

Microcontroladores y Microprocesadores

Introducción a los microcontroladores con RUST

Author: Casimiro Gómez & Carlos Gómez

Institute: Naktan

Date: Primavera, 2022

Version: 0.1

Bio: Director del laboratorio de Sistemas Embebidos, UPAEP



El único camino al éxito, es la disciplina

Índice general

1. Configuración del ecosistema Rust	2
1.1. ¿Cuándo debo usar un microcontrolador?	2
1.2. ¿Cuándo no debo usar un microcontrolador?	3
1.3. Descripción del Hardware	4
1.4. Configurando el ambiente de desarrollo	5
1.5. Instalación en windows	9
1.6. Instalación de MacOS	9
1.7. Verificando cargo-embed	9
2. Mapa de Memoria	12
2.1. El lado de Rust	12
2.2. El lado del script del enlazador	13
2.3. Uniendo todo	16
2.4. revisando	17
3. Control de GPIO	18
3.1. Configurando el proyecto	18
3.2. Encendiendo un LED en la tarjeta STM32F411RE	19
3.3. Controlando un bus de GPIO	22

Prólogo

Los microcontroladores utilizan distintos ecosistemas para su manejo, algunas de las cuales implican niveles de conocimiento desde simples hasta complejos en cuanto al hardware y la arquitectura. Desde el ecosistema de **mbed** para microcontroladores de arquitectura ARM donde el enfoque se va más a la programación, las librerías HAL de ST que implican un poco de conocimiento de arquitectura y más de programación, hasta la programación en C usando las librerías CMSIS de ARM o bien la que ocuparemos en este curso que es el ecosistema de desarrollo RUST.

Se ha seleccionado el ecosistema de RUST por ser este uno de los lenguajes con mayor crecimiento y demanda en un futuro cercano. Ello implica que debemos desarrollar un conocimiento del hardware y del lenguaje a lo largo del curso.

¡Bienvenidos y empecemos!

El autor

Casimiro Gómez González

Departamento de I+D, SMARTTEST

Capítulo Configuración del ecosistema Rust

Un microcontrolador es un sistema en un chip. Mientras que su computadora se compone de varios componentes discretos: un procesador, RAM, almacenamiento, un puerto Ethernet, etc.; un microcontrolador tiene todos esos tipos de componentes integrados en un solo “chip” o paquete. Esto hace posible construir sistemas con menos partes.

Los microcontroladores son la parte central de lo que se conoce como “sistemas embebidos”. Los sistemas embebidos están en todas partes, pero normalmente no los nota. Ellos controlan las máquinas que lavan tu ropa, imprimen tus documentos y cocinan tu comida. Los sistemas embebidos mantienen los edificios en los que vive y trabaja a una temperatura agradable y controlan los componentes que hacen que los vehículos en los que viaja se detengan y arranquen.

La mayoría de los sistemas embebidos funcionan sin la intervención del usuario. Incluso si exponen una interfaz de usuario como lo hace una lavadora; la mayor parte de su operación se realiza por su cuenta.

Los sistemas embebidos se utilizan a menudo para controlar un proceso físico. Para que esto sea posible, tienen uno o más dispositivos que les informan sobre el estado del mundo (“sensores”) y uno o más dispositivos que les permiten cambiar las cosas (“actuadores”). Por ejemplo, el sistema de control climático de un edificio podría tener:

- Sensores que miden la temperatura y la humedad en varios lugares.
- Actuadores que controlan la velocidad de los ventiladores.
- Actuadores que hacen que se agregue o se quite calor del edificio.

1.1 ¿Cuándo debo usar un microcontrolador?

Muchos de los sistemas integrados enumerados anteriormente podrían implementarse con una computadora que ejecute **Linux** (por ejemplo, una “Raspberry Pi”). ¿Por qué usar un microcontrolador en su lugar? Parece que podría ser más difícil desarrollar un programa.

Algunas razones pueden incluir:

- **Costo.** Un microcontrolador es mucho más económico que una computadora de propósito general. No solo el microcontrolador es más barato; también requiere muchos menos componentes eléctricos externos para funcionar. Esto hace que las placas de circuito impreso (**PCB**) sean más pequeñas y económicas de diseñar y fabricar.
- **El consumo de energía.** La mayoría de los microcontroladores consumen una fracción de la potencia de un procesador completo. Para aplicaciones que funcionan con baterías, eso marca una gran diferencia.
- **Sensibilidad.** Para lograr su propósito, algunos sistemas integrados siempre deben reaccionar dentro de un intervalo de tiempo limitado (por ejemplo, el sistema de frenado

“antibloqueo” de un automóvil). Si el sistema no cumple con este tipo de fecha límite, podría ocurrir una falla catastrófica. Tal fecha límite se denomina requisito de “tiempo real estricto”. Un sistema integrado que está sujeto a dicho plazo se denomina “sistema duro en tiempo real”. Una computadora de uso general y un sistema operativo generalmente tienen muchos componentes de software que comparten los recursos de procesamiento de la computadora. Esto hace que sea más difícil garantizar la ejecución de un programa dentro de las estrictas limitaciones de tiempo.

- **Fiabilidad.** En sistemas con menos componentes (tanto de hardware como de software), ¡hay menos cosas que pueden fallar!

1.2 ¿Cuándo no debo usar un microcontrolador?

Donde se involucran cálculos pesados. Para mantener bajo su consumo de energía, los microcontroladores tienen a su disposición recursos computacionales muy limitados. Por ejemplo, algunos microcontroladores ni siquiera tienen soporte de hardware para operaciones de punto flotante. En esos dispositivos, realizar una simple suma de números de precisión simple puede llevar cientos de ciclos de CPU.

1.2.1 ¿Por qué usar Rust y no C?

Con suerte, no necesito convencerlo aquí, ya que probablemente esté familiarizado con las diferencias de idioma entre Rust y C. Un punto que quiero mencionar es la administración de paquetes. C carece de una solución de gestión de paquetes oficial y ampliamente aceptada, mientras que Rust tiene Cargo. Esto hace que el desarrollo sea mucho más fácil. Y, en mi opinión, la fácil administración de paquetes fomenta la reutilización del código porque las bibliotecas se pueden integrar fácilmente en una aplicación, lo que también es bueno ya que las bibliotecas obtienen más “pruebas de batalla”.

1.2.2 ¿O por qué debería preferir C sobre Rust?

El ecosistema C es mucho más maduro. Ya existe una solución lista para usar para varios problemas. Si necesita controlar un proceso sensible al tiempo, puede tomar uno de los sistemas operativos en tiempo real (RTOS) comerciales existentes y resolver su problema. Todavía no hay RTOS comerciales de grado de producción en Rust, por lo que tendría que crear uno usted mismo o probar uno de los que están en desarrollo. Puede encontrar una lista de ellos en el repositorio *Awesome Embedded Rust*.

1.3 Descripción del Hardware

La placa STM32 Nucleo-64 proporciona una forma asequible y flexible para que los usuarios prueben nuevos conceptos y construyan prototipos eligiendo entre las diversas combinaciones de funciones de rendimiento y consumo de energía que proporciona el microcontrolador STM32. Para las placas compatibles, el *SMPS* externo reduce significativamente el consumo de energía en modo Run. El soporte de conectividad ARDUINO Uno V3 y los encabezados ST morpho permiten la fácil expansión de la funcionalidad de la plataforma de desarrollo abierta STM32 Nucleo con una amplia variedad de placas de expansión especializados.

La placa STM32 Nucleo-64 no requiere ninguna sonda separada ya que integra el depurador/programador ST-LINK.

La placa STM32 Nucleo-64 viene con las completas bibliotecas de software gratuito STM32 y ejemplos disponibles con el paquete STM32Cube MCU.

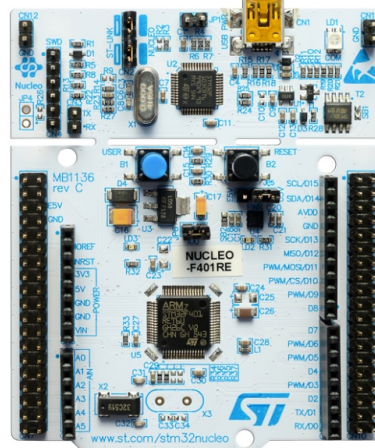


Figura 1.1: Tarjeta núcleo F411RE de ST

Los dispositivos STM32F411XC/XE se basan en el núcleo RISC de 32 bits Arm® Cortex®-M4 de alto rendimiento que funciona a una frecuencia de hasta 100 MHz. El núcleo Cortex®-M4 cuenta con una unidad de punto flotante (FPU) de precisión simple que admite todas las instrucciones de procesamiento de datos y tipos de datos de precisión simple de Arm. También implementa un conjunto completo de instrucciones DSP y una unidad de protección de memoria (MPU) que mejora la seguridad de la aplicación.

El STM32F411xC/xE pertenece a la línea de productos STM32 Dynamic Efficiency™ (con productos que combinan la eficiencia energética, el rendimiento y la integración) al tiempo que agrega una nueva característica innovadora llamada Modo de adquisición por lotes (BAM) que permite ahorrar aún más el consumo de energía durante el procesamiento por lotes de datos.

El STM32F411xC/xE incorpora memorias integradas de alta velocidad (hasta 512 Kbytes de memoria Flash, 128 Kbytes de SRAM) y una amplia gama de E/S y periféricos mejorados conectados a dos buses APB, dos buses AHB y un bus de 32 bits. matriz de bus multi-AHB. Todos

los dispositivos ofrecen un ADC de 12 bits, un RTC de bajo consumo, seis temporizadores de 16 bits de uso general, incluido un temporizador PWM para el control del motor, dos temporizadores de 32 bits de uso general. También cuentan con interfaces de comunicación estándar y avanzadas.

- Hasta tres I2C
- Cinco SPI
- Cinco I2S de los cuales dos son full dúplex. Para lograr una precisión de clase de audio, los periféricos I2S se pueden sincronizar mediante un PLL de audio interno dedicado o mediante un reloj externo para permitir la sincronización.
- Tres USART
- Interfaz SDIO
- Interfaz USB 2.0 OTG de máxima velocidad

El STM32F411xC/xE opera en el rango de temperatura de - 40 a + 125 °C desde una fuente de alimentación de 1,7 (PDR OFF) a 3,6 V. Un conjunto completo de modo de ahorro de energía permite el diseño de aplicaciones de bajo consumo.

Estas características hacen que los microcontroladores STM32F411xC/xE sean adecuados para una amplia gama de aplicaciones:

- Control de aplicaciones y accionamiento de motores
- Equipo medico
- Aplicaciones industriales: PLC, inversores, disyuntores
- Impresoras y escáneres
- Sistemas de alarma, video portero y climatización
- Electrodomésticos de audio para el hogar
- Concentrador de sensores de teléfonos móviles

1.3.1 Mapeo de Memoria

El mapa de memoria de procesador es necesario para determinar la región de donde se almacenarán los códigos de los programas. En la figura 1.3 se muestra el mapeo de memoria para este procesador.

1.4 Configurando el ambiente de desarrollo

Tratar con microcontroladores involucra varias herramientas, ya que estaremos tratando con una arquitectura diferente a la de su computadora y tendremos que ejecutar y depurar programas en un dispositivo “remoto”.

Usaremos todas las herramientas enumeradas a continuación. Cuando no se especifica una versión mínima, cualquier versión reciente debería funcionar, pero hemos enumerado la versión que hemos probado.

- **Rust** 1.53.0 o una nueva cadena de herramientas

Peripherals		STM32F411xC			STM32F411xE		
Flash memory in Kbytes		256			512		
SRAM in Kbytes	System	128					
Timers	General-purpose	7					
	Advanced-control	1					
Communication interfaces	SPI/ I ² S	5/5 (2 full duplex)					
	I ² C	3					
	USART	3					
	SDIO	1					
	USB OTG FS	1					
GPIOs		36	50	81	36	50	81
12-bit ADC		1					
Number of channels		10	16		10	16	
Maximum CPU frequency		100 MHz					
Operating voltage		1.7 to 3.6 V					
Operating temperatures		Ambient temperatures: - 40 to +85 °C / - 40 to + 105 °C/ - 40 to + 125 °C					
		Junction temperature: – 40 to + 130 °C					
Package		WLCSP49 UFQFPN48	LQFP64	UFBGA100 LQFP100	WLCSP49 UFQFPN48	LQFP64	UFBGA100 LQFP100

Figura 1.2: Características de la Tarjeta núcleo F411RE de ST

- **gdb-multiarch**. Versión probada: 10.2. Sin embargo, es probable que otras versiones también funcionen. Si su distribución/plataforma no tiene gdb multiarch disponible, arm-none-eabi-gdb también funcionará. Además, algunos binarios gdb normales también están contruidos con capacidades multiarquitectura, puede encontrar más información sobre esto en los subcapítulos.
- **cargo-binutils**. Versión 0.3.3 o más nueva.
- **cargo-embed**. Version 0.11.0 o más nueva.
- **minicom** en linux o mac.
- **PuTTY** en Windows.

1.4.1 Instalación de rustc y Cargo

Rustup es un instalador para el lenguaje de programación de sistemas Rust. Ejecute lo siguiente en su terminal, luego siga las instrucciones en pantalla.

```
1 $ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Después de ejecutar el comando anterior en una terminal obtenemos la respuesta que aparece en la figura 1.4.

De la cual elegimos **la opción 1**, después de lo cual nos instala Rust y nos modifica la variable path de nuestras terminales para poder ejecutar el comando rustc y cargo. Ahora verificamos la versión de rustc y cargo que se instalaron:

```
1 $ rustc -V
```

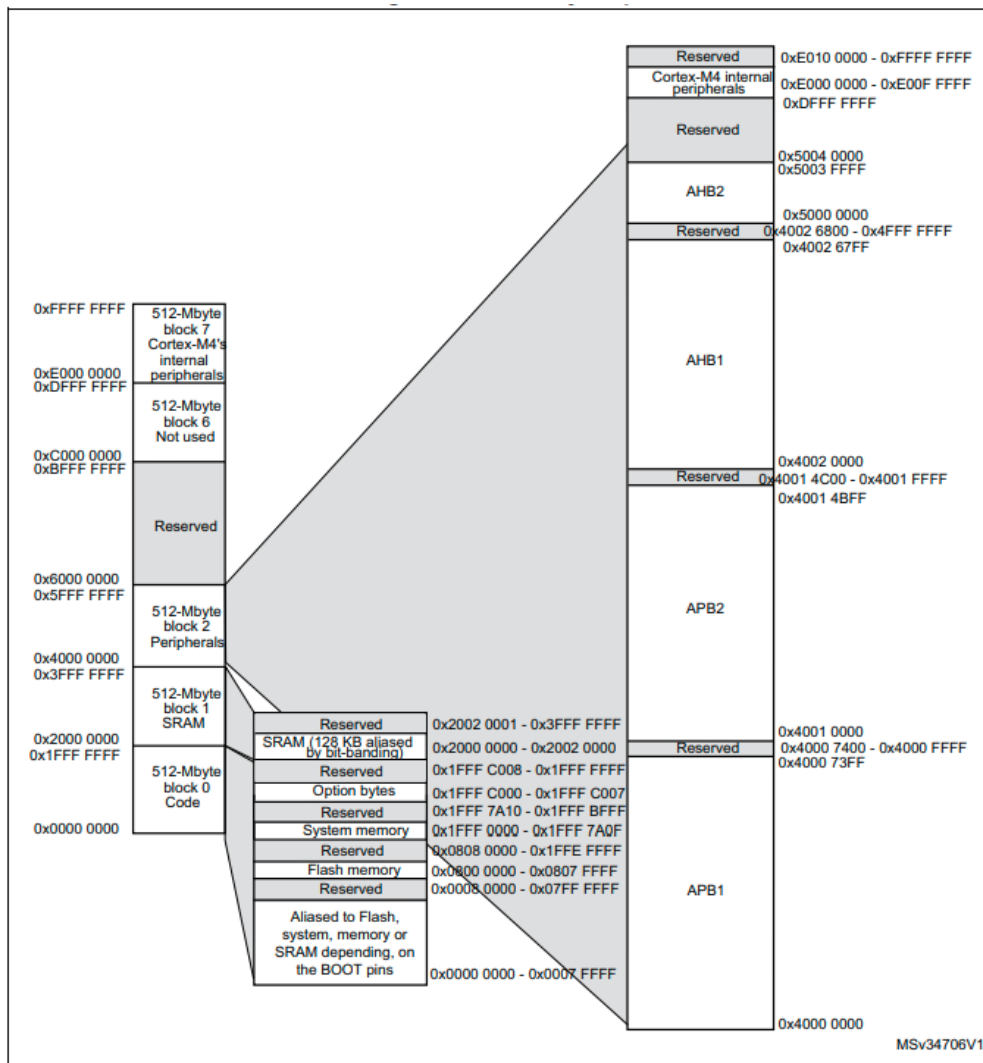



Figura 1.3: Mapa de memoria de la Tarjeta núcleo F411RE de ST

```

2 rustc 1.57.0 (f1edd0429 2021-11-29)
3 $ cargo -V
4 cargo 1.57.0 (b2e52d7ca 2021-10-21)

```

1.4.2 Instalación de cargo-binutils

```

1 $ rustup component add llvm-tools-preview
2
3 $ cargo install cargo-binutils
4
5 $ cargo-size --version

```

1.4.3 cargo-embed

```

1 $ cargo install cargo-embed
2

```

```

Current installation options:

  default host triple: x86_64-unknown-linux-gnu
  default toolchain:  stable (default)
                    profile: default
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>

```

Figura 1.4: Opciones de instalación de Rust

```
3 $ cargo embed --version
```

Si marca error el primer comando es porque el sistema no tiene instalado la librería *libudev*

```
1 $ sudo apt update
2 $ sudo apt-get install -y libudev-dev
```

Después de instalar la librería **libudev** ya es posible instalar **cargo-embed**.

1.4.4 Instalación en linux

Estos son los comandos de instalación para algunas distribuciones de Linux.

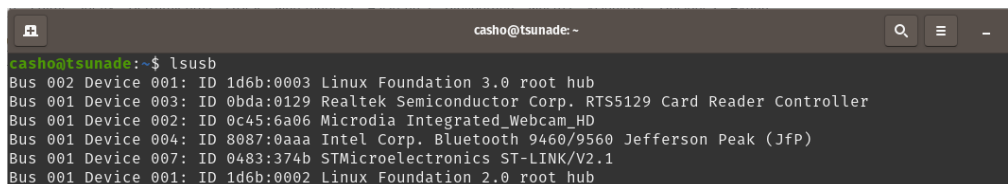
```
1 $ sudo apt-get install gdb-multiarch minicom
```

1.4.4.1 reglas udev

Para establecer las reglas udev y que linux detecte la tarjeta, **debemos primero conectar la tarjeta** y posteriormente ejecutar el siguiente comando:

```
1 $ lsusb
```

Lo cual produce la salida que se muestra en la figura 1.5



```

casho@tsunade: ~
casho@tsunade:~$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID 0bda:0129 Realtek Semiconductor Corp. RTS5129 Card Reader Controller
Bus 001 Device 002: ID 0c45:6a06 Microdia Integrated Webcam HD
Bus 001 Device 004: ID 8087:0aaa Intel Corp. Bluetooth 9460/9560 Jefferson Peak (JfP)
Bus 001 Device 007: ID 0483:374b STMicroelectronics ST-LINK/V2.1
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub

```

Figura 1.5: Salida del comando lsusb

De la figura 1.5 podemos ver los datos

```
1 Bus 001 Device 007: ID 0483:374b STMicroelectronics ST-LINK/V2.1
```

De donde los datos del ID corresponden a: **idVendor:idProduct**. Que utilizaremos para la creación de las reglas udev.

Estas reglas le permiten usar dispositivos USB como ST-LINK sin privilegios de root, es decir, sudo. Cree un archivo en directorio **/etc/udev/rules.d** con el contenido que se muestra a continuación.

```
1 $ sudo nano /etc/udev/rules.d/99-stlink.rules
```

El contenido del archivo debe ser:

```
1 # CMSIS-DAP for stlink
2 SUBSYSTEM=="usb", ATTR{idVendor}=="0483", ATTR{idProduct}=="374b", MODE:="666"
```

Luego recarga las reglas de udev con:

```
1 $ sudo udevadm control --reload-rules
```

Si tenía alguna placa conectada a su computadora, desenchúfela y luego vuelva a enchufarla.

1.5 Instalación en windows

ARM proporciona instaladores .exe para Windows. Tome uno de aquí (<https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads>) y siga las instrucciones. Justo antes de que finalice el proceso de instalación, marque/seleccione la opción “Agregar ruta a la variable de entorno”. Luego verifique que las herramientas estén en su **PATH**:

```
1 $ arm-none-eabi-gcc -v
```

1.5.1 Instalación de PuTTY

Descargue el último **putty.exe** de este sitio(<https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>) y colóquelo en algún lugar de su **PATH**.

1.6 Instalación de MacOs

Todas las herramientas se pueden instalar usando Homebrew:

```
1 $ brew tap ArmMbed/homebrew-formulae
2 $ brew install arm-none-eabi-gcc
3 $ brew install minicom
```

1.7 Verificando cargo-embed

Primero, conecte la tarjeta a su computadora usando un cable USB.

1.7.1 Paso 1: Cree un proyecto

Primero creamos un nuevo proyecto:

```
1 $ cargo new ejemplo1
2 $ cd ejemplo1
3 $ code .
```

A continuación, deberá crear el archivo **Embed.toml** en el directorio donde se encuentre el archivo **Cargo.toml**. En la sección **default.general** encontrará dos variantes de chips comentadas:

```
1 # Archivo Embed.toml
2 [default.probe]
3 protocol = "Swd"
4
5 [default.general]
6 # chip = "nrf52833_xxaa" # uncomment this line for micro:bit V2
7 # chip = "nrf51822_xxaa" # uncomment this line for micro:bit V1
8
9 [default.rtt]
10 enabled = true
11
12 [default.gdb]
13 enabled = false
```

Ahora se modifica el archivo **Cargo.toml**

```
1 # Archivo Cargo.toml
2 [package]
3 name = "ejemplo1"
4 version = "0.1.0"
5 edition = "2021"
6
7 [dependencies]
8 cortex-m = "0.7.3"
9 cortex-m-rt = "0.7.0"
10 rtt-target = { version = "0.3.1", features = ["cortex-m"] }
11 panic-rtt-target = { version = "0.1.2", features = ["cortex-m"] }
```

Ahora se crea un archivo **memory.x** donde se indica la distribución de memoria, este archivo debe estar en el directorio donde se encuentre el archivo **Cargo.toml**.

```
1 # archivo de memory.x
2 MEMORY
3 {
4     /* NOTE K = KiBi = 1024 bytes */
5     FLASH : ORIGIN = 0x00000000, LENGTH = 512K
6     RAM : ORIGIN = 0x20000000, LENGTH = 128K
7 }
```

Ahora modificamos el archivo **src/main.rs**

```

1 # archivo de src/main.rs
2 #![no_std]
3 #![no_main]
4
5 use panic_rtt_target as _;
6 use rtt_target::{rtt_init_print, rprintln};
7
8 use cortex_m_rt::entry;
9
10 #[entry]
11 fn main() -> ! {
12     rtt_init_print!();
13     rprintln!("Hola_Mundo");
14     loop {}
15 }

```

Ahora en el raíz creamos un directorio llamado **.cargo**, en donde dentro de este creamos el archivo **config**, con el siguiente contenido:

```

1 [target.'cfg(all(target_arch="arm", target_os="none"))']
2 rustflags = [
3 "-C", "link-arg=-Tlink.x",
4 ]

```

Ahora instalamos la compilación cruzada:

```

1 $ rustup target add thumbv7em-none-eabihf
2 $ cargo embed --target thumbv7em-none-eabihf

```

Después de lo cual debemos observar “Hola Mundo” en la terminal.

Capítulo Mapa de Memoria

Los dispositivos Cortex-M requieren que una tabla de vectores esté presente al comienzo de su región de memoria de código. La tabla de vectores es una matriz de punteros; los primeros dos punteros son necesarios para iniciar el dispositivo, el resto de los punteros están relacionados con excepciones. Los ignoraremos por ahora.

Los enlazadores deciden el diseño final de la memoria de los programas, pero podemos usar scripts de enlazadores para tener cierto control sobre ellos. La granularidad de control que nos dan los scripts de linker sobre el layout es a nivel de secciones. Una sección es una colección de símbolos dispuestos en memoria contigua. Los símbolos, a su vez, pueden ser datos (una variable estática) o instrucciones (una función de Rust).

Cada símbolo tiene un nombre asignado por el compilador. A partir de Rust 1.28, los nombres que el compilador de Rust asigna a los símbolos son de la forma: `_ZN5krate6module8function17he1dfc17c86fe16daE`, que desarma a `krate::module::function::he1dfc17c86fe16da` donde **krate::module::function** es la ruta de la función o variable y **he1dfc17c86fe16da** es una especie de hash. El compilador de Rust colocará cada símbolo en su propia sección única; por ejemplo, el símbolo mencionado anteriormente se colocará en una sección denominada `.text._ZN5krate6module8function17he1dfc17c86fe16daE`.

No se garantiza que estos símbolos y nombres de sección generados por el compilador permanezcan constantes en diferentes versiones del compilador de Rust. Sin embargo, el lenguaje nos permite controlar los nombres de los símbolos y la ubicación de las secciones a través de estos atributos:

- `#[export_name = "foo"]` establece el nombre del símbolo en **foo**.
- `#[no_mangle]` significa: use el nombre de la función o variable (no su ruta completa) como su nombre de símbolo.
- `#[no_mangle] fn bar()` producirá un símbolo llamado **bar**.
- `#[link_section = ".bar"]` coloca el símbolo en una sección llamada `.bar`.

2.1 El lado de Rust

Como se mencionó anteriormente, para los dispositivos Cortex-M, debemos completar las dos primeras entradas de la tabla de vectores. El primero, el valor inicial para el puntero de la pila, se puede completar usando solo la secuencia de comandos del enlazador. El segundo, el vector de reinicio, debe crearse en código Rust y colocarse correctamente mediante el script del enlazador.

El vector de reinicio es un puntero al controlador de reinicio. El controlador de restablecimiento es la función que ejecutará el dispositivo después de un restablecimiento del sistema o después de que se encienda por primera vez. El controlador de reinicio siempre es el primer

marco de pila en la pila de llamadas de hardware; regresar de él es un comportamiento indefinido ya que no hay otro marco de pila al que regresar. Podemos hacer cumplir que el controlador de reinicio nunca regrese convirtiéndolo en una función divergente, que es una función con la firma `fn(* .. *) -> !`.

Con estos atributos, podemos exponer un ABI estable del programa y usarlo en el script del enlazador.

```

1 #[no_mangle]
2 pub unsafe extern "C" fn Reset() -> ! {
3     let _x = 42;
4
5     // can't return so we go into an infinite loop here
6     loop {}
7 }
8
9 // The reset vector, a pointer into the reset handler
10 #[link_section = ".vector_table.reset_vector"]
11 #[no_mangle]
12 pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;

```

El hardware espera un determinado formato aquí, al que nos adherimos usando `extern C` para decirle al compilador que reduzca la función usando C ABI, en lugar de Rust ABI, que es inestable.

Para hacer referencia al controlador de reinicio y al vector de reinicio desde la secuencia de comandos del enlazador, necesitamos que tengan un nombre de símbolo estable, por lo que usamos `#[no_mangle]`. Necesitamos un control preciso sobre la ubicación de `RESET_VECTOR`, por lo que lo colocamos en una sección conocida, `.vector_table.reset_vector`. La ubicación exacta del propio controlador de restablecimiento, Restablecer, no es importante. Nos limitamos a la sección generada por el compilador predeterminado.

El enlazador ignorará los símbolos con enlace interno (también conocidos como símbolos internos) mientras recorre la lista de archivos de objetos de entrada, por lo que necesitamos que nuestros dos símbolos tengan enlace externo. La única forma de hacer que un símbolo sea externo en Rust es hacer que su elemento correspondiente sea público (`pub`) y accesible (sin un módulo privado entre el elemento y la raíz de la caja).

2.2 El lado del script del enlazador

A continuación se muestra un script de enlace mínimo que coloca la tabla de vectores en la ubicación correcta. Recorrámoslo.

```

1 /* Memory layout of the STM32F411RE microcontroller */
2 /* 1K = 1 KiBi = 1024 bytes */
3 MEMORY

```



```

4 {
5     FLASH : ORIGIN = 0x08000000, LENGTH = 512K
6     RAM : ORIGIN = 0x20000000, LENGTH = 128K
7 }
8
9 /* The entry point is the reset handler */
10 ENTRY(Reset);
11
12 EXTERN(RESET_VECTOR);
13
14 SECTIONS
15 {
16     .vector_table ORIGIN(FLASH) :
17     {
18         /* First entry: initial Stack Pointer value */
19         LONG(ORIGIN(RAM) + LENGTH(RAM));
20
21         /* Second entry: reset vector */
22         KEEP(*(.vector_table.reset_vector));
23     } > FLASH
24
25     .text :
26     {
27         *(.text .text.*);
28     } > FLASH
29
30     /DISCARD/ :
31     {
32         *(.ARM.exidx .ARM.exidx.*);
33     }
34 }

```

2.2.1 MEMORIA

Esta sección del script del enlazador describe la ubicación y el tamaño de los bloques de memoria en el destino. Se definen dos bloques de memoria: FLASH y RAM; corresponden a la memoria física disponible en el destino. Los valores utilizados aquí corresponden al microcontrolador STM32F411RE.

2.2.2 ENTRADA

Aquí le indicamos al enlazador que el controlador de reinicio, cuyo nombre de símbolo es Reiniciar, es el punto de entrada del programa. Los enlazadores descartan agresivamente las secciones no utilizadas. Los enlazadores consideran el punto de entrada y las funciones llamadas desde él como usados, por lo que no los descartarán. Sin esta línea, el enlazador descartaría la

función Restablecer y todas las funciones subsiguientes llamadas desde ella.

2.2.3 EXTERNO

Los enlazadores son perezosos; dejarán de buscar en los archivos de objetos de entrada una vez que hayan encontrado todos los símbolos a los que se hace referencia recursivamente desde el punto de entrada. **EXTERN** obliga al enlazador a buscar el argumento de **EXTERN** incluso después de que se hayan encontrado todos los demás símbolos a los que se hace referencia. Como regla general, si necesita que un símbolo que no se llama desde el punto de entrada esté siempre presente en el binario de salida, debe usar **EXTERN** junto con **KEEP**.

2.2.4 SECCIONES

Esta parte describe cómo se organizan las secciones de los archivos de objetos de entrada (también conocidas como secciones de entrada) en las secciones del archivo de objetos de salida (también conocidas como secciones de salida) o si se deben descartar. Aquí definimos dos secciones de salida:

```
1 .vector_table ORIGIN(FLASH) : { /* .. */ } > FLASH
```

.vector_table contiene la tabla de vectores y se encuentra al comienzo de la memoria **FLASH**.

```
1 .text : { /* .. */ } > FLASH
```

.text contiene las subrutinas del programa y se encuentra en algún lugar de **FLASH**. No se especifica su dirección de inicio, pero el enlazador la colocará después de la sección de salida anterior, **.vector_table**.

La sección de salida de la tabla de vectores **.vector_table** contiene:

```
1 /* First entry: initial Stack Pointer value */
2 LONG(ORIGIN(RAM) + LENGTH(RAM));
```

Colocaremos la pila (llamada) al final de la RAM (la pila desciende por completo; crece hacia direcciones más pequeñas) para que la dirección final de la RAM se use como el valor inicial del puntero de pila (SP). Esa dirección se calcula en el propio script del enlazador usando la información que ingresamos para el bloque de memoria RAM.

```
1 /* Second entry: reset vector */
2 KEEP*(.vector_table.reset_vector);
```

A continuación, usamos **KEEP** para obligar al enlazador a insertar todas las secciones de entrada denominadas **.vector_table.reset_vector** justo después del valor SP inicial. El único símbolo ubicado en esa sección es **RESET_VECTOR**, por lo que esto colocará efectivamente a **RESET_VECTOR** en segundo lugar en la tabla de vectores.

La sección de salida **.text** contiene:

```
1 *.text .text.*);
```

Esto incluye todas las secciones de entrada denominadas `.text` y `.text.*`. Tenga en cuenta que no usamos `KEEP` aquí para permitir que el enlazador descarte las secciones no utilizadas.

Finalmente, usamos la sección especial `/DESCARTAR/` para descartar

```
1 *.ARM.exidx .ARM.exidx.*);
```

secciones de entrada denominadas `.ARM.exidx.*`. Estas secciones están relacionadas con el manejo de excepciones, pero no estamos deshaciendo la pila en caso de pánico y ocupan espacio en la memoria Flash, por lo que simplemente las descartamos.

2.3 Uniendo todo

Ahora podemos vincular la aplicación. Como referencia, aquí está el programa Rust completo:

```
1 #![no_main]
2 #![no_std]
3
4 use core::panic::PanicInfo;
5
6 // The reset handler
7 #[no_mangle]
8 pub unsafe extern "C" fn Reset() -> ! {
9     let _x = 42;
10
11     // can't return so we go into an infinite loop here
12     loop {}
13 }
14
15 // The reset vector, a pointer into the reset handler
16 #[link_section = ".vector_table.reset_vector"]
17 #[no_mangle]
18 pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;
19
20 #[panic_handler]
21 fn panic(_panic: &PanicInfo<'_> _-> ! {
22     loop {}
23 }
```

Tenemos que modificar el proceso del enlazador para que use nuestro script del enlazador. Esto se hace pasando el indicador `-C link-arg` a `rustc`. Esto se puede hacer con `cargo-rustc` o `cargo-build`.

IMPORTANTE: asegúrese de tener el archivo `.cargo/config` que se agregó al final de la última sección antes de ejecutar este comando.

Usando el subcomando `cargo-rustc`:

```
1 cargo rustc -- -C link-arg=-Tlink.x
```

O puede configurar los `rustflags` en `.cargo/config` y continuar usando el subcomando `cargo-build`. Haremos lo último porque se integra mejor con `cargo-binutils`.

```
1 # modify .cargo/config so it has these contents
2 cat .cargo/config
```

quedando

```
1 [target.thumbv7m-none-eabi]
2 rustflags = ["-C", "link-arg=-Tlink.x"]
3
4 [build]
5 target = "thumbv7m-none-eabi"
```

La parte `[target.thumbv7m-none-eabi]` dice que estas banderas solo se usarán cuando se realice una compilación cruzada para ese objetivo.

2.4 revisando

Ahora, inspeccionemos el binario de salida para confirmar que el diseño de la memoria se vea como queremos (esto requiere `cargo-binutils`):

```
1 cargo objdump --bin app -- -d -no-show-raw-insn
```

Este es el desmontaje de la sección `.text`. Vemos que el controlador de reinicio, llamado Restablecer, se encuentra en la dirección `0x8`.

```
1 cargo objdump --bin app -- -s -section .vector_table
```

Esto muestra el contenido de la sección `.vector_table`. Podemos ver que la sección comienza en la dirección `0x0` y que la primera palabra de la sección es `0x2001_0000` (la salida de `objdump` está en formato little endian). Este es el valor SP inicial y coincide con la dirección final de la RAM. La segunda palabra es `0x9`; esta es la dirección del modo pulgar del controlador de reinicio. Cuando se va a ejecutar una función en modo pulgar, el primer bit de su dirección se establece en 1.

Capítulo Control de GPIO

stm32f4xx-hal contiene una abstracción de hardware multidispositivo además de la API de acceso periférico para los microcontroladores de la serie STMicro STM32F4. La selección de la MCU se realiza mediante puertas de características, generalmente especificadas por cajas de soporte de placa. Las configuraciones admitidas actualmente son:

Microcontrolador			
stm32f410	stm32f401	stm32f405	stm32f407
stm32f411	stm32f412	stm32f413	stm32f415
stm32f417	stm32f423	stm32f427	stm32f429
stm32f437	stm32f439	stm32f446	stm32f469
stm32f479			

La idea detrás de esta caja es pasar por alto las ligeras diferencias en los diversos periféricos disponibles en esos MCU para que se pueda escribir un HAL para todos los chips de esa misma familia sin tener que cortar y pegar cajas para cada modelo.

¡La colaboración en esta caja es muy bienvenida, al igual que las solicitudes de extracción!

Esta caja se basa en la fantástica **crate stm32f4** de **Adam Greig** para proporcionar definiciones de registro apropiadas e implementa un conjunto parcial de rasgos incrustados.

Parte de la implementación se adaptó descaradamente de la **crate stm32f1xx-hal** iniciada originalmente por **Jorge Aparicio**.

3.1 Configurando el proyecto

Compruebe si existe el **BSP** para su placa en la tabla del microcontrolador. Si existe, la **crate stm32f4xx-hal** ya debería estar incluida, por lo que puede usar el **bsp** como **BSP** para su proyecto.

De lo contrario, cree un nuevo proyecto de Rust como lo hace normalmente con **cargo init**. El "hola mundo" del desarrollo integrado generalmente consiste en hacer parpadear un LED. El código para hacerlo está disponible en ejemplos **/delay-syst-blinky.rs**. Copie ese archivo en **main.rs** de su proyecto.

También necesita agregar algunas dependencias a su **Cargo.toml**:

```
1 [dependencies]
2 embedded-hal = "0.2"
3 nb = "1"
4 cortex-m = "0.7.3"
5 cortex-m-rt = "0.7"
6 # Panic behaviour, see https://crates.io/keywords/panic-impl for alternatives
7 panic-halt = "0.2"
```

```

8
9 [dependencies.stm32f4xx-hal]
10 version = "0.10"
11 features = ["rt", "stm32f411"] # replace the model of your microcontroller here

```

También debemos decirle a Rust cómo vincular nuestro ejecutable y cómo distribuir el resultado en la memoria. Para lograr todo esto, copie **.cargo/config** y **memory.x** del repositorio **stm32f4xx-hal** a su proyecto y asegúrese de que los tamaños coincidan con la hoja de datos. También tenga en cuenta que puede haber diferentes tipos de memoria que no son iguales; para estar seguro, solo especifique el tamaño del primer bloque en la dirección especificada.

3.2 Encendiendo un LED en la tarjeta STM32F411RE

Para controlar el GPIO de la tarjeta stm32f411re primero debemos configurar el archivo **Cargo.toml**

```

1 #Archivo Cargo.toml
2 [package]
3 name = "prender_led2"
4 version = "0.1.0"
5 edition = "2021"
6
7 [dependencies]
8 embedded-hal = "0.2"
9 nb = "1"
10 cortex-m = "^0.7"
11 cortex-m-rt = "^0.6"
12 rtt-target = { version = "0.3.1", features = ["cortex-m"] }
13 panic-halt = "0.2"
14
15 [dependencies.stm32f4xx-hal]
16 version = "0.10"
17 features = ["rt", "stm32f411"]

```

Ahora es necesario crear el **Embed.toml** en el mismo directorio donde se encuentra el directorio **Cargo.toml**.

```

1 # Archivo Embed.toml
2 [default.probe]
3 protocol = "Swd"
4
5 [default.general]
6 chip = "STM32F411RETx" # uncomment this line for micro:bit V2
7 # chip = "nrf51822_xxAA" # uncomment this line for micro:bit V1
8
9 [default.rtt]

```

```

10 enabled = true
11
12 [default.gdb]
13 enabled = false

```

En el mismo directorio también se crea un archivo llamado **memory.x** en donde se escribe la distribución de memoria del microcontrolador stm32f411re

```

1 MEMORY
2 {
3     /* NOTE K = KiBi = 1024 bytes */
4     FLASH : ORIGIN = 0x08000000, LENGTH = 512K
5     RAM : ORIGIN = 0x20000000, LENGTH = 128K
6 }
7 /* This is where the call stack will be allocated. */
8 /* The stack is of the full descending type. */
9 /* NOTE Do NOT modify '_stack_start' unless you know what you are doing */
10 _stack_start = ORIGIN(RAM) + LENGTH(RAM);

```

El siguiente archivo para crear **build.rs** para procesar el archivo de memoria **memory.x**.

```

1 // Archivo build.rs
2 use std::env;
3 use std::fs::File;
4 use std::io::Write;
5 use std::path::PathBuf;
6
7 fn main() {
8     // Put 'memory.x' in our output directory and ensure it's
9     // on the linker search path.
10    let out = &PathBuf::from(env::var_os("OUT_DIR").unwrap());
11    File::create(out.join("memory.x"))
12        .unwrap()
13        .write_all(include_bytes!("memory.x"))
14        .unwrap();
15    println!("cargo:rustc-link-search={}", out.display());
16
17    // By default, Cargo will re-run a build script whenever
18    // any file in the project changes. By specifying 'memory.x'
19    // here, we ensure the build script is only re-run when
20    // 'memory.x' is changed.
21    println!("cargo:rerun-if-changed=memory.x");
22 }

```

Ahora se crea un directorio llamado **.cargo** en donde creamos un archivo llamado **config**, dicho archivo nos permite compilar y enlazar nuestro proyecto

```

1 [target.thumbv7em-none-eabihf]
2 runner = 'probe-run -c chip_STM32F411CEUx'

```



```

3 rustflags = [
4 "-C", "link-arg=-Tlink.x",
5 ]
6
7 [build]
8 target = "thumbv7em-none-eabi"

```

Ahora se modifica el archivo **src/main.rs**

```

1 // ***** Archivo src/main.rs *****
2 #![deny(unsafe_code)]
3 #![allow(clippy::empty_loop)]
4 #![no_main]
5 #![no_std]
6
7 // Halt on panic
8 use panic_halt as _; // Manejador de panic
9
10 use cortex_m_rt::entry;
11 use stm32f4xx_hal as hal;
12
13 use crate::hal::{pac, prelude::*};
14 use rtt_target::{rtt_init_print, rprintln};
15
16
17 #[entry]
18 fn main() -> ! {
19     rtt_init_print!();
20     rprintln!("Prendiendo Led Cada Segundo");
21     if let (Some(dp), Some(cp)) = (
22         pac::Peripherals::take(),
23         cortex_m::peripheral::Peripherals::take(),
24     ) {
25         // Configuracion del led de la tarjeta stm32f411
26         let gpioa = dp.GPIOA.split();
27         let mut led = gpioa.pa5.into_push_pull_output();
28
29         // configuracion del reloj de la tarjeta a 48Mhz
30         let rcc = dp.RCC.constrain();
31         let clocks = rcc.cfgr.sysclk(48.mhz()).freeze();
32
33         // Creacion del retardo con SysTick
34         let mut delay = hal::delay::Delay::new(cp.SYST, &clocks);
35         loop {
36             // On for 1s, off for 1s.
37             led.set_high();
38             delay.delay_ms(2000_u32);
39             led.set_low();

```

```

40     delay.delay_ms(1000_u32);
41 }
42
43 }
44 loop{}
45 }

```

La estructura del proyecto se muestra en la figura 3.1

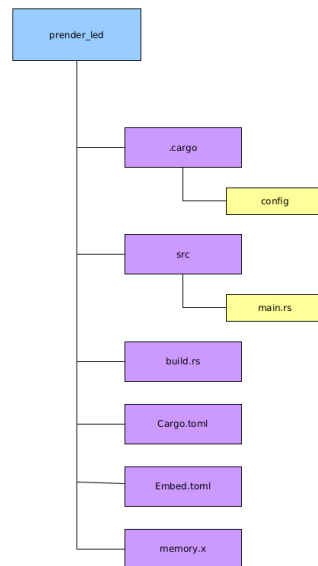


Figura 3.1: Estructura general del proyecto para controlar GPIO

Ahora si ejecutamos el comando:

```
1 cargo embed --target thumbv7em-none-eabihf
```

La salida obtenida por dicho comando se muestra en la figura

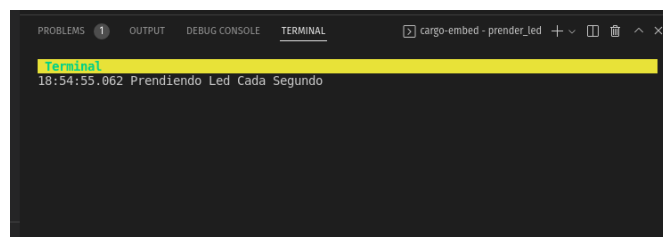
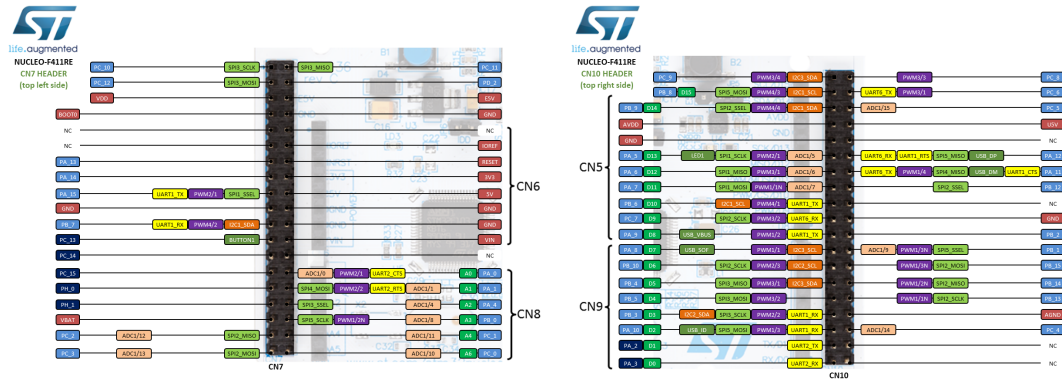


Figura 3.2: Salida en la terminal al ejecutar el proyecto

3.3 Controlando un bus de GPIO

Para controlar los GPIO del microcontrolador necesitamos conocer los nombres de estos pines

De la figura 3.3 vamos a tomar los pines del lado derecho para la creación del bus. Los pines



(a) Izquierdo

(b) Derecho

Figura 3.3: Pins de la tarjeta núcleo F411RE

van a ser (PA_5, PA_6, PA_7, PA_9) y llamaremos a este BUS BCD ¹.

¹Los pines se pueden observar claramente en la figura 3.3b