Vorlesungsskript zu „Vertiefung Programmieren“ Speicherlayout + Linker

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

DHBW
Duale Hochschule
Baden-Württemberg

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

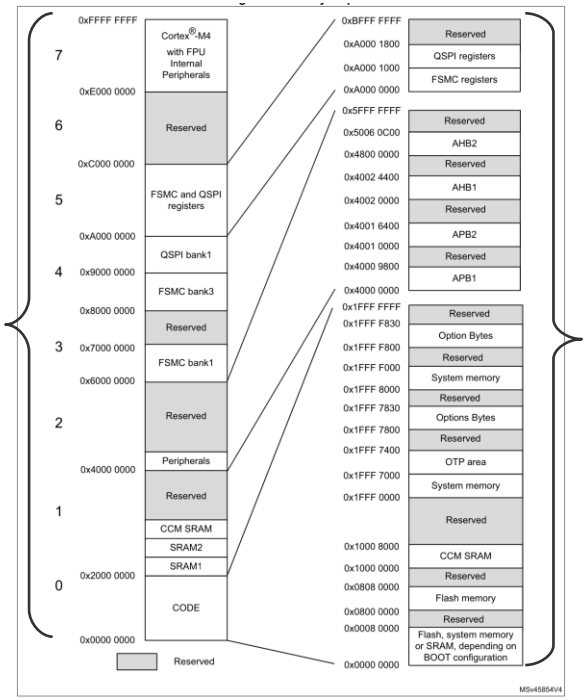
Speicherlayout und Linker

GNU Compiler Toolchain

Speicherlayout

Mikrocontroller und Prozessorsysteme haben in der Regel ein vorgegebenes Speicherlayout (engl. Memory Map).
Hierbei ist festgelegt, an welchen Adressen des verfügbaren Adressraums, welche Speicher- bzw. Peripherie-Komponenten verfügbar sind.

32-Bit Adressraum

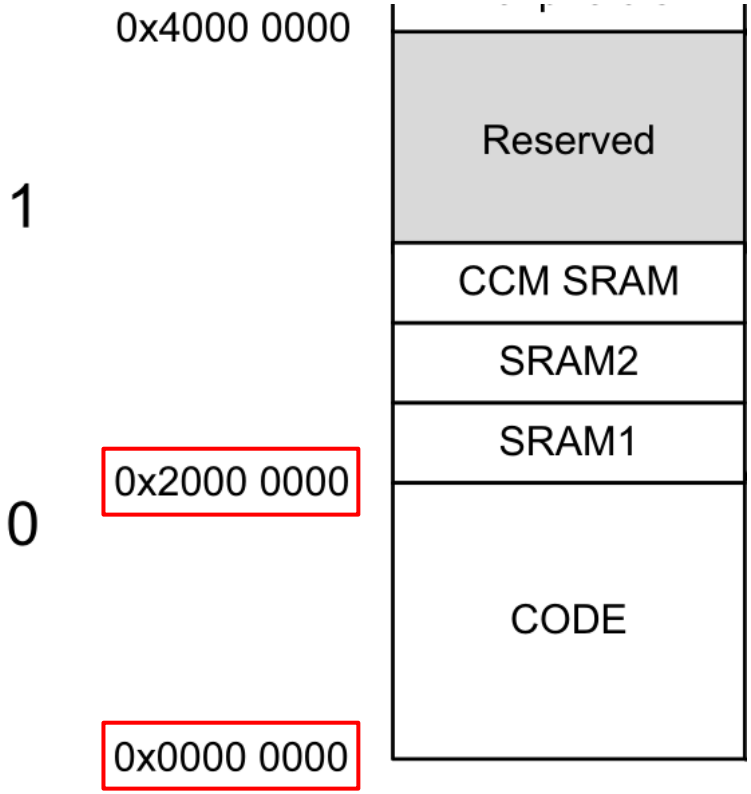


Speicher- und Peripherie Zuordnung

Quelle: intern

GNU Compiler Toolchain

Speicherlayout



Der gezeigte Ausschnitt aus einer Memory Map des STM32G474 Microcontrollers zeigt z.B. an welchen Adressen sich Flash Speicher (Code) sowie RAM Speicher (SRAM1 u. SRAM2) befinden.

The STM32G4 Series category 3 devices feature up to 128 Kbytes SRAM:

- 80 Kbytes SRAM1 (mapped at address 0x2000 0000)
- 16 Kbytes SRAM2 (mapped at address 0x2001 4000)
- 32 Kbytes CCM SRAM (mapped at address 0x1000 0000 and end of SRAM2)

The STM32G4 Series category 4 devices feature up to 112 Kbytes SRAM:

- 80 Kbytes SRAM1 (mapped at address 0x2000 0000)
- 16 Kbytes SRAM2 (mapped at address 0x2001 4000)
- 16 Kbytes CCM SRAM (mapped at address 0x1000 0000 and end of SRAM2)

The STM32G4 Series category 2 devices feature up to 32 Kbytes SRAM:

- 16 Kbytes SRAM1 (mapped at address 0x2000 0000)
- 6 Kbytes SRAM2 (mapped at address 0x2000 4000)
- 10 Kbytes CCM SRAM (mapped at address 0x1000 0000 and end of SRAM2)

Quelle: intern

GNU Compiler Toolchain

Speicherlayout

Da der Compiler und der Assembler keinerlei Kenntnis über das Speicherlayout des Zielsystems haben, und diese Information auch nicht benötigen, muss der Linker als letzte Komponente in der Toolchain diese Informationen bereitgestellt bekommen.

Der Linker benötigt Kenntnis über die verfügbaren Speicherbereiche (Startadresse und Länge bzw. Größe des Speichers) sowie eine Information darüber, welche Inhalte in die Speicherbereiche gelegt werden sollen

Das Speicherlayout sowie die genutzten Sections können sehr komplex sein. Daher werden diese Informationen in der Regel nicht mehr über Kommandozeilen Parameter an den Linker übergeben, sondern mit Hilfe eines sog. Linker Files

Quelle: intern

GNU Compiler Toolchain

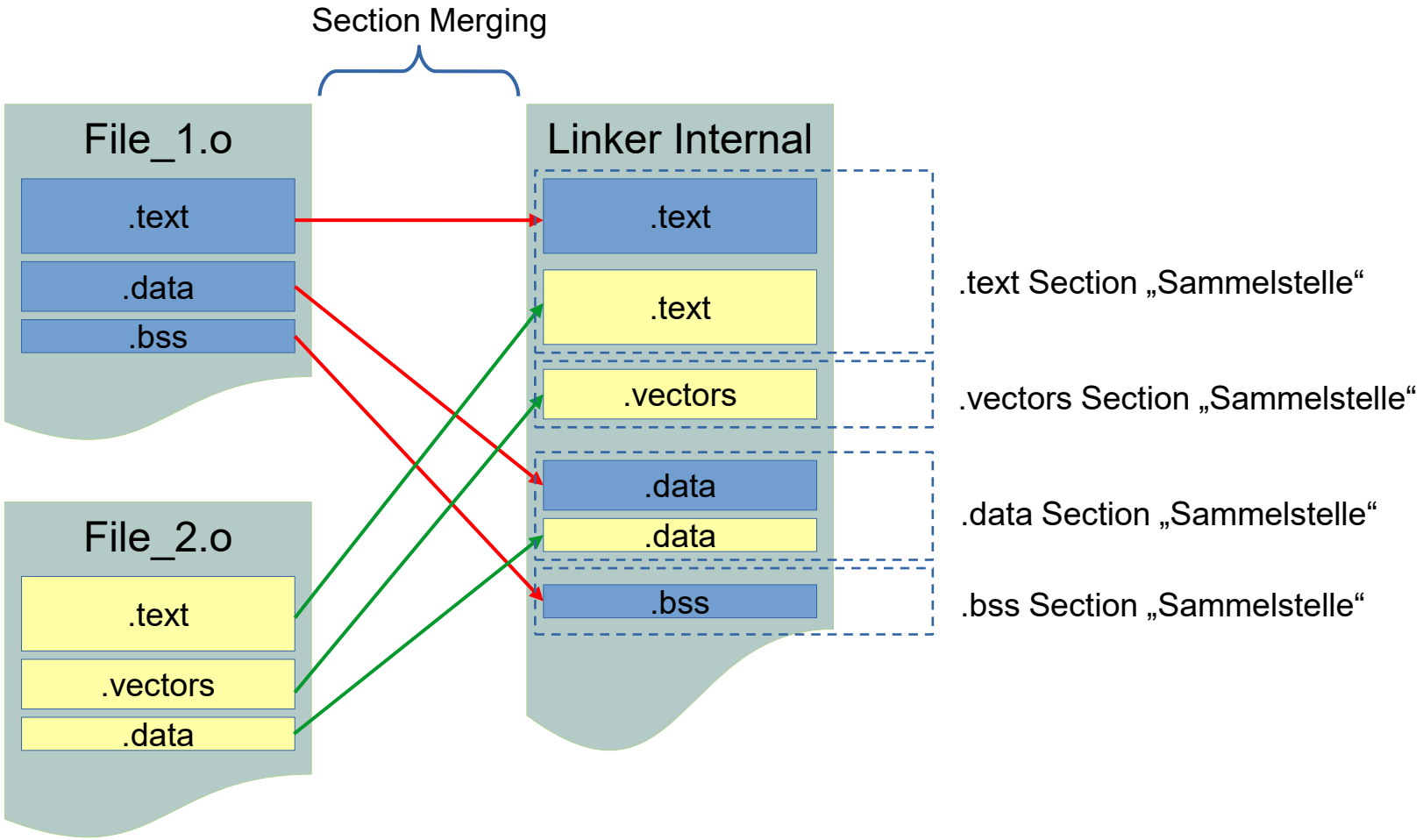
Compiler – Speicherreservierung/Platzierung

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

Quelle: intern

GNU Compiler Toolchain

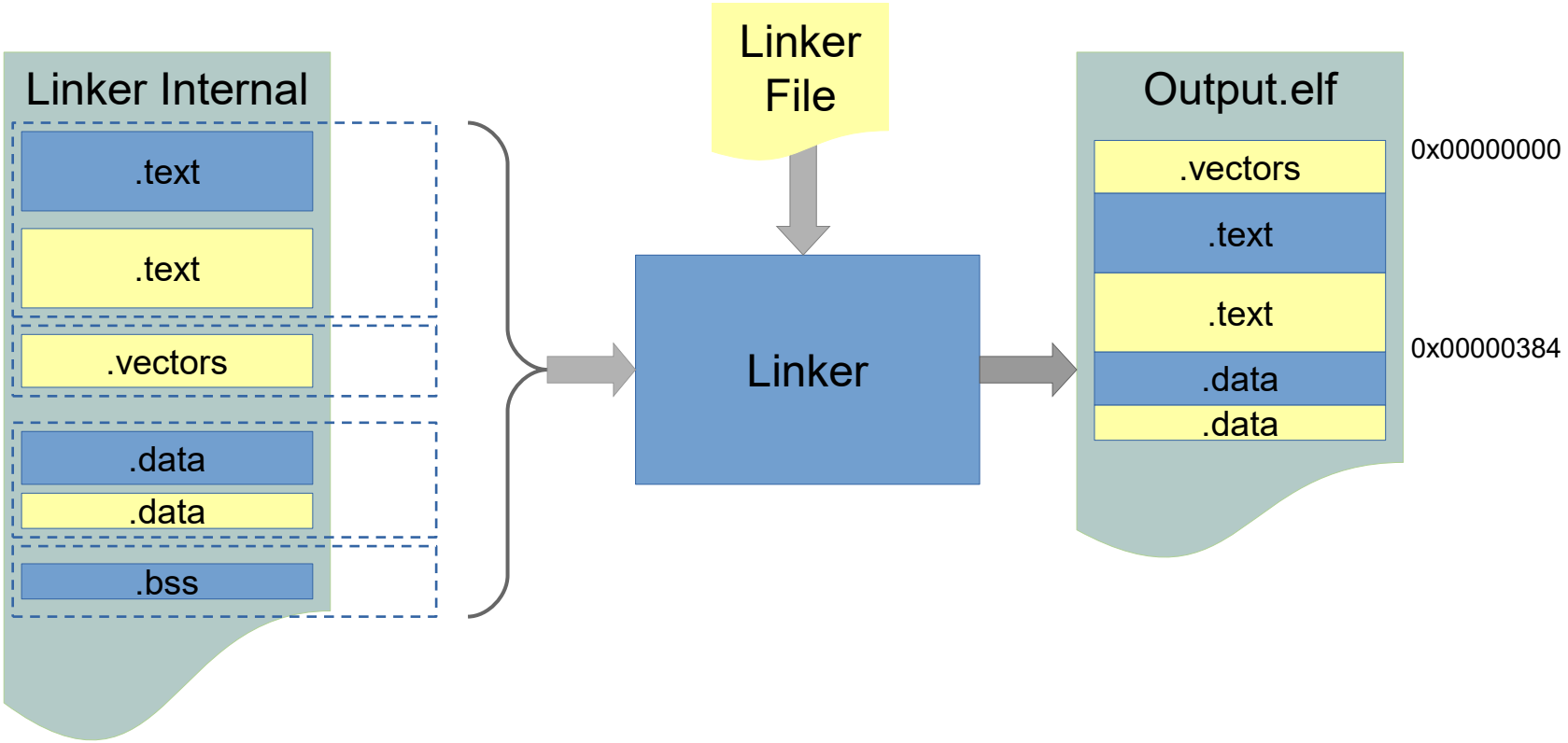
Linker File



Quelle: intern

GNU Compiler Toolchain

Linker File



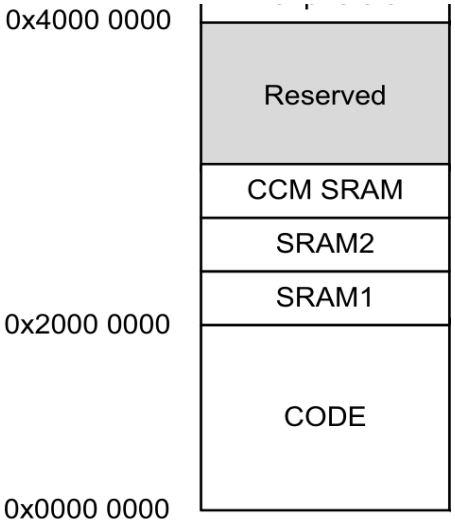
Quelle: intern

Linker File - Memory

Das Linker File definiert als erstes die grundlegenden, physikalischen Speicherbereiche, die das Zielsystem besitzt.

Hierbei können im Rahmen der Memory Map bzw. der Adressen des Systems die Speicherbereich frei definiert werden. Z.B. Könnte der RAM Bereich auch in zwei unterschiedliche Bereiche SRAM1 und SRAM2 aufgeteilt werden.

```
MEMORY
{
  RAM     (xrw)      : ORIGIN = 0x20000000,   LENGTH = 128K
  FLASH   (rx)       : ORIGIN = 0x00000000,   LENGTH = 512K
}
```



Name des Speicherbereichs

Attribute R/W/X

Start Adresse

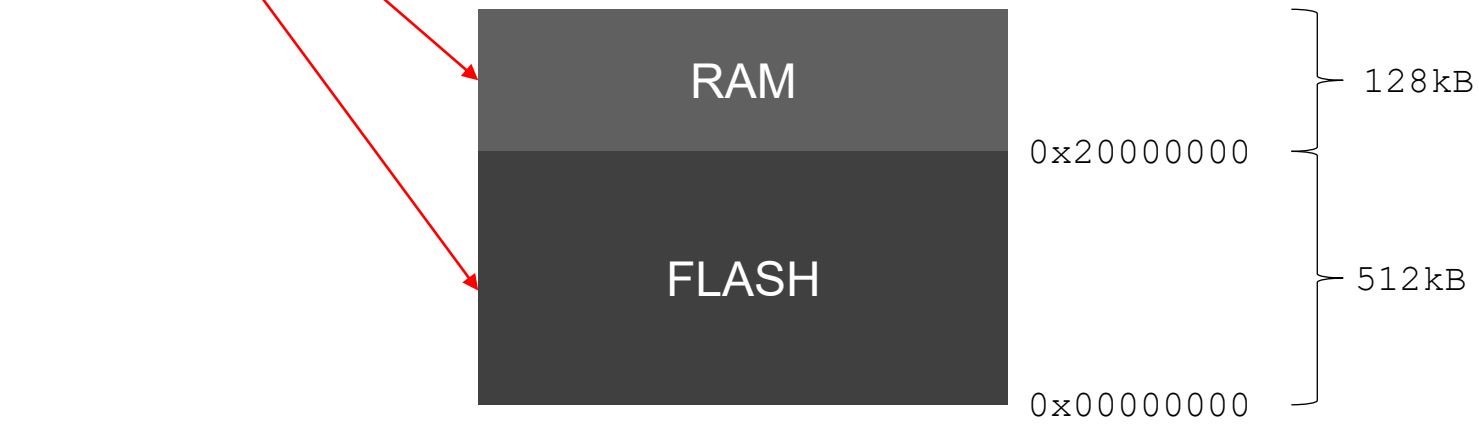
Größe

Quelle: intern

Linker File - Memory

Das Linker File definiert als erstes die grundlegenden, physikalischen Speicherbereiche, die das Zielsystem besitzt.

```
MEMORY
{
  RAM    (xrw)      : ORIGIN = 0x20000000,   LENGTH = 128K
  FLASH  (rx)       : ORIGIN = 0x00000000,   LENGTH = 512K
}
```



Quelle: intern

Linker File - Memory

A linker script may contain many uses of the MEMORY command, however, all memory blocks defined are treated as if they were specified inside a single MEMORY command. The syntax for MEMORY is:

```
MEMORY
  {
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
  }
```

The *name* is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name within the MEMORY command

The *attr* string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

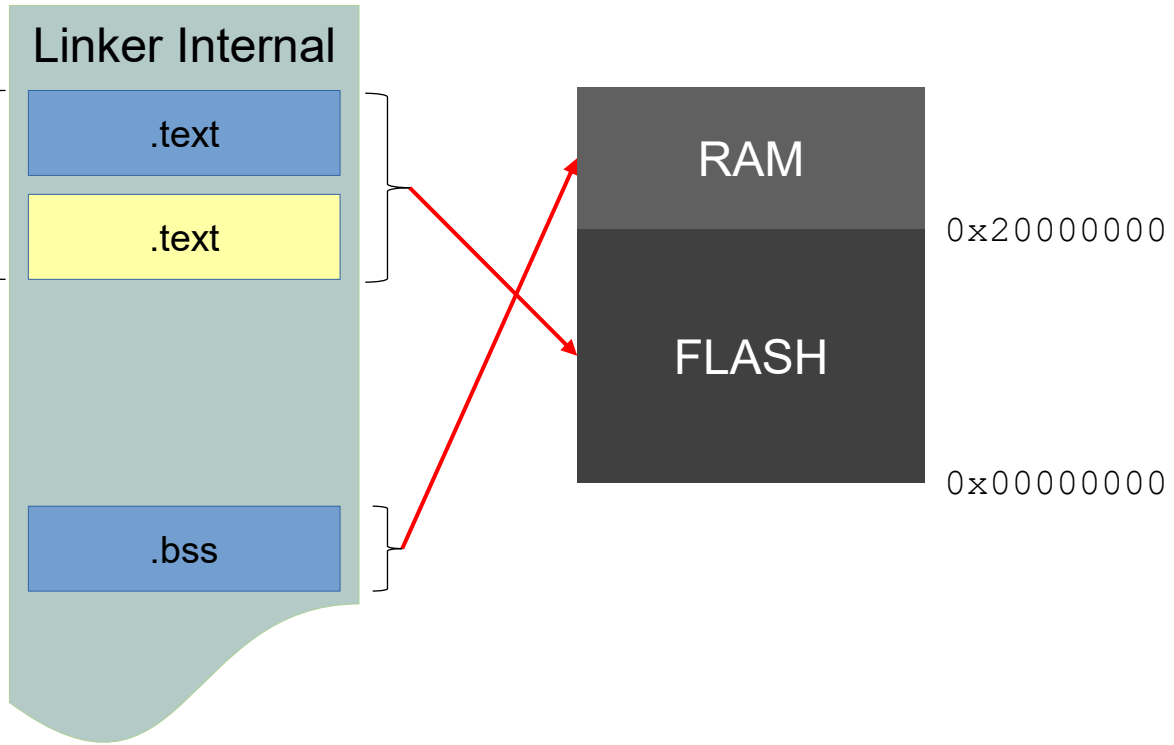
GNU Compiler Toolchain

Linker File - Sections

Nachdem die Speicher-Struktur festgelegt ist, werden dem Linker die vorhandenen Sections sowie deren Position im definierten Speicher mitgeteilt

```
SECTIONS
{
  .text :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
    . = ALIGN(4);
  } >FLASH

  .bss :
  {
    . = ALIGN(4);
    *(.bss)
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
  } >RAM
}
```



Quelle: intern

Linker File - Sections

Nachdem die Speicher-Struktur festgelegt ist, werden dem Linker die vorhandenen Sections sowie deren Position im definierten Speicher mitgeteilt

```
SECTIONS
{
  .text :
  {
    . = ALIGN(4);
    *(.text)
    *(.text*)
    . = ALIGN(4);
  } >FLASH

  .bss :
  {
    . = ALIGN(4);
    *(.bss)
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
  } >RAM
}
```

Name der Section

Inhalt der Section

Ziel-Speicherbereich der Section

Quelle: intern

Linker File - Sections

The full description of an output section looks like this:

```
section [address] [(type)] :
  [AT(lma)]
  [ALIGN(section_align) | ALIGN_WITH_INPUT]
  [SUBALIGN(subsection_align)]
  [constraint]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
[,]
```

Most output sections do not use most of the optional section attributes.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

Linker File - Sections

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

NOLOAD
The section should be marked as not loadable, so that it will not be loaded into memory when the program is run. When generating an ELF output file, the memory space is allocated for the section at run-time, except for SHT_NOTE sections. For other output files, no memory space is allocated at run-time.

Note - the ELF behaviour is a bug that may change to be consistent with non-ELF targets.

READONLY
The section should be marked as read-only.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

Linker File - Sections

Every loadable or allocatable output section has two addresses. The first is the *VMA*, or virtual memory address. This is the address the section will have when the output file is run. The second is the *LMA*, or load memory address. This is the address at which the section will be loaded. In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system). In this case the ROM address would be the LMA, and the RAM address would be the VMA.

The AT keyword takes an expression as an argument. This specifies the exact load address of the section. The AT> keyword takes the name of a memory region as an argument. See MEMORY Command. The load address of the section is set to the next free address in the region, aligned to the section's alignment requirements.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

Linker File – Symbol Assignments

You may assign to a symbol using any of the C assignment operators:

symbol = *expression* ;
symbol += *expression* ;
symbol -= *expression* ;
symbol *= *expression* ;
symbol /= *expression* ;
symbol <<= *expression* ;
symbol >>= *expression* ;
symbol &= *expression* ;
symbol |= *expression* ;

The first case will define *symbol* to the value of *expression*. In the other cases, *symbol* must already be defined, and the value will be adjusted accordingly.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

Linker File – Location Counter

The special linker variable *dot* '.' always contains the current output location counter. Since the . always refers to a location in an output section, it may only appear in an expression within a SECTIONS command. The . symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to . will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may not be moved backwards inside an output section, and may not be moved backwards outside of an output section if so doing creates areas with overlapping LMAs.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

Linker File – Location Counter

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x12345678;
}
```

In the previous example, the '.text' section from file1 is located at the beginning of the output section 'output'. It is followed by a 1000 byte gap. Then the '.text' section from file2 appears, also with a 1000 byte gap following before the '.text' section from file3. The notation '= 0x12345678' specifies what data to write in the gaps

Quelle: https://sourceware.org/binutils/docs/ld/index.html

DHBW
Duale Hochschule
Baden-Württemberg

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

ELF Sections & Debug - Infos

GNU Compiler Toolchain

ELF Sections & Debug Infos

Die im Linker-File definierten Sections sowie deren Inhalt werden durch den Linker i.d.R. in einem ELF Binary gespeichert.

Neben dem eigentlichen Inhalt der im Linker-File definierten Sections enthält das resultierende ELF Binary, abhängig von den angegeben Compiler- und Linker-Optionen, auch sog. Debug Informationen.

Der Debugger nutzt die Debug-Informationen im ELF Binary um ein sog. Source Level Debugging zu ermöglichen sowie Informationen über Variablen, Funktionen und Datentypen zu erhalten.

Mit Hilfe der Tools `arm-none-eabi-nm`, `arm-none-eabi-objcopy`, `arm-none-eabi-objdump` sowie `arm-none-eabi-readelf` lassen sich Informationen über das generierte ELF Binary und dessen Daten ermitteln und teilweise auch modifizieren.

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

ELF Informationen

Mit Hilfe des Linux-Kommandozeilen Tools `file` lassen sich allgemeine Informationen über ein Binary sowie dessen Typ und Inhalt ermitteln

```
$ file firmware.elf

build\firmware.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1
(SYSV), statically linked, with debug_info, not stripped
```

```
$ file obj\main.o

obj\main.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV),
with debug_info, not stripped
```

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

ELF Sections

```
$ arm-none-eabi-readelf -S build\firmware.elf
There are 23 section headers, starting at offset 0x4632c:

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 008304 00  AX  0   0  8
  [ 3] .rodata           PROGBITS        080084e0 0094e0 0000d0 00   A  0   0  8
  [ 4] .ARM              ARM_EXIDX       080085b0 0095b0 000008 00  AL  2   0  4
  [ 5] .data             PROGBITS        20000000 00a000 00000c 00  WA  0   0  4
  [ 6] .bss              NOBITS          2000000c 00a00c 000268 00  WA  0   0  4
  [ 7] .heap             PROGBITS        20000274 00a00c 000000 00   W  0   0  1
  [ 8] .stack            NOBITS          20000274 00a274 01fd8c 00  WA  0   0  1
  [ 9] .ARM.attributes   ARM_ATTRIBUTES  00000000 00a00c 00002f 00      0   0  1
  [10] .debug_line       PROGBITS        00000000 00a03b 01020b 00      0   0  1
  [11] .debug_info       PROGBITS        00000000 01a246 015319 00      0   0  1
  [12] .debug_abbrev     PROGBITS        00000000 02f55f 003371 00      0   0  1
  [13] .debug_aranges    PROGBITS        00000000 0328d0 001600 00      0   0  8
  [14] .debug_str        PROGBITS        00000000 033ed0 006a44 01  MS  0   0  1
  [15] .debug_ranges     PROGBITS        00000000 03a918 000020 00      0   0  8
  [16] .debug_rnglists   PROGBITS        00000000 03a938 0010db 00      0   0  1
.
.
.
```

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

ELF Sections

```
$ arm-none-eabi-readelf -S build\firmware.elf
There are 23 section headers, starting at offset 0x4632c:

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 008304 00  AX  0   0  8
  [ 3] .rodata           PROGBITS        080084e0 0094e0 0000d0 00   A  0   0  8
  [ 4] .ARM              ARM_EXIDX       080085b0 0095b0 000008 00  AL  2   0  4
  [ 5] .data             PROGBITS        20000000 00a000 00000c 00  WA  0   0  4
  [ 6] .bss              NOBITS          2000000c 00a00c 000268 00  WA  0   0  4
  [ 7] .heap             PROGBITS        20000274 00a00c 000000 00   W  0   0  1
  [ 8] .stack            NOBITS          20000274 00a274 01fd8c 00  WA  0   0  1
```

Die Sections aus dem Linker File existieren ebenfalls im ELF Binary. Hierbei sind auch die Informationen über die physikalische Adresse und Größe der Sections enthalten

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

ELF Debug-Sections

```
$ arm-none-eabi-readelf -S build\firmware.elf
There are 23 section headers, starting at offset 0x4632c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [10] .debug_line       PROGBITS        00000000 00a03b 01020b 00      0   0  1
  [11] .debug_info       PROGBITS        00000000 01a246 015319 00      0   0  1
  [12] .debug_abbrev     PROGBITS        00000000 02f55f 003371 00      0   0  1
  [13] .debug_aranges    PROGBITS        00000000 0328d0 001600 00      0   0  8
  [14] .debug_str        PROGBITS        00000000 033ed0 006a44 01  MS  0   0  1
  [15] .debug_ranges     PROGBITS        00000000 03a918 000020 00      0   0  8
  [16] .debug_rnglists   PROGBITS        00000000 03a938 0010db 00      0   0  1
  [17] .comment          PROGBITS        00000000 03ba13 000045 01  MS  0   0  1
  [18] .debug_frame      PROGBITS        00000000 03ba58 005f3c 00      0   0  4
  [19] .debug_line_str   PROGBITS        00000000 041994 0000df 01  MS  0   0  1
  [20] .symtab           SYMTAB          00000000 041a74 002c40 10     21 433  4
  [21] .strtab           STRTAB          00000000 0446b4 001b92 00      0   0  1
  [22] .shstrtab         STRTAB          00000000 046246 0000e6 00      0   0  1
```

Zusätzlich haben der Compiler/Assembler sowie der Linker noch Sections erzeugt, welche die sog. Debug- sowie Symbol-Informationen enthalten

Quelle: https://sourceware.org/binutils/docs/ld/index.html

GNU Compiler Toolchain

ELF Debug-Sections

```
.debug_abbrev     Abbreviations used in the .debug_info section
.debug_aranges    A mapping between memory address and compilation
.debug_frame      Call Frame Information
.debug_info       The core DWARF data containing DWARF Information Entries (DIEs)
.debug_line       Line Number Program
.debug_loc        Location descriptions
.debug_macinfo    Macro descriptions
.debug_pubnames   A lookup table for global objects and functions
.debug_pubtypes   A lookup table for global types
.debug_ranges     Address ranges referenced by DIEs
.debug_str        String table used by .debug_info
.debug_types      Type descriptions
```

Quelle: https://sourceware.org/binutils/docs/ld/index.html

Compiler Toolchain

Präprozessor, Compiler, Assembler und Linker

Linker MAP File

GNU Compiler Toolchain

Linker MAP File

Der Linker kann, bei angabe entsprechende Optionen, ein sog. MAP File erzeugen. Dieses MAP File ist eine Textdatei mit lesbaren Informationen über das vom Linker erzeugte Binary sowie dessen Inhalt.

Das MAP File beinhaltet u.a.

- Zugrundeliegende Speicher-Konfiguration
- Adresse für Variablen und Funktionen
- Größe von Variablen/Funktionen
- Informationen über entfernte oder ausgelassene Eingabedaten/Sections
- Herkunft von Variablen und Funktionen (z.B. Dateiname)

Das Format des MAP Files ist, anders als z.B. das ELF Format, nicht standardisiert und jeder Tool-Hersteller verwendet hierzu einen anderen Aufbau und teilweise auch andere Informationen.

Quelle: intern

GNU Compiler Toolchain

Linker MAP File

```
Memory Configuration

Name             Origin             Length             Attributes
RAM              0x20000000         0x00020000         xrw
FLASH            0x08000000         0x00080000         xr
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD obj/libstm32.a
LOAD obj/startup.o
LOAD obj/System.o
.
.
.
```

Quelle: intern

GNU Compiler Toolchain

Linker MAP File

```
                0x00000200                        _size_of_stack = 0x200
                0x00000000                        _size_of_heap = 0x0
                0x2001fffc                        _end_of_ram = ((ORIGIN (RAM) + LENGTH
(RAM)) - 0x4)

.isr_vector     0x08000000      0x1d8
                0x08000000                        . = ALIGN (0x4)
 *(.isr_vector)
 .isr_vector    0x08000000      0x1d8 obj/startup.o
                0x08000000                g_pfnVectors
                0x080001d8                        . = ALIGN (0x4)
.
.
.
*(.text*)
 .text.Reset_Handler
                0x08000b48       0x4c obj/startup.o
                0x08000b48                Reset_Handler
 .text.Default_Handler
                0x08000b94        0x2 obj/startup.o
                0x08000b94                RTC_Alarm_IRQHandler
.
.
.
```

Quelle: intern