

Sistemas Empotrados

Tema 3: Cargador de arranque

Lección 8:
Diseño de un boot loader básico



Contenidos

Tema 3: Cargador de arranque

Necesidad de un boot loader en cualquier sistema empotrado

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

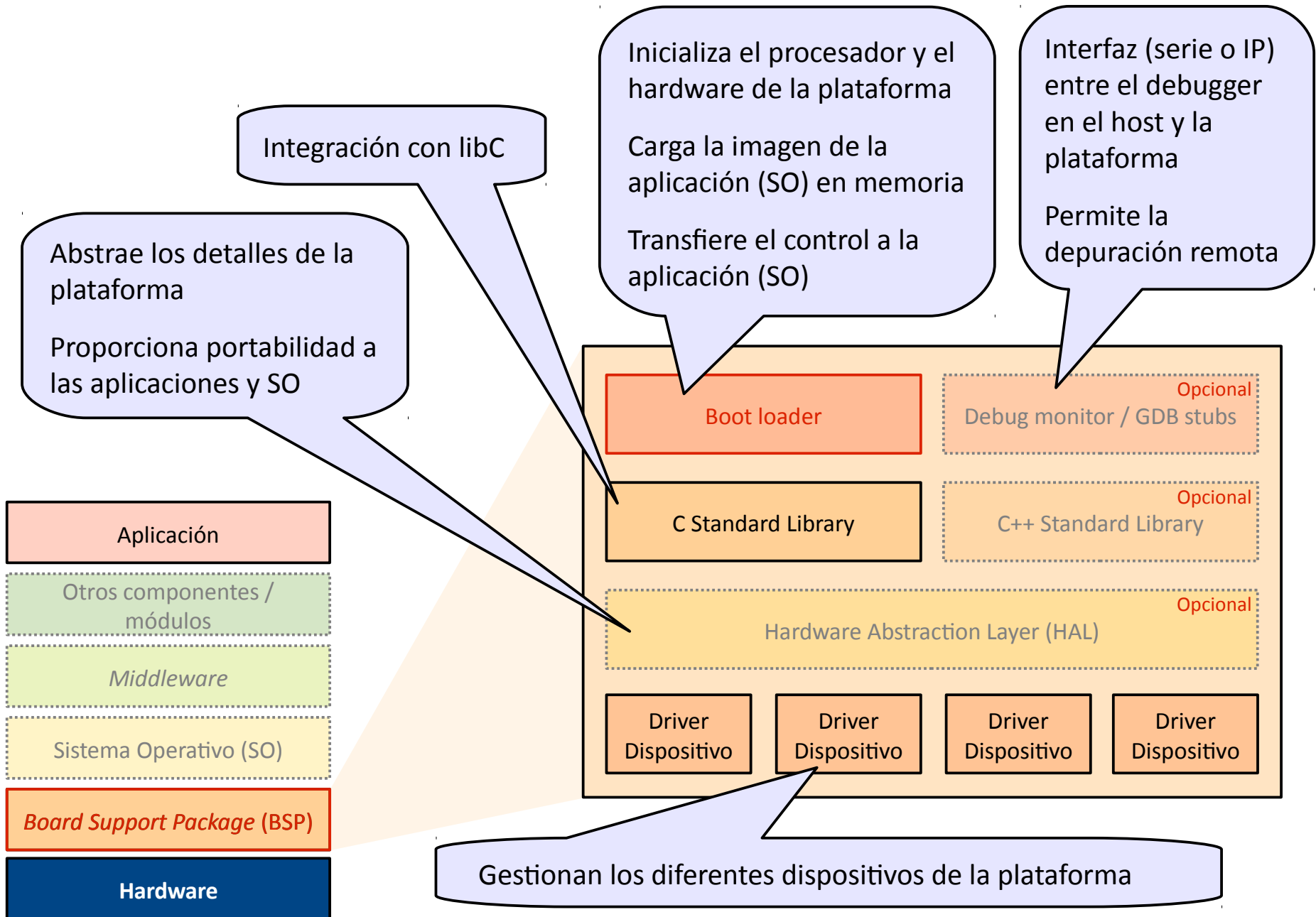
Soporte para excepciones

Soporte para memoria dinámica

Soporte para todos los modos de ejecución

Inicialización de los dispositivos

Componentes del *Board Support Package* (BSP)



Necesidad de un *boot loader*

Una aplicación en C asume que...

- El programa empieza en la función main, pero..., ¿quién llama a la función main?
- Se puede hacer llamadas a funciones, luego..., es necesario crear e inicializar una pila
- Se puede usar memoria dinámica, luego..., es necesario crear e inicializar un heap
- Se pueden usar variables globales, pero..., ¿quien las inicializa antes de llamar a la función main?
- Existe E/S estándar (printf asume que existe stdout), pero..., ¿quien los define e inicializa?
- Los periféricos van a funcionar adecuadamente, pero..., ¿quién los inicializa?
- Hay soporte de excepciones e interrupciones, pero..., ¿quién los inicializa?

El cargador de arranque (*boot loader*)

- Se ejecuta tras un reset del sistema
- Inicializa las memorias, los dispositivos y controladores del sistema
- Configura el entorno de ejecución de C (C Runtime, crt0)
- Copia el código de la aplicación y sus datos (de la ROM, flash, usb, etc.) a la memoria RAM
- Llama a la función main de la aplicación
- Algunas de estas tareas sólo se pueden hacer en ensamblador

Desarrollo sin *runtime* de C

Hola mundo en C sin runtime de C

```
/* Registro de control de dirección del GPIO32-GPIO63 */
#define REG_GPIO_PAD_DIR1      ((volatile int *) 0x80000004)

/* Registro de activación de bits del GPIO32-GPIO63 */
#define REG_GPIO_DATA_SET1     ((volatile int *) 0x8000004c)

/* Registro de limpieza de bits del GPIO32-GPIO63 */
#define REG_GPIO_DATA_RESET1   ((volatile int *) 0x80000054)

/* El led rojo está en el GPIO 44 */
#define LED_RED_MASK           (0x00001000)

/* Retardo para la espera */
#define RETARDO                 (0x100000)
```

No hay variables globales,
hay que usar #define

```
void _start(void)
{
```

El linker debe colocar _start en la dirección 0x00000000

```
    register int i;
```

No hay variables locales, hay que usar registros

```
    /* Configuramos el GPIO44 para que sea de salida */
    *REG_GPIO_PAD_DIR1 = LED_RED_MASK;
```

Acceso directo a los puertos
de E/S mediante punteros

```
    while (1)
    {
```

```
        *REG_GPIO_DATA_SET1 = LED_RED_MASK;
        for (i=0 ; i<RETARDO ; i++);
```

```
        *REG_GPIO_DATA_RESET1 = LED_RED_MASK;
        for (i=0 ; i<RETARDO ; i++);
```

No hay funciones, hay que volver
a copiar el código para el retardo

```
    }
```

```
}
```

Desarrollo sin *runtime* de C

Linker script

```
/*  
 * Punto de entrada del programa  
 */
```

```
ENTRY(_start)
```

El programa empieza en la etiqueta `_start`

```
/*  
 * Mapa de memoria de la placa  
 */
```

```
MEMORY
```

```
{  
    flash : org = 0x00000000, len = 0x00100000  
}
```

El mapa variará de una plataforma a otra, pero al arrancar, siempre existirá una ROM o una Flash mapeada a la dirección 0x00000000

```
/*  
 * Secciones de salida  
 */
```

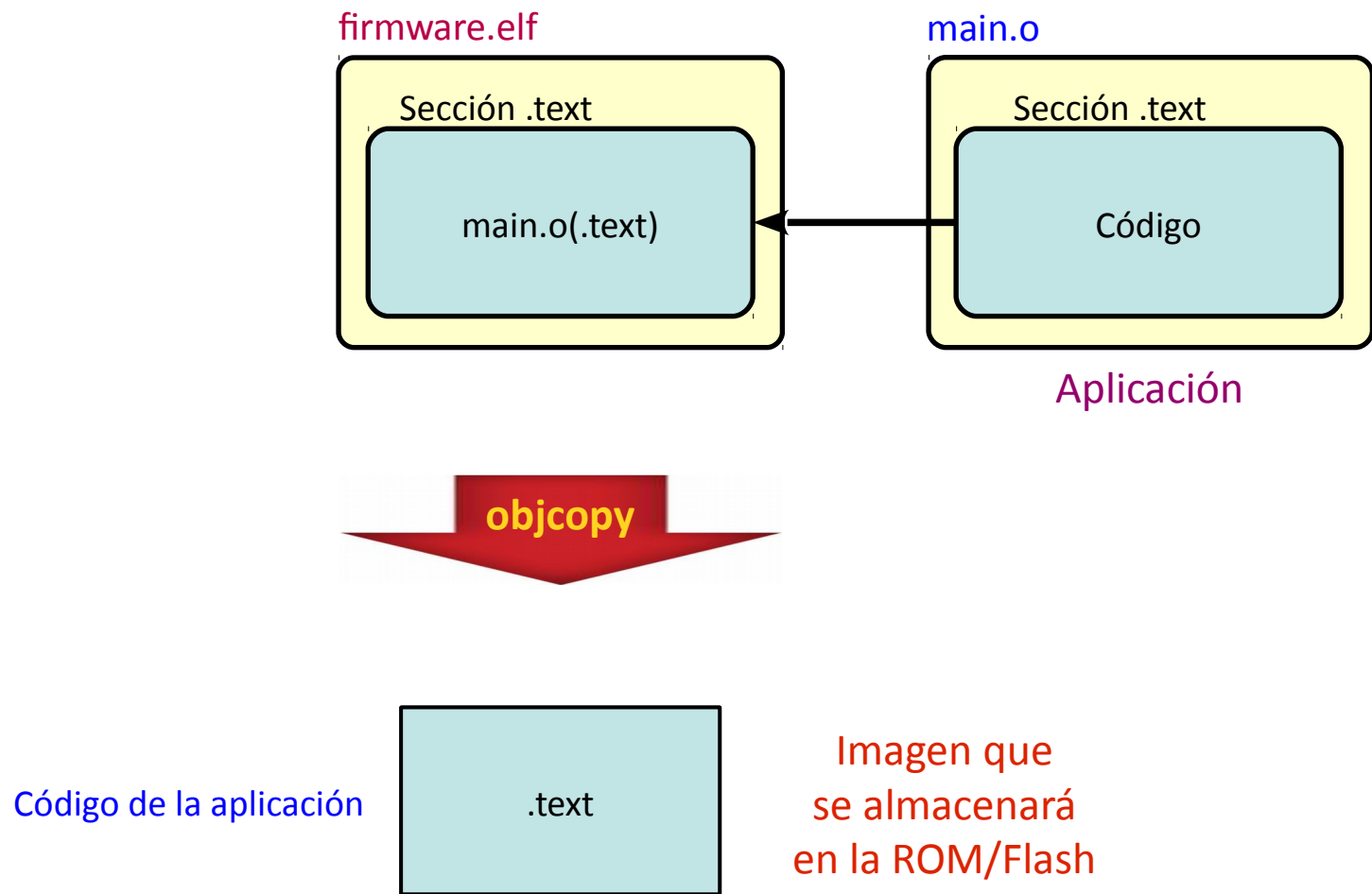
```
SECTIONS
```

```
{  
    /* Código del programa */  
    .text :  
    {  
        *(.text);  
    } > flash  
}
```

Ejecutamos desde la flash. $VMA = LMA$

Como la primera instrucción del código es la que tiene la etiqueta `_start`, y la Flash está mapeada a la dirección 0x00000000, hemos asignado la dirección 0x00000000 a la etiqueta `_start`

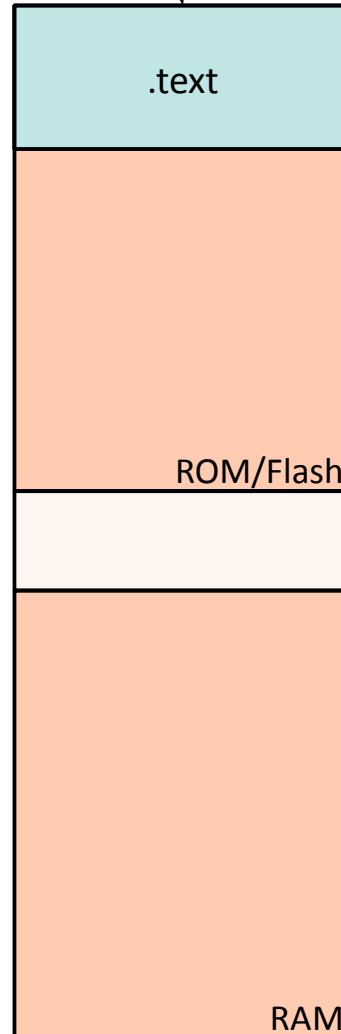
Estructura de nuestro firmware



Ejecución de nuestro firmware

`_start=0x00000000`

Código de la aplicación



Como de momento no
usamos funciones ni
variables, y ejecutamos
la aplicación desde la
ROM/Flash, no usamos
la RAM del sistema

Contenidos

Tema 3: Cargador de arranque

Necesidad de un boot loader en cualquier sistema empotrado

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: Entrada tras el reset

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

Al arrancar el sistema, el procesador está en el modo privilegiado supervisor (SVC) y PC = 0x00000000

La dirección 0 debe estar mapeada a una memoria no volátil que contenga los vectores de excepción

Tras los vectores de excepción se puede poner una nota de copyright, para que aparezca al principio de la ROM/flash

@ Sección de código de arranque

```
.code 32
.section .startup, "xa"
```

Definimos una sección para el crt

@ Vectores de excepción en la ROM

```
.global _reset
_reset: b _start
_und: b .
_swi: b .
_pabt: b .
_dabt: b .
_rsv: b .
_irq: b .
_fiq: b .
```

El linker debe colocar _reset en la dirección 0x00000000

@ Nota del copyright al principio de la ROM

```
.string "Copyright (C) UGR"
.align 4
```

@ Comienza el cargador

```
.global _start
.type _start, %function
_start:
```

El boot loader paso a paso: Salto a main

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

```
@ Salto a main
ldr    ip, =main

@ lr guarda la dirección de retorno
@ pc apunta 2 instrucciones más abajo
mov    lr, pc

@ Salto global a la función main
bx     ip

@ Colgamos el sistema si main retorna
b      .
```

El código de la aplicación
puede estar en ARM o Thumb

Nuestro programa C ya empieza en main, aunque seguimos sin soporte de variables (ni locales ni globales) y sin llamadas a función

Main no debería retornar. Si lo hiciera, entramos en un bucle infinito

Ya se entra desde la función main

Hola mundo en C

```
/* Registro de control de dirección del GPIO32-GPIO63 */
#define REG_GPIO_PAD_DIR1      ((volatile int *) 0x80000004)

/* Registro de activación de bits del GPIO32-GPIO63 */
#define REG_GPIO_DATA_SET1     ((volatile int *) 0x8000004c)

/* Registro de limpieza de bits del GPIO32-GPIO63 */
#define REG_GPIO_DATA_RESET1   ((volatile int *) 0x80000054)

/* El led rojo está en el GPIO 44 */
#define LED_RED_MASK           (0x00001000)

/* Retardo para la espera */
#define RETARDO                 (0x100000)
```

Seguimos sin variables globales, hay que usar #define

```
int main (void) {
    register int i;
```

Seguimos sin variables locales, hay que usar registros

```
/* Configuramos el GPIO44 para que sea de salida */
*REG_GPIO_PAD_DIR1 = LED_RED_MASK;
```

Acceso directo a los puertos de E/S mediante punteros

```
while (1)
{
    *REG_GPIO_DATA_SET1 = LED_RED_MASK;
    for (i=0 ; i<RETARDO ; i++);

    *REG_GPIO_DATA_RESET1 = LED_RED_MASK;
    for (i=0 ; i<RETARDO ; i++);
}
return 0;
}
```

Seguimos sin funciones, hay que volver a copiar el código para el retardo

Entrada tras el reset y salto a la aplicación

Linker script

```
/*  
 * Punto de entrada del cargador  
 */
```

```
ENTRY(_reset)
```

El cargador empieza en el vector de `_reset`

```
/*  
 * Mapa de memoria de la placa  
 */
```

```
MEMORY
```

```
{  
    flash : org = 0x00000000, len = 0x00100000  
}
```

```
/*  
 * Secciones de salida  
 */
```

```
SECTIONS
```

```
{
```

```
/* Código del cargador */
```

```
.startup :
```

```
{  
    *(<code>.startup);
```

```
} > flash
```

Código del fichero crt.o.

El vector de reset es la primera instrucción del cargador, que está mapeado al principio de la Flash, a la dirección 0x00000000

De momento sólo salta a main

El cargador se ejecuta desde la flash. `VMA = LMA`

```
/* Código del programa */
```

```
.text :
```

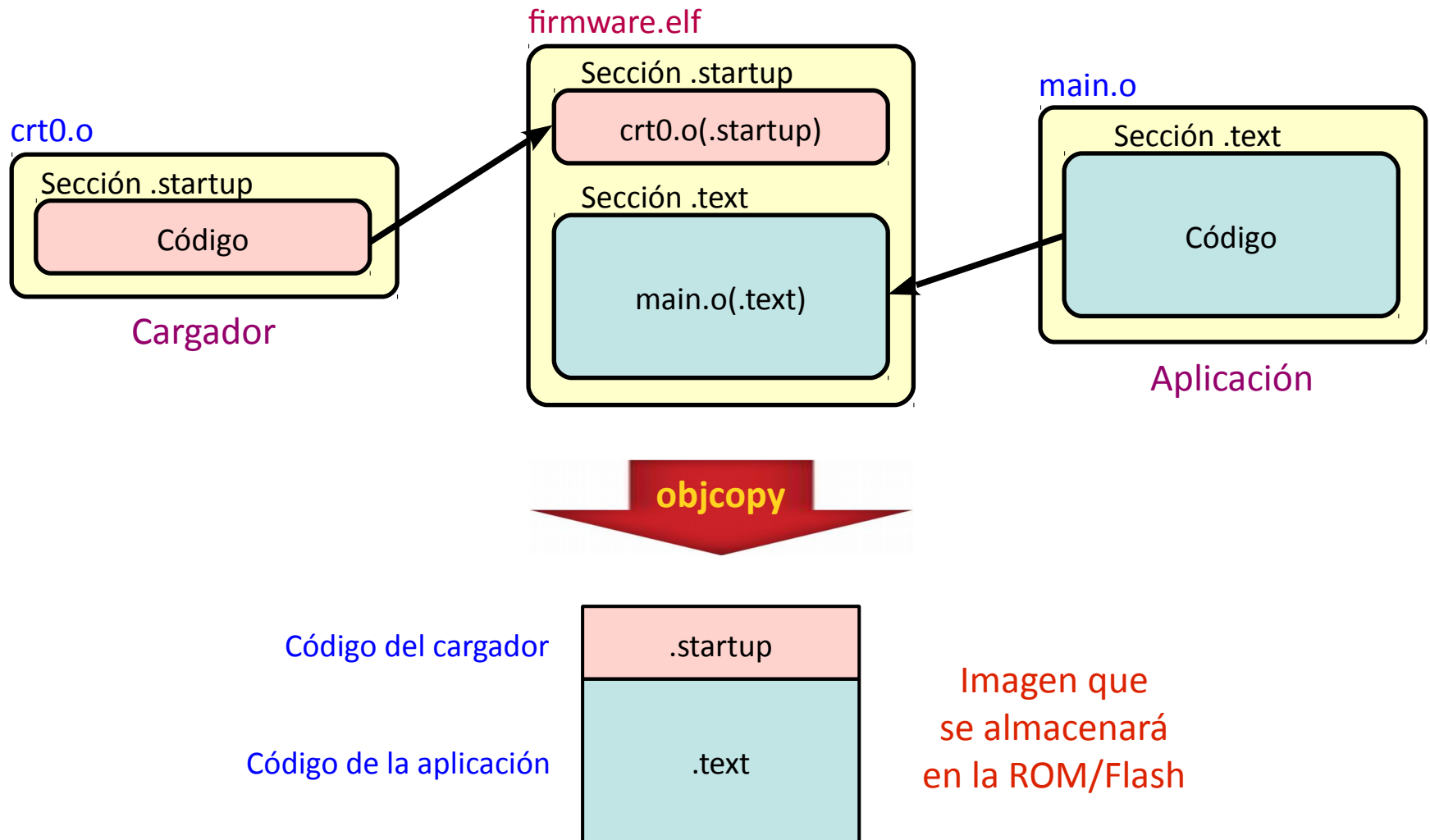
```
{  
    *(<code>.text);
```

```
} > flash
```

Código del resto de ficheros objeto

La aplicación se ejecuta desde la flash. `VMA = LMA`

Estructura de nuestro firmware

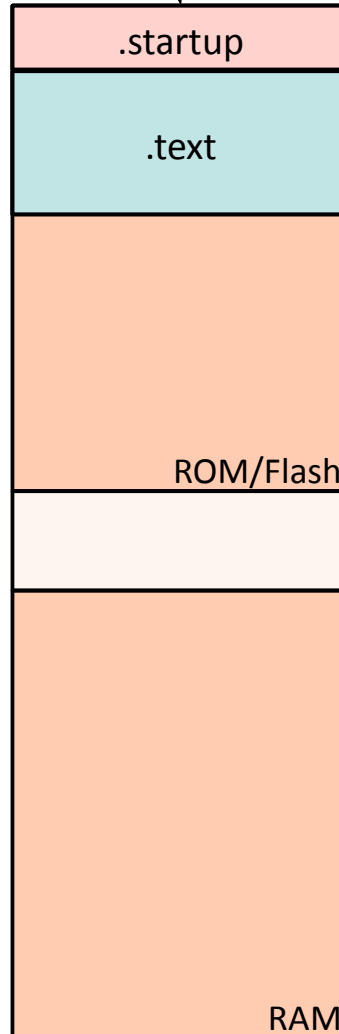


Ejecución de nuestro firmware

_reset=0x00000000

Código del cargador

Código de la aplicación



1 Entrar tras un reset

2 Saltar a main

Como de momento no
usamos funciones ni
variables, y ejecutamos
la aplicación desde la
ROM/Flash, no usamos
la RAM del sistema

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 variables globales

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

El compilador emite las variables globales inicializadas a cero a la sección .bss y las variables sin inicializar a la sección COMMON.

El enlazador genera direcciones VMA en la RAM para las variables de las secciones .bss y COMMON.

El cargador debe inicializar a cero todas estas variables en sus correspondientes direcciones en la RAM

@ Fijamos a 0 la memoria RAM reservada para las
@ variables sin inicializar

```
ldr    a1, =_bss_start
ldr    a2, =_bss_end
ldr    a3, =0
bl     _ram_init
```

Direcciones
generadas
por el linker

@ Rutina para inicializar una zona de memoria RAM
@ a1: Posición inicial en la memoria RAM
@ a2: Posición final en la memoria RAM
@ a3: Valor para la inicialización

```
.type    _ram_init, %function
_ram_init:
    cmp    a1, a2
    strne  a3, [a1], #+4
    bne    _ram_init

    mov    pc, lr
```

El boot loader paso a paso: Soporte para variables globales

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

El compilador emite las variables globales inicializadas a la sección .data

El enlazador genera direcciones VMA para las variables en la RAM y direcciones LMA para sus valores iniciales, que están en la imagen del programa, en la ROM/Flash

El cargador deberá copiar los valores iniciales de las variables de la ROM/Flash a su correspondiente dirección de memoria en la RAM

@ Copiamos los valores de las variables de su
@ dirección LMA a la VMA

```
ldr    a1, =_data_start
ldr    a2, =_data_end
ldr    a3, =_data_flash_start
bl     _ram_copy
```

Direcciones
generadas
por el linker

@ Rutina para copiar bloques de memoria

@ a1: Dirección inicial en la RAM

@ a2: Dirección final en la RAM

@ a3: Dirección inicial en la ROM

.type _ram_copy, %function

_ram_copy:

```
cmp    a1, a2
bge    1f
ldrb   a4, [a3], #+1
strb   a4, [a1], #+1
b      _ram_copy
```

1: mov pc, lr

El boot loader paso a paso: Soporte para variables globales

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

El compilador emite las constantes globales a las secciones
.rodata, .rodata.str, etc.

De momento dejaremos las constantes globales en la
ROM/Flash, ya que no van a cambiar a lo largo de la
ejecución de la aplicación

Ya podemos usar variables globales

Hola mundo en C

```
/* Registro de control de dirección del GPIO32-GPIO63 */
#define REG_GPIO_PAD_DIR1      ((volatile int *) 0x80000004)

/* Registro de activación de bits del GPIO32-GPIO63 */
#define REG_GPIO_DATA_SET1     ((volatile int *) 0x8000004c)

/* Registro de limpieza de bits del GPIO32-GPIO63 */
#define REG_GPIO_DATA_RESET1   ((volatile int *) 0x80000054)

/* El led rojo está en el GPIO 44 */
#define LED_RED_MASK           (0x00001000)

/* Variables globales */
int retardo1 = 0x100000;      /* Variable inicializada -> .data */
const int retardo2 = 0x10000; /* Constante -> .rodata */
int i = 0;                   /* Variable inicializada a cero -> .bss */
int j;                       /* Variable sin inicializar -> COMMON */

int main (void) {
    /* Configuramos el GPIO44 para que sea de salida */
    *REG_GPIO_PAD_DIR1 = LED_RED_MASK;

    while (1) {
        *REG_GPIO_DATA_SET1 = LED_RED_MASK;
        for (i=0 ; i<retardo1 ; i++);

        *REG_GPIO_DATA_RESET1 = LED_RED_MASK;
        for (j=0 ; j<retardo2 ; j++);
    }
    return 0;
}
```

Ya tenemos soporte
para variables globales

Usamos variables globales. Todavía no tenemos pila

Seguimos sin funciones, hay que volver
a copiar el código para el retardo

Soporte para variables globales

Linker script

```
MEMORY
{
  flash : org = 0x00000000, len = 0x00100000
  ram   : org = 0x00300000, len = 0x00040000
}
```

Ya usamos la RAM para las VMA de las variables

```
SECTIONS
{
  .rodata : {
    *(.rodata*);
    . = ALIGN(4) ;
  } > flash
```

Las secciones `.startup` y `.text` no cambian. No las repetimos

Como las constantes podrían tener cualquier tamaño, conviene alinear el final de la sección a una frontera de 32 bits

Dejamos las constantes en la ROM/Flash. VMA = LMA

```
  .data : {
    _data_start = . ;
    *(.data);
    . = ALIGN(4) ;
    _data_end = . ;
  } > ram AT > flash
  _data_flash_start = LOADADDR(.data);
```

Direcciones VMA de comienzo y fin de las variables inicializadas en la RAM

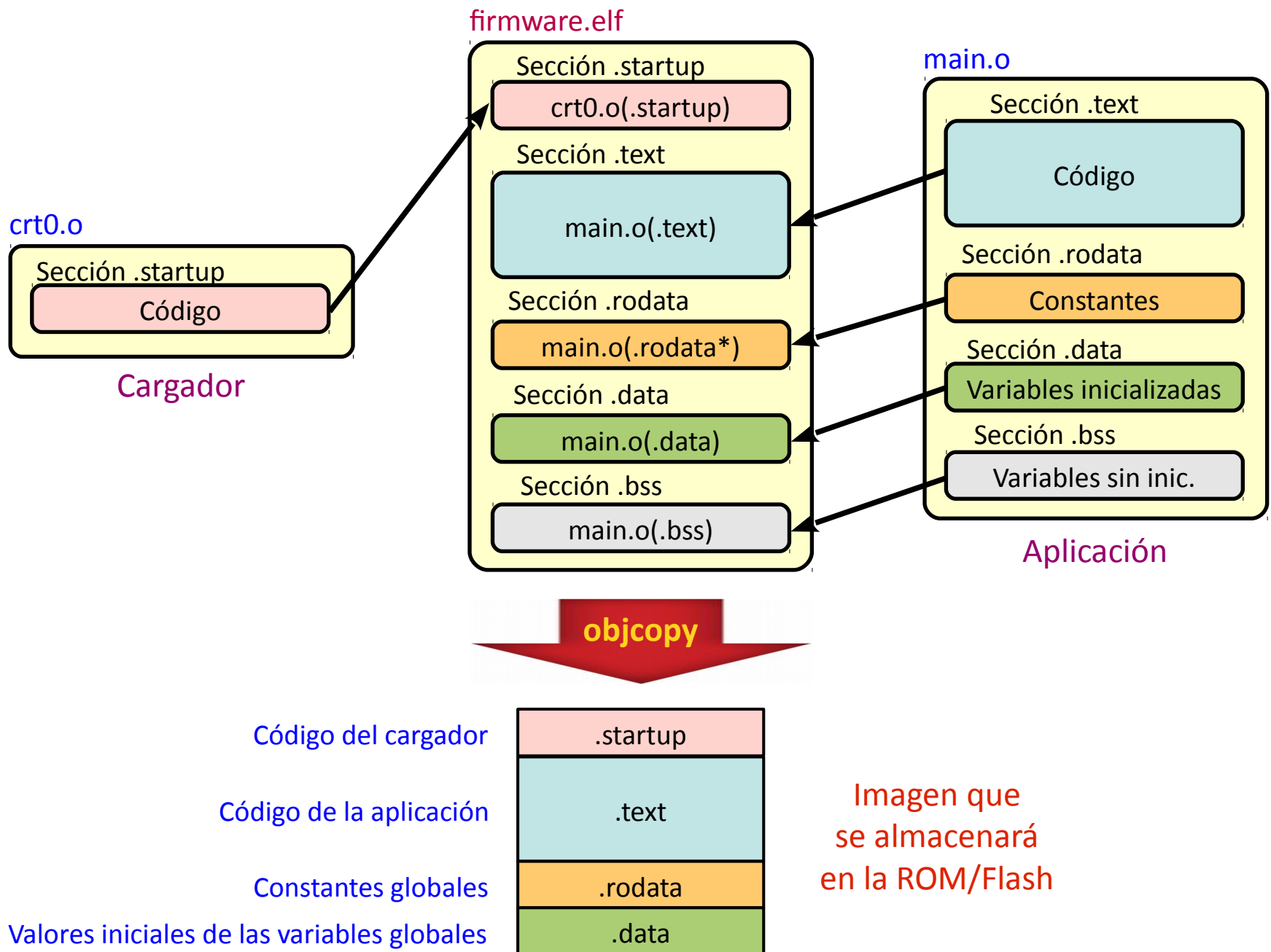
Dirección LMA de comienzo de los valores iniciales de las variables en la Flash/ROM

```
  .bss : {
    _bss_start = . ;
    *(.bss);
    . = ALIGN(4) ;
    *(COMMON);
    . = ALIGN(4) ;
    _bss_end = . ;
  } > ram
```

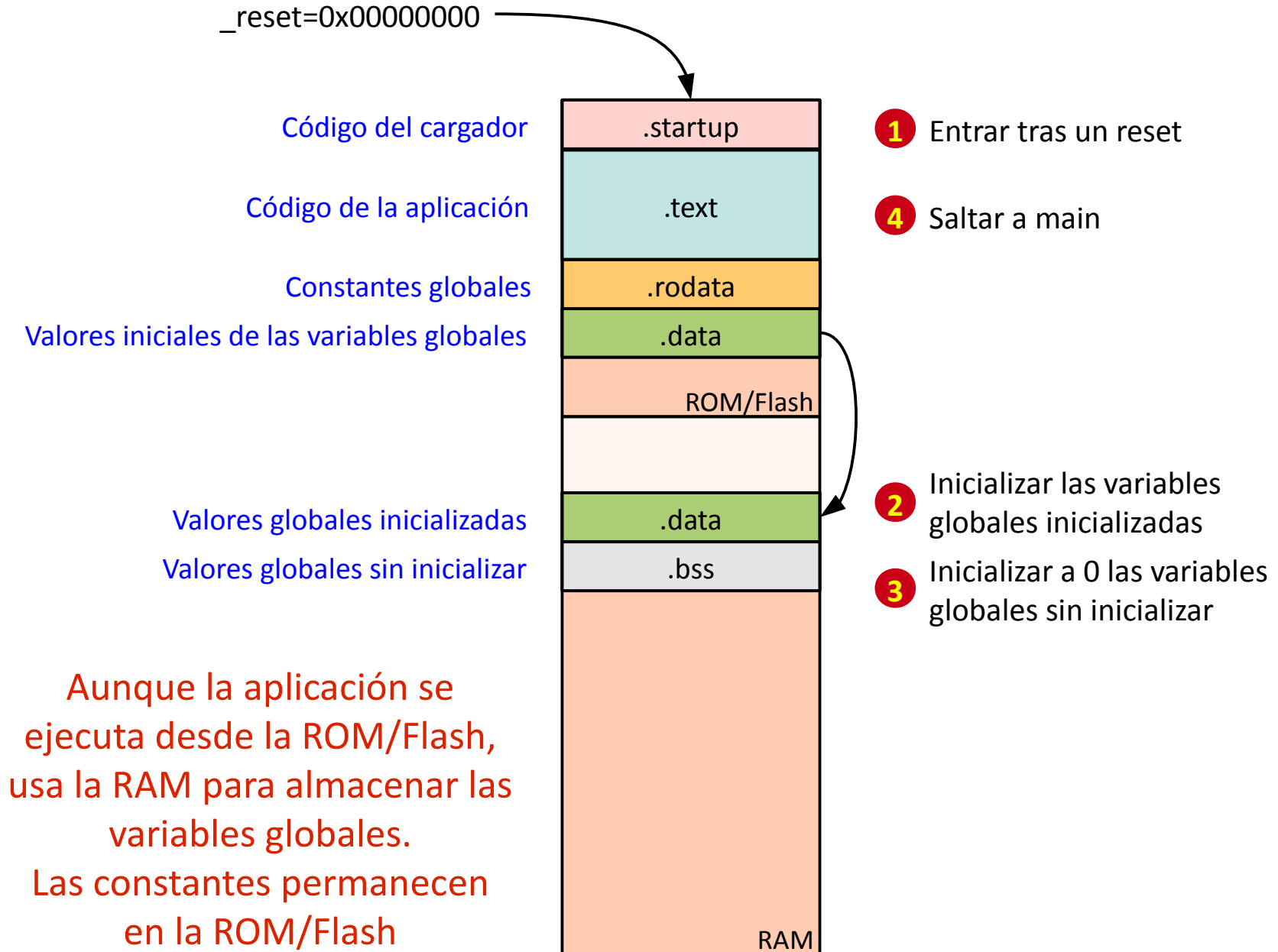
Direcciones VMA de comienzo y fin de las variables sin inicializar en la RAM

Sólo asignamos direcciones VMA en la RAM

Estructura de nuestro firmware



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 funciones y variables locales

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

Necesitamos una pila para realizar el paso de parámetros en las llamadas a función, así como para que las funciones puedan alojar sus variables locales

Es conveniente inicializar la pila a un valor conocido para poder estimar (a posteriori) el uso que se ha hecho de ella, de cara optimizar su tamaño

El cargador debe inicializar la pila y el registro SP para que apunte a su tope

Aunque la arquitectura ARM soporta una pila para cada modo de ejecución, de momento sólo vamos a usar el modo SVC, por lo que sólo creamos una pila

@ Valor para inicializar la pila

```
.set _STACK_FILLER, 0xdeadbeef
```

@ Inicializamos la pila

```
ldr    a1, =_stack_bottom
ldr    a2, =_stack_top
ldr    a3, =_STACK_FILLER
bl     _ram_init
```

```
ldr    sp, =_stack_top
```

Direcciones
generadas
por el linker

Ya podemos usar variables locales y funciones

Hola mundo en C

```
#define REG_GPIO_PAD_DIR1      ((volatile int *) 0x80000004)
#define REG_GPIO_DATA_SET1    ((volatile int *) 0x8000004c)
#define REG_GPIO_DATA_RESET1  ((volatile int *) 0x80000054)
#define LED_RED_MASK          (0x00001000)

/* Variables globales */
int retardo = 0x100000;      /* Variable inicializada -> data */

/* Función para esperar */
void esperar (void)
{
    int i;
    for (i=0 ; i<retardo ; i++);
}

int main ()
{
    /* Configuramos el GPIO44 para que sea de salida */
    *REG_GPIO_PAD_DIR1 = LED_RED_MASK;
    while (1)
    {
        *REG_GPIO_DATA_SET1 = LED_RED_MASK;
        esperar();

        *REG_GPIO_DATA_RESET1 = LED_RED_MASK;
        esperar();
    }

    return 0;
}
```

Ya tenemos soporte para variables locales

Ya podemos llamar a funciones

Soporte para funciones y variables locales

Linker script

```
ENTRY(_reset)

MEMORY
{
    flash : org = 0x00000000, len = 0x00100000
    ram    : org = 0x00300000, len = 0x00040000
}

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

    .text :    { ... } > flash

    .rodata :  { ... } > flash

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

    .bss :     { ... } > ram

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

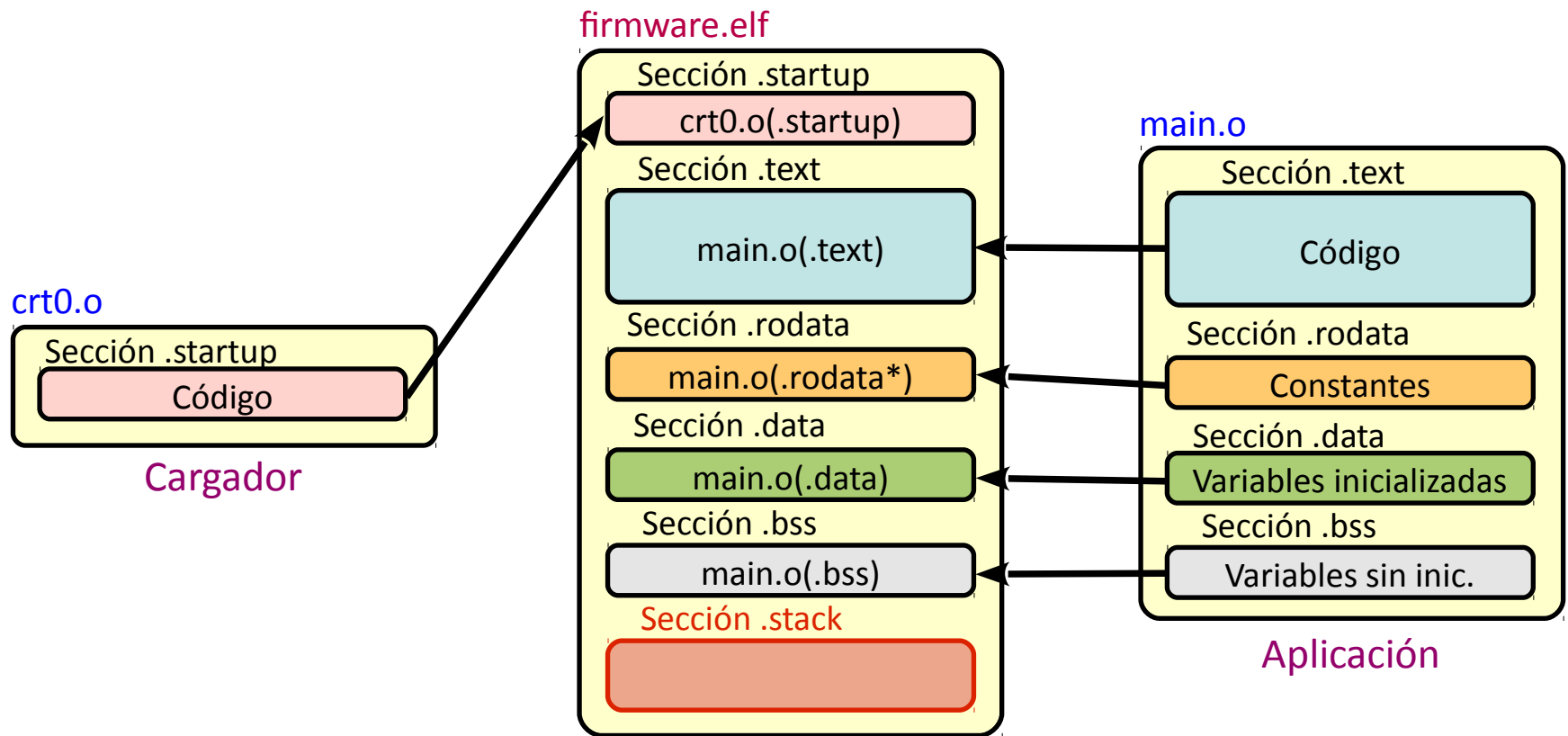
    .stack _ram_limit - _stack_size :
    {
        _stack_bottom = . ;
        . += _stack_size ;
        _stack_top = . ;
    }
}
```

El resto del script no varía

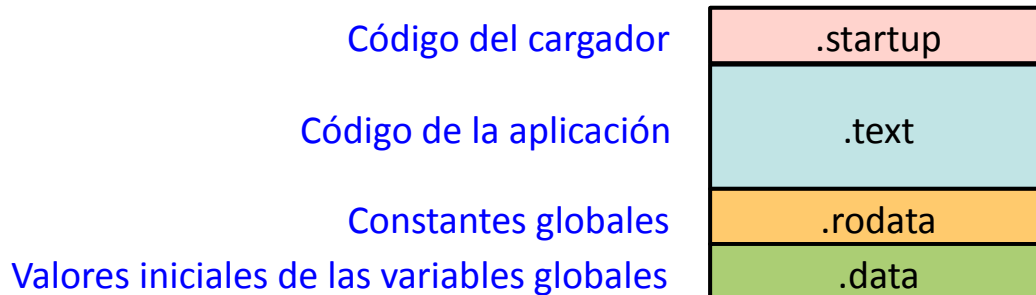
Definimos una zona de 2 KB al final de la RAM para alojar la pila

En este caso nos interesa usar direcciones VMA en vez de regiones de memoria

Estructura de nuestro firmware



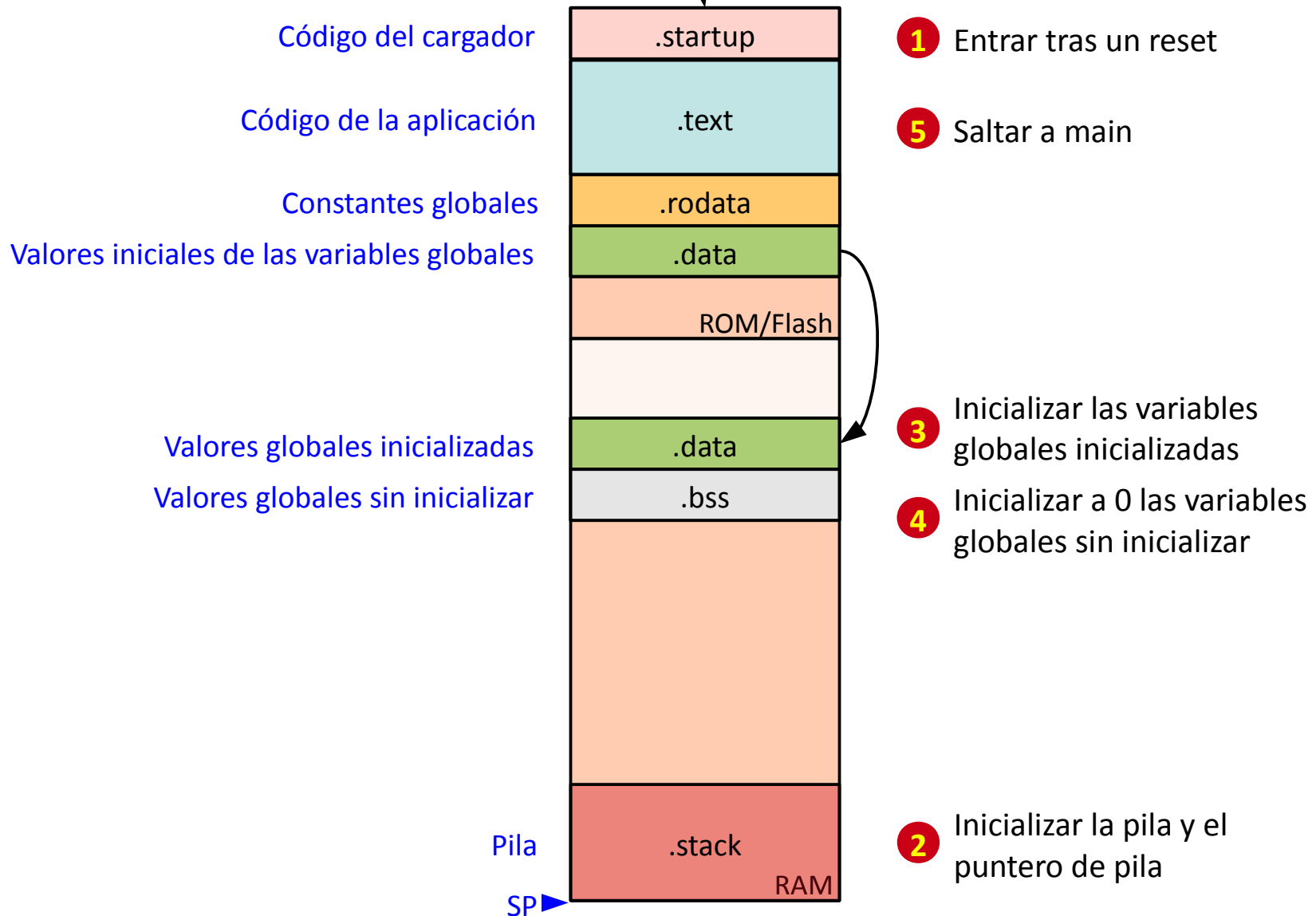
objcopy



La imagen no varía. El linker ha añadido una nueva sección para la pila y el cargador contiene código para inicializarla

Ejecución de nuestro firmware

_reset=0x00000000



Contenidos

Tema 3: Cargador de arranque

- Necesidad de un boot loader en cualquier sistema empotrado

- 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: Carga de la aplicación en la RAM

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

Lo más habitual es que el firmware y el cargador formen parte de la misma imagen grabada en la ROM/Flash

Algunos cargadores proporcionan la posibilidad de dejar el firmware en una flash externa, una tarjeta SD, o bien de descargar el firmware por un puerto serie (UART o USB)

En este último caso, será necesario detectar dónde está el firmware para poder cargarlo a la RAM

Si la aplicación está en una ROM o una NOR Flash se puede ejecutar directamente desde su ubicación. Sin embargo, el tiempo de lectura de estas memorias es significativamente más lento que el de la RAM. La copia a la RAM del código mejorará las prestaciones

Si la aplicación proviene de otra fuente (tarjeta SD, USB, UART, etc.), es necesario copiarla a la RAM para poder ejecutarla

@ Copiamos el código de la aplicación
@ de la ROM/Flash a la RAM

```
ldr    a1, =_text_start
ldr    a2, =_text_end
ldr    a3, =_text_flash_start
bl     _ram_copy
```

Direcciones
generadas
por el linker

Soporte para la carga de la aplicación en la RAM

Linker script

```
ENTRY(_reset)

MEMORY
{
    flash : org = 0x00000000, len = 0x00100000
    ram    : org = 0x00300000, len = 0x00040000
}

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

    .text : {
        _text_start = . ;
        *(.text);
        text_end = . ;
    } > 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;

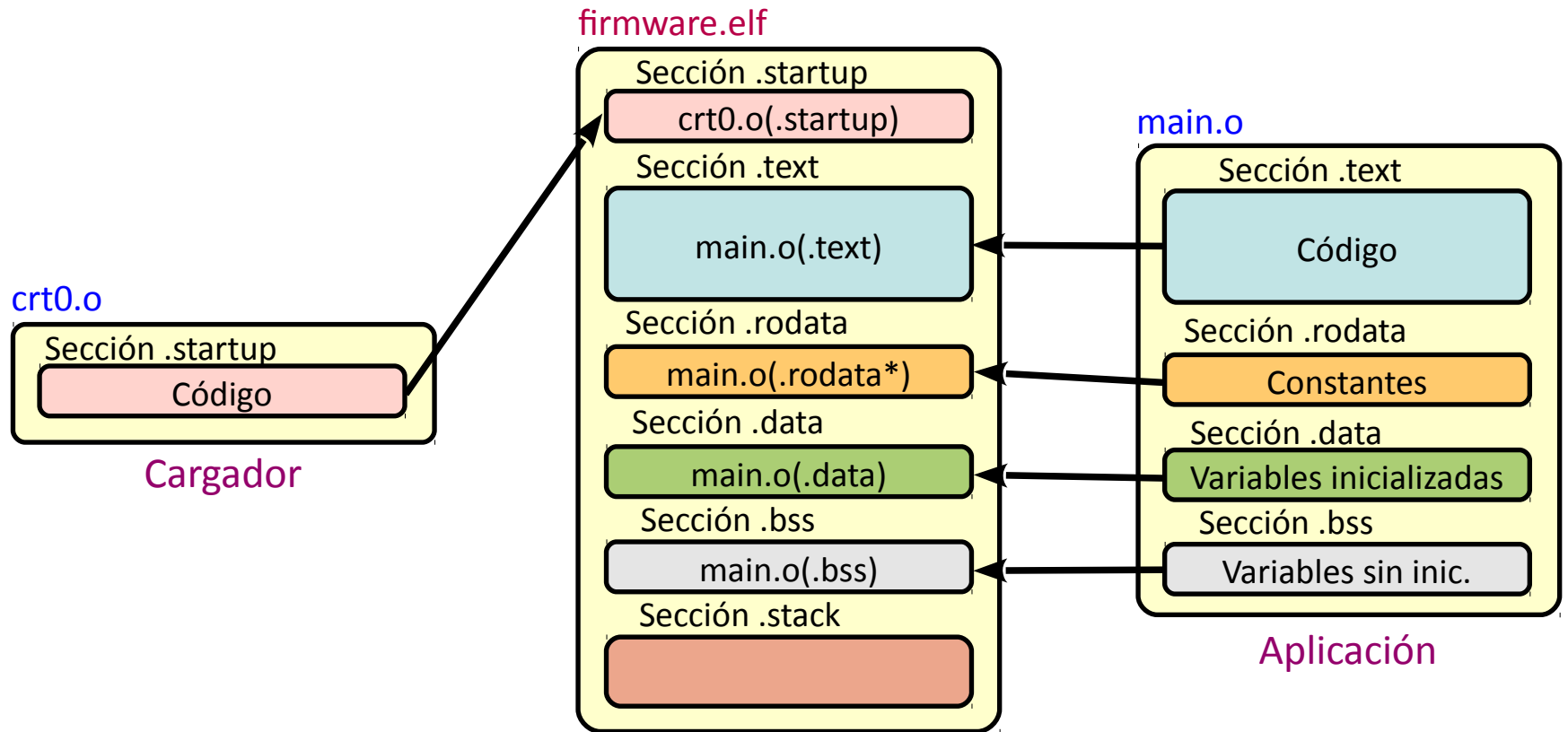
    .stack _ram_limit - _stack_size : { ... }
}
```

Definimos las direcciones VMA de inicio y fin del código en la RAM

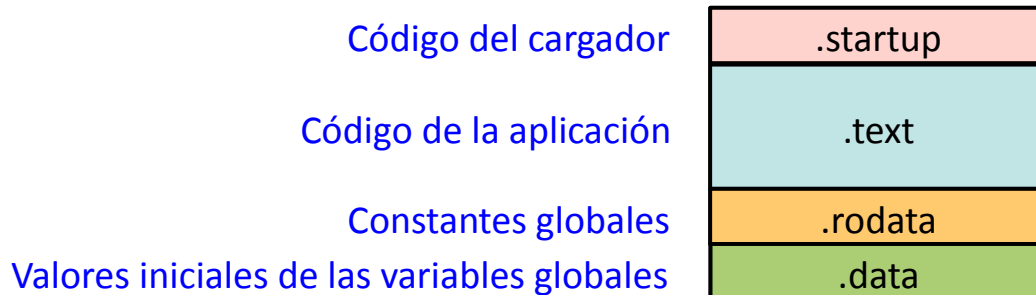
Asignamos direcciones LMA diferentes de las VMA

Dirección LMA de inicio del código en la ROM/Flash

Estructura de nuestro firmware

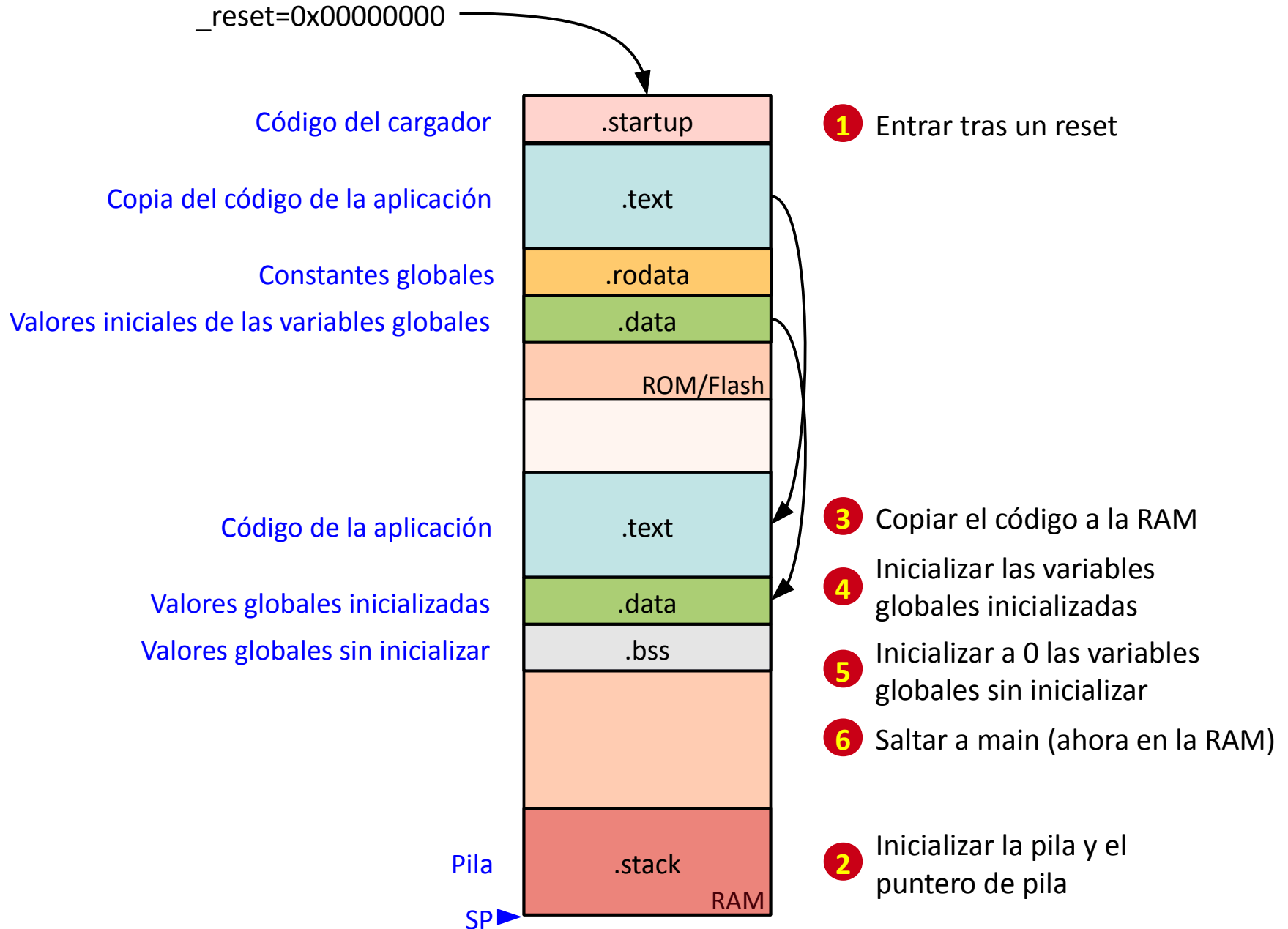


objcopy



Aunque la estructura de la imagen no varía, el código no se puede ejecutar desde su ubicación en la ROM/Flash, ya que el linker a mapeado todas las direcciones a la RAM. Los saltos saltarán a direcciones en la RAM

Ejecución de nuestro firmware



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>