

Índice general

1. Introducción	9
1.1. ¿Qué son las memorias Flash?	9
1.2. Motivación	12
1.3. Objetivos	13
2. Antecedentes	15
2.1. Estado de la tecnología	15
2.1.1. Descripción de equipos	15
2.1.2. Características de la memoria	16
2.2. Utilidades	17
2.2.1. Comandos y herramientas utilizadas	17
2.2.2. ¿Cómo trabajar desde fuera de la ETSiT?	19
3. Desarrollo	23
3.1. Experimentos	23
3.1.1. Tamaño del bloque de memoria	23
3.1.2. Tabla de nombres	27
3.1.3. Movimiento de datos entre sectores	31
3.1.4. Vida útil	33
3.2. Código	39
3.2.1. Archivos marcados	39

3.2.2. Flash	40
3.2.3. Monvimiento de datos	46
3.2.4. Tamaño de bloque	47
3.2.5. Tabla de nombres	50
4. Resultados	51
4.1. Conclusiones	51
4.2. Líneas futuras	52

Índice de figuras

1.1. Diferencia entre Flash NOR y NAND [1].	11
1.2. Evolución de los precios de los discos SSD [2].	12
2.1. Esquema de red.	21
3.1. Fichero con CC para calcular el tamaño de sector.	25
3.2. Tabla de nombre con los ficheros.	26
3.3. Fichero con CC para calcular el tamaño de sector.	26
3.4. Fichero con BB para calcular el tamaño de sector.	27
3.5. Empiece de la tabla de nombres.	28
3.6. Final de la tabla de nombres.	29
3.7. Siguiente tabla de nombres.	29
3.8. Último fichero antes de llenarse la memoria.	30
3.9. Tabla de nombres llena con 65535 ficheros.	31
3.10. Errores de lectura en la memoria.	34
3.11. Inicio de una prueba para determinar la vida de una memoria.	37
3.12. Estado de la prueba para determinar la vida de una memoria tras 26003 vueltas.	38

Listings

3.1. Limpiado y formateo de memoria	24
3.2. Creación de ficheros con datos distintos para el tamaño de bloque	25
3.3. Estado inicial de la memoria	33
3.4. Log del script 3.9	34
3.5. Log del kernel cuando se produce el error en el sistema de ficheros	34
3.6. Log del sistema con memoria dañada	35
3.7. Salida del fdisk	36
3.8. Script que genera datos de varios tamaños.	39
3.9. Script para flashear una memoria con un fichero un gran número de veces.	40
3.10. Script para flashear una memoria con un archivo fácil de encontrar.	42
3.11. Última versión de script para flashear una memoria.	44
3.12. Script para verificar si en el montado y desmontado existe movimiento de datos.	46
3.13. Script para generar ficheros de un tamaño concreto	47
3.14. Script para generar ficheros de un tamaño concreto	48
3.15. Script para crear muchos ficheros iguales	50

Agradecimientos

Una amiga dice que el proyecto es como la culminación de la carrera, el cierre, si lo tomo así, tendría que dar las gracias a mucha gente.

A mi familia, supongo que por todo en general. A los compañeros de la facultad, por los buenos momentos y por los días interminables en el aula. Por todo lo aprendido o que dejamos de aprender. De todo camino se puede sacar buenas experiencias, sobre todo de uno tan largo como este.

Tengo que dar las gracias a mi tutor, Juan Carlos por la paciencia que ha tenido conmigo, sobre todo en estos últimos meses ya que no es fácil coordinarse con alguien que está viviendo fuera. Además se agradece que haya estado siempre dispuesto a ayudar con cualquier dificultad.

A Jose y Natalia también muchas gracias por preguntarme cada vez que me venían por el proyecto y estar dándome con el palo para que acabase.

Pero sobre todo gracias a María por el apoyo, por estar siempre encima animándome a ponerme a escribir, a terminar el proyecto este año, y a no dejarlo. Por corregirme faltas, ayudarme a redactar, en fin, por ser alguien con quien siempre puedo contar.

Por último ya no un agradecimiento que también, si no más bien una mención a mi Madre que estoy seguro que la hubiese gustado verme aquí, ahora.

A todos, muchas gracias!

Capítulo 1

Introducción

Este proyecto de final de carrera tiene como objetivo el estudio del comportamiento de las memorias Flash cuando han tenido un uso excesivo y su capacidad se ve afectada por la pérdida de bloques de memoria. También se analizará la persistencia de los datos borrados. ¿Por qué son tan importantes este tipo de memorias?

Existen multitud de dispositivos de nuestro día a día que usan este tipo de tecnología. Tanto de uso de general, como puede ser ordenadores portátiles, discos de estado sólido (SSD), tablets, sistemas de posicionamiento global (GPS), reproductores de música, cámaras digitales, teléfonos móviles, instrumentos musicales electrónicos, decodificadores de televisión, etc. Como en muchas aplicaciones industriales, como pueden ser sistemas de seguridad, productos de redes y comunicación, productos de administración de ventas (lectores portátiles) o sistemas militares [3].

1.1. ¿Qué son las memorias Flash?

La memoria Flash es una evolución de la memoria EEPROM que permite la lectura y escritura de múltiples posiciones de memoria en la misma operación. Gracias a ello, la

tecnología Flash permite velocidades de funcionamiento muy superiores a la tecnología EEPROM, que sólo permitía actuar sobre una única celda de memoria en cada operación de programación.

Las memorias Flash utilizan una tecnología de almacenamiento que mediante impulsos eléctricos es capaz de leer, escribir o borrar información. Estas memorias están basadas en transistores de puerta flotante que se juntan formando celdas. El elemento básico de funcionamiento de las memorias son los transistores MOS de puerta flotante [4].

Fujio Masuoka en 1984 inventó este tipo de memoria como evolución de las EEPROM existentes por aquel entonces, mientras trabajaba en Toshiba. Intel intentó atribuirse la creación sin éxito, aunque sí comercializó la primera memoria Flash de uso común en 1988 [5].

Se dividen en dos clases según el tipo de puertas usado en su fabricación:

- NOR: este tipo de memoria se ha usado típicamente para almacenar el software que luego es ejecutado en los dispositivos portátiles, ya que proporciona capacidades de acceso aleatorio de alta velocidad, pudiendo leer y escribir datos en lugares específicos de la memoria sin tener que acceder a la memoria en modo secuencial. A diferencia de la memoria Flash NAND, la memoria Flash NOR permite la recuperación de datos desde un solo byte. La memoria Flash NOR es excelente en aplicaciones donde los datos se recuperan o se escriben de manera aleatoria. Se encuentra integrada en teléfonos móviles (para almacenar el sistema operativo), y también se utiliza en los ordenadores para almacenar el programa BIOS que se ejecuta en el arranque.
- NAND: fue inventada después de la NOR. Están diseñadas con unas celdas muy pequeñas, que permiten tener un precio muy pequeño por bit de almacenamiento. Por eso se encuentra comúnmente en unidades de disco duro de estado sólido y unidades de almacenamiento. La memoria Flash NAND lee y escribe a alta velocidad, en modo secuencial, manejando datos en bloques de tamaño pequeño. La memoria

Flash NAND puede recuperar o escribir datos como páginas únicas, pero no puede recuperar bytes individuales como la memoria Flash NOR. La memoria Flash NAND es más económica que la memoria Flash NOR y puede guardar una mayor cantidad de datos en el mismo tamaño de bloque, ver figura 1.1 [3].

En lo que al consumo energético se refiere, algo bastante importante dependiendo del dispositivo, es algo menor en las NAND cuando son usadas en aplicaciones que tienen mucha carga de escritura. Sin embargo el consumo instantáneo sin demasiada carga de trabajo es bastante parecido entre los dos tipos de memorias [1].

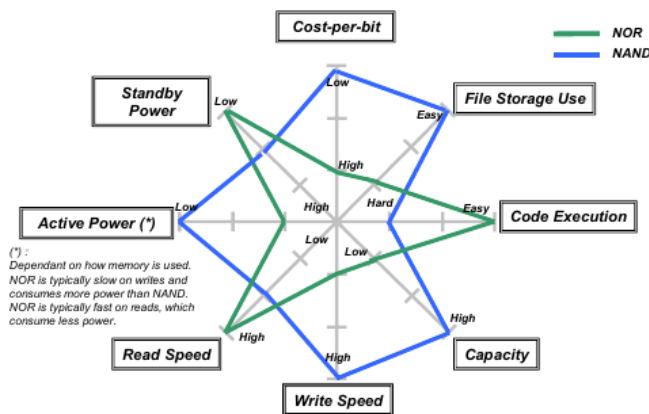


Figura 1.1: Diferencia entre Flash NOR y NAND [1].

En la actualidad la diferencia entre los dos tipos es cada vez menor [1] [5].

La memoria Flash con la que hemos trabajado ha sido es una *BIWIN BW29F64G08CBABA*, que son de tipo *Micron 20nm MLC Flash*. El controlador que utiliza es un *Alcor AU6989ANHL*.

Por último hay que mencionar que este tipo de tecnología va a ser muy importante en el futuro, como por ejemplo en aplicaciones de automoción donde se han conseguido microcontroladores con una memoria Flash de 40 nm de muy bajo consumo [6]. Hay también proyectos para usar las memorias Flash como RAM, pero de momento son más lentas que las DRAM, aunque puede llegar a consumir diez veces menos de energía y son algo más baratas [7].

1.2. Motivación

Como hemos visto anteriormente, el uso de las memorias Flash está bastante extendido. La sociedad demanda dispositivos más pequeños y rápidos y las memorias Flash se adaptan a estas necesidades. Ha habido una creciente implantación en portátiles conforme el precio por Giga byte ha ido disminuyendo, ver figura 1.2 [2].

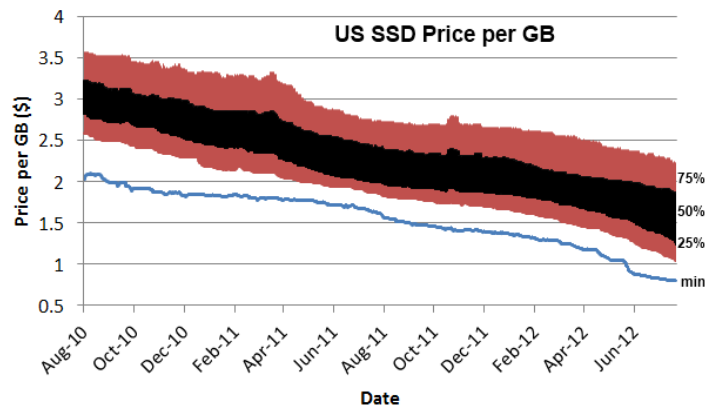


Figura 1.2: Evolución de los precios de los discos SSD [2].

Además esta tecnología supone un ahorro energético, algo también muy importante en dispositivos móviles o portables. En un artículo de la web www.anandtech.com realizan una comparación con un MacBook Air usando un disco sólido y un disco magnético. En sus resultados puede verse una diferencia en el consumo energético de 0.9W en el HDD en lectura y escritura frente a 0.24/0.32W del SSD, o de un 16.8 % más de batería usando el portátil para navegar por internet y escuchar música. El único punto donde la memoria Flash pierde es en el consumo hibernando, en que el disco duro no consume energía y el disco sólido 0.015W [8].

De momento solo hemos contado las virtudes de esta tecnología, pero las memorias Flash tienen un problema: cada bloque de memoria soporta unos ciclos de escritura/borrado. Nadie quiere estar haciendo fotos con su cámara digital y que un buen día haya fotos que se pierdan, o comprarse un disco sólido de una capacidad y que después de un tiempo esa capacidad se haya visto reducida sensiblemente. Todo esto nos hace plantearnos las

siguientes preguntas: ¿qué pasa cuando se usa más veces de las soportadas?, ¿cuántos ciclos son los realmente soportados?, ¿qué información queda si un bloque se rompe?, y, ¿se puede recuperar la información que había?

Normalmente en la mayoría de los dispositivos con memorias Flash, el chip de memoria va acompañado de un controlador. Éste procesa los datos en la lectura y la escritura. Su presencia supondrá un problema añadido porque existirá una diferencia entre los datos que se ven desde fuera al utilizar el dispositivo, que han sido procesados por el controlador, y los datos presentes que están realmente en el chip de memoria [9].

En este proyecto hemos tratado de profundizar sobre funcionamiento de las memorias flash, sobre todo en lo que se refiere al funcionamiento interno, a bajo nivel, pero sin entrar en el hardware. En qué posiciones de memoria guarda la tabla de nombres, cómo gestiona los archivos, cuál es el tamaño que tiene un sector, cuántos ciclos de vida útil tiene un sector y qué información queda cuando un sector se deteriora.

Para ello hemos realizado varios experimentos, los cuales detallamos en el Capítulo 3.

1.3. Objetivos

- **Determinar el tamaño de sector en hardware y software:** hay que diferenciar entre lo que el sistema ve y lo que realmente está guardado en memoria. El sistema de ficheros puede tener un tamaño de bloque distinto del que físicamente tiene la memoria.
- **Determinar como funciona la tabla de nombres:** en la tabla de nombres están los nombres de los ficheros que tiene guardados la memoria, ¿se guarda también un puntero al fichero? ¿esta tabla tiene algún formato especial?, son algunas de las preguntas que intentaremos resolver en este apartado.
- **¿El controlador mueve datos entre bloques para alargar la vida de los mismos?:** cada bloque tiene unos ciclos de vida, si solo unos pocos están libres y

siempre se usan los mismos, ¿puede el controlador mover datos a bloques que estén menos “gastados”?

- **¿Un borrado implica cambios “físicos”?**: Cuando se borra un fichero desde el sistema operativo, sus datos siguen existiendo hasta que se sobrescribe, ¿se borra también su referencia en la tabla de nombres?
 - **¿El montar y desmontar un dispositivo implica movimiento de datos?**: Puede pasar que el supuesto movimiento de datos al que nos referíamos antes pase en el montado y desmontado del dispositivo.
- **Determinar la vida útil (ciclos de vida) de un sector**: cada bloque de memoria tiene unos ciclos de borrado/escritura. Vamos a determinar cuántos ciclos aguanta un bloque.
 - **Ver que información queda después en un sector roto**: una vez que un bloque de memoria deja de estar reconocible por el sistema, ¿queda algún tipo de información útil en él?

Capítulo 2

Antecedentes

2.1. Estado de la tecnología

2.1.1. Descripción de equipos

Para poder realizar los experimentos hemos contado con dos equipos en el laboratorio. Son dos equipos con procesador AMD Phenom (tm) X6 1055T y 7'6 GiB de RAM. En ellos hemos instalado Debian 6.0.6, que viene con el kernel 2.6.32-5-amd64.

Ambos dos forman parte de una red interna de gf.tel.uva.es, teniendo cada uno de ellos los siguientes hostnames:

- Leonardo, con la IP: 192.168.0.50.
- Donatello, con la IP: 192.168.0.51.

Como esta red no está accesible desde el exterior de la escuela, para poder trabajar desde casa y tener monitorizado el trabajo, hemos recurrido a los túneles ssh y al comando screen de Linux, que detallamos en la sección [2.2.2](#).

2.1.2. Características de la memoria

La diferencia entre lo que se ve desde el software y lo que físicamente está en la memoria, es algo muy importante en este Proyecto. Como ya hemos comentado, todas las memorias disponen de un controlador, y este puede recolocar los datos de manera interna, y completamente transparente desde el exterior. El controlador que utiliza es un *Alcor AU6989ANHLL*. Sobre el controlador no hemos encontrado mucha información ya que esta normalmente suele ser privativa de la empresa que lo diseña.

La memoria Flash con la que hemos trabajado ha sido es una *BIWIN BW29F64G08CBABA*, que son de tipo *Micron 20nm MLC Flash*. Las memorias MLC (Multi-level cell) tienen la capacidad de guardar más de un bit de información, solo se da en las NAND Flash y permite guardar más información usando el mismo número de transistores [10]. Salvo en dispositivos de red, las memorias MLC son de las más extendidas [11].

Este tipo de memorias están formateadas con FAT32. FAT es un sistema de archivos desarrollado para MS-DOS. FAT32 es su última evolución, utiliza direcciones de cluster de 32 KiB (aunque sólo 28 de esos KiB se utilizan realmente) y el tamaño máximo de un archivo en FAT32 es 4 Gigabytes.

Tamaño de cluster que impone FAT32 es de 4K en memorias de hasta 8GB, [12], y esto puede llegar a ser una limitación a la hora de obtener datos fiables de nuestros experimentos, pues no sabemos cómo puede o no condicionarnos.

2.2. Utilidades

2.2.1. Comandos y herramientas utilizadas

Comandos Linux

En Linux existen multitud de comandos que nos han facilitado el trabajo a la hora de realizar este Proyecto. Uno de los más importantes ha sido `dd`.

- `dd`: *Copia un fichero, convirtiendo y formateando según los operandos*. En principio este comando se usa para crear copias de un disco. En nuestro caso creamos archivos `.iso` de las memorias USB. Pero en unas pruebas nos dimos cuenta de que tenía mayor rendimiento que el comando `cp`, que es el originario de Linux para copiar ficheros, y desde ese momento lo hemos usado también para la copia “normal”. Por defecto sus bloques de copiado son de 1024 bytes y sus parámetros son los siguientes:

```
$ dd if=ruta_o_fichero_entrada of=fichero_salida
```

- `df`: *Uso del espacio del sistema de ficheros*. Este comando te da información sobre el espacio de los dispositivos montados en el sistema. Aparte de ver el espacio disponible en las memorias, principalmente se ha utilizado para ver la capacidad de las memorias y comprobar si su capacidad se ha visto reducida, ver [3.1.4](#). Nota importante:

```
$ df --block-size=kB \\ Muestra un tamaño de bloque de 1000bytes
```

```
$ df --block-size=k \\ Muestra un tamaño de bloque de 1024bytes
```

- `hexdump`: *Muestra el contenido de un fichero en ascii, decimal, hexadecimal, u octal*. Gracias a la orden:

```
$ hexdump -c .iso > .txt
```

Podemos crear un archivo de texto que luego compararemos con `meld`, que es una herramienta gráfica de Linux para comprar ficheros de texto, de esta manera podemos comparar imágenes `iso` de forma muy fácil y ver sus diferencias.

- `iotop`: *Muestra el proceso de las peticiones de lectura o escritura que implican a distintos dispositivos del sistema.* Gracias a este comando podemos monitorizar si una lectura o escrita a la memoria sigue en curso o murió su proceso.
- `ls -l | wc -l`: Esta orden muestra la cantidad de elementos en un directorio, ha sido de mucha utilidad sobre todo en el experimento [3.1.2](#).
- `od -x .txt`: *Muestra un fichero en octal y otros formatos.* Hemos utilizado ficheros en hexadecimal, como ficheros base de lectura y escritura en este proyecto, este comando ha ayudado a visualizar esos ficheros hexadecimales.

[\[13\]](#) [\[14\]](#)

Bash y Shell script

Bash es un interprete de órdenes, está basado en la shell de Unix y es compatible con POSIX. Su nombre viene del acrónimo “*Bourne-Again SHell*”, un juego de palabras de Stephen Bourne, el autor del antecesor de la Shell actual de Unix [\[15\]](#). Bash dispone de una gran cantidad de comandos que te permiten hacer casi cualquier cosa [\[16\]](#).

Una **Shell** es un marco que ejecuta comandos. La Shell puede aceptar de entrada órdenes por teclado o leer instrucciones de un fichero [\[15\]](#).

Un script de Shell es una secuencia de comandos que se ejecutan uno detrás de otro, ya que es un tipo de lenguaje interpretado. Un fichero de texto con extensión `.sh` al que se le da permisos de ejecución “`chmod u+x script.sh`” para que pueda ser ejecutado en Linux.

Gracias a estos Shell script hemos podido automatizar procesos que podrían llevar muchos días y filtrar los resultados para una mejor comprensión de los mismos.

Bash tiene algunas peculiaridades a la hora de tratar con variables que puede llamar la atención a alguien acostumbrado a otros lenguajes de programación:

- La asignación de variables se realiza sin “declaración”: `variable=valor`.
- Para modificar esas variables se usan dobles paréntesis: `((variable=variable+10))`.
- Para usar las variables deben estar precedidas de `$`: `echo ‘‘Hola’’ $variable`.

Control de versiones y Git

Git es una herramienta para el control de versiones. Tiene muchas virtudes, pero la más importante en este contexto es poder tener un histórico de cada cambio “comiteado”, lo que nos permite tener una copia de seguridad casi perfecta tanto del código como de la redacción de Proyecto. En cualquier momento podemos volver a un punto anterior o ver qué se cambió de un fichero en concreto [17]. Todo el Proyecto se ha ido guardando en [github.com](https://github.com/jilgue/flashcrash), que permite crear repositorios git de manera gratuita siempre que estos sean de libre acceso, el repositorio del proyecto se encuentra en: <https://github.com/jilgue/flashcrash>.

2.2.2. ¿Cómo trabajar desde fuera de la ETSiT?

Se puede tardar días en conseguir resultados. Para monitorizar el proceso es necesario conectarse a los equipos y poder recuperar la shell donde tenemos lanzado el script. Para ellos nos hemos ayudado del protocolo SSH y el comando screen.

SSH

SSH es un protocolo de shell remota segura. Gracias a ello podemos conectarnos al terminal de un ordenador y ejecutar ordenes en él.

SSH usa una autenticación con cable público/privada. Si queremos conectarnos por SSH sin tener que escribir la contraseña cada vez, tenemos que generar una pareja de claves y añadir la pública a nuestro servidor:

```
$ ssh-keygen -t rsa -b 2048
```

Esto nos genera un `.pub` dentro de la carpeta `.ssh` y tenemos que copiar su contenido en el archivo `.ssh/authorized_keys` del servidor. Con esto la próxima vez que nos conectemos no tendremos que escribir la contraseña [18].

Tunelando con ssh

En la escuela tenemos una arquitectura parecida a la de la figura 2.1:

Desde “Mi equipo” no puedo conectarme al “Otro equipo”. Para solventar este problema creamos un túnel:

```
$ ssh -L 2222:donatello:22 -L 2221:leonardo:22 usuario@gf.tel.uva.es
```

Con esta línea hemos abierto un túnel desde nuestro puerto 2222 y 2221 al 22 de donatello y leonardo respectivamente a través de `gf.tel.uva.es`. Ahora para conectarnos a nuestro túnel escribimos en nuestro terminal:

```
$ ssh usuario@localhost -p2221 -X
```

```
$ ssh usuario@localhost -p2222 -X
```

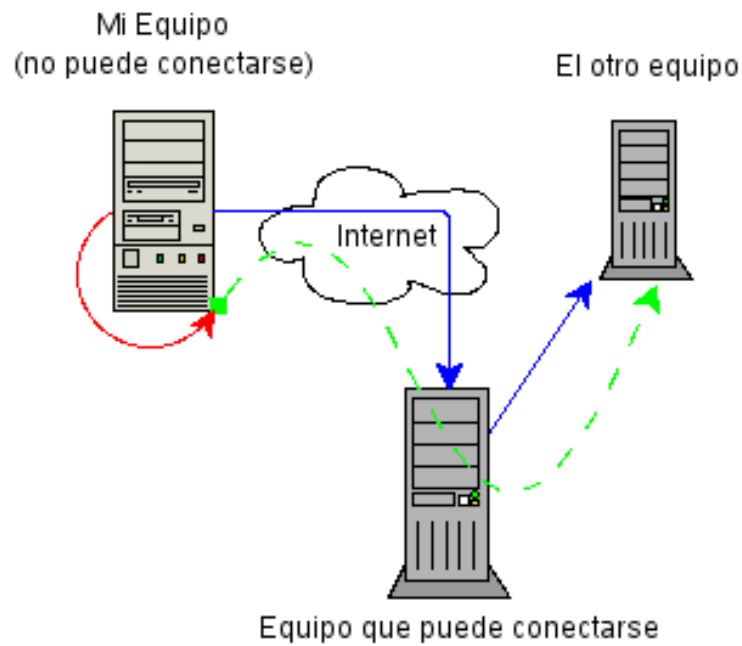


Figura 2.1: *Esquema de red.*

Screen

Screen es una herramienta que nos permite recuperar una sesión shell. Podemos dejar corriendo un script cerrar la conexión, irnos a casa, abrir una conexión nueva y recuperar la misma terminal que teníamos antes.

Uso básico de `screen`:

```
$ screen // abre una sesión
```

```
Ctrl + a + d // deja la sesión en background.
```

```
$ screen -r // recupera la sesión
```

[19] [20]

Capítulo 3

Desarrollo

3.1. Experimentos

3.1.1. Tamaño del bloque de memoria

Como hemos visto anteriormente, la memoria se divide en bloques o sectores. Por las especificaciones del sistema de ficheros, sección 2.1.2, el tamaño de bloque que manejamos desde el sistema operativo es de 4K. Al no haber encontrado las especificaciones físicas de la memoria, hemos realizado una serie de experimento de ingeniería inversa para tratar de averiguar cual es el tamaño del sector físico de memoria.

Hicimos varios experimentos, como llenar todo el disco menos 8K, escribir un fichero de 8K, borrarlo, escribir uno de 6K y buscar si quedaban restos del de 8. Nos dimos cuenta de que los 2K restantes están intactos, como veremos más adelante, los datos y las referencias a esos datos se guardan en zonas distintas, sección 3.1.2, y al repetir el mismo experimento, los datos habían sido recolocados. Por lo que este método no iba a darnos resultados válidos y por lo tanto no pudimos obtener las conclusiones que queríamos.

Como la idea inicial no funcionó, probamos a mirarlo al revés, copiamos un archivo de

un byte y miramos si había cambiado algo más. Para ello primero llenamos el disco con un fichero de *00* en hexadecimal, y después lo formateamos, código 3.1, de esta manera la memoria se queda “limpia” para poder tener resultados más fácilmente interpretables. Después de hacer una copia de su imagen actual, copiamos un archivo de un byte con *CC* en hexadecimal. Pero al abrir la imagen con **bless** vimos que no había nada cambiado alrededor de los datos del fichero como podríamos haber podido esperar.

Listing 3.1: Limpiado y formateo de memoria

```
$ ./tam_bloque_b2.sh
$ dd if=datos/datosfin.txt of=/media/cesar/usb2/datos.txt
# umount usb2
# mkfs.vfat /dev/sdc1
# mount -o gid=1000,uid=1000 /dev/sdc1 usb2
```

Por último probamos a grabar dos archivos de 2 bytes y mirar “cuánto” se separan. El tamaño de bloque será el espacio que queda entre ambos ficheros en memoria.

```
$ ./archivos_marcados_b3.sh
$ ./tam_bloque_b2.sh
$ dd if=datos/datosfin.txt of=/media/cesar/4F35-D764/datosfin
$ cp datos2/datos1.txt /media/cesar/4F35-D764/datos1.txt
$ cp datos2/datos1.txt /media/cesar/4F35-D764/datos2.txt
$ sudo dd if=/dev/sdb1 of=tambloque.iso
```

Al copiar dos veces el mismo archivo tuvimos el problema de que no encontrábamos los archivos, así que lo repetimos pero el segundo archivo que copiamos contenía *BB*. Pero nos encontramos con otro problema, **bless** solo encontraba el fichero con los datos *CC*, figura 3.1, pero no el que contenía *BB*. Al ser unos ficheros con tan pocos datos se pueden perder entre el resto de información que contiene el dispositivo.

Así que repetimos el experimento, después de haber aplicado el proceso de limpieza,

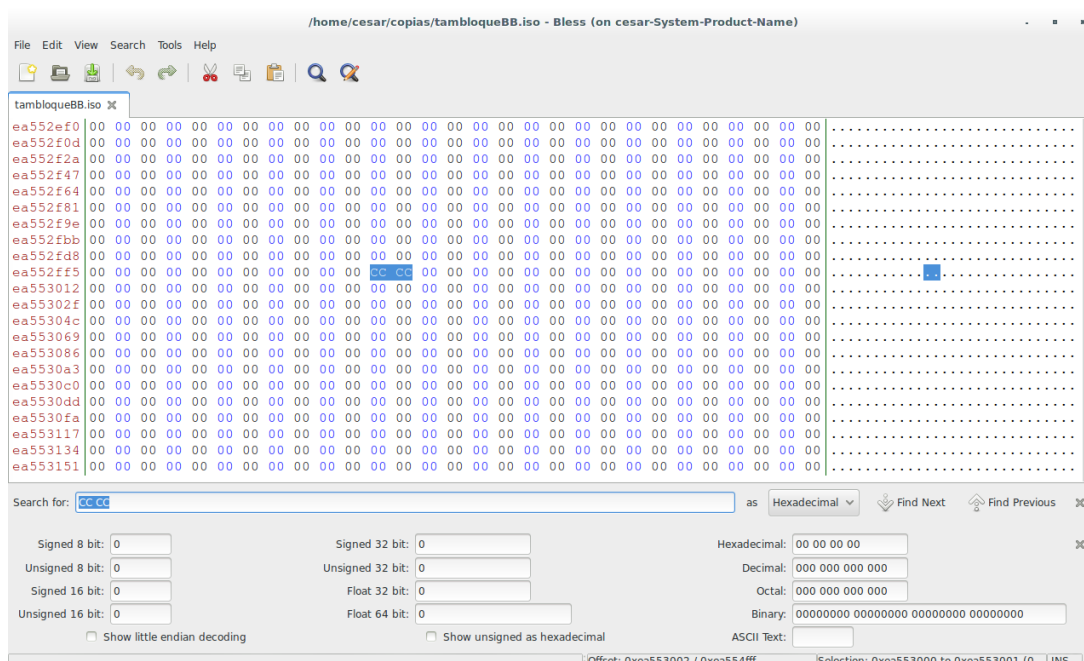


Figura 3.1: *Fichero con CC para calcular el tamaño de sector.*

código 3.1, copiando dos ficheros con distintos datos, código 3.2. Y este fue el resultado de que obtuvimos:

Lo primero que encontramos es la tabla de nombre con la referencia a los ficheros, figura 3.2, veremos más sobre cómo funciona esta tabla en la sección 3.1.2. Como puede verse en las figuras 3.3 y 3.4 los datos empiezan en la posición 7692288 y 7696384 respectivamente, hemos puesto el offset de **bless** para que muestre la posición de bit, por lo que su separación es de 4096 bytes, que es lo mismo a 4Kbytes. Por lo que el controlador no recoloca los datos en este sentido, respeta el tamaño de bloque que tiene el sistema de ficheros FAT32 que como ya hemos visto es de 4K.

Listing 3.2: Creación de ficheros con datos distintos para el tamaño de bloque

```
$ echo -e -n '\xBB' > datosBB.txt
$ cat datosBB.txt >> datosBBBB.txt
$ cat datosBB.txt >> datosBBBB.txt
$ echo -e -n '\xCC' > datosCC.txt
$ cat datosCC.txt >> datosCCCC.txt
```

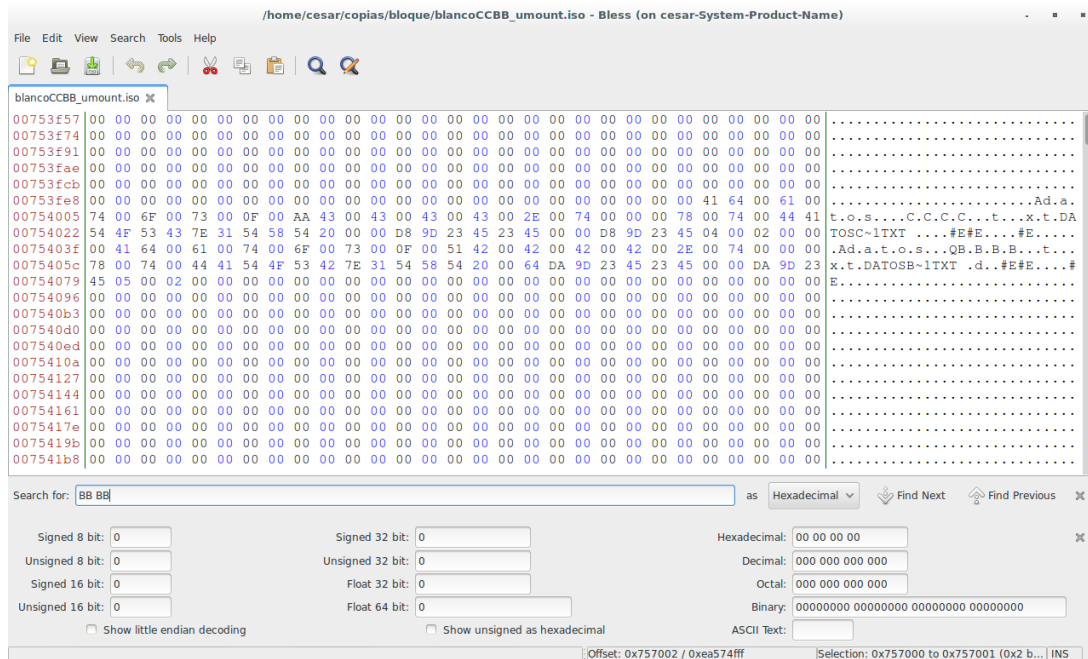


Figura 3.2: Tabla de nombre con los ficheros.

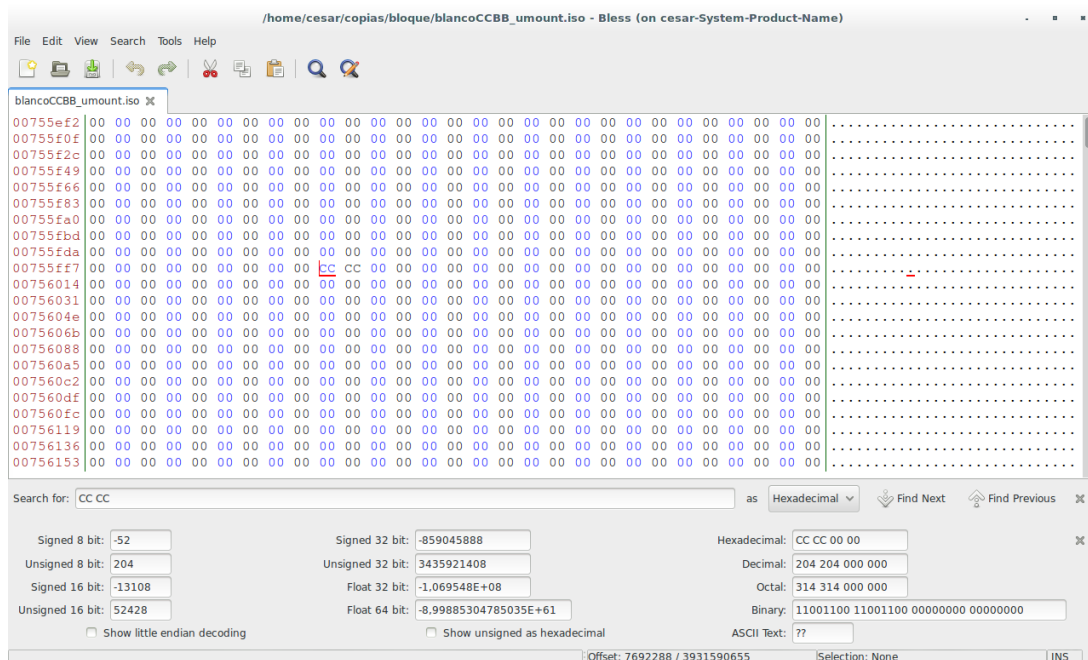


Figura 3.3: Fichero con CC para calcular el tamaño de sector.

```
$ cat datosCC.txt >> datosCCCC.txt
```

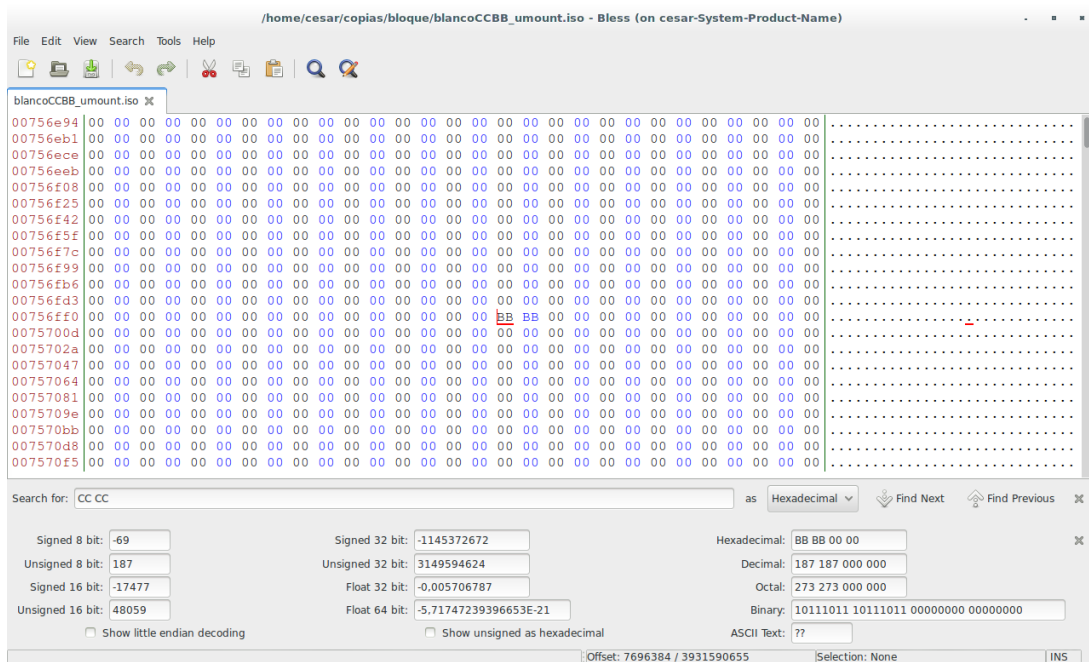


Figura 3.4: *Fichero con BB para calcular el tamaño de sector.*

3.1.2. Tabla de nombres

Como hemos visto en la sección anterior, 3.1.1, hay una parte de la memoria reservada para guardar puramente datos y otra donde está el registro de los ficheros. Esta última es la tabla de nombres.

Lo que hemos hecho para descubrir un poco su funcionamiento es, con el script 3.15, llenar toda la memoria, ya que en teoría si independientemente del tamaño del fichero, en memoria ocupa 4K, si yo copio tamañoDeLaMemoriaExpresadoEnK / 4 ficheros, la memoria se llena.

Después de ejecutar el script el resultado fue tras crear 32.767 el dispositivo dijo que estaba lleno. De este primer experimento pudimos sacar varias conclusiones:

- La tabla de nombres empieza en el byte 7831552, figura 3.5, por lo que antes de esa posición solo hay información de control y propia del sistema de ficheros.

- La tabla termina en el 7835648, lo que ocupa 4096 bytes, figura 3.6.
- El tamaño reservado para datos por “página” es de 256K, ya que en la posición 8097792 se terminan los datos y empieza una nueva tabla de nombres, figura 3.7.
- El `df -h` muestra que se están usando 130M, por lo que no se ha llenado el dispositivo, si no la tabla de nombres.
- La última página de la memoria que tiene contenidos empieza en la posición 1823698944, si cada 256K hay una página nueva, en teoría debería haber espacio para muchas más, esto nos hace pensar que igual es otro el motivo por lo que se ha llenado tan rápido en este experimento.

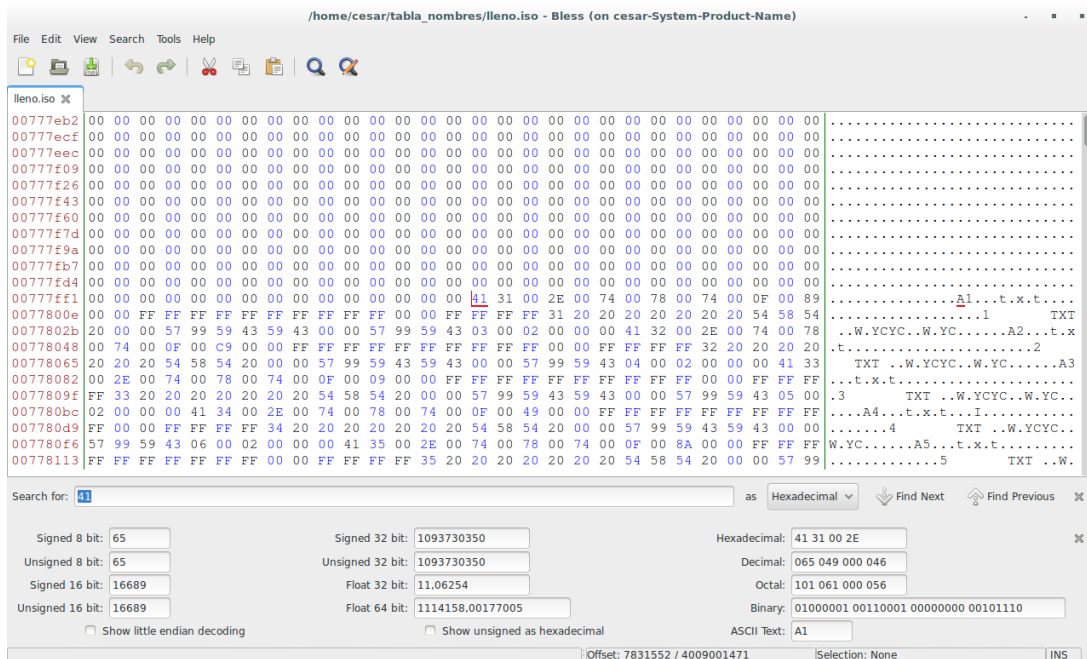


Figura 3.5: *Empiece de la tabla de nombres.*

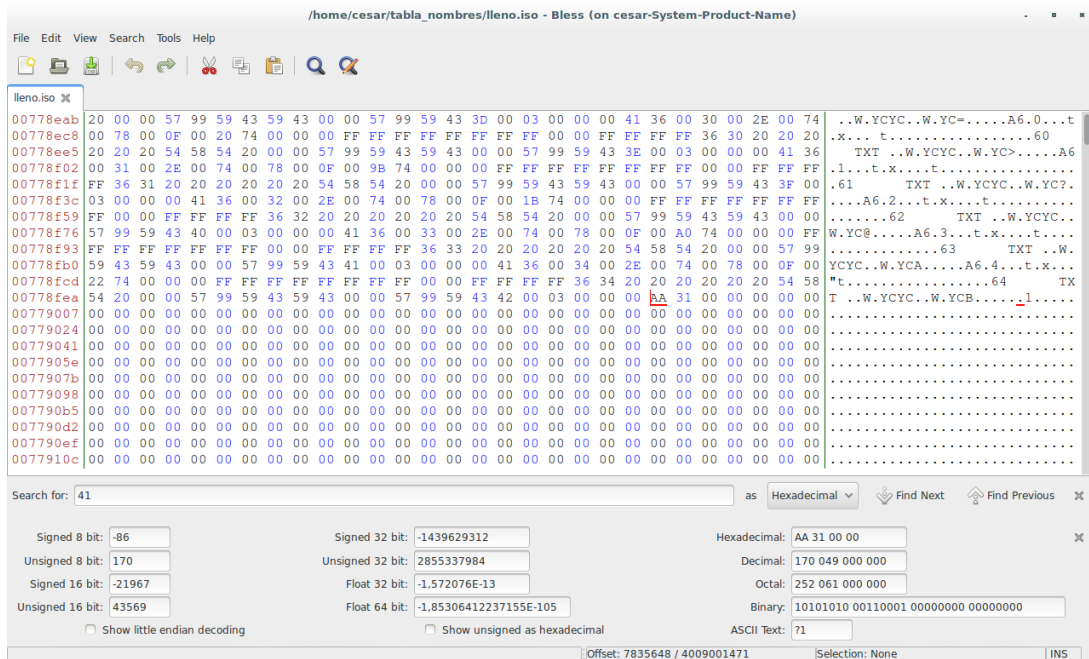


Figura 3.6: Final de la tabla de nombres.

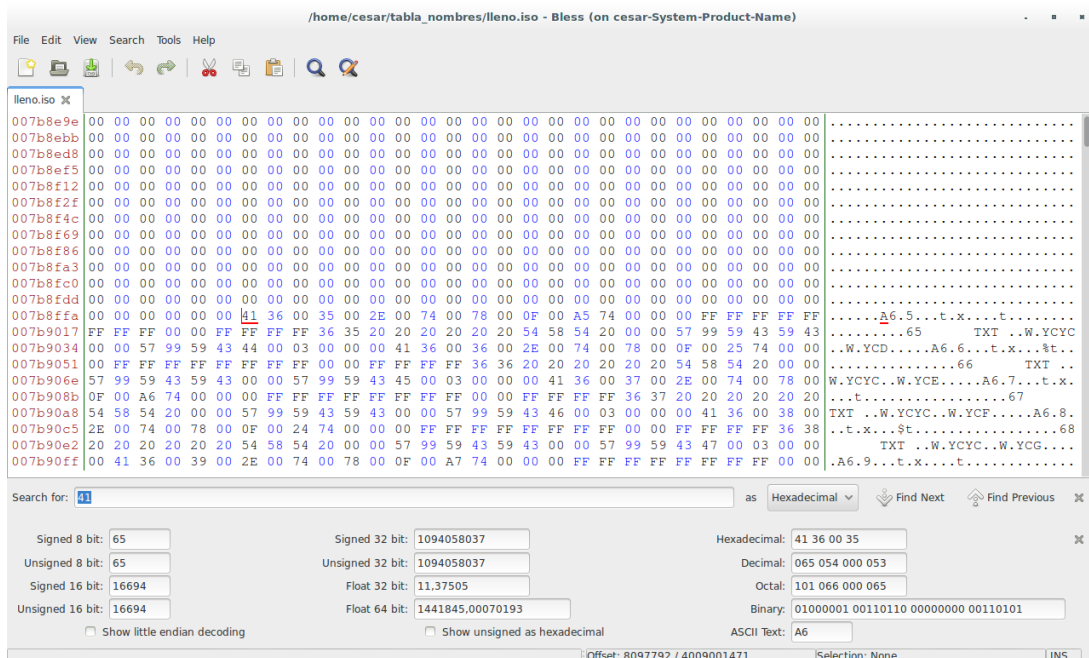


Figura 3.7: Siguiente tabla de nombres.

En las siguientes pruebas mejoramos el script 3.15, quitándole la extensión “.txt” de los ficheros de esta manera conseguimos que la memoria se llenase con 65535 ficheros:

```
$ ls | wc -l
```

```
65535
```

Ver figura 3.8.

Para ello seguimos la teoría de que al ser tantos ficheros, el nombre de los ficheros iba aumentando tanto que aunque pudiesen caber más ficheros por tener páginas de memoria libres, la tabla de nombres se llenaba antes por culpa de la longitud de los nombres.

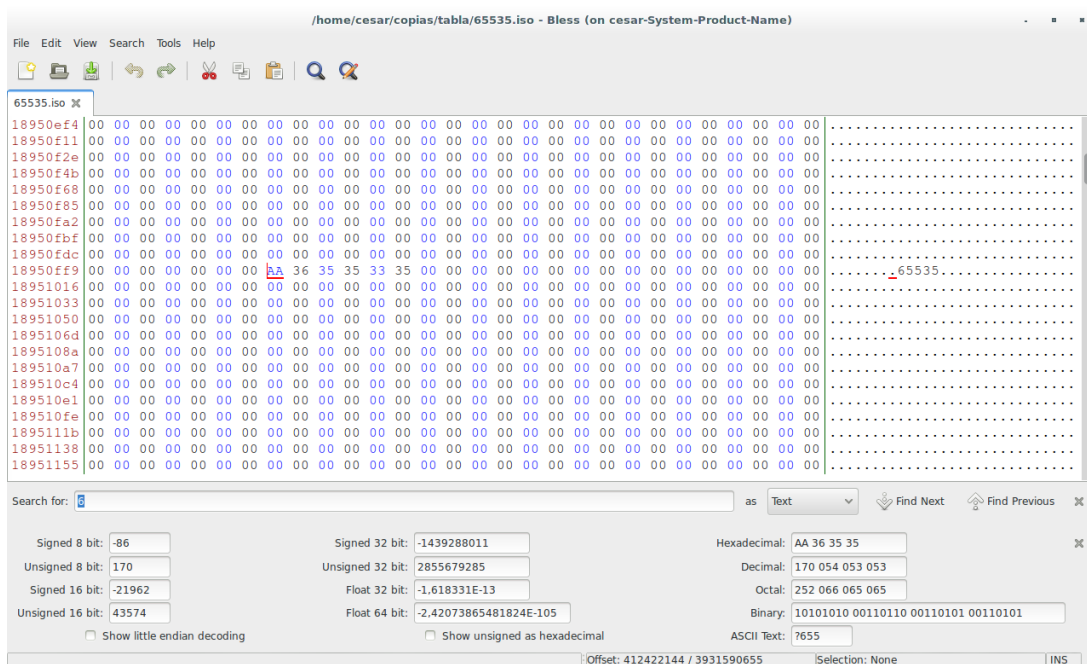


Figura 3.8: Último fichero antes de llenarse la memoria.

Como se puede ver en la figura 3.9, la tabla de nombres está bastante saturada de información. Y aunque no estamos del todo seguros, creemos que el tamaño máximo de ficheros que entran depende de la longitud del nombre, ya que en este segundo experimento hemos modificado el script para que genere ficheros con nombres lo más corto posible, y han entrado casi el doble de ficheros antes de llenarse la memoria.

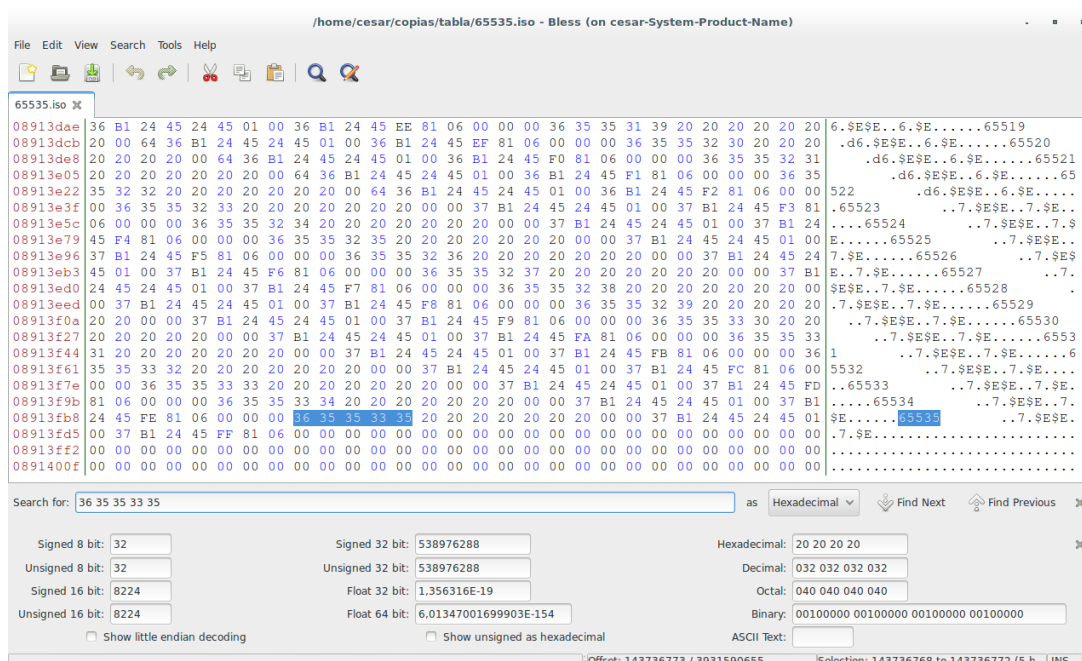


Figura 3.9: Tabla de nombres llena con 65535 ficheros.

3.1.3. Movimiento de datos entre sectores

En esta sección partimos de la suposición inicial de que el controlador puede reordenar los datos a distintos sectores para salvaguardar la vida del conjunto, haciendo que todos se usen de manera equitativa. Esto se conoce como “wear levelling” y suelen usar algoritmos propietarios de empresas, secreto industrial. Para averiguar si esto es cierto vamos a realizar varios experimentos:

En todos ellos es importante que la memoria contenga varios ficheros cuyos datos que comprobaremos si sufren algún tipo de movimiento.

- Con ayuda del script 3.12, determinar si el montado y desmontado del dispositivo implica algún tipo de movimiento de datos.
- Copiar un fichero, borrarlo, y comparar las imágenes de antes y después.
- Copiar varios ficheros, crear la imagen, borrarlos, volver a copiarlos y comprar imágenes.

- Copiar un fichero varias veces y comparar si los datos de los ficheros iniciales sufren cambios de posición.

Vamos a llenar el dispositivo con un fichero con *00*. Después borramos el archivo y copiamos un archivo de 64K que contiene *CC*. Hacemos una imagen *.ISO* de la memoria, borramos el fichero, creamos otra imagen, volvemos a copiar el mismo archivo y hacemos otra imagen del dispositivo.

```
$ ./archivos_marcados_b3.sh
$ ./tam_bloque_b2.sh
$ dd if=datos/datosfin.txt of=/media/cesar/4F35-D764/datosfin.txt
$ rm /media/cesar/4F35-D764/datosfin.txt
$ cp datos2/datos16.txt /media/cesar/4F35-D764/
$ sudo dd if=/dev/sdb1 of=datos16.iso
$ rm /media/cesar/4F35-D764/datos16.txt
$ sudo dd if=/dev/sdb1 of=datos16borrado.iso
$ cp datos2/datos16.txt /media/cesar/4F35-D764/
$ sudo dd if=/dev/sdb1 of=datos16copiado.iso
```

Con *bles* hemos visto que los datos siguen estado después de haber borrado el fichero. Pero *md5sum* nos dice que las imágenes son distintas, así que vamos a crear ficheros de texto con *hexdum* para poder comparar las imágenes con *meld*.

```
$ md5sum *
5b47c568523dbb3bb3128e3f4193148d  datos16.iso
5704c3cc3b5264aecfc5e5a4a54f0bf5  datos16borrado.iso
9fbc3200e1f61767c5c0da7353fe1c72  datos16copiado.iso
```

En este punto del experimento nos encontramos con una limitación que no fuimos capaces de superar, el tamaño de las imágenes era demasiado grande para poder obtener

resultados válidos. Comparar las imágenes con `bles` no era viable, `meld` no conseguía calcular las diferencias de los ficheros de texto con el *dump* de la imagen. Dividimos esos ficheros de texto con `split` pero `meld` no calculaba bien las diferencias con archivos partidos. El comando `cmp -l` conseguía mostrar datos pero no eran legibles ni interpretables. Y el comando `diff -y` si era capaz de calcular todas las diferencias de los ficheros con el *dump* pero no lo hacía bien ya que mostraba cada linea como si fuese una diferencia.

La comparación binaria de grandes cantidades de datos es complicada por las herramientas que proporciona Shell. Desarrollamos un pequeño programa en C que comparaba un binario bit a bit, pero era muy lento y no mostraba información fácilmente entendible. Est punto sería algo interesante a continuar como líneas futuras.

3.1.4. Vida útil

Después de varias pruebas con el script `flash_b2.sh` 3.9 y `flash_b3.sh` 3.10 no conseguimos el resultado esperado, que es obtener datos para generar una gráfica que muestre cómo disminuye la capacidad según se van deteriorando bloques de memoria.

Al principio pensamos que como este tipo de memorias tienen bloques reservados para que el deterioro de la capacidad no se vea reflejado, nosotros tampoco éramos capaces de verlo. Entonces hicimos pruebas más largas para evitar este posible problema. Pero solo conseguimos que la memoria diese errores de entrada/salida y la memoria dejase de estar accesible Figura 3.10.

En una prueba con una memoria con el estado inicial, código 3.3, llegamos a realizar 50.356.021 ciclos de escritura/borrado antes de registrar algún error. Como se puede ver en las últimas líneas del log 3.4 del script donde vamos apuntado cada ciclo de escritura/borrado y el estado de la memoria.

Listing 3.3: Estado inicial de la memoria

```
$ df --block-size=KB
```


Listing 3.5: Log del kernel cuando se produce el error en el sistema de ficheros

```
kern.log:May 15 19:32:56 donatello kernel: [1699652.958924]
    FAT: Filesystem error (dev sde1)
kern.log:May 15 19:32:56 donatello kernel: [1699652.972408]
    FAT: Filesystem error (dev sde1)
kern.log:May 15 19:32:56 donatello kernel: [1699652.972426]
    FAT: Filesystem error (dev sde1)
kern.log:May 15 19:51:50 donatello kernel: [1700786.560947]
    sd 12:0:0:0: [sde] Attached SCSI removable disk
messages:May 15 19:51:50 donatello kernel: [1700786.560947]
    sd 12:0:0:0: [sde] Attached SCSI removable disk
syslog:May 15 19:32:56 donatello kernel: [1699652.958924]
    FAT: Filesystem error (dev sde1)
syslog:May 15 19:32:56 donatello kernel: [1699652.972408]
    FAT: Filesystem error (dev sde1)
syslog:May 15 19:32:56 donatello kernel: [1699652.972426]
    FAT: Filesystem error (dev sde1)
syslog:May 15 19:51:50 donatello kernel: [1700786.560947] sd
    12:0:0:0: [sde] Attached SCSI removable disk
```

En otros dos casos no conseguimos obtener datos en el momento que la memoria empezó a fallar. Pero quedó completamente inservible, ya que aunque el sistema la reconoce 3.6, ni mantiene las particiones originales 3.7, ni se puede acceder a ella.

Listing 3.6: Log del sistema con memoria dañada

```
[ 4673.313154] usb 3-1: new high-speed USB device number 3
    using
ehci-pci
[ 4673.444924] usb-storage 3-1:1.0: USB Mass Storage device
detected
```

```
[ 4673.446328] scsi3 : usb-storage 3-1:1.0
```

Listing 3.7: Salida del fdisk

```
# fdisk /dev/sdc
Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x1670df5f
.
```

Nos dimos cuenta, con una memoria que habíamos usado para varias pruebas que aunque aparentemente estaba vacía ya que no contenía ningún fichero, ni tenía datos ocultos, `df` nos mostraba que tenía espacio usado.

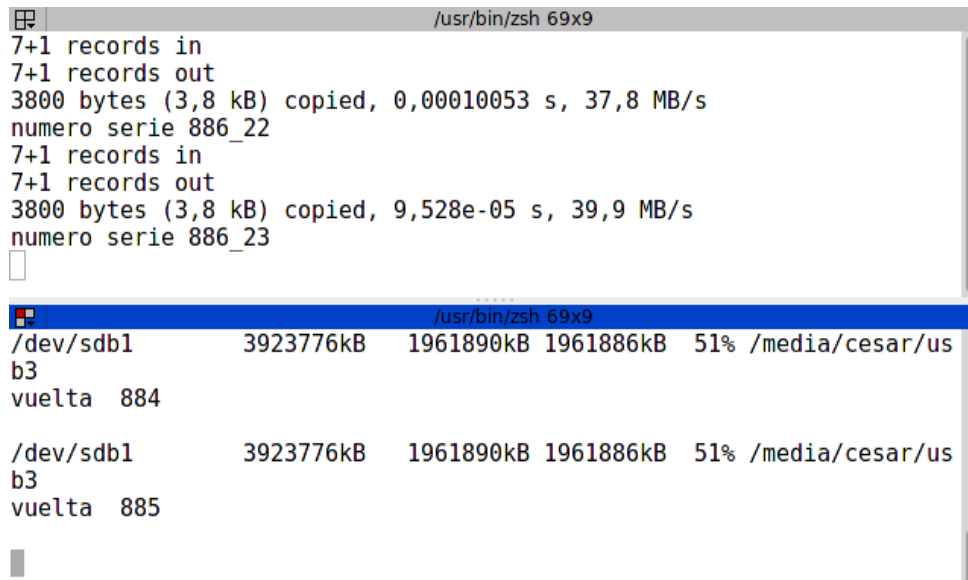
```
$ df
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sdc1	3831940	716640	3115300	19%	/media/cesar/usb1

Esto nos lleva a pensar que igual nos habíamos planteado mal esta sección, y no es la capacidad de memoria lo que disminuye con su uso, si no su espacio “útil”. Modificamos el script para poder sacar datos más precisos siguiendo esta hipótesis, que no terminamos por llevar a cabo ya que al formatear la memoria ese espacio “en uso” desapareció.

Por último hicimos otra versión cuyo resultado fue el script [3.11](#), en ella montamos y desmontamos el dispositivo antes del borrado y antes de la escritura, para asegurarnos que los datos que se están escribiendo llegan a materializarse en la memoria.

En la figura [3.11](#) puede verse el inicio de una de las pruebas para determinar la vida útil de una memoria y en la figura [3.12](#) como tras 26003 vueltas la capacidad y su espacio disponible no han variado. Por lo que no conseguimos los resultados esperados.



The image shows two terminal windows. The top window, titled '/usr/bin/zsh 69x9', displays the output of a memory test. It shows two iterations of copying 3800 bytes (3.8 kB) from /dev/sdb1 to /media/cesar/us b3. The first iteration took 0.00010053 seconds at 37.8 MB/s, and the second iteration took 9.528e-05 seconds at 39.9 MB/s. The bottom window, also titled '/usr/bin/zsh 69x9', shows the progress of the test. It displays the source and destination paths, the size of the data being copied (3923776kB), and the current progress (51%). The test is currently on 'vuelta 884' and 'vuelta 885'.

```
/usr/bin/zsh 69x9
7+1 records in
7+1 records out
3800 bytes (3,8 kB) copied, 0,00010053 s, 37,8 MB/s
numero serie 886_22
7+1 records in
7+1 records out
3800 bytes (3,8 kB) copied, 9,528e-05 s, 39,9 MB/s
numero serie 886_23

```

```
/usr/bin/zsh 69x9
/dev/sdb1      3923776kB    1961890kB 1961886kB  51% /media/cesar/us
b3
vuelta  884

/dev/sdb1      3923776kB    1961890kB 1961886kB  51% /media/cesar/us
b3
vuelta  885

```

Figura 3.11: Inicio de una prueba para determinar la vida de una memoria.

```

/usr/bin/zsh 69x20
8+1 records in
8+1 records out
4200 bytes (4,2 kB) copied, 0,000111654 s, 37,6 MB/s
numero serie 26004_26
8+1 records in
8+1 records out
4200 bytes (4,2 kB) copied, 0,000113411 s, 37,0 MB/s
numero serie 26004_27
8+1 records in
8+1 records out
4200 bytes (4,2 kB) copied, 0,000115227 s, 36,4 MB/s
numero serie 26004_28
8+1 records in
8+1 records out
4200 bytes (4,2 kB) copied, 0,000112469 s, 37,3 MB/s
numero serie 26004_29
8+1 records in
8+1 records out
4200 bytes (4,2 kB) copied, 0,000114159 s, 36,8 MB/s
[ ]

****
/usr/bin/zsh 69x20
b3
vuelta 25999

/dev/sdb1      3923776kB   1961890kB 1961886kB  51% /media/cesar/us
b3
vuelta 26000

/dev/sdb1      3923776kB   1961890kB 1961886kB  51% /media/cesar/us
b3
vuelta 26001

/dev/sdb1      3923776kB   1961890kB 1961886kB  51% /media/cesar/us
b3
vuelta 26002

/dev/sdb1      3923776kB   1961890kB 1961886kB  51% /media/cesar/us
b3
vuelta 26003

```

Figura 3.12: Estado de la prueba para determinar la vida de una memoria tras 26003 vueltas.

3.2. Código

3.2.1. Archivos marcados

`archivos_marcados_b3.sh`: este script genera ficheros desde un byte hasta el límite que se ponga en la variable `valor_final`. Se ha usado para generar datos para llenar la memoria y realizar pruebas.

Listing 3.8: Script que genera datos de varios tamaños.

```
#
# Crea ficheros de todos los tamaños posibles hasta llegar a
#   'valor_final'
#
valor_final=3907388
((tamfinal=valor_final*1024))

aux=2
cont=0
while [ $aux -le $tamfinal ]; do

    ((aux=aux+aux))
    ((cont++))

done

echo "auxiliar" $aux
echo "contado" $cont
rm datos/*
echo -e -n "\x00" > datos/datos0.txt

((aux=cont))
```

```
echo "vamos_a_contar_desde_1_a_" $aux

for (( a=1 ; a<=aux ; a++ ))

do

    ((b=a-1))

    cat datos/datos$b.txt > datos/datos$a.txt
    cat datos/datos$b.txt >> datos/datos$a.txt

    echo $b

    echo $a

done
```

3.2.2. Flash

Cada una de las versiones de este script se ha creado con una finalidad concreta tal como describimos en cada uno de ellos.

`flash_b2.sh` : su función es escribir y borrar muchas veces varios datos en una memoria, y monitorizar mediante la escritura en un log la evolución de su capacidad.

Listing 3.9: Script para flashear una memoria con un fichero un gran número de veces.

[illegible]


```
for (( a=1 ; a<3 ; a++ ))
do
    echo "numero_serie_${b}_${a}"
    echo -e -n "numero_serie_${b}_${a}" > datos/${b}_${a}
    datos.txt
    for (( c=1 ; c<200 ; c++ ))
    do
        echo -e -n "numero_serie_${b}_${a}" >>
datos/${b}_${a}datos.txt
    done
    #cat datos/datos.txt >> datos/${b}_${a}datos.txt
    #echo -e -n "numero serie ${b}_${a}" >> datos/${b}_${
    a}datos.txt
    dd if=datos/${b}_${a}datos.txt
of=/media/${dispo}/${b}_${a}datos.txt
    rm datos/*_datos.txt
done

rm -f /media/${dispo}/*datos.txt

df --block-size=KB | grep ${sdd} >> stam.txt
#cut -c32-52 tam.txt >> stam.txt

frase="vuelta_"
echo "$frase$vuelatas" >> stam.txt
echo "_" >> stam.txt

#sudo umount /media/usb0
```

```
#sudo mount /dev/sdc1 /media/usb0
```

```
done
```

flash_b3.sh: esta versión se ha hecho con la idea de tener archivos fácilmente localizables dentro de la imagen de 4GB que generamos y de la cual tenemos que sacar información y resultados.

Listing 3.10: Script para flashear una memoria con un archivo fácil de encontrar.

```
#
# Crea un fichero con un texto reconocible y su numero de
# serie
#
rm stam.txt
rm diff.txt
for (( b=0 ; b<10 ;b++))
do
    let vueltas=$b

for (( a=1 ; a<3 ; a++ ))
do
    echo "numeroserie_${b}_${a}"
    echo -e -n "En un lugar de la Mancha, de cuyo nombre
        no quiero
acordarme, no ha mucho tiempo que vivía un hidalgo de los de
    lanza en astillero, adarga antigua, rocín flaco y galgo
    corredor. Una olla de algo más vaca que carnero, salpicón
    las más
noches, y duelos y quebrantos los sábados, lentejas los viernes
    , algún palomino de añadidura los domingos, consumían las
```

```
tres partes de su
hacienda. El resto della concluían sayo de velarte, calzas de
velludo para las fiestas con sus pantuflos de lo mismo,
los días de entre
semana se honraba con su vellori de lo más fino. Tenía en su
casa una ama que pasaba de los cuarenta, y una sobrina que
no llegaba a los
veinte, y un mozo de campo y plaza, que así ensillaba el
rocín como tomaba la podadera. Frisaba la edad de nuestro
hidalgo con los
cincuenta años, era de complexión recia, seco de carnes,
enjuto de rostro; gran madrugador y amigo de la caza.
Quieren decir que tenía el
sobrenombre de Quijada o Quesada (que en esto hay alguna
diferencia en los autores que deste caso escriben), aunque
por conjeturas
verosímiles se deja entender que se llama Quijana; pero esto
importa poco a nuestro cuento; basta que en la narración
del no se salga un
punto de la verdad. "${b}_${a}" > datos/${b}_${a}datos.txt
#cat datos/datos.txt >> datos/${b}_${a}datos.txt
#echo -e -n "numero serie ${b}_${a}" >> datos/${b}_${
a}datos.txt
dd if=datos/${b}_${a}datos.txt of=/media/usb0/${b}_${
a}datos.txt
rm datos/*_datos.txt
done

dd if=/dev/sdc1 of=copias/${b}.iso
```

```
echo "vuelta_{$b}" >> diff.txt

if (($b != 0 ))
then
((c=b-1))
cmp -l copias/${b}.iso copias/${c}.iso >> diff.txt
fi

echo "" >> diff.txt

rm -f /media/usb0/*datos.txt

df --block-size=KB | grep sdc1 > tam.txt
cut -c32-52 tam.txt >> stam.txt

frase="vuelta_"
echo "$frase$vueltas" >> stam.txt
echo "_" >> stam.txt

done
```

flash_b4.sh: Es la última versión del script para flashear una memoria. El único cambio importante es que entre la escritura y el borrado de datos, monta y desmonta el dispositivo, con esto nos aseguramos que los datos son escritos correctamente en el dispositivo.

Listing 3.11: Última versión de script para flashear una memoria.

```
ruta="/media/cesar/usb3"
sdd="sdb1"
```

[illegible]

```
echo "umount"

sudo mount -o gid=1000,uid=1000 /dev/${sdd} ${ruta}

echo "mount"


df --block-size=KB | grep ${sdd} >> stam.txt


frase="vuelta_"
echo "$frase_$vueltas" >> stam.txt
echo "_" >> stam.txt


echo "-----"
echo "-----"
echo "vuelta"
echo $vueltas


done


exit
```

3.2.3. Monvimiento de datos

`mount.sh`: Este script se ha usado para comprobar si en el montado y desmontado, el dispositivo sufre algún tipo de cambio. Funciona montado y desmontado el dispositivo varias veces y creado una suma de MD5 que luego se puede comparar.

Listing 3.12: Script para verificar si en el montado y desmontado existe movimiento de datos.

```
rm md5.txt
```

```

for (( a=1 ; a<9999999999999999 ; a++ ))
do
    sudo umount /media/usb0
    sudo mount /dev/sdb1 /media/usb0
    echo "$a" >> md5.txt
    dd if=/dev/sdb1 | md5sum >> md5.txt

done

```

3.2.4. Tamaño de bloque

`tam_bloque_b1.sh`: este script usa los ficheros generados por el script 3.8 para generar uno con un tamaño fijo. Se ha usado para llenar una memoria hasta un punto en concreto, de esta manera sabíamos en que posiciones estábamos grabando el resto de los datos.

Listing 3.13: Script para generar ficheros de un tamaño concreto

```

df --block-size=KiB | grep sdb1 > tam.txt
tam_final=$(cut -c56-63 tam.txt)
echo $tam_final
((valor_final=tam_final-6))
((valor_final_bits=valor_final*1024))
echo "valor_ final" $valor_final_bits
final_binario=$(echo "obase=2; $valor_final_bits" | bc)
echo $final_binario > long.txt
echo "valor_ en_ binario" $final_binario
long=${#final_binario}

((long--))

```

```

total=0

((exp=long))

for ((a=1;a<=long;a++))
do
bit=$(cut -c$a-$a long.txt)
#echo "a="$a "bit sacado de long.txt" $bit
if((bit==1))
then
    ((exponente=2**exp))
#    echo "exponente" $exp
    echo $exp $exponente
    ((total=total+exponente))

    cat datos/datos$exp.txt >> datos/datosfin.txt
fi
((exp--))
done

echo $total

tam_bloque_b2.sh

```

Listing 3.14: Script para generar ficheros de un tamaño concreto

```

((valor_final=3907388))
#((valor_final=3907388/4))
((valor_final_bits=valor_final*1024))
echo "valor_□final" $valor_final_bits

```



```
final_binario=$(echo "obase=2;_ $valor_final_bits" | bc)
echo $final_binario > long.txt
echo "valor_en_binario" $final_binario
long=${#final_binario}

((long--))

total=0

((exp=long))
rm datos/datosfin.txt
for ((a=1;a<=long;a++))
do
bit=$(cut -c$a-$a long.txt)
#echo "a="$a "bit sacado de long.txt" $bit
if((bit==1))
then
    ((exponente=2**exp))
#    echo "exponente" $exp
    echo $exp $exponente
    ((total=total+exponente))

    cat datos/datos$exp.txt >> datos/datosfin.txt
fi
((exp--))
done

echo $total
```

3.2.5. Tabla de nombres

`crear_mismo_tam.sh`: la finalidad de este script es copiar el número máximo de archivos posibles dentro de una memoria, al principio se usó para varias pruebas, pero al final ha sido muy útil a la hora de sacar resultados sobre el funcionamiento de la tabla de nombres.

Listing 3.15: Script para crear muchos ficheros iguales

```
valor_final=3907388
((aux=valor_final/4))

echo "vamos_a_contar_desde_1_a_" $aux

for (( a=1 ; a<=aux ; a++ ))
do
    echo -e -n "\xAA$a" > /media/B030-FF46/$a.txt
    echo "vuelta_$a"
done
```

Resultados

4.1. Conclusiones

Los resultados a los que hemos llegado son los siguientes:

- El tamaño de bloque de memoria que impone el sistema de ficheros se mantiene a nivel físico.
- El montado y desmontado del dispositivo no genera cambios en él.
- Analizar las diferencias entre imágenes `iso` tan grandes como las que hemos manejado no es una tarea trivial.
- Son más frágiles los componentes que acompañan a la memoria que la propia memoria.
- Es necesario encontrar un procedimiento más eficiente para “flashear” memorias que el utilizado en este Proyecto si se quiere continuar con la línea de trabajo.

4.2. Líneas futuras

Este Proyecto deja varias puertas abiertas a futuras invesgitaciones.

Sería ideal intentar mejorar el script 3.11 hasta conseguir los resultados esperados. Uno de los puntos a mejorar es la sobrecarga del dispositivo cuando está tanto tiempo encendido, trabajando, se podría añadir tiempos de espera, de esta manera posiblemente las memorias no sufran el fallo del controlador o de componentes internos antes de obtener los datos del deterioro de la memoria.

Otro punto que sería interesante analizar es el moviento de datos entre sectores de forma más eficaz. Esto puede ser evitado usando memorias de menos capacidad, ordenadores con mayor procesado (aunque en este sentido creemos que el fallo de la herramienta `meld` para comparar ficheros tan grandes no depende del ordenador si no del programa en sí), o idear otras maneras de comparar imágenes `.iso` y obtener resultados facilmente interpretables. En nuestro Proyecto hemos desarrollado un pequeño programa en C que compara bit a bit dos ficheros binarios, pero este no llegó a ser funcional ya que era muy lento y pesado.

Una parte que no nos ha dado tiempo a tratar es la recuperación de datos de una memoria dañada. Más allá de las herramientas de software tipo ..., lo interesante es sacar los datos directamente de la memoria física. Teníamos pensando utilizar un prototipo con una FPGA y un lector de chips de los que se usan para flashear consolas que esta en fase de desarrollo.

Por último, y siguiendo esta línea, se podría diseñar un grabador con una FPGA de esa manera se quitaría la limitación del controlador y todos los componentes intermedios que pueden fallar antes que la memoria. El inconveniente de este punto es encontrar un dispositivo que aguante mucha carga de trabajo durante mucho tiempo, ya que los lectores de chips que hemos mencionado anteriormente no están pensados para estas aplicaciones.

Bibliografía

- [1] Toshiba America Electronic Components. Nand vs. nor flash memory. Toshiba America Electronic Components. http://maltiel-consulting.com/NAND_vs_NOR_Flash_Memory_Technology_Overview_Read_Write_Erase_speed_for_SLC_MLC_semiconductor_consulting_expert.pdf.
- [2] Kristopher Kubicki. Solid state society: The magic per Gigabyte reality arrives. Dynamite Data. <http://dynamitedata.com/news/cat=15/index.html>.
- [3] Kingston technology. Guía de productos de memoria Flash. Kingston technology. http://media.kingston.com/pdfs/FlashMemGuide_LA.pdf.
- [4] R. Bez, E. Camerlenghi, A. Modelli y A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91, April 2003.
- [5] Wikipedia contributors. Flash memory. Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Flash_memory&oldid=622001976.
- [6] convertronic.net. Serie de microcontroladores de ultra-bajo consumo de energía rh850/flx chip de memoria flash de 40 nm para aplicaciones en automoción. convertronic.net. <http://www.convertronic.net/Microcontroladores/serie-de-microcontroladores-de-ultra-bajo-consumo-de-energia-rh850flx-chip-de->
[html](http://www.convertronic.net/Microcontroladores/serie-de-microcontroladores-de-ultra-bajo-consumo-de-energia-rh850flx-chip-de-).

-
- [7] Douglas Perry. Replacing RAM with Flash can save massive power. Princeton University. <http://www.tomshardware.com/news/fusio-io-flash-ssdalloc-memory-ram,16352.html>.
 - [8] Anand Lal Shimpi. The impact of ssd on battery life. Anandtech. <http://www.anandtech.com/show/2445/16>.
 - [9] Oron Ogdan Amir Ban, Dov Moran. Architecture for a universal serial bus-based pc flash disk., April 2003.
 - [10] Wikipedia contributors. Multi-level cell. Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Multi-level_cell&oldid=605050764.
 - [11] Micron. Tlc, mlc, and slc devices. Micron. <http://www.micron.com/products/nand-flash/tlc-mlc-and-slc-devices>.
 - [12] Oron Ogdan Amir Ban, Dov Moran. Architecture for a universal serial bus-based pc flash disk., April 2003. <http://support.microsoft.com/kb/192322/es>.
 - [13] Escrito originalmente por John W. Eatonx. man de linux.
 - [14] Alberto Luna Fernández y Pablo Sanz Mercado. *Programación de Shell Scripts*. UAM Ediciones, 2011.
 - [15] GNU. Bash reference manual. www.gnu.org. <http://www.gnu.org/software/bash/manual/bash.htm>.
 - [16] SS64.com. An a-z index of the bash command line for linux. <http://ss64.com/bash/index.html>. <http://ss64.com/bash/index.html>.
 - [17] La comunidad de <http://git-scm.com>. Git fast-version-control. <http://git-scm.com>. <http://git-scm.com/about>.
 - [18] Wikipedia contributors. Criptografía asimétrica. Wikipedia, The Free Encyclopedia. http://es.wikipedia.org/w/index.php?title=Criptograf%C3%ADa_asim%C3%A9trica&oldid=73152741.

-
- [19] César Martín Cristóbal. Uso de túneles ssh y screen, Diciembre 2013. <http://dev.callepuzzle.com/uso-de-tuneles-ssh-y-screen/>.
- [20] guimi.net. Uso básico de screen en linux, November 2008. <http://guimi.net/blogs/hiparco/uso-basico-de-screen-en-linux/>.