Universität zu Köln
Musikwissenschaft
Systematik

Skript

Assembler in der Arduino-IDE: ein Einblick

C-Code

Die Arduino-IDE baut auf dem GNU-Assembler auf. Daher ist es möglich, mit der avr-gcc den µC über die ISP-Schnittstelle in Assembler programmieren. Man kann aber viele Assembler Mnemonics und Macros auch in der C-Umgebung verwenden. Dies ermöglicht die durch die Arduino-IDE automatisch eingebundene Bibliothek `<avr/io.h>`, die u.a. wiederum `<avr/sfr_defs.h>` und diverse weitere io- Bibliotheken einfügt. Neben den unten genannten Vor- und Nachteilen soll an dieser Stelle ein erster Einblick in die direkte Verwendung von Mnemonics, aber auch ein erster Einblick in Assembler und den Kompilierungs-Vorgang gegeben werden. Als Bespiel dient der „Blink"-Sketch, der ähnlich in der IDE unter „basics" geöffnet werden kann.

```
void setup()
{
      pinMode(13, OUTPUT);
}

void loop()
{
      digitalWrite(13, HIGH);
      delay(1000);
      digitalWrite(13, LOW);
      delay(1000);
}
```

Hier soll einmal nur die Anweisung `pinMode()` in der Setup-Funktion entsprechend ersetzt werden. Zunächst muß anhand des Arduino Schaltplans [Arduino Schematic] festgestellt werden, um welchen Port des µC es sich handelt. PIN 13 des Uno-Boards ist direkt mit PB5 des ATmega 328P verbunden, gehört also zum Port B des µC [Atmel Datenblatt 2009, Figure 2-1, S.5]. Alle Porteinstellungen sind in 8bit-Registern (8-bit Architektur, bit 0 bis 7) organisiert [Atmel Datenblatt 2009, S.77]. Die DDRx (Data Direction Register) bestimmen die Richtung eines Pins, eine logische „1" konfiguriert einen Ausgang, eine logische „0" einen Eingang. Für Port B bekommt das x dann den entsprechend Wert DDRB [Atmel Datenblatt 2009, S77, S.92]. Es soll ja nun im Port B PB5 als Ausgang gesetzt werden, das 5. bit des 8bit-Registers muß also eine „1" sein: 00100000. Hexadezimal entspricht das 20, dezimal 32. Die restlichen sieben bits sind für unseren sketch nicht relevant, können „0" bleiben (wodurch die übrigen PBs alle Eingänge werden). Im C-Code sieht das dann so aus:

```
void setup()
{
      DDRB = 0b00100000; // oder hex.: DDRB = 0x20; oder dez.: DDRB = 32;
}
void loop()
{
      digitalWrite(13, HIGH);
      delay(1000);
      digitalWrite(13, LOW);
      delay(1000);
}
```

Inline Assembler

Um kleinere Assembler-Stücke im C-Code direkt einzubinden, stellt gcc *Inline Assembler* zur Verfügung [GCC EXTENDED-ASM]. Eingebettet wird dies durch

```
asm ("hier steht der assembler code");
```

bzw., damit der compiler den Code nicht ungewollt wegoptimiert

```
asm volatile ("hier steht der assembler code");
```

Als Schutzmaßnahme können in Assembler die DDRx-Register nicht direkt mit Inhalten beschrieben werden[1]. Der µC besitzen daher 32 sogenannte Arbeitsregister r0 bis r31 (General-Purpose-Working-Registers) [Atmel Datenblatt 2009, S11]. Diese werden z.B. als Zwischenspeicher zwischen den I/O-Registern und dem RAM genutzt.

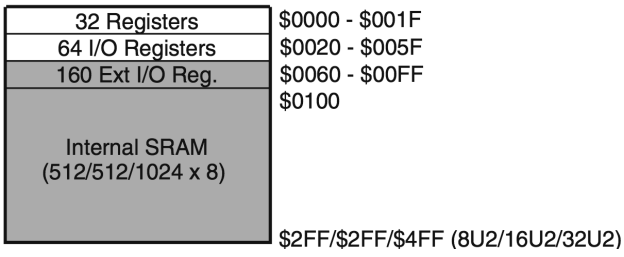

Abbildung 1: Arbeitsspeicher mit 32 Registern bei den ATmega-Mikrocontrollern, aus [ATMEL Datenblatt AVR Microcontroller 2009]

Die Register ab r16 können von jedem sinnvollen Assmeblerbefehl genutzt werden. Um einen Wert in ein solches Arbeitsregister zu schreiben, dient der Befehl `ldi` (LoaD Immediate) [ATMEL INSTRUCTION SET S.94]. Anschließend wir der Wert aus r16 in das DDRB geladen, dazu dient der Befehl `out` [ATMEL INSTRUCTION SET, S.111]. Das DDRB hat die Adresse `0x04` [Atmel Datenblatt 2009, Tabelle S.426], so daß sich jetzt folgendes ergibt:

[1] Allerdings vgl. die Anweisung sbi [ATMEL INSTRUCTION SET, S. 128]

```
void setup()
{
      asm volatile
            (
            "ldi r16,0x20\n"
            "out 0x04,r16\n"
            );
}
void loop()
{
      digitalWrite(13, HIGH);
      delay(1000);
      digitalWrite(13, LOW);
      delay(1000);
}
```

Der Ausdruck \n dient dabei als Kennzeichnung eines Zeilenumbruchs zum Trennen von asm-Befehlszeilen. Es sei an dieser Stelle betont, daß es sich hier um ein recht einfaches und auch einfach zu realisierendes Besipiel handelt. Neben den üblichen Assembler Befehlen, die in der Instruktions-Anweisung von Atmel [ATMEL INSTRUCTION SET] beschrieben ist, muss grundsätzlich bei der Programmierung in Assembler einiges mehr berücksichtigt werden, als man dies von Hochsprachen gewöhnt ist, z.B. spezifische Bedeutungen der Anweisungen, Orte, an denen Parameter in einer bestimmten Form übergeben werden, das Verbindung von Variablen und Registern und eine genauere Auseinandersetzung mit der Verwendeten Hardware. Letzteres macht das Studium des µC-Handbuches unumgänglich. Dennoch ist das Arbeiten mit der Hardwarenahen Assembler-Sprache sinnvoll und mit etwas Übung nicht unbedingt schwerer zu erlernen als eine Hochsprache.

Assembler: Vor- und Nachteile

Vorteile:

Assembleranweisungen sind adäquat zu Maschinencodes, sind daher auch „Maschinen-näher“ als Hochsprachen wie z.B. C. Grundsätzlich entspricht ein einzelner Assemblerbefehl (Mnemonic) einem entsprechenden Maschinenbefehl. Daher sind in reinem Assembler (nicht inline-Assembler) Programme grundsätzlich schlanker und schneller als kompilierte Kodex, die durch Hochsprachen erzeugt werden. Z.B. kann man in extrem kurzer Zeit Pins gleichzeitig an und ausschalten, da man die wiring.c-Bibliothek umgehen kann. Diese braucht für jeden Port viele Zeilen, die alle nacheinander im Prozessortakt abgearbeitet werden müssen. So ist für eine zeitkritische Anwendung grundsätzlich reiner Assembler einer Hochsprache zu bevorzugen.

Inline-Assembler bietet - da ja in eine C-Umgebung eingebunden - Diese optimale Ausnutzung nicht. Allerdings lassen sich so kleine Assembler-Stückchen direkt in den C-Code einbetten. Dies kann von der Registerverwendung her deutlich günstiger sein, da gcc genau weiß, welche Register gebraucht werden und welche nicht. Für Zeitkritische Anwendungen kann inline-Assembler aus ebenfalls von großem Vorteil sein.

Nachteile:

- Einarbeitung in Assembler, Fehlersuche und Programmpflege ist schwieriger als bei Hochsprachen.
- Eingeschränkt oder gar nicht auf andere µC portierbar.

Maschinensprache

Der Compiler / Assembler / Debugger der Ardunio-IDE erzeugt nun eine .hex-Datei, die automatisch durch ein integriertes Ladeprogramm über die USB-Schnittstelle des Arduino-Boards in den Speicher des ATmega geladen werden. Anschaulich gesprochen werden menschenverständliche Kürzel in einen vom Mikrocontroller verständlichen Binärcode übersetzt. Diese .hex-Datei liegt bei geöffneter IDE auf der Festplatte des Programmierrechnes. Um sich den Ort anzeigen zu lassen, kann man in der IDE unter „Einstellungen“ die Haken bei „Ausführliche Ausgabe anzeigen bei Kompilierung“ setzten, dann erscheint im Kommentarfeld der IDE bei „Überprüfung“ ein entsprechender Hinweis.
Reduziert man der Einfachheit halber nun den o.g. Code auf das Beschreiben des Arbeitsregisters:

```
void setup()
{
      asm volatile
            (
            "ldi r16,0x20\n"
            );
}
void loop()
{
}
```

so ergibt sich folgende .hex-Datei:

```
:100000000C9434000C9451000C9451000C94510049
:100010000C9451000C9451000C9451000C9451001C
:100020000C9451000C9451000C9451000C9451000C
:100030000C9451000C9451000C9451000C945100FC
:100040000C9465000C9451000C9451000C945100D8
:100050000C9451000C9451000C9451000C945100DC
:100060000C9451000C94510011241FBECFEFD8E026
:10007000DEBFCDBF11E0A0E0B1E0E4EDF1E002C0F1
:1000800005900D92A030B107D9F711E0A0E0B1E0E2
:1000900001C01D92A930B107E1F70E9456000C94EF
:1000A000E8000C940000 00E2 08950895CF93DF93D8
:1000B0000E94AD000E945300C0E0D0E00E945500B5
:1000C0002097E1F30E940000F9CF1F920F920FB624
:1000D0000F9211242F933F938F939F93AF93BF93CE
:1000E000809104019091050 1A0910601B091070152
:1000F000309108010196A11DB11D232F2D5F2D37D1
:1001000020F02D570196A11DB11D20930801809369
:10011000040190930501A0930601B093070180911B
:10012000000190910101A0910201B091030101969B
:10013000A11DB11D8093000190930101A0930201C4
:10014000B0930301BF91AF919F918F913F912F91F8
:100150000F900FBE0F901F901895789484B5826011
:1001600084BD84B5816084BD85B5826085BD85B55B
:10017000816085BDEEE6F0E0808181608083E1E80A
:10018000F0E0108280818260808380818160808342
:10019000E0E8F0E0808181608083E1EBF0E0808145
:1001A00084608083E0EBF0E0808181608083EAE717
:1001B000F0E0808184608083808182608083808 1A0
:1001C000816080838081806880831092C10008955F
:0401D000F894FFCFD1
:00000001FF
```

Es handelt sich hier um das sogenannte INTEL Hex-Format, welches bereits in den 70er Jahren zum laden von Programmen von Lochstreifen verwendet wurde. Ein Satz beginnt hier immer mit einem Doppelpunkt, gefolgt von verschiedenen Kontrollbits wie Adressbits, Typ-Idendifikationsbits, den eigentlichen Datenbits und Prüfsummen für jede Zeile. Das Uploader-Programm entfernt dann die zusätzlichen Daten. Um nun die Anweisung für r16 wiederzufinden, wird der Opcode (operation-code) für den `ldi`-Befehl benötigt [ATMEL INSTRUCTION SET, S.94]:

1110	KKKK	dddd	KKKK

Der Syntax lautet: `LDI Rd,K`. Der Wert für `K` ist hexadezimal „20“, entsprechend binär „100000“. Damit ergibt sich für den Opcode schon mal

1110	0010	dddd	0000

Das verwendete Register ist das erste nutzbare, erhält daher im Opcode den Wert „0“:

1110	0010	0000	0000

Somit ergibt sich hexadezimal für die 2 Byte: „e200“

Nun steht im mm INTEL Hex--Code immer das Lowbyte vor dem Highbyte, somit müssen diese beiden vertauscht werden: „00e2“

Dieser Kodeschnipsel findet sich tatsächlich einmal im o.g. .hex-Code (gelb hervorgerufen).

Literatur

Hardware-Informationen

www.atmel.com/Images/doc8161.pdf [ATMEL DATENBLATT 2009]
http://arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf [ARDUINO SCHEMATIC]
http://arduino.cc/hu/Hacking/PinMapping
http://arduino.cc/en/Reference/PortManipulation

Hardware-Informationen einführend (Beispiele)

Skript *Grundlagen Mikrocontroller – ganz kurz gefaßt*
BRINKSCHULTE, UWE; UNGERER, THEO (2007): *Mikrocontroller und Mikroprozessoren, 2. Auflage*, eXamen press, Springer, Berlin
TIETZE, ULRICH; SCHENK, CHRISTOPH (1991): *Halbleiter-Schaltungstechnik*, 2. Auflage. Auflage, Springer Vieweg (historisch nicht aktuell, aber gut einführend)

Atmel AVR instruction set

www.atmel.com/Images/doc0856.pdf [ATMEL INSTRUCTION SET]

Assembler und AVR

SCHMIDT, GERHARD: *Anfängerkurs zum Erlernen der Assemblersprache von ATMEL-AVR-Mikroprozessoren*, http://avr-asm-download.de/beginner_de.pdf [SCHMIDT]
ZITT, HUBERT: *Programmiertechniken für Embedded Systems, Arduino-Assembler*, Eigenverlag [ZITT]
HEINRICHS, G: Assemblieren, http://www.g-heinrichs.de/attiny/Assemblieren.pdf [HEINRICHS]

Inline Assembler

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-As [GCC EXTENDED-ASM]
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
http://www.nongnu.org/avr-libc/user-manual/inline_asm.html
http://savannah.nongnu.org/download/avr-libc/avr-libc-user-manual-1.8.0.pdf.bz2 [AVR-LIB MANUAL 2012]
http://www.mikrocontroller.net/articles/AVR_Assembler_Makros [ASSEMBLER MACROS]
http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial
http://www.lowlevel.eu/wiki/Inline-Assembler_mit_GCC
http://www.rn-wissen.de/index.php/Inline-Assembler_in_avr-gcc [ROBOTERNETZ 2012]
http://www.k2.t.u-tokyo.ac.jp/members/alvaro/courseMaterials/MicroProgramming/

& separate Literaturliste