Vorlesungsskript zu „Vertiefung Programmieren“ Toolchain

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

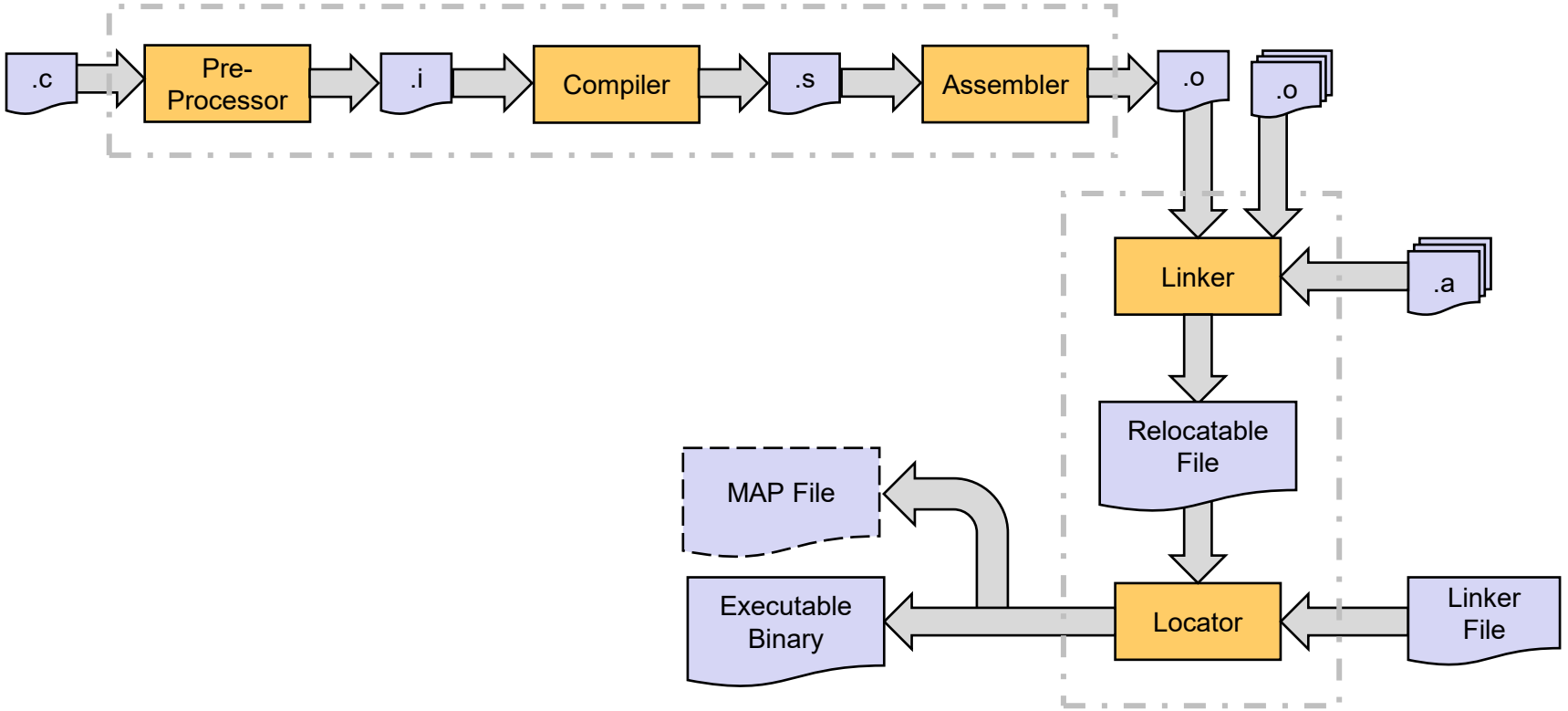
DHBW
Duale Hochschule
Baden-Württemberg

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Compiler Toolchain

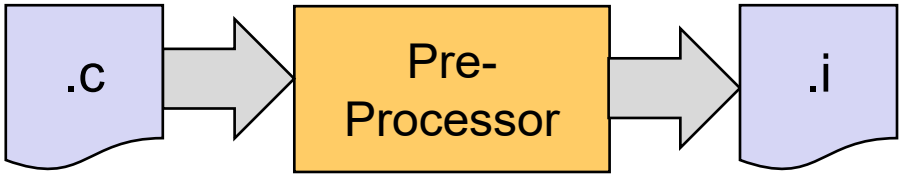
Grundsätzlicher Aufbau einer „C/C++ Compiler Toolchain“



Quelle: intern

Compiler Toolchain

Pre-Processor



- Entfernt alle Kommentare
- Verarbeitet die sog. „Präprozessor Direktiven“
  - Code-Zeilen welche mit „#“ beginnen
  - Ersetzt Makros und Konstanten im Source-Code

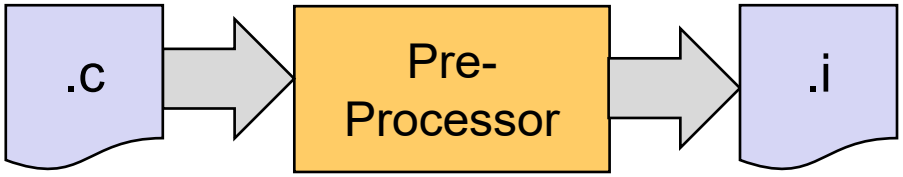
  → modifiziert den Code, welcher der Compiler zum „Übersetzen“ bekommt

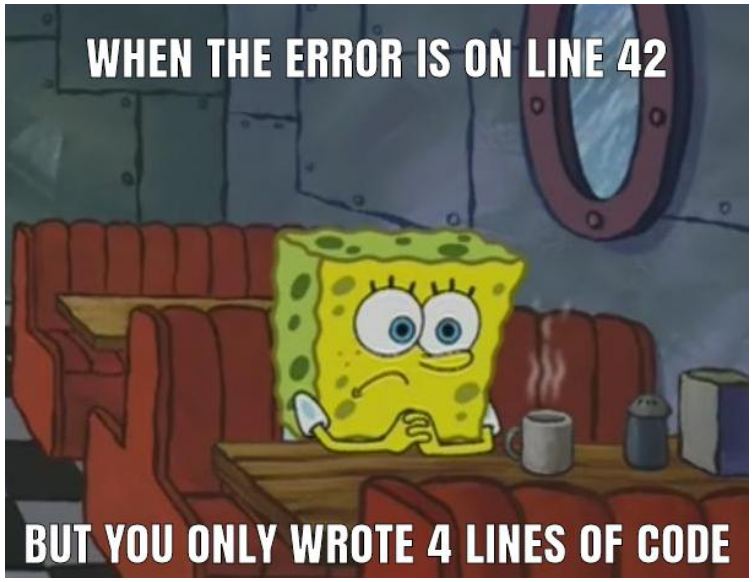
https://gcc.gnu.org/onlinedocs/gcc-14.3.0/cpp/

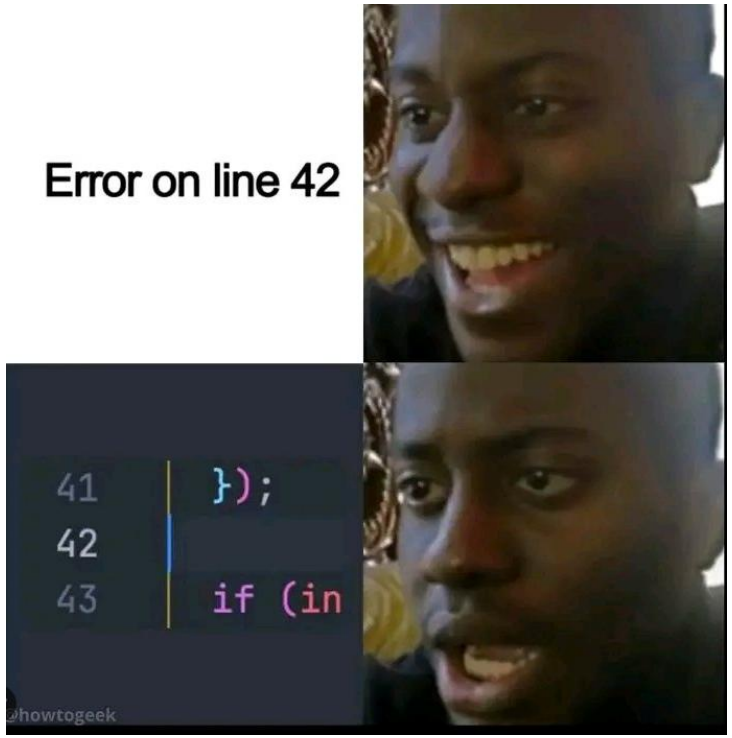
Quelle: intern

Compiler Toolchain

Pre-Processor



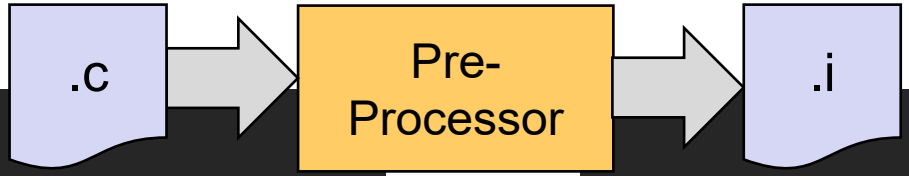




Quelle: intern

Compiler Toolchain

Pre-Processor




```c
/*
 * Some Header Comments
 * Author: Andreas Schmidt
 */

#define SPECIAL_VERSION

#ifdef SPECIAL_VERSION
    #define INIT_VALUE 0xABCD
#else
    #define INIT_VALUE 0xDCBA
#endif

static int globalVarInit = INIT_VALUE;
static int globalVarNoInit;

static int newResult = 0xDEADBEEF;

/**
 * Main Entry for Application
 */
int main(void)
{
    /* Set a value for the non-initialized variable */
    globalVarNoInit = 0x20;

    /* Do some super complicated calculations */
    int result = globalVarInit + globalVarNoInit;
    newResult = result + 0x10;

    return result;
}
```

```c
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "main.c"
# 17 "main.c"
static int globalVarInit = 0xABCD;
static int globalVarNoInit;

static int newResult = 0xDEADBEEF;




int main(void)
{

 globalVarNoInit = 0x20;


 int result = globalVarInit + globalVarNoInit;
 newResult = result + 0x10;

 return result;
}
```

Quelle: intern

Compiler Toolchain

Prä-Prozessor Direktiven

`#define`
Definiert ein Symbol mit einem optionalen Inhalt und speichert es in der Symboltabelle des Prä-Prozessors. Wird das Symbol innerhalb der aktuell verarbeitenden Datei verwendet, so ersetzt der Prä-Prozessor das Symbol mit dem definierten Inhalt.

Die definierten Symbole sind nur für die aktuell verarbeitete Datei gültig.
Der Prä-Prozessor unterscheidet zwischen sog. „object-like macros“ und „function-like macros“

`#undef`
Löscht ein Symbol aus der internen Symbol-Tabelle des Prä-Prozessors, wodurch es nicht mehr im Quelltext ersetzt wird.

Quelle: intern

Compiler Toolchain

Prä-Prozessor Direktiven

`#ifdef`…`#elif`…`#else`…`#endif`
Direktiven für sog. bedingte Kompilierung. Hierbei kann mit Hilfe von `#ifdef` geprüft werden, ob ein bestimmtest Prä-Prozessor Symbol definiert ist. Der umgekehrte Fall ist `#ifndef` mit dessen Hilfe geprüft werden kann ob ein Symbol nicht definiert ist.

Der Quelltext, welcher zwischen den Direktiven `#ifdef/#ifndef` sowie `#elif/#else` bzw. `#endif` eingefasst ist, wird abhängig von der verwendeten Direkten und dem Vorhandensein des entsprechenden Symbols in die Ausgabedatei eingefügt oder auch nicht.

Dadurch lassen sich Teile des Quelltexts, abhängig von der Existenz von Symbolen, einfügen bzw. entfernen, bevor der Compiler den Quelltext übersetzt.

Quelle: intern

Compiler Toolchain

Prä-Prozessor Direktiven

`#include`
Für den Inhalt der angegebenen Datei an der Stelle in den Quelltext ein, an der die `#include` Direktive steht.

Der Prä-Prozessor erwartet hierbei nicht zwingend eine Datei mit korrektem C-Syntax. Der Prä-Prozessor kann jede beliebige Textdatei mit Hilfe von `#include` einfügen.

Nach dem Einfügen des Inhalts führt der Prä-Prozessor erneut eine Analyse durch und verarbeitet eventuell in der include-Datei enthaltenen Prä-Prozessor Direktiven

Quelle: intern

Compiler Toolchain

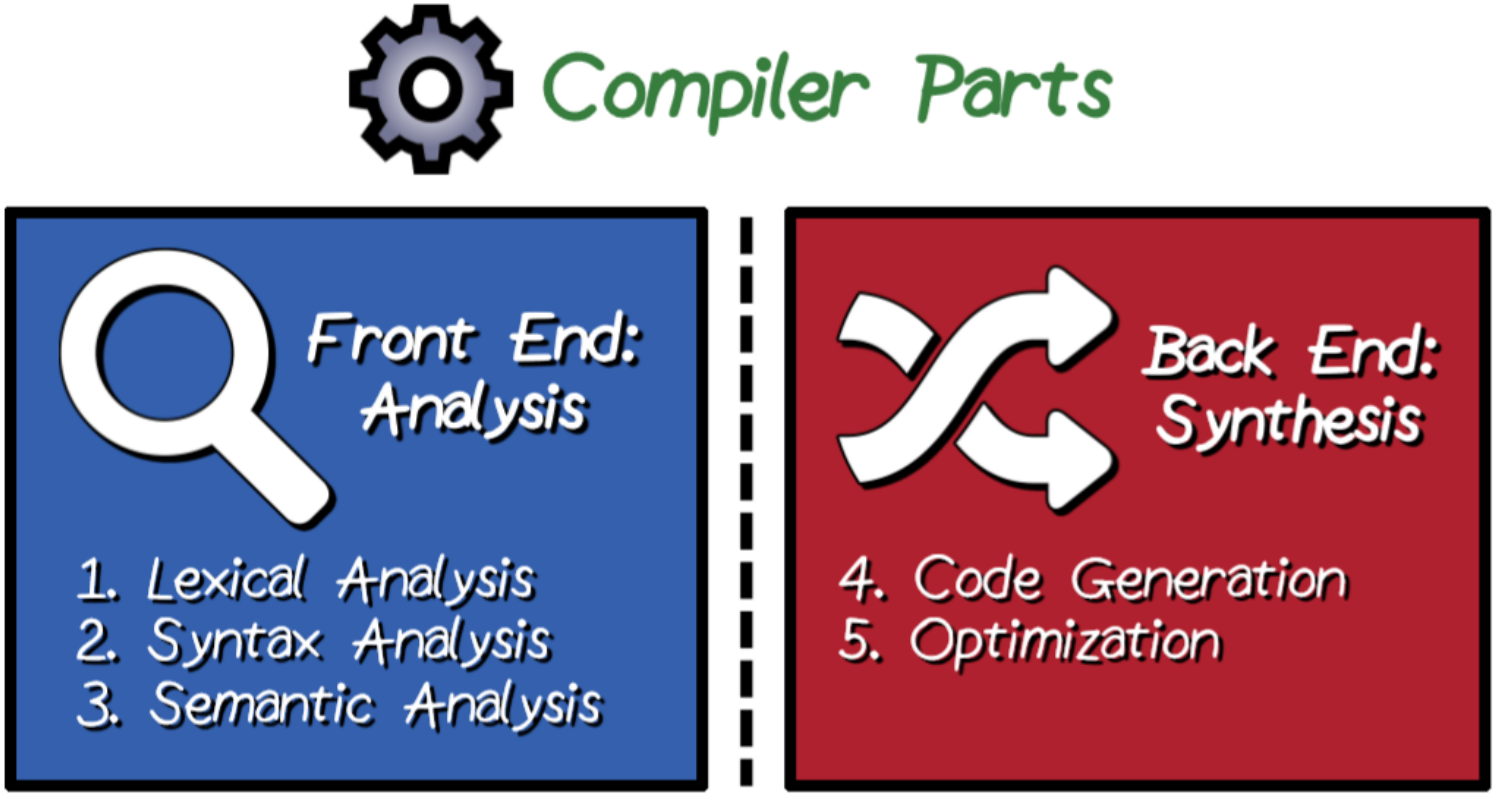
Compiler

- Generiert prozessorspezifischen Assembler Code aus dem vorbereiteten C/C++ Code
  - inkl. möglicher Optimierungen
- „Reserviert“ Speicherplatz für Variablen und Konstanten
- Platziert Code und Daten in entsprechende Speicherregionen

Quelle: intern

Compiler Toolchain

Compiler



Quelle: https://www.linkedin.com/pulse/c-build-process-details-abdelaziz-moustafa/

Compiler Toolchain

Compiler – Front End Analyse

Lexikalische Analyse

- Erste Phase der Kompilierung
- Wandelt die zeichenbasierte Eingabe des Quelltexts in sog. Token um
- Prüft auf „Rechtschreibfehler“
  - Z.B. fehlerhafte Variablen-Bezeichner wie int myVariable$1

Syntax Analyse

- Zweite Phase der Kompilierung
- Erstellt einen sog. Abstract Syntax Tree (AST) und Symbol-Tabellen
- Prüft die Token-Folge der lexikalischen Analyse auf korrekte „Grammatik“
  - Z.B. fehlende schließende Klammer in einem if-Ausdruck

Semantische Analyse

- Dritte Phase der Kompilierung
- Analysiert auf korrekte Verwendung von Ausdrücken
  - Z.B. Verwendung einer nicht deklarierten Variable

Quelle: intern

Compiler Toolchain

Compiler – Backend Synthese

Optimierung (Middle-End)

- Interne Repräsentation des Source-Codes wird mit Hilfe unterschiedlicher Optimierungsverfahren optimiert

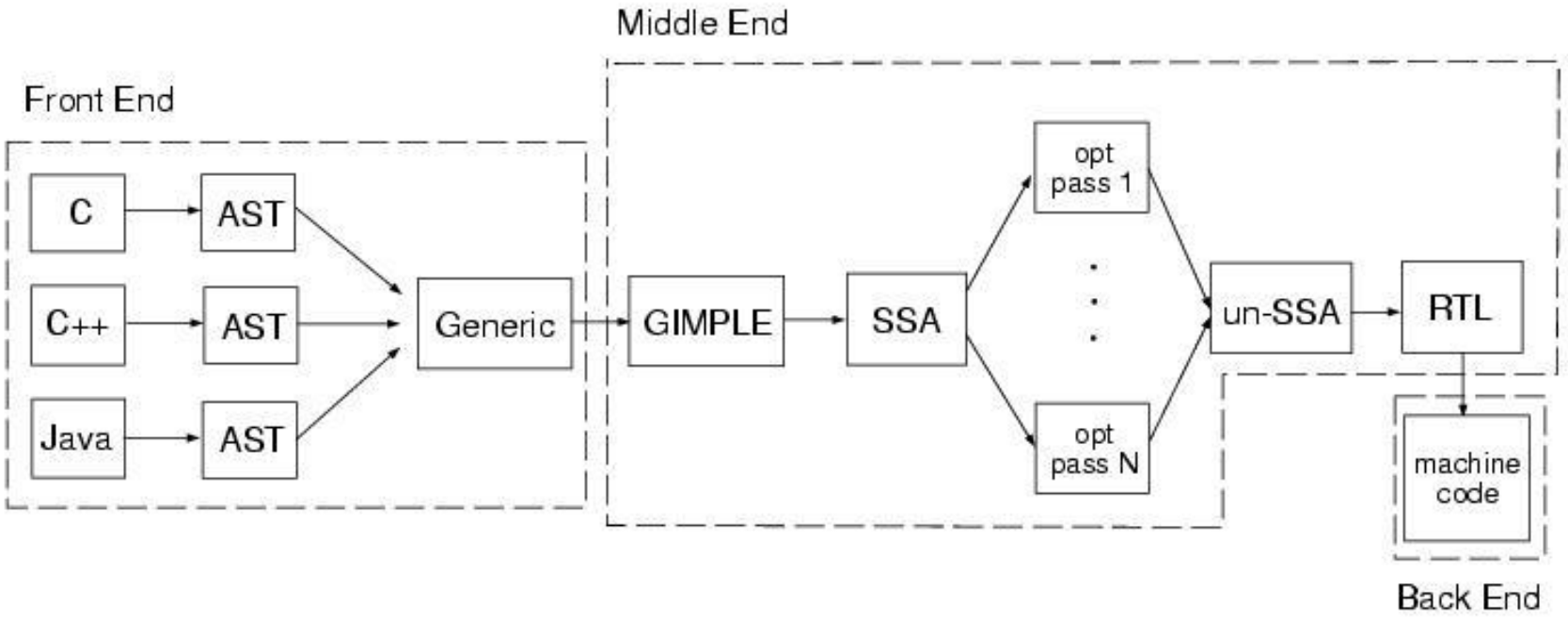
Code Generierung (Backend)

- Generiert prozessorspezifischen Assembler-Code

Quelle: intern

Compiler Toolchain

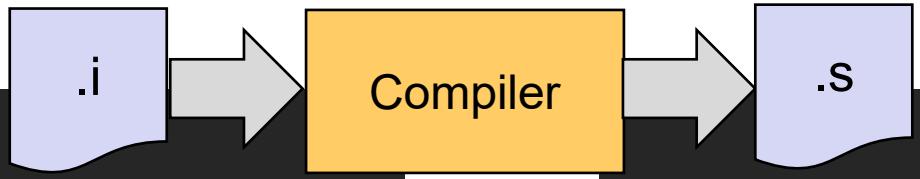
Compiler – GCC Architecture



Quelle: https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture

Compiler Toolchain

Compiler




```
# 0 "main.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "main.c"
# 17 "main.c"
static int globalVarInit = 0xABCD;
static int globalVarNoInit;

static int newResult = 0xDEADBEEF;



int main(void)
{

 globalVarNoInit = 0x20;


 int result = globalVarInit + globalVarNoInit;
 newResult = result + 0x10;

 return result;
}
```

```
globalVarNoInit:
    .space  4
    .size   globalVarNoInit, 4
    .data
    .align  2
    .type   newResult, %object
    .size   newResult, 4
newResult:
    .word   -559038737
    .text
    .align  2
    .global main
    .syntax unified
    .arm
    .type   main, %function
main:
    str fp, [sp, #-4]!
    add fp, sp, #0
    sub sp, sp, #12
    ldr r3, .L3
    mov r2, #32
    str r2, [r3]
    ldr r3, .L3+4
    ldr r2, [r3]
    ldr r3, .L3
    ldr r3, [r3]
    add r3, r2, r3
    str r3, [fp, #-8]
    ldr r3, [fp, #-8]
    add r3, r3, #16
```

Quelle: intern

Compiler – Speicherreservierung/Platzierung

Der Compiler platziert Daten und generierten Code in unterschiedliche Speicherbereiche.

Da der Compiler keinerlei Kenntnisse über das Speicher-Layout des Zielsystems hat, nutzt er sogenannte *Sections* um die entsprechenden Elemente einzuordnen.

Eine Section ist eine benannte „Sammelstelle“ ohne Bezug zu physikalischem Speicher oder speziellen Speicheradressen.

Quelle: intern

Compiler – Standard Sections

Speicherbereich	Verwendung
`.text`	Ausführbarer Code
`.rodata`	Read-Only Daten (Konstanten)
`.data`	Initialisierte Variablen
`.bss`	Nicht-initialisierte Variablen

Quelle: intern

Compiler Toolchain

Compiler – Speicherreservierung/Platzierung

```
        .data
arr:    .word 10, 20, 30, 40, 50
len:    .word 5
        .text
start:  mov r1, #10
        mov r2, #20
        .data
result: .skip 4
        .text
        add r3, r2, r1
        sub r3, r2, r1
```

.data section

```
0000_0000 arr:    .word 10, 20, 30, 40, 50
0000_0014 len:    .word 5
0000_0018 result: .skip 4
```

.text section

```
0000_0000 start:  mov r1, #10
0000_0004         mov r2, #20
0000_0008         add r3, r2, r1
0000_000C         sub r3, r2, r1
```

Quelle: intern

Compiler – Speicherreservierung und Wert-Initialisierung

```
    .data
    .align  2
    .type   globalVarInit, %object
    .size   globalVarInit, 4
globalVarInit:
    .word   43981


    .bss
    .align  2
globalVarNoInit:
    .space  4
    .size   globalVarNoInit, 4


    .data
    .align  2
    .type   newResult, %object
    .size   newResult, 4
newResult:
    .word   -559038737
```

Initialisierte Variable im .data Bereich

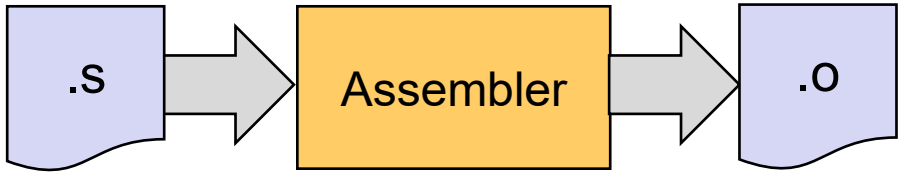
Nicht-Initialisierte Variable im .bss Bereich

Initialisierte Variable im .data Bereich

Quelle: intern

Compiler Toolchain

Assembler



- Übersetzt den generierten Assembler Code in prozessorspezifischen Maschinen-Code

Quelle: intern

Compiler Toolchain

Assembler

Der Assembler ist das erste Werkzeug in der Toolchain, welches keine Textdatei als Ausgabe-Datei erzeugt. Die erzeugten Maschinen-Codes werden als Binär-Daten abgespeichert.

Durch das Generieren von Maschinen-Code ist es notwendig, dass der Assembler in der Ausgabe-Datei auch mit virtuellen Adressen arbeitet

- Jede Section (z.B. .text, .data etc.) erhält in der Ausgabe-Datei einen eigenen Bereich
- Die virtuellen Adressen jeder Section beginnen bei 0

Neben dem Maschinen-Code erzeugt der Assembler auch eine sog. Symbol-Tabelle in der Ausgabe-Datei.

Quelle: intern

Compiler Toolchain

Assembler

Der Assembler ist das erste Werkzeug in der Toolchain, welches keine Textdatei als Ausgabe-Datei erzeugt. Die erzeugten Maschinen-Codes werden als Binär-Daten abgespeichert.

Durch das Generieren von Maschinen-Code ist es notwendig, dass der Assembler in der Ausgabe-Datei mit sog. virtuellen Adressen arbeitet

- Jede Section (z.B. .text, .data etc.) erhält in der Ausgabe-Datei einen eigenen Bereich
- Die virtuellen Adressen jeder Section beginnen bei 0

Neben dem Maschinen-Code erzeugt der Assembler auch eine sog. Symbol-Tabelle in der Ausgabe-Datei.

Quelle: intern

Compiler Toolchain

Assembler

```
1   int arr[] = { 1, 10, 4, 5, 6, 7 };
2   const int n = sizeof(arr) / sizeof(arr[0]);
3
4   extern int sum(int values[], int len);
5
6   int main()
7   {
8       int result = 0;
9
10      result = sum(arr, n);
11
12      return result;
13  }
```

```
41  main:
42      @ args = 0, pretend = 0, frame = 8
43      @ frame_needed = 1, uses_anonymous_args = 0
44      push    {r7, lr}
45      sub sp, sp, #8
46      add r7, sp, #0
47      movs    r3, #0
48      str r3, [r7, #4]
49      movs    r3, #6
50      mov r1, r3
51      ldr r0, .L3
52      bl  sum
53      str r0, [r7, #4]
54      ldr r3, [r7, #4]
55      mov r0, r3
56      adds    r7, r7, #8
57      mov sp, r7
58      @ sp needed
59      pop {r7, pc}
```

Quelle: intern

Compiler Toolchain

Assembler

```
3   int sum(int values[], int len)
4   {
5       int result = 0;
6
7       for (int i=0; i<len; i++)
8       {
9           result += values[i];
10      }
11
12      return result;
13  }
```

```
16      .global sum
17      .syntax unified
18      .thumb
19      .thumb_func
20      .type   sum, %function
21  sum:
22      @ args = 0, pretend = 0, frame = 16
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      push    {r7}
26      sub sp, sp, #20
27      add r7, sp, #0
28      str r0, [r7, #4]
29      str r1, [r7]
30      movs    r3, #0
31      str r3, [r7, #12]
32      movs    r3, #0
33      str r3, [r7, #8]
34      b   .L2
35  .L3:
36      ldr r3, [r7, #8]
```

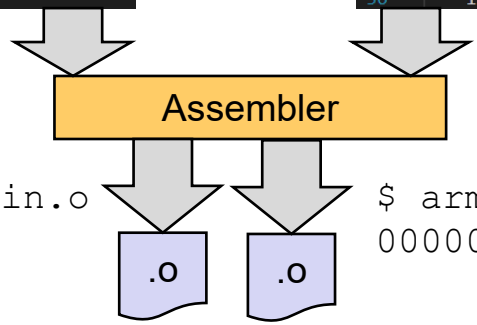
Quelle: intern

Compiler Toolchain

Assembler

```
41  main:
42      @ args = 0, pretend = 0, frame = 8
43      @ frame_needed = 1, uses_anonymous_args = 0
44      push    {r7, lr}
45      sub sp, sp, #8
46      add r7, sp, #0
47      movs    r3, #0
48      str r3, [r7, #4]
49      movs    r3, #6
50      mov r1, r3
51      ldr r0, .L3
52      bl  sum
53      str r0, [r7, #4]
54      ldr r3, [r7, #4]
55      mov r0, r3
56      adds    r7, r7, #8
57      mov sp, r7
58      @ sp needed
59      pop {r7, pc}
```

```
16      .global sum
17      .syntax unified
18      .thumb
19      .thumb_func
20      .type   sum, %function
21  sum:
22      @ args = 0, pretend = 0, frame = 16
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      push    {r7}
26      sub sp, sp, #20
27      add r7, sp, #0
28      str r0, [r7, #4]
29      str r1, [r7]
30      movs    r3, #0
31      str r3, [r7, #12]
32      movs    r3, #0
33      str r3, [r7, #8]
34      b   .L2
35  .L3:
36      ldr r3, [r7, #8]
```




```
$ arm-none-eabi-nm main.o
00000000 D arr
00000000 T main
00000000 R n
         U sum
```

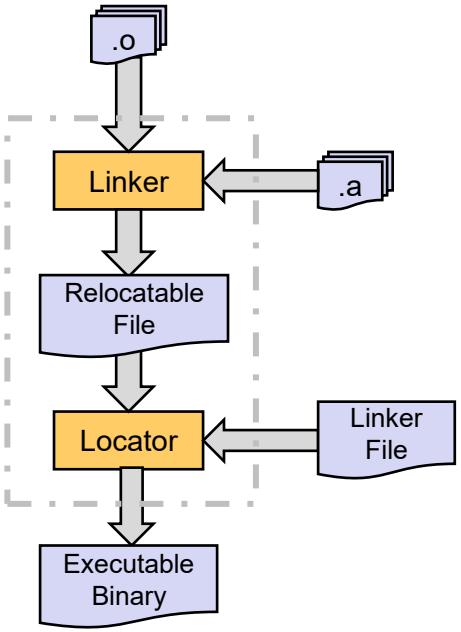
```
$ arm-none-eabi-nm sum.o
00000000 T sum
```

Quelle: intern

Linker

Linker & Locator

- Linker erstellt aus den Object-Files und statischen Bibliotheken ein sog. Relocatable-File (zumeist nur intern)
- Hierbei führt der Linker folgende Aktionen durch
  - Symbol Resolution
  - Section Merging
- Der Locator führt dann, basierend auf dem „Linker File“oder gegebenen Parametern, ein sog. Section Placement durch

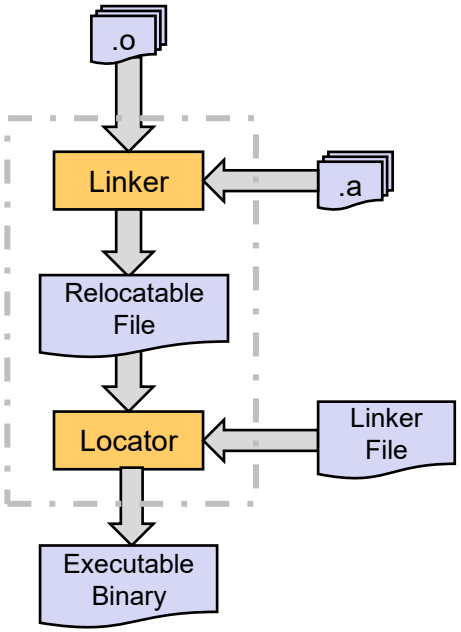


Quelle: intern

Linker

Linker & Locator

- Linker erstellt aus den Object-Files und statischen Bibliotheken ein sog. Relocatable-File (zumeist nur intern)
- Hierbei führt der Linker folgende Aktionen durch
  - **Symbol Resolution**
  - Section Merging
- Der Locator führt dann, basierend auf dem „Linker File“oder gegebenen Parametern, ein sog. Section Placement durch



Quelle: intern

Linker & Locator – Symbol Resolution

main.s

```
1         .text
2         b start                 @ Skip over the data
3   arr:  .byte 10, 20, 25        @ Read-only array of bytes
4   eoa:                          @ Address of end of array + 1
5
6         .align
7   start:
8         ldr   r0, =arr          @ r0 = &arr
9         ldr   r1, =eoa          @ r1 = &eoa
10
11        bl    sum               @ Invoke the sum subroutine
12
13  stop: b stop
```

sum.s

```
1         @ Args
2         @ r0: Start address of array
3         @ r1: End address of array
4         @
5         @ Result
6         @ r3: Sum of Array
7
8         .global sum
9
10  sum:  mov   r3, #0            @ r3 = 0
11  loop: ldrb  r2, [r0], #1      @ r2 = *r0++    ; Get array element
12        add   r3, r2, r3        @ r3 += r2      ; Calculate sum
13        cmp   r0, r1            @ if (r0 != r1) ; Check if hit end-of-array
14        bne   loop              @    goto loop  ; Loop
15        mov   pc, lr            @ pc = lr       ; Return when done
```

```
$ arm-none-eabi-nm main.o
00000004 t arr
00000007 t eoa
00000008 t start
00000014 t stop
         U sum
```

Linker

```
$ arm-none-eabi-nm sum.o
00000004 t loop
00000000 T sum
```

```
$ arm-none-eabi-nm sum.elf
00010038 T __bss_end__
00010038 T __bss_start
00010038 T __bss_start__
00010038 T __data_start
00010038 T __end__
00010038 T _bss_end__
00010038 T _edata
00010038 T _end
00080000 B _stack
         U _start
00000004 t arr
00000007 t eoa
00000024 t loop
00000008 t start
00000014 t stop
00000020 T sum
```

Quelle: http://bravegnu.org/gnu-eprog/linker.html

Exkurs: Speicherklassen

Speicherklassen für Variablen

Speicherklassen für Variablen beeinflussen neben dem Speicherort und dem Initial-Wert auch die sog. „Sichtbarkeit“ der Variable.

Hierbei sind die Speicherklassen „**extern**“ und „**static**“ für den Linker relevant.

Storage Classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	Stack	Garbage	Within block	End of block
extern	Data segment	Zero	Global, Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Quelle: intern

Exkurs: Speicherklassen

Speicherklassen für Funktionen

Speicherklassen für Funktionen beeinflussen nur die sog. „Sichtbarkeit“ der Funktion.

Hierbei sind nur die Speicherklassen „**extern**“ und „**static**“ möglich.

```c
#ifndef FLIB_H
#define FLIB_H
void f(void);              // function declaration with external linkage
extern int state;          // variable declaration with external linkage
static const int size = 5; // definition of a read-only variable with internal linkage
enum { MAX = 10 };         // constant definition
inline int sum (int a, int b) { return a + b; } // inline function definition
#endif // FLIB_H
```

Library implementation, source file "flib.c":

```c
#include "flib.h"

static void local_f(int s) {} // definition with internal linkage (only used in this file)
static int local_state;       // definition with internal linkage (only used in this file)

int state;                    // definition with external linkage (used by main.c)
void f(void) { local_f(state); } // definition with external linkage (used by main.c)
```

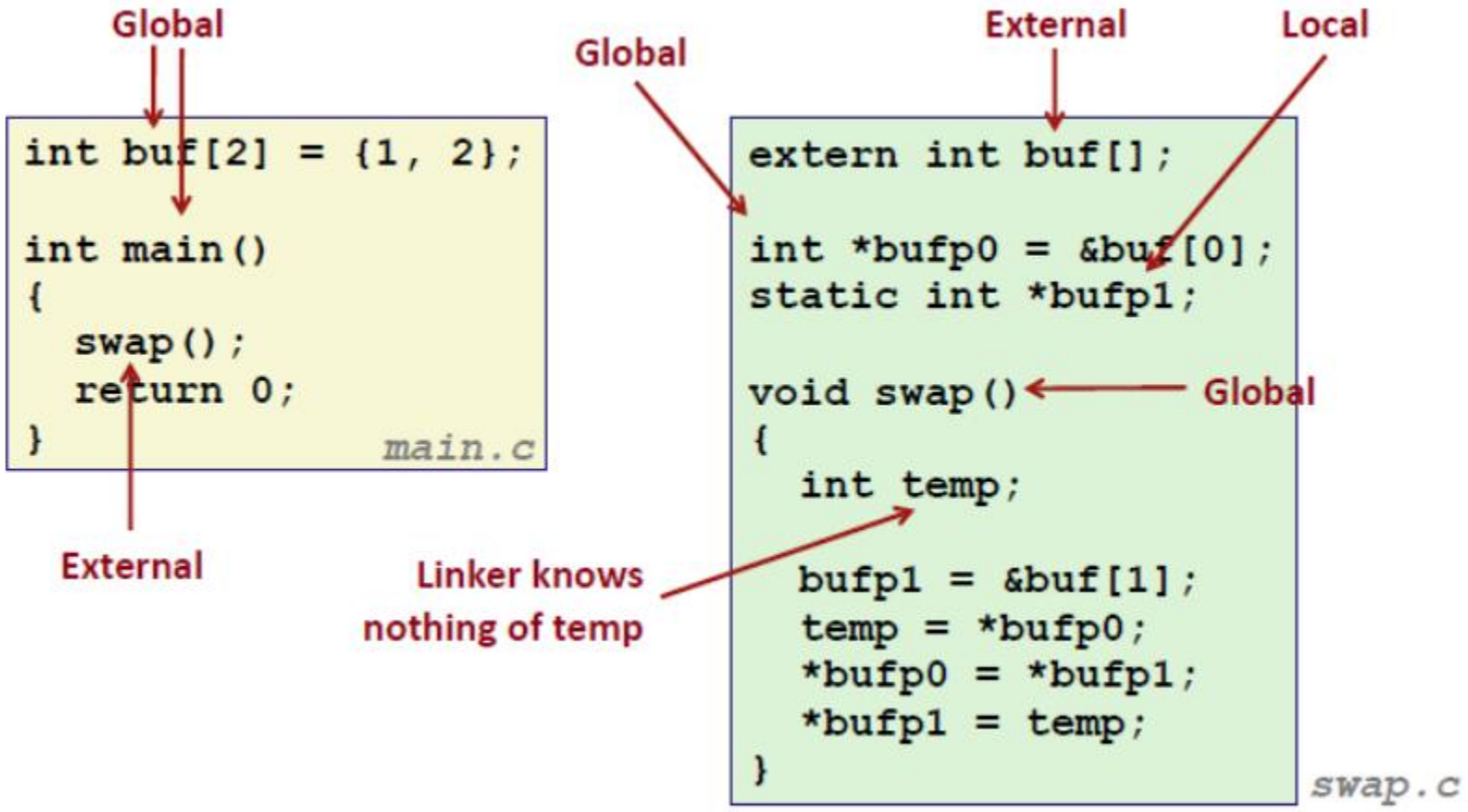
Application code, source file "main.c":

```c
#include "flib.h"

int main(void)
{
    int x[MAX] = {size}; // uses the constant and the read-only variable
    state = 7;           // modifies state in flib.c
    f();                 // calls f() in flib.c
}
```

Quelle: intern

Linker

Linker & Locator – Symbol Resolution



Quelle: intern

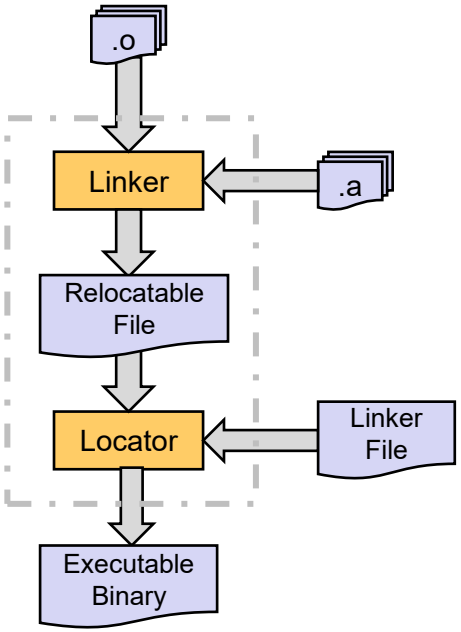
Linker

Linker & Locator

- Linker erstellt aus den Object-Files und statischen Bibliotheken ein sog. Relocatable-File (zumeist nur intern)

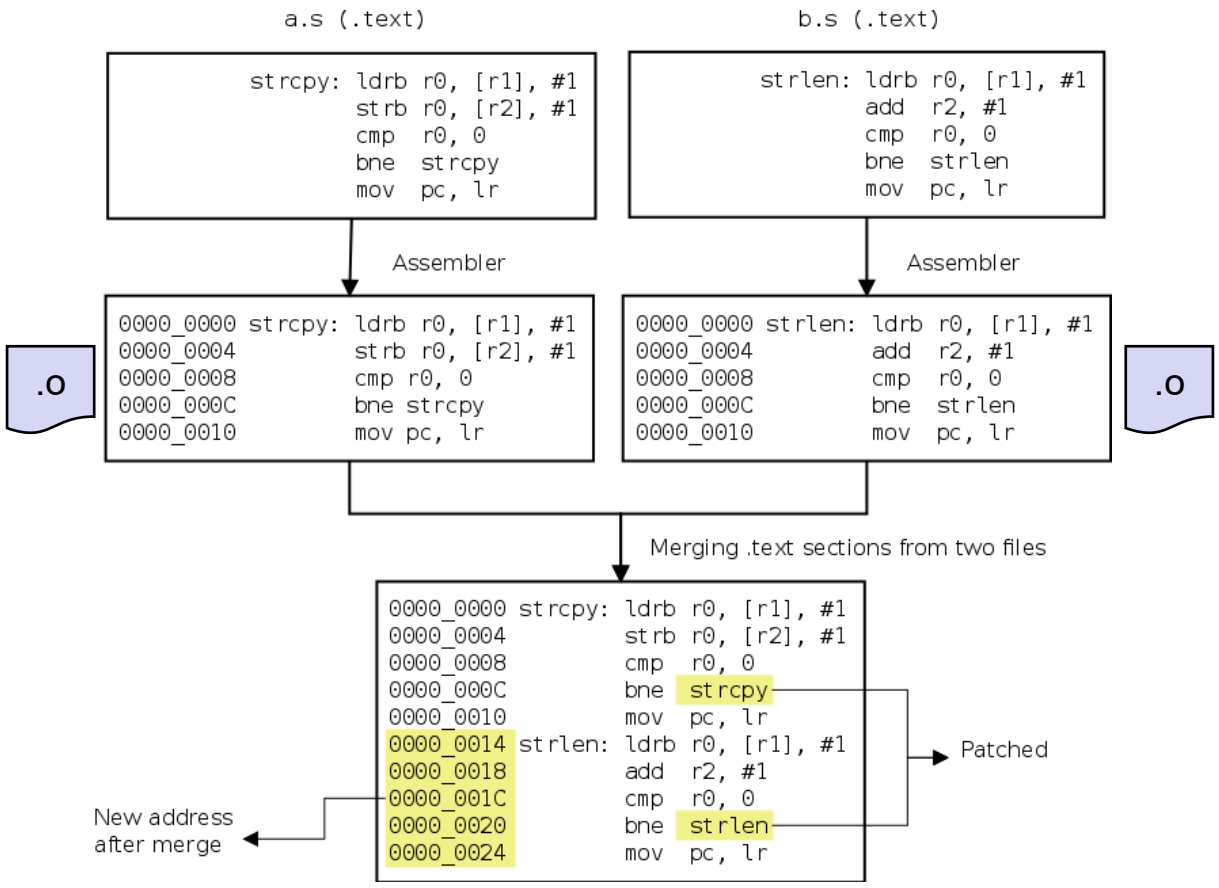
- Hierbei führt der Linker folgende Aktionen durch
  - Symbol Resolution
  - **Section Merging**

- Der Locator führt dann, basierend auf dem „Linker File“oder gegebenen Parametern, ein sog. Section Placement durch



Quelle: intern

Linker & Locator – Section Merging



Quelle: http://bravegnu.org/gnu-eprog/linker.html

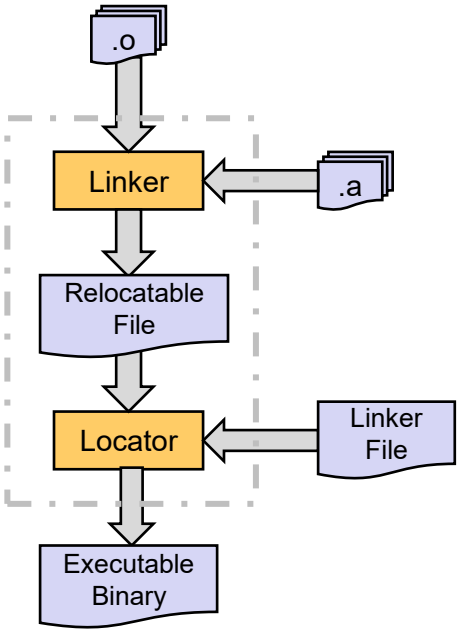
Linker

Linker & Locator

- Linker erstellt aus den Object-Files und statischen Bibliotheken ein sog. Relocatable-File (zumeist nur intern)

- Hierbei führt der Linker folgende Aktionen durch
  - Symbol Resolution
  - Section Merging

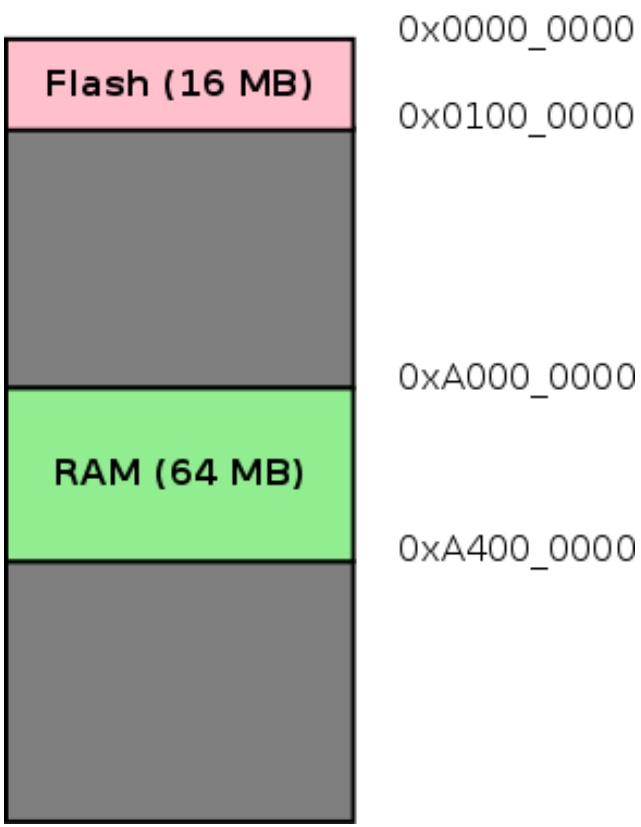
- **Der Locator führt dann, basierend auf dem „Linker File“ oder gegebenen Parametern, ein sog. Section Placement durch**



Quelle: intern

Compiler Toolchain

Linker – Speicher-Platzierung



Flash

- Als Read-Only Memory (ROM) ausgelegt.
- Über spezielle Konfiguration des sog. Flash-Controllers (interne Peripherie) beschreibbar.
- Enthält in der Regel Programm-Code und konstante Daten.

RAM

- Schreib-/Lese Speicher (Random Access Memory)
- Enthält in der Regel Variablen und veränderbare Daten

Quelle: http://bravegnu.org/gnu-eprog/using-ram.html

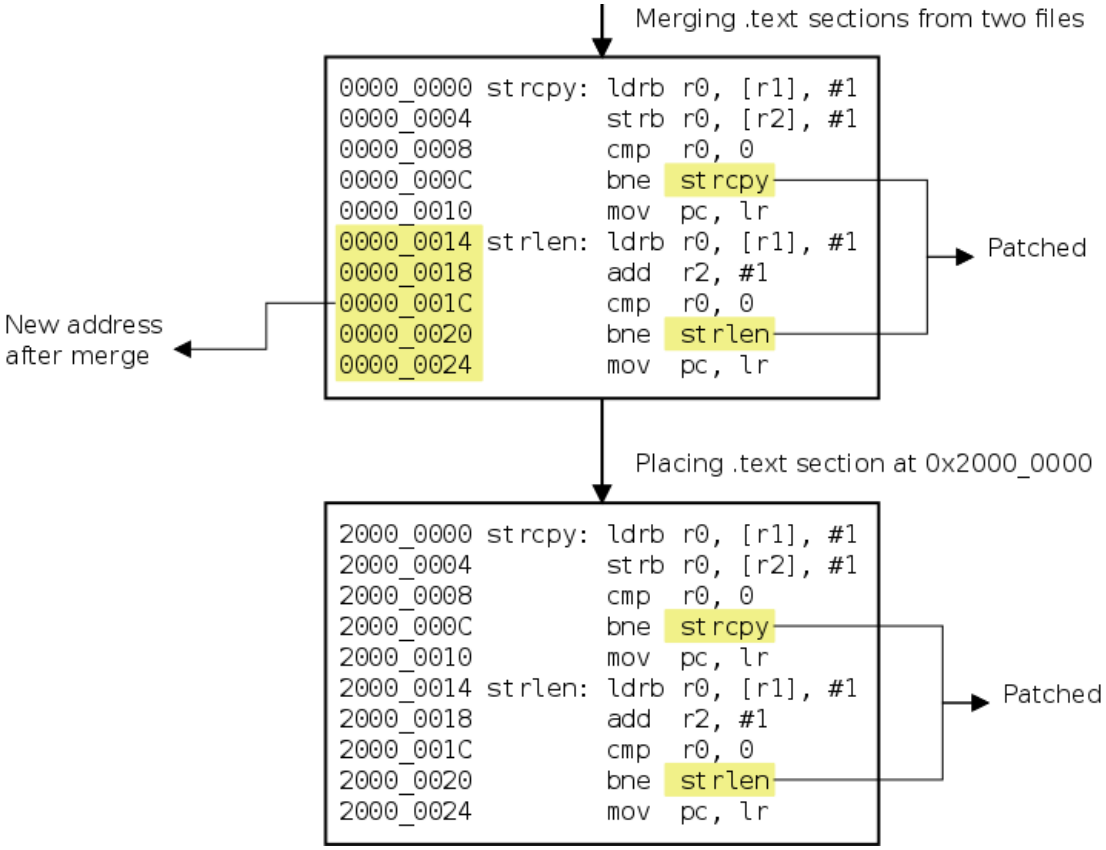
Compiler Toolchain

Linker – Speicher-Platzierung

Speicherbereich	Verwendung	phys. Speicher
`.text`	Ausführbarer Code	Flash
`.rodata`	Read-Only Daten (Konstanten)	Flash
`.data`	Initialisierte Variablen	Flash + RAM
`.bss`	Nicht-initialisierte Variablen	RAM

Quelle: intern

Linker & Locator – Section Relocation



Quelle: http://bravegnu.org/gnu-eprog/linker.html

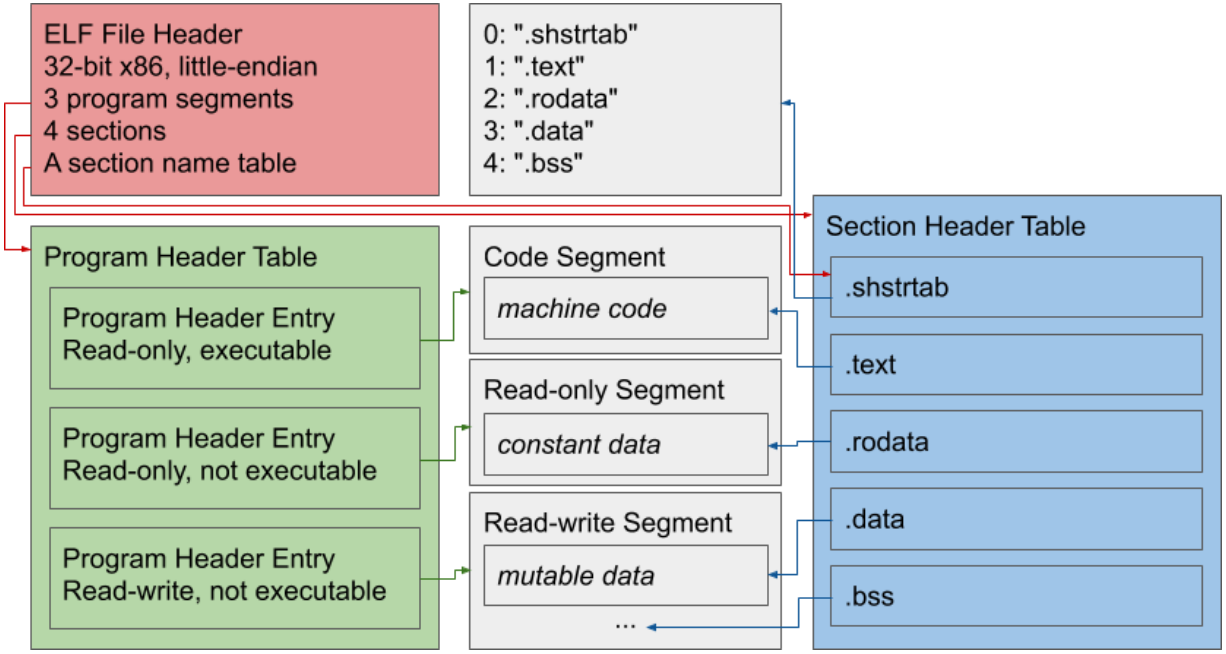
Compiler Toolchain

ELF, BIN und HEX Dateien

ELF Binaries

ELF Binary Files

- ELF – Executable and Linkable Format
  - ELF Header
  - Segments
  - Sections



Quelle: https://eoj.ie/15/

ELF Binary Files – Header

```
$ arm-none-eabi-readelf -h firmware.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8000251
  Start of program headers:          52 (bytes into file)
  Start of section headers:          29596 (bytes into file)
  Flags:                             0x5000200, Version5 EABI, soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         25
  Section header string table index: 24
```

Quelle: https://eoj.ie/15/

ELF Binary Files – Segments

```
$ arm-none-eabi-readelf -l firmware.elf

Elf file type is EXEC (Executable file)
Entry point 0x8000251
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x08000000 0x08000000 0x002a0 0x002a0 R E 0x1000
  LOAD           0x002000 0x20000000 0x080002a0 0x00018 0x00618 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .isr_vector .text .rodata
   01     .data .bss ._user_heap_stack
```

Quelle: https://eoj.ie/15/

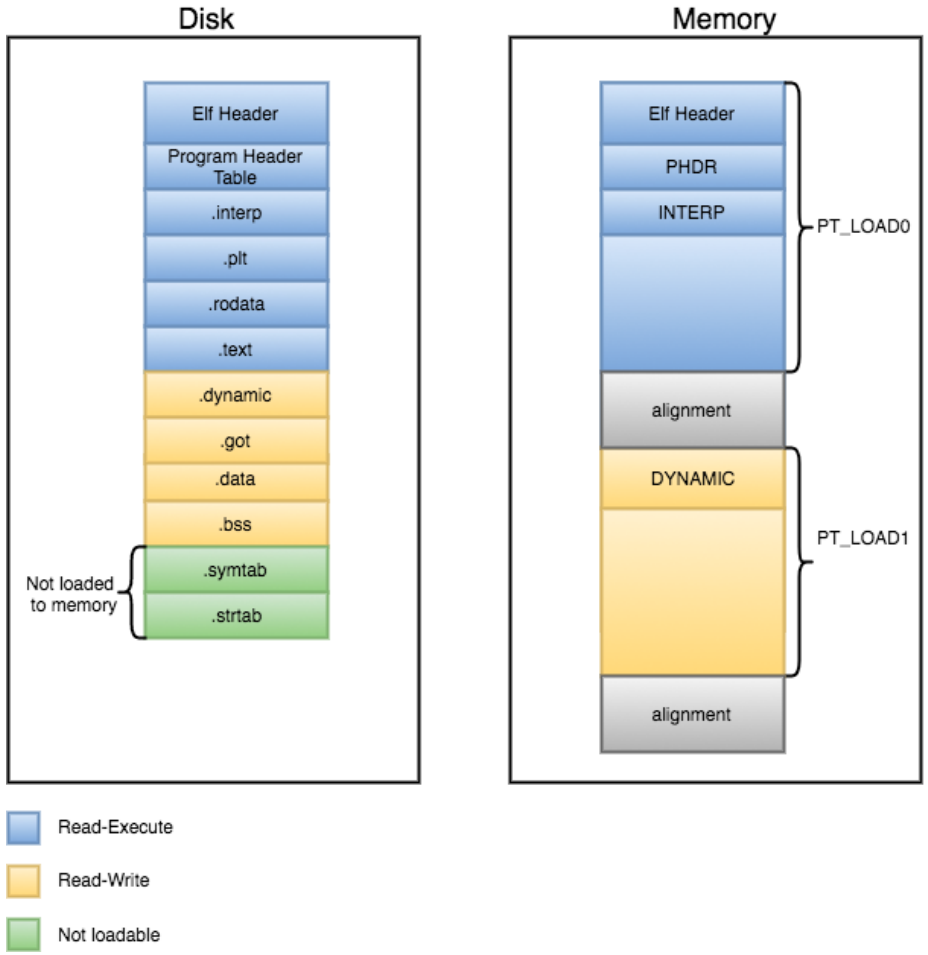
ELF Binary Files – Sections

```
λ arm-none-eabi-readelf -S firmware.elf
There are 25 section headers, starting at offset 0x739c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .isr_vector       PROGBITS        08000000 001000 0001d8 00   A  0   0  1
  [ 2] .text             PROGBITS        080001d8 0011d8 0000c4 00  AX  0   0  4
  [ 3] .rodata           PROGBITS        0800029c 00129c 000004 00   A  0   0  4
  [ 4] .ARM.extab        PROGBITS        080002a0 002018 000000 00   W  0   0  1
  [ 5] .ARM              PROGBITS        080002a0 002018 000000 00   W  0   0  1
  [ 6] .preinit_array    PREINIT_ARRAY   080002a0 002018 000000 04  WA  0   0  1
  [ 7] .init_array       INIT_ARRAY      080002a0 002018 000000 04  WA  0   0  1
  [ 8] .fini_array       FINI_ARRAY      080002a0 002018 000000 04  WA  0   0  1
  [ 9] .data             PROGBITS        20000000 002000 000018 00  WA  0   0  4
  [10] .bss              NOBITS          20000018 002018 000000 00  WA  0   0  1
  [11] ._user_heap_stack NOBITS          20000018 002018 000600 00  WA  0   0  1
  [12] .ARM.attributes   ARM_ATTRIBUTES  00000000 002018 00002c 00      0   0  1
  [13] .debug_info       PROGBITS        00000000 002044 00017c 00      0   0  1
  [14] .debug_abbrev     PROGBITS        00000000 0021c0 00015b 00      0   0  1
  [15] .debug_aranges    PROGBITS        00000000 002320 000068 00      0   0  8
  [16] .debug_macro      PROGBITS        00000000 002388 000adc 00      0   0  1
  [17] .debug_line       PROGBITS        00000000 002e64 000105 00      0   0  1
  [18] .debug_str        PROGBITS        00000000 002f69 002dfe 01  MS  0   0  1
  [19] .comment          PROGBITS        00000000 005d67 000045 01  MS  0   0  1
  [20] .debug_frame      PROGBITS        00000000 005dac 00006c 00      0   0  4
  [21] .debug_ranges     PROGBITS        00000000 005e18 000020 00      0   0  8
  [22] .symtab           SYMTAB          00000000 005e38 000ab0 10     23  41  4
  [23] .strtab           STRTAB          00000000 0068e8 0009a8 00      0   0  1
  [24] .shstrtab         STRTAB          00000000 007290 00010a 00      0   0  1
```

ELF Binaries

ELF Binary Files

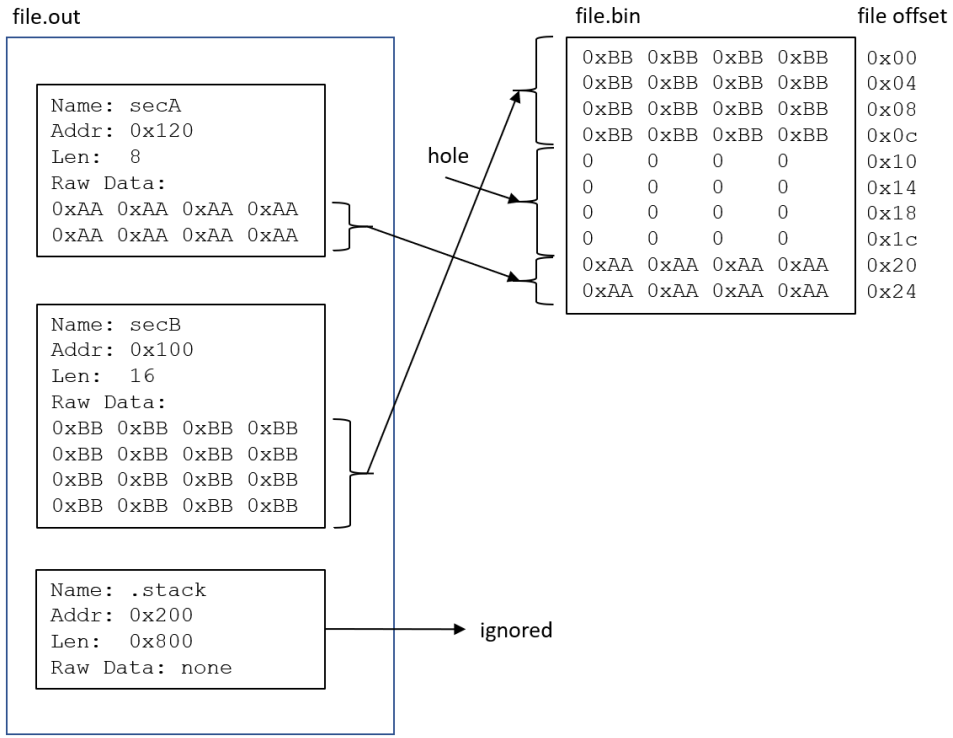
- Segmente werden u.a. für das Laden der Executable auf einem System wie z.B. Linux verwendet



Quelle: intern

Binary Files - .BIN

- Enthalten in der Regel ein Binärabbild eines Speicherbereichs
- Enthält keine Adress-Angaben für den Inhalt



Quelle: intern

Topic

Binary Files - .BIN

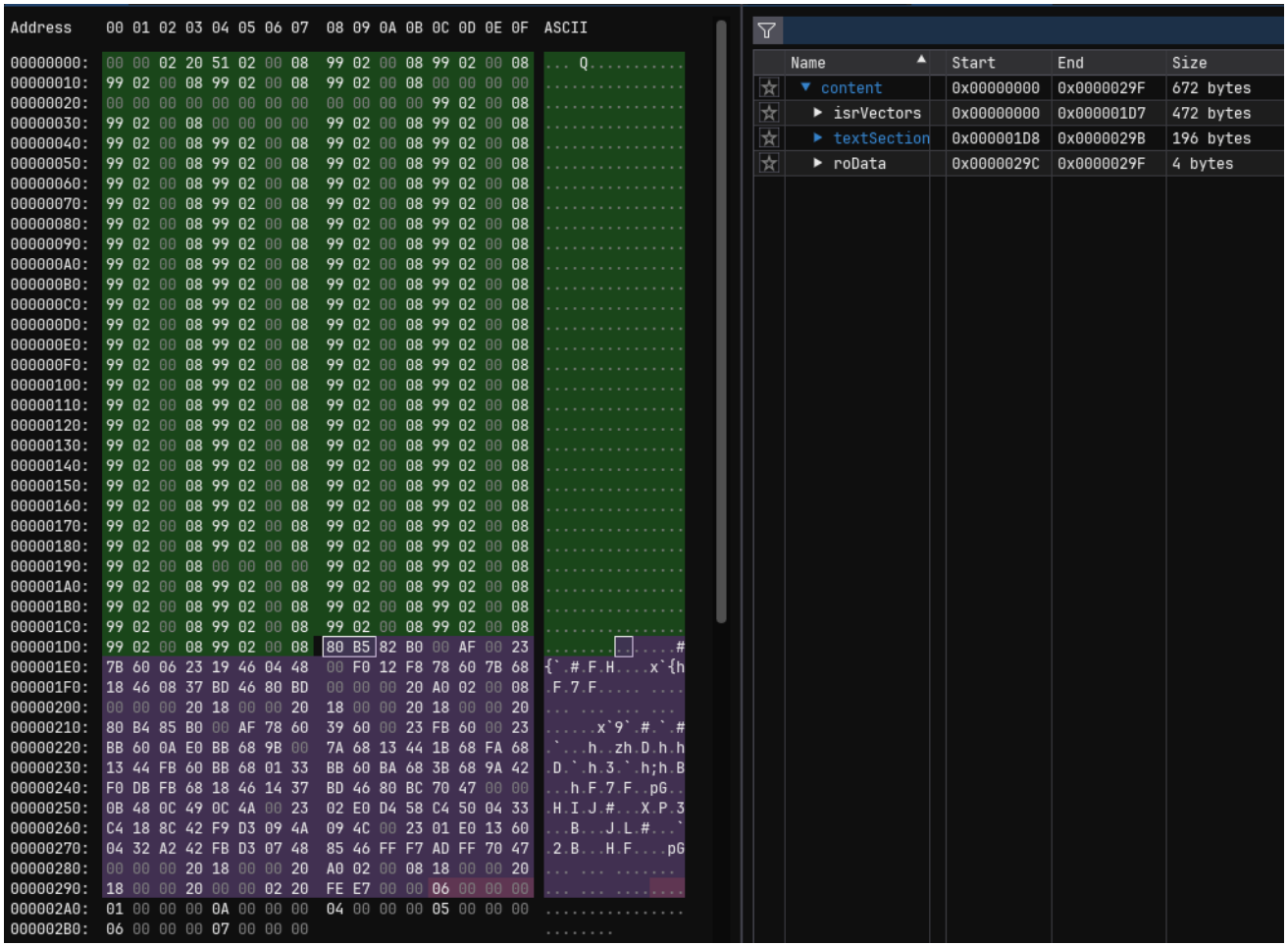
```
Address    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ASCII

000000A0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000B0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000C0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000D0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000E0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000F0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000100:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000110:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000120:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000130:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000140:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000150:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000160:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000170:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000180:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000190:  99 02 00 08 00 00 00 00  99 02 00 08 99 02 00 08  ................
000001A0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000001B0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000001C0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000001D0:  99 02 00 08 99 02 00 08  80 B5 82 B0 00 AF 00 23  ...............#
000001E0:  7B 60 06 23 19 46 04 48  00 F0 12 F8 78 60 7B 68  {`.#.F.H....x`{h
000001F0:  18 46 08 37 BD 46 80 BD  00 00 00 20 A0 02 00 08  .F.7.F..... ....
00000200:  00 00 00 20 18 00 00 20  18 00 00 20 18 00 00 20  ... ... ... ...
00000210:  80 B4 85 B0 00 AF 78 60  39 60 00 23 FB 60 00 23  ......x`9`.#.`.#
00000220:  BB 60 0A E0 BB 68 9B 00  7A 68 13 44 1B 68 FA 68  .`...h..zh.D.h.h
00000230:  13 44 FB 60 BB 68 01 33  BB 60 BA 68 3B 68 9A 42  .D.`.h.3.`.h;h.B
00000240:  F0 DB FB 68 18 46 14 37  BD 46 80 BC 70 47 00 00  ...h.F.7.F..pG..
00000250:  0B 48 0C 49 0C 4A 00 23  02 E0 D4 58 C4 50 04 33  .H.I.J.#...X.P.3
00000260:  C4 18 8C 42 F9 D3 09 4A  09 4C 00 23 01 E0 13 60  ...B...J.L.#...`
00000270:  04 32 A2 42 FB D3 07 48  85 46 FF F7 AD FF 70 47  .2.B...H.F....pG
00000280:  00 00 00 20 18 00 00 20  A0 02 00 08 18 00 00 20  ... ... ....... 
00000290:  18 00 00 20 00 00 02 20  FE E7 00 00 06 00 00 00  ... ... ........
000002A0:  01 00 00 00 0A 00 00 00  04 00 00 00 05 00 00 00  ................
000002B0:  06 00 00 00 07 00 00 00                          ........
```

Beispiel .bin File enthält nur die Inhalte, welche in den Flash geladen werden müssen

Quelle: intern

Topic

Binary Files - .BIN



```
Address    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ASCII
00000000:  00 00 02 20 51 02 00 08  99 02 00 08 99 02 00 08  ... Q...........
00000010:  99 02 00 08 99 02 00 08  99 02 00 08 00 00 00 00  ................
00000020:  00 00 00 00 00 00 00 00  00 00 00 00 99 02 00 08  ................
00000030:  99 02 00 08 00 00 00 00  99 02 00 08 99 02 00 08  ................
00000040:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000050:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000060:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000070:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000080:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000090:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000A0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000B0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000C0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000D0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000E0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000000F0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000100:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000110:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000120:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000130:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000140:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000150:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000160:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000170:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000180:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
00000190:  99 02 00 08 00 00 00 00  99 02 00 08 99 02 00 08  ................
000001A0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000001B0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000001C0:  99 02 00 08 99 02 00 08  99 02 00 08 99 02 00 08  ................
000001D0:  99 02 00 08 99 02 00 08  80 B5 82 B0 00 AF 00 23  ...............#
000001E0:  7B 60 06 23 19 46 04 48  00 F0 12 F8 78 60 7B 68  {`.#.F.H....x`{h
000001F0:  18 46 08 37 BD 46 80 BD  00 00 00 00 20 A0 02 00  .F.7.F...... ...
00000200:  00 00 00 20 18 00 00 20  18 00 00 20 18 00 00 20  ... ... ... ...
00000210:  80 B4 85 B0 00 AF 78 60  39 60 00 23 FB 60 00 23  ......x`9`.#.`.#
00000220:  BB 60 0A E0 BB 68 9B 00  7A 68 13 44 1B 68 FA 68  .`...h..zh.D.h.h
00000230:  13 44 FB 60 BB 68 01 33  BB 60 BA 68 3B 68 9A 42  .D.`.h.3.`.h;h.B
00000240:  F0 DB FB 68 18 46 14 37  BD 46 80 BC 70 47 00 00  ...h.F.7.F..pG..
00000250:  0B 48 0C 49 0C 4A 00 23  02 E0 D4 58 C4 50 04 33  .H.I.J.#...X.P.3
00000260:  C4 18 8C 42 F9 D3 09 4A  09 4C 00 23 01 E0 13 60  ...B...J.L.#...`
00000270:  04 32 A2 42 FB D3 07 48  85 46 FF F7 AD FF 70 47  .2.B...H.F....pG
00000280:  00 00 00 20 18 00 00 20  A0 02 00 08 18 00 00 20  ... ... ........
00000290:  18 00 00 20 00 00 02 20  FE E7 00 00 06 00 00 00  ... ... ........
000002A0:  01 00 00 00 0A 00 00 00  04 00 00 00 05 00 00 00  ................
000002B0:  06 00 00 00 07 00 00 00                          ........
```

Name	Start	End	Size
content	0x00000000	0x0000029F	672 bytes
isrVectors	0x00000000	0x000001D7	472 bytes
textSection	0x000001D8	0x0000029B	196 bytes
roData	0x0000029C	0x0000029F	4 bytes

Quelle: intern

Topic

Intel HEX Files - .HEX

- Textdatei in speziellem Format
- Speicherinhalte in hexadezimaler Darstellung
- Enthält auch Adressangaben

```
:020000021000EC
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

- Startcode
- Byte count
- Adresse
- Typ
- Datenfeld
- Prüfsumme

Die Prüfsumme für den zweiten Beispiel-Datensatz berechnet sich wie folgt: $10+01+00+00+21+46+01+36+01+21+47+01+36+00+7E+FE+09+D2+19+01 = 3C0 \rightarrow C0 \rightarrow \neg C0 + 1 = 40.$

Quelle: https://de.wikipedia.org/wiki/Intel_HEX

Topic

Motorola HEX Files - .SREC

- Textdatei in speziellem Format
- Speicherinhalte in hexadezimaler Darstellung
- Enthält auch Adressangaben

```
S00F000068656C6C6F212020202000003B
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026
S11F001C4BFFFFE5398000007D83637880010014382100107C0803A64E800020E9
S1110003848656C6C6F20776F726C642E0A0042
S5030003F9
S9030000FC
```

- Start code
- Record type (Datensatztyp)
- Byte count
- Adresse
- Daten
- Checksum (Prüfsumme)

Die Prüfsumme für den ersten Beispiel-Datensatz berechnet sich wie folgt: $0F + 00 + 00 + 68 + 65 + 6C + 6C + 6F + 21 + 20 + 20 + 20 + 20 + 00 + 00 = 2C4 \rightarrow C4 \rightarrow \neg C4 = 3B$.

Quelle: https://de.wikipedia.org/wiki/S-Record