

SOFTWARE DE BASE

INDICE

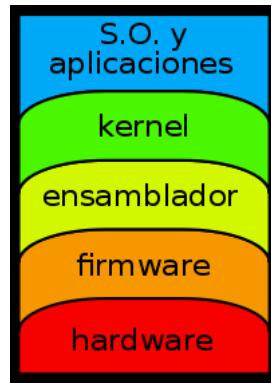
- Software de Base ¿Qué es?
 - Existencia del Software de Base
 - BIOS teoría y ejemplo
 - BIOS-UEFI
 - CoreBoot y EDKII
 - Sumario y Conclusiones
 - Apendices
- Bibliografia

Prefacio

El propósito de este documento es exponer de manera superficial a los ingenieros de computación al *Software de Sistema* o *Software de base*, para concientizar a los lectores sobre "**que es**" lo que hace posible que una computadora pueda ser una herramienta útil programable que pueda manejar y procesar información, así como advertir sobre los posibles problemas que trae la falta de conocimiento sobre este tema de ciencias computacionales, los cuales pueden ser tan graves como volver a una computadora un sistema inmanejable.

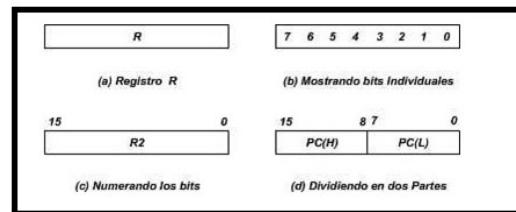
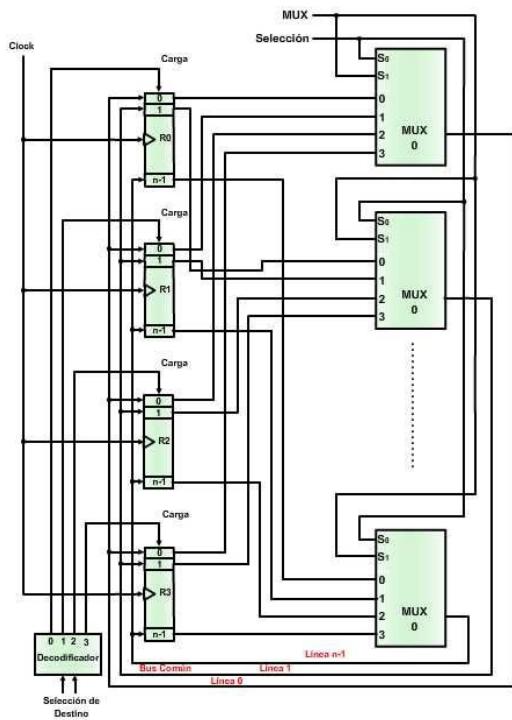
- Software de Base ¿Qué es?

En las ciencias computacionales los temas de arquitectura de computadoras hablan sobre sobre la compuertas lógicas en circuitos electrónicos que agrupados constituyen físicamente una CPU (o un Microprocesador), la implementación de dichos circuitos con transistores pertenecen al área de la Ingeniería Electrónica, la cual se apoya en la teoría de Sistemas Digitales para el diseño de tales componentes del sistema de computo, esta área de conocimiento es el punto de partida para que un Ingeniero en sistemas Computacionales comprenda el funcionamiento del Hardware de un computadora, sobre el cual se maneja un modelo de abstracción de capas para el software, el cual se muestra a continuación:



A partir de la capa del Firmware se comienza a dejar de pensar en las características "físicas del hardware" conforme nos movemos a una capa superior, es decir nos comenzamos a preocupar más en conceptos de programación que van desde el "Ensamblador" (de bajo nivel) hasta los "Lenguajes de Alto Nivel".

Un breve recorrido por las capas mostrara las diferencias en los niveles de abstracción, en la capa de *Hardware*, nos encontramos en A) con esquemáticos propios de los Sistemas Digitales, o Representaciones de los Registros como en B), es usada la Notación de Transferencia de Registros C) cuando se utilizan los lenguajes HDL para el diseño de los componentes digitales físicos.



B) Representación de Registros

Tabla 2.1
Símbolos Básicos para las Transferencia de Registros

Simbolo	Descripción	Ejemplos
Letras y números	Denota un Registro	AR, R2
Parentesis ()	Denota una parte de un Registro	R2(0-7), R2(L)
Flecha ←	Denota Transferencia de Información	R2 ← R1
Coma ,	Separa dos Microoperaciones	R2 ← R1, R1 ← R2
Parentesis []	Especifica una Dirección de Memoria	DR ← M[AR]
Cuadrados []		

C) Notación de Transferencia de Registros (lenguajes HDL)

A) Sistema Digital

En la capa de *Firmware*, toda la información es presentada en formato binario, es posible usar la

representación hexadecimal para facilitar la lectura del contenido de los archivos:

```
(17:50) dmullholl ~/dev
>> ./hexdump -n 128 hexdump.c
 0 | 23 69 6E 63 6C 75 64 65 20 22 61 72 67 73 2E 68 | #include "args.h
10 | 22 0A 23 69 6E 63 6C 75 64 65 20 3C 73 74 64 69 | ".#include <stdi
20 | 6F 2E 68 3E 0A 23 69 6E 63 6C 75 64 65 20 3C 73 | o.h>.#include <s
30 | 74 64 6C 69 62 2E 68 3E 0A 23 69 6E 63 6C 75 64 | tdlib.h>.#includ
40 | 65 20 3C 73 74 64 62 6F 6F 6C 2E 68 3E 0A 23 69 | e <stdbool.h>.#i
50 | 6E 63 6C 75 64 65 20 3C 73 74 64 69 6E 74 2E 68 | nclude <stdint.h
60 | 3E 0A 0A 0A 63 68 61 72 2A 20 68 65 6C 70 74 65 | >...char* helpte
70 | 78 74 20 3D 0A 20 20 20 20 22 55 73 61 67 65 3A | xt =.      "Usage:

(--:--) dmullholl ~/dev
>>
```

Para la capa de **ensamblador** podemos encontrar un "lenguaje" de programación con una estrecha relación a la operación de una CPU, ya es posible en este punto para un programador crear programas para un sistema de computo de una manera mas "humana".

```
-u 100 la
OCFD:0100 BA0B01
OCFD:0103 B409
OCFD:0105 CD21
OCFD:0107 B400
OCFD:0109 CD21
MOV DX,010B
MOV AH,09
INT 21
MOV AH,00
INT 21

-d 10b 13f
OCFD:0100 20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67
OCFD:0110 72 61 60 61 20 68 65 63-68 6F 20 65 6E 20 61 73
OCFD:0120 73 65 60 62 6C 65 72 20-70 61 72 61 20 6C 61 20
OCFD:0130 57 69 6B 69 70 65 64 69-61 24
48 6F 6C 61 2C

Hola,
este es un progr
rama hecho en as
sembler para la
Wikipedia$
```

La capa de *kernel*, encontramos programas especiales que son la base del Sistema Operativo, y usualmente y contamos con Lenguajes de Programacion de Alto nivel que facilitan la creacion del software enormemente, en la figura se muestran a ejemplo dos pedazos de código escritos en C y Rust, para modulos de Kernel.

```
1 #include <linux/module.h>
2 #include <linux/version.h>
3 #include <linux/kernel.h>
4
5 static int __init skm_init(void) /* Constructor */
6 {
7     printk(KERN_INFO "Hello Universe");
8     return 0;
9 }
10
11 static void __exit skm_exit(void) /* Destructor */
12 {
13     printk(KERN_INFO "Bye Bye Universe");
14 }
15
16 module_init(skm_init);
17 module_exit(skm_exit);
18
19 MODULE_LICENSE("GPL");
20 MODULE_AUTHOR("Pradeep Tewani");
21 MODULE_DESCRIPTION("Simple Kernel Module");
```

Lenguaje C

```
.config - Linux/x86 6.2.0 Kernel Configuration
> Search (CONFIG_SAMPLE_HELLO_RUST)

Symbol: SAMPLE_HELLO_RUST [=n]
Type : tristate
Defined at samples/rust/Kconfig:167
Prompt: Rust Hello World module
Depends on: SAMPLES [=n] && SAMPLES_RUST [=n]
Location:
    → Kernel hacking
(1)   → Sample kernel code (SAMPLES [=n])
        → Rust samples (SAMPLES_RUST [=n])
        → Rust Hello World module (SAMPLE_HELLO_RUST [=n])
```

Lenguaje Rust

Por ultimo la capa superior se ejecuta el Sistema Operativo, esta capa es donde usualmente

interactua la mayoria de usuarios de una computadora, el SO permite lanzar programas ya sea en un ambiente grafico o en modo Consola (terminal de caracteres) que pueden ayudan a completar diversas tareas tipicas de una computadora (almacenaje de informacion, reproduccion de video, impresion de documentos, etc). Tambien permite la programación de tareas mediante el uso de interpretes de comandos como BASH o Phyton, los cuales son lenguajes de alto nivel interpretados que "no requieren compilación" para ejecutarse.

```

Processes: 421 total, 3 running, 418 sleeping, 1283 threads
CPU usage: 1.49%, 1.20, 1.11 CPU usage: 0.48% user, 0.84% sys, 98.67% idle
Shared Libraries: 1004 loaded
Memory: 512M total, 324M used, 188M free
VM: 2441G
Networks: 0
Disk: 244G
PhysicalMem: 5 Processes: 421 total, 3 running, 418 sleeping, 1283 threads
CPU usage: 1.49%, 1.20, 1.11 CPU usage: 0.48% user, 0.84% sys, 98.67% idle
Shared Libraries: 1004 loaded
Memory: 512M total, 324M used, 188M free
VM: 2441G
Networks: 0
Disk: 244G
PhysicalMem: 5

```

Interprete de ordenes (BASH o CMD)



Interprete de Python

La tendencia es incrementar el numero de capas de abstraccion tal que la interaccion usuario sistema sea mas directa a la realizada con un ser humano, actualmente ya contamos con aplicaciones que permiten aceptar comandos por voz, puede que en un futuro el reconocimiento de imagenes se anexe como algo cotidiano.

- Existencia del Software de Base

El modelo de abstraccion de capas de software permite mejorar la interaccion del usuario con la computadora, actualmente se requiere que el nivel minimo de interaccion sea en la capa de Sistema Operativo.



Sistema Operativo Escritorio Windows



Sistema Operativo Escritorio Linux

La instalacion y posterior "carga" del Sistema Operativo cada que encienda e inicie el sistema de computo es posible gracias al software BIOS (Basic Input Output System).



Dicho software puede ser de varios tipos GRUB, Linux Boot, UEFI-BIOS, BIOS legacy, etc. El cual puede estar formado por uno o varios programas, este conjunto de programas se le conoce como Software de Base, aparte de "cargar" al Sistema Operativo en cada inicio del sistema también se encarga de "gestionar" la comunicación entre el Sistema Operativo y el Hardware, esto evita que los programadores de aplicaciones de la ultima capa se olviden sobre los detalles del funcionamiento del hardware y se centren en programar solo lo que la aplicación debe hacer para el usuario, por ejemplo, si se desea realizar una comunicación entre procesos mediante sockets en java.

The screenshot shows a Java application window titled "Chat Version Premium" with fields for "IP:" and "Mensaje:". Below the window is a terminal window titled "Adaptador de Ethernet Ethernet 2:" displaying network configuration details:

```

Sufijo DNS específico para la conexión. . .
Vínculo: dirección IPv6 local. . . : fe80::b836:9af7:5508:5435%19
Dirección IPv4 de configuración automática: 169.254.17.218
Máscara de subred . . . . . : 255.255.0.0
Puerta de enlace predeterminada . . . . :

```

On the left, there is a Java code snippet for a thread that listens on port 3122 and receives data from a DatagramSocket. Annotations (A) and (B) highlight specific parts of the code:

```

class Receptor extends Thread
{
    private DatagramSocket socketRecepcion;
    public void run()
    {
        int puertoEntrada=3122; <(A)
        try {
            socketRecepcion=new DatagramSocket(puertoEntrada);

            byte[] buffer=new byte[1024];

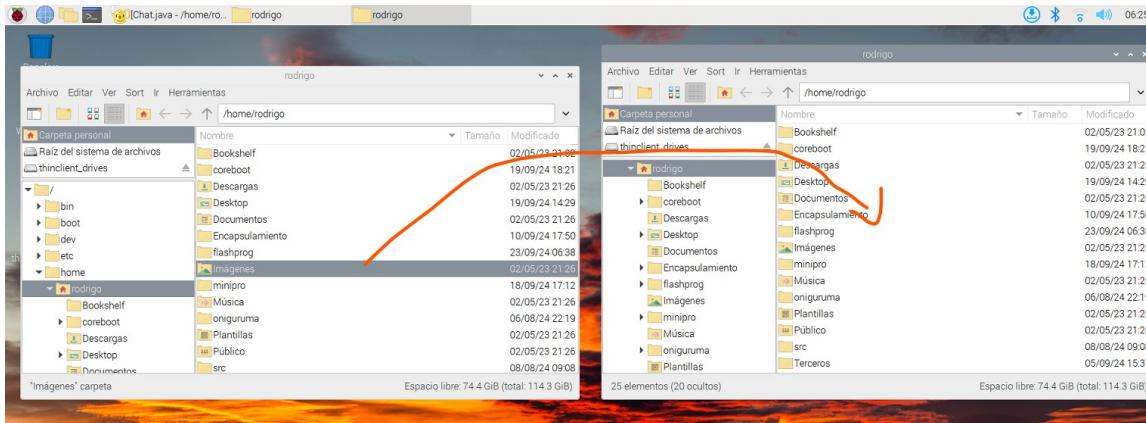
            DatagramPacket dp;
            dp=new DatagramPacket(buffer,buffer.length);

            while(true)
            {
                try {
                    socketRecepcion.receive(dp);
                    areaChat.append("IP emisora: "+dp.getAddress().getHostAddress()+": ");
                    areaChat.append(new String(buffer,0,dp.getLength())+"\n"); <(B)

                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
    }
}

```

Como programadores solo tenemos que preocuparnos por indicar en qué socket enviaremos la información (A) e ingresar la IP correcta en (B), el programa al ejecutarse automáticamente manejará los detalles de la comunicación, lo mismo si nosotros deseamos copiar archivos, simplemente los movemos de una ventana a otra



El sistema operativo se encargara de interpretar la accion como copia de archivos y enviara los mensajes correspondientes a la capa inferior (kernel) y esta a su vez se comunicara con el Firmware (BIOS) y este contactara al dispositivo (en este caso un disco SSD o HDD) para indicarle el movimiento de la informacion.

Es decir el programador en las capas superiores no necesita preocuparse sobre como funciona un disco HDD o uno SSD, este es trabajo del programador de la capa inferior, por ejemplo el que se encarga de escribir el controlador (driver) del disco duro, asi el modelo de capas permite tambien delinear las funciones y responsabilidades de cada capa, asi tendremos entonces un programador de aplicaciones (capa superior), uno de Sistema Operativo, otro para el BIOS y asi sucesivamente, el Software de Base permite entonces dividir la configuracion de un sistema de computo en varias tareas, asi como permitir que se pueda "cargar" un ambiente de Sistema Operativo cada vez que se inicia la computadora.

- BIOS teoria y ejemplo

Antes de comenzar a hablar sobre el código BIOS, se realizara una breve explicacion teorica de lo que es una computadora y como trabaja, esto para poder exponer la razon de ser del programa BIOS, posteriormente se mostrara un ejemplo de arranque usando un sistema Virtual y un BIOS de Código sencillo, se compilara como una prueba del concepto usando una raspberry pi, la cual se ha elegido para reforzar el ejemplo ya que la arquitectura de este sistema es ARM, diferente al sistema emulado x86.

El modelo de Von Neuman

Uno de los aportes mas importantes en las ciencias computacionales fue propuesto por Jhon Von Neuman, y quedo expresado en su diagrama sobre un sistema de computo (figura 1).

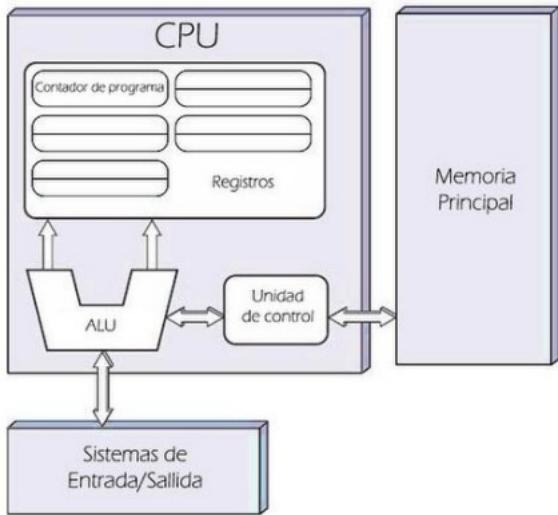


Figura 1

En este modelo la memoria principal tomaba un papel muy importante al ser el lugar donde residia el software, que es como fue llamado al conjunto de instrucciones que la CPU del sistema de computo debia ejecutar, este tipo de sistema es conocido como computadora de programa almacenado. Y su forma de operar se basa en la interaccion que tiene la CPU con la memoria, mediante sus registros, lee y actua acorde a las instrucciones residentes en la memoria y los resultados son guardados tambien en la memoria, por lo que pueden usarse para nuevos calculos, toma de decisiones o como instrucciones nuevas.

La forma de interacción de la CPU con la memoria es dirigida por la Unidad de Control, la cual es una maquina de estados (vista de una manera formal), que ejecuta el ciclo de instrucciones mostrado en la figura 2.

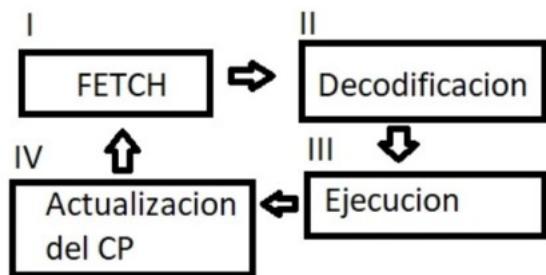


Figura 2

Un recorrido por los Estados seria como el listado siguiente:

- 1) Fetch se busca la información en la memoria de la instrucción a ejecutar, la cual es indicada por el registro "Contador de programa" de la CPU (el como ese registro se cargo con la dirección sera explicado mas delante).
- 2) la cpu Decodifica la informacion que obtuvo de la memoria para averiguar cual instruccion de

las que puede realizar tiene que hacer (dicha lista de instrucciones puede variar entre computadoras diferentes).

3) La CPU "lleva a cabo" (Ejecuta) la tarea indicada por la instrucción, la cual puede ser mover un dato, o realizar una operación.

4) tras finalizar la ejecución se "Actualiza el Registro Contador de Programa (CP)", el cual usualmente incrementa su valor para indicar que la celda siguiente contigua en memoria contiene la siguiente instrucción.

La CPU se mantiene moviéndose en los estados del ciclo de fetch mientras esté encendida y operando, similar a un motor de combustión interna de cuatro tiempos, el cual inclusive cuando está esperando el vehículo en un alto, este ciclo no se detiene.

Para que este modelo funcione, se deben tener ciertas consideraciones iniciales, primero el programa a ejecutar debe estar residente en la memoria de la computadora (el término coloquial es cargado). Los registros de la CPU deben tener valores que permitan la transición de los estados del ciclo de fetch, el más importante sería que el registro "Contador del Programa" contenga la dirección de inicio de la primera instrucción del programa a ejecutar. Surge la primera pregunta, ¿quién coloca el programa en la memoria? y la segunda pregunta, Quién inicializa los registros para que la CPU inicie el ciclo de fetch y así comienza la ejecución del programa?.

La respuesta a la primera pregunta sería que un programa ejecutado por la misma CPU podría realizar la tarea de colocar las instrucciones en la memoria del programa que se desea ejecutar y ese programa al terminar la carga de las instrucciones, sería encargado de actualizar los valores de los registros para dejar todo preparado para la ejecución del nuevo programa recién cargado en la memoria, es decir la misma CPU debería cargar los programas que va a ejecutar. La pregunta se vuelve al problema inicial ¿quién carga a este programa "cargador" de programas? , este problema donde la CPU intenta "iniciarse así misma", se le conoce como BootStrap, la solución es usar un "mecanismo externo" que permita preparar la CPU para "valerse por sí misma", a este proceso "externo" se le conoce como **Boot**.

La primer parte del proceso de Boot, inicializar la CPU al estado en el que pueda ejecutar el ciclo de Fetch se resuelve usando una máquina de estados externa cuyo único trabajo sea configurar los registros de la CPU (usualmente implementada en un circuito digital especializado en la CPU). Donde el valor del registro contador del programa (CP) es fijado a una dirección inicial acordada con antelación por el grupo de diseño de la CPU.

La segunda parte es "proveer de antemano" el primer programa a ejecutar ya "cargado" en celdas de memoria contiguas a la dirección inicial en el CP y reside en un tipo de memoria que es "persistente" para que siempre esté "listo" y no requiera ser "cargado" nuevamente. Este programa tiene como objetivo poder "cargar" otros programas e inicializar los registros de la CPU para que comience a ejecutar dichos programas así como otras tareas adicionales, dicho programa se le conoce como BIOS.

Tenemos que la "razon de ser" del programa BIOS es formar parte del proceso de Boot de una CPU, el cual permite tener un sistema listo para cargar programas en memoria y configura los registros de la CPU para que pueda comenzar el ciclo de fetch la ejecución de estos nuevos programas.

Una explicación teorica mas detallada de este proceso se puede obtener del libro del Doctor Guillermo Levinne (información en la sección de Bibliografia de este documento), los siguientes párrafos se ocuparan de mostrar los aspectos técnicos de la implementación del proceso de BOOT en Hardware usando como ejemplo la implementación propuesta por la arquitectura x86.

Proceso de Boot (Arranque) de arquitectura x86.

Construimos un modelo que representa una arquitectura "Ficticia" con el fin de poder explicar mejor el proceso Técnico que es bastante complejo, así una mejora moderna del modelo de Von Neuman, consiste en la adición del módulo de acceso directo a memoria.

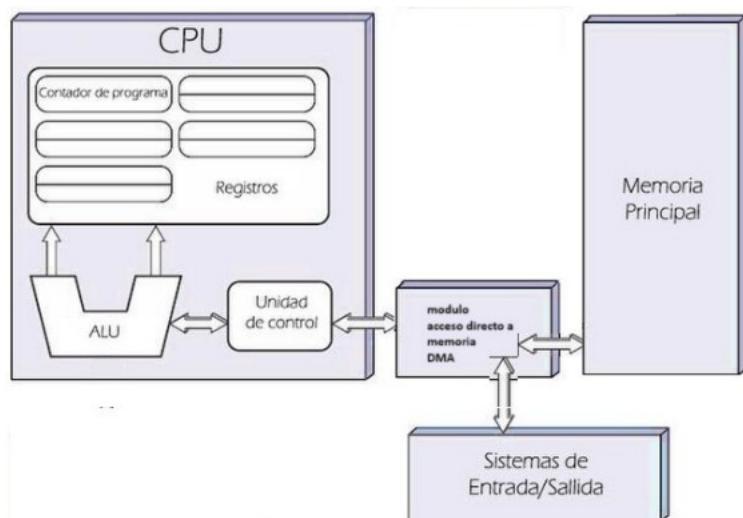


Figura 3 (no es una arquitectura real)

De esta manera la interacción de periféricos y el CPU se realiza mediante regiones especiales de memoria (Esto es parcialmente cierto). Esto significa que el CPU solo puede acceder a regiones de memoria, las cuales pueden ser casillas en RAM, casillas en ROM, el teclado, el monitor o algún otro dispositivo periférico más complejo, en este esquema toma importancia en el concepto de mapa de la Memoria (Memory Map) para el proceso de arranque (Boot). Tomando el siguiente párrafo del manual Minimal Boot publicado por Intel:

Power-Up (Reset Vector) Handling

When an IA bootstrap processor (BSP) powers on, the first address that is fetched and executed is at physical address **0xFFFFFFFF0**, also known as the reset vector. This accesses the ROM / Flash device at the top of the **ROM - 0x10**. The boot loader must always contain a jump to the initialization code in these top 16 bytes.

La dirección marcada con verde, sería la cual acordaron los diseñadores que sería la primera dirección que se colocaría en el Registro Contador de Programa (CP), que como vimos anteriormente es requerida "a priori" para la ejecución del ciclo de Fetch. Como marca el documento Minimal Boot, la dirección 0x10 es colocada en nuestra flash por el programa Enlazador como lo muestra la línea 25 del script de "enlazado" (bios.ld en el apéndice A parte 2):

```

20 } >ROM      /* Places this first section at the beginning of the ROM */
21 /* the --gap-fill option of objcopy will be used to fill the gap to .main */
22 .main main_address : {
23     *(main)
24 }
25 .reset 4096M - 0x10 : {      /* First instruction executed after reset */
26     *(reset)
27 }
```

El vector de reset está indicado por la etiqueta .reset en nuestro código del archivo bios.S (Apéndice A parte1) como se muestra en la línea 186:

```

183 ****
184 /* reset: this section must reside at 0xffffffff0, and be exactly 16 bytes */
185 ****
186 .reset reset, "ax"
187     /* Issue a manual jmp to work around a binutils bug.
188     * See coreboot's src/cpu/x86/16bit/reset16.inc
189     */
190     .byte 0xe9
191     .int  init - ( . + 2 )
192     .align 16, 0xff /* fills section to end of ROM (with 0xFF)
193 ****
```

Otro acuerdo hecho por los diseñadores fue el siguiente:

Initial Processor Mode

When the processor is first powered-on, it will be in a special mode similar to Real Mode, but with the top 12 address lines being asserted high, allowing boot code to be accessed directly from NVRAM (physical address 0xFFFFxxxx). Upon execution of the first long jump, these 12 address lines will be driven according to instructions by firmware. If one of the Protected Modes is not entered before the first long jump, the processor will enter Real Mode, with only 1MB of addressability. In order for Real Mode to work without memory, the chipset needs to be able to alias memory below 1MB to just below 4GB, to continue to access NVRAM. Some chipsets do not have this aliasing and a forcible switch to a normal operating mode will be required before performing the first long jump.

Figura del Documento Minimal Boot Intel

Como se explica ahí la memoria RAM no se encuentra disponible aún y como vimos anteriormente el ciclo de Fetch depende de que el programa almacenado se encuentre residente en la memoria, por lo que la única memoria disponible es la ROM y justamente contiene el código que necesitamos para inicializar la memoria RAM (y demás dispositivos), esto no es coincidencia si no producto del modelo (cita del Doctor Levinne), también debemos acomodar nuestro código en la ubicación indicada, lo cual es expresado al Enlazador en el script bios.ld en la línea 5:

```

1 OUTPUT_ARCH(i8086)           /* i386 for 32 bit, i8086 for 16 bit      */
2
3 /* Set the variable below to the address you want the "main" section, from bios.S, */
4 /* to be located. The BIOS should be located at the area just below 4GB (4096 MB). */
5 main_address = 4096M - 4K;        /* Use the last 4K block             */
6
7 /* Set the BIOS size below (both locations) according to your target flash size   */
8 MEMORY {
9     ROM (rx) : org = 4096M - 512K, len = 512K
10 }

```

Tambien debemos asegurar que nuestro código tenga una longitud de 512 kb. En este punto nuestra CPU esta lista para ejecutar el program BIOS, ha salidode reset, se encuentra en Modo Real, el registro contador de programa (CP) ha sido inicializado apuntando a la direccion de la memoria donde esta ubicado nuestro programa BIOS ya almacenado, por lo que puede comenzar a ejecutar el ciclo de fetch. En el código ejemplo el hardware que incializaremos sera el Super IO para mostrar un mensaje en pantalla y asi indicar que la CPU esta lista, se ha resuelto el proceso BootStrap, mas no ha terminado todo el proceso de inicializacion Boot, aun que funcional nuestro sistema es aun muy "primitivo".

Damos por terminido el proceso de bootstrap de la arquitectura x86, se da acontinuacion una explicacion somera de lo que hace el resto de codigo del archivo bios.S, remarcando que desde aqui deberia de comenzar el código propiamente del BIOS.

Para inicializar nuestro Chipset (SuperIO), se encuentra con el alias SUPERIO_BASE (linea 76 archivo bios.S):

```

75     init_superio:
76         mov dx, SUPERIO_BASE    /* The PC97338 datasheet says we are supposed    */
77         in al, dx    /* to read this port twice on startup, but the    */
78         in al, dx    /* VMware virtual chip doesn't seem to care... */

```

Cuya direccion de entrada esta indicada en la linea 30 del archivo bios.S:

```

29     /* The VMware platform uses an emulated NS PC97338 as SuperIO          */
30     SUPERIO_BASE = 0x2e    /* Do NOT believe what you see in the BIOS bootblock:    */
31     |    /* the VMware SuperIO base is 0x2e and not 0x398.           */

```

Esta direccion es la indicada por el fabricante del Chip en su hoja de datos :

TABLE 8. INDEX and DATA Register Address Options and Configuration Register Accessibility

BADDR Pin		INDEX Register Address	DATA Register Address	Accessible after Reset
1	0	398h	399h	Yes
0	1	undefined	undefined	No ^a
1	0	15Ch	15Dh	Yes
1	1	2Eh	2Fh	Yes

a. Apply Plug and Play protocol.

Esto significa que para poder acceder al Super IO chip debemos acceder a su direccion en memoria:

```

64  /* 'init' doesn't have to be at the beginning, so you can move it around, as      */
65  /* long as remains reachable, with a short jump, from the .reset section.          */
66  .section main, "ax"
67  .globl init      /* init must be declared global for the linker and must */
68  init:           /* point to the first instruction of your code section */
69  cli      /* NOTE: This sample BIOS runs with interrupts disabled */
70  cld      /* String direction lookup: forward */
71  (A) { mov ax, cs /* A real BIOS would keep a copy of ax, dx as well as */
72  mov ds, ax /* initialize fs, gs and possibly a GDT for protected */
73  mov ss, ax /* mode. We don't do any of this here. */
74
75  init_superio:
76  (B) { mov dx, SUPERIO_BASE /* The PC97338 datasheet says we are supposed */
77  in al, dx /* to read this port twice on startup, but the */
78  in al, dx /* VMware virtual chip doesn't seem to care... */
79
80  /* Feed the SuperIO configuration values from a data section */
81  mov si, offset superio_conf /* Don't forget the 'offset' here! */
82  mov cx, (serial_conf - superio_conf)/2

```

Cargamos la direccion incial del programa BIOS en el par de registros DS:SS (A)para despues cargar la direccion del Super IO en la seccion "baja" del registro (B), asi ahora el conjunto DS:SS contienen la direccion segmento desplazamiento para "ubicar" al SuperIO.

Con el proposito de enviar un mensaje (cadena de texto guardada en el segmento e datos con la etiqueta hello string) por el puerto serial, debemos seguir las instrucciones de la hoja de datos del Super IOS PC397338, la cual indica la direccion de memoria donde esta el dispositivo que maneja la salida serial, la cual usualmente esta conectada a un circuito interno dentro del Super IO que convierte señales seriales en RS232 (en ocasiones puede ser un IC externo) y las envia por un cabezal de Debug, el apendice B comenta mas sobre el protocolo RS232.

```

89  init_serial:      /* Init serial port
90  mov si, offset serial_conf
91  mov cx, hello_string - serial_conf)/2
92 write_serial_conf:
93  mov ax, [si]
94  ROM_CALL serial_out
95  add si, 0x02
96  loop write_serial_conf ←
97
98 print_hello:      /* Print a string
99  mov si, offset hello_string
100 ROM_CALL print_string

```

Como muestra el fragmento de codigo del archivo bios.S desde la linea 89 a la 100 realizamos un proceso de bajo nivel para mover un caracter a la vez y enviarlo hacia el puerto de salida (linea 94), mediante un Loop que continua hasta que hallamos leido y enviado los caracteres del mensaje alojado en el segmento de datos(B):

```

164 /* Data: */  

165 /******  

166 superio_conf:  

167 /* http://www.datasheetcatalog.org/datasheet/nationalsemiconductor/PC97338.pdf */  

168 .byte PC97338_FER, 0x0f /* Enable COM, PAR and FDC */  

169 .byte PC97338_FAR, 0x10 /* LPT=378, COM1=3F8, COM2=2F8 */  

170 .byte PC97338_PTR, 0x00 /* Make sure COM1 test mode is cleared */  

171 serial_conf: /* See http://www.versalogic.com/kb/KB.asp?KBID=1395 */  

172 .byte COM_MCR, 0x00 /* RTS/DTS off, disable loopback */  

173 .byte COM_FCR, 0x07 /* Enable & reset FIFOs. DMA mode 0. */  

174 .byte COM_LCR, 0x80 /* Set DLAB (access baudrate registers) */  

175 .byte COM_BRD_LO, 0x01 /* Baud Rate 115200 = 0x0001 */  

176 .byte COM_BRD_HI, 0x00  

177 .byte COM_LCR, 0x03 /* Unset DLAB. Set 8N1 mode */  

178 hello_string: /*(B)*/  

179 .string "\r\nHello BIOS world!\r\n" /* .string adds a NUL terminator */  

180 /*****
```

El inciso (A) muestra los datos que son necesarios para que el Super IO active a su circuito interno que realiza la conversion Serial a RS232.

Aqui concluye la explicacion somera de lo que hace el codigo de ejemplo del archivo bios.S al inicio de estos parrafos es donde deberia de comenzar el código de nuestro BIOS. En conclusion, un circuito electronico se encarga de encender y "sacar" de Reset a nuestra CPU, apartir de ahí nuestro CPU comenzara a ejecutar instrucciones de una direccion de memoria especifica, como dicha direccion no cambia simplemente debemos asegurar que el inicio de nuestro codigo (etiqueta .section main linea 66 del archivo bios.S) se encuentre en esa direccion, para esto le indicamos al programa enlazador mediante un script el punto de entrada requerido, dicho script aunque tiene alguna complejidad tecnica, no cambiara mucho durante el proceso de diseño de nuestro BIOS, por lo que basta con preocuparse solo algunas veces, por lo que debemos centrarnos mas en como debemos inicializar nuestro hardware, lo cual require seguir las especificaciones de las hojas de datos de los fabricantes del hardware conectado a nuestro sistema, utilizando programacion bajo nivel en lenguaje ensamblador, el siguiente capitulo muestra como se implementan los archivos bios.S y bios.Id para generar un archivo bios.rom el cual puede probarse en maquina virtual y comprobar la efectividad de lo mostrado, mas delante usaremos el ambiente de desarrollo del proyecto libre Coreboot el cual nos permitira abstraer mas los detalles tecnicos y enfocarnos en el desarrollo de un BIOS mas completo.

Proceso de Compilacion

Debido a que estamos usando una Raspeberry Pi como estacion de diseño, necesitamos realizar una Cross-Compilacion de ARMx64 (microprocesador usado en la tarjeta) para generar código x64_32/64, así como una maquina virtual para probar el programa ejecutable.

El proceso de compilacion debera hacerse por partes, debido a que debemos de configurar varios aspectos técnicos, la figura muestra los comandos requeridos para llevar acabo cada etapa.

```

rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ ls Inicio  

bios_BackUP.rom bios.ld bios.rom bios.S Makefile PDF_Docs  

rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ x86_64-w64-mingw32-gcc -c -o bios.o -m32 bios.S 1  

rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ x86_64-w64-mingw32-ld -nostartfile -Tbios.ld -o bios.out bios.o -Map xMemLayout.map 2  

rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ x86_64-w64-mingw32-objcopy -O binary -j .begin -j .main -j .reset --gap-fill=0x0eff bios.out bios.rom 3  

rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ ls Final  

bios_BackUP.rom bios.ld bios.o bios.out bios.rom bios.S Makefile PDF_Docs xMemLayout.map  

rodrigo@raspberrypi:~/Documents/vmbios-1.0 $ ls
```

Lista de comandos requerida:

1)**x86_64-mingw32-gcc -c -o bios.o -m32 bios.S**

Creamos el archivo objeto bios.o a partir del archivo fuente (en ensamblador) bio.S

2)**x86_64-mingw32-ld -nostartfile -Tbios.ld -o bios.out bios.o -Map xMemLayout.map**

Enlazamos el archivo ajustando a direcciones fijas de memoria (a partir del inicio del programa), para cumplir con las especificaciones del Boot Strap para arquitectura X86, usando un listado (script) de enlazado bios.ld y se le aplica al archivo objeto bios.o, generando el archivo bios.out.

3)**x86_64-mingw32-objcopy -O binary -j .begin -j .main -j .reset --gap-fill=0x0ff bios.out bios.rom**

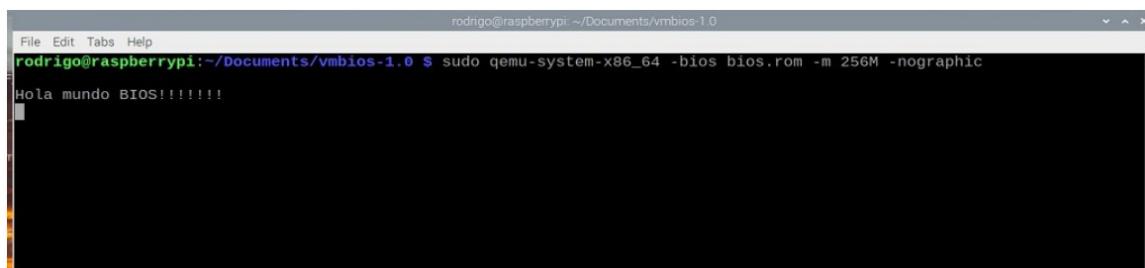
Vaciamos el contenido binario del archivo bios.out con la herramienta objcopy para construir el sistema de archivos que es compatible con las memorias SP(donde se alojara nuestro BIOS),esto ayuda a llenar el espacio requerido para la particion con 0. El resultado de la compilacion es un archivo bios.rom, el cual puede cargarse directamente en la memoria SPI fisica en la plataforma especifica para la cual construimos el BIOS (El cual no es un programa general). O podemos usarlo para inicializar nuestra maquina virtual.

Inicializar la maquina virtual con un BIOS especifico.

Las maquinas virtuales tienen la opción de emular su funcionamiento con un bios especifico, para ello debemos indicarle donde reside el archivo que contiene el programa de arranque que usara el mecanismo de Boot Strap de la arquitectura especifica a emular (En este caso x86_32/64), se muestra el comando para inicializa una maquina virtual con Qemu y nuestro archivo bios.rom.

```
sudo qemu-system-x86_64 -bios bios.rom -m 256M -nographic
```

Nos genera la ventana:



Donde automaticamente redirigimos la salida del puerto de debug (RS232) de la VM a la terminal de trabajo.

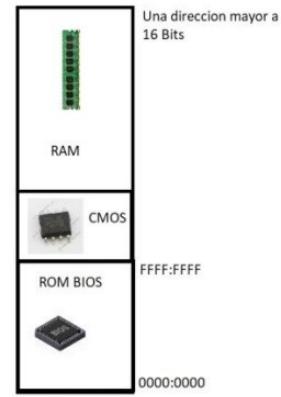
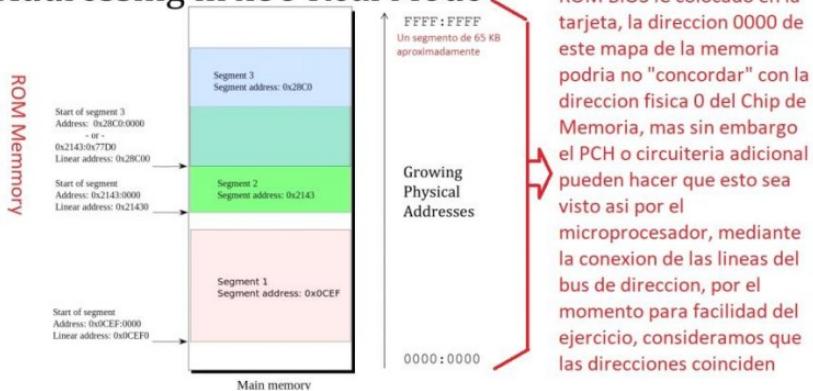
- BIOS-UEFI

En el capitulo anterior vimos que el BIOS es el programa encargado de inicializar el hardware a

bajo nivel de una tarjeta madre, una vez funcional el hardware el programa comenzara a cargar al Sistema Operativo del equipo de computo, el cual hara uso de los servicios del recien inicializado Hardware (como lo indica el esquema de arquitectura de software por capas).

Durante decadas se manejo la idea de un BIOS monolitico cuya funcion era inicializar el hardware y cargar el sistema operativo, asi como el BIOS tenia una direccion de entrada 0xFFFFFFFF0, entonces era comun que el sistema operativo tambien tuvieran una dirección especifica, ya que recordemos que la CPU solo ve el mapa de la memoria, pero no sabe si esta dirección, es una localidad de RAM, ROM, USB o Disco, (lo cual es propio del modelo de Von Neuman).

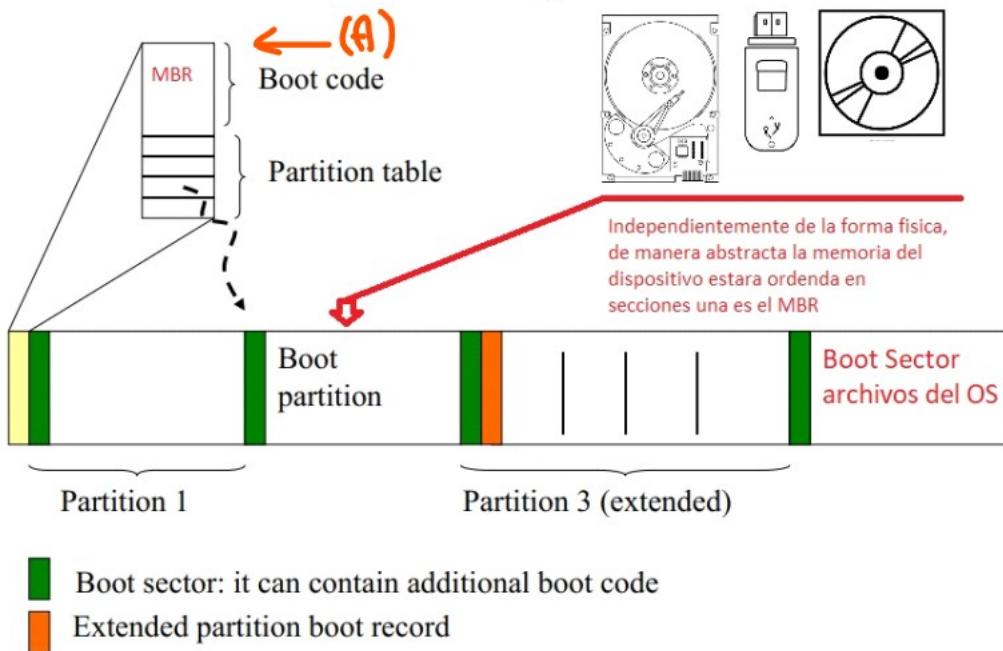
Addressing in x86 Real Mode



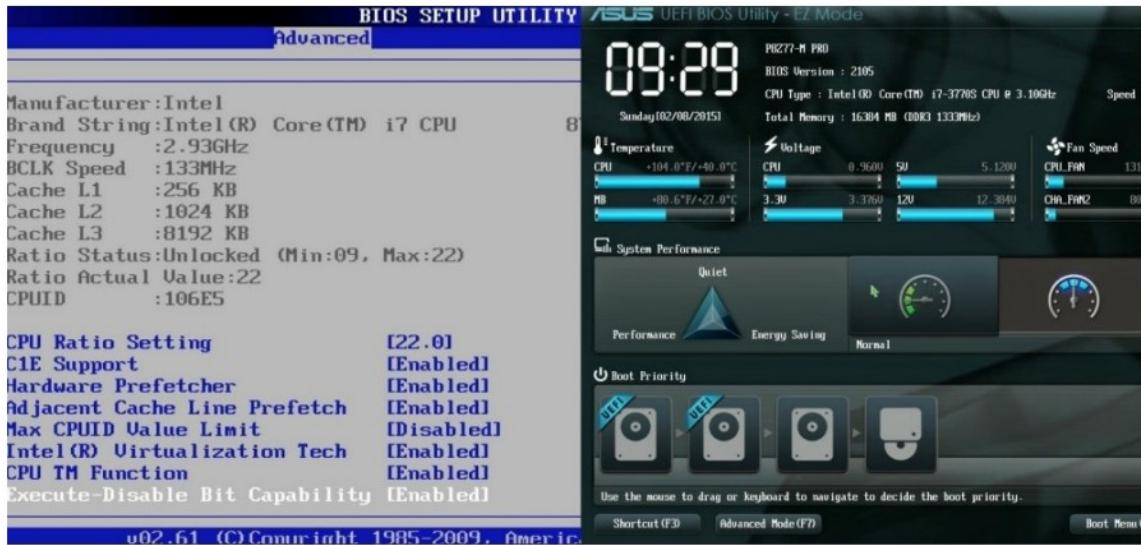
Mapa de la memoria (PCH = Platform Control Hub, dispositivo igual a Super IO)

Esa sección de la memoria se le conoce como el MBR (Master BOOT Record). Inciso (A) de la siguiente figura, se "agrega" esta porción de la memoria, al mapa de la memoria de nuestra computadora (el término coloquial es "se mapea").

The Device Organization



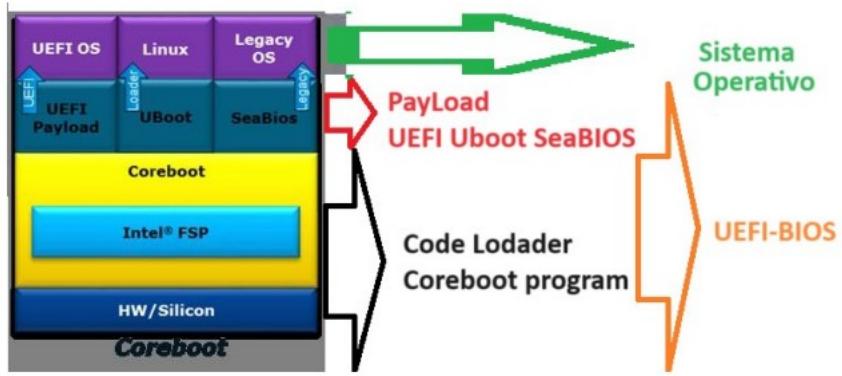
La sección de la memoria del dispositivo que almacena al Sistema Operativo, contendrá una región llamada el MBR, donde el BIOS hallará el nombre del archivo del sistema operativo que ha de ejecutar primero para que este comience su ejecución, el cual estará ubicado en el "Boot Sector", una vez que comience a ejecutarse el sistema operativo, el BIOS "le cederá el control" y este será el único programa que se ejecutara durante el tiempo que la computadora esté encendida y operando, el usuario tratará con el sistema operativo para pedir cargar sus aplicaciones, administrar sus archivos y manejar los dispositivos. Hasta este punto el esquema de abstracción por capas funciona perfectamente. La siguiente figura muestra a la izquierda un menú de un BIOS Legacy (monolítico) descrito anteriormente, la parte derecha muestra un BIOS-UEFI (o simplemente UEFI).



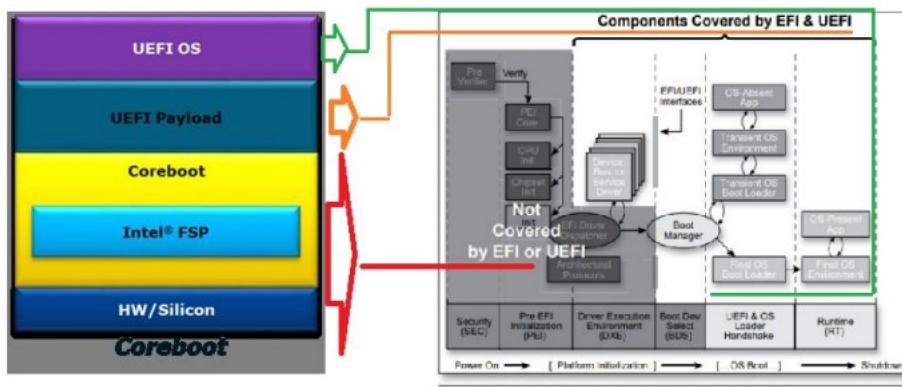
A parte de las apariencias graficas entre BIOS, existe una ventaja adicional que provee UEFI, como dijimos en el sistema de capas el BIOS Legado era un programa usualmente escrito en ensamblador para inicializar el Hardware y cargar al Sistema Operativo, el problema es que si se desea agregar "nuevo" hardware o mejorar el software de acceso a los dispositivos, todo el BIOS debe escribirse y compilarse nuevamente, debido a que existe una restriccion de tamaño, un ejemplo seria si el codigo del BIOS indica que despues de la direccion de inicio 0xFFFFFFFFF0 en la direccion 0xFFFFFFFFFA por decir esta la direccion del Sistema Operativo, quiere decir que el código del BIOS no puede estar mas alla de dicha dirección tendra que reubicarse y esto conyeva volver a escribir el código y recomilar (un ejemplo muy extremo).

Siguiendo la lógica del modelo de capas entonces seria posible dividir al BIOS en partes que puedan tener mas flexibilidad de cambios y a su vez se puedan escribir en lenguajes de alto nivel para facilitar el proceso de diseño. Esto dio origen al concepto BIOS-UEFI donde el BIOS se "particiona" en varias etapas de software, unas incializaran el hardware minimo requerido para poder ejecutar programas en "Modo Real" y sobre esta capa se cargara software que pueda aceptar cambios y modificaciones mas facilmente que el concepto Monolitico anteriormente usado. Asi pues Unified Extensible Firmware Interface (UEFI) es el software que que acepta cambios en el BIOS "mas facilmente" que el modelo monolitico y esta sobre una capa de software compuesta de un BIOS minimalista.

Existen varios software de base que cumple la tarea del "BIOS minimalista" entre ellos mencionamos U-Boot, Open-Bios, Libreboot, por mencionar algunos, y de los que permite implementar una especificación UEFI (), tenemos a SlimBoot y Coreboot como algunos candidatos, para este documento escogeremos este ultimo, el cual permite implementar como una "payload" varios tipos de especificaciones para "cargadores de sistema operativo", observe la siguiente figura:

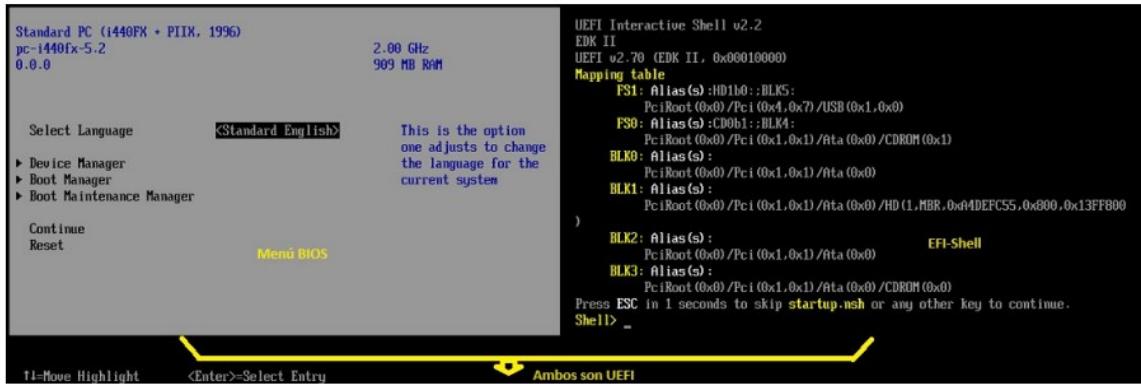


Usando CoreBoot como base Elegiremos a UEFI como payload por lo que tendremos un esquema como la parte izquierda de la siguiente figura:



La sección derecha de la figura anterior muestra a detalle los servicios que "espera" la implementación UEFI (sombreado oscuro linea roja) y los servicios que "provee"(linea naranja) dicha implementación, podemos ver que algunos servicios de UEFI, se "traslapan" (línea verde) con la especificación UEFI, dichos servicios no son proporcionados por la sección UEFIPayload, mas bien creados a partir de esta en "tiempo de ejecución del sistema operativo" y acompañaran al sistema operativo mientras este "corriendo", esta característica es una de los beneficios de la migración del anterior BIOS PC de IBM (que mencionamos anteriormente).

El mayor beneficio es que el esquema BIOS-UEFI nos proporciona el ambiente de desarrollo y pruebas UEFI-Shell (a la derecha de la siguiente figura):



Para contar con una maquina virtual donde probar UEFI-Shell revise el apendice D.

EFI-Shell

La interface no es muy diferente a una Ventana de Comandos en Windows o a la terminal Bash de Linux, compuesta de un conjunto de comandos, los cuales pueden agruparse en Scripts para realizar tareas mas complejas. Usando la instrucción `help` se nos presenta una lista de comandos disponibles por *defacto*.

```
Shell> help
alias      - Displays, creates, or deletes UEFI Shell aliases.
attrib     - Displays or modifies the attributes of files or directories.
bcfg      - Manages the boot and driver options that are stored in NVRAM.
cd        - Displays or changes the current directory.
cls       - Clears the console output and optionally changes the background
and foreground color.
```

Si desesamos informacion de un comando en especifico escribimos `help comando`, como se muestra en la imagen:

```

Shell> help alias
Displays, creates, or deletes UEFI Shell aliases.

ALIAS [-d|-v] [alias-name] [command-name]

-d - Deletes an alias. Command-name must not be specified.
-v - Makes the alias volatile.
alias-name - Specifies an alias name.
command-name - Specifies an original command's name or path.

NOTES:
1. This command displays, creates, or deletes aliases in the UEFI Shell
environment.
2. An alias provides a new name for an existing UEFI Shell
command or UEFI application. Once the alias is created, it can be used
to run the command or launch the UEFI application.
3. There are some aliases that are predefined in the UEFI Shell environment.
These aliases provide the MS-DOS and UNIX equivalent names for the file
manipulation commands.
4. Aliases will be retained even after exiting the shell unless the -v option
is specified. If -v is specified then the alias will not be valid after
leaving the shell.

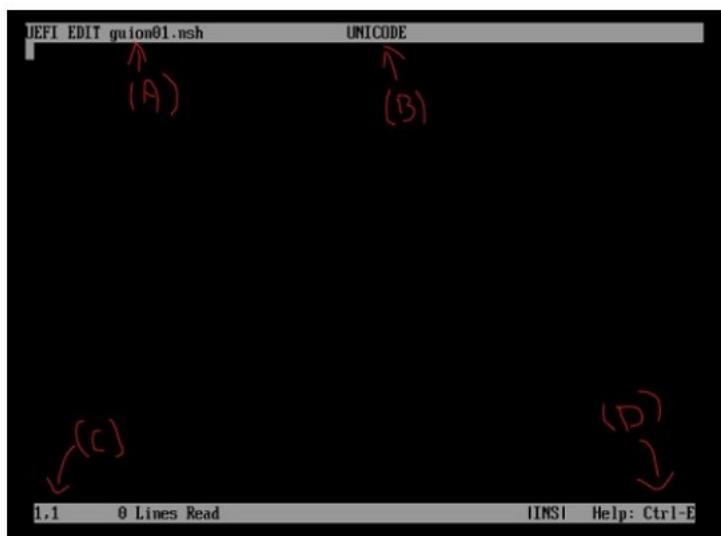
EXAMPLES:
* To display all aliases in the UEFI Shell environment:

```

Shell Scripting en EFI

Como hemos observado con forme usemos los comandos de EFI-Shell, llegara un punto en que necesitaremos repetir varias veces una lista de secuencia de instrucciones, por lo que se necesita una manera eficiente de tratar el problema de la repetitividad, lo cual es muy usual en tareas de mantenimiento.

Cuenta con un editor sencillo para crear nuestros Scripts, invocado con la instrucción `edit <nombreScript>.nsh`



Describimos brevemente el area de edicion (A) el nombre del archivo que estamos editando, (B) el juego de caracteres usado (consulte help para mayor informacion), (C) El indicador en pares LINEA-COLUMNA, (D) La combinación de teclas para entrar al menu del editor que nos brinda mas informacion sobre el uso del programa. Escribimos lo siguiente:



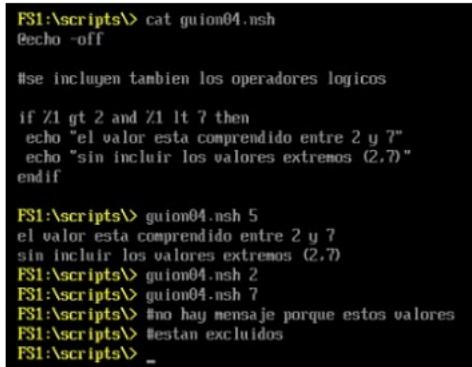
```
UEFI EDIT guion01.nsh          UNICODE          Modified
echo "hola mundo"
-
```

Presionamos ctrl + s para guardar los cambios del archivo (puede que necesite presionar la tecla de flecha hacia abajo para que se muestre el menu de guardado en la parte inferior derecha del area de edicion) y ctrl + q para salir del editor, en el CLI (Comand Line Interface), Podemos ejecutar el script simplemente con su nombre:



```
F1:\scripts> .\guion01.nsh
F1:\scripts> echo "hola mundo"
hola mundo
```

Podemos escribir Scripts sencillos usando el editor y el conjunto de instrucciones provistas por el UEFI-Shell:



```
F1:\scripts> cat guion04.nsh
echo -off

#se incluyen tambien los operadores logicos

if %1 gt 2 and %1 lt 7 then
echo "el valor esta comprendido entre 2 y 7"
echo "sin incluir los valores extremos (2,7)"
endif

F1:\scripts> guion04.nsh 5
el valor esta comprendido entre 2 y 7
sin incluir los valores extremos (2,7)
F1:\scripts> guion04.nsh 2
F1:\scripts> guion04.nsh 7
F1:\scripts> #no hay mensaje porque estos valores
F1:\scripts> #estan excluidos
F1:\scripts>
```

Otra ventaja de usar el esquema BIOS-UEFI es que podemos agregar mas instrucciones para ejecución en UEFI-Shell o de manera automatica durante la carga de BIOS-UEFI, esto nos permite extender la funcionalidad de nuestro Software de Base (que era la principal limitante del BIOS Legacy). Mas delante en este documento se expone dicha posibilidad.

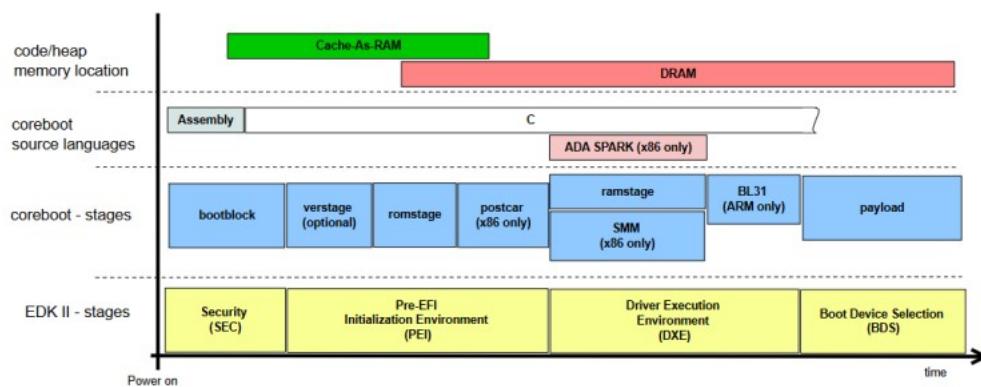
- CoreBoot y EDKII

En los parrafos anteriores (BIOS teoria y ejemplo) fuimos desde una explicación teorica, pasando por un ejemplo real, el cual se describio a grandes rasgos y se compilo, para probar en un ambiente virtual, aunque el código generaba un BIOS demasiado sencillo, sirve para intentar conectar la teoria y la practica. Dicho código de ejemplo no es suficiente para funcionar como un BIOS Legacy y mucho menos para soportar un ambiente UEFI, como la creación de un BIOS es un proceso complejo existen proyectos libres compuestos con varias personas que se encargan de realizar esta tarea dividiendolo en varias etapas acorde al modelo de abstraccion de

software por capas, a continuación se describen las partes que conforman un CoreBoot BIOS.

Introducción al proyecto CoreBoot

El proyecto busca generar código abierto para la inicialización de sistemas de computo modernos, su filosofía es hacer una cosa y hacerla bien, por lo que solamente se encarga de la etapa básica de inicialización del sistema, para permitir la carga de un Sistema Operativo, se puede agregar como Payloads a la compilación del archivo Binario otro código de BIOS complementarios como SEABIOS o LibreBoot, también permite código propietario en formato de archivos BLOBS (archivos binarios crudos) de vendedores como Intel y AMD. La siguiente Figura muestra un diagrama de tiempos de los procesos de ejecución del software de base que inicializa un sistema de computo, podemos ver como se listan las etapas de CoreBoot con respecto a las de EDKII (UEFI) por ejemplo.



Coreboot está armado con varias etapas, las cuales son compiladas en archivos Binarios separados y posteriormente "pegadas" usando el mecanismo de compresión personalizado CBFS (CoreBoot File System, el cual no es un sistema de archivos real). Excepto la etapa de BootBlock (escrita en Ensamblador). Las etapas de Coreboot se describen brevemente a continuación:

Bootblock

La primera etapa en ser ejecutada tras que la CPU sale de reset o sleep mode. Esta escrita en ensamblador y su principal función es preparar todo para poder "cargar" un ambiente para ejecutar C (CPU pasa a modo Real, se crea una Pila, etc). Las tareas principales en esta etapa son las siguientes:

- Cache-As-RAM para el área de "heap" y la "stack" (pila).
- Ajustar el apuntador de Pila
- Limpiar la memoria para el BSS

-Descomprimir y cargar la siguiente etapa (Load the next estage)

En maquinas con arquitectura x86 se realiza adicional:

°Actualizacion del Microcode

°Inicialización del Timmer

°Cambiar del modo Real de 16-bit hacia el Modo Protegido 32-bit (o 64 segun el caso) Bootblock cargara la etapa Verstage si esta habilitada (esta etapa es opcional) o pasara a la siguiente Roomstage.

Bootblock

La primera etapa en ser ejecutada tras que la CPU sale de reset o sleep mode. Esta escrita en ensamblador y su principal función es preparar todo para poder "cargar" un ambiente para ejecutar C (CPU pasa a modo Real, se crea una Pila, etc).

Las tareas principales en esta etapa son las siguientes:

-Cache-As-RAM para el area de "heap" y la "stack" (pila).

-Ajustar el apuntador de Pila

-Limpiar la memoria para el BSS

-Descomprimir y cargar la siguiente etapa (Load the next estage)

En maquinas con arquitectura x86 se realiza adicional:

°Actualizacion del Microcode

°Inicialización del Timmer

°Cambiar del modo Real de 16-bit hacia el Modo Protegido 32-bit (o 64 segun el caso)

Bootblock cargara la etapa Verstage si esta habilitada (esta etapa es opcional) o pasara a la siguiente Roomstage.

Cache-As-Ram

Cache como RAM, debido a que la memoria actual es bastante compleja de inicializar usando simplemente Ensamblador, se deja para una etapa posterior, por lo que en su defecto se utiliza una parte de la memoria Cache del Microprocesador para cargar el BIOS. La cual se activa

usando la especificación de cada vendedor de CPUs. Las siguientes etapas se ejecutan cuando la CAR (Cache as Ram) está activa:

bootblock romstage verstage postcar

Verstage

En esta etapa comienzan los mecanismos de seguridad, se ejecutan código de seguridad y se realizan las verificaciones correspondientes del BIOS y del Hardware, no puede ser modificado en campo (aquí es donde entra la TPM). Los mecanismos de verificación de arranque seguro permiten confiar en actualizaciones de Firmware en campo combinadas con un sistema de recuperación "seguro".

Romstage

La RAM es inicializada y se prepara todo para inicializar los dispositivos. Tareas Comunes:

°Preinicialización de Dispositivos

°Inicialización de la DRAM

Postcar

Para dejar la configuración CAR y ejecutar código en la DRAM regular, esta etapa desactiva al CAR y carga el código en la RAM, la etapa es pequeña en comparación con las otras etapas.

Ramstage

Comienza inicialización principal de los dispositivos:

°Inicialización de dispositivos PCI

°Inicalización de dispositivos "On-chip"

°Inicialización de la TPM (sin no fue inicializada en la etapa Verstage)

°Inicialización de Graficos es opcional.

°Inicialización del CPU (Como ajustar el SMM)

Tras la inicialización de las tablas de sistema correspondientes se le informa a la siguiente etapa Payload (la cual es opcional) o al Sistema Operativo. El informe incluye:

°ACPI tables (x86 specific)

°SMBIOS tables (x86 specific)

- °coreboot tables
- °devicetree updates (ARM specific)
- °Tambien asegura (bloquea) el hardware and firmware :
 - Protección contra escritura del boot media
 - Bloque los registros relacionados (importantes)
 - Asegura el modo SMM(x86 specific)

Payload

Este código se ejecuta tras que Coreboot a terminado. Reside dentro del CBFS y no hay posibilidad de escogerlo durante el tiempo de ejecución. Usualmente aqui se carga un programa BIOS mas completo como SEABIOS o LibreBoot, para complementar el proceso de BOOT y ofrecer mas servicios, en maquinas x86 es posible cargar la especificacion UEFI.

EDKII Explicando la implementacion de la especificacion UEFI

Se usara el siguiente programa de Ejemplo:

```

1 //efi.c programa simple que muestra en pantalla un mensaje, hay que reiniciar o apagar el sistema
2 //para terminar el programa
3 #include "efi.h"
4
5 //Punto de Entrada de nuestro programa, Note que no existe funcion main()
6 EFI_STATUS efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable) { ← El inicio de nuestro programa no es la
7   //PORDEFINIR: definir parametros de entrada de linea de comandos.
8   (void)ImageHandle, (void)SystemTable; //Esta linea evita advertencias del compilador
9
10  // Reset Console Output Reinicia al PROTOCOLO de salida a Consola
11  SystemTable->ConOut->Reset(SystemTable->ConOut, false); → Reset, para inicializar la salida de Texto
12  Simple
13
14  //CLEAR console outpout (Borra la pantalla y por efecto colateral
15  // configura el color de fondo a Negro y Reubica el cursor a la posicion LINEA 0 COLUMN 0)
16  SystemTable->ConOut->ClearScreen(SystemTable->ConOut); → ClearScreen, limpiar pantalla para que nuestro
17  // mensaje se vea claramente
18  SystemTable->ConOut->OutputString(SystemTable->ConOut, u"TESTING, Hello UEFI World!\r\n");
19  // Envíar el mensaje a la pantalla
20
21  // Loop Infinito para ver el mensaje, para terminar hay que apagar el sistema
22  while (1); → El programa esperara oiosamente mostrando el mensaje
23
24  return EFI_SUCCESS;
25 }
```

Archivo efi.c, para ver el contenido del archivo efi.h revise el appendice C, describimos brevemente las lineas de código:

#3 Archivo efi.h, donde estan definidas las Tablas y los Protocolos usados tal cual como marca la especificacion UEFI.

#6 El punto de inicio de nuestro programa no es la funcion Main(), esto lo marca tambien la especificación.

#8 Aunque no hemos implementado todos los Protocolos de la especificacion, debemos incluirlos,

ya que los sistemas con UEFI esperan que esten presentes, aunque no se usen, por lo que debemos incluirlos aunque sea "simbolicamente".

Protocolos usados:

#11 de la Tabla SystemTable, buscamos al miembro ConOut el cual es tambien una Tabla compuesta por varios elementos, buscamos al que lleve el nombre de Reset el cual es un Protocolo por lo que ingresamos los parametros requeridos. hay que notar que uno de los Parametros parece una referencia circular "SystemTable->ConOut", es raro ver este tipo de instrucciones en Lenguajes de Alto Nivel, mantenga en mente que estamos trabajando en bajo Nivel, pero el Lenguaje C lo "oculta" para evitar que nos preocupemos en cumplir con la especificacion de Arquitectura del Microprocesador.

#15 Este Protocolo limpia la pantalla y reubica nuestro cursor, esto ayuda a que nuestro mensaje se muestre con una mejor presentación.

#18 El Protocolo que envia la cadena de Texto: "Hello UEFI World!" completando la tarea que deseabamos hacer.

#21 El bucle infinito mantiene al sistema en un estado Ocioso para que podamos apreciar el mensaje.

Para compilar el código usamos nuestro Cross Compiler con los parametros:

```
x86_64-w64-mingw32-gcc -WI,--subsystem,10 -e efi_main -std=c17 -Wall -Wextra -
```

```
Wpedantic -mno-red-zone -ffreestanding -nostdlib -o Hello.EFI efi.c
```

Como podemos ver se pueden incluir código estandar de C, esto nos permite crear procesos y funciones como se realiza normalmente con el lenguaje, dado que estamos en una etapa Pre OS contamos con un nivel de acceso privilegiado a los recursos del sistema, no existen muchos mecanismos que impidan realizar acciones maliciosas que afecten al HARDWARE, si usted adquiere un programa de aplicacion .EFI de terceros ya compilado listo para su uso, podría contener código injectado, por esta razón se recomienda usar programas de fuentes confiables y es usual que este sea presentado en formato código fuente, para que pueda ser inspeccionado antes de compilarlo para asi asegurar que no presentará algún problema de seguridad. Para mayor información revise el documento de la Especificación UEFI en la pagina de EDKII.

- Sumario y Conclusiones

En los parrafos anteriores (BIOS teoria y ejemplo)fuimos desde una explicación teorica, pasando por un ejemplo real, el cual se describio a grandes razgos y se compilo, para probar en un ambiente virtual, aunque el código generaba un BIOS demasiado sencillo, sirve para intentar conectar la teoria y la practica.

Resumiendo lo anterior podemos ver que la programación en UEFI es igual la programacion normal en C, donde se añaden los Protocolos cuando requerimos acceder a algun recurso del Hardware del sistema, como enviar mensajes hacia la pantalla o pedir entradas del teclado, acceso a dispositivos de almacenamiento externo o caracteristicas fisicas del Hardware como puertos PCI o I2C, como requisito para poder realizar pruebas de Diagnostico, las cuales son posibles debido a que estamos en un nivel privilegiado sin la supervision del OS, Esta facilidad es proporcionada por las definiciones de las Tablas de acceso definidas en el archivo efi.h, el cual es el que crea la comodidad de programar en un ambiente Estandarizado, solo basta con familiarizarse con los Protocolos Disponibles y agruparlos para construir nuestras herramientas de Software.

A diferencia del BIOS Legacy Monolitico, el BIOS-UEFI es mas flexible permite cambios posteriores en el Software de Base generando servicios adicionales (programas UEFI), dichos servicios pueden "pasarse" al Sistema Operativo como indica la especificación UEFI, esto extiende las capacidades tanto del Sotware de Base como las del Sistema Operativo.

- Apendice

Apendice A

1ra parte Código fuente BIOS Sencillo

Extraido del blog del autor:

<https://pete.akeo.ie/2011/06/crafting-bios-from-scratch.html>

```
1  ****  
2  /*          VMware BIOS ROM example          */  
3  /*      Copyright (c) 2011 Pete Batard (pete@akeo.ie) - Public Domain */  
4  ****  
5  
6  
7  ****  
8  /* GNU Assembler Settings:                      */  
9  ****  
10 .intel syntax noprefix /* Use Intel assembler syntax (same as IDA Pro) */  
11 .code16             /* After reset, the x86 CPU is in real / 16 bit mode */  
12 ****  
13  
14  
15 ****  
16 /* Macros:                                     */  
17 ****  
18 /* This macro allows stackless subroutine calls */  
19 .macro ROM_CALL addr  
20     mov sp, offset lf /* Use a local label as we don't know the size */  
21     jmp \addr /* of the jmp instruction (can be 2 or 3 bytes) */  
22 l: /* see http://sourceware.org/binutils/docs-2.21/as/Symbol-Names.html */  
23 .endm
```

```

26  /*************************************************************************/
27  /* Constants: */
28  /*************************************************************************/
29  /* The VMware platform uses an emulated NS PC97338 as SuperIO */
30  SUPERIO_BASE = 0x2e /* Do NOT believe what you see in the BIOS bootblock: */
31  /* the VMware SuperIO base is 0x2e and not 0x398. */
32  PC97338_FER = 0x00 /* PC97338 Function Enable Register */
33  PC97338_FAR = 0x01 /* PC97338 Function Address Register */
34  PC97338_PTR = 0x02 /* PC97338 Power and Test Register */
35
36 /* 16650 UART setup */
37 COM_BASE = 0x3f8 /* Our default COM1 base, after SuperIO init */
38 COM_RB = 0x00 /* Receive Buffer (R) */
39 COM_TB = 0x00 /* Transmit Buffer (W) */
40 COM_BRD_LO = 0x00 /* Baud Rate Divisor LSB (when bit 7 of LCR is set) */
41 COM_BRD_HI = 0x01 /* Daud Rate Divisor MSB (when bit 7 of LCR is set) */
42 COM_IER = 0x01 /* Interrupt Enable Register */
43 COM_FCR = 0x02 /* 16650 FIFO Control Register (W) */
44 COM_LCR = 0x03 /* Line Control Register */
45 COM_MCR = 0x04 /* Modem Control Registrer */
46 COM_LSR = 0x05 /* Line Status Register */
47 /*************************************************************************/
48
49
50 /*************************************************************************/
51 /* begin : Dummy section marking the very start of the BIOS. */
52 /* This allows the .rom binary to be filled to the right size with objcopy. */
53 /*************************************************************************/
54 .section begin, "a" /* The 'ALLOC' flag is needed for objcopy */
55 .ascii "VMBIOS v1.00" /* Dummy ID string */
56 .align 16
57 /*************************************************************************/
58
59
60 /*************************************************************************/
61 /* main: */
62 /* This section will be relocated according to the bios.ld script. */
63 /*************************************************************************/
64 /* 'init' doesn't have to be at the beginning, so you can move it around, as */
65 /* long as remains reachable, with a short jump, from the .reset section. */
66 .section main, "ax"
67 .globl init /* init must be declared global for the linker and must */
68 init: /* point to the first instruction of your code section */
69     cli /* NOTE: This sample BIOS runs with interrupts disabled */
70     cld /* String direction lookup: forward */
71     mov ax, cs /* A real BIOS would keep a copy of ax, dx as well as */
72     mov ds, ax /* initialize fs, gs and possibly a GDT for protected */
73     mov ss, ax /* mode. We don't do any of this here. */
74
75 init_superio:
76     mov dx, SUPERIO_BASE /* The PC97338 datasheet says we are supposed */
77     in al, dx /* to read this port twice on startup, but the */
78     in al, dx /* VMware virtual chip doesn't seem to care... */
79
80     /* Feed the SuperIO configuration values from a data section */
81     mov si, offset superio_conf /* Don't forget the 'offset' here! */
82     mov cx, (serial_conf - superio_conf)/2
83 write_superio_conf:
84     mov ax, [si]
85     ROM_CALL superio_out
86     add si, 0x02
87     loop write_superio_conf
88
89 init_serial: /* Init serial port */
90     mov si, offset serial_conf
91     mov cx, (hello_string - serial_conf)/2

```

```

92     write_serial_conf:
93         mov ax, [si]
94         ROM_CALL serial_out
95         add si, 0x02
96         loop write_serial_conf
97
98     print_hello:      /* Print a string
99         mov si, offset hello_string
100        ROM_CALL print_string
101
102    serial_repeater: /* End the BIOS with a simple serial repeater */
103        ROM_CALL readchar
104        ROM_CALL putchar
105        jmp serial_repeater
106
107    /*************************************************************************/
108    /* Subroutines: */
109    /*************************************************************************/
110    superio_out:      /* AL (IN): Register index, AH (IN): Data to write */
111        mov dx, SUPERIO_BASE
112        out dx, al
113        inc dx
114        xchg al, ah
115        out dx, al
116        jmp sp
117
118
119    serial_out:      /* AL (IN): COM Register index, AH (IN): Data to Write */
120        mov dx, COM_BASE
121        add dl, al /* Unless something is wrong, we won't overflow to DH */
122        mov al, ah
123        out dx, al
124        jmp sp
125
126
127    putchar:          /* AL (IN): character to print
128        mov dx, COM_BASE + COM_LSR
129        mov ah, al
130    tx_wait:
131        in al, dx
132        and al, 0x20 /* Check that transmit register is empty */
133        jz tx_wait
134        mov dx, COM_BASE + COM_TB
135        mov al, ah
136        out dx, al
137        jmp sp
138
139
140    readchar:         /* AL (OUT): character read from serial */
141        mov dx, COM_BASE + COM_LSR
142    rx_wait:
143        in al, dx
144        and al, 0x01
145        jz rx_wait
146        mov dx, COM_BASE + COM_RB
147        in al, dx
148        jmp sp

```

```

151 print_string:      /* SI (IN): offset to NUL terminated string      */
152     lodsb
153     or    al, al
154     jnz   write_char
155     jmp   sp
156 write_char:
157     shl   esp, 0x10 /* We're calling a sub from a sub => preserve SP      */
158     ROM_CALL putchar
159     shr   esp, 0x10 /* Restore SP                                         */
160     jmp   print_string
161
162
163 /***** Data: ****/
164 /* Data: */
165 /***** ****/
166 superio_conf:
167 /* http://www.datasheetcatalog.org/datasheet/nationalsemiconductor/PC97338.pdf */
168     .byte PC97338_FER, 0x0f /* Enable COM, FAR and FDC          */
169     .byte PC97338_FAR, 0x10 /* LPT=378, COM1=3F8, COM2=2F8      */
170     .byte PC97338_PTR, 0x00 /* Make sure COM1 test mode is cleared */
171 serial_conf: /* See http://www.versalogic.com/kb/KB.asp?KBID=1395 */
172     .byte COM_MCR, 0x00 /* RTS/DTS off, disable loopback   */
173     .byte COM_FCR, 0x07 /* Enable & reset FIFOs. DMA mode 0. */
174     .byte COM_LCR, 0x80 /* Set DLAB (access baudrate registers) */
175     .byte COM_BRD_LO, 0x01 /* Baud Rate 115200 = 0x0001        */
176     .byte COM_BRD_HI, 0x00
177     .byte COM_LCR, 0x03 /* Unset DLAB. Set 8N1 mode         */
178 hello_string:
179     .string "\r\nHello BIOS world!\r\n" /* .string adds a NUL terminator */
180 /***** ****/

```



```

183 /***** ****/
184 /* reset: this section must reside at 0xffffffff0, and be exactly 16 bytes */
185 /***** ****/
186 .section reset, "ax"
187     /* Issue a manual jmp to work around a binutils bug.           */
188     /* See coreboot's src/cpu/x86/16bit/reset16.inc                 */
189     .byte 0xe9
190     .int init - (. + 2)
191     .align 16, 0xff /* fills section to end of ROM (with 0xFF) */
192 /***** ****/

```

2da parte Código para el Enlazador del Bios Sencillo

Extraido del blog del autor:

<https://pete.akeo.ie/2011/06/crafting-bios-from-scratch.html>

```

1 OUTPUT_ARCH(i8086)           /* i386 for 32 bit, i8086 for 16 bit      */
2
3 /* Set the variable below to the address you want the "main" section, from bios.S, */
4 /* to be located. The BIOS should be located at the area just below 4GB (4096 MB). */
5 main_address = 4096M - 4K;        /* Use the last 4K block                      */
6
7 /* Set the BIOS size below (both locations) according to your target flash size   */
8 MEMORY {
9     ROM (rx) : org = 4096M - 512K, len = 512K
10 }
11
12 /* You shouldn't have to modify anything below this                                */
13 SECTIONS {
14     ENTRY(init)          /* To avoid antivirus false positives      */
15     /* Sanity check on the init entrypoint          */
16     _assert = ASSERT(init >= 4096M - 64K,
17         "'init' entrypoint too low - it needs to reside in the last 64K.");
18     .begin : { /* NB: ld section labels MUST be 6 letters or less      */
19         *(begin)
20     } >ROM          /* Places this first section at the beginning of the ROM */
21     /* the --gap-fill option of objcopy will be used to fill the gap to .main */
22     .main main_address : {
23         *(main)
24     }
25     .reset 4096M - 0x10 : {      /* First instruction executed after reset */
26         *(reset)
27     }
28     .igot 0 : {             /* Required on Linux                         */
29         *(.igot.plt)
30     }
31 }
32

```

Apendice B

Protocolo RS23

El funcionamiento de esta interfaz es bastante sencillo, proviene de las viejas terminales TTY que se usaron para transmitir datos mecanografiados durante la época del telegrafo, posteriormente dieron un salto con los sistemas multiusuario de las primeras computadoras, también formaron parte del origen del sistema ASCII, aunque es un sistema muy confiable tiene un ancho de banda limitado y es probable sea desplazado por alguna otra interfaz como la USB.

La siguiente figura representa como se implementa esta interfaz:

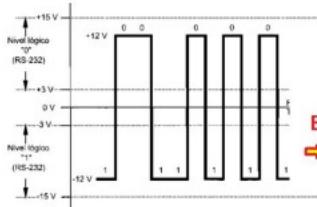


Figura 20-3 Niveles de tensión lógicos para RS232

El código ASCII es representado con valores lógicos típicos de los componentes TTL (A), para posteriormente ser convertidos en valores RS232 (B),

Carácter	ASCII	Carácter	ASCII
A	0100 0001	W	0101 0111
B	0100 0010	X	0101 1000
C	0100 0011	Y	0101 1001
D	0100 0100	Z	0101 1010
E	0100 0101	0	0011 0000
F	0100 0110	1	0011 0001
G	0100 0111	2	0011 0010
H	0100 1000	3	0011 0011
I	0100 1001	4	0011 0100
J	0100 1010	5	0011 0101
K	0100 1011	6	0011 0110
L	0100 1100	7	0011 0111
M	0100 1101	8	0011 1000
N	0100 1110	9	0011 1001
O	0100 1111	+	0010 1011
P	0101 0000	-	0010 1101
Q	0101 0001	*	0010 1010
R	0101 0010	:	0011 1010
S	0101 0011	=	0011 1101
T	0101 0100	<	0011 1100
U	0101 0101	>	0011 1011
V	0101 0110	-	0011 1010

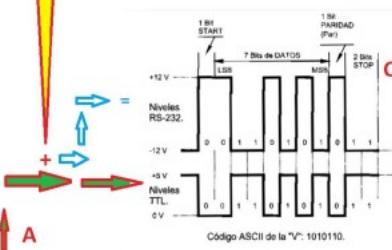
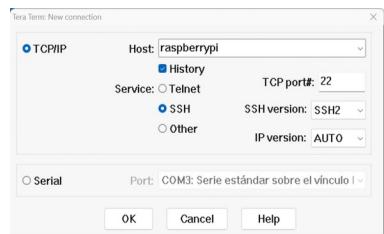


Figura 20-4 Ejemplo de envío de un byte según normas RS232

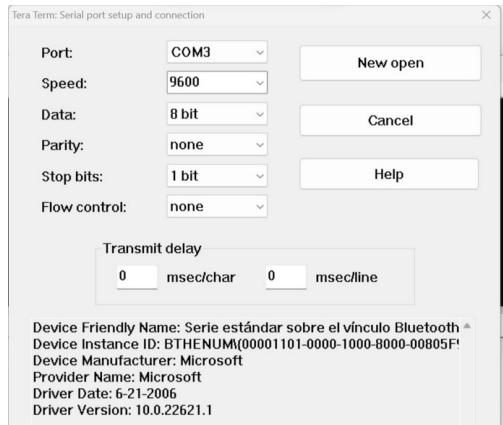
Basicamente el Código ASCII nos indica como se codifican las letras en binario, posteriormente esta señal es convertida a una señal RS232, este tipo de señal pertenece al grupo de pares diferenciales, estas señales exhiben características útiles para la transmisión a distancia, debido a que no cuentan con una línea de reloj la frecuencia de transmisión debe de conocerse de antemano, una medida usada para cuantificar la cantidad de información usada es el Baudio, definido como el numero de bits enviados por segundo.

Como los puertos RS232 no cuentan con linea de reloj existe de antemano una lista de "convenciones" de la velocidad de transmisión, entre las que encontramos 1200, 4800, 9600 115200, etc, los cuales son velocidades muy lentas, pero lo suficientemente rápidas para sostener una interfaz serie de servicio.

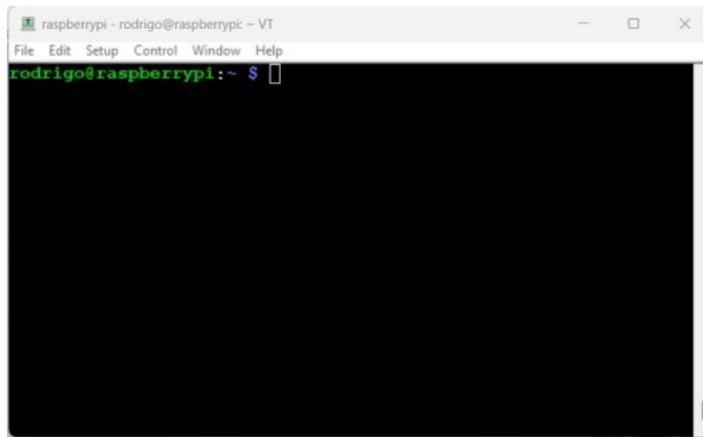
Existen varios software que emula este tipo de comunicación Tera Term, Moba Xterm y Putty por mencionar algunos, basicamente requieren dos cosas, el tipo de conexión que se realizará:



La configuración de la velocidad del transmisión y el puerto en Baudios



Para poder presentar una ventana con una "terminal de caracteres" emulando así una comunicación TTY.



Apendice C

Código completo del archivo de inclusión efi.c

```

2 // No ha sido implementado, pero debe estar presente ya que
3 // la especificacion dicta que asi debe ser
4 #include <stdint.h>
5 #include <stdbool.h>
6 #if __has_include(<uchar.h>)
7 #include <uchar.h>
8#endif
9 //UEFI Especificacion 2.10 seccion 2.4
10#define IN //Ayudas visuales que nos permiten saber el flujo de los datos
11#define OUT //como no cuentan con definicion solo titulo se pueden tratar
12#define OPTIONAL //como etiquetas en el codigo similar a las usadas en Ensamblador
13#define CONST const
14//EFI API define las llamadas de las funciones UEFI
15#define EFIAPI __attribute__((ms_abi)) //Revise la informacion sobre las llamadas x86_64 impuestas por Microsoft
16//Tipos de datos usados en UEFI Especificacion 2.10 seccion 2.3
17//typedef uint8_t UNIT8; //Redefinicion del lenguaje para enmascarar los tipos usados en la especificacion
18typedef uint8_t BOOLEAN; //0 = False 1 = True
19typedef int64_t INTN;
20typedef uint64_t UINTN;
21typedef int8_t INT8;
22typedef uint8_t UINT8;
23typedef int16_t INT16;
24typedef uint16_t UINT16;
25typedef int32_t INT32;
26typedef uint32_t UINT32;
27typedef int64_t INT64;
28typedef uint64_t UINT64;
29typedef char CHAR8;
30//UTF-16 Es el equivalente de tipo , para caracteres del código UCS-2
31//codepoints <= 0xFFFF_FFFF
32#ifndef _UCHAR_H
33    typedef uint_least16_t char16_t;
34#else
35    typedef char16_t CHAR16;
36#endif
37typedef char16_t CHAR16;
38typedef void VOID;
39typedef struct EFI_GUID{ //Mas informacion sobre EFI_GUID en la especificacion UEFI APENDICE A
40    UINT32 TimeLow;
41    UINT16 TimeMid;
42    UINT16 TimeHighAndVersion;
43    UINT8 ClockSeqHighAndReserved;
44    UINT8 ClockSeqLow;
45    UINT8 Node[6];
46}__attribute__((packed)) EFI_GUID;
47typedef UINTN EFI_STATUS;
48typedef VOID *EFI_HANDLE;
49typedef VOID *EFI_EVENT;
50typedef UINT64 EFI_LBA;
51typedef UINTN EFI_TPL;
52//EFI STATUS (Códigos de estado) - UEFI Especificacion 2.10 Appendice D
53#define EFI_SUCCESS 0ULL
54//PORDEFINIR: Agregar EFI_ERROR()para verificar los valores retornados por EFI_STATUS
55//Si el Bit mas alto esta activo , verificar si hubo un estado de ERROR
56//EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL: UEFI Especificacion 2.10 seccion 12.2
57typedef struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL; //Resulta extraña la definicion del
58//del protocolo
59//EFI_SIMPLE_TEXT_RESET_PROTOCOL: UEFI Especificacion 2.10 seccion 12.4.1
60typedef
61EFI_STATUS
62(EFIAPI *EFI_TEXT_RESET) (
63    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
64    IN BOOLEAN ExtendedVerification
65);
66//EFI_SIMPLE_TEXT_STRING_PROTOCOL: : UEFI Especificacion 2.10 seccion 12.4.3
67typedef
68EFI_STATUS
69(EFIAPI *EFI_TEXT_STRING) (
70    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
71    IN CHAR16 *String
72);
73//EFI_SIMPLE_TEXT_CLEAR_SCREEN_PROTOCOL: UEFI Spec 2.10 section 12.4.1
74typedef
75EFI_STATUS
76(EFIAPI *EFI_TEXT_CLEAR_SCREEN) (
77    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This
78);

```

```

79 //EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL: UEFI Spec 2.10 section 12.4.8
80 #ifndef _EfiSimpleTextOutputProtocol_h_
81 #define _EfiSimpleTextOutputProtocol_h_
82
83 #include <Efi.h>
84
85 #include <Protocol/TextServices.h>
86
87 #include <Protocol/TextOutput.h>
88
89 #include <Protocol/TextInput.h>
90
91 #endif // _EfiSimpleTextOutputProtocol_h_
92
93 // EFI_TABLE_HEADER: UEFI Spec 2.10 section 4.2.1
94 #ifndef _EfiTableHeader_h_
95 #define _EfiTableHeader_h_
96
97 #include <Protocol/TextOutput.h>
98
99 #include <Protocol/TextInput.h>
100
101 //EFI_SYSTEM_TABLE: Spec 2.10 section 4.3.1
102 #ifndef _EfiSystemTable_h_
103 #define _EfiSystemTable_h_
104
105 #include <Protocol/TextOutput.h>
106
107 #include <Protocol/TextInput.h>
108
109 #include <Protocol/TextOutput.h>
110
111 #include <Protocol/Runtime.h>
112
113 #include <Protocol/ConfigurationTable.h>
114
115 //EFI_IMAGE_ENTRY_POINT: UEFI Spec 2.10 section 4.1
116 //El compilador arroja este error cuando la funcion MAIN() no es la principal o no existe
117 //Hay que especificar explicitamente cual es la funcion de Inicio del programa principal.
118
119 #ifndef _EfiImageEntryPoint_h_
120 #define _EfiImageEntryPoint_h_
121
122 #endif // _EfiImageEntryPoint_h_

```

// Hay que especificar explicitamente cual es la funcion de Inicio del programa principal.

//efi.h:69:1 warning: 'ms abi' attribute directive ignored [-Wattributes]

//check the MICROSOFT CALLING CONVENTIONS on Wiki

//Nota IN solo indica la direccion del flujo de la informacion, esto es ayuda ilustrativa

// para el programador

Describiremos algunas partes del archivo de Inclusion efi.h

Nombre	Descripción
<code>BOOLEAN</code>	Logical Boolean. 1-byte value containing a 0 for FALSE or a 1 for TRUE. Other values are undefined.
<code>INTN</code>	Signed value of native width. 4 bytes on supported 32-bit processor instructions, 8 bytes on supported 64-bit processor instructions, 16 bytes on supported 128-bit processor instructions.
<code>UINTN</code>	Unsigned value of native width. 4 bytes on supported 32-bit processor instructions, 8 bytes on supported 64-bit processor instructions, 16 bytes on supported 128-bit processor instructions.
<code>INT8</code>	1-byte signed value.
<code>UINT8</code>	1-byte unsigned value.
<code>INT16</code>	2-byte signed value.
<code>UINT16</code>	2-byte unsigned value.
<code>INT32</code>	4-byte signed value.
<code>UINT32</code>	4-byte unsigned value.
<code>INT64</code>	8-byte signed value.
<code>UINT64</code>	8-byte unsigned value.
<code>INT128</code>	16-byte signed value.
<code>CHAR8</code>	1-byte character. Unless otherwise specified, all 1-byte or ASCII characters and strings are stored in 8-bit ASCII encoding format, using the ISO-Latin-1 character set.
<code>CHAR16</code>	2-byte Character. Unless otherwise specified all characters and strings are stored in the UCS-2 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.
<code>VOID</code>	Undeclared type.
<code>EFI_GUID</code>	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned to a 48-bit boundary.
<code>EFI_STATUS</code>	Status code. Type <code>UINTN</code> .
<code>EFI_HANDLE</code>	A collection of related interfaces. Type <code>VOID*</code> .
<code>EFI_CODE</code>	Hexadecimal value. Type <code>UINTN</code> .
<code>EFI_LMA</code>	Logical block address. Type <code>UINT64</code> .
<code>EFI_TPL</code>	Task priority level. Type <code>UINTN</code> .
<code>EFI_MAC_ADDRESS</code>	8-byte buffer containing a network Media Access Control address.
<code>EFI_IPv4_ADDRESS</code>	4-byte buffer containing an IPv4 internet protocol address.
<code>EFI_IPv6_ADDRESS</code>	16-byte buffer containing an IPv6 internet protocol address.
<code>EFI_IPv6_ADDRESS</code>	16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 internet protocol address.

continues on next page

```

1 // NOTA: Usamos "VOID *" para marcar un elemento que aun
2 // No ha sido implementado, pero debe estar presente ya que
3 // la especificacion dice que asi debe ser
4 #include <stdbool.h>
5 #include <stdbool.h>
6 #if __has_include(<uchar.h>)
7 #include <uchar.h>
8 #endif
9 //UEFI Especificacion 2.10 sección 2.4
10 #define IN     //Ayudas visuales que nos permiten saber el flujo de los datos
11 #define OUT    //como no cuentan con definición solo título se pueden tratar
12 #define OPTIONAL //como etiquetas en el código similar a las usadas en ensamblador
13 #define CONST const
14 //EFIAPI define las llamadas de las funciones UEFI
15 #define EFIAPI __attribute__((ms_abi)) //Revisa la información sobre las llamadas x64 e impuestas por Microsoft (A)
16 //Las definiciones usadas en UEFI Especificacion 2.10 sección 2.3. Se indica la sección donde se puede encontrar la información en el documento
17 #typedef uint8_t UNIT8; //Redefinición del lenguaje para emascarar los tipos usados en la especificación
18 #typedef uint8_t BOOLEAN; //0 = False 1 = True
19 #typedef int8_t INTN;
20 #typedef uint16_t UINT16; //Definimos los nombres de los tipos datos que lleva la especificación,
21 #typedef int8_t INT8; //con el tamaño en bytes que deben tener, para hacer esto, realizamos una
22 #typedef uint8_t UINT8; //sustitución de los nombres usando TYPEDEF, esto ayuda a que el código se
23 #typedef int16_t INT16; //asemeje lo mas posible al mostrado en la Especificación.
24 #typedef uint16_t UINT16;
25 #typedef int32_t INT32;
26 #typedef uint32_t UINT32;
27 #typedef int64_t INT64;
28 #typedef uint64_t UINT64;
29 #define char CHAR8;
30 //UINT16 Es el equivalente de tipo , para caracteres del código UCS-2
31 //codepoints <= 0xFFFF FFFF

```

Las definiciones con solo ayudas visuales al programador, se indican con un cuadro naranja, el compilador sustituye el código no introduce nada debido a que no hay definición

Las definiciones con solo ayudas visuales al programador, se indican con un cuadro naranja, el compilador sustituye el código similar a las usadas en ensamblador

//Ayudas visuales que nos permiten saber el flujo de los datos

//como no cuentan con definición solo título se pueden tratar

//como etiquetas en el código similar a las usadas en ensamblador

//Redefinición del lenguaje para emascarar los tipos usados en la especificación

//0 = False 1 = True

//Definiciones usadas en UEFI Especificacion 2.10 sección 2.3. Se indica la sección donde se puede encontrar la información en el documento

//UINT16 Es el equivalente de tipo , para caracteres del código UCS-2

Se describirán solo algunas secciones del código debido a que la especificación ya provee información completa sobre el tema, la idea es mostrar como se puede implementar lo estipulado en el documento.

En la figura anterior se resalta la traducción de algunos tipos de datos usados en nuestro programa para que concuerden con los nombres en la especificación, esto ayuda a "normalizar" el código.

```

31 //codepoints <= 0xFFFF_FFFF
32 #ifndef _UCHAR_H
33     typedef uint_least16_t char16_t;
34 #else
35     typedef char16_t CHAR16; ➔ typedef existing-type new-type;
36 #endif
37     typedef char16_t CHAR16;
38     typedef void VOID;
39     typedef struct EFI_GUID; //Mas informacion sobre EFI_GUID en la especificacion UEFI APENDICE A
40     UINT32 TimeLow;
41     UINT16 TimeMid;
42     UINT16 TimeHighAndVersion;
43     UINT8 ClockSeqHighAndReserved;
44     UINT8 ClockSeqLow;
45     UINT8 Node[6];
46 }__attribute__((packed)) EFI_GUID;
47     typedef UINTN EFI_STATUS;
48     typedef VOID *EFI_HANDLE;
49     typedef VOID *EFI_EVENT;
50     typedef UINT64 EFI_LBA;
51     typedef UINTN EFI_TPL;

```

Acorde a la definición:
El tipo existente char16_t
Ahora se llama también CHAR16

La definición se aplica como sigue:
`typedef struct EFI_GUID (...) EFI_GUID;`

Esto evita tener que usar el prefijo struct cada que se haga referencia a la variable EFI_GUID, también facilita la notación de acceso a los elementos si queremos acceder a `TimeLow`, podemos hacerlo así:
`EFI_GUID->TimeLow.`

Note también que `TimeLow` es de tipo `UINT32` el cual es una redefinición del tipo `uint32_t`

Dichos tipos fueron redefinidos entre las líneas 18 a 29 en el archivo efi.h

Los cuales son usados para, a su vez definir las "Tablas" de datos mas complejos que marca la especificación.

```

79 //EFI SIMPLE TEXT OUTPUT_PROTOCOL: UEFI Spec 2.10 section 12.4.8
80 typedef struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
81     EFI_TEXT_RESET Reset;
82     EFI_TEXT_STRING OutputString;
83     void *TestString;
84     void *QueryMode;
85     void *SetMode;
86     void *SetAttribute;
87     EFI_TEXT_CLEAR_SCREEN ClearScreen;
88     void *SetCursorPosition;
89     void *EnableCursor;
90     void *Mode;
91 } EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;

```

Continuamos redefiniendo los tipos de datos estructurados complejos a nombres mas simples de acceso, esto aplica también a los PROTOCOLOS
struct `EFI_SIMPLE_TEXT_OUTPUT` ahora es simplemente `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`
Sus elementos apuntan a la ubicación donde deben ir los datos donde los espera el Protocolo para poder funcionar, Note que los elementos marcados con VOID no han sido implementados, pero están incluidos como marca la especificación

Lo mismo ocurre con los PROTOCOLOS, son definidos como una "Tabla" compuesta de varios elementos.

```

57 typedef struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;
58 //EFI_SIMPLE_TEXT_RESET_PROTOCOL: UEFI Especificacion 2.10 seccion 12.4.1
59 typedef EFI_STATUS
60 (EFIAPI *EFI_TEXT_RESET) (
61     IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
62     IN BOOLEAN ExtendedVerification
63 );
64 //EFI_SIMPLE_TEXT_STRING_PROTOCOL: : UEFI Especificacion 2.10 seccion 12.4.3
65 typedef EFI_STATUS
66 (EFIAPI *EFI_TEXT_STRING) (
67     IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
68     IN CHAR16 *String
69 );
70 //EFI_SIMPLE_TEXT_CLEAR_SCREEN_PROTOCOL: UEFI Spec 2.10 section 12.4.1
71 typedef EFI_STATUS
72 (EFIAPI *EFI_TEXT_CLEAR_SCREEN) (
73     IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
74     ...
75 );
76 // Write String Protocol que permite mostrar cadenas de texto simple en el CLI
77 IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL
78 );

```

Lista de las definiciones de los tres PROTOCOLOS usados en nuestro programa:
`EFI_SIMPLE_TEXT_RESET_PROTOCOL`
`EFI_SIMPLE_TEXT_STRING`
`EFI_SIMPLE_TEXT_CLEAR_SCREEN`

// Reset Console Output Reinicia al PROTOCOLO de salida a Consola
`SystemTable->ConOut->Reset(SystemTable->ConOut, false);`

// CLEAR console output (Borra la pantalla y por efecto colateral // configura el color de fondo a Negro, y Reabica el cursor a la posicion LINEA 0 COLUMN 0)
`SystemTable->ConOut->ClearScreen(SystemTable->ConOut);`

// Write String Protocol que permite mostrar cadenas de texto simple en el CLI
`SystemTable->ConOut->OutputString(SystemTable->ConOut, u"TESTING, Hello UEFI World!\r\n");`

efi.c

Para mayor informacion sobre los protocolos consulte el manual de especificacion de UEFI.

Apendice D

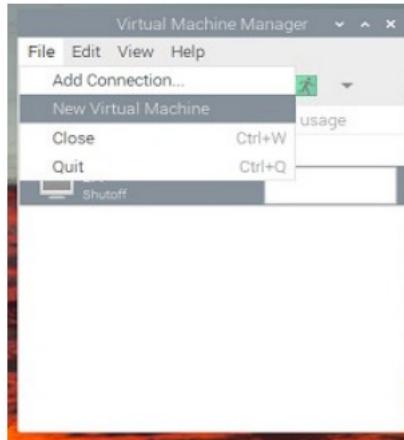
Implementacion de una maquina virtual en Raspberry pi usando Qemu para probar aplicaciones de UEFI-Shell

Acontinuacion se muestra como ejemplo una configuracion de ambiente en una Raspberry Pi:

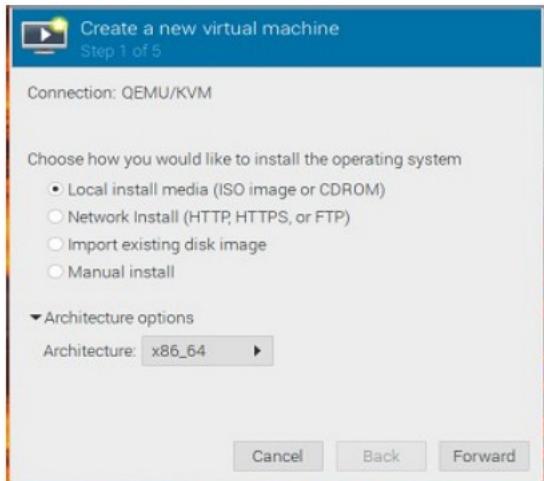
```
File Edit Tabs Help
rodrigo@raspberrypi:~$ sudo apt-get install mingw-w64
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
mingw-w64 is already the newest version (8.0.0-1).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
rodrigo@raspberrypi:~$ sudo apt-get install qemu
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
qemu is already the newest version (1:5.2+dfsg-11+deb11u3).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
rodrigo@raspberrypi:~$ sudo apt-get install vim
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
vim is already the newest version (2:8.2.2434-3+deb11u1).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
rodrigo@raspberrypi:~$
```

figura1.1 las aplicaciones habian sido instaladas previamente

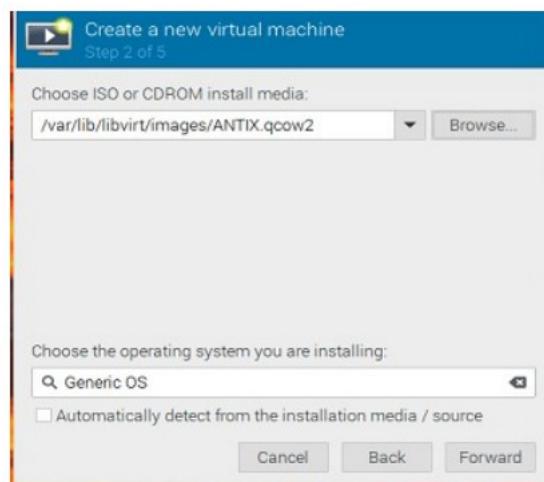
Creamos una nueva maquina virtual (MV) con el programa Qemu:



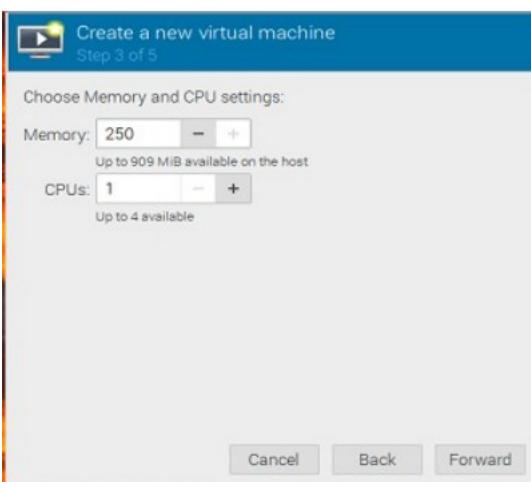
Nos preguntara desde donde deseamos instalar el OS de nuestra Maquina Virtual, seleccionamos local, para que podamos tomarlo desde nuestra Maquina HOST(donde se ejecuta la MV).



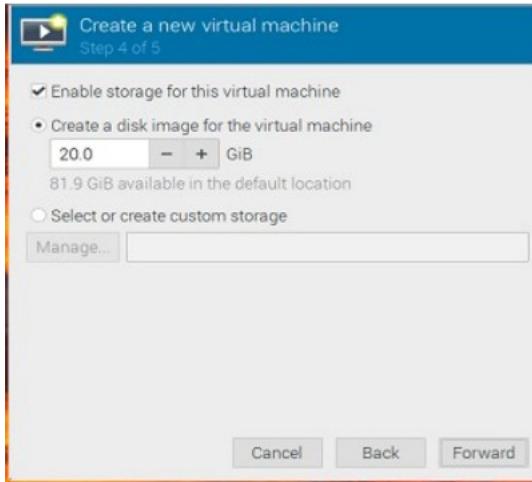
Elegimos la opcion de arquitectura x86_64 para poder trabajar con UEFI.



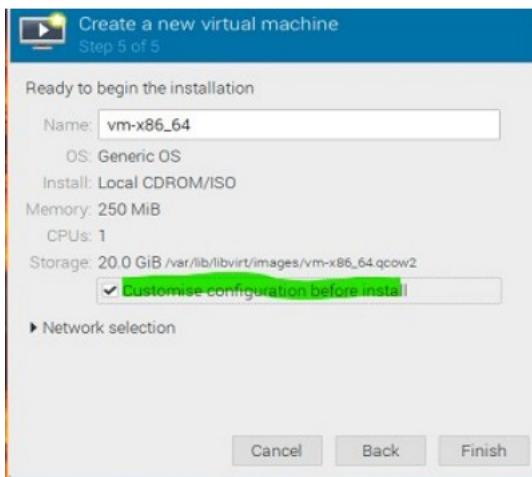
Rellenamos los campos de ubicacion de nuestra imagine ISO del OS y rellenamos los datos del tipo de OS que estamos instalando.



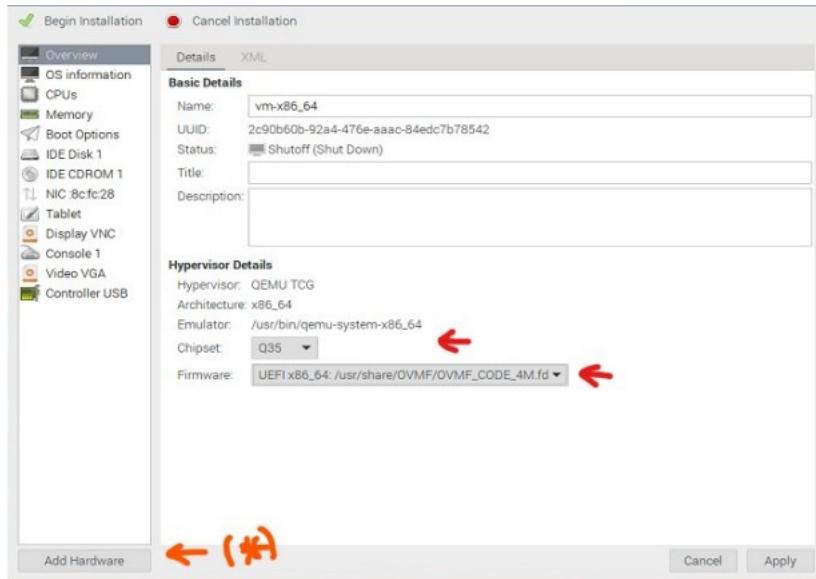
Tambien aseguramos la cantidad de recursos que le proporcionaremos a nuestra MV, con 250 MB y un CPU es suficiente para desarrollo UEFI, si desea usar la Maquina Virtual con el OS ajuste los parametros acorde a lo requerido por su OS, tome en cuenta la cantidad de recursos que ofrece el HOST.



Asigne tambien cierta cantidad de espacio libre del Disco Duro de la maquina Anfritiona (HOST), para crear un disco virtual para nuestra MV.

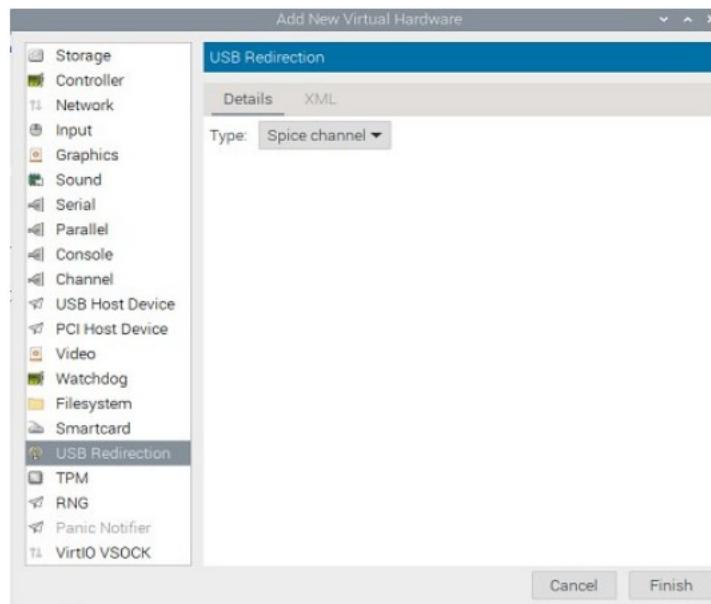


Muy importante seleccione la casilla de ajustes personalizados, para poder realizar las asignaciones de Hardware y seleccionar al momento el BIOS que usara la MV, (esta operacion puede realizarla despues).

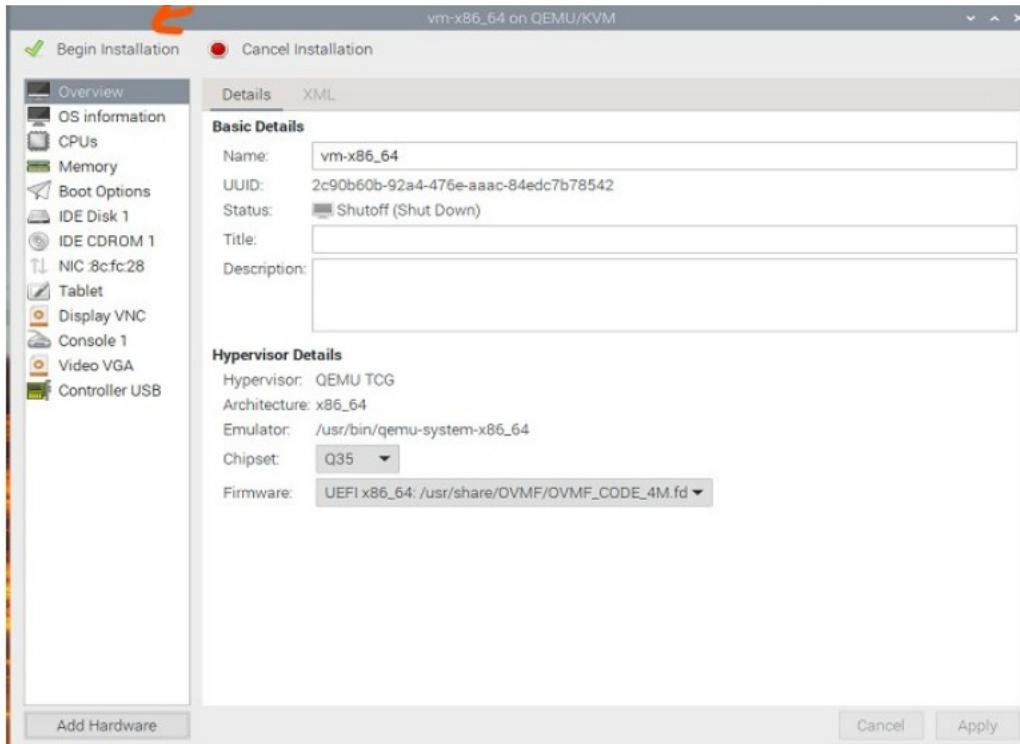


Importante debe seleccionar como Firmware UEFI, para poder tener acceso al menu de EFI-Shell, se recomienda usar Q35 como Chipset.

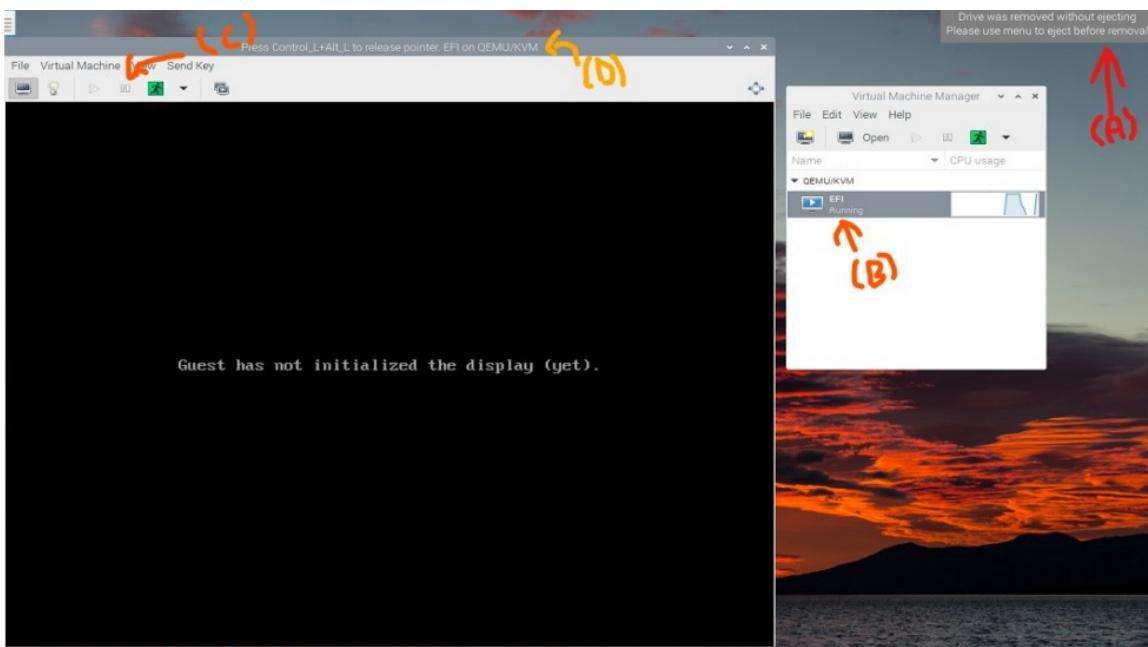
Presione el boton de "Aregar Hardware", para continuar con la configuración (*).Para ubicar la opción de Redirección USB, esto permite que la Maquina Virtual enlace una USB conectada físicamente al HOST para que pueda ser "alcanzable" desde "dentro" de la MV.



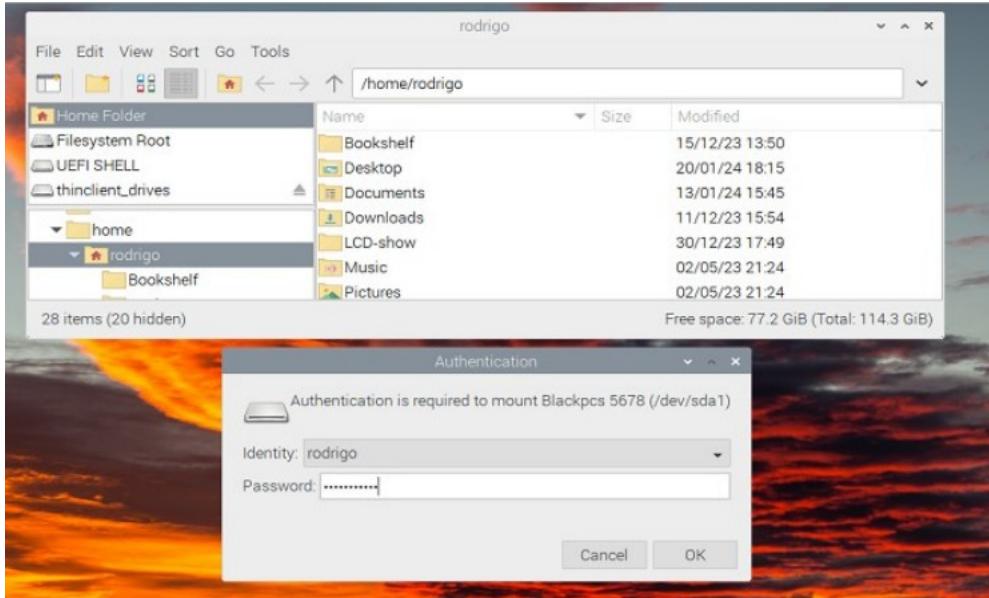
Asegure que la USB física que desea enlazar con la MV sea reconocida por el sistema (conectada físicamente y "montada" en el sistema de archivos) Al terminar la configuracion presione iniciar la instalacion y la MV será creada.



Al comenzar a ejecutarse la maquina virtual el HOST anunciará que la memoria USB ha sido removida (A), Qemu redireccionara la USB hacia la MV, podremos ver el desempeño de la ejecución de nuestra MV y su nombre en (B), En la ventana de ejecución de nuestra MV ponemos atención a la barra de botones (C) donde podremos Detener la ejecución de nuestra MV, especial atención requiere el título de la ventana (D) donde podremos ver el estado de acceso a los controles de la ventana de ejecución de la MV, use la combinación de teclas para efectuar la acción que se muestra.



Tras terminar la ejecución de nuestra MV, Qemu "liberara" la USB y el OS de nuestro HOST, nos preguntara si deseamos reincorporarla a nuestro sistema de archivos para poder acceder a la información que contiene.



Es posible lanzar una MV de Qemu desde la ventana de comandos, consulte las documentacion de su software de virtualización para ejecutarlas de esta manera.

-Bibliografia

°Guillermo Levinne, Computación y programación moderna (una perspectiva integral de la informatica), editorial Addison Wesley.

°Jenny M Peiner, James A Peiner, White Paper Minimal Intel Architecture Boot Loader, Intel Corporation.

°White Paper de Intel sobre el incio de la arquitectura X86

<https://www.intel.com/content/www/us/en/intelligent-systems/intel-boot-loader-development-kit/minimal-intel-architecture-boot-loader-paper.html>

°Proyecto CoreBoot Wep page:

<https://www.coreboot.org/>

<https://doc.coreboot.org/tutorial/part1.html>

°Manual de Docker:

<https://github.com/AngelSanchezT/books-1/blob/master/docker/the-docker-book.pdf>

ºHoja de datos del SuperIO chip PC97338:

<https://pdf.datasheetcatalog.com/datasheet/nationalsemiconductor/PC97338.pdf>

ºInformación de las herramientas de compilación

-Libro para informacion teorica sobre enlazadores y compiladores y su desempeño en un el diseño de software de base:

System Software an introduction to system programming by Leland L. Beck

-Pagina web del manual oficial GNAT el enlazador (linker) del compilador GNU C

https://gcc.gnu.org/onlinedocs/gcc-4.3.6/gnat_ugn_unw.pdf

ºIntel-Basic Instructions for Using the Extensible Firmware Interface (EFI)

ºIntel-Shell Command Reference Manual

Enlaces Web

MinGW

<https://www.mingw-w64.org/>

Debian Download package para EFI_Shell

<https://packages.debian.org/sid/all/efi-shell-x64/download>

Pagina de Tianocore para

<https://github.com/tianocore/tianocore.github.io/wiki/OVMF>

Especificacion UEFI:

https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf