Alexander Lennartz, B. Eng.

Workshop:
Arduino Uno, Atmel Studio 7.0 und C

Einfache Ein- und Ausgänge
Grundlagen Timer

Krefeld, 2016

Dieses Script ist als Begleitung zu einem Workshop gedacht. Obwohl darauf geachtet wurde, dass möglichst wenige Vorkenntnisse notwendig sind sollte der Teilnehmer/Leser folgende Vorkenntnisse mitbringen:

- Grundkenntnisse in C oder einer ähnlichen Programmiersprache
- Binäre und hexadezimale Zahlen
- Grundkenntnisse Elektronik (wie schließe ich eine LED an?)
- Einfaches Englisch (viele Englische Begriffe werden der Einfachheit halber nicht übersetzt)

Hat man sich bereits mit Arduino beschäftigt, so hat man schnell genug Vorkenntnisse.

Dieses Script und zusätzliche Unterlagen, wie Programmbeispiele, können unter folgenden Link herunter geladen werden:

https://github.com/al-1/AS7-Workshop

Inhalt

1 Einrichten von Atmel Studio 1
1.1 Download 1
1.2 Verknüpfen mit Arduino IDE 1
2 Ein- und Ausgänge 5
2.1 Ein und Ausgangs Ports 5
2.1.1 Pin Mode setzen 6
2.1.2 Ausgänge benutzen 7
2.2 Neues Projekt in Atmel Studio 8
2.3 Programm Aufbau 10
2.4 Blinken 10
2.4.1 Programm Kompilieren und auf den Arduino laden 10
2.5 Mehr Blinken 12
2.6 Einfaches Auslesen von einem Eingang 13
2.7 Flankenerkennung 14
2.8 Blinken mit XOR 15
2.9 Flexibles Pinmapping mit defines 16
3 Timer 17
3.1 Grundlagen und Interrupts 17
3.2 Blinken mit Timer und Overflow Interrupt 17
3.3 PWM mit Timer 19
3.3.1 LED faden 21
3.3.2 LED Helligkeit Korrektur 22
3.4 Blinken mit compare match 24
4 Quellen 25

Abkürzungen

LSb	Least signifacant Bit – Bit mit dem geringsten Wert
MSb	Most signifacnt Bit – Bit mit dem höchsten Wert
IDE	Integrated prgramming Enviroment – Entwicklungsumgebung

1 Einrichten von Atmel Studio

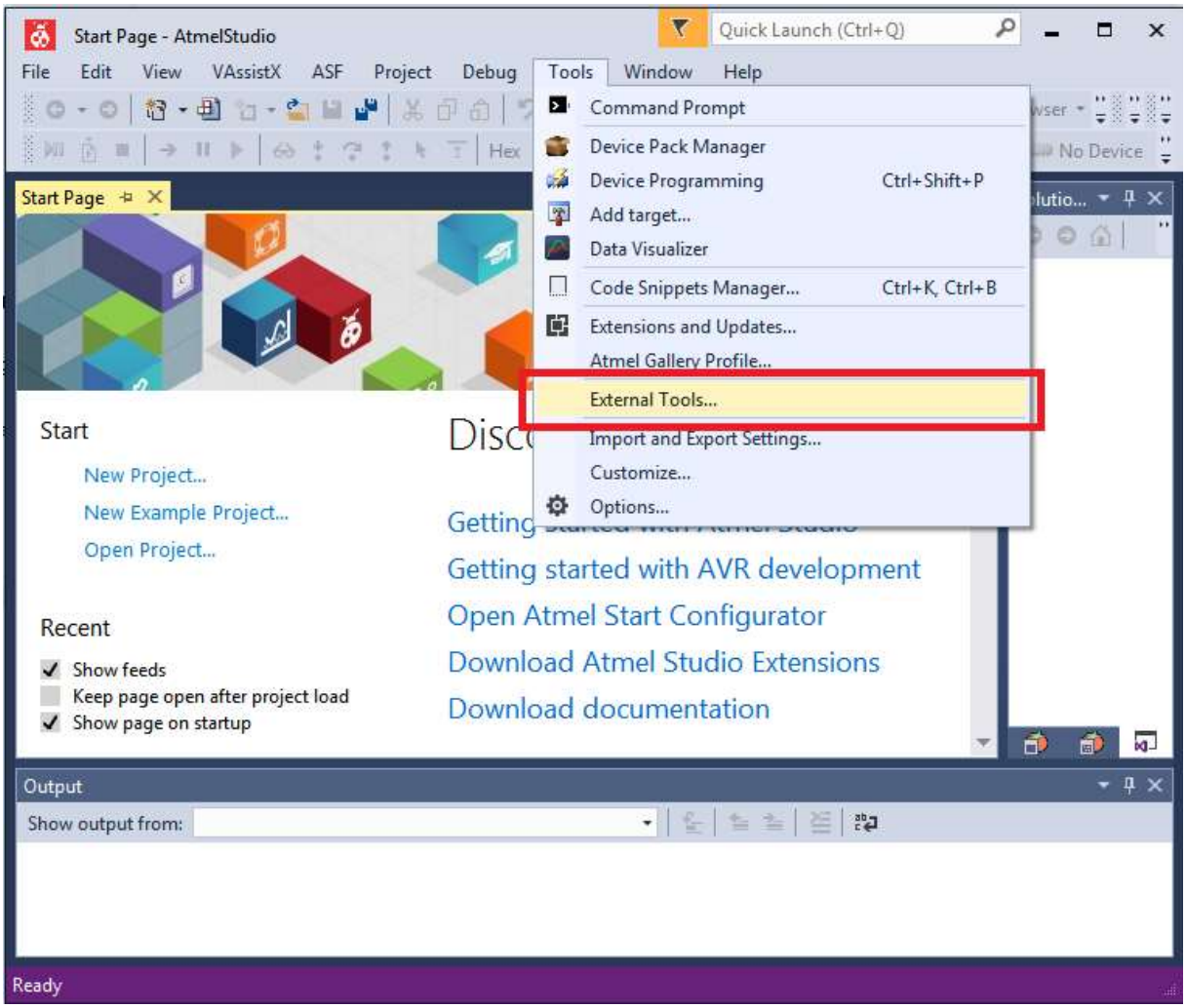
1.1 Download

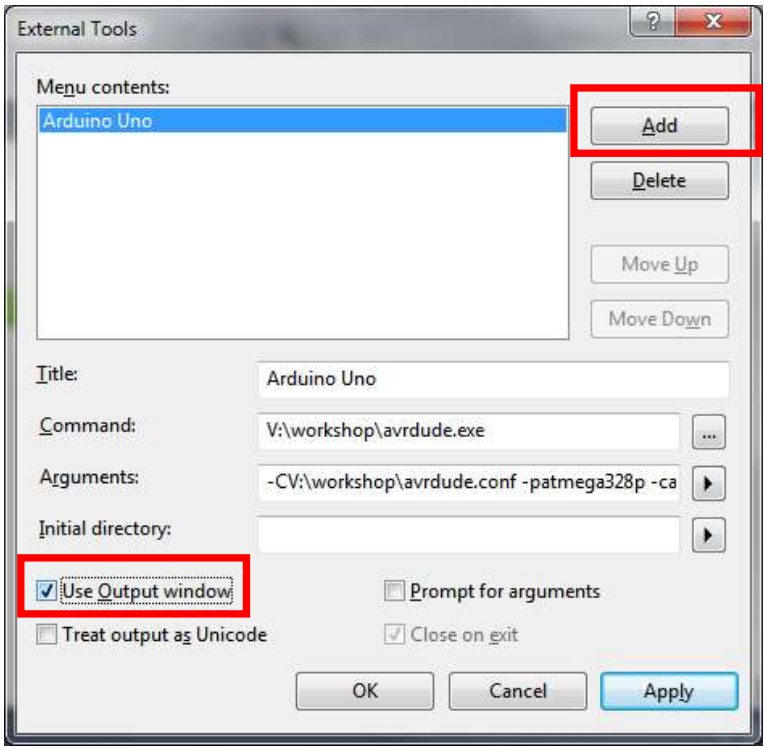
1.2 Verknüpfen mit Arduino IDE

Sucht die folgenden zwei Dateien innerhalb der Arduino Installation:

Datei	Typischer Ort
avrdude.exe	… `ArduinoInstallDir\hardware\tools\avr\bin\avrdude.exe`
avrdude.conf	… `ArduinoInstallDir\hardware\tools\avr\etc\avrdude.conf`

Kopiert beide in einen leichter zu finden Ordner. Hier z.B. „V:\workshop“. Nun öffnet Atmel Studio 7.0:

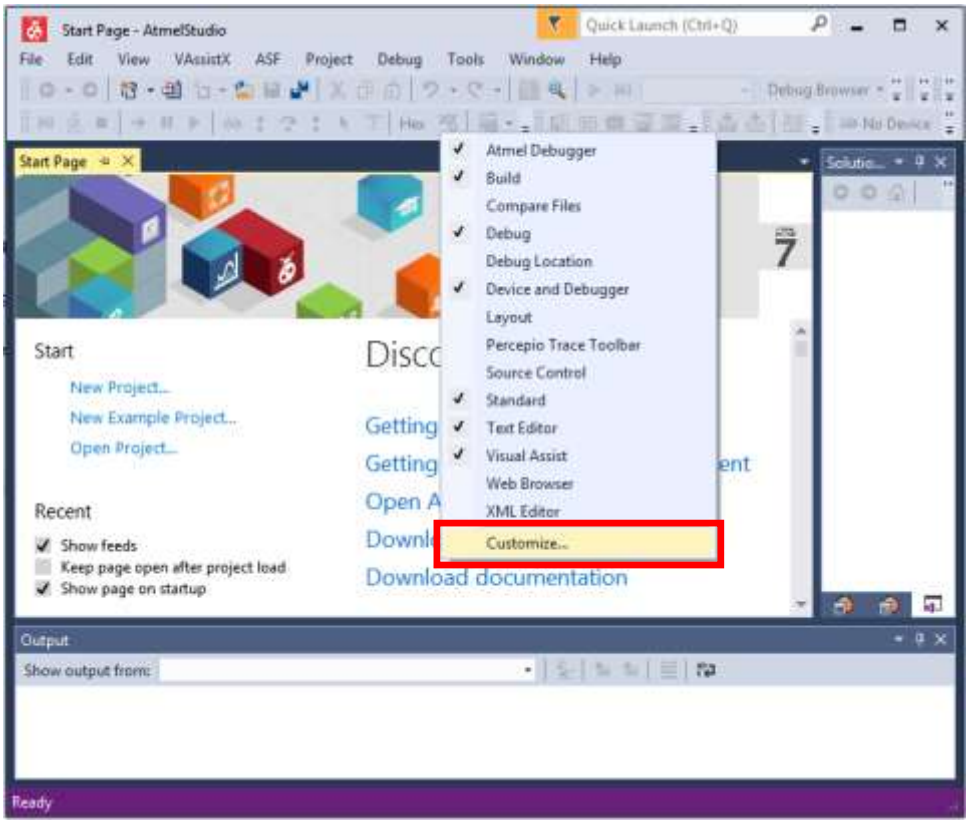


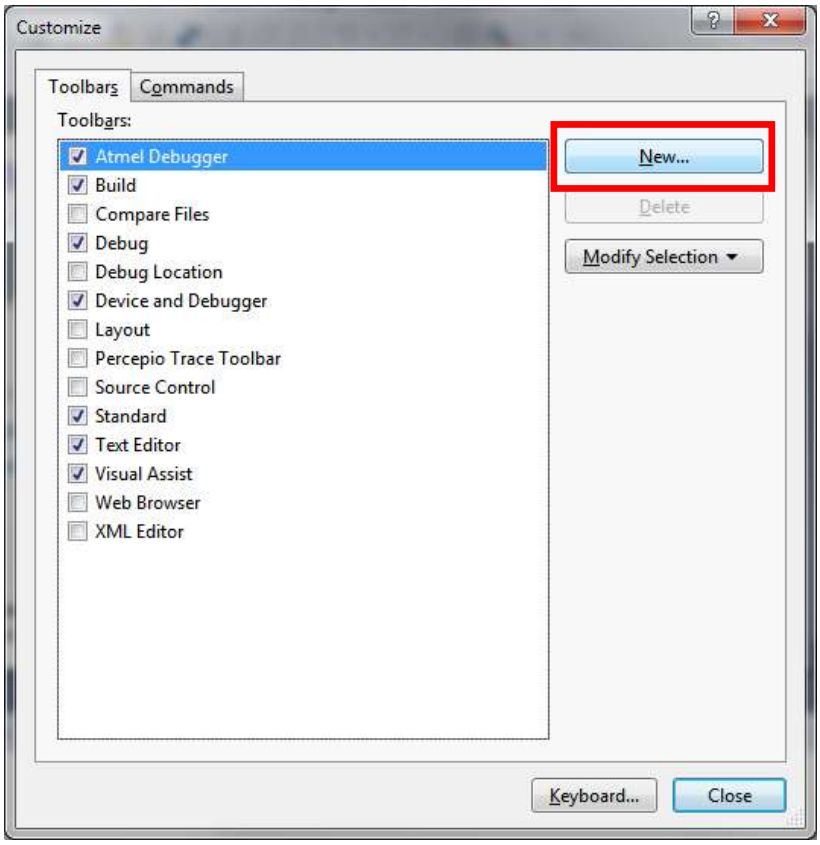


Command: `V:\workshop\avrdude.exe`

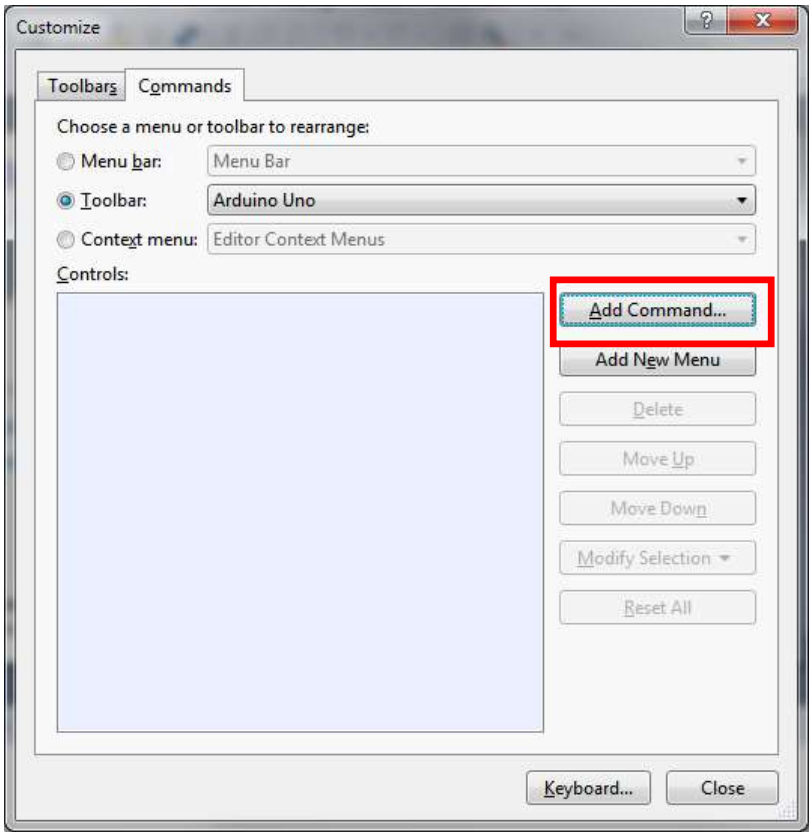
Argument: `-CV:\workshop\avrdude.conf -patmega328p -carduino -P\\.\COM14 -b57600 -Uflash:w:"$(ProjectDir)Debug\$(ItemFileName).hex":i`

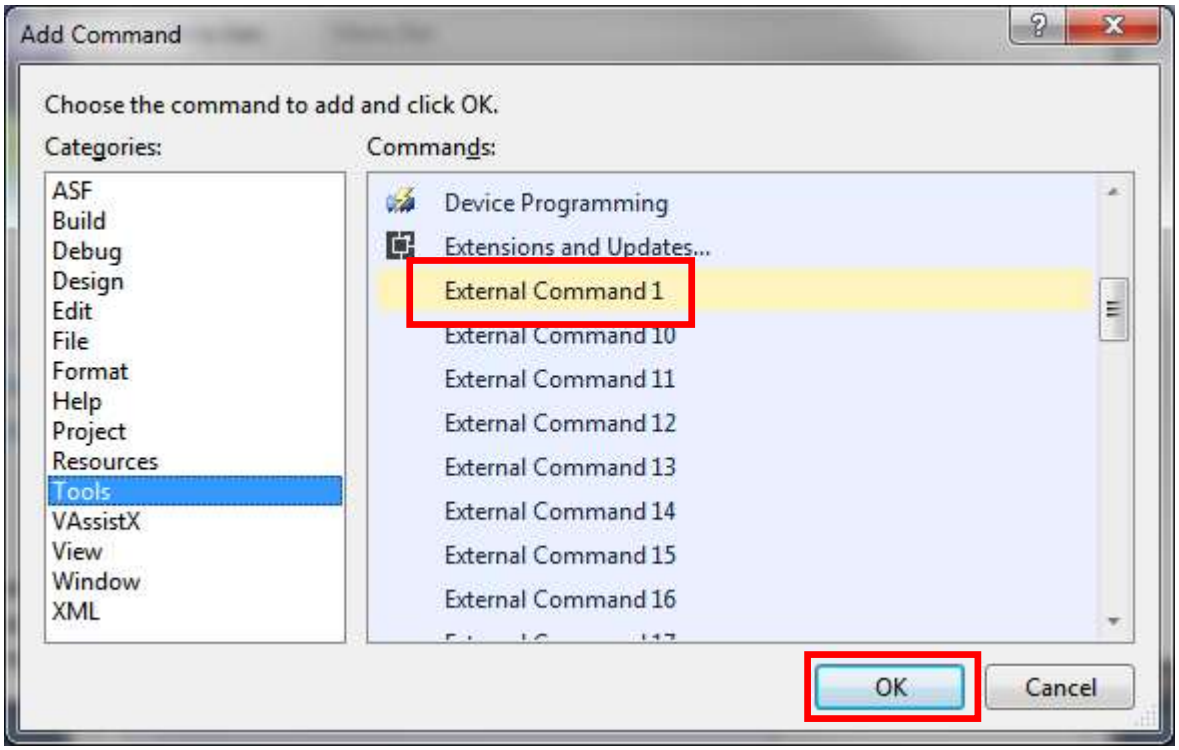
Dateipfade und COM-Port müssen ggf. angepasst werden. Nun erstellen wir uns noch einen Button zum Programmieren:



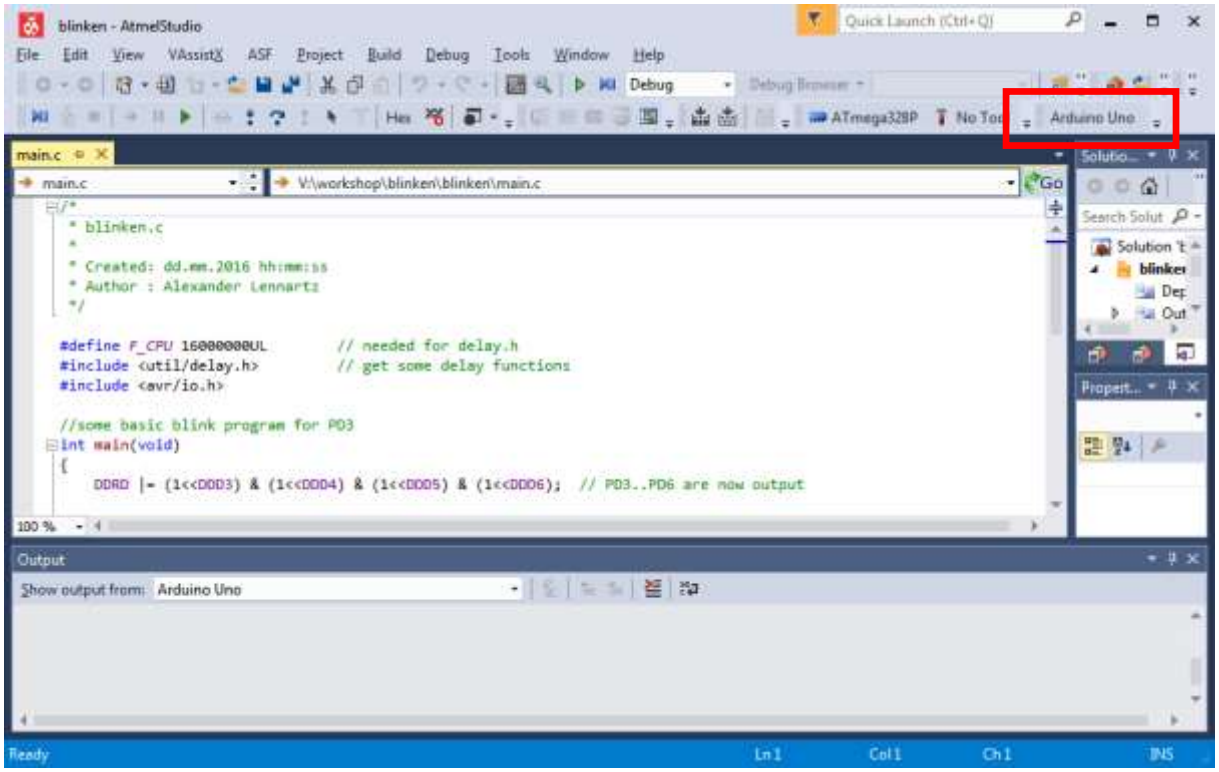


Es folgt ein Fenster welches uns nach einem Namen für unsere neue Toolbar fragt. Hier im Beispiel habe ich „Arduino Uno" gewählt. Anschließend können wir im zweiten Reiter „Commandos" unsere neue Toolbar konfigurieren:





OK, drücken Und dann auf „close“. Nun sollte es oben in Toolbar (oft leider noch etwas versteckt – muss eventuell etwas hin und her geschoben werden) ein neuer Button sein: Das ganze sollte so ähnlich aussehen:

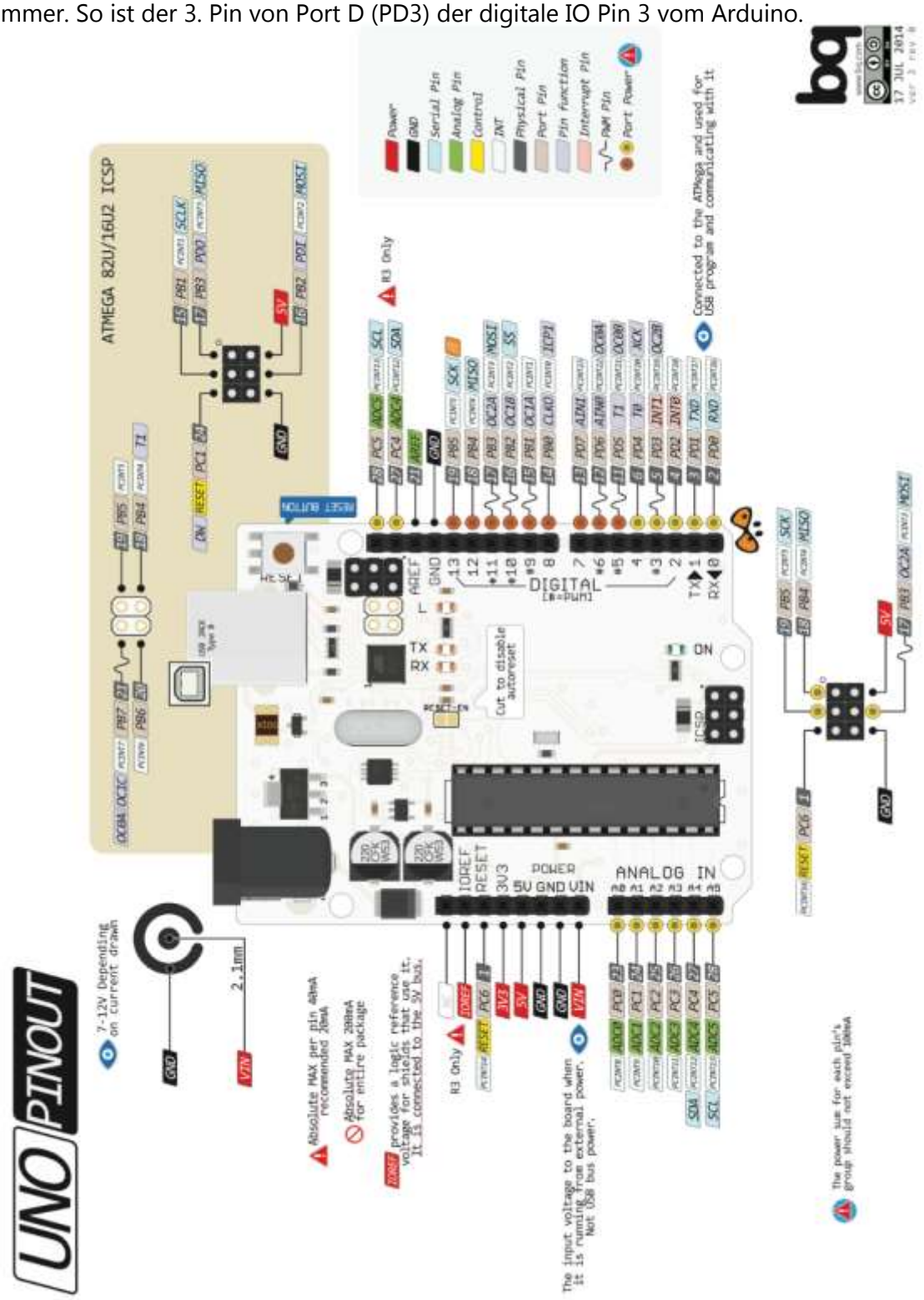


Klickt man auf ihn, hat **zuvor sein Programm kompiliert** und ist der richtige COM-Port eingestellt, so wird der Arduino Programmiert. Im unterem „output“ Fenster sollten auch die gewohnten Zeilen Text aus der Arduino IDE zu sehen sein.

2 Ein- und Ausgänge

2.1 Ein und Ausgangs Ports

Die Ein- und Ausgänge des Atmega328P sind mit verschiedenen Registern verbunden. Register sind vom Prozessor adressierbare Speicherstellen. Die Ausgänge sind zusammengefasst in verschiedene Ports. Jeder Port hat bis zu acht zugehörige Pins. Auf der folgenden Abbildung lässt sich die Zuordnung der Pins beim Arduino Uno ablesen (hell orange Kästen). Die Ports werden mit den Buchstaben D,B und C bezeichnet ihnen folgt eine Pin Nummer. So ist der 3. Pin von Port D (PD3) der digitale IO Pin 3 vom Arduino.



Zu jedem Port gehören drei verschiedene Register mit verschiedenen Aufgaben. Alle Register sind 8 Bit breit. Die Bitnummer innerhalb des Registers ist äquivalent zur Pinnummer. So ist das dritte Bit für PD3 zuständig. Die verschiedenen Register sind:

- Das Data Direction Register (DDR) bestimmt die ob ein Pin ein Ein- oder Ausgang ist
  0-> Eingang (default), 1->Ausgang
- Das Zustandsregister (PIN) gibt den logischen Zustand des Pins wieder. Egal ob Ein- oder Ausgang.
  0-> LOW/FALSE, 1-> HIGH/TRUE
- Das Port Register (PORT). Zunächst nur relevant für Ausgänge. Jedes Bit spiegelt dabei einen Ausgang wieder.
  0-> Ausgang LOW (default), 1-> Ausgang HIGH.

Diese drei Register müssen nun beschrieben werden. Macht man es richtig wird die gleiche Funktionalität wie `pinMode()` und `digitalWrite()` beim Arduino erreicht nur schneller in der Ausführung.

2.1.1 Pin Mode setzen

Bleiben wir beim Pin PD3 (Pin 3 vom Arduino). Um ihn als Ausgang zu setzten müssen wir das entsprechende Bit im DDR Register vom Port D setzen. Der Name von diesem Register ist DDRD (Data Direction Register Port D).

Bit	7 (MSb)	6	5	4	3	2	1	0 (LSb)
Portpin	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0

Um nur das entsprechende Bit zu ändern und alle anderen auf den gleichen Wert zu belassen. Kann DDRD mit einer Bitmaske logisch bitweise mit ODER verknüpft werden. In der Maske ist Dabei das entsprechende Bit gesetzt.

Bit	7	6	5	4	3	2	1	0	
Alter Wert von DDRD	u	u	u	u	u	u	u	u	u: Unbestimmt
Bitmaske	0	0	0	0	1	0	0	0	
ODER Verknüpfung	u	u	u	u	1	u	u	u	u ODER 0 = u u ODER 1 = 1

Weisen wir nun diesen Wert DDRD zu haben wir unser Ziel erreicht. PD3 wäre dann Ausgang. In C gibt es für solche bitweise Operationen eigen Befehle Hier eine kurze Zusammenfassung:

	Operation	Beispiel	Ergebnis
&	Bitweise UND	`result = 0b11000011 & 0b00111100;`	`0b11111111`
|	Bitweise ODER	`result = 0b11001100 | 0b11110000;`	`0b11111100`
<<	Bitschift nach links	`result = 3 << 6`	`0b11000000`
>>	Bitshift nach rechts	`result = 4 >> 1`	`0b00000010`
~	Bitweise NICHT	`result = ~(0b10101010)`	`0b01010101`
^	Bitweise XOR	`result = 0b11001100 ^ 0b11110000;`	`0b00111100`

Für unser kurzes Beispiel sähe es dann im code wie folgt aus:

```
uint8_t mask = (1<<3);            // Erzeuge eine Maske mit einer 1 an der Stelle drei
//uint8_t mask = 0b00001000       // schlechter lesbare Alternative
DDRD = DDRD | mask;               // weise DDRD den Wert von DDRD bitweise ODER der
                                  Maske zu

DDRD|=(1<<3)                      // macht genau das gleiche nochmal nur mit weniger
                                  Tipp Arbeit (inklusive der Masken Erzeugung)

DDRD|=(1<<DDD3);                  // alternative mit Benutzung von eines defines DDDn
                                  ist ein Bit von DDRD
```

2.1.2 Ausgänge benutzen

Nun da wir PD3 als Ausgang gesetzt haben müssen wir ihn Ansteuern. Zum anschalten müssen wir das entsprechende Bit im PORTD Register setzen. Das geht vom Prinzip genau wie eben nur mit einem anderen Register:

```
PORTD |= (1<<3);
PORTD |= (1<<PORTD3);             // Äquvalent aber besser zu lesen und "sauberer"
```

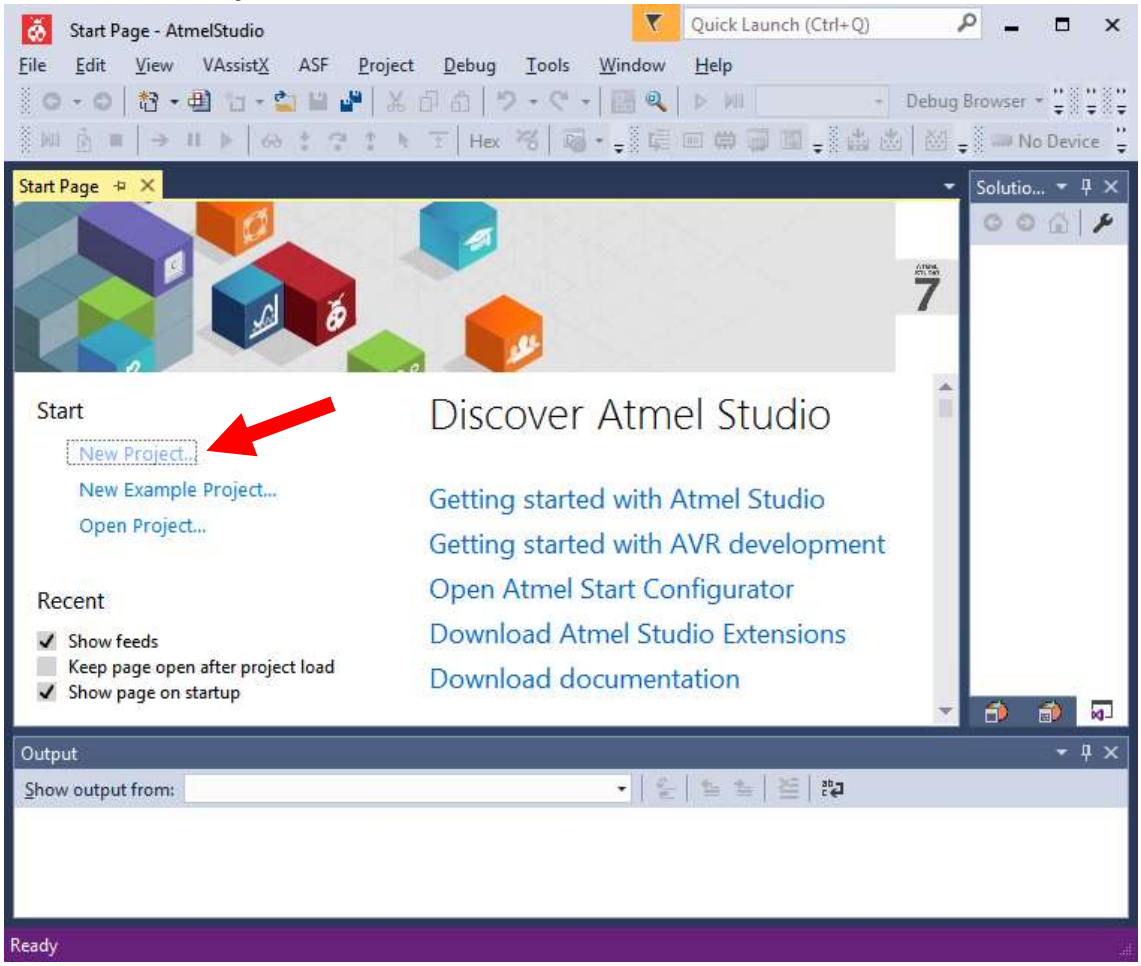
Um nun eine Bit in einem Register zurückzusetzen (auf null setzen) ist eine ein wenig komplizierteres Verfahren notwendig:

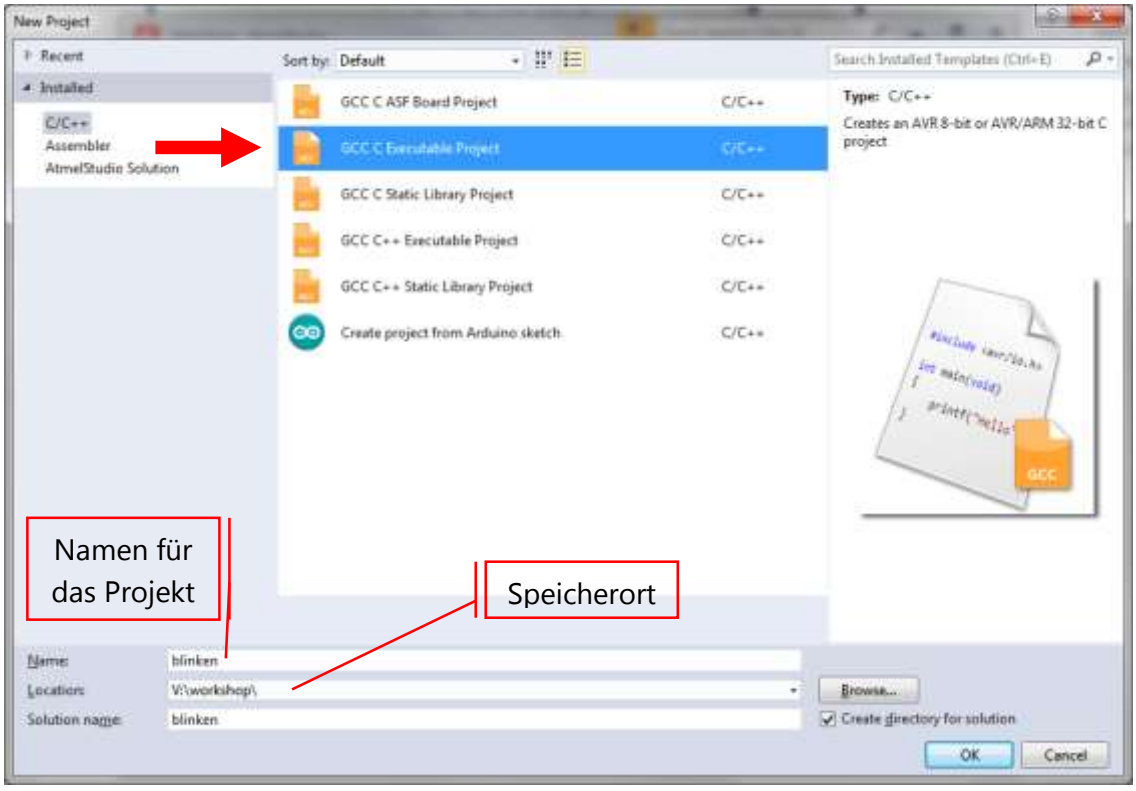
Bit	7	6	5	4	3	2	1	0	
Alter Wert von PORTD	u	u	u	u	1	u	u	u	u: Unbestimmt
Bitmaske	0	0	0	0	1	0	0	0	
NICHT Bitmaske (~)	1	1	1	1	0	1	1	1	
UND Verknüpfung (~Bitmaske & alter Wert)	u	u	u	u	0	u	u	u	u UND 1 = u 0 UND 1 = 0

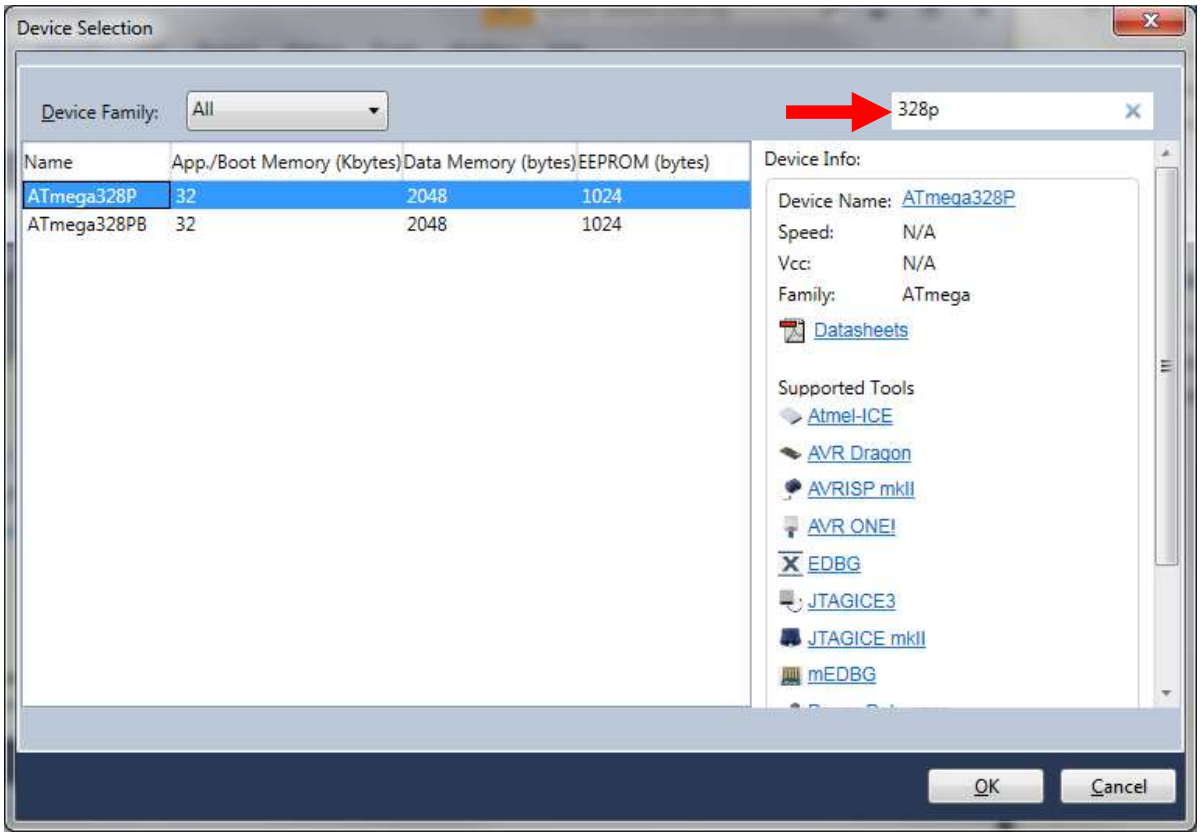
Im C sieht das Ganze dann wie folgt aus:

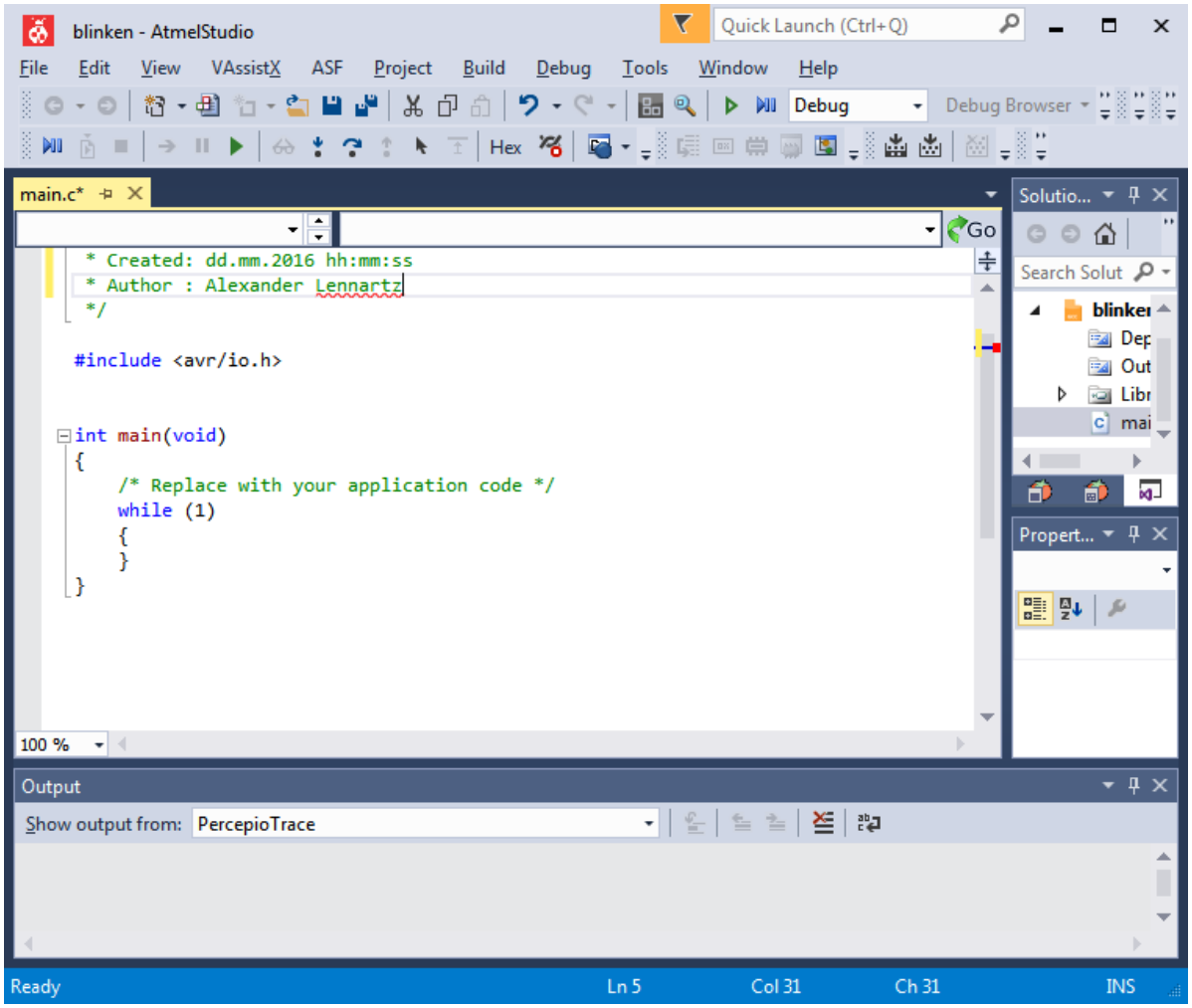
```
PORTD &=~(1<<PORTD3);             // schalte PD3 aus
```

2.2 Neues Projekt in Atmel Studio






Device Selection
Device Family:
All
328p
Name
App./Boot Memory (Kbytes)
Data Memory (bytes)
EEPROM (bytes)
ATmega328P
32
2048
1024
ATmega328PB
32
2048
1024
Device Info:
Device Name: ATmega328P
Speed: N/A
Vcc: N/A
Family: ATmega
Datasheets
Supported Tools
Atmel-ICE
AVR Dragon
AVRISP mkII
AVR ONE!
EDBG
JTAGICE3
JTAGICE mkII
mEDBG
OK
Cancel


blinken - AtmelStudio
Quick Launch (Ctrl+Q)
File Edit View VAssistX ASF Project Build Debug Tools Window Help
Debug
Debug Browser
Hex
main.c*
Go
* Created: dd.mm.2016 hh:mm:ss
* Author : Alexander Lennartz
*/
#include <avr/io.h>
int main(void)
{
/* Replace with your application code */
while (1)
{
}
}
100 %
Solutio...
Search Solut
Propert...
Output
Show output from: PercepioTrace
Ready
Ln 5
Col 31
Ch 31
INS

2.3 Programm Aufbau

Bevor wir nun selbst ein kleines Testprogramm schreiben kurz noch eine Sache, die wir benötigen. Ähnlich wie bei Arduino sketchen haben wir zwei Bereiche: Einen Bereich der nur einmalig zu Beginn ausgeführt wird (vergleichbar zu `setup()`) und einen der zyklisch wiederholt wird. (vergleichbar mit `loop()`). Im code sieht das Ganze dann so aus:

```
int main(void)
{
	//"setup()"
	DDRD |= (1<<DDD3);		// PD3-> Ausgang

    while (1)
    {  //"loop()"
		…
    }
}
```

2.4 Blinken

Nun schreiben wir ein kleines Programm um eine LED, die an PD3 angeschlossen ist zum Blinken zu bringen, dafür nutzen wir die zuvor kennengelernten Methoden um DDRD und PORTD zu ändern. Zusätzlich brauchen wir noch eine delay Funktion. Diese können wir recht einfach einbinden. Wir müssen einfach

```
#define F_CPU 16000000UL    // needed for delay.h – CPU frequency in Hz
#include <util/delay.h>     // get some delay functions
```

am Begin ergänzen und können nun mit

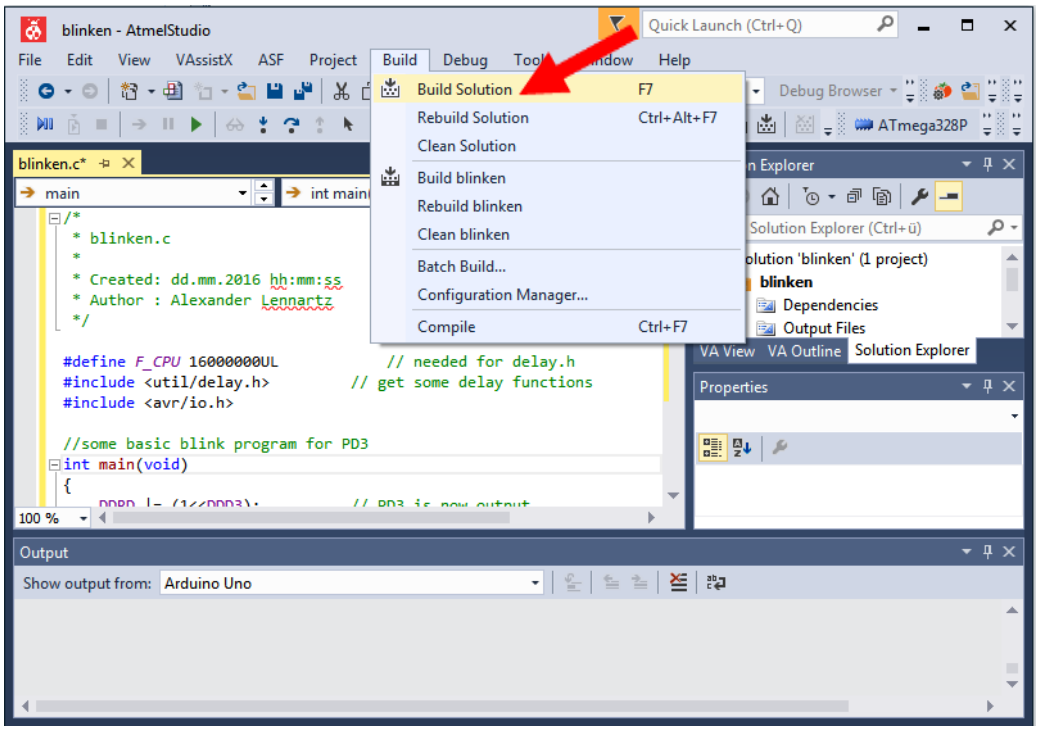
```
_delay_ms(500);
```

z.B. eine halbe Sekunde Warten.

2.4.1 Programm Kompilieren und auf den Arduino laden

Falls eure C-Datei links im „SolutionExplorer“ „main.c“ heißt. Nennt sie bitte genauso wie das Projekt. (so wie den übergeordneten Ordner)

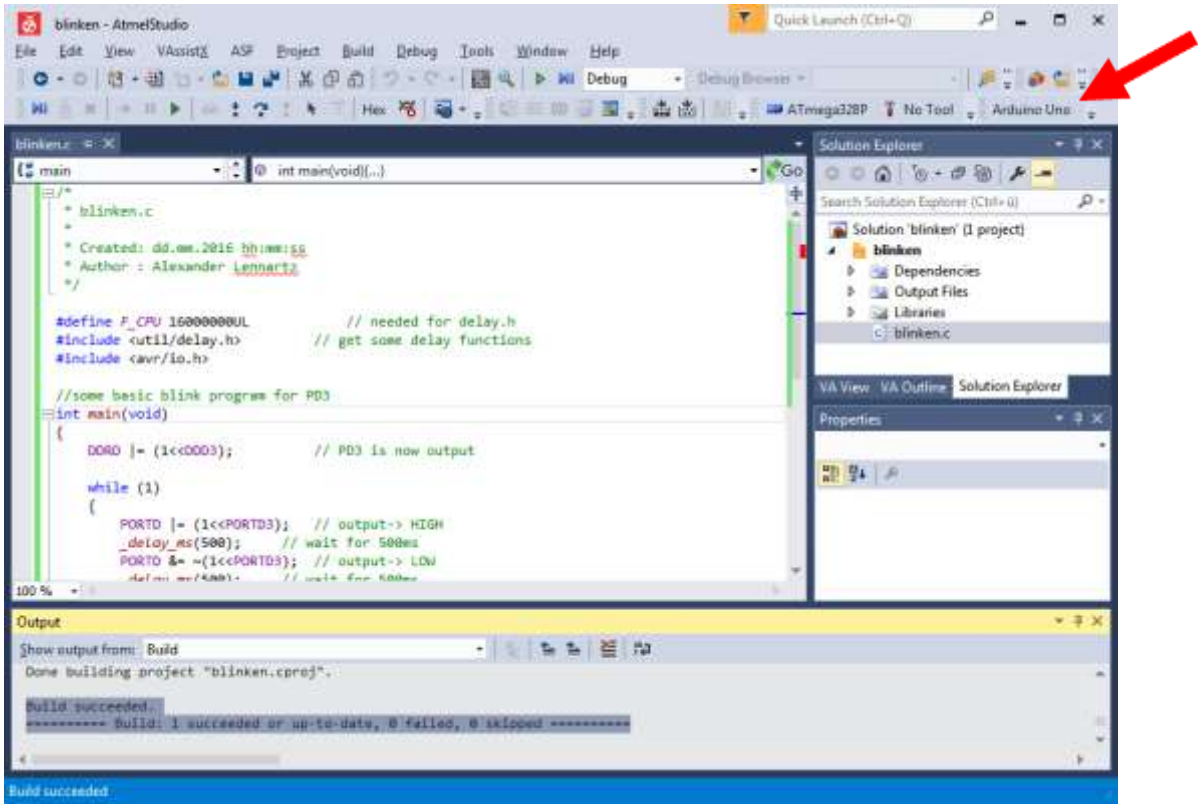
Um unser Programm auf den Arduino zu laden müssen wir es zunächst kompilieren. Entweder über das Menü wie auf dem Bild oder einfach F7 drücken.



Haben wir keinen Fehler gemacht kommt in dem „output" Fenster folgende Meldung:

```
Build succeeded.
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Nun können wir mit dem in Kapitel 1 eingerichtetem Button das Programm auf den Arduino laden



Auch hier erhalten wir im „output" Fenster unten wieder eine Rückmeldung. Ihr Ende sollte so in etwa aussehen:

```
avrdude.exe: verifying ...
avrdude.exe: 176 bytes of flash verified

avrdude.exe: safemode: Fuses OK (H:00, E:00, L:00)
```

Lösung:

```
/*
 * blinken.c
 *
 * Created: dd.mm.2016 hh:mm:ss
 * Author : Alexander Lennartz
 */

#define F_CPU 16000000UL                // needed for delay.h
#include <util/delay.h>                 // get some delay functions
#include <avr/io.h>

//some basic blink program for PD3
int main(void)
{
    DDRD |= (1<<DDD3);                  // PD3 is now output

    while (1)
    {
            PORTD |= (1<<PORTD3);       // output-> HIGH
            _delay_ms(500);             // wait for 500ms
            PORTD &= ~(1<<PORTD3);      // output-> LOW
            _delay_ms(500);             // wait for 500ms
    }
}
```

2.5 Mehr Blinken

Ein großer Vorteil von diesem direkten Zugriff auf Register ist, dass mehrere Bits auf einmal geändert werden können. Beispiel:

```
DDRD |= (1<<DDD3) | (1<<DDD4);          // PD3 and PD4 are now output
```

Gleiches gilt für die Port Register:

```
PORTD |= (1<<PORTD3) | (1<<PORTD4);     // outputs-> HIGH
PORTD &= ~( (1<<PORTD3) | (1<<PORTD4) ); // outputs-> LOW
                                        take care of outer bracket needed here
```

Aufgabe: Lasse insgesamt vier LEDs (angeschlossen an Pins von Port D) blinken. Mit der gleichen Frequenz aber in Gruppen von zwei abwechselnd.

Lösung:

```
/*
 * blinken.c
 *
 * Created: dd.mm.2016 hh:mm:ss
 * Author : Alexander Lennartz
 */

#define F_CPU 16000000UL          // needed for delay.h
#include <util/delay.h>           // get some delay functions
#include <avr/io.h>

//some basic blink program for PD3
int main(void)
{
       DDRD |= (1<<DDD3) | (1<<DDD4) | (1<<DDD5) | (1<<DDD6);
       // PD3..PD6 are now output

    while (1)
    {
              PORTD |=    (1<<PORTD3) | (1<<PORTD4);   // PD3/PD4 -> HIGH
              PORTD &= ~( (1<<PORTD5) | (1<<PORTD6) ); // PD5/PD6 -> LOW
              _delay_ms(500);                          // wait for 500ms
              PORTD |=    (1<<PORTD5) | (1<<PORTD6);   // PD5/PD6 -> HIGH
              PORTD &= ~( (1<<PORTD3) | (1<<PORTD4) ); // PD3/PD4 -> LOW
              _delay_ms(500);                          // wait for 500ms
    }
}
```

2.6 Einfaches Auslesen von einem Eingang

Ähnlich wie Ausgänge können wir mit Eingängen umgehen. Nur benötigen wir nun nur das PIN Register. In das DDR Register müsste zwar eine null reingeschrieben werden, damit ein Pin Eingang ist, dies ist allerding der Normalzustand. Kurzes Beispiel:

```
uint8_t myInput = PIND & (1<<PIND3); // read input pin and write value to variable
```

Die variable „myInput“ (vom typ uint8_t) ist nun ungleich null falls am Eingangspin PD3 eine logische eins anlag.

Aufgabe: Steuere über einen Eingang eine LED an. Die LED soll immer leuchten wenn am Eingang eine logische null /LOW Pegel anliegt.

Lösung

```
#include <avr/io.h>


int main(void)
{
	uint8_t myInput;
	DDRD |= (1<<DDD2);                          // output pin for LED

	while (1)
	{
		myInput = PIND & (1<<PIND3);
                    // read input pin and write value to variable
		if (myInput)                            // input==HIGH
		{
			PORTD &= ~(1<<PORTD2);      // LED off
		}
		else
		{
			PORTD |= (1<<PORTD2);       // LED on
		}
	}
}
```

Nun könntet ihr festgestellt haben, dass sich das Ganze unerwartet oder nicht immer gleich verhält, wenn Garnichts an den Eingang angeschlossen ist. Dies liegt daran das dann der Zustand des Einganges nicht sauber bestimmt ist. Dies löst man durch einen Widerstand (in der Größenordnung von 10kOhm). Der Widerstand wird mit 5V (HIGH) **oder** der Masse (LOW) verbunden. So gibt zumindest immer der Widerstand einen Eingangszustand vor. Dieser kann aber leicht durch ein externes Signal „überschrieben" werden. In unserem Mikrocontroller haben wir die Möglichkeit solch einen Wiederstand intern zu Aktivieren. Dafür müssen wir einfach ins PORT Register für den entsprechenden Eingang eine Eins schreiben:

```
PORTD |= (1<<PORTD3);                    // enable internal pullup
```

Haben wir dies ergänzt, sollte unsere LED nur noch leuchten wenn der Eingangspin mit der Masse verbunden ist.

2.7 Flankenerkennung

(optional)

In diesem Beispiel/Aufgabe wird gezeigt, wie mit Hilfe einer Flankenerkennung eine LED an- und ausgeschaltet werden kann. Dafür wird an einen Eingang ein Taster angeschlossen: Der Taster schaltet den Eingang wenn gedrückt auf Masse. Bei jedem kurzen Drücken vom Taster, soll die LED ihren Zustand wechseln. Hier wird eine fallen oder negative Flanke detektiert.

Das Prinzip ist folgendes: Es wird eine Variable für einen alten Tasterzustand angelegt. Dann wird zyklisch ein neuer Tasterzustand eingelesen - haben die beiden Tasterzustände die gewünschte Flanke wird die LED bedient. Anschließend wird immer der alte Tasterzustand mit dem Neuen überschrieben und es beginnt von vorne.

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

int main(void)
{
    DDRD  |= (1<<DDD2);           // PD2 output
       PORTD |= (1<<PORTD3);      // enable pullup on PD3

       uint8_t oldIn=1;           // variable for old input state
       uint8_t newIn;             // variable for new input state
       uint8_t led=0;             // variable for LED state

    while (1)
    {
              newIn = PIND & (1<<PIND3); // read new input
              if (newIn && !(oldIn) )    // if newIn==1 and oldIn==0
              {
                     if (led==0)                        // if led is off
                     {
                            led=1;
                            PORTD |=(1<<PORTD2);        // led on
                     }
                     else
                     {
                            led=0;
                            PORTD &= ~(1<<PORTD2);      // led off
                     }
              }
              oldIn=newIn;
              _delay_ms(10);                            // some delay
    }
}
```

2.8 Blinken mit XOR

Beim letzten und auch bei vorherigen Beispiel wäre es sinnvoll gewesen den Zustand von einem Ausgang mit nur einem Befehlt zu wechseln. Dies ist auch möglich mit einem exklusiven ODER (XOR)

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

a XOR 1 ergibt immer a invertiert. In C wird XOR als „^“ geschrieben:

```
…
int main(void)
{
    DDRD |= (1<<DDD7);                  //PD7 output
    while (1)
    {
              PORTD ^= (1<<PORTD7);     //toggle pin
              _delay_ms(200);
    }
}
```

2.9 Flexibles Pinmapping mit defines

(optional)

Will man sein Pinmmapping ein weniger flexibler gestellten, kann man sich mit defines ein flexibles Pinmapping aufbauen. Der Vorteil ist, dass wenn nachträglich der Pin z.B. einer LED geändert werden soll, muss nur an einer Stelle etwas am Programm geändert werden. Hier ein kurzes Beispiel, es müsste nur oben an den defiens etwas geändert werden, um der LED einen anderen Pin zuzuweisen:

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

#define LED_DDR DDRD                    // change only these three defines
#define LED_PORT PORTD                  // to remap the LED pin
#define LED_BIT 7

int main(void)
{
    LED_DDR |= (1<<LED_BIT);            //LED pin output
    while (1)
    {
              LED_PORT ^= (1<<LED_BIT);  //toggle LED
              _delay_ms(200);
    }
}
```

3 Timer

In diesem Zweiten Abschnitt des Workshops werden wir uns mit einem der Peripheriemodule beschäftigen. Exemplarisch ist hier einer der drei Timerbausteine (timer0) ausgewählt wurden. Mit den Timer Bausteinen können verschiede zeitgesteuerte Abläufe realisiert werden.

3.1 Grundlagen und Interrupts

Vereinfacht beschrieben handelt es sich bei den Timern um Zählbausteine. Jedes Mal wenn sie ein Taktsignal bekommen zählen sie ihren Zählerwert um eins hoch. Dabei ist ihre Taktquelle konfigurierbar. Sie kann zum Beispiel der CPU Takt oder ein Teiler des CPU Taktes sein.

Der hier behandelte Timer ist ein 8-Bit Timer. Das heißt, dass sein Zählerwert 8-Bit breit ist und so maximal den Wert 255 annehmen kann. Ist dieser erreicht läuft der Zähler mit dem nächsten Takt über und beginnt wieder bei 0. Die Timer können zu verschiedenen Zählerstände Interrupts auslösen. Interrupts sind Prozesse, die unabhängig vom normalen Programm Ablauf ausgeführt werden. Sie werden so zu sagen da zwischen geschoben.

Mit Hilfe des sogenannten Overflow-Interrupts, der auftritt wenn der Zähler von 255 auf 0 springt kann ein blink Programm entwickelt werden, dass ohne die delay-Funktion auskommt. Dies hat den entschieden Vorteil, dass zwischen durch die CPU für weitere Aufgaben frei bleibt.

3.2 Blinken mit Timer und Overflow Interrupt

Zunächst müssen wir den Timer0 konfigurieren. Der Timer lässt sich über verschiedene Register einstellen:

Register	Bedeutung
TCCR0A	Timer0 Control Register A
TCCR0B	Timer0 Control Register B
TCNT0	Timer counter Register - Zählerwert
OCR0A	Output compare Register A/B
OCR0B	
TIMSK0	Timer Interrupt Mask Register

Für unser erstes Programm brauchen wir zunächst nur die Register TCCR0B und TIMSK0. Dazu hier zunächst der Aufbau vom TCCR0B Register aus dem Datenblatt:

TCCR0B – Timer/Counter Control Register B

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Von den Einzelnen Bits interessiert uns zunächst auch nur CS02..CS00. Diese bestimmen die Taktquelle für unseren Timer. Dazu gibt es wiederum auch im Datenblatt eine kleine Tabelle:

Table 15-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

$Clk_{I/O}$ entspricht im Normalfall der CPU Frequenz. Haben wir nun das Ziel eine LED zum Blinken zu bringen müssen wir erreichen, dass der Timer so überlauft, dass das blinken der LED noch zu sehen ist. (Was hier das Hauptproblem darstellt, ist das menschliche Auge, welches im Vergleich zu einen Mikrocontroller extrem langsam ist). Es gelten ein paar Formeln:

$$f_{overflow} = \frac{f_{CPU}}{N * 256}, \quad T_{owerflow} = \frac{N * 256}{f_{CPU}} \quad N: prescaler[1, 8, 64, 256, 1024]$$

Für N=1024 läuft der Timer alle 16ms über. Dies ergibt schon ein recht schnelles Blinken, reicht aber zu demonstrationszwecken aus.

Im code sieht das Beschreiben von TCCRB0 dann so aus:

```
TCCR0B |= ((1<<CS02) | (1<<CS00));       // prescaler: 1024
```

Das zweite Register ist TIMSK0. In ihm legt man fest, welcher Interrupt (ein Timer hat mehrere) ausgelöst werden kann:

TIMSK0 – Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Uns interessiert der overflow Interrupt, welcher hier durch das Bit TOIE0 (timer overflow interrupt enable 0) dargestellt wird. Mit dem folgenden Code setzen wir das Bit und aktivieren so den Interrupt:

```
TIMSK0 |= (1<<TOIE0);                    // enable overflow interrupt
```

Um das gesamte Programm zu erhalten brauchen wir noch die Interrupt Service Routine. Dies ist der Programmteil der aufgerufen wird wenn der Interrupt ausgelöst wird. Auch müssen wir `interrupt`.h einbinden und Interrupts global aktivieren mit `sei()`. Zusammen gesetzt sieht der code dann so aus:

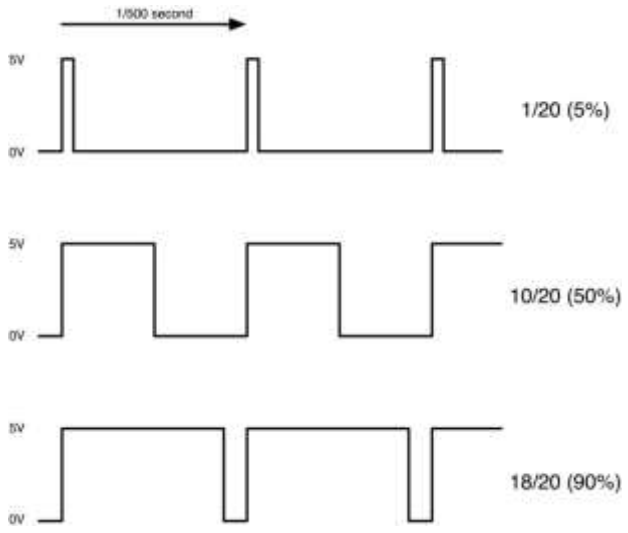
```c
#include <avr/io.h>
#include <avr/interrupt.h>

int main(void)
{
	DDRD |= (1<<DDD7);				// LED pin output
	TCCR0B |= ((1<<CS02) | (1<<CS00));// prescaler: 1024
	TIMSK0 |= (1<<TOIE0);			// enable overflow interrupt
	sei();						// enable interrupts

	while (1)
	{
		;						// nothing to do (yet)
	}
}

ISR(TIMER0_OVF_vect)				// timer0 overflow interrupt service routine
{
	PORTD ^= (1<<PORTD7);			//toggle LED
}
```

Der Programmteil innerhalb der ISR (hier nur das wechseln vom LED Zustand) wird jedes Mal wenn der Interrupt auftritt ausgeführt. Dies geschieht asynchron zum normalen Programmablauf (hier das ausführen der Endlosschleife).

Die Timer sind generell weniger gedacht um LEDs blinken zu lassen. Mit ihnen können aber z.B. Messwerte zyklisch erfasst und weiter verarbeitet werde. Auch können Ausgangssignale (wie z.B. ein Rechtecksignal) erzeugt werden. Eine weitere Anwendung zeigt das nächste Kapitel.

3.3 PWM mit Timer

Eine sehr weit verbreitete Anwendung von Timern ist die Erzeugung sogenannter PWM-Signale. Mit ihnen können Verbraucher wie LEDs gedimmt werden. Dabei wird der Verbraucher nicht die ganze Zeit, sondern nur zu einem Prozentanteil angeschaltet. Wird das ganze schneller gemacht, als das menschliche Auge es z.B. es bei LEDs sehen kann, erscheint die LED gedimmt.



Bildquelle: learn.adafruit.com (CC BY 3.0)

Diesmal brauchen wir zur Konfiguration des Timers zwei verschiedene Register:

- TCCR0A zur PWM Konfiguration
- TCCR0B um einen Eingangstakt zu wählen

Da wir diesmal ohne Interrupts auskommen, brauchen wir das TIMSK0 Register nicht mehr. Zunächst zu den Optionen im TCCR0A Register (Abbildung aus dem Datenblatt):

TCCR0A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Relevant sind hier die folgenden Bits: COM0A1/2 und WGM01/00. Zunächst zu den WGM01/0 Bits:

Table 15-8. Waveform Generation Mode Bit Description

Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on(1)(2)
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Wir nehmen hier den Modus 3. Zuvor beim Blinken hatten wir den Modus 0 und mussten uns deswegen nicht um diese Einstellung kümmern. Zu den COM0A1/0 Bits bestimmen wie sich einzelne IO-Pins die direkt mit dem Timer gekoppelt werden verhalten sollen. Abhängig vom Modus gibt's es auch dazu im Datenblatt Tabellen:

Table 15-3. Compare Output Mode, Fast PWM Mode(1)

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	WGM02 = 0: Normal Port Operation, OC0A Disconnected. WGM02 = 1: Toggle OC0A on Compare Match.
1	0	Clear OC0A on Compare Match, set OC0A at BOTTOM, (non-inverting mode).
1	1	Set OC0A on Compare Match, clear OC0A at BOTTOM, (inverting mode).

Wir wählen hier sinnvoller Weise die Einstellungen [1, 0]. Dieser bringt uns dann ein PWM-Signal im non-inverting-mode an den Pin OC0A. Ein Blick auf das Pinout vom Anfang sagt uns, dass OC0A mit PD6 einen Pin teilt. Wir erzeugen also ein PWM Signal an diesen Pin.

Neben TCCR0A müssen wir wieder TCCR0B beschreiben. Hier wählen wir wieder einen Taktquelle für den Timer. Diesmal nehmen wir aber einfach eins zu eins den CPU Takt um unseren Timer mit der maximalen Frequenz laufen zu lassen.

Im code ergeben sich dann zur Konfiguration dann die folgenden Zeilen:

```
TCCR0A |= ( (1<<WGM01) | (1<<WGM00) );   // Set Mode 3 (fast PWM)
TCCR0A |=   (1<<COM0A1);                 // OCOA non inverting PWM output
TCCR0B |=   (1<<CS00);                   // prescaler=1 / use f_CPU
```

Den Duty cycle, also das Verhältnis zwischen an und aus, können wir nun über das sogenannte output compare Regsiter (OCR0A) festlegen. Zu jedem PWM Ausgang gibt es ein solches Register. (Der Timer0 hätte noch einen zweiten PWM Ausgang zu Verfügung). OCR0A kann mit Werten von 0 (=0%) bis 255 (=100%) beschrieben werden.

Führen wir mal etwas Code zusammen:

```
#include <avr/io.h>

int main(void)
{
       DDRD   |=   (1<<DDD6);                  // set OC0A pin to output
       TCCR0A |= ( (1<<WGM01) | (1<<WGM00) );  // Set Mode 3 (fast PWM)
       TCCR0A |=   (1<<COM0A1);                // OCOA non inverting PWM output
       TCCR0B |=   (1<<CS00);                  // prescaler=1 / use f_CPU
       OCR0A   = 127;                          // set duty cycle to 50%
    while(1)
    {
        ;
    }
}
```

3.3.1 LED faden

Aufgabe: Ergänze das Programm, so dass die LED langsam hochgedimmt wird. Dazu kann in der while Schleife langsam OCR0A erhöht werden. Nutze die delay Funktion um das Ganze noch sichtbar zu lassen.

Lösung:

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

int main(void)
{
	DDRD   |=   (1<<DDD6);                          // set OC0A pin to output
	TCCR0A |= ( (1<<WGM01) | (1<<WGM00) );          // Set Mode 3 (fast PWM)
	TCCR0A |=   (1<<COM0A1);                        // OCOA non inverting PWM output
	TCCR0B |=   (1<<CS00);                          // prescaler=1 / use f_CPU
	OCR0A   = 0;                                    // set duty cycle to 0%
	while(1)
	{
		OCR0A = OCR0A +1;                       // increment OCR0A
		_delay_ms(10);

	}
}
```

3.3.2 LED Helligkeit Korrektur

(optional)

Beim letzten Beispiel könnte man sehen, dass die LED für das menschliche Auge sich nicht linear verhalten hat. Dies liegt daran, dass Licht nicht linear wahrgenommen wird. Mit folgender Formel kann eine Korrektur vorgenommen werden:

$$R[i] = round\left((z-1) * \left(\frac{i}{n-1}\right)^{\frac{1}{y}} \right)$$

Mit n: Anzahl der Diskreten Werte
i: Laufindex [0..n-1]
z: Anzahl der PWM-Stufen (in unserem Fall 256)
y: Gammakorrekturwert ~[0,5..0,33]

Für n wird jetzt der Wert 100 gewählt. So erhalten wir 100 verschiedene Helligkeitswerte. Mit Zahlenwerten (y=1/2,2=0,454 Wert wird bei VGA Monitoren genutzt) ergibt sich dann:

$$R[i] = round\left(255 * \left(\frac{i}{99}\right)^{2,2} \right)$$

Funktionen für round() und den Exponenten stehen uns in der Bibliothek `math.h` zu Verfügung. Wir binden sie mit `#include <math.h>` zu Beginn ein. Wir können dann die Funktionen `round()` und `pow(a,b)` (=a^b) Nutzen

Aufgabe: Schreibe ein Programm welches die LED nun auch für das menschliche Auge linear faden lässt. Dazu kann die Korrekturberechnung gut in einer einzelnen zusätzlichen Funktion umgesetzt werden.

Eine Lösung:

```
#define F_CPU 16000000
#include <util/delay.h>
#include <avr/io.h>
#include <math.h>

uint8_t correction(uint8_t index)
{
	float temp=index;                       // now is i as float in temp
	temp = temp/99      ;                   // temp=i/99
	temp = pow(temp, 2.2);                  // (i/99)^2,2
	temp = round(255*temp);

	return (uint8_t)temp;                   //typecast the result
}

int main(void)
{
	uint8_t  led_value=0;                   // variable for the led brightness
	int8_t   led_increment=1;

	DDRD   |=   (1<<DDD6);                  // set OC0A pin to output
	TCCR0A |= ( (1<<WGM01) | (1<<WGM00) );  // Set Mode 3 (fast PWM)
	TCCR0A |=   (1<<COM0A1);                // OCOA non inverting PWM output
	TCCR0B |=   (1<<CS00);                  // prescaler=1 / use f_CPU
	OCR0A   = 0;                            // set duty cycle to 0%

	while(1)
	{
		OCR0A =correction(led_value);     // set PWM value
		led_value = led_value+led_increment;
		if (led_value>=99)
			led_increment=-1;
		if (led_value==0)
			led_increment=1;

		_delay_ms(5);
	}
}
```

Würde man bei dem ganzen darauf achten möglichst wenig CPU last zu verbrauchen, würde man einen anderen Ansatz wählen: Die Berechnung der werte zur Programm Laufzeit ist eigentlich wenig sinnvoll, da ständig wieder die gleichen Werte berechnet werden. Sinnvoller wäre unter diesem Gesichtspunkt alle 100 möglichen Werte von der Funktion `correction()` in ein Array zu schreiben und dann nur noch auf diese zu zugreifen.

3.4 Blinken mit compare match

(optional)

Zur Erzeugung von PWM Signalen haben wir OC0A genutzt. Darüber war es uns möglich einen Ausgang direkt mit einem Timer zu verknüpfen. OC0A kann neben der Funktion als PWM Ausgang auch dazu genutzt werden, ähnlich wie beim ersten Timer-Beispiel, einen Ausgang zum Blinken zu bringen. Mit dieser Methode brauchen wir auch kein Interrupt.

Wir müssen einfach in den beiden Konfigurationsregistern des Timers die richtigen Einstellungen machen: Das TCCROB wird erneut nur genutzt um den größten Frequenzteiler auszuwählen. Die wichtigsten Einstellungen finden im Register TCCR0A statt:

TCCR0A – Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Die WGM0x Bits brauchen wir nicht zu verändern, da wir uns wieder im normalen Modus 0 befinden. Wir können allerdings auch in diesem Modus mit den COM0x Bits Arbeiten:

Table 15-2. Compare Output Mode, non-PWM Mode

COM0A1	COM0A0	Description
0	0	Normal port operation, OC0A disconnected.
0	1	Toggle OC0A on Compare Match
1	0	Clear OC0A on Compare Match
1	1	Set OC0A on Compare Match

Wie wir in dieser Tabelle sehen, können wir Einstellungen vornehmen um OC0A bei „compare Match“ zu toggeln. Der „compare Match“ tritt immer dann auf wenn der Zählerwert gleich dem OCR0A ist. Schreiben wir nichts auf OCR0A ist das Register gleich null. Es wird also jedes Mal wenn unser Zählerwert den Wert null hat unser Ausgang getoggelt. Dies erzeugt das gesuchte Blinkmuster.

Wird das beschriebene Verfahren in code umgesetzt erhält man:

```
#include <avr/io.h>

int main(void)
{
	DDRD   |= (1<<DDD6);                    // LED pin output
	TCCR0B |= ((1<<CS02) | (1<<CS00));      // prescaler: 1024
	TCCR0A |= (1<<COM0A0);                  // toggle OC0A on compare match

    while(1);                               // nothing to do in cpu
}
```

Wie man hier sieht verbraucht das Blinken gar keine CPU-Leistung mehr und wird komplett vom Timer erledigt.

4 Quellen

Atmel, *Datasheet ATmega328p* (pdf -> link)

Atmel, *AVR Libc Reference Manual* (online)

Mikrocontroller.net, *LED-Fading* (link)

Wikipedia (diverse Artikel)