Der Speicher des Arduino ATmega 2560

Wie bereits in der Hardwareübersicht erwähnt, hat dieses Funduinoboard mit dem ATmega 2560 aufgrund der Harvard-Architektur einen vom Programmspeicher (256 KByte, organisiert in der Form 128KBit x 16 Bit) getrennten Datenspeicher. Dieser Datenspeicher umfasst 8KByte statisches RAM. Da es sich nicht um dynamisches RAM handelt, ist keine Refreshlogik erforderlich und der Speicher kann durch eine Spannungsquelle gepuffert werden. Des weiteren enthält das Board noch einen separaten EEPROM-Speicher in der Größe von 4 KByte.

Memory-mapped I/O

Die im Mikrocontroller enthaltenen Komponenten, wie z.B. die Timer/Counter oder Analog/Digital-Wandler, sind programmierbar über ihnen jeweils zugeordnete Register.

Prinzipiell gibt es zwei Konzepte, wie die Register solcher Komponenten zugreifbar sind:

- Isolated I/O oder auch Port-mapped I/O (PMIO)
- Memory-mapped I/O (MMIO)

Im ersten Fall sind die Register dieser Hardware-Komponenten in einem vom Datenspeicher separaten Adressraum unter gebracht. Typischerweise hat der Mikrocontroller im internen Steuerbus dann ein Bussignal, welches zwischen dem Zugriff auf eine Speicheradresse oder auf eine I/O-Adresse unterscheidet. Der Zugriff auf den Speicher bzw. auf den I/O-Bereich wird dann durch unterschiedliche Assembler-Befehle abgewickelt.

Im zweiten Fall liegen die Adressen der programmierbaren Steuerregister von Hardware-Komponenten im regulären Speicher-Adressbereich des Mikrocontrollers. Diese Register werden quasi in den Speicher-Adressbereich „eingebunden" und sind somit mittels der regulären Assembler-Befehle für Speicherzugriffe benutzbar. Es gibt dann keinen separaten I/O-Adressbereich.

Dieses Konzept ist bei Mikrocontrollern weit verbreitet.

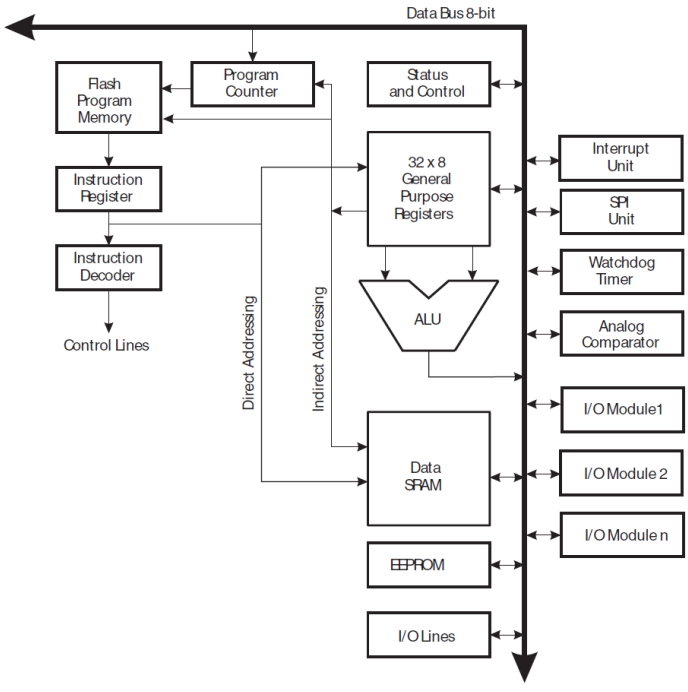
Hardware-Architektur ATmega 2560

Das neben dargestellte Bild[1] zeigt einen Überblick der Hardware-Architektur.

Intern existiert ein 8 Bit breiter *Datenbus*, der alle Komponenten miteinander verbindet.

Weiterhin erkennbar ist der vom *Datenspeicher* separate *Programmspeicher*. Der im Programmspeicher nächste auszuführende Assembler-Befehl wird durch den *Program Counter* vorgegeben und in das *Instruction Register* geladen. Der *Instruction Decoder* ermittelt dann, um welchen Befehl es sich handelt und erzeugt die notwendigen Steuersignale (*Control Lines*) für alle betroffenen Komponenten.

Die Arithmetisch-logische Einheit (*ALU*) wird von den 32 Allzweck-Registern (*General Purpose Register*) mit zwei Werten versorgt, die



1 Entnommen aus dem Datenblatt Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf Kapitel 7.2

dann, je nach Assembler-Befehl (z.B. **`ADD`**, **`SUB`**, **`AND`**, **`OR`** etc.) einen Ergebniswert liefert und wiederum in der Regel in einem Allzweck-Register ablegt. Mit Datentransport-Befehlen kann dann der Inhalt eines solchen Allzweck-Registers in eine Speicherstelle oder in eines der Steuerregister einer Hardwarekomponente über den Datenbus transferiert werden.

Die Adresse einer Speicherstelle kann entweder *direkt* oder *indirekt* angegeben werden. Bei der *direkten Adressierung* enthält der Programmcode konkret die Adresse der gewünschte Speicherstelle. Bei der *indirekten Adressierung* befindet sich die Adresse der gewünschten Speicherstelle eben nicht im Programmcode, sondern wird durch den Wert in einem Allzweck-Register vorgegeben. Somit kann durch Verändern dieses Wertes in einem Allzweck-Register die gewünschte Speicherstelle verändert werden.

Speicherkarte des Datenspeichers

Der Adressraum des Mikrocontrollers umfasst 64 KByte und ist folgendermaßen aufgeteilt[2]:

Address (HEX)	
0 - 1F	32 Registers
20 - 5F	64 I/O Registers
60 - 1FF	416 External I/O Registers
200	Internal SRAM (8192 × 8)
21FF	
2200	External SRAM (0 - 64K × 8)
FFFF	

In die ersten 512 Byte (**`0x0`** bis **`0x1FF`**) dieses Adressraums werden verschiedene Register eingebunden. Der „eigentliche" interne Datenspeicher beginnt ab der Speicheradresse **`0x0200`** und reicht bis zur Speicheradresse **`0x21FF`**. Hier schließt sich dann ein ggf. vorhandener externe Datenspeicher an, wenn der interne für den Einsatzzweck nicht ausreicht.

Einen Überblick über alle Register findet man in Kapitel „*33. Register Summary*" des Datenblatts.

Hierbei sind teilweise zwei Adressen angegeben: Ein Adresse **ohne** runde Klammern, beginnend bei **`0x00`** bis **`0x3F`** und eine zweite Adresse **mit** runden Klammern, beginnend bei **`0x20`** bis **`0x5F`**. Diese Adressen beziehen sich auf die jeweils zugehörigen Befehle **`IN`**/**`OUT`** bzw. **`LDS`**/**`STS`**.

Befehle für den Zugriff auf die 64 I/O-Register

Die Befehle **`IN`**/**`OUT`** sind nur für die 64 I/O-Register im Adressbereich **`0x20`** bis **`0x5F`** (Adressangabe **mit** Klammer!) verwendbar! Wirft man einen Blick in die Dokumentation dieser beiden Befehle, sieht man jedoch, dass bei dem **`IN`**-Befehl als Quelloperand und bei dem **`OUT`**-Befehl als Zieloperand lediglich ein Werft zwischen 0 (**`0x00`**) und 63 (**`0x3F`**) erlaubt ist. Beide Befehle verwenden zur Adressierung den I/O-Bereich! Dieser I/O-Bereich ist um 32 Adressen gegenüber

2 Entnommen aus dem Datenblatt Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf Kapitel 8.2

dem Datenbereich verschoben! Die I/O-Adresse **0x00** wird im Datenbereich an der Adresse **0x20** „eingeblendet". Alle I/O-Adressen bis hin zur Adresse **0x3F** finden sich entsprechend an den Adressen im Datenbereich bis hin zur Adresse **0x5F**.

Für alle Register kann deren Adresse durch eine symbolische Konstante ersetzt werden.

Im Atmel Studio wird bei Erzeugung eines Projekts automatisch durch Auswahl eines bestimmten Mikrocontrollers eine Datei mit eingebunden, die für alle Komponenten eine symbolische Konstante definiert. Für den ATmega 2560 heißt diese Datei **m2560def.inc**.

Man findet diese Datei im Projektmappen-Explorer unter dem Zweig „*Dependencies*". Sucht man in dieser Datei z.B. nach dem Begriff „**PORTE**" findet sich die Direktive „`.equ PORTE = 0x0e`". Mit Hilfe dieser Direktive ersetzt der Assembler im Programm alle Fundstellen der symbolischen Konstanten „**PORTE**" durch den Wert „**0x0e**". Dieser Zahlenwert ist die Adresse von **PORTE im I/O-Bereich**. Deshalb funktioniert diese Konstante in Verbindung mit den Befehlen **IN**/**OUT**, auch wenn die Adresse von **PORTE** im Datenbereich den Wert **0x2E** hat!

Kurz gesagt: Für alle 64 I/O-Register sollte man bei Verwendung von symbolischen Konstanten die Befehle IN/OUT und *nicht* die nachfolgend genannten Befehle verwenden!

Eine Besonderheit gilt noch für diese 64 I/O-Register: Alle diese Register sind auf Bit-Ebene zugreifbar. Das heißt, das man jedes einzelne Bit in einem solchen Register setzen oder rücksetzen kann. Die zugehörigen Befehle **SBI** (**S**et **B**it in **I**/O Register) und **CBI** (**C**lear **B**it in **I**/O Register) wurden bereits angewendet.

Auch hier werden häufig symbolische Konstanten verwendet. Der Befehl „`sbi PORTA, PORTA0`" setzt in **PORTA** das Bit **PORTA0**. Sieht man in der zuvor genannten Datei **m2560def.inc** nach, findet die Direktiven „`.equ PORTA = 0x02`" und „`.equ PORTA0 = 0`".

Durch die Verwendung von symbolischen Konstanten werden Assemblerprogramme deutlich besser lesbar!

Befehle für den Zugriff auf die 416 externen I/O-Register und das SRAM

Hier funktionieren die Befehle **IN**/**OUT** nicht! Der Assembler wird einen Fehler melden.

Für den Zugriff auf Adressen im Datenbereich verwendet man u.a. die Befehle **LDS** (**L**oad **D**irect From Data**s**pace) und **STS** (**S**tore **D**irect To Data**s**pace).

Diese Befehle sollte man nicht in Verbindung mit den zuvor genannten 64 I/O-Registern verwenden; zumindest wenn man symbolische Konstanten nutzt!

Der Befehl **LDS** lädt den Inhalt einer Speicherstelle im Datenbereich direkt in eines der 32 Allzweck-Register. Der Befehl **STS** hingegen speichert den Inhalt eines der 32 Allzweck-Register in die angegebene Speicherstelle.

Aufgaben

1. Informieren Sie sich auf der Webseite
   `http://www.avr-asm-tutorial.net/avr_de/beginner/diraus.html`
   über die zur Verfügung stehenden Assemblerdirektiven.

2. Es soll ein 4-Bit Rückwärtszähler mit vier LEDs visualisiert werden.

   Verbinden Sie das Funduino-Board mit vier LEDs, die an den Bits 0 bis 3 des PORTA angeschlossen sind. Ermitteln Sie hierzu die notwendigen Pins des Funduino-Boards.

   Verwenden Sie das unten angegebene Assembler-Rumpfprogramm.
   Erweitern Sie das Rumpfprogramm im MAIN-Teil so, dass der Wert an der Speicherstelle **`0x0200`** mit den vier LEDs visualisiert wird.
   Hierbei soll der Wert nach jeder Ausgabe dekrementiert werden und wieder an der Speicherstelle **`0x0200`** gesichert werden.

   Damit die Anzeige sichtbar zählt, verwenden Sie geeignete Warteschleifen, die momentan nicht dargestellt sind.

3. Das Bit 0 im PORTE soll mit einem Taster verbunden werden.
   Erweitern Sie das Programm so, dass die Visualisierung angehalten wird, solange der Taster gedrückt wird (Bit 0 in PORTE hat den Wert 0).

4. Verwenden Sie anstelle der Warteschleifen aus einen Timer/Counter. Nutzen Sie die ISR zur Visualisierung. Überlegen Sie, wie Sie nun das Anhalten der Visualisierung realisieren können.

```
;=====================================================================
; Projekt Basics
; Demonstriert den Zugriff auf den I/O-Bereich und Datenbereich
;=======Start des Assembler-Programms=================================
.org         0x0000
;=======Initialisierung===============================================
INIT:
    sbi     DDRA,   DDA0        ; Setze Bit 0 in Port A als Ausgang
    sbi     DDRA,   DDA1        ; Setze Bit 1 in Port A als Ausgang
    sbi     DDRA,   DDA2        ; Setze Bit 2 in Port A als Ausgang
    sbi     DDRA,   DDA3        ; Setze Bit 3 in Port A als Ausgang

    sbi     PORTE,  DDE0        ; Schalte Pullup für Bit 0 in Port E ein

    ldi     R16,    15
    sts     $0200,  R16         ; Speichere 15 an der Adresse $0200
    rjmp    MAIN

;=======Konstante Werte===============================================
.db   "Hallo Welt!",13

;===Haupt-Programm in Endlosschleife==================================
MAIN:

      rjmp   MAIN                       ; Zurück zum Anfang
;=====================================================================
```