# 1 Hello Blinking World

## Student Group

| First Name | Surname | Matrikel Nr. |
|---|---|---|
| | | |
| | | |
| | | |

## Table of Contents

# 1 Hello Blinking World

## 1.1 AVR Programmierung für Dummies

### Ziele

Nach dieser Lektion sollten Sie:

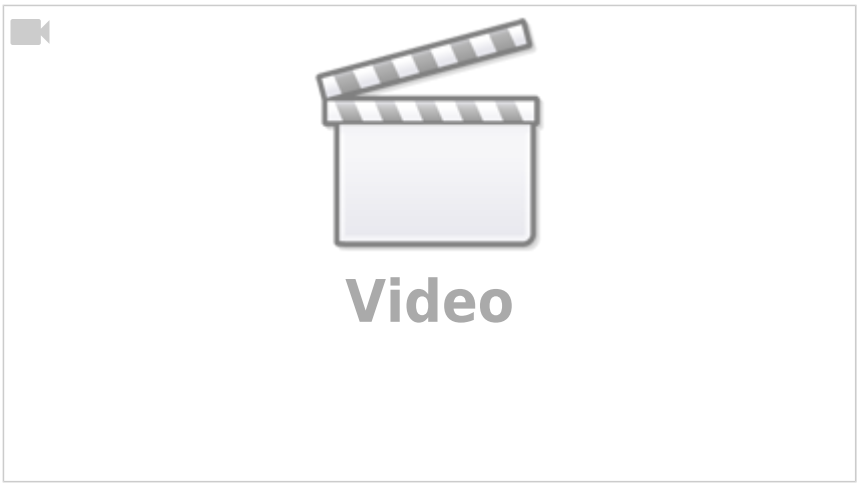
1. die verschiedenen Pins von AVR Chips kennen.
2. die verschiedenen Arten von Speichern in Chips kennen.
3. wissen was Register sind.
4. wissen für was das Spezialregister DDRx (x={A, B, C, D}) nützlich ist.

### Video

Im Video werden im ersten Teil (bis 15min30sec) die Grundkonzepte für die AVR Programmierung erklärt.

#### Achtung!

Ab der 16. Minute geht es in die Assembler Programmierung. Dies ist nicht Teil des Kurses Mikroprozessortechnik.




## 1.2 Es werde Licht! Oder auch nicht...

### Ziele

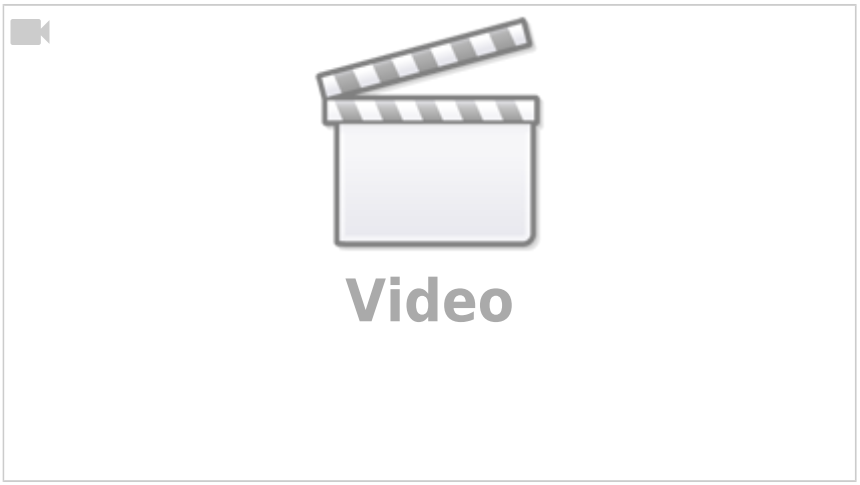
Nach dieser Lektion sollten Sie:

1. wissen, wie man im Microchip Studio ein Projekt anlegt.
2. wissen, wie der Programmierumgebung die Taktfrequenz des Microcontrollers festgelegt wird.
3. die wichtigsten Bitmanipulationen (Bitmaske zum setzten und löschen eines einzelnen Bits, togglen) kennen und anwenden können.

## weiterführende Links

- kurzes Youtube Video: ein einzelnes Bit setzen oder löschen mit bitweisen Operationen

## Video




## Übung

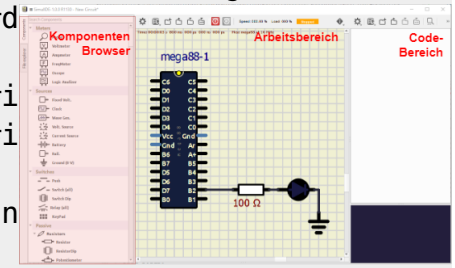
I. Vorarbeiten

1. installieren Sie SimulIDE und Microchip Studio
2. falls es Probleme bei der Programmierung gibt: nutzen Sie die tipps_fuer_die_fehlersuche

II. Eingabe in Microchip Studio

1. öffnen Sie Microchip Studio
2. Anlegen eines neuen Projekts
    1. `File` » New » `Project…`
    2. Wählen Sie “GCC C Executable Project”, da ein ausführbares C Code Beispiel erstellt werden soll
    3. Geben Sie bei Name `Einfuehrung_v01` und drücken Sie auf Ok
    4. Der Cursor sollte nun im Eingabefeld des Suchfenster für die Microcontroller stehen. Geben Sie dort 88 ein
    5. Wählen Sie den mega88 aus den möglichen Chips aus und drücken Sie auf Ok
3. Eingabe und Kompilieren des Code
    1. Ersetzen Sie den vorhandenen Code, durch den rechts stehenden Code
    2. Kompilieren Sie den Code durch `Build` » `Build solution`
    (oder dem Button “Build Solution”

Fig. 1: einfache Diodenschaltung in SimulIDE




```
#d

#i
#i

in
{
    DDRB=0b00000100;
    while (1)
    {
        PORTB |= (1<<2);
        _delay_ms(1000);
        PORTB &= ~(1<<2);
        _delay_ms(1000);
    }
}
```

 oder <F7>)

3. Im unteren Teil des Fensters sollte nun die Ausgabe des Kompilers sichtbar werden. Diese sollte `========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========` lauten

4. Auswählen der hex-Datei
    1. im Microchip Studio finden Sie rechts im Fenster den “Solution Explorer”
    2. gehen Sie dort im Solution Explorer zu `Solution` » `Einfuehrung_v01` » `Output Files`
    3. klicken Sie mit rechter Maustaste auf `Einfuehrung_v01.hex` und wählen Sie Pfad und Name aus

III. Ausführung in Simulide

1. Öffnen Sie SimulIDE (unter …\bin\simulide.exe)
    1. links in SimulIDE sollten Sie den Komponenten Browser finden. Wählen Sie dort `Micro`»AVR»`atmega`»`mega88`
    2. Ziehen Sie den Eintrag `mega88` per Drag and Drop in den Arbeitsbereich (rechter, beiger Teil des Fensters)
    3. Es sollte nun ein Chip names `mega88-1` dargestellt sein
2. Erstellen der Ausgangsschaltung
    1. Im Programm wurde im auf PortB das 2bit angesprochen. Entsprechend soll auch hier am Port B der Ausgang 2 genutzt werden. Am Chip ist dieser mit B2 gekennzeichnet
    2. Fügen Sie eine LED (im Komponenten Browser über Output LED) und ein Massepotential ein (Sources Ground)
    3. Die Komponenten können mit dem Kontextmenu (Rechtsklick) gedreht und gespiegelt werden. Außerdem ist mit der Auswahl von `Properties` im Kontextmenu die Änderung von
    4. Verbinden Sie die LED mit Masse und mit Port B2. Achten Sie auf die richtige Richtung der LED. Die Verbindungen lassen sich dadurch erstellen, dass auf ein Komponenten-Pin geklickt wird und die Linie zu einem nächsten Komponenten-Pin gezogen wird.
3. Flashen der Software
    1. Klicken Sie rechts auf den Microcontroller

und wählen Sie `Load firmware`
2. Fügen Sie hier den Pfad und Name des oben erstellten `Einfuehrung_v01.hex` ein und öffnen Sie dieses

4. Starten der Simulation
    1. klicken Sie im Menu den Power-on Button
    2. Die Simulation startet
5. Bugfixing
    1. vermutlich ist bei Ihnen zu sehen, dass die Diode nicht gleichmäßig an und aus dargestellt wird. Dies ist kein Fehler des Simulationsprogramms. Es wurde noch eine wichtige Komponente vergessen, welche immer bei der Verwendung von diskreten LEDs verwendet werden muss. Fügen Sie diese ein und Testen Sie die Schaltung nochmal

Sie sollten sich nach der Übung die ersten Kenntnisse mit dem Umgang der Umgebung angeeignet haben. Zum Festigen des der Fähigkeiten bieten sich folgende Aufgaben an:

Aufgaben

1. Welche Vorgaben für die Softwareentwicklung wurden verletzt, trotzdem das Programm lauffähig ist? (Interrupts werden erst in späteren Übungen erklärt)
2. Wie könnte ein Ampel-Licht-Abfolge oder Lauflicht aus 4 Dioden erstellt und programmiert werden? Welche Optimierungen könnten im Code vorgenommen werden? Welche Komponente in SimulIDE kann genutzt werden? Wie kann die Farbe der LEDs geändert werden?
3. Lesen Sie auf Mikrocontroller.net im Kapitel Warteschleifen die “erste Seite”, also bis: **Abhängig von der Version der Bibliothek verhalten sich die Bibliotheksfunktionen etwas unterschiedlich.**
4. Registeranalyse
    1. Öffnen Sie in Simulide die RAM Table (Rechtsklick auf Microcontroller `Open Mcu Monitor.` » `RAM Table`)
    2. Analysieren Sie das Verhalten der Register `PORTB` und der Mehrzweckregister `R0…R31` (eintragen von z.B. `PORTB` unter Name in der ersten Zeile und drücken von <Return>, oder anklicken bzw. markieren

des Registers mit dem Cursor und <Return>).
Welches Mehrzweckregister scheint ein Auf- oder Abwärtszähler zu sein? (siehe dazu auch “Was ist eigentlich ein Register?” unter Weiterführende Fragen und Infos)

3. Zählt der Zähler aufwärts oder abwärts? Ändern Sie dazu die Simulationsgeschwindigkeit bei den Einstellungen. Die Einstellungen sind über das Zahnrad in der Menüleiste oben links zu erreichen. Im Reiter `Simulation` kann die Geschwindigkeit geändert werden.

## 1.3 Weiterführende Fragen und Infos

Was hat es mit "Release" und "Debug" in der Standard-Toolbar auf sich?

Mittels dieser Auswahl lassen sich verschiedene Konfigurationen des Compilers setzen. Der Compiler übersetzt den C-Code in maschinenlesbaren Code und wird bei bei Bedarf Hinweise/Warnungen/Fehler ausgeben, diverse Optimierungen vornehmen und die ausführbaren Dateien (“Executables”) ablegen.
Über eine Konfiguration kann damit folgende Dinge geändert werden:

- Kriterien für Hinweise/Warnungen/Fehler
- Optimierungen bei der Kompilierung
- Ablageort der Dateien

Was ist DDRB, PORTB?

Die Anschlüsse (Pins) des Chips sind in 8er Gruppen sortiert, den sogenannten Ports. Für jeden Port sind jeweils drei Register-Bytes vorhanden: DDRx, PORTx und PINx. Diese Speicherstellen ermöglichen die Konfiguration des Ports. Die Bits in den Registern stehen für die einzelnen Anschlüsse: So ist zum Beispiel das 2. Bit in DDRB, PORTB und PINB für den Anschluss B2 zuständig.

Das Bit im **DDRx** (Data Direction Register) wählt die Richtung des Pins aus. Wenn dort logisch Eins geschrieben wird, wird der entsprechende Pin als Ausgangspin konfiguriert. Wenn dort logisch Null geschrieben wird, wird der entsprechende Pin als Eingangspin konfiguriert.

Das Bit im **PORTx** Register hat mehrere Eigenschaften: Wenn das gewünschte Bit in PORTx logisch Eins geschrieben wird und der Pin als Ausgangspin konfiguriert ist, wird der Portpin auf high (eins) gesetzt. Wenn das gewünschte Bit in PORTx logisch Null geschrieben wird und der Pin als Ausgangspin konfiguriert ist, wird der Portpin auf Low (Null) getrieben.

Auch wenn ein Pink als Eingangspin konfiguriert wurde, hat PORTx eine Funktion. Wenn in diesem Fall das gewünschte Bit in PORTx logisch eins geschrieben wird, wird der Pull-up-Widerstand aktiviert. Ein Pull-up-Widerstand ist ein höherohmiger Widerstand (im Bereich $20~\rm k\Omega$ … $100~\rm k\Omega$), der bei nicht weiter verbundenem Pin den

ausgegebenen Wert auf logisch Eins zieht. Um den Pull-up-Widerstand auszuschalten, muss das gewünschte Bit in PORTx logisch Null geschrieben werden oder der Pin muss als Ausgangspin konfiguriert werden.

Das Einlesen der Signale wird in einem späteren Kapitel erklärt.

Was ist eigentlich ein Register?

Vielleicht kennen Sie noch die Begriffe Register oder Schieberegister aus den Grundlagen der Digitaltechnik (siehe Anwendungen mit Flipflops). Register sind im allgemeinen Speicherzellen mit besonderen Funktionen, auf denen der Rechenkern besonders schnell zugreifen kann. Im Folgenden sind eine Auswahl an Registern kurz beschrieben:

- AVR-Microcontroller haben 32 **General Purpose Register (R0…R31)**, also Mehrzweckregister. Diese sind sozusagen der Schreibtisch des Prozessors, d.h. in diesen befinden sich lokale Variablen, Zwischenergebnisse oder Zieladressen zum Ein- und Auslesen. Diese Register werden auch im Ram in den ersten 32 Speicherzellen abgebildet und können in Simulide über die “RAM Table” ausgegeben werden (z.B. Eingabe von 24 für R24).
- Der **Programm Counter (PC)** ist ein internes Register, welches auf den nächsten Befehl für den Rechenkern zeigt. Intern bedeutet, es kann nicht direkt über den Code ausgelesen oder geändert werden. Indirekt geht dies schon, da der PC nach jedem abgearbeiteten Befehl auf den darauffolgenden zeigt. Bei jedem Sprung in eine Funktion oder Wiederholung einer Schleife weist der PC i.d.R. auf eine Stelle, die nicht direkt aufsteigend erreichbar wäre.
- Das **Status Register (SREG)** zeigt verschiedene Stati des letzten ausgeführten Befehls an. War z.B. das Ergebnis der letzten Berechnung Null, so wird das Z-Bit im SREG gesetzt. Die angezeigten Werte im SREG helfen dem Prozessor schnell auf Ergebnisse zu reagieren. Weitere Details zum SREG finden sich im Kapitel 7. Der Kern der AVR-CPU des deutschen AVR Datenblatts.
- Verschiedene **Special Function Register (SFR)**, also Spezialregister, ermöglichen den Zugriff auf Pins, den Analog-Digital-Wandler, oder den Timern. Eine Übersicht der SFR findet sich im Kapitel 36. Übersicht I/O-Register des deutschen AVR Datenblatts.

Wie findet man die Namen der Anschlüsse?

Die Namen sind im Datenblatt des verwendeten Microcontrollers zu finden. Datenblätter lassen sich allgemein mittels auf zwei Wegen finden:

1. Mittels einer Suchmaschine über `[IC Name] “datasheet” site:[Herstellername].com filetype:pdf`, da es sich beim Datasheet um ein PDF handelt.
2. Direkt über die Hersteller-Seite

Leider gibt es gerade bei dem ATMEGA88 auch ein **veraltetes** Datenblatt, welches just das ist, wass sich z.B. über Google leichter finden lässt.
in diesem Fall muss also über die Herstellerseite gesucht werden, bzw. bei der Suchmaschine ATMEGA88 eingeben und anschließend auf die Herstellerseite klicken.

Zum Lesen der Datenblätter empfiehlt sich ein Download und die Betrachtung über einen PDF-Viewer, welcher ein Inhaltsverzeichnis als Seitenleiste ermöglicht (z.B. Acrobat Reader). Ansonsten ist das Inhaltsverzeichnis häufig auch auf den hinteren Seiten des Datenblatts zu

finden.

Die gesuchte Pinbelegung ist für den ATmega88 konkret auf Seite 3 unter “1. Pin Configurations”:

Falls Sie mit dem englischen Datenblatt Probleme haben, kann ich die deutsche Übersetzung des ATmega88 empfehlen, welcher sich im Wesentlichen ähnlich verhält. Beachten Sie aber, dass viele weitere Datenblätter nur in englisch vorhanden sind. Ein Umgang mit der englischen Dokumentation sollte also erlernt werden.

In Simulide fehlen die Anschlüsse für PORTA und weitere PINs. Außerdem kann man GND und VCC nicht verbinden.

Die ATmega Microcontroller basieren alle auf einer ähnlichen Struktur. Bei verschiedenen Varianten (insbesondere bei denen mit geringem “Pin Count”) werden Anschlüsse zusammengefasst oder weggelassen. PORTA ist bei älteren Chips der Anschlussbereich für die Eingänge zum Analog-Digital-Wandler gewesen. Diese sind nun auf PORTC zu finden. Aus Kompatibilitätsgründen ist PORTA weggelassen worden. Ähnlich verhält es sich mit Pin PC7.

Die Anschlüsse GND (Masse) und VCC (Versorgung) sind in der Simulation automatisch verbunden. Der Microcontroller ist also stets betriebsbereit.

Äh.. wir haben 1000ms als Wartezeit eingegeben, aber die LED blinkt doppelt so schnell..

Richtig bemerkt. Das liegt daran, dass der verwendete Chip in Simulide auf 16 MHz läuft. Dies ist in der Simulation oben unter dem Start-Button zu sehen.

In Realität wird die Taktfrequenz durch die Randbedingungen wie Performance, Stromverbrauch, Spannungsversorgung oder Platinengröße vorgegeben. Die reale Hardware des ATmega 88 beinhaltet einen internen Taktgeber mit 8 MHz, welche durch einen Teiler auf 1 MHz reduziert sind. Über einen externen Quarz an den Pins PB6 und PB7 kann bei der realen Hardware (mit weiteren Einstellung der Fuses) ein externen Takt eingespeist werden. Häufig werden hierbei - neben ganzzahligen MHz - auch Vielfache von 256 genutzt, wie z.B. 12,288 MHz. Dies vereinfacht das exakte Abzählen von (Milli)Sekunden, da intern hierzu 8-Bit-Zähler genutzt werden können. Der Takt kann bei der realen Hardware in der Regel nicht zur Laufzeit beliebig geändert werden, sondern liegt fest vor.

Der Controller kenn bei der Ausführung des Codes die Taktfrequenz nicht. Sollen genaue Bruchteile einer Sekunde erzeugt werden, muss dem Compiler die Frequenz über einen define mitgeteilt werden. Im Code geschieht dies über `#define F_CPU 8000000UL`

In der Simulation kann über `rechte Maustaste auf den uC » Eigenschaften » MHz` die Frequenz zwischen 1kHz und 100MHz beliebig geändert werden. Dies ist in Realität nicht möglich. Die höchste Frequenz ist laut Datenblatt 16MHz.

In diesem Fall wäre also die Lösung entweder den Wert von F_CPU oder die Frequenz in der Simulation anzupassen.

Wie ist die Ansteuerung von den Pins nun genau zu verstehen?

Ziel der kleinen Projektes ist es einen Pin wechselnd auf logisch Eins und logisch Null zu ändern. Wichtig ist bei jeder Umsetzung zu prüfen, was **genau** umgesetzt werden soll.

Im Prinzip würde folgender Code das auch ermöglichen:

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    // Die Zahl in folgender Zeile gibt die Bitposition in der nächsten
Zeile an:
    //     76543210
    DDRB=0b00000100;  // B2 soll als Ausgang genutzt werden
    while (1)
    {
        PORTB = 0b00000100; // Das Bit für B2 wird gesetzt; am Ausgang
liegt VCC an
        _delay_ms(1000);
        PORTB = 0b00000000; // Das Bit für B2 wird gelöscht; am Ausgang
liegt 0V an
        _delay_ms(1000);
    }
}
```

Hierbei gibt es aber mehrere Probleme:

1. Bitte geben Sie bei der Kommentierung Ihres Codes keine Trivialitäten an. D.h. alle angegebenen Kommentare sollen so nicht im Code stehen; Sie dienen hier dem Verständnisaufbau
2. Es wird hier nicht nur das gewünschte Bit gesetzt, sondern auch die anderen Bits manipuliert. Das ist in diesem konkreten Fall zunächst kein Problem. ABER: Jede Software sollte so entwickelt werden, dass Sie den höchsten Grad an Wartbarkeit und Erweiterbarkeit aufzeigt. Es ist also wichtig auf eine gewisse Codehygiene (“Clean Code”) zu achten. Es ergeben sich sonst Probleme, wenn eine komplexere Umsetzung nach Jahren weiterentwickelt werden soll.

Achten Sie also darauf, dass die Umsetzung im Code genau das tut was gewünscht ist. In diesem Fall soll nur das Bit B2 manipuliert und alle anderen unverändert belassen werden.

Hiermit ergibt sich die Frage, wie genau nur ein Bit in einem Register geändert werden kann. Dies ist über die Funktionen der logischen Bitmanipulation möglich (UND, ODER, etc.). Hierzu wird das gewünschte Register mit einer Maskierung verknüpft.

Über die Disjunktion (ODER bzw. in C über `|`) mit logisch Eins können nur bestimmte Bitpositionen auf logisch Eins gesetzt werden:

```
Register-Byte .... R01 = 0b01100101 (Beispielwert)
```

```
Maske ............ MSK = 0b0000**11**00
Disjunktion .. R01|MSK = 0b0110**11**01
```

Die fett markierten Bits mit logisch Eins in der Maske MSK gewährleisten also, dass die Bits in der Disjunktion R01|MSK gesetzt sind. In C wäre hierfür `R01 = R01|MSK;` zu schreiben - R01 ist im LED-Code PORTB; in MSK sollte nur das Bit 2 gesetzt sein. Die Programmiersprache C bietet die Möglichkeit mathematische Operatoren auch noch etwas umzuschreiben: `R01 |= MSK;` Auch für das Setzen eines einzelnen Bits in MSK kann eine logische Bitmanipulation genutzt werden: das Verschieben aller Bits nach links. Der Code 1«2 erzeugt ein Byte mit dem Inhalt `0b00000100`. Für den Neuling mag 1«2 etwas schwerer zu lesen sein - für den fortgeschrittenen Entwickler ist dies leichter zu lesen. Daneben bietet die inkludierte Bibliothek `<avr/io.h>` die Möglichkeit auf weitere defines zuzugreifen, z.B. PB2 welches durch 2 ersetzt wird. Damit ließe sich der Code für des Setzen eines Bits schreiben zu:

```
        PORTB |= 1<<PB2; // Das Bit für B2 wird gesetzt; am Ausgang liegt
VCC an
```

Für das Löschen kann ein ähnliches Konzept genutzt werden. Statt eine “VerODERung mit 1” (im Code `R01|0b01000000`) zum Setzen, muss hier eine “VerUNDung mit 0” (im Code `R01|0b10111111`) genutzt werden:

```
Register-Byte .... R01 = 0b01100101 (Beispielwert)
Maske ............ MSK = 0b1111**00**11
Konjunktion .. R01&MSK = 0b0110**00**01
```

Die Maske MSK kann hier durch die Negation des gesetzten Bits erfolgen:

```
Orginalmaske ............... ORG_MSK = 0b0000**11**00
negierte Maske ..... MSK = ~ORG_MSK = 0b1111**00**11
```

Der Code ergibt sich dann zu:

```
        PORTB &= ~(1<<PB2); // Das Bit für B2 wird gelöscht; am Ausgang
liegt 0V an
```

Was macht ein define? Warum wird manchmal ein 'static const' und manchmal ein 'define' genutzt?

> Alle Befehle mit # sind nur Compileranweisungen. Der Compiler setzt den C-Code in maschinenlesbaren Code um. Die Compileranweisungen werden aber nicht in maschinenlesbaren Code umgesetzt, sondern weisen den Compiler an verschiedene Dinge zu tun. `#define` speziell weist den Compiler an eine dargestellte Zeichenfolge durch eine andere zu ersetzen. Beispiel:
>
> ```
> #define DUMMY 5
> ```

```
...

int main(void)
{
    ...
    A = DUMMY + 2;
    ...
}
```

In diesem Beispiel wird vor der Übersetzung des Codes die Zeichenfolge DUMMY durch 5 ersetzt und dann erst kompiliert. Beim Ersetzen wird keine Typisierung (Datentypen) beachtet. Weiterhin wird der definierte Code nicht vorher berechnet, was zu Problemen bei der Priorisierung führen kann:

```
#define DUMMY1 500
#define DUMMY2 5 + 3

...

int main(void)
{
    ...
    uint8_t a = DUMMY1; // DUMMY1 ist größer als 500
    uint8_t b = DUMMY2 * 2; // Es wird 5 + 3 *2 = 11 ausgegeben, und nicht
(5 + 3)*2 = 16
    ...
}
```

Prinzipiell kann also #define für Zahlenwerte genutzt werden, sofern die obengenannten Grenzen beachtet werden. Diese Probleme lassen sich für Zahlenwerte über die Verwendung von static const umgehen (z.B. `static const int var = 5;`). In Microcontrollern werden von der Programmierumgebung häufig auch defines vorgegeben (z.B. PORTB, PB2). Der große Vorteil von Compiler-Definitionen besteht darin, dass so die verschiedene eher schwer lesbare, aber häufig verwendete Codeschnipsel anschaulicher umschrieben werden können. Folgende defines werden beispielsweise häufig genutzt:

```
#define SET_BIT(BYTE, BIT)    ((BYTE) |=  (1 << (BIT))) // Bit Zustand in
Byte setzen
#define CLR_BIT(BYTE, BIT)  ((BYTE) &= ~(1 << (BIT))) // Bit Zustand in Byte
loeschen
#define TGL_BIT(BYTE, BIT)  ((BYTE) ^=  (1 << (BIT))) // Bit Zustand in Byte
wechseln (toggle)
```

Wie sähe der Code aus, wenn man sich an die Vorgaben für sauberen Code hielte?

Hierbei ist zu beachten, dass auch die Verwendung von delays vermieden werden soll. Diese sind ein “aktives Nichtstun” des Prozessors. Das heißt er kann in der Zeit nicht andere Aufgaben erledigen. Ein Multitasking ist damit nicht möglich. Da dies einer nachträglichen Erweiterung des Codes im Weg steht, sollten **generell keine delays** verwendet werden. Eine Ausnahme davon bildet Treiber-Code in welchem Pegeländerungen an einem Pin Microsekunden-genau umgesetzt werden müssen.

```
#define F_CPU               8000000UL
#define SET_BIT(BYTE, BIT)    ((BYTE) |=  (1 << (BIT))) // Bit Zustand
in Byte setzen
#define TGL_BIT(BYTE, BIT)  ((BYTE) ^=  (1 << (BIT))) // Bit Zustand in
Byte wechseln (toggle)

#define LED_WAIT_TIME       1000 // Dauer bis zum Taktwechsel am Pin
#define LED_PIN             PB2 // Pin an dem die LED anschlossen ist

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    SET_BIT(DDRB, LED_PIN);
    while (1)
    {
        TGL_BIT(PORTB, LED_PIN);
        _delay_ms(LED_WAIT_TIME);
    }
}
```

Beim "sauberen Code" wird PB2 genutzt - wo kommt das her? Wo liegt DDRB eigentlich ab?

Klicken Sie auf den fraglichen Code-Snipsel (z.B. Variablen- oder Funktionsname) mit einem Rechts-Klick und wählen Sie `Goto Implementation` (Alternativ <Alt>+<G>) . Wird beispielsweise PB2 eingegeben und die Implementierung angezeigt, ist zu sehen, dass PB2 auch nur ein define ist: `#define PB2 2`. Selbiges ist bei DDRD zu finden; dort wird aber über den define der Text durch `_SFR_IO8(0x05)` ersetzt. `_SFR_IO8(0x05)` ist ein Befehl für den Microcontroller, um auf ein Special Function Register des Input-Output-Registerbereichs zugreifen, welches im Speicher unter `0x05` abliegt.

Wie finde ich heraus, was wo definiert wurde?

siehe vorheriger Punkt