



Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Hardwarenahe Softwareentwicklung

Subroutinen

V5.1, ©2023 roger.weber@bfh.ch

Lernziele

Sie sind in der Lage:

- ▶ Die Mechanismen beim Aufruf einer Subroutine zu erklären.
- ▶ Subroutinen korrekt zu implementieren und aufzurufen.



Inhaltsverzeichnis

1. Eigenschaften und Aufruf von Subroutinen
2. Stack-Operationen
3. Rücksprungadressen
4. Retten von Registern
5. Parameterübergabe
6. Lokale Variablen
7. Häufige Fehlerquellen
8. Makros oder Subroutinen?

Eigenschaften und Aufruf von Subroutinen

Eigenschaften und Vorteile von Subroutinen

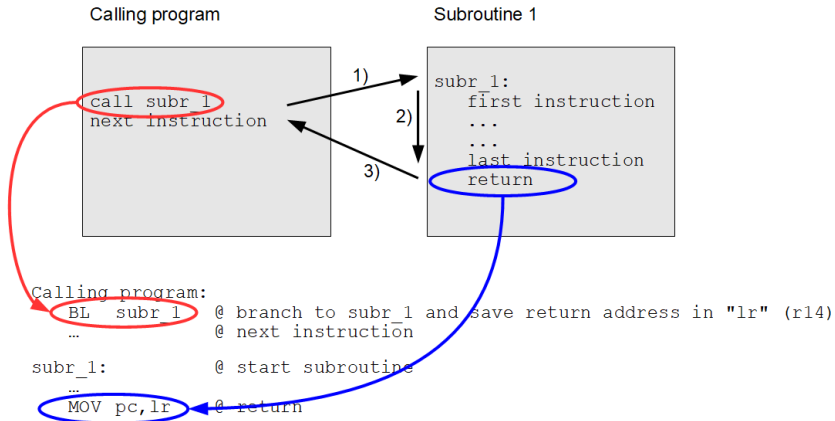
Subroutinen in Assembler entsprechen Funktionen in C.



- ▶ Was sind die Eigenschaften von Subroutinen / Funktionen?
- ▶ Was sind die Vorteile von Subroutinen / Funktionen?

Aufruf von Subroutinen

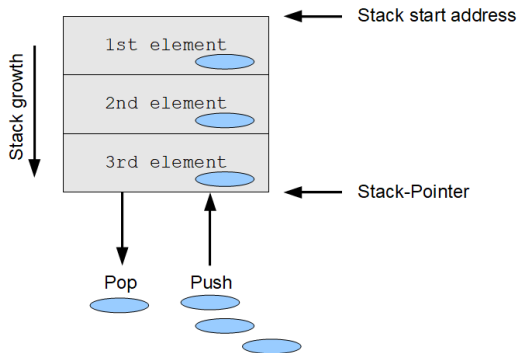
- Für den Aufruf einer Subroutine wird bei ARM die Branch-and-Link Instruktion (**BL**) verwendet.



Stack-Operationen

Stack

- ▶ Ein Stack (Stapel) ist eine dynamische Datenstruktur, die nach dem Prinzip “**Last In First Out**“ arbeitet.
- ▶ Typische Operationen:
 - ▶ Initialisierung
 - ▶ **Push** (ein Datenelement auf den Stack legen)
 - ▶ **Pop** (das letzte Datenelement vom Stack holen)
- ▶ Der **Stack-Pointer** zeigt auf das Ende des Stacks, d.h. dorthin, wo die nächste Operation ausgeführt wird.



Stack

- ▶ Es gibt verschiedene Ansätze, einen Stack zu implementieren:
- ▶ **descending** (nach unten, zu tieferen Adressen) oder **ascending** (nach oben, zu höheren Adressen)
- ▶ **full** (SP → zuletzt geschriebenes Element) oder **empty** (SP → nächstes freies Element)
- ▶ ARM Architecture Procedure Call Standards (AAPCS) → “**full descending**“ Stack.
- ▶ Push-Operation: **PUSH**-Instruktion (als Variante auch STR oder STMDB).
- ▶ Pop-Operation : **POP**-Instruktion (als Variante auch LDR oder LDMIA).

Stack initialisieren

- ▶ Die Initialisierung des Stacks ist je nach CPU unterschiedlich.
- ▶ Cortex-Mx: Der Stackpointer wird mit dem ersten Eintrag in der Vektortabelle initialisiert:

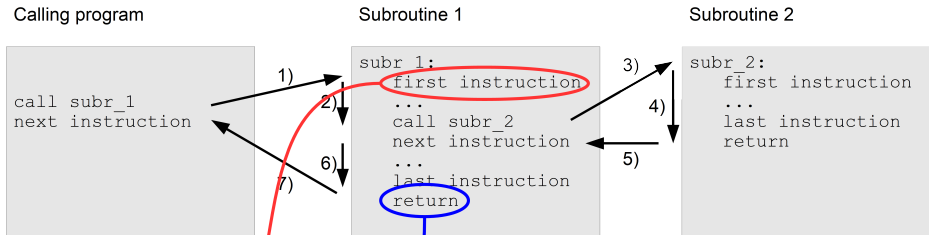
```
/* _____ Section .isr_vector _____ */  
/**  
 * \ brief The minimal vector table for a Cortex M4.  
 */  
...  
g_pfnVectors:  
.word _estack           @ start address stackpointer  
.word Reset_Handler    @ start address reset handler  
...
```

- ▶ Der Wert `_estack` wird im Linkerscript "stm32f4_flash.ld" definiert:

```
/* Highest address of the user mode stack */  
_estack = 0x20020000 ; /* end of 128 K RAM */
```

Rücksprungadressen

Verschachtelte Aufrufe von Subroutinen

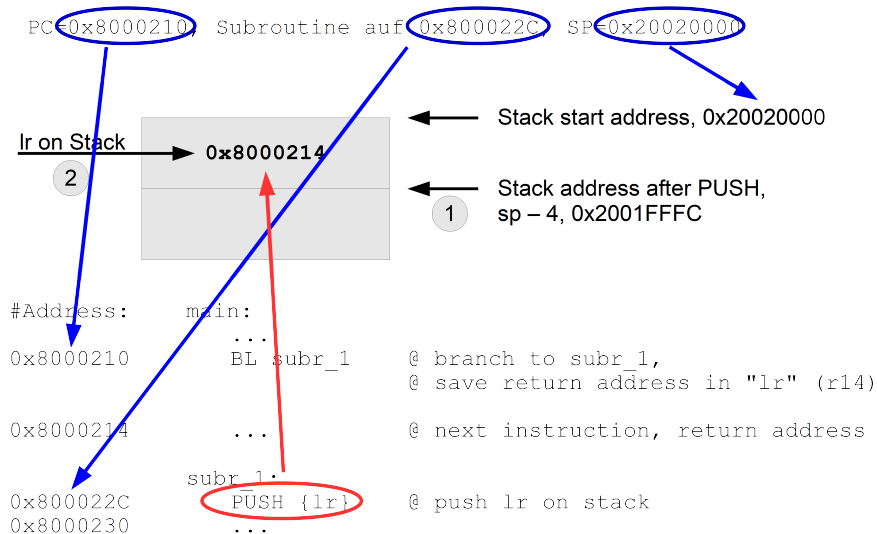


```
main:
    BL subr_1    @ branch to subr_1, save return address in "lr" ( r14 )
    ...         @ next instruction

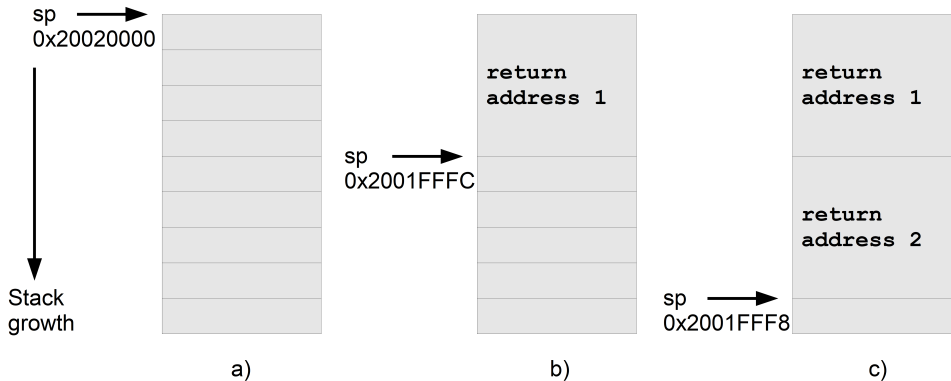
subr_1:
    PUSH {lr}    @ push lr on stack
    ...         @ code subr_1
    BL subr_2    @ call subr_2
    ...
    POP {pc}     @ return, pop return address from stack to pc and increment sp
```

Rücksprungadresse auf dem Stack retten

► Beispiel:



Rücksprungadresse auf dem Stack retten



- a) Stack im aufrufenden Programm
- b) Stack innerhalb von "subr1"
- c) Stack innerhalb von "subr2"

Wichtige Instruktionen für Subroutinen

Anweisung	Syntax-Beispiel	Operation
Branch-and-Link	BL subr_name	$lr \leftarrow$ Rücksprungadresse $pc \leftarrow pc + \text{Sprungdistanz}$
push single	PUSH {lr}	$sp \leftarrow sp - 4$ Stack \leftarrow Rücksprungadresse
push multiple	PUSH {r0, r2-r4, lr}	$sp \leftarrow sp - 5 \cdot 4$ Stack \leftarrow Rücksprungadresse + Arbeitsregister
pop single + Rücksprung	POP {pc}	$pc \leftarrow$ Rücksprungadresse $sp \leftarrow sp + 4$ return
pop multiple + Rücksprung	POP {r0, r2-r4, pc}	$pc \leftarrow$ Rücksprungadresse Arbeitsregister \leftarrow Stack $sp \leftarrow sp + 5 \cdot 4$ return

Retten von Registern

Retten von Registern

► Beispiel, um die Problematik zu illustrieren:

```
start:
    MOV    r4,#10                @ initialize loop variable r4, 10 loops
loop:
    BL     subr3                 @ branch to subroutine
    SUBS   r4,r4,#1              @ decrement loop variable
    BNE    loop                  @ r4!=0? yes -> branch to label loop
    ...

subr3:
    PUSH   {lr}                  @ push lr on the stack
    ...
    MOV    r4,#0                 @ modify r4
    ...
    POP    {pc}                  @ pop return address
```

Retten von Registern

- ▶ Es gilt das Verursacherprinzip: Wenn eine Subroutine Register verändert, muss sie deren Inhalt zuerst auf dem Stack retten und am Schluss wieder herstellen.
- ▶ Ausnahmen gemäss Spezifikation AAPCS

```
subr3:  PUSH    {r4, lr}      @ push r4 and lr on the stack
        ...
        MOV     r4, #0        @ modify r4
        ...
        POP     {r4, pc}      @ pop register and return address
```

Parameterübergabe

Parameterübergabe

- ▶ Möglichkeiten für die Parameterübergabe:
 - ▶ Parameterübergabe über Register (schnell, beschränkte Anzahl Parameter)
 - ▶ Parameterübergabe über den Stack (langsam, beliebige Anzahl Parameter)
- ▶ Bei reinen Assembler-Projekten ist man frei, wie die Übergabe definiert wird.
- ▶ Bei gemischten Projekten C / Assembler muss man sich an die Richtlinien des C-Compilers halten.
- ▶ Bei ARM wird die Parameterübergabe in der AAPCS festgelegt:
 - ▶ **Die ersten vier Parameter werden in den Registern r0 bis r3 übergeben.**
 - ▶ **Die übrigen Parameter werden in umgekehrter Reihenfolge auf dem Stack abgelegt.**
 - ▶ **Rückgabewert in r0.**

Parameterübergabe über Register

- ▶ **Call by Value:** Als Parameter wird eine Kopie des Wertes in einem Register (r0 bis r3) übergeben.

```
LDR    r0,=var    @ r0 points to address var
LDR    r0,[r0]     @ copy value of var into r0
BL     mySubr      @ call subroutine,
                  @ parameter is passed in r0
```

- ▶ **Call by Reference:** Als Parameter wird die Adresse einer Variablen in einem Register (r0 bis r3) übergeben.

```
LDR    r0,=var    @ r0 points to address of var
BL     mySubr      @ call subroutine,
                  @ parameter is passed in r0
```

Parameterübergabe über Register

► Beispiel: `void sum(int p1, int p2, int* res) {*res = p1 + p2;}`

```
.data
v1:      .word 2          @ 1st parameter
v2:      .word 5          @ 2nd parameter
res:     .word 0          @ result
        .text
main:
    ...
# calling subroutine sum
    LDR    r0,=v1          @ copy address of v1 into r0
    LDR    r0,[r0]         @ read value v1, 1st parameter in r0, by value
    LDR    r1,=v2          @ copy address v2 into r1
    LDR    r1,[r1]         @ read value v2, 2nd parameter in r1, by value
    LDR    r2,=res         @ copy address res, 3rd parameter in r2, by ref.
    BL     sum             @ branch to subroutine sum
    ...
# subroutine sum
sum:     ADD    r3,r0,r1    @ r3 = p1 + p2, p1 passed in r0, p2 in r1
        STR    r3,[r2]     @ *res = r3, res in r2 by reference
        MOV    pc,lr       @ return
```

Parameterübergabe über den Stack

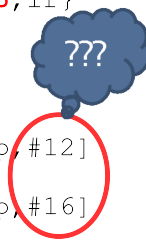
► Beispiel:

```
int super_sum(int p1, int p2, int p3, int p4, int p5, int p6);
```

► AAPCS → p1 bis p4 in Registern, p5 und p6 auf dem Stack, return in r0.

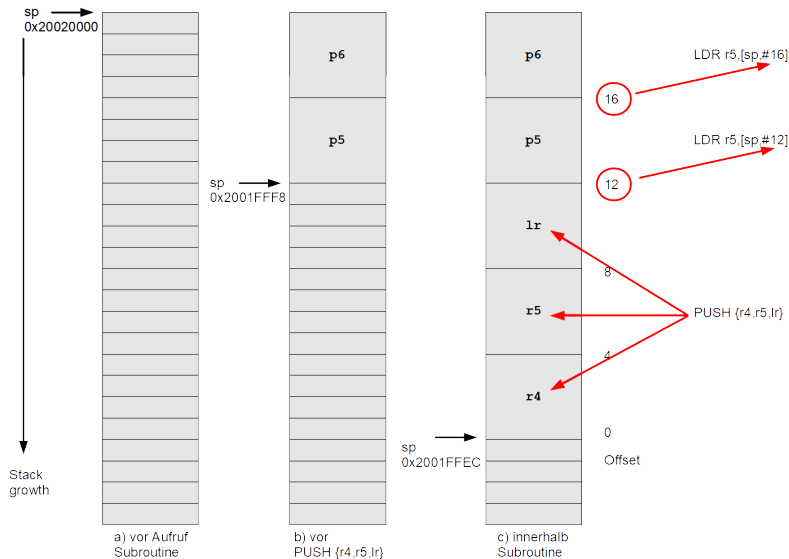
super_sum :

PUSH	{ r4, r5 , lr}	@ push r4, r5 and lr on stack	
MOV	r4, r0	@ temp = p1	} Register
ADD	r4, r1	@ temp += p2	
ADD	r4, r2	@ temp += p3	
ADD	r4, r3	@ temp += p4	
LDR	r5, [sp, #12]	@ buffer = p5	} Stack
ADD	r4, r5	@ temp += p5	
LDR	r5, [sp, #16]	@ buffer = p6	
ADD	r4, r5	@ temp += p6	
MOV	r0, r4	@ copy return value to r0	
POP	{ r4, r5 , pc}	@ pop r4, r5 and lr, return	



► Optimierung obiger Code: r0 statt r4 für temporäre Variable, →2 Instruktionen weniger.

Parameterübergabe über den Stack



ARM Architecture Procedure Call Standard (AAPCS)

Register	AAPCS Synonym	Beschreibung
r0	a1	Argument 1 / Integer-Rückgabewert 32-Bit / Scratch-Register
r1	a2	Argument 2 / Integer-Rückgabewert 64-Bit / Scratch-Register
r2	a3	Argument 3 / Scratch-Register
r3	a4	Argument 4 / Scratch-Register
r4	v1	Variable Register 1
r5	v2	Variable Register 2
r6	v3	Variable Register 3
r7	v4	Variable Register 4
r8	v5	Variable Register 5
r9	v6 / SB / TR	Platformspezifisches Register / Variable Register 6
r10	v7	Variable Register 7
r11	v8	Variable Register 8
r12	ip	Intra-Procedure Call / Scratch-Register
r13	sp	Stack-Pointer
r14	lr	Link-Register
r15	pc	Program-Counter

Lokale Variablen

Anlegen von lokalen Variablen

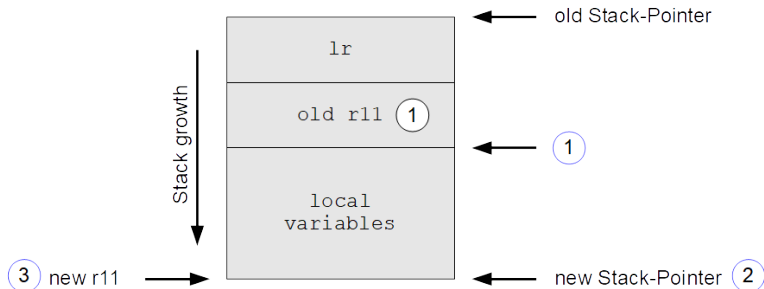
- ▶ Werden nur wenige lokale Variablen benötigt, können diese in den Registern r0 bis r12 abgelegt werden. Diese sind gemäss AAPCS zu retten.
- ▶ Wenn zusätzliche Variablen benötigt werden, müssen diese auf dem Stack gespeichert werden. Dazu wird ein Variablenbereich auf dem Stack angelegt.
- ▶ Beispiel in C:

```
void myFunc(void) {  
    unsigned int myArray[10];  
    ...  
}
```

- ▶ Zusätzlich kann optional ein Register verwendet werden, welches auf den Bereich dieser lokalen Variablen zeigt: der Frame-Pointer (üblicherweise r11).

Anlegen von lokalen Variablen

```
sub _localvar:  
    PUSH    {r11, lr}    @ push r11 and return address, (1)  
    SUB     sp, sp, #40  @ allocate 10 * 4 bytes local variable space, (2)  
    MOV     r11, sp      @ frame pointer r11 points to first local variable, (3)  
    ...  
    ...  
    ADD     sp, sp, #40  @ free local variable space  
    POP     {r11, pc}    @ pop r11 and lr, return
```



Häufige Fehlerquellen

Häufige Fehlerquellen

Es ist besondere Vorsicht geboten! Generell gilt:

- ▶ Subroutinen immer mit “BL” (und nicht mit “B”) aufrufen!
- ▶ Subroutinen verlassen, indem die Rücksprungadresse in den Program-Counter kopiert wird (entweder direkt aus dem Link-Register oder vom Stack holen)!
- ▶ Werden in einer Subroutine Register verändert (ausser Scratch-Register), so sind diese auf dem Stack zu retten (Reihenfolge beachten, d.h. zuletzt gerettet → zuerst entfernt). Die Anzahl der “Pop” muss mit der Anzahl “Push” übereinstimmen!
- ▶ **Stack genügend gross dimensionieren**, damit es keinen Überlauf gibt! Diese Fehlerquelle kann auch nur sporadisch auftreten und ist besonders schwierig zu lokalisieren!!! Die Stack-Grösse können Sie im Linkerskript definieren:
_MIN_STACK_SIZE.
- ▶ Fehlerhafte Berechnung der Offsets bei der Parameterübergabe über den Stack vermeiden!

Makros oder Subroutinen?

Subroutine oder Makro?

- ▶ Subroutine: Code ist einmal im Speicher abgelegt.
 - (+) braucht wenig Speicher.
 - (-) Aufruf ist langsamer (Instruktion für Aufruf und Rücksprung).
- ▶ Makro: Code wird anstelle des Makroaufrufs im Speicher abgelegt.
 - (-) Braucht mehr Speicher (ausser Makro ist sehr kurz).
 - (+/-) Keine Rekursion möglich.
 - (+) Aufruf ist schneller (keine Instruktion für Aufruf und Rücksprung).
- ▶ Generell gilt deshalb:
 - ▶ Verwenden Sie im “Normalfall” Subroutinen.
 - ▶ Wenn Sie sehr zeitkritische Codesequenzen haben, dann verwenden Sie Makros.