ELF and Linker Script

Where are the compiled source codes?

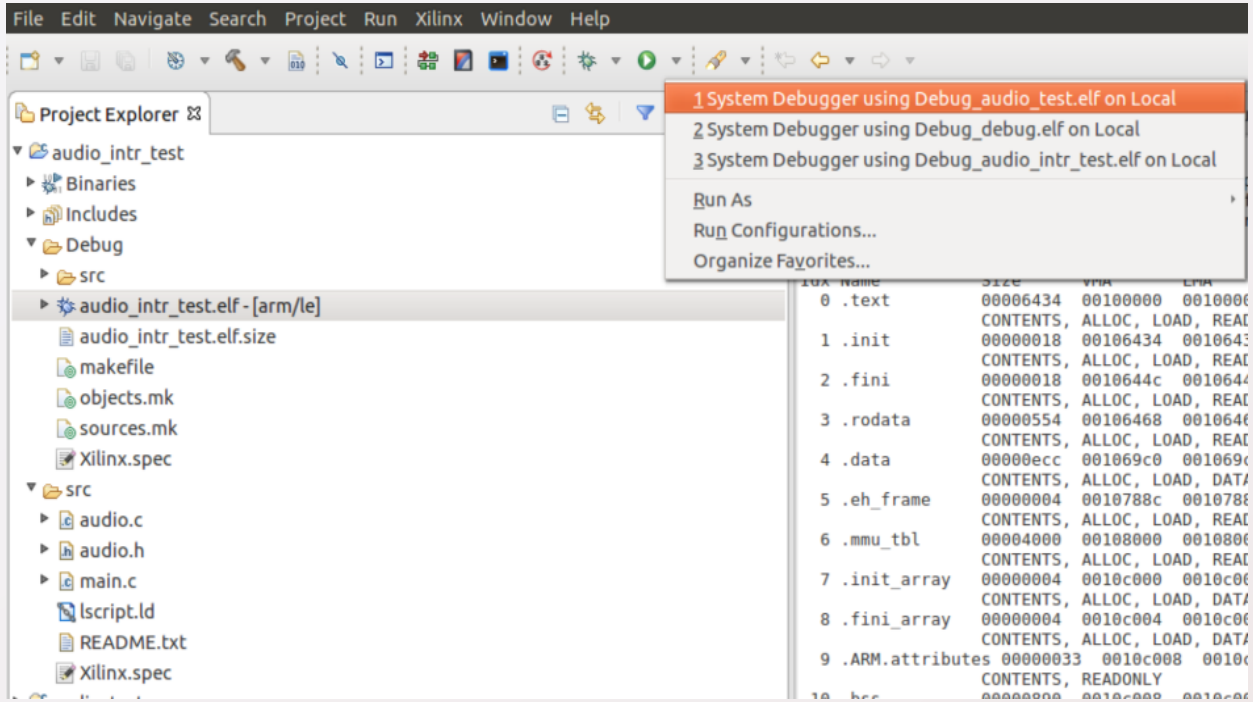
ELF (*.elf) File

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?

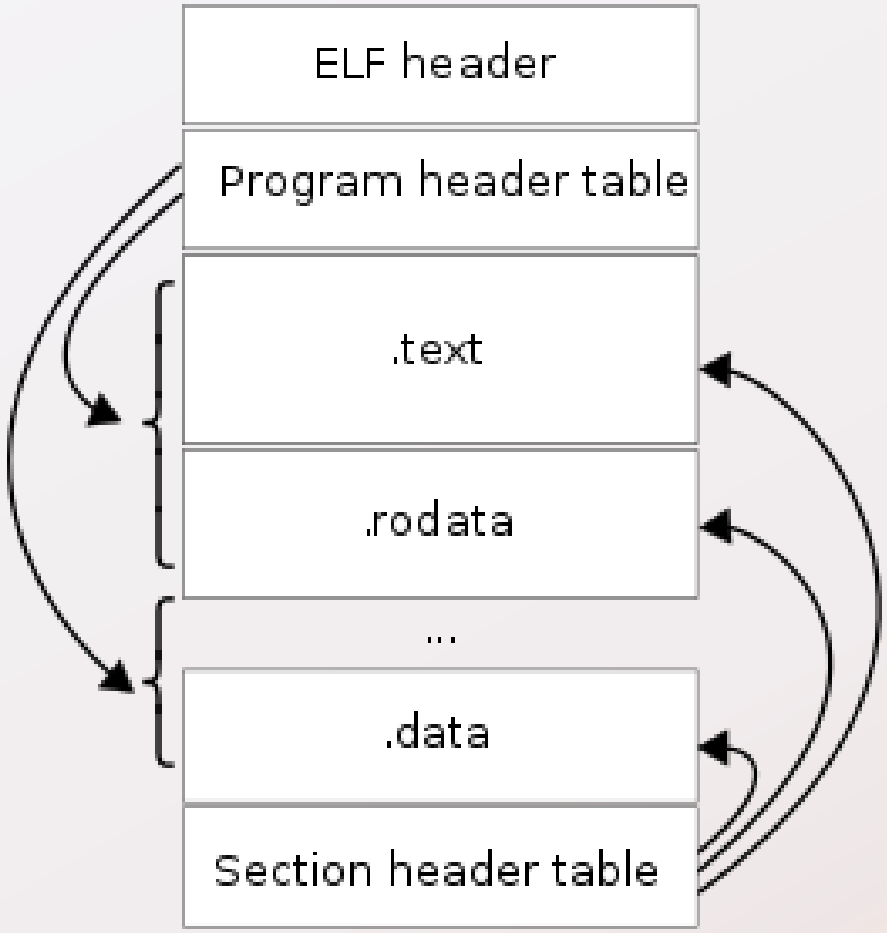


Executable and Linkable Format (.elf)

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

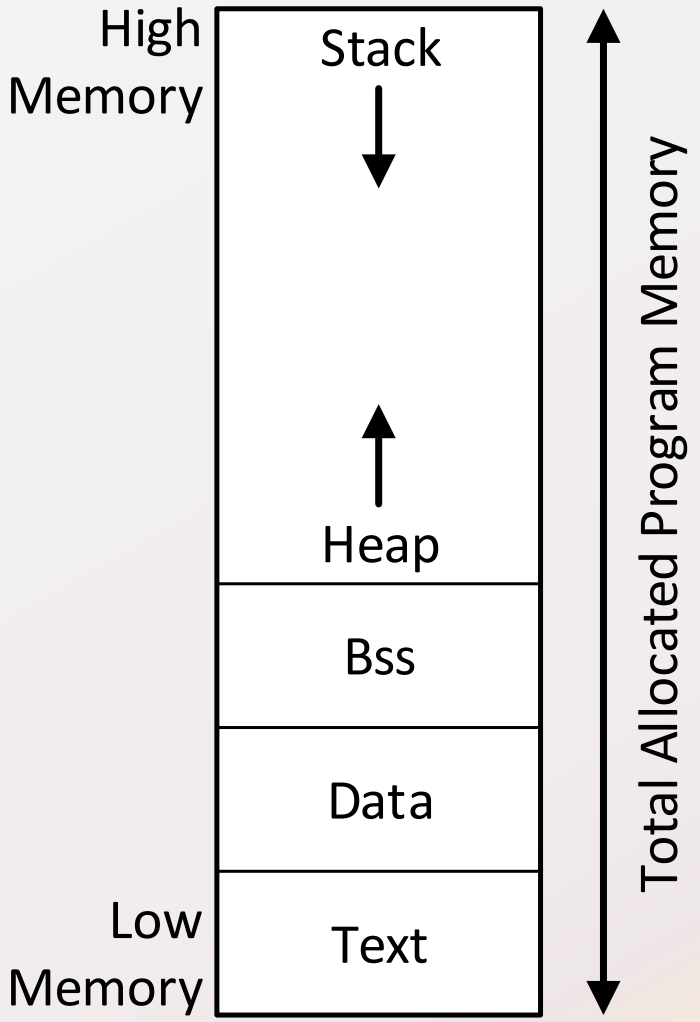
Reference: Wikipedia – Executable and Linkable Format
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

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.

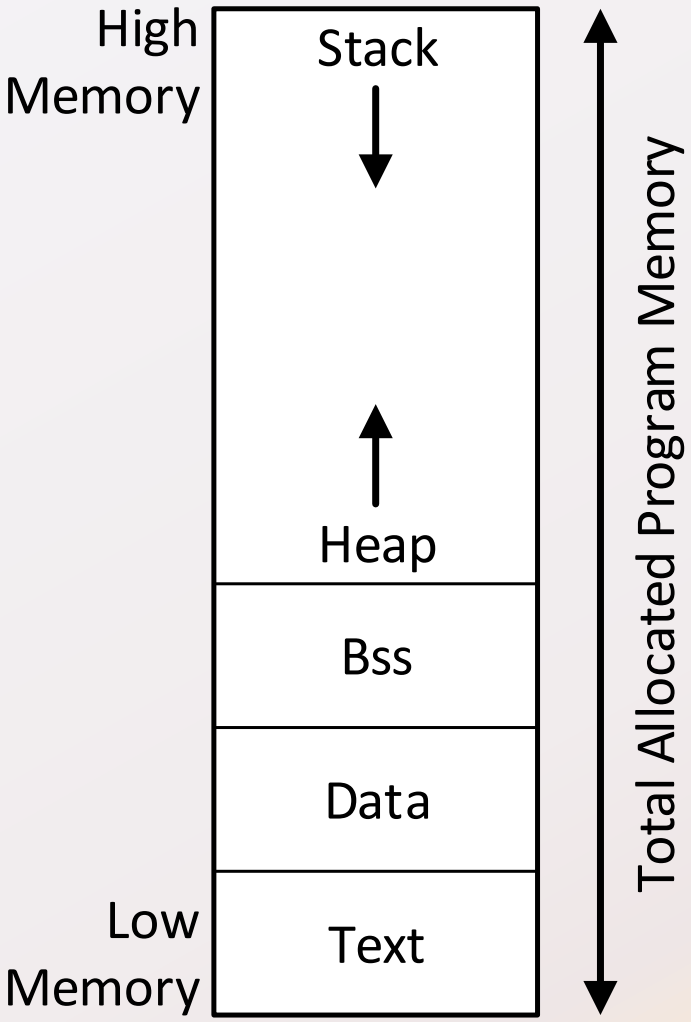


Sections

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.

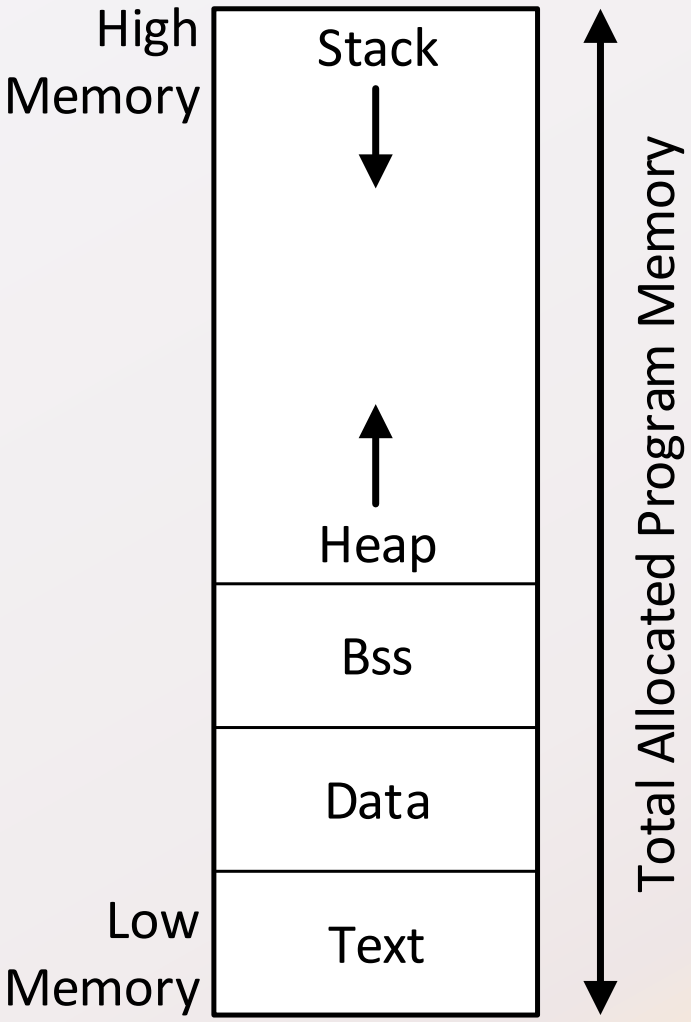
The .stack section is for run-time, dynamically allocated memory used for passing parameters between subprograms and storing local context. It typically starts in high memory and grows towards heap.



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 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).

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.

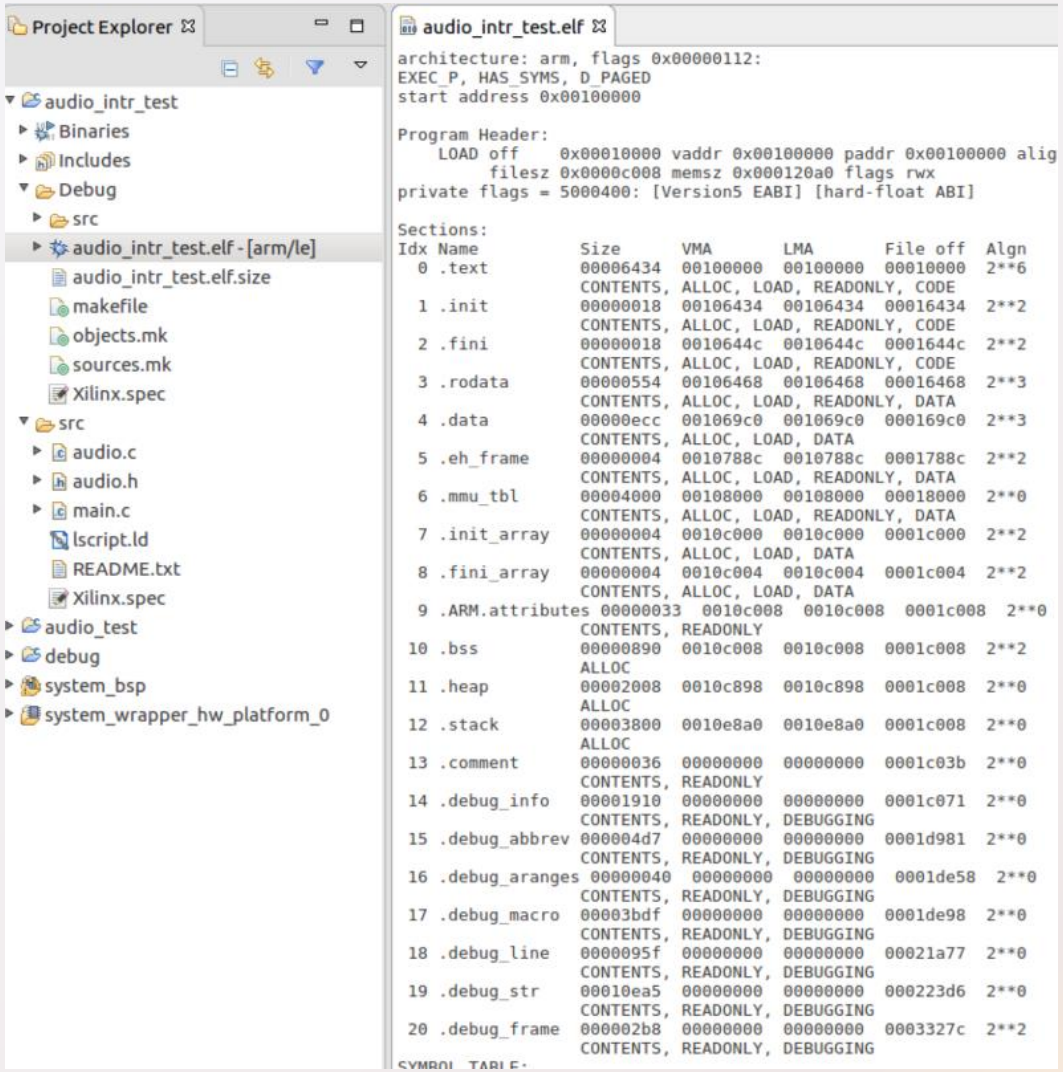
The ELF file can be directly executed after it is loaded into main memory.

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 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).

Example of an ARM Program

- In Xilinx SDK, you can get the information about the ELF file by double clicking the compiled ".elf" file.
- The program shown on the right has 21 segments defined, and it shows the size, memory location of each segments.
- As microcontroller has limited resources, these information are very important.



Linker (LD) and linker script (.ld)

- Linker (LD) is a program that combines one or more object file (compiled source codes) generated by the compiler and combines them into an executable/binary file – in our case, an ELF file.
- The **linker** needs instruction about how to map your compiled source codes with physical memory locations.
- **Linker script** (usually a file with .ld extension) provide such information by specifying the memory location of each program segment, and mapping your source codes into the program segments they belong.

A Simple Linker Script

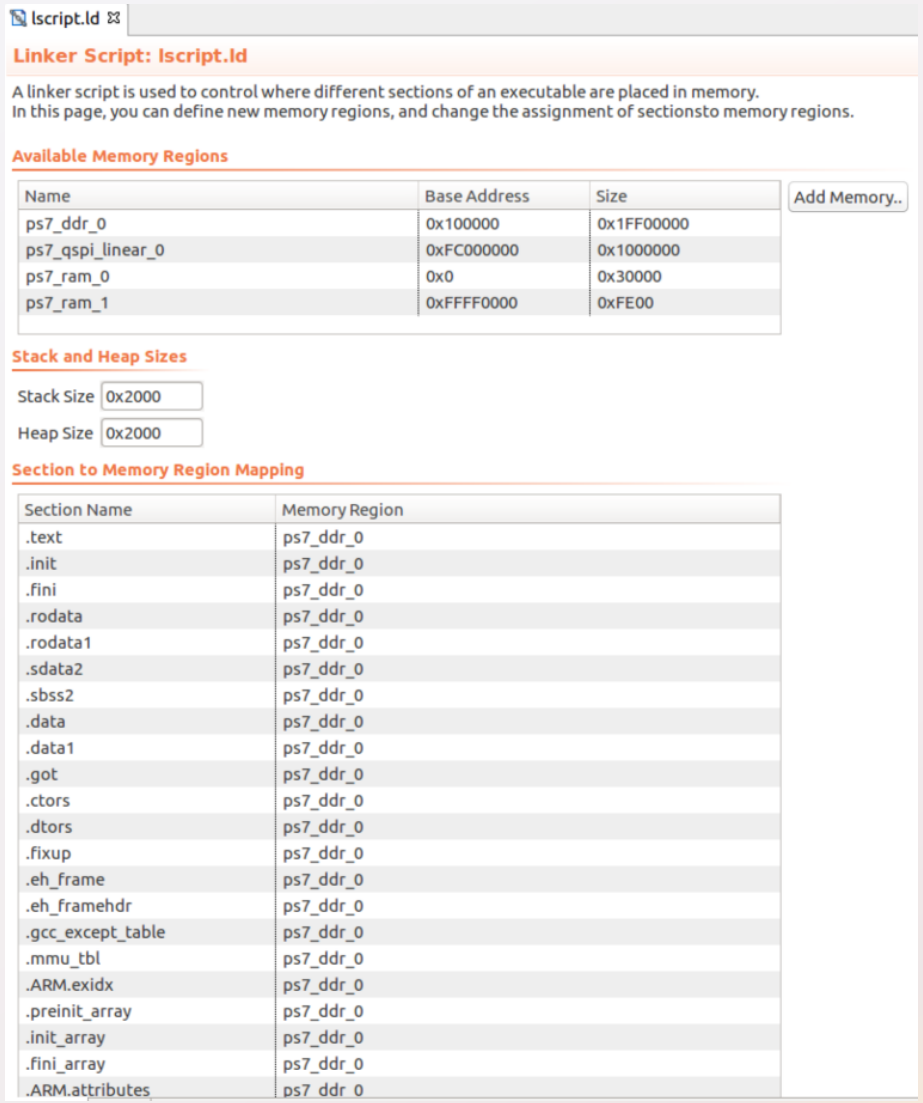
- The file on the right is a simple linker script.
- **SECTIONS** command is used to describe the location of each sections:
  - Code Segment starts at 0x10000
  - Data segment starts at 0x8000000
  - Unintialized data segment starts after the data segment.

```
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

- You will always find an linker script (usually named `lscript.ld`) in the project you created in XSDK.
- You can open the linker script by double clicking the file. XSDK will parse the basic information in the file and load it in a table as shown on the right.



lscript.ld

Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory.
In this page, you can define new memory regions, and change the assignment of sectionsto memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0	0x100000	0x1FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFF0000	0xFE00

Add Memory..

Stack and Heap Sizes

Stack Size 0x2000

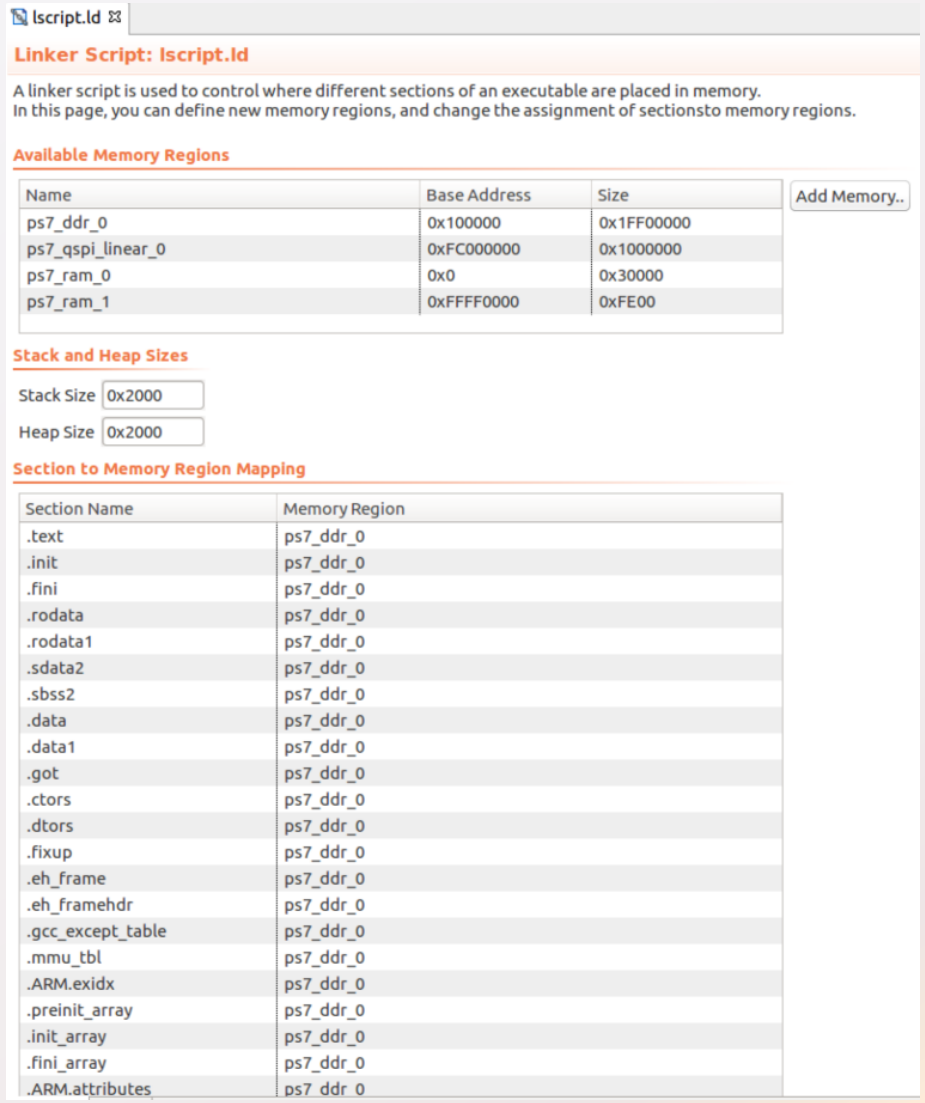
Heap Size 0x2000

Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.sdata2	ps7_ddr_0
.sbss2	ps7_ddr_0
.data	ps7_ddr_0
.data1	ps7_ddr_0
.got	ps7_ddr_0
.ctors	ps7_ddr_0
.dtors	ps7_ddr_0
.fixup	ps7_ddr_0
.eh_frame	ps7_ddr_0
.eh_framehdr	ps7_ddr_0
.gcc_except_table	ps7_ddr_0
.mmu_tbl	ps7_ddr_0
.ARM.exidx	ps7_ddr_0
.preinit_array	ps7_ddr_0
.init_array	ps7_ddr_0
.fini_array	ps7_ddr_0
.ARM.attributes	ps7_ddr_0

Example in Xilinx SDK

- In this linker script, it defines all available memory regions in the design and the location of all program sections in the memory region.
- It also provides you with the options to change the size of Stack and Heap.



lscript.ld

Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory.
In this page, you can define new memory regions, and change the assignment of sectionsto memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0	0x100000	0x1FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFF0000	0xFE00

Add Memory..

Stack and Heap Sizes

Stack Size 0x2000

Heap Size 0x2000

Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.sdata2	ps7_ddr_0
.sbss2	ps7_ddr_0
.data	ps7_ddr_0
.data1	ps7_ddr_0
.got	ps7_ddr_0
.ctors	ps7_ddr_0
.dtors	ps7_ddr_0
.fixup	ps7_ddr_0
.eh_frame	ps7_ddr_0
.eh_framehdr	ps7_ddr_0
.gcc_except_table	ps7_ddr_0
.mmu_tbl	ps7_ddr_0
.ARM.exidx	ps7_ddr_0
.preinit_array	ps7_ddr_0
.init_array	ps7_ddr_0
.fini_array	ps7_ddr_0
.ARM.attributes	ps7_ddr_0

Example in Xilinx SDK

- You can also click the "source" tab to view the linker script in text editor.
- **MEMORY** command specifying the size of volatile and non-volatile memory in the system.
- In the **SECTIONS** command, for example, code 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 : {
```