El proceso de compilación en sistemas bare-metal

…y algunos detalles de bajo nivel

- Sistemas bare-metal vs. “hosteados”, Compilación cruzada y Toolchain GNU para ARM
- Proceso de compilación y enlazado
- Secciones de memoria y manipulación mediante Linker script
- Ubicación en memoria de los vectores de excepción
- Iniciación de variables globales
- Enlazado de bibliotecas estándar de C

Sistemas bare-metal vs. “hosteados”, Compilación cruzada y Toolchain GNU para ARM

Sistemas “Hosteados” vs sistemas “bare-metal”

Un Sistema Embebido puede ser …

- de tipo ***Hosteado****:* Con sistema operativo de propósitos generales (Linux/Windows/Andorid…). El desarrollador cuenta con el soporte brindado por el SO para acceder a los distintos recursos del sistema (*File system*, red, gestión de procesos). El sistema SO se encarga de acondicionar la memoria para la ejecución del programa y de liberarla una vez finalizado. Si opta por programar en C/C++, contará con las bibliotecas estándar de ambos lenguajes.
- o ***bare-metal***: No hay un SO de propósitos generales. Puede haber un pequeño RTOS (en general es un módulo más del proyecto). El desarrollador debe incluir en el proyecto el código que inicializa la memoria y gestionar el acceso a los distintos recursos. El uso de las bibliotecas estándar es limitado y deberá “completar” con drivers o código de bajo nivel para operaciones de I/O

Sistemas “Hosteados” vs sistemas “bare-metal”

Un Sistema Embebido puede ser …

- de tipo ***Hosteado****:* Con sistema operativo de propósitos generales (Linux/Windows/Andorid…). El desarrollador cuenta con el soporte brindado por el SO para acceder a los distintos recursos del sistema (*File system*, red, gestión de procesos). El sistema SO se encarga de acondicionar la memoria para la ejecución del programa y de liberarla una vez finalizado. Si opta por programar en C/C++, contará con las bibliotecas estándar de ambos lenguajes.
- o ***bare-metal***: No hay un SO de propósitos generales. Puede haber un pequeño RTOS (en general es un módulo más del proyecto). El desarrollador debe incluir en el proyecto el código que inicializa la memoria y gestionar el acceso a los distintos recursos. El uso de las bibliotecas estándar es limitado y deberá “completar” con drivers o código de bajo nivel para operaciones de I/O

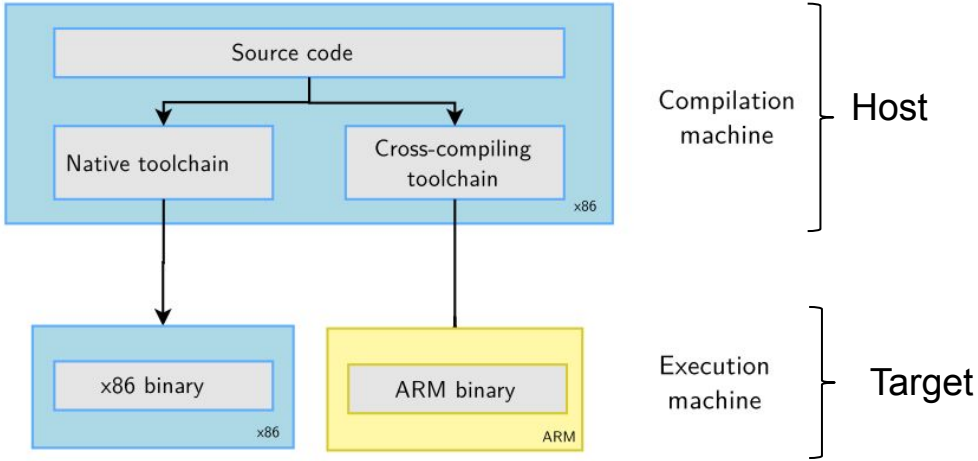
Proceso de desarrollo del firmware

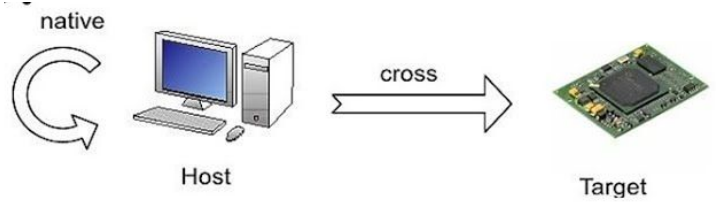
- Diseño del software embebido (herramientas de modelado/diagramas de clases/diagramas de estado/diagramas de flujo/pseudocódigos/elección de bibliotecas y frameworks)
- Codificación (IDE, editor de texto, bibliotecas de terceros)
- Generación del binario ejecutable: “compilación”=compilación+ensamblado+enlazado (toolchain)
- Descarga (Circuito de interfaz para programación + SW para descargar)
- Prueba + Depuración (Circuito de interfaz para debug + SW debugger)

Proceso de desarrollo del firmware

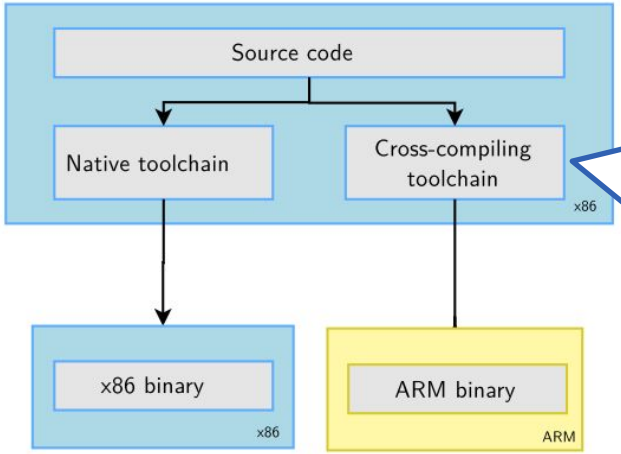
- Diseño del software embebido (herramientas de modelado/diagramas de clases/diagramas de estado/diagramas de flujo/pseudocódigos/elección de bibliotecas y frameworks)
- Codificación (IDE, editor de texto, bibliotecas de terceros)
- **Generación del binario ejecutable**: “compilación”=compilación+ensamblado+enlazado (toolchain)
- Descarga (Circuito de interfaz para programación + SW para descargar)
- Prueba + Depuración (Circuito de interfaz para debug + SW debugger)

Compilación Nativa vs. Cruzada



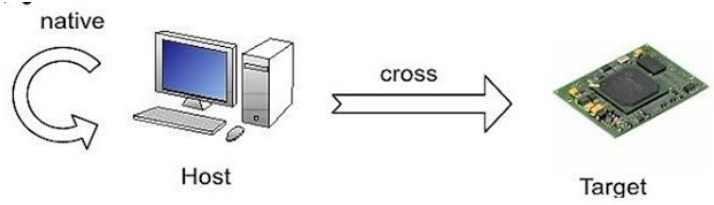


Compilación Nativa vs. Cruzada

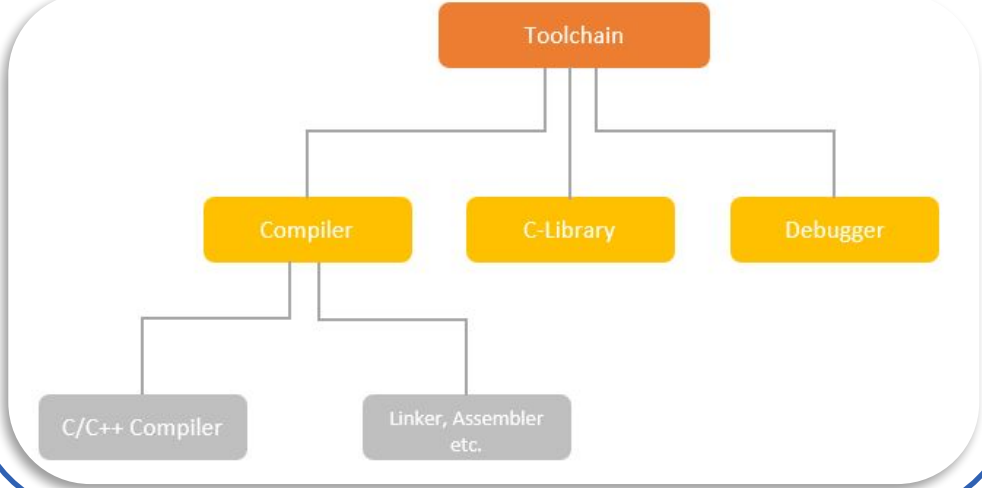


Compilat

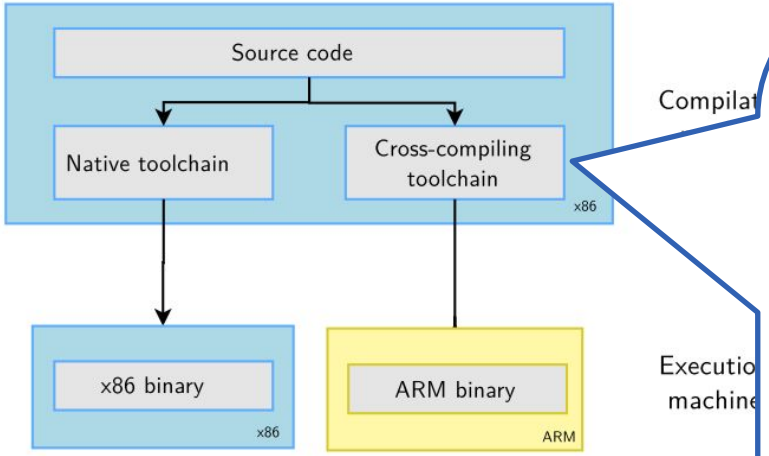
Executio
machine

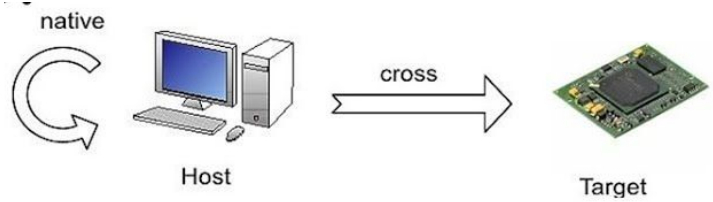


Se requiere un *toolchain* que produzca el binario ejecutable para el embebido



Compilación Nativa vs. Cruzada



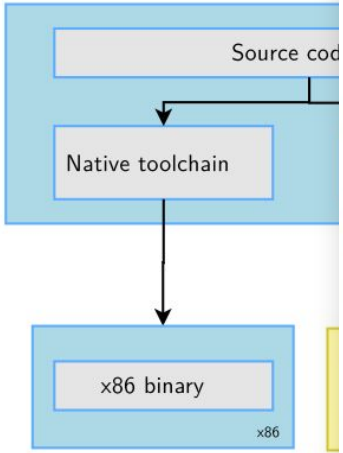


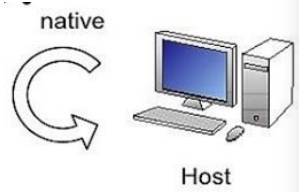
En esta clase tomamos como referencia el ***GNU Arm Embedded Toolchain*** , que es el que usa el STM32CubeIDE y está compuesto por herramientas estándar y abiertas.

Otros *toolchains "similares"* son:

- *IAR*
- *Keil*
- *TI ARM Clang*

Compilación





arm Developer

Downloads

Arm GNU Toolchain is a community supported pre-built GNU compiler toolchain for Arm based CPUs.

Arm GNU Toolchain releases consists of cross toolchains for the following host operating systems:

- GNU/Linux
  - Available for x86_64 and AArch64 host architectures
  - Available for bare-metal and Linux targets
- Windows
  - Available for x86 host architecture only (compatible with x86_64)
  - Available for bare-metal and Linux targets
- macOS
  - Available for x86_64 host architecture only
  - Available for bare-metal targets only

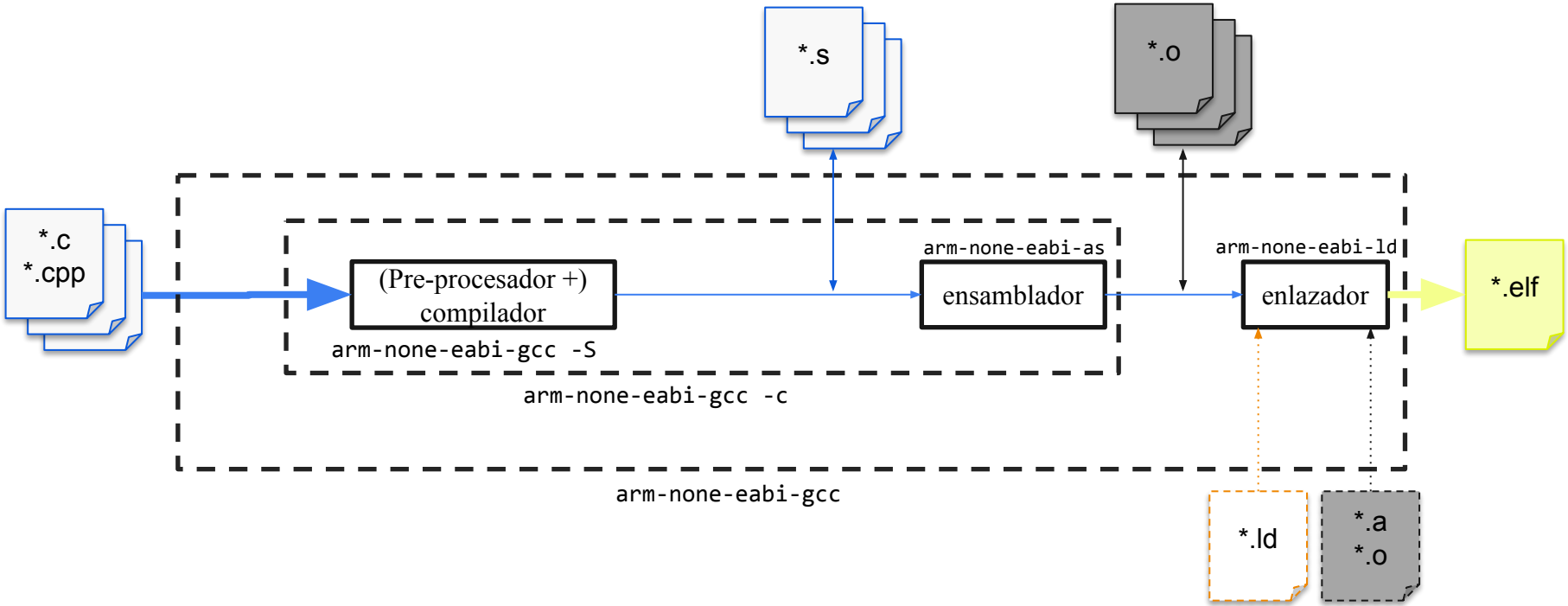
Please download the correct toolchain variant that suits your development needs.

…s como referencia el
…***l Toolchain*** , que es
…CubeIDE y está
…mientas estándar y

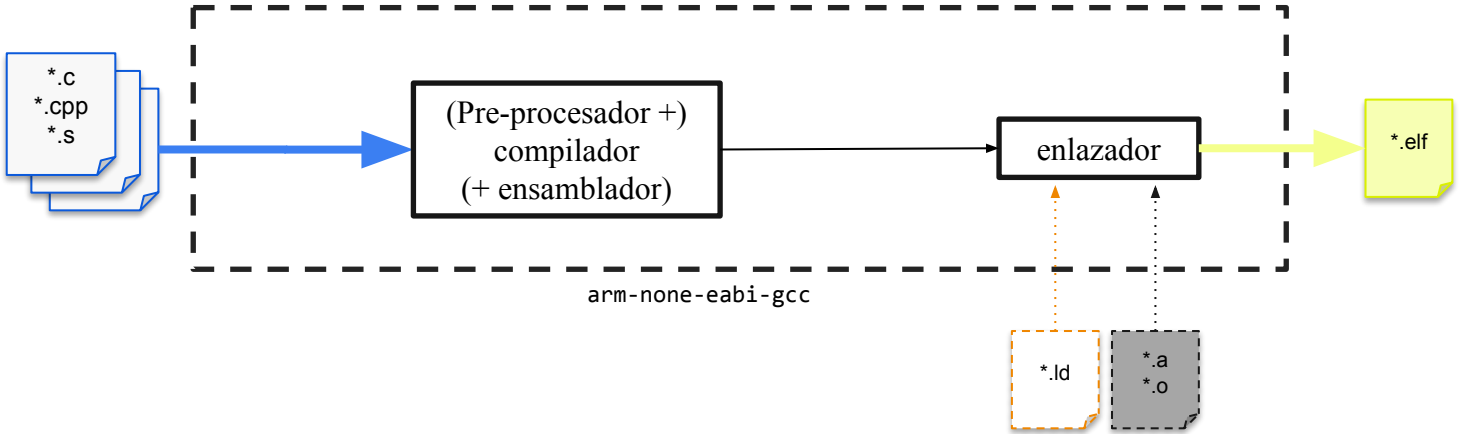
…*lares"* son:

Proceso de compilación y enlazado

Fases del proceso de construcción del binario ejecutable



Construcción directa

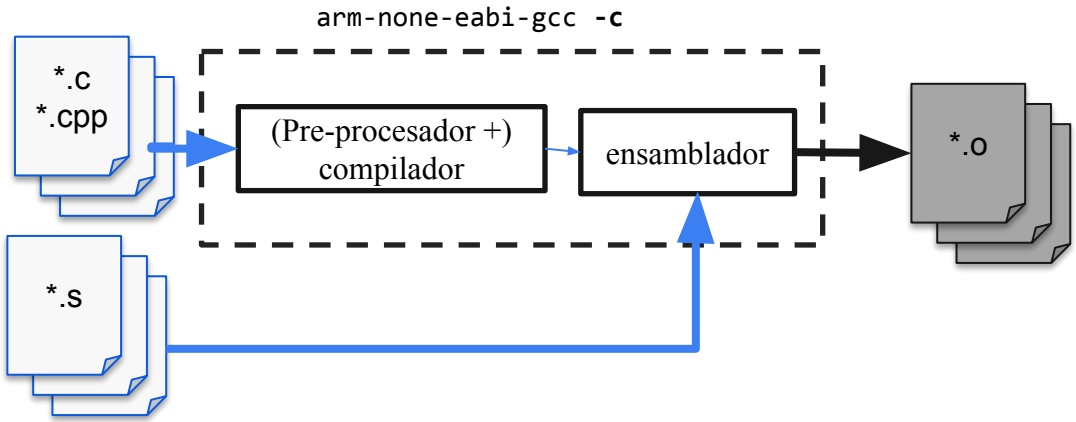


Ej:

```
arm-none-eabi-gcc fuente1.c fuente2.cpp fuente3.s mod_precomp.o -llibreria -o ejecutable.elf  -T script.ld
```

Uso típico (**compilación** y enlazado separados)

1. Compilación / ensamblado del código fuente: Se interrumpe el proceso antes del enlazado y se genera un archivo de código objeto (*.o) por cada archivo de código (*. c,*.cpp, *.s)

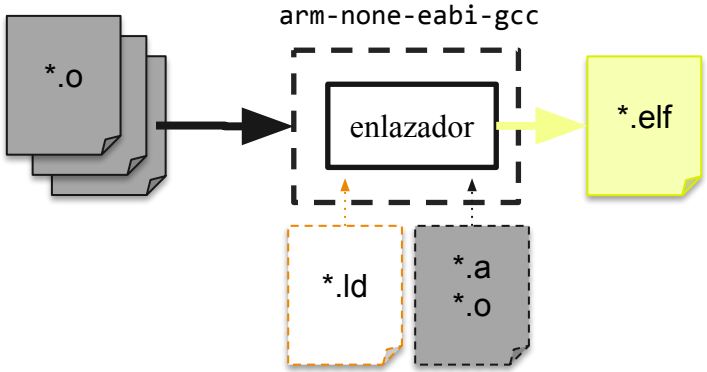


Ej:

```
arm-none-eabi-gcc -c fuente1.c -o fuente1.o
arm-none-eabi-gcc -c fuente2.cpp -o fuente2.o
arm-none-eabi-gcc -c fuente3.s -o fuente3.o
```

Uso típico (compilación y **enlazado** separados)

2. Enlazado: a partir de los código objeto generados (*.o), se conforma un único binario (*.elf), agregando si es necesario código de la biblioteca estándar provisto con el toolchain y bibliotecas de terceros (*.o/*.a). Las reglas para determinar en qué zona de memoria se aloja cada objeto de código (funciones, variables, constantes, etc…) se determinan en el archivo linker script (*.ld)



Ej:

```
arm-none-eabi-gcc fuente1.o fuente2.o fuente3.o mod_precomp.o -llibreria -o ejecutable.elf  -T script.ld
```

Herramientas del ARM GNU Toolchain

El GNU Toolchain posee un conjunto de herramientas (binutils), varias de las cuales permiten realizar distinto tipo de análisis sobre los binarios producidos (.o / .elf):

- **arm-none-eabi-size** permite ver tamaño en bytes y ubicación en memoria de cada *sección* de uno o varios binarios
- **arm-none-eabi-strings** muestra las cadenas de caracteres incrustadas en uno o más binarios
- **arm-none-eabi-nm** lista los símbolos (nombres de funciones, variables, cttes., etc.) de uno o más binarios
- **arm-none-eabi-objdump** permite desensamblar funciones, visualizar la tabla de símbolos, tamaño y ubicación de las *secciones* de uno o más binarios
- **arm-none-eabi-readelf** similar a **objdump**, además permite visualizar la información binaria por *secciones*.
- **arm-none-eabi-objcopy** convierte entre diferentes formatos de archivos binarios (.elf, .bin, .hex)

Herramientas del ARM GNU Toolchain

El GNU Toolchain posee un conjunto de herramientas (binutils), varias de las cuales permiten realizar distinto tipo de análisis sobre los binarios producidos (.o / .elf):

- **arm-none-eabi-size** permite ver tamaño en bytes y ubicación en memoria de cada *sección* de uno o varios binarios
- **arm-none-eabi-strings** muestra las cadenas de caracteres incrustadas en uno o más binarios
- **arm-none-eabi-nm** lista los símbolos (nombres de funciones, variables, cttes., etc.) de uno o más binarios
- **arm-none-eabi-objdump** permite desensamblar funciones, visualizar la tabla de símbolos, tamaño y ubicación de las *secciones* de uno o más binarios
- **arm-none-eabi-readelf** similar a **objdump**, además permite visualizar la información binaria por *secciones*.
- **arm-none-eabi-objcopy** convierte entre diferentes formatos de archivos binarios (.elf, .bin, .hex)

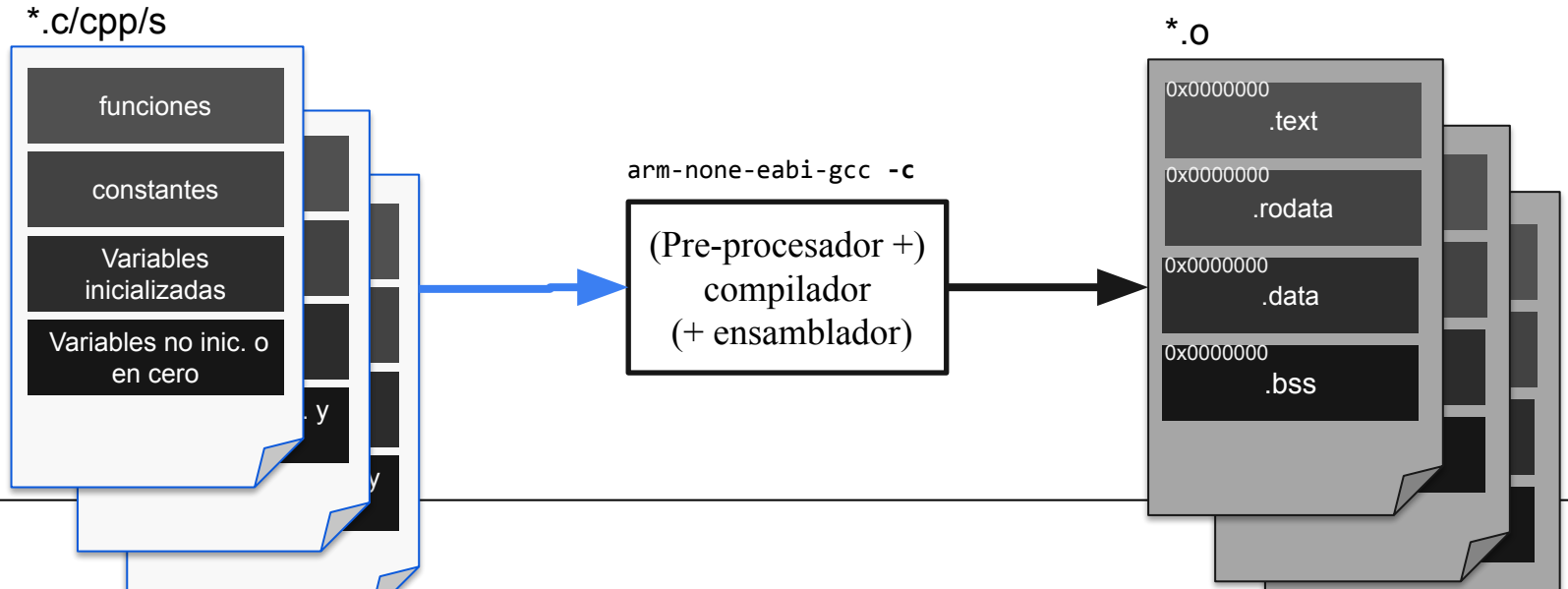
Secciones de memoria y manipulación mediante linker script

Secciones

- Los archivos objeto (.o/.elf) están organizados por secciones
- Cada sección almacena un **tipo de información** binaria diferente
- Por defecto existen cuatro secciones:
  - **.text**: instrucciones ejecutables (funciones compiladas)
  - .**rodata**: constantes globales y locales static
  - **.data**: variables globales y locales static inicializadas (≠ 0)
  - **.bss**: variables globales y locales static no inicializadas (== 0)
- Dependiendo de las necesidades de la aplicación se usan más o menos secciones
- Puede haber subsecciones. Por ejemplo, es común compilar para que en los archivos .o cada función tenga su propia sección: por ejemplo *.text.main*

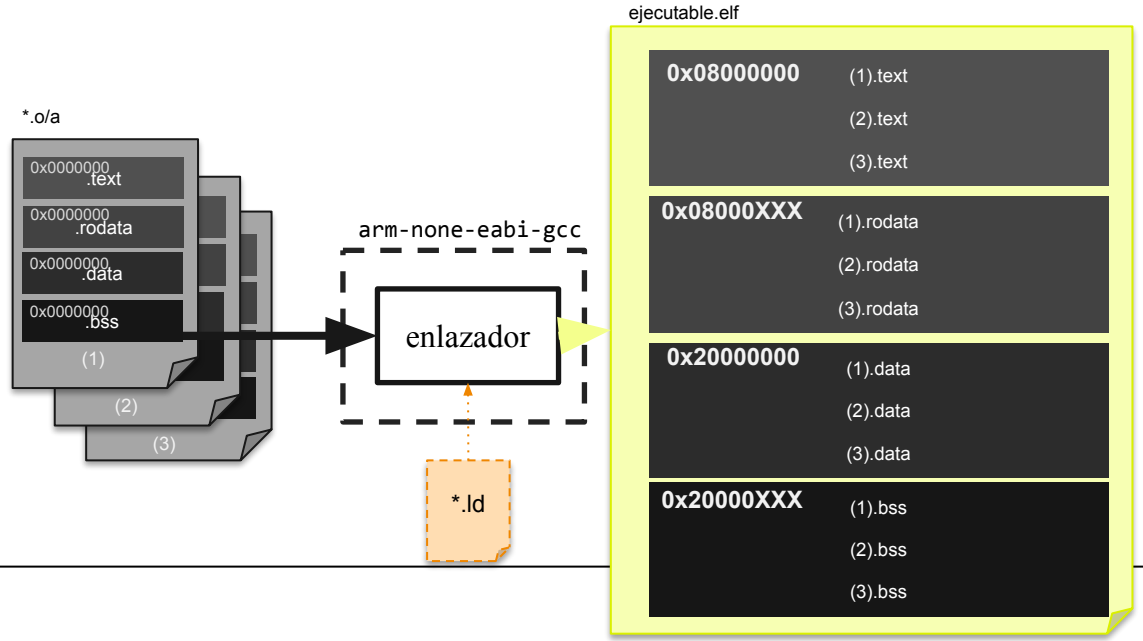
Secciones

- Luego de la **compilación** cada archivo .o tiene su conjunto de secciones.
- Cada una de las cuales comienza en la dirección de memoria cero (`0x00000000`)
- En el caso de que el código haga llamadas a funciones o utilice datos definidos en otros módulos (definidos como *extern*) las direcciones no se resuelven



Secciones

- Luego de la fase **enlazado** se unen las secciones de entrada (de los distintos archivos objeto) para formar las secciones de salida (del archivo .elf)
- A cada sección de salida se le asigna una dirección de memoria física
- Se resuelven las direcciones en las llamadas a función y variables externas



Ejemplo básico: volvemos al ejemplo “Blink” original

```c
#include <stdint.h>
#define ITER_ESPERA 400000

void configura_led();
void prende_led();
void apaga_led();
void espera(uint32_t);

void main()
{
    configura_led();

    while(1){
        prende_led();
        espera(ITER_ESPERA);
        apaga_led();
        espera(ITER_ESPERA);
    }
}
```

```c
void espera(uint32_t iter){
    unsigned b = 0;
    while (b++ < iter)
        ;
}
```

fuente.c

```c
#define RCC_BASE    0x40021000                                  /** p. 50  **/
#define RCC_APB2ENR *((volatile uint32_t *)(RCC_BASE + 0x18))   /** p. 112 **/
#define RCC_APB2ENR_IOPCEN (1<<4)                               /** p. 114 **/

#define GPIOC_BASE  0x40011000                                  /** p 51   **/
#define GPIOC_CRH *((volatile uint32_t *)(GPIOC_BASE + 0x04))   /** p. 172 **/
#define GPIOC_CRH_MODE13_0  (1<<20)
#define GPIOC_CRH_MODE13_1  (1<<21)
#define GPIOC_CRH_CNF13_0  (1<<22)
#define GPIOC_CRH_CNF13_1  (1<<23)
void configura_led(){
    // habilita el clock del GPIOC
    RCC_APB2ENR |= RCC_APB2ENR_IOPCEN;
    // establece PC13 como salida push-pull / 2MHz max
              //pongo los 4 bits en cero
    GPIOC_CRH &=  ~(GPIOC_CRH_MODE13_0|GPIOC_CRH_MODE13_1|
                    GPIOC_CRH_CNF13_0|GPIOC_CRH_CNF13_1);
    GPIOC_CRH |=  GPIOC_CRH_MODE13_1;
}
```

```c
#define GPIOC_ODR *((volatile uint32_t *)(GPIOC_BASE + 0x0C))   /** p. 173 **/
void prende_led(){ GPIOC_ODR &= ~(1 << 13);}
void apaga_led(){ GPIOC_ODR |= (1 << 13);}
```

- Compilamos con el comando: `arm-none-eabi-gcc -c fuente.c -o fuente.o`
- Analizamos archivo objeto:

```
arm-none-eabi-objdump  -d fuente.o
```

```
fuente.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   ebffffe         bl      2c <configura_led>
   c:   ebffffe         bl      88 <prende_led>
  10:   e59f0010        ldr     r0, [pc, #16]   ; 28 <main+0x28>
  14:   ebffffe         bl      e8 <espera>
  18:   ebffffe         bl      b8 <apaga_led>
  1c:   e59f0004        ldr     r0, [pc, #4]    ; 28 <main+0x28>
  20:   ebffffe         bl      e8 <espera>
  24:   eafffff8        b       c <main+0xc>
  28:   00061a80        .word   0x00061a80
```

```
0000002c <configura_led>:
  2c:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
  30:   e28db000        add     fp, sp, #0
  34:   e59f3044        ldr     r3, [pc, #68]   ; 80 <configura_led+0x54>
  38:   e5933000        ldr     r3, [r3]
  3c:   e59f203c        ldr     r2, [pc, #60]   ; 80 <configura_led+0x54>
  40:   e3833010        orr     r3, r3, #16
  44:   e5823000        str     r3, [r2]
  48:   e59f3034        ldr     r3, [pc, #52]   ; 84 <configura_led+0x58>
  4c:   e5933000        ldr     r3, [r3]
  50:   e59f202c        ldr     r2, [pc, #44]   ; 84 <configura_led+0x58>
  54:   e3c3360f        bic     r3, r3, #15728640       ; 0xf00000
  58:   e5823000        str     r3, [r2]
  5c:   e59f3020        ldr     r3, [pc, #32]   ; 84 <configura_led+0x58>
  60:   e5933000        ldr     r3, [r3]
  64:   e59f2018        ldr     r2, [pc, #24]   ; 84 <configura_led+0x58>
  68:   e3833602        orr     r3, r3, #2097152        ; 0x200000
  6c:   e5823000        str     r3, [r2]
  70:   e1a00000        nop                     ; (mov r0, r0)
  74:   e28bd000        add     sp, fp, #0
  78:   e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
  7c:   e12fff1e        bx      lr
  80:   40021018        .word   0x40021018
  84:   40011004        .word   0x40011004
```

- Compilamos con el comando: `arm-none-eabi-gcc -c fuente.c -o fuente.o`
- Analizamos archivo objeto:

```
arm-none-eabi-objdump  -d fuente.o
```

```
fuente.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   ebfffffe        bl      2c <configura_led>
   c:   ebfffffe        bl      88 <prende_led>
  10:   e59f0010        ldr     r0, [pc, #16]   ; 28 <main+0x28>
  14:   ebfffffe        bl      e8 <espera>
  18:   ebfffffe        bl      b8 <apaga_led>
  1c:   e59f0004        ldr     r0, [pc, #4]    ; 28 <main+0x28>
  20:   ebfffffe        bl      e8 <espera>
  24:   eafffff8        b       c <main+0xc>
  28:   00061a80        .word   0x00061a80
```

Dirección dentro de .text dónde comienza main

desensamblado

Código objeto (binario ejecutable)

Offset de cada instrucción

```
0000002c <configura_led>:
  2c:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
  30:   e28db000        add     fp, sp, #0
  34:   e59f3044        ldr     r3, [pc, #68]   ; 80 <configura_led+0x54>
  38:   e5933000        ldr     r3, [r3]
  3c:   e59f203c        ldr     r2, [pc, #60]   ; 80 <configura_led+0x54>
  40:   e3833010        orr     r3, r3, #16
  44:   e5823000        str     r3, [r2]
  48:   e59f3034        ldr     r3, [pc, #52]   ; 84 <configura_led+0x58>
  4c:   e5933000        ldr     r3, [r3]
  50:   e59f202c        ldr     r2, [pc, #44]   ; 84 <configura_led+0x58>
  54:   e3c3360f        bic     r3, r3, #15728640       ; 0xf00000
  58:   e5823000        str     r3, [r2]
  5c:   e59f3020        ldr     r3, [pc, #32]   ; 84 <configura_led+0x58>
  60:   e5933000        ldr     r3, [r3]
  64:   e59f2018        ldr     r2, [pc, #24]   ; 84 <configura_led+0x58>
  68:   e3833602        orr     r3, r3, #2097152        ; 0x200000
  6c:   e5823000        str     r3, [r2]
  70:   e1a00000        nop                     ; (mov r0, r0)
  74:   e28bd000        add     sp, fp, #0
  78:   e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
  7c:   e12fff1e        bx      lr
  80:   40021018        .word   0x40021018
  84:   40011004        .word   0x40011004
```

- Compilamos con el comando: `arm-none-eabi-gcc -c fuente.c -o fuente.o`
- Analizamos archivo objeto:

```
arm-none-eabi-objdump  -d fuente.o

fuente.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <main>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp,
   8:   ebffffe         bl      2c <cont
   c:   ebffffe         bl      88 <pren
  10:   e59f0010        ldr     r0, [pc,
  14:   ebffffe         bl      e8 <espe
  18:   ebffffe         bl      b8 <apag
  1c:   e59f0004        ldr     r0, [pc,
  20:   ebffffe         bl      e8 <espe
  24:   eafffff8        b       c <main+
  28:   00061a80        .word   0x00061a

0000002c <configura_led>:
  2c:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
  30:   e28db000        add     fp, sp, #0
                                                ; 80 <configura_led+0x54>
                                                ; 80 <configura_led+0x54>
                                                ; 84 <configura_led+0x58>
                                                ; 84 <configura_led+0x58>
640         ; 0xf00000
                                                ; 84 <configura_led+0x58>
                                                ; 84 <configura_led+0x58>
52          ; 0x200000
                                                ; (mov r0, r0)
                                                ; (ldr fp, [sp], #4)
  7c:   e12fff1e        bx      lr
  80:   40021018        .word   0x40021018
  84:   40011004        .word   0x40011004
```

Dirección dentro de .text dónde comienza main

Código objeto (binario ejecuta

Offset de cada instrucción

Cada instrucción ocupa 32 bits, lo cual es el set de instrucciones ARM (no Thumb2)

- No se compiló para Cortex-M3!!!!

- Compilamos teniendo en cuenta la arquitectura destino:

```
arm-none-eabi-gcc -c fuente.c -o fuente.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
```

```
arm-none-eabi-objdump  -d fuente.o
```

```
fuente.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <main>:
   0:   b580            push    {r7, lr}
   2:   af00            add     r7, sp, #0
   4:   f7ff fffe       bl      24 <configura_led>
   8:   f7ff fffe       bl      5c <prende_led>
   c:   4804            ldr     r0, [pc, #16]   ; (20 <main+0x20>)
   e:   f7ff fffe       bl      94 <espera>
  12:   f7ff fffe       bl      78 <apaga_led>
  16:   4802            ldr     r0, [pc, #8]    ; (20 <main+0x20>)
  18:   f7ff fffe       bl      94 <espera>
  1c:   e7f4            b.n     8 <main+0x8>
  1e:   bf00            nop
  20:   00061a80        .word   0x00061a80
```

```
00000024 <configura_led>:
  24:   b480            push    {r7}
  26:   af00            add     r7, sp, #0
  28:   4b0a            ldr     r3, [pc, #40]   ; (54 <configura_led+0x30>)
  2a:   681b            ldr     r3, [r3, #0]
  2c:   4a09            ldr     r2, [pc, #36]   ; (54 <configura_led+0x30>)
  2e:   f043 0310       orr.w   r3, r3, #16
  32:   6013            str     r3, [r2, #0]
  34:   4b08            ldr     r3, [pc, #32]   ; (58 <configura_led+0x34>)
  36:   681b            ldr     r3, [r3, #0]
  38:   4a07            ldr     r2, [pc, #28]   ; (58 <configura_led+0x34>)
  3a:   f423 0370       bic.w   r3, r3, #15728640       ; 0xf00000
  3e:   6013            str     r3, [r2, #0]
  40:   4b05            ldr     r3, [pc, #20]   ; (58 <configura_led+0x34>)
  42:   681b            ldr     r3, [r3, #0]
  44:   4a04            ldr     r2, [pc, #16]   ; (58 <configura_led+0x34>)
  46:   f443 1300       orr.w   r3, r3, #2097152        ; 0x200000
  4a:   6013            str     r3, [r2, #0]
  4c:   bf00            nop
  4e:   46bd            mov     sp, r7
  50:   bc80            pop     {r7}
  52:   4770            bx      lr
  54:   40021018        .word   0x40021018
  58:   40011004        .word   0x40011004
```

- Compilamos teniendo en cuenta la arquitectura destino:

```
arm-none-eabi-gcc -c fuente.c -o fuente.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
```

```
arm-none-eabi-objdump  -t fuente.o

fuente.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 fuente.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     F .text  00000024 main
00000024 g     F .text  00000038 configura_led
0000005c g     F .text  0000001c prende_led
00000094 g     F .text  00000026 espera
00000078 g     F .text  0000001c apaga_led
```

- Ya tenemos las funciones compiladas para ser ejecutadas por el STM32F103…
- Ahora es trabajo del enlazador posicionar las instrucciones en la memoria flash, que comienza en 0x08000000

Table 5. Flash module organization (medium-density devices)

Block	Name	Base addresses	Size (bytes)
Main memory	Page 0	0x0800 0000 - 0x0800 03FF	1 K
	Page 1	0x0800 0400 - 0x0800 07FF	1 K
	Page 2	0x0800 0800 - 0x0800 0BFF	1 K
	Page 3	0x0800 0C00 - 0x0800 0FFF	1 K
	Page 4	0x0800 1000 - 0x0800 13FF	1 K
	. . .	. . .	. . .
	Page 127	0x0801 FC00 - 0x0801 FFFF	1 K
Information block	System memory	0x1FFF F000 - 0x1FFF F7FF	2 K
	Option bytes	0x1FFF F800 - 0x1FFF F80F	16
Flash memory interface registers	FLASH_ACR	0x4002 2000 - 0x4002 2003	4
	FLASH_KEYR	0x4002 2004 - 0x4002 2007	4
	FLASH_OPTKEYR	0x4002 2008 - 0x4002 200B	4
	FLASH_SR	0x4002 200C - 0x4002 200F	4
	FLASH_CR	0x4002 2010 - 0x4002 2013	4
	FLASH_AR	0x4002 2014 - 0x4002 2017	4
	Reserved	0x4002 2018 - 0x4002 201B	4
	FLASH_OBR	0x4002 201C - 0x4002 201F	4
	FLASH_WRPR	0x4002 2020 - 0x4002 2023	4

Aquí deben colocarse:

- la tabla de vectores de excepciones
- las funciones compiladas (sección .text)
- Constantes globales/locales static (sección .rodata)

Reference Manual RM0008

Ubicación en memoria de la tabla de vectores de excepciones

Vectores de excepciones

- Recordar que para el proceso de arranque (o luego de un reset) son necesarios al menos los dos primeros vectores:
  1. Se carga el puntero de pila (MSP) con el contenido de 0x00000000*
  2. Se carga el contador de programa (PC) con el contenido de 0x00000004*

* Estas direcciones de memoria no existen en el STM32F103 (son alias)… En el modo de booteo “normal” desde memoria flash esas direcciones están “mapeadas” a partir de 0x08000000

- La definición de la tabla de vectores en lenguaje C podría hacerse de la siguiente manera (además de las dos primeras entradas obligatorias se agregan como ejemplo las excepciones que pueden producirse en caso de falla):

```
const uint32_t vectores[] ={
          0x20005000,               /*final de la ram*/
          (uint32_t)main,           /*Reset*/
          (uint32_t)null_handler,   /*NMI*/
          (uint32_t)null_handler,   /*Hard Fault*/
          (uint32_t)null_handler,   /*Memory Management Fault*/
          (uint32_t)null_handler,   /*Bus Fault*/
          (uint32_t)null_handler,   /*Usage Fault*/

            //acá habría que incluir los otros
            //vectores de excepción que se usen
      };
```

```
void null_handler(){
  while(1);
}
```

Vectores de excepciones

```
const uint32_t vectores[] ={
        0x20005000,              /*final de la ram*/
        (uint32_t) main,         /*Reset*/
        (uint32_t) null_handler, /*NMI*/
        (uint32_t)null_handler,  /*Hard Fault*/
        (uint32_t)null_handler,  /*Memory Management Fault*/
        (uint32_t)null_handler,  /*Bus Fault*/
        (uint32_t)null_handler,  /*Usage Fault*/

         //acá habría que incluir los otros
         //vectores de excepción que se usen
    };
```

```
void main(){
 ...
}
```

```
void null_handler(){
  while(1);
}
```

Recordemos: En lenguaje C, se usa el nombre de una función …

- seguido con un par de paréntesis llamada
- Solo (sin los paréntesis) puntero a función –dirección en memoria de las instrucciones de la función–

Hay que instruir al enlazador para que:

- Posicione la tabla de vectores a partir de la posición 0x08000000
- Posicione el código objeto de las funciones a continuación de la tabla.

Esto se hace parte en el **código fuente** y parte en el **linker script**.

En el **código fuente** se puede asignar un *atributo* al objeto que nos interese (función, variable, constante…) para obligar al compilador a colocarlo en una sección arbitraria

```
__attribute__((section(".isr_vectors")))
const uint32_t vectores[] ={
    0x20005000,                 /*final de la ram*/
    (uint32_t)main,             /*Reset*/
      /*...*/
};
```

En el **linker script** se indica en qué tipo de memoria y dirección se ubica cada sección

```
MEMORY
{
    flash(rx) : ORIGIN = 0x08000000, LENGTH = 64K
    sram(wrx)   : ORIGIN = 0x20000000, LENGTH = 20K
}

SECTIONS
{
    .isr_vectors :{KEEP(*(.isr_vectors))} > flash
    .text :{*(.text)}       > flash
    .rodata :{*(.rodata)}   > flash
    .data : { *(.data) }    > sram AT > flash
    .bss : { *(.bss) }      > sram
}
```

- **Compilamos** nuevamente:

```
arm-none-eabi-gcc -c fuente.c -o fuente.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
```

`arm-none-eabi-objdump -t fuente.o`

```
fuente.o:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 fuente.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
00000000 l    d  .isr_vectors   00000000 .isr_vectors
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 g     F .text  00000024 main
00000024 g     F .text  00000038 configura_led
0000005c g     F .text  0000001c prende_led
00000094 g     F .text  00000026 espera
00000078 g     F .text  0000001c apaga_led
00000000 g     O .isr_vectors   00000008 vectores
```

`arm-none-eabi-objdump -h fuente.o`

```
fuente.o:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000bc  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  000000f0  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  000000f0  2**0
                  ALLOC
  3 .isr_vectors  00000008  00000000  00000000  000000f0  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  4 .comment      0000004a  00000000  00000000  000000f8  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002d  00000000  00000000  00000142  2**0
                  CONTENTS, READONLY
```

- Ahora **enlazamos**, indicándole al linker dónde ubicar cada sección a través del script:

```
arm-none-eabi-gcc fuente.o -o ejecutable.elf  -T script.ld -nostartfiles
```

Esto evita que se enlace el archivo objeto de “inicialización genérica” crt0.o provisto por la biblioteca estándar, que incluye (entre otros elementos) la función _start que realiza una inicialización más compleja que la necesaria en sistemas *bare-metal*

```
arm-none-eabi-gcc fuente.o -o ejecutable.elf  -T script.ld -nostartfiles
```

```
arm-none-eabi-objdump  -t ejecutable.elf

ejecutable.elf:     file format elf32-littlearm

SYMBOL TABLE:
08000000 l    d  .isr_vectors   00000000 .isr_vectors
08000008 l    d  .text  00000000 .text
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 l    df *ABS*  00000000 fuente.c
08000064 g     F .text  0000001c prende_led
08000000 g     O .isr_vectors   00000008 vectores
0800002c g     F .text  00000038 configura_led
08000008 g     F .text  00000024 main
08000080 g     F .text  0000001c apaga_led
0800009c g     F .text  00000026 espera
```

```
arm-none-eabi-objdump  -h ejecutable.elf

ejecutable.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .isr_vectors  00000008  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         000000bc  08000008  08000008  00010008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .comment      00000049  00000000  00000000  000100c4  2**0
                  CONTENTS, READONLY
  3 .ARM.attributes 0000002d  00000000  00000000  0001010d  2**0
                  CONTENTS, READONLY
```

```
arm-none-eabi-gcc fuente.o -o ejecutable.elf  -T script.ld -nostartfiles
```

```
arm-none-eabi-readelf ejecutable.elf -x .isr_vectors -x .text

Hex dump of section '.isr_vectors':
  0x08000000 00500020 09000008                    .P. ....

Hex dump of section '.text':
  0x08000008 80b500af 00f00cf8 00f028f8 044800f0 ..........(..H..
  0x08000018 41f800f0 31f80248 00f03cf8 f4e700bf A...1..H..<.....
  0x08000028 801a0600 80b400af 0a4b1b68 094a43f0 .........K.h.JC.
  0x08000038 10031360 084b1b68 074a23f4 70031360 ...`.K.h.J#.p..`
  0x08000048 054b1b68 044a43f4 00131360 00bfbd46 .K.h.JC....`...F
  0x08000058 80bc7047 18100240 04100140 80b400af ..pG...@...@....
  0x08000068 044b1b68 034a23f4 00531360 00bfbd46 .K.h.J#..S.`...F
  0x08000078 80bc7047 0c100140 80b400af 044b1b68 ..pG...@.....K.h
  0x08000088 034a43f4 00531360 00bfbd46 80bc7047 .JC..S.`...F..pG
  0x08000098 0c100140 80b485b0 00af7860 0023fb60 ...@......x`.#.`
  0x080000a8 00bffb68 5a1cfa60 7a689a42 f9d800bf ...hZ..`zh.B....
  0x080000b8 00bf1437 bd4680bc 704700bf          ...7.F..pG..
```

0x20005000

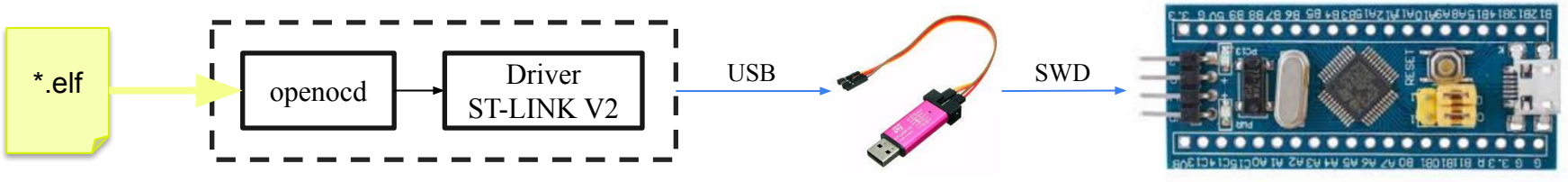
0x08000009 = 0x08000008 + 1

Modo Thumb2

Descarga del ejecutable al dispositivo

- Mediante el adaptador ST-LINK V2 se puede descargar el binario a la flash del dispositivo utilizando el software:
  - STM32CubeProgrammer (versión de línea de comandos o GUI)
  - openocd (Open On-Chip Debugger)

```
openocd
-f interface/stlink.cfg
-c "transport select hla_swd"
-f target/stm32f1x.cfg
-c "program ejecutable.elf verify reset exit"
```



Iniciación de variables globales

Ejemplo 2: agregando variables globales

Al ejemplo blink previo le modificamos que

- ITER_ESPERA se reemplaza por una variable global
- Se agrega otra variable global para usarse como contador de “blinks”

fuente.c

```
#include <stdint.h>

void configura_led();
void prende_led();
void apaga_led();
void espera(uint32_t);

uint32_t iter_espera = 50000;
uint32_t contador=0;

void main()
{
    configura_led();

    while(1){
        prende_led();
        espera(iter_espera);
        apaga_led();
        espera(iter_espera);
        contador++;
    }
}
...
```

```
arm-none-eabi-gcc -c fuente.c -o fuente.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
arm-none-eabi-gcc fuente.o -o ejecutable.elf  -T script.ld -nostartfiles
```

VMA: Virtual Memory Address

LMA: Load Memory Address

```
arm-none-eabi-objdump  -t ejecutable.elf

ejecutable.elf:     file format elf32-littlearm

SYMBOL TABLE:
08000000 l    d  .isr_vectors   00000000 .isr_vectors
08000008 l    d  .text  00000000 .text
20000000 l    d  .data  00000000 .data
20000004 l    d  .bss   00000000 .bss
00000000 l    d  .comment       00000000 .comment
00000000 l    d  .ARM.attributes        00000000 .ARM.attributes
00000000 l    df *ABS*  00000000 fuente.c
08000078 g     F .text  0000001c prende_led
08000000 g     O .isr_vectors   00000008 vectores
08000040 g     F .text  00000038 configura_led
20000000 g     O .data  00000004 iter_espera
20000004 g     O .bss   00000004 contador
08000008 g     F .text  00000038 main
08000094 g     F .text  0000001c apaga_led
080000b0 g     F .text  00000026 espera
```

```
arm-none-eabi-objdump  -h ejecutable.elf

ejecutable.elf:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .isr_vectors  00000008  08000000  08000000  00010000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         000000d0  08000008  08000008  00010008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data         00000004  20000000  080000d8  00020000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000004  20000004  080000dc  00020004  2**2
                  ALLOC
  4 .comment      00000049  00000000  00000000  00020004  2**0
                  CONTENTS, READONLY
  5 .ARM.attributes 0000002d  00000000  00000000  0002004d  2**0
                  CONTENTS, READONLY
```

Iniciación de variables globales / locales static

```
MEMORY
{
    flash(rx) : ORIGIN = 0x08000000, LENGTH = 64K
    sram(wrx)   : ORIGIN = 0x20000000, LENGTH = 20K
}

SECTIONS
{
    .isr_vectors :{KEEP(*(.isr_vectors))} > flash
    .text :{*(.text)}       > flash
    .rodata :{*(.rodata)}   > flash
    .data : { *(.data) }    > sram AT > flash
    .bss : { *(.bss) }      > sram
}
```

Carmine Noviello – Mastering STM32

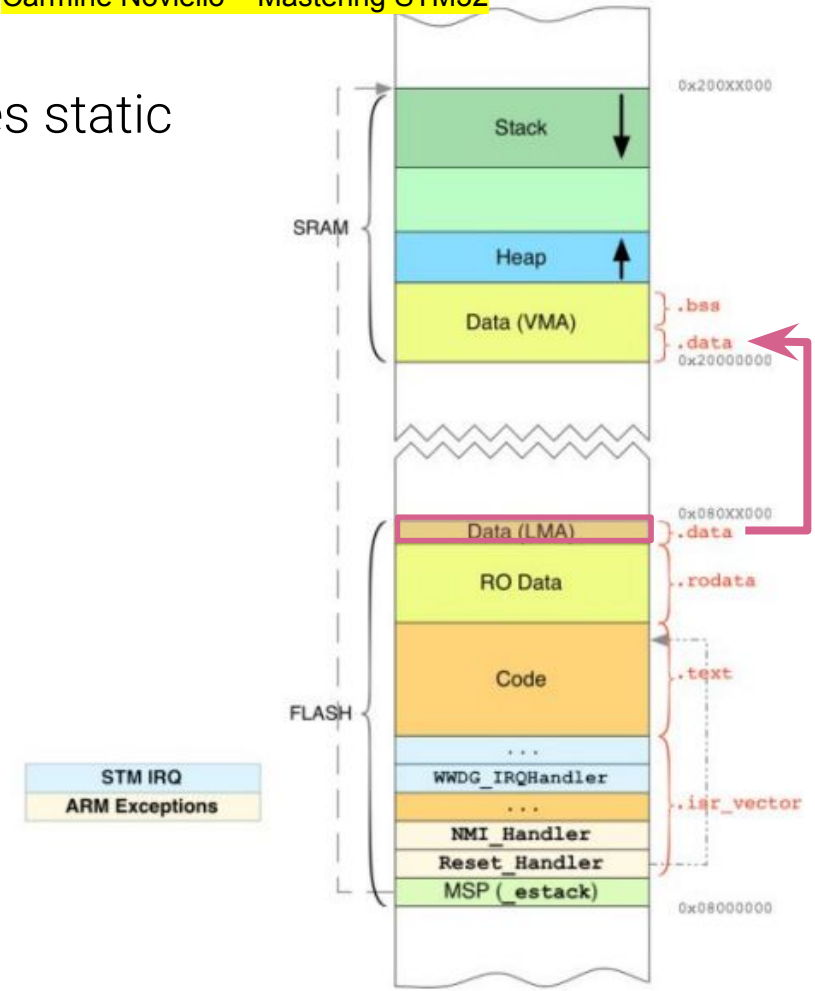


Figure 1: The typical layout of flash and SRAM memories

Iniciación de variables globales / locales static

- El programa va a fallar dado que no hay ningún código que inicialice las variables globales.
- A pesar de que están inicializadas en la declaración global **NO HAY CÓDIGO** que copie esos valores a RAM

fuente.c

```
#include <stdint.h>

void configura_led();
void prende_led();
void apaga_led();
void espera(uint32_t);

uint32_t iter_espera = 50000;
uint32_t contador=0;

void main()
{
    configura_led();

    while(1){
        prende_led();
        espera(iter_espera);
        apaga_led();
        espera(iter_espera);
        contador++;
    }
}
...
```

Iniciación de variables globales / locales static

- Si bien se podrían agregar las líneas de código al principio del programa principal que inicializan las variables globales, esta **no es una buena práctica**
- Es imposible de hacer con bibliotecas de terceros (no podemos conocer todas las variables que usan, ni sus valores de inicialización)

fuente.c

```
#include <stdint.h>

void configura_led();
void prende_led();
void apaga_led();
void espera(uint32_t);

uint32_t iter_espera = 50000;
uint32_t contador=0;

void main()
{
    iter_espera = 50000;
    contador=0;
    configura_led();

    while(1){
        prende_led();
        espera(iter_espera);
        apaga_led();
        espera(iter_espera);
        contador++;
    }
}
...
```

Iniciación de variables globales

SE IMPLEMENTA UNA SOLUCIÓN QUE SIRVE PARA LA TOTALIDAD DE LOS CASOS

- Para **TODAS** las variables globales/estáticas **inicializadas/no-nulas** hay que incluir el código que copie el bloque almacenado en la **LMA** de la sección .data a la **VMA**
- Para **TODAS** las variables globales/estáticas **no-inicializadas/nulas** hay que incluir el código que asigne 0x00 a todos los bytes de la sección .bss
- Desde el **código fuente**, puede accederse a las direcciones de inicio y final de cada sección través de símbolos globales definidos en el **linker script**

fuente.c (al comienzo de main())

```
// sección .bss (inicializada en cero)
extern uint8_t _sbss, _ebss;
uint8_t *dst;
for (dst = &_sbss; dst < &_ebss;)
  *dst++ = 0;

// sección .data (se copian valores de la FLASH a la RAM)
uint8_t *src;
extern uint8_t _sidata, _sdata, _edata;
for (dst = &_sdata, src = &_sidata; dst < &_edata;)
  *dst++ = *src++;
```

script.ld

```
MEMORY
{
    flash(rx) : ORIGIN = 0x08000000, LENGTH = 64K
    sram(wrx)   : ORIGIN = 0x20000000, LENGTH = 20K
}

SECTIONS
{
    .isr_vectors :{KEEP(*(.isr_vectors))} > flash
    .text :{*(.text)}       > flash
    .rodata :{*(.rodata)}   > flash
    _sidata = LOADADDR(.data);
    .data : { _sdata = .;
        *(.data)
        _edata = .; } > sram AT > flash
    .bss : { _sbss = .;
        *(.bss)
        _ebss = .;} > sram
}
```

Finalizamos el ejemplo modularizando la aplicación construida en distintos fuentes y compilando cada módulo por separado.

- main.c : implementa la función principal y la de espera
- led.c : implementa las funciones referidas al LED on-board
- startup.c: se define la tabla de vectores y una nueva función _start (será el nuevo Reset_Handler) desde la cual se invoca a main.

main.c

```c
#include <stdint.h>

void configura_led();
void prende_led();
void apaga_led();
void espera(uint32_t);

uint32_t iter_espera = 200000;
uint32_t contador=0;

void main()
{

    configura_led();

    while(1){
        prende_led();
        espera(iter_espera);
        apaga_led();
        espera(iter_espera);
        contador++;
    }
}


void espera(uint32_t iter){
    volatile unsigned b = 0;
    while (b++ < iter)
        ;
}
```

led.c

```c
#include <stdint.h>

void configura_led();
void prende_led();
void apaga_led();

#define RCC_BASE    0x40021000                          /** Reference manual - p.
50 **/
#define RCC_APB2ENR *((volatile uint32_t *)(RCC_BASE + 0x18))   /** p. 112 **/
#define RCC_APB2ENR_IOPCEN (1<<4)                       /** p. 114 **/

#define GPIOC_BASE  0x40011000                          /** Reference manual - p.
51     **/
#define GPIOC_CRH *((volatile uint32_t *)(GPIOC_BASE + 0x04))   /** p. 172 **/
#define GPIOC_CRH_MODE13_0  (1<<20)
#define GPIOC_CRH_MODE13_1  (1<<21)
#define GPIOC_CRH_CNF13_0  (1<<22)
#define GPIOC_CRH_CNF13_1  (1<<23)



void configura_led(){
    // habilita el clock del GPIOC
    RCC_APB2ENR |= RCC_APB2ENR_IOPCEN;
    // establece PC13 como salida push-pull / 2MHz max
    GPIOC_CRH &=  ~(GPIOC_CRH_MODE13_0|GPIOC_CRH_MODE13_1|
                    GPIOC_CRH_CNF13_0|GPIOC_CRH_CNF13_1); //pongo los 4 bits en cero
    GPIOC_CRH |=  GPIOC_CRH_MODE13_1;
}

#define GPIOC_ODR *((volatile uint32_t *)(GPIOC_BASE + 0x0C))   /** p. 173 **/
void prende_led(){
    // prende el LED - poniendo a GND PC13
    GPIOC_ODR &= ~(1 << 13);
}

void apaga_led(){
    // apaga el LED - poniendo a VDD PC13
    GPIOC_ODR |= (1 << 13);
}
```

Finaliz
fuente
- ma
- led
- sta

á el
nue

startup.c

```c
#include <stdint.h>

void _start(void);
int main();
void null_handler();

__attribute__((section(".isr_vectors")))
const uint32_t vectores[] ={
    0x20005000, //final de la ram para dispositivo de 20kB
    (uint32_t)_start,
    (uint32_t)null_handler,   /*NMI*/
    (uint32_t)null_handler,   /*Hard Fault*/
    (uint32_t)null_handler,   /*Memory Management Fault*/
    (uint32_t)null_handler,   /*Bus Fault*/
    (uint32_t)null_handler,   /*Usage Fault*/
 };


void _start(void)
{
  // sección .bss (inicializada en cero)
  extern uint8_t _sbss, _ebss;
  uint8_t *dst;
  for (dst = &_sbss; dst < &_ebss;)
    *dst++ = 0;

  // sección .data (se copian valores de la FLASH a la RAM)
  uint8_t *src;
  extern uint8_t _sidata, _sdata, _edata;
  for (dst = &_sdata, src = &_sidata; dst < &_edata;)
    *dst++ = *src++;

  main();

  while (1);
}

void null_handler(){
  while(1);
}
```

Finalizamos el ejemplo modularizando la aplicación construida en distintos fuentes y compilando cada módulo por separado.

- main.c : implementa la función principal y la de espera
- led.c : implementa las funciones referidas al LED on-board
- startup.c: se define la tabla de vectores y una nueva función _start (será el nuevo Reset_Handler) desde la cual se invoca a main.

```
arm-none-eabi-gcc -c main.c -o main.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
arm-none-eabi-gcc -c led.c -o led.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
arm-none-eabi-gcc -c startup.c -o startup.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft

arm-none-eabi-gcc *.o -o ejecutable.elf  -T script.ld -nostartfiles
```

Enlazado de Bibliotecas Estándar de C

Biblioteca Estándar de C

Una de las ventajas de programar en C es el soporte de las funciones de su biblioteca estándar.
El toolchain GNU ARM viene con dos versiones de la biblioteca estándar:

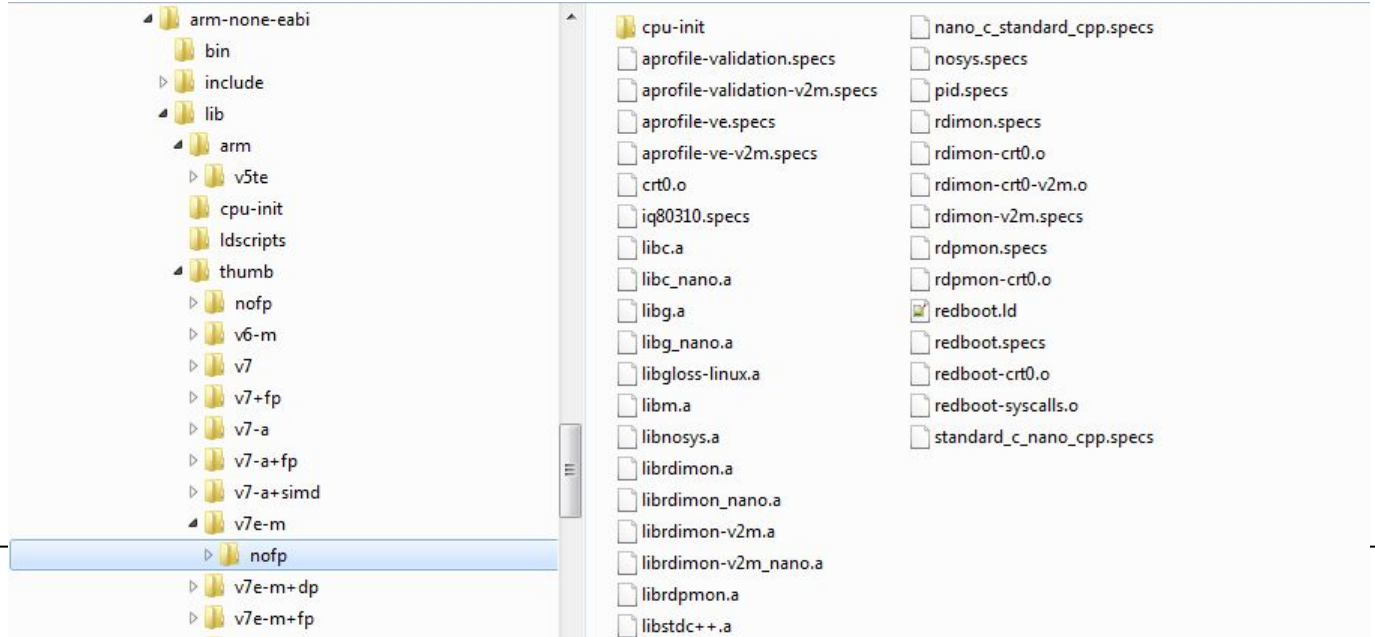
- **newlib** : una versión open source (y con algunas limitaciones) de la biblioteca estándar de C, desarrollada con la intención de su utilización en embebidos (tanto hosteados como bare-metal).
- **newlib-nano** : basada en la anterior pero optimizada para ocupar menor espacio en flash.

Estas vienen compiladas y empaquetadas en archivos de biblioteca estática **libc.a** y **libc_nano.a**. Estos archivos no son más que un empaquetamiento de archivos objeto .o con las distintas funciones de la librería que se incorporarán en la etapa de enlazado.
Por fuera de estos archivos quedan:

- Las funciones matemáticas (math.h) en el archivo **libm.a**
- Implementaciones “elementales” (stubs) de las funciones de bajo nivel utilizadas por funciones de I/O (stdio.h), o de tiempo (time.h) en el archivo **libnosys.a**
- Archivo objeto para la inicialización del sistema **crt0.o**
- Bibliotecas para C++ (versiones normales y nano) y versiones “debugueables” de las bibliotecas de C

Biblioteca Estándar de C

Hay un conjunto de estos archivos de biblioteca para cada arquitectura, por lo que hay que enlazar repitiendo las opciones referidas a la arquitectura, para que el linker sepa qué versión de bibliotecas utilizar: `-mcpu=cortex-m3 -mthumb -mfloat-abi=soft`



Biblioteca Estándar de C

La mayoría de las funciones se utiliza normalmente, incluyendo el archivo de cabecera y realizando la invocación.
Por ejemplo, para utilizar strlen() hay que incluir string.h y para atan(), math.h. Con eso basta para compilar.

En la fase de enlazado hay que indicar con cuáles librerías se enlaza:
`-lm` (para libm.a)
`-lc` o `-lc_nano` (para libc.a/libc_nano.a)
`-lnosys` (para libnosys.a)

Biblioteca Estándar de C –un ejemplo con printf

En funciones de más alto nivel que

- Produzcan algún efecto en la salida estándar: printf(), puts(), putchar(), …
- Reciban datos de la entrada estándar: scanf(), gets(), getchar(), …
- Interactúen con un file-system: fopen(), fclose(), fseek(), …
- Trabajen con memoria dinámica: malloc(), …
- Accedan a determinado hardware: clock(), …

newlib y newlib-nano dejan a criterio del programador la implementación de las interacciones con el hardware o un eventual sistema operativo, por medio de llamadas a funciones que se realizan desde estas funciones de la biblioteca.
A estas funciones se las denomina llamadas al sistema (syscalls) y son las que deben ser implementadas por el programador del embebido.
Si se usa una función que las requiera y no se provee una definición de la misma fallará el proceso de enlazado.
El listado de funciones requeridas por las bibliotecas se encuentra en la documentación de newlib: _exit, _write,_wait, _unlink, _times, _symlink, _stat, _sbrk, _readlink, _read, _open, _lseek, _kill, _isatty, _gettod, _getpid, _fstat, _fork, _execve, _close

Biblioteca Estándar de C –un ejemplo con printf

Al usar printf() en algún lugar del código, surgirán los siguientes errores de enlazado:

```
.../ld.exe: .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-sbrkr.o): in function `_sbrk_r':
sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
.../ld.exe: .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-writer.o): in function `_write_r':
writer.c:(.text._write_r+0x10): undefined reference to `_write'
.../ld.exe: .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-closer.o): in function `_close_r':
closer.c:(.text._close_r+0xc): undefined reference to `_close'
.../ld.exe:   .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-lseekr.o): in function `_lseek_r':
lseekr.c:(.text._lseek_r+0x10): undefined reference to `_lseek'
.../ld.exe:   .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-readr.o): in function `_read_r':
readr.c:(.text._read_r+0x10): undefined reference to `_read'
.../ld.exe:   .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-fstatr.o): in function `_fstat_r':
fstatr.c:(.text._fstat_r+0xe): undefined reference to `_fstat'
.../ld.exe:   .../arm-none-eabi/lib/thumb/v7-m/nofp\libc_nano.a(lib_a-isattyr.o): in function `_isatty_r':
isattyr.c:(.text._isatty_r+0xc): undefined reference to `_isatty'
```

Biblioteca Estándar de C –un ejemplo con printf

La solución es la siguiente:

- Implementar la función `int _write(int file, char *ptr, int len)`, para que envíe al periférico deseado los `len` caracteres a partir de la dirección `ptr`. En este ejemplo se utilizará el periférico USART1.

- Definir en el linker script el símbolo `end` al final de la sección .bss, el cual marca el comienzo del *heap*. Este símbolo es utilizado por la función `_sbrk()`, invocada por `malloc()`, que es utilizada por `printf()`.

- Enlazar con libnosys.a (opción `-lnosys` en el comando de enlazado) que provee una implementación elemental de las *system calls.*

- Para el caso de `_write()`, que está definida tanto en el código fuente del proyecto como en la biblioteca nosys, el enlazador prioriza la definición provista en el código del usuario e ignora la implementación de la biblioteca.

Biblioteca Estándar de C –un ejemplo con printf

main.c

```c
#include <stdio.h>
#include <stdint.h>

void usart_setup();

uint32_t contador = 0;

void main(void){

    usart_setup();

    do{
        printf("contador = %ld\n", contador++);
        for(volatile int i=500000;i>0;i--);
    }while(1);
}
```

Printf_helper.c

```c
#include <stdint.h>
void usart_tx_buffer(uint8_t *byte, uint16_t length);
int _write(int file, char *ptr, int len);

int _write(int file, char *ptr, int len)
{
    usart_tx_buffer((uint8_t*)ptr,len);
    return len;
}
```

Biblioteca Estándar de C –un ejemplo con printf

usart.c

```c
#include <stdint.h>
void usart_setup();
void usart_tx(uint8_t byte);
void usart_tx_buffer(uint8_t *byte, uint16_t length);


/** Reference manual - p. 827 **/
typedef struct{
  uint32_t SR;         /*!< USART Status register,                   Address offset: 0x00 */
  uint32_t DR;         /*!< USART Data register,                     Address offset: 0x04 */
  uint32_t BRR;        /*!< USART Baud rate register,                Address offset: 0x08 */
  uint32_t CR1;        /*!< USART Control register 1,                Address offset: 0x0C */
  uint32_t CR2;        /*!< USART Control register 2,                Address offset: 0x10 */
  uint32_t CR3;        /*!< USART Control register 3,                Address offset: 0x14 */
  uint32_t GTPR;       /*!< USART Guard time and prescaler register, Address offset: 0x18 */
} USART_t;

#define USART1_BASE         0x40013800  /** Reference manual - p. 51 **/
#define USART1        (*(volatile USART_t *) (USART1_BASE))  /** Reference manual - p. 51 **/

void usart_tx(uint8_t byte){
    // espera a que se pueda escribir en el Data Register
    while((USART1.SR &= (1<<7)) == 0){
        __asm volatile ("nop");
    }
    //Escribe byte para transmitir.
    USART1.DR = byte;
}
```

Biblioteca Estándar de C –un ejemplo con printf

usart.c

```
void usart_setup(){

    volatile uint32_t * rcc_apb2enr = (volatile uint32_t *) (0x40021000 + 0x18);
    volatile uint32_t * gpioa_crh   = (volatile uint32_t *) (0x40010800 + 0x04);

    // habilita el clock del GPIOA (El TX1 es PA9)
    *rcc_apb2enr |= (1<<2);
    // habilita función alternativa
    *rcc_apb2enr |= (1<<0);
    //configura PA9 como salida con función alternativa push-pull
    *gpioa_crh &= ~(0b1111  << 4);
    *gpioa_crh |= (0b1011  << 4);
    // habilita el clock del USART1
    *rcc_apb2enr |= (1<<14);
    /** configura la USART1 según Referenc Manual - p. 792 **/
    //1. Enable the USART by writing the UE bit in USART_CR1 register to 1.
    USART1.CR1 |= (1<<13);
    //2. Program the M bit in USART_CR1 to define the word length.
    USART1.CR1 &= ~(1<<12);    /** 8 bits      **/
    //3. Program the number of stop bits in USART_CR2.
    USART1.CR2 &= ~(3<<12);    /** 1 stop bit  **/
    //5. Select the desired baud rate using the USART_BRR register.
    USART1.BRR = (52<<4)+1;    /** (52+1/16) para obtener 9600 bps  **/
    //6. Set the TE bit in USART_CR1 to send an idle frame as first transmission.
    USART1.CR1 |= (1<<3);
}
```

```
void usart_tx_buffer(uint8_t *byte, uint16_t length){
    uint16_t i=0;
    for(i = 0; i < length; i++){
        usart_tx(byte[i]);
    }
}
```

Biblioteca Estándar de C –un ejemplo con printf

script.ld

```
MEMORY
{
    flash(rx) : ORIGIN = 0x08000000, LENGTH = 64K
    sram(wrx)   : ORIGIN = 0x20000000, LENGTH = 20K
}

SECTIONS
{
    .isr_vectors :{KEEP(*(.isr_vectors))} > flash
    .text :{. = ALIGN(4); *(.text) *(.text*) }       > flash
    .rodata :{. = ALIGN(4); *(.rodata) *(.rodata*) . = ALIGN(4);}   > flash
    _sidata = LOADADDR(.data);
    .data : {. = ALIGN(4); _sdata = .; *(.data) *(.data*) . = ALIGN(4); _edata = .;} > sram AT > flash
    .bss : { _sbss = .; *(.bss)  *(.bss*) _ebss = .;} > sram
    end = .;
}
_estack = 0x20005000;
```

```
arm-none-eabi-gcc -c main.c -o main.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
arm-none-eabi-gcc -c usart.c -o usart.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
arm-none-eabi-gcc -c startup.c -o startup.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
arm-none-eabi-gcc -c printf_helper.c -o printf_helper.o -mcpu=cortex-m3 -mthumb -mfloat-abi=soft

arm-none-eabi-gcc *.o -o ejecutable.elf -mcpu=cortex-m3 -mthumb -mfloat-abi=soft
-T script.ld -nostartfiles -lc_nano -lnosys
```