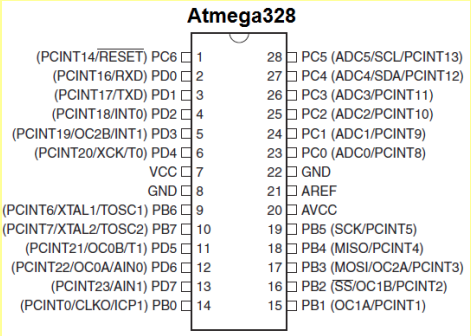
AVR Ein/Ausgabe

Inhaltsverzeichnis

1 Die Begriffe PIN und PORT



Als Pins werden die Kontaktpunkte eines IC (integrated circuit) bezeichnet. Die Pinbelegung (pinout) zeigt diese Pins und ihre logische Funktion. Es gibt sehr wenige Pins, daher sind viele mehrfach belegt.

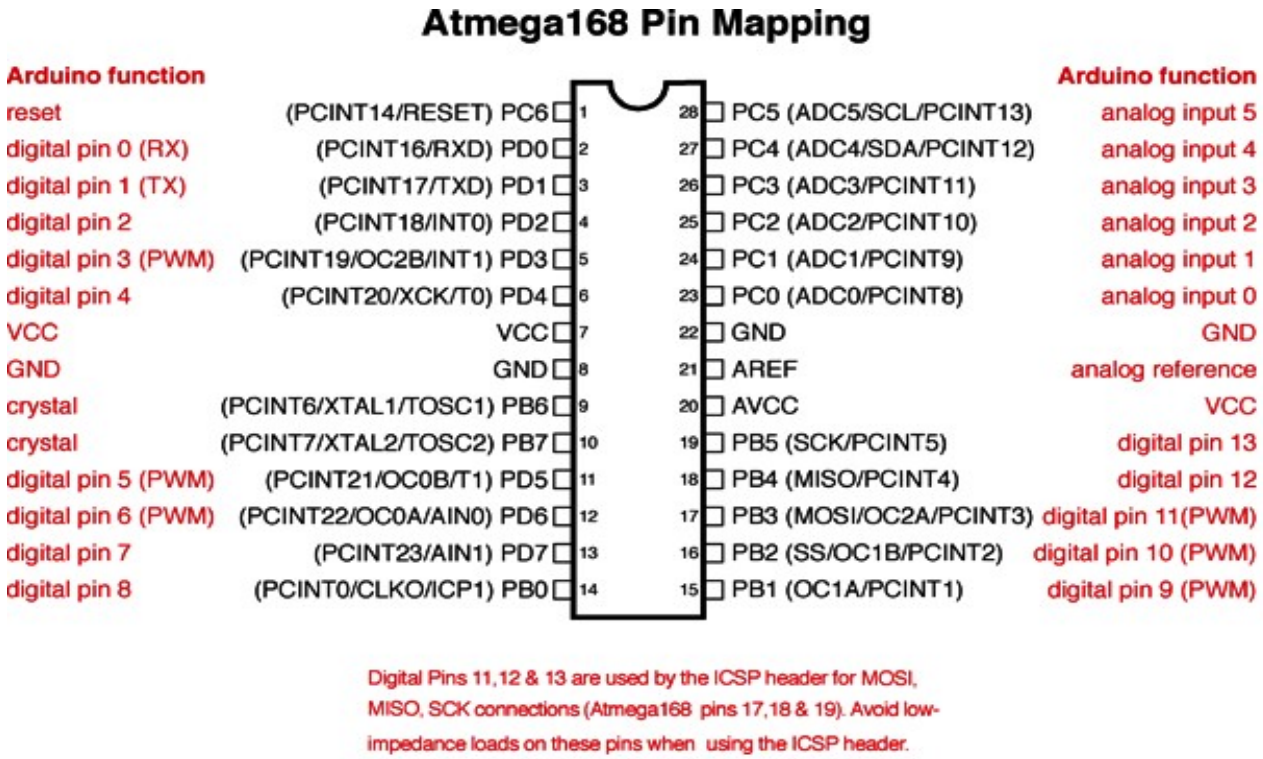
Pin 14 kann daher als normaler Input und Ouput für digitale Signale arbeiten, aber auch als Input für einen Pin-Change Interrupt (PCINT0) oder als Input um den Zählerstand eines laufenden Zählers zu schnappen (input capture pin ICP1).

PORT

Als PORT werden mehrere zusammengehörige Pins bezeichnet (z.B: PORTB = PB0,PB1 … PB7).

In der Programmierung bedeutet PORT allerdings die hinter jedem Pin liegende Speicherzelle (8-Bit Register), die sich den Zustand eines Pins (HIGH oder LOW output) merkt oder für die Input-Pins die Steuerung der Pullup-Widerstände regelt.

Pinout für den Arduino UNO

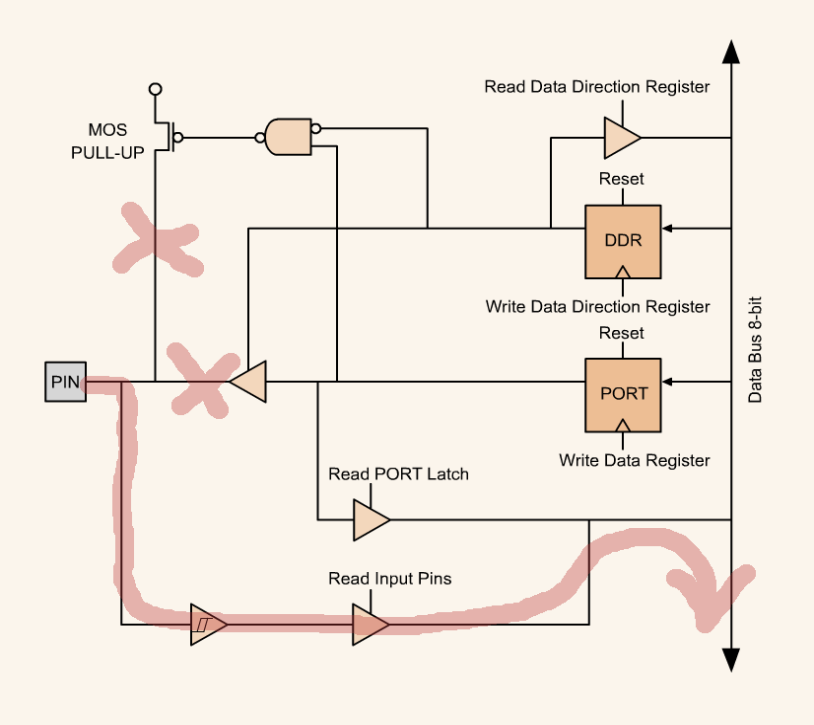


2 AVR PORT - I N P U T → PINB

- C-Zeile für das Einlesen des Ports in eine Variable x
- Schaltplan für den Datenpfad „Input“

```
uint8_t  x = PINB;
```

Man erkennt, dass der Wert aller 8 Pins direkt eingelesen wird. Ob ein einzelner Pin HIGH oder LOW ist, muss man im Programm abgefragt werden (Maske!)



3 AVR PORT Architektur Bausteine

Was ist/Wozu dient

- Schmitt-Trigger
- Gesteuerter Puffer
- D-FLIPFLOP
- Register
- Special Function Register (SFR)
- Pullup
- PMOS Transistor
- NAND
- INVERTER
- Bus

Schmitt-Trigger: verfügt über unterschiedliche Schaltschwellen für Ein/Aus und kann so Störungen auf einem Eingangssignal des Bausteins ausblenden
Gesteuerter Puffer: hat einen Enable-Eingang über den er ein/ausgeschaltet werden kann; ist im Ausgeschalteten Zustand hochohmig (Z), blockiert also die Leitungen am Ausgang nicht
D-Flipflop: ein 1-Bit-Speicher; speichert bei steigender Taktflake das Eingangssignal bis zum nächsten Takt oder bis der Strom abgeschaltet wird (statischer Speicher); Grundbaustein für Register
Register: meist 8/16/32/64/128 Flipflops werden parallelgeschaltet und können dann die Daten auf dem mehrspurigen Datenhighway (Bus) mit einem Taktschlag abspeichern
SFR: Register im IO Bereich zur Steuerung des Prozessors und/oder der Peripheriegeräte z.B: PORTB, PINB, DDRB, SREG
Pullup: schwacher Widerstand; erzeugt eine 1 auf der Leitung; kann aber leicht überschrieben werden; verhindert, dass eine Leitung in den hochohmigen Zustand Z geht
PMOS-Transistor: Source hängt auf VDD; bei einer Null am Gate wird der Transistor eingeschaltet, es bildet sich ein Kanal zwischen Source und Drain aus; je nach Dotierung ist dieser Kanal ein Kurzschluss oder ein Widerstand; der PMOS Transistor kann also als gesteuerter Pullup verwendet werden
NAND: nur wenn beide Eingänge true sind ist der Ausgang false
INVERTER: bildet das Komplement des Eingangssignals
Bus: Datenhighway; transportiert die Digitaldaten; mehrere parallel geführte Leitungen; Busbreiten als Vielfaches von 2 (beim AVR 8 oder 16 Bit breit); darf immer nur von einem Register beschrieben werden; lange Leitungen, daher hohe parasitäre Kapazitäten, die Baugruppen brauchen starke Treiber um den Ladestrom für diese Kapazitäten liefern zu können $\Delta U = I * \Delta t$

Bustreiber: Digitalgatter mit starken Ausgangstreibern

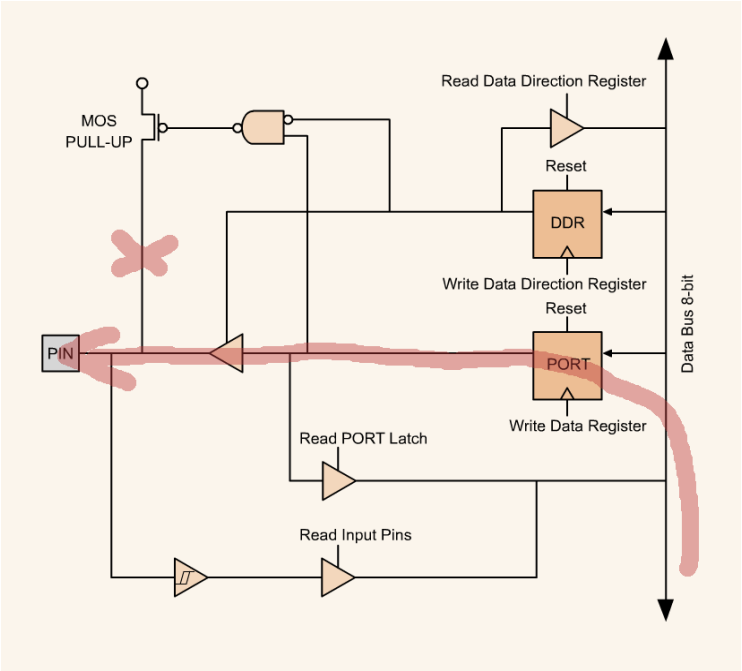
AVR Port – O U T P U T → PORTB

- C-Zeile für das Schreiben des Ports: Bitmuster 0000 1100 ausgeben
- Schaltplan für den Datenpfad „Output“

```
x = 12; // oder  x = 0b1100; x = 0xC;
PORTB = x;
```

schreibt man auf PORTB, so werden alle Bits des Registers gleichzeitig gespeichert und der Inhalt des Registers an die Pins ausgegeben.

Einzelne Bits müssen durch Maskieren auf HIGH oder LOW gesetzt werden.



4 Bitmuster und Datentypen

- Schreibweise von Konstanten in C: Dezimal, (Binär), Hexadezimal

Schreibweise in C, C++, C#

x = 12; x = 0x0C; 0xC; 0xc;

x = 0b0000 1100; x= 0b1100;

Das Bitmuster 0x41 wird wahlweise als 65 oder ‘A‘ interpertiert, abhängig davon, in welchem Zusammenhang es benutzt wird.

```
char x =  ‘A‘;   // ist gleichbedeutend mit
x = 65; // oder
x = 0x41
```

Achtung!

```
unsigned char x = -1; // hat das gleiche Bitmuster wie
signed char y = -1;     // aber x wird als 255 interpretiert (-1 hat das Bitmuster 0xFF)
```

5 Starkes/ Schwaches Digitalsignal

- Unendlich stark (kann beliebig viel Strom liefern) : mathematisches Signal 0,1
- Stark (Ströme > mA) H, L
- schwach (Ströme < mA) h, l
- sehr schwach (Ströme < uA) Z
- undefiniert: das Signal kann nicht ausgewertet werden; Widerspruch X

5.1 Fragen:

- Pegel von Pullups
- Pegel von Pulldown
- Pegel von Digitalgattern
- Pegel der Versorgungsleitungen
- was liefert ein Kurzschluss zwischen h und l
- was liefert ein Kurzschluss zwischen h und L
- was liefert ein Kurzschluss zwischen 1 und L
- Pegel eines Tasters, Umschalters, Taster mit Pullup

Pegel von Pullups: h

Pegel von Pulldown: l

Pegel von Digitalgattern: H, L

Pegel der Versorgungsleitungen: 0,1

Ein Kurzschluss zwischen h und l liefert X

Ein Kurzschluss zwischen h und L liefert L;

was liefert ein Kurzschluss zwischen 1 und L?

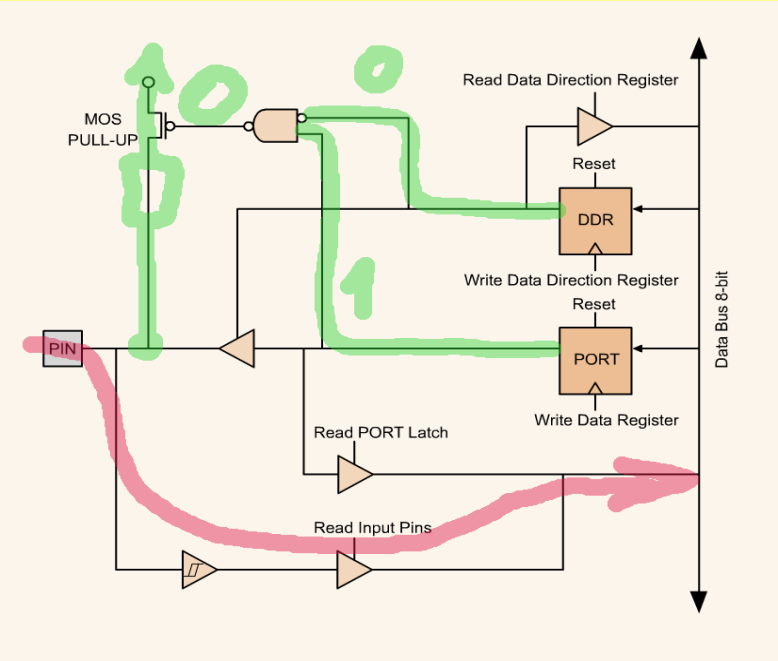
Taster: 0,Z oder 1,Z usw.

Umschalter: H,L oder 1,0

Taster mit Pullup: 0,h, L,h

6 AVR Port – INPUT MIT PULLUP → PINB/PORTB

```
DDRB = 0; //alle Pins auf Input
PORTB = 0xff;  //alle Pullups aktivieren
uint8_t x = PINB;    // Pins einlesen
```



7 MASKIEREN: Setzen eines einzelnen Bits

ohne die anderen Bits in einem Register zu verändern

```
PORTB = 0b0000010;      //ACHTUNG! Es werden alle Bits gesetzt!
```

1. MASKE = 0b0000010;
2. BITWEISES ODER

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

Beispiel: Setzen des Bits 1

```
uint8_t mask = 0b0000 0010;
PORTB = PORTB | mask;
```

oder

```
PORTB |= 0b0000 0010;
```

8 Maske für Setzen

Für jedes Bit das in einem SFR (special function register) gesetzt oder gelöscht werden soll muss in der Maske das zugehörige Bit auf 1 gesetzt werden.

8.1 Schiebeoperation

```
uint8_t maske = 1 << 1;
```

0000 0001 „1“
0000 0010 „1<<1“

8.2 Mehrere Bits

```
uint8_t maske = (1 << 1) | (1 << 3);
```

0000 0010
0000 1000 „1<<3“

0000 1010 „(1 << 1) | (1 << 3)“

Zur besseren Lesbarkeit: Bits in den SFR (special function register) mit Namen und Werten

```
#define PB1 1
#define PB3 3
uint8_t maske = (1 << PB1)|(1 << PB3);
```

8.3 _BV (Byte Value) Makro

In der Stdlib.c ist in io.h ein Makro zur Bildung von Masken definiert:

```
#define _BV(bit)    (1 << (bit))
```

Beispiel:

```
#include <avr/io.h>
uint8_t maske = _BV(PB1) | _BV(PB3)
```

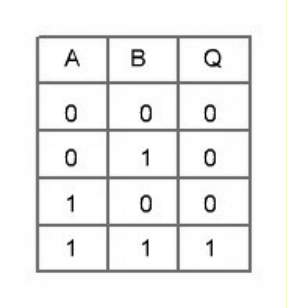
Aufgabe: Erzeugen Sie folgende Masken: 0000 0001, 1000 0000, 1000 1000

9 MASKIEREN: Löschen eines einzelnen Bits

ohne die anderen Bits in einem Register zu verändern

```
PORTB = 0b1111 1101;      //ACHTUNG! Es werden alle Bits gesetzt!
```

3. MASKE = 0b1111 1101;
4. BITWEISES UND



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Beispiel: Löschen des Bits 1

```
uint8_t mask = 0b1111 1101;
PORTB = PORTB & mask;
```

oder

```
PORTB &= 0b1111 1101;
```

10 Maske für Löschen: Einerkomplement

0b1111 1101 kann gebildet werden durch das Einerkomplement von 0b0000 0010

Vorgangsweise:

1. Maske bilden wie beim Setzen eines Bits
2. Einerkomplement (Operator „~“ in C, C++, C#)

Frage: Maske für Setzen von Bit 1 und Bit 3?

Beispiel:

```
uint8_t maske = ~ 0b0000 0010;

uint8_t maske1 = ~ _BV(PB1);

uint8_t maske2 = ~( _BV(PB3) | _BV(PB1) );   //1111 0101
```

11 MASKIEREN: Abfragen eines einzelnen Bits

11.1 Ist das Bit gesetzt?

if (PINB & _BV(PB3)) …

z.B.	PINB	0b0000 1011	
	_BV(PB3) = 1 << PB3 = 1 << 3	0b0000 1000	
	Ergebnis	0b0000 1000	**TRUE**
z.B.	PINB	0b0000 0011	
	_BV(PB3) = 1 << PB3 = 1 << 3	0b0000 1000	
	Ergebnis	0b0000 0000	**FALSE**

11.2 Ist das Bit gelöscht?

Abfragen wie vorher, ob das Bit gesetzt ist und das Ergebnis der Abfrage negieren

If (**!**(PINB & _BV(bit))) …

z.B.	PINB	0b0000 1011	
	_BV(PB3) = 1 << PB3 = 1 << 3	0b0000 1000	
	Ergebnis	0b0000 1000	**gesetzt → FALSE**
z.B.	PINB	0b0000 0011	
	_BV(PB3) = 1 << PB3 = 1 << 3	0b0000 1000	
	Ergebnis	0b0000 0000	**nicht gesetzt → TRUE**

12 Komfort-Funktionen zur Bitmanipulation stdlib.c

#include <avr/io.h> //Stdlib.c einbinden

#define bit_is_set(sfr, bit)	(_SFR_BYTE(sfr) & _BV(bit))
#define bit_is_clear(sfr, bit)	(!(_SFR_BYTE(sfr) & _BV(bit)))
#define loop_until_bit_is_set(sfr, bit)	do { } while (bit_is_clear(sfr, bit))
#define loop_until_bit_is_clear(sfr, bit)	do { } while (bit_is_set(sfr, bit))

z.B.

```
#include <avr/io.h>
...
 loop_until_bit_is_clear(PINB, PB3);
…
 if (bit_is_set(PINB, PB1)) ...
```