

# Systemnahe Programmierung in C (SPiC)

## 29 Signale

**Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

Sommersemester 2022

<http://sys.cs.fau.de/lehre/SS22/spic>



- Mikrocontroller kann auf nebenläufige Ereignisse (Interrupts) mit Interrupt-Service-Routinen reagieren.
- Ähnliches Konzept auf Prozess-Ebene: **Signale**



**Interrupt:** asynchrones Signal aufgrund eines „externen“ Ereignisses

- CTRL-C auf der Tastatur gedrückt
- Timer abgelaufen
- Kind-Prozess terminiert
- ...

**Exception:** synchrones Signal, ausgelöst durch die Aktivität des Prozesses

- Zugriff auf ungültige Speicheradresse
- Illegaler Maschinenbefehl
- Division durch 0
- Schreiben auf eine geschlossene Kommunikationsverbindung
- ...

**Kommunikation:** ein Prozess will einem anderen ein Ereignis signalisieren



# Signale (3)

## CTRL-C:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
^C
~>
```

## Inter-Prozess-Kommunikation:

```
int main(void)
{
    while (1) {
    }
}
```

```
~> ./test
Terminated
~>
```

## Illegaler Speicherzugriff:

```
int main(void)
{
    *(int *) NULL = 0;
    return 0;
}
```

```
~> ./test
Segmentation fault
~>
```



## abort:

erzeugt Core-Dump (Speicher- und Registerinhalte werden in Datei `./core` geschrieben) und beendet Prozess  
Standardeinstellung für alle Exceptions (zum nachträglichen Debuggen)

## exit:

beendet Prozess (ohne Core-Dump)  
Standardeinstellung für z.B. CTRL-C, Kill-Signal

## ignore:

Signal wird ignoriert  
Standardeinstellung für alle „unwichtigen“ Signale (z.B. Kindprozess terminiert, Größe des Terminal-Fensters hat sich geändert)

...



## Reaktion auf Signale (2)

---

...

### handler:

Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

nie Standardeinstellung, da Programm-abhängig

### stop:

stoppt Prozess

Standardeinstellung für Stop-Signal

### continue:

setzt Prozess fort

Standardeinstellung für Continue-Signal

Reaktion über System-Aufruf (`sigaction`) änderbar



- Einstellen der Signalbehandlungsfunktion  
(entspricht dem Setzen der ISR-Funktion)

```
#include <signal.h>
```

```
int sigaction(int sig, struct sigaction *new, struct sigaction *old);
```

struct sigaction enthält:

```
void (*sa_handler)(int sig); /* handler function  
                             or SIG_DFL or SIG_IGN */  
sigset_t sa_mask;           /* list of blocked signals while  
                             handler is executed */  
int sa_flags;               /* 0 or SA_RESTART ... */
```

- ...



- ...
- Blockieren/Freigeben von Signalen  
(entspricht `cli()`, `sei()`)

```
#include <signal.h>

int sigprocmask(int how, sigset_t *nmask, sigset_t *omask);
```

- `SIG_BLOCK`: angegebene Signale blockieren
- `SIG_UNBLOCK`: angegebene Signale deblockieren
- `SIG_SETMASK`: Signalmaske setzen
- Freigeben + Passives Warten auf Signal + wieder Blockieren  
(entspricht `sei()`; `sleep_cpu()`; `cli()`;) )

```
#include <signal.h>

int sigsuspend(sigset_t *mask);
```

...





# Programmierschnittstelle

- ...
- Erstellen einer leeren Signal-Liste

```
#include <signal.h>
int sigemptyset(sigset_t *mask);
```

- Erstellen einer vollen Signal-Liste

```
int sigfillset(sigset_t *mask);
```

- Hinzufügen eines Signals zu einer Signal-Liste

```
int sigaddset(sigset_t *mask, int sig);
```

- Entfernen eines Signals aus einer Signal-Liste

```
int sigdelset(sigset_t *mask, int sig);
```



- Typische Signale:
  - SIGSEGV: „Segmentation Fault“ (ungültiger Speicherzugriff)
  - SIGINT: „Interrupt“ (CTRL-C)
  - SIGALRM: „Alarm“ (Timer abgelaufen)
  - SIGCHLD: „Child“ (Kindprozess terminiert)
  - SIGTERM: „Terminate“ (Abbruch des Prozesses; abfangbar)
  - SIGKILL: „Kill“ (Abbruch des Prozesses; nicht abfangbar)



# Signal-Beispiel 1

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
    // Call handler when
    // CTRL-C signal is received.
    struct sigaction sa;
    sa.sa_handler = handler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    for (int i = 0; ; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

```
static void handler(int sig)
{
    char s[] = "CTRL-C!\n";
    write(STDOUT_FILENO,
          s, strlen(s));
}
```

(Fehlerbehandlung  
weggelassen...)

```
~> ./test
...
146431
146432
146433
14^CCTRL-C!
6434
146435
146436
...
~>
```



## Signal-Beispiel 2

```
int main(void)
{
    // Call handler when
    // timer signal is received.
    struct sigaction sa;
    sa.sa_handler = handler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, NULL);

    // Send timer signal every sec.
    struct itimerval it;
    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);

    // Wait for timer ticks.
    sigset_t mask;
    sigemptyset(&mask);
    while (1) sigsuspend(&mask);
}
```

```
static void handler(int sig)
{
    write(STDOUT_FILENO,
         "Tick\n", 5);
}
```

(Fehlerbehandlung  
weggelassen...)

```
~> ./test
Tick
Tick
Tick
^C
~>
```



# Signal-Beispiel 3

```
#include <signal.h>
#include <unistd.h>

int main(void)
{
    // Call handler when
    // I/O signal is received.
    struct sigaction sa;
    sa.sa_handler = handler;
    sigfillset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGIO, &sa, NULL);

    // Send I/O signal when
    // STDIN can be read.
    int flags = fcntl(STDIN_FILENO,
                     F_GETFL);
    flags |= O_ASYNC;
    fcntl(STDIN_FILENO, F_SETFL,
          flags);

    while (1) sleep(1);
}
```

```
static void handler(int sig)
{
    char buf[256];
    int len;

    // Read chars from STDIN.
    len = read(STDIN_FILENO, buf,
              sizeof(buf));

    // Handle chars in buf.
    ...
}
```

(Fehlerbehandlung  
weggelassen...)



- Signale erzeugen Nebenläufigkeit innerhalb von Prozessen
- resultierende Probleme völlig analog zu Nebenläufigkeit bei Interrupts auf einem Mikrocontroller



# Nebenläufigkeitsbeispiel

```
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) {
        int s = sec, m = min, h = hour;
        printf("%02d:%02d:%02d\n", h, m, s);
        do_work();
    }
}
```

↓ hier Signal

(Fehlerbehandlung weggelassen...)

```
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
00:00:00
...
```

← hier Problem!



# Lösungen Nebenläufigkeitsbeispiel

## 1. Lösung

```
sigset_t nmask, omask;

/* Block SIGALRM. */
sigemptyset(&nmask);
sigaddset(&nmask, SIGALRM);
sigprocmask(SIG_BLOCK,
            &nmask, &omask);

/* Get current time. */
int s = sec, m = min, h = hour;

/* Restore signal mask. */
sigprocmask(SIG_SETMASK,
            &omask, NULL);

/* Print current time. */
printf("%02d:%02d:%02d\n",
        h, m, s);
```

## 2. Lösung

```
/* Get current time. */
int s, m, h;
do {
    s = sec;
    m = min;
    h = hour;
} while (s != sec
        || m != min
        || h != hour);

/* Print current time. */
printf("%02d:%02d:%02d\n",
        h, m, s);
```

Weitere Lösungen existieren...





# Nebenläufigkeitsprobleme

- Zusätzliches Problem:  
interne Funktionsweise von Bibliotheksfunktionen i.A. unbekannt
- Beispiel 1:  
`printf` fügt Zeichen in Puffer ein  
=> Nutzung von `printf` im Hauptprogramm *und* in  
Signal-Behandlungsfunktion u.U. gefährlich
- Beispiel 2:  
`malloc` durchsucht Liste nach freiem Speicherbereich; `free` fügt  
Block in Liste ein  
=> Nutzung von `malloc/free` im Hauptprogramm *und* in  
Signal-Behandlungsfunktion u.U. gefährlich
- Lösung:
  - Signale während der Ausführung kritischer Bereiche blockieren oder
  - keine unbekanntes Bibliotheksfunktionen aus  
Signal-Behandlungsfunktionen heraus aufrufen

