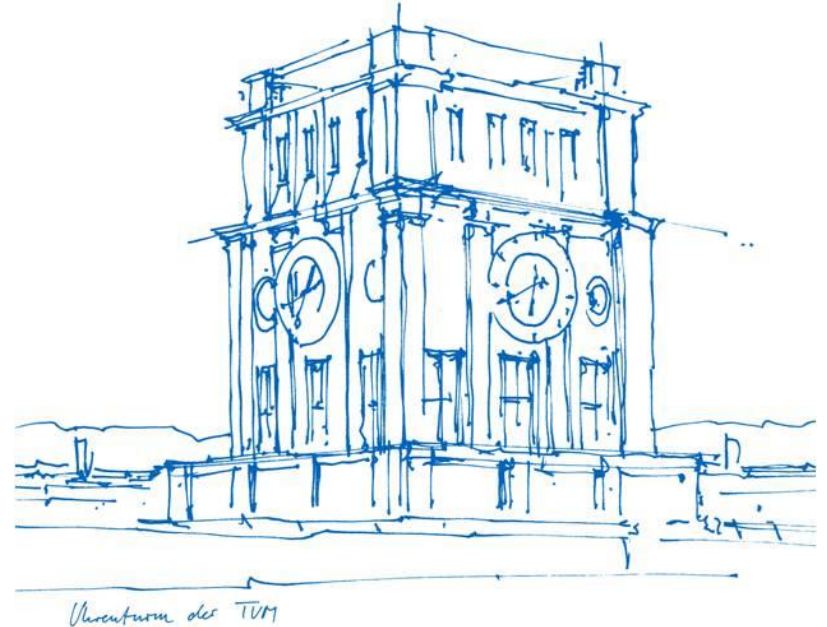


# Grundlagenpraktikum: Rechnerarchitektur

SoSe 2024

~ *Danial Arbabi*

[danial.arbabi@tum.de](mailto:danial.arbabi@tum.de)



# Zulip-Gruppen

Gruppe 29

FR 12:00



<https://zulip.in.tum.de/#narrow/stream/2276-GRA-Tutorium---Gruppe-29>

Gruppe 32

FR 15:00



<https://zulip.in.tum.de/#narrow/stream/2279-GRA-Tutorium---Gruppe-32>

# Tutoriums-Website



<https://home.in.tum.de/~arb/>

Disclaimer:

*Dies sind keine offiziellen  
Materialien, somit besteht keine  
Garantie auf Korrektheit und  
Vollständigkeit.*

*Falls euch Fehler auffallen, bitte  
gerne melden.*

# Tutoriumsumfrage



<https://forms.gle/EMw5pzybhN69SL9b6>

# Commandline Parsing mit getopt()

```
int getopt(int argc, char *argv[], const char *optstring);
```

- argc und argv aus der main-Funktion
- const char \*optstring: Alle unterstützten Optionen
  - : => Argument **benötigt** (befindet sich in globaler Variable **optarg**)
  - :: => Argument **optional** (falls vorhanden in globaler Variable **optarg**, sonst 0)
  - <nichts> => **Kein** Argument
- Beispiele:
  - hta:b:c::
  - -h und -t
  - a 1.0 oder a1.0 und b5
  - c5 oder c

# Commandline Parsing mit getopt()

```
int getopt(int argc, char *argv[], const char *optstring);
```

- Rückgabewert:
  - Nächster options-Character
  - -1 bei Ende der Optionen
  - ? (oder : s. manpage) falls Fehler und printet error message auf stderr
- Wenn fertig:
  - Globale Variable optind auf Index des ersten nicht-option (positional) Arguments in argv gesetzt
- getopt\_long() (s. Manpage)

# Umwandlung von Strings zu Zahlen

- `strtol()`
- `strtoul()`
- `strtoull()`
- `strtof()`
- `strtod()`
- KEIN `atoi()` und `atof()` benutzen, da keine Fehlererkennung;

```
long strtol(const char *ptr, char **endptr, int base);  
double strtod(const char *ptr, char **endptr);
```

- `const char *ptr`: zu konvertierender String
- `char **endptr`: Adresse eines Pointers, der auf den fehlerhaften Teil des ptr Strings zeigen wird
- `int base`: Zahlenbasis

# Umwandlung von Strings zu Zahlen

Beispiel:

```
int a; // Zu ändernde Variable
errno = 0; // Errno für Fehlerbehandlung
char* endptr; // Endptr für Fehlerbehandlung

a = strtol(currentOpt, &endptr, 10);

if (*endptr != '\0' || errno) {
    fprintf(stderr, "Invalid number\n");
    /* Weitere Fehlerbehandlung bzgl errno Wert*/
    return EXIT_FAILURE;
}
```



# File-I/O

## fopen

- Öffnen von Files:

```
FILE *fopen(const char *pathname, const char *mode);
```

- `const char *pathname`: Pfad der Datei
- `const char *mode`: Berechtigungen, mit denen die Datei geöffnet wird
  - ☐ `O_RDONLY`
  - ☐ `O_WRONLY`
  - ☐ `O_RDWR`
  - ☐ ...

Rückgabe: Pointer auf FILE-Struktur

# File-I/O

## fclose

- Schließen von Files:

```
int fclose(FILE *stream);
```

- FILE \*stream: Filestruktur

Rückgabe: 0 wenn erfolgreich, sonst EOF und errno gesetzt

# File-I/O

## fread

- Lesen von Files:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- void \*ptr: Speicherbereich, in dem Fileinhalt gespeichert wird
- size\_t size: Größe der zu lesenden Items in Bytes
- size\_t nmemb: Anzahl der zu lesenden Items
- FILE \*stream: File, aus der gelesen werden soll

Rückgabe: Anzahl der gelesenen Items

# File-I/O

## fwrite

- Schreiben von Files:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Const void \*ptr: Pointer auf einen konstanten Speicherbereich, aus dem gelesen wird
- size\_t size: Größe der zu schreibenden Items in Bytes
- size\_t nmemb: Anzahl der zu lesenden Items
- FILE \*stream: File, die beschrieben wird

Rückgabe: Anzahl gelesener Items

# File-I/O

## fstat

- Erhalten von File Informationen:

```
int fstat(int fd, struct stat *statbuf);
```

- `int fd`: Filedeskriptor (bekommt man mit `fileno(FILE*)`)
- `struct stat *statbuf`: Pointer auf stat Struktur, worin die Information geschrieben wird

Rückgabe: 0 wenn erfolgreich, ansonsten -1 (errno wird gesetzt)

# Makefile

## Was passiert in den beiden Zeilen?

```
main: main.c xor_cipher.c
    $(CC) $(CFLAGS) -o $@ $^
```

- Ein solches Konstrukt heißt „Rule“

target: prerequisite1 prerequisite2 ... (prerequisite kann auch ein weiteres Target sein)

    recipe1

    recipe2

- Quelldateien: main.c und xor\_cipher.c
- Shell-Befehle: \$(CC) \$(CFLAGS) -o \$@ \$^ (kann z.B. auch „echo Hallo“ sein)
- Output-Name: main

# Makefile

Womit wird nun `$@` und `$$` ersetzt?

```
main: main.c xor_cipher.c
    $(CC) $(CFLAGS) -o $@ $$
```

- `$@` = The file name of the target of the rule
- `$$` = The names of all the prerequisites, with spaces between them

# Module in SystemC

Stichworte: Konstruktor, Sensitivity-Lists, behaviour

```
1 SC_MODULE(M1) {  
2     sc_signal<bool> x;  
3     sc_signal<bool> y;  
4     sc_signal<bool> output;  
5  
6     SC_CTOR(M1) {  
7         SC_THREAD(bhaviour);  
8         sensitive << x << y;  
9     }  
10  
11     void behaviour() {  
12         while (true) {  
13             output = x.read() | (!x.read() & y.read());  
14             wait();  
15         }  
16     }  
17 };
```



# Module in SystemC

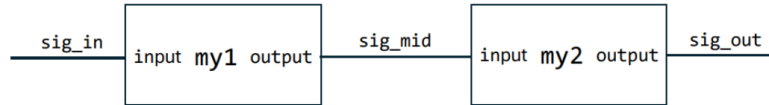
Stichworte: Ports (input, output)

```
1 SC_MODULE(MyModule) {  
2     sc_in<bool> input;  
3     sc_out<bool> output;  
4  
5     SC_CTOR(MyModule) {  
6         SC_THREAD(behaviour);  
7     }  
8  
9     void behaviour() {  
10         while (true) {  
11             output->write(!input->read());  
12             wait();  
13         }  
14     }  
15 };
```

- Lesen vom Input
- Schreiben auf den Output

# Module in SystemC

## Kommunikation zwischen Modulen (Kabel verbinden)



```
1 MyModule my1("my1");
2 MyModule my2("my2");
3 sc_signal<bool> sig_in, sig_mid, sig_out;
4 // Die Tatsächlichen Signale werden außerhalb der Module
  erstellt.
5
6 my1.input.bind(sig_in);
7 my1.output.bind(sig_mid);
8 // port.bind(signal) weist einem Port ein Signal zu.
9
10 my2.input(sig_mid);
11 my2.output(sig_out);
12 // Alternative Schreibweise: port(signal).
```

- Schreiben / Lesen von Signalen
- Verbinden dieser Signale mit den Ports

# Clocks in SystemC

- Best Practice: Wir erstellen eine Clock in `sc_main` und binden sie an Input Ports in Modulen.

```
1 SC_MODULE(MY_MODULE) {  
2     sc_in<bool> clk;  
3     SC_CTOR(MY_MODULE) { }  
4 };  
5  
6 int sc_main(int argc, char* argv[]) {  
7     sc_clock clk("clk", 2, SC_SEC);  
8  
9     MY_MODULE my_module("my_module");  
10    my_module.clk(clk);  
11  
12    sc_start(10, SC_SEC);  
13    return 0;  
14 }
```

# Clocks in SystemC

## SC\_CTHREAD

Nutzen der Clock mit SC\_CTHREAD():

```
1 SC_MODULE(MY_MODULE) {  
2     sc_in<bool> clk;  
3  
4     SC_CTOR(MY_MODULE) {  
5         SC_CTHREAD(behaviour, clk.pos());  
6     }  
7  
8     void behaviour() { ... }  
9 };
```

# Clocks in SystemC

## SC\_CTHREAD

SC\_CTHREAD beginnt erst ab erster Flanke

- ▶ SC\_CTHREAD(behaviour, clk.pos())
  - ▶ sc\_in.pos(): Event für steigende Flanke.
  - ▶ sc\_in.neg(): Event für sinkende Flanke.
  - ▶ sc\_in.value\_changed(): Event für jeden Flankenwechsel.

```
1 ... // SC_CTHREAD(behaviour, clk.pos())
2 void behaviour() {
3     while(true) {
4         wait();
5         std::cout << sc_time_stamp() << std::endl;
6     }
7 }
8 // 2 s
9 // 4 s
10 // 6 s
11 // 8 s
```

# Clocks in SystemC

## SC\_THREAD und SC\_METHOD

SC\_THREAD beginnt sofort → wait() am Anfang

- ▶ SC\_METHOD und SC\_THREAD bieten Möglichkeiten, die Prozesse bei steigender Flanke eines `sc_in<bool>` auszuführen.
- ▶ SC\_METHOD:

```
1 void behaviour() {  
2     std::cout << sc_time_stamp() << std::endl;  
3     next_trigger(clk.posedge_event());  
4 }
```

- ▶ SC\_THREAD:

```
1 ...  
2 SC_THREAD(behaviour);  
3 sensitive << clk.pos();
```

# Datenspeicherung in SystemC Modulen

```
1 SC_MODULE(MY_STORAGE_MODULE) {  
2     sc_in<int> address;  
3     sc_in<int> value;  
4     sc_in<bool> clk;  
5     int storage[256];  
6  
7     ...  
8  
9     void behaviour() {  
10         while(true) {  
11             wait();  
12             storage[address->read()] = value->read();  
13         }  
14     }  
15 };
```

# Tracefile

## Erstellen

1. Erstellen der Tracefile (Pfad bzw. Name vom User über Commandline übergeben) und zuweisen auf eine Pointervariable `trace_file`

```
sc_trace_file* trace_file = sc_create_vcd_trace_file(argv[1]);
```

2. Beobachten einer Variablen `var` in `trace_file` und nenne sie beliebig (z.B. „var“)

```
sc_trace(trace_file, var, „var“);
```

3. Starten der Simulation für `time` Sekunden

```
sc_start(time, SC_SEC);
```

4. Schließen der `trace_file`

```
sc_close_vcd_trace_file(trace_file);
```



# Trace File

## I/O-Analyse

```
1 SC_MODULE(MY_MODULE) {  
2     int reads;  
3     int writes;  
4     ...  
5  
6     void behaviour() {  
7         while (true) {  
8             wait();  
9             bool result = a->read() ^ b->read();  
10            reads += 2;  
11            out->write(result);  
12            writes++;  
13        }  
14    }  
15 };
```

# T8 - SystemC und C

## Anleitung zum Projektaufbau

### ■ main.c:

- ☐ Commandline Argumente parsen (mit Fehlerbehandlung) und Simulationsfunktion aus `modules.cpp` aufrufen
- ☐ Alle Funktionen, die von SystemC/C++ aufgerufen werden sollen, deklarieren  
`extern <typ> <funktion>(<parameter>);`

### ■ modules.hpp:

- ☐ Module definieren (wie wir es bereits kennen)
- ☐ Alle Funktionen, die von C aus verwendet werden sollen, deklarieren:  
`extern "C" <typ> <funktion>(<parameter>);`

### ■ modules.cpp:

- ☐ `sc_main()` mit Fehlercode return hinschreiben (wird aber nicht aufgerufen)
- ☐ **Simulationsfunktion/Mainfunktion** des Projekts, die in `modules.hpp` deklariert wurde, definieren:  
Module initialisieren, Signale setze, Tracefiles erstellen, Simulation starten, return des Ergebnisses, etc.

## Umfrage zum Tutorium



<https://kurzelinks.de/em4f>

**Viel Erfolg bei euren  
Projekten!**