

Sistemas Empotrados

Tema 3: Cargador de arranque

Lección 9:
Diseño de un boot loader completo



Contenidos

Tema 3: Cargador de arranque

- Necesidad de un boot loader en cualquier sistema empujado

- Entrada tras el reset y salto a la aplicación

- Soporte para variables globales

- Soporte para funciones y variables locales

- Carga de la aplicación en la RAM

- Remapeo de la memoria

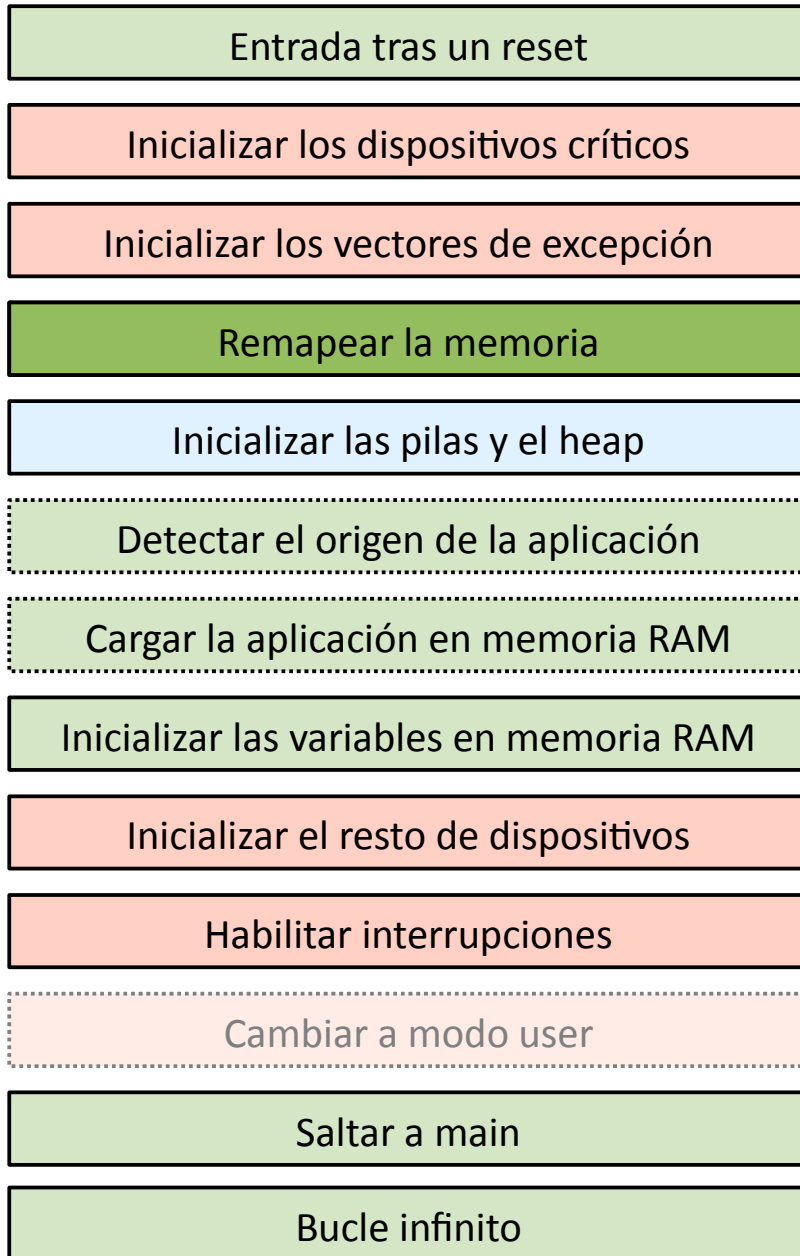
- Soporte para excepciones

- Soporte para memoria dinámica

- Soporte para todos los modos de ejecución

- Inicialización de los dispositivos

El boot loader paso a paso: Remapeo de la memoria



Al arrancar el sistema $PC = 0x00000000$. Esta dirección debe estar mapeada a una ROM/Flash en el arranque y debe contener el vector de reset

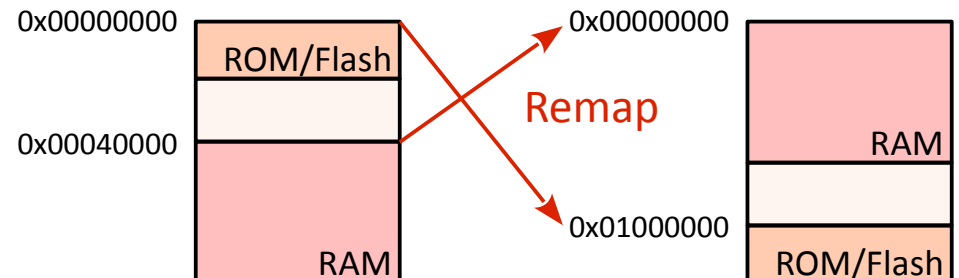
Posteriormente, nuestra aplicación querrá cambiar los vectores de excepción, por lo que la dirección $0x00000000$ debería estar mapeada a una memoria RAM. Además, la RAM es más rápida, por lo que las interrupciones tendrán menos latencia

El cargador de arranque es el encargado de cambiar el mapa de memoria

Tanto el cargador como la aplicación se compilan para ejecutarse con el segundo mapa de memoria, por lo que el cambio del mapa debe hacerse cuanto antes

Antes del remapeo el cargador sólo podrá usar direcciones relativas a PC, ya que las direcciones absolutas sólo serán válidas tras el remapeo

El cambio de mapa es dependiente de cada plataforma. Normalmente se realiza cambiando los parámetros del controlador de memoria o de la MMU del sistema



Remapeo de memoria en el Atmel AT91R40008

Address	Function
0xFFFFFFF	On-chip Peripherals
0xFFC00000	
0xFFBFFFFF	
0x00400000	
0x003FFFFF	On-chip Primary RAM Bank
0x00300000	
0x002FFFFF	Reserved On-chip Device
0x00200000	
0x001FFFFF	On-chip ROM or Secondary RAM Bank
0x00100000	
0x000FFFFF	External Device Selected by NCS0 or On-chip ROM or Secondary RAM Bank
0x00000000	

Se ejecuta con el mapa de memoria de arranque

@ Remapeo de memoria

```

@ Obtenemos la dirección real del salto
@ tras el remapeo
@ ldr usa direccionamiento relativo a pc
@ para acceder al puntero
ldr    r12, _pt_init_remap

@ Código para hacer el remapeo
@ En este caso, solo hay que cambiar la
@ configuración del controlador de memoria
@ Como la tabla está en el código del
@ cargador, no se puede usar su dirección,
@ hay que obtener su dirección a partir de pc
sub    r10, pc, #(8+._mem_ctrl_init_table)
ldr    r11, =_mem_ctrl_base
ldmia  r10!, {r0-r9}
stmia  r11!, {r0-r9} @ Aqui se cambia el mapa
    
```

```

@ Salto global a dirección absoluta de la
@ siguiente instrucción
mov    pc, r12

@ Como el linker asigna las direcciones de
@ acuerdo al segundo mapa de memoria,
@ init_remap contiene la dirección adecuada
_pt_init_remap:
    .word _init_remap

_init_remap:
    
```

Address	Function
0xFFFFFFF	On-chip Peripherals
0xFFC00000	
0xFFBFFFFF	
0x00400000	External Devices (Up to 8)
0x003FFFFF	
0x00300000	Reserved
0x002FFFFF	
0x00200000	
0x001FFFFF	
0x00100000	RAM Bank
0x000FFFFF	
0x00000000	On-chip Primary RAM Bank

Se ejecuta con el mapa de memoria definitivo

Antes del remapeo

Después del remapeo

Soporte para el remapeo de memoria

Linker script

```
ENTRY(_reset)

MEMORY
{
    ram    : org = 0x00000000, len = 0x00040000
    flash  : org = 0x01000000, len = 0x00100000
}

SECTIONS
{
    .startup : { ... } > flash

    .text :      { ... } > ram AT > flash
                _text_flash_start = LOADADDR(.text);

    .rodata :    { ... } > flash

    .data :      { ... } > ram AT > flash
                _data_flash_start = LOADADDR(.data);

    .bss :       { ... } > ram

    _ram_limit = ORIGIN(ram) + LENGTH(ram);
    _stack_size = 0x800;

    .stack _ram_limit - _stack_size : { ... }
}
```

El linker script define las direcciones de las regiones tal y como estarán después del remapeo

Por eso hasta que no se ha llevado a cabo el remapeo sólo se pueden hacer saltos locales

Ejecución de nuestro firmware

_reset=0x00000000



Código del cargador

.startup

Copia del código de la aplicación

.text

Constantes globales

.rodata

Valores iniciales de las variables globales

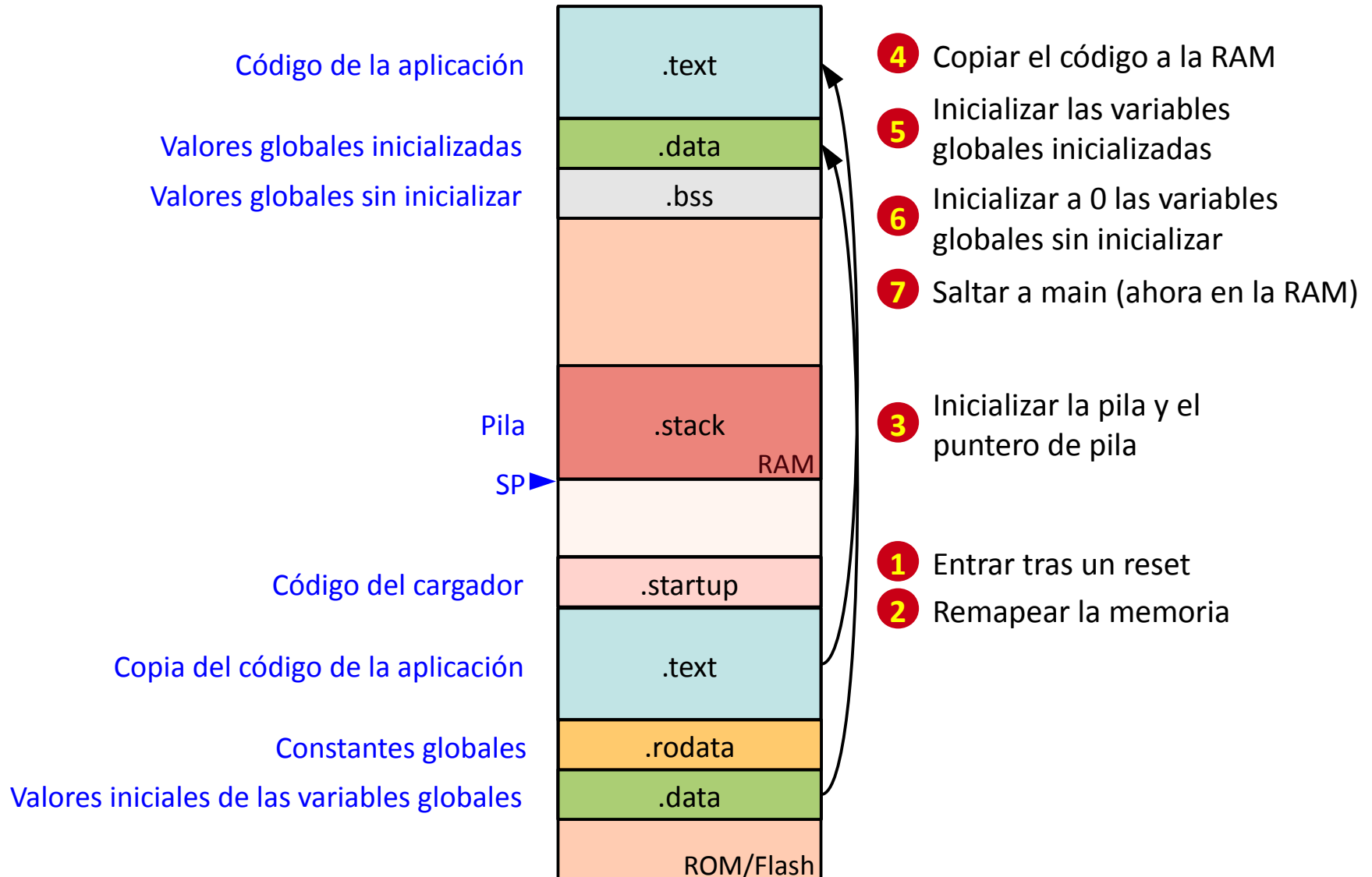
.data

ROM/Flash

RAM

- 1 Entrar tras un reset
- 2 Remapear la memoria

Ejecución de nuestro firmware



Contenidos

Tema 3: Cargador de arranque

- Necesidad de un boot loader en cualquier sistema empujado

- Entrada tras el reset y salto a la aplicación

- Soporte para variables globales

- Soporte para funciones y variables locales

- Carga de la aplicación en la RAM

- Remapeo de la memoria

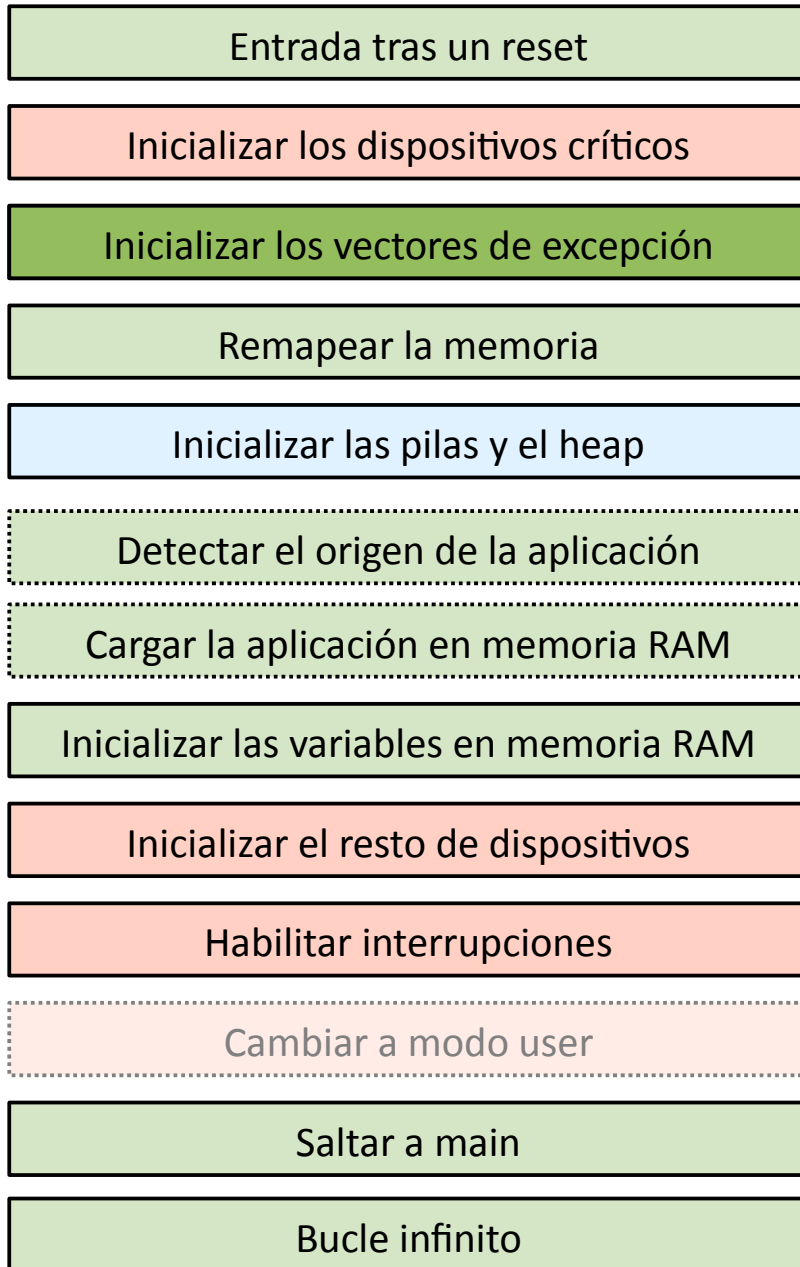
- Soporte para excepciones

- Soporte para memoria dinámica

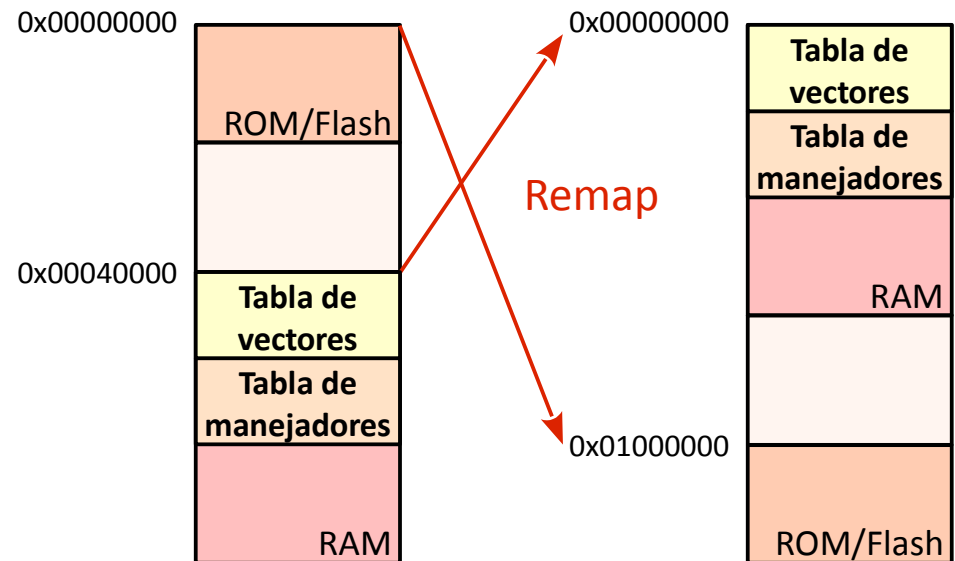
- Soporte para todos los modos de ejecución

- Inicialización de los dispositivos

El boot loader paso a paso: Soporte para excepciones



Antes del remapeo debemos asegurarnos de que en las primeras posiciones de la RAM habrá unos vectores de excepción válidos una vez que se haya remapeado la memoria



El cargador copia la tabla de vectores y la de manejadores al comienzo de la RAM antes de remapear

Al remapear los vectores de excepción estarán ubicados en la dirección correcta

El boot loader paso a paso: Inicialización de los vectores de excepción

b _setup_vectors

@ Tabla de vectores que se copiará a la RAM

```
.globl _vector_table
_vector_table:
    ldr    pc, [pc, #24]    @ Soft reset
    ldr    pc, [pc, #24]    @ Undefined
    ldr    pc, [pc, #24]    @ SWI
    ldr    pc, [pc, #24]    @ Prefetch abort
    ldr    pc, [pc, #24]    @ Data abort
    nop                    @ Reserved
    ldr    pc, [pc, #24]    @ IRQ
    ldr    pc, [pc, #24]    @ FIQ
```

@ Tabla de direcciones absolutas de los
@ manejadores

```
.word    _soft_reset_handler
.word    _undef_handler
.word    _swi_handler
.word    _pabt_handler
.word    _dabt_handler
nop
.word    _irq_handler
.word    _fiq_handler
```

@ Manejadores por defecto
_soft_reset_handler:

```
b        .
_undef_handler:
b        .
_swi_handler:
b        .
_pabt_handler:
b        .
_dabt_handler:
b        .
_irq_handler:
b        .
_fiq_handler:
b        .
```

@ Copiamos las tablas de vectores y
@ de manejadores a la RAM

```
.globl _setup_vectors
_setup_vectors:
    sub    r8, pc, #(8+._vector_table)
    ldr    r9, =_ram_base_boot
    ldmbia r8!, {r0-r7}
    stmbia r9!, {r0-r7}
    ldmbia r8!, {r0-r7}
    stmbia r9!, {r0-r7}
```

Se debe usar una
dirección relativa.
Todavía no hemos
remapeado la
memoria

Definida
por el
linker

La tabla de vectores es una tabla de saltos globales a las direcciones de los manejadores

Las tablas de vectores y manejadores deben copiarse al principio de la RAM antes de remapear

Soporte para excepciones

Linker script

```
MEMORY
{
    ram    : org = 0x00000000, len = 0x00040000
    flash  : org = 0x01000000, len = 0x00100000
}
```

SECTIONS

```
{
    .startup : { ... } > flash

    .vectors : {
        . += 0x20 ;
        _excep_handlers = . ;
        . += 0x20 ;
    } > ram
    _ram_base_boot = 0x00300000 ;

    .text :      { ... } > ram AT > flash
    _text_flash_start = LOADADDR(.text);

    .rodata :    { ... } > flash

    .data :      { ... } > ram AT > flash
    _data_flash_start = LOADADDR(.data);

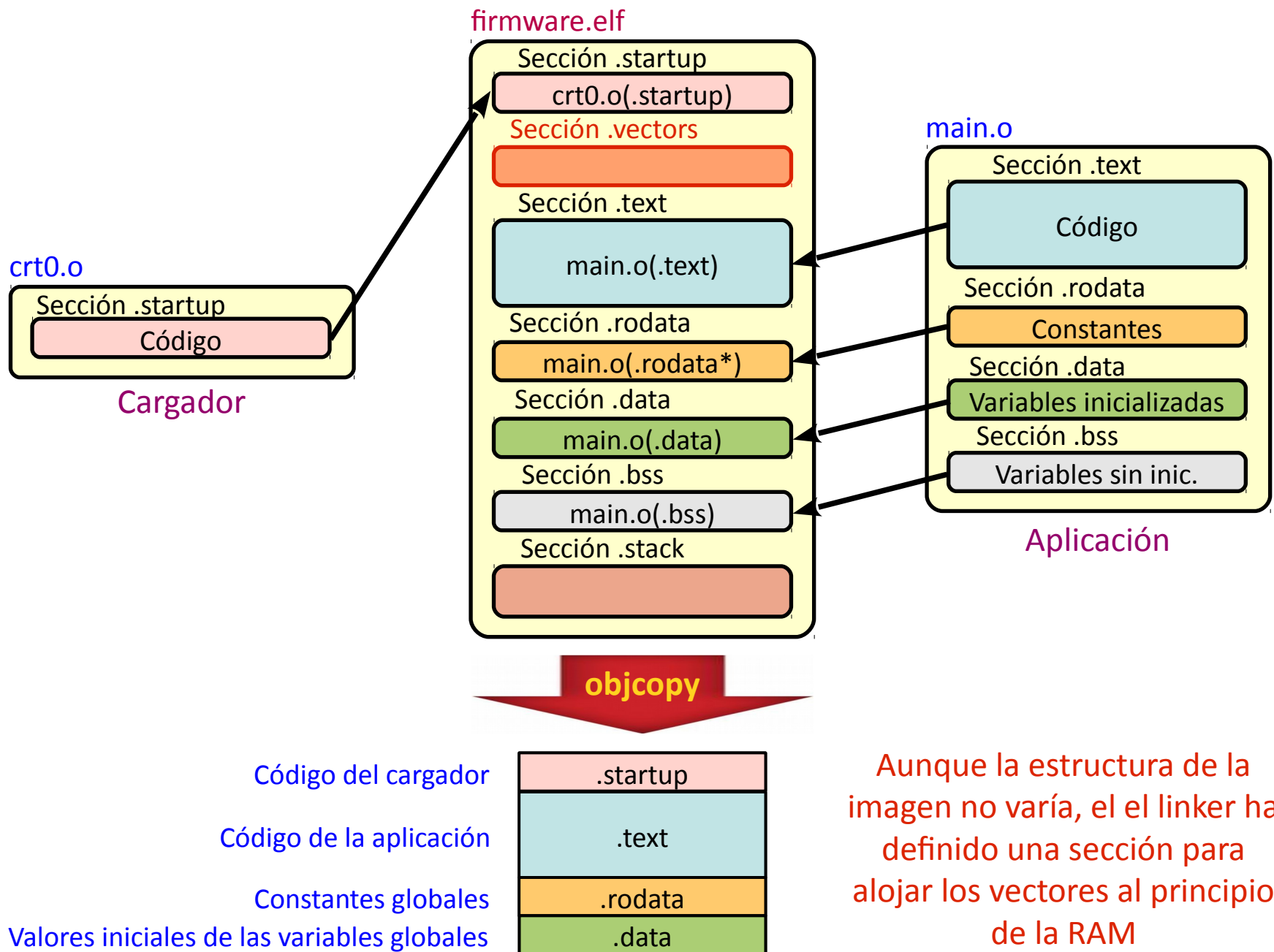
    .bss :       { ... } > ram

    _ram_limit = ORIGIN(ram) + LENGTH(ram);
    _stack_size = 0x800;
    .stack _ram_limit - _stack_size : { ... }
}
```

Reservamos espacio para la tabla de vectores y la tabla de manejadores

Dirección de inicio de la RAM antes del remapeo

Estructura de nuestro firmware



Ejecución de nuestro firmware

_reset=0x00000000



Código del cargador

.startup

Copia del código de la aplicación

.text

Constantes globales

.rodata

Valores iniciales de las variables globales

.data

ROM/Flash

Tablas de vectores y manejadores

.vectors

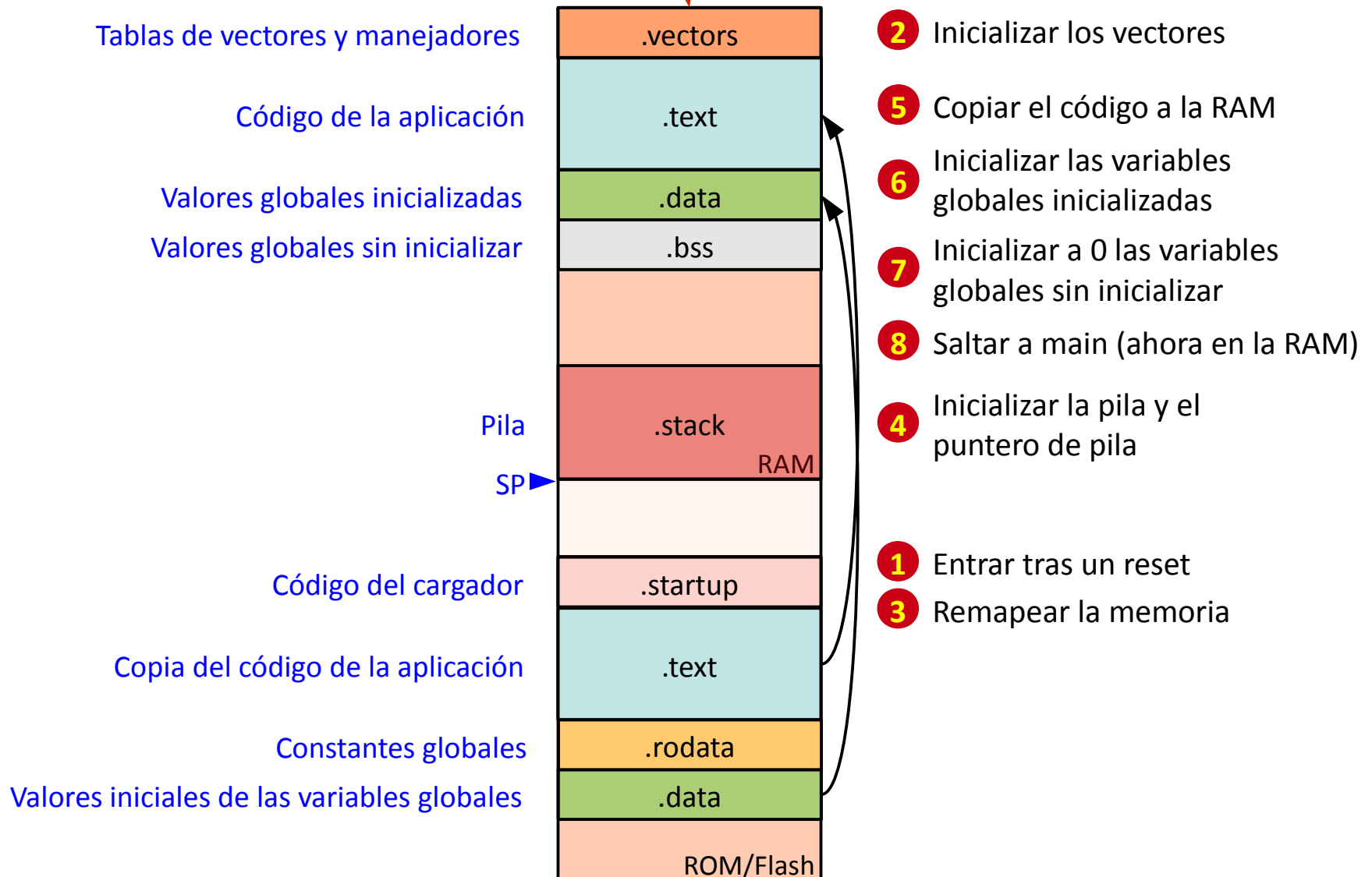
RAM

- 1 Entrar tras un reset
- 3 Remapear la memoria

- 2 Inicializar los vectores

Ejecución de nuestro firmware

`_vector_table=0x00000000`
tras el remapeo



Contenidos

Tema 3: Cargador de arranque

- Necesidad de un boot loader en cualquier sistema empujado

- Entrada tras el reset y salto a la aplicación

- Soporte para variables globales

- Soporte para funciones y variables locales

- Carga de la aplicación en la RAM

- Remapeo de la memoria

- Soporte para excepciones

- Soporte para memoria dinámica

- Soporte para todos los modos de ejecución

- Inicialización de los dispositivos

El boot loader paso a paso: Soporte para memoria dinámica

Entrada tras un reset

Inicializar los dispositivos críticos

Inicializar los vectores de excepción

Remapear la memoria

Inicializar las pilas y el heap

Detectar el origen de la aplicación

Cargar la aplicación en memoria RAM

Inicializar las variables en memoria RAM

Inicializar el resto de dispositivos

Habilitar interrupciones

Cambiar a modo user

Saltar a main

Bucle infinito

La gestión de memoria dinámica necesita que exista un heap en la memoria RAM

El linker script definirá la zona de memoria que se usará como heap

Aunque no es estrictamente necesario, nuestro cargador inicializará el heap a cero

@ Fijamos a 0 la memoria RAM
@ reservada para el heap

```
ldr    a1, =_heap_start
ldr    a2, =_heap_end
ldr    a3, =0
bl     _ram_init
```

Direcciones
generadas
por el linker

Soporte para memoria dinámica

Linker script

```
SECTIONS
{
    .startup : { ... } > flash

    .vectors : { ... } > ram
    _ram_base_boot = 0x00300000 ;

    .text :      { ... } > ram AT > flash
    _text_flash_start = LOADADDR(.text);

    .rodata :    { ... } > flash

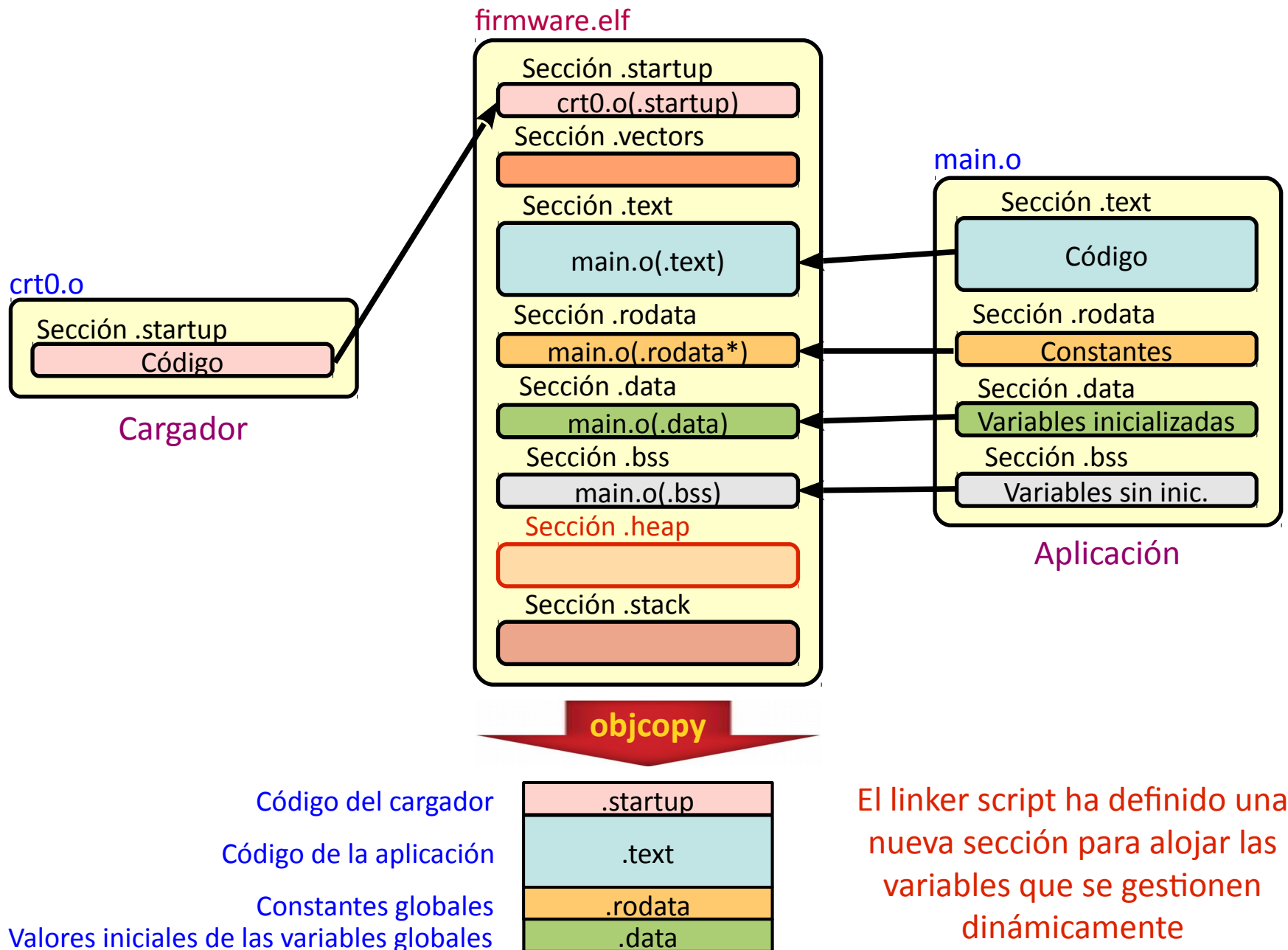
    .data :      { ... } > ram AT > flash
    _data_flash_start = LOADADDR(.data);

    .bss :       { ... } > ram

    _ram_limit = ORIGIN(ram) + LENGTH(ram);
    _stack_size = 0x800;
    .stack _ram_limit - _stack_size : {
        _stack_bottom = . ;
        . += _stack_size ;
        _stack_top = . ;
    }
    _heap_size = _stack_bottom - _bss_end ;
    .heap _bss_end : {
        _heap_start = . ;
        . += _heap_size ;
        _heap_end = . ;
    }
}
```

Ubicamos el heap desde el final de la sección bss hasta que llegamos a la pila

Estructura de nuestro firmware



Ejecución de nuestro firmware

_reset=0x00000000



Código del cargador

.startup

Copia del código de la aplicación

.text

Constantes globales

.rodata

Valores iniciales de las variables globales

.data

ROM/Flash

Tablas de vectores y manejadores

.vectors

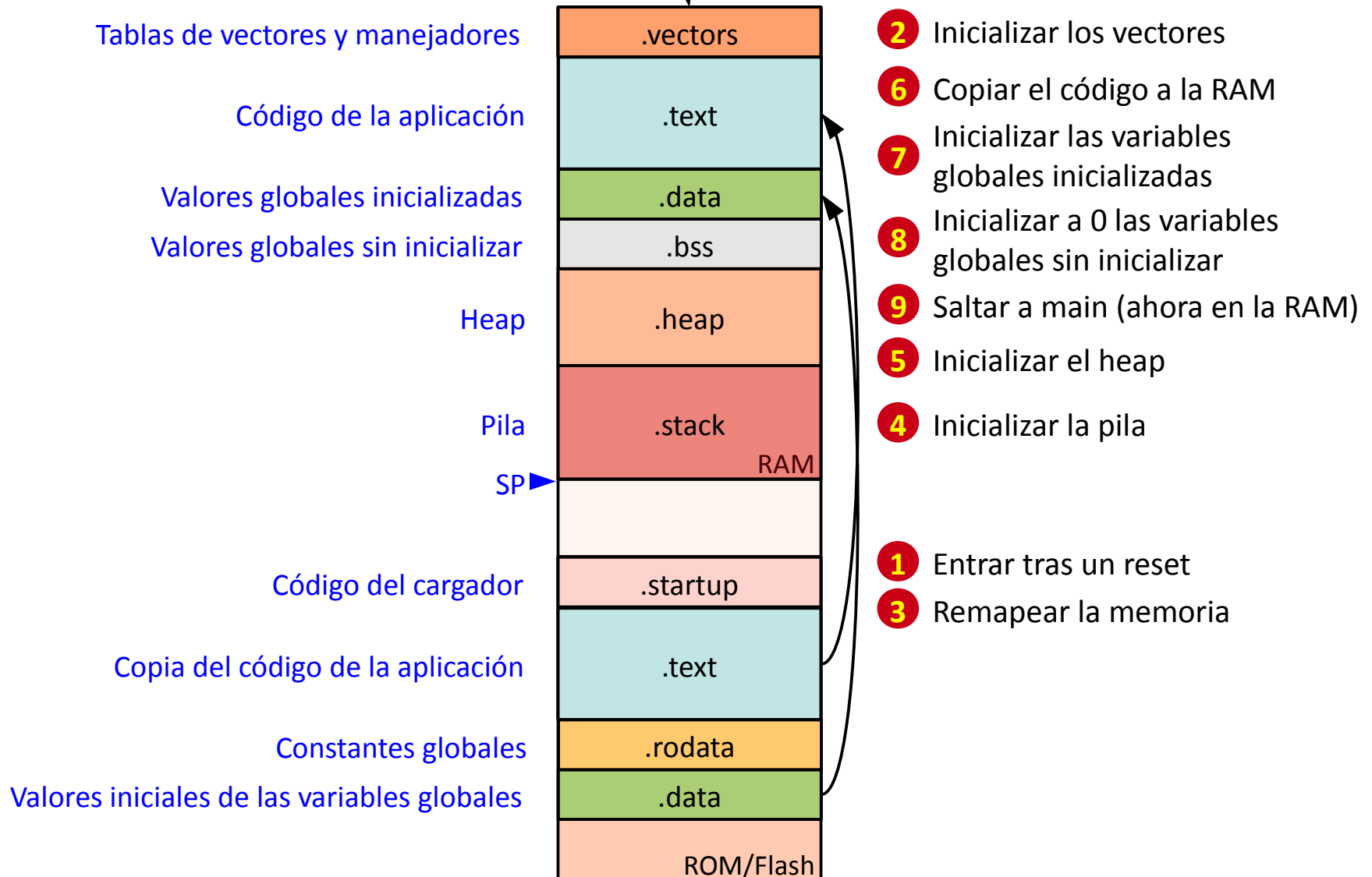
RAM

- 1 Entrar tras un reset
- 3 Remapear la memoria

- 2 Inicializar los vectores

Ejecución de nuestro firmware

_vector_table=0x00000000
tras el remapeo



Contenidos

Tema 3: Cargador de arranque

- Necesidad de un boot loader en cualquier sistema empujado

- Entrada tras el reset y salto a la aplicación

- Soporte para variables globales

- Soporte para funciones y variables locales

- Carga de la aplicación en la RAM

- Remapeo de la memoria

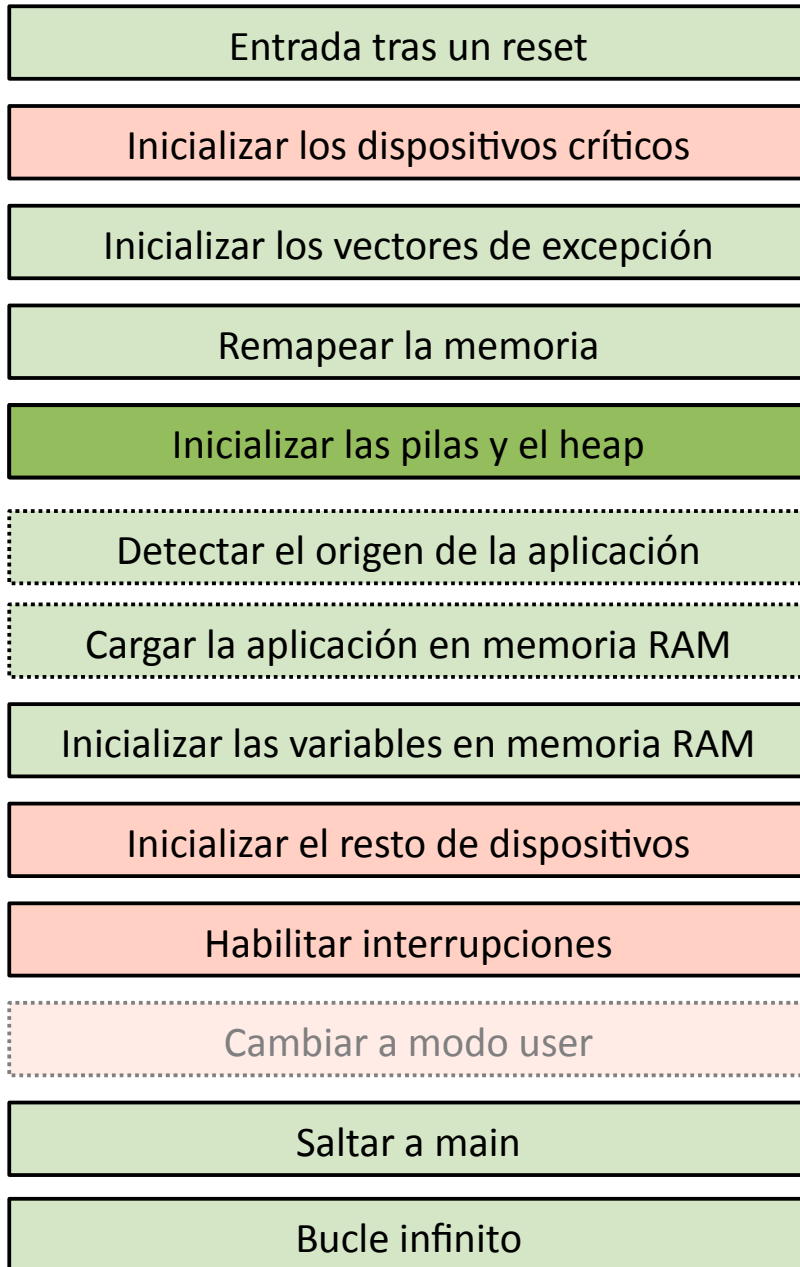
- Soporte para excepciones

- Soporte para memoria dinámica

- Soporte para todos los modos de ejecución

- Inicialización de los dispositivos

El boot loader paso a paso: Soporte todos los modos de ejecución

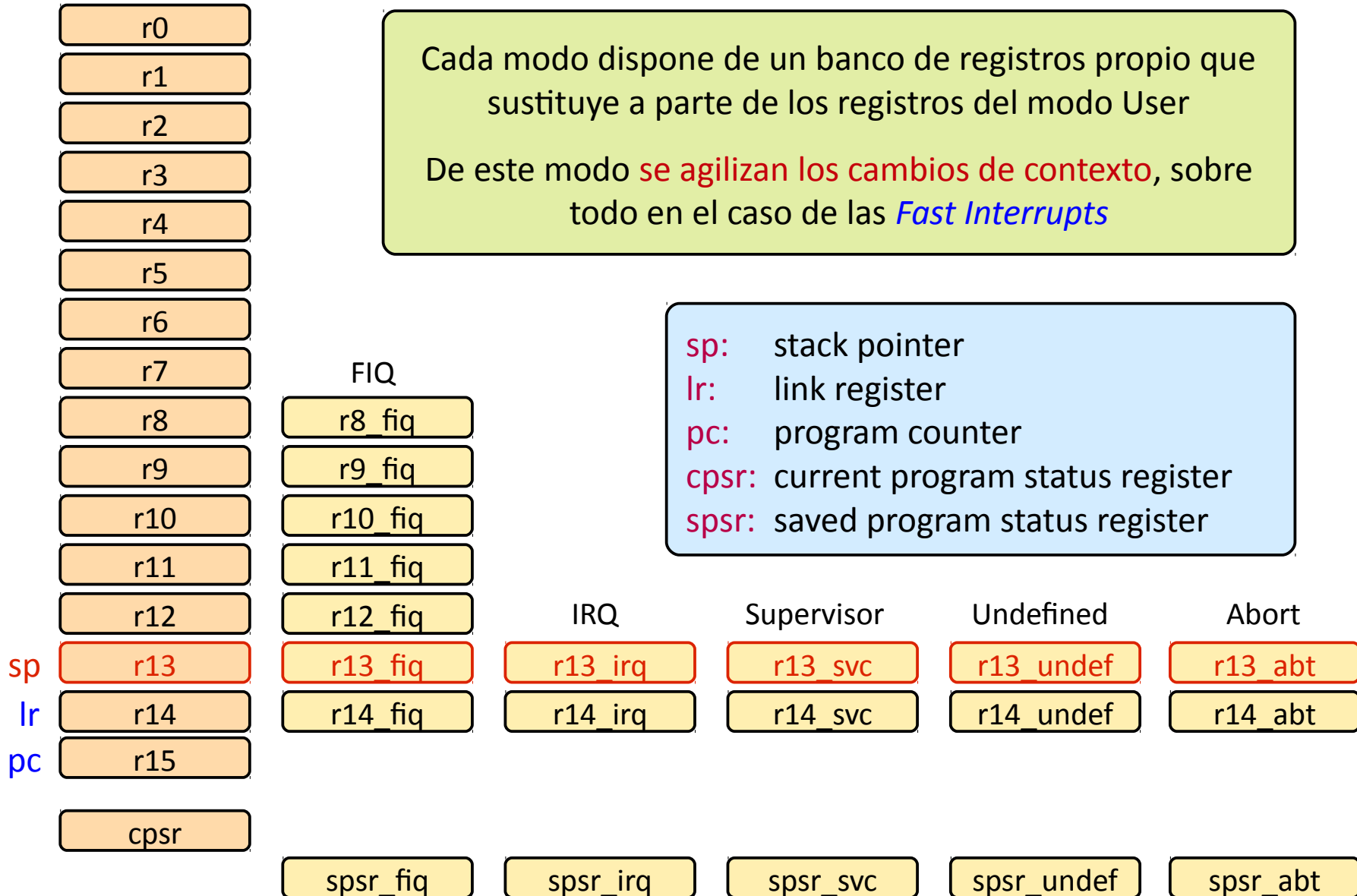


En la arquitectura ARM hay siete modos de ejecución y seis de ellos tienen una pila independiente (y un registro SP). El cargador deberá inicializar cada una de las pilas del sistema

Modo	Descripción	
Supervisor (SVC)	Se entra cada vez que se hace un reset y cuando ocurre una Interrupción Software (SWI)	Modos privilegiados Para servir excepciones o acceder a recursos protegidos
FIQ	Se entra cada vez que ocurre una Interrupción de alta prioridad (Fast)	
IRQ	Se entra cada vez que ocurre una Interrupción de baja prioridad (normal)	
Abort	Para manejar violaciones de acceso a memoria	
Undef	Para manejar instrucciones no definidas	
System	Modo privilegiado que usa los mismos registros que el modo User	Modo sin privilegios
User	Modo en el que se ejecutan la mayoría de aplicaciones y tareas del sistema operativo	

Registros y modos de ejecución

User y
System



Inicialización de las pilas en todos los modos de ejecución

@ Fijamos un valor inicial para las pilas

```
ldr a1, =_stacks_bottom
ldr a2, =_stacks_top
ldr a3, =_STACK_FILLER
bl _ram_init
```

@ Pila del modo Undefined

```
msr cpsr_c, #(_UND_MODE | _IRQ_DISABLE | _FIQ_DISABLE)
ldr sp, =_und_stack_top
```

@ Pila del modo Abort

```
msr cpsr_c, #(_ABT_MODE | _IRQ_DISABLE | _FIQ_DISABLE)
ldr sp, =_abt_stack_top
```

@ Pila del modo System

```
msr cpsr_c, #(_SYS_MODE | _IRQ_DISABLE | _FIQ_DISABLE)
ldr sp, =_sys_stack_top
```

@ Pila del modo FIQ

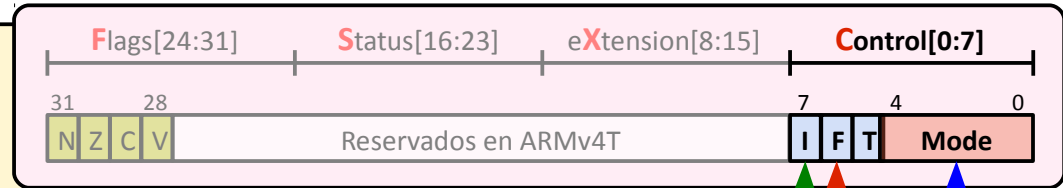
```
msr cpsr_c, #(_FIQ_MODE | _IRQ_DISABLE | _FIQ_DISABLE)
ldr sp, =_fiq_stack_top
```

@ Pila del modo IRQ

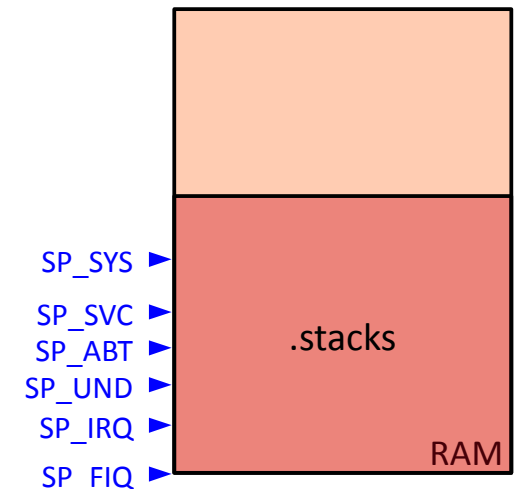
```
msr cpsr_c, #(_IRQ_MODE | _IRQ_DISABLE | _FIQ_DISABLE)
ldr sp, =_irq_stack_top
```

@ Pila del modo Supervisor

```
@ La última para que el cargador siga en modo SVC
msr cpsr_c, #(_SVC_MODE | _IRQ_DISABLE | _FIQ_DISABLE)
ldr sp, =_svc_stack_top
```



```
.set _IRQ_DISABLE, 0x80
.set _FIQ_DISABLE, 0x40
.set _USR_MODE, 0x10
.set _FIQ_MODE, 0x11
.set _IRQ_MODE, 0x12
.set _SVC_MODE, 0x13
.set _ABT_MODE, 0x17
.set _UND_MODE, 0x1B
.set _SYS_MODE, 0x1F
```



El boot loader paso a paso: Cambio a modo user

Entrada tras un reset

Inicializar los dispositivos críticos

Inicializar los vectores de excepción

Remapear la memoria

Inicializar las pilas y el heap

Detectar el origen de la aplicación

Cargar la aplicación en memoria RAM

Inicializar las variables en memoria RAM

Inicializar el resto de dispositivos

Habilitar interrupciones

Cambiar a modo user

Saltar a main

Bucle infinito



@ Cambiamos a modo User y
@ habilitamos las interrupciones

```
msr      cpsr_c, #_USR_MODE
```

En la arquitectura ARMv4T, si fijamos a 0 los bits I y F de cpsr_c se habilitan las interrupciones IRQ y FIQ. Como estos bits están en el mismo campo que el modo, la habilitación se puede hacer en la misma instrucción

En otras arquitecturas hará falta algo más de código

Cambiamos a modo user. Este paso es opcional. Las aplicaciones muy sencillas se suelen ejecutar en modo SVC

Soporte para todos los modos de ejecución

Linker script

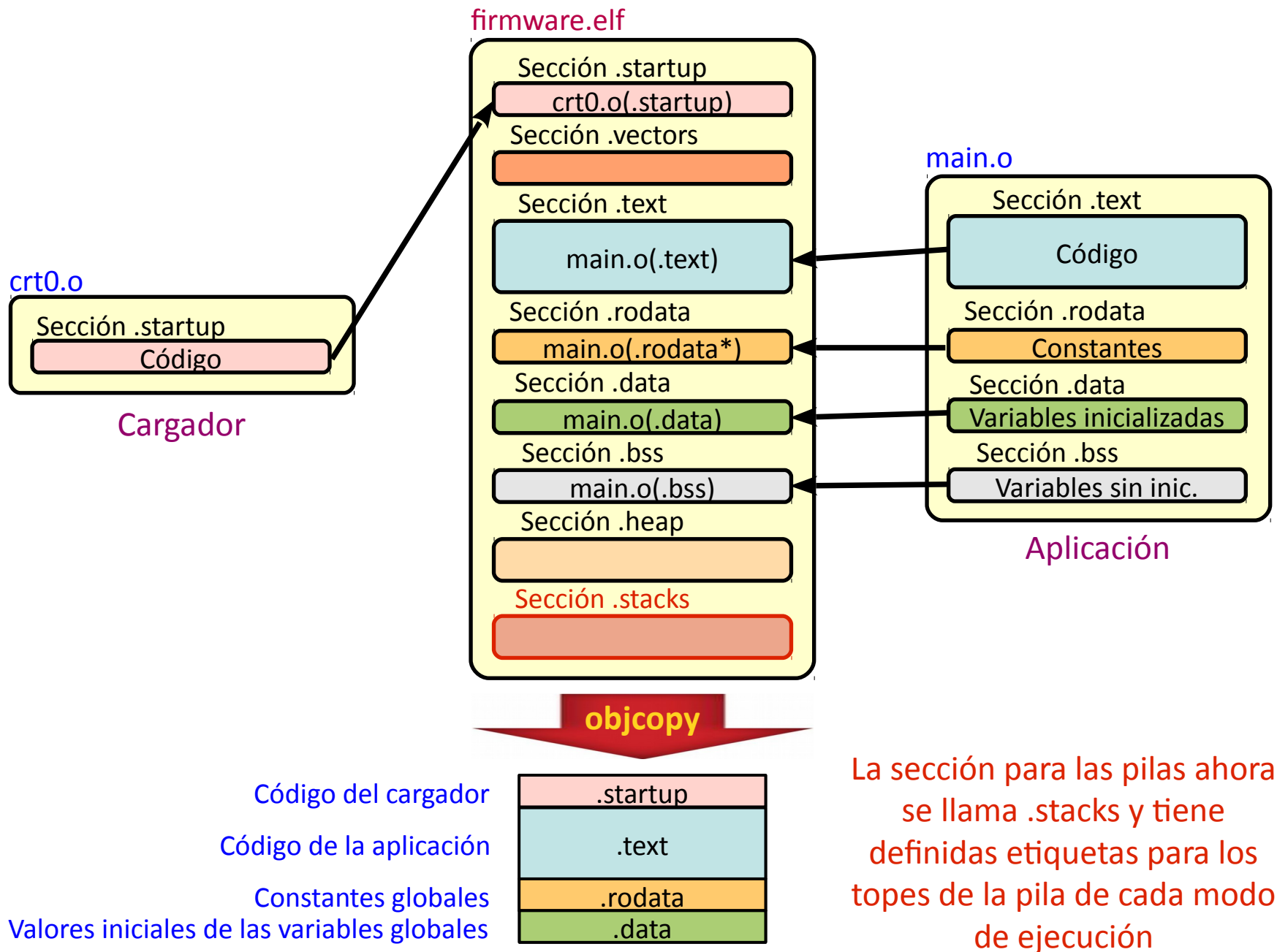
```
ENTRY(_reset)

MEMORY {
    ram    : org = 0x00000000, len = 0x00040000
    flash  : org = 0x01000000, len = 0x00100000
}

SECTIONS {
    .startup : { *(.startup); } > flash
    .vectors : { . += 0x20 ;
                 _excep_handlers = . ;
                 . += 0x20 ;
    } > ram
    _ram_base_boot = 0x00300000 ;
    .text : { _text_start = . ;
              *(.text);
              _text_end = . ;
    } > ram AT > flash
    _text_flash_start = LOADADDR(.text);
    .rodata : { *(.rodata*);
                .=ALIGN(4) ;
    } > flash
    .data : { _data_start = . ;
              *(.data);
              . = ALIGN(4) ;
              _data_end = . ;
    } > ram AT > flash
    _data_flash_start = LOADADDR(.data);
    .bss : { _bss_start = . ;
             *(.bss);
             . = ALIGN(4) ;
             *(COMMON);
             . = ALIGN(4) ;
             _bss_end = . ;
    } > ram
```

```
    _ram_limit = ORIGIN(ram) + LENGTH(ram);
    _sys_stack_size = 1024 ;
    _irq_stack_size = 256 ;
    _fiq_stack_size = 256 ;
    _svc_stack_size = 256 ;
    _abt_stack_size = 16 ;
    _und_stack_size = 16 ;
    _stacks_size = _stacks_top - _stacks_bottom ;
    .stacks _ram_limit - _stacks_size :
    {
        _stacks_bottom = . ;
        . += _sys_stack_size ;
        _sys_stack_top = . ;
        . += _svc_stack_size ;
        _svc_stack_top = . ;
        . += _abt_stack_size ;
        _abt_stack_top = . ;
        . += _und_stack_size ;
        _und_stack_top = . ;
        . += _irq_stack_size ;
        _irq_stack_top = . ;
        . += _fiq_stack_size ;
        _fiq_stack_top = . ;
        _stacks_top = . ;
    }
    _heap_size = _stacks_bottom - _bss_end;
    .heap _bss_end :
    {
        _heap_start = . ;
        . += _heap_size ;
        _heap_end = . ;
    }
}
```

Estructura de nuestro firmware



Ejecución de nuestro firmware

_reset=0x00000000



Código del cargador

.startup

Copia del código de la aplicación

.text

Constantes globales

.rodata

Valores iniciales de las variables globales

.data

ROM/Flash

Tablas de vectores y manejadores

.vectors

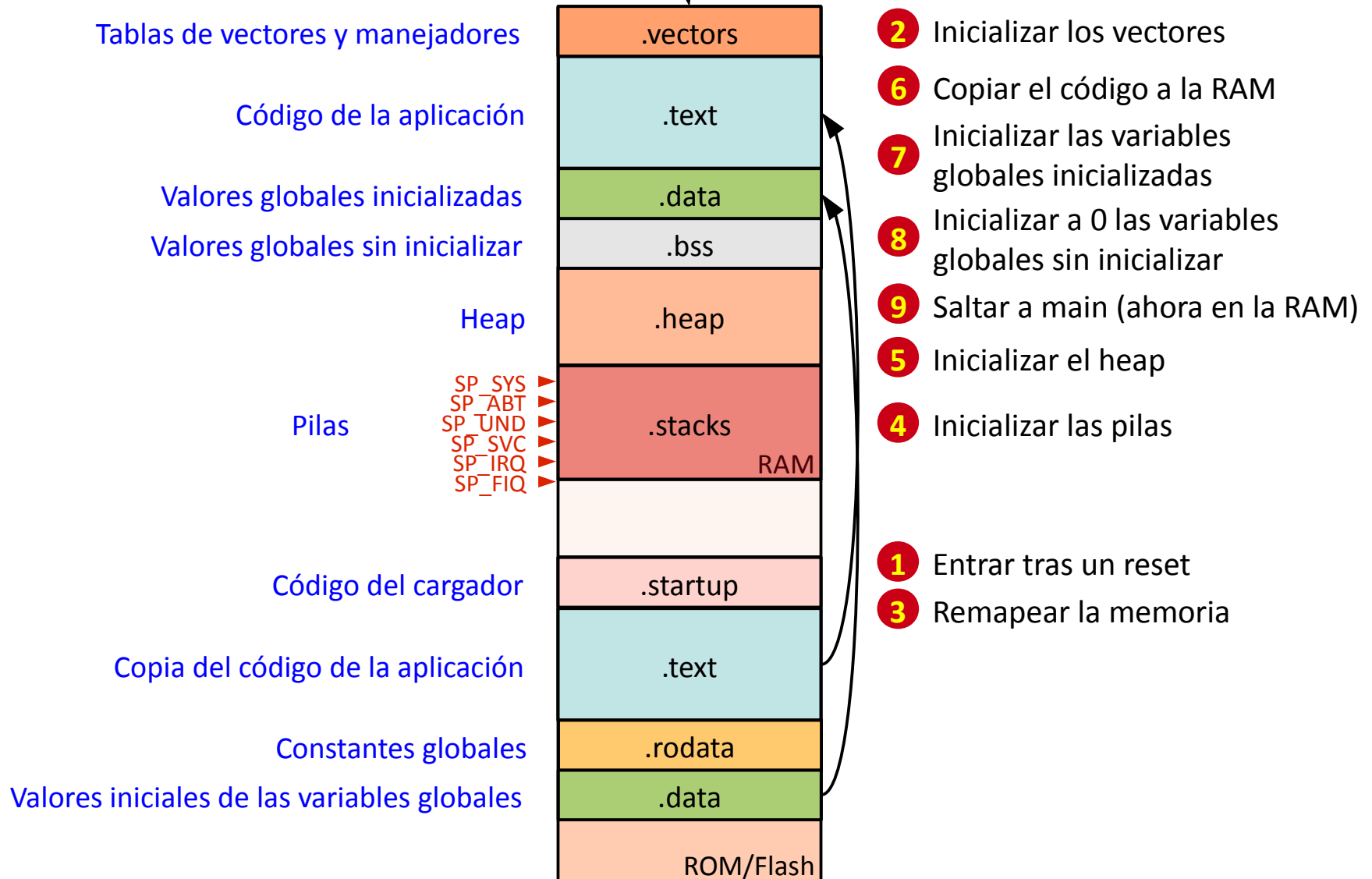
RAM

- 1 Entrar tras un reset
- 3 Remapear la memoria

- 2 Inicializar los vectores

Ejecución de nuestro firmware

_vector_table=0x00000000
tras el remapeo



Contenidos

Tema 3: Cargador de arranque

- Necesidad de un boot loader en cualquier sistema empujado

- Entrada tras el reset y salto a la aplicación

- Soporte para variables globales

- Soporte para funciones y variables locales

- Carga de la aplicación en la RAM

- Remapeo de la memoria

- Soporte para excepciones

- Soporte para memoria dinámica

- Soporte para todos los modos de ejecución

- Inicialización de los dispositivos

El boot loader paso a paso: Cambio a modo user

Entrada tras un reset

Inicializar los dispositivos críticos

Dispositivos del sistema, como el controlador de interrupciones, controladores de memorias externas, caches, MMU, etc.

Inicializar los vectores de excepción

Remapear la memoria

Inicializar las pilas y el heap

Detectar el origen de la aplicación

Cargar la aplicación en memoria RAM

Inicializar las variables en memoria RAM

Inicializar el resto de dispositivos

Resto de dispositivos

Habilitar interrupciones

Cambiar a modo user

Saltar a main

Bucle infinito

La inicialización de los dispositivos es totalmente dependiente de la plataforma

@ Inicialización de la plataforma

```
ldr    ip, =bsp_init
mov    lr, pc
bx     ip
```

La inicialización se puede hacer en C

Ejemplo de inicialización del sistema

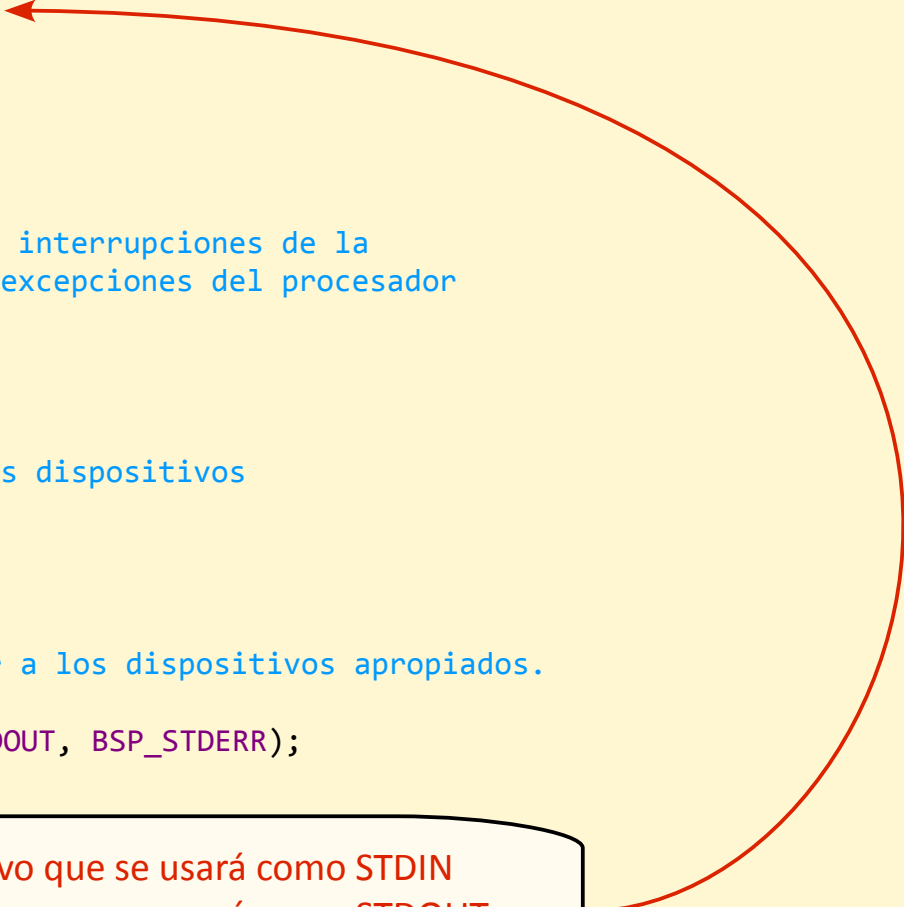
Inicialización del BSP

```
/*
 * Configuración de E/S estándar
 */
#define BSP_STDOUT    "/dev/uart1"
#define BSP_STDIN     "/dev/uart1"
#define BSP_STDERR    "/dev/uart1"

void bsp_init (void)
{
    /*
     * Inicializamos el controlador de interrupciones de la
     * plataforma y la gestión de las excepciones del procesador
     */
    bsp_excep_init();

    /*
     * Inicializamos los drivers de los dispositivos
     */
    bsp_sys_init();

    /*
     * Redireccionamos la E/S estándar a los dispositivos apropiados.
     */
    bsp_io_redirect(BSP_STDIN, BSP_STDOUT, BSP_STDERR);
}
```



BSP_STDIN	→	Dispositivo que se usará como STDIN
BSP_STDOUT	→	Dispositivo que se usará como STDOUT
BSP_STDERR	→	Dispositivo que se usará como STDERR

Lecturas recomendadas

Cargador de arranque:

Q. Li, C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003. Capítulo 3

A. N. Sloss, D. Symes y C. Wright. *ARM Systems Developer's Guide. Designing and Optimizing System Software*. Morgan Kaufmann, 2004. Capítulo 10

M. Samek. *Building Bare-Metal ARM Systems with GNU: Part 1 Getting Started*. Embedded.com, 2007.

M. Samek. *Building Bare-Metal ARM Systems with GNU: Part 2 Startup Code and Low-level Initialization*. Embedded.com, 2007.

Atmel. *AT91 Assembler Code Startup Sequence for C Code Applications Software*, 2002.
<http://www.atmel.com/Images/doc2644.pdf>

J. Bennett. *Howto: Porting Newlib. A Simple Guide*, 2010. Capítulo 5.2
<http://www.embecosm.com/download/ean9.html>

Rob Savoye. *Embed With GNU. Porting The GNU Tools To Embedded Systems*. Cygnus Support, 1995. Capítulo 3 <http://www.gnuarm.com/pdf/porting.pdf>