Vorlesungsskript zu „Vertiefung Programmieren“ Stack & Heap

Dozent: Dipl.-Inf. (FH) Andreas Schmidt

DHBW
Duale Hochschule
Baden-Württemberg

Memory Management

Stack und Heap-Speicher

Stack und Heap Speicher

Stack Speicher

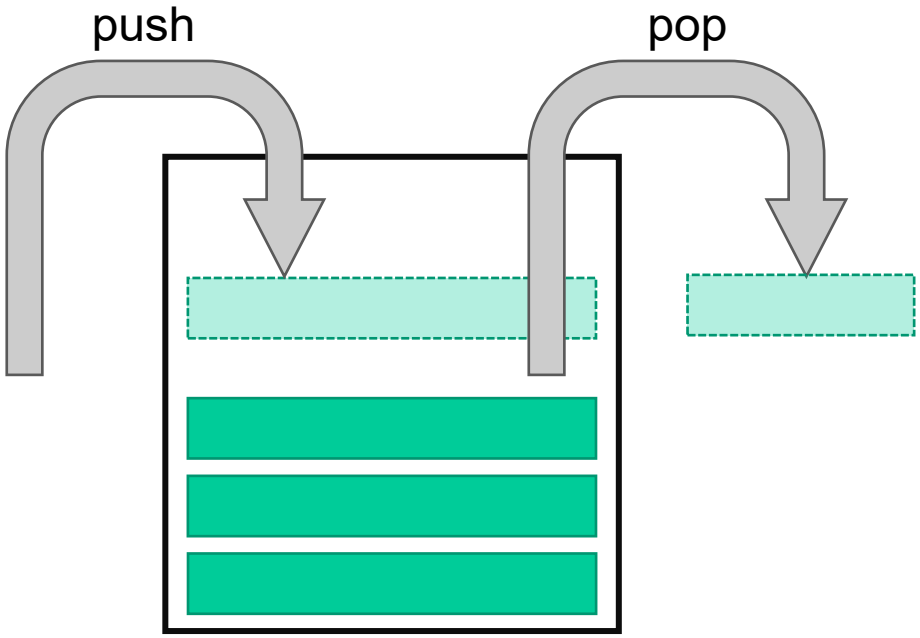
- Ein Stack-Speicher („Stapel“) wird vom Prozessor für Funktionsaufrufe und lokale Variablen genutzt
- Wird eine Funktion aufgerufen, so wird u.a. die Rücksprungadresse des Aufrufers auf dem Stack gespeichert und beim Verlassen (Rücksprung) der Funktion vom Stack gelesen und der Program-Counter entsprechend gesetzt
- Zusätzlich werden vom Compiler lokale Variablen auf dem Stack gespeichert. Hierzu reserviert der Compiler einen entsprechenden Bereich im Speicherbereich des Stacks und nutzt eine „relative Adressierung basierend auf dem Stack-Pointer“
- Die Daten auf dem Stack (i.d.R. Rücksprungadresse, Parameter und lokale Variablen) werden als „Stack Frame“ bezeichnet

Quelle: intern

Stack und Heap Speicher

Stack Speicher

- Ein Stack (Datenstruktur) ist eine sog. Last In – First Out Struktur
  - d.h. das zuletzt gespeicherte Datum wird als erstes wieder ausgelesen
- Die Operationen eines Stack heißen:
  - push → Daten auf den Stack legen
  - pop → Daten vom Stack holen

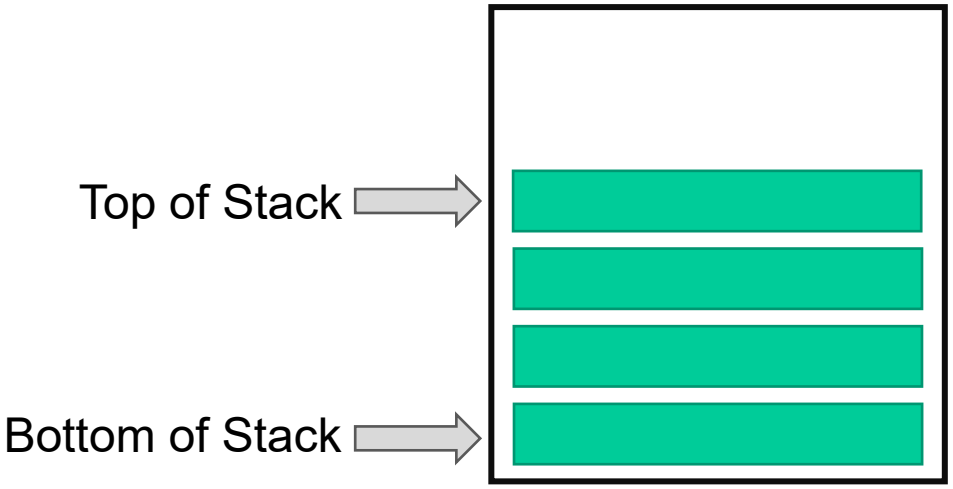


Quelle: intern

Stack und Heap Speicher

Stack Speicher

- Bei der Betrachtung als Datenstruktur werden folgende Begriffe verwendet
  - *Top of Stack* – Oberster Eintrag im Stack
  - *Bottom of Stack* – Letzter Eintrag im Stack (manchmal auch End of Stack)



Quelle: intern

Stack und Heap Speicher

Stack Speicher des Prozessor

- Der Stack Speicher des Prozessors kann z.B. bei einer ARM Architektur in unterschiedlichen Varianten umgesetzt werden:
  - **full descending stack** → Der Stack-Pointer zeigt auf den aktuellen (belegten) Speicherplatz (daher der Name „full“) und bei der nächsten Push-Operation wird zuerst der Stack-Pointer dekrementiert (descending) und danach der neue Eintrag gespeichert.
  - **full ascending stack** → Wie beim full descending stack, allerdings wird der Stack-Pointer inkrementiert (ascending)
  - **empty descending stack** → Der Stack Pointer zeigt auf den freien Speicher (empty), bei der Push-Operation wird das Datum gespeichert, und danach der Stack-Pointer dekrementiert (descending)
  - **empty ascending stack** → Wie beim empty descending stack, allerdings wird der Stack-Pointer inkrementiert (ascending)

Quelle: intern

Stack und Heap Speicher

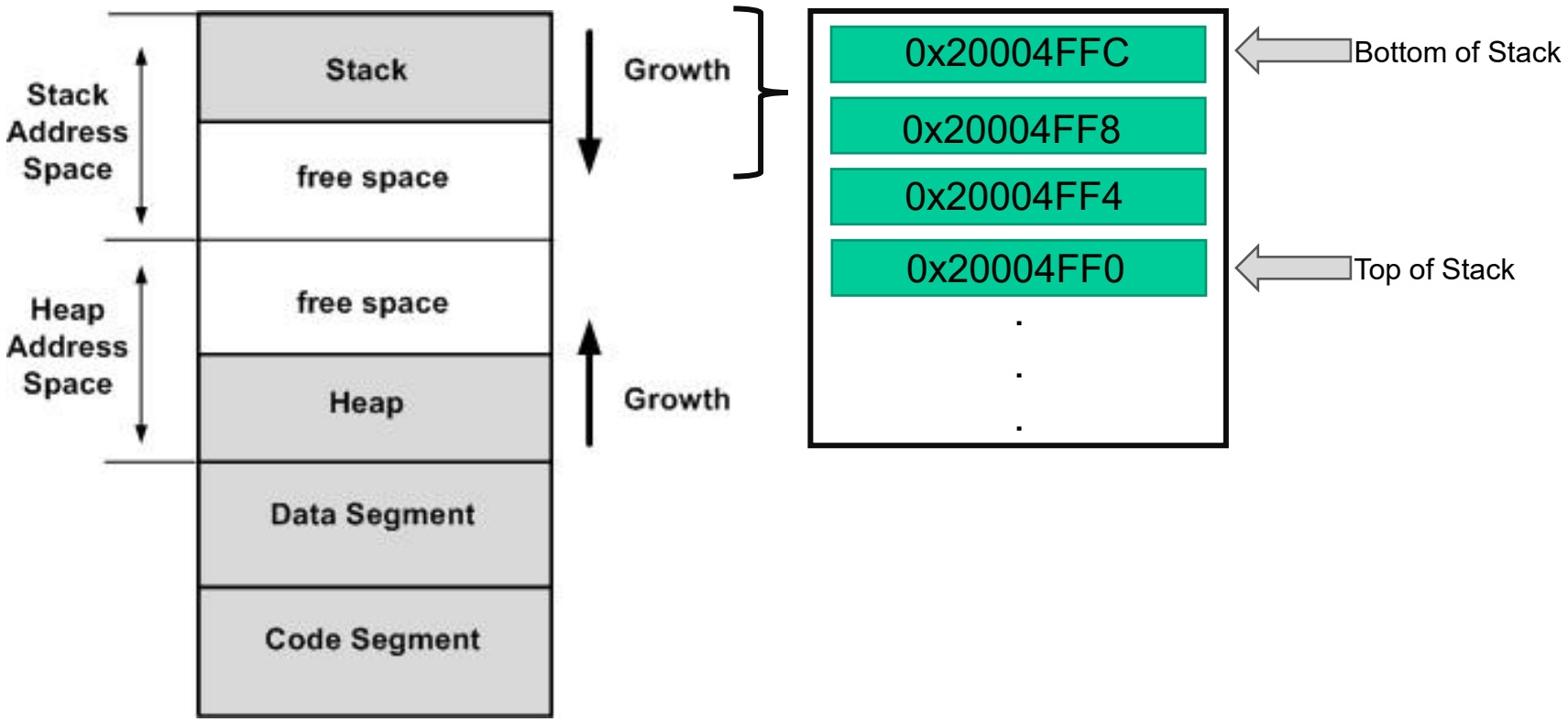
Stack Speicher des Prozessor

- Der Stack Speicher beim STM32G474 (Cortex-M4) ist als full-descending Stack implementiert.
- Daher wird häufig in den Embedded-Systemen der Stack an das Ende des RAM Bereichs gelegt
- Durch die Implementierung als „descending“ Stack, bewegt sich der Stack-Pointer von den hohen Adressen zu den niedrigen
  - man sagt auch: „der Stack wächst von oben nach unten“

Quelle: intern

Stack und Heap Speicher

Stack Speicher des Prozessor



Quelle: intern

Memory Layout

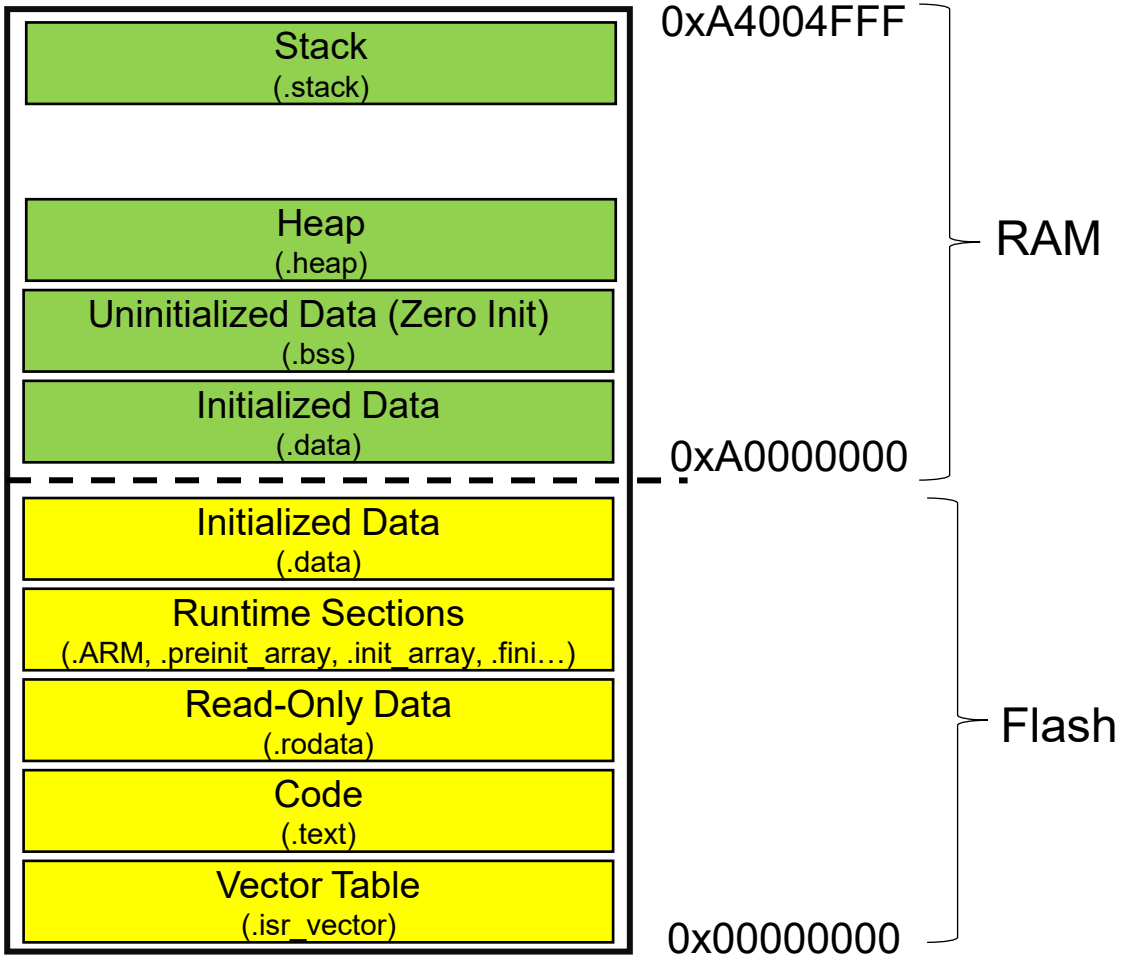
Beispiel eines Linker-Files und dem dazugehörigen Memory Layout

```
/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */

    . = ALIGN(4);
    _etext = .;        /* define a global symbols at end of code */
  } >FLASH

  /* Constant data goes into FLASH */
  .rodata :
  {
    . = ALIGN(4);
    *(.rodata)         /* .rodata sections (constants, strings, etc.) */
    *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
    . = ALIGN(4);
  } >FLASH
```

Quelle: intern

Memory Layout

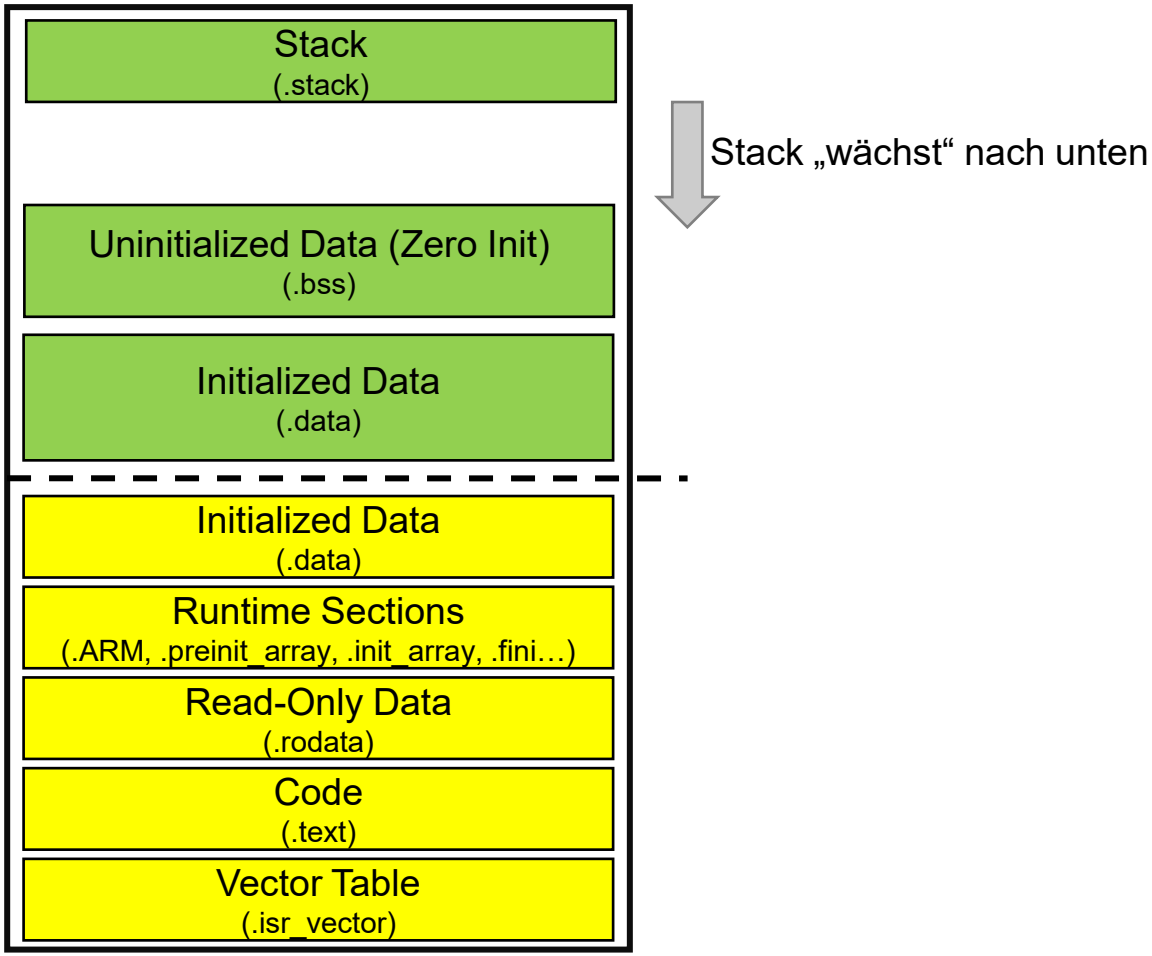
Beispiel eines Linker-Files und dem dazugehörigen Memory Layout



Quelle: intern

Memory Layout

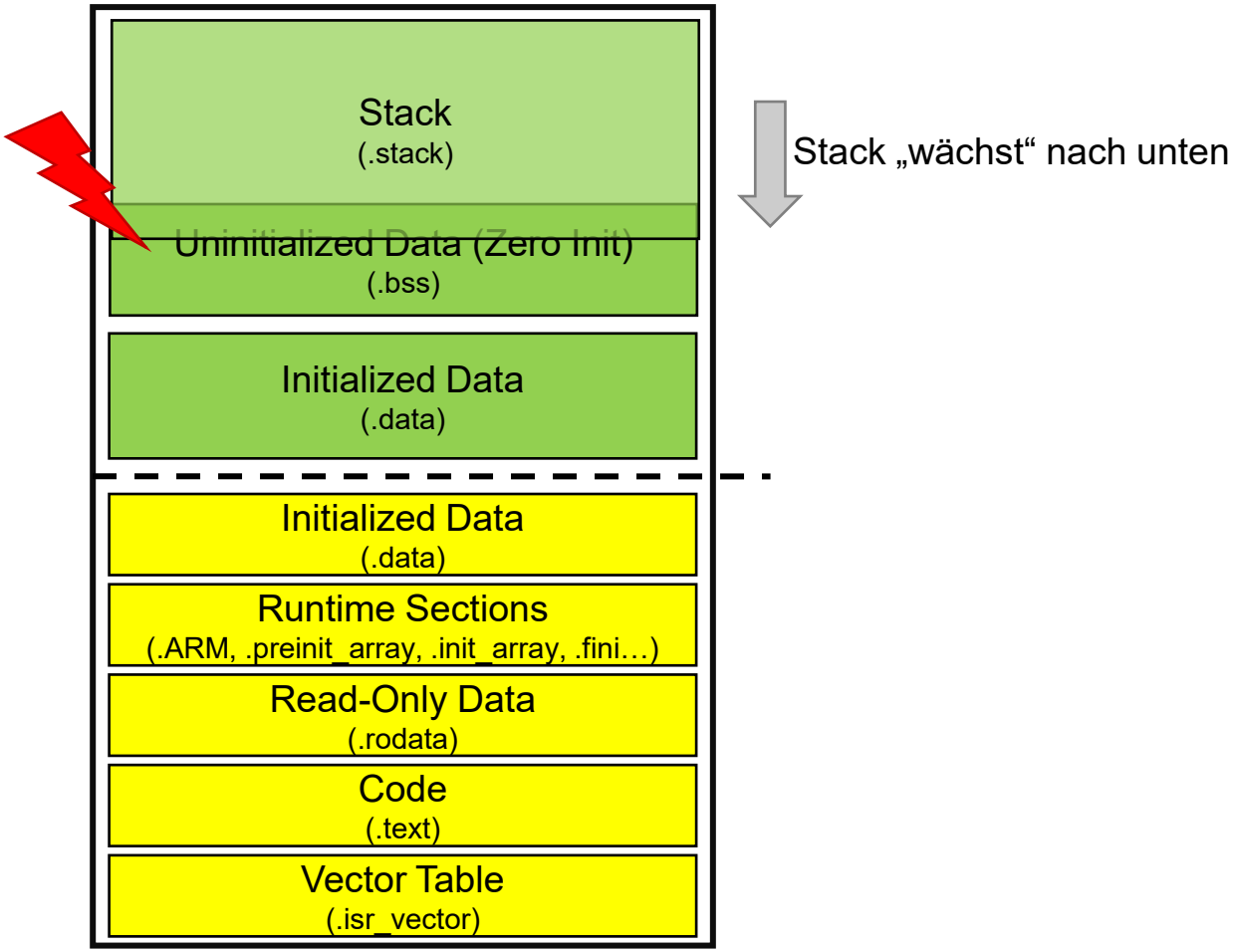
Stack-Overflow



Quelle: intern

Memory Layout

Stack-Overflow



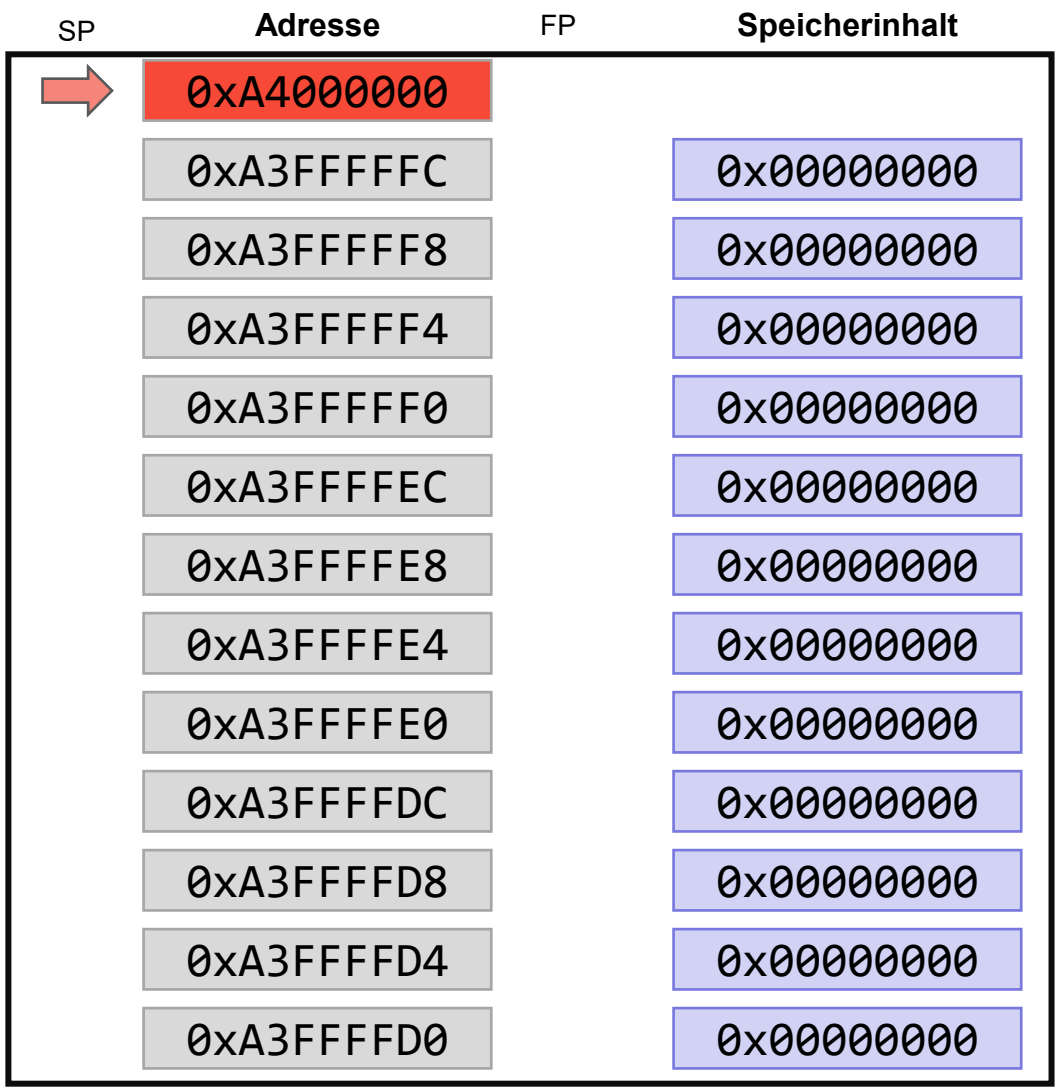
Quelle: intern

DHBW
Duale Hochschule
Baden-Württemberg

Stack und Compiler

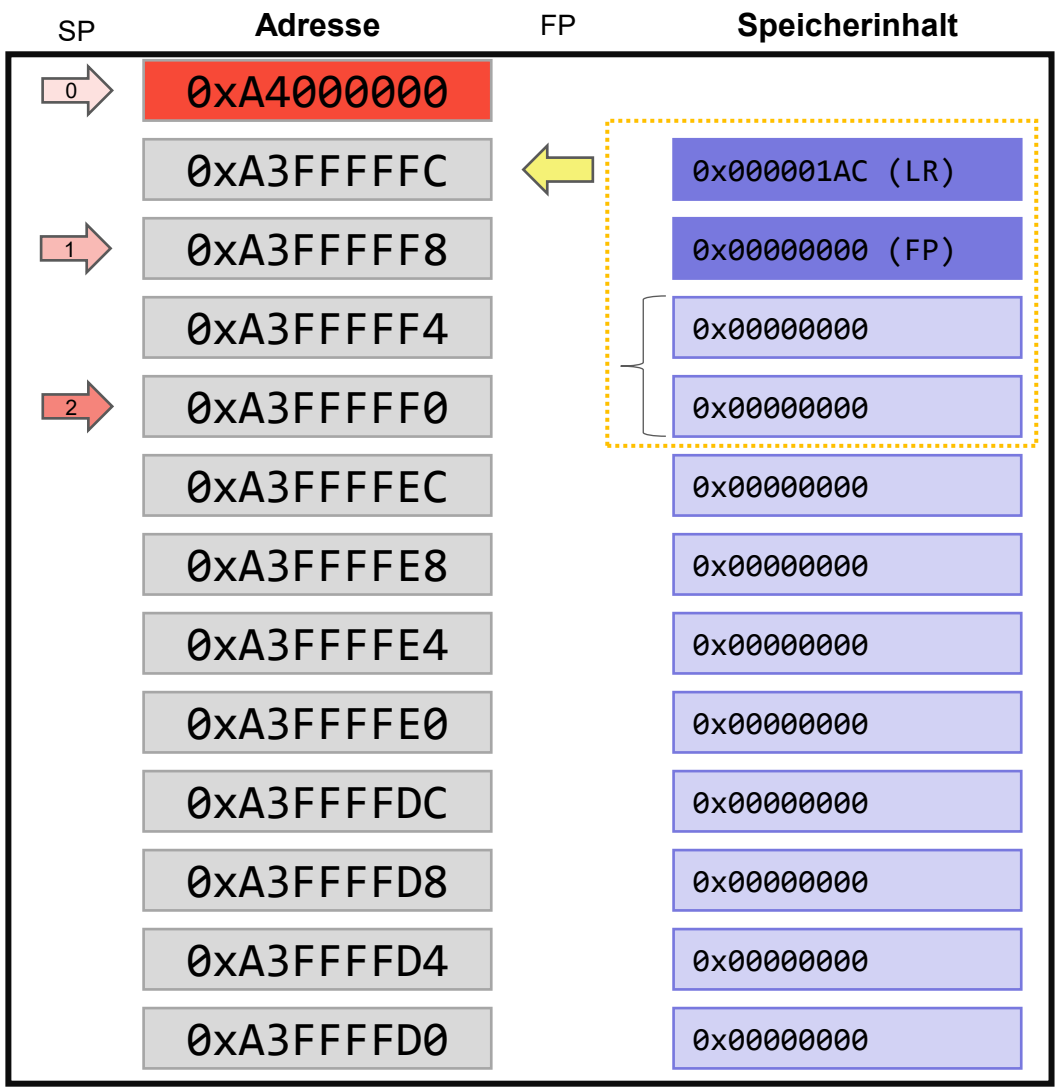
Code-Generierung für Stack-Handling

Stack und Code-Generierung



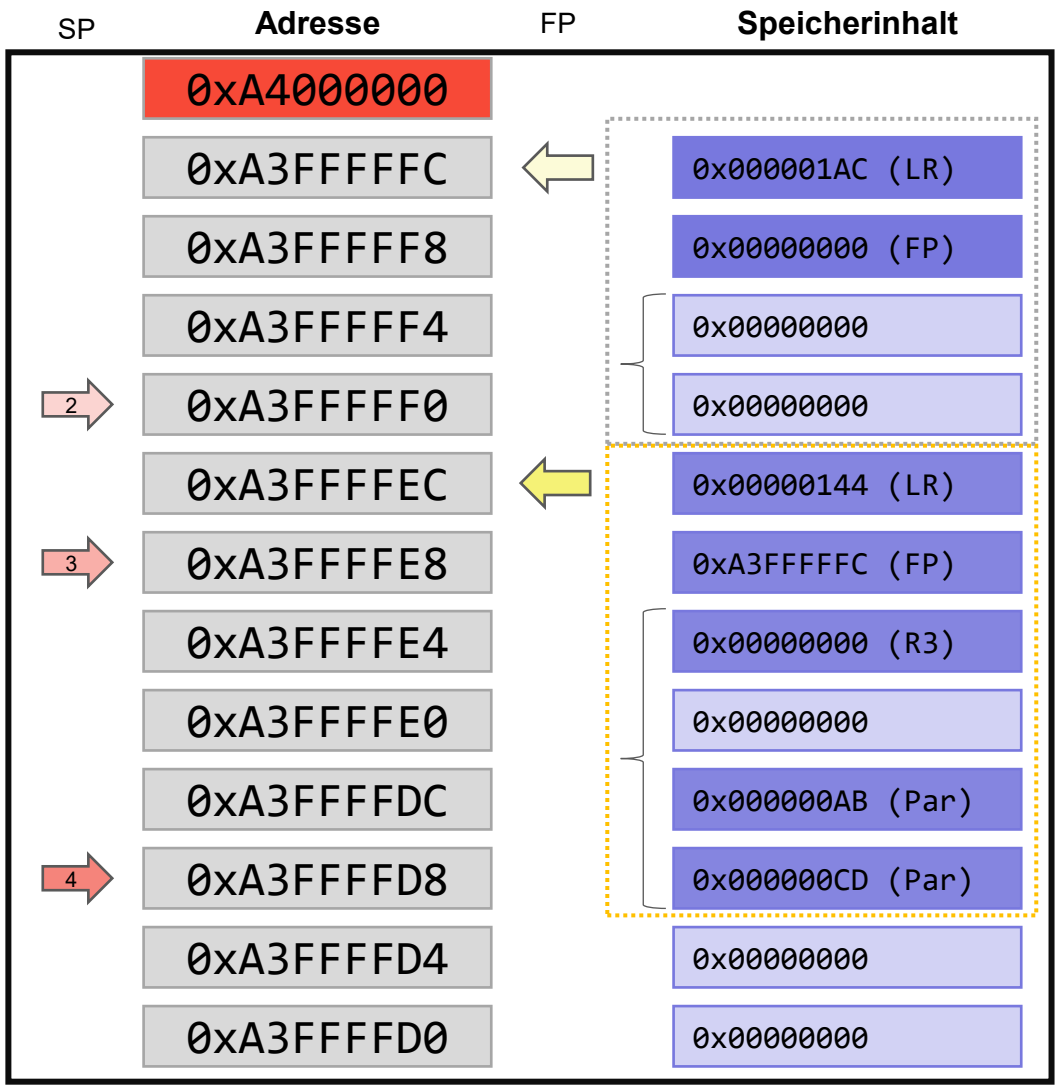

```
init_stack:
        @@ Initialize the stack pointer
        ldr   sp, =(0xA4000000)
        bl    main
```

Stack und Code-Generierung




```
main:   push    {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8

        mov     r1, #205
        mov     r0, #171
        bl      simpleFunctionCallTree
        str     r0, [fp, #-8]

        mov     r3, #0

        mov     r0, r3
        sub     sp, fp, #4
        pop     {fp, lr}
        bx      lr
```

Stack und Code-Generierung




```
simpleFunctionCallTree(int, int):
        push    {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #16
        str     r0, [fp, #-16]
        str     r1, [fp, #-20]

        mov     r3, #0
        str     r3, [fp, #-8]

        bl      simpleFunctionWithoutParam

        ldr     r0, [fp, #-16]
        bl      simpleSecondCallLevel

        ldr     r0, [fp, #-20]
        bl      simpleFuncWithParamReturn
        str     r0, [fp, #-8]

        ldr     r3, [fp, #-8]

        mov     r0, r3
        sub     sp, fp, #4
        pop     {fp, lr}
        bx      lr
```

DHBW
Duale Hochschule
Baden-Württemberg

Heap Speicher

Dynamische Speicherverwaltung

Stack und Heap Speicher

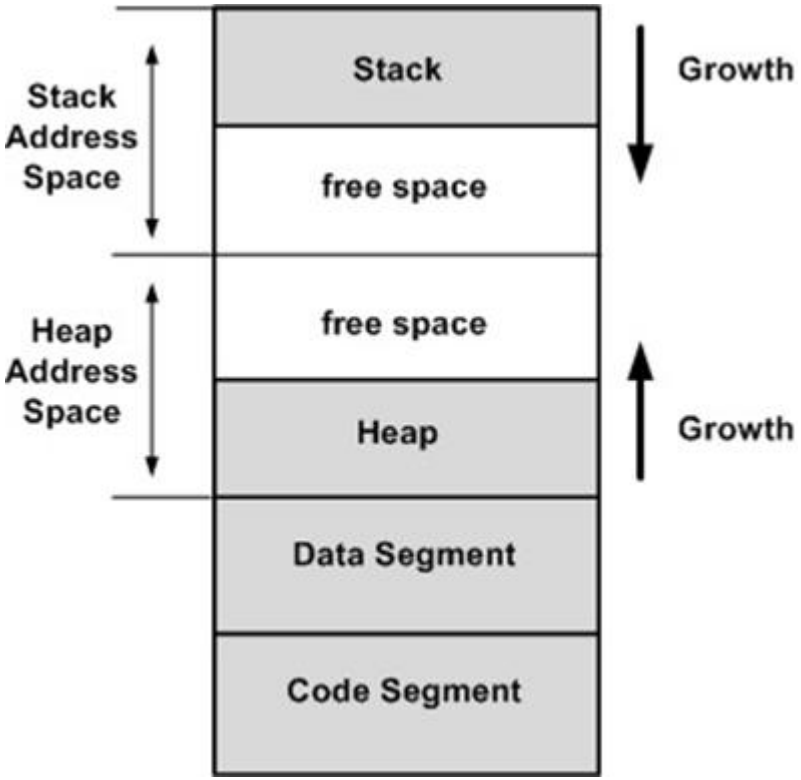
Heap Speicher

- Ein Heap-Speicher („Haufen“) ist ein Speicherbereich, der für die dynamische Speicherverwaltung verwendet wird.
- Dynamisch bedeutet hier, dass Speicherblöcke während der Ausführung des Programms reserviert (allokiert) werden und sobald diese nicht mehr benötigt werden, auch wieder freigegeben werden.
- D.h. der Speicherbedarf ist in diesem Fall nicht zur Compilierzeit bekannt, sondern ergibt sich erst während der Laufzeit
- In C werden klassisch die Funktionen `malloc()` und `free()` für die Verwendung von dynamischem Speicher verwendet.

Quelle: intern

Stack und Heap Speicher

Heap Speicher

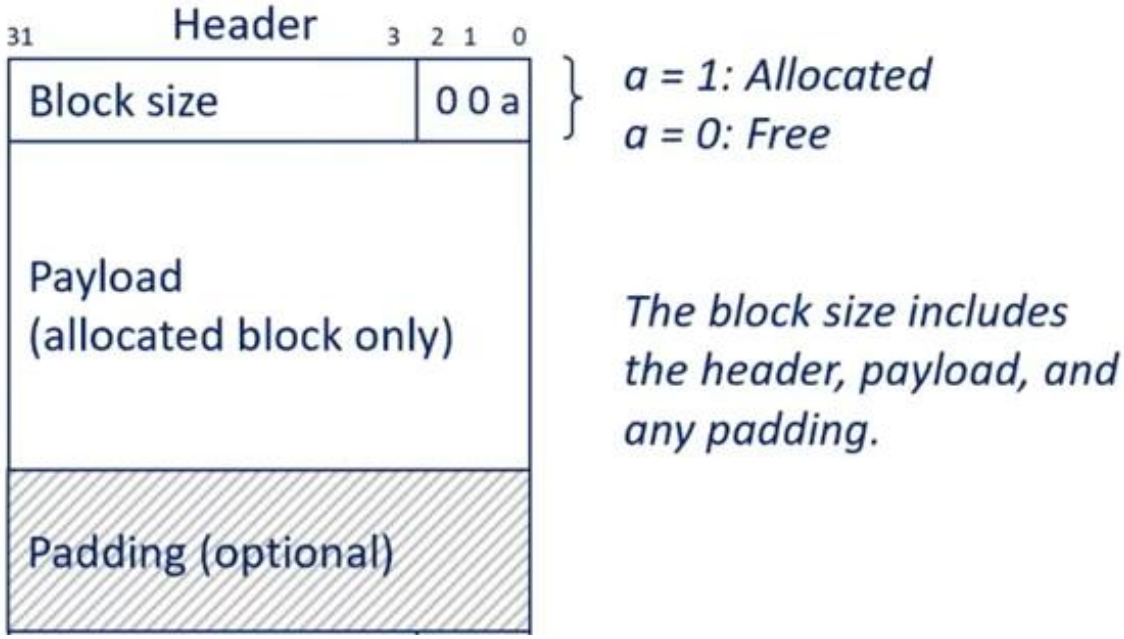


- Damit Funktionen wie `malloc()` „wissen“, in welchem Speicherbereich die Speicherblöcke reserviert werden sollen, muss der Heap Bereich i.d.R. in der Memory Map (z.B. Linker-File) definiert werden.
- Häufig wird der Heap Bereich an das Ende des Data-Segments gelegt, wodurch der Heap-Speicher „nach oben wächst“.

Quelle: intern

Dynamische Speicherverwaltung

Memory Allocator (e.g. malloc())

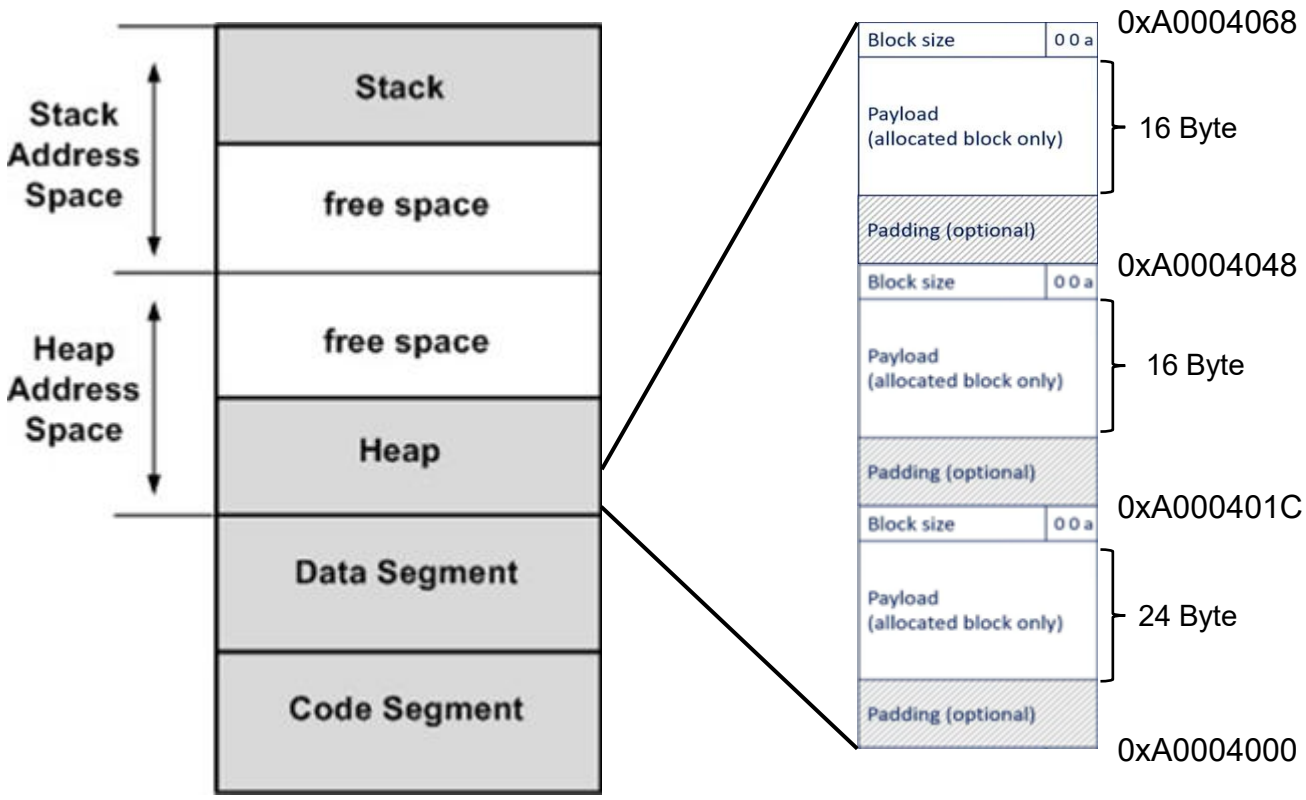


- Für die Verwaltung der reservierten und genutzten Speicherblöcke, sind zusätzliche Verwaltungsinformationen notwendig.
- Diese Verwaltungsinformationen sind i.d.R. für den Anwender nicht sichtbar und werden intern durch Funktionen wie `malloc()` und `free()` genutzt.

Quelle: intern

Dynamische Speicherverwaltung

Memory Allocator (e.g. malloc())



Quelle: intern

Stack und Heap Speicher

Probleme der dyn. Speicherverwaltung in Embedded Systems

- **Speicher Fragmentierung**
  Der Heap Bereich wird durch Reservieren und Freigeben von Speicher fragmentiert

- **Determinismus**
  Durch Fragmentierung und nicht optimale Algorithmen, kann die Laufzeit einer Speicherreservierung/Freigabe u.U. zeitlich variieren

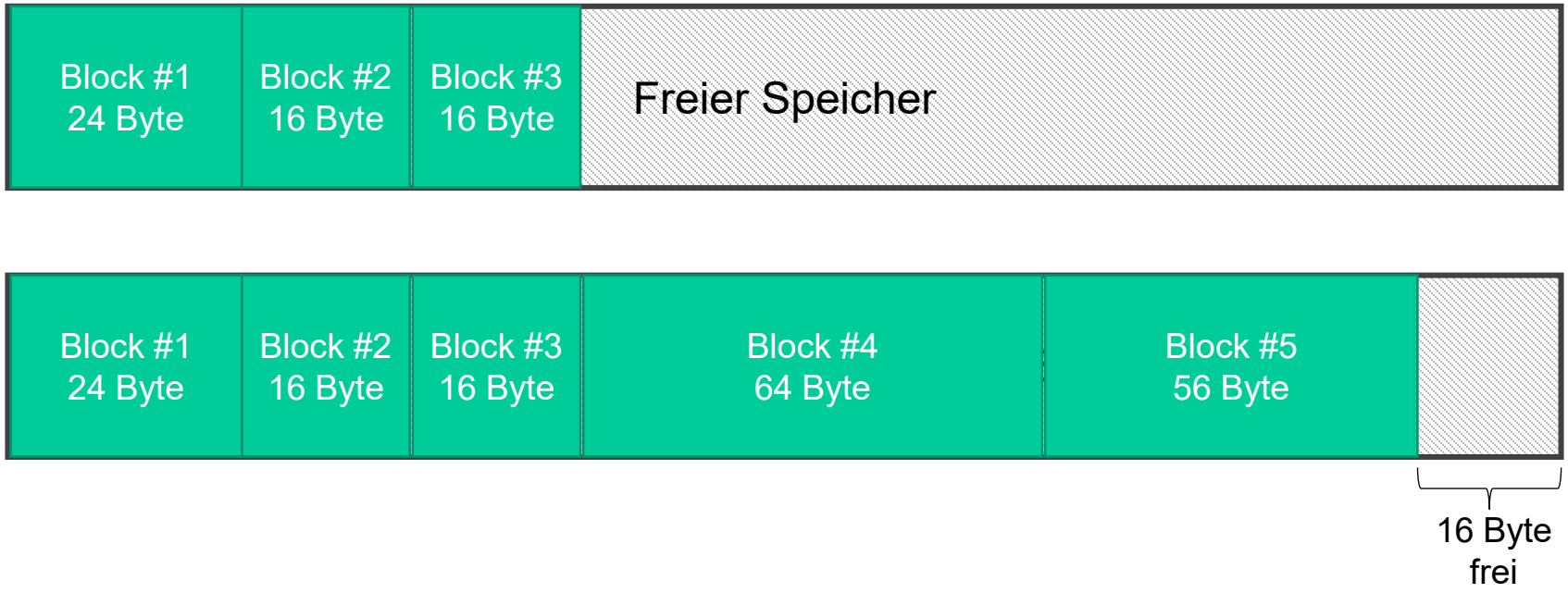
- **Speicherlecks**
  Vergessene Speicherfreigaben führen zu sog. Speicherlecks und damit zu Speicherblöcken, die nicht mehr genutzt werden können

- **Management Overhead**
  Die Verwaltung der Speicherblöcke benötigt zusätzlichen Speicher sowie zusätzlichen Code

Quelle: intern

Dynamische Speicherverwaltung

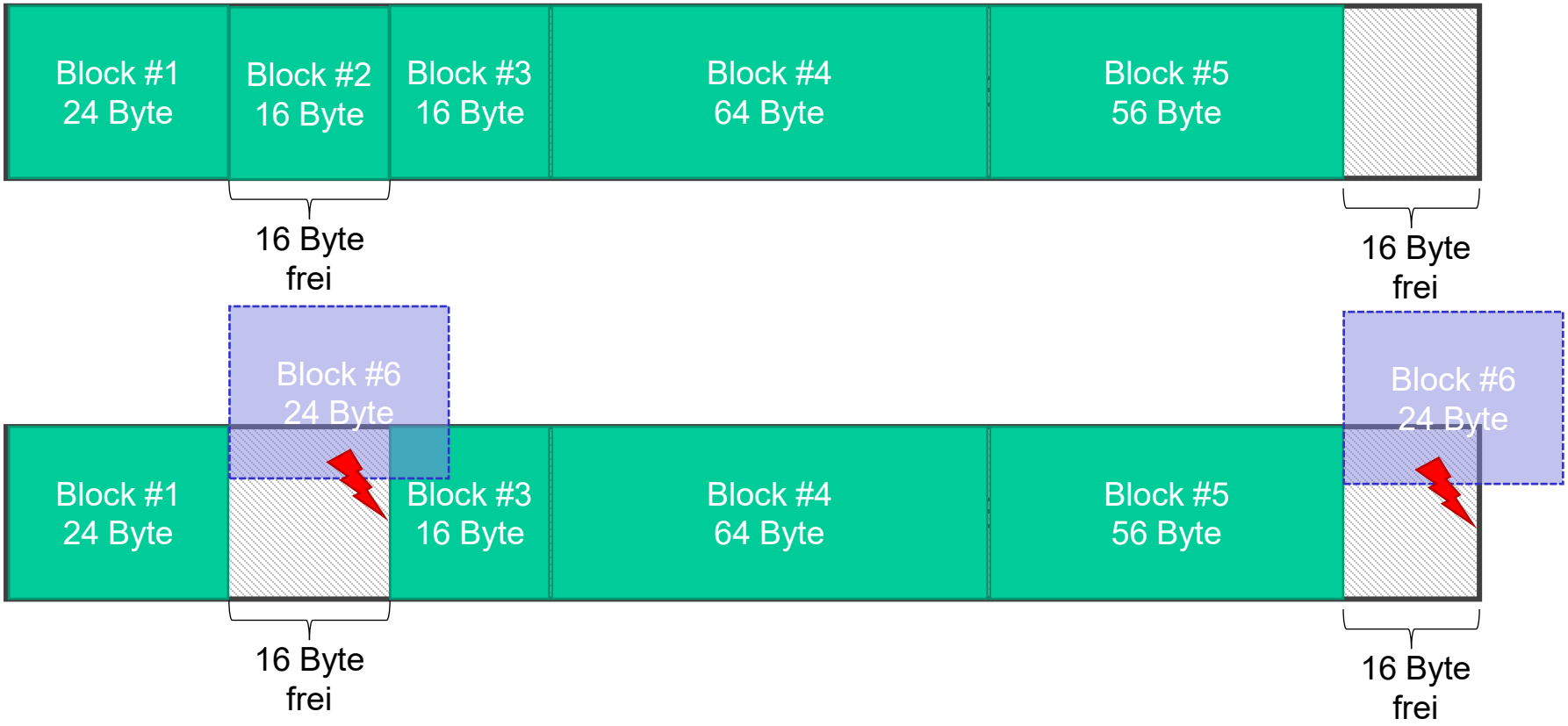
Problem mit der „Freigabe“ der Blöcke



Quelle: intern

Dynamische Speicherverwaltung

Problem mit der „Freigabe“ der Blöcke



Quelle: intern

Dynamische Speicherverwaltung

malloc() Beispiele

https://github.com/Snaipe/malloc/blob/master/malloc.c

https://gist.github.com/mshr-h/9636fa0adcf834103b1b

Quelle: intern