Linux For Embedded Systems

For Arabs

Cairo University
Computer Eng. Dept.
CMP445-Embedded Systems

Ahmed ElArabawy

Lecture 14:
Introduction to the Toolchain
Part 2: Binary Utilities

Binary Files

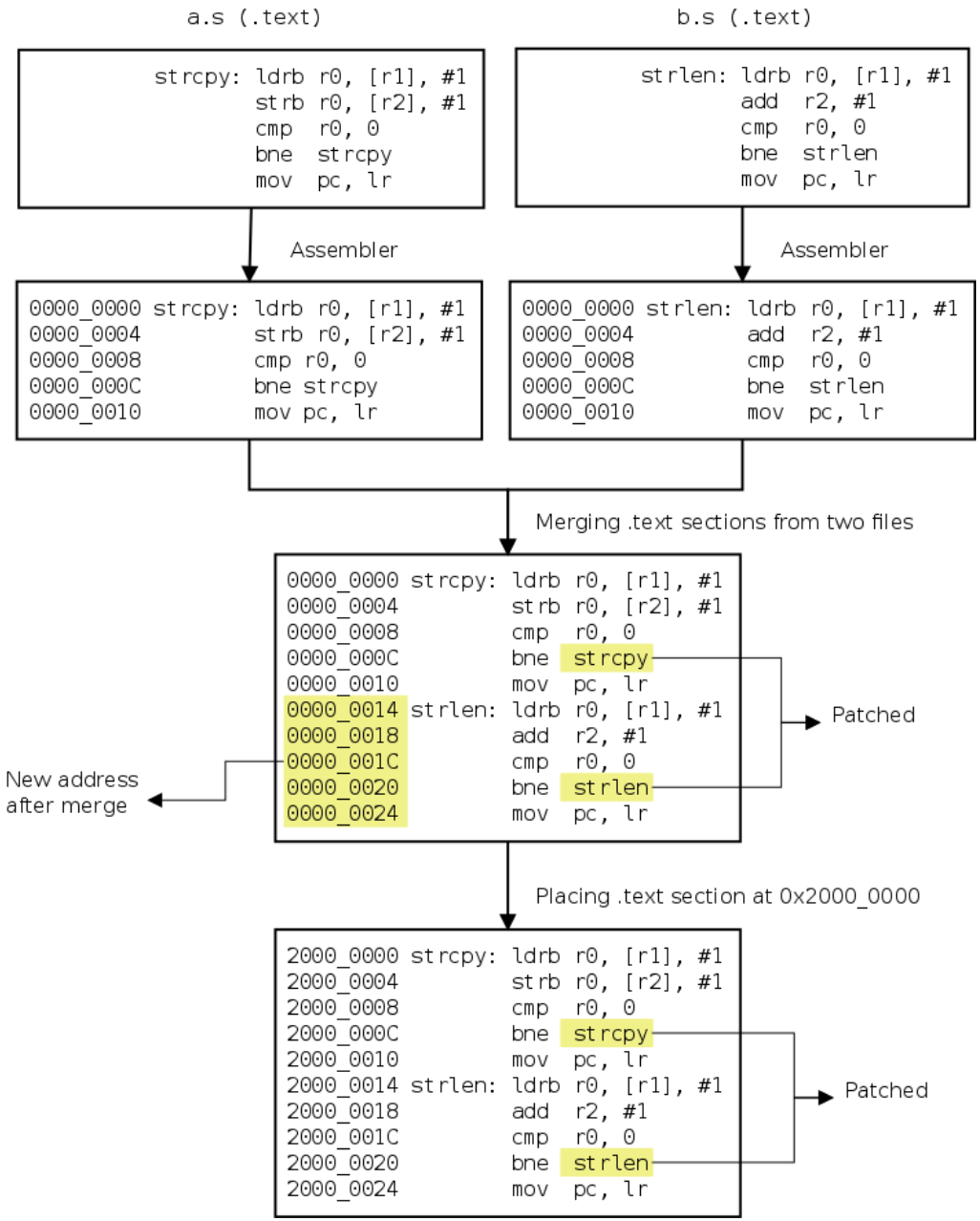
- What is a Binary File?
  - A Binary File is the **machine language instructions** that should be executed on the target
- This is **not** an accurate or complete answer….
  - Executable files are just one type of binary files. Not all binary files can execute on the target
  - Machine language instructions are just one one component of the binary file. Other components exist
- **So what would be a more accurate answer ??**

Binary File Types

Binary File Types

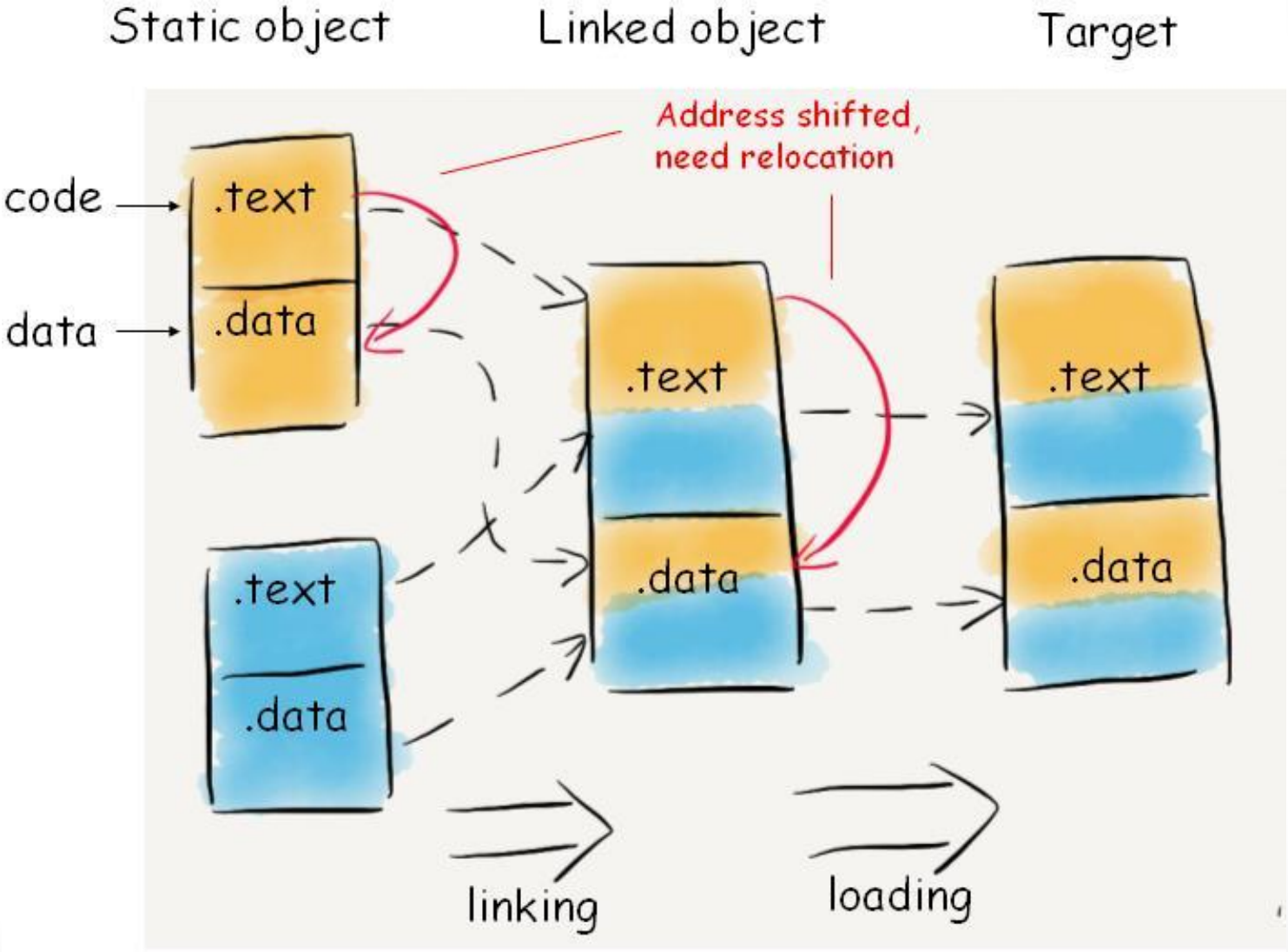
1. Relocatable Object Files (*.o)

- A relocatable object file is the outcome of the compilation/assembly process of one source code file
- This file can not execute since it has some unresolved symbols, and needs to be linked with other object files or libraries
- Even if the file does not use any unresolved symbols, it is still not ready to execute on the target
- Why is it called **Relocatable** ?
  - The different components of the object file is set to start at an address independent from other object files (for example they can start at the address ***0x00000000)***
  - Addresses of all symbols inside the object files are offsets from the start address
  - This applies to both machine language instructions and other components of the object file
  - It is the job of the **linker** to relocate these components into other addresses when combining multiple object files (so they don't overlap in memory)
  - The relocation process performed by the linker includes,
    - Merging of code and data sections from the object files
    - Moving the start address of the resulting sections into an address suitable to the target or the OS


a.s (.text)
b.s (.text)
strcpy: ldrb r0, [r1], #1
strb r0, [r2], #1
cmp r0, 0
bne strcpy
mov pc, lr
strlen: ldrb r0, [r1], #1
add r2, #1
cmp r0, 0
bne strlen
mov pc, lr
Assembler
Assembler
0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0000 strlen: ldrb r0, [r1], #1
0000_0004 add r2, #1
0000_0008 cmp r0, 0
0000_000C bne strlen
0000_0010 mov pc, lr
Merging .text sections from two files
0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0014 strlen: ldrb r0, [r1], #1
0000_0018 add r2, #1
0000_001C cmp r0, 0
0000_0020 bne strlen
0000_0024 mov pc, lr
Patched
New address after merge
Placing .text section at 0x2000_0000
2000_0000 strcpy: ldrb r0, [r1], #1
2000_0004 strb r0, [r2], #1
2000_0008 cmp r0, 0
2000_000C bne strcpy
2000_0010 mov pc, lr
2000_0014 strlen: ldrb r0, [r1], #1
2000_0018 add r2, #1
2000_001C cmp r0, 0
2000_0020 bne strlen
2000_0024 mov pc, lr
Patched

Binary File Types
2. Executable Binary Files

- This is the outcome of the linking process of multiple object files as well as (static/dynamic) libraries
- This file has all of its symbols either resolved, or pointing to some shared objects (to be resolved at load/run time)
- The code and data sections of this file is the outcome of merging code/data sections of the object (*.o) and archive (*.a) files used to generate it
- This file should be placed on the target storage, and is ready to execute on target


Static object
Linked object
Target
Address shifted,
need relocation
code
data
.text
.data
.text
.data
.text
.data
.text
.data
linking
loading

Binary File Types
3. Shared Object Files (*.so)

- This is the outcome of the linking process of multiple object files to form a dynamic library file
- Since it is the outcome of a linking process, then the sections of object files are merged and relocated (same as an executable)
- One shared object file may rely on some symbols in another shared object files
- This file is ready to load on the target by the Dynamic Linker (***ld-linux.so***) whenever it is needed by an executable file
- Shared object files are not executable on their own (they don't have an entry function), they are only called by executable files

Binary File Types
4. Core Dump Files

- A core dump file (also called core file) is a binary file that is automatically generated by the Linux kernel when the executable faces a fatal problem that causes it to exit abruptly (crash)
- For example, it is generated,
  - When the executable has a segmentation fault (illegal memory access)
  - When the executable executes the abort() function
  - Upon other faults such as floating point faults (eg. Divide by Zero)
- It contains records of the state of the program (memory, registers, variables, stack trace, ….) at the point of the crash
- It is useful for debugging purposes to analyze the system crashes in an offline way. Core file can be copied from the target to the host machine, and debugged on the host machine

Controlling Core File Generation

- In some cases, core files are very useful for debugging faults that can not be debugged on the target
- In other cases, core file generation is not desired, to avoid filling the storage media with them
- The following can be done to control the generation of core files
  - To enable core file generation

    $ ulimit -c unlimited
  - To disable core file generation

    $ ulimit -c 0
  - To check on the current setting

    $ ulimit -c
- The core file location is set at ***/proc/sys/kernel/core_pattern***
- To set the location of the core file generation (make sure the program has write access on the folder)

  $ echo '/home/user/cores/core_%e.%p' | sudo tee /proc/sys/kernel/core_pattern

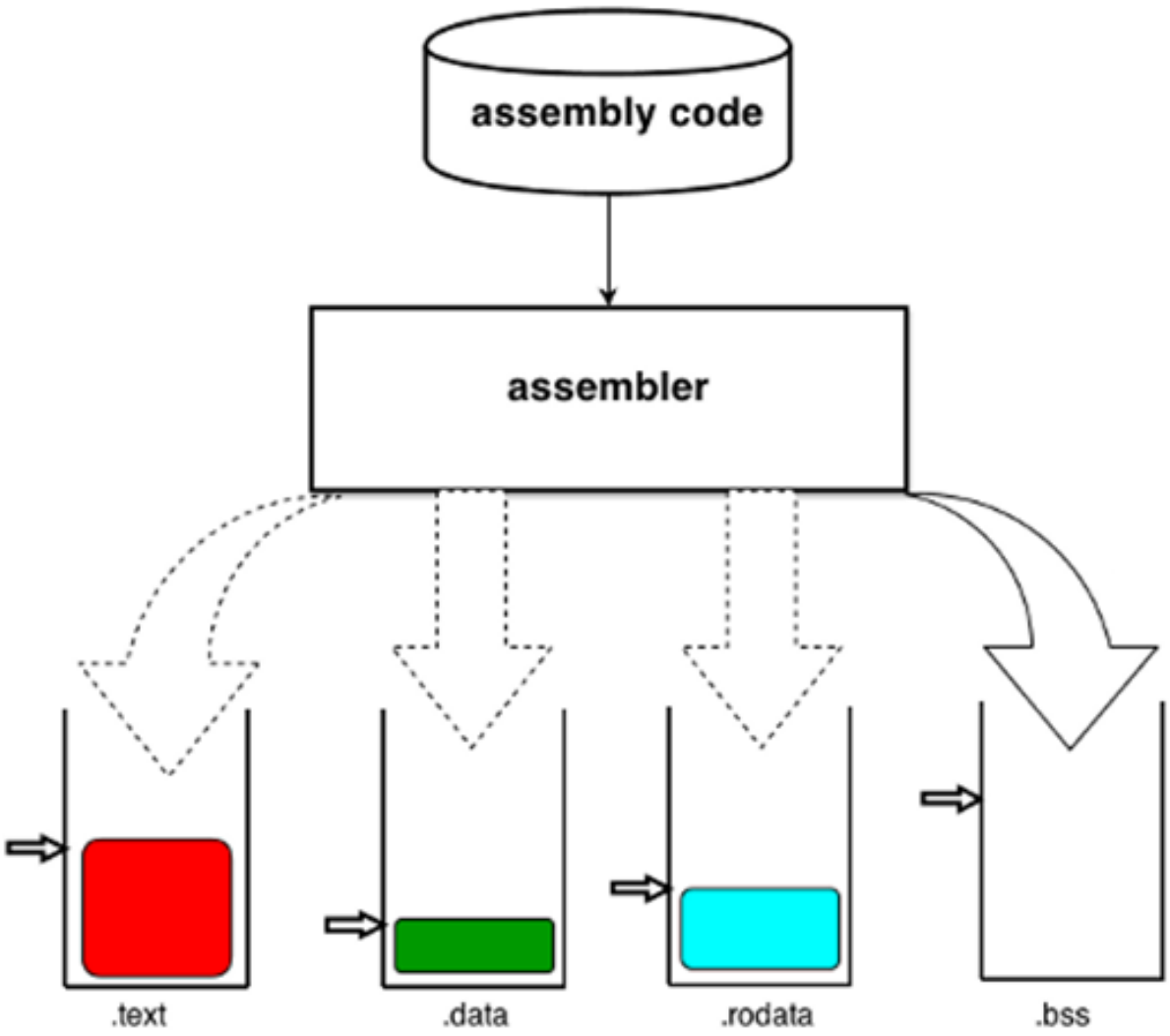
Binary File Components

Binary File Components

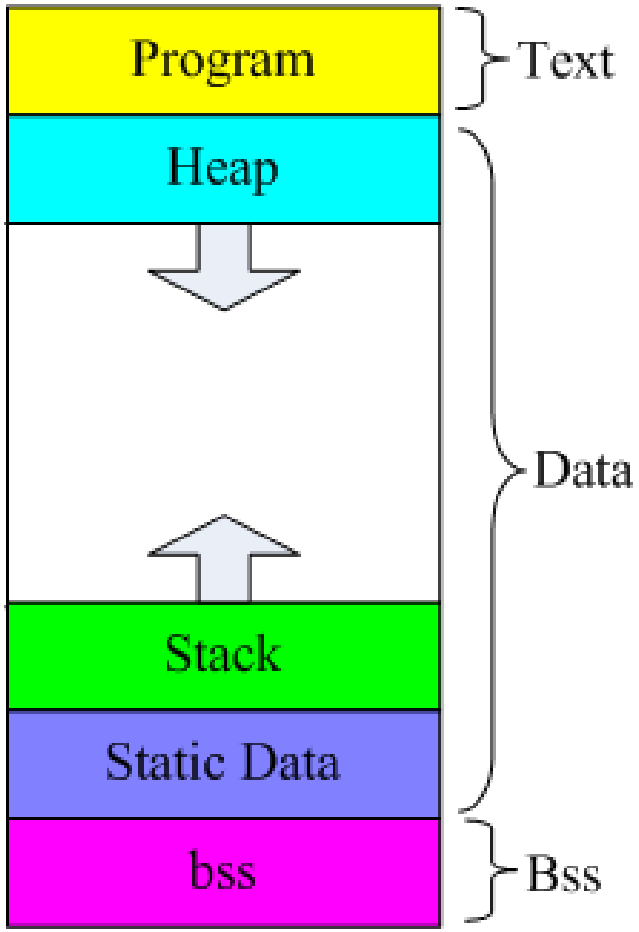
- Binary file Contains a lot of blocks of information:
  - The code section (also called the **.text** section):
    - This is the machine language instructions that the assembler generates during the compilation process
    - Each object file (*.o) will contain its .**text** section
    - The linker merges the .**text** sections of object files
    - The .**text** section has references to **symbols** (such as data that it uses, or functions that it needs to jump to)
  - A group of Data Sections
    - The **.data** section that contains global/static data that are initialized in the code (to other values than zero)
    - The **.bss** section that contains global/static data that are not initialized or initialized to zero
    - The **.rodata** section that contains Read-only Data in the code (such as strings in ***printf*** statements)
  - A symbol table section that contains a table of variables and functions defined or used in the code
  - A group of debug info sections for use by debuggers
  - Other sections

More on Data Sections

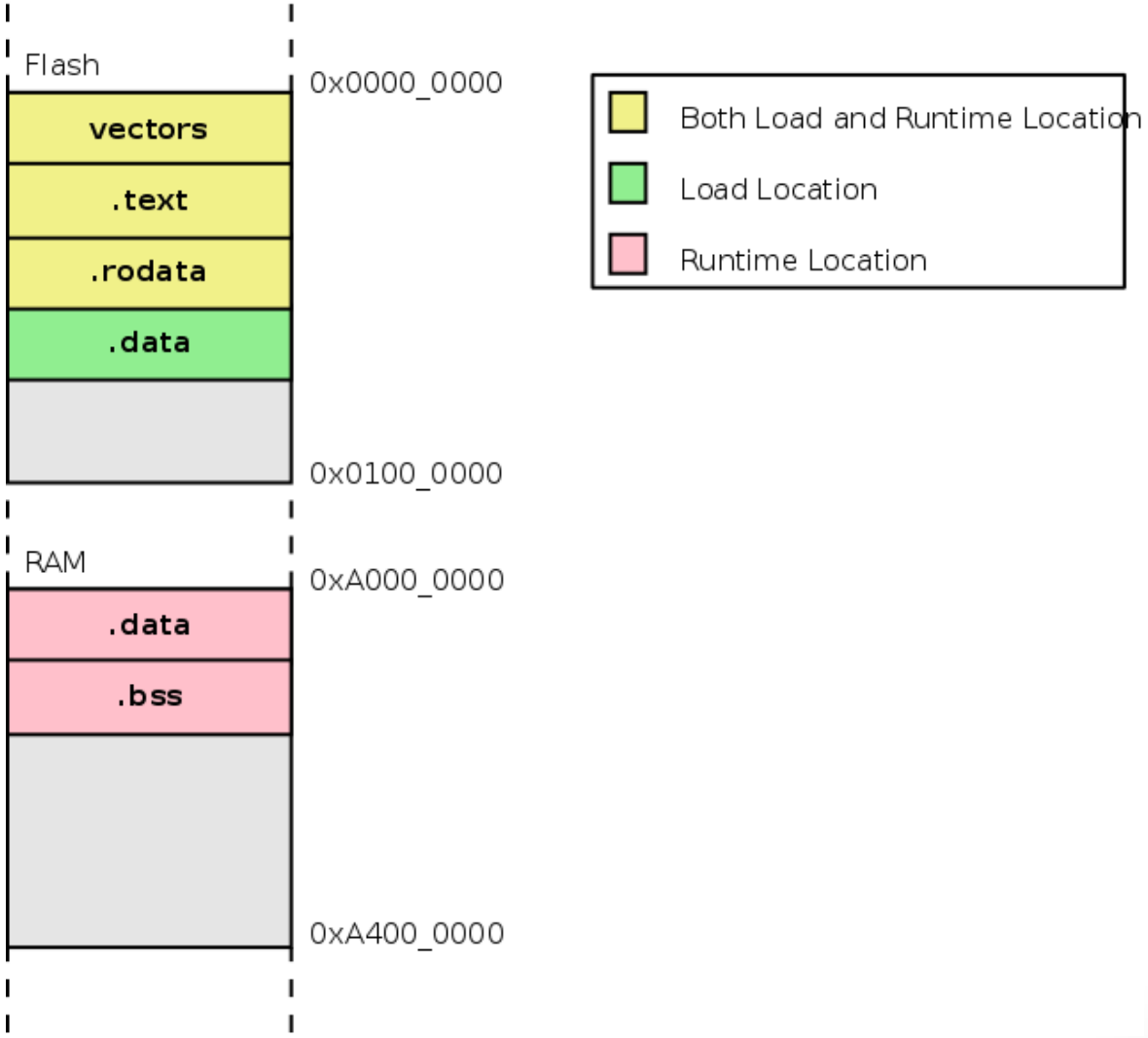
- What is the difference between **.bss** and **.data** sections?
  - When the program is loaded in memory, the program global and static data is also loaded from the storage to the RAM to start execution
  - If the data was initialized in the program, then the object file needs to store the initial value in the **.data** section
  - If the data is not initialized, then it is initialized to zero at load time
  - Since the data is initialized at load time, then there is no need to carry the data value of zero in the object file (what is the value of having a part of the object file which is all filled with zeros)
  - Accordingly, the **.bss** section is normally empty, and used only as a place holder. It has a start address and a length to reserve the space in memory at load time, but no contents in the object file (stored in Flash)
  - Note that the data initialized to zero in the program, is also considered with the uninitialized data (since all will be initialized to zero by the loader)
- What about local data ? Where are they located ?
  - Local data are created at run time inside the stack
  - The stack is not part of the object file stored on the Flash, it is created directly in the target RAM at program load in Memory


assembly code
assembler
.text
.data
.rodata
.bss

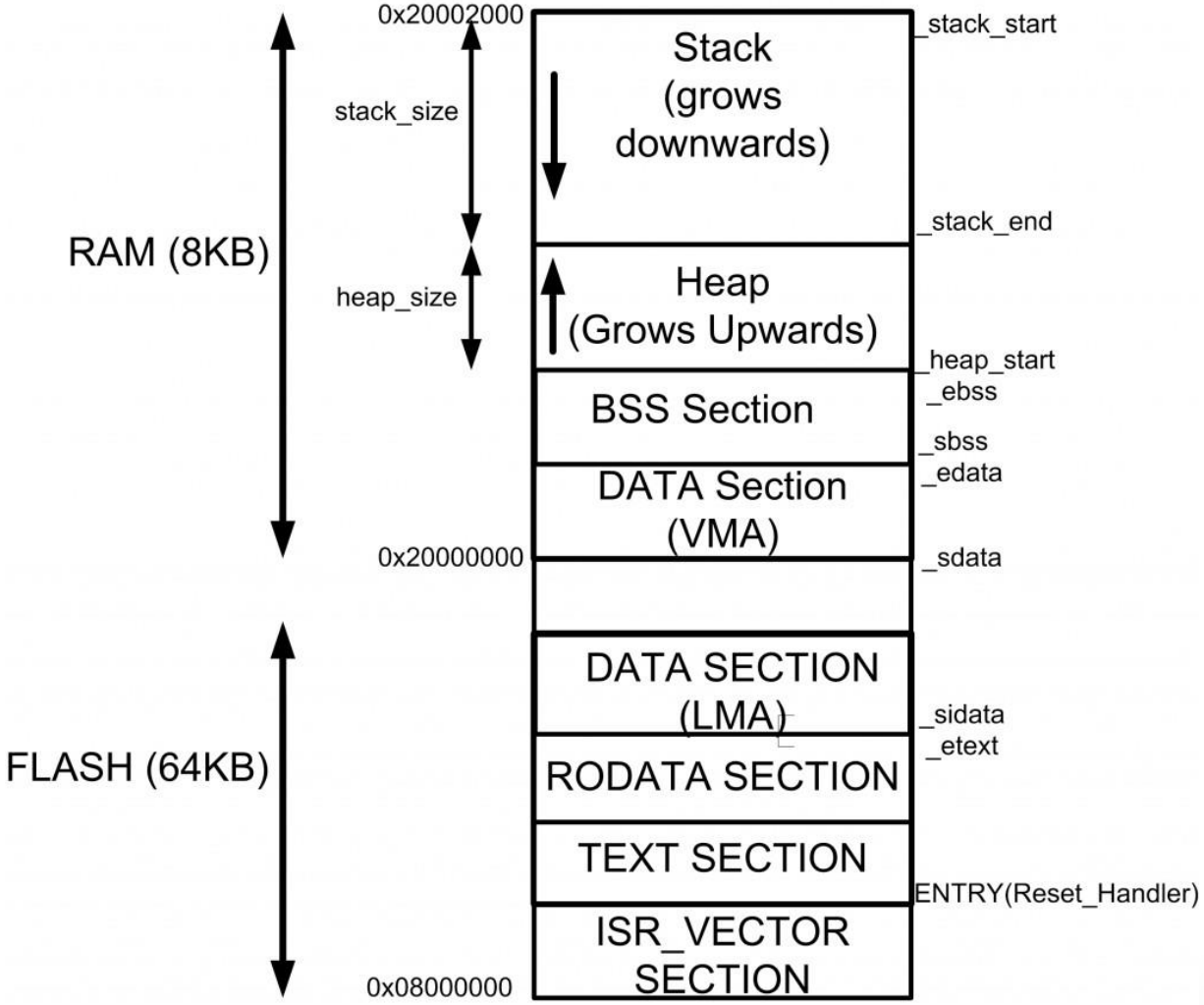
Memory Map Example:
(For Simple Embedded Systems)



Memory Map Example:
(For Simple Embedded Systems)



Memory Map Example:
(For Simple Embedded Systems)



Conclusion:

- Binary files are of different types and perform different roles
- Binary files are not just a bunch of machine language instructions ready for execution
- There are a lot of information inside a binary file (code, data, debug info, tables, ….)
- Due to all of that, we need to have some clean, extendable, and flexible way to carry all of these info in one file and facilitate the use of it
- Different OSs use different file formats to carry this information:
  - Unix introduced the ***COFF*** (Common Object File Format) file format
  - Windows uses the ***PE*** (Portable Executable) file format
  - Linux uses the ***ELF*** (Executable and Linkable Format) file format

ELF File Format

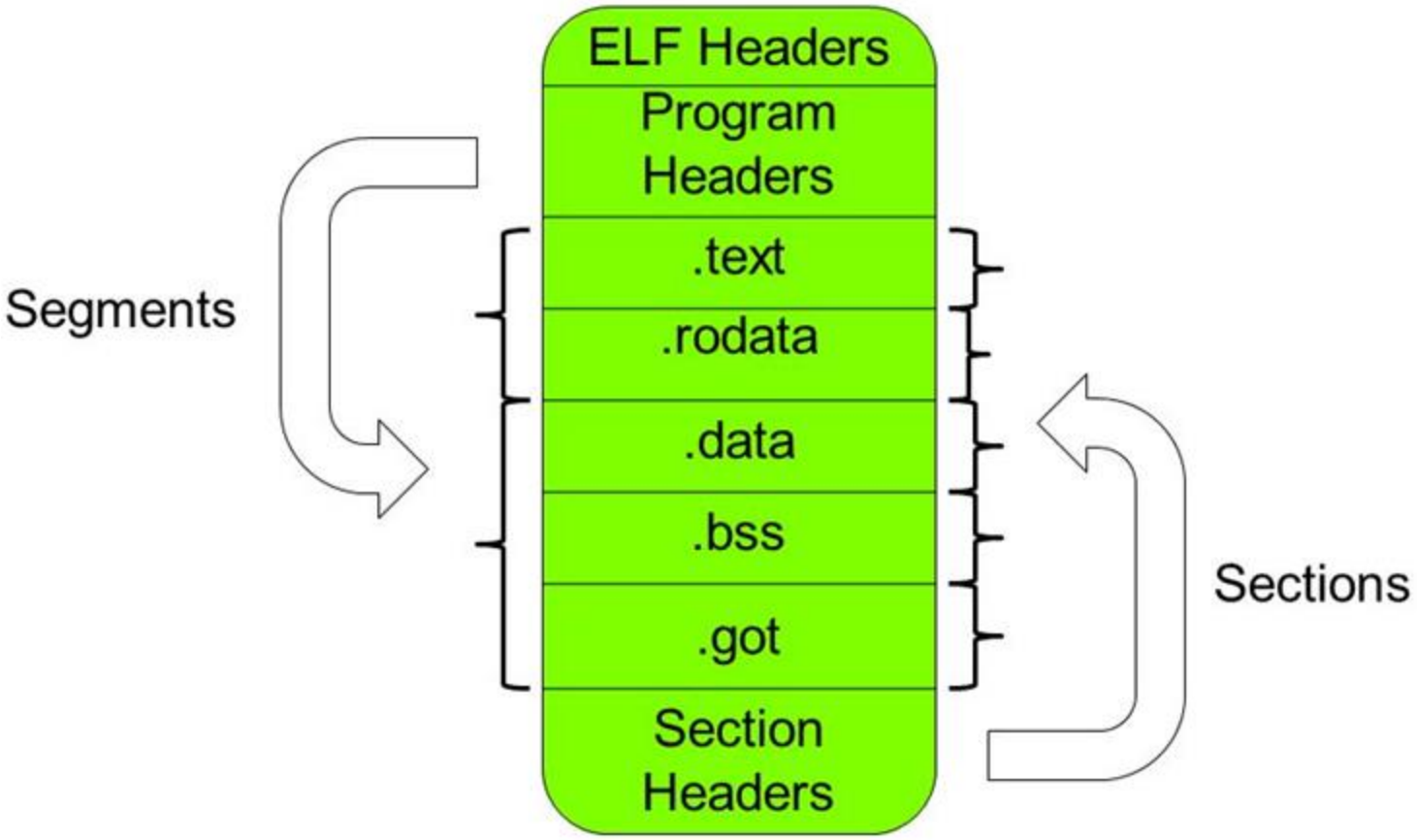
What is an ELF File

- ELF stands for “Executable and Linkable Format”
- ELF is a file format used for,
  - Relocatable Object files
  - Executable files
  - Shared Library files
  - Core dump files
- ELF files are extensible and have a lot of optional fields
- They are not specific to any processor architecture
- Used in Unix and Linux OS (and may be adopted by other OSs as well)
- Generated by GNU GCC
- Used by the GNU toolchain

ELF File Layout

- An ELF file contains the following,
  - An ***ELF Header*** at the beginning of the file. This header describes high level attributes of the file and target processor such as,
    - Type of file (object, executable, core, shared object)
    - Used Processor (ARM, x86, x86-64, …)
    - Little Endian/Big Endian format
    - 32/64 bit format
    - It has Pointers to the other parts of the file
  - ***Program Header Table*** that points to zero or more **segments**
  - ***Section Header Table*** that describes zero or more **sections**
  - A group of **segments** and **sections** pointed by the two headers

ELF File Layout



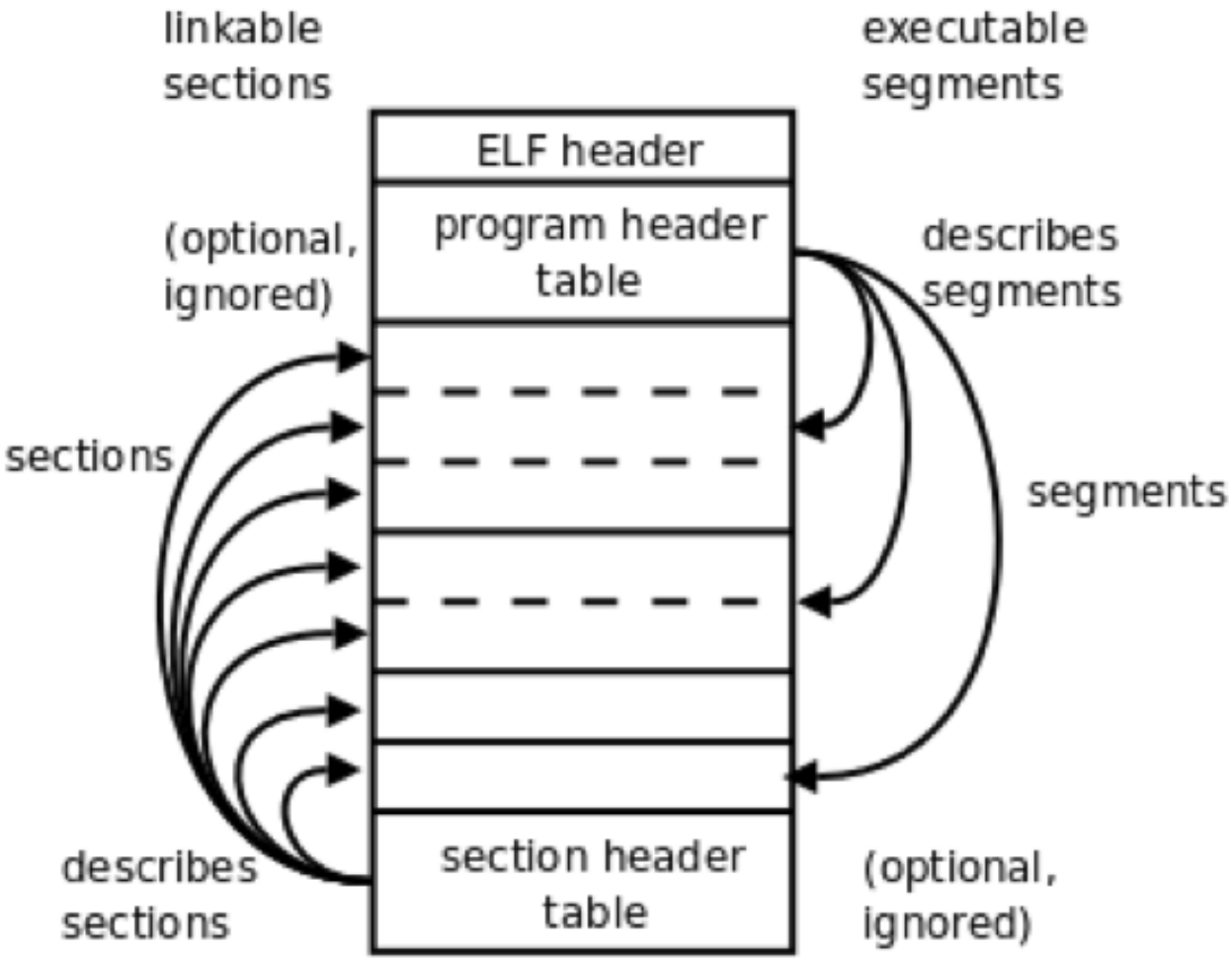
What is a Segment

- A ***segment*** is part of the executable image for the program
- It is more relevant to executable file (not a object file)
- It is necessary for runtime execution of the file
- There are 3 major types of segments:
  - **Text Segment**
    - Contains binary code for the executable (instructions of the program)
  - **Data Segment**
    - Contains the program variables that are initialized in the program with non zero values
  - **BSS Segment**
    - Contains the program variables that are not initialized in the program (or initialized to zero)

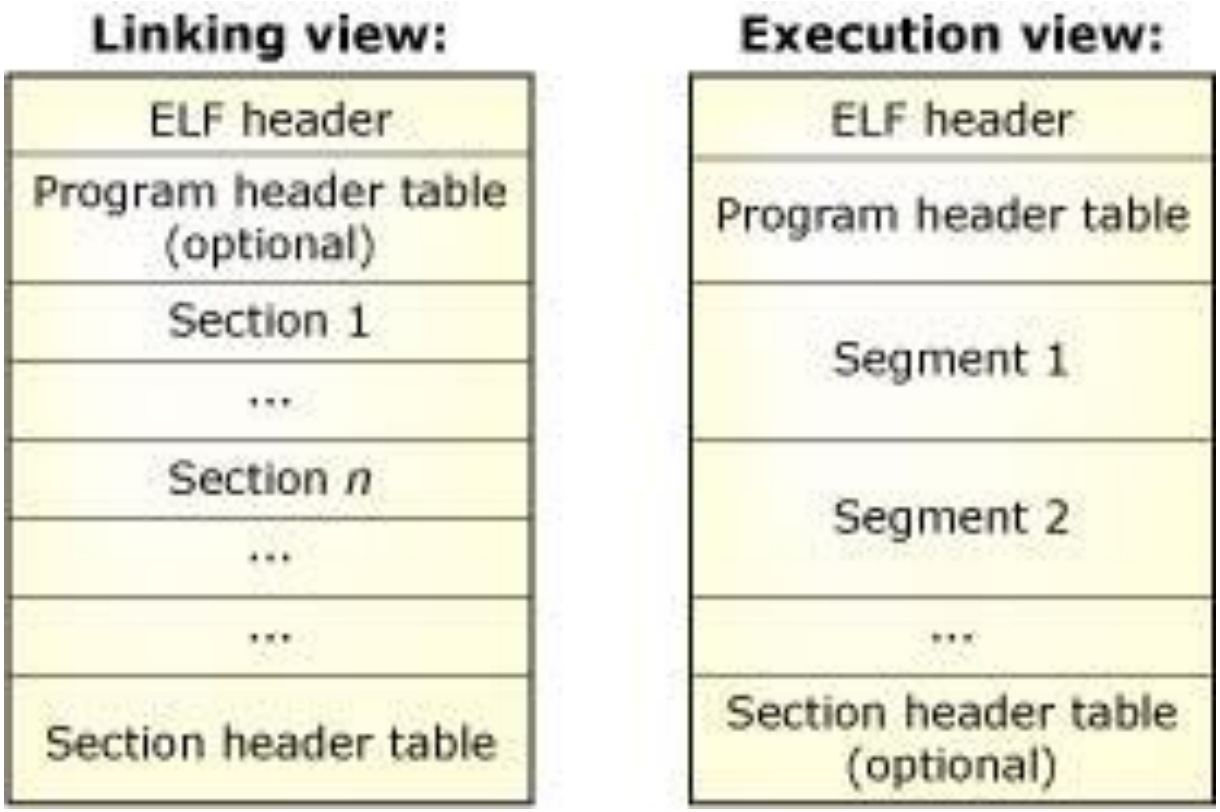
What is a Section

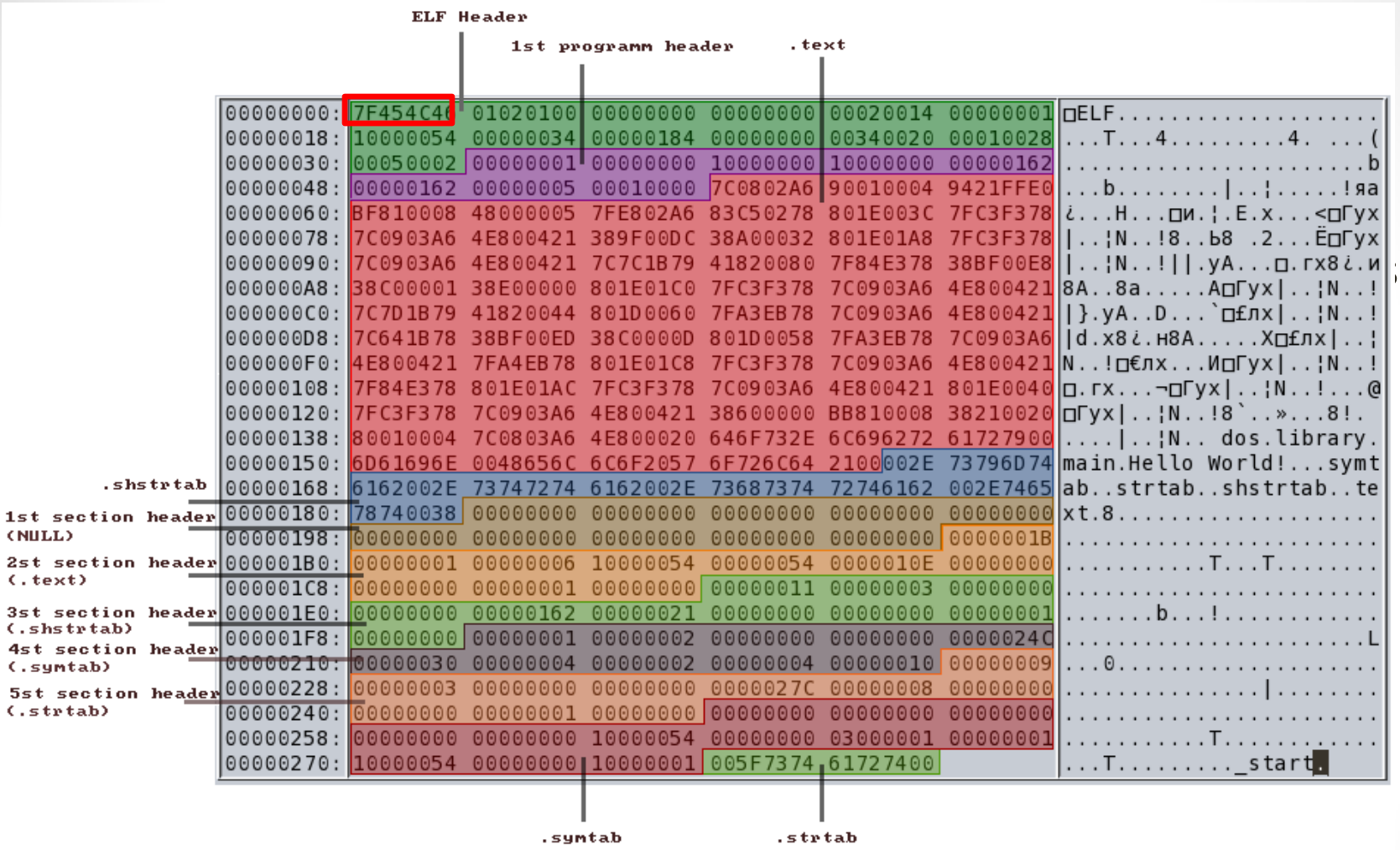
- ***Sections*** are used for linking and debugging purposes
- In relocatable object files, sections contain code and data such as in **.text** , **.data** and **.bss** sections
- During the linking process, the linker maps the TEXT, DATA, and BSS segments from these sections
- Other sections are not required for the execution of the program and are used for debugging purposes
- Removing those sections will not affect running of the executable, but may reduce debugging capabilities
- Main Sections are:
  - Symbol Table Section (**.symtab**) : Used for low level debugging info and for linking purposes
  - Strings Table Section (**.strtab**) : Carries all strings used by other sections in the executable
  - Section Name String Table Section (**.shstrtab**) : Carries the names of the sections in the table
  - Debug Info Sections (multiple sections) : available when the executable contains source level debug info
- Some tools can be used to strip an ELF file from its sections (such as ***strip*** command)

ELF File Layout

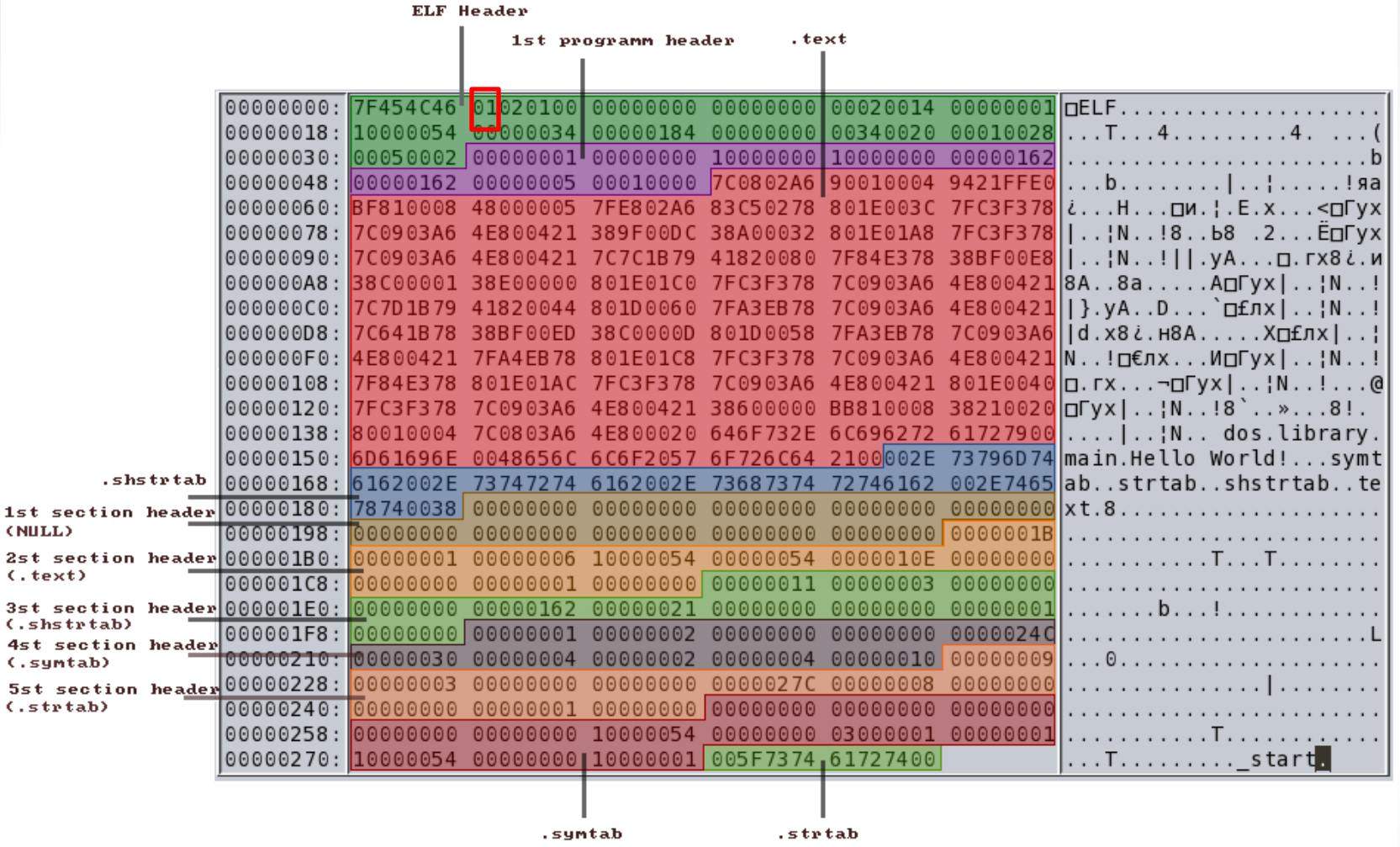


ELF File Layout

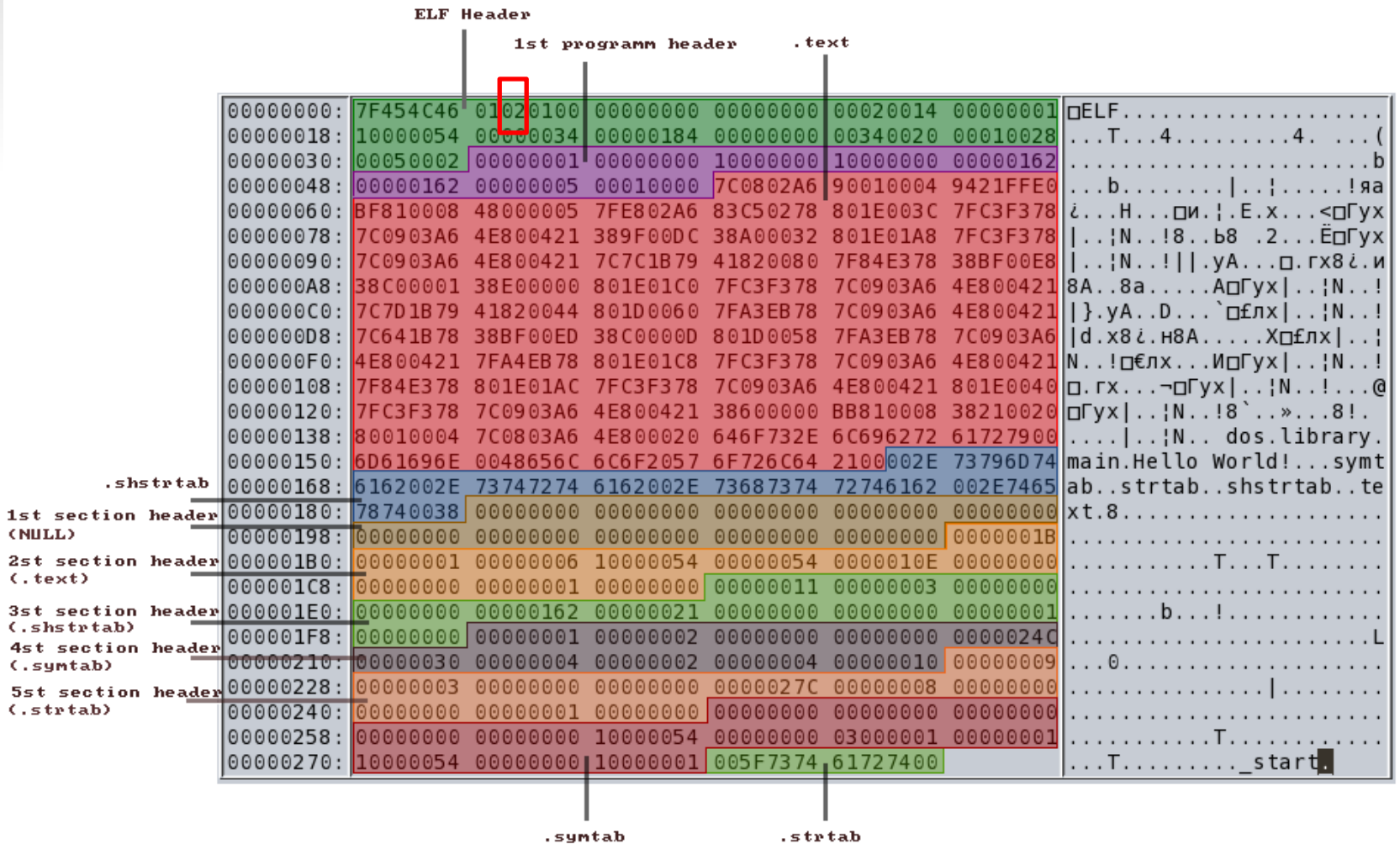




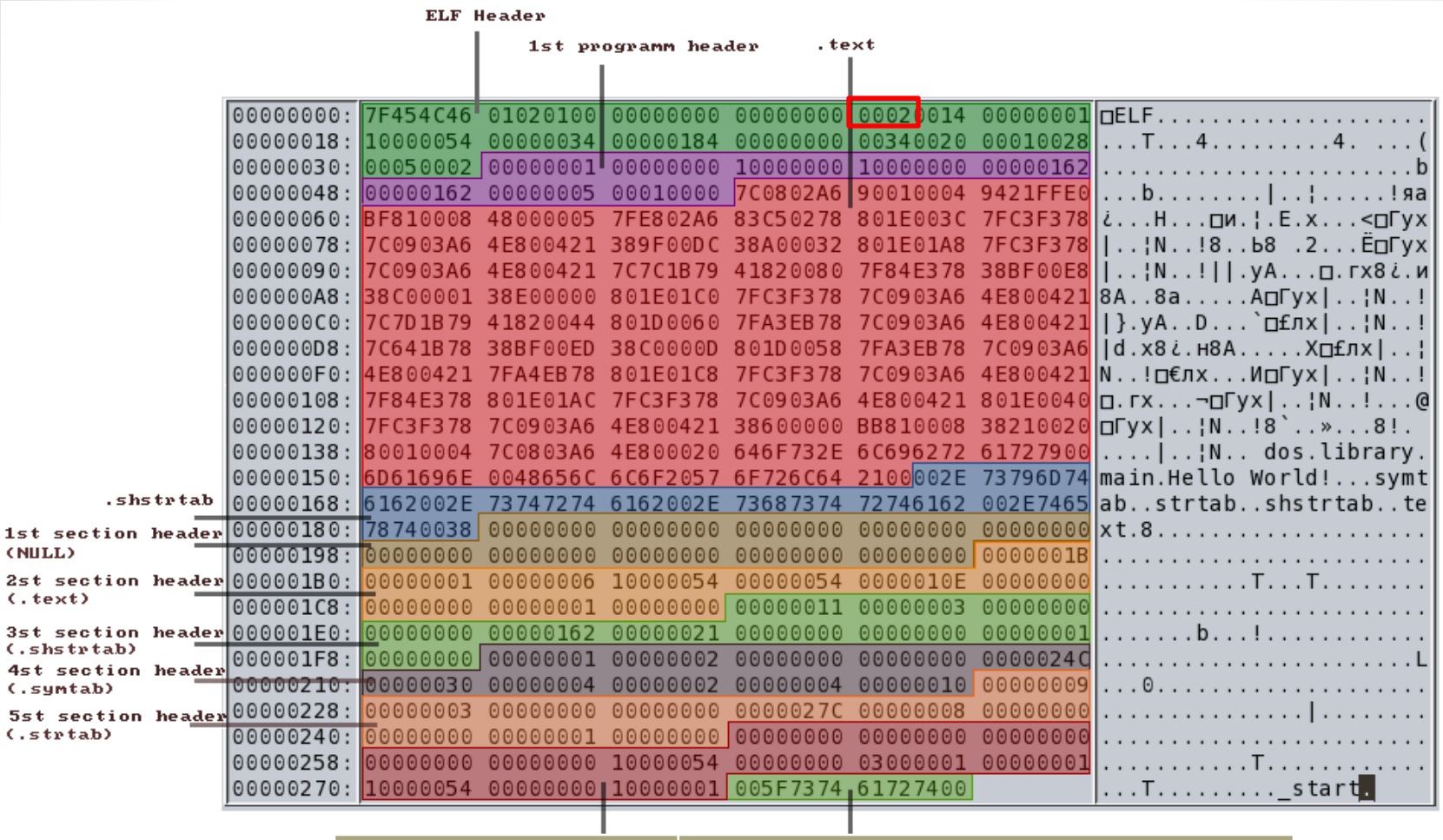
- The first 4 bytes represent a magic number to identify that it is an ELF file
- The magic number is : ***7F 'E' 'L' 'F'*** which is ***7F454C46***



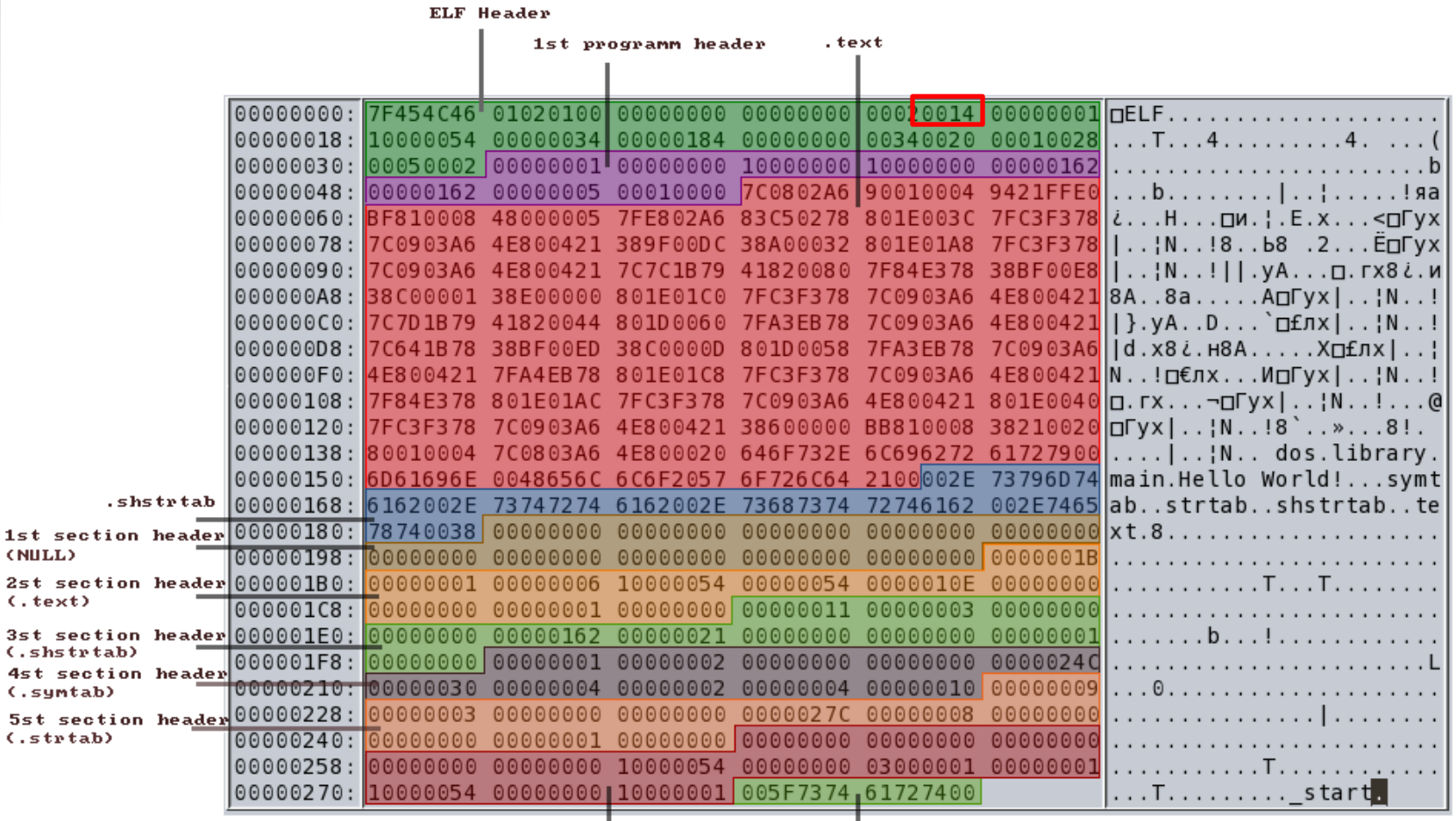
Value	Word Size
01	32 bit Format
02	64 bit Format



Value	Byte Order
01	Little Endian
02	Big Endian

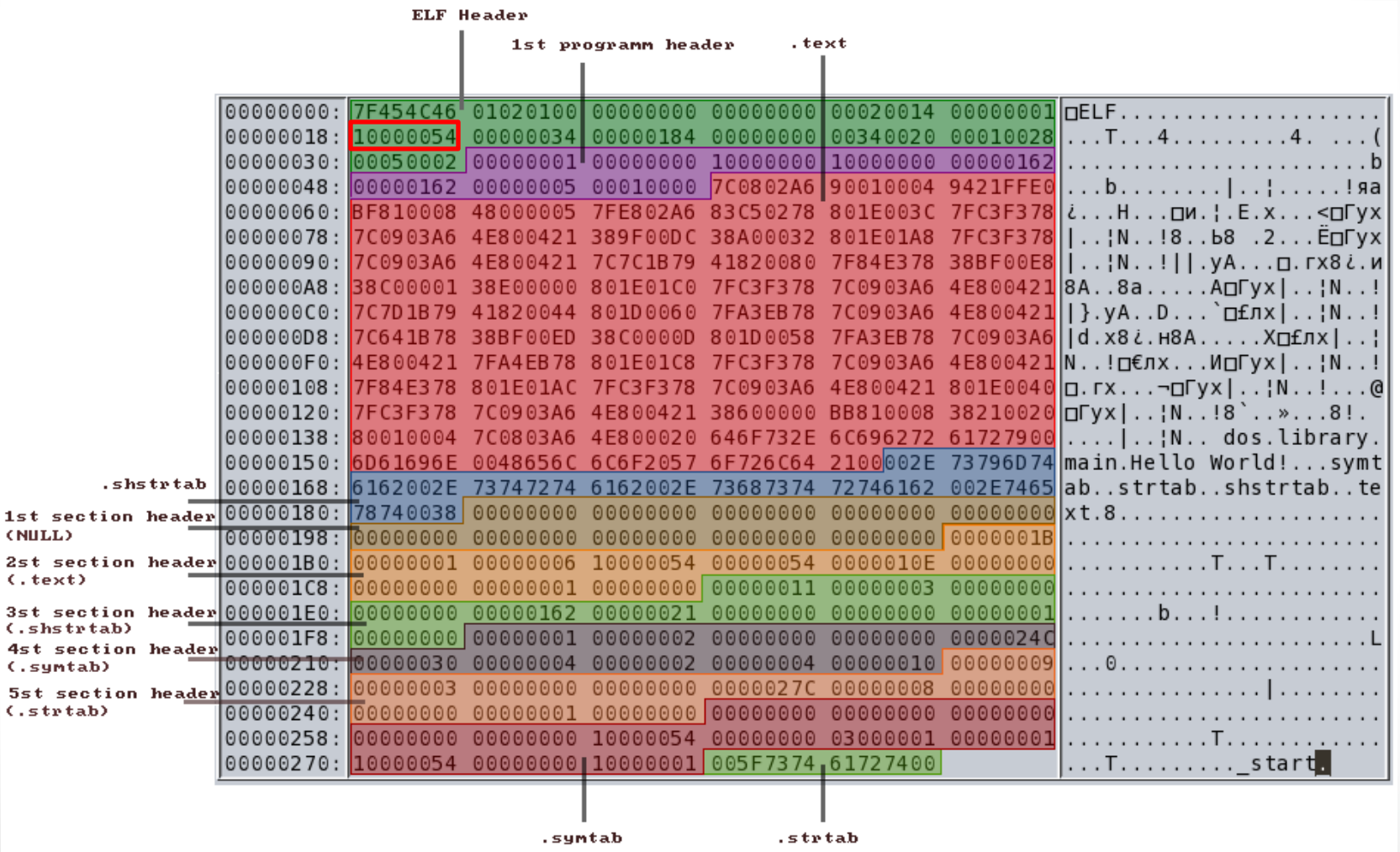


Value	File Type
00 01	Relocatable Object
00 02	Executable
00 03	Shared Object
00 04	Core dump

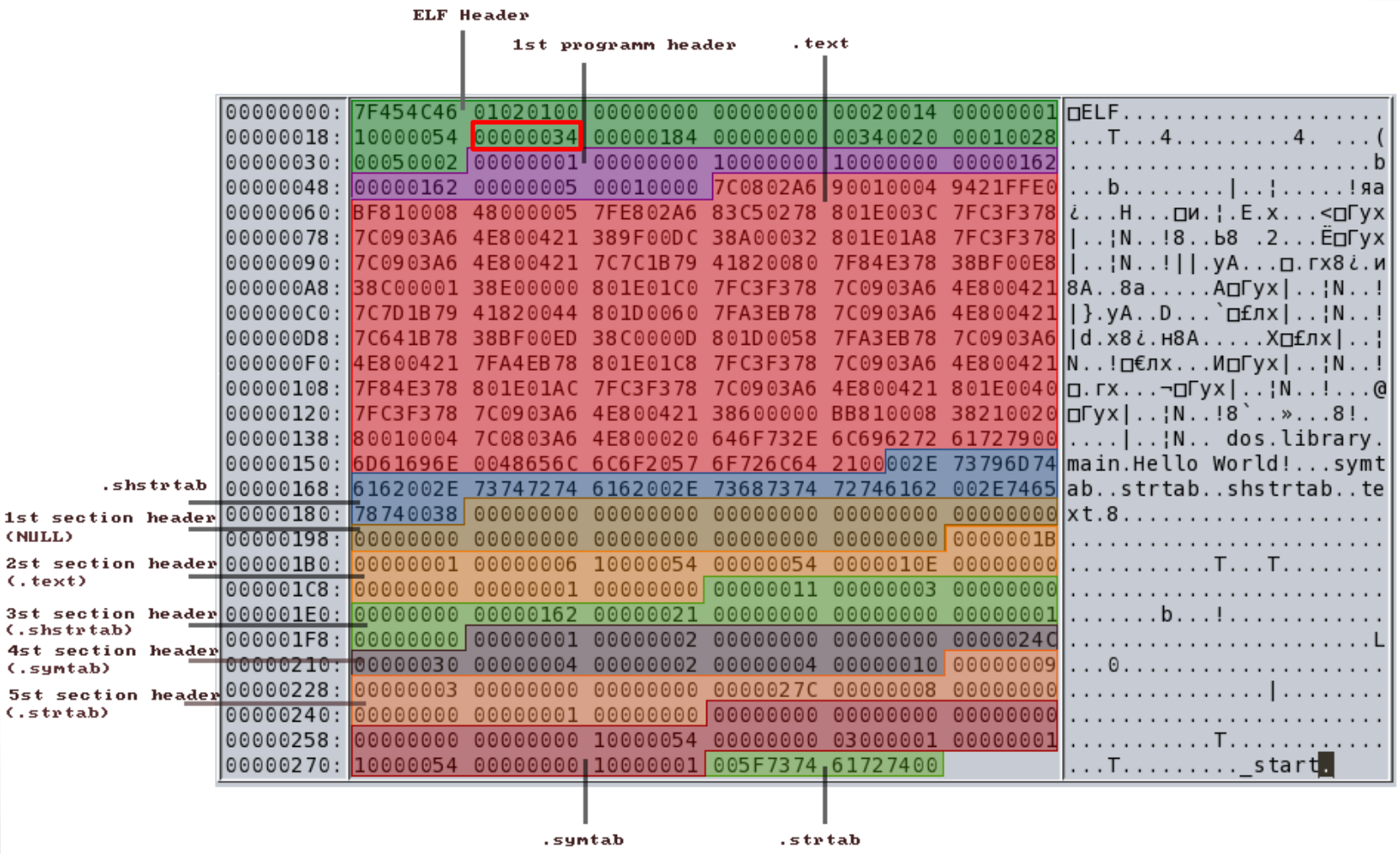


Value	Target Processor
00 03	x86
00 14	PowerPC
00 28	ARM
00 3E	x86-64

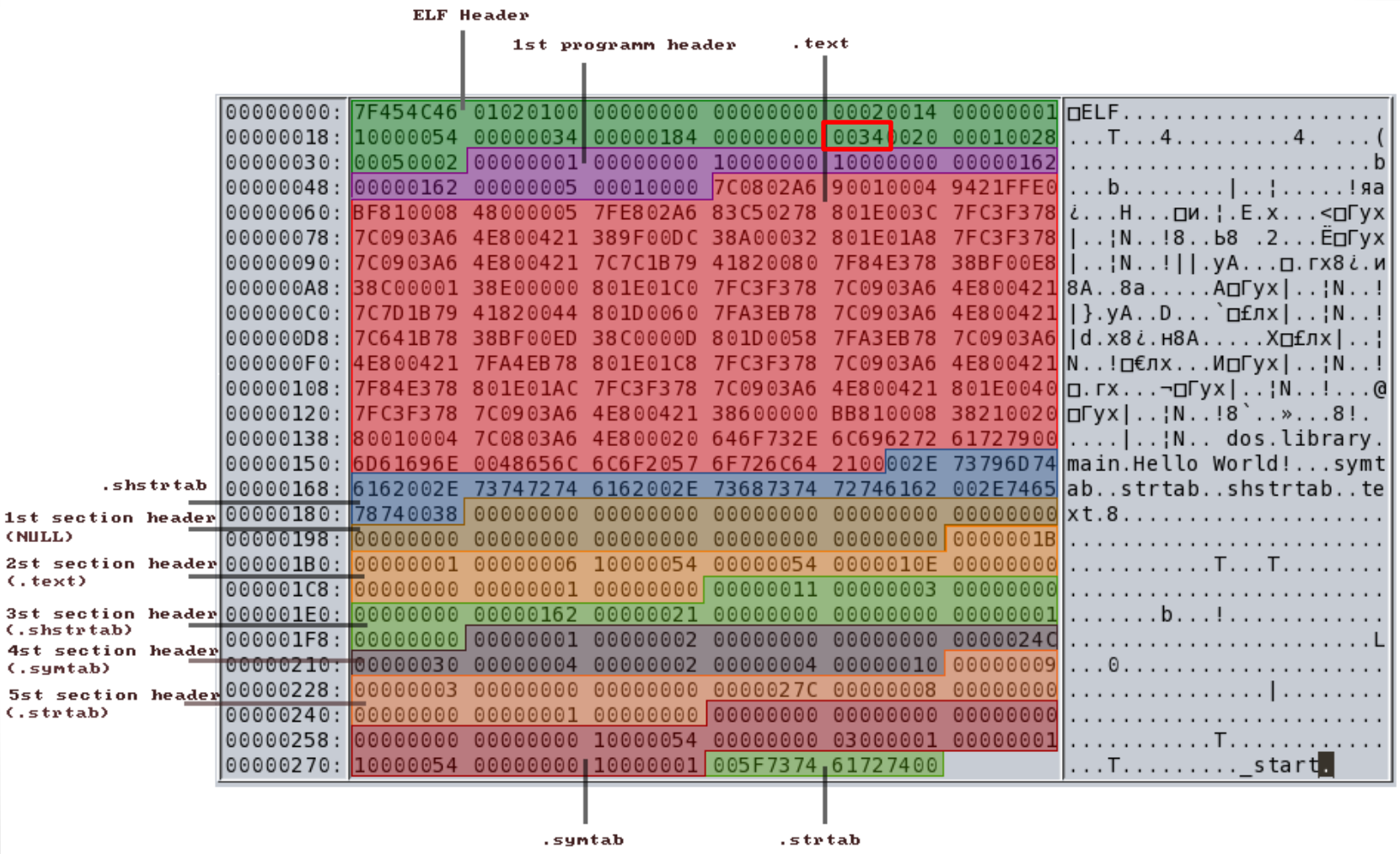
Program Entry Address In Memory



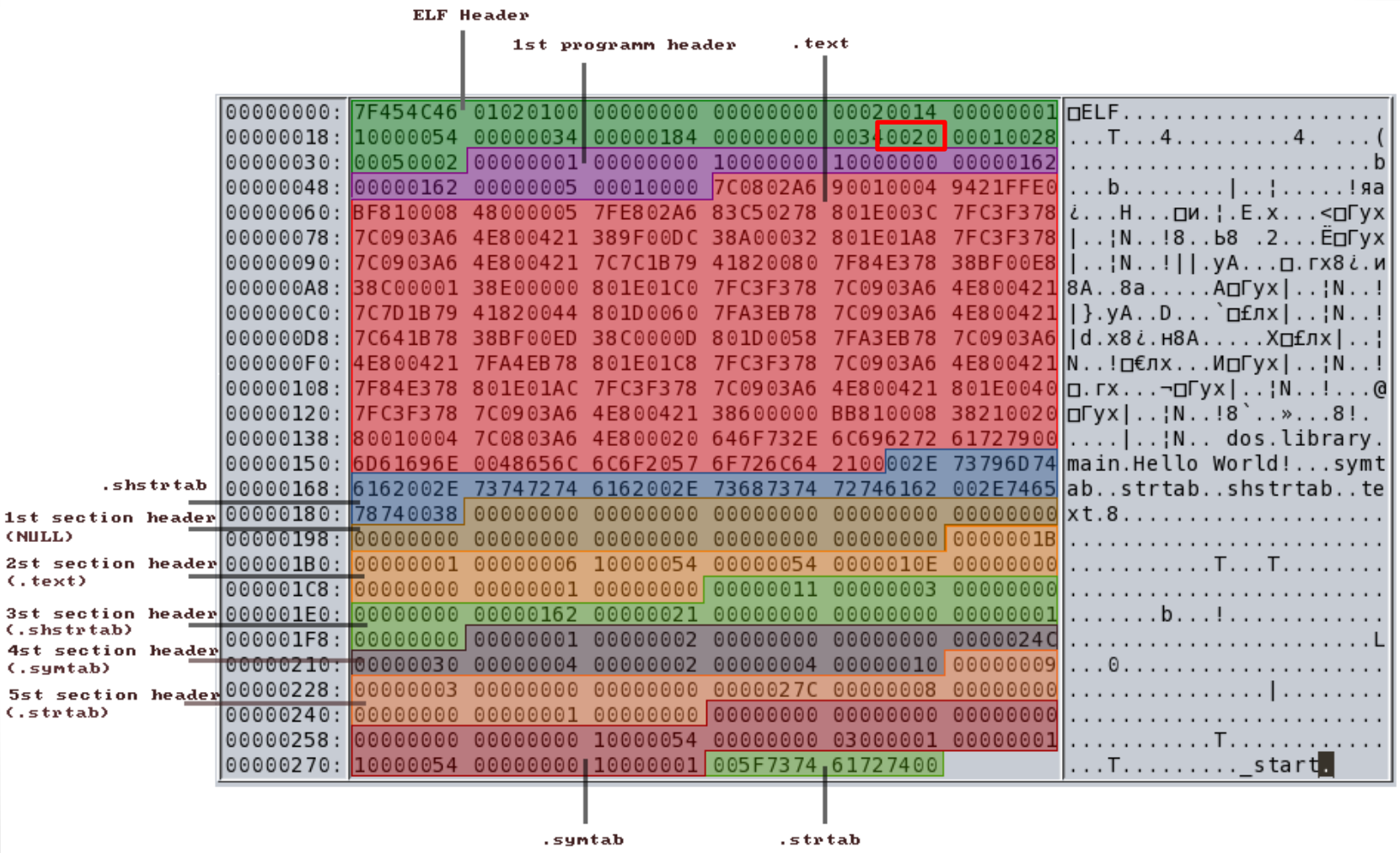
Section Header Table Location



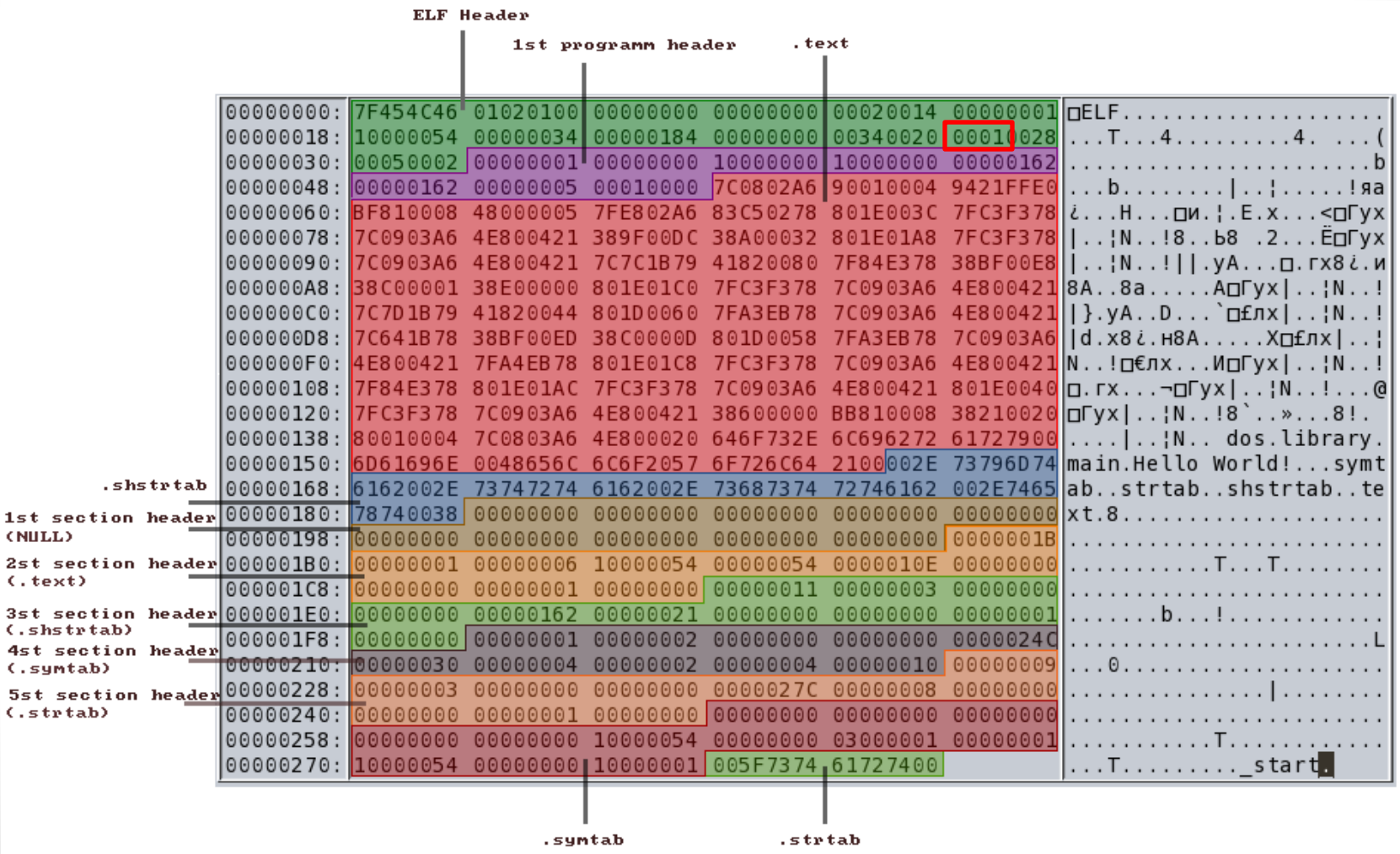
ELF Header Size



Program Header size



Program Table Entry Count



GNU readelf

Read ELF Files (readelf Command)

- This command is used to read the contents of an ELF file
- This includes
  - Main file header
  - Section Headers
  - Sections
  - Symbol table
  - … etc
- Can be used for,
  - Resolving linking problems, such as “Unresolved Symbol” error
  - Debugging a crash
  - Hacking an executable
  - Reverse engineering a binary file

Read ELF File Main Header

$ readelf -h <ELF file>

```
brook@vista:~/test/link_and_loader$ readelf -h a.out
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:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4007a0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8656 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

Other Options for readelf

- To Show the file sections

  $ readelf -S <ELF file>
- To show the file segments

  $ readelf -l <ELF file>
- To show the symbol table

  $ readelf -s <ELF File>
- To show all the elf file contents

  $ readelf -a <ELF File>

GNU objdump

Reading the ELF File (objdump Command)

$ objdump [options] <ELF file>

- This is another program to read from the ELF files
- It has a lot of usages depending of the chosen options
- For example, it can be used for,
  - Reading the ELF file Header (**readelf** does a better job with that)
  - Reading the ELF file sections headers
  - Reading the assembly code of the binary code
  - Reading the Symbol table(s)
  - Reading the Debug Information

Showing the ELF File Header


```
$ objdump -f factorial

factorial:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400440
```

- This usage shows the information of the main header for the ELF file
- This is equivalent to use of ***readelf -h*** (but shows less information)

Display Sections Headers

```
$ objdump -h factorial

factorial:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000400274  0000000000400274  00000274  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .hash         00000024  0000000000400298  0000000000400298  00000298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  ....
 14 .fini         0000000e  0000000000400668  0000000000400668  00000668  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata       0000001b  0000000000400678  0000000000400678  00000678  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .eh_frame_hdr 00000024  0000000000400694  0000000000400694  00000694  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .eh_frame     0000007c  00000000004006b8  00000000004006b8  000006b8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .ctors        00000010  0000000000600e18  0000000000600e18  00000e18  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 19 .dtors        00000010  0000000000600e28  0000000000600e28  00000e28  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 ...
 23 .got.plt      00000028  0000000000600fe8  0000000000600fe8  00000fe8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .data         00000010  0000000000601010  0000000000601010  00001010  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          00000010  0000000000601020  0000000000601020  00001020  2**3
                  ALLOC
 26 .comment      00000023  0000000000000000  0000000000000000  00001020  2**0
                  CONTENTS, READONLY
```

Showing Assembly Code of an ELF

```
$ objdump -d factorial

factorial:     file format elf64-x86-64

Disassembly of section .init:

00000000004003f0 :
  4003f0:       48 83 ec 08             sub    $0x8,%rsp
  4003f4:       e8 73 00 00 00          callq  40046c
  ..
Disassembly of section .plt:

0000000000400408 :
  400408:       ff 35 e2 0b 20 00       pushq  0x200be2(%rip)        # 600ff0
  40040e:       ff 25 e4 0b 20 00       jmpq   *0x200be4(%rip)        # 600ff8
  400414:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400418 :
  400418:       ff 25 e2 0b 20 00       jmpq   *0x200be2(%rip)        # 601000
  40041e:       68 00 00 00 00          pushq  $0x0
  400423:       e9 e0 ff ff ff          jmpq   400408
```

Showing Assembly Code of an ELF

- **objdump** can show the assembly code for the program
- It performs disassembly for all the ELF file sections that contain code

  $ objdump -d my-binary
- If you are interested in a specific section, then we need to identify what section to use,

  $ objdump -j .text -d my-binary
- Keep in mind,
  - You must be using the **objdump** for the same target platform that the program was compiled for
  - In case a processor supports to run in both little endian or big endian formats (such as ARM), you need to specify which format you want to use,

    $ objdump -EB -j .text -d my-binary (Big endian)

    $ objdump -EL -j .text -d my-binary (Little endian)

GNU nm

List Symbols (nm Command)

$ nm [options] <elf file>

- This command lists the symbols in the provided ELF file
- For each symbol, it presents,
  - Virtual Address of the symbol
  - Symbol type (Local, Global, Data, BSS, Undefined, …)
  - Name of the symbol
  - Size of the symbol (use the option **-S**)

The nm Command Usage

- To find the object files that use or define a certain symbol:

  $ nm -A ./*.o | grep var_1

  This command lists the symbol tables of all object files in the current directory along with the file name, and then filters that with the name of the symbol we are looking for

- To list all undefined symbols in an ELF file:

  $ nm -u my-obj-file

  These undefined symbols need to be resolved through static linking at build time or at run time via dynamically linking with a shared object

- To list dynamic symbols of an executable

  $ nm -D my-bin-file

GNU strings

List Strings in an ELF File (strings Command)

$ strings [options] <ELF File>

- This command lists all strings in a non-textual file
- It looks for any set of printable characters of 4 or more letters within the file
- Examples:

  $ strings a.out (Lists a set of strings within the binary file)

  $ strings -f /bin/* | grep "Copy"

  This Command searches for the Copyright in all binary files in /bin directory

GNU strip

Reduce ELF File Size (strip Command)

$ strip [options] <object file>

- The strip command is used to reduce the size of the ELF file
- This applies for both executable or object files
- This is performed by removing some of the tables and sections that the binary can run without
- This is useful for:
  - Reduce the requirement for Flash and memory usage (specially useful for embedded systems)
  - Protect the code from being reverse engineered

Usage Examples

- To strip an executable from its symbol table

  $ strip -s my-bin-file

- To remove debug symbols

  $ strip --strip-debug my-bin-file

- Remove all un-needed symbols

  $ strip --strip-unneeded my-bin-file

- To keep the original file, and create a stripped file

  $ strip -s -ostripped-file my-bin-file

GNU addr2line

addr2line Command

- To know the location in the source code for a specific address,

  $ addr2line -e my-bin-file 0x400534

- This can be useful when handling a crash, and we know the address of the instruction that crashed, and need to know the source code line that caused the crash
- To know also the function name for that address

  $ addr2line -f -e my-bin-file 0x400534

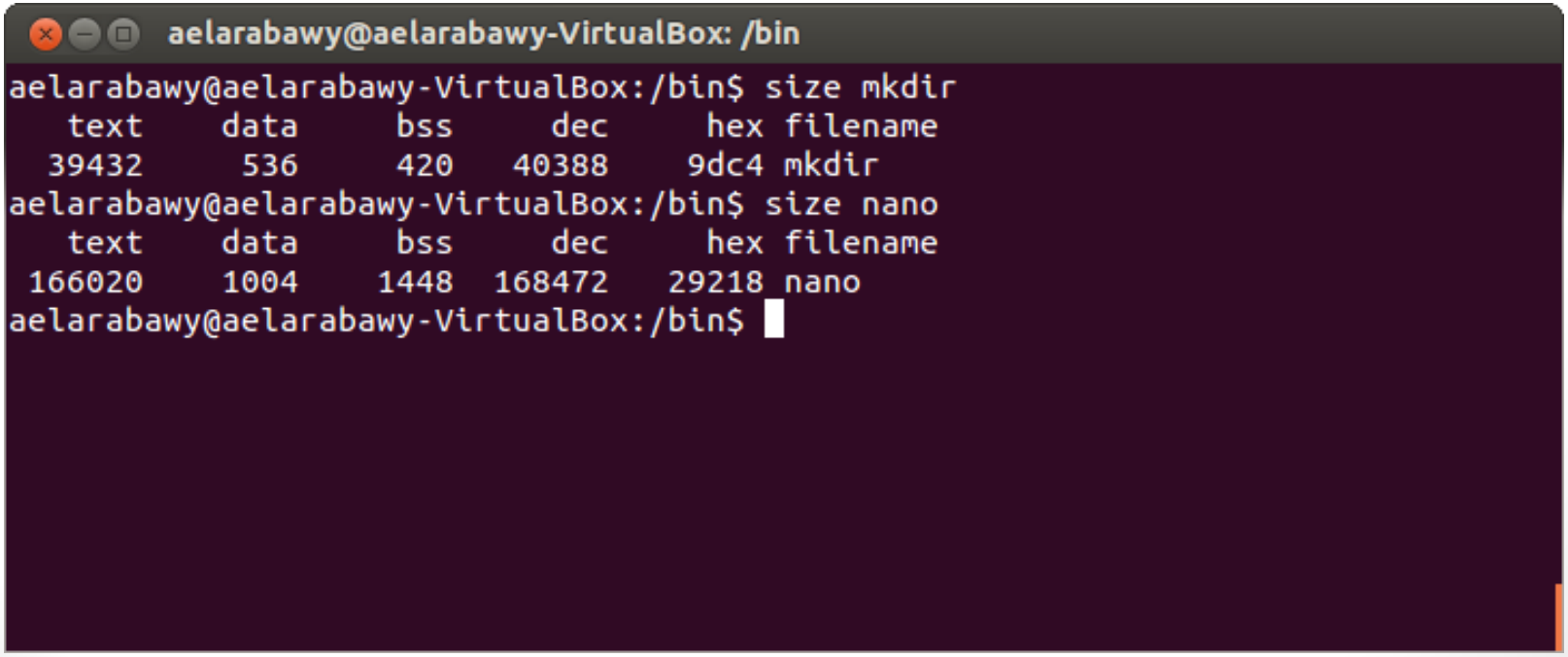
GNU size

size Command

$ size [options] <ELF File>

- This command is used to list the sizes of the different sections inside an elf file

  $ size my-bin-file



size Command

- **In Linux based target platforms,**
  - We load the whole ELF file into the target
  - Hence, the Flash size of the target needs to accommodate the full size of the ELF file
  - Accordingly, using the ***size*** command becomes useful to find out the real size of each section inside the ELF file
  - This helps us identify where to optomize,
    - Is it the code (**text** section) ?
    - Is it the defined data (**data** and **bss** sections) ?
  - We may need to use the ***strip*** command to remove some of the optional sections to reduce the requirements on the Flash size
- **In simpler platforms (bare bone, or simple RTOS),**
  - The Flash Loader only loads the necessary parts to the target
  - The debug sections are left in the version on the host machine for debugging purposes, but the target only gets the binary code, and the data sections
  - Accordingly, using the ***strip*** command does not really help
  - The ***size*** command becomes very important, since we don't care about the size of the ELF file, we care more about the size of some specific sections

Linux 4 Embedded Systems

http://Linux4EmbeddedSystems.com