



## 9. Intel IA-32 Prozessoren: Stack und Stackoperationen

- Was ist ein Stack?
- Systemstack
- Stackpointer

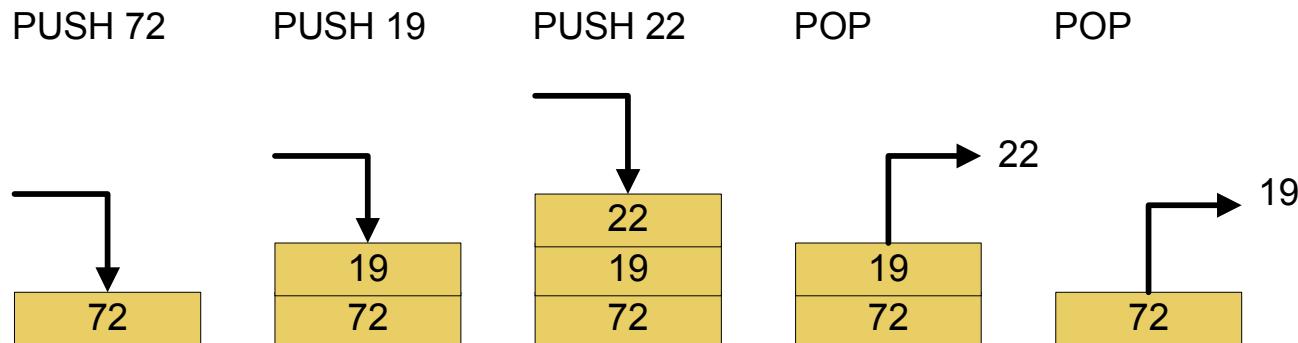


## 9.1 Funktionsweise eines Stacks

**Stack = Stapspeicher = LIFO (Last in – First out).**

**PUSH x** legt ein Element x auf dem Stapel ab

**POP** liest (und entfernt) ein Element vom Stapel





## 9.2 Systemstack

Das System stellt jedem Programm einen speziellen Speicherbereich zur Verfügung, der als Stack arbeitet. → **Systemstack**

Dieser Systemstack hat 2 Hauptfunktionen:

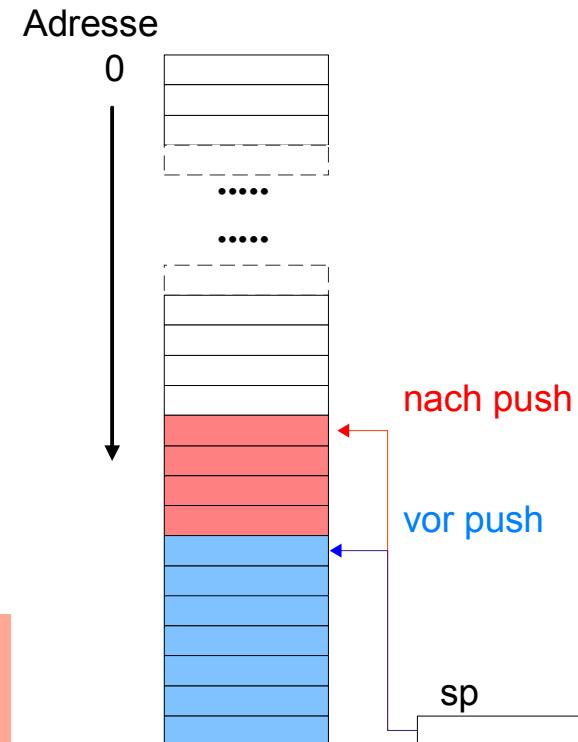
- Übergabe von Funktionsparametern,
- Bereitstellen von Speicher für lokale Variablen von Funktionen

Der Stackboden liegt auf einer hohen Adresse und der Stack wächst in Richtung kleinerer Adressen.

Als **Stackpointer** dient das r13-Register (r13 = sp).

Wird also ein Wert auf dem Stack abgelegt (*Push*), dann verkleinert sich der Wert im Stackpointer.

Wird dein Wert vom Stack entfernt (*Pop*), dann vergrößert sich der Wert im Stackpointer.





## 9.3 Ablegen und Rücklesen von Werten auf dem Systemstack

### 9.3.1 8-Byte-Alignment des Systemstack (Cortex M4)

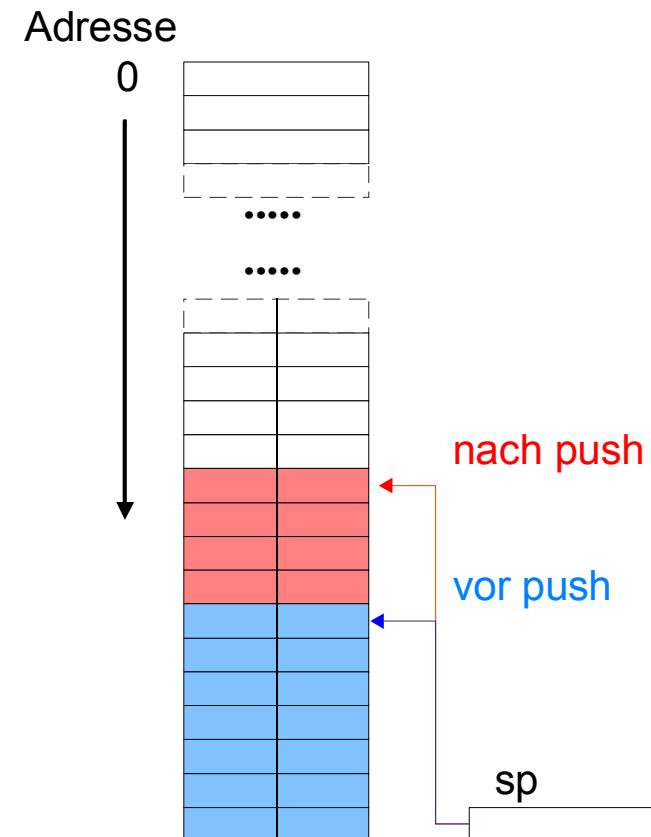
PUSH und POP werden wie folgt realisiert:

```
push {r0, r1, r4-r9}  
      . . .  
      . . .  
pop  {r0, r1, r4-r9}
```

**Der Systemstack des Cortex-M4 ist  
8-Byte-aligned !!!**

Es muss also immer eine gerade Anzahl von  
Registern (je 4 Byte) auf dem Stack abgelegt  
werden.

Anm.: ... auch wenn man nur ein Byte retten  
möchte .





### 9.3.2 Reihenfolge der Parameter auf dem Stack

#### push:

Die Register werden beginnend bei der höchsten Registernummer auf den Stack abgelegt, d.h.

**die kleinste Registernummer steht oben auf dem Stack.**

Die kleinste Registernummer steht also auf der niedrigsten Adresse.

#### Beispiel:

```
push {r1,r0,r9,r8,r4-r7}
```

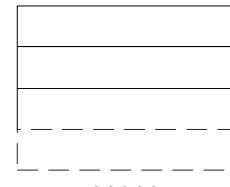
#### pop:

Die Register werden beginnend mit der kleinsten Registernummer vom Stack gelesen.

Die Schreibreihenfolge spielt also keine Rolle.

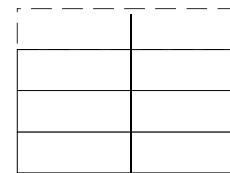
#### Adresse

0



.....

.....



r0	r1
r4	r5
r6	r7
r8	r9

nach push



vor push

sp



### 9.3.2 Alternative Befehle für PUSH und POP (z.B. für Userstacks)

PUSH und POP mehrerer Register können auch mit den Befehlen

**STMFD** (*store multiple, full descending*) und  
**LDMFD** (*load multiple, full descending*)

durchgeführt werden.

```
stmfd      sp!, {r3-r6, r8, r10-r12}    @ PUSH r12,  
          . . .                                @ PUSH r11 .....,  
          . . .  
ldmfd      sp!, {r3-r6, r8, r10-r12}    @ POP r3,  
          . . .                                @ POP r4,
```

**stmfd:** Die Register werden beginnend bei der höchsten Registernummer auf den Stack abgelegt, d.h.  
die kleinste Registernummer steht oben auf dem Stack.

# 10. ARM/Cortex - Prozessor: Unterprogramme

- Zweck von Unterprogrammen
- Übergabemechanismen (globale/lokale Daten, call-by-value, call-by-reference)
- Grundlegende Gedanken
- Thumb-2-Befehle
- Rolle des Stackpointers **sp**
- Registerbehandlung (Übergabe, Retten der Register)
- Parameter- und Ergebnisübergabe



## 10.1 Fragestellungen zu Unterprogrammen

### 10.1.1 *Wozu Unterprogramme?*

#### Zweck von Unterprogrammen:

- Wiederverwendbarkeit von Codeteilen
- Übersichtlichkeit durch Abstraktion
  - Funktionssammlungen (z.B. Vektorrechnung, trig. Funktionen)
  - Softwareschichten (z.B. Hardwareabstraktionsschichten)
  - Anwendungsmodule (z.B. OCR-Modul, Grafikmodule, ...)

#### Definition eines Unterprogramms (Pseudocode):

**Funktionsname (IN: Arg.1, Arg.2 OUT: Erg.1, Erg.2 )**

....

....

**RETURN**



## 10.1.2 Wie können Funktionsargumente übergeben werden ?

### 10.1.2.1 Globale Daten

**Prinzip:** Das aufrufende Programm und das Unterprogramm arbeiten auf den gleichen Daten. Der Speicherort ist hart einprogrammiert und wird nicht explizit übergeben.

#### Vorteile:

- Der Datenort muss nicht explizit übergeben werden, d.h.
- die Datenübergabe muss nicht implementiert werden.

#### Nachteile:

- Es ist nicht offensichtlich, mit welchen Daten das Unterprogramm arbeitet.
- Schlecht wiederverwendbar, da das Unterprogramm das Vorhandensein der globalen Daten voraussetzt (*feste Kopplung*).
- Gefahr von Seiteneffekten, da man schnell die Übersicht darüber verliert, wer die Daten verändert und wann sie verändert werden.



### 10.1.2.2 Call-by-value

**Prinzip:** Das aufrufende Programm über gibt dem Unterprogramm die notwendigen Eingangsdaten als Kopie.

#### Vorteile:

- Ohne Seiteneffekte, da das Unterprogramm keinen Zugriff auf externe Daten hat.
- Der anwendende Programmierer benötigt kein Internwissen über das verwendete Unterprogramm (Geheimnisprinzip = information hiding).

#### Nachteile:

- Auf einzelne Werte beschränkt.  
Strukturierte Datentypen (z.B. Strings, Vektoren, Arrays, ... ) können nur sehr umständlich und ineffizient übergeben werden.



### 10.1.2.3 Call-by-Reference

**Prinzip:** Das aufrufende Programm über gibt dem Unterprogramm  
die Adresse der Ein-/Ausgangsdaten.

#### Vorteile:

- Kontrollierbare Seiteneffekte, da der Programmierer bei Anwendung des Unterprogramms explizit den Zugriff auf die externen Daten erlaubt.
- Der anwendende Programmierer benötigt kein Internwissen über das Unterprogramm (Geheimnisprinzip = information hiding).
- Auch strukturierte Datentypen sind leicht übergebbar.

#### Nachteile:

- Restgefahr von Seiteneffekten bleibt, da das Unterprogramm externe Daten verändert.



## 10.1.3 Wie müsste ein Unterprogrammaufruf arbeiten ?

### 10.1.3.1 Vorüberlegung

#### Sprung zum Unterprogramm:

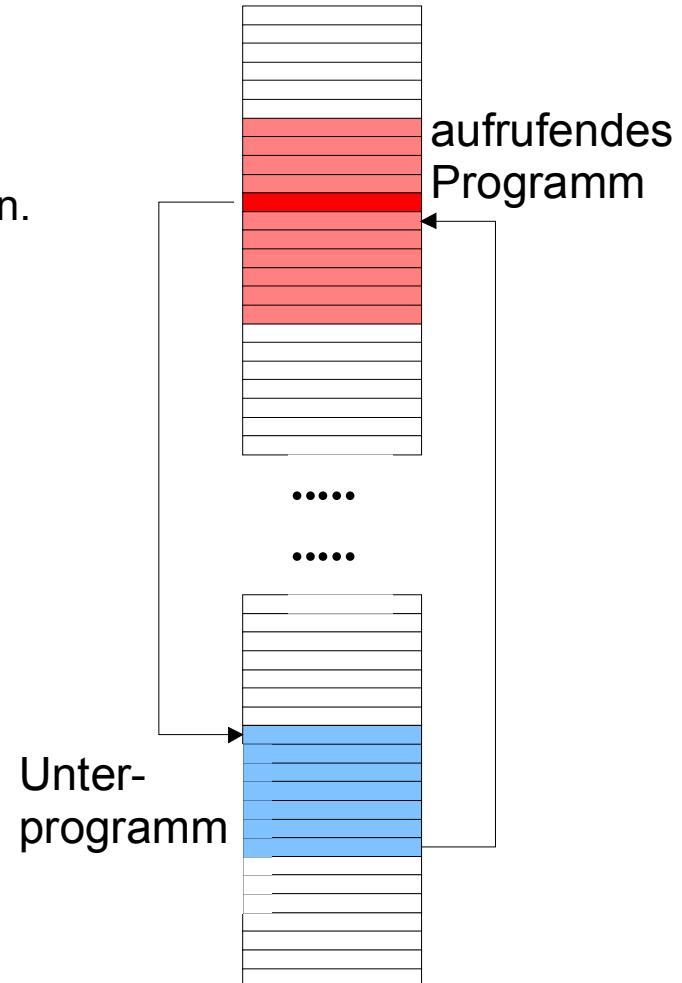
Um zu einem Unterprogramm zu verzweigen, muss der *Program Counter (pc = r15)* auf den ersten Befehl des Unterprogramms gesetzt werden.

#### Rücksprung vom Unterprogramm:

Um mit dem Kontrollfluss nach Abarbeitung des Unterprogramms wieder an der Aufrufstelle fortzusetzen, muss der Program Counter auf den dem Aufruf nachfolgenden Befehl gesetzt werden.

#### Fazit:

Mit dem Sprung in ein Unterprogramm  
muss die Rückkehradresse zum aufrufenden  
Programmteil gespeichert werden.





### 10.1.3.2 Befehlssequenz

Um das Problem des Speicherns der Rücksprungstelle zu lösen, verwendet man die folgenden Befehle:

- **bl <Label>** : Transfer des Kontrollflusses zum Unterprogramm (*relative branch with link*)
- **bx lr** : Rückkehr zum Hauptprogramm (*branch and exchange*)

**bl** rettet die Programmadresse nach dem Sprungbefehl (pc) im Linkregister (lr = r14)  
Danach wird die Programmausführung an der Unterprogrammadresse fortgesetzt!

„**bx lr**“ kopiert die in lr gespeicherte Rücksprungadresse wieder in den pc.

**Anm.:** **bl** erlaubt nur Sprünge von +/-32MB. „*Long branches*“ sind möglich durch:

```
mov lr, pc  
ldr pc, =MySubroutine
```

#### Beispiel:

.....  
**bl MySubroutine**  
.....  
.....

MySubroutine

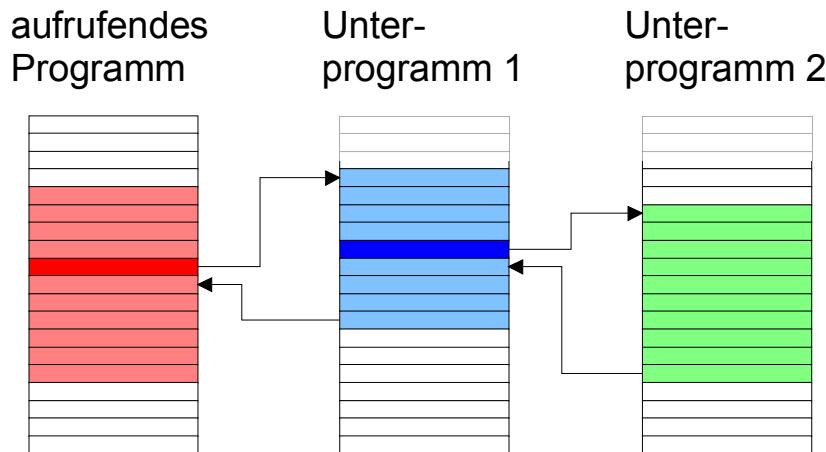
bx lr



## 10.1.4 Geschachtelte Unterprogrammaufrufe (*nested calls*)

### 10.1.4.1 Vorüberlegung

Das Prinzip der Sicherung und Rekonstruktion von Rücksprungadressen muss auch im Fall beliebig geschachtelter Unterprogrammaufrufe funktionieren.



Sicherung und Rekonstruktion haben LIFO-Charakter (Last-in, First-out).  
**Ein Stack ist also die ideale Datenstruktur für diese Aufgabe.**



### 10.1.4.2 Befehlssequenz

Im Unterprogramm

- wird zunächst das Linkregister auf den Stack gerettet (push),
- der Unterprogrammcode wird ausgeführt und anschließend
- wird das Linkregister wieder restauriert (vom Stack gelesen, pop).

**Beispiel:**

```
.....  
.....  
bl MySubroutine  
.....  
.....
```

MySubroutine

```
push { r1, lr }  
.....  
.....  
pop sp!, { r1, lr }  
bx lr
```

**Achtung:** Es ist zu bedenken, dass immer eine gerade Anzahl von Registern auf den Stack gerettet wird ! ( deswegen hier : push {r1, lr} )



## 10.1.5 Registerrettung

### 10.1.5.1 Vorüberlegung

- Beim Unterprogrammsprung wird nichts gerettet:

- keine Registerinhalte,
- keine Condition codes,
- keine lokale Variablen.

Wie rettet man  
Register ?

**Also:** Der Programmierer trägt die Verantwortung für die Rettung von Registerinhalten, Condition codes und lokalen Variablen. !

- Der Stackpointer **sp** sollte nur für Unterprogramme genutzt werden. Andernfalls ist die Gefahr falscher Rücksprungadressen sehr groß.

**Also:** Finger weg vom Register sp (Stackpointer). !



### 10.1.5.2 Retten der verwendeten Register

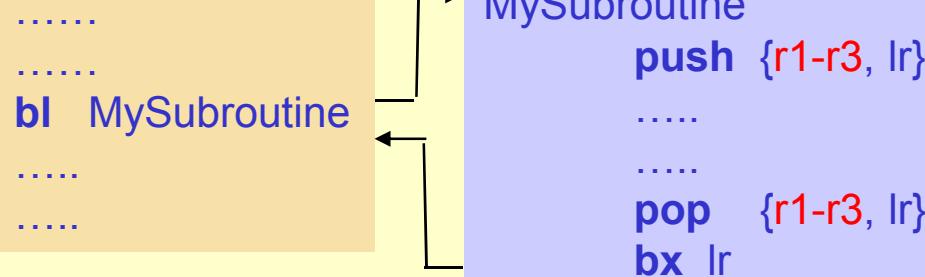
#### Konsistenz der CPU-Register wahren

Zur Vermeidung unerwarteter Effekte (Seiteneffekte) nach Unterprogrammen sollte darauf geachtet werden, dass die im Unterprogramm verwendeten Register nach dem Rücksprung aus dem Unterprogramm wieder die gleichen Inhalte haben wie vor dem Unterprogrammaufruf.

**Lösung:** Die verwendeten Register zu Beginn des Unterprogramms auf den Stack retten und vor dem Rücksprung zum Hauptprogramm wieder restaurieren.

**Ausnahme:** Die für die Parameterübergabe verwendeten Register.

**Beispiel:**





## 10.2 Weitere konzeptionelle Gedanken zu Unterprogrammen

### 10.2.1 Verschiedenen Übergabemechanismen

Der Rückgabewert einer Funktion wird meist über Register **r0** zurückgegeben.

Für die Eingabewerte (Parameter) stehen folgende Mechanismen zur Verfügung:

Was	Wie	Register r0, r1, r2, r3 (ARM) - begrenzte Parameterzahl	Stack + unbegrenzte Parameterzahl
<b>Datenkopie</b> <i>Call-by-value</i> + Seiteneffektfrei + information hiding - nur einfache Datentypen		CbV-Reg	CbV-Stk
<b>Adressen</b> <i>Call-by-reference</i> + beliebige Datentypen + information hiding - Restgefahr von Seiteneffekten		CbR-Reg	CbR-Stk



## ÜBUNG: Parameterübergabe: „Call-by-value“ über Register (CbV-Reg)

Schreiben Sie ein Unterprogramm, welches die *Fakultät*  $n!$  zu einer gegebenen Zahl  $n$  berechnet.

Verwenden Sie das Register r0 zur Übergabe des Eingabeparameters ( $n$ ) und des Ergebnisses ( $n!$ ).

Was sind die Beschränkungen dieser Methode?



## ÜBUNG: Parameterübergabe: „Call-by-reference“ über Register (CbR-Reg)

Schreiben Sie ein Unterprogramm, welches die Länge eines Strings bestimmt.

Zur Übergabe des Strings soll das Register r0 verwendet werden.  
Das Ergebnis soll ebenfalls in r0 stehen.

Was sind die Beschränkungen dieser Methode?



## 10.2.2 Parameterübergabe über den Stack

### 10.2.2.1 Einleitende Anmerkungen

Eine übliche Konvention beim ARM Thumb-Befehlssatz ist, die ersten vier Parameter über r0 – r3 zu übergeben und alle weiteren Parameter über den Stack.

Dies ermöglicht für die meisten Unterprogramme (0 ... 4 Parameter) einen sehr schnellen Unterprogrammsprung (mit wenig Overhead), ermöglicht aber auch mehr (5 ... beliebig viele) Übergabeparameter (mit etwas mehr Overhead).

Bei vielen anderen Prozessoren (mit weniger Registern) ist die Parameterübergabe über den Stack der Standard.

Die Parameter r4 - r11 stehen üblicherweise als lokale Variablen zur Verfügung. Werden mehr lok. Variablen benötigt, so kann auch hierfür der Stack verwendet werden (s.u.).

Die Parameterübergabe nach Konvention ist dann einzuhalten, wenn die Assembler-Unterprogramme auch von C-Programmen aufgerufen werden sollen.



### 10.2.2.2 Realisierungsidée 1 (*nicht gut*)

; Aufrufendes Programm

```
push {r3, r4}          ; PUSH VarA und VarB
bl MySubroutine
add sp, #8            ; Stack korrigieren
```

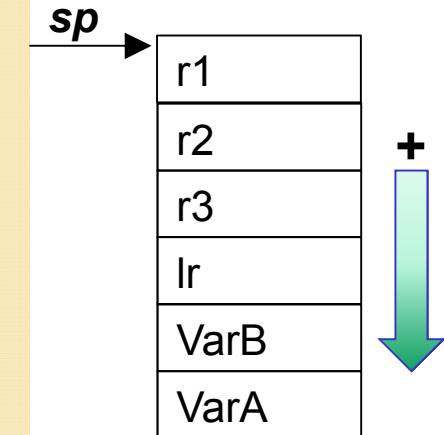
#### MySubroutine:

- 2 Parameter
- Übergabe über Stack
- intern genutzte Register:  
r1 und r2
- Ergebnis in r0

; geändertes Unterprogramm

#### MySubroutine

```
push {r1-r3, lr}      ; PUSH, Register retten
ldr r2, [sp, #16]      ; [r2] ← VarB
ldr r1, [sp, #20]      ; [r1] ← VarA
.....                 ; irgendwas berechnen .....
.....                 ; und Ergebnis → [r0]
pop {r1-r3, lr}        ; POP, Register restaurieren
bx lr
```



## ***Worin liegt der Nachteil dieses Ansatzes ?***

Werden im Unterprogramm weitere Register auf den Stack kopiert, dann müssen alle Versatzwerte (*Offsets*) korrigiert werden !!

- schwer wartbar
- sehr fehleranfällig



### 10.2.2.2 Realisierungsidée 2 ( *besser, aber auch noch nicht gut* ) → fp

#### # Aufrufendes Programm

```
push {r3, r4}           ; PUSH VarA und VarB
bl MySubroutine
add sp, #8             ; Stack korrigieren
```

#### MySubroutine:

- 2 Parameter
- Übergabe über Stack
- intern genutzte Register:  
r1 und r2
- Ergebnis in r0

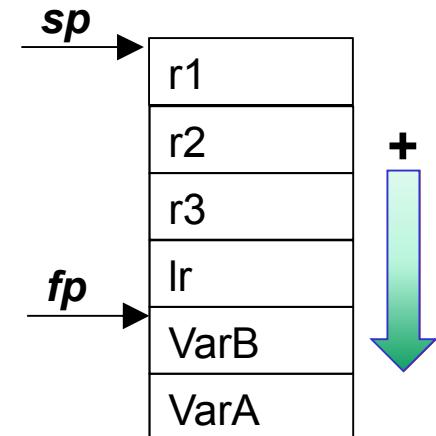
#### Unterprogramm

##### MySubroutine

```
mov fp, sp            ; aktuelle Stackpos. merken
push {r1-r3, lr}      ; PUSH, Register retten
ldr r2, [fp, #0]       ; [r2] ← VarB
ldr r1, [fp, #4]       ; [r1] ← VarA
.....                ; irgendwas berechnen .....
.....                ; und Ergebnis → [r0]
pop {r1-r3, lr}        ; POP, Register restaurieren
bx lr
```

**fp** ist der sog.  
**Framepointer**

$$fp = r11$$





## **Worin liegt der Nachteil dieses Ansatzes ?**

Im Falle von „*nested calls*“ wird der alte Inhalt des Framepointer fp überschrieben.

→ nicht schachtelbar



### 10.2.2.3 Realisierungsweise 3 ( so geht's !!! )

#### # Aufrufendes Programm

```
push {r3, r4}           ; PUSH VarA und VarB
bl MySubroutine
add sp, #8             ; Stack korrigieren
```

#### MySubroutine:

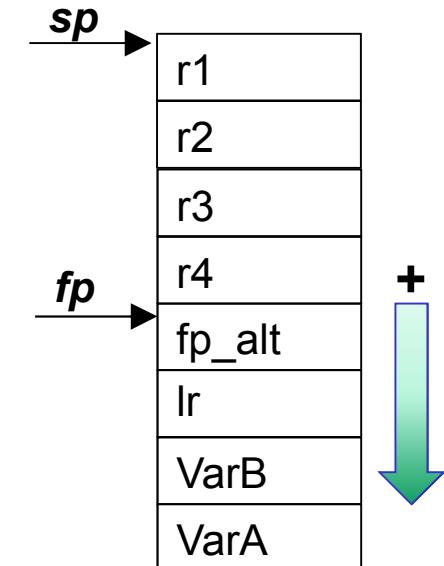
- 2 Parameter
- Übergabe über Stack
- intern genutzte Register:  
r1- r4
- Ergebnis in r0

#### Unterprogramm

##### MySubroutine

```
push {fp, lr}           ; PUSH fp , fp_alt retten
mov fp, sp
push {r1-r4}            ; aktuelle Stackpos. merken
                        ; PUSH, Register retten
ldr r2, [fp, #8]         ; [r2] ← VarB
ldr r1, [fp, #12]         ; [r1] ← VarA
.....                  ; irgendwas berechnen .....
.....                  ; und Ergebnis → [r0]
```

```
pop {r1-r4}              ; POP, Register restaurieren
pop {fp,lr}               ; POP fp u. lr restaurieren
bx lr
```





## ÜBUNG: Parameterübergabe über Stack (1)

Schreiben Sie ein Unterprogramm, welches folgende Berechnung durchführt:

$$\text{Erg} = (z1+z2)*(z3+z4)$$

- Die Parameter z1 ... z4 sollen über den Stack übergeben werden.
- Das Ergebnis soll über r0 zurückgegeben werden.
- Alle verwendeten Register sollen gerettet und wieder restauriert werden.

## 10.2.3 Lokaler Speicher für Unterprogramme

### 10.2.3.1 Was ist „lokaler Speicher“ ?

**Lokaler Speicher ist ...**

Speicherplatz, der bereitgestellt wird, wenn das Unterprogramm startet und der wieder freigegeben wird, wenn das Unterprogramm endet.

**Vorteile:**

- effiziente Speichernutzung
- Kapselung interner Daten
- es können gleichzeitig mehrere Instanzen eines Unterprogramms existieren ohne sich gegenseitig zu beinflussen  
(z.B. Multitasking, rekursive Unterprogramme)



### 10.2.3.2 Realisierung

#### # Aufrufendes Programm

```

str    r3, [sp, #-4]!      ; PUSH VarA
str    r4, [sp, #-4]!      ; PUSH VarB
bl     MySubroutine
add    sp, #8              ; Stack korrigieren

```

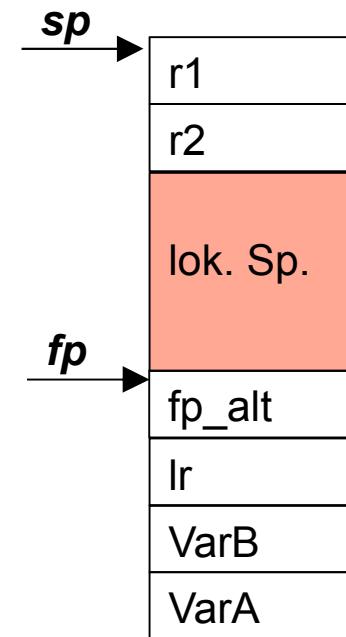
Über den Framepointer fp kann man bequem auf den lokalen Speicher zugreifen !

#### MySubroutine

```

push   {fp, lr}            ; fp und lr retten
mov    fp, sp              ; aktuelle Stackpos. merken
sub    sp, #16              ; 16 Byte lok. Speicher (nx8Byte)
push   {r1-r2}              ; PUSH, Register retten
ldr    r2, [fp, #8]          ; [r2] ← VarB
ldr    r1, [fp, #12]          ; [r1] ← VarA
.....                   ; irgendwas berechnen .....
.....                   ; und Ergebnis → [r0]
pop    {r1-r2}              ; POP, Register restaurieren
mov    sp, fp              ; sp unter lok. Speicher setzen
pop    {fp,lr}              ; POP fp u. lr restaurieren
bx    lr

```





## ÜBUNG: Parameterübergabe über Stack (2) und lok. Speicher

Schreiben Sie ein Unterprogramm, welches folgende Berechnung durchführt:

$$\text{Erg} = a (1 + b^2 + b^3 + b^4)$$

- Die Parameter a und b (word) sollen über den Stack übergeben werden.
- Das Ergebnis soll über r0 zurückgegeben werden.
- Alle verwendeten Register sollen gerettet und wieder restauriert werden.
- Für Zwischenergebnisse soll 16 Byte lokaler Speicher bereitgestellt werden.



## 11. ARM : Ein- und Ausgabeschnittstellen und -befehle

- Memory-Mapped-Ein-/Ausgabe
- Polling
- Steuerregister
- Beispiel: General Purpose Input/Output Port (GPIO)

## 11.1 Einführung

### 11.1.1 **Basisschnittstellen**

Für die Interaktion des Prozessors mit der Aussenwelt Schnittstellen für die Ein-/Ausgabe von Signalen benötigt.

Grundlegende Interfaces sind Beispielsweise:

- **GPIO** (General Purpose Input/Output) :
  - Ansteuerung von LED, Lampen, Relais, Signalleitungen, .....
  - Abfrage von Schaltern, Tastaturen, binären Signalpegeln, .....
- **ADC** (Analog Digital Converter)
  - Messen einer analogen Eingangsspannung (z.B. Messwertaufnehmer)
  - Abtasten (Samplen) von Zeitsignalen (z.B. von Musik)
- **DAC** (Digital Analog Converter)
  - Umwandeln einer Digitalzahl in eine analoge Ausgangsspannung
  - Ausgabe von Tönen und Musik

## 11.1.2 Komplexere Schnittstellen

- **UART** (*Universal Asynchronous Receiver Transmitter*) :
  - serielle Schnittstelle
  - EIA 485 (früher RS 232)
- **PWM** (*Pulse Width Modulation*)
  - z.B. Ansteuerung von Motoren,
- **I2C** (*Inter-Integrated Circuit*)
  - 2-Draht-Bus, Master-Slave-Bus,
  - für einfache Gerätekommunikation
  - Beispiel: LEGO-NXP-Roboter für Sensoren und Antriebe
- **CAN** (*Controller Area Network*)
  - Feldbusssystem (bis 40m bei 1MBit/s, bis zu 110 Teilnehmer, Linienstruktur)
  - Beispiel: Standard bei der Kfz-internen Gerätekommunikation
- **Ethernet**
- **USB**



## 11.2 I/O-Ports (GPIO)

Die Ein- und Ausgabe von Werten sowie die Programmierung der I/O-Bausteine erfolgt über die Steuer- und Ein-/Ausgaberegister der I/O-Bausteine.

Diese Register werden ähnlich wie Speicher über Adressen angesprochen.

Zwei Prinzipien sind üblich:

- **Memory-mapped-I/O:**

- I/O-Baustein und Speicher teilen gemeinsam den Adressraum.
- Die Register der I/O-Bausteine werden wie Speicher angesprochen ( $\rightarrow \text{ldr, str}$ ).
- **Beispiel:** ARM-Cortex, Motorola 68000

- **I/O-mapped:**

- Die I/O-Bausteine haben einen eigenen Adressraum.
- Die Register der I/O-Bausteine werden über eigene Befehle angesprochen ( $\rightarrow \text{in / out}$ )
- **Beispiel:** Intel IA-32-Prozessoren



## 11.3 Reaktion auf Eingangsereignisse

Für die Detektion von Eingangsereignissen (z.B. Tastendruck) sowie die daran anschließende Reaktion darauf können verschiedene Mechanismen angewendet werden:

- **Polling:**

- Zyklisches Abfragen eines Eingangswertes, solange bis eine bestimmte Bedingung erfüllt ist

- **Interrupt:**

- asynchrone Unterbrechung der normalen Programmausführung durch EA-Bausteine über spezielle Steuerleitungen (Interruptleitung):

- Sicherung des aktuellen Prozessorzustandes (Register, Flags)
- Ausführung der Interruptroutine
- Wiederherstellung des gespeicherten Prozessorzustandes
- Fortführung des unterbrochenen Programms



## 11.4 Ein-/Ausgabe

### 11.4.1 Cortex-M4 : General-Purpose-Input/Output-Ports

- 9 Ports mit je 16 Bit : GPIO A, GPIO B, ..... GPIO I
- Bitweise als Input- oder Outputports programmierbar
- verschiedene I/O-Taktraten programmierbar 2MHz, 25MHz, 50MHz, 100MHz
- Adressbereiche der Ports

0x4002 2000 - 0x4002 23FF	GPIOI
0x4002 1C00 - 0x4002 1FFF	GPIOH
0x4002 1800 - 0x4002 1BFF	GPIOG
0x4002 1400 - 0x4002 17FF	GPIOF
0x4002 1000 - 0x4002 13FF	GPIOE
0x4002 0C00 - 0x4002 0FFF	GPIOD
0x4002 0800 - 0x4002 0BFF	GPIOC
0x4002 0400 - 0x4002 07FF	GPIOB
0x4002 0000 - 0x4002 03FF	GPIOA



## ***Initialisierungssequenz der Ports beim Cortex-M4***

1. Takt der verwendeten Ports einschalten
2. Zu jedem Port
  - verwendete Pins festlegen
  - Modus festlegen (Input, Output)
  - Ausgangsbeschaltung festlegen (Pullup, Pulldown, ....)
  - Taktrate festlegen

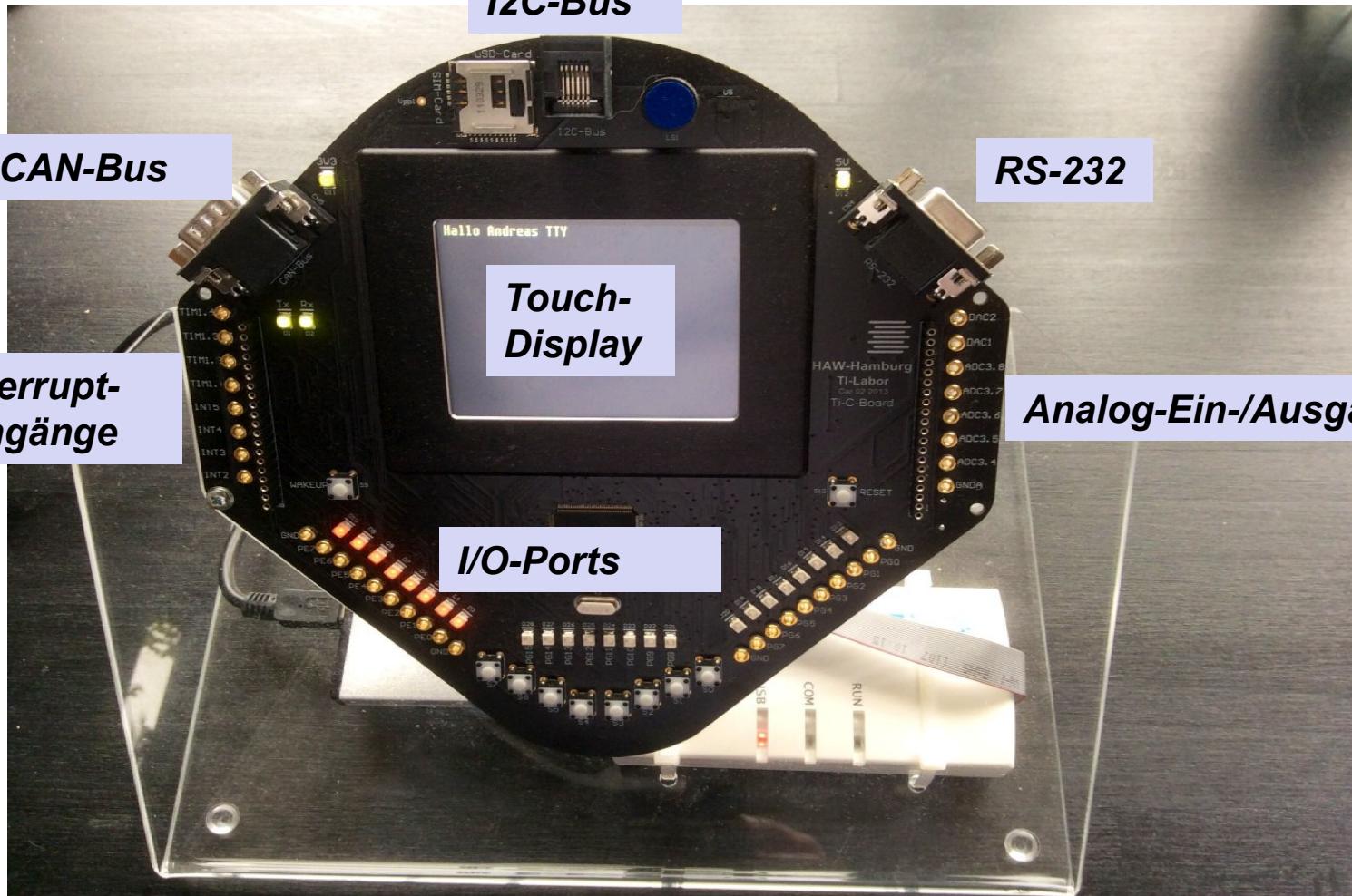
**Beim TI-Board steht für die korrekte Initialisierung  
der Ports die Funktion**

**`Init_TI_Board()` → `Init_IO()`**

**zur Verfügung.**

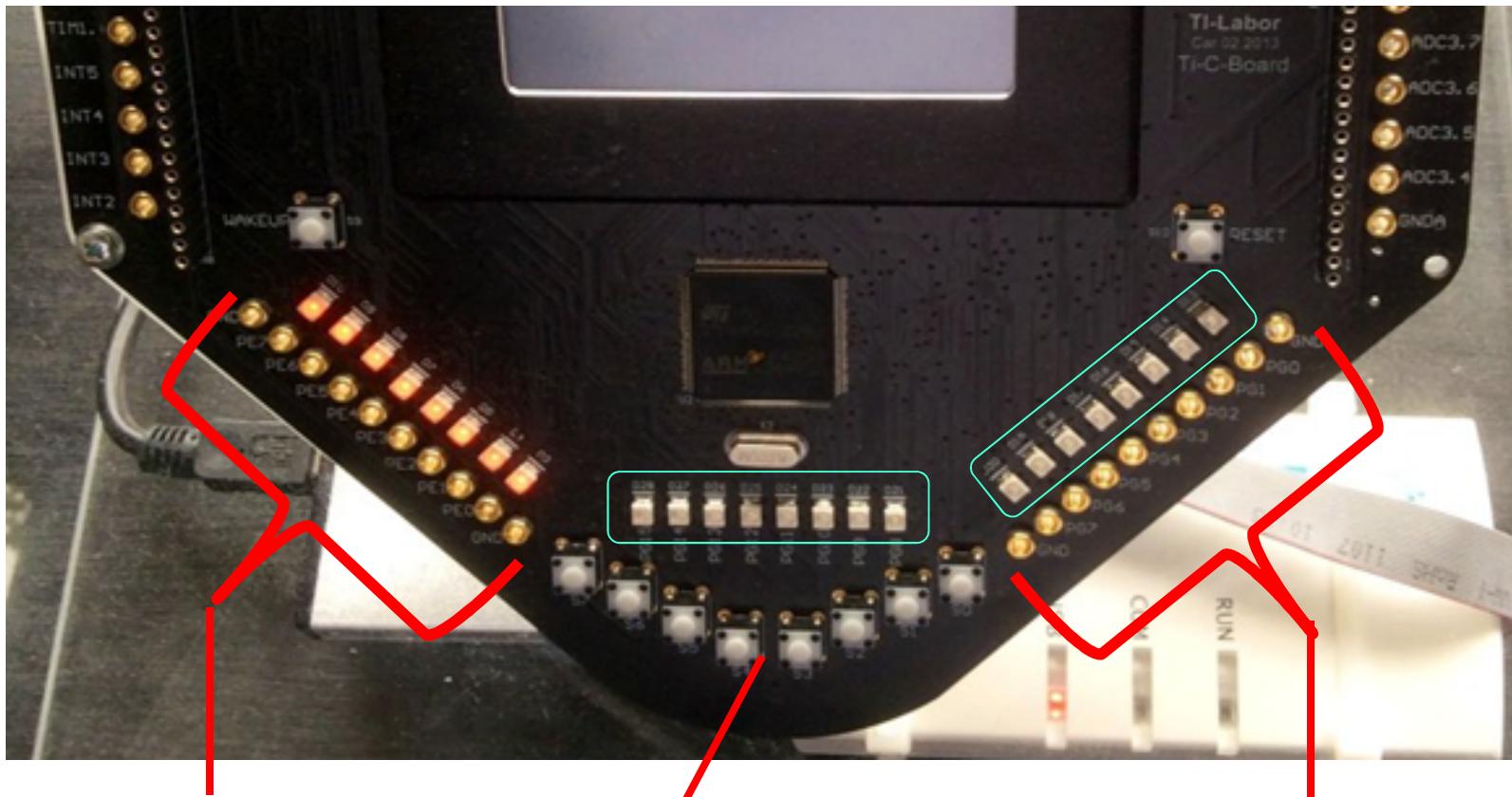


### 11.4.1 TI-Board





- Port GPIO\_E: 8 Eingänge, auf Buchsen und LED herausgeführt (Bit 0 ... 7)
- Port GPIO\_G: 8 Ausgänge, auf Buchsen herausgeführt (0 ... 7)  
16 Ausgänge, auf LED herausgeführt (0 ... 15)
- Initialisierung mit `Init_TI_Board()`
- 50 MHz I/O-Taktrate
- **Achtung:** offene Eingänge haben den Wert 1



Taster zu den Eingängen



### 11.4.2 Ein-/Ausgabe auf den GPIO-Ports

Die Ansteuerung bzw. das Auslesen eines 16-bit-GPIO-Ports erfolgt über Register mit den folgenden Adressen:

#### Lesen von Port E

GPIO\_E\_PIN ((= 0x 40 02 10 10) : Lesen der Eingangswerte von GPIO\_E

#### Schreiben auf Port G

GPIO\_G\_PIN (= 0x 40 02 18 10) : Lesen der aktuellen Ausgangswerte

GPIO\_G\_SET (= 0x 40 02 18 18) : Setzen von Ausgangswerten

GPIO\_G\_CLR (= 0x 40 02 18 1A) : Löschen von Ausgangswerten

#### **Beispiel:**

2\_0000000000000001 → GPIO\_G\_SET setzt Bit 0  
und lässt alle anderen Bits unverändert

2\_0000000000000001 → GPIO\_G\_CLR löscht Bit 0  
und lässt alle anderen Bits unverändert



## **Beispiel:** GPIO\_E lesen und auf GPIO\_G ausgeben (Eingabe reflektieren)

### **1. Portadressen definieren**

```
PERIPH_BASE      equ      0x40000000
AHB1PERIPH_BASE equ      (PERIPH_BASE + 0x00020000)

;blaue LEDs: output
GPIOG_BASE       equ      (AHB1PERIPH_BASE + 0x1800)
GPIO_G_SET        equ      GPIOG_BASE + 0x18
GPIO_G_CLR        equ      GPIOG_BASE + 0x1A
GPIO_G_PIN         equ      GPIOG_BASE + 0x10

;rote LEDs / Taster: input
GPIOE_BASE       equ      (AHB1PERIPH_BASE + 0x1000)
GPIO_E_PIN        equ      GPIOE_BASE + 0x10
```



## 2. Asm-Programm

```
Reflect_PortE_to_PortG PROC
loop
    ldr r0, =GPIO_E_PIN           ; Eingangsadr. laden
    ldrb r3, [r0]                 ; Eingangswert lesen

    mov r2, #0xFFFF               ; Löschbits laden
    ldr r1, =GPIO_G_CLR          ; Adr. CLR-Register laden
    strh r2, [r1]                 ; LED 0...15 löschen

    ldr r1, =GPIO_G_SET          ; Adr. SET-Register laden
    strh r3, [r1]                 ; LED setzen

    b loop
ENDP
```