Linker

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

Volátil	0x00000000	Código
	0x0000002F	
	0x00000030	Datos
	0x00400030	
ROM	.....	

```
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

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 autocontenida 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
- ✓ Generación de código

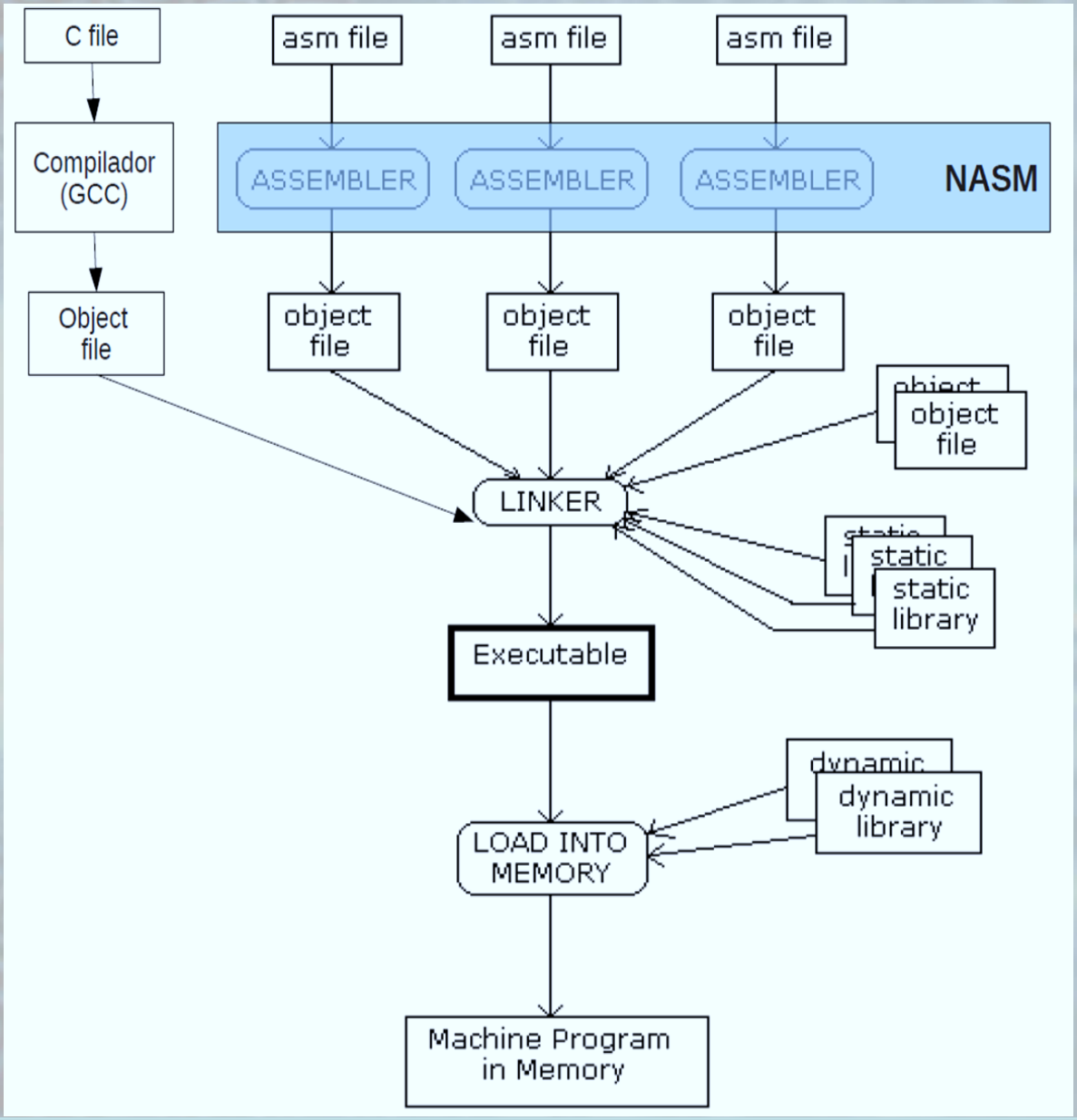
Tareas del linker (enlazador)

- ✓ 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:
…….
```

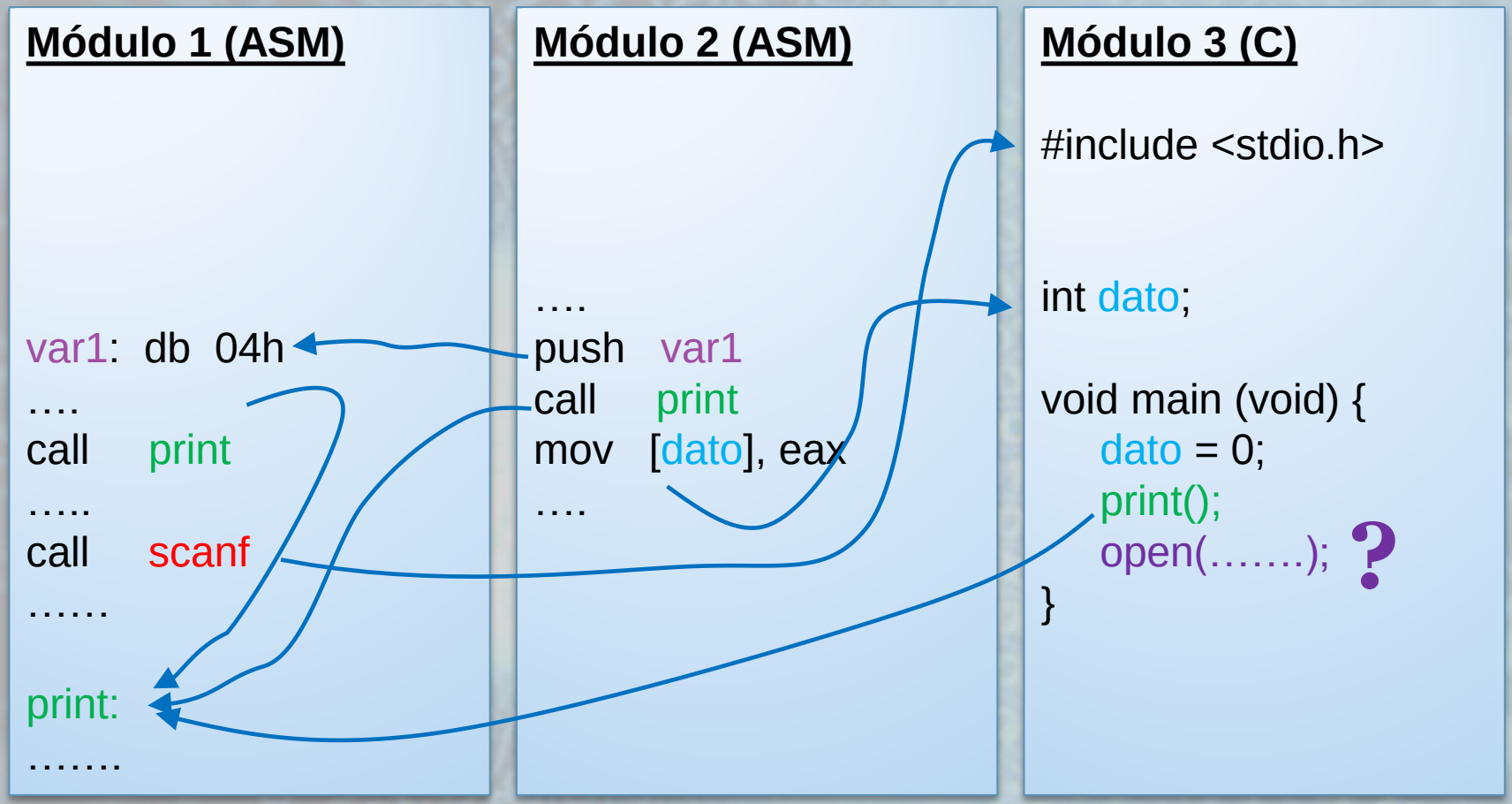
Módulo 2 (ASM)

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

Módulo 3 (C)

```
#include <stdio.h>


int i;

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

Dependencia entre símbolos - Falla



Dependencia entre símbolos externos

Módulo 1 (ASM)

```
global var1
global print
extern scanf

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»

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

Dirección física

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

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*

¿Cómo especificar esos bloques de código?

Se las llama secciones "section "

Secciones por defecto:

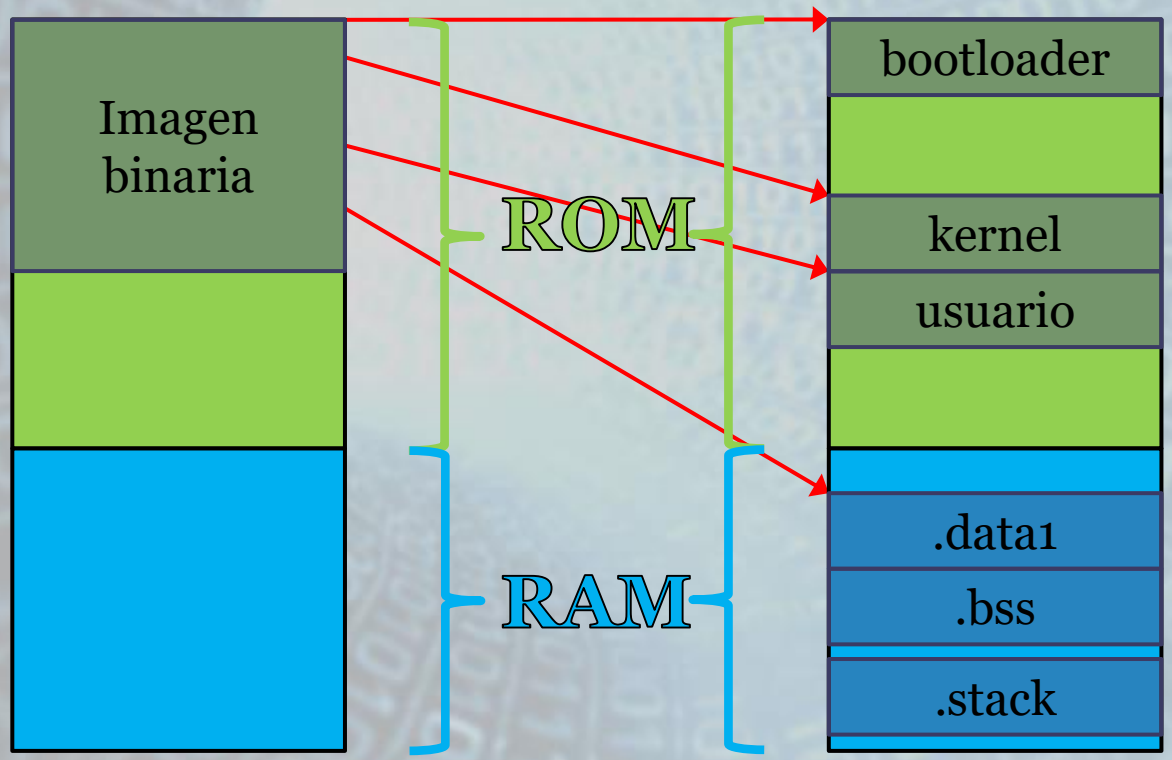
.text	: Código ejecutable
.data	: Datos inicializados
.bss	: Datos no inicializados
.rodata	: Datos constantes
.eh_frame	: Stack de C
Otras	: .eeprom, .init, .finit, etc

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

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

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

- 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)

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?

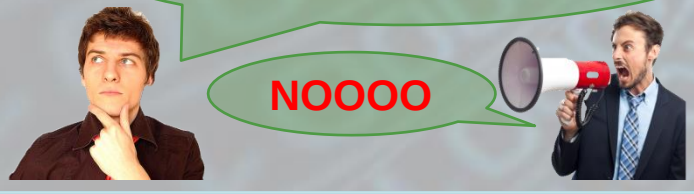


¿Quién ubica el código en memoria?

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

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

Modo ama de casa



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

Originalmente teníamos el primer banco ROM/RAM.
Luego expandimos la memoria al segundo bloque dejándola alineada.

```
section .text
_start:
  ……
  ……

section .data
i: TIMES db 1024

section .subs
print:
 ……
clear:
 ……
memcpy:

section .data
buffer: db 0x24
```

```
SECTIONS
{
    .text :
      {  *(.text);  }
    . = 0x00002000;
    .data :
       AT (LOADADDR(.text) + SIZEOF(.text))
      { *(.data*);  }
    . = 0x00040000;
    .codigo_adicional :
       AT (LOADADDR(.data) + SIZEOF(.data))
      { *(.subs); }
}
```

Tipo	Dirección	Obs.
ROM	00000000h-00001FFFh	8kB
RAM	00002000h-000031FFFh	192kB
.....	000032000h-0003F000h	Nada
ROM	00040000h-000403FFF	16kB
RAM	000404000h-000503FFFh	1MB

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

¿Cómo lo invoco?

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

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

Ejemplo sin linker script

```
.asm
 1 bits 32
 2
 3 global Inicio
 4
 5 dato:   dw 0x1234
 6
 7 Inicio:
 8   xor edi, edi
 9   mov dword edi, 0x01
10   xor esi, esi
11   mov dword esi,0x03
12   mov dword [variable], 0x04
13
14   xor eax, eax
15   add eax, edi
16   add eax, esi
17   add dword eax, [variable]
18   mov dword [resultado], eax
19   hlt
20
21   resultado:  dw 0x9876
22   variable:   resd 1
```

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

```
.bin
34 12

31 ff
bf 01 00 00 00
31 f6
be 03 00 00 00
c7 05 2e 80 00
00 04 00 00 00
31 c0
01 f8
01 f0
03 05 2e 80 00 00
a3 2c 80 00 00
f4

76 98
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

```
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
}
```

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
}
```

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);
```

attr:

'R'	**Read-only section**
'W'	**Read/write section**
'X'	**Executable section**
'A'	**Allocatable section (debe reservarse, en gral. sin contenido e inicilizada 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);
```

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*); }

    . = 0x0A000;
    __datos_no_iniciali_inicio = .;
    .bss   : { *(.dat_no_inic*); }

    . = 0x0B000;
    .codigo_adicional : AT (ADDR(.text) + SIZEOF(.text)) { *(.codigo_funciones); }
}
```

.asm

```
 1 bits 32
 2
 3 global Inicio
 4
 5 SECTION     .codigo_principal
 6
 7 Inicio:
 8   mov dword edi, 0x1
 9   mov dword [parametro_a], edi
10   mov dword esi, 0x3
11   mov dword [parametro_b], esi
12   call far [_suma]
13   mov dword eax, [resultado]
14   add dword eax, [variable_a]
15   mov dword [array_a], eax
16   hlt
17
18 SECTION .dat_inic_a progbits
19   resultado:  dw 0x9876
20   texto_a:    db "Hola"
21
22 SECTION .dat_no_inic_a nobits1
23   variable_a:  resd 1

24   parametro_a: resw 1
25   parametro_b: resw 1
26   array_a:     resb 4
27
28 SECTION .codigo_funciones
29   _suma:
30      push edi
31      push esi
32      push eax
33
34      xor edi, edi
35      mov dword edi,[parametro_a]
36      xor esi, esi
37      mov dword esi,[parametro_b]
38      xor eax, eax
39      add eax, edi
40      add eax, esi
41      mov dword [resultado], eax
42
43      pop eax
44      pop esi
45      pop edi
46
47      ret
```

.lst

```
 1                                  bits 32
 2
 3                                  global Inicio
 4
 5 SECTION                          .codigo_principal
 6
 7                                  Inicio:
 8 00000000 BF01000000                      mov dword edi, 0x1
 9 00000005 893D[04000000]                  mov dword [parametro_a], edi
10 0000000B BE03000000                      mov dword esi, 0x3
11 00000010 8935[06000000]                  mov dword [parametro_b], esi
12 00000016 FF1D[00000000]                  call far [_suma]
13 0000001C A1[00000000]                    mov dword eax, [resultado]
14 00000021 0305[00000000]                  add dword eax, [variable_a]
15 00000027 A3[08000000]                    mov dword [array_a], eax
16 0000002C F4                              hlt
17
18                                  SECTION .dat_inic_a progbits
19 00000000 7698                            resultado: dw 0x9876
20 00000002 486F6C61                        texto_a:   db "Hola"
21
22                                  SECTION .dat_no_inic_a nobits1
23 00000000 <res 00000004>                  variable_a:  resd 1

24 00000004 <res 00000002>                  parametro_a: resw 1
25 00000006 <res 00000002>                  parametro_b: resw 1
26 00000008 <res 00000004>                  array_a:     resb 4
27
28 SECTION .codigo_funciones
29                                      _suma:
30 00000000 57                              push edi
31 00000001 56                              push esi
32 00000002 50                              push eax
33
34 00000003 31FF                            xor edi, edi
35 00000005 8B3D[04000000]                  mov dword edi, [parametro_a]
36 0000000B 31F6                            xor esi, esi
37 0000000D 8B35[06000000]                  mov dword esi, [parametro_b]
38 00000013 31C0                            xor eax, eax
39 00000015 01F8                            add eax, edi
40 00000017 01F0                            add eax, esi
41 00000019 A3[00000000]                    mov dword [resultado], eax
42
43 0000001E 58                              pop eax
44 0000001F 5E                              pop esi
45 00000020 5F                              pop edi
46
47 00000021 C3                              ret
```

.bin

```
bf 01 00 00 00
89 3d 04 a0 00 00
be 03 00 00 00
89 35 06 a0 00 00
ff 1d 00 b0 00 00
a1 00 90 00 00
03 05 00 a0 00 00
a3  08 a0 00 00
f4

76 98
48 6f 6c 61

00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00

57
56
50

31 ff
8b 3d 04 a0 00 00
31 f6
8b 35 06 a0 00 00
31 c0
01 f8
01 f0
a3 00  90 00 00

58
5e
5f

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

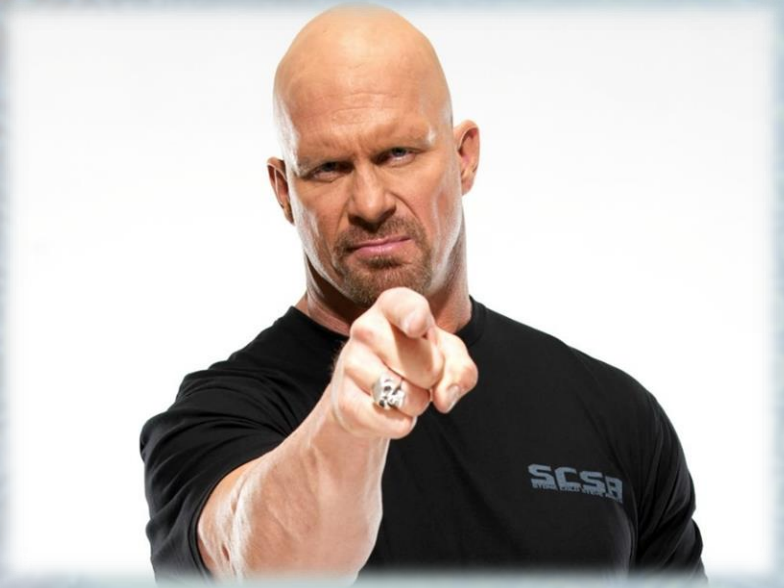
.sys_tables
……..

main.asm

start32:
……..
Ejercicios

Section	LMA	VMA
reset_vector	0xFFFFFFF0	0xFFFFFFF0
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