

Was ist ein Microcontroller?

Wednesday, 7 September 2022 09:09

Zitat Wikipedia:

Als **Mikrocontroller** (auch *μController*, *μC*, *MCU* oder *Einchipmikrorechner*) werden **Halbleiterchips** bezeichnet, die einen **Prozessor** (oder mehrere Prozessoren) und zugleich auch **Peripheriefunktionen** enthalten. In vielen Fällen befindet sich auch der **Arbeits- und Programmspeicher** teilweise oder komplett auf demselben Chip. Ein Mikrocontroller ist ein **Ein-Chip-Computersystem**. Für manche Mikrocontroller wird auch der Begriff **System-on-a-Chip** oder **SoC** verwendet.

Auf modernen Mikrocontrollern finden sich häufig auch **komplexe Peripheriefunktionen** wie z. B. **CAN**- (Controller Area Network), **LIN**- (Local Interconnect Network), **USB**- (Universal Serial Bus), **I²C**- (Inter-Integrated Circuit), **SPI**- (Serial Peripheral Interface), **serielle** oder **Ethernet**-Schnittstellen, **PWM**-Ausgänge, **LCD**-Controller und -Treiber sowie **Analog-Digital-Umsetzer**. Einige Mikrocontroller verfügen auch über programmierbare digitale und/oder **analoge** bzw. hybride Funktionsblöcke.



Einsatzbereiche

Der Mikrocontroller tritt in Gestalt von **eingebetteten Systemen** im Alltag oft unbemerkt in technischen Gebrauchsartikeln auf, zum Beispiel in Waschmaschinen, **Chipkarten** (Geld-, Telefonkarten), Unterhaltungselektronik (**Videorecordern**, **CD-/DVD-Spieler**, **Radios**, **Fernsehgeräten**, **Fernbedienungen**), Büroelektronik, Segways, Kraftfahrzeugen (**Steuergeräte** für z. B. **ABS**, **Airbag**, **Motor**, **Kombiinstrument**, **ESP** usw.), **Mobiltelefonen** und sogar in **Uhren** und **Armbanduhren**. Darüber hinaus sind sie in praktisch allen Computer-**Peripheriegeräten** enthalten (**Tastatur**, **Maus**, **Drucker**, **Monitor**, **Scanner** usw.).

Mikrocontroller sind in Leistung und Ausstattung auf die jeweilige Anwendung angepasst. Daher haben sie gegenüber „normalen“ Computern Vorteile bei den Kosten und der Leistungsaufnahme. Kleine Mikrocontroller sind in höheren Stückzahlen für wenige Cent verfügbar.

Abgrenzung zu Mikroprozessoren

Die Grenze zwischen Mikrocontrollern und **Mikroprozessoren** ist fließend...

Der Arduino UNO als Beispiel ...

Korrektur: Der ATmega328p als Beispiel

Wednesday, 7 September 2022 09:09

Arduino

Arduino Uno ist ein ATmega328p

Mikrocontroller mit Board,

Spannungsversorgung, USB-Anschluss, ...

Der ATmega328p des Arduino UNO

hat einen besonderen Bootloader,

um die einfache Neuprogrammierung durch die

Arduino Programmierungsgebung zu

ermöglichen
... was vom Betriebssystem übrig bleibt
UEFI, BIOS trifft es noch besser

vs Atmel

war ein Hersteller von Mikrocontrollern.

Unter anderen baute Atmel Mikrocontroller mit

AVR-Architektur

2016 wurde Atmel von Microchip Technology gekauft

vs AVR

ist ein Prozessorarchitektur (vgl. x86, ARM, ...)

Diese Prozessorarchitektur gibt es in 8, 16, 32-Bit Ausführung

Der ATmega328p gehört zur 8-Bit AVR Familie

Voraussetzung:
Der Prozessor arbeitet vorrangig mit 8, 16, 32-Bit Zahlen

Achtung ☺☺

Der Begriff Arduino - bestehend aus einem Hardwaresteil und einem Softwaresteil - wird von vielen µC-Entwicklern belächelt.

In MCT werden wir AVR µC kennenlernen, studieren, programmieren

Programmiersprachen auf Mikrocontrollern: meistens Assembler, C

selten C++
Arduinos Programmierungsgebung

▷ für Programmierung wichtig

Daten des ATmega328p:

entspricht am PC:

32 kB Flash-Speicher 0,5 kB Bootloader → Festplatte

2048 kB SRAM → RAM

1024 kB EEPROM

16 MHz Taktfrequenz

10 bit Auflösung ADC

Little endian

16bit Befehlsatz

RISC Architektur

ATmega328p Datenblatt

ATmega328p Befehlsatz

Dezimal, Binär- und Hexadezimalsystem

Wednesday, 7 September 2022 09:10

Dezimalzahl: 1271

dezi heißt 10
Dezimale: 10  Rekt.

Dezimalzahlen haben 10 Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binärzahlen haben 2 Ziffern: 0, 1 oder auch low, high oder ,  ...

Hexadezimalzahlen haben 16 Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

Fernglas auf Englisch:
binocular
bicycle

$$\begin{array}{cccc} 1 & 2 & 7 & 1 \\ \uparrow & \uparrow & \uparrow & \uparrow \\ \text{Einer} & 1 & - 10^0 \\ \text{Zehner} & 10 & - 10^1 \\ \text{Hunderter} & 100 & - 10^2 \\ \text{Tausender} & 1000 & - 10^3 \\ \vdots & & & \end{array} \quad 1 \ 2 \ 7 \ 1 = 1 \cdot 1 + 7 \cdot 10 + 2 \cdot 100 + 1 \cdot 1000$$

Dies ist entsprechend in jedem Zahlensystem gültig!

$$\begin{array}{cccc} 1 & 0 & 1 & 1_b \\ \uparrow & \uparrow & \uparrow & \uparrow \\ \text{Achter} & \text{Vierer} & \text{Zweier} & \text{Einer} \\ & 1 & - 2^0 \\ & 2 & - 2^1 \\ & 4 & - 2^2 \\ & 8 & - 2^3 \\ \vdots & & & \end{array} = \left\{ \begin{array}{l} 1 \cdot 1_d + 1 \cdot 2_d + 0 \cdot 4_d + 1 \cdot 8_d = 11_d \\ \text{Achtung Dezimalzahlen!} \\ 1 \cdot 1_b + 1 \cdot 10_b + 0 \cdot 100_b + 1 \cdot 1000_b = 1011_b \end{array} \right. \quad \text{Binär} \rightarrow \text{Dezimal}$$

Binärzahlen

$$\begin{array}{cccc} 1 & 2 & 7 & 1 \\ \uparrow & \uparrow & \uparrow & \uparrow \\ \text{1000er} & \text{100er} & \text{10er} & \text{1er} \\ & & 10 & 1 \\ & & 100 & 100 \\ & & 1000 & 1000 \end{array} \quad 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \quad \begin{array}{l} \text{machbar aber} \\ \text{nicht praktikabel!} \end{array}$$

Viel besser:

2er Potenzen:

$$\begin{aligned} 2^0 &= 1 \\ 2^1 &= 2 \\ 2^2 &= 4 \\ 2^3 &= 8 \\ 2^4 &= 16 \\ 2^5 &= 32 \\ 2^6 &= 64 \\ 2^7 &= 128 \\ 2^8 &= 256 \\ 2^9 &= 512 \\ 2^{10} &= 1024 \\ 2^{11} &= 2048 \\ \vdots & \end{aligned}$$

$$\begin{array}{l} \text{In } 1271 \text{ geht einmal } 2^{10} \\ 1271 \\ - 1024 \\ \hline 247 \\ \text{In } 247 \text{ geht einmal } 2^7 \\ 247 \\ - 128 \\ \hline 119 \\ \text{In } 119 \text{ geht einmal } 2^6 \\ 119 \\ - 64 \\ \hline 55 \\ \text{In } 55 \text{ geht einmal } 2^5 \end{array}$$

In 55 geht einmal 2^5

$$\begin{array}{r} 55 \\ -32 \\ \hline 23 \end{array}$$

In 23 geht einmal 2^4

$$\begin{array}{r} 23 \\ -16 \\ \hline 7 \end{array}$$

In 7 geht einmal 2^2

$$\begin{array}{r} 7 \\ -4 \\ \hline 3 \end{array}$$

In 3 geht einmal 2^1

$$\begin{array}{r} 3 \\ -2 \\ \hline 1 \end{array}$$

In 1 geht einmal 2^0

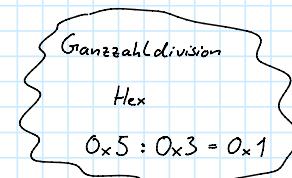
$$\begin{array}{r} 1 \\ -1 \\ \hline 0 \end{array}$$

$$\rightarrow 10011110111$$

$2^{10} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$

TR: Neu \rightarrow 3

"blaue Tasten" \rightarrow Ganzzahldivision
 $+,-,\cdot,: \quad \text{Ganzzahldivision}$



Im Hexadezimalsystem werden jeweils 4 bit zu einer Hexziffer zusammengefasst!

Übungen:

Eingabe von Zahlen im Dezimal, Binär, Oktal und Hexadezimalsystem

Ausgabe von Zahlen in anderen Zahlsystemen

Hex 7A4C in Dec, handisch, TR, Programm

Dec 14593 in Hex, handisch, TR, Programm

Übersetzertabelle

Binär	Hexadezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	a
1011	b
1100	c
1101	d
1110	e
1111	f

11100	c
1101	d
1110	e
1111	f

Übungen

Donnerstag, 29. September 2022 10:36

```
void setup() {  
    // put your setup code here, to run once:  
    word sepp = 0xffff;  
    int hias = 0b10011110111;  
  
    Serial.begin(9600);  
    Serial.print("sepp dec: ");  
    Serial.println(sepp);  
    Serial.print("sepp hex: ");  
    Serial.println(sepp, HEX);  
    Serial.print("hias dec: ");  
    Serial.println(hias);  
    Serial.print("hias hex: ");  
    Serial.println(hias, HEX);  
  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
    double sepp = 3.0;  
}
```



sketch_za...

Ganzzahl Darstellung

Wednesday, 7 September 2022 09:11

[char - Arduino Reference](#)
[short - Arduino Reference](#)
[int - Arduino Reference](#)
[long - Arduino Reference](#)

→ Arduino Reference → Data Types

Wir widmen uns den Ganzzahl Datentypen aus dieser Auflistung

und versuchen Strukturen in das Chaos zu bringen.

1 Byte ist aus 8 Bit aufgebaut:
Byte in Hex
 $4D = 01001101$
most significant bit
1 MSB
least significant bit
LSB
Bit: 0 oder 1

Bytes	Windows 64	Arduino Uno	explizite Datentypen	unsigned Datentypen
1	char	char	int8_t	byte
2	short	short, int	int16_t	word
4	long, int	Long	int32_t	
8	long long	Long long	int64_t	
				uint8_t
				uint16_t
				uint32_t
				uint64_t
				:

gibt es auch mit Vorsilbe
unsigned ? z.B.:
unsigned long long x;

→ Übung: anwenden am Arduino

z.B. $\text{int } x = 5;$
 $\text{long } y = 4;$ Siehe Referenz sizeof
 $\text{Serial.println(sizeof(x));}$
 $\text{Serial.println(sizeof(y));}$
 $\text{Serial.println(x+y);}$

Das Zweierkomplement

Das Zweierkomplement ist ein genialer Trick um negative ganze Zahlen intern im Computer darzustellen (zu speichern).

$$6_{\text{dec}} = 00000110_b$$

$$-6_{\text{dec}} = ?$$

Die bitweise Negation / Einerkomplement

$$\begin{array}{c} \text{Eingabe} \quad 0 \mid 1 \\ \hline \text{Ausgabe} \quad 1 \mid 0 \end{array}$$

Bitweise Negation bei der Programmierung

Übung: gebe die bitweise Negation der Zahl 6 aus.

$$\begin{array}{l} b = 1011 \\ \sim b = 0100 + 1 \\ \hline \dots \end{array}$$

BITWEISE NEGATION BEI DER PROGRAMMIERUNG

Übung: gebe die bitweise Negation der Zahl 6 aus.

Übung: Was ist Zahl + (bitweise Negation derselben Zahl) ?

Übung: Was ist Zahl + (bitweise Negation derselben Zahl) + 1 ?

$$6 + (-6) =$$

$$4 + (-4) =$$

$$\left. \begin{array}{l} b = 1011 \\ \sim b = 0100+1 \\ \sim b+1 = 0101 \\ b = 1011 \\ + (\sim b+1) = 0101 \\ \hline 0000 \end{array} \right\}$$

Das Zweierkomplement von a ist die bitweise Negation von a - anschliessend um 1 erhöht

-6 wird im ATmega328p als Zweierkomplement von 6 dargestellt.

$$\begin{array}{rcl} 6_{dec} & = & 00000110_b \\ \text{Vorzeichen} & \swarrow & 11111001_b \quad \text{bitweise Negation / Einerkomplement} \\ & + & 00000001_b \\ -6_{dec} & = & 11111010_b \quad \text{Zweierkomplement von 6} \end{array}$$

[Video zum Zweierkomplement](#)

Binärzahl x	Beträge von x - $\sim(x-1)$ wenn "x negativ"	x in Hex	Dezimal als int	Dezimal als unsigned int
$\left\{ \begin{array}{l} 0111111111111111 \\ 1000000000000000 \\ 1000000000000001 \end{array} \right.$	0111111111111111 1000000000000000 0111111111111111	ffff 8000 8001	32767 -32768 -32767	32767 32768 32769
$\left. \begin{array}{l} \\ \\ \\ \text{diese Zahlen folgen} \\ \text{direkt aufeinander} \\ \text{V2 bei int} \end{array} \right\}$				

→ Übung: Wie verhält sich folgendes Programm? Wie könnten wir es korrigieren?

```
int a = 0;
do {
    a++;
} while (a <= 1000000)
```

Übungen

Donnerstag, 6. Oktober 2022 07:45

```
void setup() {  
    // put your setup code here, to run once:  
    unsigned int sepp = 0xffff;  
    int hias = 0b10011110111;  
    int a = 0x7fff;  
    int b = 0x0001;  
    int c = a+b;  
    char sechs = 0xfa;  
  
    Serial.begin(9600);  
    Serial.print("sepp dec: ");  
    Serial.println(sepp, DEC);  
    Serial.print("sepp hex: ");  
    Serial.println(sepp, HEX);  
  
    Serial.print("Einserkomplement von sepp hex: ");  
    Serial.println(~sepp, HEX);  
  
    Serial.print("hias dec: ");  
    Serial.println(hias);  
    Serial.print("hias hex: ");  
    Serial.println(hias, HEX);  
  
    Serial.print("sechs dec: ");  
    Serial.println(sechs, DEC);  
  
    Serial.print("Größe in Bytes von sepp: ");  
    Serial.println(sizeof(sepp));  
  
    Serial.print("a: ");  
    Serial.println(a);  
    Serial.print("b: ");  
    Serial.println(b);  
    Serial.print("a+b: ");  
    Serial.println(c);  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```



sketch_zw...

Exkurs: Messung der Laufzeit am Arduino

Dienstag, 11. Oktober 2022 10:39

Wie lange dauert es einen char, int, long, long long in der AVR
Architektur zu addieren?

Exkurs: Zeitmessung

Themen: ungewollte Compiler Optimierungen
Zeitmessung mit AVR

[millis - Arduino Reference](#)

[micros - Arduino Reference](#)

Achtung! Die halbwegs aussagekräftige Zeitmessung auf einem Computer (egal ob PC oder Mikrocontroller) ist eine Wissenschaft für sich. Ergebnisse unbedingt hinterfragen!

Der ATmega328p hat eine Taktfrequenz von 16MHz

Die aller einfachsten Rechenoperationen auf einer CPU benötigen genau einen Arbeitsschritt.

→ Übung: sucht aus dem [ATmega328p Befehlssatz](#) einen Befehl, der nur einen Takt benötigt (Takt engl. cycles)

→ Wie oft kann dieser Befehl auf dem ATmega328p in einer Sekunde ausgeführt werden?

→ Wieviele Arbeitsschritte sind vorgenommen, wenn sich `micros()` um 1 erhöht hat?

→ Versuchen Sie mithilfe von `micros()` festzustellen, wie lange auf dem ATmega328p `unsigned char + unsigned char` dauert.

z.B. ADC = Add with carry

ein ADC braucht $T = \frac{1}{f}$ d.h. $6,25 \cdot 10^{-9}$. Also können 16 Millionen ADC in einer Sekunde ausgeführt werden

16 Takte in 1μs

```
void setup()
{
    Serial.begin(9600);
}

void loop()
{
    unsigned char a = 0x7;
    unsigned char b = 0xff;
    long time_start = micros();
    b = b + a; // b = 0x106 262 Dezimal
    long time_stop = micros();
    delay(10);
    Serial.print("Benötigte Zeit: ");
    Serial.println(time_stop - time_start);
}
```

Achtung:

time_start kann 0xffff sein und time_stop 0x02

Dies ist aber kein Problem, siehe [Arithmetik mit Ganzzahlen \(Subtraktion\)](#)

→ Warum wird oft 0x0 angezeigt? Löschen Sie die Rechenoperation und vergleichen Sie die Ergebnisse? Was ist da los?

→ Ändert sich etwas, wenn wir b wirklich benutzen? z.B. `Serial.println(b);`

Ergebnis:

Der Computer ist ein verdammt komplexes Programm, dass sich extrem bemüht unseren geschriebenen Code zu optimieren. Bei der Zeitmessung und allgemein beim Timing beraten uns aber die Optimierungen des Compilers riesige Kopfschmerzen!

Der C-Compiler merkt, dass wir zwar $b+a$ rechnen, dass Ergebnis aber nicht verwendet wird. Deshalb löscht der Compiler die Zeile `b = b + a;`, weil sie nichts bewirkt.

Vorsicht: Man könnte machen durch ein `Serial.println(b);` muss der Compiler $b = b + a;$ ausführen. Nein, das wird er nicht tun, sondern während der `Compilat` ausrechnen, was $a+b$ ist, da der Compiler versucht, dass a und b fiktive Zahlen sind. Er wird letztlich so tun, als hätten wir geschrieben `Serial.println("262")`

Lösung für die Probleme:

① Die gemessenen Zeiten sind grundsätzlich sehr hoch.

Viel effizienter addieren, z.B. 1000 Mal

② Der Compiler optimiert die zu messende Addition weg.

Die Variablen a und b als `volatile` definieren. [volatile - Arduino Reference](#)

/** Zeitmessung zweiter Anlauf */

void setup()
{
 Serial.begin(9600);
}

```

        }
        void loop()
        {
            volatile unsigned char a = 0x7;
            volatile unsigned char b = 0xff;
            long time_start = micros();
            for (int i = 0; i < 1000; i++)
            {
                b = b + a;
            }
            long time_stop = micros();
            delay(10);
            Serial.print("Benoetigte Zeit: ");
            Serial.println(time_stop - time_start);
        }
    }
}

```

Das ist schon nicht schlecht. Wir müssen natürlich bedenken, dass nur die $\frac{7000}{700}$ $\frac{7000}{700}$ Gesamtzeit: Schleife, Variable SRAIT \rightarrow CPU Register, Addition, Variable CPU Register \rightarrow SRAIT gemessen wird.

Aber: Warum schwanken die Zeiten überhaupt noch?

Weil **Interrupts**! Interrupts sind Routinen (C-Funktionen), die der Microcontroller immer wieder **auf unplannmäßig** abarbeitet. Dazu **unterbricht** er unseren Code, führt dann die Interrupt-Routine aus und führt anschließend wieder unser Programm aus. Übrigens werden wir in TCT eigene Interrupt-Routinen schreiben!

Lösung: mit den folgenden zwei Funktionen können **Interrupts** verboten/erlaubt werden.

Dies sollte nur **kurzzeitig** geschehen, da Interrupt-Routinen **wichtige Aufgaben** erledigen.

noInterrupts - Arduino Referenz

interrupts - Arduino Referenz



Timing with
Macros

```

void setup()
{
    Serial.begin(9600);
    noInterrupts();
}
void loop()
{
    volatile unsigned char a = 0x7;
    volatile unsigned char b = 0xff;
    noInterrupts();
    long time_start = micros();
    for (int i = 0; i < 1000; i++)
    {
        b = b + a;
    }
    long time_stop = micros();
    interrupts();
    delay(10);
    Serial.print("Benoetigte Zeit: ");
    Serial.println(time_stop - time_start);
}

```

```

/** Zeitmessung: jetzt fangen wir an zu spinnen ... */
/* Durch timer2 gibt es einen overhead, der natürlich selber clockcycles frisst. Evtl könnte man versuchen diesen overhead durch inline von timer2.reset() und timer2.get_count() noch weiter zu reduzieren.
 * Aber diesen overhead kann man sowieso herausrechnen indem man die Zeit 8 mal b += a mit der Zeit 16 mal b += a vergleicht!
 */
/* Ergebnisse:
 * Ein b += a braucht:
 * unsigned char: 7/8 * 0.5us -> 7 clockcycle
 * unsigned int: 7/4 * 0.5us -> 14 clockcycle
 * unsigned long: 7/2 * 0.5us -> 28 clockcycle
 * unsigned long long: 9 * 0.5us -> 72 clockcycle
 *
 * 1 clockcycle entspricht 0.0625us
 */
#define MEASURE_START    noInterrupts(); \
                      TCCR1A = 0; \
                      TCCR1B = 1 << CS10; \
                      TCNT1 = 0;

/** offset empty loop is 4 cycles with optimization off */
/** offset empty loop is 1 cycles with standard optimization */
#define MEASURE_STOP(c, offset) c = TCNT1; \
                           c = c - offset; \
                           interrupts();

```

```

typedef unsigned char integer_to_test;
void setup()
{
    Serial.begin(9600);
}

//#pragma GCC push_options
//#pragma GCC optimize ("O0")
void loop()
{
    volatile integer_to_test a = 0x7;
    volatile integer_to_test b = 0xff;
    unsigned int cycles;
    MEASURE_START

    b = b + a + a;

    MEASURE_STOP(cycles, 1)
    delay(10);
    Serial.print("Benoetigte Zeit: ");
    Serial.println(cycles);
}
//#pragma GCC pop_options

```

Auflösung 4μs \rightarrow 64 Takte
1 Arbeitsschritt: 16MHz 0,0625μs

$0 \times 0002 \ll 0 \times 0001$ 10 Takte
immidiat

4.) Division mit einer 2er-Potenz:

analog: Siehe Bitshift nach rechts

Achtung: Hierbei entsteht niemals eine Gleitkommazahl (float, double)

Übung: → Was ist $5 \div 2$ auf dem ATmega328p?

→ Wieviele Taktzyklen benötigt

$0 \times ffff / 0 \times 0002$

→ Wieviele Taktzyklen benötigt

$0 \times ffff \% 0 \times 0002$

→ Wie kann ich diese Rechnungen ablaufen lassen?

$\times / 2$ bzw. $\times \% 2$

Noch ausführen? Wieviele Takte brauchen diese Instruktionen?

2

224 444

224 444

Das ist katastrophal langsam!!!

$\times \gg 0 \times 0001$
immidiat

$\times \& 0 \times 0001$

10

10

Zusammenfassung:

$\times \cdot 1$	unmöglich	$\times / 1$	unmöglich	$\times \% 1$	unmöglich, immer 0
$\times \cdot 2$	$\ll 0 \times 1$	$\times / 2$	$\gg 0 \times 1$	$\times \% 2$	$\& 0 \times 1$
$\times \cdot 4$	$\ll 0 \times 2$	$\times / 4$	$\gg 0 \times 2$	$\times \% 4$	$\& 0 \times 3$
$\times \cdot 8$	$\ll 0 \times 3$	$\times / 8$	$\gg 0 \times 3$	$\times \% 8$	$\& 0 \times 7$
$\times \cdot 16$	$\ll 0 \times 4$	$\times / 16$	$\gg 0 \times 4$	$\times \% 16$	$\& 0 \times f$
$\times \cdot 32$	$\ll 0 \times 5$	$\times / 32$	$\gg 0 \times 5$	$\times \% 32$	$\& 0 \times 1f$
$\times \cdot 64$	$\ll 0 \times 6$	$\times / 64$	$\gg 0 \times 6$	$\times \% 64$	$\& 0 \times 3f$
$\times \cdot 128$	$\ll 0 \times 7$	$\times / 128$	$\gg 0 \times 7$	$\times \% 128$	$\& 0 \times 7f$
$\times \cdot 256$	$\ll 0 \times 8$	$\times / 256$	$\gg 0 \times 8$	$\times \% 256$	$\& 0 \times ff$
$\times \cdot 512$	$\ll 0 \times 9$	$\times / 512$	$\gg 0 \times 9$	$\times \% 512$	$\& 0 \times 1ff$
$\times \cdot 1024$	$\ll 0 \times a$	$\times / 1024$	$\gg 0 \times a$	$\times \% 1024$	$\& 0 \times 3ff$
⋮	⋮	⋮	⋮	⋮	⋮

$\times / 2^{\text{er-Potenz immidiat}}$ und $\times \% 2^{\text{er-Potenz immidiat}}$ wird vom Computer optimiert

Bit-Plachen

Übung: → vgl. $\times \% 0 \times 0100$ mit $\times \% 0ca3$

9 Takte vs 208 Takte Achtung

$0 \times 01a5 \% 0 \times 0008$

$0000000011010101 \ll 0000000000000000 = 0 \dots + 011$

Dieselbe Wirkung kann durch folgende Operation erzielt werden

0000000011010101 Bitmaske
 $\& 0 \dots - 0111$
 $\dots \dots 0000011$

Wiederholung: Bitoperationen in C

① Not	② And	③ Or	④ Xor
$A \mid \sim A$	$A \mid B$	$A \& B$	$A \mid B$
0 1	0 0	0 0	0 0
1 0	0 1	0 1	0 1
1 0	0 0	1 0	1 1
1 1	1 1	1 1	1 0

Achtung: Verwechseln Sie die Bitoperationen ($\sim, \&, \mid, \wedge$) nicht mit den Logischen Verknüpfungen ($!, \&&, \mid\mid$)

In C ist jeder Ausdruck, der nicht Null ist true
Ausdrücke die Null sind false

$7 \mid\mid 3 \rightarrow \text{true} \mid\mid \text{true} \rightarrow \text{true} \rightarrow 0 \times 1$
 $7 \mid 3 \rightarrow 0111 \mid 0011 \rightarrow 0111$

0111
 0011
 0111

und $8-6 = a = 7$,

if ($a == 0$) {
 false
}

false ist 0×00
true sonst

und $8-6 = a = 7$

if ($a == 0$) || ($a == 7$)
 false
 true

true
if ($a == 0$) || ($a == 7$)
 false
 true
 0x00
 0x12

A

B,

A

if ($a == 0$) | ($a == 7$)

false true

0x00 0x12

0x12 ≡ true

0x00 00000000

00010010

00010010

Howto: C-Programmierung

Thursday, 13 October 2022 11:40

(1) Scope / Sichtbarkeitsbereich

Ein Bereich "zwischen" [...] wird **scope** (Sichtbarkeitsbereich) genannt.
Variablen, die in einer scope **deklariert** wurden, sind außerhalb der scope **unsichtbar**!

```
void f()
{
    int a = 0;
    {
        a++;
    }
    a++; // Fehler: Compilation error: 'a' was not declared in this scope
}
int b = a;
```

Scope ist sehr wichtig, damit es bei größeren Programmen nicht ständig zu Kollisionen mit **zufällig** gleichen Variablennamen kommt.

→ Fehlern, deklarieren eine Variable in der kleinstein (minimale) möglichen scope, um nicht zufällig andere Programmabschnitte mit Variablennamen zu verdecken.

Variablen, die überhaupt nicht von [...] eingeschlossen sind heißen **globale** Variablen. Alle anderen Variablen heißen **lokale** Variablen.

```
int a = 0; // a ist eine globale Variable
void f()
{
    long b = 11; // b ist eine lokale Variable
    a++;
    b++;
}
```

Werden Variablen mit **gleichen** Namen in verschiedenen scopes deklariert, so ist die Variable außerhalb scope zumindest nicht mehr **zugänglich** - die Variable ist überdeckt.

```
int a = 0; // a ist eine globale Variable
void f()
{
    long a = 11; // lokale Variable a überdeckt globales a in dieser scope
    a++;
    Serial.println(a); // lokales a ist jetzt 12
}
Serial.println(a); // globales a ist weiterhin 0
```

(2) Funktionen

In der Programmiersprache C sind **Funktionen**

meistens die einzige Möglichkeit, um das Programm zu strukturieren, also übersichtlich und nachvollziehbar für einen Programmierer zu gestalten.

In anderen Programmiersprachen werden weitere Strukturen entwickelt, um die Code **nachvollziehbar** und **wettert** zu machen:

classes / Klassen, namespaces, Templates, gronee, decoultor, etc.

Jede C-Funktion kann höchstens **einen** Return-wert (Rückgabewert) haben. Liefert die Funktion nichts zurück muss in der Deklaration der Datentyp **void** verwendet werden. Auf die **selben** Anweisung kann in diesem Fall **verzweigt** werden.

```
void f() // Return-Datentyp der Funktion
{
}
int xyz(float x) // Funktionsname (kein Schieklieken, Variablen, andere Funktionen)
{
    return (int) x; // Argument/Argumente der Funktion mit Datentyp
}
int xyz(float x, float y) // return-Anweisung (sichernde Rückkopplung zur aufgerufenen Funktion)
{
    return (int) (x+y);
}
```

Übung: → Schreiben Sie eine Funktion f, die "nichts" macht.
→ Schreiben Sie eine Funktion quadrat, die das Quadrat eines int berechnet.

→ Schreiben Sie eine Funktion hochdrei, die multipliziert die Funktion quadrat, x^3 für einen beliebigen bestimmt.

→ Verknüpfen Sie die Reihenfolge der Funktionen quadrat und hochdrei.

→ Schreiben Sie eine Funktion umfang_rechteck, die den Umfang eines ganzstetigen Rechtecks zurück liefert.

```
void f()
{
}
int quadrat(int x)
{
    return x*x;
}
```

```
int hochdrei(int x)
{
    int y = quadrat(x);
    return y*y;
}
```

```
int umfang_rechteck(int a, int b)
{
    return 2*(a+b);
}
```

Achtung

Normalerweise darf man in C und C++ keine Variablen und Funktionen verwenden, bevor sie **deklariert** wurden. Die Ausnahme Entwicklungsangestellt ist in dieser Hinsicht eine **Ausnahme**, die vorher sogenannte **forward-declarations** automatisch generiert werden.

Übung → Task folgender Code aus! Visual Studio Code

```
int hochrei(int x)
{
    int y = quadrat(x);
    return y*x;
}

int quadrat(int x)
{
    return x*x;
}

int main()
{
}
```

```
File Edit Selection View Go Run Terminal Help main.cpp - Untitled (Workspace) - Visual Studio Code

D:\VSCode\UNTITLED (WORKSPACE)\Visual Studio Code vs Arduino IDE\main.c
Get Started C main.c Extension: C/C++ Extension Pack main.cpp 1 PIO Home

1 #include <Arduino.h>
2
3 int hochrei(int x)
4 {
5     int y = quadrat(x);
6     return y*x;
7 }
8
9 int quadrat(int x)
10 {
11     return x*x;
12 }
13
14 void setup() {
15     // put your setup code here, to run once:
16 }
17
18 void loop() {
19     // put your main code here, to run repeatedly:
20 }

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL ...
Archiving .pio/build/uno/libFrameworkArduinoVariant.a
Compiling .pio/build/uno/Framework\Arduino\CDC.cpp.o
Compiling .pio/build/uno/Framework\Arduino\HardwareSerial.cpp.o
Compiling .pio/build/uno\Framework\Arduino\HardwareSerial10.cpp.o
src\main.cpp: In function 'int hochrei(int)':
src\main.cpp:5:11: error: 'quadrat' was not declared in this scope
    int y = quadrat(x);
           ^~~~~~
*** [.pio/build/uno\src\main.cpp.o] Error 1
[FAILED] Took 3.33 seconds
The terminal process "C:\Users\bernd\platformio\penv\Scripts\platformio.exe 'debug'" terminated with exit code: 1.
Terminal will be reused by tasks, press any key to close it.

@ 1 ▲ 0 DIO Debug (Visual Studio Code vs Arduino) | ✓ → ⚡ Default (Visual Studio Code vs Arduino) | Line 13 Col 1 | Sources: 2 | UTF-8 | CRLF | C++ | PlatformIO | 21°C Meist sonnig | 10:12 | 10/10/2022
```

Die Lösung zu diesem Problem ist eine sogenannte *forward declaration*:

Funktionsskopf mit Strichpunkt:

```
int quadrat(int x); // Dies ist ein forward-declaration.

int hochrei(int x)
{
    int y = quadrat(x);
    return y*x;
}

int quadrat(int x)
{
    return x*x;
}

int main()
{
}
```

Howto: C-Programmierung: "Vorfahrtsregeln"

Donnerstag, 19. Januar 2023 10:49

C Operator Precedence

[C](#)

[C language](#)

[Expressions](#)

The following table lists the precedence and associativity of C operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	<code>++ --</code>	Suffix/postfix increment and decrement	Left-to-right
	<code>()</code>	Function call	
	<code>[]</code>	Array subscripting	
	<code>.</code>	Structure and union member access	
	<code>-></code>	Structure and union member access through pointer	
	<code>(type){list}</code>	Compound literal(C99)	
2	<code>++ --</code>	Prefix increment and decrement [note 1]	Right-to-left
	<code>+ -</code>	Unary plus and minus	
	<code>! ~</code>	Logical NOT and bitwise NOT	
	<code>(type)</code>	Cast	
	<code>*</code>	Indirection (dereference)	
	<code>&</code>	Address-of	
	<code>sizeof</code>	Size-of [note 2]	
	<code>_Alignof</code>	Alignment requirement(C11)	
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code>	For relational operators <code><</code> and <code>≤</code> respectively	
	<code>> >=</code>	For relational operators <code>></code> and <code>≥</code> respectively	
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	
8	<code>&</code>	Bitwise AND	
9	<code>^</code>	Bitwise XOR (exclusive or)	
10	<code> </code>	Bitwise OR (inclusive or)	
11	<code>&&</code>	Logical AND	
12	<code> </code>	Logical OR	
13	<code>?:</code>	Ternary conditional [note 3]	Right-to-left
14 [note 4]	<code>=</code>	Simple assignment	
	<code>+= -=</code>	Assignment by sum and difference	
	<code>*= /= %=</code>	Assignment by product, quotient, and remainder	
	<code><<= >>=</code>	Assignment by bitwise left shift and right shift	

	<code>&= ^= =</code>	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

- ↑ The operand of prefix `++` and `--` can't be a type cast. This rule grammatically forbids some expressions that would be semantically invalid anyway. Some compilers ignore this rule and detect the invalidity semantically.
- ↑ The operand of `sizeof` can't be a type cast: the expression `sizeof(int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
- ↑ The expression in the middle of the conditional operator (between `?` and `:`) is parsed as if parenthesized: its precedence relative to `?:` is ignored.
- ↑ Assignment operators' left operands must be unary (level-2 non-cast) expressions. This rule grammatically forbids some expressions that would be semantically invalid anyway. Many compilers ignore this rule and detect the invalidity semantically. For example, `e = a < d ? a++ : a = d` is an expression that cannot be parsed because of this rule. However, many compilers ignore this rule and parse it as `e = ((a < d) ? (a++) : a) = d`, and then give an error because it is semantically invalid.

When parsing an expression, an operator which is listed on some row will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it. For example, the expression `*p++` is parsed as `*(p++)`, and not as `(*p)++`.

Operators that are in the same cell (there may be several rows of operators listed in a cell) are evaluated with the same precedence, in the given direction. For example, the expression `a=b=c` is parsed as `a=(b=c)`, and not as `(a=b)=c` because of right-to-left associativity.

Notes

Precedence and associativity are independent from [order of evaluation](#).

The standard itself doesn't specify precedence levels. They are derived from the grammar.

In C++, the conditional operator has the same precedence as assignment operators, and prefix `++` and `--` and assignment operators don't have the restrictions about their operands.

Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`sizeof ++*p` is `sizeof(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity is meaningful for member access operators, even though they are grouped with unary postfix operators: `a.b++` is parsed `(a.b)++` and not `a.(b++)`.

References

- C17 standard (ISO/IEC 9899:2018):
 - A.2.1 Expressions
- C11 standard (ISO/IEC 9899:2011):
 - A.2.1 Expressions
- C99 standard (ISO/IEC 9899:1999):
 - A.2.1 Expressions
- C89/C90 standard (ISO/IEC 9899:1990):
 - A.1.2.1 Expressions

See also

[Order of evaluation](#) of operator arguments at run time.

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
<code>a = b</code>	<code>++a</code>	<code>+a</code>	<code>!a</code>	<code>a == b</code>	<code>a[b]</code>	<code>a(...)</code>
<code>a += b</code>	<code>--a</code>	<code>-a</code>	<code>a && b</code>	<code>a != b</code>	<code>*a</code>	<code>a, b</code>
<code>a -= b</code>	<code>a++</code>	<code>a + b</code>	<code>a b</code>	<code>a < b</code>	<code>&a</code>	<code>(type)a</code>
<code>a *= b</code>	<code>a--</code>	<code>a - b</code>		<code>a > b</code>	<code>a->b</code>	<code>a ? b : c</code>
<code>a /= b</code>		<code>a * b</code>		<code>a <= b</code>	<code>a.b</code>	<code>sizeof</code>
<code>a %= b</code>		<code>a / b</code>		<code>a >= b</code>		
<code>a &= b</code>		<code>a % b</code>				<code>_Alignof</code>
<code>a = b</code>		<code>~a</code>				<code>(since C11)</code>
<code>a ^= b</code>		<code>a & b</code>				
<code>a <= b</code>		<code>a b</code>				
<code>a <<= b</code>		<code>a ^ b</code>				
<code>a >>= b</code>		<code>a << b</code>				
		<code>a >> b</code>				

[C++ documentation](#) for C++ operator precedence

Hintergrundwissen: Timer

Dienstag, 11. Oktober 2022 — 12:11



Timer,
Counter un...

Timer, Counter und Interrupts

01.08.2016 07:00 Uhr Dr. Michael Stal

In dieser Extraausgabe kommt das Thema Timer zur Sprache. Dabei geht es um mehr als den simplen Aufruf der `delay()`-Funktion.

In dieser Extraausgabe kommt das Thema Timer zur Sprache. Dabei geht es um mehr als den simplen Aufruf der `delay()`-Funktion.

Die Funktion `delay()` war bisher unser treuer und ständiger Begleiter in Sachen Zeitmessung. Sie steht standardmäßig in der Arduino-Bibliothek zur Verfügung, und erlaubt millisekundengenaue Wartezeiten. Was will man also mehr? Bei der Entwicklung eingebetteter Systeme – übrigens nicht nur bei Echtzeitfragestellungen – spielt Zeitmessung eine zentrale Rolle. `delay()` ist allerdings ein Mittel fürs Grobe, nicht für ausgefeilte Einsatzgebiete.

Was stört konkret an den simplen Funktionen à la `delay()` oder `millis()`?

- Beschränkte Auflösung: Eine Genauigkeit in Millisekunden mag für Wartezeiten im Maßstab menschlicher Wahrnehmung passabel erscheinen. Im Rahmen technischer Ereignisse ist diese Auflösung ungenügend. Bei einer ATMega- oder ATTiny-CPU auf Arduino-Boards haben wir es in der Regel mit Taktfrequenzen von 8 MHz oder 16 MHz zu tun. Bei 16 MHz dauert ein einzelner Takt 0,000000625 sec. Das ist im Vergleich zu einer Millisekunde (0,001 sec) ein Faktor von 1:16.000. Während jeder Millisekunde verarbeitet ein ATMega Tausende von Maschinenbefehlen. Daher sind Reaktionszeiten in dieser Größenordnung alles andere als akzeptabel.
- Aktives Warten: Während eines Delays wie `delay(42)` ist der Arduino-Sketch zum aktiven Warten verurteilt. Die gewählte Wartezeit in Millisekunden ist zudem oft Ergebnis einer groben Schätzung nach dem Motto: "Vermutlich braucht die Initialisierung rund 2 Sekunden. Zur Sicherheit verwenden wir aber einfach 3 Sekunden." Das führt zu Ineffizienz.
- Mangelnde Flexibilität: Timernutzung mittels `delay()` dient einzig zum Integrieren von Wartezeiten. Weitere Dienste, insbesondere periodische Trigger wie ein Watchdog-Timer, sind hiermit nicht oder nur unzureichend umsetzbar.

Timer on the Chip

Würde sich ein AVR-Prozessor mit dieser Art von grober Zeitmessung zufrieden geben, müssten wir bei Arduino-Boards viele funktionale Abstriche in Kauf nehmen. So benötigt die Pulse-Width-Modulation an digitalen Ports eine hohe Timerauflösung, um genaue Aktivitätszyklen (Duty-Cycles) einstellen zu können. Benötigen Sie zum Beispiel eine durchschnittliche Ausgangsspannung von 2,765643 V an einem digitalen Ausgang, lässt sich das nur durch Genauigkeiten jenseits der Möglichkeiten von `delay()` oder `millis()` bewerkstelligen.

Bei Bussystemen wie I²C oder SPI sind Vorgänge im Mikrosekundenbereich typisch und notwendig. Will ein Programm angeschlossene Komponenten über diese Bussysteme ansteuern, wäre eine Auflösung von groben Millisekunden ein Show-Stopper. Nicht zuletzt erweisen sich für die Kontrolle von Servomotoren oder zur Tonerzeugung zeitliche Bedingungen als notwendig, die durch Millisekunden-getriggerte Timer schlicht nicht umsetzbar wären.

Da wir aus bisheriger Erfahrung wissen, dass Arduino-Boards sehr gute Unterstützung für PWM, Bussysteme, Servo-Ansteuerung und Tonerzeugung leisten, stellen sich zwei Fragen:

1. Wie schafft es ein AVR-Prozessor intern, die dafür erforderlichen zeitlichen Auflösungen umzusetzen?
2. Können wir uns als Entwickler diese Funktionen ebenfalls zunutze machen?

Den Takt angeben

Für die genannten Aufgaben integrieren die AVR-Microcontroller diverse Timer mit zugeordneten Zählregistern von 8 oder 16 Bits Breite. Diese Register starten mit einem initialen Wert von 0. Ihr Inkrementieren, also das eigentliche Hochzählen, erfolgt automatisch und periodisch. Laufen die jeweiligen Register über, wird ein Timer-Überlauf-Interrupt ausgelöst.

Ein häufiger Irrtum lautet übrigens, dass die CPU des Arduino die Timer antreibt. Die Timer eines Arduino bzw. eines AVR-Mikrocontrollers von ATmel sind von der CPU bzw. MCU unabhängig. Diese Tatsache sollten Sie sich bei den nachfolgenden Diskussionen vor Augen halten.

Um eine möglichst hohe Auflösung zu erhalten, könnte der erste Ansatz darin liegen, die Updates der Counter (d.h. deren Inkrementieren) synchron zum Prozessortakt vorzunehmen. Das hat allerdings einen entscheidenden Schönheitsfehler. Bei einer angenommenen Taktfrequenz von 16 MHz wäre der Überlauf eines 8-Bit-Timers nach 16 Mikrosekunden erreicht, der eines 16-Bit-Registers nach rund 4,1 Millisekunden.

Das ist natürlich vorteilhaft für kurzzeitige Zeitintervalle. Was aber, wenn wir längere Zeiträume überdecken wollen? Um dies zu ermöglichen, bieten die Mikrocontroller sogenannte Prescaler. Diese konfigurieren, nach wie vielen Taktzyklen das System ein Zählregister inkrementieren soll. Mögliche Werte liegen bei 8, 64, 256 oder 1024. Eine Prescale-Einstellung von 1024 führt beispielsweise zum Inkrementieren des Zählers jeweils nach 64 Mikrosekunden bzw. 1024 Taktzyklen bei 16 MHz Taktfrequenz, sodass ein 16-Bit-Zähler erst nach 4,2 Sekunden überläuft.

Um präzise Intervalle zu programmieren, lassen sich Timer-Register mit Zählerständen vorbelegen statt sie bei 0 starten zu lassen. Einmal angekommen, würden wir gerne alle 0,5 Sekunden eine LED abwechselnd ein und ausschalten. Die gewünschte Frequenz des Timers wäre somit 2 Hz. Ein Takt besteht im Auslösen eines Timer-Überlaufs. Wie genau lässt sich dies erreichen?

Wir haben es mit folgenden Parametern zu tun:

- *bits* definiert die Größe des Zählerregister in Bits, etwa 16 für einen 16-Bit-Timer.
- *maxcount* entspricht dem maximalen Zahlenwerts 2^{bits} .
- *prescale* ist der oben erläuterte konfigurierbare Prescalewert, also die Zahl der Taktzyklen bis ein weiteres Inkrementieren des Timeregisters erfolgt.
- *cpufreq* repräsentiert die CPU-Frequenz. Der Taktzyklus berechnet sich folglich aus $1 / \text{cpufreq}$.
- *initcount* ist der vorbelegte Startwert des Zählregisters.
- *count* ist die notwendige Zahl von Inkrementierungen, um einem Timeroverflow auszulösen. Es gilt: $\text{count} = \text{maxcount} - \text{initcount}$.
- *deltaT* bezeichnet das gewünschte Zeitintervall bis zum Auslösen des Timer-Overflows. Man könnte auch definieren: $\text{deltaT} = 1 / \text{timerfreq}$ (gewünschte Zahl von Timer Overflows pro Sekunde).

Es gilt $\text{prescale} / \text{cpufreq} * \text{count} = \text{deltaT}$

$$\Rightarrow \text{count} = \text{deltaT} * \text{cpufreq} / \text{prescale}$$

$$\Rightarrow \text{maxcount} - \text{initcount} = \text{deltaT} * \text{cpufreq} / \text{prescale}$$

$$\Rightarrow \text{initcount} = \text{maxcount} - \text{deltaT} * \text{cpufreq} / \text{prescale}$$

Beispielsrechnung: Alle 0,5 Sekunden soll ein Timer-Overflow-Interrupt stattfinden.

- Wir verwenden einen 16-Bit-Timer: $\text{bits} = 16 \Rightarrow \text{maxcount} = 2^{16} = 65536$.
- Wir benötigen einen Timer Overflow pro halbe Sekunde. $\text{deltaT} = 0,5 \text{ sec} = 1 / \text{timerfreq}$
- Die Taktfrequenz des Arduino-Board beträgt $\text{cpufreq} = 16 \text{ MHz} = 16.000.000 \text{ Hz}$
- Als Prescale-Wert liegt $\text{prescale} = 256$ vor.

Der Timer startet statt mit 0 mit folgendem Anfangszählerstand $\text{initcount} = 65.536 - 8.000.000 / 256 = 34.286$

Das Timer-Register muss initial mit 34.286 starten, damit bis zum Timer Overflow – bei Überschreiten von 65.636 – genau eine halbe Sekunde vergeht. In jedem Durchlauf der Interrupt-Service-Routine ist der Zähler jeweils wieder mit 34.286 initialisieren.

Ein entsprechender Sketch könnte wie folgt aussehen. Auf die darin erwähnten Register kommen wir später noch zu sprechen.

```
#define ledPin 13

void setup()
{
  pinMode(ledPin, OUTPUT); // Ausgabe LED festlegen
```

```

// Timer 1
noInterrupts();           // Alle Interrupts temporär abschalten
TCCR1A = 0;
TCCR1B = 0;

TCNT1 = 34286;           // Timer nach obiger Rechnung vorbelegen
TCCR1B |= (1 << CS12); // 256 als Prescale-Wert spezifizieren
TIMSK1 |= (1 << TOIE1); // Timer Overflow Interrupt aktivieren
interrupts();            // alle Interrupts scharf schalten
}

// Hier kommt die selbstdefinierte Interruptbehandlungsroutine
// für den Timer Overflow
ISR(TIMER1_OVF_vect)
{
    TCNT1 = 34286;           // Zähler erneut vorbelegen
    digitalWrite(ledPin, digitalRead(ledPin) ^ 1); // LED ein und aus
}

void loop()
{
    // Wir könnten hier zusätzlichen Code integrieren
}

```

World of Timers

Ein Arduino weist nicht nur einen einzelnen Timer sondern mehrere Timer auf. Kein Wunder, sind Timer doch essenzielle Grundkomponenten für verschiedene Aufgaben eines Mikrocontrollers.

- Timer 0 (8 Bit) Verwendet für Funktionen wie *delay()*, *millis()*, *micros()*
- Timer 1 (16 Bit) Verwendet von der Servo-Bibliothek
- Timer 2 (8 Bit) Verwendet von der Tone-Bibliothek
- Timer 3 (16 Bit) Nur Mega
- Timer 4 (16 Bit) Nur Mega
- Timer 5 (16 Bit) Nur Mega

Beim Arduino:

- PWM Pins 5 und 6 kontrolliert durch Timer 0
- PWM Pins 9 und 10 kontrolliert durch Timer 1
- PWM Pins 3 und 11 kontrolliert durch Timer 2

Beim Arduino Mega:

- PWM Pins 4 und 13 kontrolliert durch Timer 0
- PWM Pins 11 und 12 kontrolliert durch Timer 1
- PWM Pins 9 und 10 kontrolliert durch Timer 2
- PWM Pins 2, 3 und 5 kontrolliert durch Timer 3
- PWM Pins 6, 7 und 8 kontrolliert durch Timer 4
- PWM Pins 44, 45 und 46 kontrolliert durch Timer 5

Es gibt zusätzlich diverse Einschränkungen zu beachten:

- Pin 11 des Arduino ist zugleich Pin mit PWM-Fähigkeit und Master-Out-Slave-In-Pin des SPI-Busses. Demzufolge lassen sich beide Funktionen nicht gleichzeitig nutzen.
- Für die Tonerzeugung ist mindestens Timer 2 im Einsatz. Daher lassen sich die Pins 3, 11 (Arduino) bzw. 9, 10 (Arduino Mega) nicht für PWM nutzen, solange die Funktion `tone()` im Einsatz ist.
- Beim Anschluss von Servos müssen sich Timer exklusiv dieser Aufgabe widmen, weshalb sich die Zahl der Digitalpins mit PMW-Unterstützung reduziert.

Timer-Register

Wie im Sketch weiter oben ersichtlich, erfolgt die Steuerung der Timer-Funktionalität über verschiedene Register. Das Symbol μ repräsentiert die Nummer des jeweiligen Timers, also 0, 1, 2, ..., . TCNT μ ist daher das Zählregister von Timer μ . Das Zählregister für Timer 1 lautet dementsprechend TCNT1, das für Timer 0 TCNT0.

Der Einfachheit halber beziehen sich die nachfolgenden Diskussionen wie auch die beiden Beispiel-Sketches auf Timer 1. Des Weiteren erspare ich Ihnen eine Aufzählung sämtlicher Details, sondern fokussiere mich auf die relevanten Eigenschaften.

TCCR1A (Timer Counter/Control Register): die Flags PWM10 und PWM11 erlauben eine Festlegung der Auflösung für den Fall, dass Timer 1 zur PWM-Steuerung dient. Ausgangsbasis sei die Vereinbarung `TCCR1A = 0;`:

- Kein PWM: `no-op`
- 8-Bit PWM: `TCCR1A |= (1 << PWM10);`
- 9-Bit PWM: `TCCR1A |= (1 << PWM11);`
- 10-Bit PWM: `TCCR1A |= (1 << PWM10); TCCR1A |= (1 << PWM11);`

TCCR1B (Timer Counter/Control Register): Konfiguration des Prescaler.

- Kein Prescaler: `TCCR1B = 0; TCCR1B |= (1 << CS10);`
- Prescale = 8: `TCCR1B = 0; TCCR1B |= (1 << CS11);`
- Prescale = 64: `TCCR1B = 0; TCCR1B |= (1 << CS10); TCCR1B |= (1 << CS11);`
- Prescale = 256: `TCCR1B = 0; TCCR1B |= (1 << CS12);`
- Prescale = 1024: `TCCR1B = 0; TCCR1B |= (1 << CS10); TCCR1B |= (1 << CS12);`

Weitere Kombinationen ermöglichen die externe Steuerung über den T1-Pin.

TCNT1 (Timer/Counter Register): d.h. der eigentliche Zähler.

OCR1 (Output Compare Register): Ist der Zähler in TCNT1 gleich dem Inhalt des OCR1, erfolgt ein Timer Compare Interrupt.

ICR1 (Input Capture Register, nur für 16-Bit-Register): Messung der Zeit zwischen zwei Flanken des Input Capture Pins, die durch externe Schaltungen zustande kommen. Lässt sich auch zur Messung der Umdrehungszahl eines Motors einsetzen. Wird auch über Einstellungen von TCCR1A mit beeinflusst.

TIMSK1 (Timer/Counter Interrupt Mask Register): hier lassen sich Timer Interrupts unterbinden oder erlauben.

- Scharf schalten des Output Compare Interrupts: `TIMSK1 |= (1 << OCIE1A)`
- Scharf schalten des Timer Overflow Interrupts (16 Bit): `TIMSK1 |= (1 << TOIE1)`
- Scharf schalten des Timer Overflow Interrupts (16 Bit): `TIMSK1 |= (1 << TOIE0)`

TIFR1 (Timer/Counter Interrupt Flag Register): Hier lassen sich noch unverarbeitete Interrupts feststellen. Die Bits korrespondieren mit denen von TIMSK1.

Alternative Methode CTC

Statt einen Interrupt bei Überlauf eines Timer-Registers auszulösen wie im oberen Sketch, gibt es die alternative Option namens CTC (Clear Timer on Compare Match). Bei dieser vergleicht der Mikrocontroller, ob der Inhalt des Zählerregisters identisch mit dem Inhalt des zum Timer gehörigen OCR (Output Compare Registers) ist. Falls ja, wird ein Timer Compare Interrupt ausgelöst und das Register auf 0 zurückgesetzt. Wiederum soll jede halbe Sekunde ein Interrupt stattfinden.

Bei einem Prescaling von 256 und einer Taktfrequenz von 16 MHz können wir die obige Formel für count anwenden: $count = deltaT * cpufreq / prescale = 0.5 * 16.000.000 / 256 = 31.256$.

Obiger Sketch würde sich in diesem Fall also ändern in:

```
#define ledPin 13

void setup()
{
    pinMode(ledPin, OUTPUT); // Ausgabe LED festlegen

    // Timer 1
    noInterrupts();          // Alle Interrupts temporär abschalten
```

```

TCCR1A = 0;
TCCR1B = 0;
TCNT1 = 0;           // Register mit 0 initialisieren
OCR1A = 31250;      // Output Compare Register vorbelegen
TCCR1B |= (1 << CS12); // 256 als Prescale-Wert spezifizieren
TIMSK1 |= (1 << OCIE1A); // Timer Compare Interrupt aktivieren
interrupts();        // alle Interrupts scharf schalten
}

// Hier kommt die selbstdefinierte Interruptbehandlungsroutine
// für den Timer Compare Interrupt
ISR(TIMER1_COMPA_vect)
{
    TCNT1 = 0;           // Register mit 0 initialisieren
    digitalWrite(ledPin, digitalRead(ledPin) ^ 1); // LED ein und aus
}

void loop()
{
    // Wir könnten hier zusätzlichen Code integrieren
}

```

Zusammenfassung

In diesem Extra ging es um den Umgang mit ausgefeilten Timer-Funktionen bei Prozessoren der ATmel-Familie (ATMega, ATTiny). Damit sollten Sie jetzt ein tiefergehendes Verständnis der Thematik besitzen. Wollen Sie es noch detaillierter wissen, verweise ich Sie auf Dokumente des Herstellers ATmel wie zum Beispiel [das hier \[1\]](#).

Mehr über die praktische Anwendung dieser Funktionalität erfahren Sie in fortgeschrittenen Anwendungen wie etwa [Sound Synthesis \[2\]](#), [PWM \[3\]](#) oder [Servomotoren \[4\]](#). Eine weitere lohnenswerte Lektüre ist dieser [Artikel von Adafruit. \[5\]](#)

Es ist aber auch eine gute Idee, das Gelernte durch eigene Experimente zu vertiefen.

MEHR INFOS

Was bisher geschah:

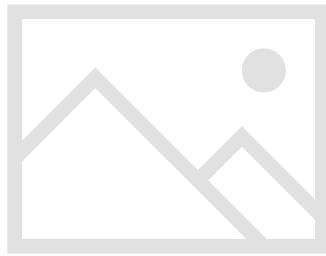
- [Arduino goes ESP8266 \[6\]](#)
- [Bright Side of Arduino \[7\]](#)
- [Genuino bzw. Arduino unplugged \[8\]](#)
- [Arduino-/IoT-Extra – Bibliotheken selbst implementieren \[9\]](#)

Warten Sie, während OneNote diesen Ausdruck lädt...



✗

Warten Sie, während OneNote diesen Ausdruck lädt...



✗

Projekt: Wärmebildkamera

Donnerstag, 10. November 2022 07:51



Druckversi...
- Portable...

Ideen: SD-Karte nutzen

aug 8833 8x8 Sensor verwenden

mlx 90640 32x24 Sensor verwenden Performance ??

st 7789 240x240 Display verwenden

Performance optimieren: hrv → rgb schneller

interpolation schneller

mlx 90621 sehr floating point Lastig.

kann man da was optimieren?

code Verbesserungen: Buffer-size wieder auf 32 setzen (I2C ☹)

Höhere Takt I2C, SPI möglich?

tft Bibliothek durch tft-espi ersetzen

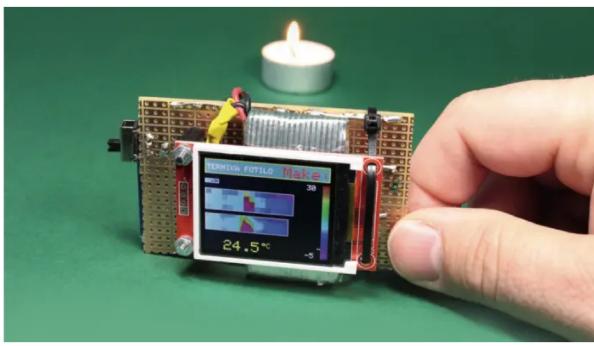
06.10.22, 12:14

Druckversion - Portable Wärmebildkamera selber bauen | Make

Make:

Portable Wärmebildkamera selber bauen

30.09.2022 18:28 Uhr Felix Pfeifer, Florian Schäffer



Termika Fotilo: Esperanto für Wärmebildkamera

Unsere Wärmebildkamera ist unschlagbar günstig. Die Auflösung von 16x4 Pixel klingt zwar niedrig, das Bild lässt sich aber mit ein wenig Mathematik verbessern.

Der Sensor MLX90621 von Melexis ist hauptsächlich für Industrieanwendungen zur kontaktlosen Temperaturmessung konzipiert. Eine mögliche Anwendung ist beispielsweise das Suchen von Kältebrücken in Haus oder Wohnung, eine Raumüberwachung und Zutrittskontrolle oder ein Brandmelder, der auf Hitze und nicht auf Verrauchung reagiert.

Die geringe Auflösung schließt eine Nutzung zur (pseudo-)fotorealistischen Bildgewinnung eher aus. Natürlich kochen auch wir nur mit Wasser und können nicht zaubern, aber ein bisschen was geht da schon noch. Wir haben den Sensor mit einem Arduino und einem farbigen Grafikdisplay kombiniert (siehe Schaltbild). Ausgestattet mit einem Akku liefert unsere *Termika Fotilo* getaufte Wärmebildkamera ganz ansehnliche Resultate.

MEHR ZU: ARDUINO-PROJEKTE

- [Termika Fotilo: Bauanleitung für eine einfache portable Wärmebildkamera \[1\]](#)
- [Bastelanleitung: Roboter-Mülltonne "Tonni" mit einem Arduino bauen \[2\]](#)
- [Anleitung: Wie Sie mit dem Arduino smarte Treppenbeleuchtung einrichten \[3\]](#)
- [Anleitung: Rommé-Buzzer mit Arduino bauen \[4\]](#)
- [Smart Garden: Rasenkabelfinder mit Arduino bauen \[5\]](#)

<https://www.heise.de/ratgeber/Termika-Fotilo-Bauanleitung-fuer-eine-einfache-portable-Waermebildkamera-7280578.html?view=print>

1/22

- [Anleitung: Wie Sie mit einem Roboter Ihre Sitzhaltung verbessern \[6\]](#)

Mit dem Aufbau zeigen wir, wie Sie den Sensor in Betrieb nehmen und die Daten auswerten können. Anschließend dürfte es kein Problem sein, den Sensor auch in Ihren eigenen Applikationen zu nutzen, die vielleicht keine grafische Darstellung der Temperaturen benötigt, sondern lediglich die Umgebung thermisch überwachen soll.

KURZINFO

Darum geht's

- Low-Budget-Kamera für Infrarot-Wärmebilder
- Farbräume
- Interpolation von Bilddaten

Checkliste

Zeitaufwand: 2 Stunden

Kosten: 85 Euro

Programmieren: Code auf ATmega übertragen

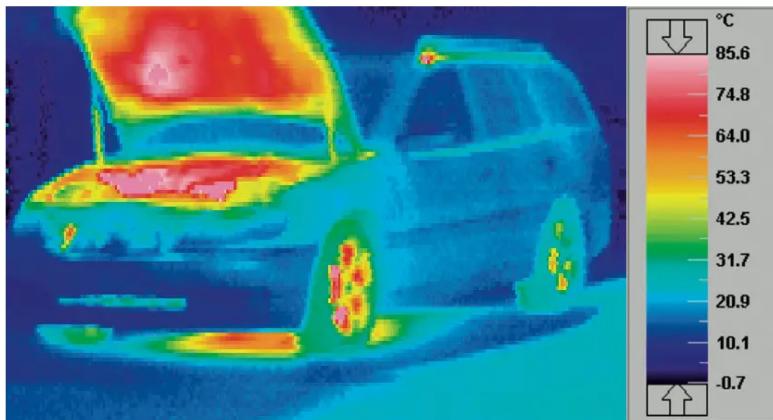
Löten: Bauteile auf Lochrasterplatine anordnen und verbinden

Material

- MLX90621 IR-Sensor
- Arduino Nano
- Platine Loch-/Streifenraster, ca. 9 cm x 5 cm
- 1,8" 28 x 160, SPI, TFT/LCD-Modul ST7735S 5 V/3,3 V, oft mit SD-Kartenslot (wird nicht gebraucht)
- Lithium-Polymer-Akkumulator ca. 1000 mAh/3,7 V
- USB-LiPo-Ladegerät TP4056/TE420, 3,6 V mit LiPo-Schutz
- USB-Steckernetzteil mit USB-Kabel, 5 V/500 mA
- LD33V Festspannungsregler oder LD1117V33 oder anderer, siehe Text
- 2 x 100 nF Kondensator
- 2,2 µF/35 V Elektrolytkondensator
- 2 x 4,7 kΩ Widerstände

- 1N4004 Diode
- Schalter Schiebeschalter 1x UM, liegend, Print
- Schaltlitz/Drahtbrücke oder LD1117V33 oder anderer, siehe Text

Alles zum Artikel unter: [make-magazin.de/xgk8 \[7\]](https://make-magazin.de/xgk8)



Wärmebild eines Autos mit heißem Motorraum und erwärmten Felgen

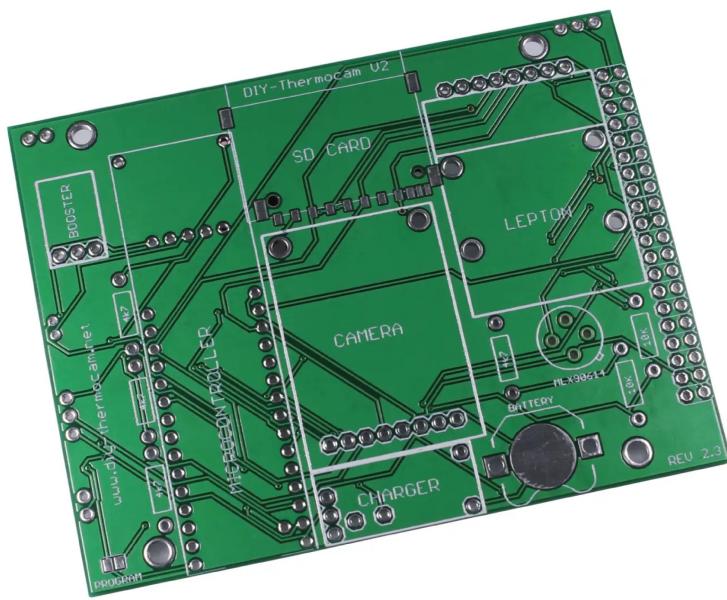
Wärmebildkameras vom Markenhersteller (Flir) sind teuer und für Privatanwender meistens unerschwinglich. Das Modell TG165 (ein Pyrometer, was im Prinzip eine Wärmebildkamera mit geringer Auflösung ist) kostet mit etwa 350 Euro schon fast so viel wie Thermokameras, die ans Smartphone gedockt werden. Nach oben ist bis in den Bereich von um die 2500 Euro alles möglich.

MARKE UND TECHNIK

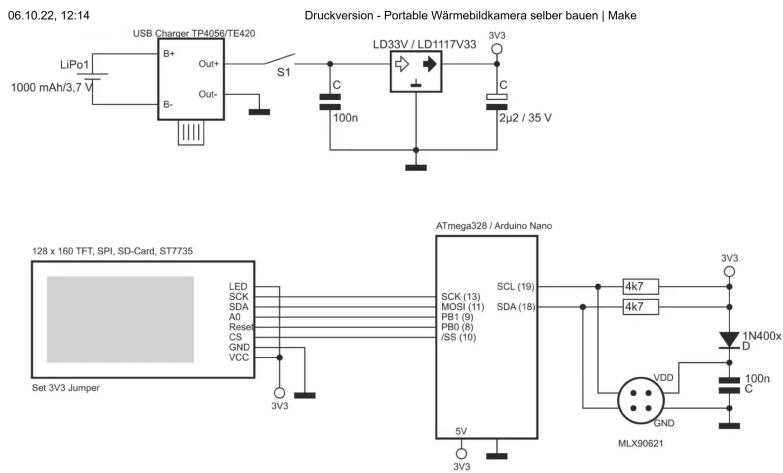
FLIR (Forward Looking Infrared, etwa *vorwärts gerichtetes Infrarotgerät*) bezeichnet zum einen die (militärische) Technik, Infrarotstrahlung in Richtung von Fahrzeug- oder Flugkörperachsen zu bestimmen. Flir Systems ist zum anderen ein amerikanischer Hersteller von Wärmebildsystemen.

Eine Alternative stellt die DIY-Thermocam von Max Ritter [8] dar, der sein Projekt auch auf der Maker Faire 2016 in Berlin zeigte. Hier wird der Lepton-3-Sensor von Flir genutzt, der eine thermische Auflösung von 160×120 Pixel aufweist. Als Bausatz kostet das Set um die 540 US-Dollar (plus Versand, Steuern und Zoll). Der ältere Sensor Lepton 2.5 (80×60 Pixel) kann alternativ aus gebrauchten Kameras ausgebaut werden, wodurch die Kosten um etwa 50 bis 100 Euro reduziert

werden. Gegenüber unserer Billigvariante haben die kommerziellen Produkte natürlich einige Vorteile, zu denen beispielsweise die Software zur Analyse der Bilder gehört.



Platine aus dem Bausatz für die DIY-Thermocam von Max Ritter



Schaltplan für die Thermocam

Vom Pyrometer zur Wärmebildkamera

Pyrometer sind einfache Infrarot-Thermometer zur berührungslosen Temperaturnmessung ohne Wärmebilddarstellung. Bei den meisten Modellen, die ab etwa 40 Euro von deutschen Händlern angeboten werden, markiert ein integrierter Laserpointer den Messpunkt. Ein Infrasensor (LWIR, long-wavelength infrared) misst die vom Objekt abgegebene Wärmestrahlung an einer Stelle. Wärmebildkameras besitzen im Prinzip einfach mehrere IR-Sensoren in einer Matrix angeordnet. Diese liefern dann ein ortsaufgelöstes Thermografiebild. Je mehr Sensoren und somit Pixel vorhanden sind, desto höher ist das Bild aufgelöst und es können auch feine Strukturen erkannt werden. Wärmekameras werden zum Beispiel von Rettungskräften eingesetzt, um Personen in verrauchten Räumen oder im Nebel zu finden. Bei Gebäuden lassen sich mit den Kameras schlecht isolierte Bereiche und Wärmebrüchen schnell aufspüren. In der Medizin sucht man damit nach Entzündungen unter der Haut, weil diese Bereiche besser durchblutet werden.

Sensor MLX90621

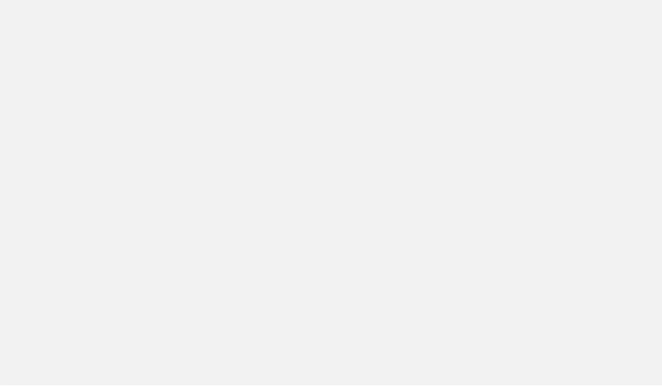
Den Sensor bekommt man beispielsweise bei DigiKey oder Mouser. Ein dreistelliger Buchstabencode nach der Typnummer gibt an, welchen Öffnungswinkel (FOV: field of view, Sichtfeld) der Chip hat. Der von uns genutzte BAB mit der mittleren Bauhöhe weist einen Bereich von $60^\circ \times 16^\circ$ auf. Im Bauteil ist bereits die notwendige Optik integriert, sodass er direkt eingesetzt werden kann. Diese Optik bestimmt den Öffnungswinkel für die einfallende Infrarotstrahlung: je größer dieser ist, desto weitwinkliger wird der Sichtbereich. Die Kommunikation läuft bequem per I²C ab, sodass nur zwei Pullup-Widerstände für die Verbindung zu einem ATmega erforderlich sind. Es gibt ein paar ähnliche Sensortypen wie den MLX90620, der günstiger, aber nicht kompatibel ist. Die Adressen zum Auslesen der Daten und die Berechnung der Messwerte weichen dann ab. Gegenüber diesem bietet der von uns verwendete MLX90621 mehr Präzision und Geschwindigkeit, sodass theoretisch auch

Videoaufnahmen möglich sind. Von den IR-Sensoren können Temperaturen (T_o) zwischen -20°C und $+300^{\circ}\text{C}$ gemessen werden. Die Umgebungstemperatur (T_a) kann zwischen -40°C und $+85^{\circ}\text{C}$ bestimmt werden.



Sensor MLX90621

Der Sensor beinhaltet auch noch die Controller-Logik zur Ansteuerung und Kommunikation. Im EEPROM des Chips sind werkseitig die Konfiguration und Parameter zur Temperaturkorrektur jedes einzelnen der 64 IR-Sensoren gespeichert. Zudem gibt es allgemeine Korrekturwerte, die in die Berechnung der tatsächlichen Messwerte einfließen. Im RAM werden die IR-Werte abgespeichert und können ausgelesen werden. Das Datenblatt beschreibt, wie die Werte ausgelesen und interpretiert werden. Die Berechnung der tatsächlichen Temperaturen ist etwas knifflig, aber ebenso gut dokumentiert und soll hier nicht weiter betrachtet werden.



[9]

Mini-Thermocam

Unsere eigene Wärmebildkamera soll billig und einfach werden. Mit dem ATmega 328 auf einem Arduino Nano wird der Bildsensor ausgelesen. Auf einem farbigen TFT mit 128×160 Pixeln erfolgt die Ausgabe des Wärmebildes und der Umgebungstemperatur. Wie bei Thermocams üblich, nutzen wir eine Falschfarbendarstellung. Dabei wird jedem Temperaturwert ein Farbwert zugeordnet. Analog zum Sprachgebrauch sind kalte Temperaturen blau und warme rot. Weil bei 16×4 Pixeln nicht viel zu sehen ist, gibt es eine vergrößerte Darstellung, bei der einfach jeder Pixel siebenmal nebeneinander gezeigt wird. Damit die Thermocam mobil wird, versorgen wir die Schaltung über einen Lithium-Polymer-Akku mit etwa 3,7V.



Bildschirmsdarstellung der Mini-Wärmebildkamera: Unter der Titelzeile erst das Bild mit einem Pixel pro Sensorpunkt, dann die vergrößerte Darstellung, schließlich die interpolierte Anzeige. Ganz unten die Umgebungstemperatur, rechts die Temperaturskala

Beim Display greifen wir auf einen einfachen Typ mit ST7735S-Controller zurück, der per SPI angesteuert wird. Für den gibt es bereits eine Library für Arduinos, sodass sich ein Nano-Board zur Steuerung anbietet. So können wir den Nachbau auf weniger Baugruppen beschränken und alles auf einer Lochrasterplatine unterbringen. Der Akku wird über einen USB-Laderegler angeschlossen, der auch gleich Schutz vor Tiefentladung bietet.



Eine (schlechte) Digicam liefert von der Straßenszene dieses Foto mit geringer Auflösung. Natürlich ist das Bild kleiner als die echten Gebäude am Las Vegas Strip. Flüchtig betrachtet sind die Qualitätsverluste daher kaum

sichtbar. Es fehlen aber unwiderruflich Informationen: Die Schrift ist nicht mehr zu lesen, sondern besteht nur noch aus einzelnen Punkten.

Ein Low-Drop-Festspannungsregler vom Typ LD33V versorgt die restliche Schaltung mit 3,3 V. Die Drop-Out-Spannung liegt bei diesem Typ (baugleich zum LD1117) bei 1 V. Damit ist er eigentlich nicht für den Akkubetrieb geeignet, es klappt aber, so dass Sie sich an den Schaltplan halten können. Besser wären Very-Low-Drop-Out-Typen wie der LF33CV oder ein XC6204 (als SMD-Variante), die schon ab einer Eingangsspannung arbeiten können, die lediglich nur 200 mV oder 60 mV größer als die Ausgangsspannung sein muss (je nach Strombelastung). Abhängig vom Typ ist darauf zu achten, dass die Anschlüsse nicht einheitlich sind und unterschiedliche Werte für die Stützkondensatoren empfohlen werden. Unterschreitet die Eingangsspannung die minimale Spannung für den Spannungsregler, wird es etwas unsicher, wie die Schaltung sich verhält. Der Arduino und das Display werden noch arbeiten, aber der Sensor kann nicht mehr initialisiert oder ausgelesen werden.

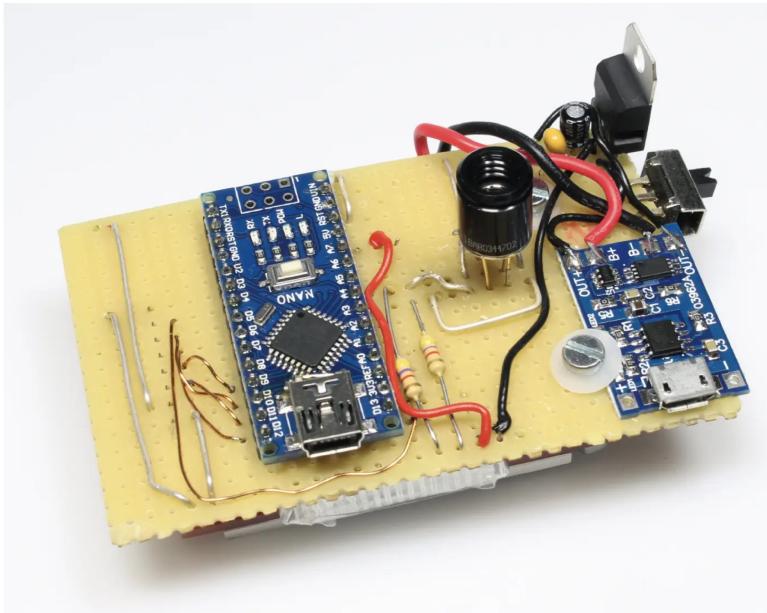
DROP-OUT-SPANNUNG

Die Drop-Out-Spannung gibt an, wie viel höher die Eingangsspannung gegenüber der Ausgangsspannung bei einem Linearregler sein muss. Die Festspannungsregler vom Typ LM78xx benötigen 2 V am Differenzspannung. Um beispielsweise mit einem 7805 die Ausgangsspannung auf 5 V zu regeln, müssen am Eingang mindestens 7 V anliegen. Je kleiner die Drop-Out-Spannung eines Reglers ist, desto näher kann die Eingangsspannung an der Ausgangsspannung liegen. Der LD33V regelt den Ausgang auf 3,3 V und benötigt mindestens 4,3 V am Eingang. Fällt die Spannungsdifferenz unter die Drop-Out-Spannung, kann das Verhalten des Spannungsreglers nicht mehr vorhergesagt werden – in gewissen Grenzen funktionieren sie meistens trotzdem noch.



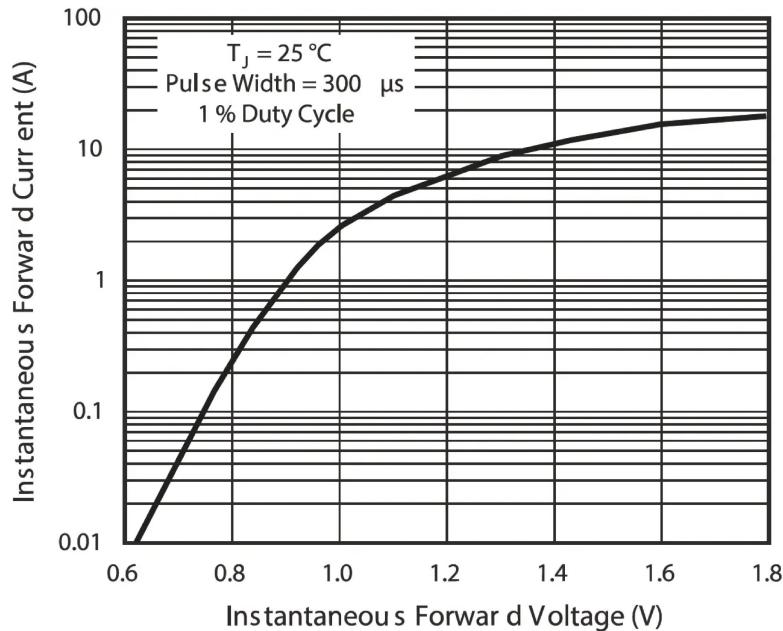
Eine Straßenszene aus Las Vegas, wie sie ein Betrachter vor Ort mit eigenen Augen wahrnehmen würde: Man könnte mit guten Augen den Text "NO ROOMS ..." lesen.

Aus Sicherheitsgründen wird auf der Arduino-Platine an der Mini-USB-Buchse der Pin für die 5-V-Spannungsversorgung aufgetrennt, sodass der ATmega darüber nur noch programmiert werden kann und ausschließlich die externe Spannungsversorgung die Schaltung versorgt. Andernfalls würde der Controller beim Anschluss an den PC mit 5 V arbeiten, was zu Schäden an Display und Thermosensor führen kann.



Vorderseite unserer Termika Fotilo mit dem IR-Sensor.

Auf der Display-Rückseite muss die kleine Lötbrücke bei J1 neben dem dreipoligen Festspannungsregler für den Betrieb an 3,3 V gesetzt werden. Der Thermosensor kann zwar mit 3,3 V betrieben werden, liefert aber bei 2,6 V die besten Ergebnisse. Mit einer Diode 1N4001 (bis 1N4007) reduzieren wir deshalb die Versorgungsspannung auf etwa 2,7 V. Laut Datenblatt liegt die Vorwärtsspannung, die an der Diode abfällt, bei etwas über 0,6 V bei der geringen Strombelastung (ca. 5 mA) durch den IR-Sensor.



Typical Instantaneous Forward Characteristics

Verlauf der Vorwärtsspannung der Diode 1N400x (Datenblatt Vishay).

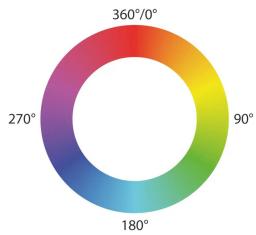
SCHWARZER KÖRPER ALS STRAHLUNGSREFERENZ

Der Emissionsgrad ϵ eines Objekts gibt an, wie viel Strahlung er im Vergleich zu einem idealen Wärmestrahlern, einem schwarzen Körper, abgibt. Ein schwarzer Körper ist ein hypothetischer idealisierter Gegenstand, der jegliche auf ihn treffende elektromagnetische Strahlung bei jeder Frequenz vollständig absorbiert, also nichts reflektiert. Wärmebildkameras messen die abgegebene Strahlung eines Objektes. Je nach Material gibt ein Körper bei gleicher Temperatur unterschiedlich viel Wärmestrahlung ab. Um die tatsächliche Temperatur des Körpers anhand der Strahlung zu bestimmen, muss der Emissionsgrad bekannt sein und berücksichtigt werden. Viele Materialien haben einen Wert zwischen 0,8 und 0,9. Bei (blanken) Metallen liegt der Emissionsgrad hingegen oft deutlich unter 0,3, was zu entsprechenden stark verfälschten Temperaturwerten führt, wenn dies nicht berücksichtigt wird. Die von uns entwickelte Software bietet bisher keine Vorgabe des Emissionsgrads.

Den gesamten Sourcecode können Sie auf GitHub herunterladen [10]. Den laden Sie dann in die Arduino IDE, kompilieren ihn und laden ihn auf den Nano hoch. In C haben wir eine Klasse geschrieben, die sich um die Initialisierung und das Auslesen der Messwerte des Sensors kümmert. Im Hauptprogramm wird ein zweidimensionales Array für die Temperaturwerte angelegt und dann an die Klasse MLX90621 übergeben. Anschließend wird die Temperatur in einem Farbwert umgerechnet und auf dem Display angezeigt. Zuerst erfolgt eine kleine Darstellung, bei der jedem IR-Sensor ein Pixel zugeordnet wird. Weil diese Grafik sehr klein ist, zeigt das Programm zusätzlich eine siebenfach vergrößerte Grafik an. Die eigentliche Besonderheit ist die interpolierte Darstellung im dritten Feld. Die niedrige Auflösung wird dadurch visuell erheblich gesteigert und einfache Strukturen lassen sich erkennen.

Eine einfache Umsetzung von Messwerten in Farben lässt sich mit dem HSV-Farbmodell abbilden.

HSV, RGB, CMY(K)



Die meisten Computeranwender kennen das RGB-Farbmodell. In diesem wird eine Farbe durch Angabe der Helligkeit für die drei Grundfarben Rot, Grün und Blau festgelegt. Weiß ergibt sich, wenn alle drei farbigen Lichtquellen eines Bildschirmpixels maximal hell leuchten. Schwarz ist die Abwesenheit von Licht. Durch Mischen der drei Leuchtkörpern können die meisten Farben gebildet werden (additive Farbmischung).

Für den Druck wird hingegen das CMY(K)-Verfahren (subtraktive Farbmischung) mit den drei Grundfarben Cyan (Hellblau), Magenta (Rosa) und Gelb (Yellow) angewendet. Weiß wird durch das bedruckte Papier geliefert und Schwarz ergibt sich beim Mischen der drei Farben zu je 100 Prozent. Weil dabei in der PraCxis kein tiefes Schwarz entsteht, sondern eher ein dunkles Graubraun, wird im Vierfarbdruck noch zusätzlich die Farbe Schwarz (Key) beigelegt.

Das HSV-Farbmodell orientiert sich gegenüber RGB und CMYK mehr an der menschlichen Farbwahrnehmung. Die Farbe wird mit Hilfe des Farbwerts (Hue), der Farbsättigung (Saturation) und des Hellwerts (Value) bestimmt. Der Farbwert kann als Gradzahl zwischen 0° (rot) und 360° auf einem Farbkreis liegen. Bei 100 Prozent Sättigung und 100 Prozent Helligkeit ergibt sich dann der gezeigte Farbverlauf für die Farbwerte.

Hohe Temperaturen werden mit kleinen Gradzahlen für den Farbwert abgebildet und niedrige Temperaturen mit Farben, die einen hohen Farbwert haben. Weil der Bereich ab einem Farbwert von etwa 300° wieder röthlich wird, beschränken wir uns auf die Farbwerte von 0 bis 300 Grad (HUEMAX im Code). Mit der folgenden Anweisung wird für eine Temperatur der zugehörige Farbwert bestimmt, wobei die Skalierung berücksichtigt wird, die am Farverlaufsbalken am rechten Bildschirmrand abzulesen ist. MINTEMP und MAXTEMP bestimmen im Code die Temperaturgrenze.

```
hue = HUEMAX - (temps[x][y] + abs(MINTEMP)) * (HUEMAX / (float)(MAXTEMP + abs(MINTEMP)))
```

Übersteigt die anzulegende Temperatur die maximale Temperatur, wird aus dem HUE-Farbkreis der Farbwert benutzt, der sich bei einem weiteren Kreisumlauf ergibt. Das führt dazu, dass überhöhte Temperaturen violett erscheinen.

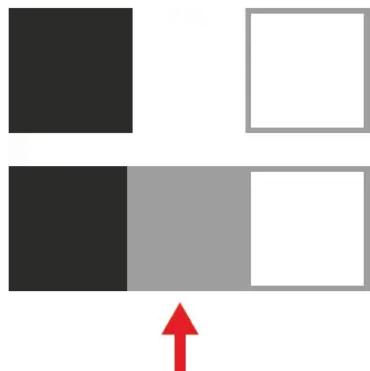
Interpolation

Vergrößert man ein digitales Bild, dann wachsen die Pixel irgendwann zu deutlich sichtbaren Quadraten heran. Zwischen den Pixeln ist eben keine Bildinformation vorhanden. Bei einem selbst programmierten Zoom-Algorithmus hat man die Wahl, die Pixel einfach anschwellen zu lassen oder Zwischenwerte zu berechnen. Zwischen ein schwarzes Pixel und ein benachbartes weißes Pixel würde man dann ein graues Pixel schieben. So wird das Bild zwar unscharf, aber nicht verpixelt. Die Unschärfe stört im Bild der Wärmebildkamera nicht. Sie hilft sogar, das Bild besser interpretieren zu können, da die harten Kanten der klobigen Pixel weniger der Realität entsprechen als die weichen Übergänge des berechneten Bilds.



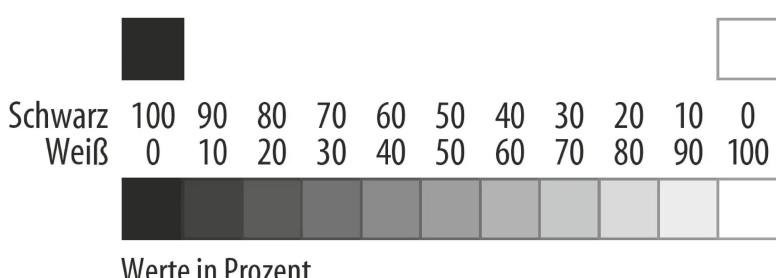
Wird das verlustbehaftete aufgenommene Foto nachträglich unter Anwendung einer bilinearen Interpolation vergrößert, wirkt das Ergebnis für das Auge gefälliger. Die verloren gegangenen Informationen/Bilddaten können aber nicht rekonstruiert werden.

Das Verfahren zur Berechnung von Zwischenwerten wird in der Mathematik als Interpolation bezeichnet. Es gibt verschiedene Interpolationsverfahren. Das einfachste ist die lineare Interpolation. Sie heißt linear, weil zwischen zwei Werten quasi linear übergeblendet wird. Wenn im einfachsten Fall nur ein einzelner Wert zwischen zwei gegebenen Werten berechnet werden soll, bedeutet das, dass der Durchschnitt gebildet wird.



Interpolation von zwei Pixeln: der ermittelte Grauwert entspricht dem Durchschnitt von Weiß und Schwarz..

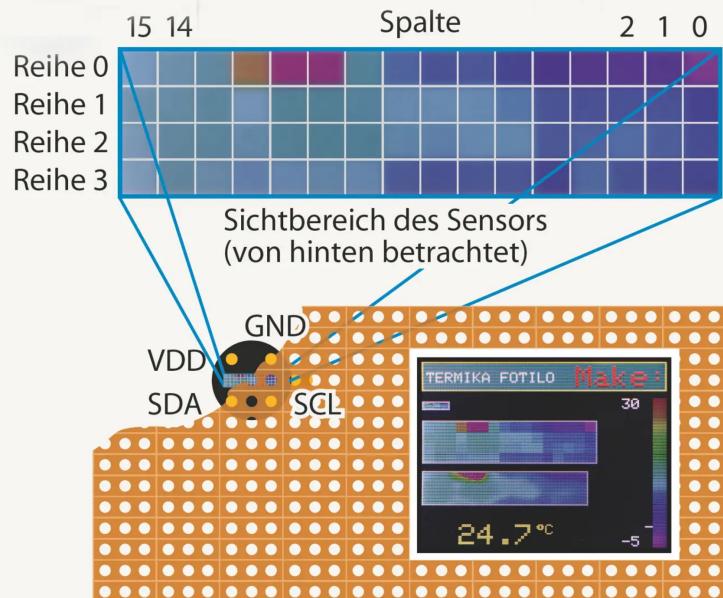
Sollen zwischen zwei Werten y_1 und y_2 mehrere Zwischenwerte y_x mit $1 < x < 2$ interpoliert werden, dann entspricht $y_{1,5}$ wieder genau dem Durchschnitt der beiden Werte. Alle Werte werden nämlich anteilig je nach ihrer Nähe zu y_1 beziehungsweise y_2 summiert. Das entspricht der Formel $y_x = y_1 * (2 - x) + y_2 * (x - 1)$. Der Wert y_1 geht mit dem Anteil $2 - x$ in die Summe ein. Für $x = 1,1$ sind das $2 - 1,1 = 0,9$, beziehungsweise 90 Prozent. Der Wert y_2 geht mit dem Anteil $x - 1$ in die Summe ein. Für $x = 1,1$ sind das $1,1 - 1 = 0,1$ beziehungsweise 10 Prozent. Der Wert $y_{1,1}$ ist eben noch sehr nahe an y_1 . Für den Sonderfall des Durchschnitts gilt, dass $y_{1,5}$ genauso nahe an y_1 wie an y_2 liegt.



Zwischen zwei Pixeln werden mehrere Zwischenwerte berechnet. Die gegebenen Pixel gehen anteilig je nach Entfernung zum Zwischenwert in die Berechnung ein.

Um die Formeln in ein passendes Code-Fragment zu übertragen, müssen die Wertebereiche noch angepasst werden. Bisher lag y_x zwischen y_1 und y_2 . Bei Bildpixeln gibt es aber nur ganzzahlige Positionen. Auch beim Zoom-Faktor kann man es sich einfach machen, indem man nur ganze Zahlen zulässt. Der Zoom-Faktor entspricht in unserer Berechnung der Anzahl der zu berechnenden Zwischenwerte.

Anordnung von Display und IR-Sensor



Weil der Sensor nach vorne (vom Benutzer weg) schaut, während das Display nach hinten zeigt, müssen die Pixel jeder Zeile bei der Displaydarstellung horizontal gespiegelt werden. Der erste Messwert der ersten Spalte (0), ersten Reihe (0) aus dem EEPROM des Sensors muss in der 16. Spalte (rechts) auf dem Display gezeigt werden.

In der für ganzzahlige Positionen angepassten Formel für die lineare Interpolation der Temperaturwerte einer einzelnen Zeile kommen folgende Variablen vor:

ZOOM Zoomfaktor (entspricht der Anzahl der zu berechnenden Zwischenwerte)

x1 Spaltenposition 1 in der interpolierten Tabelle der Temperaturwerte

x2 Spaltenposition 2 in der interpolierten Tabelle der Temperaturwerte

x Spaltenposition des zu berechnenden Zwischenwerts

t1 Temperatur an der Spaltenposition x1

t2 Temperatur an der Spaltenposition x2

t interpolierte Temperatur an der Spaltenposition x

Angepasste Formel:

```
int8_t MINTEMP = -5;
uint8_t MAXTEMP = 30;
uint16_t HUEMAX = 320;      // Schrittweite fuer Farbstrahl. Eigentlich 360,
aber wir wollen die lila Farben etwas vermeiden, weil Ã¤hnlich wie rot
uint8_t ZOOM = 7;           // Vergroesserungsfaktor

MLX90621 MLXtemp;          // Objekt fuer Tempsensor erzeugen

float temps[16][4];
while (!MLXtemp.init())        // MLX90620 init failed
    delay (100);
MLXtemp.read_all_irfield (temps);
// Reihe umkehren
for (y = 0; y < 4; y++)
{
    for (x = 0; x < 8; x++)
    {
        i = temps[15 - x][y];           // Wert von rechts retten
        temps[15 - x][y] = temps[x][y];
        temps[x][y] = i;
    }
}
// Ausgabe des Temperaturfeldes auf Display
for (y = 0; y < 4; y++)
{
    for (x = 0; x < 16; x++)
```

```

{
    hue = HUEMAX - (temps[x][y] + abs(MINTEMP)) * (HUEMAX / (float)(MAXTEMP
+ abs(MINTEMP)));
    HSVtoRGB (R, G, B, hue, 1, .5);
    TFTscreen.stroke (B * 255, G * 255, R * 255);
    TFTscreen.fill (B * 255, G * 255, R * 255);
    TFTscreen.point (x + 1, y + 28); // 1:1
Datenfeld
    TFTscreen.rect (x * ZOOM + 1, y * ZOOM + 41, ZOOM, ZOOM); // Daten
xZOOM
}
}

```

Die Spaltenposition spielt in der Berechnung eine wichtige Rolle. In dem Array, in dem die Temperaturwerte abgespeichert sind, wird die Spalte von Position 0 bis 15 angegeben, was 16 Pixeln entspricht. Wenn Zwischenwerte gebildet werden, dann gibt es mehr als 16 Spalten, in unserem gezoomten Bild sind es $16 \times 7 = 112$, da der Zoom-Faktor auf den Wert 7 festgelegt ist. Das Pixel an der Position 0 aus dem Temperatur-Array bleibt auch bei dem interpolierten und gezoomten Bild an Position 0. Das Pixel an der Position 1 aus dem Temperatur-Array rutscht jedoch im interpolierten oder gezoomten Bild an Position 7. Das entspricht einer Multiplikation der Position mit dem Zoom-Faktor. Die Spaltenpositionen x_1 , x_2 und x in der Formel beziehen sich auf das interpolierte Bild. Die Formel beziehungsweise der Pseudocode lautet:

```
t = t1 * (x2 - x)/ZOOM + t2 * (x - x1)/ZOOM
```

An einem Beispiel wird deutlich, wie die Formel funktioniert. Angenommen, wir haben zwei benachbarte Temperaturwerte t_1 und t_2 mit der Spaltenposition $x_1=0$. Bei einem Zoomfaktor von 7 ($ZOOM=7$) entspricht dann die Spaltenposition $x_2=7$. Wenn $x=2$ ist, dann geht t_1 mit einem Anteil von $(x_2-x)/ZOOM=(7-2)/7=5/7$ in das neu berechnete Pixel an der Position x ein. t_2 geht mit einem Anteil von $(x-x_1)/ZOOM=(2-0)/7=2/7$ ein.

Im Programmcode für die Interpolation sieht das Ganze dann so aus:

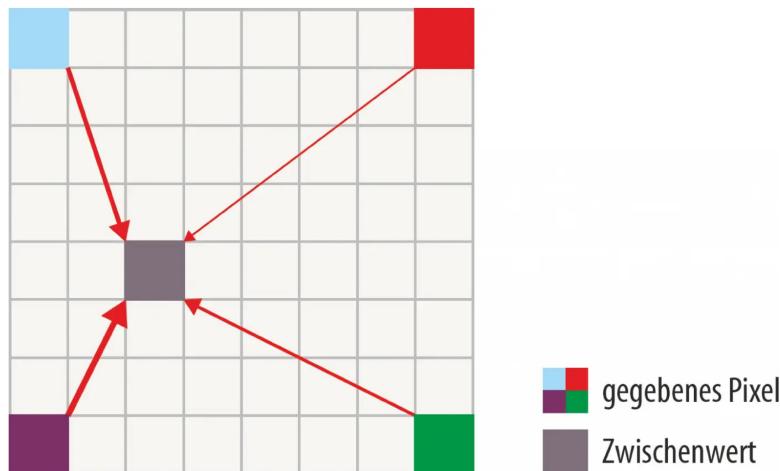
```

float LinInterpol (float t1, float t2, uint8_t x1, uint8_t x2, uint8_t x)
{
    return t1 * (x2 - x) / ZOOM + t2 * (x - x1) / ZOOM;
}

```

Für die Berechnung eines kompletten Bildes statt einer Pixelzeile können wir die gleichen Formeln verwenden. Man kann jedoch nicht einfach Pixel für Pixel interpolieren, sondern berechnet zuerst die Spalten und dann die Zeilen (oder umgekehrt). So reduziert man das Interpolationsproblem immer

auf je zwei Werte, denn bei einem zweidimensionalen Bild liegen die Interpolationswerte zwischen vier Pixeln, die alle in die Berechnung einfließen müssen. Eine einzige Formel für die bilineare Interpolation, die alle vier Werte in einem Schritt mit einberechnet, wäre unnötig kompliziert.



Interpolation von Werten auf einer Ebene: Für die Interpolation eines Pixels zwischen vier anderen Werten müssen alle vier gegebenen Pixel in die Berechnung einfließen.

Die interpolierten Zeilen können im Speicher abgelegt werden, um Rechenzeit zu sparen. Da aber im verwendeten ATmega328 wenig RAM-Speicher zur Verfügung steht, werden die Zeilenwerte im Programm bei jeder Berechnung eines Pixels neu ermittelt, um die Werte zwischen zwei Zeilen berechnen zu können. So wird auf Kosten der Rechenzeit Speicher gespart. Wir berechnen die Interpolation zwar wie gezeigt, können aber nicht auf zuvor berechnete Werte zurückgreifen, sondern müssen für jeden Pixel immer alle Nachbarn erneut ermitteln.

Im Programmcode wird die Interpolation des Wärmebildes wie folgt berechnet:

```

01  for(x = 0; x < (ZOOM * 15); x++)
02  {
03      for(y = 0; y < (ZOOM * 3); y++)
04      {
05          xmod = x % ZOOM;
06          xorg1 = x - xmod;
07          xorg2 = xorg1 + ZOOM;
08          ymod = y % ZOOM;

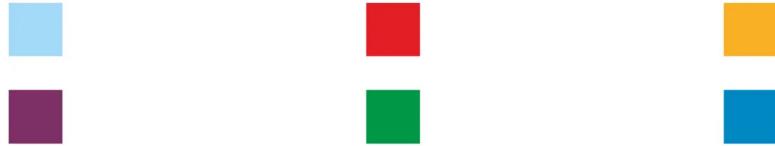
```

```

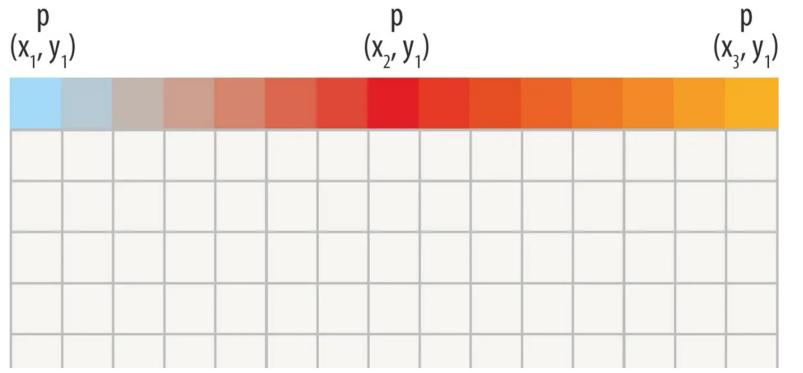
09      yorg1 = y - ymod;
10      yorg2 = yorg1 + ZOOM;
11      i1 = LinInterpol (temps[x / ZOOM][y / ZOOM], temps[x / ZOOM + 1]
[y / ZOOM], xorg1 , xorg2, x); // horizontale Interpol. Daten Zeile n
12      i2 = LinInterpol (temps[x / ZOOM][y / ZOOM + 1], temps[x / ZOOM
+ 1][y / ZOOM + 1], xorg1 , xorg2, x); // horizontale Interpol. Daten Zeile
n+1
13      interpoltemp = LinInterpol (i1, i2, yorg1, yorg2, y); //
vertikale Interpolation
14  }
15 }
```

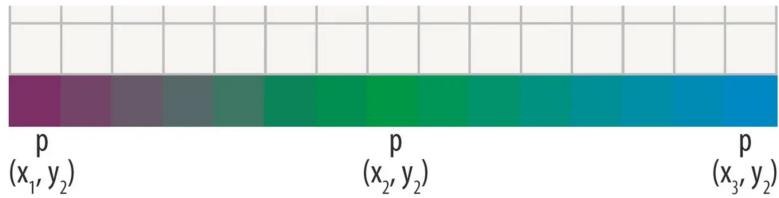
Die for-Schleifen sorgen dafür, dass das interpolierte Wärmebild Spalte für Spalte und Zeile für Zeile aufgebaut wird. Die Variablen x und y geben die Koordinaten im interpolierten Bild an. Das originale kleine Bild ist 16 Pixel breit, deswegen muss die Zeilenbreite mit dem Zoom-Faktor abzüglich 1 multipliziert werden, um die Breite des interpolierten Bildes zu erhalten. Es wird also mit 15 anstatt mit 16 multipliziert. Das liegt daran, weil nicht über den letzten vorhandenen Temperaturwert hinaus interpoliert werden kann.

1. Nicht interpoliert

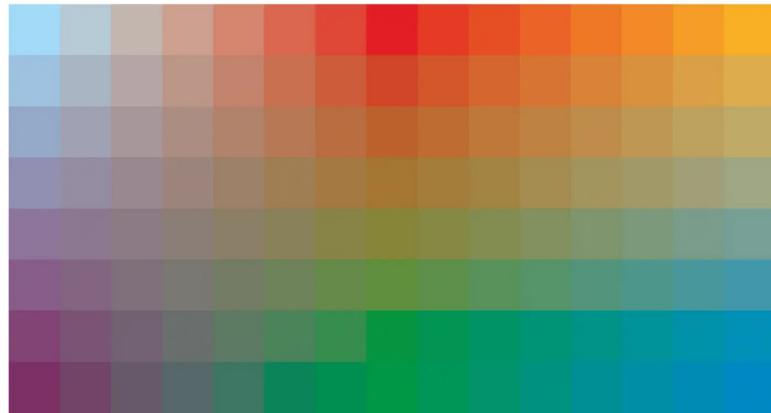


2. Zeilen interpoliert



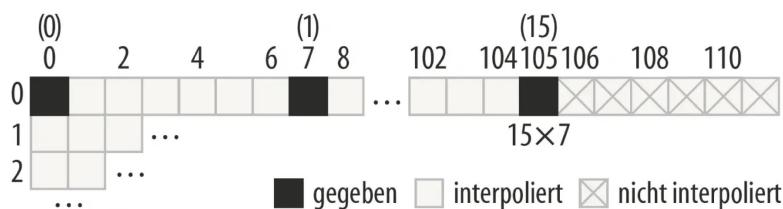


3. Spalten interpoliert



Bilineare Interpolation in zwei Schritten: Zuerst werden die Zeilen interpoliert, dann die Spalten mit Hilfe der neu berechneten Zwischenwerte in den Zeilen.

Das gezoomte Bild ist also breiter und höher als das interpolierte. Beim einfachen Zoom mit harten Pixelkanten wird einfach jedes Pixel mehrfach kopiert, auch das letzte 16. Pixel. Im Temperaturwert-Array entspricht das der Position 15. Auch dieses Pixel wird mehrfach auf dem Display kopiert und angezeigt. Beim interpolierten Bild ist dort jedoch Schluss, denn danach gibt es keine Werte mehr zum interpolieren.



Das interpolierte Bild ist kleiner als das gezoomte.

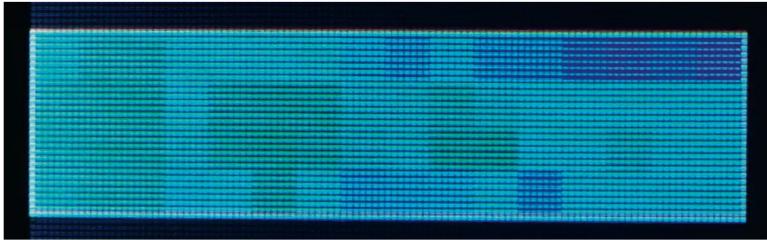
Innerhalb der Schleifen erfolgt die eigentliche Interpolation. Dort stehen die aktuell zu berechnenden Koordinaten für ein Pixel, also `x` (von 0 bis 105) und `y` (von 0 bis 21) zur Verfügung. Für jedes Pixel werden außerdem die vier Koordinaten der gegebenen Temperaturwerte benötigt, die in die Berechnung mit eingehen. Diese werden in den Zeilen 5 bis 10 berechnet und mit `xorg1`, `xorg2`, `yorg1`, `yorg2` angegeben. Es handelt sich dabei um die Werte 0, 7, 14 und so weiter. Von einer beliebigen Stelle `x` kann dieser Wert wie folgt berechnet werden: Zuerst berechnet man den Rest der Division von `x` durch 7 (`xmod = x % ZOOM`). Das Ergebnis `xmod` entspricht dem Abstand der Stelle `x` zur Stelle `xorg1`. Demzufolge ergibt sich `xorg1 = x - xmod`. Der nächste gegebene Temperaturwert ist dann 7 (Zoom-Faktor) Schritte weiter `xorg2 = xorg1 + ZOOM`. Entsprechend werden die `y`-Werte berechnet.

Nun wird ein Pixel `i1` zwischen zwei gegebenen Zeilenwerten interpoliert. Dazu werden in die Interpolationsfunktion zwei benachbarte Werte aus dem Temperatur-Array eingesetzt:

```
temps[x / ZOOM][y / ZOOM]
temps[x / ZOOM + 1][y / ZOOM]
```

Zusätzlich werden die dazugehörigen `x`-Koordinaten `xorg1` und `xorg2`, und außerdem die aktuell zu berechnende `x`-Koordinate `x` benötigt. Dann wird ein Pixel `i2` zwischen zwei gegebenen Zeilenwerten auf die gleiche Weise interpoliert, jedoch eine Zeile tiefer. Deshalb wird für das Temperatur-Array die `y`-Koordinate jeweils durch `y / ZOOM + 1` eingesetzt. Im letzten Schritt wird dann aus den beiden interpolierten Zeilenwerten `i1` und `i2` der Spaltenwert `interpoltemp` berechnet.

Der Code berechnet jedes Pixel aus dem Temperatur-Array noch einmal neu, also auch die Werte, die eigentlich nicht interpoliert werden müssen. Man würde zwar Rechenzeit sparen, wenn man die Berechnung in diesen Fällen nicht ausführt, aber das Programm wäre dann wesentlich unübersichtlicher, weil man diesen Fall gesondert abfangen müsste.



Trotz einheitlicher Oberflächentemperatur werden unterschiedliche Temperaturen gemessen.

Erweiterungspotential

Die Wärmebildkamera kann noch deutlich ausgebaut werden. Auf dem Displaymodul sitzt ein SD-Kartenslot auf der Rückseite. Wir haben ihn sogar abgelötet, um den Aufbau kompakt zu halten. Die Arduino-Library fürs Display bietet schon die notwendigen Routinen für den Zugriff, sodass es nicht aufwendig ist, die Thermografiebilder auch noch zu speichern, wenn man den Slot behält. Wir haben uns in unserem Beispielcode auch auf einen Temperaturbereich von –5 °C bis 30 °C beschränkt, weil dann die Farben unter normalen Raumbedingungen schick aussehen. Soll die Kamera stark unterschiedliche Temperaturbereiche abbilden, wäre eine adaptive Anpassung der Skala praktisch. Dazu könnte der Mittelwert der letzten Tiefst- und Höchstwerte ermittelt werden. Eine Kalibrierung nach dem Einschalten könnte für noch bessere Ergebnisse sorgen. Wie sich zeigte, unterliegt jeder Sensor einer gewissen Streuung bei den Messwerten und liefert selbst bei einer homogenen Wärmequelle kein gleichmäßiges Bild. Mit einem Taster könnte man auch das aktuelle Bild einfrieren, um es in Ruhe zu betrachten. (pek [11])

URL dieses Artikels:

<https://www.heise.de/-7280578>

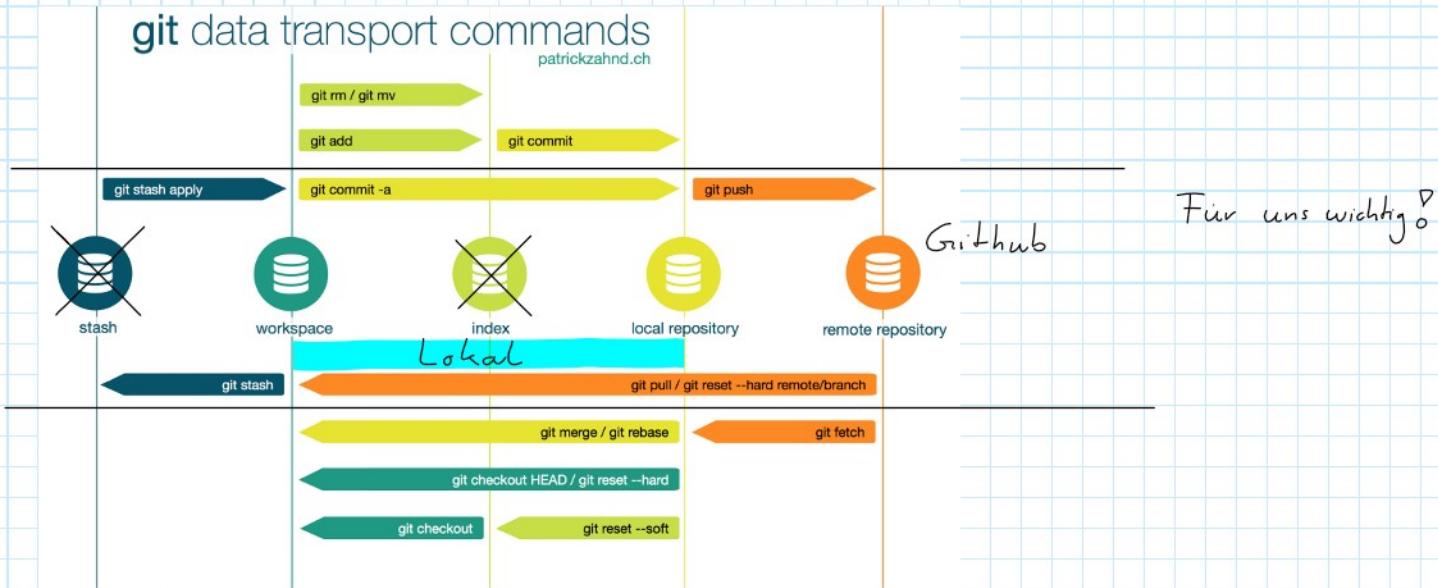
Links in diesem Artikel:

- [1] <https://www.heise.de/ratgeber/Termika-Fotilo-Bauanleitung-fuer-eine-einfache-portable-Waermebildkamera-7280578.html>
- [2] <https://www.heise.de/ratgeber/Bastelanleitung-Roboter-Muelltonne-Tonni-mit-einem-Arduino-bauen-6525842.html>
- [3] <https://www.heise.de/ratgeber/Anleitung-Wie-Sie-mit-dem-Arduino-smarte-Treppenbeleuchtung-einrichten-7081394.html>
- [4] <https://www.heise.de/ratgeber/Anleitung-Romme-Buzzer-mit-Arduino-bauen-7067882.html>
- [5] <https://www.heise.de/ratgeber/Smart-Garden-Rasenkabelfinder-mit-Arduino-bauen-7070891.html>
- [6] <https://www.heise.de/ratgeber/Anleitung-Wie-Sie-mit-einem-Roboter-Ihre-Sitzhaltung-verbessern-6289555.html>
- [7] https://www.heise.de/select/make/2017/3/softlinks/xgk8?wt_mc=pred.red.make032017.016.softlink.softlink
- [8] <https://github.com/maxritter/DIY-Thermocam>
- [9] <https://www.heise.de/make/>
- [10] <https://github.com/MakeMagazinDE/TermikaFotilo>
- [11] <mailto:pek@ct.de>

Copyright © 2022 Heise Medien

Die git-Versionsverwaltung

Donnerstag, 8. Dezember 2022 06:38



- 1.) Bei GitHub registrieren
- 2.) In ein Verzeichnis wechseln in dem Sie alle ihre git-repositories unterbringen möchten (!!! Dies sollte kein ownCloud, googleDrive, etc Verzeichnis sein !!!)
- 3.) In diesem Verzeichnis "git bash here" öffnen.
- 4.) in die git bash die Zeile:
`git clone https://github.com/BerndDonner/TermikaFotilo.git`
Eingeben
- 5.) Änderungen am code vornehmen z.B. --- Programm besser dokumentieren
- 6.) Die Änderungen ins lokale repository committen mit:
`git commit -am "Programm besser dokumentiert."`
- 7.) mit dem Kommando
`git push`
Änderungen nach GitHub hochladen (Benutzer muss gesetzt werden, GitHub Account notwendig, wird aber alles als Hilfe ausgegeben...)
- 8.) Bei dem nächsten Änderungen nicht mit 4.) sondern mit
`git pull`
beginnen

---> um mit dem donner branch zu arbeiten:
`git checkout donner`

Exkurs: Bilineare Interpolation

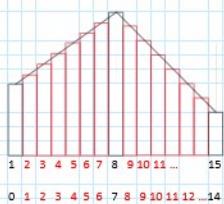
Tuesday, 13 December 2022 17:49

Zur Bilinearen Interpolation:

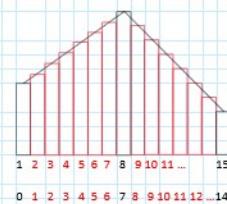
Bei uns: zur ansehnlichen Vergrößerung der Bilder um den Faktor 7 eingesetzt

Zur linearen Interpolation (schwarz - weiß Bild)

Zoomfaktor: 7



Wenn etwas linear ist, so kann man jeden Eintrag als a^* linker Pixelwert + b^* rechter Pixelwert darstellen.



0 1 2 3 4 5 6 7 8 9 10 11 12 14

0 1 2 3 4 5 6 7 8 9 10 11 12 14

Ganz analog geht das auch in 2D:



Komplexe Formel für dieses Pixel:

$$\frac{56}{49} \cdot uL + \frac{59}{49} \cdot vR + \frac{26}{49} \cdot uR + \frac{29}{49} \cdot vL$$

Dies sind die Gewichte für das Pixel oben-rechts (or)

- Wie sehen die Gewichte für das Pixel or in diesem Bereich aus?

Research Article

A Study of Digital Image Enlargement and Enhancement

Hsueh-Yi Lin, Chi-Yuan Lin, Cheng-Jian Lin, Sheng-Chih Yang, and Cheng-Yi Yu

Department of Computer Science and Information Engineering, National Chin-Yi University of Technology,
No. 57, Section 2, Zhongshan Road, Taiping District, Taichung 41270, Taiwan

Correspondence should be addressed to Cheng-Yi Yu, yuyj@ccut.edu.tw

Received: 21 February 2014; Accepted: 13 April 2014; Published: 28 April 2014

Academic Editor: Hsueh-Yi Lin

Copyright © 2014 Hsueh-Yi Lin et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Most image enlargement techniques suffer the problem of zigzagged edges and jagged images following enlargement. Blurriness occurs to the edges of objects if the edges in the image are sharp, the visual is considered to be high quality. To solve this problem, this paper presents a new and effective method for image enlargement and enhancement based on adaptive inverse hyperbolic tangent (AIHT) algorithm. Conventional image enlargement and enhancement methods enlarge the image using interpolation and subsequently enhance the image without considering image features. However, this study presents the method based on Adaptive Inverse Hyperbolic Tangent algorithm to enhance images according to image features before enlarging the image. Experimental results indicate that the proposed algorithm is capable of adaptively enhancing the image and extruding object details, thereby improving enlargements by smoothing the edge of the objects in the image.

1. Introduction

The digital images are affordable and easy to preserve, transmit, and modify; they are widely used across different fields. Since digital image sampling is a low-quality and incomplete, digital image resolution is often limited. Therefore, it is necessary to employ image enhancement technologies when viewing portions of an image in details.

The quality of an image is commonly determined by factors in the natural environment. Factors present in the natural environment usually relate to light. If light distribution is extreme, the target object in the picture becomes hard to identify. This study proposes an image enlargement method that combines image enhancement with image enlargement. The proposed image enhancement process employs AIHT algorithm to enhance the image and smooth its edges. The enlargement process uses a bilinear enlargement algorithm to enlarge the enhanced image. This method adaptively enhances images while simultaneously extracting details in the target image, in order to smooth the edges created by enlargement. Image quality is often reduced when images are enlarged. The proposed method can improve image quality following enlargement. The most significant advantage of this method is the ability to reduce rough edges and image distortion even after image enlargement.

This paper is structured as follows: Section 2 discusses related image enlargement technology algorithms; Section 3 introduces the AIHT algorithm; Section 4 describes the proposed algorithm; Section 5 presents the test and simulation results; the conclusion provides a closing discussion of the proposed algorithm and suggests future research directions and applications.

2. Review of Image Enlargement Algorithms

Images are enlarged to enhance image resolution, increase image quality, and improve identification. The goal of this approach is to maintain image quality while eliminating image distortion, such as blurring and rough edges, upon image enlargement.

Traditional interpolation methods are commonly employed in image enlargement due to their simplicity and efficiency. Interpolation generally comprises two methods: (1) nearest neighbor interpolation [1] and (2) bilinear interpolation [2]. When the continuous function passes through, the interpolation function can be used to calculate the sampling points. In theory, higher order interpolation functions are similar to continuous functions. However, this is not the case in practice [3].

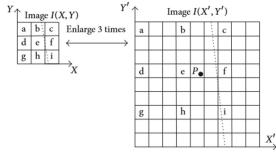


FIGURE 1: Nearest-neighbor interpolation method.

2.1. Nearest Neighbor Interpolation. Nearest-neighbor interpolation, otherwise known as the substitution method, uses the gray value of neighboring pixels to interpolate the gray value of new pixel points [4, 5]. This method's approach is to find the neighboring integer pixel points nearest to the noninteger pixel points. The gray value of these integer pixel points is then used to interpolate the gray value of new pixel points (shown in Figure 1).

In Figure 1 we can see the enlarged image (x', y') of the pixel P ; then, with conversion back to the original image $I(x, y)$, P is interpolated between the pixel e and f . Nearest-neighbor interpolation algorithm is to calculate the P point in the image $I(x, y)$ and its surrounding pixels e, f, h , and i the distance and then choose the shortest distance between the gray values of the pixels, as their gray values. The enlargement process using the nearest-neighbor interpolation algorithm will be encounter with the problem of blocking the effect.

The nearest neighbor interpolation function is the simplest and most efficient interpolation algorithm. However, since it is easier to calculate its results in lower image quality, enlarged images usually display jagged and blocked features. The mathematical function is

$$h_t(t) = \begin{cases} 1, & -\frac{1}{2} < t < \frac{1}{2} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

2.2. Bilinear Interpolation. In mathematics, bilinear interpolation is an extension of linear interpolation for interpolating functions of two variables on a regular 2D grid image. The key idea is to perform linear interpolation first in one direction and then again in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location.

In computer vision and image processing, bilinear interpolation is one of the basic resampling techniques. When an image needs to be scaled up, each pixel of the original image needs to be moved in a certain direction based on the scale constant. However, when scaling up an image by a nonintegral scale factor, there are pixels that are not assigned appropriate pixel values. In this case, those holes should be assigned appropriate RGB or grayscale values so that the output image does not have nonvalued pixels.

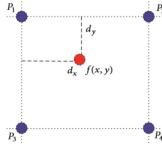


FIGURE 2: Bilinear interpolation method.

Bilinear interpolation can be used where perfect image transformation with pixel matching is impossible, so that one can calculate and assign appropriate intensity values to pixels. Unlike other interpolation techniques such as nearest neighbor interpolation, bilinear interpolation uses only the 4 nearest pixel values which are located in diagonal directions from a given pixel in order to find the appropriate color intensity values of that pixel.

Bilinear interpolation considers the closest 2×2 neighborhood of known pixel values surrounding the unknown pixel's computed location. It then takes a weighted average of these 4 pixels to arrive at its final, interpolated value. The weight on each of the 4 pixel values is based on the computed pixel's distance from each of the known points (shown in Figure 2).

Similar to nearest-neighbor interpolation, this method calculates new values based on four neighboring integer pixels [6, 7]. Interpolation is calculated as follows:

$$f(x, y) = (1 - d_x)(1 - d_y)p_1 + d_x(1 - d_y)p_2 + (1 - d_x)d_y p_3 + d_xd_y p_4 \quad (2)$$

The $f(x, y)$ refers to the pixels of the new position point after image enlargement and p_1, p_2, p_3 , and p_4 indicate the four vertices of the interpolation pixel P . The closer these vertices are to $f(x, y)$, the greater their contribution to $f(x, y)$ will be and vice versa. The neighboring pixels in images enlarged using bilinear interpolation are more continuous, or smoother, than when the integer points are acquired directly.

3. Adaptive Inverse Hyperbolic Tangent (AIHT) Algorithm

The world is filled with various images, which are representations of objects and scenes in the real world. Images are represented by a matrix of pixels, which can represent the gray levels or colors of the image. There are many aspects of images that are ambiguous and uncertain. Examples of these vague aspects include determining the border of a blurred object and determining which gray values of pixels are bright and which are dark [8]. If an image containing both objects and scenery gets too dark or blurred, it would be hardly recognized. Thus, the image enhancement technique

Howto: C-Programmierung: Arrays und Zeiger

Wednesday, 7 September 2022 09:12

Alle Variablen, mit denen wir bis jetzt gearbeitet haben, hat der Mikrocontroller im SRAM abgelegt

Wie ist der SRAM aufgebaut?

Jedes Register steht für ein Byte aus dem SRAM



Genauso aus dem AVR-Datensheet Kapitel 7.3 / Seite 18

Der ATmega328p hat 2kB = 2048 Bytes SRAM

Data Memory	
32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Registers	0x0060 - 0x0FFF
Internal SRAM (1048 x 8)	0x0100
	0x03FF
	0x0BFF

Beachte die Fehler in diesem Diagramm!*

Jedes Byte im SRAM des Mikrocontrollers hat eine eigene Adresse. Die Adressen von ATmega328p sind 16-bit groß.

Mit dem &-Operator kann man sich die Adresse einer Variablen beschaffen.

Ein Zeiger (eng. pointer) ist nichts anderes als die Adresse für einen bestimmten Datentyp.

→ Legen Sie nachanmerende folgende Variablen an:

- 1) long
- 2) byte
- 3) int

b: 0x03FB 22 99
c: 0x03FA 22 97
d: 0x03F9 22 95
e: 2 3
f: 6 7
g: 5 6 7 8

Lassen Sie sich die Adressen der 3 Variablen anzeigen.

Was sind die Adressen von Variablen übergeben? *

(1) Bei großen Funktionsaufrufen wird die Werte der Argumente kopiert werden "Call by value"

Für "große" Argumente (Strukturen, Arrays) ist dies unerheblich aber zulässig.

→ **Arrays:** Der Funktion wird nur die Adresse des Arguments (Struktur, Array) übergeben, es werden also nur 16-Bit übergeben ("Call by reference"). Dies hat aber sofort zur Folge, dass die Funktion nun (durch die Adresse) mit den original Daten arbeitet. Also gute Veränderung der referenzierten Variable ist sofort in der Hauptfunktion (long, main) sichtbar.

Warum? Weil man mit dem Original arbeiten nicht will oder kann.

(2) Arrays werden in C nur durch ihre Adresse repräsentiert.

siehe unten

(3) malloc und free (nicht in der Ausgabe Relevanz)

(4) Viele viele weitere ...

linkisch - links

funktion - pointer

... Das Gegengleich zum &-Operator ist der *-Operator:

Mit * p holt man über Inhalt der Speicherzelle und die Adresse p

Werte Bytes für den "Inhalt der Speicherzelle" benötigt werden geht aus einem Datentyp für *p herree.

→ Erstellt aus "call by reference" Funktion, die kein Zeichenwert vertragen

Inhalt einer Zelle

Variablen für das Zeichen

Adresse von text[0]	Adresse von text[0][0]	a	b	c	d	inhalt[0][0]
text[0]	text[0][0]	→ ad[0][0] ad[0][1]				inhalt[0][0]
Adresse von text[0][0]	text[0][0]	A	B	C	D	
text[0][0]	→ ad[0][0]	1	2	3	4	
Adresse von text[0][1]	text[0][1]	w	x	y	z	
text[0][1]	→ ad[0][1]	W	X	Y	Z	inhalt[0][1]
Adresse von text[0][2]	text[0][2]	→ ad[0][2]				inhalt[0][2]
text[0][2]						

8: liefere die Adresse einer Variablen

*: liefere den Inhalt einer Adresse

* ad[0] ist 'W'

In Wirklichkeit ist die Anordnung im SRAM eher folgendermaßen:

text[0][3] ist eine Abkürzung für *(text[0]+3)

und das ist eine Abkürzung für *(text[0]+0)+3)

text[0][3] = *(text[0]+3) = *(*(text+0)+3)

```
void setup() {
    // put your setup code here, to run once:
    Serial.begin(9600);
    long a; //4 bytes
    byte b; //1 byte
    int c = 668; //2 bytes 668 = 0x029C
    // byte spinon;
    // &spinon = 0xB8FC;
    byte *spinon;
    spinon = (byte *) &c + 1;
    // spinon = (byte *) 0xB8F9;
    //nur zum nachschauen von der Adresse von a: &a
    Serial.print("a: ");
    Serial.println(a);
    Serial.print("b: ");
    Serial.println(b);
    Serial.print("c: ");
    Serial.println(c);
    Serial.print("d: ");
    Serial.println(d);
    Serial.print("Spinon: ");
    Serial.println(*spinon, HEX);
}
void loop() {
    // put your main code here, to run repeatedly:
}
```

Adresse von text[0]	Adresse von text[1]	Adresse von text[2]	Adresse von text[3]	Adresse von text[4]	Adresse von text[5]	a	b	c	d	h	B	C	D	1	2	3	4	w	x	y	z	W	X	Y	Z
text	text + 1	text + 2	text + 3	text + 4	text + 5	*text + 0	*text + 1	*text + 2	*text + 3	*text + 4	*text + 5	*text + 6	*text + 7	*text + 8	*text + 9	*text + 10	*text + 11	*text + 12	*text + 13	*text + 14	*text + 15	*text + 16	*text + 17	*text + 18	*text + 19

Adressen, Zeiger, Arrays, Call by Reference und Zeigerarithmetik

Deklaration: 2 Dimension von Index 0...3

char text[5][4];
 ↑
 ↑

Datentyp 1 Dimension von Index 0...4

Übungen: Was wäre ein Serial.println gewollt ausgegeben?

- *text[0]
- *text[0] + 2
- *text[0] + 5
- &text[0]

Aus dem Datenblatt des Displays ST7735R ist folgendes zu finden:

ST7735R

9.8 Data Color Coding

9.8.1 8-bit Parallel Interface (IM2, IM1, IM0 = "100")

Different display data formats are available for three Colors depth supported by listed below.

- 4k colors, RGB 4,4,4-bit input.
- 65k colors, RGB 5,6,5-bit input.
- 262k colors, RGB 6,6,6-bit input.

Was macht folgende Funktion? Quelle: Adsfritz_GFxx.cpp Zeile 507

- Erstellen Sie ein Diagramm in dem erkennbar wird, welche Bits der Eingabedaten wie in den Ausgabe eingeht.
- Welche Bits der Eingabedaten werden nicht benutzt
- In welchen Zusammenhang steht das Datenblatt mit dieser Funktion?
- Welchen return-Wert liefert die Funktion bei $r = 0x100, g = 0x141, b = 0x0020$
- Erstellen Sie eine Funktion `uint16_t newColor666(uint8_t r, uint8_t g, uint8_t b)` die einen Farbwert im direkten Farbformat des ST7735R Displays zurückliefert.

```

void Adafruit_GFX::newColor(uint8_t r, uint8_t g, uint8_t b)
{
    return ((r & 0xF8) <> 8) | ((g & 0xFC) <> 3) | (b >> 3);
}

Was macht diese Funktion? Quelle: Adafruit_ST7735.cpp

inline uint16_t swapcolor(uint16_t x) {
    return (x << 11) | (x & 0x07E0) | (x >> 11);
}

```

In dem Wärmebildkamera Projekt ist in der innersten Schleife folgender Code zu finden:

```
HSVtoRGB (R, G, B, hue, 1, .5);  
TFTScreen.stroke (RGB (R * 255, G * 255, B * 255)); //-->Adafruit_GFX::newColor Adafruit_GFX.cpp Zeile 507  
TFTScreen.point (*x1*ZOOM+x+1, y1*ZOOM+yd + 75 + 1); //-->void Adafruit_ST7735S::drawPixel(int16_t x,int16_t y,uint16_t t_color) Adafruit_GFX.cpp Zeile 45
```

- Die Funktion `TFTscreen.stroke` ruft letztlich Adafruit_GFX::newColor auf. Verwendet wird die Farbe aber erst im letzten Funktionsaufruf `TFTscreen.point` die wiederum Adafruit_ST7735::drawPixel aufruft.

 - 1) Erläutern Sie welches Verbesserungspotential Sie bei der Handhabung der Farbe erkennen und wie Sie die Implementation abändern würden
 - 2) Messen Sie die Laufzeit der bestehenden Implementierung und speichern Sie die Ergebnisse
 - 3) (**Das ist eine echte Herausforderung**) Verbessern Sie den Umgang mit den Farben.
 - 4) Messen Sie die Laufzeit den verbesserten Codes und vergleichen Sie die Ergebnisse.

In unserem Wärmebildkameraprojekt ist folgender Code zu finden

```
uint8_t irpixels[128];  
  
int16_t vir = (int16_t)(irpixels[1] << 8 | irpixels[0]);
```

- Bauen Sie Ausgaben ein, so dass die übermittelten Zeichen übersichtlich und hexadezimal auf der seriellen Console ausgegeben werden.
 - Messen Sie nun die Laufzeit der Zeile `int16_t w = (int16_t)(irpixels[i] <> 8) * irpixels[i]);`
 - Speichern Sie dieses Programm unter dem Namen `uebung_8` zur Abgabe ab und notieren Sie hier die Anzahl der Taktzyklen, die diese Codezeile benötigt.
 - In folgenden erstellen Sie das Programm `uebung_new`. Bitte schaute Sie darauf, dass Sie nicht versehentlich das bereits fertige Programm `uebung_8` überschreiben.
Erstellen Sie ein Programm mit identischer Funktionalität, dass dieselbe Aufgabe möglichst schnell mithilfe von `int16_t` Zeiger bewältigt. Messen Sie die Laufzeit der neuen Implementation, so dass sie einen Vergleich mit beiden möglichen Implementierungen vornehmen können. Welche Version ist schneller? Um die Ursachen für die Geschwindigkeitsunterschiede eindeutig festzustellen, müsten man den generierten Assembler Code betrachten. Dies ist leider unter der Arduino Entwicklungsumgebung unnötig kompliziert. Trotzdem versuchen Sie mögliche Ursachen für die Geschwindigkeitsunterschiede zu erörtern.

git clone <https://github.com/BerndDonner/Microcontrollertechnik.git>

zu 4) alte Version: mindstens zwei Bit-operatoren (\ll , \gg) \Rightarrow mindstens 2 Takte
neue Version: Zeiger-cast wird weggemischt

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  
  uint8_t irpixels[2] = {0x12, 0x34};
  int16_t* spixel = (int16_t*) &irpixels[0];
  
  int16_t* spixel = (int16_t*) &irpixels[0];
}
```

```
Serial.println("Wert der Variablen");
Serial.print("Variante Bitoperationen: ");
Serial.println(int16_t vir, HEX);
Serial.print("Variante Zeiger: ");
Serial.println(*spion, HEX);
}
```

```
void loop() {
    // put your main code here, to run repeatedly:
}
```

Versucht man die HSVtoRGB Funktion unseres Wärmebildkameraprojekts zu optimieren kann man folgende Codezeilen kommen:

```
    uint8_t f;  
    uint8_t q = (0x00ff * (0x0100 - f)) >> 9;
```

- Wieso liefert diese Codezeile für $f = 129$ nicht den Wert 63?!
 - Korrigieren sie die Codezeile so, dass hier der gewünschte Wert von 63 ausgegeben wird.

```
float ai = (acommon + deltaai * pow(2, aiscale)) / pow(2, ((configreg >> 4) & 0x03)); // Bit 5:4 Config Register  
bi = bi / (pow(2, biscale) * pow(2, 3 - ((configreg >> 4) & 0x03)));
```

Schreiben Sie die obigen Zeilen als shift-Operationen um

```
*****  
uint16_t rgbcConvert24to16(uint32_t rbg24)  
{  
    uint16_t r = (rbg24 & 0xFF);  
    uint16_t g = (rbg24 & 0xFF) >> 8;  
    uint16_t b = (rbg24 & 0xFF) >> 16;  
    uint16_t t RGB565 = 0;  
  
    t = ((r << 11) | (g << 5) | b);  
    return t;  
}
```

```
g = (g * 253 + 505) >> 10;  
b = (b * 249 + 1014) >> 11;  
  
RGB565 |= (r << 11);  
RGB565 |= (g << 5);
```

```
return RGB565;
```

```
}
```

Was müssen wir wissen:

- 1.) Arduino: setup, loop, Serial.begin(), Serial.print, Bedienung Arduino-Oberfläche, Leistungsdaten des ATmega328p
- 2.) Skopes: {}, lokale, globale Variablen (Micheal Kipp, Howto C-Programmierung, Arrays)
- 3.) Ganzzahltypen: int8_t, int16_t, int32_t, int64_t und unsigned Varianten, Zweierkomplement, Type-cast
- 4.) Bitoperationen und deren Bedeutung: ^, &, |, ^, >>, <<
- 5.) Möglichkeiten der Zeitmessung: eigene Routine, millis, micros, wann nimmt man was?, Compiler Optimierungen, volatile
- 6.) Arrays und Zeiger: Wie liegen Daten im Speicher, Adressoperator &, Zeigeroperator *, call-by reference, spion mit Zeiger-Type-cast

Aufgabe zu Zweierkomplement und überlauf

Aufgabe zu Bitoperationen

Aufgabe zu Arrays (steht im groben)

Aufgabe Zeiger und Zeitmessung (fertig)

Geben Sie an, in welcher Datenstruktur Sie den kompletten Inhalt des TFT Displays (160 horizontal und 128 vertikal) bei horizontaler Ausrichtung speichern könnten. Sie dürfen davon ausgehen, dass die linke obere Ecke die Indexposition (0, 0) trägt. Diese Datenstruktur bezeichnete man übrigens als "framebuffer".

Welches Problem ergibt sich bei der Erstellung dieses Framebuffers auf einem ATmega328p?

In einem neuen Modus, hat nun die linke untere Ecke die Indexposition (0, 0). Erstellen Sie ein kleines Programm, dass den alten framebuffer auf den neuen so umkopiert, dass der Inhalt für den Benutzer weiterhin korrekt dargestellt wird.

```
uint16_t frame[128][160]; //pixel folgen zeilenweise aufeinander; uint16_t wegen 5,6,5 Farbkodierung  
//Wieviel Bytes: 128*160*2=40960Bytes, wir haben nur 2048 Bytes SRam zur Verfügung!!!!
```