

Real-Time and Embedded Systems @ SIT

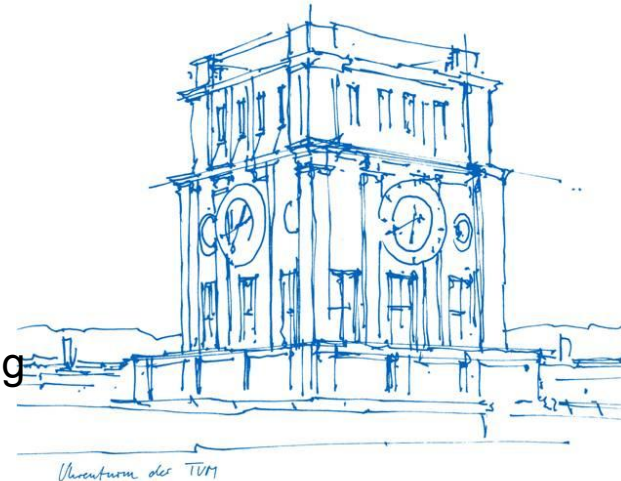
Running software

Alexander Hoffman

Technical University of Munich

Department of Electrical and Computer Engineering

Chair of Real-Time Computer Systems



What happens when I run my program?

- Firstly we need to understand (roughly) how an executable file is structured
- For all examples where required I am talking about a Linux machine compiled with GCC
 - Thus, our executable file is an ELF file (Executable and Linkable Format)
 - Standardized binary file for Unix and Unix-like systems on x86 processors
 - Linux, Solaris, *BSD, ...
 - Output from a compiler after linking
- File extension is not always .elf
 - .bin, .so, .ko, ...

ELF File Structure

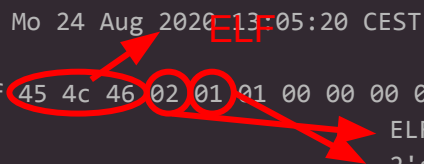
- ELF file contains two sections:
 - ELF header
 - 32 bytes
 - Identifies the file format
 - Specifies file bit format (32 bit or 64 bit)
 - Specifies endianness
 - OS target
 - File data
 - Soon

ELF File Header - Example

```
readelf bar -h
      30.4s  Mo 24 Aug 2020 13:05:20 CEST
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                               UNIX - System V
  ABI Version:                         0
  Type:                                 DYN (Shared object file)
  Machine:                             Advanced Micro Devices X86-64
  Version:                             0x1
  Entry point address:                 0x1040
  Start of program headers:            64 (bytes into file)
  Start of section headers:           15512 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           11
  Size of section headers:             64 (bytes)
  Number of section headers:           34
  Section header string table index:   33
```

ELF File Header - Example

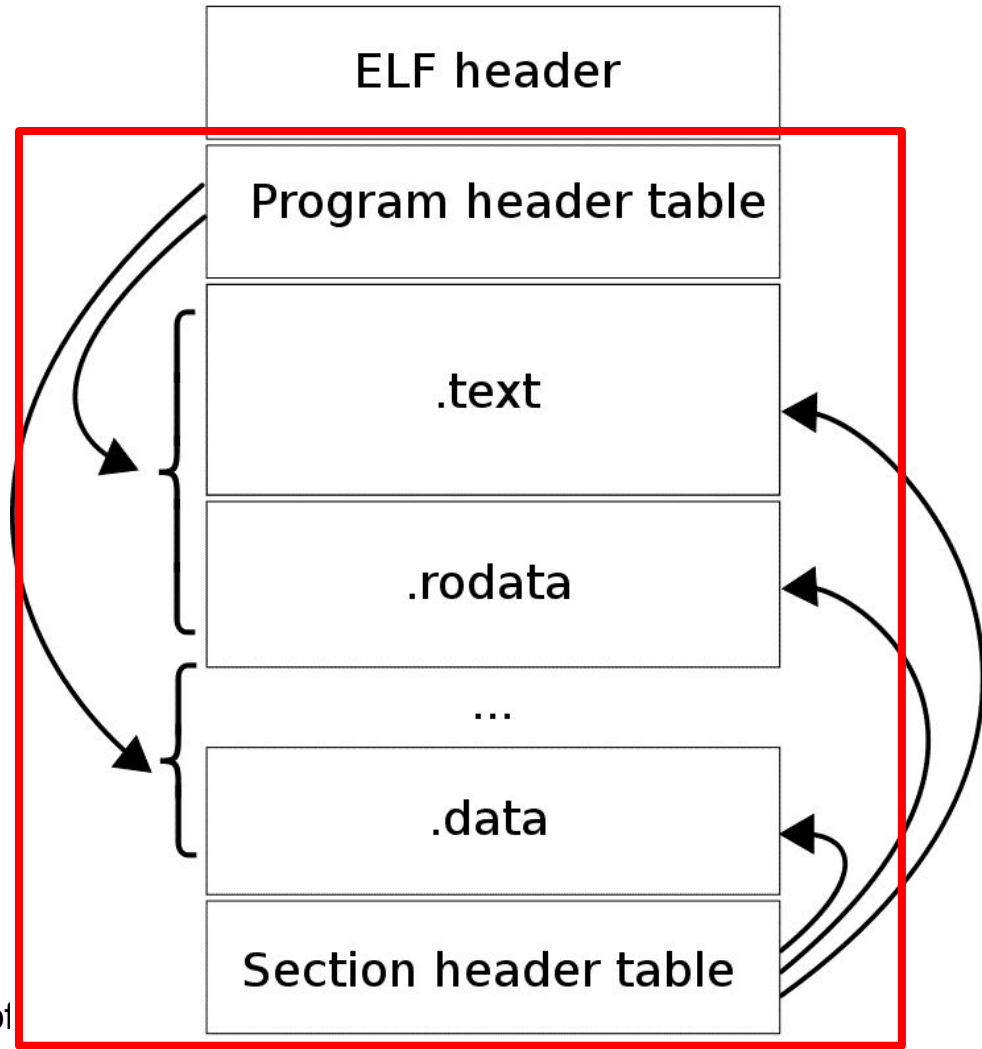
```
readelf bar -h
30.4s Mo 24 Aug 2020 13:05:20 CEST
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                              UNIX - System V
  ABI Version:                         0
  Type:                                DYN (Shared object file)
  Machine:                             Advanced Micro Devices X86-64
  Version:                             0x1
  Entry point address:                 0x1040
  Start of program headers:            64 (bytes into file)
  Start of section headers:           15512 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           11
  Size of section headers:            64 (bytes)
  Number of section headers:           34
  Section header string table index:   33
```



File Data

Can include:

- Program header table
 - Describes memory segments
- Section header table
 - Describes sections
- Program data
 - Referred to from header table



Memory Segments

- Described by program header table
- Used to define different classes of memory
 - Eg. code and data segments
- Memory locations are identified by a segment and an offset
- Used at run-time to find and execute code and data

Memory Sections

- Defined by the section header table
- Contain all information in an object file
- Each section is described by a section header
- Data can only exist in one section (no overlapping)
- Special sections:
 - `.bss`
 - `.data`
 - `.rodata`
 - `.text`
 - `.interp`

Special Sections (System Reserved)

- `.bss` : Holds uninitialized data, generally initialized to zero
 - Eg. `int foo;`
- `.data` : holds initialized data
 - Eg. `int bar = 2;`
- `.rodata` : holds read-only data and is put onto read-only memory when possible
- `.text` : holds executable instructions of a program (code)
- `.interp` : Holds the name of the program's interpreter which loads loadable segments, eg. shared libraries. On a Linux system this is [ld.so](#)

Loading File

- We now know that an executable is made up of many sections containing our variables and code
- Various forms of headers tell our system what we have, how large it is and where it can be found
- OS Kernel executes the file in two steps:
 - Opening a file (loading “stuff” into memory)
 - Executing the file by transferring control to program’s code

Opening File

A C file will be opened by generally doing the following:

- `.text` section (code) is loaded as read-only into memory
- `.data` section is loaded into memory
- `.bss` is loaded and usually initialized to zero
- Shared library calls are resolved which are detailed in the `.interp` section
- Control now passed to program as it is loaded

Executing Loaded Program

- The symbol (remember from compilation) `_start` is invoked
- In turn invokes `__libc_start_main` in glibc (compiler) passing in appropriate arguments, ie. `argc`, `argv` etc.
- `_init` is then called which can do a few things, such as initialize global variables with values from `.bss`
- `main` is then called
- If `main` exits then `_fini` gets called to handle program destruction
- Program exits, ie. surrenders control

[here](#)

Now on an embedded system?

- Biggest difference is we don't have an OS to load and execute our program's code
- Relies more heavily on predefined laying out of program's code and its location in memory so that we know what needs to be executed, when and where to find it in memory where our binary was loaded into (flashing)
- What goes where is usually hardware dependent and handled for us by the compiler and flashing software

Microcontroller Boot Process

- A lot simpler than you'd expect
- Two clear steps:
 - Apply voltage to the system and point microcontroller to the *reset vector* in memory
 - Execution of the *reset vector* is responsible for the startup on the CPU and system, Eg. peripherals
- System specific assembly code telling the CPU EXACTLY what to do
- Placed into memory/structured as specified by the linker file
 - Linker file is responsible for structuring binary appropriately
- The average embedded developer will never write this, provided by silicon toolchains
- I've never done more than read these to get an understanding of them

So Where Does the Real “Booting” Happen?

The reset vector is where the more complex boot process is programmed.

It “generally” does the following:

- Copy vector tables (interrupts) from flash to RAM
- Copy initialized data (Eg. `int foo = 2`) from the `.data` section into RAM
- Copy uninitialized/zero initialized data from the `.bss` section into RAM
- Copy RAM functions from flash to RAM
- Set up processor's stack pointer
- Set up the heap
- Set up any system hardware, Eg. peripherals
- Jump to main

Startup Code Example

```
/* Copy the data segment initializers from
flash to SRAM */
movs r1, #0
b LoopCopyDataInit

CopyDataInit:
ldr r3, =_sdata
ldr r3, [r3, r1]
str r3, [r0, r1]
adds r1, r1, #4

LoopCopyDataInit:
ldr r0, =_sdata
ldr r3, =_edata
adds r2, r0, r1
cmp r2, r3
bcc CopyDataInit
ldr r2, =_sbss
b LoopFillZerobss

/* Zero fill the bss segment. */
FillZerobss:
movs r3, #0
str r3, [r2], #4

LoopFillZerobss:
ldr r3, =_ebss
cmp r2, r3
bcc FillZerobss

/* Call the clock system initialization
function.*/
bl SystemInit
/* Call the application's entry point.*/
bl main
bx lr
.size Reset_Handler, .-Reset_Handler
```

Startup assembly code used on a STM32 by my students @ TUM can be seen [here](#)

This is just a bit of the reset vector, it does:

- Sets up stack pointer
- Sets up ISR vector table with appropriate addresses
- Points CPU to “Reset_Handler”
 - Copies all required data from flash to RAM
 - Configures the CPU’s clock system and external SRAM
 - Branches to main

Startup Code Example

```
/* Copy the data segment initializers from
flash to SRAM */
movs r1, #0
b LoopCopyDataInit

CopyDataInit:
ldr r3, =_sdata
ldr r3, [r3, r1]
str r3, [r0, r1]
adds r1, r1, #4

LoopCopyDataInit:
ldr r0, =_sdata
ldr r3, =_edata
adds r2, r0, r1
cmp r2, r3
bcc CopyDataInit
ldr r2, =_sbss
b LoopFillZerobss

/* Zero fill the bss segment. */
FillZerobss:
movs r3, #0
str r3, [r2], #4

LoopFillZerobss:
ldr r3, =_ebss
cmp r2, r3
bcc FillZerobss

/* Call the clock system initialization
function.*/
bl SystemInit
/* Call the application's entry point.*/
bl main
bx lr
.Sym _Reset_Handler, .-Reset_Handler
```

Startup assembly code used on a STM32 by my students @ TUM can be seen [here](#)

This is just a bit of the reset vector, it does:

- Sets up stack pointer
- Sets up ISR vector table with appropriate addresses
- Points CPU to “Reset_Handler”
 - Copies all required data from flash to RAM
 - Configures the CPU’s clock system and external SRAM
 - Branches to main