

Linker

el linker script no mueve código, sino que indica a qué posición de memoria mirar, es decir, si hacemos referencia a x1, el ld indicará donde está ubicada.

Linker Script



Planteo del problema

Se desea realizar un programa que modifique un área de 4MB de memoria no inicializada a '0xFF', sabiendo que el sistema solo cuenta con una ROM de 1kB y una RAM de 8MB.

Tenga en cuenta que el sistema no dispone de ningún sistema operativo.

Plantéese:

- ✓ ¿Donde se encuentra el programa durante la ejecución?
- ✓ ¿Cómo llegó a la ubicación de la respuesta anterior?
- ✓ ¿Qué tamaño, aunque más no sea basándose en el tamaño de los datos, estima que tendría el programa?



Primer propuesta

```
#define MEM_4MB 4*1024*1024
```

```
char mem_ptr[MEM_4MB] = {0xFF, .....};
```

```
int main(void) {
    mem_ptr = 0x0000;
    .....
    .....
}
```

Primer propuesta

```
#define MEM_4MB 4*1024*1024
```

```
char mem_ptr[MEM_4MB] = {0xFF, .....};
```

```
int main(void) {
    mem_ptr = 0x0000;
    .....
    .....
}
```

Resulta evidente que el binario generado ocupará al menos 4MBytes, por lo cual no cabrá en la ROM del sistema.
A parte de que no compila ya que no se puede ubicar un puntero de esa manera.

Segunda propuesta

```
char mem_ptr[];  
  
int main(void) {  
    unsigned int i;  
    for (i=0; i<MEM_4MB; i++) {  
        mem_ptr[i] = 0xFF;  
    }  
}
```

Segunda propuesta

```
char mem_ptr[];
```

```
int main(void) {
```

```
    unsigned int i;
```

```
    for (i=0; i<MEM_4MB; i++) {
        mem_ptr[i] = 0xFF;
    }
```

```
}
```

- Ahora el código es mas compacto
- Pero el código queda ubicado en cualquier lugar, no en el espacio esperado
- La RAM queda ubicada en el espacio de ROM

Volátil	oxoooooooo	Código
	oxoooooooo2F	
	oxoooooooo30	Datos
	ox00400030	
ROM	

La respuesta a este conflicto es el LINKER

Algunas definiciones

Símbolo: El programador usa símbolos para nombrar cosas, el linker los utiliza para referenciar, representan variables, direcciones, constantes

Módulo: Es una porción de programa autocontenido en un archivo. Los programas están compuestos por módulos desarrollados independientemente que no son combinados hasta que el programa es linkeado.

Tareas del compilador

- ✓ **Preprocesamiento (directivas)**
- ✓ **Parsing**
- ✓ **Análisis léxico**
- ✓ **Análisis sintáctico**
- ✓ **Conversión a representaciones intermedias**
- ✓ **Optimización de código** detecta cualquier error en el código y lo optimiza
- ✓ **Generación de código**

de aca sale el archivo objeto

para generar mi ejecutable
voy a tener el compilador y
el linker.

Tareas del linker (enlazador)

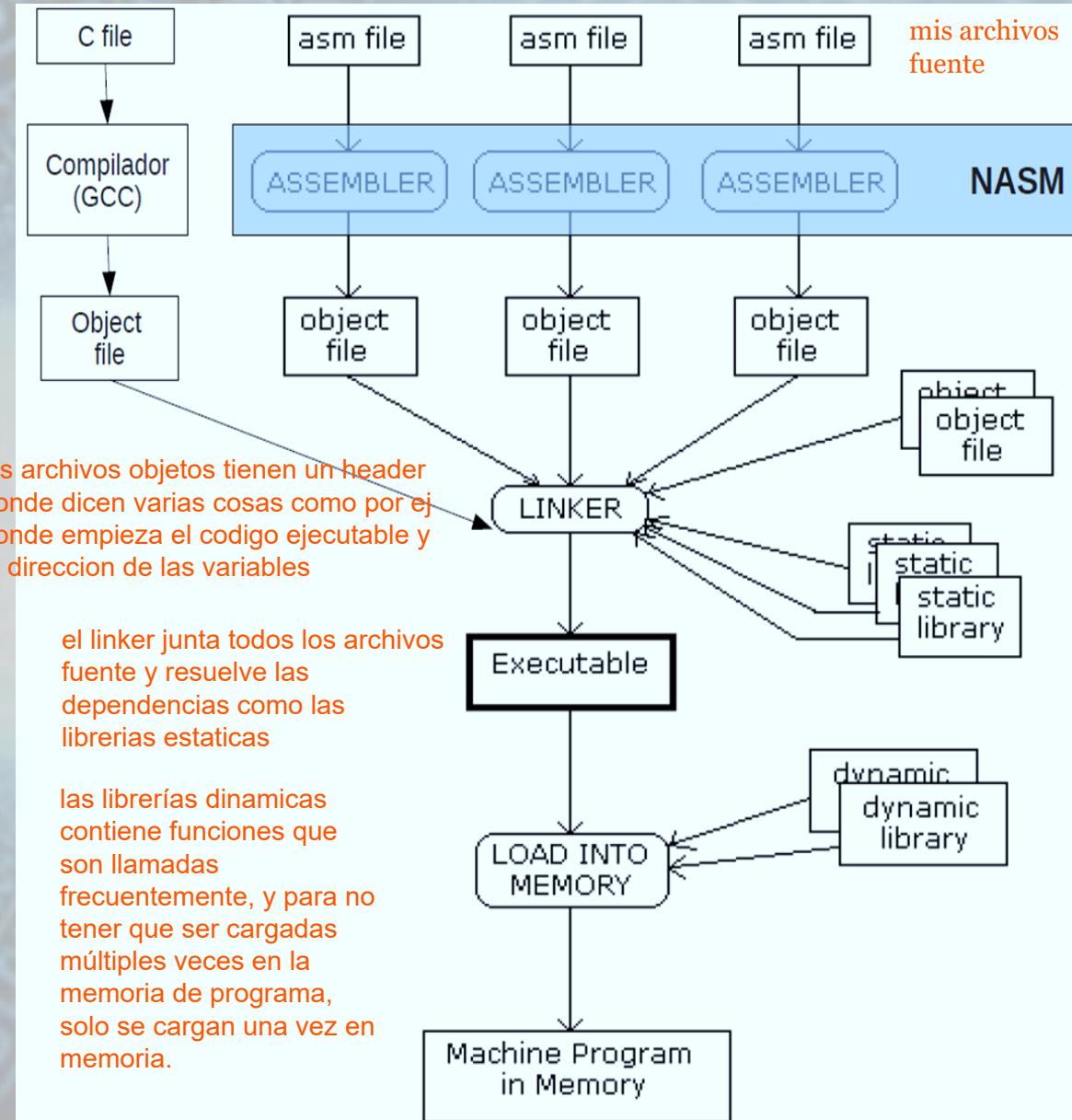
este linkea las librerías estandar que nosotros usamos pero no sabemos que hay dentro.

- ✓ **Toma los objetos generados por el Compilador**
- ✓ **Toma otros recursos necesarios (bibliotecas externas)**
- ✓ **Quita lo que no necesita (ej. libc)**
- ✓ **Produce un archivo ejecutable o biblioteca final (salvo carga dinámica)**

Misión del Linker

El enlazador tiene por objetivo combinar varios fragmentos de código y datos en un único archivo binario.

En base a ello mantiene las referencias relativas entre todos los símbolos de todos los módulos que componen el proyecto.



Dependencia entre símbolos

Módulo 1 (ASM)

```
var1: db 04h
.....
call print
.....
call scanf
.....
print:
.....
```

Módulo 2 (ASM)

```
.....
push var1
call print
mov [i], eax
.....
```

Módulo 3 (C)

```
#include <stdio.h>

int i;

void main (void) {
    i = 0;
    print();
    open(.....);
}
```

Los modulos tienen que estar interconectados, sino no tiene sentido linkear los archivos

Dependencia entre símbolos - Falla

Módulo 1 (ASM)

```
var1: db 04h  
....  
call print  
....  
call scanf  
....  
print:  
....
```

Módulo 2 (ASM)

```
....  
push var1  
call print  
mov [dato], eax  
....
```

Módulo 3 (C)

```
#include <stdio.h>  
  
int dato;  
  
void main (void) {  
    dato = 0;  
    print();  
    open(...); ?  
}
```

Dependencia entre símbolos externos

Módulo 1 (ASM)

```
global var1
global print
extern scanf
```

con estas
directivas hago
que el compilador
sepa que algunas
dependencias no
estan en el
archivo

```
var1: db 04h
.....
call print
.....
call scanf
.....
print:
```

Módulo 2 (ASM)

```
extern var1
extern print
extern i
....
push var1
call print
mov [i], eax
....
```

Módulo 3 (C)

```
#include <stdio.h>
extern void print(char*);
int i;

void main (void) {
    i = 0;
    print();
    open(.....);
}
```

open() {
....
....
}

Biblioteca estática (.a o .lib)

Misión del Linker o «Enlazador»

Debemos poder ubicar cada bloque de código donde queramos tanto en el binario como en ejecución ???

El binario de salida puede presentar la misma organización de la memoria en la cual se carga, por ejemplo una ROM, o diferir «virtualmente» (en tiempo de ejecución) como es el caso de requerir RAM.

¿Qué sucedía hasta ahora? -> Lo resolvía el linker a su criterio

¿Y ahora? -> Le vamos a especificar al linker el «lay-out» que queremos del binario y de la memoria

¿Cómo? -> mediante un script denominado «linker script»

tenemos 2 tipos de direcciones

LMA : Load Memory Address

Dónde quedarán ubicados los datos en el binario y en el medio físico

VMA : Virtual Memory Address

Cual es la ubicación lineal en memoria al momento de su ejecución

➤ El linker especificará las referencias en el código en base a ésta

un uso comun de esto es cuando tengo que escribir direcciones de memoria y estas estan en ROM. Tengo que moverlas a RAM. O tal vez por un tema de timing (la RAM es mas rapida) necesito ejecutar en RAM (se lo llama requerimiento no funcional).

Dirección física

Donde deben estar ubicados los datos realmente al momento de su ejecución

los perifericos van a tener una direccion fisica y nosotros vamos a darles una direccion virtual que se obtiene a partir de una conversion|

Tipos y atributos:

- Reubicables o No reubicables
- progbits : *se almacena en la imagen de disco.*
- nobits : *se ubica e inicializa en la carga (opuesto a progbits)*
- Etc

8.1. RAM is Volatile!

RAM is volatile memory, and hence it is not possible to directly make the data available in RAM, on power up.

All code and data **should** be stored in Flash before power-up. On power-up, a startup code is supposed to copy the data from Flash to RAM, and then proceed with the execution of the program. So the program's `.data` section has two addresses, a **load address** in Flash and a **run-time address** in RAM.



Tip

In `ld` parlance, the load address is called LMA (Load Memory Address), and the run-time address is called VMA (Virtual Memory Address.).

The following two modifications have to be done, to make the program work correctly.

1. The linker script has to be modified to specify both the load address and the run-time address, for the `.data` section.
2. A small piece of code should copy the `.data` section from Flash (load address) to RAM (run-time address).

8.2. Specifying Load Address

The run-time address is what that should be used for determining the address of labels. In the previous linker script, we have specified the run-time address for the `.data` section. The load address is not explicitly specified, and defaults to the run-time address. This is OK, with the previous examples, since the programs were executed directly from Flash. But, if data is to be placed in RAM during execution, the load address should correspond to Flash and the run-time address should correspond to RAM.

A load address different from the run-time address can be specified using the `AT` keyword. The modified linker script is shown below.

```
SECTIONS {
    . = 0x00000000;
    .text : { * (.text); }
    etext = .; ①

    . = 0xA0000000;
    .data : AT (etext) { * (.data); } ②
}
```

- ❶ Symbols can be created on the fly within the `SECTIONS` command by assigning values to them. Here `etext` is assigned the value of the location counter at that position. `etext` contains the address of the next free location in Flash right after all the code. This will be used later on to specify where the `.data` section is to be placed in Flash. Note that `etext` itself will not be allocated any memory, it is just an entry in the symbol table.
- ❷ The `AT` keyword specifies the load address of the `.data` section. An address or symbol (whose value is a valid address) could be passed as argument to `AT`. Here the load address of `.data` is specified as the location right after all the code in Flash.

¿Cómo especificar esos bloques de código?

el linker manda los bloques de código a cualquier parte de la memoria.

Si yo quisiera tenerlo en una cierta parte de la memoria, debo identificar los bloques a través del término section

Se las llama secciones "section "

las secciones terminan al final del código o donde empieza otra.

Secciones por defecto:

.text	: Código ejecutable
.data	: Datos inicializados
.bss	: Datos no inicializados
.rodata	: Datos constantes
.eh_frame	: Stack de C

Otras	: .eeprom, .init, .finit, etc
-------	-------------------------------

- Puedo crear mis propias secciones
- Notar que el binario contiene datos al comienzo, sin embargo el entry point se especifica con _start (por defecto en ASM) o main() (por defecto en C)

después de linkear también voy a tener secciones, las "secciones de salida"

```
section .data
msg: db 'Hola mundo' , 0

section .text
_start:
    push [msg]
    call mi_print
    ...
    ...

section .mi_code
mi_print:
    push ebp
    mov  esp, ebp
    mov  eax, [ebp-4]
    ...
    ...
```

6.2. Relocation

Relocation is the process of changing addresses already assigned to labels. This will also involve patching up all label references to reflect the newly assigned address. Primarily, relocation is performed for the following two reasons:

1. Section Merging
2. Section Placement

To understand the process of relocation, an understanding of the concept of sections is essential.

Code and data have different run time requirements. For example code can be placed in read-only memory, and data might require read-write memory. It would be convenient, if code and data is **not** interleaved. For this purpose, programs are divided into sections. Most programs have at least two sections, `.text` for code and `.data` for data. Assembler directives `.text` and `.data`, are used to switch back and forth between the two sections.

It helps to imagine each section as a bucket. When the assembler hits a section directive, it puts the code/data following the directive in the selected bucket. Thus the code/data that belong to particular section appear in contiguous locations. The following figures show how the assembler re-arranges data into sections.

Figure 3. Sections

```
.data
arr: .word 10, 20, 30, 40, 50
len: .word 5
.text
start: mov r1, #10
        mov r2, #20
.data
result: .skip 4
.text
        add r3, r2, r1
        sub r3, r2, r1
```

.data section

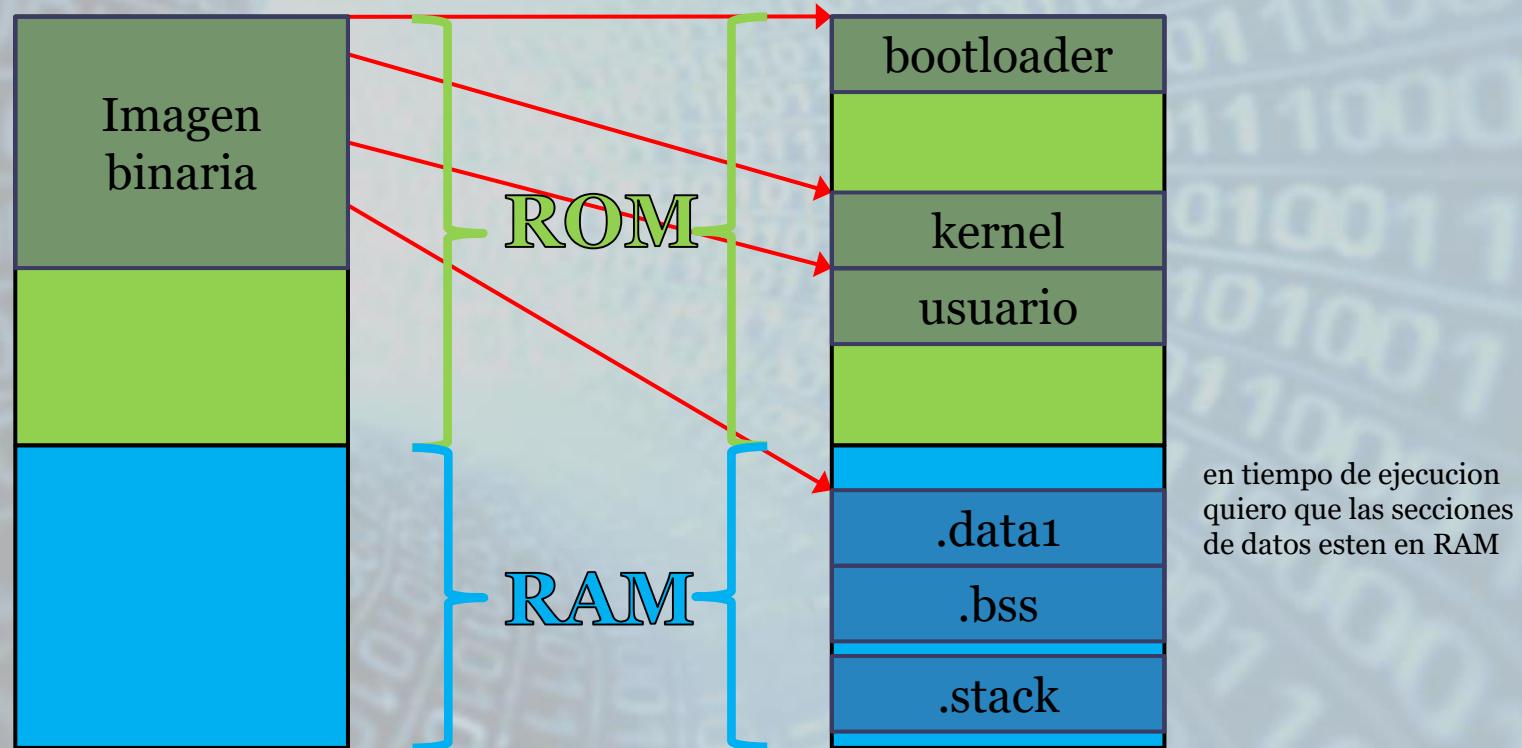
```
0000_0000 arr: .word 10, 20, 30, 40, 50
0000_0014 len: .word 5
0000_0018 result: .skip 4
```

.text section

```
0000_0000 start: mov r1, #10
0000_0004
0000_0008
0000_000C        mov r2, #20
                    add r3, r2, r1
                    sub r3, r2, r1
```

No entiendo, ¿si los datos están compactados en un binario, inclusive ahí se encuentran las variables globales?:

¿Cómo y quién reubica ese código y mueve las variables a zona de RAM?



¿Quién ubica el código en memoria?

- Trabajando sobre un sistema operativo lo hace el «loader» o alguna rutina interna al mismo
- En nuestro caso -> **NOSOTROS**

```
int no_se_oooo_llegue(void)  
{  
    // escriba su código aquí  
}
```

Modo amo de casa



¿Todo esto pasa porque trabajo con un procesador complicado?



NOOOO



No solo se usa en una PC, todos los linker lo hacen, en forma automática o explícitamente (donde logramos toda su potencia).

Hasta un PIC de 8 bits tiene su linker y linker script

Ejemplo práctico de hardware

En este link esta mas claro
(Ver parte 8 tambien)



Originalmente teníamos el primer banco ROM/RAM.

Luego expandimos la memoria al segundo bloque dejándola alineada.

las variables inicializadas deben estar en ROM y luego tambien en RAM para cuando tenga que utilizarla. Si no esta inicializada va a RAM

```
section .text
_start:
.....
.....
section .data
i: TIMES db 1024
section .subs
print:
.....
clear:
.....
memcpy:
section .data
buffer: db 0x24
```

SECTIONS

{ esto me genera las secciones de salida

.text : este es el nombre de una de las secciones de salida
{ *(.text); }

. = 0x00002000;

.data :

AT (LOADADDR(.text) + SIZEOF(.text))
{ *(.data*); }

. = 0x00040000;

.codigo_adicional :

AT (LOADADDR(.data) + SIZEOF(.data))
{ *(.subs); }

Tipos	Dirección	Obs.
ROM	00000000h-00001FFFh	8kB
RAM	00002000h-000031FFFh	192kB
.....	000032000h-0003Foooh	Nada
ROM	00040000h-000403FFF	16kB
RAM	000404000h-000503FFFh	1MB

Existen formas de generar mapas de memoria
(Ver «*The GNU Linker*»)

MEMORY

REGION

una cosa es el código que yo escribo (bl func() ---> 0x80
func : ---> 0x86

están una instrucción debajo de la otra.

Yo puedo querer que la dirección donde se encuentre func sea en la RAM

puedo tener secciones duplicadas (en este caso .data)

¿Cómo lo invoco?

ld <opciones> objeto_entrada_1 objeto_entrada_n -o archivo_salida -T linker_script

si no agrego esto el linker va a ubicar las secciones donde quiera

ld -m32 hola.o aux.o -o hola -T mi_archivo.lds

Ejemplo sin linker script

.asm	.lst	.bin
1 bits 32	1	bits 32
2	2	
3 global Inicio	3	global Inicio
4	4	
5 dato: dw 0x1234	5 00000000 3412	dato: dw 0x1234
6	6	
7 Inicio:	7	Inicio:
8 xor edi, edi	8 00000002 31FF	xor edi, edi
9 mov dword edi, 0x01	9 00000004 BF01000000	bf 01 00 00 00
10 xor esi, esi	10 00000009 31F6	31 f6
11 mov dword esi, 0x03	11 0000000B BE03000000	be 03 00 00 00
12 mov dword [variable], 0x04	12 00000010 C705[2E000000]0400-	c7 05 2e 80 00
13	13 00000018 0000	00 04 00 00 00
14 xor eax, eax	14 0000001A 31C0	31 c0
15 add eax, edi	15 0000001C 01F8	01 f8
16 add eax, esi	16 0000001E 01F0	01 f0
17 add dword eax, [variable]	17 00000020 0305[2E000000]	03 05 2e 80 00 00
18 mov dword [resultado], eax	18 00000026 A3[2C000000]	a3 2c 80 00 00
19 hlt	19 0000002B F4	f4
20	20	
21 resultado: dw 0x9876	21 0000002C 7698	76 98
22 variable: resd 1	22 0000002E <res 00000004>	00 00 00 00

REGIONS

Sección	Variante A	Variante B	Variante C
.text	RAM	ROM	ROM
.rodata	RAM	ROM	ROM2
.data	RAM	RAM/ROM	RAM/ROM2
.bss	RAM	RAM	RAM

REGIONS

con estos archivos generamos regiones de memoria donde ubicar las secciones y luego el linker interpreta donde mover cada bloque. Es un punto intermedio entre que lo haga todo el linker y que nosotros le digamos específicamente donde ubicar las secciones

INCLUDE linkcmds.memory en este archivo van a estar definidas las regiones de memoria

SECTIONS

```
{  
    .text :  
    {  
        *(.text)  
    } > REGION_TEXT  
    .rodata :  
    {  
        *(.rodata)  
        rodata_end = .;  
    } > REGION_RODATA  
    .data : AT (rodata_end)  
    {  
        data_start = .;  
        *(.data)  
    } > REGION_DATA  
    data_size = SIZEOF(.data);  
    data_load_start = LOADADDR(.data);  
    .bss :  
    {  
        *(.bss)  
    } > REGION_BSS  
}
```

Definidas con
«MEMORY»

REGIONS

```
INCLUDE linkcmds.memory
```



```
SECTIONS
```

```
{  
    .text :  
    {  
        *(.text)  
    } > REGION_TEXT  
    .rodata :  
    {  
        *(.rodata)  
        rodata_end = .;  
    } > REGION_RODATA  
    .data : AT (rodata_end)  
    {  
        data_start = .;  
        *(.data)  
    } > REGION_DATA  
    data_size = SIZEOF(.data);  
    data_load_start = LOADADDR(.data);  
    .bss :  
    {  
        *(.bss)  
    } > REGION_BSS  
}
```

al poner >LUGAR_DE_MEMORIA lo que hago es que el linker ubique la sección en el lugar de memoria indicado, sin necesidad de indicar exactamente la posición de memoria. Se ubican en el primer lugar vacío en memoria que se encuentra, no dejando huecos vacíos.

estos lugares de memoria se indican con la directiva MEMORY

REGIONS

Archivo “*linkcmds.memory*” variante A:

```
MEMORY
{
    RAM [(attr)] : ORIGIN = 0, LENGTH = 4M
}
```

```
REGION_ALIAS("REGION_TEXT", RAM);
REGION_ALIAS("REGION_RODATA", RAM);
REGION_ALIAS("REGION_DATA", RAM); REGION_ALIAS("REGION_BSS", RAM);
```

Syntax :

```
MEMORY
{
    . name (attr) : ORIGIN = origin, LENGTH = len
}
```

Defines name of the memory region which will be later referenced by other parts of the linker script

Defines origin address of the memory region

Defines the length information

attr:

- ‘R’ Read-only section
- ‘W’ Read/write section
- ‘X’ Executable section
- ‘A’ Allocatable section (debe reservarse, en gral. sin contenido e inicializada a cero)
- ‘I’ Initialized section
- ‘L’ Same as ‘I’
- ‘!’ Invert the sense of any of the preceding attributes

Loadable section (debe reubicarse en tiempo de ejecución)

REGIONS

Archivo “*linkcmds.memory*” variante B:

```
MEMORY
{
    ROM : ORIGIN = 0, LENGTH = 3M
    RAM : ORIGIN = 0x10000000, LENGTH = 1M
}

REGION_ALIAS("REGION_TEXT", ROM);
REGION_ALIAS("REGION_RODATA", ROM);
REGION_ALIAS("REGION_DATA", RAM);
REGION_ALIAS("REGION_BSS", RAM);
```

If an unmapped section matches any of the listed attributes other than ‘!’, it will be placed in the memory region. The ‘!’ attribute reverses the test for the characters that follow, so that an unmapped section will be placed in the memory region only if it does not match any of the attributes listed afterwards. Thus an attribute string of ‘RW!X’ will match any unmapped section that has either or both of the ‘R’ and ‘W’ attributes, but only as long as the section does not also have the ‘X’ attribute.

The origin is a numerical expression for the start address of the memory region. The expression must evaluate to a constant and it cannot involve any symbols. The keyword ORIGIN may be abbreviated to org or o (but not, for example, ORG).

The len is an expression for the size in bytes of the memory region. As with the origin expression, the expression must be numerical only and must evaluate to a constant. The keyword LENGTH may be abbreviated to len or l.

In the following example, we specify that there are two memory regions available for allocation: one starting at ‘0’ for 256 kilobytes, and the other starting at ‘0x40000000’ for four megabytes. The linker will place into the ‘rom’ memory region every section which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the ‘ram’ memory region.

```
MEMORY
{
    rom (rx) : ORIGIN = 0, LENGTH = 256K
    ram (!rx) : org = 0x40000000, l = 4M
}
```

REGIONS

Archivo “*linkcmds.memory*” variante C:

```
MEMORY
{
    ROM : ORIGIN = 0, LENGTH = 2M
    ROM2 : ORIGIN = 0x10000000, LENGTH = 1M
    RAM : ORIGIN = 0x20000000, LENGTH = 1M
}

REGION_ALIAS("REGION_TEXT", ROM);
REGION_ALIAS("REGION_RODATA", ROM2);
REGION_ALIAS("REGION_DATA", RAM);
REGION_ALIAS("REGION_BSS", RAM);
```

MAS DIRECTIVAS

ALIGN

```
.text ALIGN(0x10) : { *(.text) }
```

ESPECIFICACIÓN DE ARCHIVOS FUENTE

```
SECTIONS {
    outputa 0x10000 :
    {
        object1.o
        object2.o (.input1)
    }
    outputb :
    {
        object2.o (.input2)
        object3.o (.input1)
    }
    outputc :
    {
        *(.input1)
        *(.input2)
    }
}
```

Ejemplo con linker script

```
SECTIONS
{
    . = 0x08000;
    _codigo_inicio = .;
    .text : { *(.codigo_principal); }

    . = 0x09000;
    _datos_iniciali_inicio = .;
    .data : { *(.dat_inic*); }

    . = 0xA000;
    _datos_no_iniciali_inicio = .;
    .bss : { *(.dat_no_inic*); }

    . = 0xB000;
    .codigo_adicional : AT (ADDR(.text) + SIZEOF(.text)) { *(.codigo_funciones); }
}
```

Linker script

.asm	.lst	.bin
1 bits 32	1	bits 32
2	2	
3 global Inicio	3	global Inicio
4	4	
5 SECTION .codigo_principal	5 SECTION	.codigo_principal
6	6	
7 Inicio:	7	Inicio:
8 mov dword edi, 0x1	8 00000000 BF01000000	mov dword edi, 0x1
9 mov dword [parametro_a], edi	9 00000005 893D[04000000]	mov dword [parametro_a], edi
10 mov dword esi, 0x3	10 0000000B BE03000000	mov dword esi, 0x3
11 mov dword [parametro_b], esi	11 00000010 8935[06000000]	mov dword [parametro_b], esi
12 call far [_suma]	12 00000016 FF1D[00000000]	call far [_suma]
13 mov dword eax, [resultado]	13 0000001C A1[00000000]	mov dword eax, [resultado]
14 add dword eax, [variable_a]	14 00000021 0305[00000000]	add dword eax, [variable_a]
15 mov dword [array_a], eax	15 00000027 A3[08000000]	mov dword [array_a], eax
16 hlt	16 0000002C F4	hlt
17	17	
18 SECTION .dat_inic_a progbits	18	SECTION .dat_inic_a progbits
19 resultado: dw 0x9876	19 00000000 7698	resultado: dw 0x9876
20 texto_a: db "Hola"	20 00000002 486F6C61	texto_a: db "Hola"
21	21	
22 SECTION .dat_no_inic_a nobits1	22	SECTION .dat_no_inic_a nobits1
23 variable_a: resd 1	23 00000000 <res 00000004>	variable_a: resd 1
24 parametro_a: resw 1	24 00000004 <res 00000002>	parametro_a: resw 1
25 parametro_b: resw 1	25 00000006 <res 00000002>	parametro_b: resw 1
26 array_a: resb 4	26 00000008 <res 00000004>	array_a: resb 4
27	27	
28 SECTION .codigo_funciones	28 SECTION .codigo_funciones	_suma:
29 _suma:	29	push edi
30 push edi	30 00000000 57	push esi
31 push esi	31 00000001 56	push eax
32 push eax	32 00000002 50	
33	33	xor edi, edi
34 xor edi, edi	34 00000003 31FF	mov dword edi, [parametro_a]
35 mov dword edi,[parametro_a]	35 00000005 8B3D[04000000]	xor esi, esi
36 xor esi, esi	36 0000000B 31F6	mov dword esi, [parametro_b]
37 mov dword esi,[parametro_b]	37 0000000D 8B35[06000000]	xor eax, eax
38 xor eax, eax	38 00000013 31C0	add eax, edi
39 add eax, edi	39 00000015 01F8	add eax, esi
40 add eax, esi	40 00000017 01F0	mov dword [resultado], eax
41 mov dword [resultado], eax	41 00000019 A3[00000000]	
42	42	pop eax
43 pop eax	43 0000001E 58	pop esi
44 pop esi	44 0000001F 5E	pop edi
45 pop edi	45 00000020 5F	
46	46	ret
47 ret	47 00000021 C3	c3

Plantilla para ejercicios

init16.asm

Inicialización de hardware
Pasaje a MP
Far Jmp start

Rutinas

init32.asm

.reset_vector

jmp inicio

.init

INCBIN «init16.asm»

Movimiento de secciones

jmp start32

.sys_tables

.....

main.asm

start32:

.....

Ejercicios

Section	LMA	VMA
reset_vector	0xFFFFFFFF0	0xFFFFFFFF0
init	0xFFFF0000	0xFFFF0000
sys_tables	0xFFFF0154	0X00100000
mdata	0xFFFF01FE	0X00120000
bss	0xFFFF0205	0X00130000

Ahora les toca a ustedes



Es hora de empezar con los ejercicios