergebnisse liefert. Genau aus diesem Grund wurden "höhere" Programmiersprachen erfunden, weil man damit nicht immer wieder "das Rad neu erfinden" muss. Das gilt besonders, wenn vorbereitete Programmblöcke zur Verfügung stehen, die man miteinander kombinieren kann. Wer regelmäßig programmieren und auch längere Programme schreiben möchte, dem sei deshalb geraten, nach diesem Assembler-Tutorial C zu lernen, zum Beispiel mit dem AVR-GCC-Tutorial.

Wer C schon kann, für den bietet es sich an, das Tutorial parallel in C und Assembler abzuarbeiten. Die meisten hier vorgestellten Assemblerprogramme lassen sich relativ einfach in C umsetzen. Dabei sollte großes Augenmerk darauf gelegt werden, dass die dem Programm zugrunde liegende Idee verstanden wurde. Nur so ist ein vernünftiges Umsetzen von Assembler nach C (oder umgekehrt) möglich. Völlig verkehrt wäre es, nach sich entsprechenden 'Befehlen' zu suchen und zu glauben, damit hätte man dann ein Programm von Assembler nach C übersetzt.

Weitere Informationen

Weiterführende Informationen u. A. zu den berüchtigten Fuse-Bits, zu Programmierhard- und Software, dem Softwarepool und einer Checkliste mit Hinweisen zur Lösung üblicher Probleme finden sich im Hauptartikel AVR.

vor zum ersten Kapitel

Kategorien: AVR | AVR-Tutorial