

# LSE004-2015: Programando en el lenguaje RUST

Dr. Casimiro Gómez González  
Facultad de Electrónica, UPAEP  
correo: [casimiro.gomez@upaep.mx](mailto:casimiro.gomez@upaep.mx)  
Tel: 222 229 9428

Primavera 2015

# Prólogo

El presente texto es una introducción a la programación en RUST, incluye las herramientas necesarias para el desarrollo de proyectos de nivel intermedio en este excitante lenguaje y sus aplicaciones en el desarrollo de sistemas embebidos.

El autor  
Casimiro Gómez González  
Doctor en Ingeniería Mecatrónica  
correo: [casimiro.gomez@upaep.mx](mailto:casimiro.gomez@upaep.mx)

# Índice general

<b>1. Instalación de RUST</b>	<b>5</b>
1.1. Instalando RUST en Linux . . . . .	6
1.2. Instalando Rust Nightly . . . . .	6
1.2.1. Desinstalando . . . . .	6
1.3. Instalación de RustDT . . . . .	6
1.3.1. Instrucciones . . . . .	7
1.3.2. Actualizar . . . . .	7
<b>2. Características Principales de RUST</b>	<b>8</b>
2.1. Una corta introducción a RUST . . . . .	8
2.2. Programa “Hola, Mundo” . . . . .	11
2.3. Conviertiendo a Cargo . . . . .	12
2.3.1. Primer paso . . . . .	13
<b>3. Programación embebida usando la toolchain GNU</b>	<b>14</b>
3.1. Programación Ensamblador para ARM . . . . .	14
3.1.1. Sumador . . . . .	14
3.1.2. Sumar datos en Memoria . . . . .	15
<b>4. Ensamblador en linux</b>	<b>18</b>
4.1. Desplegando informacion en linux con ensamblador . . . . .	18
4.1.1. Hola Mundo en ensamblador . . . . .	19
4.1.2. Enviando resultados a la salida estandar . . . . .	20
4.2. Usando gdb . . . . .	22
4.2.1. El comando Print del depurador . . . . .	23
<b>5. Diseñando el Sistema Operativo <i>UtVitam</i> con Rust</b>	<b>25</b>
5.1. Construyendo un <b>Cross-Compiler</b> . . . . .	25

5.1.1. Anfitrión y Objetivos . . . . .	26
5.1.2. Raspberry Pi bootloader . . . . .	27
5.2. Comenzando la programación . . . . .	27
5.3. Haciendo el LED parpadear . . . . .	31
5.3.1. Preparando el código . . . . .	32
5.4. Tips de compilación y ensamblador . . . . .	33
<b>6. Sistema Operativo para x86</b>	<b>38</b>

# Capítulo 1

## Instalación de RUST

Rust es un lenguaje de programación compilado, de propósito general y multiparadigma que está siendo desarrollado por Mozilla. Ha sido diseñado para ser "un lenguaje seguro, concurrente y práctico", soportando programación funcional pura, por procedimientos, imperativa y orientada a objetos.

El lenguaje surgió de un proyecto personal desarrollado por Graydon Hoare (trabajador de Mozilla), quién empezó a trabajar en él en 2006; y Mozilla se involucró en este proyecto en 2009 y lo dio a conocer oficialmente en 2010. El mismo año, el trabajo pasó del compilador inicial (escrito en OCaml) al compilador auto contenido, escrito en sí mismo, en Rust. Conocido como `rustc`, se compiló a sí mismo en 2011. El compilador auto contenido usa LLVM como su back-end.

La primera versión alfa numerada del compilador de Rust apareció en enero de 2012. Según la política de Mozilla, Rust es desarrollado de forma totalmente abierta y solicita la opinión y contribución de la comunidad. El diseño del lenguaje se ha ido perfeccionando a través de las experiencias en el desarrollo del motor de navegador, Servo, y el propio compilador de Rust. Aunque es desarrollado y patrocinado por Mozilla y Samsung, es un proyecto comunitario. Una gran parte de las contribuciones proceden de los miembros de la comunidad.

El objetivo de Rust es ser un buen lenguaje para la creación de grandes programas del lado del cliente y del servidor que se ejecuten en Internet. Esto ha llevado a un conjunto de características con un énfasis en la seguridad, el control de distribución de la memoria y la concurrencia. Se espera que el rendimiento de código seguro sea más lento que C++, si el rendimiento es la única consideración, pero si lo comparamos con el código C++ hecho para

que tome precauciones comparables a las que toma Rust, este último puede ser incluso más rápido

## 1.1. Instalando RUST en Linux

El primer paso para usar Rust es instalarlo. Hay distintas formas de instalar Rust, pero la forma más fácil es usar el script **rustup**. Se necesita ejecutar el siguiente comando:

```
1 $ curl -sf -L https://static.rust-lang.org/rustup.sh | sudo sh
```

También es posible instalarlo en dos pasos:

```
1 $ curl -f -L https://static.rust-lang.org/rustup.sh -O
2 $ sudo sh rustup.sh
```

## 1.2. Instalando Rust Nightly

Se instala Rust Nightly con el siguiente comando:

```
1 curl -sSf https://static.rust-lang.org/rustup.sh | sudo sh -s -- --channel=nightly --yes
```

### 1.2.1. Desinstalando

Si se desea desinstalar Rust se debe ejecutar el siguiente comando:

```
1 $ sudo /usr/local/lib/rustlib/uninstall.sh
```

## 1.3. Instalación de RustDT

Para la instalación de RustDT se necesita tener instalado lo siguiente:

- Eclipse 4.5 (Mars) or later.
- Java VM version 8 or later.
- The Racer tool.

### 1.3.1. Instrucciones

- Usa el Eclipse que tengas instalado en tu sistema o baja el paquete de <http://www.eclipse.org/downloads/><sup>1</sup> debes bajar la "*Platform Runtime Binary*".
- Iniciar Eclipse, e ir a **HELP ->Install New Software...**
- Pulsa el botón **Add...** para agregar un nuevo sitio, introduce el URL: **<http://rustdt.github.io/releases/>** en el campo de localización, pulsa **OK**.
- Selecciona el sitio de las actualizaciones recientes en **work with:** menu. Selecciona **RustDT** en la caja de opciones. Ahora la característica de **RustDT** aparecerá en la parte inferior
- Selecciona la característica **RustDT** y completa el wizard.<sup>2</sup>
- Reiniciar Eclipse
- Para la configuración inicial sigue las instrucciones en la sección de configuración en la guía de usuario.

### 1.3.2. Actualizar

Si tienes instalado RustDT y quieres actualizar pulsa **Help / Check for Updates...**

---

<sup>1</sup>Para un paquete de eclipse sin ningún otro IDE o extra (tal como la herramienta VCS)

<sup>2</sup>Las dependencias de **RustDT** tales como CDT automáticamente se instalarán durante el proceso de la instalación

## Capítulo 2

# Características Principales de RUST

RUST es un lenguaje de programación de sistemas enfocado en tres objetivos: Seguridad, velocidad y concurrencia. Se logran estos tres objetivos sin tener un colector de basura, provocando esto que sea un lenguaje útil para aplicaciones donde otros lenguajes no lo son: Ser embebidos en otros lenguajes, programas con datos específicos de tiempo y espacio, escritura de código de bajo nivel, tales como drivers de dispositivos y sistemas operativos. Mejora de los lenguajes actuales dirigidas a este espacio por tener una serie de controles de seguridad en tiempo de compilación que no producen sobrecarga de tiempo de ejecución, mientras que la eliminación de todas las carreras de datos. Rust también tiene como objetivo lograr “abstracciones de costo cero” a pesar de que algunas de estas abstracciones sienten como los de un lenguaje de alto nivel. Incluso entonces, Rust todavía permite un control preciso como un lenguaje de bajo nivel lo haría.

### 2.1. Una corta introducción a RUST

EL principal concepto que hace a Rust único es llamado "propiedad". Considere el siguiente ejemplo:

```
1 fn main() {  
2     let mut x = vec!["Hola", "mundo"];  
3 }
```



Este programa realiza un enlace de variable llamado **x**. El valor de este enlace es un `Vec<T>`, un “vector” a través de una macro definida en la librería estándar. Esta macro es llamada **vec**, y se invoca a los macros con un **!**. Esto sigue el principio general de Rust: **Haz las cosas explícitamente**. Las macros pueden hacer cosas significativamente mas complicadas que las funciones, es por ello que son visualmente distintas. El símbolo **!** tambien ayuda con el parsing, haciendo la herramienta de macros fácil de escribir, lo cual es muy importante.

Se utiliza **mut** para hacer **x** mutable: los enlaces son inmutables por defecto en Rust.

También vale la pena señalar que no necesitábamos una anotación de tipo aquí: mientras Rust se escribe con tipado estático, no lo necesitamos para anotar explícitamente el tipo . Rust tiene inferencia de tipos para equilibrar el poder de tipos estáticos con el nivel de detalle de la anotación de tipos .

Rust prefiere asignación de pila que asignación de memoria dinámica: **x** es colocada directamente en la pila. Sin embargo, el tipo `Vec<T>` coloca espacios para elementos del vector en la memoria dinámica. Como lenguaje de programación de sistemas, RUST proporciona la habilidad de controlar como las memoria es colocada.

Como mencionamos anteriormente, la “propiedad” es el concepto principal en Rust. En el lenguaje Rust, **x** se dice que es “Dueño” del vector. Esto significa que cuando **x** se elimina, la memoria del vector se descoloca. Esto es realizado determinísticamente por el compilador de Rust, mas que a través de un mecanismo tal como el colector de basura. En otras palabras, en RUST, no se llama a la función como **malloc** y **free** por ti mismo: El compilador estáticamente determina cuando se necesita colocar o liberar memoria, e introduce estas llamadas por el mismo. Equivocarse es humano, pero el compilador nunca olvida.

Si agregamos otra linea a nuestro ejemplo:

```
1 fn main() {  
2 let mut x = vec!["Hello", "world"];  
3  
4 let y = &x[0];  
5 }
```

En el programa anterior se introdujo otra ligadura, **y**. En este caso, **y** es una “referencia” a el primer elemento de el vector. La referencias en Rust son similares a los apuntadores en otros lenguajes, pero con revisiones de seguridad adicionales en tiempo de compilación. Las referencias interactuan

con las el sistema de propiedades “prestando” a lo que apunta, mas que adueñándose de él. La diferencia es, cuando la referencia se elimina, no será liberada de la memoria. Si así fuera, sería liberada dos veces lo cual es malo.

Ahora adicionemos una tercera línea.

```
1 fn main() {  
2     let mut x = vec!["Hello", "world"];  
3  
4     let y = &x[0];  
5  
6     x.push("foo");  
7 }
```

**push** es un método en los vectores que adiciona otro elemento al final del vector. Cuando se trata de compilar este programa, se genera el siguiente error:

```
1 error: cannot borrow 'x' as mutable because it is also  
   borrowed as immutable  
2 x.push(4);  
3 ^  
4 note: previous borrow of 'x' occurs here; the immutable  
   borrow prevents  
5 subsequent moves or mutable borrows of 'x' until the borrow  
   ends  
6 let y = &x[0];  
7 ^  
8 note: previous borrow ends here  
9 fn main() {  
10  
11 }  
12 ^
```

El compilador Rust da información bastante detallada de los errores algunas veces, esta es una de esas veces. Como se señala en el error, mientras hacemos nuestra ligadura mutable no se puede ejecutar **push**. Esto es debido a que aún tenemos una referencia a un elemento del vector, **y**. Mutar algo mientras otra referencia existe es peligroso, porque se podría invalidar la referencia. En este caso específico, cuando creamos un vector, solamente tendremos espacio de memoria para tres elementos. Adicionar un cuarto significa colocar un espacio de memoria para esos elementos nuevos, copiando valores viejos y actualizando el apuntador interno a esa memoria. Todo esto suena bien. El problema es que **y** no se puede actualizar, por lo que tendremos un apuntador colgante. Esto es malo. Cualquier uso de **y** generará un

error en este caso, así que el compilador captura el error para nosotros.

Así que ¿Cómo resolvemos este problema? Se puede utilizando dos técnicas. La primera es haciendo una copia mas que una referencia:

```
1 fn main() {  
2     let mut x = vec!["Hello", "world"];  
3  
4     let y = x[0].clone();  
5  
6     x.push("foo");  
7 }
```

Rust tiene una semántica para mover datos, de tal forma que si queremos hacer una copia de alguna dato, llamamos al método **clone()**. En este ejemplo, **y** no es más una referencia al vector guardado en **x**, pero copia su primer elemento “hola”. Ahora no tenemos referencia, y el método **push()** trabajará bien.

Si realmente se requiere de utilizar una referencia, se necesita otra opción, esta es: asegurarse que nuestra referencia esta fuera del ambiente antes de que tratemos de hacer la mutación. Esto quería de la siguiente forma:

```
1 fn main() {  
2     let mut x = vec!["Hello", "world"];  
3  
4     {  
5         let y = &x[0];  
6     }  
7  
8     x.push("foo");  
9 }
```

En este programa se ha creado un ambiente interno utilizando el conjunto de llaves adicionales. **y** se saca del ambiente antes de llamar al método **push()**, de tal forma que todo funciona adecuadamente.

El concepto de “propiedad ” no es bueno para prevenir apuntadores colgados, pero un conjunto de problemas relacionados, como iteraciones inválidas, concurrencia y otros si se previenen.

## 2.2. Programa “Hola, Mundo”

Antes de escribir nuestro tradicional programa “Hola, Mundo”, vamos a crear nuestro subdirectorio de trabajo

*LSE004-2015*

*Elaboró: Dr. Casimiro Gómez González*

```
1 $ mkdir ~/projects
2 $ cd ~/projects
3 $ mkdir hello_world
4 $ cd hello_world
```

Ahora haremos un nuevo archivo fuente. Y llamaremos al archivo **main.rs**. Los archivos en Rust siempre terminan con la extensión **.rs**. Si utilizas mas de una palabra en el nombre de tu archivo, se recomienda usar un guión inferior **Hola\_Mundo.rs** mas que **HolaMundo.rs**.

Ahora, una vez el archivo este abierto, escribir el siguiente código dentro:

```
1 fn main() {
2     println!("Hola, Mundo");
3 }
```

Grabar el archivo, y luego escribir los siguientes comandos en la terminal:

```
1 $ rustc main.rs
2 $ ./main
3 Hello, world!
```

¡Éxito! Ahora analizaremos el programa en detalle

```
1 fn main() {
2
3 }
```

Estas líneas definen una función en Rust. La función **main** es especial: Es el principio de todo programa Rust. La primera linea dice “Estoy declarando una función llamada **main** la cual no necesita argumentos y no regresa nada”. Si necesitara algún argumento estos irían dentro de los paréntesis, y debido a que no estamos regresando nada se puede omitir la palabra clave **return** completamente.

## 2.3. Convirtiendo a Cargo

Vamos a convertir el programa “Hola Mundo” a Cargo, para ello necesitamos hacer tres cosas:

- Hacer el archivo de configuración **Cargo.toml**
- Poner los archivos fuentes en los directorios adecuados
- Ejecutar el archivo creado

### 2.3.1. Primer paso

```
1 $ mkdir src
2 $ mv main.rs src/main.rs
3 $ rm main
```

Si deseamos crear una librería en lugar de un ejecutable, debemos usar el nombre `lib.rs`. Esta convención es usada por Cargo para compilar exitosamente nuestros proyectos. Localizaciones propias pueden ser especificadas con las palabras claves `[bin]` o `[lib]` en el archivo TOML.

Cargo espera que tus archivos fuentes se encuentren en el directorio **src**. Esto deja los archivos de nivel alto para otras cosas, como READMEs, información de licencias, y cualquier otra cosa no relacionado con tu código. Cargo te ayuda a mantener tus proyectos agradables. Un lugar para cada cosa y cada cosa en su lugar.

Ahora, nuestro archivo de configuración:

## Capítulo 3

# Programación embebida usando la toolchain GNU

Primero debemos actualizar el sistema operativo, en este caso se considera ubuntu:

```
1 $ sudo apt-get update
2 $ sudo apt-get upgrade
3 $ sudo apt-get dist-upgrade
```

Despues debemos instalar el toolchain, para ello anexamos el PPA<sup>1</sup>:

```
1 sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
2 sudo apt-get update
3 sudo apt-get install gcc-arm-embedded
```

### 3.1. Programación Ensamblador para ARM

#### 3.1.1. Sumador

```
1 .thumb
2 .syntax unified
3 sp: .word 0x200
4 reset: .word start+1
5
```

---

<sup>1</sup>Si el PPA anterior aún no tiene una versión del compilador cruzado debemos usar la versión que trae por defecto por ello usamos lo siguiente: `sudo apt-get install gcc-arm-none-eabi`

```

6 start:
7         mov r0, #4
8         mov r1, #5
9         add r2, r1, r0
10 stop:   b stop

```

Los comandos son:

```

1 $ arm-none-eabi-as -mcpu=cortex-m3 add.s -o add.o
2 $ arm-none-eabi-ld -Ttext=0x0 -o add.elf add.o
3 $ arm-none-eabi-objcopy -O binary add.elf add.bin

```

Los comandos qemu para monitorear es el siguiente:

```

1 $ qemu-system-arm -M lm3s811evb -kernel add.bin -monitor
   stdio

```

### 3.1.2. Sumar datos en Memoria

```

1         .syntax unified
2         .thumb
3         .data
4 num1:    .word 0x10
5 num2:    .word 0x20
6 result:  .word 0x0
7         .text
8 sp:      .word 0x200
9 reset:   .word start+1
10
11 start:
12         ldr r0, =sdata           @ Load the address of
13         sdata
14         ldr r1, =edata           @ Load the address of
15         edata
16         ldr r2, =etext           @ Load the address of
17         etext
18
19 copy:    ldrb r3, [r2]            @ Load the value from Flash
20         strb r3, [r0]            @ Store the value in
21         RAM
22         add r2, r2, #1           @ Increment Flash
23         pointer
24         add r0, r0, #1           @ Increment RAM pointer
25         cmp r0, r1              @ Check if
26         end of data
27         bne copy

```

```

23         ldr r0, =num1           @ Load the address of
           num1
24         ldr r1, [r0]           @ Load the value in
           num1
25
26         ldr r0, =num2           @ Load the address of
           num2
27         ldr r2, [r0]           @ Load the value in
           num2
28
29         add r3, r1, r2          @Add num1 and num2
30
31         ldr r0, =result         @ Load the address of
           result
32         str r3, [r0]           @ Store the value in
           result
33
34 stop:    b stop

```

Script del enlazador add-ram.s:

```

1 MEMORY
2 {
3     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000
4     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00002000
5 }
6 SECTIONS {
7     .text : {
8         * (.text);
9         etext = .;
10    } > FLASH
11    .data : {
12        sdata = .;
13        * (.data);
14        edata = .;
15    } > SRAM AT> FLASH
16 }

```

Los comandos son:

```

1 $ arm-none-eabi-as -mcpu=cortex-m3 add-ram.s -o add-ram.o
2 $ arm-none-eabi-ld -T add-ram.lds -o add-ram.elf add-ram.o
3 $ arm-none-eabi-objcopy -O binary add-ram.elf add-ram.bin

```

Comandos de monitor qemu:

```

1 $ qemu-system-arm -M lm3s811evb -kernel add-ram.bin -monitor
  stdio

```



```
2 xp /1xw 0x20000000
3 xp /1xw 0x20000004
4 xp /1xw 0x20000008
```

# Capítulo 4

## Ensamblador en linux

Para los programas de esta sección se utiliza el ensamblador de gcc en una computadora intel con ubuntu. Algunos códigos podrán transportarse al tarjetas con Arm y linux instalado pero solo aquellos que dependan del sistema operativo y no del hardware.

### 4.1. Desplegando informacion en linux con ensamblador

El siguiente código despliega un mensaje en la pantalla utilizando una llamada al sistema de linux (int \$0x80). El kernel de linux proporciona muchas funciones preestablecidas que pueden fácilmente ser accedidas desde ensamblador. Para acceder a estas funciones del kernel, se debe utilizar el código de instrucción **int** , el cual genera una interrupción por software, con el valor 0x80. La función específica que es realizada es determinada por el valor del registro EAX. Las llamadas al sistema de linux ahorran mucho tiempo al programador de ensamblador.

```
1 movl $4, %eax
2 movl $1, %ebx
3 movl $output, %ecx
4 movl $42, %edx
5 int $0x80
```

Las llamadas de escritura al sistema linux es usado para escribir bytes a un archivo. Los siguientes son los parámetros para las llamadas de escritura al sistema:

- EAX contiene el valor de la llamada al sistema
- EBX Contiene el descriptor de archivo al cual se va a escribir
- ECX contiene el inicio de la cadena
- EDX contiene el largo de la cadena

La salida estandar (STDOUT) representa la terminal de pantalla de la sesión actual, y tiene un **descriptor de archivo de 1**. Escribiendo a este archivo descriptor se despliega la información en la terminal de la consola. Los bytes a desplegar son definidos como localidades de memoria desde donde se lee la información, y el número de bytes que se desplegarán. El registro **ECX** es cargado con la localidad de memoria de la etiqueta **output**, la cual define el inicio de la cadena. Debido a que a lo largo del programa el largo de la cadena es siempre el mismo, se define como constante el tamaño en el registro **EDX**. En el código anterior el valor de la llamada al sistema es 4 (**EAX**), el descriptor de archivo es 1 (**EBX**), el inicio de la cadena **output** se guarda en **ECX** y en **EDX** se pone el tamaño de la cadena que es de 42 bytes o caracteres. Una vez escritos los valores en los registros pertinentes se llama a la interrupción por software con **int \$0x80**.

En el siguiente código se muestra el código en ensamblador para salir del programa. Se mueve el valor de 1 al registro EAX lo cual define el valor de llamada a sistema (función exit) y el descriptor de archivo, registro EBX, se define como 0 lo cual indica el valor que regresará el programa a el shell de linux. Esto puede ser usado para producir diferentes resultados en el programa del shell, dependiendo de la situación dentro del programa en ensamblador. El valor de cero indica que el programa se terminó exitosamente. Posteriormente a la colocación de los valores en los registros se llama a la interrupción por software **int \$0x80**.

```
1 movl $1, %eax
2 movl $0, %ebx
3 int $0x80
```

#### 4.1.1. Hola Mundo en ensamblador

A continuación se muestra un ejemplo de un programa en ensamblador que despliega información de una cadena, en este caso el mundialmente conocido "Hola Mundo". A continuación se muestra el código en ensamblador

*LSE004-2015*

*Elaboró: Dr. Casimiro Gómez González*

```

1 #Hola_Mundo.s Programa Hola Mundo en ensamblador
2 .section .data
3 output:
4     .ascii "Hola_Mundo_Cabron\n"
5 .section .text
6 .globl _start
7 _start:
8     movl $4, %eax
9     movl $1, %ebx
10    movl $output, %ecx
11    movl $18, %edx
12    int $0x80
13    movl $1, %eax
14    movl $0, %ebx
15    int $0x80

```

Para compilar el programa y volverlo ejecutable

```

1 $ as -o hola.o Hola_Mundo.s
2 $ ld -o Hola hola.o

```

posteriormente se ejecuta el programa con el comando:

```

1 $ ./Hola

```

#### 4.1.2. Enviando resultados a la salida estandar

Es importante recibir información de salida de un programa y esta información reenviarla al usuario. Para ello se muestra el siguiente programa que utiliza el programa del sistema CUID para capturar la información del sistema. En la siguiente tabla se muestran las diferentes opciones de salida

Tomando en cuenta los valores de EAX de la tabla 4.1 se escribe el código para llamar al programa que se muestra a continuación:

```

1 #cpuid.s Ejemplo de programa que obtiene Vendor ID del
   procesador
2 .section .data
3 output:
4     .ascii "El_ID_del_vendedor_del_procesador_es_XXXXXXXXXXXXX
       '\n"
5 .section .text
6 .globl _start
7 _start:
8     movl $0, %eax
9     cpuid

```

Valor EAX	Salida CPUID
0	Cadena del vendedor del microprocesador
1	Tipo de Procesador, familia, modelo
2	Configuración de cache del procesador
3	Número de serie del procesador
4	Configuración de cache(Número de hilos, número de núcleos, y propiedades físicas)
5	Información del Monitor
80000000h	Cadena extendida del vendedor y niveles soportados
80000001h	Tipo de procesador extendido, familia, modelo
80000002h - 80000004h	Cadena extendida del procesador

Cuadro 4.1: Opciones EAX para el comando CPUID.

```

10  movl $output, %edi
11  movl %ebx, 38(%edi)
12  movl %edx, 42(%edi)
13  movl %ecx, 46(%edi)

```

En el código anterior se crea una cadena output en donde los valores de x serán sustituidos por la salida que produce el comando cpuid con la opción 0 en el registro EAX, que como indica la tabla 4.1 produce el ID del vendedor del microprocesador. Esta cadena se almacena en los registros EBX, EDX y ECX donde la cadena empieza en el registro EBX y se almacenan 4 bytes o caracteres y los 4 siguientes en EDX y los últimos 4 en ECX. Los valores de que el comando CPUID almacenó en el registro EBX se pasan a la cadena cuyo valor inicial apunta el registro EDI. Como puede observarse en el código se mueven los valores de las cadenas EBX, EDX y ECX a las posiciones 38, 42 y 46 de la cadena apuntada por el registro EDI, el número indica la posición relativa señalada por la posición de memoria base del registro EDI. Una vez que la cadena señalada por el registro EDI se modificó con los resultados entonces se puede hacer una llamada al sistema para desplegar esta cadena modificada en la salida estandar de tal forma que el programa completo queda:

```

1 #cpuid.s Ejemplo de programa que obtiene Vendor ID del
  procesador
2 .section .data
3 output:

```

```

4  .ascii "El_ID_del_vendedor_del_procesador_es_XXXXXXXXXXXX
   '\n"
5  .section .text
6  .globl _start
7  _start:
8  movl $0, %eax
9  cpuid
10 movl $output, %edi
11 movl %ebx, 38(%edi)
12 movl %edx, 42(%edi)
13 movl %ecx, 46(%edi)
14 movl $4, %eax
15 movl $1, %ebx
16 movl $output, %ecx
17 movl $52, %edx
18 int $0x80
19 movl $1, %eax
20 movl $0, %ebx
21 int $0x80

```

Para compilar el programa y volverlo ejecutable

```

1 $ as -o cpuid.o cpuid.s
2 $ ld -o cpuid cpuid.o

```

Posteriormente se ejecuta el programa con el comando:

```

1 $ ./cpuid

```

## 4.2. Usando gdb

Para poder realizar una depuración de un programa en ensamblador, primero se debe reensamblar el código usando el parámetro `-gstab`:

```

1 $ as -gstab -o cpuid.o cpuid.s
2 $ ld -o cpuid cpuid.o

```

Especificando el parámetro `-gstab` información extra es ensamblada dentro del ejecutable para ayudar al programa `gdb` para recorrer el programa fuente. Ahora el programa ejecutable contiene la información necesaria de depuración, y el programa con `gdb` se puede correr:

```

1 $ gdb cpuid

```

Con el comando anterior el depurador inicia, con el programa cargado en la memoria. Es posible ejecutar el programa dentro del depurador con los siguientes comandos:

- **run.**- Sirve para ejecutar el programa
- **break \*label+offset.**- Sirve para definir un breakpoint, por ejemplo **break \*\_\_start**, define un breakpoint en la etiqueta **\_\_start** del programa, o bien **break \*\_\_start+1** que coloca un breakpoint en la instrucción siguiente a la etiqueta **\_\_start**.
- **next.**- Ejecuta la siguiente línea de código
- **step.**- Ejecuta al igual que **next** la siguiente línea de código
- **cont.**- Continúa la ejecución normal del código

Además de los comandos de control de ejecución en el depurador son importantes los comandos de despliegue de datos, en la siguiente tabla se muestran estos comandos con sus ejemplos:

Comando de datos	Descripción
<b>info registers</b>	Despliega el valor de todos los registros
<b>print</b>	Imprime el valor de un registro o variable
<b>x</b>	Imprime el valor de una localidad de memoria

Cuadro 4.2: Opciones para imprimir registros y variables en el depurador.

### 4.2.1. El comando Print del depurador

El comando **print** del depurador se utiliza para imprimir valores individuales de registros. Incluyendo modificadores que pueden dar formato a los valores de salida del comando:

- **print/d.**- Imprime el valor en decimal
- **print/t.**- Imprime los valores en binario
- **print/x.**- Imprime los valores en hexadecimal

Por ejemplo:

*LSE004-2015*  
*Elaboró: Dr. Casimiro Gómez González*

```
1 (gdb) print/x $ebx
2 $9 = 0x756e6547
3 (gdb) print/x $edx
4 $10 = 0x49656e69
5 (gdb) print/x $ecx
6 $11 = 0x6c65746e
7 (gdb)
```

### 4.2.2. Comando x del depurador

El comando **x** es usado para imprimir los valores de direcciones específicas de memoria. Similar al comando **print**, el comando **x** puede ser formateada. El formato es el siguiente:

```
1 x/nyz
```

donde **n** es el número de caracteres o bytes a imprimir, **y** es el formato de salida los cuales pueden ser:

- c .- para caracteres
- d .- Para decimales
- x .- para decimales

y **z** es el tamaño de la palabra en bits:

- b .- para un byte (8 bits)
- h .- Para media palabra de 16 bits
- w .- Para una palabra de 32 bits

A continuación se muestran algunos ejemplos del comando **x**:

```
1 (gdb) x/42cb &output
```

Este comando imprime los primeros 42 bytes de la variable salida (el amperser indica que se toma la dirección de memoria) en modelidad de caracteres.



## Capítulo 5

# Diseñando el Sistema Operativo *Ut Vitam* con Rust

En este trabajo se supondrá que se utilizá un sistema operativo parecido a Unix como plataforma de diseño, como por ejemplo Linux.

### 5.1. Construyendo un Cross-Compiler

Hay una convención llamada “el triple objetivo” para describir una plataforma en particular. Es triple porque consta de tres partes:

```
1 arch-kernel-userland
```

Así que, un triple objetivo para una computadora que tiene un procesador x86-64 bit ejecutando un kernel de linux y programas de usuario (userland) de GNU se vería así:

```
1 x86_64-linux-gnu
```

Si esta compuesto por cuatro partes, se le llama “Objetivo” mas que “triple objetivo”.

Los kernels por si mismos no necesitan ser para un específico conjunto de programas de usuario, y por eso tu verás ‘none’ :

```
1 x86_64-unknown-none
```

### 5.1.1. Anfitrión y Objetivos

La razón por los cuales son llamados “objetivos” es porque es la arquitectura para la cual estas compilando. La arquitectura desde la cual compilas se llama “Arquitectura Anfitrión”.

Si el objetivo y el anfitrión son los mismos, nosotros los llamamos “Compilación”. Si son diferentes, entonces se llama “Compilación cruzada”. Así se puede afirmar:

#### Compile cruzado desde x86\_64-linux-gnu a x86-unknown-none

Esto significa que la computadora de desarrollo fue una con GNU/linux de 64 bits, pero el binario final es para una máquina x86 de 32 bits sin sistema operativo.

Por lo tanto se necesita un ambiente en particular construir nuestro Sistema Operativo: Se necesita un compilador cruzado desde cualquier tipo de computadora se este usando para nuestro objetivo.

Para instalar en linux debian o ubuntu, se necesita ejecutar lo siguiente:

```
1 sudo apt-get install nasm xorriso qemu build-essential
```

Lo primero que se debe realizar es configurar un Cross-Compiler para la plataforma **arm-none-eabi**. No serás capaz de compilar correctamente tu sistema operativo sin un *Cross compiler*. Ahora necesitamos compilar **libcore** de Rust a nuestra arquitectura objetivo . Se trata de una base mínima libre de la dependencia de la biblioteca estándar. Para eso es necesario para compilar desde el código fuente :

```
1 git clone https://github.com/rust-lang/rust.git
2 cd rust
3 rustc --version    #Se toma el numero que este comando y se
                     #introduce despues del comando checkout de la linea
                     #siguiente
4 git checkout 21922e1f4
5 sudo mkdir -p /usr/local/lib/rustlib/arm-unknown-linux-
   gnueabihf/lib
6 sudo rustc --target arm-unknown-linux-gnueabihf -O -Z no-
   landing-pads src/libcore/lib.rs --out-dir /usr/local/lib/
   rustlib/arm-unknown-linux-gnueabihf/lib
```

### 5.1.2. Raspberry Pi bootloader

En el caso de las Raspberry Pi, dos archivos pueden ser encontrados en <https://github.com/raspberrypi/firmware/tree/master/boot>, el **bootcode.bin** y el **start.elf**. Teniendo un SDCARD de arranque se le borran todos los archivos y se colocan los dos: **bootcode.bin** y **start.elf** en la partición **boot** de la tarjeta.

## 5.2. Comenzando la programación

Con todo configurado, podemos empezar a escribir el código. Vamos a hacer que el led ACT de la Raspberry parpadee, es el verde que se encuentra a lado del led de encendido. Así que, nuestro programa, básicamente necesita encender el led, esperar un tiempo, y apagarlo y luego ciclarse. Así que en un editor creamos el archivo *kernel.rs*:

```
1 fn main() {  
2     println!("Hola_Mundo!");  
3 }
```

Este es el ejemplo estandar de hola mundo, pero la macro **println!** es parte de la librería estandar de Rust y usa la plataforma E/S para imprimir algo en la pantalla y en este momento no tenemos ninguna plataforma definida. De tal manera, que necesitamos informarle al compilador que nuestro código independiente será independiente:

```
1 #![feature(no_std)]  
2 #![crate_type = "staticlib"]  
3 #![no_std]  
4  
5 // kernel.rs  
6 fn main() {  
7     println!("Hola_Mundo!");  
8 }
```

En la primera línea le informamos al compilador que vamos a usar una característica **no\_std**, esto se realiza con el atributo **#![feature()]** el cual le informa este tipo de características al compilador. Con la instrucción **#![crate\_type]** le informamos al compilador que estamos compilando una librería estática, lo cual básicamente significa que todo nuestro código está auto-contenido y finalmente, con la instrucción **#![no\_std]** le informamos al

```

Terminal
Archivo Editar Ver Buscar Terminal Ayuda
casho@casholap ~/github/EmbeddedRust/RaspBerry $ rustc --target arm-unknown-linux-gnueabi kernel.rs
kernel.rs:7:5: 7:12 error: macro undefined: `println!`
kernel.rs:7      println!("Hola Mundo!");
                  ^
error: aborting due to previous error
casho@casholap ~/github/EmbeddedRust/RaspBerry $

```

Figura 5.1: Resultados de la compilación cruzada

```

Terminal
Archivo Editar Ver Buscar Terminal Ayuda
casho@casholap ~/github/EmbeddedRust/RaspBerry $ rustc --target arm-unknown-linux-gnueabi kernel.rs
error: language item required, but not found: `panic_fmt`
error: language item required, but not found: `eh_personality`
error: aborting due to 2 previous errors
casho@casholap ~/github/EmbeddedRust/RaspBerry $

```

Figura 5.2: compilación cruzada sin **println!**

compilador que no usaremos la librería estándar, de tal manera que no se-  
rá cargada automáticamente en los procesos iniciales. Ahora si intentamos  
compilar obtenemos el siguiente error de la figura 5.1

Como se muestra en la figura 5.1, la macro **println!** no está definida,  
porque no esta implementada en las rutinas de E/S y el runtime, ¡somos  
independientes!. Si removemos la instrucción **println!** e intentamos compilar  
nuevamente obtenemos lo mostrado en la figura 5.2

Esto significa que el runtime mínimo de Rust espera que estas dos fun-  
ciones sean definidas. Ambas funciones son usadas por los mecanismos de  
fallo/manejo del compilador. Para un *hola mundo* independiente, definimos  
estos como implementaciones independientes.

```

1 // kernel.rs
2 #![feature(no_std, lang_items)]
3 #![crate_type = "staticlib"]
4 #![no_std]
5
6 pub extern fn main() {
7     loop {}
8 }
9
10 #![lang = "eh_personality"]
11 extern fn eh_personality() {}
12

```

```

13 #[lang = "panic_fmt"]
14 extern fn panic_fmt() {}

```

El atributo `#[lang]` informa al compilador que la función esta siendo definida para el runtime. Ya que este atributo es una característica necesitamos agregarla a nuestra lista en el atributo `#![feature()]`. Ahora para compilar, necesitamos informarle al compilador que queremos producir un archivo objeto. Un archivo objeto es básicamente el código compilado con los símbolos definidos. Usaremos el archivo objeto generado para enlazarlos con **arm-none-eabi-gcc** y generar el archivo **kernel.elf**. Adicionamos la bandera **-O** para remover algunos símbolos innecesarios y realizar alguna optimización. También cambiamos nuestra función **main** para que sea pública y externa. Esto informa al compilador que nuestra función será usado externamente.

```

1 rustc --target arm-unknown-linux-gnueabi -O --emit=obj
  kernel.rs

```

Esto genera el archivo **kernel.o**. Podemos comprobar si nuestro compilador cruzado esta trabajando y produciendo código máquina compatible con ARM utilizando el comando *file*:

```

1 file kernel.o
2 #kernel.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1
  (SYSV), not stripped

```

Podemos revisar los símbolos del archivo objeto usando **arm-none-eabi-nm**<sup>1</sup> :

```

1 arm-none-eabi-nm kernel.o

```

Lo cual produce el resultado de la figura 5.3.

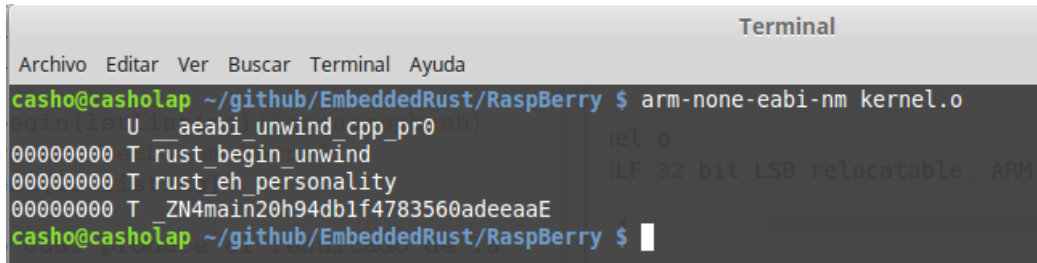
Esto nos muestra los tres símbolos definidos por nuestro código fuente. El extraño es nuestra función principal **main**. Esto sucede debido a algo que se llama renombrado, que es algo que el compilador hace para evitar conflictos de nombres. Queremos que nuestros símbolos sean claros, por lo que las herramientas externas sabe cómo hacer referencia a ella. Para indicar al compilador que no destrozará nuestra función principal, tenemos que indicarlo :

```

1 // kernel.rs
2 #![feature(no_std, lang_items)]
3 #![crate_type = "staticlib"]

```

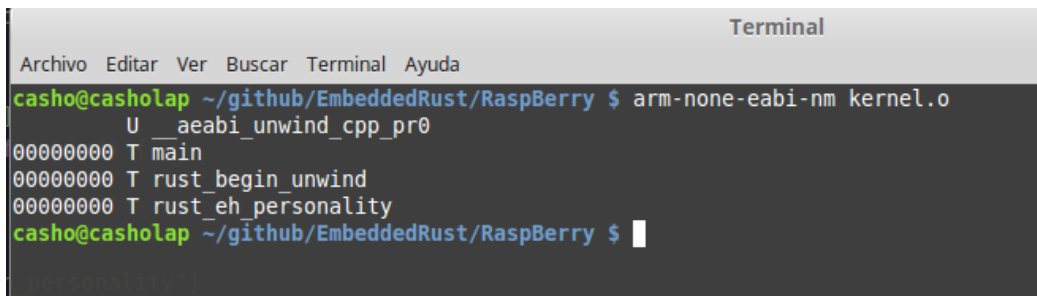
<sup>1</sup>si el comando anterior no funciona, es necesario instalar lo siguiente: `sudo apt-get install binutils-arm-none-eabi`



```

Terminal
Archivo Editar Ver Buscar Terminal Ayuda
casho@casholap ~/github/EmbeddedRust/RaspBerry $ arm-none-eabi-nm kernel.o
00000000 U __aeabi_unwind_cpp_pr0
00000000 T rust_begin_unwind
00000000 T rust_eh_personality
00000000 T _ZN4main20h94db1f4783560adeeaaE
casho@casholap ~/github/EmbeddedRust/RaspBerry $

```

Figura 5.3: Etiquetas del **kernel.o**


```

Terminal
Archivo Editar Ver Buscar Terminal Ayuda
casho@casholap ~/github/EmbeddedRust/RaspBerry $ arm-none-eabi-nm kernel.o
00000000 U __aeabi_unwind_cpp_pr0
00000000 T main
00000000 T rust_begin_unwind
00000000 T rust_eh_personality
casho@casholap ~/github/EmbeddedRust/RaspBerry $

```

Figura 5.4: Etiquetas del **kernel.o** con **no\_mangle**

```

4 #![no_std]
5
6 #[no_mangle]
7 pub extern fn main() {
8     loop {}
9 }
10
11 #[lang = "eh_personality"]
12 extern fn eh_personality() {}
13
14 #[lang = "panic_fmt"]
15 extern fn panic_fmt() {}

```

Si compilamos y checamos los símbolos obtenemos lo indicado en la figura 5.4

Ahora todos las etiquetas son claras lo cual será útil cuando lo necesitemos llamar desde el ensamblador.

## 5.3. Haciendo el LED parpadear

La dirección base de la Raspberry Pi B+ GPIO es 0x20200000. El registro que nos permitirá encender nuestro LED es la dirección base + offset 0x8 y tenemos que establecer el bit 15 al 1 para encenderlo. Para desactivarlo, tenemos que establecer el bit 15 al 0 en la dirección base + 0xB offset. Entre estos cambios , tenemos que bloquear el programa durante algún tiempo. Nuestro código se verá así :

```
1 // kernel.rs
2
3 #![feature(no_std, lang_items, asm)]
4 #![crate_type = "staticlib"]
5 #![no_std]
6
7 const GPIO_BASE: u32 = 0x20200000;
8
9 fn sleep(value: u32) {
10     for _ in 1..value {
11         unsafe { asm!(""); }
12     }
13 }
14
15 #[no_mangle]
16 pub extern fn main() {
17     let gpio = GPIO_BASE as *const u32;
18     let led_on = unsafe { gpio.offset(8) as *mut u32 };
19     let led_off = unsafe { gpio.offset(11) as *mut u32 };
20
21     loop {
22         unsafe { *(led_on) = 1 << 15; }
23         sleep(500000);
24         unsafe { *(led_off) = 1 << 15; }
25         sleep(500000);
26     }
27 }
28
29 #[lang = "eh_personality"] extern fn eh_personality() {}
30 #[lang = "panic_fmt"] extern fn panic_fmt() {}
```

Hemos añadido una función de apagado automático que básicamente es un bucle vacío para un rango dado. El **unsafe asm!()** fue la manera que se ha encontrado para evitar que el compilador para elimine a esta parte del código durante la fase de optimización, ya que no parece hacer nada.

En la función principal **main**, creamos un puntero raw para la dirección de GPIO\_BASE, entonces obtenemos un puntero mutable de los lugares que queremos cambiar, que son el uno para encender el LED y para apagarlo. Desde la llamada desplazamiento implica la eliminación de referencias un puntero raw, esto es una operación insegura, por tanto, los bloques no seguros. Con que sólo podemos hacer un bucle infinito que Enciende el LED, espera, apagarlo y esperar.

### 5.3.1. Preparando el código

Ahora podemos compilar el código para ejecutarlo en la Raspberry:

```
1 rustc --target arm-unknown-linux-gnueabihf -O --emit=obj
  kernel.rs
```

Con el archivo **kernel.o** generado se necesita enlazar el código usando **arm-none-eabi-gcc**, así que esto puede configurar nuestro símbolo **main** como el punto de entrada a nuestro programa, para generar el archivo **kernel.elf** y luego usar **arm-none-eabi-objcopy** para generar el archivo binario final **kernel.img**<sup>2</sup>.

```
1 arm-none-eabi-gcc -O0 -mcpu=vfp -mfloat-abi=hard -march=
  armv6zk -mtune=arm1176jzf-s -nostartfiles --specs=rdimon.
  specs -lgcc -lc -lm -lrdimon kernel.o -o kernel.elf
2 arm-none-eabi-objcopy kernel.elf -O binary kernel.img
```

Con nuestro **kernel.img** construido, sólo tenemos que copiarlo en nuestra frambuesa SD junto con los otros archivos de arranque ( **start.elf** y **bootcode.bin** ), se adhieren a nuestra micro SD de nuevo en el Pi de frambuesa y encenderlo para ver nuestra pequeña núcleo en ejecución y parpadea el LED !

Hay un montón de otras cosas interesantes que podemos hacer, como usar el temporizador del sistema en chip para poner en práctica nuestra función dormir o crear algunas abstracciones de la parte superior de la GPIO se dirige para encapsular todas las llamadas inseguras en una API más elegante. Pero creo que por ahora esto es una buena manera de empezar con la programación de metal desnudo con mohó.

<sup>2</sup>Si la instrucción de la línea 1 no se ejecuta es necesario instalar con el siguiente comando:  
 sudo apt-get install gcc-arm-none-eabi



## 5.4. Tips de compilación y ensamblador

Primero se escribe el programa `Hola_Mundo.c` con el siguiente código:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hola_Mundo\n");
6     return 0;
7 }
```

Compilando y checando su tamaño en bytes con los siguientes comandos:

```
1 gcc -o hola Hola_Mundo.c
2 wc -c hola
```

La última instrucción da como resultado:

```
1 8608 hola
```

¿De donde provienen los 8.4Kb del archivo ejecutable? **hola** nos da una table de simbolos de 79 entradas que pertenecen a nuestra librería estandar. Así que dejaremos de usarla. No utilizaremos *printf* de tal forma que ya no es necesario incluir la librería en el programa:

```
1 int main()
2 {
3     char *str = "Hola_Mundo";
4     return 0;
5 }
```

Recompilando y checando el tamaño en bytes:

```
1 $ gcc -o hola Hola_Mundo.c
2 $ wc -c hola
3 8560 hola
```

Oh!! Podemos observar que casi no disminuyó el número de bytes (8.36Kb).

El problema es que gcc esta todavía usando las librerías estandar cuando esta enlazando. Vamos ahora a compilar con la opción **-nostdlib** la cual de acuerdo con el manual de gcc no usa las librerías estandar cuando esta enlazando. Solamente los archivos que especifiques serán pasados al enlazador.

```
1 $ gcc -nostdlib -o hola Hola_Mundo.c
2 /usr/bin/ld: aviso: no se puede encontrar el simbolo de
   entrada _start; se usa por defecto 000000000400144
3 $ wc -c hola
4 1648 hola
```

Esto ya se ve muy bien el tamaño del archivo es más razonable (1.6Kb). Ahora ejecutamos:

```
1 $ ./hola
2 Violacion de segmento ('core' generado)
```

Como puede observarse se redujo a expensas de una violación de segmento

Desde la perspectiva del enlazador, por defecto `__start` es el punto de entrada a tu programa, no el programa principal `main`. Esto no está definido en el `crt1.o` de la tabla del archivo ejecutable `ELF`. Se puede verificar esto enlazando nuevamente `crt1.o` y nada parecido a `__start` no es encontrado<sup>3</sup>:

```
1 #Compilar el archivo fuente pero no enlazar
2 $ gcc -Os -c Hola_Mundo.c
3 #Ahora tratar de enlazar
4 $ ld /usr/lib/x86_64-linux-gnu/crt1.o -o hola Hola_Mundo.o
5 /usr/lib/x86_64-linux-gnu/crt1.o: En la función '__start':
6 (.text+0x12): referencia a '__libc_csu_fini' sin definir
7 /usr/lib/x86_64-linux-gnu/crt1.o: En la función '__start':
8 (.text+0x19): referencia a '__libc_csu_init' sin definir
9 /usr/lib/x86_64-linux-gnu/crt1.o: En la función '__start':
10 (.text+0x25): referencia a '__libc_start_main' sin definir
```

Esto nos indica donde se encuentra `__start` en la librería fuente: `sysdeps/x86_64/elf/start.S` para la máquina en particular. Este archivo bien comentado exporta el símbolo `__start`, ajusta la pila y algunos registros, y llama al programa principal `__libc_start_main`. Si revisamos al final del programa `csu/libc-start.c` podemos ver la llamada al programa principal:

```
1 result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);
```

Eso es todo acerca de `__start`. Vamos a exportar nuestro propio símbolo `__start` y llamar al programa `main` y enlazar de nuevo.

```
1 $ nano stubstart.S
2 .globl __start
3
4 __start:
5     call main
```

Compilando y ejecutando con el archivo ensamblador `stubstart.s`:

```
1 $ gcc -nostdlib stubstart.S -o hola Hola_Mundo.c
2 $ ./hola
3 Violacion de segmento ('core' generado)
```

<sup>3</sup>Es importante primero ubicar el archivo `crt1.o` en nuestro sistema con el comando: `sudo find . -name crt1.o -print`

Ahora ya no tenemos errores de compilación pero aun tenemos errores de segmentación. Ahora vamos a compilar con opciones de debugging y usaremos gdb (se introduce la opción -g). Ajustaremos un punto de quiebre en el programa principal y revisar hasta la falla:

```
1 $ gcc -g -nostdlib stubstart.S -o hola Hola_Mundo.c
2 $ gdb hola
3 GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
4 Copyright (C) 2016 Free Software Foundation, Inc.
5 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/
  licenses/gpl.html>
6 This is free software: you are free to change and
  redistribute it.
7 There is NO WARRANTY, to the extent permitted by law.  Type "
  show_copyright"
8 and "show_warranty" for details.
9 This GDB was configured as "x86_64-linux-gnu".
10 Type "show_configuration" for configuration details.
11 Para las instrucciones de informe de errores, vea:
12 <http://www.gnu.org/software/gdb/bugs/>.
13 Find the GDB manual and other documentation resources online
  at:
14 <http://www.gnu.org/software/gdb/documentation/>.
15 For help, type "help".
16 Type "apropos word" to search for commands related to "word"
  ...
17 Leyendo simbolos desde hola...hecho.
18 (gdb) break main
19 Punto de interrupcion 1 at 0x40014d: file Hola_Mundo.c, line
  3.
20 (gdb) run
21 Starting program: /home/casho/Documentos/ProgramasC/
  Hola_Mundo/hola
22
23 Breakpoint 1, main () at Hola_Mundo.c:3
24 3      char *str = "Hola_Mundo";
25 (gdb) step
26 4      return 0;
27 (gdb) step
28 5      }
29 (gdb) step
30 main () at Hola_Mundo.c:2
31 2      {
32 (gdb) step
33
```

```

34 Breakpoint 1, main () at Hola_Mundo.c:3
35 3      char *str = "Hola_Mundo";
36 (gdb) step
37 4      return 0;
38 (gdb) step
39 5      }
40 (gdb) step
41
42 Program received signal SIGSEGV, Segmentation fault.
43 0x0000000000000001 in ?? ()
44 (gdb)

```

Como se puede observar ejecutamos dos veces la rutina main. Si observamos en el ensamblador:

```

1 $ objdump -d hola
2
3 hola:      formato del fichero elf64-x86-64
4
5
6 Desensamblado de la seccion .text:
7
8 0000000000400144 <_start>:
9   400144:      e8 00 00 00 00      callq  400149 <main>
10
11 0000000000400149 <main>:
12   400149:      55                  push   %rbp
13   40014a:      48 89 e5            mov    %rsp,%rbp
14   40014d:      48 c7 45 f8 5c 01 40 movq   $0x40015c,-0x8
15   400154:      00                  (%rbp)
16   400155:      b8 00 00 00 00      mov    $0x0,%eax
17   40015a:      5d                  pop    %rbp
18   40015b:      c3                  retq

```

Revisando el programa anterior es posible observar que se ejecuta primero del programa ensamblador la línea 400144 (etiqueta <\_start> que corresponde a la línea que llama a la subrutina <main> al terminar de ejecutar esta rutina la instrucción `retq` regresa el programa a la siguiente instrucción después de ejecutar el `callq` lo cual precisamente es el inicio de <main> ya que no hay un indicio de fin de programa después de `callq` por lo que se vuelve a ejecutar el programa <main> y al encontrar el `retq` se indica un error ya que no se llama a ninguna rutina y no hay nada en la pila. Para corregir lo anterior hay que indicar en el programa principal una salida al sistema después del `callq` con `SYS_exit` dentro del registro `%eax`. Después hacemos una interrupción

dentro del kernel con `int $0x80` para indicarle que salimos del programa sin errores.

```
1 $ nano stubstart.S
2 .globl _start
3
4 _start:
5     call main
6     movl $1, %eax
7     xorl %ebx, %ebx
8     int $0x80
```

y ejecutando

```
1 $ gcc -nostdlib stubstart.S -o hola Hola_Mundo.c
2 ./hola
```

Ahora nuestro programa se ejecuta sin errores.

# Capítulo 6

## Sistema Operativo para x86

En este caso se necesitan tres archivos:

- boot.s.- Punto de entrada del kernel que configura el entorno del procesador
- kernel.rs.- Las rutinas actuales del kernel
- linker.ld.- Para enlazar los archivos anteriores

Les presento el nuevo sistema operativo UtVitam. UtVitam es un sistema operativo enfocado a dar vida a nuevos proyectos que busquen funcionar independiente y embebidos, por ello es que UtVitam proviene del latín que significa **A la vida**. La primera decisión que debemos hacer nada más plantearnos el sistema operativo es ¿cuál va a ser el bootloader?

Aquí existen múltiples variantes, e incluso podríamos crear uno nosotros; sin embargo, vamos a usar GRUB, porque la mayoría conoce más o menos algo de él. Creamos una carpeta que será el root de nuestro sistema operativo y allí creamos la carpeta /boot/grub

```
1 mkdir UtVitamroot && cd UtVitamroot
2 mkdir -p boot/grub
```

Allí creamos el fichero grub.cfg de la siguiente manera:

```
1 menuentry "UtVitam" {
2
3     echo "Booting UtVitam"
4
5     multiboot /Ut/START.ELF
6 }
```

```
7 boot
8
9 }
```

En este fichero hemos visto como GRUB cargará nuestro kernel, en este caso, en /Ut/START.ELF. Ahora debemos crear nuestro kernel.

Para ello necesitaremos el GCC y GAS (el ensamblador del proyecto GNU, suele venir con el gcc). Así pues vamos a crear el kernel.

Primero hacemos un archivo llamado kernel.asm. Este archivo contendrá el punto de inicio de nuestro kernel y además definirá el multiboot (una característica de algunos bootloaders como GRUB). El contenido de kernel.asm será:

```
1 .text
2
3 .globl start
4
5 start:
6
7 jmp multiboot_entry
8
9 .align 4
10
11 multiboot_header:
12
13 .long 0x1BADB002
14
15 .long 0x00000003
16
17 .long -(0x1BADB002+0x00000003)
18
19 multiboot_entry:
20
21 movl $(stack + 0x4000), %esp
22
23 call UtVitam_Main
24
25 loop: hlt
26
27 jmp loop
28
29 .section ".bss"
30
31 .comm stack,0x4000
```

Todo lo relacionando con multiboot es simplemente seguir la especificación nada más. Todo empezará en start, llamará a multiboot\_entry, habremos definido el multiboot header en los primeros 4k y lo pondremos (con movl).

Más tarde llamamos a UtVitam\_Main que es nuestra función en C del kernel. En el loop hacemos un halt para parar el ordenador. Esto se compila con:

```
1 as -o kernel.o -c kernel.asm
```

Ahora vamos a entrar a programar en C. Pensarás que ahora todo es pan comido, ponemos un printf en main y ya está, lo hemos hecho.

Pues no, ya que printf y main son funciones que define el sistema operativo, ¡pero nosotros lo estamos creando! Solo podremos usar las funciones que nosotros mismos definamos.

En capítulos posteriores hablaré de como poner nuestra propia librería del C (glibc, bionic, newlibc) pero tiempo al tiempo. Hemos hablado que queremos poner texto en pantalla, bueno veremos como lo hacemos.

Hay dos opciones, una es llamar a la BIOS y otra es manejar la memoria de la pantalla directamente. Vamos a hacer esto último pues es más claro desde C y además nos permitirá hacerlo cuando entremos en modo protegido.

Creamos un fichero llamado UtVitam\_Main.c con el siguiente contenido:

```
1 int UtVitam_Main()  
2  
3 {  
4  
5 char *str = "NextDiveI_says_Hello_World", *ch;  
6  
7 unsigned short *vidmem = (unsigned short*) 0xb8000;  
8  
9 unsigned i;  
10  
11 for (ch = str, i = 0; *ch; ch++, i++)  
12  
13 vidmem[i] = (unsigned char) *ch | 0x0700;  
14  
15 return 0;  
16  
17 }
```

Con esto manipulamos directamente la memoria VGA y caracter a caracter lo vamos escribiendo. Compilamos desactivando la stdlib:



```
1 gcc -o UtVitam_Main.o -c UtVitam_Main.c -nostdlib -fPIC -  
    ffreestanding
```

Si has llegado hasta aquí querrás probar ya tu nuevo y flamante sistema operativo, pero todavía no hemos terminado. Necesitamos un pequeño fichero que diga al compilador en que posición del archivo dejar cada sección. Esto se hace con un linker script. Creamos link.ld:

```
1 ENTRY(start)  
2  
3 SECTIONS  
4  
5 {  
6  
7 . = 0x00100000;  
8  
9 .multiboot_header :  
10  
11 {  
12  
13 *(.multiboot_header)  
14  
15 }  
16  
17 .text :  
18  
19 {  
20  
21 code = .; _code = .; __code = .;  
22  
23 *(.text)  
24  
25 . = ALIGN(4096);  
26  
27 }  
28  
29 .data :  
30  
31 {  
32  
33 data = .; _data = .; __data = .;  
34  
35 *(.data)  
36  
37 *(.rodata)  
38
```

```

39 . = ALIGN(4096);
40
41 }
42
43 .bss :
44
45 {
46
47 bss = .; _bss = .; __bss = .;
48
49 *(.bss)
50
51 . = ALIGN(4096);
52
53 }
54
55 end = .; _end = .; __end = .;
56
57 }

```

Con esto definimos la posición de cada sección y el punto de entrada, start, que hemos definido en kernel.asm. Ahora ya podemos unir todo este mejunje:

```

1 gcc -o START.ELF kernel.o UtVitam_Main.o -Tlink.ld -nostdlib
  -fPIC -ffreestanding -lgcc

```

Ahora copiamos START.ELF al /next dentro de nuestra carpeta que simula el root de nuestro sistema operativo. Nos dirigimos a la carpeta root de nuestro sistema operativo nuevo con la consola y verificamos que hay dos archivos: uno /boot/grub/grub.cfg y otro /Ut/START.ELF.

Vamos al directorio superior y llamamos a una utilidad de creación ISOs con GRUB llamada grub-mkrescue

```

1 grub-mkrescue -o UtVitam.iso UtVitamroot
2 grub-mkrescue -d /usr/lib/grub/i386-pc/ -o os.iso isofiles

```

Una vez hayamos hecho esto tendremos una ISO. Esta ISO puede abrirse en ordenadores x86 (64 bits también) y máquinas virtuales. Para probarlo, voy a usar QEMU. Llamamos a QEMU desde la línea de comandos:

```

1 qemu-system-i386 nextdivel.iso

```

Arrancará SeaBIOS y más tarde tendremos GRUB. Después si todo va correcto veremos nuestra frase.

## Bibliografía