Mikrocontroller strom-sparend programmieren

Uwe Berger
bergeruw@gmx.net

Uwe Berger

- Beruf: Softwareentwickler
- Freizeit: u.a. mit Hard- und Software rumspielen
- Linux seit ca. 1995
- BraLUG e.V.
- bergeruw@gmx.net

Inhalt

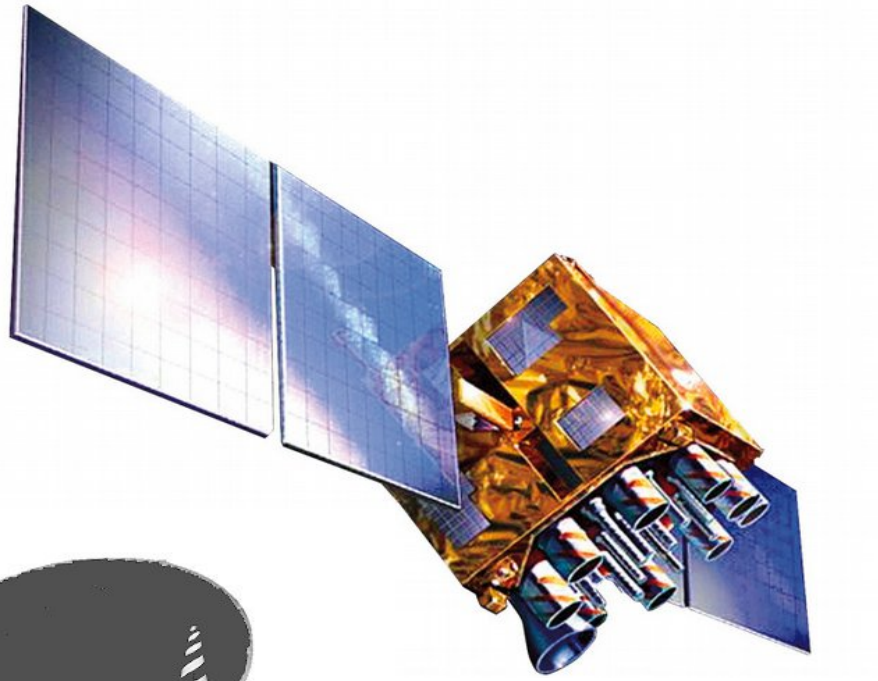
- Motivation
- Ultra-Low Power Hardware
- Stromsparende Firmware entwickeln
- Lebensdauer einer Primärbatterie

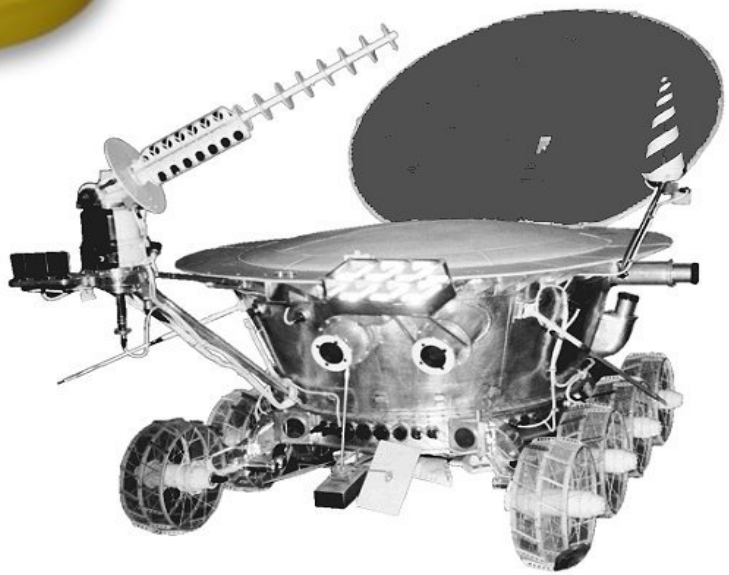
Modernes Design und tragbar...

Modernes Design und tragbar...

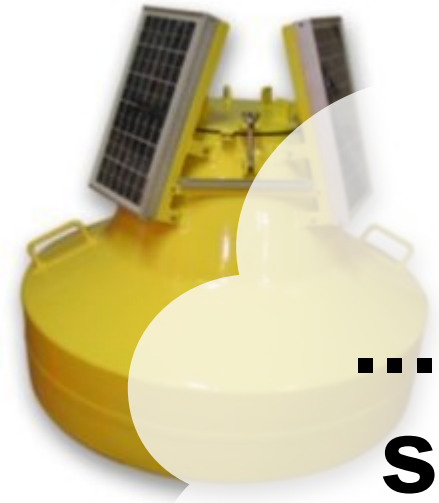
...soll lange funktionieren!

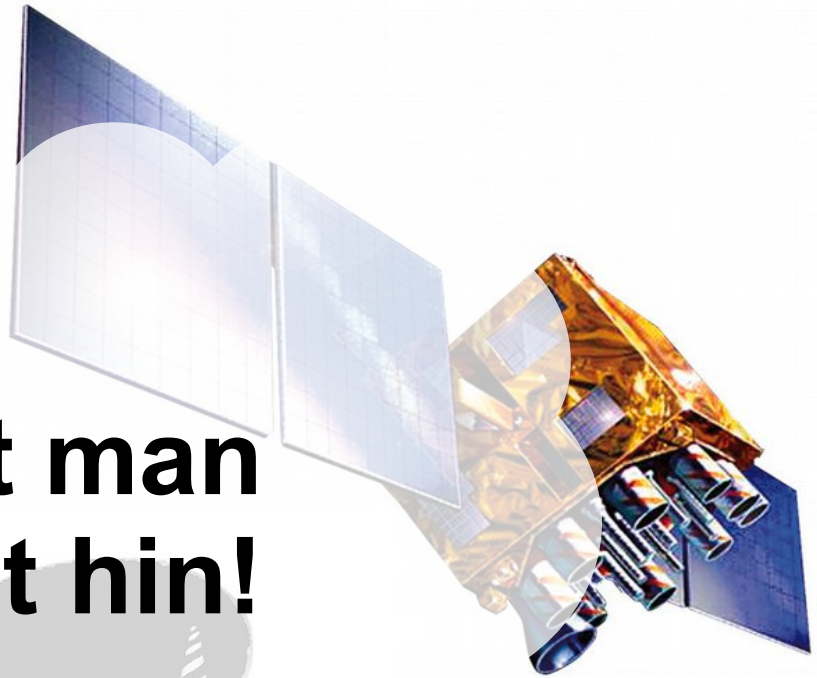
Wartungsarm und stromsparend...



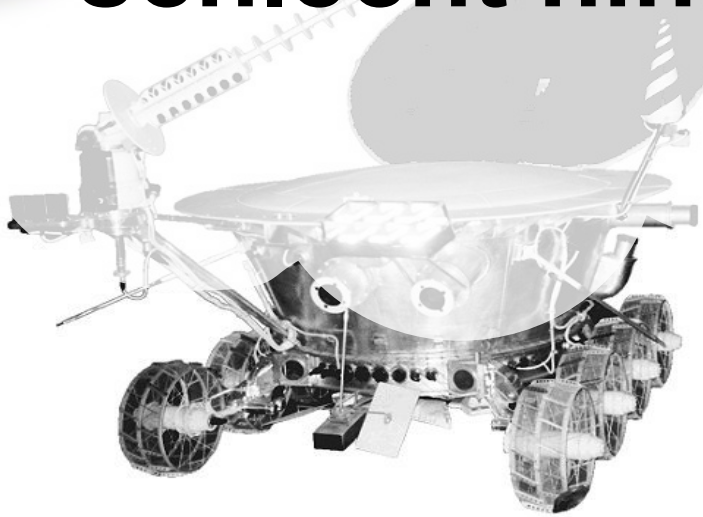


Wartungsarm und stromsparend...

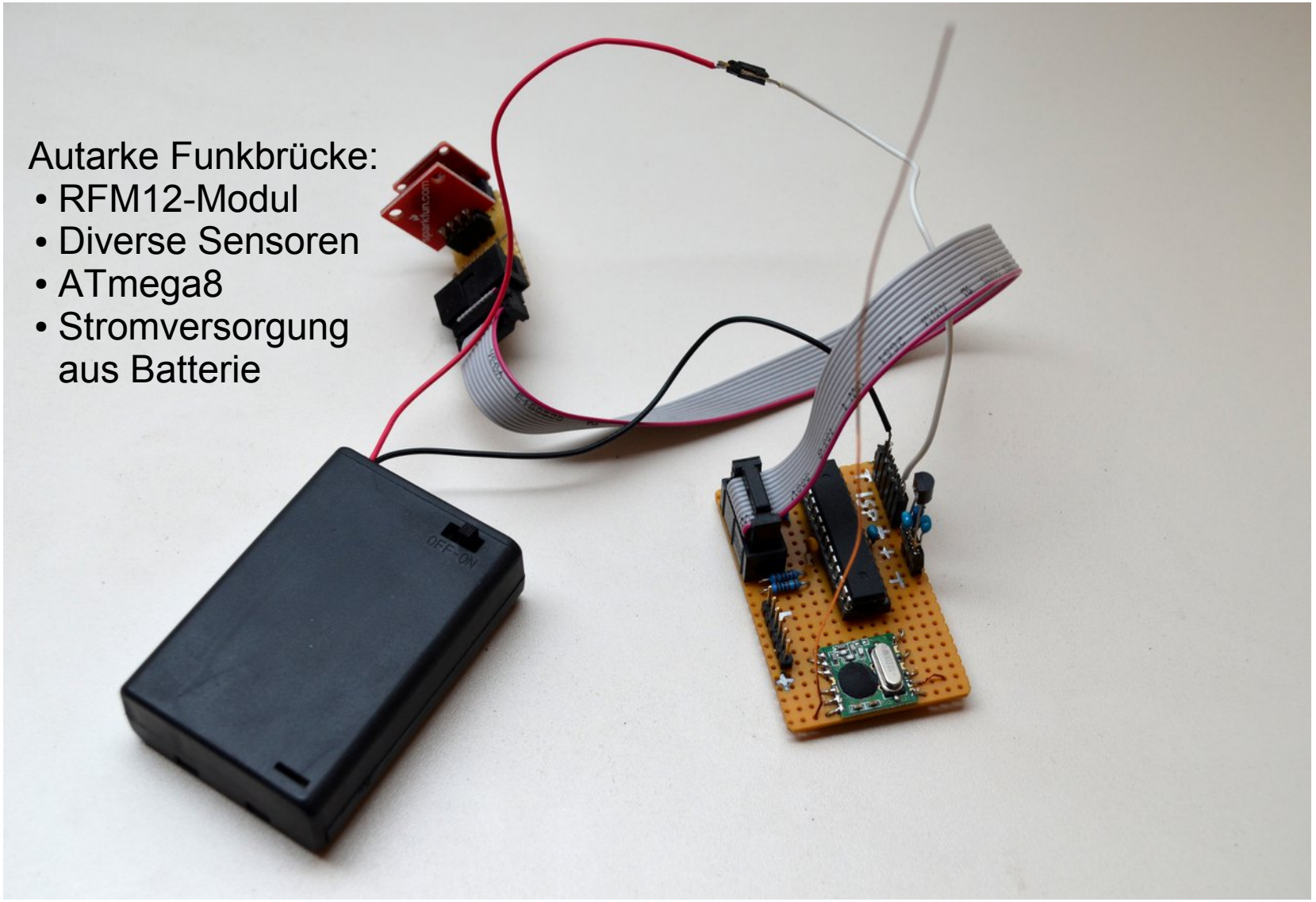




...kommt man schlecht hin!

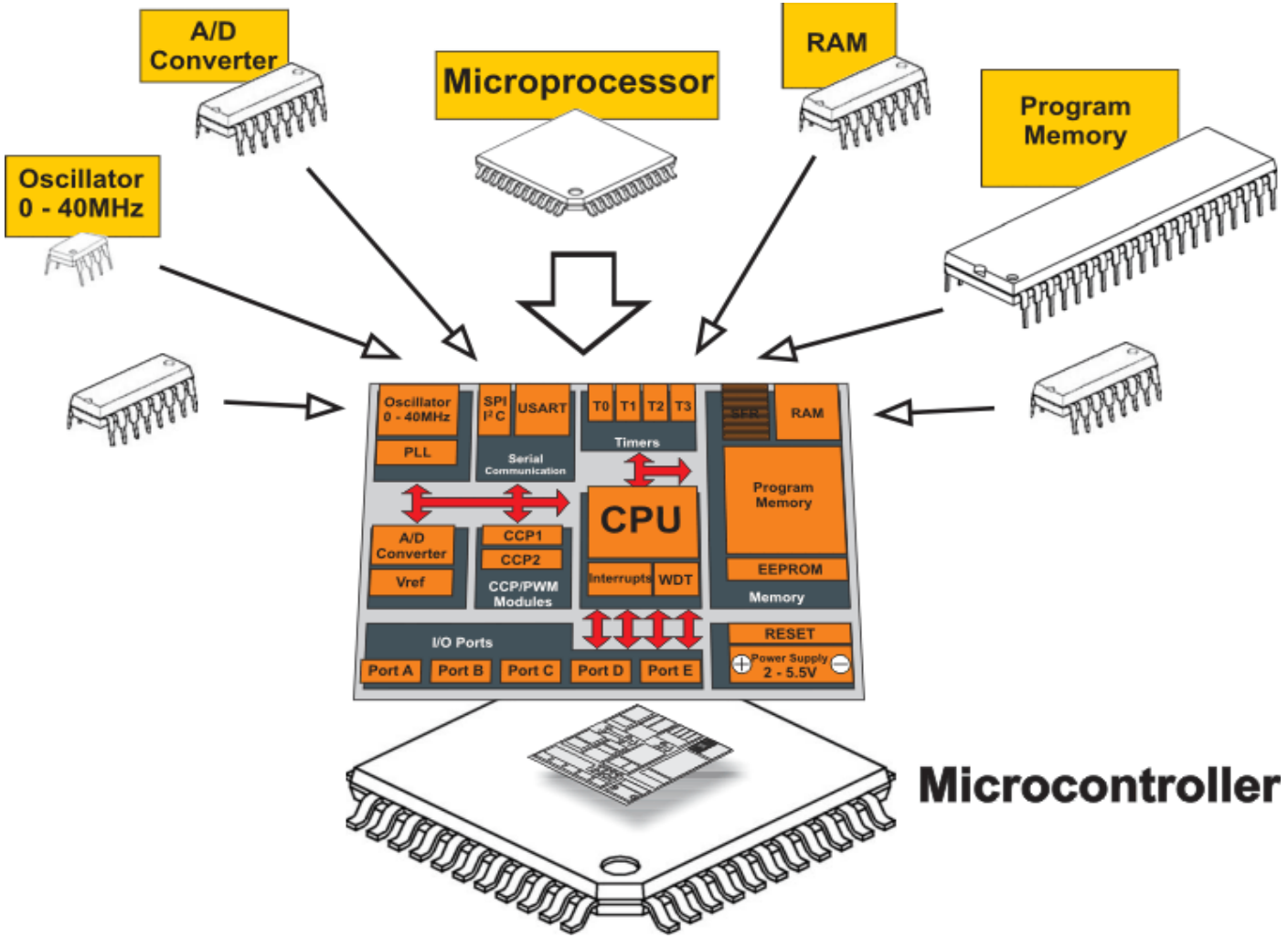


Meine Motivation...



Meine Motivation...

Aufbau Mikrocontroller (MCU)



Bildquelle: http://www.mikroe.com/chapters/view/1/

Besonderheiten stromsparender MCUs

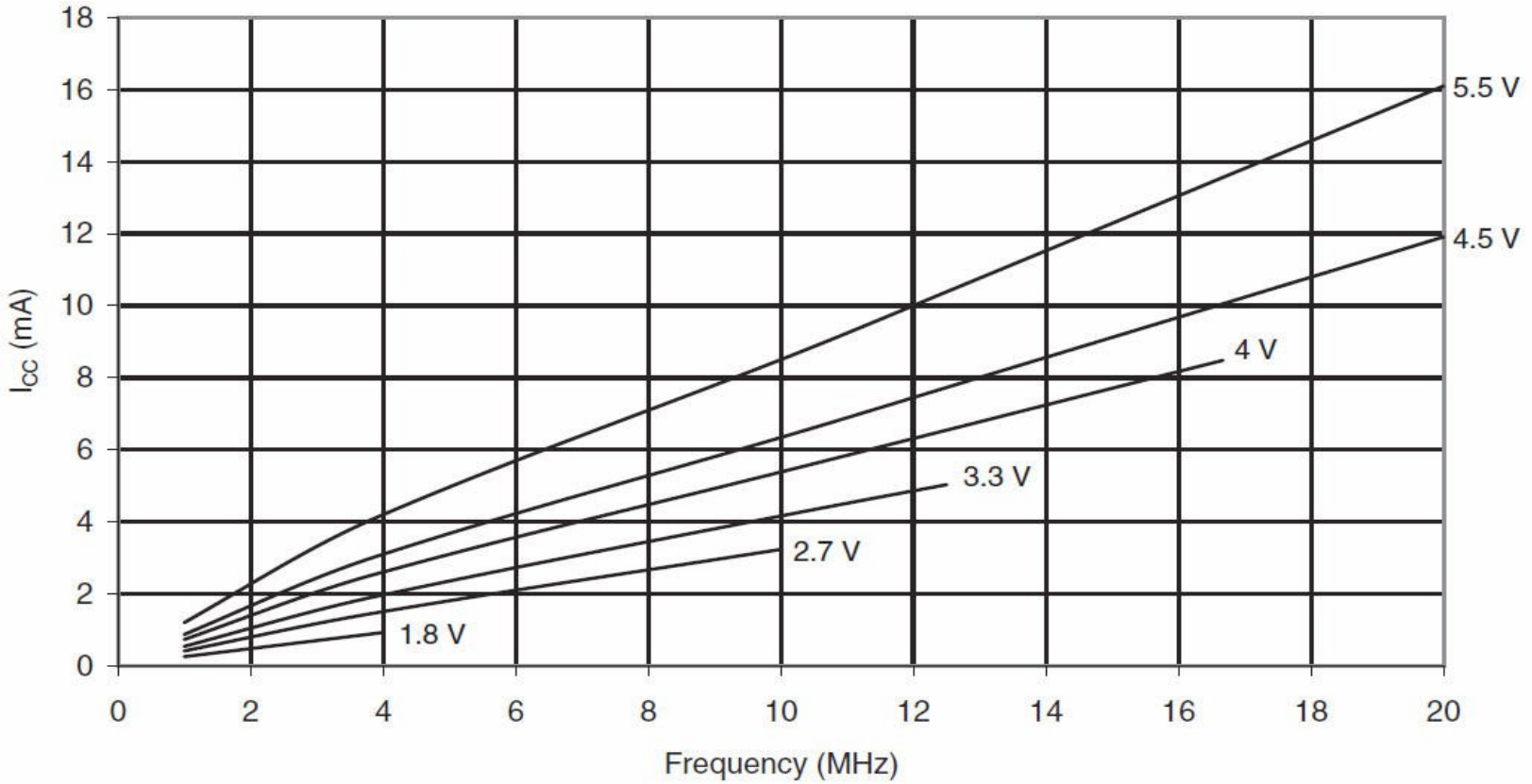
- Energieeffizientes Hardwaredesign
- Kleine Versorgungsspannungen
- **Diverse Schlaf-Modi**
- **Einzelne interne MCU-Komponenten abschaltbar**
- Beispiele verschiedener MCU-Hersteller:

Atmel:	„picoPower“-Familie
	Low-Power-Versionen diverser MCUs
TI:	„Ultra-Low Power“-MCUs (MSP430-Familie)
Microchip:	„eXtreme Low Power“-MCUs

MCU-Stromverbrauch: Aktiv-Mode

Beispiel: ATmega88P („picoPower“-Technologie; F. Atmel)

Figure 30-48. Active Supply Current vs. Frequency (1-20 MHz).



Quelle: Datenblatt ATmega88 (Firma Atmel)

Stromverbrauch: Stromspar-Modi

Beispiel: ATmega88P („picoPower“-Technologie; F. Atmel)

T_A = -40°C to 85°C, V_{CC} = 1.8V to 5.5V (unless otherwise noted)

Symbol	Parameter	Condition	Min.	Typ.(2)	Max.	Units
I_{CC}	Power Supply Current(1)	Active 1 MHz, V_{CC} = 2V		0.3	0.5	mA
		Active 4 MHz, V_{CC} = 3V		1.7	2.5	mA
		Active 8 MHz, V_{CC} = 5V		6.3	9	mA
		Idle 1 MHz, V_{CC} = 2V		0.05	0.15	mA
		Idle 4 MHz, V_{CC} = 3V		0.3	0.7	mA
		Idle 8 MHz, V_{CC} = 5V		1.4	2.7	mA
	Power-save mode(3)(4)	32 kHz TOSC enabled, V_{CC} = 1.8V		0.72	1.6	µA
		32 kHz TOSC enabled, V_{CC} = 3V		0.9	2.6	µA
	Power-down mode(3)	WDT enabled, V_{CC} = 3V		4.4	8	µA
		WDT disabled, V_{CC} = 3V		0.2	2	µA

Quelle: Datenblatt ATmega88 (Firma Atmel)

Stromverbrauch: MCU-Komponenten

Beispiel: ATmega88P („picoPower“-Technologie, Firma Atmel)

30.2.3 Supply Current of IO Modules

The tables and formulas below can be used to calculate the additional current consumption for the different I/O modules in Active and Idle mode. The enabling or disabling of the I/O modules are controlled by the Power Reduction Register. See ”Power Reduction Register” on page 43 for details.

Table 30-3. Additional Current Consumption for the different I/O modules (absolute values)

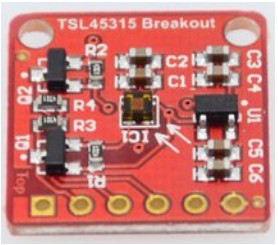
PRR bit	Typical numbers		
	V_{CC} = 2V, F = 1MHz	V_{CC} = 3V, F = 4MHz	V_{CC} = 5V, F = 8MHz
PRUSART0	4.12 uA	26.7 uA	108.3 uA
PRTWI	8.96 uA	58.6 uA	238.2 uA
PRTIM2	9.94 uA	64.1 uA	256.3 uA
PRTIM1	8.81 uA	56.9 uA	227.0 uA
PRTIM0	2.29 uA	15.5 uA	62.3 uA
PRSPI	8.31 uA	56.8 uA	260.4 uA
PRADC	9.27 uA	58.4 uA	230.8 uA

Quelle: Datenblatt ATmega88 (Firma Atmel)

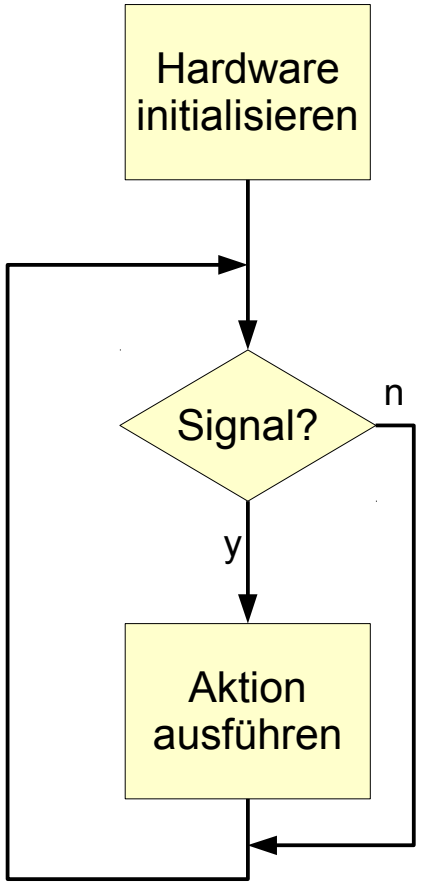
Stromverbrauch: externe Komponenten

- Viele externe Bauelemente/Baugruppen bieten ebenfalls Stromspar-Modi an → Datenblätter
- Beispiele:
  - Funkmodul RFM12
    - Aktiv (Sendebetrieb) → 21mA
    - Sleep → 0,3µA

  - Lichtsensor TSL4531
    - Aktiv → 110µA
    - Sleep → 2,2µA

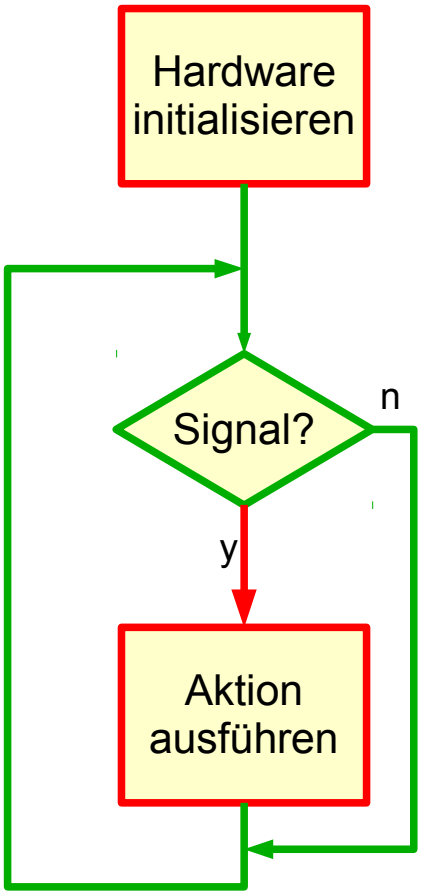


...ein typisches MCU-Programm



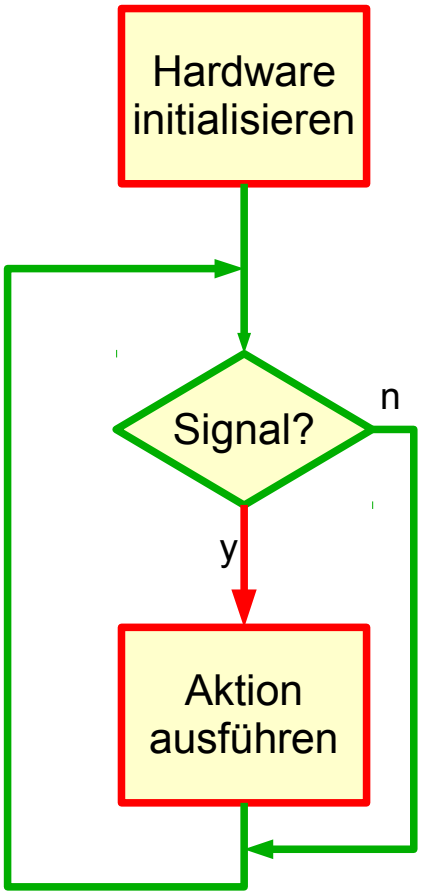
- EVA-Prinzip
  - Ereignis/Signal
  - Verarbeitung
  - Ausgabe

...ein typisches MCU-Programm

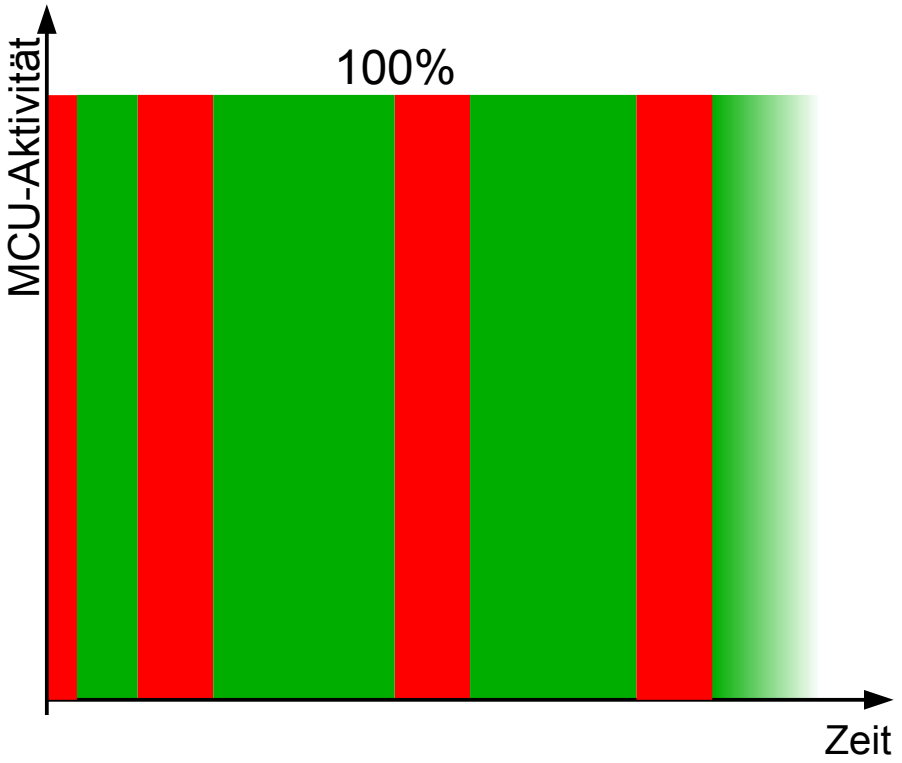


- MCU-Aktivität
  - **Aktiv**
  - **„Idle“**

...ein typisches MCU-Programm



- MCU-Aktivität
  - **Aktiv**
  - **„Idle“**



...ein typisches MCU-Programm

„MCU-Idle“ ist der Zeitraum in dem Strom gespart werden kann...!

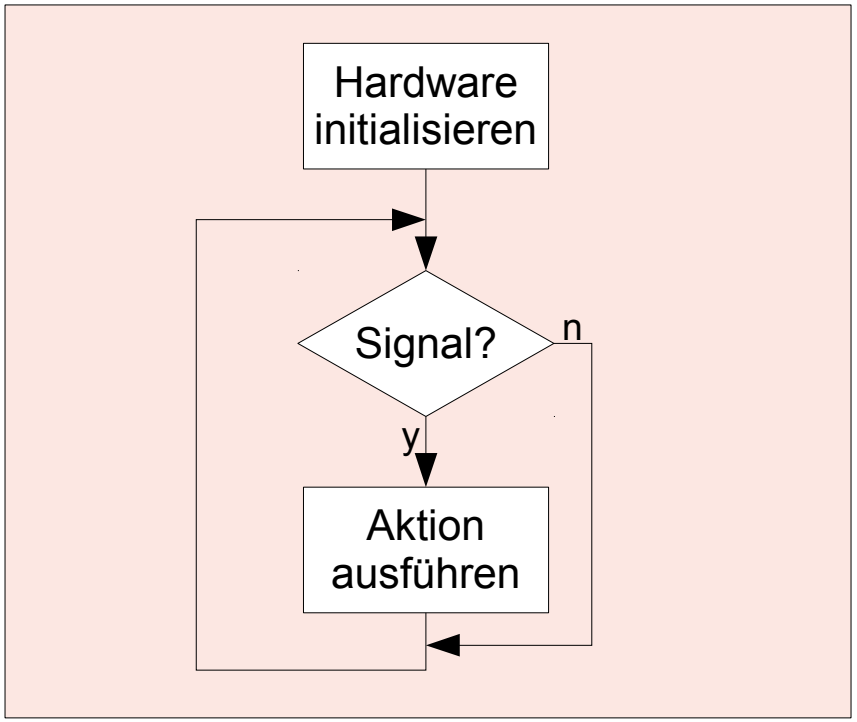
D.h. also:

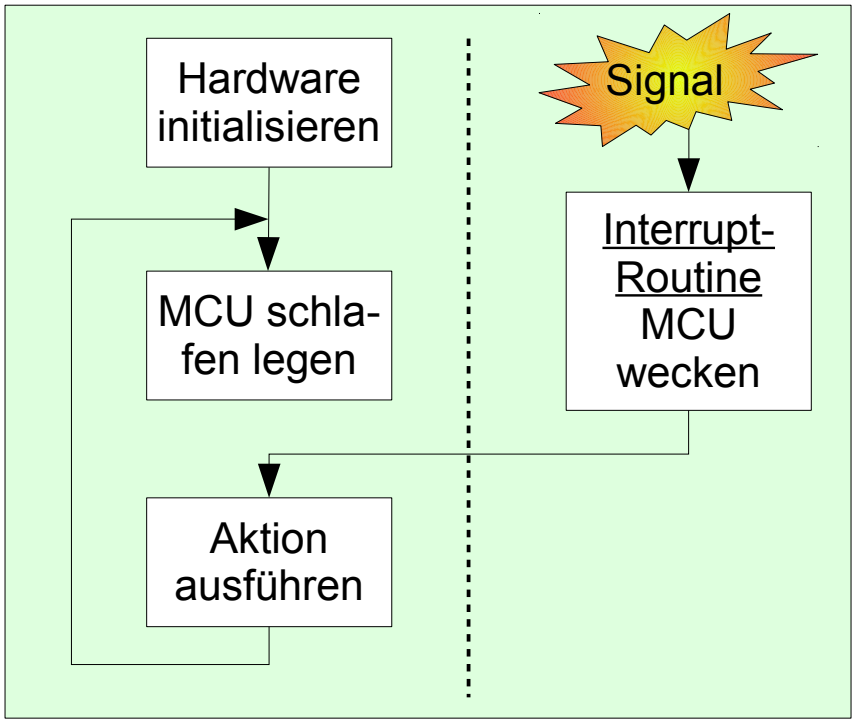
- Möglichst **kurzer und schneller Code**
- Während Wartezeiten („Idle“-Zeit) CPU **in Schlaf-Modus schalten**
- Unbenutzte **MCU- und externe Komponenten ausschalten**
- Möglichst **wenige Zugriffe auf Speicher** (RAM, FLASH, EEPROM)

„Hilfsmittel“/Tutorials zum Stromsparen

- Datenblätter
- Diverse Application Notes der Firma Atmel, z.B.: AVR035, AVR1010, AVR4013, AVR32739
- ULPAdvisor (Texas Instruments)
- Diverse weitere Veröffentlichungen

Nutze die Stromspar-Modi der MCU





- In einem „Endlos-Loop“ arbeitet die CPU ständig und verbraucht Strom
- CPU nur dann wecken, wenn auch etwas zu tun ist
  → **die meiste Zeit wird Strom gespart!**

Interrupts statt „Flag-Polling“

```
uint16_t  adc_read(uint8_t channel) {
  ADMUX |= (channel & 0x1F);
  ADCSRA |= (1<<ADSC);
  while (ADCSRA & (1<<ADSC)) {}
  return ADCW;
}
```

```
uint16_t  adc_read(uint8_t channel) {
  ADMUX |= (channel & 0x1F);
  set_sleep_mode(SLEEP_MODE_ADC);
  ADCSRA |= (1<<ADSC)|(1<<ADIE);
  sleep_mode();
  return ADCW;
}

ISR(ADC_vect){
}
```

- Beispiel ADC-Messung mit einem AVR
- Während der Messung Sleep-Mode „ADC Noise Reduction“
  → **ca. 60% weniger Stromverbrauch** in diesem Zeitraum

Timer statt „Pause-Schleife“

```
void long_delay_ms(uint16_t ms) {
   for(; ms>0; ms--) _delay_ms(1);
}

int main(void) {
      DDRC |= (1<<PC1);
      while(1) {
            PORTC ^= (1<<PC1);
            long_delay_ms(500);
      }
}
```

```
int main(void) {
      DDRC |= (1<<PC1);
      TCCR1B |= (1 << WGM12);
      TIMSK |= (1 << OCIE1A);
      OCR1A   = 7812;
      sei();
      while(1) {
            // was anderes machen...
            // ... oder Sleep-Mode...
      }
}

ISR(TIMER1_COMPA_vect) {
      PORTC ^= (1<<PC1);
}
```

- AVR-Beispiel 1Hz-Takt-Signal an Pin PC1 erzeugen
- **CPU nicht „blockiert“** und kann zwischen den Timer-Interrupts noch andere Dinge machen → insgesamt schneller

Keine langen Interrupt-Routinen

```
int main(void) {
  // INT0 konfigurieren...
  while(1) {
    // ...mache etwas
  }
}

ISR(INT0_vect) {
  printf("Signal an INT0-Pin!\n");
}
```

```
volatile uint8_t flag=0;
int main(void) {
  // INT0 kongigurieren
  while(1) {
  if (flag) {
    printf("Signal an INT0-Pin!\n");
    flag=0;}
  // ...mache etwas
  }
}

ISR(INT0_vect) {
  flag=1;
}
```

- AVR-Beispiel: Aktion, wenn Signal an INT0-Pin
- Kurze Interrupt-Routine → **schnelle Reaktion auf interne und externe Signale**; keine Interrupt-Überlappungen

Keine rechenintensive Operationen

- Floating-Point ohne FPU
- Ganzzahlmultiplikation ohne Hardware-Multiplizierer
- Modulo-/Division-Operationen
- math.h (sin(), cos(), sqrt(), etc.)
- stdio.h, string.h, stdlib.h

- Festkomma-Arithmetik
- Lookup-Tabellen
- MCU-spezifische Mathematik-Bibliotheken
- Eigenes, angepasstes „printf()“, atoi(), scanf(), etc.

- Minimierung rechenintensiver Operationen
  → **kürzerer und schnellerer Code**

Nutze DMA, wenn vorhanden

```
#define LEN 100
char str1[LEN], str2[LEN];
uin8_t i;
...

for (i=LEN; i>0; i--) {
  str2[i-1]=str1[i-1];
}
...
memcpy(str2, str1, LEN);
...
```

```
#define LEN 100
char str1[LEN], str2[LEN];
...

DMA0DA = (unsigned int)str2;
DMA0SA = (unsigned int)str1;
DMA0SZ = LEN;
DMA0CTL |= DMAEN;
DMA0CTL |= DMAREQ;
while (!(DMA0CTL & DMAIFG)) ;
...
```

- Beispiel MSP430Fxxxx: große Variable kopieren
- DMA-Nutzung → kürzerer Code; während des Transfers **kann CPU abgeschaltet werden** oder etwas anderes tun

Lokale statt globale Variablen

```
uint8_t i;
...
void do_it(void) {
  for (i=0; i<10; i++) {
    Printf("%d\n", i);
  }
}
```

```
void do_it(void) {
  uint8_t i;
  for (i=0; i<10; i++) {
    Printf("%d\n", i);
  }
}
```

- Der Compiler legt, wenn möglich, lokale Variablen gleich in CPU-Register an
  → **kürzerer, schnellerer Code; keine unnötigen Speicherzugriffe**

„Call by reference“ bei großen Variablen

```
void call_by_value(uint64_t z) {
  printf(“Antwort %d“, z);
}

uint64_t answer = 42;
...
pass_by_value(answer);
...
```

```
void call_by_reference(uint64_t *z) {
  printf(“Anwort %d“, *z);
}

uint64_t answer = 42;
...
pass_by_reference(&answer);
...
```

- „call by value“
  → Übergabewert muss erst im Speicher umkopiert werden
- „call by reference“
  → **kürzerer, schnellerer Code; weniger Speicherzugriffe**

„const“ und „static“

```
void do_it(void) {
  uint8_t answer=42;
  uint8_t zahl=23;
  printf(“%d statt %d?\n“, zahl, answer);
}
...
for (i=10; i>0; i--) {
  do_it();
}
```

```
void do_it(void) {
  const uint8_t answer=42;
  static uint8_t zahl=23;
  printf(“%d statt %d?\n“, zahl, answer);
}
...
for (i=10; i>0; i--) {
  do_it();
}
```

- „static“-Variablen werden einmal angelegt und existieren über die ganze Laufzeit → **kürzerer, schnellerer Code**
- „const“-Variablen werden direkt aus dem Programmspeicher gelesen → **schnellerer Code; weniger Speicherzugriffe**

„Ausreichenden“ Variablentyp verwenden

```
#define LEN 100
char a[LEN];
int i;
...
for (i=0; i<LEN; i++) {
  printf(“%c“, a[i]);
}
```

```
#define LEN 100
char a[LEN];
uint8_t i;
...
for (i=0; i<LEN; i++) {
  printf(“%c“, a[i]);
}
```

- Feldindizes können (in C) nicht negativ sein; die Zahl 100 passt auch in ein Byte rein
  → **kürzerer, schnellerer Code**

Bitmasken statt Bitfelder

```
struct status_t {
  unsigned s1    : 1;
  unsigned s2    : 1;
  unsigned s3    : 1
  unsigned res   : 5;
} status;
...
status.s1 = 1;
status.s2 = 1;
if (status.s2) status.s3=0;
...
```

```
#define S1   1
#define S2   2
#define S3   4

uint8_t status;
...
status |= S1 | S2;
if (status & S2) status &= ~S3;
...
```

- Quelltext mit Bitfeldern ist zwar „schöner“, aber...
- Zugriffe mit Bitmasken kann der Compiler optimieren
  → **schnellerer, kürzerer Code**

„Rückwärts“ statt „vorwärts“ zählen

```
uint16_t i;
...
for (i=0; i<10000; i++) {
  // mache etwas...
}
...
```

```
uint16_t i;
...
for (i=10000; i>0; i--) {
  // mache etwas...
}
...
```

- Bei „Vorwärts-Schleifen“ muss auf obere Grenze abgeprüft werden bevor Sprung → zusätzliche Maschinenbefehle
- Bei „Rückwärts-Schleifen“ reicht die Prüfung ob i ungleich 0; viele CPUs kennen einen Maschinenbefehl „Verzweige, wenn 0“ → **kürzerer, schnellerer Code**

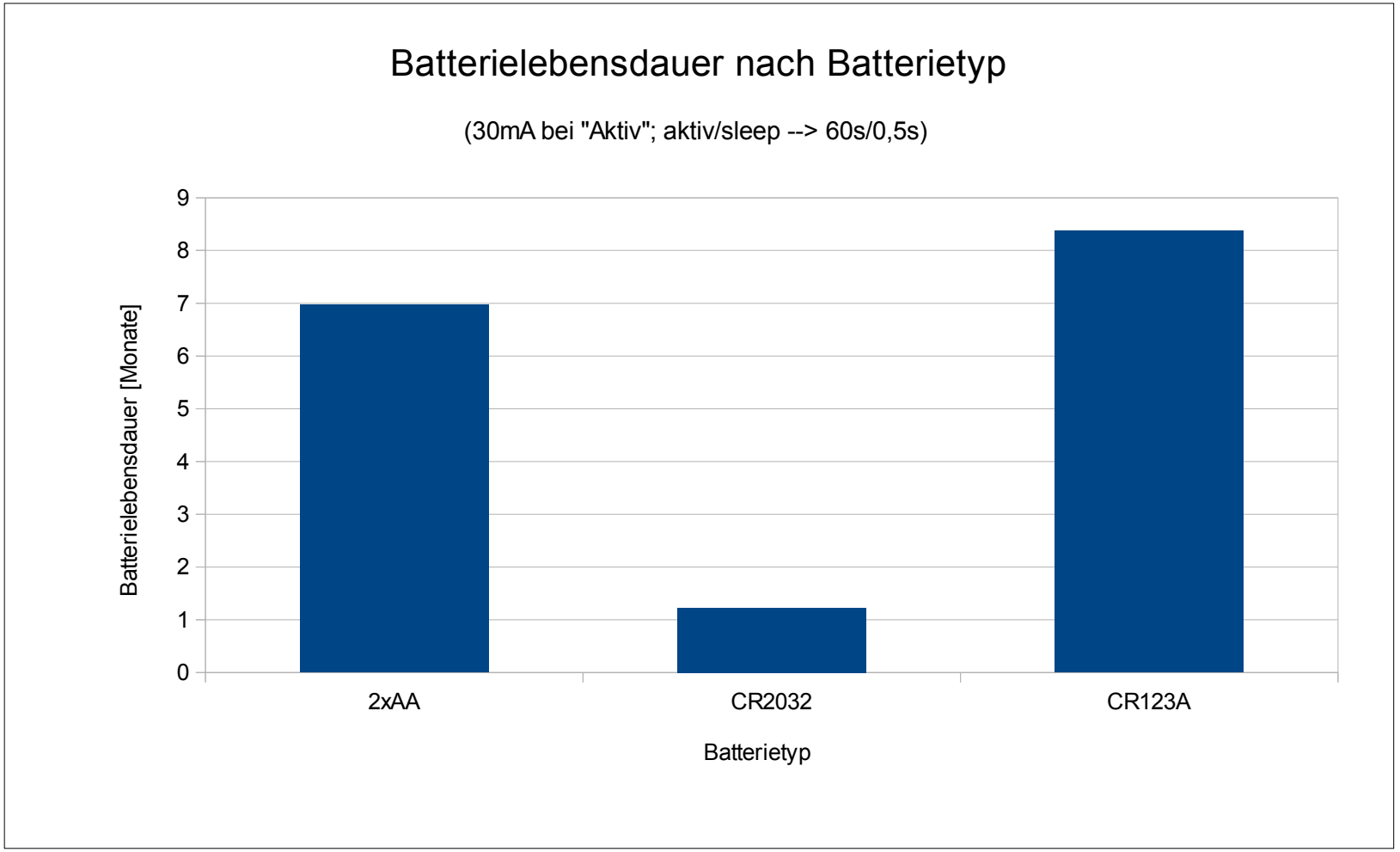
Verbreitete Primärbatterien

	AA (2x)	CR2032	CR123A
Nennspannung	2x1,5V = 3V	3V	3V
Kapazität	2500mAh	220mAh	1500mAh
Kapazität bis 2,55V	1250mAh	220mAh	1500mAh
Selbstentladung (21°C)	3%	1%	1%

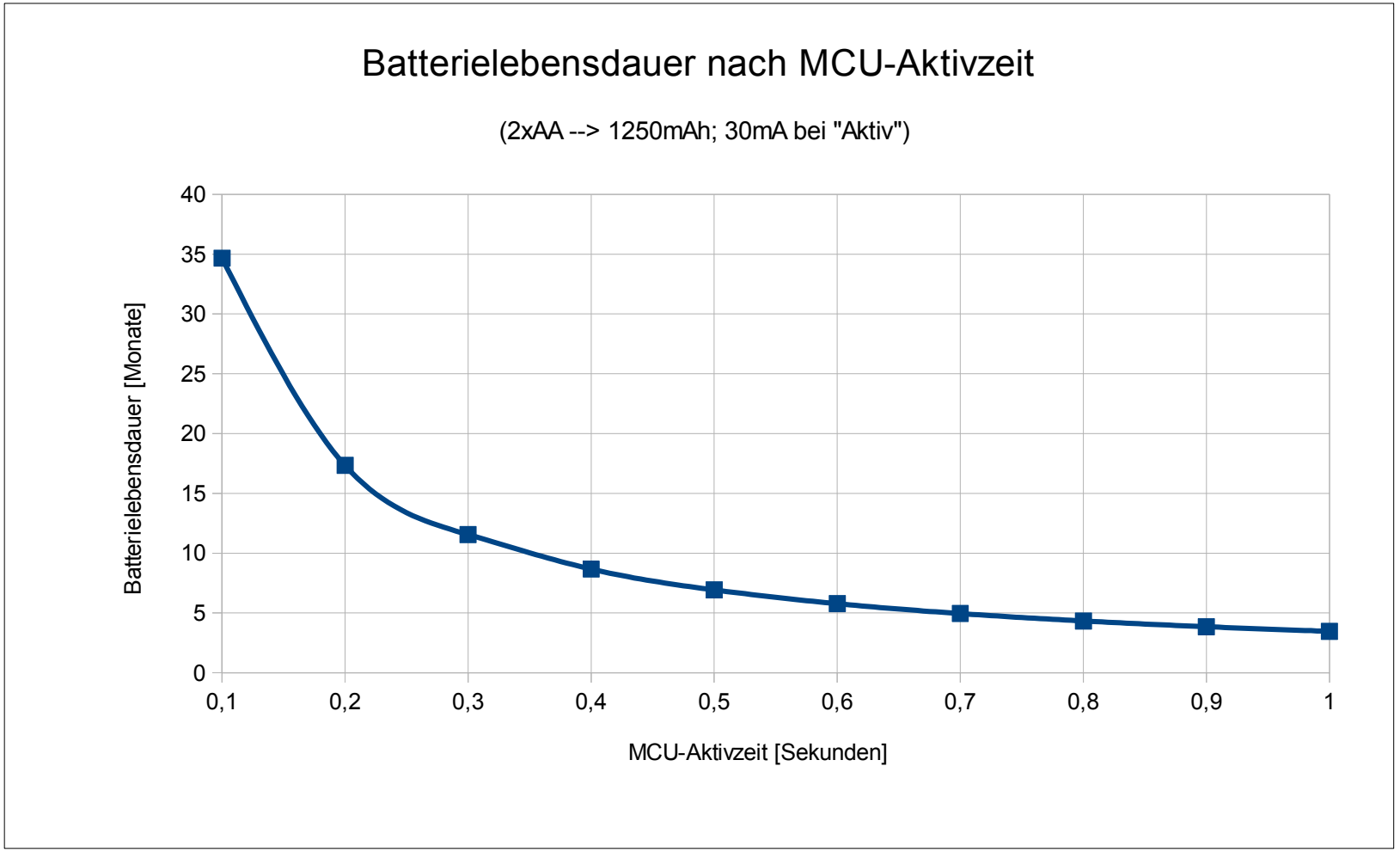
Beispielrechnung: Batterielebensdauer

- Stromverbrauch bei 3V Versorgungsspannung:
  - Aktiv-Modus: **30mA**
  - Schlaf-Modus: 1µA (erst mal vernachlässigbar...)
- 2 AA-Batterien bis 2,55V → **1250mAh**
  - 30mA permanent: 1250mAh/30mA=**41,6h**
- Verhältnis Aktiv-/Schlaf-Modus: 0,5s/60s
  - 1h=60min → 60*0,5s=30s → **0,83%** einer Stunde
- Resultierende Batterielebensdauer:
  - 41.6h/0,0083=5012h → 209d → **6,97 Monate**

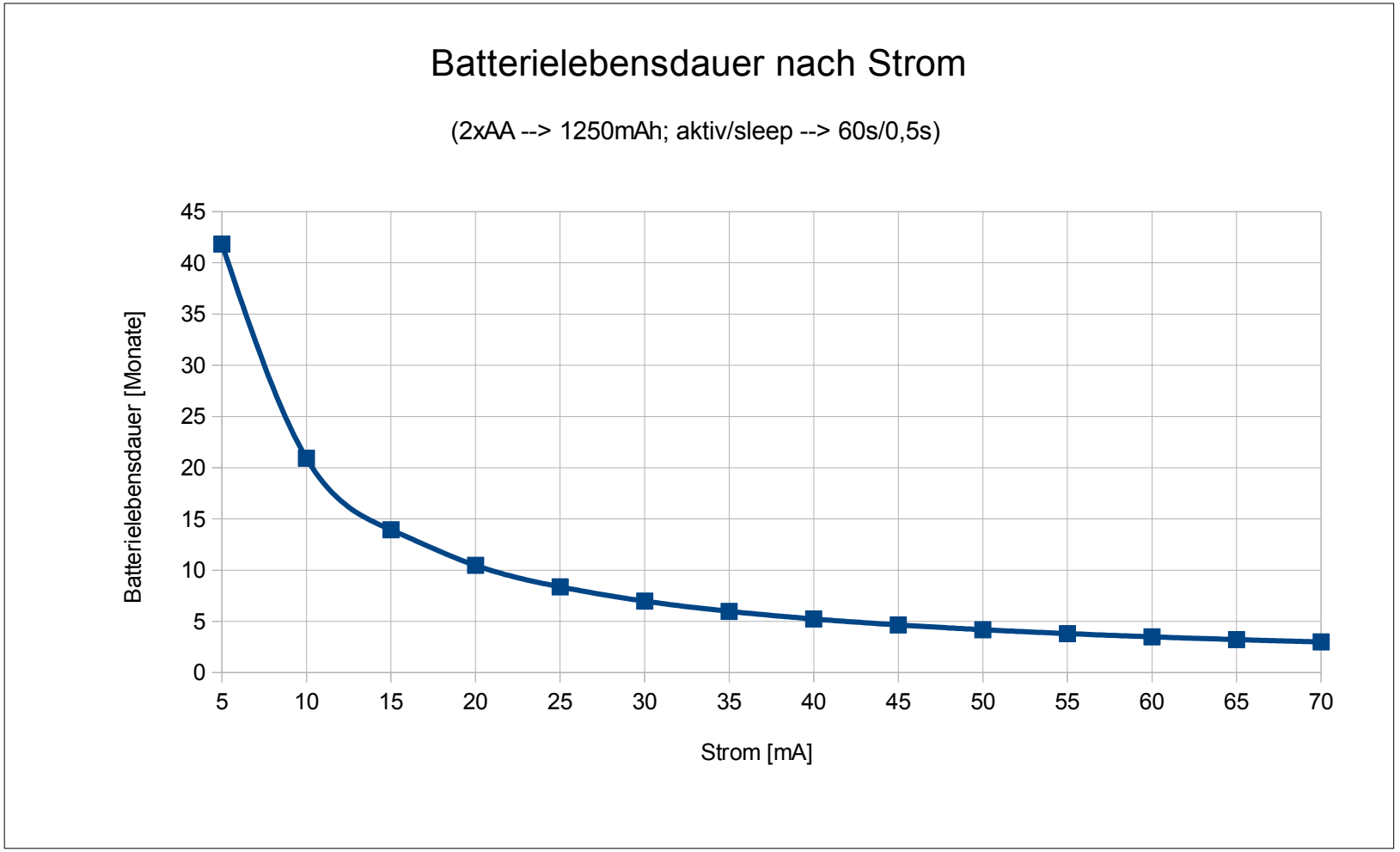
Batterielebensdauer: „Was wäre wenn?“



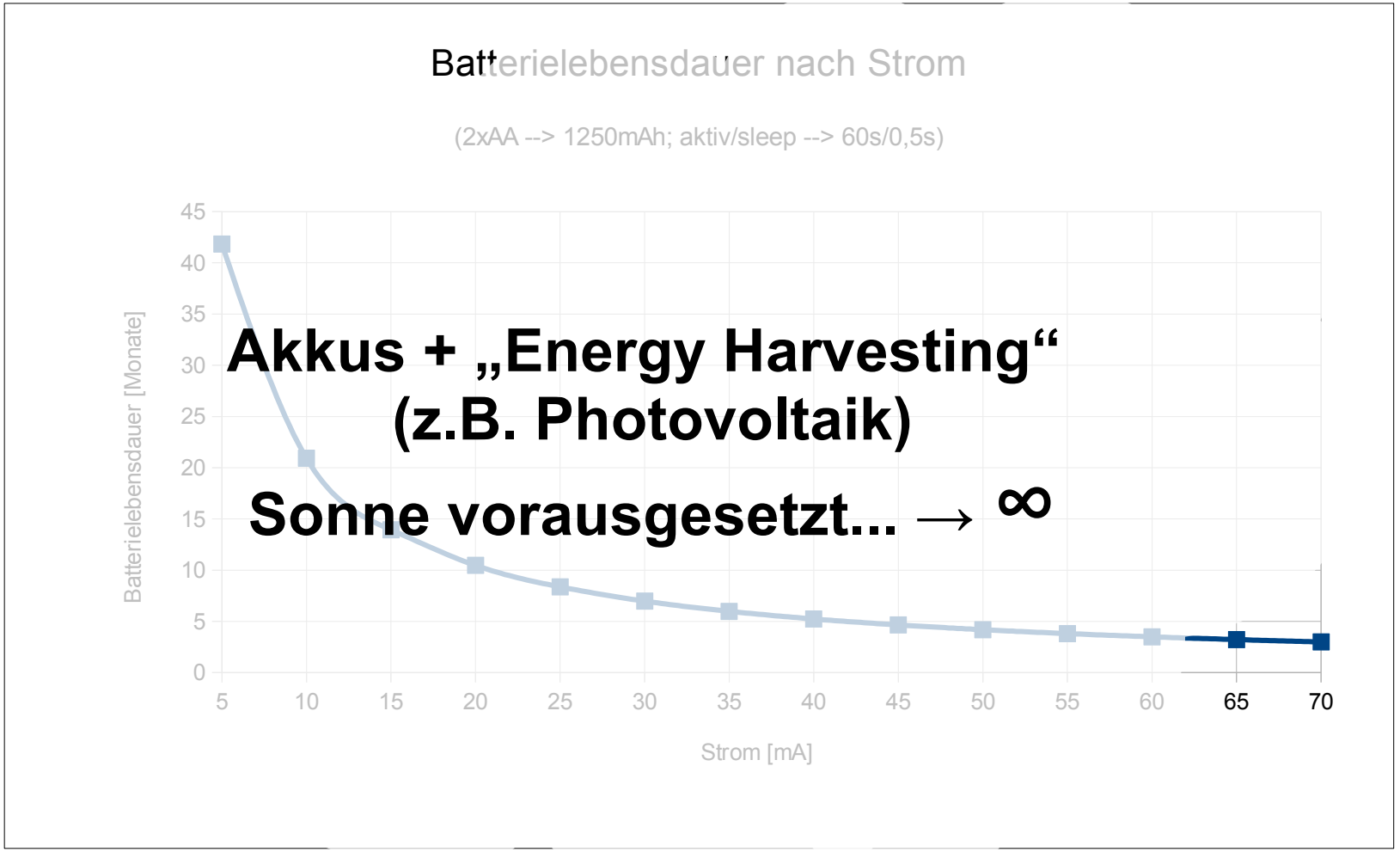
Batterielebensdauer: „Was wäre wenn?“



Batterielebensdauer: „Was wäre wenn?“



Batterielebensdauer: „Was wäre wenn?“



Eine Batterie „für immer und ewig“?

- Lebenszyklus elektronischer Geräte/Baugruppen:
  - Durchschnittlich ca. 10 Jahre,
  - Elektronische Massenproduktion 2-3 Jahre...
- Einflussfaktoren:
  - Verschleiß, Abnutzung
  - Technologischer Fortschritt
  - Geplante Obsoleszenz...

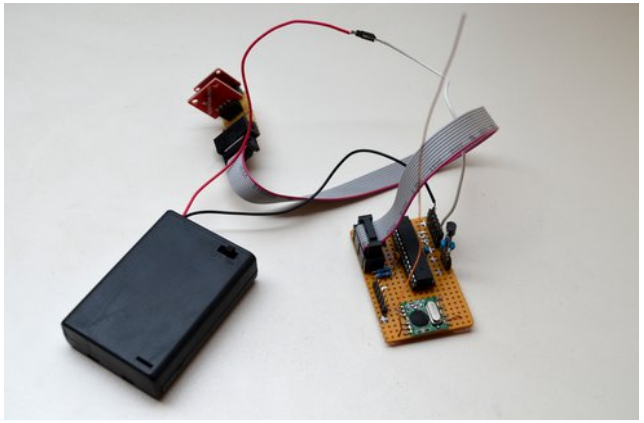
Eine Batterie „für immer und ewig“?

- Lebenszyklus elektronischer Geräte/Baugruppen:
  - Durchschnittlich ca. 10 Jahre,
  - Elektronische Mass
- Einflussfa
  - Versch
  - Te
  - Ge

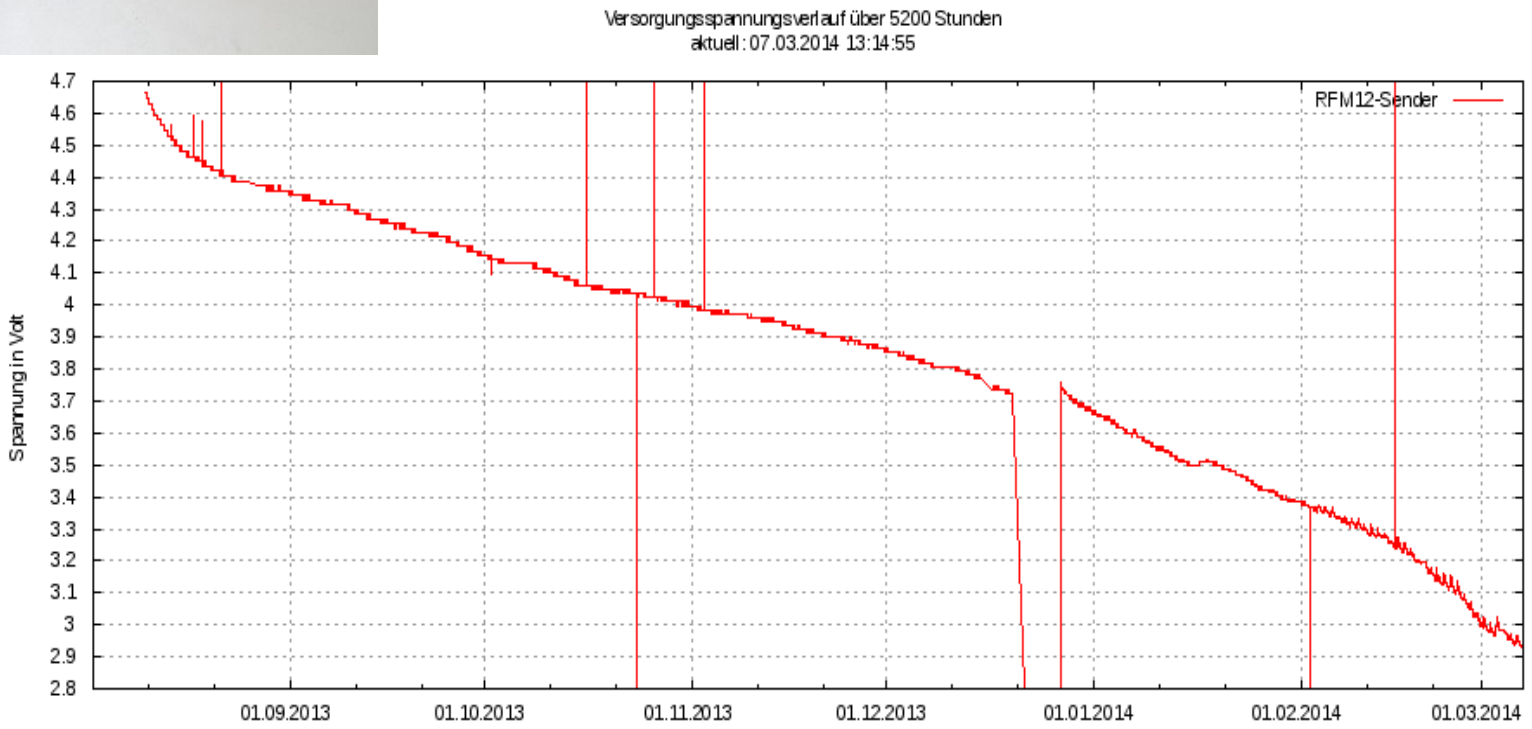
Die „10-Jahre-Batterie“:

- 2xAA → permanent 9,8µA
- CR2032 → permanent 2,2µA
- CR123A → permanent 15,4µA

Batterielebensdauer: reales Beispiel



Stromversorgung:	3xAA
Stromverbrauch:	ca. 35mA
Verhältnis aktiv/inaktiv:	0,5s/60s



Informationsquellen

- Datenblätter, Datenblätter, Datenblätter...
- http://bralug.de/wiki/Mikrocontroller_stromsparend_programmieren

Danke für die Aufmerksamkeit!

14:00 Uhr im Raum V3:

„Wenn Geeks Langeweile haben“ - reloaded