ELF and Linker Script

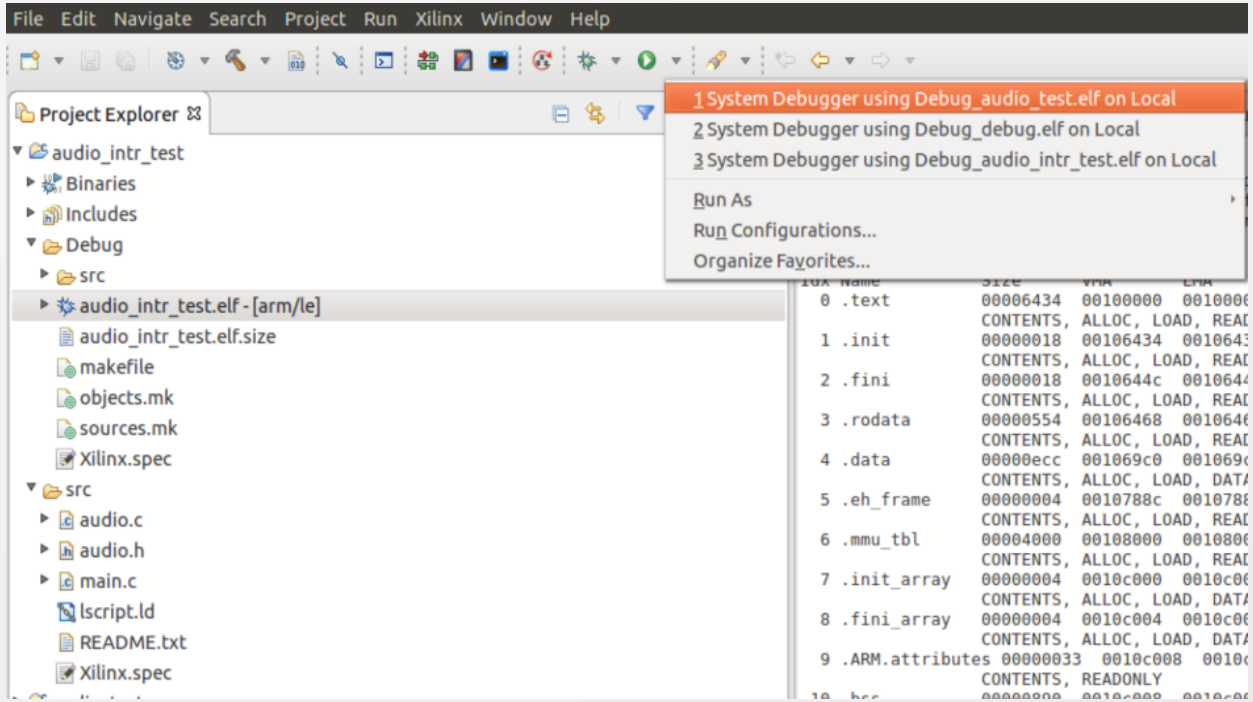
ELF (*.elf) File

The Xilinx SDK uses the ARM GCC compiler/assembler.

The GCC tools “assemble” (or process) your source code to produce a .elf file that contains the binary/machine code.

This executable binary code can be loaded in memory and executed by the processor.

But what is .elf file, and how is it generated?

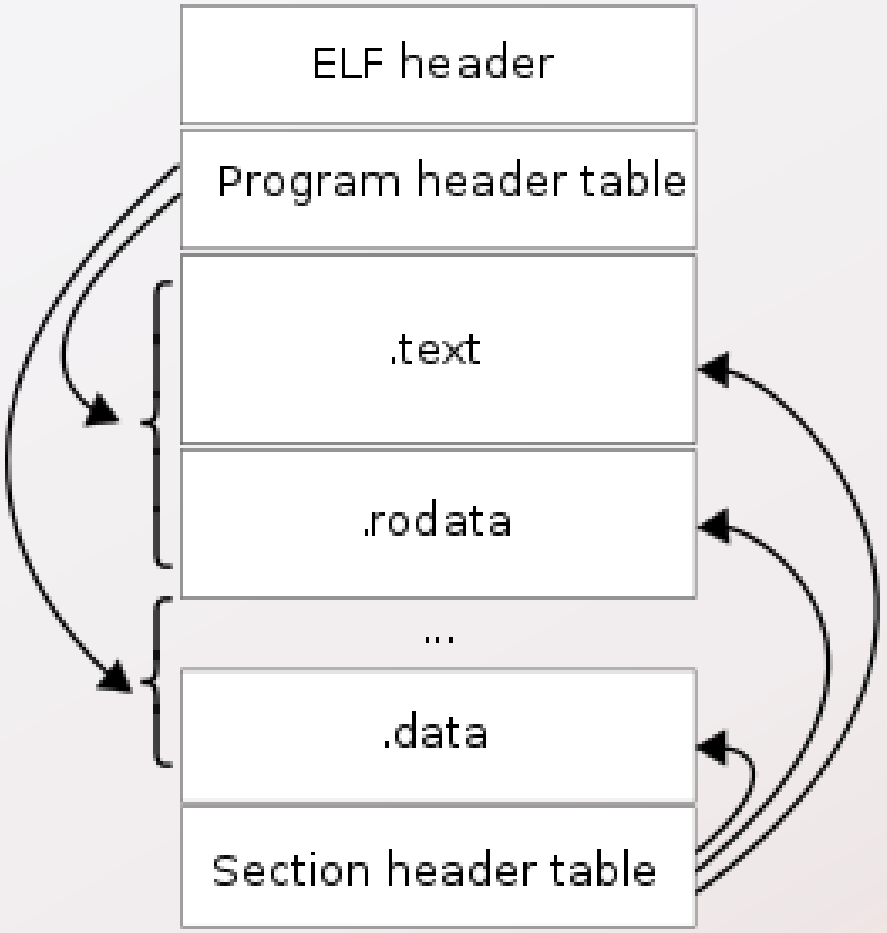


“Executable and Linkable File” format (.elf)

ELF is the standard format for executable files, object code, shared libraries, and core dumps.

ELF files are flexible, extensible, cross-platform, and not bound to any CPU or instruction set.

ELF files have been adopted by many different operating systems on many different hardware platforms.



http://www.cirosantilli.com/elf101.png

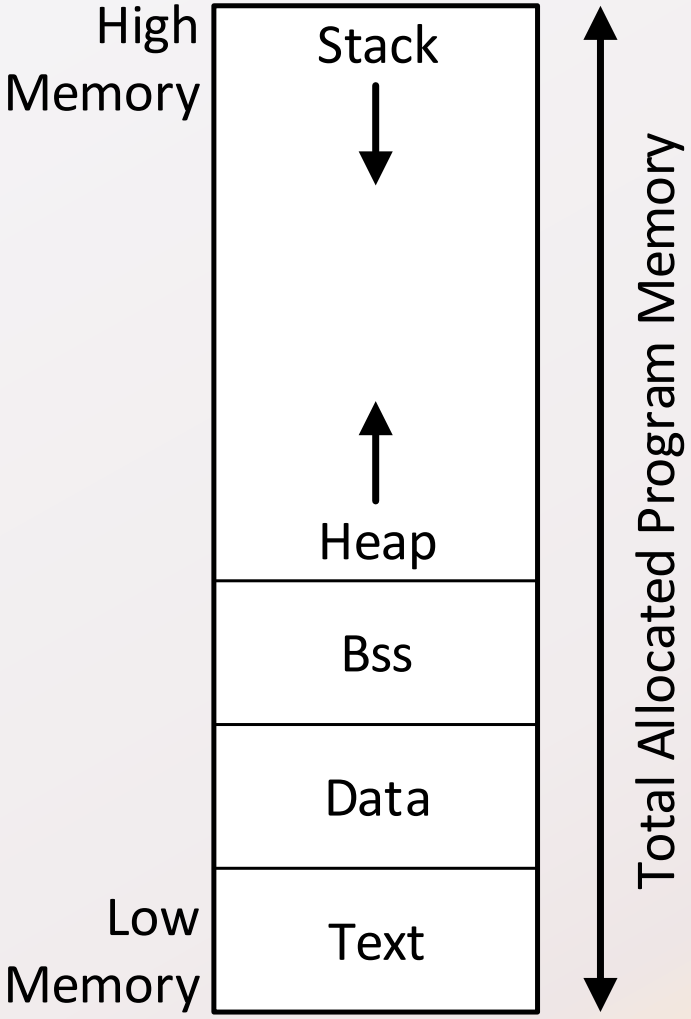
Reference: Wikipedia – Executable and Linkable Format
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Sections

"Programs" are divided into segments, or sections of memory. The ELF file defines and uses the segments for general organization purposes.

The .text section contains opcodes (executable statements). The size is fixed by assembler/compiler. It is (typically) read only, may be loaded in ROM.

The .data segment contains initialized static variables, global variables, and static local variables. Size is fixed by assembler/complier; data in this segment may be read and written. The .rodata (read only data - if present) segment contains static constants.

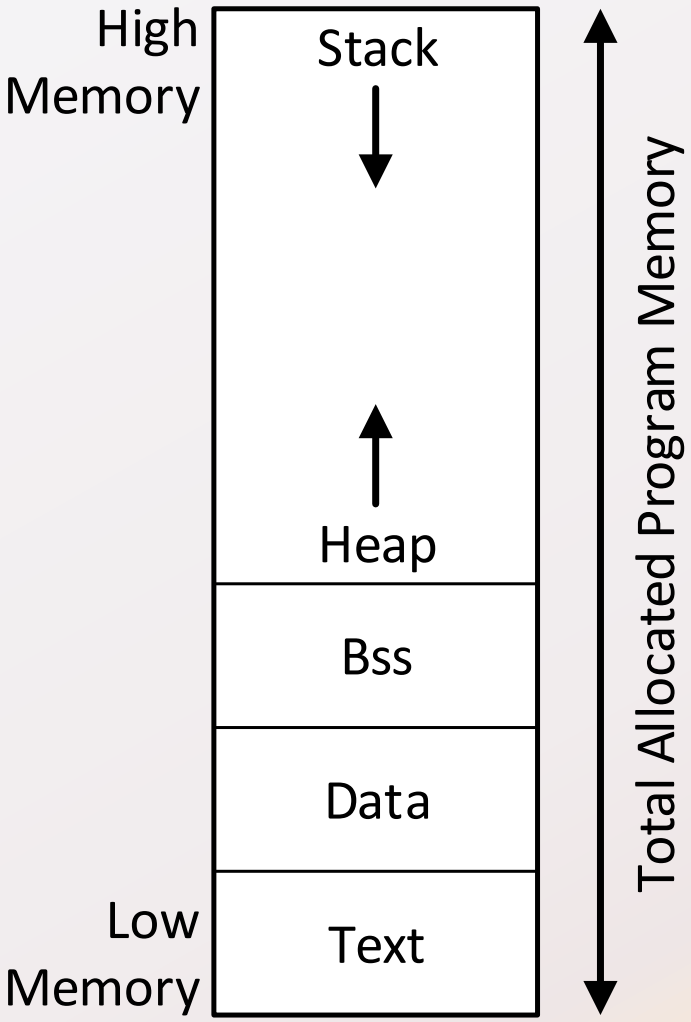


Sections

The .bss section (“block started by symbol”) contains uninitialized statically allocated variables. It is a read/write data segment, with size fixed at assemble/compile time, that is set aside for known needs. Only its size is stored in .elf file.

The .heap section is for run-time allocated dynamic memory needs. It typically begins at end of .bss and grows into larger memory.

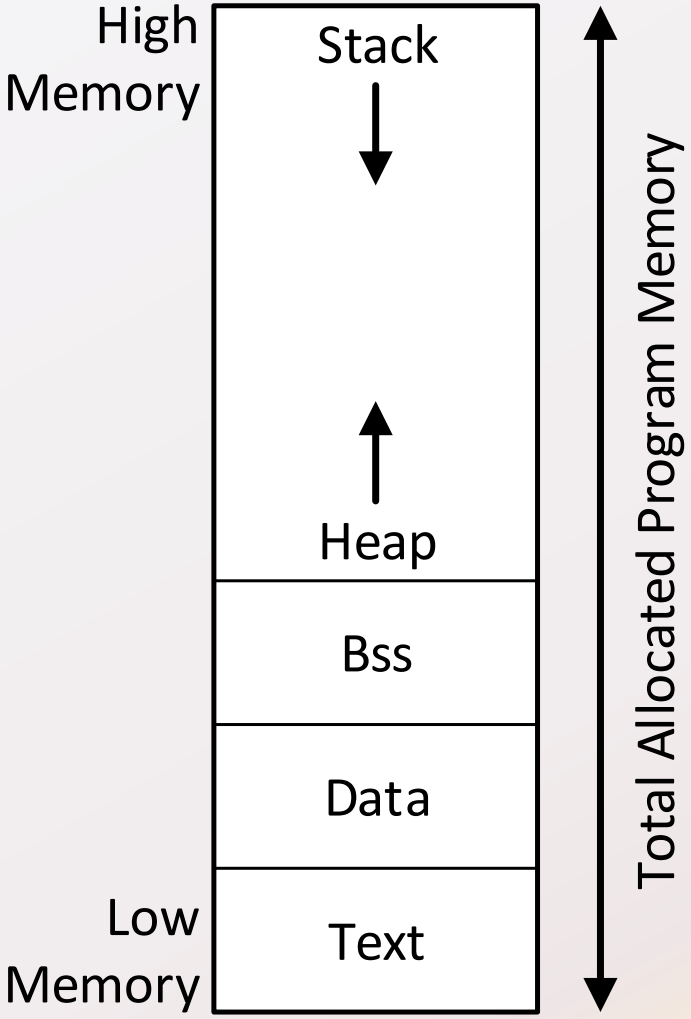
The .stack section is for run-time, dynamically allocated memory used for passing parameters between subprograms and storing local context. It typically starts in high memory and grows towards heap.



Sections

Many Assemblers define additional sections as well to define areas of memory for initialization data, or operating parameters.

Sections are used to organize the source file. Historically, there was some correlation between sections on memory requirements (RAM, ROM, high-speed, off-line, etc.). In a large, flat/uniform memory space, they are really just “organizational artifacts”, or labels identifying code segments and data areas.



Assembler

The Assembler reads your source file, resolves references, replaces mnemonics with opcodes, and produces some or all of the sections of an ELF file.

The ELF file can be directly executed after it is loaded into main memory.

The assembler passes through your code twice: once to make a symbol table, and a second time to replace mnemonics, labels and references with opcodes.

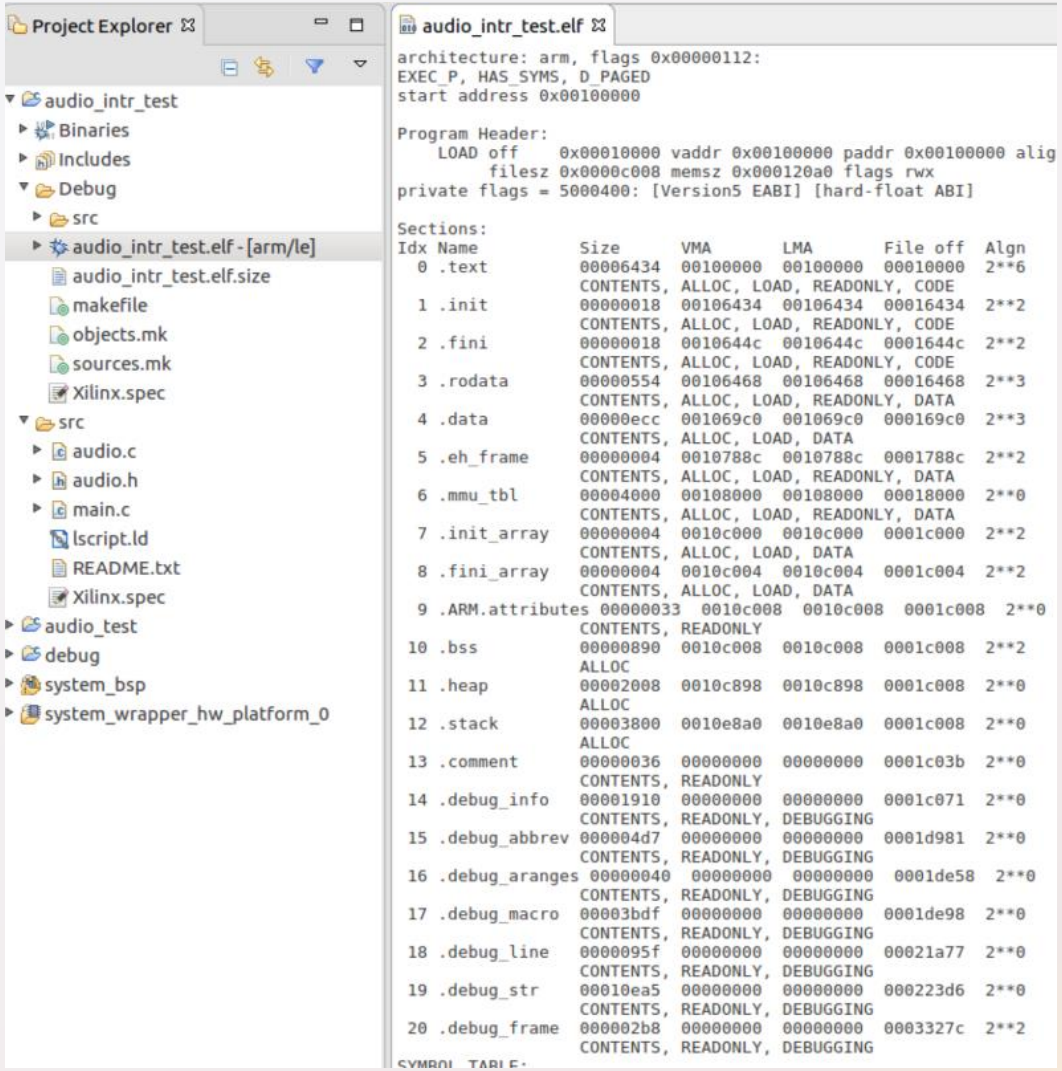
> A symbol table is not executable. It is a reference table that is built by associating all labels and section-identifying directives with “virtual” addresses (virtual addresses may be adjusted by adding a constant when the program is actually placed in memory).

An .elf file example for an ARM Program

In Xilinx SDK, you can open and read the .elf file.

The program shown on the right has 21 segments defined (this is the default for .elf files prepared by SDK). The .elf file shows the size and memory location of each segment.

Segment information is used to correctly place your program in memory, and to help ensure you do not make obvious mistakes.



Linker (LD) and linker script (.ld)

Linker (LD) is a program that combines one or more object files (compiled/assembled source code files generated by the compiler/assembler) and combines them into an executable/binary file (the .elf file).

The Linker script (usually a file with .ld extension) includes information needed by the linker, including the memory locations of each segment. Segment physical addresses are needed so your source code can be properly located in physical memory (so the processor can access and execute it).

The debugger also uses segment information to access and display relevant memory dumps.

A Simple Linker Script

The file on the right shows a simple linker script. The **SECTIONS** area defines the location of each section:

- Code (text) segment starts at 0x10000
- Data segment starts at 0x8000000
- Unintialized data segment starts after the data segment.

```
SECTIONS {
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Reference: binutils 2.29 documentation – 3.3 Simple Linker Script Example
https://sourceware.org/binutils/docs/ld/Simple-Example.html#Simple-Example

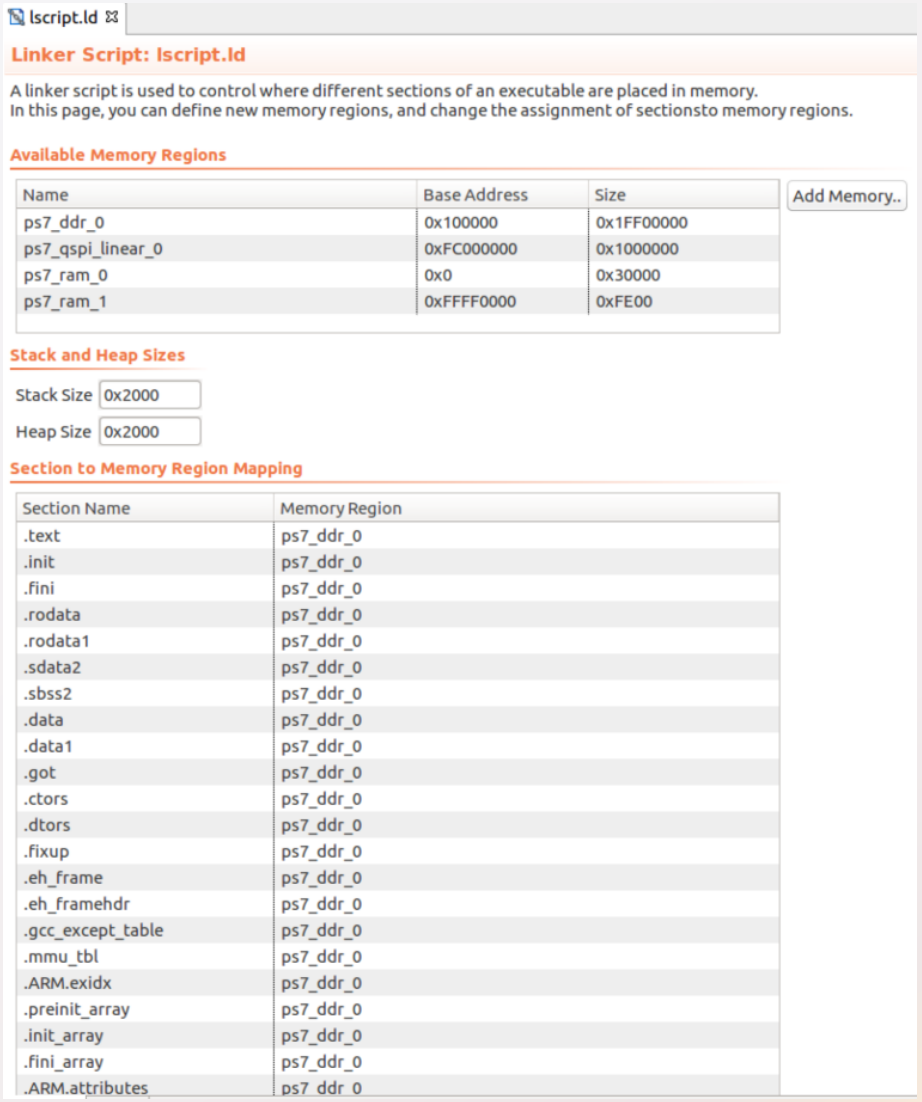
Example in Xilinx SDK

A linker script (usually named `lscript.ld`) is placed in your project directory.

You can open the linker script by double clicking the file. SDK will parse the basic information in the file and load it in a table as shown on the right.

The linker script defines all available memory regions, and defines the location of all program sections in the memory regions.

You can edit the default linker script information (for example, you can change the size of Stack and Heap).



Example in Xilinx SDK

You can also click the “source” tab to view the linker script in a text editor.

The MEMORY area specifies the size of volatile and non-volatile memory in the system.

The SECTIONS area specifies segment locations. For example, the code (text) section is mapped to `ps7_ddr_0` which starts at 0x100000 with a size of 0x1FF00000.

lscript.ld

Linker Script: lscript.ld

```
/* Define Memories in the system */

MEMORY
{
   ps7_ddr_0 : ORIGIN = 0x100000, LENGTH = 0x1FF00000
   ps7_qspi_linear_0 : ORIGIN = 0xFC000000, LENGTH = 0x1000000
   ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
   ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
}

/* Specify the default entry point to the program */

ENTRY(_vector_table)

/* Define the sections, and where they are mapped in memory */

SECTIONS
{
.text : {
   KEEP (*(.vectors))
   *(.boot)
   *(.text)
   *(.text.*)
   *(.gnu.linkonce.t.*)
   *(.plt)
   *(.gnu_warning)
   *(.gcc_execpt_table)
   *(.glue_7)
   *(.glue_7t)
   *(.vfp11_veneer)
   *(.ARM.extab)
   *(.gnu.linkonce.armextab.*)
} > ps7_ddr_0

.init : {
   KEEP (*(.init))
} > ps7_ddr_0

.fini : {
```

The Process…

Your source files are created and assembled/complied into an .elf file. Since the .elf file can be executed, all external references and relative label addresses must be resolved (i.e., associated with a physical address).

The .elf files can be emulated, or executed on the target hardware.

Before an .elf file can be executed, it must be placed in memory so that the processor can jump to the first instruction. This is the linkers job: the linker (loader) places code according to segment definitions, and loads (links) any other external programs and determines all needed physical addresses for entry points.

The IDE (SDK) lets you “run” your program after it is loaded. The run command generates a software interrupt that causes the processor to execute an instruction from a fixed location – and that instruction is a jump to the first instruction of your code (more on this later).