Systemnahe Programmierung in C (SPiC)

91 Abschlussbemerkungen

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2020

http://www4.cs.fau.de/Lehre/SS20/V_SPIC

Hinweise Klausur

Prüfungszeitraum	GSPiC-Prüfung (60 min)	SPiC-Prüfung (90 min)	InfoEEI-Prüfung (90 min)	Manpages
August 2019	2019-08-02	2019-08-02	2019-08-02	exec(), fork(), kill(), waitpid()
April 2019	2019-04-04	2019-04-04	2019-04-04	opendir()/readdir() /closedir(), lstat()/stat(), String functions
Juli 2018	2018-07-20	2018-07-20	2018-07-20	fopen(), fgets()/fputs(), strcmp()
März 2018	2018-03-21	2018-03-21		fork(), opendir()/readdir() /closedir(), wait()
August 2017	2017-08-04	2017-08-04		fork(), strcat(), strcpy(), wait()
April 2017	2017-04-22	2017-04-22		stat(), fopen(), fgetc()
Juli 2016	2016-07-22	2016-07-22		exec(), fgets()/fputs(), waitpid(), fork()
März 2016	2016-03-23	2016-03-23		exec(), fopen()/fgets(), strtok(), wait()
Juli 2015	2015-07-27	2015-07-27		opendir()/readdir() /closedir()
März 2015	2015-03-25	2015-03-25		exec(), fork(), stat(),

https://www4.cs.fau.de/Lehre/SS20/V_SPIC/Pruefung/

- Fragestunde 16.7., 16:00
- Forum Studon `https://www.studon.fau.de/`
- `i4spic@lists.cs.fau.de`
- `i4spic-orga@lists.cs.fau.de`

enum ↦ int [≠Java]

- Technisch sind enum-Typen Integers (int)
  - enum-Konstanten werden von 0 an durchnummeriert

```
typedef enum { RED0,      // value: 0
               YELLOW0,   // value: 1
               GREEN0,    // value: 2
               ··· } LED;
```

  - Es ist auch möglich, Werte direkt zuzuweisen

```
typedef enum { BUTTON0 = 4, BUTTON1 = 8 } BUTTON;
```

  - Man kann sie verwenden wie ints (z. B. mit ihnen rechnen)

```
sb_led_on(RED0 + 2);  // -> LED GREEN0 is on
sb_led_on(1);         // -> LED YELLOW0 is on
for( int led = RED0, led <= BLUE1; led++ )
  sb_led_off(led);    // turn off all LEDs
// Also possible...
sb_led_on(4711);      // no compiler/runtime error!
```

- ⇝ Es findet **keinerlei Typprüfung** statt!

Typumwandlung in Ausdrücken (Forts.)

- unsigned-Typen gelten dabei als „größer" als signed-Typen

```
int s = -1, res;          // range: -32768 --> +32767
unsigned u = 1;           // range: 0 --> 65535

res = s < u;              // promotion to unsigned: -1 --> 65535
int: 0   unsigned: 65535
         unsigned: 0
```

⇝ Überraschende Ergebnisse bei negativen Werten!

⇝ Mischung von signed- und unsigned-Operanden vermeiden!

Schleifensteuerung [=Java][↪ GDI, 08-09]

- Die continue-Anweisung beendet den aktuellen Schleifendurchlauf
  ⇝ Schleife wird mit dem nächsten Durchlauf fortgesetzt

```
for (uint8_t led = 0; led < 8; led++) {
  if (led == RED1) {
    continue;          // skip RED1
  }
  sb_led_on(led);
}
```

- Die break-Anweisung verlässt die (innerste) Schleife
  ⇝ Programm wird *nach* der Schleife fortgesetzt

```
for (uint8_t led = 0; led < 8; led++) {
  if (led == RED1) {
    break;          // break at RED1
  }
  sb_led_on(led);
}
```

Funktionsdeklaration [≠Java]

- Funktionen müssen vor ihrem ersten Aufruf im Quelltext deklariert (↦ bekannt gemacht) worden sein
  - Eine voranstehende Definition beinhaltet bereits die Deklaration
  - Ansonsten (falls die Funktion „weiter hinten" im Quelltext oder in einem anderen Modul definiert wird) muss sie explizit deklariert werden
- Syntax: *Typ Bezeichner* **(** *FormaleParam* **) ;**
- Beispiel:

```
// Deklaration durch Definition
int max(int a, int b) {
  if (a > b) return a;
  return b;
}

void main(void) {
  int z = max(47, 11);
}
```

```
// Explizite Deklaration
int max(int, int);

void main(void) {
  int z = max(47, 11);
}

int max(int a, int b) {
  if (a > b) return a;
  return b;
}
```

Spezialitäten der C-Programmierung

Präprozessor – Gefahren [≠ Java]

- Funktionsähnliche Makros sind keine Funktionen!
  - Parameter werden nicht evaluiert, sondern textuell eingefügt
    Das kann zu unangenehmen Überraschungen führen

```
#define POW2(a) 1 << a                 << hat geringere Präzedenz als *
n = POW2(2) * 3                  ⇝ n = 1 << 2 * 3
```

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

```
#define POW2(a) (1 << a)
n = POW2(2) * 3                  ⇝ n = (1 << 2) * 3
```

  - Aber nicht alle

```
#define max(a, b) ((a > b) ? a : b)    a++ wird ggf. zweimal ausgewertet
n = max(x++, 7)                  ⇝ n = ((x++ > 7) ? x++ : 7)
```

11-Praeprozessor: 2020-05-08

Module in C – Header (Forts.) [≠ Java]

Modulschnittstelle: `foo.h`

```
// foo.h
#ifndef _FOO_H
#define _FOO_H

// declarations
extern uint16_t a;
void f(void);

#endif // _FOO_H
```

Modulimplementierung `foo.c`

```
// foo.c
#include <foo.h>


// definitions
uint16_t a;
void f(void){
  ...
}
```

Modulverwendung `bar.c`
(vergleiche ↪ 12–11)

```
// bar.c
extern uint16_t a;
void f(void);
#include <foo.h>

void main() {
  a = 0x4711;
  f();
}
```

12-Module: 2020-04-27

Felder sind Zeiger sind Felder – Zusammenfassung

- In Kombination mit Zeigerarithmetik lässt sich in C jede Feldoperation auf eine äquivalente Zeigeroperation abbilden.
- Für `int i, array[N], *ip = array;` mit $0 \leq i < N$ gilt:

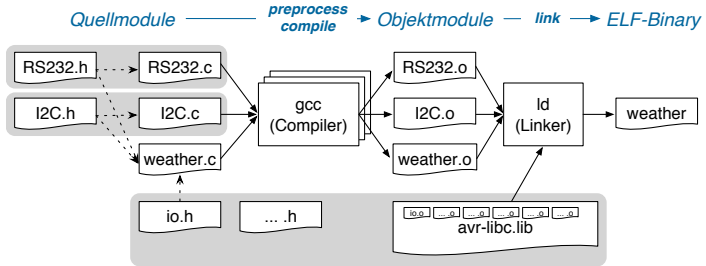
`array`	≡	`&array[0]`	≡	`ip`	≡	`&ip[0]`
`*array`	≡	`array[0]`	≡	`*ip`	≡	`ip[0]`
`*(array + i)`	≡	`array[i]`	≡	`*(ip + i)`	≡	`ip[i]`
		`array++`	≢	`ip++`		

Fehler: `array` ist konstant!

- Umgekehrt können Zeigeroperationen auch durch Feldoperationen dargestellt werden.
  Der Feldbezeichner kann aber nicht verändert werden.

13-Zeiger: 2020-05-06

Zurück zum Beispiel: Wetterstation



- Jedes Modul besteht aus Header- und Implementierungsdatei(en)
  - `.h`-Datei definiert die Schnittstelle
  - `.c`-Datei implementiert die Schnittstelle, inkludiert `.h`-Datei, um sicherzustellen, dass Deklaration und Definition übereinstimmen
- Modulverwendung durch Inkludieren der modulspezifischen `.h`-Datei
- Das Ganze funktioniert entsprechend bei Bibliotheken

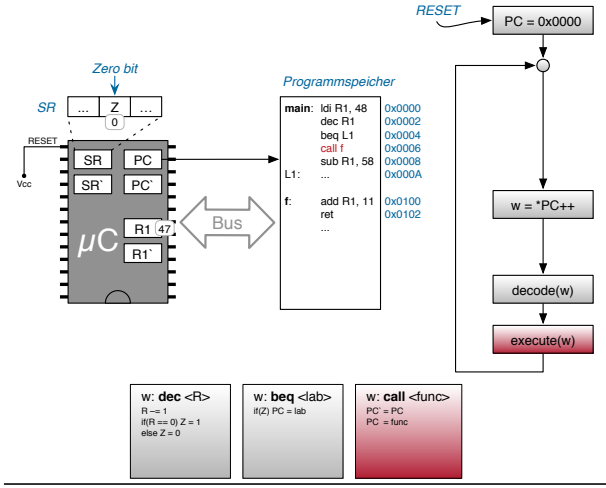
12-Module: 2020-04-27

Was tut ein Compiler?

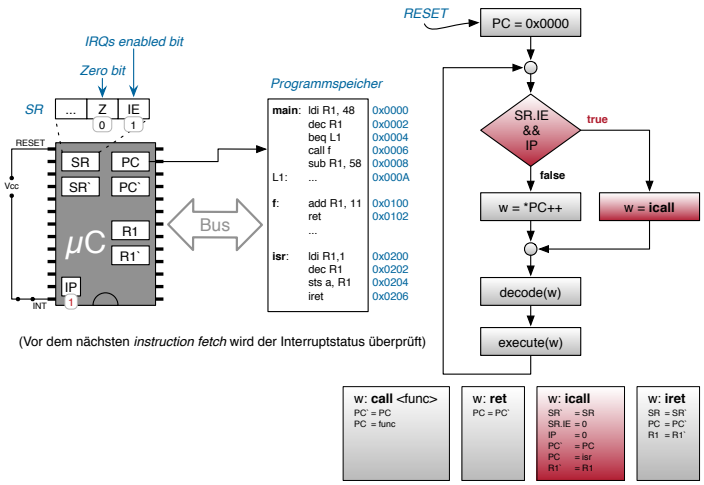
Beispielprogramm:

vereinfachter C-Code		Assembler-Code	
	uint8_t n;	10	
	uint8_t i;	11	
	...	...	
	r0 = n;	20	ld r0, 10
	if (r0 == 0) goto endif;	21	cmpi r0, 0
		22	beq 33
	r0 = 0;	23	ldi r0, 0
	i = r0;	24	st 11, r0
	goto test;	25	jmp 30
loop:	output();	26	call 70
	r0 = i;	27	ld r0, 11
	r0++;	28	inc r0
	i = r0;	29	st 11, r0
test:	r0 = i;	30	ld r0, 11
	if (r0 != 10) goto loop;	31	cmpi r0, 10
		32	bneq 26
endif:		33	
	...	...	...
	output(...)	70	...

Wie arbeitet ein Prozessor?



Ablauf eines Interrupts – Details



Der `volatile`-Typmodifizierer

- **Lösung:** Variable `volatile` (*„flüchtig, unbeständig“*) deklarieren
  - Compiler hält Variable nur so kurz wie möglich im Register
    - ⇝ Wert wird unmittelbar vor Verwendung gelesen
    - ⇝ Wert wird unmittelbar nach Veränderung zurückgeschrieben

```
// C code
#define PIND \
  (*(volatile uint8_t*) 0x10)
void foo(void) {
  ...
  if (! (PIND & 0x2)) {

    // button0 pressed
    ...
  }

  if (! (PIND & 0x4)) {

    // button 1 pressed
    ...
  }
}
```

```
// Resulting assembly code

foo:
  lds  r24, 0x0010 // PIND->r24
  sbrc r24, 1      // test bit 1
  rjmp L1
  // button0 pressed
  ...
L1:
  lds  r24, 0x0010 // PIND->r24
  sbrc r24, 2      // test bit 2
  rjmp L2
  ...
L2:
  ret
```

`PIND` ist `volatile` und wird deshalb vor dem Test erneut aus dem Speicher geladen.

Interrupts

Externe Interrupts: Verwendung (Forts.)

- **Schritt 2**: Konfigurieren der Interrupt-Steuerung
  - Steuerregister dem Wunsch entsprechend initialisieren
  - Unterstützung durch die `avrlibc`: Makros für Bit-Indizes (Modul `avr/interrupt.h` und `avr/io.h`)

```
...
void main(void) {
  DDRD  &= ~(1<<PD3);                 // PD3: input with pull-up
  PORTD |= (1<<PD3);
  EICRA &= ~(1<<ISC10 | 1<<ISC11);    // INT1: IRQ on level=low
  EIMSK |= (1<<INT1);                 // INT1: enable
  ...
  sei();                              // global IRQ enable
  ...
}
```

- **Schritt 3**: Interrupts global zulassen
  - Nach Abschluss der Geräteinitialisierung
  - Unterstützung durch die `avrlibc`: Befehl `sei()` (Modul `avr/interrupt.h`)

19-IRQ-Beispiel: 2020-05-15

Nebenläufigkeitsprobleme: *Lost-Wakeup*-Anomalie

```
void waitsec(uint8_t sec) {
  ...                  // setup timer
  sleep_enable();
  event = 0;
  while (! event) {
    sleep_cpu();
  }
  sleep_disable();
}
```

```
static volatile int8_t event;

// TIMER1 ISR
//   triggers when
//   waitsec() expires

ISR(TIMER1_COMPA_vect) {
  event = 1;
}
```

- Angenommen, an **dieser Stelle** tritt der Timer-IRQ (↯) auf
  - `waitsec` hat bereits festgestellt, dass `event` nicht gesetzt ist
  - `ISR` wird ausgeführt ⇝ `event` **wird gesetzt**
  - Obwohl `event` gesetzt ist, wird der Schlafzustand betreten
    ⇝ Falls kein weiterer IRQ kommt, **Dornröschenschlaf**

20-IRQ-Nebenlaeufigkeit: 2020-05-22

Zeiger, Felder und Zeichenketten (6)

- Um eine ganze Zeichenkette einem anderen char-Feld zuzuweisen, muss sie kopiert werden: Funktion `strcpy` in der Standard-C-Bibliothek
- Implementierungsbeispiele:

```
/* 1. Version */
void strcpy(char s[], char t[]) {
    int i = 0;
    while ((s[i] = t[i]) != '\0') {
        i++;
    }
}
```

```
/* 2. Version */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++, t++;
    }
}
```

```
/* 3. Version */
void strcpy(char *s, char *t) {
    while (*s++ = *t++) {
    }
}
```

21-Misc-Zeiger: 2020-05-22

Zeichenweises Lesen und Schreiben

- Lesen eines einzelnen Zeichens
  - von der Standardeingabe

```
#include <stdio.h>
int getchar(void);
```

  - aus einer Datei

```
#include <stdio.h>
int fgetc(FILE *fp);
```

  - lesen das nächste Zeichen
  - geben das Zeichen als `int`-Wert zurück
  - geben bei Eingabe von CTRL-D bzw. am Ende der Datei `EOF` als Ergebnis zurück
- Schreiben eines einzelnen Zeichens
  - auf die Standardausgabe

```
#include <stdio.h>
int putchar(int c);
```

  - in eine Datei

```
#include <stdio.h>
int fputc(int c, FILE *fp);
```

  - schreiben das Zeichen `c`
  - geben im Fehlerfall `EOF` als Ergebnis zurück

22-Misc-IO: 2020-05-22

Fehlerbehandlung

- Vorgehensweise:
  - Rückgabewert von Systemaufruf/Bibliotheksaufruf abfragen
  - Im Fehlerfall (häufig durch Rückgabewert -1 oder `NULL` angezeigt): Fehlercode steht in globaler Variablen `errno`
- Fehlermeldung kann mit der Funktion `perror` auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```

- Zwischenergebnisse auf Plausibilität überprüfen

```
#include <assert.h>
void assert(int condition);
```

Wenn Bedingung `condition` nicht „wahr" ist, wird das Programm mit Fehlermeldung abgebrochen.

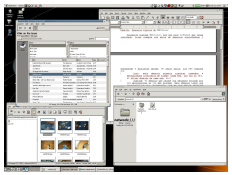
23-Misc-Fehler: 2020-05-22

Betriebssystem

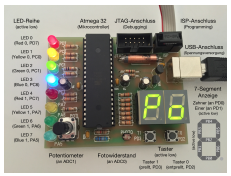
Überblick

Bisher:

- **Ein** Programm, das
- **alleine**,
- **beim Booten** gestartet
- mit **Hardware-Zugriffen**
- seine Umgebung steuert.



Quelle: www.wikipedia.org

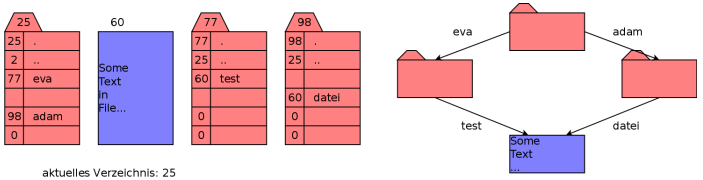


Jetzt:

- **Mehrere** Programme, die
- **nebenläufig**,
- **dynamisch** gestartet/beendet
- über **definierte E-/A-Funktionen**
- ihre Umgebung steuern.

Pfadnamen (3)

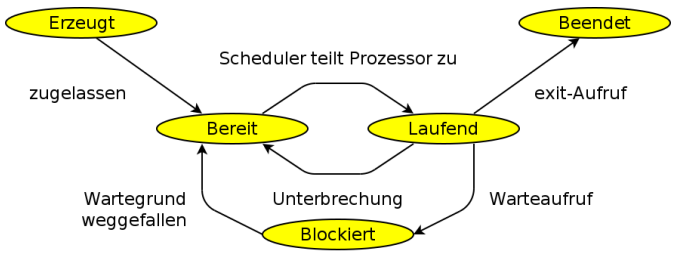
- Es können mehrere Verweise (**Hard Links**) auf eine Datei existieren:



- Beispiel Pfadauflösung „`adam/datei`”:
  - 25 + „`adam/datei`” ⇝ 98 + „`datei`”
  - 98 + „`datei`” ⇝ 60
- Beispiel Pfadauflösung „`eva/test`”:
  - 25 + „`eva/test`” ⇝ 77 + „`test`”
  - 77 + „`test`” ⇝ 60
- Datei wird gelöscht, wenn keine Verweise auf sie mehr existieren

Prozesszustände (2)

- Zustandsdiagramm mit Übergängen:



Nach Silberschatz, 1994

Nebenläufigkeitsbeispiel

```c
int main(void) {
  struct sigaction sa;
  struct itimerval it;
  /* Setup timer tick handler. */
  sa.sa_handler = tick;
  sa.sa_flags = 0;
  sigfillset(&sa.sa_mask);
  sigaction(SIGALRM, &sa, NULL);
  /* Setup timer. */
  it.it_value.tv_sec = 1;
  it.it_value.tv_usec = 0;
  it.it_interval.tv_sec = 1;
  it.it_interval.tv_usec = 0;
  setitimer(ITIMER_REAL, &it, NULL);
  /* Print time while working. */
  while (1) {                    ↓ hier Signal
    int s = sec, m = min, h = hour;
    printf("%02d:%02d:%02d\n", h, m, s);
    do_work();
  }
}
```

```c
volatile int hour = 0;
volatile int min = 0;
volatile int sec = 0;

static void tick(int sig) {
  sec++;
  if (60 <= sec) {
    sec = 0; min++;
  }
  if (60 <= min) {
    min = 0; hour++;
  }
  if (24 <= hour) {
    hour = 0;
  }
}
```

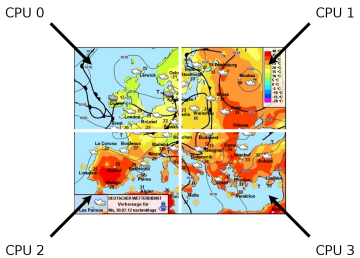
```
~> ./test
...
23:59:59
00:59:59   ← hier Problem!
00:00:00
...
```

(Fehlerbehandlung weggelassen...)

Multiprozessor-Systeme

Beispiel: Berechnung einer Wetterkarte (2)

- Z.B. Berechnung der Wetterkarte aufgeteilt auf 4 Prozessoren:



- Alle Prozessoren greifen auf einen gemeinsamen Speicher zu, in dem das Ergebnis berechnet wird.

Gegenseitiger Ausschluss (Mutual Exclusion)

- Einfache Implementierung durch **mutex**-Variablen

```
volatile int m = 0; /* 0: free; 1: locked */
volatile int counter = 0;
```

```
...       /* Thread 1 */
lock(&m);
counter++;
unlock(&m);
...
```

```
...       /* Thread 2 */
lock(&m);
printf("%d\n", counter);
counter = 0;
unlock(&m);
...
```

- Nur der Faden, der `lock` aufgerufen hat, darf `unlock` aufrufen!
- Realisierung (nur konzeptionell!)

```
void lock(volatile int *m) {
    while (*m == 1) {
        /* Wait... */
    }
    *m = 1;
}
```

```
void unlock(volatile int *m) {
    *m = 0;
}
```

`lock` (und ggf. `unlock`) müssen **atomar** ausgeführt werden!

Speicherorganisation

Dynamische Speicherallokation: Heap

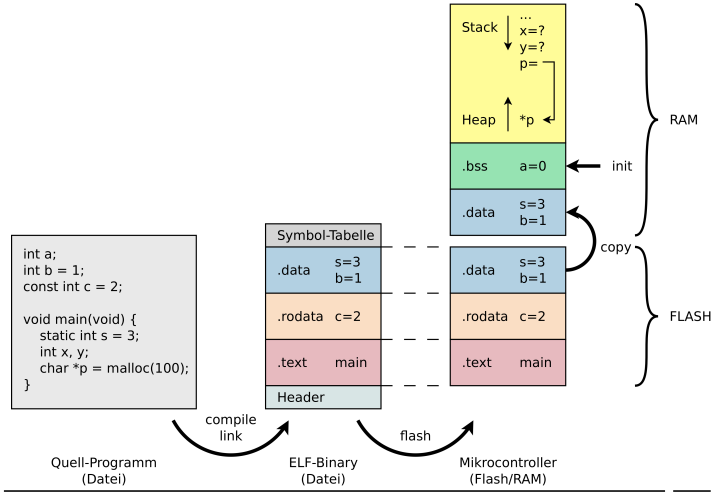
- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
  - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
  - `void *malloc(size_t n)` fordert einen Speicherblock der Größe *n* an; Rückgabe bei Fehler: `NULL`-Zeiger
  - `void free(void *pmem)` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```c
#include <stdlib.h>

int *intArray(size_t n) { /* alloc int[n] array */
    return (int *) malloc(n * sizeof int);
}

void main(void) {
    int *array = intArray(100); /* alloc memory for 100 ints */
    if (array == NULL) { /* error handling... */ }
    ...
    array[99] = 4711; /* use array */
    ...
    free(array); /* free allocated block (** IMPORTANT! **) */
}
```

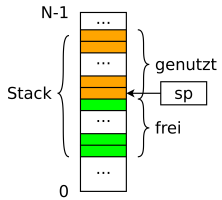

Speicherorganisation auf einem μC



Dynamische Speicherallokation – Stack

- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem Stack (Stapel, Keller) verwaltet
- Stack ist Teil des normalen Hauptspeichers
- Prozessorregister `sp` „**Stack Pointer**" zeigt immer auf das zuletzt abgelegte Datum (architekturabhängig)
- Stack „wächst" „von oben nach unten" (architekturabhängig)

=> `sp` zeigt damit immer auf den Anfang des genutzten Teil des Stacks



Statische versus dynamische Allokation

- Bei der **μC-Entwicklung** wird statische Allokation bevorzugt
  - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
  - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ 1–4)

```
~> size sections.avr
text    data     bss     dec     hex filename
682       10       6     698     2ba sections.avr
```

Sektionsgrößen des Programms von ↪ 34–1

- ⇝ Speicher möglichst durch `static`-Variablen anfordern
  - Regel der geringstmöglichen Sichtbarkeit beachten ↪ 12–6
  - Regel der geringstmöglichen Lebensdauer „sinnvoll" anwenden
- Ein Heap ist verhältnismäßig teuer ⇝ wird möglichst vermieden
  - Zusätzliche Speicherkosten durch Verwaltungsstrukturen und Code
  - Speicherbedarf zur Laufzeit schlecht abschätzbar
  - Risiko von Programmierfehlern und Speicherlecks

Dank

Danke!

an alle

- die mitgemacht haben
- die konstruktive Kritik geäußert haben
- Übungsleiter
- Tutoren

für den speziellen Einsatz in Corona-Zeiten!

Viel Erfolg für das weitere Studium!