

# Echtzeitverarbeitung

R. Kaiser, K. Beckmann, R. Kröger

(HTTP: <http://www.cs.hs-rm.de/~kaiser>

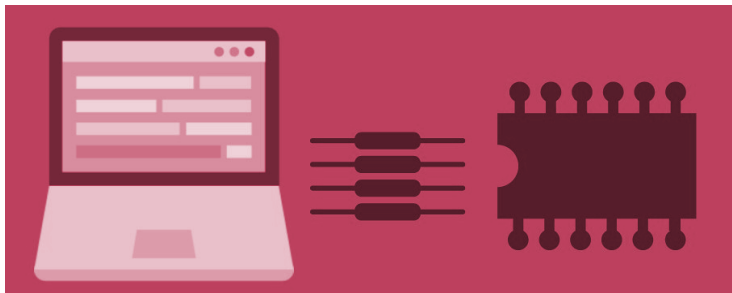
E-Mail: [robert.kaiser@hs-rm.de](mailto:robert.kaiser@hs-rm.de))

Sommersemester 2022

### 3. C-Programmierung eingebetteter Systeme



Hochschule RheinMain



<https://freevideolectures.com/course/3624/embedded-systems-programming>

# Inhalt



## 3. C-Programmierung eingebetteter Systeme

### 3.1 Cross-Entwicklung

### 3.2 Hardwarenahes Programmieren in C

# 3.1 Cross-Entwicklung



[https://blog.ergodirekt.de/index.php?aam\\_media=13065&size=full](https://blog.ergodirekt.de/index.php?aam_media=13065&size=full)

# Cross-Entwicklung

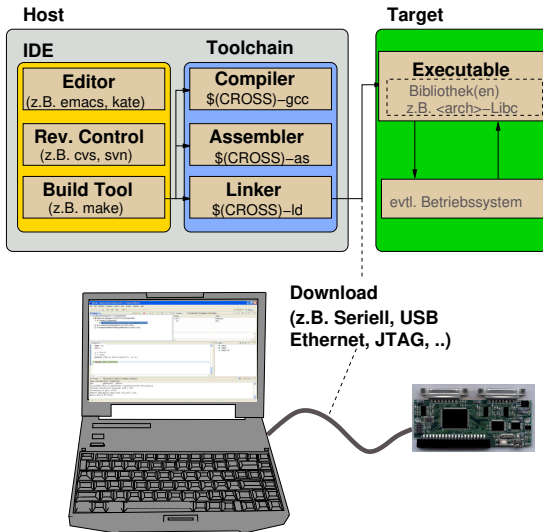


- Bisher bekannt: *self-hosted*-Entwicklung: Programme werden auf dem Rechner entwickelt, auf dem sie ausgeführt werden
- Eingebettete Systeme verfügen i.d.R. nicht über geeignete / genügende Betriebsmittel für eine Entwicklungsumgebung:
  - ▶ Speicher
  - ▶ Dateisystem
  - ▶ Ein-/Ausgabe

## → Cross-Entwicklung:

- Entwicklungswerkzeuge (IDE, Compiler, Linker) auf einem Entwicklungsrechner (*Host*)
- Generieren Code für ein Zielsystem (*Target*)
- Ausführbarer Code (*executable*) wird über geeignete Schnittstellen ins Zielsystem geladen und dort ausgeführt
- Host and Target können (müssen aber nicht) unterschiedliche Prozessorarchitekturen besitzen

# Cross-Entwicklung



# Cross-Entwicklung: Entwicklungsumgebung



- Entwicklungsumgebung: Sammlung von Programmen zum Erzeugen und Verwalten von Quellcode
  - ▶ Editor, Build-Tool, Revisionskontrolle, Handbuch, etc.
  - ▶ **Nicht** Target-spezifisch
  - ▶ Z.T. mit grafischer Benutzerschnittstelle
- Beispiele:
  - ▶ Editoren: emacs, vi, vim, kate, notepad, ...
  - ▶ Build-Tools: make, nmake, qmake, ...
  - ▶ Revisionskontrolle: Git, Mercurial, Subversion, CVS, RCS, SCCS,
- IDE: Integrierte Entwicklungsumgebung: (grafische) Benutzeroberfläche für die Programme der Entwicklungsumgebung
- Beispiele: Eclipse, Source Navigator, kscope, KDevelop, ...

# Cross-Entwicklung: Toolchain



- *Cross-Toolchain*: Programme zum Erzeugen, Bearbeiten und Analysieren von ausführbarem Code:
  - ▶ Target-spezifischer Teil der Entwicklungsumgebung
  - ▶ I.d.R. Kommandozeilen-Programme
  - ▶ Aufruf aus bzw. Anpassung an IDE (s.o.) ggf. über „Plugins“

- Beispiel: GNU ARM Toolchain:

<code>arm-linux-gnueabi-cpp</code>	C-Präprozessor
<code>arm-linux-gnueabi-gcc</code>	C-Compiler
<code>arm-linux-gnueabi-g++</code>	C++-Compiler
<code>arm-linux-gnueabi-as</code>	Assembler
<code>arm-linux-gnueabi-ld</code>	Linker
<code>arm-linux-gnueabi-ar</code>	Archiver/Librarian
<code>arm-linux-gnueabi-nm</code>	Symbole anzeigen
<code>arm-linux-gnueabi-objdump</code>	Inhalte der Sektionen anzeigen
<code>arm-linux-gnueabi-objcopy</code>	Binärformate konvertieren
<code>arm-linux-gnueabi-size</code>	Größen von Sektionen anzeigen
<code>arm-linux-gnueabi-strip</code>	Symbolinformationen entfernen
<code>arm-linux-gnueabi-gcov</code>	Coverage-Analyse
...	...



# Cross-Entwicklung: Plattformabhängigkeiten



- Bei GNU allgemein üblich: <Plattform>-<Programmname> (z.B.: avr-gcc, ppc\_60x-gcc, arm-elf-gcc, ...)

**Tipp: Toolchain-Anpassung in Makefile z.B. so:**

```
CROSS =  
CC = $(CROSS) gcc
```

- Bei Aufruf mit „make CROSS=arm-linux-gnueabi-“ wird arm-linux-gnueabi-gcc verwendet (sonst: gcc)

**Maschinenspezifischer Code in C z.B. so:**

```
#ifdef __ARMEL__  
... ARMEL-spezifischer Code ...  
#endif
```

- (Tipp: Anzeigen der vordefinierten Präprozessorkonstanten:  
touch empty.c; <Plattform>-gcc -E -dM empty.c)

# Cross-Entwicklung: Download



- Ziel: Ausführbares Programm (Executable) ins Target bringen
- Je nach Zielsystem unterschiedlichste Vorgehensweisen, z.B.:
  - ▶ Einspeichern in ein (E)(E)PROM mit Hilfe eines (externen) Programmiergerätes → Speicher muss zum Programmieren aus- und wiedereingebaut werden
  - ▶ Einspeichern in Flash-Speicher über in-System-Programmer- (ISP-) Schnittstelle → Schnittstellenspezifisches Programmier-Tool (z.B. `stlink`, `avrdude`) erforderlich
  - ▶ Laden des Codes über serielle Schnittstelle oder Netzwerk → erfordert ein *Bootloader*-Programm auf der Target-Seite
  - ▶ Laden des Programmcodes von einem Massenspeicher (z.B. CompactFlash oder USB-Stick) → erfordert ebenfalls einen *Bootloader* auf der Target-Seite
- Typische Dateiformate für Executables: ELF (`.elf`), Intel-Hex (`.hex`), S-Record (`.srec`, `.sr`), (Raw) Binary (`.bin`)
- Umwandeln zwischen diesen Formaten z.B. mit `objcopy`

# Debugging



- Eigentlich: engl. *Bug* = Käfer, Insekt, Wanze, Laus
- In der Informationstechnologie: *Bug* = Programmierfehler

→ *Debugging*: Finden und Entfernen von Fehlern

- *Debugger*: Hilfsprogramm, das die Fehlersuche ermöglicht bzw. erleichtert
- Ausführen des (evtl. fehlerhaften) Programmes unter der Kontrolle des Debuggers
  - ▶ Laden und Starten des Programmes
  - ▶ Ausführung –u.U. bedingt– anhalten (Breakpoint) und fortführen
  - ▶ Schrittweise Programm-Abarbeitung (Single-Step)
  - ▶ Variablenwerte und Registerinhalte beobachten und manipulieren
  - ▶ Aufrufhierarchie anzeigen
- GNU-Debugger gdb: Bedienung über Kommandozeile
- Verschiedene grafische Frontends: DDD, Insight, Kgdb, Eclipse, Nemiver

# Beispiel: GDB-Frontend *DDD*



The screenshot shows the DDD interface with the following components labeled:

- Steuerung**: Points to the top menu bar (File, Edit, View, Program, Commands, Status, Source, Data).
- Display-Fenster**: Points to the main display area showing the current thread information: `3: thr (struct thread *) 0x80054e`.
- Quellcode-Fenster**: Points to the source code window showing C code with comments in German. The current line is `if(thr != (struct thread *) 0x80054e)`.
- Aktuelle Position**: Points to the current execution position in the source code, indicated by a green arrow.
- Assembler Fenster**: Points to the assembly code window showing the dump of assembler code for the current instruction.
- Benutzerdef. Buttons**: Points to the user-defined buttons at the bottom of the source code window: `attach detach hex dec`.
- Kommando-Fenster**: Points to the command window at the bottom showing the GDB session output: `(gdb) finish`, `0x0000102a in wx_thread_block (handle=0x80054e, ticks=0x32) at winzix.c:411`, and `(gdb)`.
- Menüleiste**: Points to the top menu bar.
- Funktions-Buttons**: Points to the function buttons in the top toolbar (Run, Interrupt, Step, etc.).
- Aufruf-hierarchie**: Points to the **DDD: Backtrace** window showing the call stack:
  - #2 0x0000b200 in ?? ()
  - #1 0x00000938 in blinkthread () at
  - #0 0x0000102a in wx\_thread\_block
- Aktiv-Anzeige**: Points to the status bar at the bottom right, which indicates when the program is running.

# Cross-Debugging



- Funktionsweise eines Debuggers: „Fernsteuern“ der Programmausführung
- Dazu nötige Basisfunktionen:
  - ▶ Prozessor anhalten / weiterlaufen lassen
  - ▶ Register des (angehaltenen) Prozessors lesen / schreiben
  - ▶ Daten- **und** Programmspeicher Lesen **und** Schreiben  
(Programmspeicher-Schreibzugriff ist zum Setzen von Breakpoints erforderlich)
  - ▶ Ausnahmebedingungen (z.B. Speicherzugriffsfehler, Nulldivision, Ungültiger Maschinenbefehl) abfangen und Prozessor anhalten
- *self-hosted* Debugger erreichen dies über spezielle Betriebssystemfunktionen (z.B. Linux: `ptrace(2)`)
- Ein *cross*-Debugger benötigt dazu einen (i.d.R. externen) „Debug-Server“

# Debug-Server



- Allgemein: Client-Server Architektur:

- ▶ **Server:** Bietet Dienste (hier: Debug-Basisfunktionen)
- ▶ **Client:** Fordert Dienste an, wartet auf Ergebnis
- ▶ Nachrichtenbasierte Kommunikation: Übermitteln von Anforderungen / Ergebnissen in Form von Bytesequenzen

⇒ Zwischen Debugger und Debug-Server muss lediglich ein bidirektionaler, sequenzieller Kommunikationskanal existieren, z.B.:

- Seriell (RS-232)
- Netzwerk (UDP- oder TCP/IP-Socket)
- Logisch (UNIX Domain Socket, UNIX-Pipe, Pseudo-TTY)

⇒ Debugger und Debug-Server können...

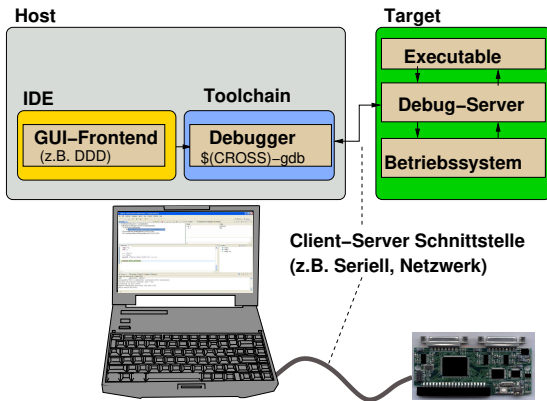
- auf getrennten Rechnern arbeiten, oder
- getrennte Tasks auf einem Rechner sein

- Verschiedene Konstellationen möglich...

# Debug-Server auf dem Target

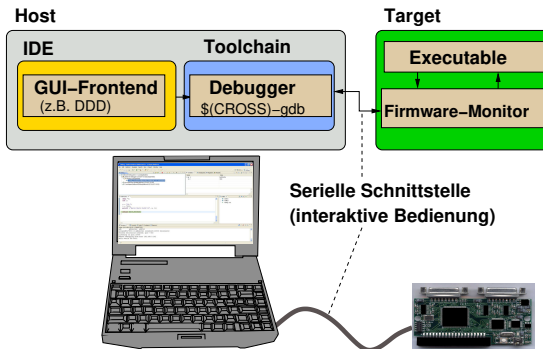


- Server agiert als „Brückenkopf“ des Debuggers im Target
- In das Betriebssystem integriert oder eigene Task (z.B. gdbserver)
- Unterstützung durch das Target-Betriebssystem erforderlich (Netzwerk-Stack, etc.)



# Nutzung der Target-Infrastruktur

- Eine ggf. vorhandene Monitor-Firmware dient als Debug-Server
- Kommunikation i.d.R. über serielle (RS-232) Schnittstelle
- Anpassung des Debuggers an das Monitor-Protokoll erforderlich

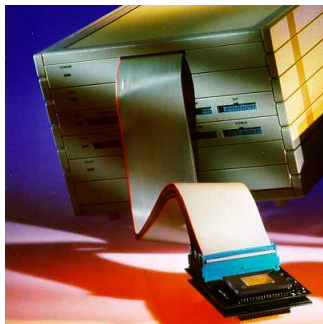




# In-Circuit Emulator (ICE)



- Hardware, ersetzt den eigentlichen Prozessor bzw. Microcontroller
- Spezielle Version des zu emulierenden Chips mit erweiterten Funktionen zur Steuerung
  - ▶ Z.B. *Bond-out-Chip*: Interne Signale herausgeführt ...
  - ▶ ... oder Implementierung der Funktionalität als FPGA
- Erweiterte Möglichkeiten
  - ▶ Tracebuffer: „Mitschneiden“ des Echtzeitverhaltens
  - ▶ Watchpoints (bedingte Unterbrechungen)
- I.d.R. **sehr** teuer!



# JTAG-Schnittstelle: Motivation



- Bausteine werden zunehmend komplexer → Anzahl der Pins steigt → Pins werden immer dichter gepackt
- Zugang zu Chip-internen Signalen wird (*Bond-out-Chip*) schwierig bis unmöglich (Ähnliche Situation auch bei Multilayer-Platinen)
- Das bisher übliche Testen von Chips und Platinen durch Anlegen von Testsignalen (*Stimuli*) und Messen der Antwortsignale (*Responses*) ist nicht mehr praktikabel
- Mitte der 80er Jahre entwickelte ein Firmenkonsortium (die *Joint Test Action Group* – JTAG) das *Boundary Scan* Verfahren
- Ziel: Ein- und Ausgangssignale Chip-intern abfragen und seriell<sup>1</sup> nach außen leiten

---

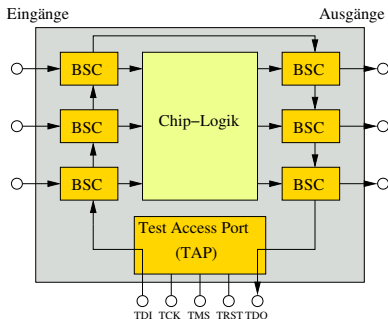
<sup>1</sup>erfordert nur wenige Pins

# JTAG-Schnittstelle: Aufbau (1)

- Ein- und Ausgangsleitungen werden über *Boundary Scan Cells* (BSCs) geführt
- Ein- und Ausgänge der BSCs werden zu einer Kette verbunden
- Ansteuerung der BSCs über einen *Test Access Port* (TAP)

- TAP-Signale:

- 1 **TDI**: Test Data In
- 2 **TDO**: Test Data Out
- 3 **TCK**: Test Clock
- 4 **TMS**: Mode Select
- 5 **TRST**: Reset (opt.)



- Ansteuerung / Abfrage **aller** Chip-Signale über nur 5 Pins

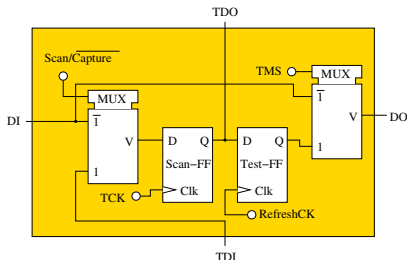
# JTAG-Schnittstelle: Aufbau (2)



- Aufbau einer *Boundary Scan Cell* (BSC):

- Betriebsmodi:

- ① **Normal:** TMS = 0
- ② **Capture:** DI → Scan-FF
- ③ **Scan:** TDI → Scan-FF
- ④ **Refresh:** Scan → Test
- ⑤ **Test:** TMS = 1



- Durch die „Kettenschaltung“ bilden die Scan-Flipflops aller BSCs im Scan-Modus ein großes Schieberegister

→ Eingänge des Chip können mit beliebigen Stimuli beaufschlagt werden, Ausgänge können seriell ausgelesen werden

# Cross-Debugging mit JTAG

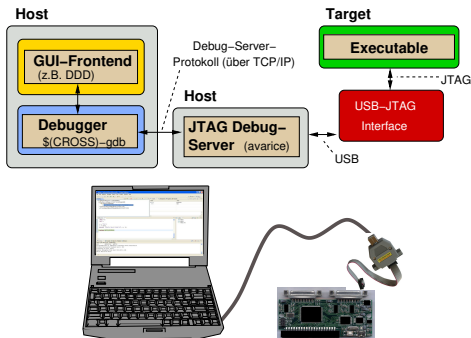


- JTAG in Mikrocontrollern bildet eine Infrastruktur zum Test, aber auch zum Zugang zu Prozessor-internen Signalen
- Chip-Hersteller haben an dieser Infrastruktur Erweiterungen vorgenommen, um über JTAG ...
  - ▶ ... Programme in den Flash-Speicher zu übertragen
  - ▶ ... Programme schrittweise abzuarbeiten
  - ▶ ... Breakpoints und Watchpoints zu setzen
- Diese Erweiterungen des JTAG-Protokolls sind herstellerspezifisch
- Damit können (mit Hilfe eines relativ preiswerten JTAG-Adapters) ähnliche Funktionen wie mit einem ICE erreicht werden

# Beispiel: AVR Cross-Debugging



- **avarice**: Debug-Server für AVR ( $\geq$  ATmega16) JTAG-Schnittstelle
  - Arbeitet auf Host
  - Kommuniziert mit Debugger über Netzwerk (*TCP/IP-Socket*)
- Kann auf demselben oder einem anderen Hostrechner arbeiten



# Virtuelle Target-Maschine



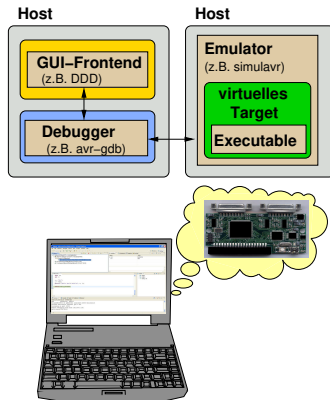
- Das Target wird durch ein Programm vollständig nachgebildet (*emuliert*)
- Dieser *Emulator* stellt damit eine *virtuelle Maschine* zur Verfügung, die –abgesehen von der Geschwindigkeit– exakt das Verhalten des Targets zeigt
- Er besitzt einen integrierten Debug-Server, über den ein Debugger die virtuelle Maschine steuern kann
- Beispiele: Bochs, Qemu (x86), simulavr (AVR), Android (Java)
  - + Softwareentwicklung ohne Target möglich
  - + Sehr weitgehende Kontrollmöglichkeiten (ähnl. ICE)
    - Nicht immer verfügbar
    - Zeitverhalten des Targets wird nicht korrekt nachgebildet (i.d.R. langsamer)
    - E/A-Bausteine werden u.U. nicht oder nur unvollständig nachgebildet

# Beispiel (1): Cross-Debugging mit simulavr



- Target existiert als virtuelle Maschine auf dem Host

- **simulavr**: Emulator für AVR Microcontroller
- Emuliert nur den Prozessor, *keine* E/A-Komponenten
- Nur bedingt brauchbar
- Kommunikation mit Debugger über Netzwerk (*TCP/IP-Socket*)



- Auch hier: Emulator kann auf demselben oder einem anderen Hostrechner arbeiten

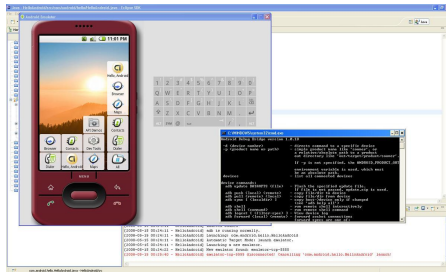


## Beispiel (2): Google Android



- Offene Entwicklungsumgebung für Smartphones<sup>2</sup>
- Hier:

- ▶ Programmierung in Java
- ▶ Peripherie wird durch die JVM emuliert
- ▶ Kein Echtzeit-Verhalten gefordert



- Anwendungsentwicklung war bereits vor Verfügbarkeit entsprechender Geräte möglich

<sup>2</sup>siehe <http://code.google.com/android/>

## 3.2 Hardwarenahes Programmieren in C



# Motivation



- Vorteil der Assemblerprogrammierung: vollkommene Kontrolle über die Maschine
  - ▶ Effizientester Code (?)
  - ▶ Größte Freiheiten in der Wahl des Programmiermodells
  - ▶ Zugriff auf Besonderheiten der Architektur (Register, Port-Mapped I/O, etc.)
- Nachteile:
  - ▶ Höchster Anspruch an Fähigkeit und Kenntnisse der ProgrammiererInnen
  - ▶ Programme (und Kenntnisse) sind nicht „portabel“
  - ▶ Komplexe Algorithmen kaum in Assembler beherrschbar
  - ▶ Fehlerträchtig: z.B. kein Typkonzept
- Heute wird nur noch in seltenen Ausnahmefällen in Assembler programmiert (i.d.R. um Dinge zu tun, die in Hochsprache „nicht gehen“, z.B. Interrupts maskieren, auf Register zugreifen, etc.)

# Was ist „hardwarenahes Programmieren“?



- Z.B. Google-Suche liefert keine klare Definition des Begriffs (Stattdessen aber sehr viele interessante Stellenangebote...)
- Vorstellung / Kenntnis der Vorgänge innerhalb des Rechensystems beim Ausführen von (Hochsprache-)Programmen
  - ▶ Speicherorganisation (Code / Daten / Stack / Heap)
  - ▶ Aufrufschnittstelle (*Call by Value* / *Reference* und Konsequenzen)
  - ▶ Fallstricke (Pufferüberlauf, Stack-Überlauf, Nebenläufigkeit, Compileroptimierungen, Caches, virtuelle Adressierung...)
  - ▶ Tipps und Tricks: Effizienzoptimierung, Fehlersuche

→ „Blick über den Tellerand“ des Hochsprachen- Modells

- Jedes Rechensystem braucht zwingend hardwarenahe Programme (Betriebssysteme / Gerätetreiber / E/A-Bibliotheken)

→ Stark nachgefragte (aber selten anzutreffende) Fähigkeit

# Warum C?



- C ist eine portable Programmiersprache, d.h. C-Programme können Im Prinzip<sup>TM</sup> auf jeder Maschine laufen, für die es einen C-Compiler gibt
- Dennoch sind typische maschinenspezifische Konzepte wie Stack, Speicheradressen, etc. in C noch zugänglich
- Sprache ist unabhängig von Laufzeitbibliothek
- In der Regel benutzt jeder C-Compiler eine exakt definierte Schnittstelle (*Application Binary Interface*, ABI) zum Maschinencode
- Dadurch können aus C heraus problemlos Assembler-Unterprogramme aufgerufen werden (und umgekehrt)
- Mit `asm` können zudem bei den meisten C-Compilern Assemblerbefehle direkt in C-Code eingebettet werden (*inline assembler*)

# Beispiel: inline-Assembler



## Inline-Assembler-Routine (GNU C)

```
int set_stack_and_go(stackp, entry)
void *stackp;
int (*entry)();
{
#ifdef __i386__
asm("mov %%eax,%0 \
    mov %%esp,%1 \
    push %%eax \
    ret " : : "p" (entry), "p" (stackp) : "%%eax", "%%esp");
#endif
#ifdef __m68k__
asm("moveal %0,%%a0 \
    moveal %1,%%sp \
    jmp    %%a0@" : : "p" (entry), "p" (stackp) : "%%a0", "%%sp");
#endif
#ifdef __ppc__
    ....
#endif
}
```

# Speicherorganisation



- Ein Programm und seine Daten werden während der Ausführung vollständig im Hauptspeicher gehalten<sup>3</sup>
- Die Programm- und Datenobjekte lassen sich in verschiedene Klassen einteilen
- Der Compiler/Linker ordnet die statischen (d.h. zur Compilezeit bekannten) Objekte je nach Klasse bestimmten „Sektionen“ zu:

Klasse	Lesen	Schreiben	Ausführen	Initialisiert	Sektion
Programmcode	x	-	x	x	.text
initialisierte Daten	x	x	-	x	.data
uninitialisierte Daten	x	x	-	-	.bss
nur-lese Daten	x	-	-	x	.rodata

- Für jede Sektion wird ein entsprechend großes, zusammenhängendes Stück Speicher eingeteilt.

<sup>3</sup>Virtueller Speicher/Paging werden hier nicht betrachtet

# Dynamische Daten



- Der restliche (freie) Speicher wird dynamisch, d.h. zur Laufzeit zugeteilt:
    - ▶ **Heap:** Dynamischer Speicher
      - ★ Wird in C mit `malloc()` zugeteilt und mit `free()` wieder freigegeben (In C++ mit `new` und `delete`)
      - ★ Wächst nach „oben“, d.h. zu höheren Adressen hin
      - ★ Anfang typischerweise direkt im Anschluss an statische Daten (`.bss`)
    - ▶ **Stack:** Lokale (dynamische) Variablen
      - ★ Sämtliche Automatic-Variablen in C (ausserdem `alloca()`-Funktion)
      - ★ Auch: Aufrufhierarchie (d.h. Rückkehradressen) und übergebene Parameter)
      - ★ Zuteilung / Freigabe automatisch durch Inkrementieren bzw. Dekrementieren des Stackpointers (z.B. mit `PUSH-` und `POP-Befehlen`)
      - ★ Wächst (i.d.R.) nach „unten“, d.h. zu kleiner werdenden Adressen hin
      - ★ Anfang typischerweise am oberen Ende des RAM-Speichers
- Stack und Heap wachsen einander entgegen. Wenn sie „zusammenstoßen“ geschehen „merkwürdige Dinge“...



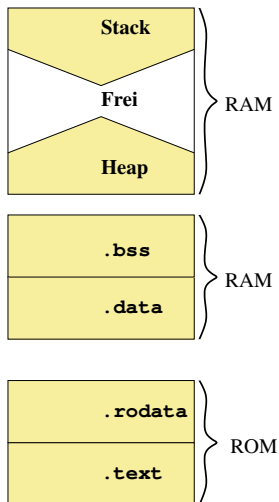
# Beispiel



## Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =     "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



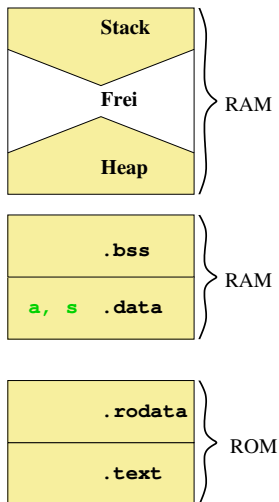
# Beispiel



## Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s      = "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



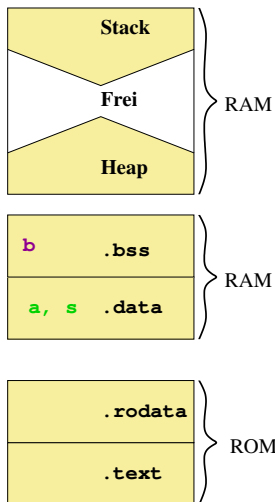
# Beispiel



## Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s       = "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



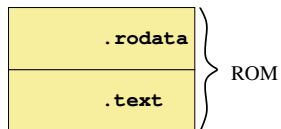
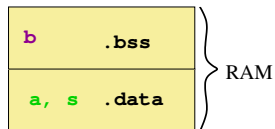
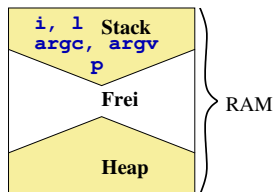
# Beispiel



## Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =     "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



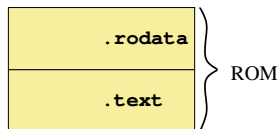
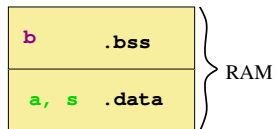
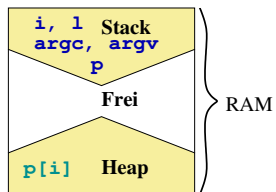
# Beispiel



## Speicherorganisation eines C-Programms

```
static int    a = 5;
char *s =     "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



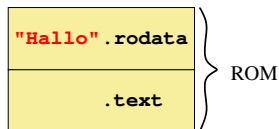
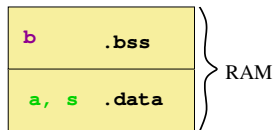
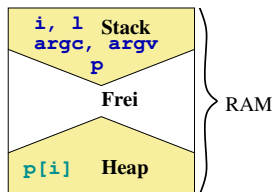
# Beispiel



## Speicherorganisation eines C-Programms

```
static int  a = 5;
char *s =   "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



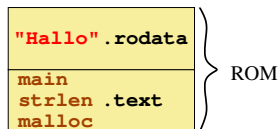
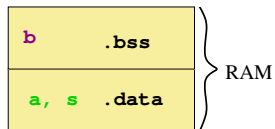
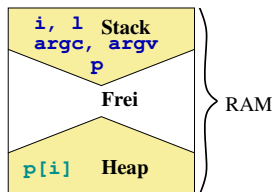
# Beispiel



## Speicherorganisation eines C-Programms

```
static int  a = 5;
char *s =   "Hallo";
int b;

main(int argc, char *argv[])
{
    int i;
    int l = strlen(s);
    char *p;
    p = malloc(l + 1);
    for(i = 0; i < l; i++)
        p[i] = s[i];
}
```



# Initialisierung – wie? (1)



- Die Sektionen `.text`, `.rodata`, `.data` und `.bss` müssen vor dem Start des Programms initialisiert (d.h. mit definiertem Inhalt versehen) werden
- Wird das Programm von einem Betriebssystem (z.B. Windows oder Linux) gestartet, so lädt dessen „Program Launcher“ die Speicherinhalte aus einer ausführbaren Datei (z.B. `program.exe` in Windows) ins RAM
- Programmcode und nur-lese-Daten liegen dann im RAM und sind –technisch gesehen– jederzeit zur Laufzeit überschreibbar
- Potenzielles Sicherheitsproblem, zugleich aber auch Möglichkeit des Debuggings mit Breakpoints, etc.



## Initialisierung – wie? (2)



- Arbeitet das Programm ohne Betriebssystem, so liegen `.text` und `.rodata` z.B. im nicht-flüchtigen ROM-Speicher, d.h. sie sind beim Programmstart bereits initialisiert
- Die Sektionen `.data` und `.bss` müssen zur Laufzeit schreibbar sein, d.h. sie können nur im (flüchtigen) RAM liegen
- Der Compiler / Linker legt dazu eine „Schattenkopie“ mit den Initialisierungswerten der `.data` Sektion ins ROM
- Vor dem Aufruf von `main()` wird eine (ebenfalls vom Linker automatisch hinzugefügte) Initialisierungsroutine aufgerufen, die die „Schattenkopie“ aus dem ROM ins RAM kopiert und die `.bss` Sektion mit Nullen füllt

→ Initialisierte Daten verursachen Platzbedarf in ROM **und** RAM

# C „Startup Code“



- Jede C-Programmierungsumgebung fügt einem kompilierten Programm implizit einen „Startup-Code“ hinzu
- Aufgabe: Umgebung für `main()` herstellen ...
  - ▶ .data Sektion initialisieren (macht ggf. auch das Betriebssystem)
  - ▶ .bss Sektion „nullen“ (dto.)
  - ▶ Ggf. Umgebungsvariablen und `argc / argv[]` aufbereiten
  - ▶ Bei C++-Programmen: Konstruktoren der statischen Klassen aufrufen
- ... dann `main()` aufrufen
  - ▶ Ggf. `argc / argv[]` übergeben
  - ▶ Falls `main()` zurückkehrt: `_exit()` aufrufen
- Startup-Code ist Maschinen-, Compiler- und Betriebssystemabhängig

# Beispiel: Multiboot-Startup (1)



Label	Code	Kommentare
<code>_begin:</code>	<code>jmp multiboot_entry</code>	
	<code>.align 4</code>	Align 32 bits boundary
<code>multiboot_header:</code>		Multiboot header magic
	<code>.long MULTIBOOT_HEADER_MAGIC</code>	
	<code>.long MULTIBOOT_HEADER_FLAGS</code>	flags
	<code>.long -(MULTIBOOT_HEADER_MAGIC</code>	
	<code>+MULTIBOOT_HEADER_FLAGS)</code>	checksum
	<code>.long multiboot_header</code>	header_addr
	<code>.long _begin</code>	load_addr
	<code>.long _edata</code>	load_end_addr
	<code>.long _end</code>	bss_end_addr
	<code>.long multiboot_entry</code>	entry_addr
<code>multiboot_entry:</code>		
	<code>movl \$_end,%esp</code>	Initialize the stack pointer
	<code>addl \$STACK_SIZE,%esp</code>	
	<code>pushl \$0</code>	Reset EFLAGS
	<code>popf</code>	
	<code>pushl %ebx</code>	Push pointer to Multiboot struct
	<code>pushl %eax</code>	Push magic value
	<code>call __init_pc</code>	Now enter the C code

## Beispiel: Multiboot-Startup (2)



```
__init_pc(int magic, multiboot_header *hdr)
{
    unsigned char *to;
    volatile int count;
    /* clear BSS section: */
    to = (unsigned char *)__bss_start;
    count = (unsigned char *)__end - to;

    if(count > 0) do
    {
        *to++ = 0;
    }
    while(--count);

    asm( /* reset stack, jump to main: */
        "movl    %0,%%esp      \n"
        "addl    $STACK_SIZE, %%esp \n"
        "jmp     main          \n"
        ":: "i"  (__end));
}
```

# Prozeduraufruf: Nötige Schritte



- Parameterkopien für aufgerufene Funktion hinterlegen
- Programmsteuerung an die Prozedur übergeben
- Arbeitsspeicher für Prozedur bereitstellen (lokale Variablen)
- Prozedur ausführen
- Ergebnis an den Aufrufer übergeben
- Arbeitsspeicher der Prozedur wieder freigeben
- zum Aufrufer zurückkehren

## Beispiel: Prozeduraufruf

```
int funktion(int p1, int p2, ..)
{
    int lokal1, lokal2, ...;
    int ergebnis;
    .....
    return(ergebnis);
}

main()
{
    int a, b, c, ...;
    int resultat;
    .....
    resultat = funktion(a, b, ...);
    .....
}
```

# Prozeduraufruf



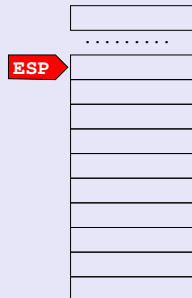
- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    ret
.....
EPC → pushl #2
      pushl #1
      call proz
rueckadr:
    move eax, rw(ebp)
    add#8, esp
```



# Prozeduraufruf

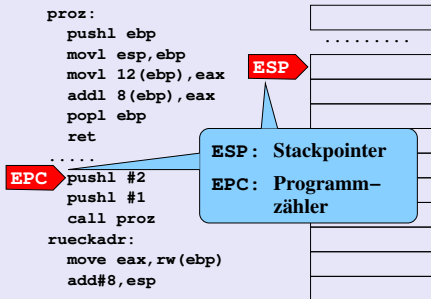


- 1 Argumentkopien auf den Stack
- 2 Call → Rücksprungadresse auf den Stack

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```



# Prozeduraufruf



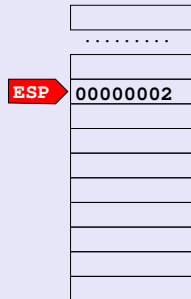
- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    ret
.....
    pushl #2
    EPC → pushl #1
    call proz
rueckadr:
    movl rw, eax
    addl $8, esp
```





# Prozeduraufruf



- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl esp
    movl esp, 12(%ebp), eax
    addl 8(%ebp), eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    movl rw, 12(%ebp)
    addl $8, esp
```

Assembler:

push x

C:

\* (--esp)=x

ESP

00000002

EPC

# Prozeduraufruf



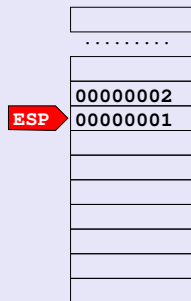
- 1 Argumentkopien auf den Stack
- 2 Call → Rücksprungadresse auf den Stack

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp,ebp
    movl 12(ebp),eax
    addl 8(ebp),eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    EPC → call proz
    rueckadr:
    move eax,rw(ebp)
    add#8,esp
```



# Prozeduraufruf



- ① Argumentkopien auf den Stack
- ② Call → Rücksprungadresse auf den Stack

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
push
mov
mov
add
pop
ret
```

**Call bewirkt:**

1. Rücksprungadresse auf Stack
2. Programmzähler = Ziel

```
call proz    * (--esp)=epc+S;
              epc=proz;
              (S=Größe des Call-Opcodes)
```

**EPC** →

```
.....
pushl #2
pushl #1
call proz
rueckadr:
move eax, rw(ebp)
add#8, esp
```


# Prozeduraufruf



- ③ „Framepointer“ (EBP) retten
- ④ Neuen Framepointer laden

## Beispiel: Prozeduraufruf

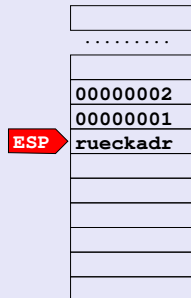
```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```

      proz:
EPC → pushl ebp
      movl esp, ebp
      movl 12(ebp), eax
      addl 8(ebp), eax
      popl ebp
      ret
      .....
      pushl #2
      pushl #1
      call proz
rueckadr:
      move eax, rw(ebp)
      add#8, esp

```



# Prozeduraufruf



- 3 „Framepointer“ (EBP) retten
- 4 Neuen Framepointer laden

## Beispiel: Prozeduraufruf

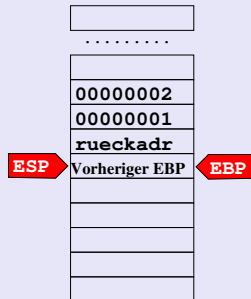
```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

proz:

```

pushl ebp
EPC → movl esp, ebp
      movl 12(ebp), eax
      addl 8(ebp), eax
      popl ebp
      ret
.....
      pushl #2
      pushl #1
      call proz
rueckadr:
      move eax, rw(ebp)
      add#8, esp
```



# Prozeduraufruf



- 5 Parameter addieren
- 6 Returnwert in Register (hier: EAX)

## Beispiel: Prozeduraufruf

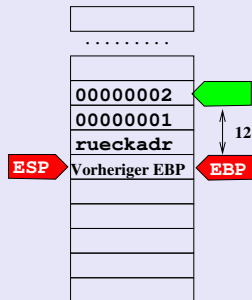
```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```

proz:
    pushl ebp
    movl esp, ebp
    EPC → movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax, rw(ebp)
    add#8, esp

```



# Prozeduraufruf



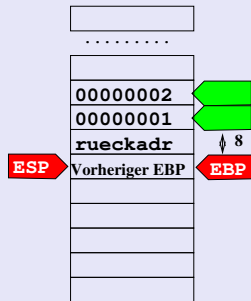
- 5 Parameter addieren
- 6 Returnwert in Register (hier: EAX)

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp,ebp
    movl 12(ebp),eax
    EPC → addl 8(ebp),eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax,rw(ebp)
    add#8,esp
```



# Prozeduraufruf



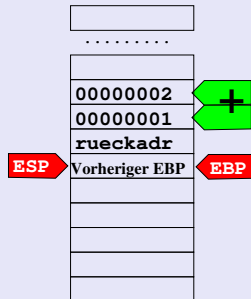
- 5 Parameter addieren
- 6 Returnwert in Register (hier: EAX)

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    EPC → addl 8(ebp), eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax, rw(ebp)
    add#8, esp
```





# Prozeduraufruf



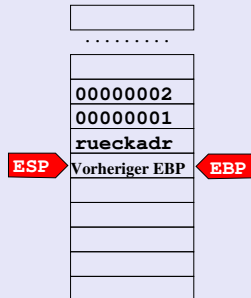
- 7 Alten Framepointer wiederherstellen
- 8 Rückkehr zum Aufrufer

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    EPC → popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax, rw(ebp)
    add#8, esp
```



# Prozeduraufruf



- 7 Alten Framepointer wiederherstellen
- 8 Rückkehr zum Aufrufer

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    EPC → popl %ebp
    ret
    .....
    pushl #2
    pushl #1
    call proz
    rueckadr:
    move %eax, %r(%ebp)
    addl $8, %esp
```

Assembler:

C:

pop x

x=\*esp++

00000002

00000001

**rueckadr**

Vorheriger EBP

**EBP****ESP**

# Prozeduraufruf



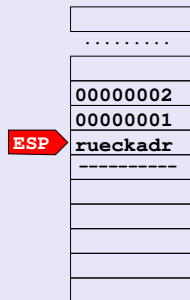
- 7 Alten Framepointer wiederherstellen
- 8 Rückkehr zum Aufrufer

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    EPC → ret
    .....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax, rw(ebp)
    add#8, esp
```



# Prozeduraufruf



- 7 Alten Framepointer wiederherstellen
- 8 Rückkehr zum Aufrufer

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(%ebp), eax
    addl 8(%ebp), eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax, rw(%ebp)
    add#8, esp
```

**EPC** →

**ESP** →

**Assembler:**

**C:**

ret      epc=\*esp++

00000002
00000001
rueckadr
-----

# Prozeduraufruf



9 Ergebnis Speichern

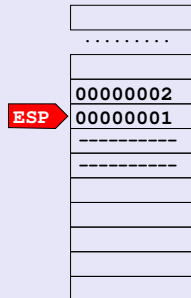
10 Stack abräumen

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    EPC → move eax, rw(ebp)
    add#8, esp
```



# Prozeduraufruf



9 Ergebnis Speichern

10 Stack abräumen

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return(a+b);
}
```

```
.....
rw = proz(1, 2);
.....
```

```
proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    ret
.....
    pushl #2
    pushl #1
    call proz
rueckadr:
    move eax, rw(ebp)
    add#8, esp
```

ESP

00000003
.....
00000002
00000001
-----
-----

EPC

# Prozeduraufruf



Hochschule RheinMain

- 9 Ergebnis Speichern
- 10 Stack abräumen

## Beispiel: Prozeduraufruf

```
int proz(int a, int b)
{
    return (a+b);
}
```

• • • • •

```
rw = proz(1, 2);
```

• • • • •

```

proz:
    pushl ebp
    movl esp, ebp
    movl 12(ebp), eax
    addl 8(ebp), eax
    popl ebp
    ret

```

• • • • •

```
pushl #2
```

```
pushl #1
```

call proz

rueckadr:

```
move eax, rw (ebp)
```

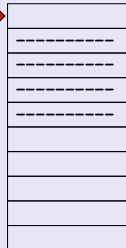
add#8, esp

**EPC** . . . .

00000003

.....

ESP



# Prozeduraufruf – Fazit (1)



- Beträchtlicher Aufwand
- Nur sinnvoll, wenn „Nutzcode“  $\gg$  Code für Prozeduraufruf  
(Im Beispiel: Eine Instruktion Nutzcode (Addition) vs. 10 Instruktionen für den Prozeduraufruf)
- In solchen Fällen effizienter:

- ① Präprozessor-Macro

```
#define proz(a, b) ((a)+(b));
```

- ② inline-Funktion

```
inline int proz(int a, int b)
{
    return(a+b);
}
```



## Prozeduraufruf – Fazit (2)



- Wertparameter (*Call by value*) → Parameterkopien werden über den Stack übergeben
  - Einige (typischerweise RISC-)Architekturen verwenden alternativ auch Prozessorregister für die Übergabe (effizienter – warum?)
  - Kein Problem wenn die Größe der Objekte  $\leq$  Registergröße
- ? .. aber wie soll dann so etwas gehen:

### Beispiel: Übergabeparameter $\geq$ Registergröße

```
struct parameterlist {  
    int elem1;  
    int elem2;  
    ....  
    int elemN;  
};  
  
main()  
{  
    ....  
    struct parameterlist x;  
    funktion(x);  
    ....  
}
```

# Auswirkungen von *Call by Value*



- Größere Objekte „passen“ nicht in Register, können nur über den Stack kopiert werden
- Laufzeitaufwand und ggf. großer Stackbedarf
- Wird von älteren C-Compilern z.T. gar nicht unterstützt
- Meistens besser: Statt Objektkopie einen **Zeiger** auf das Objekt übergeben (Eigentlich ist das *Call by reference*)

## Beispiel: Übergabe von Zeiger statt Objektkopie

```
struct parameterlist {  
    int elem1;  
    int elem2;  
    ....  
    int elemN;  
};  
  
main()  
{  
    ....  
    struct parameterlist x;  
    funktion(&x);  
    ....  
}
```

# Prozeduraufruf – Fazit (3)



- Rückgabewerte werden in einem Register zurückgeliefert
- ? Wie ist dann so etwas möglich:

## Beispiel: Rückgabeobjekt $\geq$ Registergröße

```
struct parameterlist {  
    int elem1;  
    int elem2;  
    ....  
    int elemN;  
};
```

```
struct parameterlist funktion()  
{  
    struct parameterlist x;  
    x.elem1 = ....;  
    .....  
    return(x);  
}
```

- Auch hier gilt:
  - ▶ Wird von älteren C-Compilern z.T. nicht unterstützt
  - ▶ Falls unterstützt: Aufwändiges Kopieren des Objekts (d.h. teuer)

→ Besser *nicht* verwenden!

# Große Objekte zurückliefern



- Besser: Aufrufer hält Platz für das Rückgabeobjekt vor
- Prozedur erhält Zeiger auf das Objekt, kann dieses manipulieren

## Beispiel: Rückgabeobjekt $\geq$ Registergröße

```
struct liste {  
    int elem1;  
    int elem2;  
    ....  
    int elemN;  
};  
  
void funktion(struct liste *px)  
{  
    px->elem1 = ....;  
    return;  
}  
  
void aufrufer(..)  
{  
    struct liste x;  
    ....  
    funktion(&x);  
    ....  
}
```

- Entspricht funktional der von den meisten (aber nicht allen!) Compilern verwendeten Technik


# Verschachtelte Prozeduren





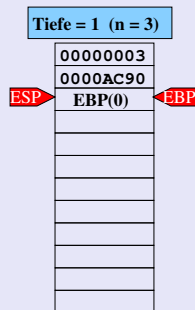
Hochschule RheinMain

- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

int fak(int n)	0100 fak: pushl ebp
{	0104
 if(0 == n)	 movl esp, ebp
return(1);	.....
else	1208 call fak
return(n * fak(n-1));	120C move eax, ...
}	.....
	135E ret
	.....
.....	AC80 push #3
f3 = fak(3));	AC8E call fak
.....	AC90 move eax, ...




# Verschachtelte Prozeduren




- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)


```

int fak(int n)
{
   if (0 == n)
    return (1);
  else
    return (n * fak(n-1));
}

.....
f3 = fak(3));
.....

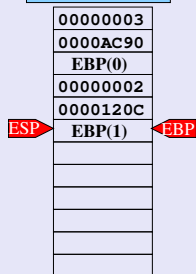
```

```

0100 fak: pushl ebp
0104
 movl esp, ebp
.....
1208      call fak
120C      move eax, ..
.....
135E      ret
.....
AC80      push #3
AC8E      call fak
AC90      move eax, ..

```

Tiefe = 2 (n = 2)




# Verschachtelte Prozeduren




- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf


(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```

int fak(int n)
{
   if (0 == n)
    return (1);
  else
    return (n * fak(n-1));
}

....
f3 = fak(3));
....

```

0100	fak: pushl ebp
0104	 movl esp, ebp
.....	.....
1208	call fak
120C	move eax, ..
.....	.....
135E	ret
.....	.....
AC80	push #3
AC8E	call fak
AC90	move eax, ..

Tiefe = 3 (n = 1)

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
00000001
0000120C
EBP(2)

ESP →


← EBP

# Verschachtelte Prozeduren




- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)


```

int fak(int n)
{
   if (0 == n)
    return (1);
  else
    return (n * fak(n-1));
}

.....
f3 = fak(3));
.....

```

```

0100 fak: pushl ebp
0104
 movl esp, ebp
.....
1208      call fak
120C      move eax, ..
.....
135E      ret
.....
AC80      push #3
AC8E      call fak
AC90      move eax, ..

```

Tiefe = 4 (n = 0)

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
00000001
0000120C
EBP(2)
00000000
0000120C
EBP(3)

ESP →

← EBP




# Verschachtelte Prozeduren




- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)


```

int fak(int n)
{
 if (0 == n)
    return (1);
else
    return (n * fak(n-1));
}

.....
f3 = fak(3));
.....

```

```

0100 fak: pushl ebp
0104
 movl esp, ebp
.....
1208      call fak
120C      move eax, ..
.....
135E      ret
.....
AC80      push #3
AC8E      call fak
AC90      move eax, ..

```

Tiefe = 3 wert=1

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
00000001
0000120C
EBP(2)
-----
-----
-----

ESP →


← EBP

# Verschachtelte Prozeduren




- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)


```

int fak(int n)
{
   if (0 == n)
    return (1);
  else
    return (n * fak(n-1));
}

.....
f3 = fak(3));
.....

```

```

0100 fak: pushl ebp
0104
 movl esp, ebp
.....
1208      call fak
120C      move eax, ..
.....
135E      ret
.....
AC80      push #3
AC8E      call fak
AC90      move eax, ..

```

Tiefe = 2 wert=1

00000003
0000AC90
EBP(0)
00000002
0000120C
EBP(1)
-----
-----
-----
-----
-----
-----
-----

ESP


EBP

# Verschachtelte Prozeduren




- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf


(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```

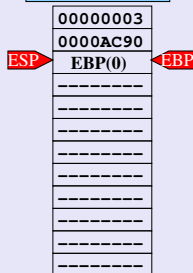
int fak(int n)
{
   if (0 == n)
    return (1);
  else
    return (n * fak(n-1));
}

.....
f3 = fak(3));
.....

```

0100	fak: pushl ebp
0104	 movl esp, ebp
.....	.....
1208	call fak
120C	move eax, ..
.....	.....
135E	ret
.....	.....
AC80	push #3
AC8E	call fak
AC90	move eax, ..

Tiefe = 1 wert=2




# Verschachtelte Prozeduren



- Im Beispiel hier: Rekursion (Berechnung der Fakultät ( $n!$ ))

## Beispiel: Rekursiver Prozeduraufruf

(Gezeigt ist jeweils der Zustand bei Erreichen der mit  markierten Zeile)

```

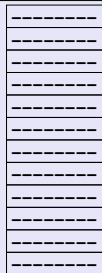
int fak(int n)
{
    if (0 == n)
        return (1);
    else
        return (n * fak(n-1));
}

.....
f3 = fak(3);
STOP .....
  
```

```

0100 fak: pushl ebp
0104
movl esp, ebp
.....
1208      call fak
120C      move eax, ..
.....
135E      ret
.....
AC80      push #3
STOP AC8E      call fak
AC90      move eax, ..
  
```

Tiefe = 0 wert=6



# Rekursion – Fazit



- Sowohl Parameterkopien als auch lokale Variablen erhalten mit jeder neuen Verschachtelung eine eigene Instanz
- Auf statische Variablen trifft das nicht zu!
- Rekursive (reentrante) Prozeduren sollten i.d.R. keine statischen Variablen benutzen

# Lokale Variablen

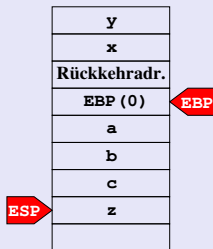


- 1 Bisher betrachtete Beispiele hatten keine lokalen Variablen
- 2 Stackpointer (ESP) und Framepointer (EBP) waren stets gleich (Ist der Framepointer damit redundant?)
- 3 Lokale Variablen liegen zwischen EBP und ESP:

## Beispiel: Platz für lokale Variablen

```
fkt(int x, int y)
{
    int a, b, c;
    int z;
    a = x;
    b = y;
    c = x - y;
    z = 2 * (a + b) - c;
}
```

```
fkt: pushl ebp
      movl esp, ebp
      subl #16, esp
      movl 8(ebp), eax
      movl eax, -4(ebp)
      movl 12(ebp), eax
      movl eax, -8(ebp)
      movl 12(ebp), edx
      movl 8(ebp), eax
      subl edx, eax
      .....
      ret
```



# Lokale Variablen



- 1 Bisher betrachtete Beispiele hatten keine lokalen Variablen
- 2 Stackpointer (ESP) und Framepointer (EBP) waren stets gleich (Ist der Framepointer damit redundant?)
- 3 Lokale Variablen liegen zwischen EBP und ESP:

## Beispiel: Platz für lokale Variablen

```
fkt(int x, int y)
{
    int a, b, c;
    int z;
    a = x;
    b = y;
    c = x - y;
    z = 2 * (a + b) - c;
}
```

```
fkt: pushl ebp
     movl esp,ebp
     subl#16,esp
     movl 8(ebp),eax
     movl eax,-4(ebp)
     movl 12(ebp),eax
     movl eax,-8(ebp)
     movl 12(ebp),edx
     movl 8(ebp),eax
     subl edx,eax
     .....
     ret
```

4 \* sizeof(int) = 16

y
x
Rückkehradr.
EBP (0)
a
b
c
z

EBP

ESP

# Dynamische lokale Variablen

- Im vorigen Beispiel: Gesamte Größe der lokalen Daten ist zur Compile-Zeit bekannt
- Allokation durch Addition eines festen Betrages zum Stackpointer
- Seit C99 Standard möglich: Lokale Felder *variabler* Größe

## Beispiel: variable Feldgröße

### Feld variabler Größe

```
fkt(int x, int y)
{
    int a[x];
    int i;

    for(i = 0; i < x; i++)
        a[i] = ....;
    ....
}
```

### Vor C99: Funktion `alloca()`

```
fkt(int x, int y)
{
    int *a;
    int i;
    a = (int*)alloca(x * sizeof(*a));
    for(i = 0; i < x; i++)
        a[i] = ....;
    ....
}
```



# Dynamische lokale Variablen

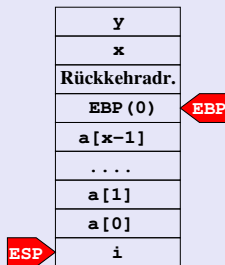


- Feldgröße kann von Aufruf zu Aufruf variieren
- Adressabstand zwischen Framepointer und Variablen (hier z.B. `i`) ist *nicht* konstant
- Aufwändige Adressberechnung bei jedem Zugriff auf lokale Variablen  
→ teuer

## Beispiel: variable Feldgröße

```
fkt(int x, int y)
{
    int a[x];
    int i;

    for(i = 0; i < x; i++)
        a[i] = ....;
    ....
}
```



# Pufferüberlauf-Attacke

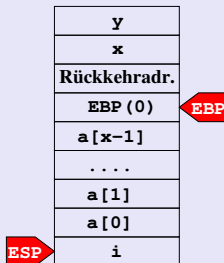


- **Vorsicht:**  $a[x+1]$  ist die Rückkehradresse
- Mit  $a[x+1]$  = Zieladresse kann die Rückkehradresse verändert werden
- Statt zum Aufrufer zurückzukehren, kann ein beliebiges Stück (womöglich als Daten eingeschleust) Schadcodes angesprungen werden

## Beispiel: Gefahr von Puffer-Überlauf

```
fkt(int x, int y)
{
    int a[x];
    int i;

    for(i = 0; i < x; i++)
        a[i] = ....;
    ....
}
```



# Frame pointer



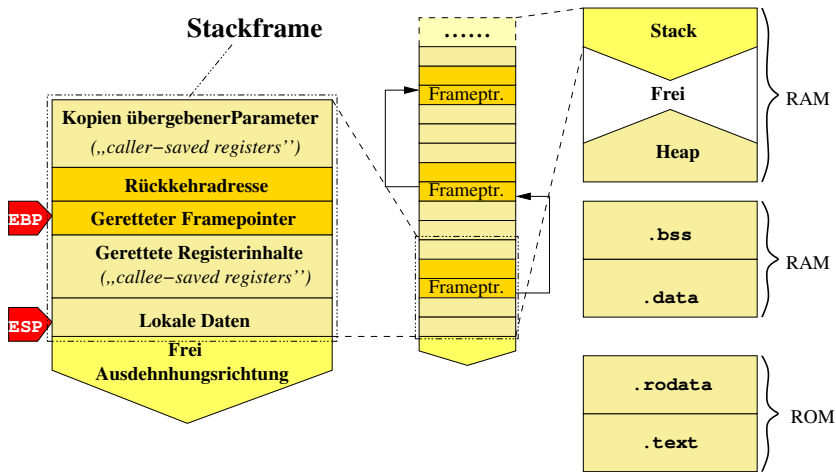
- Lokale Variablen werden über Framepointer (EBP) referenziert:
  - ▶ Positives Displacement → Parameterkopie
  - ▶ Negatives Displacement → Lokale Variable
  - ▶ Displacement = 0 → Framepointer des Aufrufers
  - ▶ Displacement = 4 → Rückkehradresse
- Eigentlich könnten alle diese Objekte auch über den Stackpointer (ESP) referenziert werden → Prinzipiell wäre der Framepointer entbehrlich
- Einige Compiler / Architekturen können auch ohne Framepointer arbeiten Z.B. GNU Compiler (gcc) mit `-fomit-frame-pointer`
- Liefert geringfügig effizienteren Code
- Nachteil: Aufrufhierarchie (*Backtrace*) ist (z.B. bei der Fehlersuche) nicht mehr rekonstruierbar

# Zusammenfassung – Prozeduren



## • Allgemeine Struktur des *Stackframe* einer Prozedur

### Stackframe



# setjmp()/longjmp()



- `setjmp()`/`longjmp()` – Funktionen der Standard C-Bibliothek
  - ▶ `setjmp()`: Speichert („rettet“) den momentanen Prozessorzustand
  - ▶ `longjmp()`: Stellt einen zuvor gespeicherten Zustand wieder her
- Implementierung **immer** in Assembler, aus C heraus aufrufbar
- Verwendung typischerweise als „nicht-lokales goto“

## Prototypen

```
#include <setjmp.h>
/* definiert u.a. jmp_buf */

int setjmp(jmp_buf zustand);

void longjmp(jmp_buf zustand, int retwert);
```

# setjmp() – Arbeitsweise



- jmp\_buf: Array zum Abspeichern des Prozessorzustandes
- setjmp() kopiert ..

- ▶ *callee saved registers* (einschl. Stackpointer)
- ▶ Rückkehradresse

## Beispiel-Implementierung für Intel i386:

```
setjmp: movl 4(esp),eax ; jmp_buf Adresse in EAX
        movl ebx,0(eax) ; callee-saved Register..
        movl esi,4(eax) ; (hier: ebx, esi,edi, ebp)
        movl edi,8(eax) ; .. in jmp_buf speichern
        movl ebp,12(eax)
        movl 0(esp),ebx ; Rückkehradresse in EBX
        movl ebx,20(eax) ; in jmp_buf speichern
        movl 4(esp),ebx ; Stackpointer in ebx
        movl ebx,16(eax) ; in jmp_buf speichern
        movl 0(eax),ebx ; ebx wiederherstellen
        xorl eax,eax ; Returnwert (eax) = 0
        ret
```

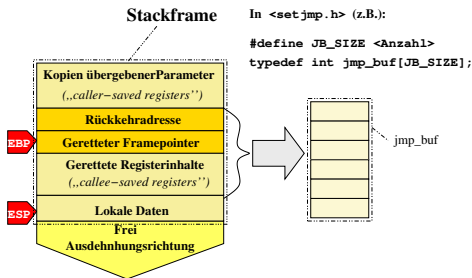
- .. in die bereitgestellte jmp\_buf Datenstruktur
  - Liefert Null als Returnwert
- Macht einen „Schnappschuss des Prozessorzustandes“

# setjmp() – Arbeitsweise



- jmp\_buf: Array zum Abspeichern des Prozessorzustandes
- setjmp() kopiert ..

- ▶ *callee saved registers* (einschl. Stackpointer)
- ▶ Rückkehradresse



- .. in die bereitgestellte jmp\_buf Datenstruktur
  - Liefert Null als Returnwert
- Macht einen „Schnappschuss des Prozessorzustandes“

## `longjmp()` – Arbeitsweise



- Lädt *callee saved registers* (einschl. Stackpointer) aus der übergebenen `jmp_buf` Datenstruktur
  - Kehrt zur Rückkehradresse aus der `jmp_buf` Datenstruktur zurück
  - Liefert vom Aufrufer übergebenen `retwert` als Returnwert
- Stellt den Prozessorzustand exakt wieder so her, wie er beim Aufruf von `setjmp()` gespeichert wurde
- Das bedeutet insbesondere auch, dass `longjmp()` zu der zugehörigen Aufrufposition von `setjmp()` zurückkehrt!
  - Einzige Möglichkeit der Unterscheidung für den Aufrufer von `setjmp()`: der Returnwert
- Achtung: niemals 0 als zweites Argument an `longjmp()` übergeben



# Typische Verwendung: „non-local goto“



## Ohne `setjmp()`/`longjmp()`

„Durchreichen“ der Fehlerbedingung über alle Aufrufebenen:

```
if (FEHLER == funktion1(..)) {
    .. Fehlerabbruch ...
else
    ....

funktion1(..)
{
    if (FEHLER == funktion2(..))
        return (FEHLER);
    else
        ....
}
....
funktionN (...)
{
    if (Fehlerbedingung)
        return (FEHLER);
}
```

## Mit `setjmp()`/`longjmp()`

Direkte Rückkehr zur obersten Aufrufebe-  
ne ohne Zwischenebenen zu involvieren

```
static jmp_buf buf;

if ((fcode = setjmp(buf))
    .. Fehlerabbruch ...
else
    ....
funktion1(..)
{
    funktion2(..);
    ....
}
....
funktionN (...)
{
    if (Fehlerbedingung)
        longjmp(buf, fcode);
}
```

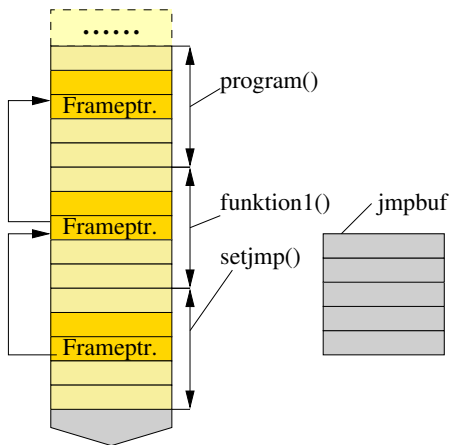
# Einschränkungen



- Achtung: immer nur „von innen nach außen“ springen!

## Beispiel: Unzulässig:

```
static jmp_buf buf;  
  
programm()  
{  
    funktion1 (...);  
    longjmp(buf);  
}  
  
funktion1(...)  
{  
    if (setjmp(buf))  
        ...  
    else  
        ...  
}
```



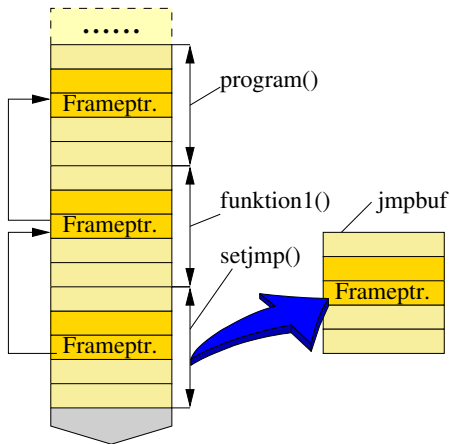
# Einschränkungen



- Achtung: immer nur „von innen nach außen“ springen!

## Beispiel: Unzulässig:

```
static jmp_buf buf;  
  
programm()  
{  
    funktion1 (...);  
    longjmp(buf);  
}  
  
funktion1(...)  
{  
    if (setjmp(buf))  
        ...  
    else  
        ...  
}
```



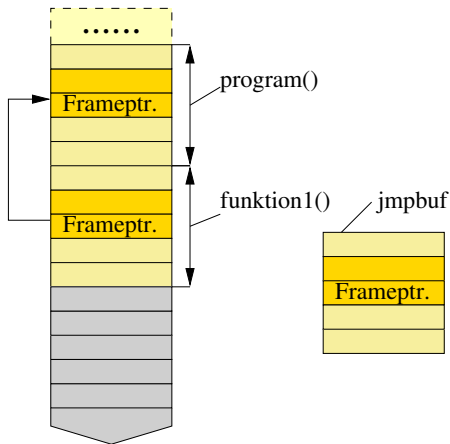
# Einschränkungen



- Achtung: immer nur „von innen nach außen“ springen!

## Beispiel: Unzulässig:

```
static jmp_buf buf;  
  
programm()  
{  
    funktion1 (...);  
    longjmp(buf);  
}  
  
funktion1(...)  
{  
    if (setjmp(buf))  
        ...  
    else  
        ...  
}
```



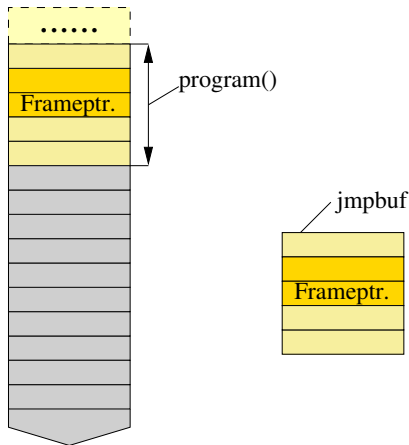
# Einschränkungen



- Achtung: immer nur „von innen nach außen“ springen!

## Beispiel: Unzulässig:

```
static jmp_buf buf;  
  
programm()  
{  
    funktion1 (...);  
    longjmp(buf);  
}  
  
funktion1(...)  
{  
    if (setjmp(buf))  
        ...  
    else  
        ...  
}
```



## Einschränkungen



Hochschule RheinMain

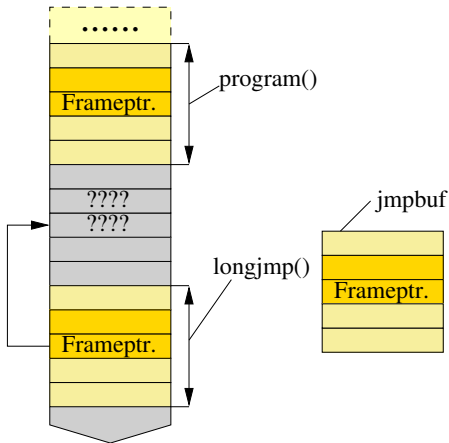
- Achtung: immer nur „von innen nach außen“ springen!

## Beispiel: Unzulässig:

```
static jmp_buf buf;

programm()
{
    funktion1 (...);
    longjmp(buf);
}

funktion1 (...)
{
    if (setjmp(buf))
        ...
    else
        ...
}
```



# Multitasking mit `setjmp()/longjmp()`



- „Missbräuchliche“ (?) Nutzung: Speichern / Wiederherstellen eines *Prozesskontexts*

## Prozesswechsel mit `setjmp()/longjmp()`

```
struct tcb { /* Task Control Block */
    .... /* Beschreibt Zustand eines Prozesses */
    jmp_buf zustand;
    ....
};

void task_wechsel(struct tcb *von, struct tcb *zu) {
    if (!setjmp(von->zustand)) {
        longjmp(zu->zustand);
        assert(0);
    }
}
```

- Vorstellung: N „virtuelle Prozessoren“, davon jeweils einer aktiv
- Umschalten mit `task_wechsel()`

# Kooperatives Multitasking



- Laufender Prozess gibt Prozessor explizit frei

## Kooperatives Multitasking (Prinzip)

```
struct tcb proztab[ANZ_PROZ]; /* Prozesstabelle */
int lfd_proz;                /* aktuell laufender Prozess */

....
if(nichts_zu_tun) {
    task_wechsel(&proztab[lfd_proz],
                 &proztab[(lfd_proz+1) % ANZ_PROZ]);
}
....
```

- + Einfach zu realisieren
- Keine Garantie, dass/wann ein Prozess „drankommt“
- Prozesse müssen „kooperativ“ sein



# Preemptives Multitasking



- Taskwechsel in festen Zeitabständen (z.B. durch Timer-Interrupt)
- Laufender Prozess wird unterbrochen wenn seine „Zeitscheibe“ abgelaufen ist

## Preemptives Multitasking (Prinzip)

```
struct tcb proztab[ANZ_PROZ]; /* Prozesstabelle */
int lfd_proz; /* aktuell laufender Prozess */
/* Timer 0 interrupt service routine: */
ISR(TIMER0_vect)
{
    /* wird (z.B.) alle 10ms aufgerufen */
    alt = lfd_proz;
    neu = lfd_proz = (lfd_proz+1) % ANZ_PROZ;
    task_wechsel(&proztab[alt],
                &proztab[neu]);
}
```

- Prozesse arbeiten „quasi-parallel“
- Besondere Vorkehrungen nötig (Kritische Abschnitte, Reentranz)

# Memory-mapped I/O



- E/A-Bausteine sind wie Speicher an den Bus angeschlossen
  - ▶ E/A-Register erscheinen wie Variablen im Speicher (unter spezieller, vorab bekannter Adresse)
  - ▶ Können ebenso wie solche –z.B. von C-Programmen– gelesen/manipuliert werden

## Beispiel AVR Mikrocontroller: LEDs blinken lassen

```
#include <avr/io.h>
void blink(int led, int times)    /* led = 1, 2, oder 1+2 */
{
    DDRD = (1<<6)|(1<<5); /* PortD6 .. PortD5 → Ausgaenge */
    while(times--){         /* blinke <times> mal */
        PORTD = (led & 3) << 5; /* LEDs an */
        _delay_ms(250);        /* warten... */
        PORTD = 0;             /* LEDs aus */
        _delay_ms(250);        /* warten... */
    }
}
```

# Memory-mapped I/O Register ansprechen (1)



- Frage: Wie spricht man in C überhaupt ein E/A-Register „unter spezieller Adresse“ an ?

Beispiel: (Vereinfachter) Auszug aus `<avr/io.h>`

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)
#define PIND      _SFR_IO8(0x10)
#define DDRD      _SFR_IO8(0x11)
#define PORTD     _SFR_IO8(0x12)
```

→ (z.B.) `DDRD` wird durch den C-Preprozessor substituiert:

- ▶ `DDRD` → `_SFR_IO8(0x11)`
- ▶ `_SFR_IO8(0x11)` → `_MMIO_BYTE((0x11)+0x20)`
- ▶ `_MMIO_BYTE((0x11)+0x20)` → `*(volatile uint8_t*)((0x11)+0x20)`

→ `*(volatile uint8_t*)0x31`

- Allgemein: `*(volatile <Typ> *)<Adresse>`

## Memory-mapped I/O Register ansprechen (2)



- `(*(volatile <Typ> *)<Adresse>)`
- Allgemein: *Typecast*, bzw. *Cast*: „(`<Typ>`)`<Objekt>`”
  - ▶ Erzwingt, dass `<Objekt>` als vom Typ `<Typ>` aufgefasst wird
  - ▶ Beispiel: `(float)10` → Gleitkommazahl 10,0000
- Hier speziell: Zahl `<Adresse>` wird als Zeiger auf Objekte vom Typ `volatile <Typ>` aufgefasst, der ...
- ...durch den vorangestellten „`*`” unmittelbar de-referenziert wird
- `<Typ>` dient zur Bestimmung der Zugriffsbreite:
  - ▶ `char`: 8 Bit (Byte)
  - ▶ `short`: i.d.R. 16 Bit („Wort“)
  - ▶ `int`: i.d.R. 16 oder 32 Bit
  - ▶ `long`: i.d.R. 32 Bit
  - ▶ `long long`: i.d.R. 64 Bit

## Memory-mapped I/O Register ansprechen (2)



- `(*(volatile <Typ> *)<Adresse>)`
- Allgemein: *Typecast*, bzw. *Cast*: „(`<Typ>`)`<Objekt>`”
  - ▶ Erzwingt, dass `<Objekt>` als vom Typ `<Typ>` aufgefasst wird
  - ▶ Beispiel: `(float)10` → Gleitkommazahl 10,0000
- Hier speziell: Zahl `<Adresse>` wird als Zeiger auf Objekte vom Typ `volatile <Typ>` aufgefasst, der ...
- ...durch den vorangestellten „`*`” unmittelbar de-referenziert wird
- `<Typ>` dient zur Bestimmung der Zugriffsbreite:
  - ▶ **char**: 8 Bit (Byte)
  - ▶ **short**: i.d.R. 16 Bit („Wort“)
  - ▶ **int**: i.d.R. 16 oder 32 Bit
  - ▶ **long**: i.d.R. 32 Bit
  - ▶ **long long**: i.d.R. 64 Bit

## Memory-mapped I/O Register ansprechen (3)



- C-Standardtypen `short`, `int`, `long`, `long long` haben keine standardisierte Größe
- C-Programmierungsumgebungen definieren i.d.R. die passenden Typen mit `typedef`
- Z.B. AVR-Libc `<stdint.h>`:
  - ▶ `int8_t`: 8 Bit vorzeichenbehaftet
  - ▶ `uint8_t`: 8 Bit vorzeichenlos
  - ▶ `int16_t`: 16 Bit vorzeichenbehaftet
  - ▶ `uint16_t`: 16 Bit vorzeichenlos
  - ▶ `int32_t`: 32 Bit vorzeichenbehaftet
  - ▶ `uint32_t`: 32 Bit vorzeichenlos
  - ▶ `int64_t`: 64 Bit vorzeichenbehaftet
  - ▶ `uint64_t`: 64 Bit vorzeichenlos

# Memory-mapped I/O Register ansprechen (4)



- Alternative („eleganter“?) Methode:

## Register in Strukturen zusammenfassen

```
typedef struct {  
    volatile uint8_t pin; /* Pegelregister          */  
    volatile uint8_t ddr; /* Data direction reg. */  
    volatile uint8_t port; /* Ausgaberegister    */  
} atmega_port;  
  
#define AVR_PORTA (( atmega_port*)0x39)  
#define AVR_PORTB (( atmega_port*)0x36)  
#define AVR_PORTC (( atmega_port*)0x33)  
#define AVR_PORTD (( atmega_port*)0x30)
```

- Abbilden einer (logisch zusammengehörigen) Gruppe von Registern durch eine Datenstruktur
- Gleichartige E/A-Geräte (hier: Ports A ... D) werden durch die gleiche Datenstruktur mit jeweils anderer Adresse beschrieben

# Memory-mapped I/O Register ansprechen (5)



- Damit: Blink-Routine (s.o.):

## Register in Strukturen zusammenfassen

```
void blink(int led, int times)
{
    AVR_PORTD->ddr = (1<<6)|(1<<5);
    while(times--){
        AVR_PORTD->port = (led & 3) << 5;
        _delay_ms(250);
        AVR_PORTD->port = 0;
        _delay_ms(250);
    }
}
```

- Vorsicht, wenn verschiedene Datentypen in der Struktur gemischt werden müssen
- Ggf. unbenutzte *padding*-Bytes einfügen, um Struktur an Hardware-gegebenheiten anzupassen



# Fallstricke: Objektgröße



p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```


→ (Unerwartete) Vorzeichenerweiterung!

# Fallstricke: Objektgröße



p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

A5	00	00	00

→ (Unerwartete) Vorzeichenerweiterung!

# Fallstricke: Objektgröße



p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

A5	00	00	00
5A	00	00	00

→ (Unerwartete) Vorzeichenerweiterung!

# Fallstricke: Objektgröße



p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

A5	00	00	00
5A	00	00	00
A5	FF	FF	FF

→ (Unerwartete) Vorzeichenerweiterung!

# Fallstricke: Objektgröße



p (0x1000) ⇒

- Beispiel:

```
int *p = (int*)0x1000;
```

```
char a = 0xA5;
```

```
char b = 0x5A;
```

```
p[0] = 0xA5;
```

```
p[1] = b;
```

```
p[2] = a;
```

A5	00	00	00
5A	00	00	00
A5	FF	FF	FF

→ (Unerwartete) Vorzeichenerweiterung!

# Fallstricke: Codeoptimierung (1)



- Codeoptimierung durch den Compiler:
- (An sich vernünftige) Annahme: „Variablenwerte ändern sich nur in Folge von Wertzuweisungen“

## Auszug aus blink-Routine (s.o.)

```
4:  while(times--) {  
5:      PORTD = (led & 3) << 5;  
6:      __delay_ms(250);  
7:      PORTD = 0;  
8:      __delay_ms(250);  
9:  }
```

- Konsequenzen:
  - ▶ Zeile 5: Wertzuweisung an PORTD
  - ▶ Zeile 6: Keine Referenz auf Wert von PORTD
  - ▶ Zeile 7: Erneute Wertzuweisung an PORTD
- Compiler: „Zeile 5 ist überflüssig“ → wird „wegoptimiert“!

# Fallstricke: Codeoptimierung (2)



- Codeoptimierung durch den Compiler:
- (An sich vernünftige) Annahme: „Variablenwerte ändern sich nur in Folge von Wertzuweisungen“

## Auszug aus blink-Routine (s.o.)

```
4:  while(times--) {  
5:  
6:      _delay_ms(250);  
7:      PORTD = 0;  
8:      _delay_ms(250);  
9:  }
```

- Nächster Schritt
  - ▶ Zeile 7: Wertzuweisung an PORTD
  - ▶ Zeilen 4,5,6,8,9: Keine Referenz auf Wert von PORTD innerhalb der Schleife
- Compiler: „Zeile 7 kann aus der Schleife herausgezogen werden“!

## Fallstricke: Codeoptimierung (2)



- Codeoptimierung durch den Compiler:
- (An sich vernünftige) Annahme: „Variablenwerte ändern sich nur in Folge von Wertzuweisungen“

### Auszug aus blink-Routine (s.o.)

```
7:  PORTD = 0;
4:  while(times--) {
5:
6:      _delay_ms(250);
8:      _delay_ms(250);
9:  }
```

- Ergebnis
  - ▶ LEDs werden, bzw. bleiben ausgeschaltet
  - ▶ Es wird  $times \cdot 500ms$  gewartet
- Wohl kaum das erwartete Ergebnis...



## Fallstricke: Codeoptimierung (3)



- Anderes Beispiel: Memory-mapped Lesezugriff

### C-Code

```
struct eageraet {
    uint8_t data;
    uint8_t __unused;
    uint8_t status;
};
.....
struct eageraet *r = ....
.....
while((r->status & (1<<2)) != 0)
    ;
```

### Assembler

```
.....
movl    r, eax
movzbl  2(eax), eax
andl    #4, eax
.L3:    testl   eax, eax
        jne    .L3
.....
```

Lesen von `r->status` wurde aus der Schleife herausgenommen

- Auch hier: keine ersichtliche Änderung von `r->status` innerhalb der Schleife
- Änderung des Wertes erfolgt durch E/A-Gerät und wird hier nicht berücksichtigt

## Fallstricke: Codeoptimierung (3)



- Anderes Beispiel: Memory-mapped *Lesezugriff*

### C-Code

```
struct eageraet {  
    uint8_t data;  
    uint8_t __unused;  
    uint8_t status;  
};  
.....  
struct eageraet *r = ....  
.....  
while((r->status & (1<<2)) != 0)  
    ;
```

### Assembler

#### Endlosschleife!!

```
.....  
movl    r, eax  
movzbl  2(eax), eax  
andl    #4, eax  
.L3: testl  eax, eax  
    jne .L3  
.....
```

Lesen von `r->status` wurde aus der Schleife herausgenommen

- Auch hier: keine ersichtliche Änderung von `r->status` innerhalb der Schleife
- Änderung des Wertes erfolgt durch E/A-Gerät und wird hier nicht berücksichtigt

# Fallstricke: Codeoptimierung (4)



- Weiteres Beispiel: Parallele Programmausführung
- Hier: Interrupt Service Routine und Hauptprogramm

## Interrupt Service

```
static int intcount = 0;

ISR(TIMER0_vector)
{
    ++intcount;
}
```

## Hauptprogramm

```
main() {
    .....
    unsigned int old_count;
    .....
    old_count = intcount;
    while(old_count == intcount)
        ;
    .....
}
```

- Änderung von `intcount` erfolgt durch nebenläufigen Prozess (hier: Interrupt Service) und wird u.U. nicht berücksichtigt
- Wieder eine Endlosschleife!

## Fallstricke: Codeoptimierung (5)



- Noch ein Beispiel: Verzögerungsschleife

### Verzögerungsschleife ..

```
int i;  
for(i = 0; i < 10000; i++)  
    ;
```

### ...ist funktional gleich mit:

```
int i = 10000;
```

- (Hier) gewünschter Effekt der Verzögerung tritt nicht ein

## Fallstricke: Codeoptimierung (6)

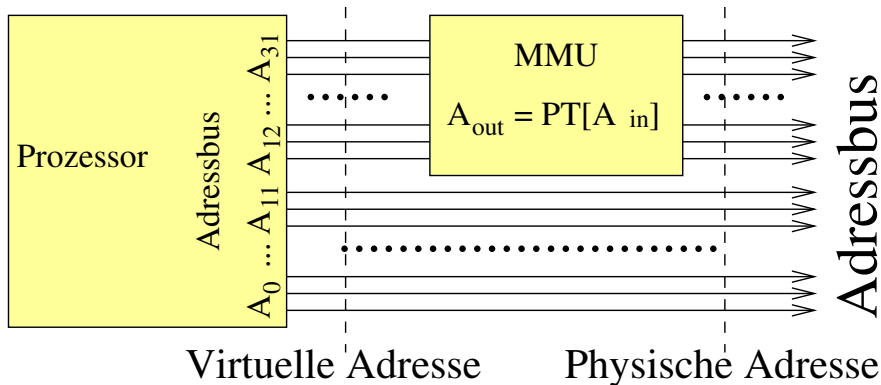


- Abhilfe in allen gezeigten Fällen: Register-Objekte, gemeinsame Daten nebenläufiger Programme oder Zählvariable von leeren Verzögerungsschleifen `volatile` („flüchtig“) erklären
  - Zwingt den Compiler, keine Annahmen über den Inhalt der bezeichneten Variablen zu machen
- Kein „Wegoptimieren“ von Wertzuweisungen oder Lesezugriffen
- Zeiger auf Memory-mapped I/O Register immer `volatile` erklären (siehe auch obige Beispiele zu AVR)

# Fallstricke: Virtuelle Adressierung (1)



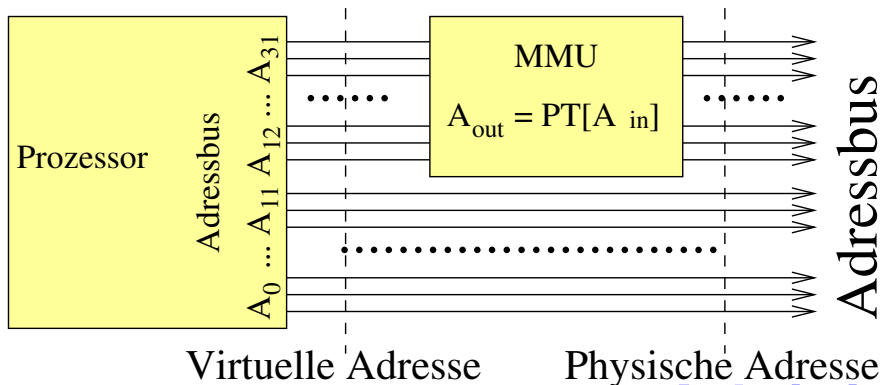
- Rechensysteme mit Speicherverwaltungseinheit (*Memory Management Unit* – MMU)
- Früher externe Hardware, heute ggf. im Prozessor integriert
- Verwendung: Speichervirtualisierung



## Fallstricke: Virtuelle Adressierung (2)



- *Virtuelle* Adressen: Adressen, die vom Adresswerk des Prozessors generiert werden
- *Physische* Adressen: Adressen, die auf dem externen Adressbus anliegen
- MMU ordnet virtuellen Adressen physische Adressen zu



## Fallstricke: Virtuelle Adressierung (3)



- Funktion der MMU ( $n$ -Bit Adressen):

- ▶ Obere  $n - m$  Adressbits der virtuellen Adresse  $A_m^{virt} \dots A_{n-1}^{virt}$  werden durch eine frei programmierbare Abbildung  $PT[x]$  auf physische Adressen  $A_m^{phys} \dots A_{n-1}^{phys}$  abgebildet:

$$A_m^{phys} \dots A_{n-1}^{phys} = PT[A_m^{virt} \dots A_{n-1}^{virt}]$$

- ▶ Untere  $m$  Adressbits der virtuellen Adresse  $A_0^{virt} \dots A_{m-1}^{virt}$  werden nicht verändert:

$$A_0^{phys} \dots A_{m-1}^{phys} = A_0^{virt} \dots A_{m-1}^{virt}$$

→ Speicher ist in „Seitenrahmen“ (*Page Frames*) fester Größe aufgeteilt, denen „Seiten“ (*Pages*) zugeordnet werden

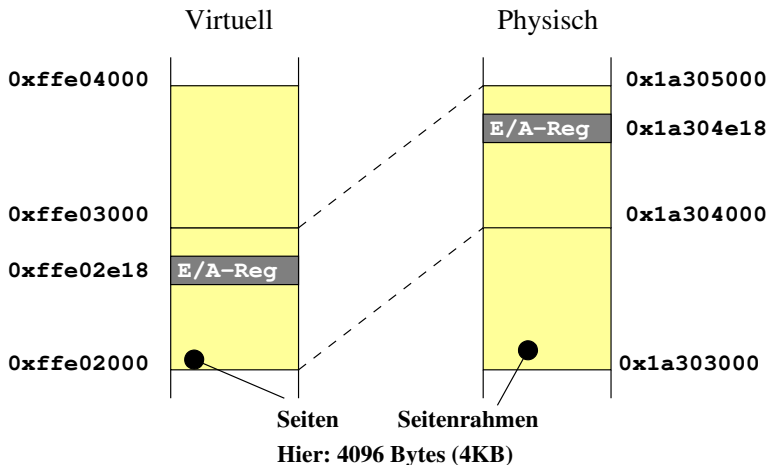
- Gängige Seitengröße:  $m = 12 \rightarrow 4096|_{10}$ , bzw.  $1000|_{16}$  Bytes



## Fallstricke: Virtuelle Adressierung (4)



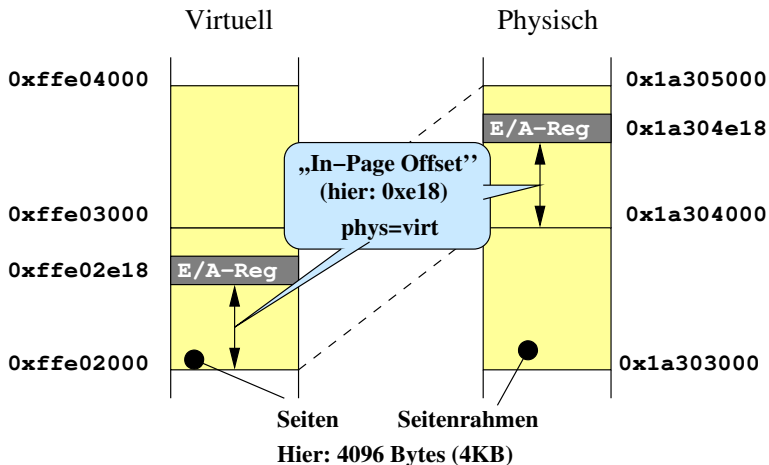
- Zugriff auf memory-mapped E/A an **physikalischer** Adresse muss über die entsprechende **virtuelle** Adresse erfolgen



## Fallstricke: Virtuelle Adressierung (4)



- Zugriff auf memory-mapped E/A an **physikalischer** Adresse muss über die entsprechende **virtuelle** Adresse erfolgen



## Fallstricke: Virtuelle Adressierung (5)



- Kenntnis bzw. Beeinflussung der Abbildung virtuelle → physische Adresse durch entsprechende Programmierung der MMU, i.d.R. mit Hilfe von Betriebssystemfunktionen

### Beispiel: Zugriff aus Linux-Anwendung

```
struct eageraet *r;
int fd;
if ((fd = open("/dev/mem", O_RDWR)) < 0) {
    perror("open(/dev/mem)");
    exit(1);
}
r = (struct eageraet*)mmap((caddr_t)0, Size,
    PROT_READ|PROT_WRITE, MAP_SHARED, fd,
    (off_t)PHYS_ADDRESS);
close(fd);
if ((long)r == -1) {
    perror("mmap()");
    exit(1);
}
r->data = ....
```

## Fallstricke: Virtuelle Adressierung (6)



- Kenntnis bzw. Beeinflussung der Abbildung virtuelle → physische Adresse durch entsprechende Programmierung der MMU, i.d.R. mit Hilfe von Betriebssystemfunktionen
- Cache muss für memory-mapped Adressregionen deaktiviert sein!

### Beispiel: Zugriff aus Linux-Gerätetreiber

```
struct eageraet *r;  
  
r = (struct eageraet*)ioremap_nocache(  
    PHYS_ADDRESS, sizeof(*r));  
r->data = ....
```

# Port-mapped I/O



- Port-mapped I/O:

- ▶ E/A-Register liegen in einem eigenen Adressraum
- ▶ Zugriffe erfordern spezielle Maschinenbefehle (in/out)

→ Nicht direkt in C/C++ implementierbar (i.d.R. stehen aber geeignete Macros zur Verfügung)

## Beispiel: Port-mapped I/O

```
#define PP_DAT 0x378
#define PP_STAT (PP_DAT+2)
#define STAT_ACK 0x02
#define STAT_STROBE 0x01
void Print(char X)
{
    while((inb(PP_STAT) & STAT_ACK) != 0) /* warten..*/ ;
    outb(X, PP_DAT);
    outb(inb(PP_STAT) | STAT_STROBE, PP_STAT);
    while((inb(PP_STAT) & STAT_ACK) == 0) /* warten..*/ ;
    outb(inb(PP_STAT) & ~STAT_STROBE, PP_STAT);
}
```

# Vergleich: Port-mapped / Memory-mapped I/O

- Port-mapped I/O ist i.W. nur bei Prozessoren der Intel ix86-Familie vorhanden
- Nicht Portabel<sup>4</sup>
- Keine Adressübersetzung durch MMU → Keine Unterscheidung zwischen physikalischen und virtuellen Adressen
- Port-mapped I/O durchläuft **nicht** den Cache
- Port-Adressraum i.d.R. kleiner als Speicheradressraum
  - ▶ z.B. 16 Bit gegenüber 32 Bit
  - Geräte mit großem Adressraum (z.B. Grafikkarten) verwenden i.d.R. auch bei Intel-Prozessoren Memory-mapped I/O

---

<sup>4</sup>Linux bietet für nicht-Intel-Maschinen in/out-Funktionen an, die aber durch Memory-mapped I/O implementiert sind