Taller de Sistemas Embebidos
STM32 MCU – Booting Process

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

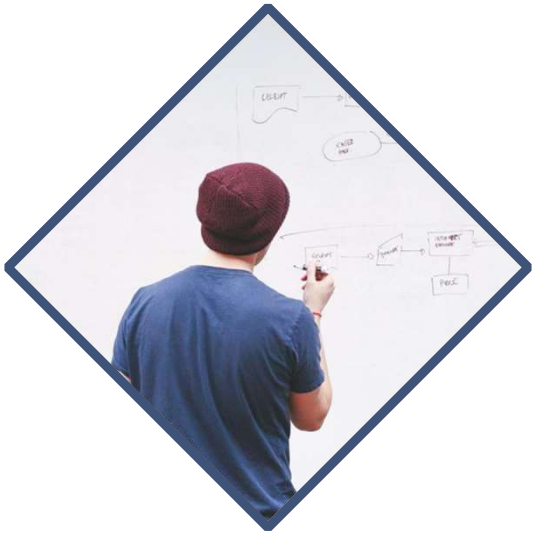
Estructura Curricular de la Carrera

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 19: Booting Process
    - En el Capítulo 15 hemos visto que el handler de la excepción Reset corresponde a la primera rutina que se ejecutará cuando se inicia la CPU. El modelo de layout de memoria fijo de los procesadores basados en Cortex-M establece que la dirección en memoria del handler de excepciones Reset se coloca justo después del Main Stack Pointer (MSP), es decir en la dirección 0x0000 0004.

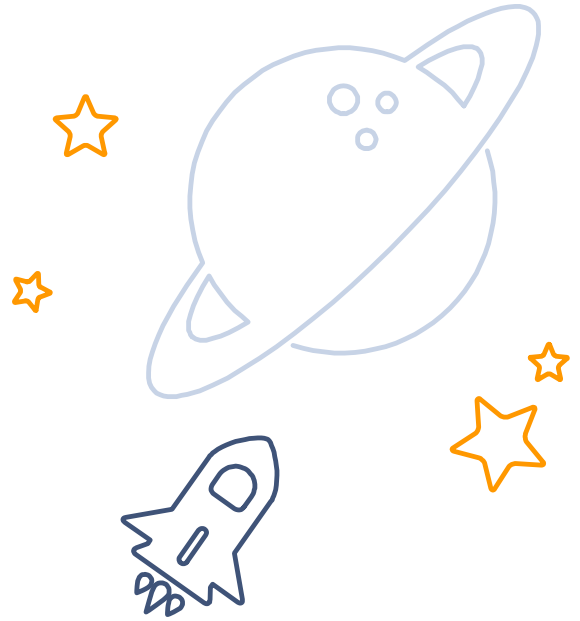
Conceptos básicos

- Esta ubicación de memoria generalmente corresponde al comienzo de memoria flash. Sin embargo, los proveedores de silicio pueden evitar esta limitación "asignando alias" a otras memorias con la dirección 0x0000 0000 con una operación llamada reasignación física (remapping). Esta operación se realiza en el hardware después de algunos ciclos de reloj y es diferente de la reubicación de la tabla de vectores que se ve en el Capítulo 15, que se realiza mediante el mismo código que se ejecuta en la MCU.
- Además, la plataforma STM32 proporciona un cargador de arranque preprogramado de fábrica, que se puede utilizar para cargar el firmware dentro de la memoria flash desde varias fuentes.

Conceptos básicos

- Dependiendo de la familia STM32 y el tipo de venta utilizado, una MCU STM32 puede cargar el código utilizando periféricos de comunicación USART, USB, CAN, I²C y SPI. El gestor de arranque (bootloader) se selecciona gracias a pines de arranque específicos.
- Este capítulo completa el Capítulo 15 mostrando el proceso de arranque (booting) realizado por los microcontroladores STM32 después de un reset del sistema. Proporciona una descripción detallada de los pasos involucrados durante el arranque (bootstrap) y muestra brevemente cómo usar el gestor de arranque preprogramado de fábrica en todas las MCU STM32. Finalmente, también se muestra un gestor de arranque personalizado, que permite actualizar el firmware “on-board” utilizando la interfaz USART y procedimiento de carga personalizado.

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 19: Booting Process

The Cortex-M Unified Memory Layout and the Booting Process

- A diferencia de las arquitecturas de microprocesadores más avanzadas, como ARM Cortex-A, los microcontroladores Cortex-M no proporcionan una Memory Management Unit (MMU), que permite asignar alias a direcciones lógicas con direcciones físicas reales.
  - Esto significa que, desde el punto de vista del Cortex-M core, el mapa de memoria es fijo y estandarizado para todas las implementaciones.
- En los microcontroladores basados en Cortex-M, el área de código comienza desde la dirección 0x0000 0000 (a la que se accede a través de los buses I-Bus/D-Bus en Cortex-M3/4/7 y a través del S-Bus en Cortex-M0/0+) mientras que el área de datos (SRAM) comienza desde la dirección 0x2000 0000 (a la que se accede a través del S-Bus).
  - Las CPU´s Cortex-M siempre obtienen la tabla de vectores del I-Bus, lo que implica que sólo arrancan (boot) desde el área de código (que normalmente corresponde a la memoria flash).

The Cortex-M Unified Memory Layout and the Booting Process

- Los microcontroladores STM32 implementan un mecanismo especial, llamado reasignación física (remap), para arrancar (boot) desde otras memorias además de flash, que consiste en muestrear dos pines MCU dedicados, llamados BOOT0 y BOOT1.
  - El estado eléctrico de estos pines establece la dirección de inicio de arranque (boot starting address) y, por tanto, la memoria fuente.

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

Table 1: The boot modes available in an STM32F401RE MCU

- La Tabla 1 muestra los modos de arranque disponibles en una MCU STM32F401RE y se extrae del manual de referencia correspondiente. La 'x' dentro de la columna BOOT1 significa que, cuando el pin BOOT0 está conectado a tierra, el estado lógico del pin BOOT1 puede ser arbitrario.

The Cortex-M Unified Memory Layout and the Booting Process

- La primera fila corresponde al modo de arranque más común: la MCU asignará un alias a la memoria flash con la dirección 0x0000 0000.
  - Los otros dos modos de arranque corresponden al arranque desde la SRAM interna y la Memoria del sistema, una memoria ROM que contiene un gestor de arranque especial en todos MCU STM32 y que estudiaremos más adelante.
- El estado de los pines BOOT se almacena en el cuarto flanco ascendente de SYSCLK después de un reinicio.
  - Depende del usuario configurar los pines de ARRANQUE después de un reinicio para seleccionar el modo de inicio requerido.
- Los pines BOOT también se vuelven a muestrear al salir del modo de espera de bajo consumo.
  - En consecuencia, deben mantenerse en la configuración del modo de inicio deseado al entrar en modo de espera.

The Cortex-M Unified Memory Layout and the Booting Process

- Una vez transcurrido este tiempo de inicio, la CPU recupera el puntero de pila principal (MSP) de la dirección 0x0000 0000 y, por lo tanto, inicia la ejecución del código desde la memoria de inicio a partir de la dirección 0x0000 0004.
  - La memoria seleccionada (flash, SRAM o ROM) siempre es accesible con su espacio de direcciones original.
- Si configuramos la MCU para que arranque desde la memoria SRAM, que es una memoria volátil, tenemos que cargar el código del programa dentro de esta memoria y asegurarnos de que exista una tabla de vectores válida (hecha de al menos un puntero a la pila base y un puntero a la excepción de reinicio) está configurada correctamente en la dirección 0x0000 0000.
  - Esto requiere que utilicemos una herramienta de depuración, que precarga todo el código necesario dentro de la SRAM antes de comenzar la ejecución. Además, también se necesita un linker script personalizado. Veremos un ejemplo más adelante.

Software Physical Remap

- Una vez que se inicia la MCU, es decir, se ejecuta la excepción de reset, aún es posible reasignar la memoria accesible a través del área de código (es decir, a través de las líneas I-Bus y D-Bus) programando algunos bits de SYSCFG memory mapped register (SYSCFG->MEMRMP en la biblioteca CMSIS).
- Dependiendo de la MCU STM32 específica, se pueden reasignar las siguientes memorias:
  - Internal flash memory
  - System Memory
  - Internal SRAM
  - FMC NVM bank 1
  - FMC SDRAM bank 1

Software Physical Remap

- Las dos últimas memorias están disponibles sólo en aquellas MCU que proporcionan el Flexible Memory Controller (FMC), un periférico que permite interconectar memorias NVM y SDRAM externas.
  - Según la Tabla 1, no se permite el arranque directo desde memorias NOR externas ni desde memorias SDRAM.
    - Estas memorias solo se pueden asignar en la dirección 0x0000 0000 mediante la reasignación física del software después de que la MCU ya esté iniciada con un firmware mínimo cargado desde la memoria flash interna.
- Una vez que una memoria externa ha sido reasignada físicamente en la dirección 0x0000 0000, la CPU puede acceder a ella a través de las líneas I-Bus y D-Bus, en lugar del abarrotado S-Bus, lo que aumenta el rendimiento general.
  - Esto es especialmente importante para MCU basadas en Cortex-M7, donde esas líneas están estrechamente acopladas con un caché L1 dedicado.

Software Physical Remap

- Cuando se inicia la CPU, el contenido del registro SYSCFG->MEMRMP se vincula a los valores de los pines de BOOT: esto significa que la reasignación física se realiza automáticamente desde la MCU al muestrear los pines de BOOT.
  - Antes de cambiar el contenido de este registro, para realizar una reasignación, es importante tener en la memoria de destino una tabla de vectores de trabajo.

Vector Table Relocation

- En el Capítulo 15 hemos visto cómo reubicar la tabla de vectores en la memoria CCM para poder aprovechar esta memoria acoplada al núcleo.
  - Cuando realizamos una reasignación física, ya sea configurando los pines BOOT o configurando el registro SYSCFG->MEMRMP en consecuencia, no hay necesidad de realizar la reubicación de la tabla de vectores ya que la MCU automáticamente asigna un alias a la dirección inicial de la memoria seleccionada a 0x0000 0000.
  - Sin embargo, a veces, queremos mover la tabla de vectores a otras ubicaciones de memoria que no corresponden a su origen.
    - Por ejemplo, es posible que queramos almacenar dos imágenes de firmware independientes dentro de la memoria flash (ver Figura 1) y seleccionar una de ellas según una condición inicial determinada.

Vector Table Relocation

- Este es el caso de los gestores de arranque, programas especiales de “sistema” que realizan importantes tareas de configuración como la actualización del firmware principal, como veremos más adelante en este capítulo.

- El Vector Table Offset Register (VTOR) es un registro en el System Control Block (SCB) (SCB->VTOR en la biblioteca CMSIS) que permite configurar la dirección base de la tabla de vectores.
  - Una vez que se establece el contenido de este registro, la CPU tratará las direcciones a partir de la nueva ubicación base como punteros para interrumpir las rutinas de servicio.

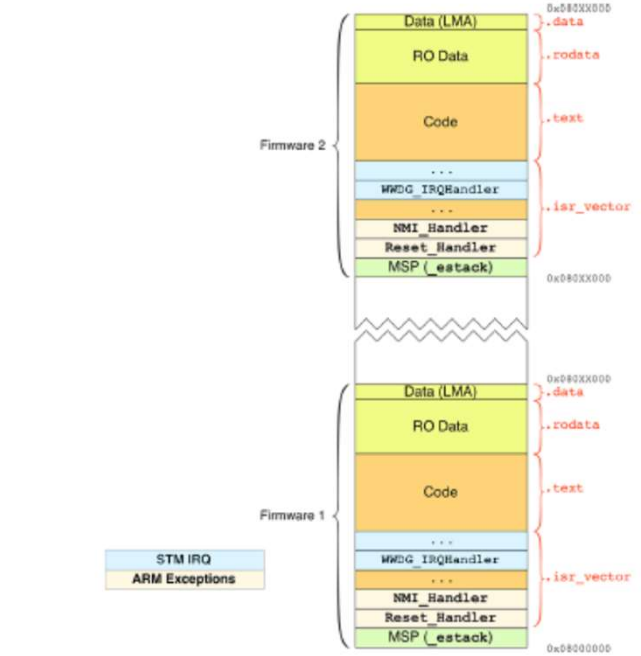


Figure 1: Two independent firmware images may be stored inside the flash memory

Vector Table Relocation

Figure 2: The structure of the VTOR register

- Al modificar el contenido del registro VTOR, es importante considerar que:
  - El registro VTOR no está disponible en la MCU basada en Cortex-M0 y, por lo tanto, no es posible reubicar la tabla de vectores sin utilizar la reasignación física (existe una forma de evitar esta limitación, como veremos más adelante).
  - En las MCU STM32F1, que se basan en la revisión del núcleo Cortex-M3 r1p0, los bits [31:30] del registro VTOR están reservados (consulte la Figura 2) y, por lo tanto, es posible reubicar la tabla de vectores solo en la memoria de código. (0x0000 0000) y en SRAM (0x2000 0000).

Vector Table Relocation

Figure 2: The structure of the VTOR register

- La especificación ARM sugiere utilizar una instrucción dmb (Data Memory Barrier) antes de actualizar el contenido del registro VTOR y una instrucción dsb (Data Synchronization Barrier) después de la actualización. Consulte el ejemplo 6 en el Capítulo 15 para obtener un ejemplo completo.
- Antes de cambiar el contenido del registro VTOR, asegúrese de que ya haya una tabla de vectores mínimos para su aplicación en la nueva ubicación.
- Si la aplicación utiliza interrupciones de periféricos, suspenda todas las interrupciones antes de iniciar el procedimiento de reubicación.

Running the Firmware From SRAM Using the GNU ARM Eclipse Toolchain

- A veces, puede resultar útil cargar el firmware binario dentro de la SRAM y arrancar desde allí.
  - Esto requiere un soporte especial del depurador y los siguientes pasos:
    - 1. Los pines de BOOT (o el bit correspondiente en la región de bytes de opción) deben configurarse para que la MCU arranque desde SRAM (ambos pines conectados a VDD en la mayoría de las MCU STM32).
    - 2. El linker script debe modificarse para que la región FLASH se asigne a la dirección inicial 0x0000 0000 (o a la dirección 0x2000 0000, que corresponde a la misma memoria si se selecciona SRAM como origen de arranque).
    - 3. Se debe indicar adecuadamente a OpenOCD que establezca el valor inicial del contador del programa en el origen de la dirección SRAM, más 4 bytes.

Running the Firmware From SRAM Using the GNU ARM Eclipse Toolchain

- El primer paso se puede lograr fácilmente en placas Nucleo conectando el pin BOOT0 (que corresponde al pin 7 en el conector morfo CN7) y el pin BOOT1 (que es el pin PB2 en casi todas las MCU STM32 con encapsulado LQFP-48, y que corresponde al pin 22 en el conector morfo CN10) a VDD, como se muestra en la Figura 3.
- El segundo paso generalmente puede limitarse a modificar el origen de la memoria FLASH dentro del linker script (el archivo mem.ld en la GNU ARM Eclipse tool-chain), estableciendo su origen en 0x0000 0000 (o la dirección 0x2000 0000 que también corresponde a la memoria SRAM). Si este procedimiento le parece nuevo, debe estudiar mejor el Capítulo 15.

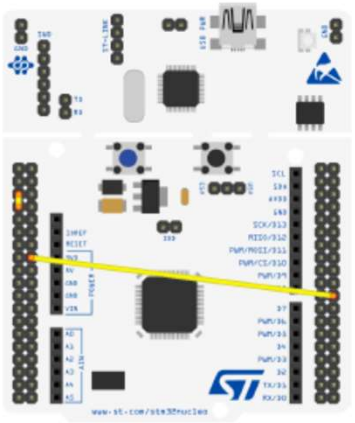


Figure 3: How to tie BOOT0 and BOOT1 pins to VDD in a Nucleo board so that MCU boots from SRAM

Running the Firmware From SRAM Using the GNU ARM Eclipse Toolchain

- Finalmente, necesitamos indicarle a OpenOCD que establezca el contador de programa (PC) en la dirección base de la memoria SRAM. Esto se puede lograr simplemente modificando la configuración de depuración de nuestro proyecto, ingresando a la sección Inicio y luego marcando la entrada Depurar desde RAM y desmarcando el reinicio de Preejecución/Reinicio. Estos ajustes también harán que el firmware se vuelva a cargar en SRAM cada vez que reseteemos el MCU desde el IDE (obviamente, si reseteamos la placa usando el botón hardware dedicado del Nucleo, el código se pierde o, al menos, se pierde). puede estar dañado).
- NOTA: Antes de solicitar soporte al autor, debido a que es posible que este procedimiento no funcione en su caso, tenga en cuenta que es posible que este procedimiento no funcione para aquellos que tienen placas Nucleo basadas en MCU STM32 con poca memoria SRAM. Esto porque podría suceder que el área de código caiga en el área de la pila. Básicamente, este procedimiento funciona para programas realmente pequeños y limitados.

Integrated Bootloader

- En la electrónica digital moderna es casi imposible distribuir dispositivos electrónicos sin publicar sucesivas actualizaciones del firmware.
  - Y esto es especialmente cierto para placas complejas con muchos circuitos integrados y periféricos.
  - Tarde o temprano, todos los desarrolladores integrados necesitarán una forma de distribuir una actualización de firmware y, lo más importante, necesitarán una forma de permitir a los clientes cargarla en la MCU sin un depurador dedicado (y a veces costoso).
  - Además, a menudo el puerto de depuración SWD no se agrega a la PCB final para una elección de diseño.

Integrated Bootloader

- Un gestor de arranque (bootloader) es una pieza de software, que generalmente se ejecuta primero cuando arranca la MCU, que tiene la capacidad de actualizar el firmware dentro de la memoria flash interna.
  - Esta operación también se conoce como In-Application Programming (IAP), que se diferencia de la programación de MCU mediante un depurador externo y dedicado: esta otra forma de programar MCU también se conoce como In-System Programming (ISP).
- Los cargadores de arranque generalmente están diseñados para aceptar comandos a través de un periférico de comunicación (USART, USB, Ethernet, etc.), que se utiliza para intercambiar el firmware binario con la MCU.
  - Generalmente también se necesita un programa dedicado, diseñado para ejecutarse en una PC externa.

Integrated Bootloader

- Todos los MCU STM32 vienen de fábrica con un gestor de arranque preprogramado en una memoria ROM, llamada System memory, que está asignada dentro del rango de direcciones 0x1FFF 0000 - 0x1FFF 77FF en la mayoría de los microcontroladores STM32.
- Dependiendo de la familia de MCU y el paquete utilizado, este gestor de arranque puede interactuar con el mundo exterior mediante:
  - USART
  - USB (DFU)
  - CAN bus
  - I2C
  - SPI

Integrated Bootloader

- Para cada uno de estos periféricos de comunicación, ST ha definido un protocolo estandarizado que permite:
  - Recuperar la versión del gestor de arranque y los comandos compatibles.
  - Obtener la identificación del chip.
  - Leer una cantidad de bytes de memoria a partir de una dirección especificada por la aplicación host.
  - Escribir una cantidad de bytes en la memoria RAM o flash a partir de una dirección especificada por la aplicación host.
  - Borrar una o más páginas/sectores de la memoria flash.
  - Saltar al código de la aplicación de usuario ubicado en la memoria flash interna o en SRAM.

Integrated Bootloader

- Activar/desactivar la protección de lectura/escritura para algunas páginas/sectores.

- Para cada protocolo de comunicación, ST proporciona una nota de aplicación dedicada llamada "protocolo PPP utilizado en el gestor de arranque STM32", donde PPP es el tipo de periférico. Por ejemplo, el AN3155 trata sobre el protocolo USART.
- Además del periférico de comunicación utilizado, el gestor de arranque utiliza otros recursos de hardware:
  - El oscilador HSI, que se selecciona como fuente de reloj.
  - El temporizador SysTick (no para todos los periféricos de comunicación).
  - Aproximadamente 2K de memoria SRAM.
  - El periférico IWDG (el preescaler está configurado en su valor máximo y el IWDG se actualiza periódicamente para evitar el reinicio en caso de que el usuario haya habilitado previamente la opción de hardware IWDG).

Integrated Bootloader

- Además, existen algunas limitaciones en cuanto a la gestión de la memoria a través del gestor de arranque:
  - Algunos microcontroladores STM32 no admiten la operación de borrado masivo.
    - Para realizar un borrado masivo usando el gestor de arranque, hay dos opciones disponibles: borrar todos los sectores uno por uno usando el comando Erase o configurar el nivel de protección de lectura flash en el Nivel 1 y luego volver a configurarlo en el Nivel 0.
    - El firmware del gestor de arranque en la serie STM32L1/L0 permite manipular EEPROM además de las memorias estándar (flash interno y SRAM, bytes de opción y memoria del sistema).
      - La dirección inicial y el tamaño de este tipo de memoria dependen del número de pieza específico.

Integrated Bootloader

- La EEPROM se puede leer y escribir, pero no se puede borrar usando el comando Erase.
- Al escribir en una ubicación EEPROM, el firmware del gestor de arranque gestiona la operación de borrado de esta ubicación antes de cualquier escritura.
- Una escritura en la EEPROM debe estar alineada con palabras (la dirección a escribir debe ser múltiplo de 4) y la cantidad de datos también debe ser múltiplo de 4.
- Para borrar una ubicación de EEPROM, puede escribir ceros en esta ubicación.

Integrated Bootloader

- El firmware del cargador de arranque en la serie STM32F2/F4/F7/L4 admite memoria OTP además de las memorias estándar (Flash interna, SRAM interna, bytes de opción y memoria del sistema).
  - La dirección inicial y el tamaño de esta área dependen del número de pieza específico.
  - Consulte el manual de referencia del producto para obtener más información.
  - La memoria OTP se puede leer y escribir, pero no se puede borrar con el comando Borrar.
  - Al escribir en una ubicación de memoria OTP, asegúrese de que el bit de protección relativo no se restablezca.

Integrated Bootloader

- Para las series STM32F2/F4/F7, el formato de operación de escritura flash interna depende del rango de tensión.
  - De forma predeterminada, las operaciones de escritura están permitidas en formato de un byte (no se permiten operaciones de media palabra, palabra y doble palabra).
  - Para aumentar la velocidad de las operaciones de escritura, el usuario debe aplicar el rango de tensión adecuado que permita operaciones de escritura por media palabra, palabra o doble palabra y actualizar esta configuración sobre la marcha utilizando el software bootloader.
  - Algunas ubicaciones virtuales están reservadas para esta operación.
  - Para obtener más información sobre esto, consulte el AN2606 de ST.

Integrated Bootloader

- Para interconectar el gestor de arranque integrado utilizando el protocolo USART, ST proporciona una herramienta conveniente, llamada STM32-FLASHER, que es una herramienta basada en Windows capaz de programar MCU STM32 utilizando el gestor de arranque USART.
  - Esto le permite programar su placa utilizando el gestor de arranque integrado y sin la necesidad de una aplicación de PC personalizada.
- Si, en cambio, su PCB final proporciona un puerto de dispositivo USB conectado a la MCU a través de sus pines dedicados, puede conectar el cargador de arranque de la MCU utilizando el protocolo estándar de actualización de firmware del dispositivo USB (Device Firmware Upgrade: DFU), un mecanismo independiente del proveedor y del dispositivo para actualizar el firmware de dispositivos USB.

Integrated Bootloader

- ST proporciona un conjunto dedicado de herramientas que permiten actualizar el firmware en la memoria flash utilizando este protocolo.
  - Además, algunas otras aplicaciones de código abierto, como la herramienta dfu-util, también se pueden utilizar tanto en Windows como en Linux y MacOS.
  - Para obtener más información sobre el modo USB DFU en los cargadores de arranque STM32, consulte el manual de usuario UM0412 de ST.

Starting the Bootloader From the On-Board Firmware

- La ejecución del gestor de arranque integrado está relacionada con el estado de los pines BOOT, que se muestrean durante los primeros ciclos de reloj.
  - Sin embargo, para varias opciones de diseño, es posible que no pueda configurar los pines de BOOT según sea necesario. Por esta razón, puede “saltar” a la System memory desde el firmware (por ejemplo, el usuario puede verse obligado a presionar un interruptor oculto).
- Forzar la ejecución del gestor de arranque desde el código de usuario no es tan difícil: solo se trata de definir un puntero de función.

```
1  __set_MSP(SRAM_END);
2  uint32_t JumpAddress = *(volatile uint32_t*)(0x1FFF0000 + 4);
3  void (*boot_loader)(void) = JumpAddress;
4  SYSCFG->MEMRMP = 0x1; //Remap 0x0000 0000 to System Memory
5  boot_loader();
6  //Never coming here
```

Starting the Bootloader From the On-Board Firmware

```
1  __set_MSP(SRAM_END);
2  uint32_t JumpAddress = *(volatile uint32_t*)(0x1FFF0000 + 4);
3  void (*boot_loader)(void) = JumpAddress;
4  SYSCFG->MEMRMP = 0x1; //Remap 0x0000 0000 to System Memory
5  boot_loader();
6  //Never coming here
```

- La instrucción en la línea 1 establece el puntero de la pila principal al final de la SRAM (normalmente esto no debería ser necesario, pero por si acaso...).
  - Luego creamos un puntero a una función cuya dirección se establece al comienzo de la System Memory y simplemente saltamos al gestor de arranque integrado llamando a la función boot_loader() después de una reasignación física a la System Memory.

Starting the Bootloader From the On-Board Firmware

- Sin embargo, debemos poner especial cuidado al saltar a System Memory.
  - De hecho, el gestor de arranque está diseñado para ser llamado justo después de un reinicio y asume que la CPU y sus periféricos están configurados en el estado inicial predeterminado.
  - Se podría lograr una mejor solución almacenando un código especial dentro de la memoria SRAM y luego forzando un reinicio del sistema en el software: podemos verificar desde el handler de excepciones de Reset este código especial y saltar a la memoria del sistema antes de cualquier otro procedimiento de inicialización.
- Este valor de protección debe almacenarse en una ubicación de memoria fuera de las regiones .data y .bss; de lo contrario, puede inicializarse durante el arranque del firmware (como alternativa, podemos colocar este código dentro del handler de excepciones Reset antes de que se inicialicen esas regiones).

The Booting Sequence in the GNU ARM Eclipse Tool-chain

- Ahora que el proceso de arranque está claro, podemos analizar un tema realmente fundamental:
  - ¿cuáles son los pasos exactos que realiza durante el arranque una aplicación desarrollada con GNU ARM Eclipse toolchain?
- La respuesta no es trivial y hay varias cosas importantes que un programador experimentado que trabaja con esta tool-chain debe saber.
- En el Capítulo 15 analizamos en profundidad la forma en que funciona una excepción de Reset.
  - Sin embargo, los ejemplos realizados en ese capítulo están aislados de la tool-chain real: hemos desarrollado una aplicación STM32 mínima que no utiliza CubeHAL ni los archivos de startup de GNU ARM Eclipse toolchain.
  - Entonces, para comprender la secuencia de inicio real, debemos comenzar desde el principio: la excepción de Reset.

The Booting Sequence in the GNU ARM Eclipse Tool-chain

```
76    .section  .text.Reset_Handler
77    .weak  Reset_Handler
78    .type  Reset_Handler, %function
79  Reset_Handler:
80    ldr   sp, =_estack                    /* set stack pointer */
81
82  /* Copy the data segment initializers from flash to SRAM */
83    movs  r1, #0
84    b  LoopCopyDataInit
85
86  CopyDataInit:
87    ldr  r3, =_sidata
88    ldr  r3, [r3, r1]
89    str  r3, [r0, r1]
90    adds  r1, r1, #4
91
92  LoopCopyDataInit:
93    ldr  r0, =_sdata
94    ldr  r3, =_edata
95    adds  r2, r0, r1
96    cmp  r2, r3
97    bcc  CopyDataInit
98    ldr  r2, =_sbss
99    b  LoopFillZerobss
100 /* Zero fill the bss segment. */
101 FillZerobss:
102   movs  r3, #0
103   str  r3, [r2], #4
104
105 LoopFillZerobss:
106   ldr  r3, = _ebss
107   cmp  r2, r3
108   bcc  FillZerobss
109
110 /* Call the clock system intitialization function.*/
111   bl  SystemInit
112 /* Call static constructors */
113     bl __libc_init_array
114 /* Call the application's entry point.*/
115   bl  main
116   bx  lr
117 .size  Reset_Handler, .-Reset_Handler
```

- En el Capítulo 7 hemos visto que los archivos assembly system/src/cmsis/startup_stm32xxxx.S contiene la definición de la tabla de vectores.
  - Estos archivos los proporciona ST y son específicos para la MCU STM32 determinada
  - Al abrir el que se adapta a su MCU, puede encontrar la definición de Reset_Handler, aproximadamente en la línea 76.
- Está escrito en assembly, pero debería ser muy fácil de entender ahora que dominamos muchos conceptos fundamentales.
  - Una nueva sección llamada .text.Reset_Handler se define en la línea 76, mientras que el cuerpo de la rutina comienza en la línea 80.

The Booting Sequence in the GNU ARM Eclipse Tool-chain

```
76    .section  .text.Reset_Handler
77    .weak  Reset_Handler
78    .type  Reset_Handler, %function
79  Reset_Handler:
80    ldr   sp, =_estack                    /* set stack pointer */
81
82  /* Copy the data segment initializers from flash to SRAM */
83    movs  r1, #0
84    b  LoopCopyDataInit
85
86  CopyDataInit:
87    ldr  r3, =_sidata
88    ldr  r3, [r3, r1]
89    str  r3, [r0, r1]
90    adds  r1, r1, #4
91
92  LoopCopyDataInit:
93    ldr  r0, =_sdata
94    ldr  r3, =_edata
95    adds  r2, r0, r1
96    cmp  r2, r3
97    bcc  CopyDataInit
98    ldr  r2, =_sbss
99    b  LoopFillZerobss
100 /* Zero fill the bss segment. */
101 FillZerobss:
102   movs  r3, #0
103   str  r3, [r2], #4
104
105 LoopFillZerobss:
106   ldr  r3, = _ebss
107   cmp  r2, r3
108   bcc  FillZerobss
109
110 /* Call the clock system intitialization function.*/
111   bl  SystemInit
112 /* Call static constructors */
113     bl __libc_init_array
114 /* Call the application's entry point.*/
115   bl  main
116   bx  lr
117 .size  Reset_Handler, .-Reset_Handler
```

- Aquí el MSP se establece en el contenido de la variable del linker _estack (coincide con el final de SRAM).
- Luego, el control se transfiere a la rutina LoopCopyDataInit, que inicializa la sección .data.
- Luego, el control se transfiere a la rutina LoopFillZerobss, que inicializa las secciones .bss y llama a la rutina SystemInit() (la analizaremos en un momento) y llama a los constructores estáticos de C++ llamando a __libc_init_array().
- Finalmente, transfiere el control a la rutina main().

The Booting Sequence in the GNU ARM Eclipse Tool-chain

```
76    .section  .text.Reset_Handler
77    .weak  Reset_Handler
78    .type  Reset_Handler, %function
79  Reset_Handler:
80    ldr   sp, =_estack                    /* set stack pointer */
81
82  /* Copy the data segment initializers from flash to SRAM */
83    movs  r1, #0
84    b  LoopCopyDataInit
85
86  CopyDataInit:
87    ldr  r3, =_sidata
88    ldr  r3, [r3, r1]
89    str  r3, [r0, r1]
90    adds  r1, r1, #4
91
92  LoopCopyDataInit:
93    ldr  r0, =_sdata
94    ldr  r3, =_edata
95    adds  r2, r0, r1
96    cmp  r2, r3
97    bcc  CopyDataInit
98    ldr  r2, =_sbss
99    b  LoopFillZerobss
100 /* Zero fill the bss segment. */
101 FillZerobss:
102   movs  r3, #0
103   str  r3, [r2], #4
104
105 LoopFillZerobss:
106   ldr  r3, = _ebss
107   cmp  r2, r3
108   bcc  FillZerobss
109
110 /* Call the clock system intitialization function.*/
111   bl  SystemInit
112 /* Call static constructors */
113     bl __libc_init_array
114 /* Call the application's entry point.*/
115   bl  main
116   bx  lr
117 .size  Reset_Handler, .-Reset_Handler
```

- Esta es la excepción de Reset proporcionada por ST.
  - ¡Pero espera! Si observa la línea 77, podrá ver que la rutina Reset_Handler está declarada weak: esto significa que otra rutina con el mismo nombre, definida en otra parte del árbol fuente, puede anular esta.
  - De hecho, si abre el archivo system/src/cortexm/exception_handlers.c, puede ver que el controlador se anula allí, aproximadamente en la línea 29, y llama a la función _start() que está definida dentro delarchivo system/src/newlib/_startup.c.

The Booting Sequence in the GNU ARM Eclipse Tool-chain

```
76    .section  .text.Reset_Handler
77    .weak  Reset_Handler
78    .type  Reset_Handler, %function
79  Reset_Handler:
80    ldr   sp, =_estack                    /* set stack pointer */
81
82  /* Copy the data segment initializers from flash to SRAM */
83    movs  r1, #0
84    b  LoopCopyDataInit
85
86  CopyDataInit:
87    ldr  r3, =_sidata
88    ldr  r3, [r3, r1]
89    str  r3, [r0, r1]
90    adds  r1, r1, #4
91
92  LoopCopyDataInit:
93    ldr  r0, =_sdata
94    ldr  r3, =_edata
95    adds  r2, r0, r1
96    cmp  r2, r3
97    bcc  CopyDataInit
98    ldr  r2, =_sbss
99    b  LoopFillZerobss
100 /* Zero fill the bss segment. */
101 FillZerobss:
102   movs  r3, #0
103   str  r3, [r2], #4
104
105 LoopFillZerobss:
106   ldr  r3, = _ebss
107   cmp  r2, r3
108   bcc  FillZerobss
109
110 /* Call the clock system intitialization function.*/
111   bl  SystemInit
112 /* Call static constructors */
113     bl __libc_init_array
114 /* Call the application's entry point.*/
115   bl  main
116   bx  lr
117 .size  Reset_Handler, .-Reset_Handler
```

- Esta rutina esencialmente realiza la inicialización de .data y .bss y transfiere el control a main(), pero antes de realizar estas operaciones, llama a la función __initialize_hardware_early() definida en el archivo system/src/cortexm/_initialize_hardware.c.
- Las líneas de código más relevantes de esa función se informan a continuación.

```
33  void __attribute__((weak))
34  __initialize_hardware_early(void)
35  {
36    // Call the CSMSIS system initialization routine.
37    SystemInit();
38
39  #if defined(__ARM_ARCH_7M__) || defined(__ARM_ARCH_7EM__)
40    // Set VTOR to the actual address, provided by the linker script.
41    // Override the manual, possibly wrong, SystemInit() setting.
42    SCB->VTOR = (uint32_t)(&__vectors_start);
43  #endif
44  ...
```

The Booting Sequence in the GNU ARM Eclipse Tool-chain

- Como puede ver, llama a la rutina SystemInit() y reubica la tabla de vectores en la dirección especificada por el símbolo del linker __vectors_start (esta operación no se realiza en Cortex-M0).
  - La rutina CMSIS SystemInit() depende de la plataforma y ST la proporciona dentro del archivo denominado system/src/cmsis/system_stm32xxxx.c.
    - Explicar el contenido exacto de esa rutina está fuera del alcance de este libro: es realmente específica para una MCU determinada y esencialmente realiza la inicialización temprana de algunos periféricos (principalmente el reloj).
    - Sin embargo, si echas un vistazo al final de esa rutina, podrás ver que ST también reubica la tabla de vectores con esta instrucción:

```
1  SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;
```

    - Como puede ver, el VTOR está configurado en la base de la memoria flash más un desplazamiento (VECT_TAB_OFFSET) que eventualmente puede definirse dentro del mismo archivo.

The Booting Sequence in the GNU ARM Eclipse Tool-chain

- Todo esto para decir que la reubicación efectiva de la tabla de vectores se realiza mediante el procedimiento de inicialización de GNU ARM Eclipse tool-chain y no mediante los archivos de inicio oficiales de ST.
  - Esto es algo importante que debe tener a mano si va a desarrollar secuencias de inicio personalizadas, como veremos más adelante.
- Finalmente, _start() también llama a la rutina __initialize_hardware(), que llama a la función CMSIS SystemCoreClockUpdate() proporcionada por ST dentro del archivo system/src/cmsis/system_stm32xxxx.c.
- Esta es una rutina dependiente de la plataforma que actualiza la variable global CMSIS SystemCoreClock de acuerdo con los registros de reloj específicos.
  - La variable SystemCoreClock se usa ampliamente dentro del código HAL y es importante mantenerla sincronizada con la configuración efectiva del árbol del reloj, como se ve en el Capítulo 10.

Developing a Custom Bootloader

- Lea cuidadosamente: El gestor de arranque descrito en este párrafo funciona correctamente si y sólo si la interfaz ST-LINK tiene una versión de firmware igual o superior a 2.27.15. Las versiones anteriores tienen un error en el VCP que impide que la interfaz USART funcione como se esperaba. Asegúrese de que su Nucleo esté actualizado.
- Los gestores de arranque integrados funcionan bien en muchos casos. Muchos proyectos reales pueden beneficiarse de su uso.
- Además, las herramientas gratuitas proporcionadas por ST pueden reducir el esfuerzo necesario para desarrollar aplicaciones personalizadas que cargan el firmware en la MCU. Sin embargo, para algunas aplicaciones es posible que necesites funcionalidades adicionales que no están implementadas en los cargadores de arranque estándar. Por ejemplo, es posible que deseemos cifrar el firmware distribuido para que solo el gestor de arranque integrado pueda decodificarlo utilizando una clave precompartida codificada dentro del código del gestor de arranque.

Developing a Custom Bootloader

- Ahora vamos a desarrollar un gestor de arranque personalizado que nos permitirá cargar un nuevo firmware en la MCU de destino. Básicamente, esto proporcionará solo una fracción de las funciones implementadas por los cargadores de arranque integrados, pero nos dará la oportunidad de revisar los pasos fundamentales necesarios para desarrollar un cargador de arranque personalizado. Proporcionará las siguientes funcionalidades:
  - Cargar un nuevo firmware usando la interfaz UART (en nuestro caso, la interfaz UART2 que proporcionan todas las placas Nucleo).
  - Recuperar el tipo de MCU.
  - Borrar una cantidad determinada de sectores/páginas flash.
  - Escribir una serie de bytes a partir de una dirección determinada.
  - Cifrar/descifrar el firmware intercambiado utilizando el algoritmo AES-128.

Developing a Custom Bootloader

- El código que analizaremos aquí se basa en el diseño de la memoria flash de los microcontroladores STM32F401RE, que se muestra en la Tabla 2 y se extrae del manual de referencia correspondiente.
  - Como puede ver, los 512 KB de memoria flash están divididos en siete sectores.
  - El primero, el sector 0 resaltado en azul en la Tabla 2, se utilizará para almacenar el gestor de arranque integrado.
  - Si está trabajando en una MCU STM32 diferente, consulte los ejemplos del libro para ver cómo se organizó el gestor de arranque para su MCU.

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Table 2: The flash memory organization in an STM32F401RE MCU

Developing a Custom Bootloader

- Una vez que la MCU se reinicia, el gestor de arranque comienza su ejecución.
  - Esto significa que el gestor de arranque se compila para que se asigne a partir de la dirección 0x0800 0000, como ocurre con todas las aplicaciones STM32 estándar que se ven en este libro.
- Se define una tabla de vectores realmente mínima, que permite a la MCU iniciar correctamente la ejecución.
  - El gestor de arranque muestrea el pin PC13, que en casi todas las placas Nucleo corresponde al botón azul de la placa.
  - Si se presiona el botón, comienza a aceptar comandos en la interfaz UART2.
  - De lo contrario, reubica inmediatamente el registro VTOR y pasa el control al handler de excepciones Reset del firmware principal.

Developing a Custom Bootloader

- También se proporciona un script complementario, escrito en Python.
  - Se llama flasher.py y puede encontrarlo dentro de los ejemplos del libro.
  - Describiremos cómo usarlo en el siguiente párrafo.
- Antes de entrar en detalles de los comandos utilizados para intercambiar mensajes con el gestor de arranque, comenzaremos a analizar los procedimientos ejecutados durante el proceso de arranque y la forma en que se transfiere el control al firmware principal.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
7  /* Global macros */
8  #define ACK                0x79
9  #define NACK               0x1F
10 #define CMD_ERASE          0x43
11 #define CMD_GETID          0x02
12 #define CMD_WRITE          0x2b
13
14 #define APP_START_ADDRESS  0x08004000 /* In STM32F401RE this corresponds with the start
15                                          address of Sector 1 */
16
17 #define SRAM_SIZE          96*1024     // STM32F401RE has 96 KB of RAM
18 #define SRAM_END           (SRAM_BASE + SRAM_SIZE)
19
20 #define ENABLE_BOOTLOADER_PROTECTION 0
21 /* Private variables ---------------------------------------------------------*/
22
23 /* The AES_KEY cannot be defined const, because the aes_enc_dec() function
24  temporarily modifies its content */
25 uint8_t AES_KEY[] = { 0x4D, 0x61, 0x73, 0x74, 0x65, 0x72, 0x69, 0x6E, 0x67,
26                       0x20, 0x20, 0x53, 0x54, 0x4D, 0x33, 0x32 };
27
28 extern CRC_HandleTypeDef hcrc;
29 extern UART_HandleTypeDef huart2;
```

- La macro APP_START_ADDRESS en la línea 14 define la dirección inicial del firmware principal.
- Según el layout de la memoria de una MCU STM32F401RE, el segundo sector comienza en esa dirección y el firmware de la aplicación principal se almacenará allí.
  - Esto significa que el MSP se colocará en 0x0800 4000 y la dirección en la memoria flash del handler de excepciones Reset en 0x0800 4004.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
7  /* Global macros */
8  #define ACK                0x79
9  #define NACK               0x1F
10 #define CMD_ERASE          0x43
11 #define CMD_GETID          0x02
12 #define CMD_WRITE          0x2b
13
14 #define APP_START_ADDRESS  0x08004000 /* In STM32F401RE this corresponds with the start
15                                          address of Sector 1 */
16
17 #define SRAM_SIZE          96*1024     // STM32F401RE has 96 KB of RAM
18 #define SRAM_END           (SRAM_BASE + SRAM_SIZE)
19
20 #define ENABLE_BOOTLOADER_PROTECTION 0
21 /* Private variables ---------------------------------------------------------*/
22
23 /* The AES_KEY cannot be defined const, because the aes_enc_dec() function
24  temporarily modifies its content */
25 uint8_t AES_KEY[] = { 0x4D, 0x61, 0x73, 0x74, 0x65, 0x72, 0x69, 0x6E, 0x67,
26                       0x20, 0x20, 0x53, 0x54, 0x4D, 0x33, 0x32 };
27
28 extern CRC_HandleTypeDef hcrc;
29 extern UART_HandleTypeDef huart2;
```

- El array AES_KEY, definida en la línea 25, contiene dieciséis bytes que forman la clave AES-128 utilizada para cifrar/descifrar el firmware cargado.
  - Analizaremos su uso más adelante.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
44  /* Minimal vector table */
45  uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
46      (uint32_t *) SRAM_END, // initial stack pointer
47      (uint32_t *) _start,   // _start is the Reset_Handler
48      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (uint32_t *) SysTick_Handler };
```

- La tabla de vectores está definida en la línea 45.
  - Solo contiene el puntero MSP, que coincide con el final de la memoria SRAM, el puntero al handler de excepciones Reset (_start en este caso, que no hace más que inicializar .data y .bss secciones y transferir el control a la rutina main()), y el puntero al SysTick_Handler.
  - Esto es necesario porque usaremos las rutinas HAL estándar para interconectar periféricos, y HAL se construye alrededor de una base de tiempo única, generalmente generada usando el temporizador SysTick. Por lo tanto, HAL necesita habilitar ese temporizador y detectar el evento de desbordamiento para que aumente la cuenta global de ticks.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
93  int main(void) {
94    uint32_t ulTicks = 0;
95    uint8_t ucUartBuffer[20];
96
97    /* HAL_Init() sets SysTick timer so that it overflows every 1ms */
98    HAL_Init();
99    MX_GPIO_Init();
100
101 #if ENABLE_BOOTLOADER_PROTECTION
102   /* Ensures that the first sector of flash is write-protected preventing that the
103    bootloader is overwritten */
104   CHECK_AND_SET_FLASH_PROTECTION();
105 #endif
106
107   /* If USER_BUTTON is pressed */
108   if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
109     /* CRC and UART2 peripherals enabled */
110     MX_CRC_Init();
111     MX_USART2_UART_Init();
112
113     ulTicks = HAL_GetTick();
114
115     while (1) {
116       /* Every 500ms the LD2 LED blinks, so that we can see the bootloader running. */
117       if (HAL_GetTick() - ulTicks > 500) {
118         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
119         ulTicks = HAL_GetTick();
120       }
121
122       /* We check for new commands arriving on the UART2 */
123       HAL_UART_Receive(&huart2, ucUartBuffer, 20, 10);
124       switch (ucUartBuffer[0]) {
125       case CMD_GETID:
126         cmdGetID(ucUartBuffer);
127         break;
128       case CMD_ERASE:
129         cmdErase(ucUartBuffer);
130         break;
131       case CMD_WRITE:
132         cmdWrite(ucUartBuffer);
133         break;
134       };
135     }
136   } else {
137     /* USER_BUTTON is not pressed. We first check if the first 4 bytes starting from
138      APP_START_ADDRESS contain the MSP(end of SRAM). If not, the LD2 LED blinks quickly. */
139     if (*((uint32_t*) APP_START_ADDRESS) != SRAM_END) {
140       while (1) {
141         HAL_Delay(30);
142         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
143       }
144     } else {
145       /* A valid program seems to exist in the second sector: we so prepare the MCU
146        to start the main firmware */
147       MX_GPIO_Deinit(); //Puts GPIOs in default state
148       SysTick->CTRL = 0x0; //Disables SysTick timer and its related interrupt
149       HAL_DeInit();
150
151       RCC->CIR = 0x00000000; //Disable all interrupts related to clock
152       __set_MSP(*((volatile uint32_t*) APP_START_ADDRESS)); //Set the MSP
153
154       __DMB(); //ARM says to use a DMB instruction before relocating VTOR */
155       SCB->VTOR = APP_START_ADDRESS; //We relocate vector table to the sector 1
156       __DSB(); //ARM says to use a DSB instruction just after relocating VTOR */
157
158       /* We are now ready to jump to the main firmware */
159       uint32_t JumpAddress = *((volatile uint32_t*) (APP_START_ADDRESS + 4));
160       void (*reset_handler)(void) = (void*)JumpAddress;
161       reset_handler(); //We start the execution from he Reset_Handler of the main firmware
162
163       for (;;)
164         ; //Never coming here
165     }
166   }
167 }
```

Developing a Custom Bootloader

- Ahora vamos a explicar las tareas que realiza la rutina main().
  - Una vez que es llamado por el handler de excepciones Reset (rutina _start()), primero inicializa el CubeHAL, reduciendo al mínimo la cantidad de operaciones realizadas en esta fase: esto ayuda a reducir el tiempo de arranque.
  - La rutina HAL_Init() también configura el temporizador SysTick para que caduque cada 1 ms.
  - El pin PC13 está así muestreado, y si el usuario mantiene presionado el BOTÓN DE USUARIO, entonces la rutina entra en un bucle infinito aceptando tres comandos en el UART2. Los analizaremos más adelante.
  - Tenga en cuenta que dejamos la fuente de reloj predeterminada como está (es decir, el oscilador HSI).

Developing a Custom Bootloader

- Si, en cambio, se deja sin presionar el BOTÓN DE USUARIO, entonces la rutina main() verifica si la primera ubicación de memoria del segundo sector contiene el MSP (simplemente verificamos que contenga el valor SRAM_-END).
  - De lo contrario, el firmware comienza a parpadear el LED LD2 muy rápido para indicar que no hay ninguna aplicación principal para ejecutar.
- Si esa ubicación de memoria contiene el puntero MSP (línea 144), podemos iniciar la secuencia de inicio.
  - Los GPIO se colocan en su estado predeterminado, el HAL se desinicializa, el temporizador SysTick se detiene y su excepción se desactiva.
  - Todas las interrupciones relacionadas con el reloj se desactivan en la línea 151 y el MSP se establece en la dirección especificada en los primeros 4 bytes del sector 1 (porque la tabla de vectores se coloca allí, como veremos más adelante).

- La ubicación base de VTOR está configurada en APP_START_ADDRESS (es decir, 0x0800 4000 para el gestor de arranque STM32F401RE).
  - La dirección de la excepción de Reset para el firmware principal se deriva de la ubicación de memoria 0x0800 4004 y se define un puntero a esa función.
    - Finalmente, en la línea 161 se invoca la excepción Reset y el gestor de arranque finaliza.

- Antes de analizar los tres comandos implementados por el gestor de arranque, es mejor echar un vistazo rápido a la otra aplicación que se incluye con los ejemplos de este capítulo.
  - Se llama main-app1.c y no es más que una simple aplicación que hace parpadear el LED LD2 e imprime un mensaje en el UART2.

Developing a Custom Bootloader

- Lo único relevante a tener en cuenta es el linker script complementario, denominado ldscript-app.ld, que define la región de memoria FLASH de la siguiente manera:

Filename: src/ldscript-app.ld

```
14  MEMORY {
15    FLASH (rx) : ORIGIN = 0x08004000, LENGTH = 512K - 16K
16    RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 96K
```

- Como puede ver, el linker reubicará el código de la aplicación a partir de la dirección 0x0800 4000.
- Además, la longitud de esta región de memoria se establece en 496 KB: dado que el primer sector tiene 16 KB de ancho, 512-16 es igual a 496. Esta definición de la región de memoria flash también nos permite cargar y depurar el firmware usando OpenOCD (o el Utilidad ST-LINK) sin sobrescribir el gestor de arranque.

Developing a Custom Bootloader

- NOTA: Según lo visto en el párrafo anterior, el valor VTOR establecido por el gestor de arranque será sobrescrito por la rutina de inicio de la aplicación principal. Sin embargo, el código seguirá funcionando sin problemas, porque en el linker script de main-app1.c el símbolo __vectors_-start coincide con la macro APP_START_ADDRESS (es decir, 0x0800 4000). Este es un aspecto importante a tener en cuenta a la hora de programar un gestor de arranque.
- Ahora es el momento adecuado para analizar los tres comandos soportados por este gestor de arranque: CMD_GETID, CMD_ERASE y CMD_WRITE.
  - Comando Get ID: El comando CMD_GETID se utiliza para recuperar el ID de la MCU y tiene la estructura que se muestra en la Figura 4.

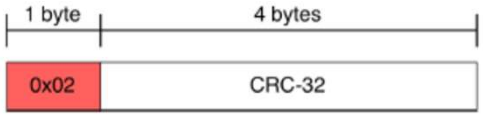


Figure 4: The structure of the CMD_GETID

  - El gestor de arranque espera recuperar el byte 0x02 seguido del CRC-32 de este byte.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
223 void cmdGetID(uint8_t *pucData) {
224   uint16_t usDevID;
225   uint32_t ulCrc = 0;
226   uint32_t ulCmd = pucData[0];
227
228   memcpy(&ulCrc, pucData + 1, sizeof(uint32_t));
229
230   /* Checks if provided CRC is correct */
231   if (ulCrc == HAL_CRC_Calculate(&hcrc, &ulCmd, 1)) {
232     usDevID = (uint16_t) (DBGMCU->IDCODE & 0xFFF); //Retrieves MCU ID from DEBUG interface
233
234     /* Sends an ACK */
235     pucData[0] = ACK;
236     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
237
238     /* Sends the MCU ID */
239     HAL_UART_Transmit(&huart2, (uint8_t *) &usDevID, 2, HAL_MAX_DELAY);
240   } else {
241     /* The CRC is wrong: sends a NACK */
242     pucData[0] = NACK;
243     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
244   }
245 }
```

- El gestor de arranque responde a la solicitud enviando un ACK (que se define en la línea 8 del archivo mainbootloader.c y es igual a 0x79) seguido de dos bytes que contienen el ID de la MCU.

El código anterior muestra cómo se implementa el comando. Como puede ver, el CRC se extrae del mensaje que llega al UART y se compara con el calculado por el periférico CRC. Si los dos valores coinciden, entonces el ID de la MCU se deriva de la interfaz DEBUG y se transmite a través del UART junto con el ACK. Si el CRC no coincide, se envía un NACK (que es igual a 0x1F).

Developing a Custom Bootloader

- Comando Erase
  - El comando CMD_ERASE se utiliza para borrar un sector determinado de la memoria flash y tiene la estructura que se muestra en la Figura 5.
    - El comando está compuesto por el id 0x43 que identifica el tipo de comando, seguido de la cantidad de sectores a eliminar (o el valor 0xFF para eliminar todos los sectores excepto el primero donde reside el gestor de arranque) y el CRC-32.

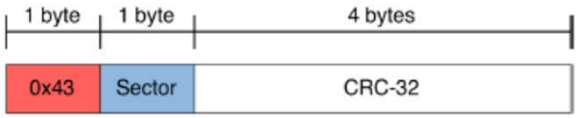


Figure 5: The structure of the CMD_ERASE

  - El gestor de arranque responde enviando un ACK cuando se completa el procedimiento de borrado.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
180 void cmdErase(uint8_t *pucData) {
181   FLASH_EraseInitTypeDef eraseInfo;
182   uint32_t ulBadBlocks = 0, ulCrc = 0;
183   uint32_t pulCmd[] = { pucData[0], pucData[1] };
184
185   memcpy(&ulCrc, pucData + 2, sizeof(uint32_t));
186
187   /* Checks if provided CRC is correct */
188   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulCmd, 2) &&
189       (pucData[1] > 0 && (pucData[1] < FLASH_SECTOR_TOTAL - 1 || pucData[1] == 0xFF))) {
190     /* If data[1] contains 0xFF, it deletes all sectors; otherwise
191      * the number of sectors specified. */
192     eraseInfo.Banks = FLASH_BANK_1;
193     eraseInfo.Sector = FLASH_SECTOR_1;
194     eraseInfo.NbSectors = pucData[1] == 0xFF ? FLASH_SECTOR_TOTAL - 1 : pucData[1];
195     eraseInfo.TypeErase = FLASH_TYPEERASE_SECTORS;
196     eraseInfo.VoltageRange = FLASH_VOLTAGE_RANGE_3;
197
198     HAL_FLASH_Unlock(); //Unlocks the flash memory
199     HAL_FLASHEx_Erase(&eraseInfo, &ulBadBlocks); //Deletes given sectors */
200     HAL_FLASH_Lock(); //Locks again the flash memory
201
202     /* Sends an ACK */
203     pucData[0] = ACK;
204     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
205   } else {
206     /* The CRC is wrong: sends a NACK */
207     pucData[0] = NACK;
208     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
209   }
210 }
```

- El código anterior muestra cómo se implementa el comando.
  - Como puede ver, el CRC se extrae del mensaje que llega al UART y se compara con el calculado por el periférico CRC.
  - Tenga en cuenta que, dado que el periférico CRC tiene un registro de datos de 32 bits de ancho y el CRC-32 se calcula en todo el registro, convertimos los primeros dos bytes en dos valores de 32 bits.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
180 void cmdErase(uint8_t *pucData) {
181   FLASH_EraseInitTypeDef eraseInfo;
182   uint32_t ulBadBlocks = 0, ulCrc = 0;
183   uint32_t pulCmd[] = { pucData[0], pucData[1] };
184
185   memcpy(&ulCrc, pucData + 2, sizeof(uint32_t));
186
187   /* Checks if provided CRC is correct */
188   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulCmd, 2) &&
189       (pucData[1] > 0 && (pucData[1] < FLASH_SECTOR_TOTAL - 1 || pucData[1] == 0xFF))) {
190     /* If data[1] contains 0xFF, it deletes all sectors; otherwise
191      * the number of sectors specified. */
192     eraseInfo.Banks = FLASH_BANK_1;
193     eraseInfo.Sector = FLASH_SECTOR_1;
194     eraseInfo.NbSectors = pucData[1] == 0xFF ? FLASH_SECTOR_TOTAL - 1 : pucData[1];
195     eraseInfo.TypeErase = FLASH_TYPEERASE_SECTORS;
196     eraseInfo.VoltageRange = FLASH_VOLTAGE_RANGE_3;
197
198     HAL_FLASH_Unlock(); //Unlocks the flash memory
199     HAL_FLASHEx_Erase(&eraseInfo, &ulBadBlocks); //Deletes given sectors */
200     HAL_FLASH_Lock(); //Locks again the flash memory
201
202     /* Sends an ACK */
203     pucData[0] = ACK;
204     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
205   } else {
206     /* The CRC is wrong: sends a NACK */
207     pucData[0] = NACK;
208     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
209   }
210 }
```

- Si el CRC coincide, entonces se completa una instancia de la estructura FLASH_EraseInitTypeDef para que los sectores flash se borren comenzando desde el segundo (línea 193) hasta la cantidad de sectores especificados (línea 194).
  - La memoria flash queda así desbloqueada (línea 198) y el procedimiento de borrado se realiza llamando a la rutina HAL_FLASHEx_Erase().

Developing a Custom Bootloader

- Comando Write
  - El comando CMD_WRITE se utiliza para almacenar dieciséis bytes (es decir, cuatro palabras) a partir de una ubicación de memoria determinada y tiene la estructura que se muestra en la Figura 6.

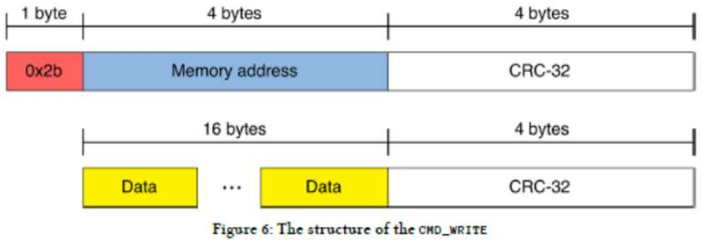


Figure 6: The structure of the CMD_WRITE

  - El comando se compone de dos partes distintas. El primero está compuesto por el id del comando 0x2b, seguido de la dirección inicial donde colocar los bytes de datos y el CRC-32 del comando.
  - Si el CRC coincide y la dirección especificada es igual o superior a APP_START_ADDRESS, el gestor de arranque responde con un ACK.
  - El gestor de arranque espera recibir otra secuencia formada por dieciséis bytes y la suma de comprobación CRC-32 de estos bytes.

Developing a Custom Bootloader

Filename: src/main-bootloader.c

```
267 void cmdWrite(uint8_t *pucData) {
268   uint32_t ulSaddr = 0, ulCrc = 0;
269
270   memcpy(&ulSaddr, pucData + 1, sizeof(uint32_t));
271   memcpy(&ulCrc, pucData + 5, sizeof(uint32_t));
272
273   uint32_t pulData[5];
274   for(int i = 0; i < 5; i++)
275     pulData[i] = pucData[i];
276
277   /* Checks if provided CRC is correct */
278   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulData, 5) && ulSaddr >= APP_START_ADDRESS) {
279     /* Sends an ACK */
280     pucData[0] = ACK;
281     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
282
283     /* Now retrieves given amount of bytes plus the CRC32 */
284     if (HAL_UART_Receive(&huart2, pucData, 16 + 4, 200) == HAL_TIMEOUT)
285       return;
286
287     memcpy(&ulCrc, pucData + 16, sizeof(uint32_t));
288
289     /* Checks if provided CRC is correct */
290     if (ulCrc == HAL_CRC_Calculate(&hcrc, (uint32_t*) pucData, 4)) {
291       HAL_FLASH_Unlock(); //Unlocks the flash memory
292
293       /* Decode the sent bytes using AES-128 ECB */
294       aes_enc_dec((uint8_t*) pucData, AES_KEY, 1);
295       for (uint8_t i = 0; i < 16; i++) {
296         /* Store each byte in flash memory starting from the specified address */
297         HAL_FLASH_Program(FLASH_TYPEPROGRAM_BYTE, ulSaddr, pucData[i]);
298         ulSaddr += 1;
299       }
300       HAL_FLASH_Lock(); //Looks again the flash memory
301
302       /* Sends an ACK */
303       pucData[0] = ACK;
304       HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
305     } else {
306       goto sendnack;
307     }
308   } else {
309     goto sendnack;
310   }
311
312 sendnack:
313   pucData[0] = NACK;
314   HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
315 }
```

Developing a Custom Bootloader

- El código anterior muestra cómo se implementa el comando.
  - Como puede ver, el CRC de la primera parte del mensaje se compara con el valor transmitido (líneas [273:278]).
  - Si corresponde se envía un ACK y se procesan los siguientes bytes. Si el CRC-32 de estos otros bytes coincide (línea 290), los bytes de datos enviados se descifran utilizando el algoritmo AES-128 y la clave precompartida.
  - Los bytes de datos se almacenan dentro de la memoria flash a partir de la ubicación de memoria dada.
- Hay una cosa más para analizar: la función CHECK_AND_SET_FLASH_PROTECTION() invocada por la función main() si la macro ENABLE_BOOTLOADER_PROTECTION está configurada en 1.

Developing a Custom Bootloader

```
317 void CHECK_AND_SET_FLASH_PROTECTION(void) {
318   FLASH_OBProgramInitTypeDef obConfig;
319
320   /* Retrieves current OB */
321   HAL_FLASHEx_OBGetConfig(&obConfig);
322
323   /* If the first sector is not protected */
324   if ((obConfig.WRPSector & OB_WRP_SECTOR_0) == OB_WRP_SECTOR_0) {
325     HAL_FLASH_Unlock(); //Unlocks flash
326     HAL_FLASH_OB_Unlock(); //Unlocks OB
327     obConfig.OptionType = OPTIONBYTE_WRP;
328     obConfig.WRPState = OB_WRPSTATE_ENABLE; //Enables changing of WRP settings
329     obConfig.WRPSector = OB_WRP_SECTOR_0; //Enables WP on first sector
330     HAL_FLASHEx_OBProgram(&obConfig); //Programs the OB
331     HAL_FLASH_OB_Launch(); //Ensures that the new configuration is saved in flash
332     HAL_FLASH_OB_Lock(); //Locks OB
333     HAL_FLASH_Lock(); //Locks flash
334   }
335 }
```

- Esta función simplemente recupera la configuración actual de Optio Bytes y verifica si el primer sector está protegido contra escritura (línea 324). De lo contrario, la protección contra escritura está habilitada para que no se pueda sobrescribir el gestor de arranque.
- Si desea experimentar con esta función, para desactivar la protección contra escritura puede utilizar la utilidad ST-LINK si trabaja en Windows. De lo contrario, los usuarios de Linux y MacOS pueden acceder a la consola OpenOCD y utilizar el siguiente comando:

```
$ telnet localhost 4444
...
$ flash protect 0 0 last off
```

- El comando anterior desactivará la protección contra escritura en todas las páginas/sectores.

- Algunas consideraciones sobre el gestor de arranque personalizado
  - El gestor de arranque personalizado que se presenta aquí está lejos de estar completo. Carece de algunas características relevantes y, lo más importante, no es lo suficientemente robusto como para cubrir condiciones de error.
  - Además, el único gestor de arranque para las plataformas STM32F0/L0 pesa aproximadamente 13 KB cuando se compila con la opción GCC -Os, que produce la imagen binaria de mayor tamaño optimizado. Definitivamente esto es demasiado para un gestor de arranque. Desafortunadamente, HAL tiene una sobrecarga no despreciable en el tamaño final de la imagen binaria. Un gestor de arranque bien diseñado está codificado reduciendo al mínimo su huella. Este aspecto está fuera del alcance de este libro, que simplemente muestra los conceptos fundamentales detrás del proceso de arranque.

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Hasta ahora, hemos visto que en los microcontroladores basados en Cortex-M0 no es posible reubicar la tabla de vectores como ocurre en las MCU Cortex-M0+/3/4/7.
  - Esto significa que no podemos usar el código visto antes (en las líneas [154:161]) para pasar el control al firmware principal, porque los núcleos Cortex-M0 siempre esperan encontrar la tabla de vectores en la dirección 0x0000 0000, y ésta coincide con la tabla de vectores del gestor de arranque en nuestro escenario.
- Sin embargo, podemos sortear esta limitación de una manera algo más astuta.
  - La idea que vamos a analizar se basa en que la reasignación física del software permite “aliasar” la memoria SRAM en la dirección 0x0000 0000, mientras que la memoria flash original siempre está accesible en la dirección 0x0800 0000.
  - De este modo, podemos reubicar la tabla de vectores del firmware principal antes de pasar el control a su handler de excepciones Reset simplemente copiando la tabla de vectores "destino" dentro de la SRAM y luego realizando la reasignación física.

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Las direcciones contenidas dentro de la tabla de vectores de destino aún son accesibles en sus ubicaciones originales, lo que permite la ejecución correcta de controladores de excepciones e ISR.

- La figura 7 intenta representar este procedimiento. En el lado izquierdo tenemos la aplicación principal (no se muestra el gestor de arranque).
  - Supongamos por simplicidad que su tabla de vectores está ubicada en la dirección 0x0800 2C00.
  - Esto significa que, a partir de la dirección 0x0800 2C04, tenemos la dirección en memoria de los controladores de excepciones e ISR Cortex-M0.

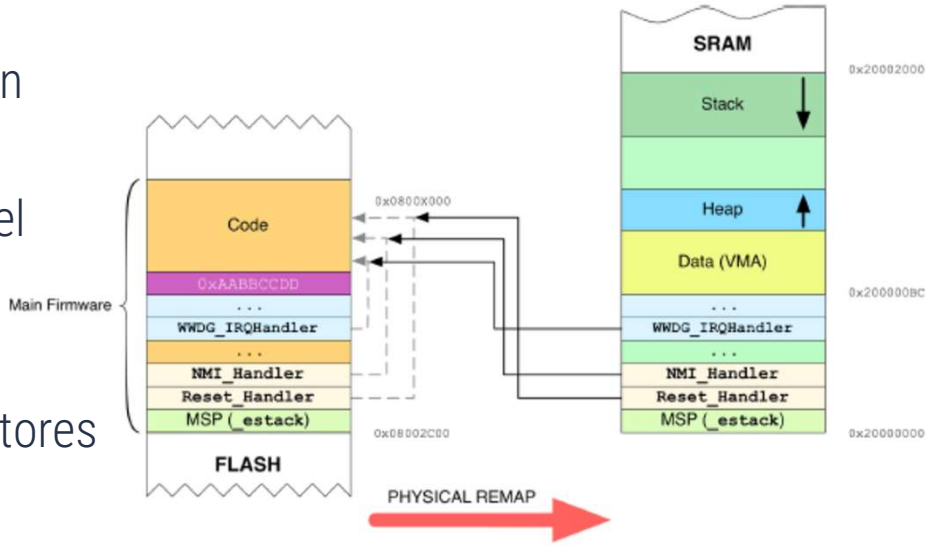


Figure 7: How the *vector table* can be relocated in STM32F0 microcontrollers

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Claramente, estas direcciones apuntan a otras ubicaciones de memoria por encima de la dirección 0x0800 2C00 (en la Figura 7 se representan como flechas grises).

- El gestor de arranque funciona de la siguiente manera. Copia la tabla de vectores dentro de la memoria SRAM, comenzando por colocar su contenido desde la dirección inicial 0x2000 0000.
  - Esto significa que desde la ubicación de memoria 0x2000 0004 tenemos las direcciones en la memoria flash de los manejadores de excepciones e ISR.

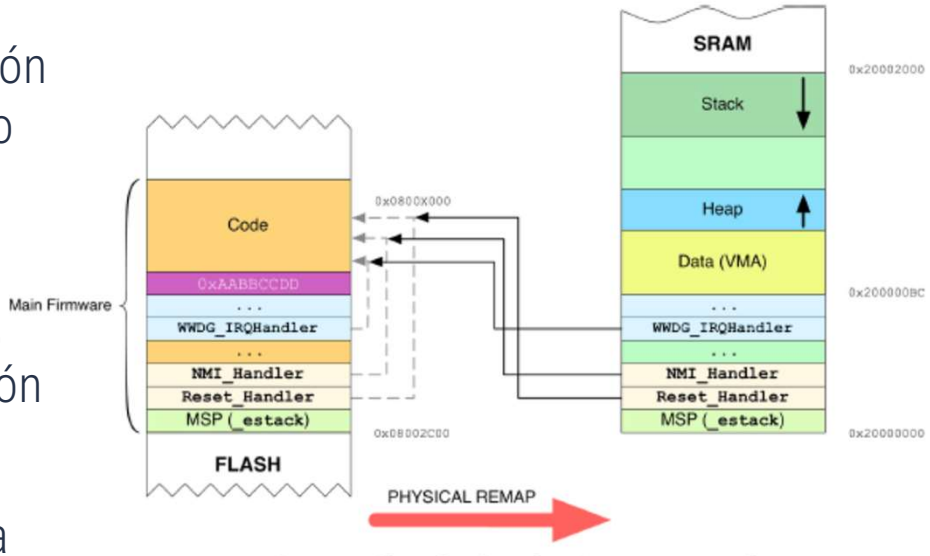


Figure 7: How the *vector table* can be relocated in STM32F0 microcontrollers

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Claramente, estas direcciones todavía apuntan a las mismas ubicaciones de memoria flash originales, como lo indican las flechas negras en la Figura 7.
  - Al final del procedimiento de copia, la memoria se reasigna, de modo que la dirección 0x0000 0000 ahora coincide con la dirección 0x2000 0000.
  - Luego, el control se transfiere al handler de excepciones Reset del firmware principal y se lleva a cabo su ejecución.
  - De esta manera hemos superado la limitación de las MCU basadas en Cortex-M0, que no permiten reubicar en la memoria la tabla de vectores.

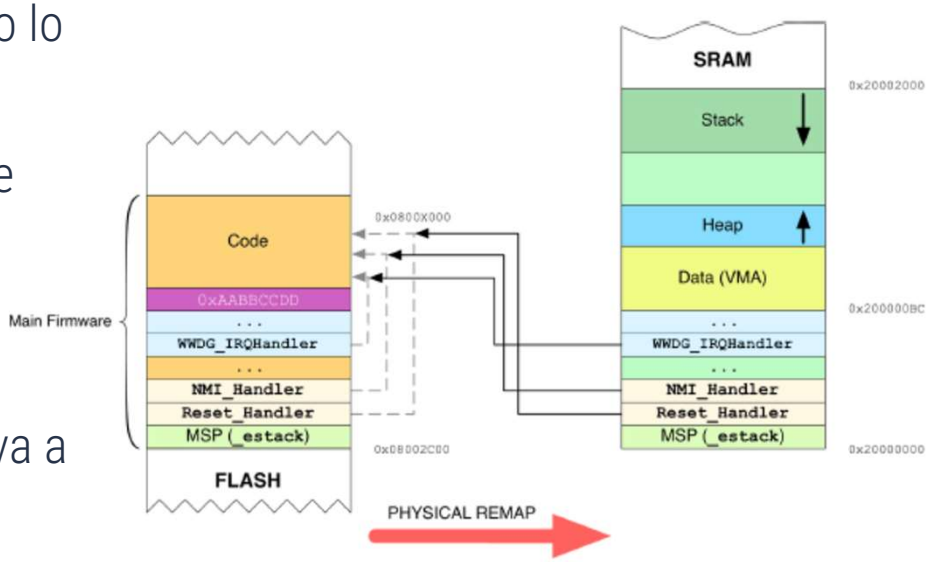


Figure 7: How the *vector table* can be relocated in STM32F0 microcontrollers

Vector Table Relocation in STM32F0/L0 Microcontrollers

- El siguiente código muestra nuestro gestor de arranque implementado para el microcontrolador STM32F030.
  - Sólo se muestra la parte relacionada con la reubicación de la tabla de vectores.

Filename: src/main-bootloader.c

```
146     } else {
147       /* A valid program seems to exist in the second sector: we so prepare the MCU
148        to start the main firmware */
149       MX_GPIO_Deinit(); //Puts GPIOs in default state
150       SysTick->CTRL = 0x0; //Disables SysTick timer and its related interrupt
151       HAL_DeInit();
152
153       RCC->CIR = 0x00000000; //Disable all interrupts related to clock
154
```

```
155       uint32_t *pulSRAMBase = (uint32_t*)SRAM_BASE;
156       uint32_t *pulFlashBase = (uint32_t*)APP_START_ADDRESS;
157       uint16_t i = 0;
158
159       do {
160         if(pulFlashBase[i] == 0xAABBCCDD)
161           break;
162         pulSRAMBase[i] = pulFlashBase[i];
163       } while(++i);
164
165       __set_MSP(*((volatile uint32_t*) APP_START_ADDRESS)); //Set the MSP
166
167       SYSCFG->CFGR1 |= 0x3; /* __HAL_RCC_SYSCFG_CLK_ENABLE()
168                                already called from HAL_MspInit() */
169
170       /* We are now ready to jump to the main firmware */
171       uint32_t JumpAddress = *((volatile uint32_t*) (APP_START_ADDRESS + 4));
172       void (*reset_handler)(void) = (void*)JumpAddress;
173       reset_handler(); //We start the execution from he Reset_Handler of the main firmware
174
175       for (;;)
176         ; //Never coming here
177     }
178   }
```

Vector Table Relocation in STM32F0/L0 Microcontrollers

- El código que nos interesa comienza en la línea 154.
  - Se definen dos punteros: uno que comienza al inicio de la memoria SRAM (pulSRAMBase) y otro al inicio del firmware principal (pulFlashBase, que es igual a 0x0800 2C00 siguiendo el ejemplo anterior).
  - El loop en las líneas [158:162] hace una copia de la tabla de vectores en SRAM, hasta que la ubicación actual de la memoria flash contiene el valor 0xAABBCCDD (más sobre esto pronto).
  - Luego, el MSP se configura al final de la SRAM (esto debería ser innecesario, pero por si acaso...) y se realiza la reasignación física (línea 166).
  - Luego, el control se transfiere al firmware principal.

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Hay varias cosas a tener en cuenta.
  - En primer lugar, para simplificar el proceso de copia y evitar que la tabla de vectores sea sobrescrita por la pila creciente, la tabla de vectores se copia en SRAM desde su inicio, y el resto de datos de la aplicación (formados por la sección .data, .bss, heap y stack) se coloca a continuación (consulte la Figura 7).
  - Esto requiere que el linker script del firmware principal esté configurado correctamente, como se muestra a continuación:

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x08002C00, LENGTH = 64K - 10K
  RAM (xrw) : ORIGIN = 0x200000B8, LENGTH = 8K - 0xB8
```

  - En segundo lugar, necesitamos una forma de saber dónde terminan las tablas de vectores.
    - Dado que no todas las IRQ suelen estar habilitadas en una aplicación, podemos colocar el valor centinela 0xAABBCCDD dentro de la primera entrada de vector que viene justo después de la última IRQ utilizada.

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Por ejemplo, suponiendo que nuestro firmware principal usa USART2 en modo de interrupción, podemos ver que esta IRQ es la entrada número 46 dentro de la tabla de vectores. Entonces podemos colocar ese valor en la entrada 47.
  - Esto se puede realizar fácilmente modificando el archivo startup_stm32f0xxx.S, como se muestra a continuación.

Filename: src/startup_stm32f030x8.S

```
180   .word  SPI1_IRQHandler          /* SPI1                */
181   .word  SPI2_IRQHandler          /* SPI2                */
182   .word  USART1_IRQHandler        /* USART1              */
183   .word  USART2_IRQHandler        /* USART2              */
184   .word  0xAABBCCDD               /* Reserved            */
185   .word  0                        /* Reserved            */
186   .word  0                        /* Reserved            */
```

  - De esta manera tenemos una forma genérica y configurable de establecer el final de la tabla de vectores.

Vector Table Relocation in STM32F0/L0 Microcontrollers

- Al observar el fragmento de secuencia de comandos del vinculador anterior, podemos ver que restamos del tamaño de la memoria SRAM el valor 0xB8, que es 184 en base 10.
- Dividiendo 184 entre 4 bytes, tenemos 46, que corresponde a la última entrada de la tabla de vectores.

- Finalmente, tenga en cuenta que SYSCFG es un periférico separado del núcleo Cortex-M y debemos habilitarlo llamando a __HAL_RCC_SYSCFG_CLK_ENABLE().

How to Use the flasher.py Tool

- Como se dijo antes, puede encontrar un script de Python llamado flasher.py dentro de los archivos fuente del libro de este capítulo.
  - Esta herramienta simplemente permite cargar en la MCU un firmware generado utilizando el formato binario Intel HEX, una especificación para archivos binarios desarrollada por Intel hace varios años y todavía muy extendida, especialmente en plataformas integradas de bajo coste.
  - El código fuente de este script no se muestra aquí, pero debería ser muy fácil entender la forma en que está hecho. Este script requiere tres módulos adicionales: bibliotecas pyserial, IntelHex y pycrypto.
- Los usuarios de Linux y Mac pueden instalarlos fácilmente usando el comando pip:

```
$ sudo pip install intelhex crypto pyserial
```

How to Use the flasher.py Tool

- En cambio, el usuario de Windows puede instalar módulos pyserial e IntelHex usando el comando pip: `$ sudo pip install intelhex pyserial`
- mientras que necesitan descargar una versión precompilada de la biblioteca pycryto desde este sitio web (elija la versión que se ajuste a su versión de Python y tipo de plataforma).
- El script está diseñado para aceptar dos argumentos en la línea de comando:
  - El puerto serie correspondiente al Nucleo VCP
  - – En Windows esto es igual a la cadena “COMx”, donde ‘x’ debe reemplazarse con el número COM correspondiente a Nucleo VCP (por ejemplo, COM3).
  - – En Linux y Mac OS esto corresponde a un archivo asignado en la ruta /dev (normalmente algo similar a /dev/tty.usbmodemXXXX).
  - El path completo al archivo HEX correspondiente al firmware principal.

How to Use the flasher.py Tool

- De forma predeterminada, GNU ARM Eclipse tool-chain genera automáticamente el archivo HEX del firmware compilado.
  - Puede encontrarlo dentro de la carpeta de compilación: esta es una carpeta de Eclipse con el mismo nombre de la configuración de compilación activa (generalmente llamada Debug or Release).
  - La Figura 8 muestra la carpeta de compilación correspondiente a la configuración activa (CH17-APP1) si está trabajando en el repositorio oficial de muestras de libros.

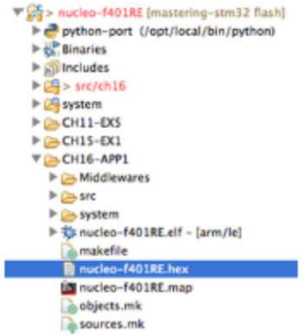


Figure 8: The binary file in HEX format inside the Eclipse build folder

How to Use the flasher.py Tool

- Puede derivar el path completo al archivo HEX haciendo clic con el botón derecho del mouse sobre él y luego seleccionando Propiedades.
  - Puede encontrar el path completo dentro de la vista de Recursos, como se muestra en la Figura 9.

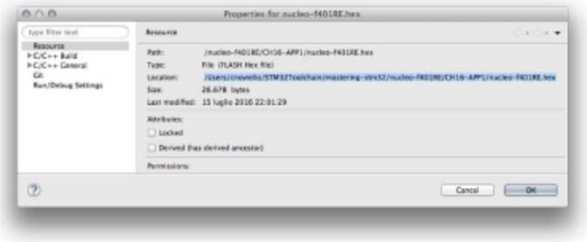


Figure 9: How to derive the full path of the HEX file

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