Taller de Sistemas Embebidos
STM32 MCU – Memory layout

Información relevante

Taller de Sistemas Embebidos

Asignatura correspondiente a la **actualización 2023** del Plan de Estudios 2020 y resoluciones modificatorias, de Ingeniería Electrónica de FIUBA

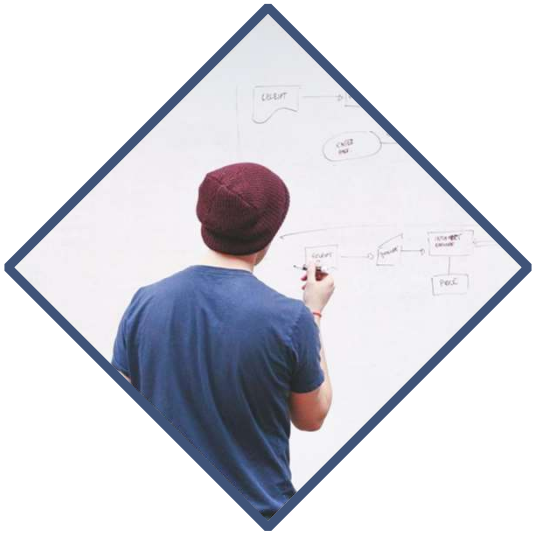
Estructura Curricular de la Carrera

El **Proyecto Intermedio** se desarrolla en la asignatura **Taller de Sistemas Embebidos**, la cual tiene un enfoque centrado en la **práctica propia de la carrera** más que en el desarrollo teórico disciplinar, con eje en la **participación de las y los estudiantes**

Más información . . .

. . . sobre la **actualización 2023** . . . https://www.fi.uba.ar/grado/carreras/ingenieria-electronica/plan-de-estudios

. . . sobre el **Taller de Sistemas Embebidos** . . . https://campusgrado.fi.uba.ar/course/view.php?id=1217



¡Hola!

Soy Juan Manuel Cruz
Taller de Sistemas Embebidos
Consultas a: jcruz@fi.uba.ar

1

Introducción

Actualización 2023 del Plan de Estudios 2020 y resoluciones . . .

Conceptos básicos

- Referencia:
  - Mastering STM32 - A step-by-step guide to the most complete ARM Cortex-M platform, using a free and powerful development environment based on Eclipse and GCC - Carmine Noviello (Author)
  - Chapter 17: Memory layout
    - Cada vez que compilamos nuestro firmware usando GCC ARM tool-chain, suceden una serie de cosas no triviales.
    - El compilador traduce el código fuente C en el assembly ARM y lo organiza para actualizarlo en una MCU STM32 determinada.

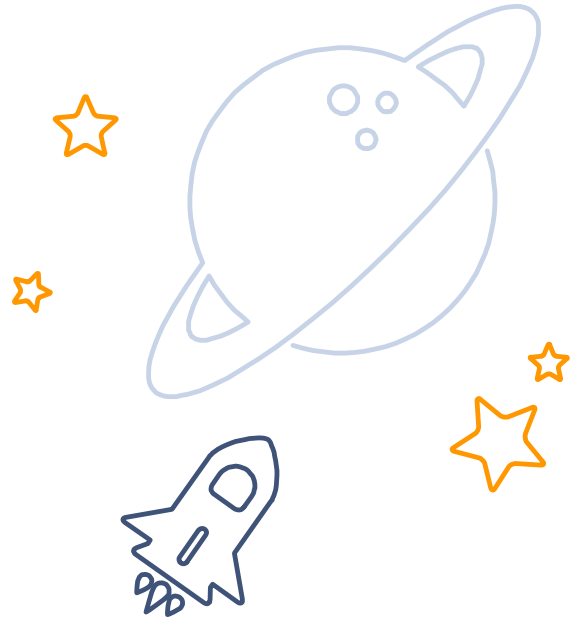
Conceptos básicos

- Cada arquitectura de microprocesador define un modelo de ejecución que debe "hacer coincidir" con el modelo de ejecución del lenguaje de programación C.
- Esto significa que durante el arranque (bootstrap) se realizan varias operaciones, cuya tarea es preparar el entorno de ejecución de nuestra aplicación: la creación del stack (pila) y del heap, la inicialización de la memoria de datos, la inicialización de la tabla de vectores son solo algunas de las actividades realizadas durante el inicio.
- Además, algunos microcontroladores STM32 proporcionan memorias adicionales o permiten conectar memorias externas mediante el controlador FSMC, que se pueden asignar a tareas específicas durante el ciclo de vida del firmware.

Conceptos básicos

- Este capítulo tiene como objetivo arrojar luz sobre aquellas preguntas que son comunes a muchos desarrolladores de STM32.
- ¿Qué sucede cuando la MCU se resetea?
- ¿Por qué es obligatorio proporcionar la función main()?
- ¿Y cuánto tiempo tarda en ejecutarse desde que se resetea la MCU?
- ¿Cómo almacenar variables en flash en lugar de SRAM?
- ¿Cómo utilizar la memoria STM32 CCM?

Solución Adecuada



. . . lo más simple posible, previa determinación del objetivo de excelencia a cumplir, obviamente contando con la documentación debida y recurriendo a la metodología de trabajo adecuada

2

Documentación debida

1er Cuatrimestre de 2024, dictado por primera vez . . .

Mastering STM32 - A step-by-step guide to the most complete ARM Cortex-M platform, using a free and powerful development environment based on Eclipse and GCC - Carmine Noviello (Author)

Chapter 17: Memory layout

The STM32 Memory Layout Model

- En el Capítulo 1 analizamos cómo las MCU STM32 organizan el espacio de direcciones de memoria de 4 GB.
- La Figura 4 de ese capítulo muestra claramente cómo los primeros 0,5 GB de memoria están dedicados al área de código.
- A su vez esta zona se subdivide en varias subregiones.
- El más importante, a partir de la dirección 0x0800 0000, está dedicado al mapeo de la memoria flash interna.
- En cambio, la memoria SRAM interna parte de la dirección 0x2000 0000, y está organizada en varias subregiones dedicadas a tareas específicas que veremos en un momento.

The STM32 Memory Layout Model

- La Figura 1 muestra el diseño típico de las memorias flash y SRAM en una MCU STM32.
- En el Capítulo 7 aprendimos que los bytes iniciales de la memoria flash están dedicados al puntero de Stack principal (MSP) y a la tabla de vectores.
- El MSP contiene la dirección donde comienza la pila.
- La arquitectura Cortex-M ofrece la máxima libertad para colocar la pila en la memoria SRAM así como en otras memorias internas (por ejemplo, la RAM CCM disponible en algunas MCU STM32) o externas (conectadas al controlador FSMC). Esto explica la necesidad del MSP.

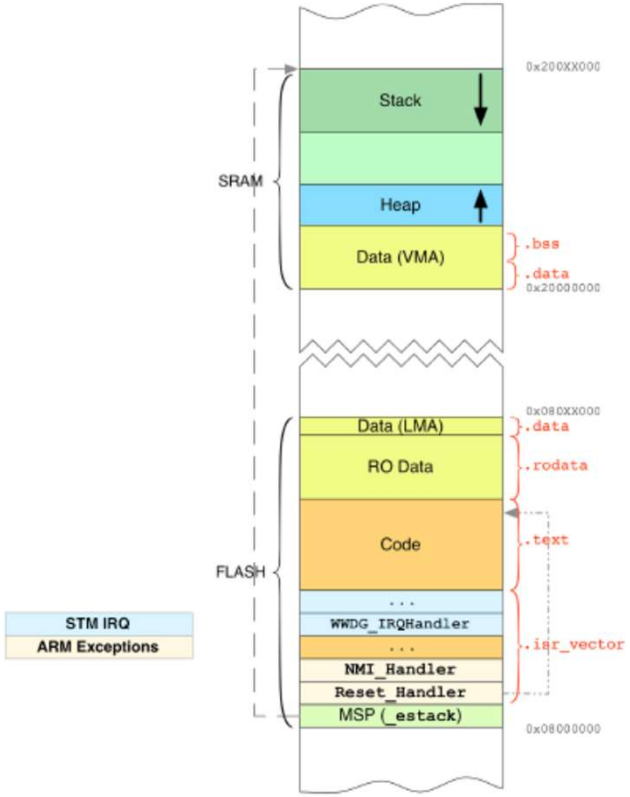


Figure 1: The typical layout of flash and SRAM memories

The STM32 Memory Layout Model

- La memoria flash también se puede utilizar para almacenar datos de sólo lectura, también conocidos como datos constantes debido a que las variables declaradas como constantes se colocan automáticamente en esta memoria.
- Finalmente, la memoria flash contiene el código assembly generado a partir del código fuente C.
- La memoria SRAM también está organizada en varias subregiones.
- Una región de tamaño variable que comienza desde el final de SRAM y crece hacia abajo (es decir, su dirección base tiene la dirección SRAM más alta) está dedicada a la pila.
- Esto sucede porque los núcleos Cortex-M utilizan un modelo de memoria de Stack llamado Stack full-descendente.

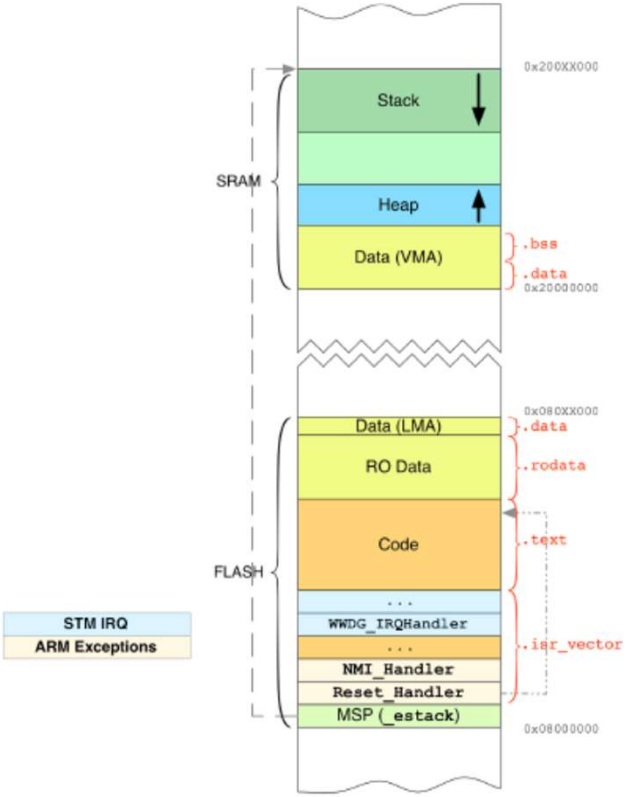


Figure 1: The typical layout of flash and SRAM memories

The STM32 Memory Layout Model

- El puntero de la pila base, también llamado puntero de Stack principal (MSP), se calcula en el momento de la compilación y se almacena en la ubicación de la memoria flash 0x0800 0000.
- Una vez que llamamos a una función, se inserta un nuevo Stack frame en la pila.
- Esto significa que el puntero al Stack frame actual (SP) disminuye automáticamente en cada llamada de función (por ejemplo, la instrucción push del assembly ARM lo disminuye automáticamente).
- La SRAM también se utiliza para almacenar datos variables y esta región generalmente comienza al comienzo de la SRAM (0x2000 0000).
- Esta región, a su vez, se divide entre datos inicializados y no inicializados.

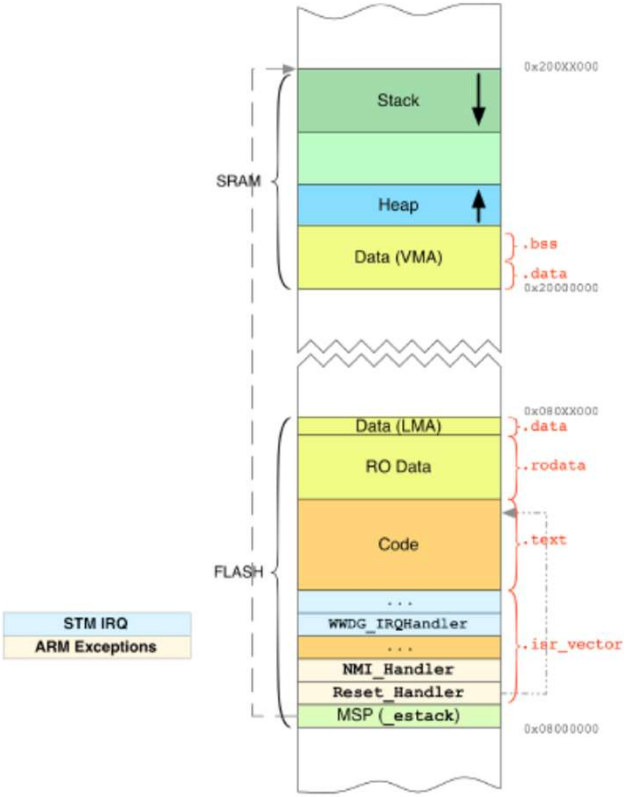


Figure 1: The typical layout of flash and SRAM memories

The STM32 Memory Layout Model

- Para entender la diferencia, consideremos este fragmento de código:

```
...
uint8_t var1 = 0xEF;
uint8_t var2;
...
```

- var1 y var2 son dos variables globales.
- var1 es una variable inicializada (fijamos su valor inicial en el momento de la compilación), mientras que el valor var2 no está inicializado: depende del tiempo de ejecución inicializarlo a cero.
- Por lo mismo tenemos dos apartados .data: uno almacenado en flash y otro en RAM, como veremos a continuación.

The STM32 Memory Layout Model

- Finalmente, la memoria SRAM podría contener otra región en crecimiento: el Heap
  - Almacena variables que se asignan dinámicamente durante la ejecución del firmware (mediante la rutina C malloc() o similar).
  - Esta área a su vez se puede organizar en varias subregiones, según el asignador utilizado (en el próximo capítulo veremos cómo FreeRTOS proporciona varios asignadores para manejar la asignación dinámica de memoria).
  - El Heap crece hacia arriba (es decir, la dirección base es la más baja de su región) y tiene un tamaño máximo fijo.
- Desde el punto de vista del compilador, estas secciones tradicionalmente reciben nombres diferentes dentro del binario de la aplicación.

The STM32 Memory Layout Model

- Por ejemplo, la sección que contiene el código ensamblador se denomina .text, .rodata es la que contiene variables y cadenas constantes, mientras que la sección para datos inicializados se denomina .data.
- Estos nombres también son comunes a otras arquitecturas informáticas, como x86 y MIPS. Otros son específicos del “mundo de los microcontroladores”.
- Por ejemplo, la sección .isr_vector es la designada para almacenar la tabla de vectores en MCU basadas en Cortex-M.
- Dado que cada MCU STM32 tiene su propia cantidad de SRAM y flash, y dado que cada programa tiene una cantidad variable de instrucciones y variables, el tamaño y la ubicación en la memoria de estas secciones difieren.
- Antes de que podamos ver cómo indicarle al compilador que genere el archivo binario para la MCU específica, debemos comprender todos los pasos y herramientas involucradas durante la generación de archivos objeto.

Understanding Compilation and Linking Processes

- El proceso que va desde la compilación del código fuente C hasta la generación de la imagen binaria final para flashear en nuestra MCU implica varios pasos y herramientas proporcionadas por la cadena de herramientas GCC.
- La Figura 2 intenta esbozar este proceso.
- Todo comienza desde los archivos fuente C
- Generalmente contienen las siguientes estructuras de programa.

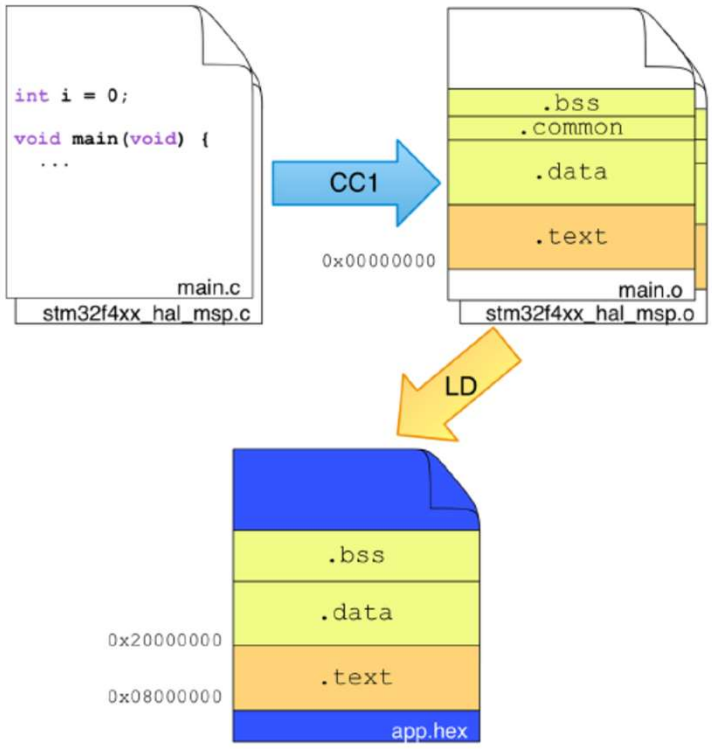


Figure 2: The compilation process from the source file to the final binary image

Understanding Compilation and Linking Processes

- Variables globales: éstas a su vez se pueden dividir entre variables no inicializadas e inicializadas; Una variable global también se puede definir como estática, es decir, su visibilidad está limitada al archivo fuente actual.
- Variables locales: se pueden dividir entre variables locales simples (también llamadas automáticas) y variables locales estáticas (es decir, aquellas variables cuya vida útil se extiende a lo largo de toda la ejecución del programa).
- Datos constantes: estos, a su vez, se pueden dividir entre tipos de datos constantes (por ejemplo, const int c = 5) y constantes de cadena (por ejemplo, "¡Hola mundo!").

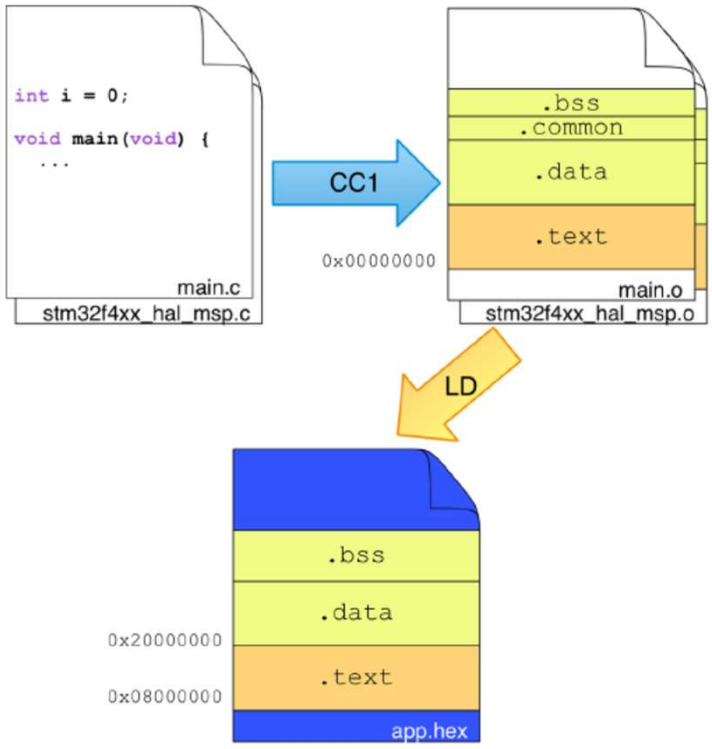


Figure 2: The compilation process from the source file to the final binary image

Understanding Compilation and Linking Processes

- Rutinas: constituyen el programa y se traducirán en instrucciones de montaje.
- Recursos externos: son tanto variables globales (declaradas como externas) como rutinas definidas en otros archivos fuente.
- Será trabajo de Linker "vincular" las referencias a estos símbolos definidos en otros archivos fuente y fusionar las secciones provenientes de los archivos binarios correspondientes.

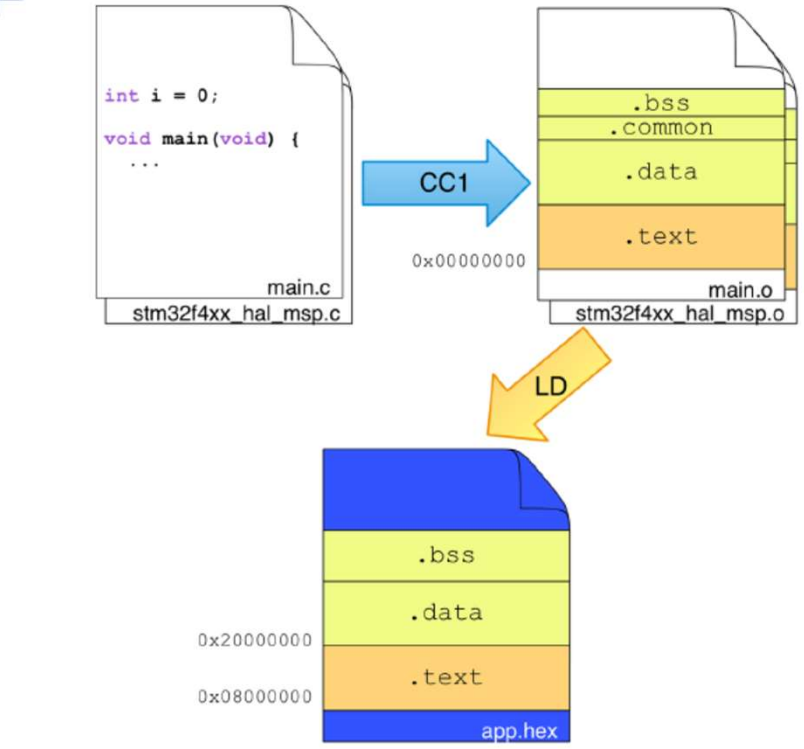


Figure 2: The compilation process from the source file to the final binary image

Understanding Compilation and Linking Processes

- Una vez que se compila un archivo fuente, las estructuras del programa anteriores se asignan dentro de secciones específicas del archivo binario.
- La Tabla 1 resume los más relevantes.

Table 1: The mapping of program structures and binary file sections

Language structure	Binary file section	Memory region at run-time
Global un-initialized variables	`.common`	Data (SRAM)
Global initialized variables	`.data`	Data (SRAM+Flash)
Global `static` un-initialized variables	`.bss`	Data (SRAM)
Global `static` initialized variables	`.data`	Data (SRAM+Flash)
Local variables	<no specific section>	Stack or Heap (SRAM)
Local `static` un-initialized variables	`.bss`	Data (SRAM)
Local `static` initialized variables	`.data`	Data (SRAM+Flash)
Const data types	`.rodata`	Code (Flash)
Const strings	`.rodata.1`	Code (Flash)
Routines	`.text`	Code (Flash)

Understanding Compilation and Linking Processes

- Para cada archivo fuente (.c) que compone nuestra aplicación, el compilador generará un archivo objeto correspondiente (.o), que contiene las secciones de la Tabla 1.
- Un archivo objeto es un tipo de archivo binario que cumple con un estándar bien conocido.
- Existen muchos estándares para archivos binarios (PE, COFF, ELF, etc.).
- El que utiliza GCC ARM es el ELF32, un estándar abierto muy popular, debido a su uso en Sistemas Operativos basados en Linux, y que es ampliamente soportado incluso por otras herramientas como OpenOCD y la utilidad ST-LINK.
- Sin embargo, los archivos que terminan en .o son un tipo especial de archivos objeto. También se conocen como archivos reubicables.
- Este nombre proviene de que todas las direcciones de memoria contenidas en este tipo de archivo son relativas al mismo archivo, y parte de la dirección 0x0000 0000.

Understanding Compilation and Linking Processes

- Esto significa que también la sección .text comenzará desde esa dirección, y sabemos que esto contrasta con la dirección inicial de la memoria flash (0x0800 0000) en una MCU STM32.
- A partir de una serie de archivos reubicables (más algunos otros archivos de configuración que veremos más adelante), el Linker ensamblará su contenido para formar un archivo objeto común que representará nuestro firmware para flashear en la MCU.
- En este proceso, llamado linkeo, el Linker reubicará todas las direcciones relativas a las direcciones de memoria reales.
- Este tipo de archivo también se conoce como archivo absoluto, porque todas las direcciones son absolutas y específicas de la MCU STM32 determinada.

Understanding Compilation and Linking Processes

- ¿Cómo sabe el Linker dónde colocar en la memoria las secciones contenidas en el archivo absoluto?
- Es gracias a los scripts del Linker (aquellos archivos que terminan en .ld) que podemos organizar el contenido del archivo absoluto de acuerdo con la distribución real de la memoria.
- Ya hemos visto un script de Linker en el Capítulo 4, cuando configuramos el archivo mem.ld para especificar la dirección de origen flash correcta.
- CubeMX también incorpora el script del Linker correcto para nuestra MCU dentro del proyecto C generado (está contenido dentro de la subcarpeta SW4STM32).
- Sin embargo, es realmente complicado estudiar el contenido de esos scripts si no dominamos varios conceptos antes.
- Por lo tanto, es mejor comenzar a crear sin problemas una aplicación STM32 básica.

The Really Minimal STM32 Application

- La mayoría de las aplicaciones vistas hasta ahora parecen realmente sencillas.
- En cambio, tanto desde el punto de vista de la organización de la memoria como desde el punto de vista de las operaciones realizadas cuando se inicia la MCU, ya ejecutan muchas operaciones bajo el capó.
- Por este motivo, vamos a construir una aplicación realmente esencial.
- El primer paso es crear un proyecto vacío usando Eclipse.
  - Vaya al menú Archivo->Nuevo->Proyecto C.
  - Elija el tipo de proyecto Vacío y seleccione la cadena de herramientas Cross ARM GCC, como se muestra en la Figura 3.
  - Complete el asistente del proyecto.
  - Cree ahora un nuevo archivo llamado main.c y coloque el siguiente código dentro de él.

The Really Minimal STM32

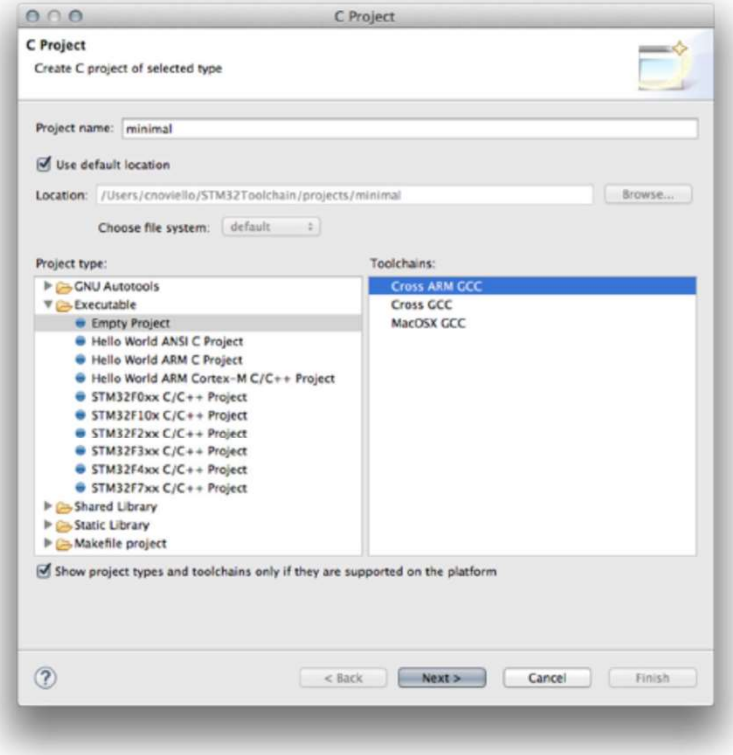


Figure 3: The project settings to choose

Filename: src/main-ex1.c

```
typedef unsigned long uint32_t;

/* Memory and peripheral start addresses (common to all STM32 MCUs) */
#define FLASH_BASE      0x08000000
#define SRAM_BASE       0x20000000
#define PERIPH_BASE     0x40000000

/* Work out end of RAM address as initial stack pointer
 * (specific of a given STM32 MCU */
#define SRAM_SIZE       96*1024     // STM32F401RE has 96 KB of RAM
#define SRAM_END        (SRAM_BASE + SRAM_SIZE)

/* RCC peripheral addresses applicable to GPIOA
 * (specific of a given STM32 MCU */
#define RCC_BASE        (PERIPH_BASE + 0x23800)
#define RCC_APB1ENR     ((uint32_t*)(RCC_BASE + 0x30))

/* GPIOA peripheral addresses
 * (specific of a given STM32 MCU */
#define GPIOA_BASE      (PERIPH_BASE + 0x20000)
#define GPIOA_MODER     ((uint32_t*)(GPIOA_BASE + 0x00))
#define GPIOA_ODR       ((uint32_t*)(GPIOA_BASE + 0x14))

/* User functions */
int main(void);
void delay(uint32_t count);

/* Minimal vector table */
uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
    (uint32_t *)SRAM_END,   // initial stack pointer
    (uint32_t *)main        // main as Reset_Handler
};

int main() {
    /* Enable clock on GPIOA peripheral */
    *RCC_APB1ENR = 0x1;
    /* Configure the PA5 as output pull-up */
    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1

    while(1) {
        *GPIOA_ODR = 0x20;
        delay(200000);
        *GPIOA_ODR = 0x0;
        delay(200000);
```



```
    }
}

void delay(uint32_t count) {
    while(count--);
}
```

The Really Minimal STM32 Application

- Las primeras 21 líneas contienen solo macros que definen las direcciones de periféricos STM32 más comunes.
- Algunos son genéricos y otros específicos de la MCU determinada.
- En la línea 26 estamos definiendo la tabla de vectores.
- Al ser “mínimo”, solo contiene dos cosas: la dirección en SRAM del SP (recuerde que esta es la primera entrada de la tabla de vectores y debe colocarse en la dirección 0x0800 0000) y el puntero al manejador de la excepción Reset.
- ¿Qué estamos haciendo exactamente?
- En el Capítulo 7 mencionamos que cuando la MCU se resetea, el controlador NVIC genera una excepción de Reset después de algunos ciclos.
- Esto significa que su handler es el punto de entrada real de nuestra aplicación, y desde allí comienza la ejecución del firmware.

The Really Minimal STM32 Application

- Aquí vamos a definir la función main() como el handler de la excepción de Reset.
- La palabra clave GCC __attribute__((section(".isr_vector"))) le dice al compilador que coloque la matriz vector_table dentro de la sección denominada .isr_vector, que a su vez estará contenida en el archivo objeto main.o.
- Finalmente, la rutina main() no contiene nada más que la clásica aplicación parpadeante.
- Antes de que podamos compilar el firmware, debemos especificar un par de configuraciones del proyecto.
- Vaya a Configuración del proyecto->Compilación C/C++->Configuración.
- En la configuración del Procesador de destino, seleccione el núcleo Cortex-M que se ajuste a su MCU.
- Luego vaya a la sección Cross ARM C Linker->General y marque la entrada No usar archivos de inicio estándar y desmarque la entrada Eliminar secciones no utilizadas, como se muestra en la Figura 4.

The Really Minimal STM32 Application

- Si intenta compilar la aplicación, verá la siguiente advertencia en la consola de Eclipse: advertencia: no se puede encontrar el símbolo de entrada _start; por defecto 0000000000008000
- ¿Qué significa?
- GCC (o mejor, LD) nos está diciendo que no sabe cuál es la rutina de entrada de nuestra aplicación (_start() - este nombre de punto de entrada es una convención en GCC) y no sabe en qué ubicación de memoria absoluta empieza a colocar el código.
- Entonces, ¿cómo podemos abordar esto?
- Necesitamos un Linker script.

The Really Minimal STM32 Application

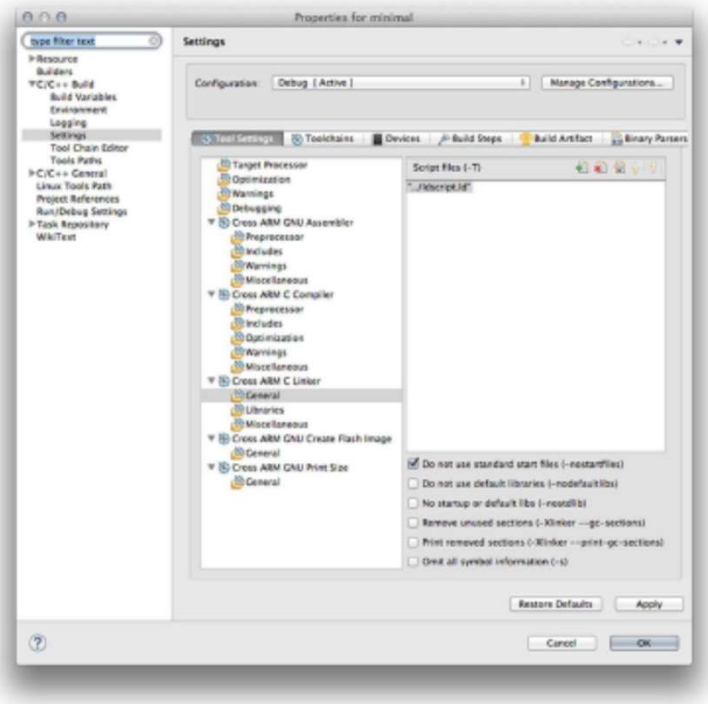


Figure 4: The project settings to choose

Filename: src/ldscript.ld

```
/* memory layout for an STM32F401RE */

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 512K
    SRAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 96K
}

ENTRY(main)

/* output sections */
SECTIONS
{
    /* program code into FLASH */
    .text : ALIGN(4)
    {
```



```
        *(.vector_table)       /* Vector table */
        *(.text)               /* Program code */
        KEEP(*(.vector_table))
    } >FLASH

    /* Initialized global and static variables (which
       we don't have any in this example) into SRAM */
    .data :
    {
        *(.data)
    } >SRAM
}
```

- Cree un nuevo archivo llamado ldscript.ld y coloque el siguiente contenido dentro de él.

The Really Minimal STM32 Application

- Veamos el contenido de este archivo.
- Las líneas [3:7] contienen la definición de las memorias flash y SRAM.
- Cada región puede tener varios atributos (w=escribible, r=legible, x=ejecutable).
- También especificamos su dirección inicial y longitud (en el ejemplo anterior están relacionados con una MCU STM32F401RE).
- La línea 9 especifica la función main() como punto de entrada de nuestra aplicación (anulando el símbolo _start predeterminado).
- Las líneas [12:28] definen el contenido de las secciones .text y .data. La sección .text estará compuesta primero por la tabla de vectores y luego por el código del programa.
- Con la directiva ALIGN(4) estamos diciendo que la sección está alineada con palabras (4 bytes), mientras que la directiva >FLASH especifica que la sección .text se colocará dentro de la memoria flash.

The Really Minimal STM32 Application

- KEEP(*(.isr_vector)) le dice a LD que mantenga la tabla de vectores dentro del archivo absoluto final; de lo contrario, la sección podría ser "eliminada" por otras herramientas que realizan optimizaciones en el archivo final.
- Finalmente, también se define la sección .data (aunque en este ejemplo no contiene nada), y se coloca dentro de la memoria SRAM.
- Antes de que podamos compilar el firmware, debemos indicarle a Eclipse que incluya el Linker script durante la compilación.
- Vaya a Configuración del proyecto->Compilación C/C++->Configuración.
- En la sección Cross ARM C Linker->General, agregue la entrada "../ldscript.ld" a la lista de archivos de script (-T).
- Ahora puedes compilar el firmware y actualizar tu Nucleo. Felicitaciones: es casi imposible tener una aplicación STM32 más pequeña.

ELF Binary File Inspection

- Un archivo binario ELF se puede inspeccionar utilizando una serie de herramientas proporcionadas por la cadena de herramientas GNU ARM.
- objdump y readelf son los más comunes.
- Describir su uso está fuera del alcance de este libro.
- Sin embargo, se recomienda encarecidamente dedicar un par de horas a jugar con sus parámetros opcionales en la línea de comandos.
- Comprender cómo se crea un archivo binario puede mejorar drásticamente el conocimiento de lo que hay debajo del capó.
- Por ejemplo, ejecutar objdump con el parámetro -h muestra el contenido de todas las secciones contenidas en el binario del firmware.

ELF Binary File Inspection

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main    00000040  08000008  08000008  00008008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay   00000020  08000048  08000048  00008048  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment      00000070  00000000  00000000  0000b1d2  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  0000b242  2**0
                  CONTENTS, READONLY
```

- Al observar el resultado anterior, vemos varias cosas con respecto a las secciones contenidas en el archivo binario.
- Cada sección tiene un tamaño, expresado en bytes.
- Una sección también tiene dos direcciones: la dirección de memoria virtual (VMA) y la dirección de memoria de carga (LMA).

ELF Binary File Inspection

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main    00000040  08000008  08000008  00008008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay   00000020  08000048  08000048  00008048  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment      00000070  00000000  00000000  0000b1d2  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  0000b242  2**0
                  CONTENTS, READONLY
```

- En sistemas embebidos como las MCU STM32, el VMA es la dirección que tendrá la sección cuando el firmware comience a ejecutarse.
- El LMA es la dirección en la que se cargará la sección.
- En la mayoría de los casos, las dos direcciones serán la misma.
- Como descubriremos en la siguiente sección, difieren para la región .data.

ELF Binary File Inspection

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main    00000040  08000008  08000008  00008008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay   00000020  08000048  08000048  00008048  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment      00000070  00000000  00000000  0000b1d2  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  0000b242  2**0
                  CONTENTS, READONLY
```

- Cada sección tiene varios atributos que le dicen al cargador (en nuestro caso, por ejemplo, el cargador es GDB junto con OpenOCD o la herramienta de utilidad ST-LINK) qué hacer con la sección dada.
- Veamos qué significan:
  - CONTENTS: este atributo le dice al cargador que la sección del archivo binario contiene datos para cargar en la dirección LMA final.

ELF Binary File Inspection

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main    00000040  08000008  08000008  00008008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay   00000020  08000048  08000048  00008048  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment      00000070  00000000  00000000  0000b1d2  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  0000b242  2**0
                  CONTENTS, READONLY
```

- Como veremos a continuación, la sección .bss no tiene contenido en un archivo binario.
- ALLOC: esto dice asignar un espacio correspondiente en la memoria LMA (que puede ser tanto memoria flash como SRAM). La dimensión del espacio asignado viene dada por la columna Tamaño.

ELF Binary File Inspection

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main    00000040  08000008  08000008  00008008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay   00000020  08000048  08000048  00008048  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment      00000070  00000000  00000000  0000b1d2  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  0000b242  2**0
                  CONTENTS, READONLY
```

- LOAD: indica cargar los datos de la sección contenida en el archivo binario a la memoria final del LMA.
- READONLY: indica que el contenido de la sección es de sólo lectura.
- CODE: indica que el contenido de la sección es código binario.

ELF Binary File Inspection

- Otra cosa interesante a destacar del resultado anterior es que el archivo binario contiene una sección dedicada para cada ejecutable contenido en el código fuente (.text.main para main() y .text.delay para delay()).
- Tenemos que especificar al Linker que fusione todas las secciones .text en una sección común completa, modificando el Linker script de esta manera:

```
.text : ALIGN(4)
{
    *(.isr_vector)    /* Vector table */
    *(.text)          /* Program code */
    *(.text*)         /* Merge all .text.* sections inside the .text section */
    KEEP(*(.isr_vector))
} >FLASH
```

ELF Binary File Inspection

```
.text : ALIGN(4)
{
    *(.isr_vector)     /* Vector table */
    *(.text)           /* Program code */
    *(.text*)          /* Merge all .text.* sections inside the .text section */
    KEEP(*(.isr_vector))
} >FLASH
```

- Como veremos más adelante, la capacidad de tener secciones separadas para cada invocable nos permite colocar selectivamente algunas funciones dentro de diferentes memorias (por ejemplo, la memoria rápida CCM en algunas MCU STM32).
- Finalmente, la columna Archivo desactivado especifica el desplazamiento de la sección dentro del archivo binario, mientras que la columna Algn indica la alineación de los datos en la memoria, que es de 4 bytes.

.data and .bss Sections Initialization

- Introduzcamos una pequeña modificación al ejemplo anterior.
- Esta vez usamos una variable global inicializada, dataVar, para iniciar el ciclo parpadeante.
- La variable ha sido declarada volátil solo para evitar que el compilador la optimice (sin embargo, al compilar este ejemplo, desactive todas las optimizaciones [-ON] en la configuración del proyecto).
- Viendo el código podemos llegar a la conclusión de que hace lo mismo que el ejemplo anterior.
- Sin embargo, si intentas hacer parpadear tu Nucleo, verás que el LED LD2 no parpadea.
- ¿Por qué no?

```
volatile uint32_t dataVar = 0x3f;

int main() {
    /* enable clock on GPIOA and GPIOC peripherals */
    *RCC_APB1ENR = 0x1 | 0x4;
    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1

    while(dataVar == 0x3f) { // This is always true
        *GPIOA_ODR = 0x20;
        delay(200000);
        *GPIOA_ODR = 0x0;
        delay(200000);
    }
}
```

.data and .bss Sections Initialization

- Para entender lo que está pasando, tenemos que revisar algunas cosas del lenguaje de programación C.
- Considere el siguiente fragmento de código:

```
...
uint32_t globalVar = 0x3f;

void foo() {
        volatile uint32_t localVar = 0x4f;

        while(localVar--);
}
```

- Aquí tenemos dos variables: una definida en el ámbito global y otra localmente.
- La variable localVar se inicializa con el valor 0x4f
- ¿Cuándo sucede esto exactamente?

.data and .bss Sections Initialization

- La inicialización se ejecuta cuando se invoca la rutina foo(), como se muestra en el siguiente código ensamblador:
- Las líneas [2:4] son el prólogo de la función.
- Cada rutina es responsable de asignar su propio marco de pila, guardando algunos registros internos de la CPU.
- Esto también se denomina convención de llamada, y la forma en que se realiza está definida por un estándar específico (en el caso de procesadores basados en ARM, está definido por el Estándar de llamada a procedimiento de arquitectura ARM (AAPCS)).
- No entraremos en detalles sobre este asunto aquí, porque analizaremos mejor la convención de llamadas de ARM en el siguiente capítulo.

```
1  void foo() {
2     0:   b480            push    {r7}            ;Save the current FP
3     2:   b083            sub     sp, #12         ;Allocate 12 bytes on the stack
4     4:   af00            add     r7, sp, #0      ;Save the new FP
5          volatile uint32_t localVar = 0x4f;
6     6:   234f            movs    r3, #79         ;Place 0x4f in r3
7     8:   607b            str     r3, [r7, #4]    ;Store r3 (that is 0x4f) in the 4-th byte
8
9          while(localVar--);
10    a:   bf00            nop
11    c:   687b            ldr     r3, [r7, #4]
12    e:   1e5a            subs    r2, r3, #1
13   10:   607a            str     r2, [r7, #4]
14   12:   2b00            cmp     r3, #0
15   14:   d1fa            bne.n   c <foo+0xc>
16 }
```

.data and .bss Sections Initialization

- Entonces, la convención de llamada define que las variables locales se inicializan automáticamente al llamar a la función.
- ¿Qué pasa con las variables globales? Dado que no participan en un proceso de llamada, deben inicializarse mediante algún código específico cuando se resetea la MCU (recuerde que la SRAM es volátil y su contenido no está definido después de un Reset).
- Esto significa que tenemos que proporcionar una función de inicialización específica.

```
1  void foo() {
2     0:   b480            push    {r7}            ;Save the current FP
3     2:   b083            sub     sp, #12         ;Allocate 12 bytes on the stack
4     4:   af00            add     r7, sp, #0      ;Save the new FP
5          volatile uint32_t localVar = 0x4f;
6     6:   234f            movs    r3, #79         ;Place 0x4f in r3
7     8:   607b            str     r3, [r7, #4]    ;Store r3 (that is 0x4f) in the 4-th byte
8
9          while(localVar--);
10    a:   bf00            nop
11    c:   687b            ldr     r3, [r7, #4]
12    e:   1e5a            subs    r2, r3, #1
13   10:   607a            str     r2, [r7, #4]
14   12:   2b00            cmp     r3, #0
15   14:   d1fa            bne.n   c <foo+0xc>
16 }
```

.data and .bss Sections Initialization

- La siguiente rutina se puede utilizar para simplemente copiar el contenido de la región flash que contiene los valores de inicialización a la región SRAM dedicada a las variables inicializadas globales.

```
void __initialize_data (unsigned int* flash_begin, unsigned int* data_begin,
                        unsigned int* data_end) {
  unsigned int *p = data_begin;
  while (p < data_end)
    *p++ = *flash_begin++;
}
```

- Antes de que podamos usar esta rutina, necesitamos definir algunas cosas más.
- En primer lugar, debemos indicarle a LD que almacene los valores de inicialización para cada variable contenida en la sección .data dentro de una región específica de la memoria flash, que corresponderá a la dirección de memoria LMA.

.data and .bss Sections Initialization

- En segundo lugar, necesitamos una forma de pasar a la función __initialize_data() el inicio y el final de la sección .data en SRAM (que llamaremos _sdata y _edata respectivamente) y la ubicación inicial (que llamaremos _sidata) donde los valores de inicialización se almacenan en la memoria flash (es importante enfatizar que cuando inicializamos una variable a un valor dado necesitamos almacenar ese valor en algún lugar de la memoria flash y usarlo para inicializar la ubicación SRAM correspondiente a la variable).
- La Figura 3 esquematiza este proceso.
- Una vez más, todas estas operaciones las podemos realizar usando el linker script, que podemos modificar de la siguiente manera:

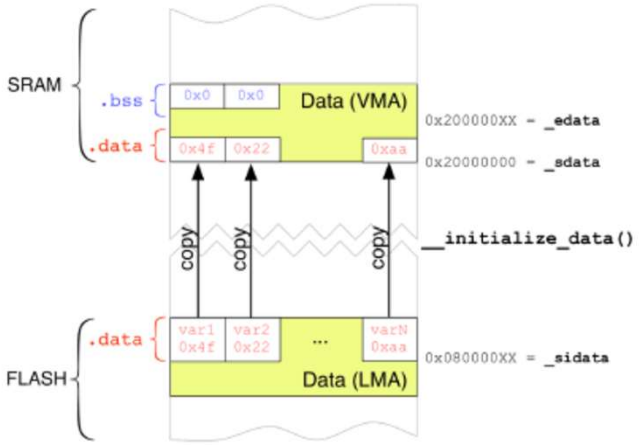


Figure 3: The copy process of initialized data from the flash to the SRAM memory

.data and .bss Sections Initialization

- La instrucción en la línea 26 define la variable _sidata, que contendrá la dirección LMA de la sección .data (es decir, la dirección inicial de la memoria flash que contiene los valores de inicialización).
- Las instrucciones en la línea [30:31] utilizan un operador especial: el "." operador. Se denomina contador de ubicación y es un contador que realiza un seguimiento de la ubicación de memoria alcanzada durante la generación de cada sección.
- El contador de ubicación cuenta de forma independiente la memoria de ubicación de cada región de memoria (SRAM, flash, etc.).
- Por ejemplo, en el código anterior, comienza desde 0x2000 0000 ya que la sección .data es la primera que se carga en SRAM.

```
25  /* Used by the startup to initialize data */
26  _sidata = LOADADDR(.data);
27
28  .data : ALIGN(4)
29  {
30      . = ALIGN(4);
31      _sdata = .;        /* create a global symbol at data start */
32
33      *(.data)
34      *(.data*)
35
36      . = ALIGN(4);
37      _edata = .;        /* define a global symbol at data end */
38  } >SRAM AT>FLASH
```

.data and .bss Sections Initialization

- Cuando se ejecutan las dos instrucciones *(.data) y *(.data*), el contador de ubicación se incrementa en el tamaño de todas las secciones .data contenidas en el archivo.
- Con la instrucción. = ALINEAR(4); simplemente estamos forzando que el contador de ubicación esté alineado con las palabras.
- Entonces, para resumir, _sdata contendrá 0x2000 0000 y _edata será igual al tamaño de la sección .data (en nuestro ejemplo, la sección .data contiene solo una variable -dataVar- y, por lo tanto, su tamaño es 0x2000 0004).
- Finalmente, la directiva >SRAM AT>FLASH le dice al editor de enlaces que la dirección VMA de la sección .data está vinculada al espacio de direcciones SRAM (es decir, 0x2000 0000), pero la dirección LMA (es decir, donde se almacenan los valores de inicialización)) se asigna dentro del espacio de la memoria flash.

```
25  /* Used by the startup to initialize data */
26  _sidata = LOADADDR(.data);
27
28  .data : ALIGN(4)
29  {
30      . = ALIGN(4);
31      _sdata = .;        /* create a global symbol at data start */
32
33      *(.data)
34      *(.data*)
35
36      . = ALIGN(4);
37      _edata = .;        /* define a global symbol at data end */
38  } >SRAM AT>FLASH
```

.data and .bss Sections Init

- Gracias a esta nueva configuración de diseño de memoria, ahora podemos organizar el archivo main.c de la siguiente manera:

Filename: src/main-ex2.c

```
22 void _start (void);
23 int main(void);
24 void delay(uint32_t count);
25
26 /* Minimal vector table */
27 uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
28     (uint32_t *)SRAM_END,   // initial stack pointer
29     (uint32_t *)_start        // main as Reset_Handler
30 };
31
32 // Begin address for the initialisation values of the .data section.
33 // defined in linker script
34 extern uint32_t _sidata;
35 // Begin address for the .data section; defined in linker script
36 extern uint32_t _sdata;
37 // End address for the .data section; defined in linker script
38 extern uint32_t _edata;
39
40
41 volatile uint32_t dataVar = 0x3f;
42
43 inline void
44 __initialize_data (uint32_t* flash_begin, uint32_t* data_begin,
45                    uint32_t* data_end) {
46   uint32_t *p = data_begin;
47   while (p < data_end)
48     *p++ = *flash_begin++;
49 }
50
51 void __attribute__ ((noreturn,weak))
52 _start (void) {
53   __initialize_data(&_sidata, &_sdata, &_edata);
54   main();
55
56   for(;;);
57 }
58
59 int main() {
60
61     /* enable clock on GPIOA and GPIOC peripherals */
62     *RCC_APB1ENR = 0x1 | 0x4;
63     *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
64
65     while(dataVar == 0x3f) {
66       *GPIOA_ODR = 0x20;
67       delay(300000);
68       *GPIOA_ODR = 0x0;
69       delay(300000);
70     }
71 }
```

.data and .bss Sections Initialization

- El punto de entrada ahora es la rutina _start(), que se utiliza como controlador para la excepción Reset.
- Cuando la MCU se resetea, se llama automáticamente y, a su vez, llama a la función __initialize_data(), pasando los parámetros _sidata, _sdata y _edata calculados por el Linker durante el proceso de vinculación. _start() luego llama a la rutina main(), que ahora funciona como se esperaba.
- Usando la herramienta objdump podemos comprobar cómo están organizadas las secciones en el archivo ELF.

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000c0  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  20000000  080000c0  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .comment      00000070  00000000  00000000  00010004  2**0
                  CONTENTS, READONLY
  3 .ARM.attributes 00000033  00000000  00000000  00010074  2**0
                  CONTENTS, READONLY
```

.data and .bss Sections Initialization

- Como puede ver, la herramienta confirma que la sección .data tiene un tamaño igual a 4 bytes, una dirección VMA igual a 0x2000 0000 y una dirección LMA igual a 0x0800 00c0, que corresponde al final de la sección .text.
- Lo mismo se aplica a la sección .bss, que está reservada para variables no inicializadas.
- Según el estándar ANSI C, el contenido de esta sección debe inicializarse en 0.
- Sin embargo, la sección .bss no tiene una región flash correspondiente que contenga todos ceros, pero depende nuevamente del código de inicio inicializar esta región.
- El siguiente fragmento de secuencia de comandos del linker muestra la definición de la sección .bss:

.data and .bss Sections Initialization

```
25   /* Uninitialized data section */
26       .bss : ALIGN(4)
27       {
28         /* This is used by the startup in order to initialize the .bss section */
29         _sbss = .;         /* define a global symbol at bss start */
30         *(.bss .bss*)
31         *(COMMON)
32
33         . = ALIGN(4);
34         _ebss = .;         /* define a global symbol at bss end */
35       } >SRAM AT>SRAM
```

- mientras que la siguiente rutina, siempre invocada desde la _start(), se utiliza para poner a cero la región .bss en SRAM:

```
void __initialize_bss (unsigned int* bss_begin, unsigned int* bss_end) {
  unsigned int *p = bss_begin;
  while (p < bss_end)
    *p++ = 0;
}
```

.data and .bss Sections Initialization

- Cambiar la rutina main() de la siguiente manera nos permite comprobar que todo funciona correctamente:

Filename: src/main-ex3.c

```
70  volatile uint32_t dataVar = 0x3f;
77  volatile uint32_t bssVar;
78
79  int main() {
80
81      /* enable clock on GPIOA and GPIOC peripherals */
82      *RCC_APB1ENR = 0x1 | 0x4;
83      *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
84
85      while(bssVar == 0) {
86        *GPIOA_ODR = 0x20;
87        delay(200000);
88        *GPIOA_ODR = 0x0;
89        delay(200000);
90      }
91  }
```

.data and .bss Sections Initialization

- Una vez más, podemos ver cómo se organiza la sección .bss invocando la herramienta objdump en el archivo binario final.

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000e8  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  20000000  080000e8  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  20000004  20000004  00010004  2**2
                  ALLOC
  3 .comment      00000070  00000000  00000000  00010004  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes 00000033  00000000  00000000  00010074  2**0
                  CONTENTS, READONLY
```

- El resultado anterior muestra que la sección tiene un tamaño igual a cuatro bytes, pero no ocupa espacio en el archivo binario final ya que la sección solo tiene el atributo ALLOC.

A Word About the COMMON Section

- En el Linker script anterior hemos utilizado la directiva especial * (COMÚN) durante la definición de la sección .bss.
- Esto simplemente le dice al LD que combine el contenido de la sección común dentro de la sección .bss.
- Pero ¿qué es exactamente la sección común?
- Para comprender mejor su función, necesitamos revisar algunas características poco conocidas del lenguaje C.
- Supongamos que tenemos dos archivos fuente y ambos definen dos variables inicializadas globales con el mismo nombre:

File A.c

```
int globalVar[3] = {0x1, 0x2, 0x3};
...
```

File B.c

```
int globalVar[3] = {0x1, 0x2, 0x3};
...
```

A Word About the COMMON Section

- Cuando intentamos generar la aplicación final vinculando los dos archivos reubicables (.o), obtenemos el siguiente error:

```
B.o:(.data+0x0): multiple definition of 'globalVar'
A.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

- La razón por la que esto sucede es evidente: estamos definiendo la misma variable global en dos archivos fuente diferentes.
- Pero ¿qué pasa si declaramos los dos símbolos como variables globales no inicializadas?

File A.c

```
int globalVar[3];
...
```

File B.c

```
int globalVar[6];
...
```

A Word About the COMMON Section

- Si intenta generar el archivo binario final, descubrirá que el Linker no genera errores.
- ¿Por qué el linker se queja de ambas definiciones de símbolos?
- Porque el Estándar C no dice nada que lo prohíba.
- Pero si el lenguaje esencialmente permite definir varias veces una variable global no inicializada, ¿cuánta memoria se asignará? (es decir, ¿globalVar será una matriz que contiene 3 o 6 elementos?).
- Este aspecto se deja a la implementación del compilador.
- Las versiones recientes de GCC colocan todas las variables globales no inicializadas (no declaradas como estáticas) dentro de una sección "común" completa, y la cantidad de memoria para un símbolo dado asumirá el valor del mayor (en nuestro caso, la matriz tendrá espacio para seis elementos de tipo int (es decir, 12 bytes).

A Word About the COMMON Section

- Entonces, para resumir, las variables globales estáticas no inicializadas son locales para un reubicable determinado y, por lo tanto, van en su sección .bss.
- Las variables globales no inicializadas son globales para toda la aplicación y van dentro de la sección común.
- El Linker script anterior coloca ambos tipos de variables globales no inicializadas dentro de la sección .bss, que la rutina __initialize_bss() pondrá a cero en tiempo de ejecución.
- Este comportamiento se puede anular especificando la opción -fno-common en el comando GCC. GCC asignará variables globales no inicializadas dentro de la sección .data, inicializándolas a cero.
- Esto significa que, si declaramos una matriz global no inicializada de 1000 elementos, la sección .data contendrá mil veces el valor 0: esto desperdiciará mucha memoria flash.
- Por lo tanto, para aplicaciones embebidas es mejor evitar usar esa opción de línea de comando.

.rodata Section

- Un programa normalmente utiliza datos constantes.
- Las cadenas y las constantes numéricas son sólo dos ejemplos, pero también se pueden inicializar grandes matrices de datos como constantes (por ejemplo, un archivo HTML utilizado para generar páginas web se puede convertir en una matriz, utilizando herramientas como el comando xxd UNIX).
- Al ser inmutables, los datos constantes se pueden colocar dentro de la memoria flash interna (o dentro de memorias flash externas conectadas a la MCU a través de la interfaz Quad-SPI) para ahorrar espacio SRAM.
- Esto se puede lograr simplemente definiendo la sección .rodata dentro del Linker script:

```
/* Constant data goes into flash */
.rodata : ALIGN(4)
{
  *(.rodata)         /* .rodata sections (constants) */
  *(.rodata*)        /* .rodata* sections (strings, etc.) */
} >FLASH
```

.rodata Section

- Por ejemplo, considerando este código C:

Filename: src/main-ex4.c

```
76  const char msg[] = "Hello World!";
77  const float vals[] = {3.14, 0.43, 1.414};
78
79  int main() {
80      /* enable clock on GPIOA and GPIOC peripherals */
81      *RCC_APB1ENR = 0x1 | 0x4;
82      *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
83
84      while(vals[0] >= 3.14) {
85        *GPIOA_ODR = 0x20;
86        delay(200000);
87        *GPIOA_ODR = 0x0;
88        delay(200000);
89      }
90  }
```

- tenemos que tanto la cadena msg como el array vals se colocan dentro de la memoria flash, como lo muestra la herramienta objdump:

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000590  08000000  08000000  00008000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000024  08000590  08000590  00008590  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000070  00000000  00000000  000085b4  2**0
                  CONTENTS, READONLY
  3 .ARM.attributes 00000033  00000000  00000000  00008624  2**0
                  CONTENTS, READONLY
```

.rodata Section

- Pointers to Const Data
  - Preste atención a declarar una cadena de esta manera:

```
char *msg = "Hello World!";
...
```

  - es completamente diferente a declararlo de esta otra manera:

```
char msg[] = "Hello World!";
...
```

- En el primer caso, declaramos un puntero a una matriz constante, lo que implica que se asignará una palabra dentro de la sección .data para almacenar la ubicación en la memoria flash de la cadena "¡Hola mundo!".
- En el segundo caso, en cambio, estamos definiendo correctamente una matriz de caracteres.
- Recuerde que en C los arreglos no son punteros.

Stack and Heap Regions

- Ya hemos visto en la Figura 1 que el Heap y el Stackson dos regiones dinámicas de la memoria SRAM que crecen en dirección opuesta.
- El Stack es una estructura descendiente, que crece desde el final de SRAM hasta el final de la sección .bss, o el final del Heap si se usa.
- El Heap crece en la dirección opuesta.
- Si bien el Stack es una estructura obligatoria en C, el Heap se utiliza sólo si se necesita una asignación de memoria dinámica.
- En algunos campos de aplicación (como en el área de la automoción) la asignación dinámica no se utiliza, o al menos se recomienda encarecidamente no utilizarla, debido al riesgo que implica.
- Una gestión decente del Heap introduce muchas penalizaciones en el rendimiento y es la fuente de posibles fugas y fragmentación de la memoria.

Stack and Heap Regions

- Sin embargo, si su aplicación necesita asignar dinámicamente algunas partes de la memoria, puede considerar utilizar la rutina clásica malloc() de la biblioteca C. Consideremos el siguiente ejemplo:

Filename: src/main-ex5.c

```
107 int main() {
108     /* enable clock on GPIOA and GPIOC peripherals */
109     *RCC_APB1ENR = 0x1 | 0x4;
110     *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
111
112     char *heapMsg = (char*)malloc(sizeof(char)*strlen(msg));
113     strcpy(heapMsg, msg);
114
115     while(strcmp(heapMsg, msg) == 0) {
116       *GPIOA_ODR = 0x20;
117       delay(200000);
118       *GPIOA_ODR = 0x0;
119       delay(200000);
120     }
121 }
```

- El código anterior es realmente simple. heapMsg es un puntero a una región de memoria asignada dinámicamente por la función malloc(). Simplemente copiamos el contenido de la cadena msg y comprobamos si ambas cadenas son iguales.
- Si es así, el LED LD2 comienza a parpadear.

Stack and Heap Regions

- Si intenta compilar el código anterior, verá el siguiente error de linkeo:

```
Invoking: Cross ARM C++ Linker
arm-none-eabi-g++ ...  ./src/ch10/main-ex5.o
/../../../../arm-none-eabi/lib/armv7e-m/libg_nano.a(lib_a-sbrkr.o): In function `_sbrk_r':
sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
collect2: error: ld returned 1 exit status
```

- ¿Lo que está sucediendo?
- malloc() se basa en la rutina _sbrk(), que es una característica que depende del sistema operativo y de la arquitectura.
- newlib deja al usuario la responsabilidad de proporcionar esta función.
- _sbrk() es una rutina que acepta la cantidad de bytes para asignar dentro de la memoria del Heap y devuelve el puntero al inicio de este "fragmento" contiguo de memoria.
- El algoritmo subyacente a la función _sbrk() es bastante simple:

Stack and Heap Regions

- Primero, debe verificar que haya suficiente espacio para asignar la cantidad de memoria deseada. Para realizar esta tarea, necesitamos una forma de proporcionar a la rutina _sbrk() el tamaño máximo de almacenamiento dinámico.
- Si el Heap tiene suficiente espacio para asignar la memoria necesaria, incrementa el tamaño del Heap actual y devuelve el puntero al comienzo del nuevo bloque de memoria.
- Si el Heap no tiene suficiente espacio (desbordamiento del Heap), entonces _sbrk() falla y corresponde al usuario proporcionar una respuesta de error.

El siguiente código muestra una posible implementación para la rutina _sbrk(). Analicemos su código.

Stack and Heap Regions

Filename: src/main-ex5.c

```
81  void *_sbrk(int incr) {
82      extern uint32_t _end_static; /* Defined by the linker */
83      extern uint32_t _Heap_Limit;
84
85      static uint32_t *heap_end;
86      uint32_t *prev_heap_end;
87
88      if (heap_end == 0) {
89        heap_end = &_end_static;
90      }
91      prev_heap_end = heap_end;
92
93  #ifdef __ARM_ARCH_6M__ //If we are on a Cortex-M0/0+ MCU
94      incr = (incr + 0x3) & (0xFFFFFFFC); /* This ensure that memory chunks are
95                                             always multiple of 4 */
96  #endif
97      if (heap_end + incr > &_Heap_Limit) {
98        asm("BKPT");
99      }
100
101     heap_end += incr;
102     return (void*) prev_heap_end;
```

- El Linker proporciona _end_static y _Heap_Limit y corresponden al final de la sección .bss y a la dirección de memoria más alta para la región del Heap (es decir, _Heap_Limit - _end_static es el tamaño del Heap).

Stack and Heap Regions

- Veremos en un momento cómo se definen dentro del Linker script.
- heap_end es una variable asignada estáticamente y se utiliza para realizar un seguimiento de la primera ubicación de memoria libre dentro del Heap.
  - Dado que es una variable local estática no inicializada, según la Tabla 1 se coloca dentro de la sección .bss y, por lo tanto, se pone a cero en tiempo de ejecución.
  - Entonces, la primera vez que se llama a _sbrk() es igual a cero y, por lo tanto, se inicializa con el valor de la variable _end_static.
- El if en la línea 97 garantiza que haya suficiente espacio en la memoria del Heap.
  - De lo contrario, se llama a la instrucción BKPT del ensamblador ARM, lo que provoca que el debugger detenga la ejecución.

Stack and Heap Regions

- La parte complicada está representada por las instrucciones en la línea [93:96].
  - La macro del preprocesador comprueba si la arquitectura ARM es ARMv6-M, es decir, las arquitecturas de procesadores basados en Cortex-M0/0+.
  - De hecho, esos procesadores no permiten el acceso a la memoria no alineada.
  - La instrucción en la línea 95 asegura que la memoria asignada sea siempre un múltiplo de 4 bytes.
- Nos queda analizar el Linker script. La parte que nos interesa comienza en la línea 51.

Filename: src/ldscript5.ld

```
51              _end_static = _ebss;
52              _Heap_Size = 0x190;
53              _Heap_Limit = _end_static + _Heap_Size;
```

Stack and Heap Regions

- _end_static no es más que un alias de la ubicación de memoria _ebss, que es el final de la sección .bss.
- _Heap_Size lo fijamos nosotros y establece la dimensión del Heap (400 bytes).
- Finalmente, _Heap_Limit no contiene nada más que la dirección final de la memoria del Heap.
- A Note About Linker Script Symbols
  - En este capítulo hemos utilizado ampliamente símbolos definidos en linker scripts del código fuente C.
  - Para cada símbolo, hemos definido una variable externa uint32_t _symbol correspondiente.
  - Cada vez que necesitamos acceder al contenido de ese símbolo, utilizamos la sintaxis &_symbol. Esto podría ser una fuente de confusión.

Stack and Heap Regions

- La forma en que se manejan los símbolos en los linker scripts es diferente a la de C.
- En C, un símbolo es de triple efecto, de símbolo, de ubicación de memoria y de valor.
- Los símbolos en escrituras similares son tuplas, formadas por el símbolo y su ubicación de memoria.
- Entonces los símbolos son contenedores de ubicaciones de memoria, como serían punteros, sin ningún valor. Entonces la siguiente instrucción:

```
extern uint32_t _symbol;
uint32_t symbol_value = _symbol;
```

- no tiene ningún significado (no hay ningún valor correspondiente para _symbol).
- Si bien esta forma de tratar con los símbolos del Linker podría ser obvia si el _symbol es una ubicación de memoria, es una fuente de muchos errores en caso de que sea un valor constante.

Stack and Heap Regions

- Por ejemplo, para recuperar el valor _Heap_Size en C tenemos que usar el siguiente código:

```
unsigned int heapSize = (unsigned int)&_Heap_Size;
```

- _Heap_Size, nuevamente contiene el tamaño del Heap como una dirección (es decir, 0x00000190), pero no es una dirección STM32 válida.
- Este hecho también se puede analizar inspeccionando la tabla de símbolos del archivo binario final, utilizando la herramienta objdump con el parámetro de línea de comando -t.

Checking the Size of Heap and Stack at Compile-Time

- Los microcontroladores tienen recursos de memoria limitados.
- Especialmente con las MCU STM32 de Value-lines, es muy común exceder la memoria SRAM máxima.
- También podemos usar el Linker script para agregar una especie de verificación "estática" sobre el uso máximo de memoria.
- La siguiente sección del linker script ayuda a garantizar que no estemos usando demasiada SRAM:

```
_Min_Stack_Size = 0x200;

/* User_heap_stack section, used to check that there is enough RAM left */
._user_heap_stack :
{
  . = ALIGN(4);
  . = . + _Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(4);
} >SRAM
```

Checking the Size of Heap and Stack at Compile-Time

- Con el código anterior, estamos definiendo una sección "ficticia" dentro del archivo binario final.
- Usando el operador de contador de ubicación (“.”), incrementamos el tamaño de esta sección para que tenga una dimensión igual al tamaño máximo de pila y al tamaño mínimo de pila “estimado”. Si la suma de las regiones .data, .bss, stack y heap es mayor que el tamaño de SRAM, el linker emitirá un error, como se muestra a continuación:

```
arm-none-eabi-g++ ...  ./src/ch10/main-ex5.o
../../../../arm-none-eabi/bin/ld: nucleo-f401RE.elf section `._user_heap_stack' will not f\
it in region `SRAM'
../../../../arm-none-eabi/bin/ld: region `SRAM' overflowed by 9520 bytes
collect2: error: ld returned 1 exit status
make: *** [nucleo-f401RE.elf] Error 1
```

- Es importante subrayar que se trata de una comprobación estática y no está relacionada con las actividades del firmware en tiempo de ejecución. Se necesitan diferentes estrategias para detectar un desbordamiento de pila y es realmente difícil tener una solución completa para un sistema integrado. Analizaremos este tema en un capítulo siguiente.

Differences With the Tool-Chain Script Files

- El linker script creado hasta ahora funciona bien para la mayoría de las aplicaciones STM32.
- Sin embargo, si va a codificar su firmware en C++, o simplemente va a utilizar bibliotecas creadas en C++, entonces esos linker scripts y secuencias de inicio no son suficientes.
- Para entender por qué, considere la siguiente aplicación C++:
- Centremos nuestra atención en la línea 14. Aquí estamos definiendo una instancia de la clase MiClase.
- La instancia se define como variable global. Pero declarar una instancia de una clase supone que se llama automáticamente al constructor de esa clase.
- Entonces, para ser claros, cuando llamamos al método increment() en la línea 17, el atributo de instancia i será igual a 101.

```
1  class MyClass {
2      int i;
3
4  public:
5      MyClass() {
6          i = 100;
7      }
8
9      void increment() {
10         i++;
11     }
12 };
13
14 MyClass instance;
15
16 int main() {
17     instance.increment();
18     for (;;);
19 }
```

Differences With the Tool-Chain Script Files

- ¿Pero quién se encarga de llamar al constructor de instancia?
- Cuando una instancia se crea localmente (es decir, desde una función global u otro método), le corresponde a esa instancia invocable realizar la inicialización de la clase.
- Pero cuando esto sucede a nivel global, depende de otras rutinas de inicialización.
- Por lo general, el compilador genera automáticamente una serie de punteros de función que contendrán rutinas de inicialización para todos los objetos asignados global y estáticamente.
- Estas matrices generalmente se llaman __init_array y __fini_array (que contiene la llamada a los destructores de objetos).

```
1  class MyClass {
2      int i;
3
4  public:
5      MyClass() {
6          i = 100;
7      }
8
9      void increment() {
10         i++;
11     }
12 };
13
14 MyClass instance;
15
16 int main() {
17     instance.increment();
18     for (;;);
19 }
```

Differences With the Tool-Chain Script Files

- Tanto los linker scripts como las rutinas de inicio proporcionadas por el complemento GNU ARM y ST en su HAL contienen todo el código necesario para manejar estas y otras actividades de inicialización.
- Explicarlos está fuera del alcance de este libro (esto también implica analizar en profundidad algunas actividades de libc realizadas en el inicio).
- Sin embargo, ahora que sabemos cómo dominar el contenido de un script de enlace, no debería ser demasiado difícil lidiar con él.

```
class MyClass {
    int i;

public:
    MyClass() {
        i = 100;
    }

    void increment() {
        i++;
    }
};

MyClass instance;

int main() {
    instance.increment();
    for (;;);
}
```

How to Use the CCM Memory

- Algunos microcontroladores de las familias STM32F3/4/7 proporcionan una memoria SRAM adicional denominada Core Coupled Memory (CCM).
- A diferencia de la SRAM normal, esta memoria está estrechamente acoplada al núcleo Cortex-M.
- Una ruta directa conecta tanto el D-Bus como el I-Bus a esta área de memoria (consulte la Figura 5), lo que permite la ejecución del estado de espera 0.
- Aunque es perfectamente posible almacenar datos en esta memoria, como tablas de consulta y vectores de inicialización, el mejor uso de esta área es almacenar rutinas críticas y computacionales intensivas, que pueden ejecutarse en tiempo real.

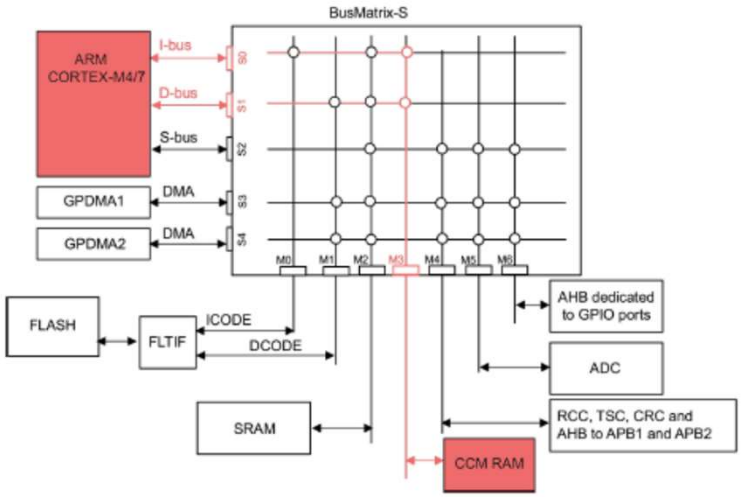


Figure 5: The direct connection between the Cortex-M core and the CCM SRAM

How to Use the CCM Memory

- Por esta razón, se dice que las MCU con memoria CCM implementan tecnología de refuerzo de rutina.
- ¿Por qué utilizar CCM para almacenar código en lugar de datos?
  - Es bastante común leer en la web que la memoria CCM se puede utilizar para almacenar datos críticos.
  - Esto garantiza un acceso rápido al mismo desde el núcleo. Si bien esto es cierto en teoría, no ofrece ventajas prácticas.
  - Todas las MCU STM32 con memoria CCM también proporcionan SRAM que se puede direccionar a la frecuencia máxima del reloj del sistema sin estados de espera.

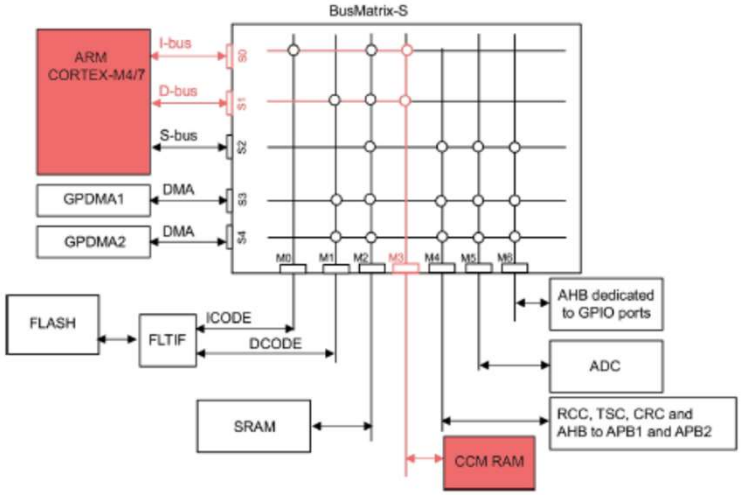


Figure 5: The direct connection between the Cortex-M core and the CCM SRAM

How to Use the CCM Memory

- Además, tanto la CPU como el DMA pueden acceder a la SRAM, mientras que al CCM solo se puede acceder a través del núcleo Cortex.
- En cambio, cuando el código se encuentra en la SRAM de CCM y los datos se almacenan en la SRAM normal, el núcleo Cortex está en la configuración Harvard óptima, porque permite el acceso a estados de espera 0 para el I-Bus (que accede a CCM) y el D- Bus (que accede en paralelo a la SRAM).
- Sin embargo, está claro que, si el rendimiento determinista no es importante para su aplicación y necesita almacenamiento SRAM adicional, entonces el CCM es una buena reserva para la memoria de datos.

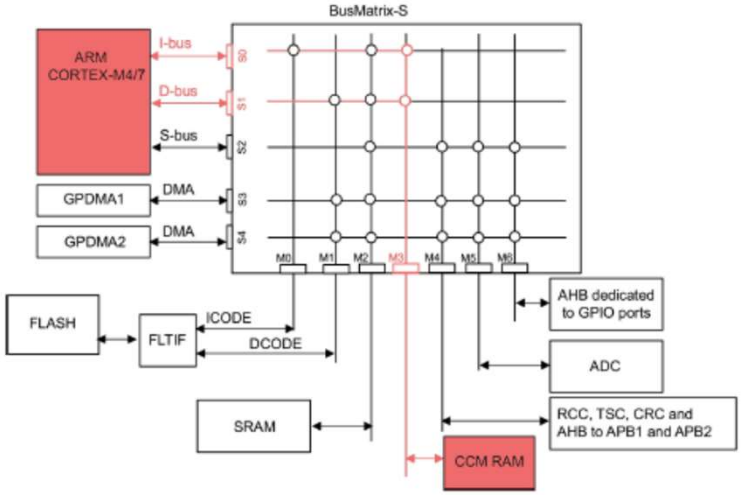


Figure 5: The direct connection between the Cortex-M core and the CCM SRAM

How to Use the CCM Memory

- En todas las MCU STM32 con esta memoria adicional, la SRAM del CCM se asigna a partir de la dirección 0x1000 0000.
- Una vez más, para usarlo necesitamos definir esta región de memoria dentro del linker script, de la siguiente manera:

```
/* memory layout for an STM32F334R8 */
MEMORY
{
    FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 64K
    SRAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 12K
    CCM  (xrw)  : ORIGIN = 0x10000000, LENGTH = 4K
}
```

- Obviamente, el atributo LENGTH debe reflejar el tamaño de la memoria CCM para la MCU STM32 específica. Una vez definida la región, tenemos que crear una sección específica dentro del Linker script:

```
.ccm : ALIGN(4) {
   *(.ccm .ccm*)
} >CCM
```

How to Use the CCM Memory

- Para reubicar una rutina específica dentro de la memoria CCM podemos usar la palabra clave GCC __attribute__, como se vio antes para la sección .isr_vector:

```
void __attribute__((section(".ccm"))) routine() {
    ...
}
```

- Si, en cambio, queremos almacenar datos dentro de la memoria CCM, también debemos inicializarlos como hemos visto para las regiones .bss y .data en la memoria SRAM normal. En este caso, necesitamos un Linker script más articulado:

```
/* Used by the startup to initialize data in CCM */
_siccm = LOADADDR(.ccm.data);

/* Initialized data section in CCM */
.ccm.data : ALIGN(4)
{
     _sccmd = .;
     *(.ccm.data .ccm.data*)
     . = ALIGN(4);
     _eccmd = .;
} >CCM AT>FLASH

/* Uninitialized data section in CCM */
.ccm.bss (NOLOAD) : ALIGN(4)
{
  _sccmb = .;

  *(ccm.bss ccm.bss*)

  . = ALIGN(4);
  _eccmb = .;
} >CCM
```

How to Use the CCM Memory

- Aquí definimos dos secciones: .ccm.data, que se usará para almacenar datos globales inicializados en CCM, y .ccm.bss que se usará para almacenar datos globales no inicializados.
- Como se hizo con la SRAM normal, será necesario llamar a las rutinas __initialize_data() y __initialize_bss() desde la rutina _start():

```
...
__initialize_data(&_sicom, &_scomd, &_ecomd);
__initialize_bss(&_scomb, &_ecomb);
...
```

- Luego, para colocar datos dentro del CCM, tenemos que indicarle al compilador que use la palabra clave del atributo:

```
uint8_t initdata[] __attribute__((section(".com.data"))) = {0x1, 0x2, 0x3, 0x4};
uint8_t uninitdata __attribute__((section(".com.bss")));
```

Relocating the vector table in CCM Memory

- La memoria CCM también se puede utilizar para almacenar rutinas ISR, reubicando toda la tabla de vectores dentro de la memoria CCM.
- Esto puede resultar especialmente útil para los ISR que deben procesarse en el menor tiempo posible.
- Sin embargo, reubicar la tabla de vectores requiere pasos adicionales, ya que la arquitectura Cortex-M está diseñada para que la tabla de vectores comience desde la dirección 0x0000 0004 (que corresponde a la dirección 0x0800 0004 de la memoria flash interna).
- Los pasos a seguir son estos:
  - definir la tabla de vectores para colocar en la RAM del CCM usando la palabra clave __attribute__((section(".isr_- vector_ccm"));
  - definir los handlers de excepciones para las excepciones e ISR de interés y colocarlos en la sección correspondiente usando la palabra clave __attribute__((section(".ccm"));

Relocating the vector table in CCM Memory

- definir una tabla de vectores mínima, compuesta por el puntero MSP y la dirección del handler de excepción Reset, para colocar en la memoria flash a partir de la dirección 0x0800 0000;
- reubicar la tabla de vectores de la excepción Reset copiando el contenido de la sección .ccm de la memoria flash a la SRAM.

Comencemos definiendo la tabla de vectores para colocar en CCM RAM. Aquí estamos definiendo un archivo llamado ccm_vector.c con el siguiente contenido:

Filename: src/ccm_vector.c

```
1  #include <stm32f3xx_hal.h>
2
3  #define GPIOA_ODR       ((uint32_t*)(GPIOA_BASE + 0x14))
4
5  extern const uint32_t _estack;
6
7  void SysTick_Handler(void);
8
9  uint32_t *ccm_vector_table[] __attribute__((section(".isr_vector_ccm"))) = {
10   (uint32_t *)&_estack,   // initial stack pointer
11   (uint32_t *) 0,         // Reset_Handler not relocatable
12   (uint32_t *) 0,
13   (uint32_t *) 0,
14   (uint32_t *) 0,
15   (uint32_t *) 0,
16   (uint32_t *) 0,
17   (uint32_t *) 0,
18   (uint32_t *) 0,

19    (uint32_t *) 0,
20    (uint32_t *) 0,
21    (uint32_t *) 0,
22    (uint32_t *) 0,
23    (uint32_t *) 0,
24    (uint32_t *) 0,
25    (uint32_t *) SysTick_Handler
26 };
27
28 void __attribute__((section(".ccm"))) SysTick_Handler(void) {
29   *GPIOA_ODR = *GPIOA_ODR ? 0x0 : 0x20; //Causes LD2 LED to blink
30 }
```

Relocating the vector table in CCM Memory

- El archivo contiene solo la tabla de vectores, que se coloca dentro de la sección .isr_vector_ccm, y el handler de la excepción SysTick, que se coloca dentro de la sección .ccm.
- A continuación, debemos organizar el Linker script de la siguiente manera:
- El Linker script no contiene nada diferente a lo visto hasta ahora.
- La sección .ccm está definida y le indicamos al linker que coloque en ella primero el contenido de la sección .isr_vector_ccm y luego el contenido de la sección .ccm, que en nuestro caso contiene solo la rutina SysTick_Handler.

Filename: src/ldscript6.ld

```
75      /* Used by the startup to load ISR in CCM from FLASH */
76        _slocm = LOADADDR(.ccm);
77
78      .ccm : ALIGN(4)
79      {
80         _sccm = .;
81         *(.isr_vector_ccm)
82         *(.ccm)
83         KEEP(*(.isr_vector_ccm .ccm))
84
85         . = ALIGN(4);
86         _eccm = .;
87      } >CCM AT>FLASH
88
89      /* Size of the .ccm section */
90      _ccmsize = _eccm - _sccm;
```

Relocating the vector table in CCM Memory

- También le indicamos al linker que almacene el contenido de la sección .ccm dentro de la memoria flash (usando la directiva CCM AT>FLASH), mientras que las direcciones VMA de la sección .ccm están vinculadas al rango de direcciones de memoria CCM (es decir, la dirección inicial es 0x1000 0000).
- Finalmente, necesitamos copiar manualmente el contenido de la sección .ccm de la memoria flash a la CCM y reubicar la tabla de vectores. Este trabajo lo realiza nuevamente la excepción Reset_Handler.

Filename: src/ldscript6.ld

```
75      /* Used by the startup to load ISR in CCM from FLASH */
76        _slocm = LOADADDR(.ccm);
77
78      .ccm : ALIGN(4)
79      {
80         _sccm = .;
81         *(.isr_vector_ccm)
82         *(.ccm)
83         KEEP(*(.isr_vector_ccm .ccm))
84
85         . = ALIGN(4);
86         _eccm = .;
87      } >CCM AT>FLASH
88
89      /* Size of the .ccm section */
90      _ccmsize = _eccm - _sccm;
```

Relocating the vector table in CCM Memory

- Las líneas [69:72] definen la tabla de vectores mínima utilizada cuando la CPU se resetea.
- Está compuesto simplemente por el puntero MSP y la dirección de la excepción Reset_Handler, que está representada por la rutina _start().
- Cuando la MCU se resetea, copiamos en la línea 77 el contenido de la sección .ccm de la memoria flash (la dirección base se almacena dentro de la variable _slccm) a la memoria CCM, y luego reubicamos toda la tabla de vectores asignando la posición en CCM. memoria de la matriz ccm_vector_table al registro VTOR en el Bloque de control del sistema (SCB) - línea 79.

Filename: src/main-ex6.c

```
68  /* Minimal vector table */
69  uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
70      (uint32_t *)&_estack,    // initial stack pointer
71      (uint32_t *)_start       // main as Reset_Handler
72  };
73
74  void __attribute__ ((noreturn,weak))
75  _start (void) {
76    /* Copy the .ccm section from the FLASH memory (_siccm) into CCM memory  */
77    memcpy(&_sccm, &_siccm, (size_t)&_ccmsize);
78
79    __DMB(); //This ensures that write to memory is completed
80
81    SCB->VTOR = (uint32_t)&_sccm;  /* Relocate vector table to 0x1000 0000 */
82    SYSCFG->RCR = 0xF;             /* Enable write protection for CCM memory */
83
84    __DSB(); //This ensures that following instructions use the new configuration
85
86    __initialize_data(&_sidata, &_sdata, &_edata);
87    __initialize_bss(&_sbss, &_ebss);
88    main();
89
90    for(;;);
91  }
92
93  int main() {
94    /* enable clock on GPIOA peripheral */
95    *RCC_APB1ENR |= 0x1 << 17;
96    *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1
97
98    SysTick_Config(4000000); //Underflows every 0.5s
99  }
100
101 void delay(uint32_t count) {
102     while(count--);
103 }
```

Relocating the vector table in CCM Memory

- A continuación, habilitamos la protección contra escritura en toda la memoria CCM para evitar escrituras no deseadas que puedan dañar el código.

Filename: src/main-ex6.c

```c
/* Minimal vector table */
uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
    (uint32_t *)&_estack,   // initial stack pointer
    (uint32_t *)_start        // main as Reset_Handler
};

void __attribute__ ((noreturn,weak))
_start (void) {
  /* Copy the .ccm section from the FLASH memory (_sloccm) into CCM memory  */
  memcpy(&_sccm, &_sloccm, (size_t)&_ccmsize);

  __DMB(); //This ensures that write to memory is completed

  SCB->VTOR = (uint32_t)&_sccm;  /* Relocate vector table to 0x1000 0000 */
  SYSCFG->RCR = 0xF;             /* Enable write protection for CCM memory */

  __DSB(); //This ensures that following instructions use the new configuration

  __initialize_data(&_sidata, &_sdata, &_edata);
  __initialize_bss(&_sbss, &_ebss);
  main();

  for(;;);
}

int main() {
  /* enable clock on GPIOA peripheral */
  *RCC_APB1ENR |= 0x1 << 17;
  *GPIOA_MODER |= 0x400; // Sets MODER[11:10] = 0x1

  SysTick_Config(4000000); //Underflows every 0.5s
}

void delay(uint32_t count) {
    while(count--);
}
```

Relocating the vector table in CCM Memory

- La RAM del CCM se subdivide en páginas de 1Kb. Cada bit del registro RCR del controlador de configuración del sistema (SYSCFG) se utiliza para configurar la protección contra escritura en páginas individuales (el bit 1 establece la protección de la primera página, el bit 2 establece la protección de la segunda página, etc.).
- Aquí, estamos protegiendo contra escritura toda la memoria CCM de una MCU STM32F334, que tiene una memoria CCM formada por cuatro páginas de 1 Kb.
- Es importante señalar que, si deshabilitamos la escritura de toda la memoria CCM, no podemos colocar variables globales o asignadas estáticamente en ella, de lo contrario ocurrirá una falla. Por otro lado, colocar tanto código como datos en la memoria CCM nos hace perder los beneficios que obtiene la memoria CCM, debido al acceso simultáneo a la misma memoria tanto por el bus D-Bus como por el I-Bus (mirando la Figura 5 se puede ver se puede ver que la memoria CCM está conectada a un solo puerto maestro del BusMatrix - el puerto M3 - por lo que el acceso desde D-Bus e I-Bus está regulado por el BusMatrix).

Relocating the vector table in CCM Memory

- La reubicación de la tabla de vectores no se limita a la memoria CCM. Como veremos en el siguiente capítulo, esta técnica también se utiliza cuando la MCU arranca desde fuentes diferentes a la memoria flash interna. En este caso, la tabla de vectores suele colocarse en SRAM y hay que reubicarla.
- La reubicación de la tabla de vectores es una característica que no está disponible en los microcontroladores Cortex-M0, mientras que sí está disponible en Cortex-M0+. Como veremos en un capítulo siguiente, existe un procedimiento que intenta abordar esta limitación.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- Además del núcleo Cortex-M0, todos los microcontroladores basados en Cortex-M pueden proporcionar opcionalmente una Unidad de Protección de Memoria (MPU).
- Y la buena noticia es que todas las MCU STM32 basadas en esos núcleos lo proporcionan.
- La MPU no debe confundirse con la Unidad de gestión de memoria (MMU), un componente de hardware avanzado disponible en microprocesadores de mayor rendimiento como Cortex-A, que se dedica principalmente a la traducción de direcciones de memoria virtual en físicas.
- La MPU se utiliza para proteger hasta ocho regiones de memoria, numeradas del 0 al 7.
- Estas, a su vez, pueden tener ocho subregiones, si la región principal tiene al menos 256 bytes.
- Todas las subregiones tienen el mismo tamaño y se pueden habilitar o deshabilitar según el número de subregión.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- La MPU se utiliza para hacer que un sistema integrado sea más robusto y seguro, y en algunos dominios de aplicaciones su uso es obligatorio (por ejemplo, en automoción y aeroespacial). La MPU se puede utilizar para:
  - Prohibir que las aplicaciones del usuario dañen los datos utilizados por tareas críticas (como el kernel del sistema operativo).
  - Definir la región de memoria SRAM como no ejecutable para evitar ataques de inyección de código.
  - Cambiar los atributos de acceso a la memoria.
- Si el núcleo de la CPU viola las definiciones de acceso de una región de memoria determinada (por ejemplo, al intentar ejecutar código desde una región no ejecutable), se genera la excepción HardFault (o la más específica Memory Fault, como veremos en el siguiente capítulo).

How to Use the MPU in Cortex-M0+/3/4/7 Based

- Las regiones de MPU pueden generar todo el espacio de direcciones de 4 GB y también pueden superponerse.
- Las características de la región están definidas por dos parámetros: el tipo de región y sus atributos. Hay tres tipos de memoria:
  - Memoria normal: permite que la CPU organice la carga y el almacenamiento de bytes, medias palabras y palabras de manera eficiente (el compilador no conoce los tipos de regiones de memoria). Para la región de memoria normal, la CPU no realiza necesariamente la carga/almacenamiento en el orden indicado en el programa. Las memorias SRAM y FLASH son dos ejemplos de memoria normal.
  - Memoria del dispositivo: dentro de la región del dispositivo, las cargas y almacenamientos se realizan estrictamente en orden. Esto es para garantizar que los registros estén configurados en el orden correcto; de lo contrario, el comportamiento del dispositivo se verá afectado.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- Memoria fuertemente ordenada: todo se hace siempre en el orden listado programáticamente, donde la CPU espera el final de la ejecución de la instrucción de carga/almacenamiento (acceso efectivo al bus) antes de ejecutar la siguiente instrucción en la secuencia del programa. Esto puede causar un impacto en el rendimiento.

Table 2: Memory region attributes

Region Attribute	Description
XN	Execute never
AP	Access permission (see Table 3)
TEX	Type Extension field (not available in Cortex-M0+
S	Shareable
C	Cacheable
B	Bufferable
SRD	Subregion disable/enable
SIZE	Size of the memory region

- Cada región de memoria tiene ocho atributos, que se muestran en la Tabla 2:
  - Execute never (XN): una región de memoria marcada con este atributo no permite la ejecución de código de programa.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- Access Permission (AP): define los permisos de acceso a la región de memoria. Los permisos se establecen tanto para código privilegiado (por ejemplo, el kernel RTOS) como para código sin privilegios (por ejemplo, un hilo individual). La Tabla 3 enumera todas las combinaciones posibles.
- TEX, C y B: estos campos se utilizan para definir las propiedades de la caché para la región y, hasta cierto punto, su capacidad para compartirla. Están codificados de acuerdo con la Tabla 4. Tenga en cuenta que en los núcleos Cortex-M0+ el campo TEX siempre es 0. Esto se debe a que los núcleos Cortex-M0+ admiten un nivel de política de caché.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- S: este campo configura una región de memoria compartible. El sistema de memoria proporciona sincronización de datos entre maestros de bus en un sistema con múltiples maestros de bus, por ejemplo, un procesador con un controlador DMA. La memoria fuertemente ordenada siempre se puede compartir. Si varios maestros de bus pueden acceder a una región de memoria no compartible, el software debe garantizar la coherencia de los datos entre los maestros de bus. Este campo no es compatible con la arquitectura ARMv6-M y, por lo tanto, siempre está configurado en 0 en los procesadores Cortex-M0+.
- SRD: define si una subregión en particular está habilitada o deshabilitada. Deshabilitar una subregión significa que en su lugar coincide otra región que se superpone al rango deshabilitado. Si ninguna otra región habilitada se superpone a la subregión deshabilitada, la MPU emite una falla.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- SIZE: especifica el tamaño de la región de memoria. El tamaño no puede ser arbitrario, pero puede asumir un valor de un conjunto conocido de tamaños de regiones (depende de la familia STM32 específica).

Table 3: Access permissions to a region

Privileged access	Unprivileged access	Description
No access	No access	All accesses to the region generate a permission fault
RW	No access	Access from a privileged software only
RW	RO	Writings by an unprivileged software generate a permission fault
RW	RW	Full access to the region
Unpredictable	Unpredictable	RESERVED
RO	No access	Read by a privileged software only
RO	RO	Read only, by privileged or unprivileged software

- Los microcontroladores STM32F7 proporcionan un caché L1 integrado, como veremos en el siguiente capítulo. Para estas MCU están disponibles los siguientes atributos de memoria adicionales:
  - Almacenable en caché/no almacenable en caché: significa que la región dedicada se puede almacenar en caché o no.

How to Use the MPU in Cortex-M0+/3/4/7 Based

- Escritura directa sin asignación de escritura: en caso de aciertos, escribe en el caché y en la memoria principal; en caso de fallos, actualiza el bloque en la memoria principal sin llevar ese bloque al caché.
- Reescritura sin asignación de escritura: en los accesos, escribe en el bit sucio de configuración de caché para el bloque, la memoria principal no se actualiza. En caso de error, actualiza el bloque en la memoria principal y no lo lleva al caché.
- Reescritura con asignación de escritura y lectura: en los accesos, escribe en la caché configurando el bit sucio para el bloque, la memoria principal no se actualiza. En caso de error, actualiza el bloque en la memoria principal y lo lleva al caché.

How to Use the MPU in Cortex-M0+/3/4/7 Based

Table 4: Region cache properties and shareability

TEX	C	B	Memory Type	Description	Shareable
000	0	0	Strongly Ordered	Strongly Ordered	Yes
000	0	1	Device	Shared Device	Yes
000	1	0	Normal	Write through, no write allocate	S bit dependent
000	1	1	Normal	Write-back, no write allocate	S bit dependent
001	0	0	Normal	Non-cacheable	S bit dependent
001	0	1	Reserved	Reserved	Reserved
001	1	0	Undefined	Undefined	Undefined
001	1	1	Normal	Write-back, write and read allocate	S bit dependent
010	0	0	Device	Non-shareable device	No
010	0	1	RESERVED	RESERVED	RESERVED

- La Tabla 5 enumera los tipos y atributos de las memorias que se encuentran en un microcontrolador STM32. Como veremos en un capítulo siguiente, el caché L1 integrado en las MCU STM32F7 también permite definir como regiones almacenables en caché memorias externas accesibles a través del controlador FMC. Esta es una gran mejora de rendimiento que ofrece esta familia de MCU.

Table 5: Memory attributes for the typical STM32 memories

Memory	Memory type	Memory attributes
ROM, flash (program memories)	Normal memory	Non-shareable, write-through C=1, B=0, TEX=0, S=0
Internal SRAM	Normal memory	Shareable, write-through C=1, B=0, TEX=0, S=1/S=0
External RAM (through FMC)	Normal memory	Shareable, write-back C=1, B=1, TEX=0, S=1/S=0
Peripherals	Device	Shareable devices C=0, B=1, TEX=0, S=1/S=0

How to Use the MPU in Cortex-M0+/3/4/7 Based

- La Tabla 6 muestra una comparación de las características de MPU en núcleos Cortex-M0+/3/4/7. La omisión de MPU es una característica ofrecida por la MPU para omitir los permisos de acceso a una región cuando el procesador ejecuta excepciones NMI o HardFault. Por ejemplo, la MPU podría usarse como mecanismo para detectar el límite de la pila asignando un pequeño espacio SRAM en la parte inferior de la pila como no accesible. Cuando se alcanza el límite de la pila, el controlador HardFault puede omitir la restricción de MPU y utilizar el espacio SRAM reservado para el manejo de fallas.

Table 6: Comparison of MPU features between the various Cortex-M cores

	Cortex®-M0+	Cortex®-M3/M4	Cortex®-M7
Number of regions	8	8	8
Region address	Yes	Yes	Yes
Region size	256 bytes to 4GB	32 bytes to 4GB	32 bytes to 4 GB
Region memory attributes	S, C, B, XN	TEX, S, C, B, XN	TEX, S, C, B, XN
Region access permission	Yes	Yes	Yes
Subregion disable	Yes	Yes	Yes
MPU bypass for	Yes	Yes	Yes
NMI/HardFault Fault exception	HardFault only	HardFault/MemManage	HardFault/MemManage

Programming the MPU With the CubeHAL

- CubeHAL proporciona toda la capa de abstracción necesaria para programar la MPU. La función

```
void HAL_MPU_ConfigRegion(MPU_Region_InitTypeDef *MPU_Init);
```

- permite configurar una región de memoria. Todas las configuraciones de región se especifican con una instancia de la estructura MPU_-Region_InitTypeDef, que se define de la siguiente manera:

```
typedef struct {
  uint8_t  Enable;          /* Specifies the status of the region. */
  uint8_t  Number;          /* Specifies the number of the region to protect. */
  uint32_t BaseAddress;     /* Specifies the base address of the region to protect. */
  uint8_t  Size;            /* Specifies the size of the region to protect.  */
  uint8_t  SubRegionDisable; /* Specifies the number of the subregion protection
                                to disable. */
  uint8_t  TypeExtField;    /* Specifies the TEX field level. */
  uint8_t  AccessPermission; /* Specifies the region access permission type. */
  uint8_t  DisableExec;     /* Specifies the instruction access status. */
  uint8_t  IsShareable;     /* Specifies the shareability status of the
                               protected region. */
  uint8_t  IsCacheable;     /* Specifies the cacheable status of the region protected. */
  uint8_t  IsBufferable     /* Specifies the bufferable status of the protected region. */
} MPU_Region_InitTypeDef;
```

Programming the MPU With the CubeHAL

- Analicemos los campos más relevantes de esta estructura.
  - Enable: especifica el estado de la región y puede asumir los valores MPU_REGION_ENABLE y MPU_REGION_DISABLE.
  - Number: es el ID de la región y puede generar desde 0 hasta 7.
  - BaseAddress: corresponde a la dirección base de la región. En Cortex-M0+, esta dirección debe estar alineada con palabras.
  - Size: especifica el tamaño de la región y corresponde a todas las potencias de dos desde 2^5 hasta 2^{32}. CubeHAL define un conjunto de 27 macros, que van desde MPU_REGION_SIZE_32B hasta MPU_-REGION_SIZE_4GB. Eche un vistazo al archivo stm32XXxx_hal_cortex.h para obtener la lista completa.
  - AccessPermission: especifica los atributos de permiso de la región y puede asumir los valores enumerados en la Tabla 7.

Programming the MPU With the CubeHAL

- DisableExec: especifica si es posible ejecutar código dentro de la región. Puede asumir los valores MPU_INSTRUCTION_ACCESS_ENABLE y MPU_INSTRUCTION_ACCESS_DISABLE.
- IsShareable: especifica si la región tiene el atributo compartible y puede asumir los valores MPU_ACCESS_SHAREABLE y MPU_ACCESS_NOT_SHAREABLE.
- IsCacheable: especifica si la región tiene el atributo cacheable y puede asumir los valores MPU_ACCESS_CACHEABLE y MPU_ACCESS_NOT_CACHEABLE.
- IsBufferable: especifica si la región tiene el atributo bufferable y puede asumir los valores MPU_ACCESS_BUFFERABLE y MPU_ACCESS_NOT_BUFFERABLE.

Programming the MPU With the CubeHAL

Table 7: CuneHAL macros to define access permissions to a region

Access permission	Description
MPU_REGION_NO_ACCESS	All accesses to the region generate a permission fault
MPU_REGION_PRIV_RW	Access from a privileged software only
MPU_REGION_PRIV_RW_URO	Writings by an unprivileged software generate a permission fault
MPU_REGION_FULL_ACCESS	Full access to the region
MPU_REGION_PRIV_RO	Read by a privileged software only
MPU_REGION_PRIV_RO_URO	Read only, by privileged or unprivileged software

- La MPU debe desactivarse antes de configurar cualquier región de memoria (o antes de cambiar sus atributos).
- Para realizar esta operación el HAL proporciona la función:

```
void HAL_MPU_Disable(void);
```

- mientras que para habilitar la MPU usamos la función:

```
void HAL_MPU_Enable(uint32_t MPU_Control);
```

- El parámetro MPU_Control especifica el modo de control de la MPU durante HardFault, NMI, FAULTMASK y el acceso privilegiado a la memoria predeterminada.

Programming the MPU With the CubeHAL

- Puede asumir un valor de los enumerados en la Tabla 8. Es importante tener en cuenta que la excepción MemFault se habilita automáticamente una vez que se habilita la MPU.

Table 8: CubeHAL macros to define MPU control during *HardFault*, NMI and FAULTMASK

Access permission	Description
MPU_HFNMI_PRIVDEF_NONE	The default memory map is used for privileged accesses, and it assumes the role of a background region (also called "region -1", where "-1" is the region ID). The access to the whole 4GB is so prohibited by unprivileged code, except in those regions that explicitly allow it.
MPU_HARDFAULT_NMI	The MPU is disabled when *HardFault* and NMI exceptions raise.
MPU_PRIVILEGED_DEFAULT	The background region is disabled and any access not covered by any enabled region will cause a fault.
MPU_HFNMI_PRIVDEF	The MPU is enabled when *HardFault* and NMI exceptions raise.

Programming the MPU With the CubeHAL

- El fragmento de código anterior muestra cómo definir una región de memoria en la memoria SRAM y evitar el acceso a ella en modo de escritura, tanto desde código privilegiado como sin privilegios. La región comienza desde la dirección 0x2000 0A00 y tiene una duración de 32 bytes. Se define un puntero al inicio de esa región (línea x) y se modifica el contenido de la primera palabra (línea x). La MPU está habilitada y los atributos de región impiden que el código modifique su contenido. El if en la línea x no coincidirá, porque la primera palabra de región contiene efectivamente el valor 0xDDEEFF00. Sin embargo, la instrucción en la línea x generará un error de MemManage debido al atributo de solo lectura de la región.

```
1   MPU_Region_InitTypeDef MPU_InitStruct;
2
3   /* Disable MPU */
4   HAL_MPU_Disable();
5
6   /* Configure RAM region as Region N°0, 8kB of size and R/W region */
7   MPU_InitStruct.Enable = MPU_REGION_ENABLE;
8   MPU_InitStruct.BaseAddress = 0x20000A00;
9   MPU_InitStruct.Size = MPU_REGION_SIZE_32B;
10  MPU_InitStruct.AccessPermission = MPU_REGION_PRIV_RO_URO;
11  MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
12  MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
13  MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
14  MPU_InitStruct.Number = MPU_REGION_NUMBER0;
15  MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
16  MPU_InitStruct.SubRegionDisable = 0x00;
17  MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
18  HAL_MPU_ConfigRegion(&MPU_InitStruct);
19
20  /* Defines a pointer to the first word of protected region */
21  volatile uint32_t *p = (uint32_t*)0x20000A00;
22  *p = 0xDDEEFF00;
23
24  /* Re-enable the MPU and enable the background region */
25  HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
26
27  if(*p != 0xDDEEFF00)
28    asm("BKPT #0");
29
30  *p = 0xAABBCCDD; //This will generate a MemManage fault
```

Referencias

- Mastering STM32 - A step-by-step guide to the most complete ARM Cortex-M platform, using a free and powerful development environment based on Eclipse and GCC - Carmine Noviello (Author)
- Programming with STM32: Getting Started with the Nucleo Board and C/C++ 1st Edición - Donal Norris (Author)
- Nucleo Boards Programming with the STM32CubeIDE, Hands-on in more than 50 projects - Dogan Ibrahim (Author)
- STM32 Arm Programming for Embedded Systems, Using C Language with STM32 Nucleo - Muhammad Ali Mazidi (Author), Shujen Chen (Author), Eshragh Ghaemi (Author)

Manos a la obra con el . . .

. . . Proyecto Intermedio

. . . un enfoque centrado en la **práctica propia de la carrera** más que en el desarrollo teórico disciplinar, con eje en la **participación de las y los estudiantes**


Las y los estudiantes preguntarán:
¿en qué lío nos metimos?

¡Muchas gracias!

¿Preguntas?

. . .

Consultas a: jcruz@fi.uba.ar