### **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?

	D 🚯 🔽	<u>1</u> System Deb	ugger using Debu	ig_audio_test	elf on Local
		2 System Deb	ugger using Debu	g_debug.elf	on Local
🔻 🐸 audio_intr_test		<u>3</u> System Deb	ugger using Debu	ig_audio_intr	test.elf on Local
🕨 👹 Binaries					-
Includes		<u>R</u> un As			
T C Debug		Ru <u>n</u> Configur	ations		
bebug		Organize Fa <u>v</u>	orites		
				512 <del>0</del>	VITIA LITIA
\$\$ audio_intr_test.	elf - [arm/le]		0.Lext	CONTENTS.	ALLOC, LOAD, R
audio_intr_test.	elf.size		1 .init	00000018	00106434 00100
🗋 makefile			2 61-1	CONTENTS,	ALLOC, LOAD, R
la objects.mk			2 .1111	00000018 CONTENTS	0010644C 00100
			3 .rodata	00000554	00106468 0010
Sources.mk				CONTENTS,	ALLOC, LOAD, R
Xilinx.spec			4 .data	00000ecc	001069c0 00100
🔻 🗁 SFC			5 .eh frame	00000004	0010788c 0010
audio.c				CONTENTS,	ALLOC, LOAD, R
audio.h			6 .mmu_tbl	00004000	00108000 00108
b S main c			7 init array	CONTENTS,	ALLOC, LOAD, R
			/ .init_array	CONTENTS,	ALLOC, LOAD, DA
🛐 lscript.ld			8 .fini_array	00000004	0010c004 00100
README.txt			0 ADM attack	CONTENTS,	ALLOC, LOAD, DA
📝 Xilinx.spec			9 .AKM.attribu	CONTENTS.	READONLY
			10 bcc	00000000	0010-000 0010

#### "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

#### 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, highspeed, 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.

ြဲ Project Explorer 🛛 🗖 🗖	audio_intr_test.elf ☎
🖻 😫 🔻 🗸	architecture: arm, flags 0x00000112: EXEC_P, HAS_SYMS, D_PAGED
🖉 😂 audio_intr_test	start address 0x00100000
Binaries	Program Header:
	LOAD off 0x00010000 vaddr 0x00100000 paddr 0x00100000 al.
P Debus	filesz 0x0000c008 memsz 0x000120a0 flags rwx
* 🔁 Debug	private flags = 5000400: [Version5 EABI] [hard-float ABI]
SIC	Sections:
standio intr_test.elf - [arm/le]	Idx Name Size VMA LMA File off Algn
audio intr test elf size	0.text 00006434 00100000 00100000 00010000 2**6
	CONTENTS, ALLOC, LOAD, READONLY, CODE
	1.1010 00000018 00100434 00100434 00010434 2**2 CONTENTS ALLOC LOAD READONLY CODE
lo objects.mk	2 fini 00000018 0010644c 0010644c 0001644c 2**2
🚡 sources.mk	CONTENTS, ALLOC, LOAD, READONLY, CODE
Xilinx spec	3 .rodata 00000554 00106468 00106468 00016468 2**3
a Annopee	CONTENTS, ALLOC, LOAD, READONLY, DATA
* 🗁 SFC	CONTENTS, ALLOC, LOAD, DATA
audio.c	5 .eh_frame 00000004 0010788c 0010788c 0001788c 2**2
audio.h	CONTENTS, ALLOC, LOAD, READONLY, DATA
main c	6 .mmu_tbl 00004000 00108000 00108000 00018000 2**0
	7 init array 00000004 0010000 0010000 0001000 2**2
Script.la	CONTENTS, ALLOC, LOAD, DATA
README.txt	8 .fini_array 00000004 0010c004 0010c004 0001c004 2**2
🕑 Xilinx.spec	CONTENTS, ALLOC, LOAD, DATA
audio test	9 AKM. ATTFIDUTES 00000033 0010C008 0010C008 0001C008 2**
C debug	10 .bss 00000890 0010c008 0010c008 0001c008 2**2
r 📨 debug	ALLOC
▶ ∰ system_bsp	11 .heap 00002008 0010c898 0010c898 0001c008 2**0 ALLOC
System_wishbei_iiw_bierioiiii_o	12 .stack 00003800 0010e8a0 0010e8a0 0001c008 2**0 ALLOC
	13 .comment 00000036 00000000 00000000 0001c03b 2**0
	CONTENTS, READONLY
	CONTENTS, READONLY, DEBUGGING
	15 .debug_abbrev 000004d7 00000000 00000000 0001d981 2**0
	CONTENTS, READONLY, DEBUGGING
	16 .debug_aranges 00000040 0000000 0000000 0001de58 2**0
	17 debug macro 00003bdf 00000000 0000000 0001de08 2**0
	CONTENTS, READONLY, DEBUGGING
	18 .debug_line 0000095f 00000000 00000000 00021a77 2**0
	CONTENTS, READONLY, DEBUGGING
	19 .debug_str 00010ea5 00000000 00000000 000223d6 2**0
	20 debug frame 000002b8 00000000 00000000 0003327c 2**2
	CONTENTS DEADONLY DEDUGGING

## 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 0x800000
- Unintialized data segment starts after the data segment.

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

### 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).

#### 🛐 lscript.ld 🛙

#### Linker Script: Iscript.Id

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	Add Memory
ps7_ddr_0	0x100000	0x1FF00000	
ps7_qspi_linear_0	0xFC000000	0x1000000	
ps7_ram_0	0x0	0x30000	
ps7_ram_1	0xFFFF0000	0xFE00	

#### 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 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.

```
S lscript.ld 🖾
Linker Script: Iscript.Id
  /* Define Memories in the system */
  MEMORY
     ps7 ddr 0 : ORIGIN = 0x100000, LENGTH = 0x1FF00000
     ps7 gspi 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.*)
     *(.qnu.linkonce.t.*)
     *(.plt)
     *(.gnu warning)
     *(.gcc execpt table)
     *(.glue 7)
     *(.glue 7t)
     *(.vfpll 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).