ARM Link

Young W. Lim

2021-12-08 Wed

Outline

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

TOC: GNU ELF Addresses

ELF object files

- the linker combines *input* files
  into a single *output* file
  - input object files
  - output object / executable file
  - all in object file format

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/l

ELF sections

- each object file has a list of sections
  - input sections
  - output sections
- each section in an object file has
  - a *name*
  - a *size*
  - section contents :
    most sections are associated with block of data

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/l

ELF section types

- a loadable section
  - the contents should be *loaded* into memory
    when the output file is *run*
- an allocatable section
  - a section with no contents may be *allocatable*
  - an area in memory should be *set aside*
    but nothing should be *loaded* there
  - in some cases, this memory must be filled with /zero/es
- sections for debugging

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/l

ELF sections

- section: tell the linker if a section is either:
  - raw data to be loaded into memory,
    - e.g. `.data`, `.text`, etc.
  - formatted metadata about other sections,
    that will be used by the linker,
    but disappear at runtime
    - e.g. `.symtab`, `.srttab`, `.rela.text`

`https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s`

ELF segments

- segment: tells the operating system:
  - *where* should a segment be loaded into virtual memory
  - *what permissions* the segments have (read, write, execute).
    - this can be efficiently enforced by the *processor*

`https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-s`

ELF views of the image at each link stage

- ELF Object file view (Linker input)
- Linker view
- ELF Image file view (Linker output)

https://www.keil.com/support/man/docs/armlink/armlink_pge1406297322750.htm

ELF object file view (linker input)

- The ELF object file view comprises input sections
- The ELF object file can be:
  - A relocatable file
    that holds code and data suitable
    - for linking with other object files
      to create an executable or a shared object file.
  - A shared object file
    that holds code and data.

https://www.keil.com/support/man/docs/armlink/armlink_pge1406297322750.htm

Linker view (1)

- The linker has two views for the address space of a program
  - The load address of a program fragment
  - The execution address of a program fragment

https://www.keil.com/support/man/docs/armlink/armlink_pge1406297322750.htm

Linker view (2)

- The load and execution addresses become *distinct*
  in the presence of the following program fragments (code or data)
  - overlaid
  - position-independent
  - relocatable
- if a fragment is *position-independent* or *relocatable*
  its *execution address* can *vary during execution*

`https://www.keil.com/support/man/docs/armlink/armlink_pge1406297322750.htm`

Linker view (3)

- The load address of a program fragment
  - the target address that the linker expects
    an external agent to *copy* the fragment from the ELF file.
    - such as a program loader, dynamic linker, or debugger
  - this might not be the address at which the fragment *executes*
- The execution address of a program fragment
  - the target address where the linker expects
    the fragment to *reside* whenever it participates
    in the *execution* of the program.

`https://www.keil.com/support/man/docs/armlink/armlink_pge1406297322750.htm`

ELF image file view (linker output)

- The ELF image file view comprises
  program segments and output sections:
  - A load region corresponds to a program segment
  - An execution region contains
    one or more of the following output sections
    - RO section.
    - RW section.
    - XO section.
    - ZI section.
- One or more execution regions make up a load region

`https://www.keil.com/support/man/docs/armlink/armlink_pge1406297322750.htm`

<Input/output sections, regions, and program segments>

- Input sections
  - RO, RW, XO, ZI attributes
- Output sections
  - a group of input sections with the same attributes
- Regions
  - upto three output sections
    - RO-RW-ZI
    - XO-RW-ZI
- Program segments

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm`

Input section

- an individual section from an input object file
- contains code, initialized data,
- or describes a fragment of memory
  - that is not initialized or
  - that must be set to zero before the image can execute.
- These properties are represented by attributes
  such as RO, RW, XO, and ZI.
- These attributes are used by armlink to group input sections
  into bigger building blocks called output sections and regions

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm

Output section

- a group of input sections
  - that have the same RO, RW, XO, or ZI attribute,
  - that are placed contiguously in memory by the linker.
- an output section has the same attributes
  as its constituent input sections
- within an output section, the input sections are
  sorted according to the section placement rules

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm`

Region (1)

- contains up to three output sections
  depending on the contents and the number of sections
  with different attributes
- By default, the output sections in a region
  are sorted according to their attributes:
- A region typically maps onto a physical memory device,
  such as ROM, RAM, or peripheral.
- You can change the order of output sections
  using scatter-loading

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm`

Region (2)

- If no XO output sections are present, (RO - RW - ZI)
  then the RO output section is placed first,
  followed by the RW output section,
  and finally the ZI output section.
- If all code in the execution region is execute-only, (XO - RW - ZI)
  then an XO output section is placed first,
  followed by the RW output section,
  and finally the ZI output section.

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm

Program segment

- a program segment
  - corresponds to a load region
  - contains execution regions
- program segments hold information
  such as text and data

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm

Relationship between sectons, regions, and segments

ELF object file view	Linker view	ELF image file view
Program Header Table (optional)	Program Header Table	Program Header Table
input section 1.1.1 input section 1.1.2 input section 1.2.1 input section 1.3.1 input section 1.3.2	Load Region 1 [Exec Region 1]	Segment 1 [Load Region 1] - output section 1.1 - output section 1.2 - output section 1.3
input section 2.1.1 input section 2.1.2 input section 2.1.3	Load Region 2 [Exec Region 2]	Segment 2 [Load Region 2] - output section 2.1
* * *	* * *	* * *
Section Header Table	Section Header Table (optional)	Section Header Table (optional)

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm

<Load view and execution view of an image>

- The memory map of an image has distinct views:
  - load view
  - execution view

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm

image regions at load time and during execution

- image regions are *placed* in the system memory map at load time.
- the location of the regions in memory might *change* during *execution*

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm`

ROM address and RAM address

- in order to execute the image,
  some of image regions must be moved
  to their execution addresses and
  create the ZI output sections
- for example, *initialized RW data*
  might have to be copied
  from its load address in ROM
  to its execution address in RAM.

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm`

Load view and execution view

- load view
  Describes each image region and section in terms of the address where it is located when the image is loaded into memory, that is, the location before image execution starts.
- execution view
  Describes each image region and section in terms of the address where it is located during image execution.

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm`

Load address, load region

- load address
  - the address where a section or region is loaded into memory before the image containing it starts executing.
  - the load address of a section or a non-root region can differ from its execution address.
- load region
  - describes the layout of a contiguous chunk of memory in load address space

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm`

Execution address, execution region

- execution address
  - the address where a section or region is located while the image containing it is being executed
- execution region
  - describes the layout of a contiguous chunk of memory in execution address space

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm

Execution Only Memory (XOM)

- a firmware protection technique
  to help *prevent* 3rd parties from *stealing* or
  reverse engineering firmware
  and at the same time allowing 3rd parties
  to add *additional software* to the chips and
  utilize the protected APIs in XOM.
- This technique is different from
  chip-level, read-out-protection,
  which can block the read-back
  of the entire firmware.

`https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/`

Memory map without XO region

	Load view	0x0FFFF	Exec view
RAM		0x0A000	ZI section
RAM		0x08000	RW section
ROM	RW section	0x06000	
ROM	RO section	0x00000	RO section

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm

Memory map with XO region

	Load view	0x0FFFF	Exec view
RAM		0x0A000	ZI section
RAM	RW section	0x08000	RW section
ROM	RO section	0x06000	RO section
XOM	XO section	0x00000	XO section

https://www.keil.com/support/man/docs/armlink/armlink_pge1362065902090.htm

Type 1 image

- one load region and contiguous execution regions
- This approach is suitable for systems that load programs into RAM, for example, an OS bootloader or a desktop system.
- Load view
  The single load region consists of the RO and RW output sections, placed consecutively. The RO and RW execution regions are both root regions. The ZI output section does not exist at load time. It is created before execution, using the output section description in the image file.
- Execution view
  The three execution regions containing the RO, RW, and ZI output sections are arranged contiguously. The execution addresses of the RO and RW execution regions are the same as their load addresses, so nothing has to be moved from its load address to its execution address. However, the ZI execution region that contains the ZI output section is created at run-time.

https://developer.arm.com/documentation/dui0474/h/image-structure-and-generation/t

Type 2 image

- one load region and non-contiguous execution regions
- for ROM-based embedded systems, where RW data is copied from ROM to RAM at startup:
- Load view
  in the load view, the single load region consists of the RO and RW output sections placed consecutively, for example, in ROM. Here, the RO region is a root region, and the RW region is non-root. The ZI output section does not exist at load time. It is created at runtime.
- Execution view
  In the execution view, the first execution region contains the RO output section and the second execution region contains the RW and ZI output sections.

`https://developer.arm.com/documentation/dui0474/h/image-structure-and-generation/t`

Type 3 image

- two load regions and non-contiguous execution regions
- A Type 3 image is similar to a Type 2 image except that the single load region is split into two root load regions
- Load view
  In the load view, the first load region consists of the RO output section, and the second load region consists of the RW output section. The ZI output section does not exist at load time. It is created before execution, using the description of the output section contained in the image file
- Execution view
  In the execution view, the first execution region contains the RO output section, and the second execution region contains the RW and ZI output sections.

https://developer.arm.com/documentation/dui0474/h/image-structure-and-generation/t

Scatter Loading (1)

- enables you to specify the memory map of an image to the linker using a description in a text file.
- gives you complete control over the grouping and placement of image components.
- generally, for images with a complex memory map
  - multiple memory regions are *scattered* in the memory map at load and execution time

`https://developer.arm.com/documentation/dui0474/f/using-scatter-files/about-scatte`

Scatter Loading (2)

- an image memory map is made up of regions and output sections
- every region in the memory map can have a different load and execution address

https://developer.arm.com/documentation/dui0474/f/using-scatter-files/about-scatte

Scatter Loading (3)

- to construct the memory map of an image, the linker must have:
  - grouping information
    describes how input sections are grouped
    into output sections and regions
  - placement information
    describes the addresses where regions
    are to be located in the memory maps.

https://developer.arm.com/documentation/dui0474/f/using-scatter-files/about-scatte

Scatter Loading (4)

- When the linker creates an image using a scatter file, it creates some region-related symbols.
- The linker creates these special symbols only if your code references them.

https://developer.arm.com/documentation/dui0474/f/using-scatter-files/about-scatte

Default section placement (1)

- By default, the linker places input sections in a specific order within an execution region.
- The sections are placed in the following order:
  - By attribute
  - By input section name if they have the *same* attributes.
  - By a tie-breaker if they have the *same* attributes and section names.

`https://www.keil.com/support/man/docs/armclang_ref/armclang_ref_Chunk1932994948.ht`

Default section placement (2)

- By default, it is the order that armlink processes the section.
  - By attribute as follows:
    - Read-only (RO) code
    - Read-only (RO) data.
    - Read-write (RW) code.
    - Read-write (RW) data.
    - Zero-initialized (ZI) data.
  - By input section name if they have the same attributes.
  - By a tie-breaker if they have the same attributes and section names.
- override the tie-breaker and sorting by input section name with the `FIRST` or `LAST` input section attribute.

`https://www.keil.com/support/man/docs/armclang_ref/armclang_ref_Chunk1932994948.ht`

Default section placement (3)

- the positions of input sections with *identical* attributes and names included from libraries depend on the order the linker processes objects.
- difficult to predict when many libraries are present on the command line.
  The `--tiebreaker=cmdline` option uses a more predictable order based on the order the section appears on the command line.

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm`

Default section placement (4)

- The base address of each input section is determined
  by the *sorting order* defined by the linker,
  and is *correctly aligned* within the output section that contains it.
- The linker produces *one* output section for *each* attribute
  present in the execution region:
  - One XO section if the execution region contains only XO sections.
  - One RO section if the execution region contains read-only code or data.
  - One RW section if the execution region contains read-write code or data.
  - One ZI section if the execution region contains zero-initialized data.

`https://www.keil.com/support/man/docs/armlink/armlink_pge1362065900278.htm`

p_vaddr and p_paddr (1)

- p_vaddr is a virtual address,
  p_paddr is a physical address.
- these are the addresses at which
  the *data in the file* will be loaded.
- they map the contents of the file
  into their corresponding memory locations

https://stackoverflow.com/questions/16812574/elf-files-what-is-a-section-and-why-c

p_vaddr and p_paddr (2)

- physical addresses are the raw memory addresses.
  - on modern operating systems,
    physical addresses are no longer used in the user space
    Instead, user space programs use virtual addresses.
- the OS deceives that
  - the user space program uses the memory alone,
  - the entire address space is available for it.
- the OS maps those virtual addresses
  to physical addresses in the actual memory,
  and it does it transparently to the program.

https://stackoverflow.com/questions/16812574/elf-files-what-is-a-section-and-why-c

p_vaddr and p_paddr (3)

- not every address in the virtual address space
  is available simultaneously
- limited by the actual physical memory available.
- the OS just maps the memory for the segments
  the program actually uses
- if the process tries to access some unmapped memory,
  the operating system incurs memory access fault
  (The program can address it, but it cannot access it)

https://stackoverflow.com/questions/16812574/elf-files-what-is-a-section-and-why-c

Base address (1)

- executable and shared object files
  have a base address :
  - the lowest virtual address associated
    with the memory image of the program's object file.
- to relocate the memory image of the program
  during dynamic linking

`https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html`

Base address (2)

- an executable or shared object file's
  base address is calculated during execution
  from three values:
  - the memory load address
  - the maximum page size
  - the lowest virtual address of a program's loadable segment

  - the virtual addresses in the program headers
    might not represent the actual virtual addresses
    of the program's memory image

`https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html`

Base address (3)

- to compute the base address
  of an executable or shared object file
  you determine the memory addreses associated with
  the lowest `p_vaddr` value
  for a `PT_LOAD` segment.
- then obtain the base address
  by *truncating* the memory address
  to the nearest multiple of the maximum page size.
- depending on the kind of file being loaded into memory,
  the memory address might not match the `p_vaddr` values.

`https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html`

A loadable segment (1)

- PT_LOAD specifies a loadable segment,
  described by p_filesz and p_memsz
- the bytes from the file are mapped
  to the beginning of the memory segment.
- loadable segment entries in the program header table

appear in ascending order, sorted on the p_vaddr member.

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html

A loadable segment (2)

- if the segment's memory size is larger than the file size (`p_memsz > p_filesz`),
  the extra bytes are defined to hold the value 0 and
  to follow the segment's initialized area
- the file size cannot be larger than the memory size
  (`p_memsz < p_filesz` not possible case)

`https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html`

Load and run addresses (1)

- The load address is the location of an object in the load image
- The run address is the location of the object as it exists during program execution
- An object is a chunk of memory. It represents a section, segment, function, or data.

https://downloads.ti.com/docs/esd/SPRU513/load-and-run-addresses-slau1317366.html

Load and run addresses (2)

- The load and run addresses for an object may be the same
  - This is commonly the case for program code and read-only data, such as the `.econst` section.
    - the program can *read the data* directly from the load address
  - sections that have no initial value, such as the `.ebss` section
    - do not have load data
    - considered to have the same load and run addresses
    - if you specify different load and run addresses for an uninitialized section, the linker provides a warning and ignores the load address.

`https://downloads.ti.com/docs/esd/SPRU513/load-and-run-addresses-slau1317366.html`

Load and run addresses (3)

- The load and run addresses for an object may be different.
  - This is commonly the case for writable data,
    such as the `.data` section.
  - The `.data` section's starting contents are placed in ROM
    and *copied to* RAM.
  - This often occurs during program startup,
    but depending on the needs of the object,
    it may be deferred to sometime later in the program

`https://downloads.ti.com/docs/esd/SPRU513/load-and-run-addresses-slau1317366.html`

LMA & VMA (1)

- every loadable or allocatable output section has two addresses.
- the VMA (Virtual Memory Address)
  - the address the *output section* will have
    when the output file is run
- the LMA (Load Memory Address)
  - the address at which the *output section* will be loaded

`https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/l`

LMA & VMA (2)

- in most cases, VMA and LMA will be the same
- VMA and LMA might be different
  when a *data section* is loaded from ROM,
  and then copied into RAM when the program starts up
  - this technique is often used to initialize global variables
    in a ROM based system
  - in this case the ROM address would be the LMA
    and the RAM address would be the VMA

https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/binutils/docs/ld/l

LMA & VMA (3)

- The section header contains a single address.
- the address in the section header is the VMA
- The program headers contain the mapping of VMA to LMA
- `objdump -x`

`https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect`

LMA & VMA (4)

ELF section header example

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
<a few lines removed>
  3 .bss          00000004  00000048  0000018c  00000240  2**1
                  ALLOC
```

- .bss has a VMA 0x048
- .bss has a LMA 0x18c

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (5)

ELF program header example

```
Program Header:
<a few lines removed>
    LOAD off    0x00000240 vaddr 0x00000048 paddr 0x0000018c align 2**0
         filesz 0x00000000 memsz 0x00000004 flags rw-
```

- a vaddr of 0x048 (VMA)
- a paddr of 0x18c (LMA)

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (6)

- ELF file segment does have the physical address attribute
  ELF file section does not have physical address attribute.
- It is possible though to map sections
  to corresponding segment memory.
- The meaning of physical address is architecture dependent
  and may vary between different OS's and hardware platforms.

https://stackoverflow.com/questions/6218384/virtual-and-physical-addresses-of-sect

LMA & VMA (7)

- VMA and LMA are GNU utility terminology
  not in the ELF specification.
- an ELF executable file has program header fields :
  - p_paddr
  - p_vaddr

`https://stackoverflow.com/questions/39888381/elf-loading-when-vma-lma`

LMA & VMA (8)

p_vaddr

- this member gives the virtual address
  at which the first byte of the segment resides in memory

p_paddr

- on systems for which physical addressing is relevant,
  this member is reserved for the segment's physical address
- because System V ignores physical addressing
  for application programs,
  this member has unspecified contents
  for executable files and shared objects.

`https://refspecs.linuxbase.org/elf/gabi4+/ch5.pheader.html`

LMA & VMA (9)

- by default, ARM IDE DS-5 uses p_vaddr, which is the standard
- Usage of p_paddr is a quality of implementation,
  and is left very loosely defined in the specification.
- The ARM Compiler, Linker and C Library
  does not generate this information (p_vaddr, p_paddr)
  since the relocation process is handled internally
  (scatter loading).

https://stackoverflow.com/questions/39888381/elf-loading-when-vma-lma

LMA & VMA (10)

- some environments use p_paddr
  - not as a physical address,
  - but the load address (hence LMA),
- some use p_paddr
  - as an address to resolve symbols
    before and after MMU is enabled

https://stackoverflow.com/questions/39888381/elf-loading-when-vma-lma