Universidad Nacional Autónoma de México

Facultad de Ingeniería

Registro de datos en tarjetas de memoria SD CARD implementando los sistemas de archivos FAT16 y FAT32

T E S I S

QUE PARA OBTENER EL TÍTULO DE :

INGENIERO ELÉCTRICO ELECTRÓNICO
(ÁREA : ELECTRÓNICA)

PRESENTA :

ANA ANGÉLICA HERNÁNDEZ LÓPEZ

DIRECTOR DE TESIS :

M.I. LAURO SANTIAGO CRUZ

México, D.F. 2009

PARA :

Mi mamá, mi papá, mi hermana y brownie.

Agradecimientos:

A mi escuela, la Universidad Nacional Autónoma de México.

Al Instituto de Ingeniería de la UNAM, por las facilidades brindadas para el desarrollo de esta tesis.

A mi director de tesis, por todo lo que me ha enseñado.

A la coordinación de Ingeniería Sísmica del Instituto de Ingeniería de la UNAM, por su interés en el tema y apoyo con los recursos para desarrollar este proyecto.

A mis compañeros del Laboratorio de Instrumentación del Instituto de Ingeniería de la UNAM, por todas las experiencias compartidas.

Contenido

4 Pruebas

Figuras

Capítulo 2

Capítulo 3

Capítulo 4

Tablas

Capítulo 2

Capítulo 3

Cualquier dispositivo electrónico que grabe información durante un periodo de tiempo, para luego utilizarla, es considerado un colector de datos (*datalogger*). Esta definición incluye a muchos sistemas de adquisición de datos que usan una computadora como medio para almacenar datos en tiempo real. En la actualidad la mayoría de los fabricantes de *dataloggers* los consideran equipos autónomos, característica que lleva implícito el uso de memoria en el sitio para almacenar datos adquiridos por el dispositivo, para luego, exportarlos a una computadora. Los primeros *dataloggers* usaban como medio de registro de datos a la cinta magnética y al papel perforado. Los *dataloggers* están cambiando ahora más que nunca, incorporando nuevas tecnologías de almacenamiento y comunicación, la tecnología actual permite el registro de datos en memorias de estado sólido, algunos ejemplos: *Static Random Access Memory* (SRAM), *Electrically Erasable Programmable Read Only Memory* (EEPROM) y flash, además, su uso hace posible reducir el tamaño de estos equipos, cualidad que los hace aún más atractivos para aplicaciones que requieren de traslados a localidades remotas.

Usos típicos de los *dataloggers* incluyen estudios de campo, monitoreo de transportación, estudios de calidad, investigación y en general, donde se requiera registrar información de eventos que ocurran alrededor, sin tener a alguien necesariamente presente.

Las tarjetas de memoria flash, con gran popularidad en el mercado actual, son versátiles y compactas, pueden ser integradas a equipos con necesidades de almacenamiento considerable, que requieran de portabilidad y practicidad, son ideales en equipos de adquisición de datos como los *dataloggers*.

En particular, en el área de Ingeniería Sismológica del Instituto de Ingeniería de la Universidad Nacional Autónoma de México, se cuenta con equipos de detección y adquisición de eventos, llevados a campo para hacer registro de datos a través de computadoras portátiles, con tarjetas de expansión de memoria *Personal Computer Card* (PC Card) o módulos de memoria en circuitos integrados. Con la intención de colaborar en la actualización de estos equipos, en el presente trabajo se desarrolla un sistema de registro de datos, que deberá permitir el grabado de información

en tarjetas de memoria flash *Secure Digital* (SD). Esta primera etapa nos permite conocer la tecnología de registro de información en las memorias *Secure Digital Card* (SD CARD), utilizando para ello los sistemas de archivos *File Allocation Table* (FAT), FAT16 y FAT32. De esta manera, la información grabada en las memorias SD CARD podrá transportarse y ser leída por cualquier computadora compatible con el sistema de archivos FAT.

El presente trabajo escrito está constituído de seis capítulos. En el capítulo uno se presenta una breve introducción, así como las consideraciones tomadas para el desarrollo de este proyecto. El segundo capítulo cubre los conceptos y características de operación de los elementos que lo integran. En el tercero y cuarto capítulo se plantea la propuesta de solución y su implementación. El capítulo 5 muestra pruebas al sistema, y se finaliza este trabajo presentando las conclusiones en el último capítulo.

A lo largo del texto se utilizan términos que se definen inmediatamente en cuanto se mencionan, y si aplica, se proporciona una abreviación para cada uno de ellos. Un resumen de estos se encuentra en el glosario de términos, dentro de los apéndices.

1

Introducción

Ha sido una necesidad humana histórica preservar información a lo largo del tiempo. Para conseguirlo, se ha hecho uso de la tecnología presente en la época.

Un dispositivo electrónico capaz de retener información es un dispositivo de almacenamiento de datos, le llamamos una memoria. En la actualidad no existe un dispositivo de almacenamiento universal, sino distintos tipos de memorias con sus ventajas y desventajas correspondientes. Depende de la aplicación particular la elección del dispositivo más conveniente.

Las memorias pueden clasificarse de muchas formas: por la tecnología que usan, por la capacidad que tienen, si son o no removibles, etcétera. Si la memoria pierde su contenido en cuanto el sistema electrónico que la incluye se apaga, se le llama memoria volátil, de otro modo son memorias no volátiles, estas últimas no requieren de alimentación de energía para mantener su contenido.

En la clasificación de memorias no volátiles se encuentran las memorias tipo **flash**, tecnología dominante en aplicaciones que requieren cantidades grandes de almacenamiento no volátil de estado sólido, a un costo razonable. Esta tecnología es la que se usa en las *memory cards* o tarjetas de memoria, las aplicaciones que las utilizan incluyen computadoras

de escritorio, computadoras portátiles, teléfonos celulares, cámaras digitales, organizadores personales, reproductores de música, consolas de videojuegos, colectores de datos, etcétera. Como vemos, las tarjetas de memoria flash han ganado popularidad en aplicaciones portátiles, su bajo consumo de energía, tamaño reducido, resistencia a cambios climáticos, durabilidad, practicidad y bajo costo las hacen una opción muy atractiva para la electrónica de consumo actual. Algunas marcas de tarjetas de memoria flash que se encuentran comúnmente son: *MultiMedia Card* (MMC), *Secure Digital* (SD CARD), *Memory Stick* (MS), *Smart Media* (SM) y *Compact Flash* (CF). Una de las más populares y ampliamente usada en dispositivos muy diversos es la SD CARD.

Las tarjetas de memoria flash, así como los discos duros, las memorias USB, los discos compactos y demás dispositivos de almacenamiento, nos proporcionan el medio físico donde se guarda la información. Requieren de un equipo especializado que grabe en ellos datos que luego puedan consultarse. La manera en que la información se graba en los dispositivos de almacenamiento depende de la tecnología que utilicen; sin embargo, es común a todos ellos la necesidad de organizar los datos que guardan para encontrarlos y acceder a ellos cuando se necesite.

La forma de poner orden a los datos es a través de los sistemas de archivos.

Un sistema de archivos se puede pensar como una base de datos de propósito específico que manipula, organiza y le da una estructura a los datos almacenados, para que posteriormente puedan ser leídos por equipos que conozcan las reglas con las que se guardó la información. Entonces en el grabado de datos queda implícita la implementación de un sistema de archivos.

Actualmente se han desarrollado muchos sistemas de archivos. La mayoría de los sistemas operativos utilizan su propio sistema de archivos y permiten compatibilidad con otros, además del propio. Algunos ejemplos de sistemas de archivos pueden ser: *Hierarchical File System Plus* (HFS PLUS) en computadoras con sistema operativo Macintosh, *Second Extended File System* (Ext2) en sistemas Linux, *File Allocation Table* (FAT) y *New Technology File System* (NTFS) en sistemas Windows. Cabe comentar que el sistema operativo Windows es el más usado en las computadoras y el sistema de archivos FAT el más común.

Este trabajo tiene por objetivo el diseño y la implementación de un módulo de almacenamiento de datos, cuyo medio de registro es a través de tarjetas de memoria SD, de forma tal que se puede dar lectura a éstas para obtener la información grabada. La lectura de las tarjetas se hace de forma convencional, por medio de un lector externo para este tipo de memorias que se conecta a una computadora, o bien a través de una computadora con lector integrado. Para cualquier caso, los datos son leídos y se muestran disponibles al usuario.

1.1. Consideraciones

El proyecto comprende la comunicación, el grabado y la recuperación de los datos registrados en la memoria SD CARD. Para los fines de este trabajo, el tipo de información grabada es irrelevante, así como la fuente de donde ésta proviene.

Así, el proyecto desarrollado se enfoca en tres aspectos funcionales:

1. El uso de memorias SD CARD y su tecnología.
2. El registro de información de interés en estas memorias.
3. Recuperación de la información registrada en ellas.

2

En este capítulo se presenta un panorama general de las memorias SD CARD, del sistema de archivos FAT y de las comunicaciones serie utilizadas.

El capítulo concluye comentando brevemente sobre el microcontrolador empleado en este proyecto.

2.1. Tecnología de las tarjetas de memoria

Los medios de almacenamiento de información que usamos vienen en una variedad de formas que sirven a distintos propósitos. En las computadoras personales o PC's, uno de los tipos de almacenamiento más comunes son las memorias *Read Only Memory* (ROM), proveen un almacenamiento no volátil y son usadas para guardar programas que se quieren tener disponibles todo el tiempo, como el *Basic Input/Output System* (BIOS) para iniciar el sistema. Mientras que en su concepción inicial el contenido de la ROMno estaba destinado a cambiarse, actualmente existen diferentes tipos de ROMy su contenido es modificable en distintos grados. Ejemplos de ROM son la *Programmable Read Only Memory* (PROM), *Erasable Programmable Read Only Memory* (EPROM) y EEPROM.

Las memorias EEPROMson el tipo más flexible de memorias ROM, son borradas y programadas eléctricamente byte a byte a través de *software*.

Las memorias **flash** son un tipo específico de EEPROM, que son borradas y programadas por bloques de bytes. Con costo significativamente menor que las EEPROM, las memorias flash se han vuelto la tecnología dominante de estado sólido en aplicaciones que requieren grandes cantidades de almacenamiento no volátil. Además, el manejo por bloques las provee de un rápido acceso al contenido, comparado con los tiempos utilizados por las EEPROM's convencionales, y cuando son empacadas en tarjetas de memoria, se les dotada de gran resistencia.

El tiempo de vida de las memorias FLASH está limitado por el desgaste provocado a sus celdas de memoria en los procesos de escritura y borrado. Típicamente, el tiempo de vida por cada bloque de bytes utlizado es de cien mil ciclos de borrado.

2.1.1. Tarjetas de memoria

Los dispositivos de memoria de estado sólido más usados incluyen circuitos integrados con memorias tipo flash, dispuestos en pequeñas tarjetas removibles. Se les llaman comúnmente tarjetas de memoria flash, o tarjetas de memoria.

Además del tipo flash, existen otros estándares de memorias que pueden ser usados en las *memory cards*, por ejemplo: *Non-Volatile Random Access Memory* (NVRAM), y memorias volátiles como la *Synchronous Dynamic Random Access Memory* (SDRAM) con batería de respaldo. Sin embargo, el uso de memoria tipo flash es el más popular.

Las tarjetas de memoria contienen varios circuitos integrados en su interior, entre ellos está la propia memoria flash y un microcontrolador, como se muestra en la figura 2.1. Aunque no todas las tarjetas de memoria en el mercado tienen un microcontrolador.

Fig. 2.1. Interior de una tarjeta de memoria.

El microcontrolador se encarga de controlar las operaciones en el arreglo de celdas de la memoria, y de la comunicación con el *host* al que la tarjeta se encuentra conectada. El *host* es el que procesa la información, de tal forma que éste almacena, lee y escribe la información en la tarjeta.

Los circuitos integrados se interconectan en un circuito impreso, que le da dureza y complejidad a la tarjeta. El circuito impreso esta rodeado por una marco de metal y una cubierta de plástico que expone los contactos eléctricos de la *memory card*.

Distintas marcas de tarjetas de memoria se encuentran disponibles en el mercado, algunas de ellas se muestran en la figura 2.2.

Fig. 2.2. Tarjetas de memoria.

En las tarjetas de memoria las diferencias físicas entre ellas son las más obvias, no así las diferencias en los estándares de comunicación. Algunas se apegan a los estándares de la *Personal Computer Memory Card International Association* (PCMCIA), mientras que otras tienen estándares propios.

2.2. Memorias Secure Digital

La tecnología y el estándar de las tarjetas de memoria *Secure Digital* es propietaria, ha sido desarrollada por el grupo SD (Panasonic, SanDisk y Toshiba), que comparten derechos de autoría con la asociación SD CARD (*SD CARD Association*).

Las especificaciones de las tarjetas SD nacieron del viejo estándar desarrollado para las memorias MMC. Las dimensiones físicas o *form factor* , interfaz eléctrica y protocolo de comunicación, son parte de las especificaciones de la SD CARD.

Las principales diferencias entre las distintas tarjetas de memoria disponibles en la actualidad son la complejidad, consumo de energía y dimensiones.

La interfaz eléctrica de la SD es relativamente simple, así como su interfaz física. Su consumo de energía no sobrepasa los 100[mA] cuando está activa, y su tamaño reducido aventaja a las de tecnologías que le compiten. Tienen el potencial para dominar el mercado de las memorias miniatura en unos años.

La figura 2.3 muestra una comparación de los distintos tamaños de las memorias SD.

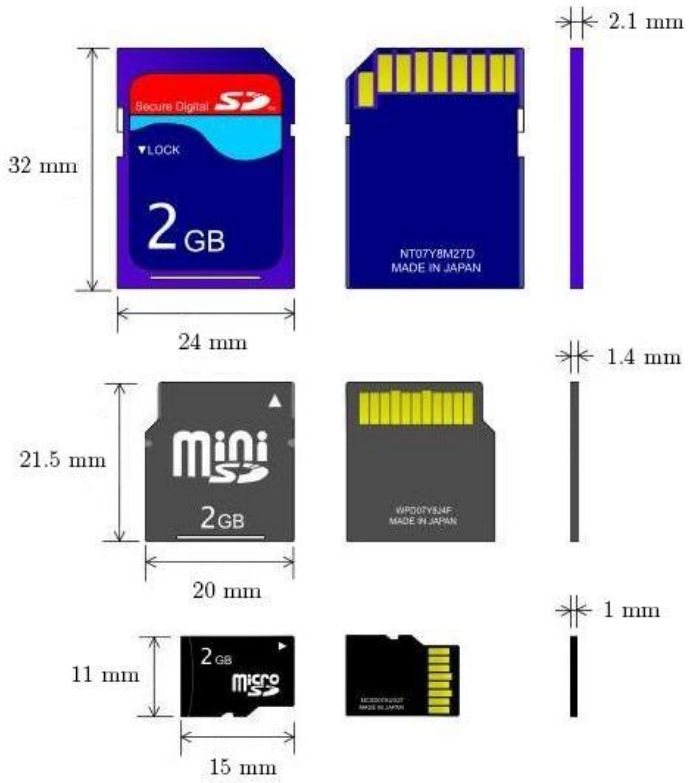


Fig. 2.3. Tamaños de las memorias SD CARD.

2.2.1. Especificaciones simplificadas versión 2.00

Para usar las tarjetas de memoria se requiere conocer las especificaciones de las mismas, y es necesario adquirir una licencia para acceder a las especificaciones completas de la SD CARD. A pesar de ello, es posible usar su tecnología a través de la versión simplificada de estas especificaciones, la *Physical Layer Simplified Specification Version 2.00*, que no tienen costo y están disponibles en el sitio oficial de la SD CARD.[1]

Las especificaciones con licencia cubren la interfaz física y el protocolo de comunicacion oficial de la SD CARD, basado en una interfaz de nueve terminales. Sin embargo, se puede tener acceso a las tarjetas a través de un segundo tipo de comunicación, al que hace referencia la versión simplificada.

Los dos modos de comunicación son el *SD Bus Mode* (modo SD) y el *SPI Bus Mode* (modo SPI). El primero es el protocolo de comunicación propio de la tarjeta de memoria SD, utiliza nueve líneas para la comunicación: cuatro para datos de forma paralela, una línea de comandos, una línea para el reloj y tres más para energizar a la tarjeta. El modo serie, o modo SPI, es a través del protocolo *Serial Peripheral Interface*. Éste es un protocolo de comunicación secundario, usa un extracto del protocolo oficial de la SD CARD y una parte del total de comandos. Esta restricción de comandos no es realmente importante, pues simplemente el modo SPI no los utiliza. Así, una implementación completamente funcional se puede lograr usando este modo, y es el que se utiliza en este proyecto.

2.2.2. Especificaciones de la memorias Secure Digital

Nos referiremos a la *Physical Layer Simplified Specification Version 2.00*, como **las especificaciones** utilizadas en este proyecto. Éstas dividen a las tarjetas de distintas formas, y para nuestro fin es de interés la clasificación por capacidad de almacenamiento, o simplemente por capacidad. Se dividen en dos tipos: SD estándar (SD) y SD *High Capacity* (SDHC).

A partir de este momento y de manera general, a las tarjetas *Secure Digital*, sin distinción de tipo, las identificaremos como **memorias SD**

[1] www.sdcard.org

CARD o simplemente **SD CARD**; a las memorias *Secure Digital* estándar las identificaremos como **SD** y a las memorias *Secure Digital High Capacity* como **SDHC**.

Antes de continuar con el tema, es conveniente hacer la aclaración sobre las unidades que se utilizan para representar la cantidad de información almacenada.

Prefijos Las computadoras guardan la información digital en potencias de base dos, utilizando el sistema binario. Mientras que los humanos usamos la numeración decimal, expresada en potencias de base diez. Véase la siguiente tabla.

$2^{10} = 1\ 024$	$\approx$	$10^3 = 1\ 000$
$2^{20} = 1\ 048\ 576$	$\approx$	$10^6 = 1\ 000\ 000$
$2^{30} = 1\ 073\ 741\ 824$	$\approx$	$10^9 = 1\ 000\ 000\ 000$

Tabla 2.1. Base binaria vs. base decimal.

Observando la similaridad entre ambos sistemas, por conveniencia se adoptaron las abreviaciones utilizadas en los números decimales para ser aplicados a los números binarios.

El *Institute of Electrical and Electronics Engineers* (IEEE) ha propuesto para eliminar la confusión, una nueva convención nombrando abreviaciones en los números binarios. Bajo esta propuesta, la tercera y cuarta letra de los prefijos usados en los números decimales, se cambian por "bi". Por ejemplo el prefijo 'Mega' se convierte en 'Mebi.' Así un Megabyte será 10^6 bytes, y un Mebibyte será 2^{20} bytes. La abreviación será 1[M*i*B] en vez de 1[MB] [**1**]. Véase la tabla 2.2.

Sin embargo, las viejas costumbres persisten y no es común encontrar aplicada esta nomenclatura. Hecha la aclaración, conservaremos en este texto los prefijos: *k*B, para referirmos a 2^{10} bytes; MB, para referirnos a 2^{20} bytes; GB, para referirnos a 2^{30} bytes y TB para referirnos a 2^{40} bytes.

De acuerdo a las especificaciones, las memorias SD soportan capacidades hasta de 2[GB]. Las memorias SDHC soportan capacidades mayores a 2[GB] y hasta 32[GB] [**2**].

DECIMAL			
Nombre	**Abreviación**	**Potencia**	**Valor**
kilobyte	kB	10^3	1 000
Megabyte	MB	10^6	1 000 000
Gigabyte	GB	10^9	1 000 000 000
Terabyte	TB	10^{12}	1 000 000 000 000
BINARIO			
Nombre	**Abreviación**	**Potencia**	**Valor**
kibibyte	kiB	2^{10}	1 024
Mebibyte	MiB	2^{20}	1 048 576
Gibibyte	GiB	2^{30}	1 073 741 824
Tebibyte	TiB	2^{40}	1 099 511 627 776

Tabla 2.2. Prefijos binarios bajo la propuesta de la IEEE.

En Abril del 2001, se dio a conocer la primera versión simplificada de las especificaciones de la SD CARD, siendo apta para tarjetas de memoria con capacidades menores a 2[GB], las SD. En Septiembre del 2006, la segunda versión simplificada de las especificaciones estuvo disponible, cubriendo memorias con capacidades de hasta 32[GB], abarcando las memorias SD y SDHC. En Abril del 2009, las más recientes especificaciones se pusieron al alcance de los miembros con licencia; comprenden tarjetas con capacidades mayores de 32[GB] y hasta 2[TB]. Es la siguiente generación de memorias SD CARD, las *SD eXtreme Capacity* (SDXC).[2]

En la tabla 2.3 se muestran algunas características de las memorias mencionadas, corresponden a las de la figura 2.4.

Existen también los tamaños mini y micro: miniSD, miniSDHC, microSD, microSDHC y microSDXC.

Los tipos SD y SDHC mostrados en la figura 2.4 son los que se usaron en este proyecto.

[2] http://www.sdcard.org/developers/tech/sdxc/

Tipos:	**SD**	**SDHC**	**SDXC**
Capacidad [GB]	2*	4 – 32	32 – 2000**
Sistema de archivos	FAT16	FAT32	exFAT
Voltaje de operación [V]	2.7 – 3.6		
Número de terminales	9		
Dimensiones [mm]	$32 \times 24 \times 2.1$		
Peso [g]	2		

* Hasta 2 [GB].
** Más de 32 [GB] y hasta 2 [TB].

Tabla 2.3. Tipos de memorias SD CARD.

Fig. 2.4. Memorias SD, SDHC y SDXC.

2.2.3. Modo serie de comunicación de datos

El *Serial Peripheral Interface* (SPI) es un un protocolo serie síncrono, extremadamente popular para comunicar dispositivos periféricos con microcontroladores. Su popularidad es la principal ventaja de la comunicación SPI, la mayoría de los microcontroladores tienen implementado este protocolo y prácticamente con cualquiera de ellos se puede establecer comunicación con las tarjetas de memoria. La desventaja del modo SPI radica en el desempeño respecto al modo SD, que es una comunicación paralela.

Protocolo El protocolo de comunicación tiene una estructura 'comando–respuesta'. Todos los comandos son iniciados y enviados por el maestro, el microcontrolador (*host*), al esclavo, que es la tarjeta. La tarjeta contesta los comandos siempre con una respuesta.

Todos los bytes transmitidos por el *host* y por la tarjeta inician con el envío de su byte más significativo.

Las memorias SD CARD manejan 89 comandos. De éstos, los comandos de lectura y escritura involucran transferencia de datos. Los datos se transmiten y reciben vía *tokens*. Un *token* es un byte con un valor especial, y se utilizan cinco de ellos; tres de estos *tokens* tienen valores fijos y el valor de los dos restantes depende del estado de la transferencia de datos.

La tarjeta envía un *token* por cada bloque de datos escrito exitosamente en ella. En la escritura y lectura de un solo bloque de datos, así como en la lectura de múltiples bloques, un *token* encabeza a los bytes de información. Si la operación de lectura falla, la tarjeta envía un *token* de error. Y en la escritura de múltiples bloques, se involucran *tokens* de inicio y fin que encapsulan a los bytes de datos.

La respuesta de la tarjeta no es inmediata. Una vez que recibe un comando se debe que esperar a que emita una respuesta. Transmitida la respuesta y dependiendo del comando, la tarjeta envía datos o una condición de error.

Existen siete tipos de respuesta: R1, R1b, R2, R3, R4, R5 y R7, todas con cero como bit de inicio, pero cada una con un formato particular. Las respuestas R4 y R5 son reservadas.

Si la tarjeta no tiene información que enviar al *host*, entrará en un estado inactivo o *idle* de unos, es decir, entrará en un estado de espera manteniendo su terminal de salida de datos en 'alto'. De la misma forma, el *host* se encontrará en estado *idle* cada vez que le envíe a la tarjeta sólo unos.

Los bloques de datos tienen una longitud fija. Como el host tiene siempre conocimiento previo del número de bytes que espera recibir en cada transacción, la respuesta de la tarjeta no incluye la longitud de los datos enviados. El estado *idle* en la tarjeta no puede ocurrir hasta que los datos hayan sido enviados, así estos se transmiten sin alteración.

Toda la información enviada por el *bus* se alínea por bloques de ocho bits, de acuerdo a los límites marcados por las transacciones de la comunicación SPI.

Todas las transacciones entre el *host* y la tarjeta están protegidas por un código de detección de error, que es un bloque de bytes adjuntado al

final de la información enviada, el *Cyclic Redundancy Code* (CRC). La tarjeta considera dos algoritmos de CRC, el CRC7 y el CRC16 [**2**]. El CRC es opcional en el modo SPI y está deshabilitado por *default*.

Formato de los comandos Todos los comandos se forman por bloques de seis bytes, como muestra la figura 2.5.

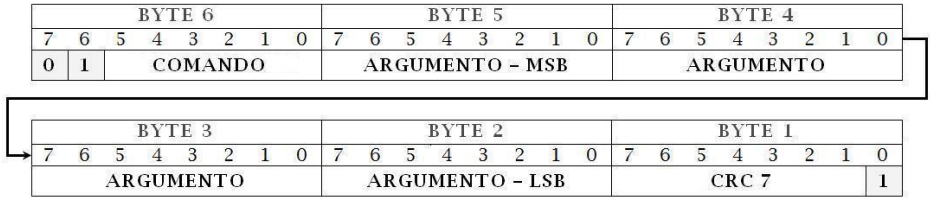


Fig. 2.5. Estructura de los comandos.

El índice o número del comando se especifica del bit 0 al bit 5 del byte 6, el bit 6 y 7 son los bits de transmisión y de inicio respectivamente, y tienen un valor fijo. Los siguientes cuatro bytes forman el argumento del comando, si es que lo requiere; el byte 5 corresponde al byte más significativo del argumento, y el byte 2 al menos significativo. Del bit 7 al bit 1 del último byte se escribe el CRC7 sobre los bytes 6 al 2, y el último bit de este byte es el bit de fin.

Existen dos tipos de comandos: los estándar y los de aplicación específica. Los comandos estándar se identifican por un número del 0 al 63. Algunos de ellos son de uso exclusivo para el modo SD y otros son reservados. Los comandos de aplicación específica son 25 y no llevan una numeración continua.

Respecto al *host*, la única diferencia entre los comandos estándar y los de aplicación específica es la forma en que los envía a la tarjeta. Los comandos de aplicación específica van siempre precedidos del comando estándard 55, que le informa a la tarjeta que el comando, a continuación de él, no es estándar, si no, de aplicación específica. En contraste, los comandos estándar se envían tal cual, sin ser ser precedidos por algún otro comando.

Secuencia de inicio Sin importar el modo, SD o SPI, antes de que cualquier comunicación se intente con la memoria SD CARD, es necesario

seguir una secuencia de inicio previa a la secuencia de inicialización: se energiza la SD CARD, se da un retardo de inicialización y se procede a la secuencia de inicialización. Ver la figura 2.6.

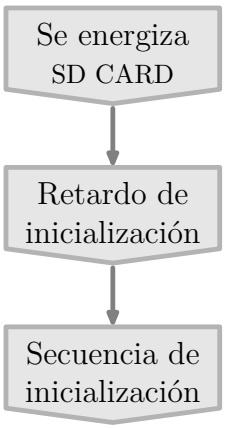


Fig. 2.6. Secuencia de inicio de las memorias SD CARD.

Para entrar al modo SPI (ver figura 2.7) se energiza la tarjeta y a continuación le sigue un retardo de inicialización, que consiste en el envío de 74 pulsos de reloj, tiempo que le permite a la tarjeta iniciar sus registros internos. La secuencia de inicialización o secuencia de comandos de inicialización, comienza con el envío del comando cero.

Inicialización El *host* debe dar un *reset* a la tarjeta enviándole el comando cero. Esto reinicia a la tarjeta y la instruye para entrar al modo SPI; el CRC en el modo SPI se ignora, pero este primer comando debe tener un valor válido de CRC, pues el modo de inicio es por *default* el SD, y, en este estado, la tarjeta todavía no ha entrado al modo SPI.

La tarjeta contesta el comando cero con la respuesta tipo R1, ver figura 2.8. En general, la respuesta R1 la envía la tarjeta después de la recepción de cada comando, con excepción del comando 13. La respuesta R1 es un byte que indica la presencia de errores, los que se señalan con '1' en el bit que corresponde al error marcado. A esta altura de la inicialización, R1 debe tener solamente encendido el bit que corresponde al 'estado *idle*'. Este bit en uno significa que la tarjeta ha entrado al estado *idle*, entonces el *reset* ha sido efectivo, y la SD CARD está en modo SPI.

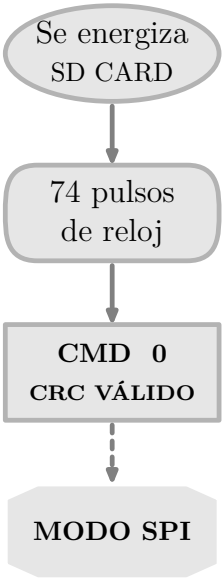


Fig. 2.7. Secuencia de entrada al modo SPI.

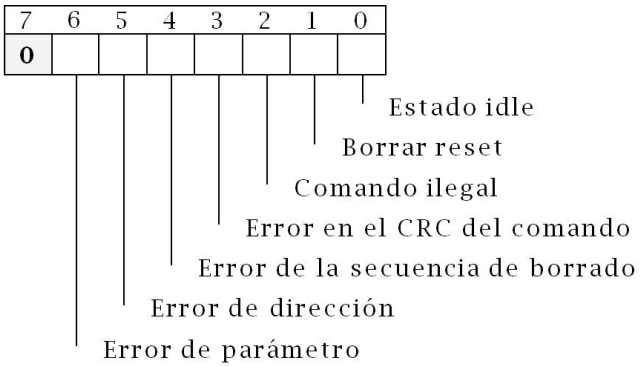


Fig. 2.8. Formato de la respuesta R1.

A continuación el host envía el comando ocho. Este comando tiene dos funciones:

– Verifica que la tarjeta pueda operar con el voltaje del *host*.
– Habilita la expansión de comandos y respuestas.

El objetivo del comando es verificar la condición de la interfaz con la tarjeta de memoria, está definido para inicializar a las tarjetas SD CARD conformes con la versión 2.00 de las especificaciones. A través de este comando el *host* pueden accesar a las memorias SDHC. Otros fallan al inicializarlas, como se ve en la figura 2.9. Es por lo tanto obligatorio para el *host*, conforme con la versión 2.00, el envío del comando ocho.

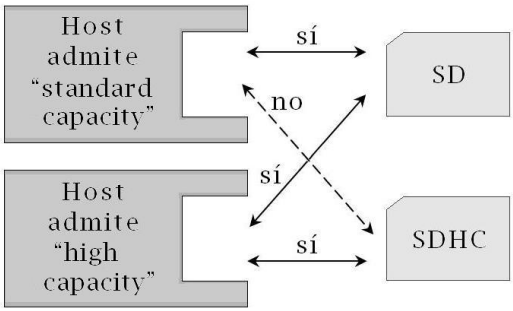


Fig. 2.9. Compatibilidad Host–SD CARD.

La tarjeta analiza el argumento del comando y le verifica al *host* la validez de la condición de operación a través de la respuesta tipo R7, de cinco bytes.

En la parte superior de la figura 2.10 se indica el orden de los 40 bits que forman la respuesta. La nomenclatura 'bX,BY' se debe leer como 'el bit X del Byte Y'; por ejemplo, b4,B2 será el bit 4 del Byte 2 de la respuesta R7.

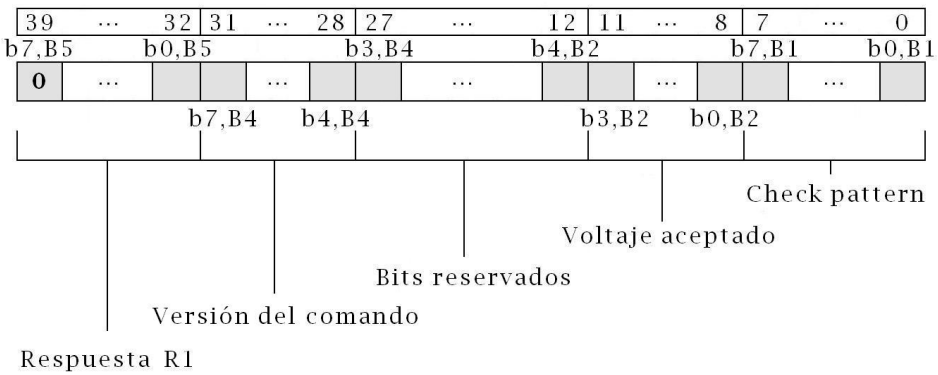


Fig. 2.10. Formato de la respuesta R7.

La secuencia continúa con el comando 58, su envío no es obligatorio. Está diseñado para que el *host* tenga una manera de identificar a las tarjetas que no verifican su rango de voltaje.

La respuesta de la tarjeta al comando 58 es de cinco bytes y es tipo R3. Se muestra en la figura 2.11.

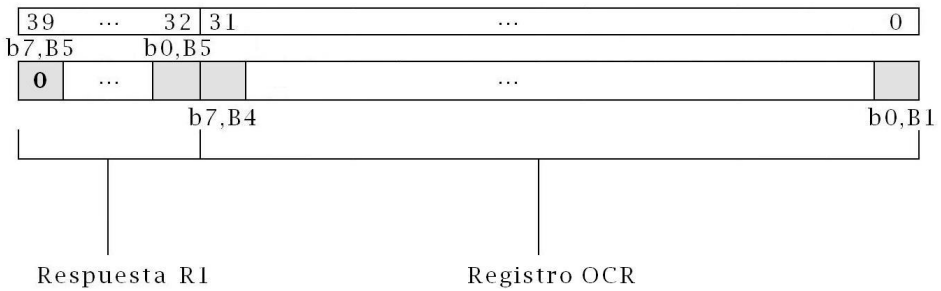


Fig. 2.11. Formato de la respuesta R3.

Si la tarjeta no verifica el rango de voltaje del *host*, el proceso de inicialización deberá detenerse y la tarjeta no podrá usarse.

A continuación se envía el comando de aplicación específica 41. Este comando activa el proceso de inicialización de la tarjeta; la respuesta es tipo R1. La tarjeta usa el bit de estado *idle* de la respuesta para informar al *host* si la inicialización ha sido completada: si el bit cambia su valor a '0' el proceso de inicialización se ha completado; si se mantiene en '1', la tarjeta sigue inicializándose.

El comando 41 de aplicación específica deberá enviarlo el *host* repetidamente a la tarjeta, hasta que el bit de estado *idle* de la respuesta R1 sea '0'.

Completada la secuencia de comandos de inicialización, el *host* verifica la respuesta del comando 58, comando que envía para finalizar el proceso.

En el diagrama de flujo de la figura 2.12 se resume la inicialización de la SD CARD.

El diagrama es una simplificación del algoritmo de inicialización. Se han obviado varias consideraciones que se retomarán en el capítulo siguiente.

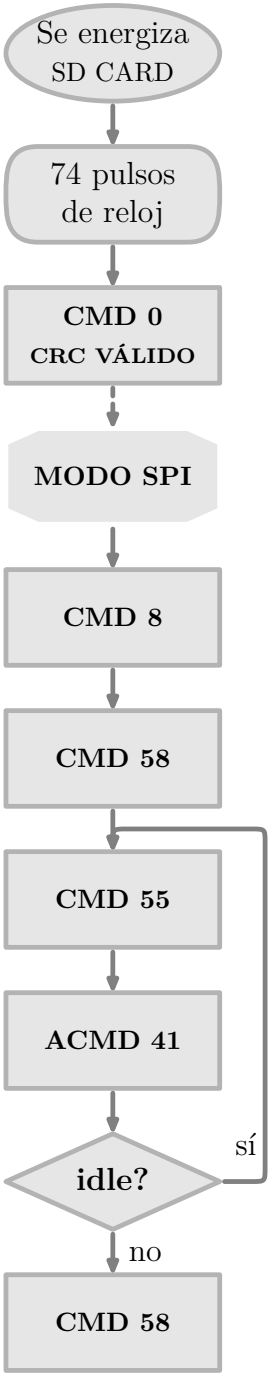


Fig. 2.12. Algoritmo de inicialización.

2.2.4. Detección de errores

Las técnicas de detección de error proporcionan al receptor un medio para determinar si el mensaje que recibió del transmisor se ha corrompido, generalmente por efecto de ruido en el canal de comunicación.

La idea tras las técnicas de detección de error es la siguiente: el transmisor construye un valor (llamado *checksum*) que es una función del mensaje, lo adjunta al final de éste, y lo envía al receptor. El receptor usa la misma función para calcular el *checksum* del mensaje recibido y lo compara con el *checksum* adjuntado, para ver si el mensaje se recibió correctamente.

El término *checksum* se usaba para describir los primeros algoritmos, basados en sumas. El nombre se ha adoptado, y toma ahora un significado más amplio, abarcando algoritmos más sofisticados como los del CRC.

La figura 2.13 muestra el ejemplo de un mensaje con checksum.

Fig. 2.13. Mensaje con *checksum*.

El CRC protege contra errores de transmisión en el *bus* a los comandos, respuestas y a las transferencias de datos entre el *host* y la SD CARD.

La idea básica de los algoritmos CRC es tratar al mensaje como un gran número binario, dividirlo entre otro número binario fijo, y hacer que el residuo de la división sea el checksum. Una vez recibido el mensaje, el receptor calcula la misma división y compara el residuo obtenido con el checksum.

Los algoritmos CRC utilizan "polinomios"; esto significa simplemente que el divisor, el dividendo (mensaje), el cociente y el residuo (*checksum*), se ven como polinomios con coeficientes binarios. Por ejemplo, el polinomio que corresponde al número binario 10111, es: $1 \times x^4 + 0 \times x^3 + 1 \times x^2 + 1 \times x^1 + 1 \times x^0$ o simplemente $x^4 + x^2 + x^1 + x^0$. En general se dice que estos algoritmos operan usando aritmética de polinomios, que es la aritmética binaria, sin acarreos.

Para calcular la división se necesita un dividendo (que está dado por el mensaje) y un divisor (llamado "polinomio generador" o "polinomial") que es el parámetro principal de cada algoritmo CRC.

Las memorias SD CARD implementan dos algoritmos de CRC: el CRC7 y el CRC16. El CRC7 se genera para cada comando enviado y el CRC16 para cada bloque de datos transferido. Las especificaciones proporcionan los polinomios generadores para cada uno de estos algoritmos.

Ejemplo de cálculo El algoritmo del CRC es simplemente el cálculo de una división operada con aritmética binaria sin acarreos. El único truco es que el dividendo se "completa" o se le agregan tantos ceros como la posición del '1' más significativo del divisor. Se muestra a continuación un ejemplo.

Mensaje original	:	1101011011
Polinomio generador	:	10011
Mensaje después de agregar ceros	:	11010110110000

El '1' más significativo del polinomio generador (divisor) es el bit 4, es decir, el '1' está en la posición cuatro; se agregan entonces cuatro ceros al mensaje original, formando el dividendo. La división es como se muestra en la figura 2.14.

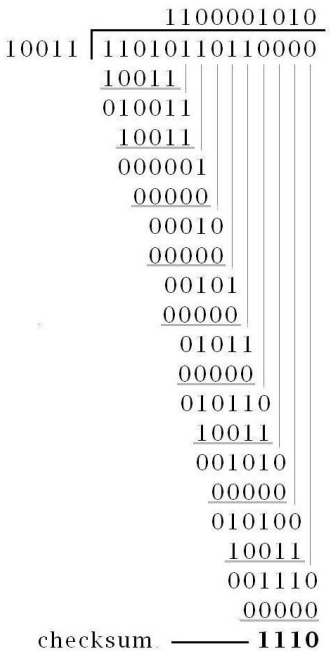


Fig. 2.14. Ejemplo de cálculo del CRC.

2.3. Sistema de archivos

Un sistema de archivos se refiere a las estructuras que se utilizan para organizar datos dentro de una unidad de almacenamiento; es un método para almacenar y organizar archivos y su contenido.

Un archivo es una colección de bytes almacenados en grupo con un nombre para identificarlo, puede contener cualquier tipo de información: imagenes, texto, una película, etcétera. Lo significativo de un archivo lo determina su contenido y el programa que se utiliza en él.

En una PC el sistema operativo utiliza el sistema de archivos para almacenar y recuperar información en una unidad cuando se requiere; en otras palabras, el sistema de archivos sirve como interfaz entre el sistema operativo y la unidad de almacenamiento. Por ejemplo, cuando una aplicación como MS Word pide leer un archivo que está en el disco duro, el sistema operativo (Windows) le pide al sistema de archivos (FAT) que abra el archivo. Ver la figura 2.15.

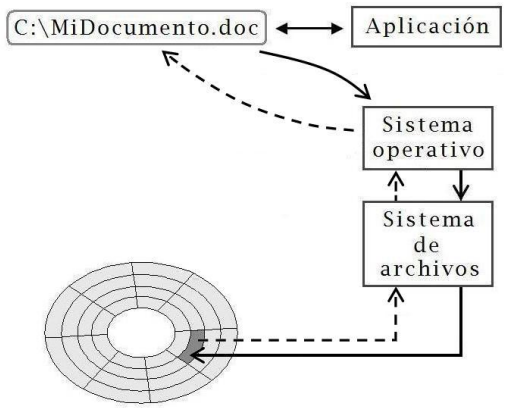


Fig. 2.15. Sistema operativo y sistema de archivos.

El sistema de archivos sabe donde se guardó el archivo, encuentra el espacio de disco donde está almacenado y entrega los datos al sistema operativo.

Todo sistema operativo tiene su sistema de archivos que abstrae y administra la información guardada en una unidad o dispositivo de almacenamiento. Las características particulares del sistema de archivos dependen del sistema operativo.

La información guardada en una unidad de almacenamiento puede ser o no accesada a través de un sistema de archivos, pero si se quiere formalizar la lectura de su contenido con una computadora debe utilizarse uno.

La computadora da formato o 'formatea' una unidad o dispositivo de almacenamiento para organizarlo y prepararlo para recibir datos. Al formatearlo se dice que recibe un sistema de archivos.

El formateo se puede comparar con empezar una biblioteca; antes de poder consultar los libros y tener acceso a ellos, se instalan primero las repisas sobre las que se colocan los libros y el sistema de catálogos. Hecho esto, la biblioteca está lista para recibir los libros. De forma similar, al formatear un dispositivo de almacenamiento, se instalan las estructuras del sistema de archivos que le permiten al dispositivo ser usado para almacenar datos, quedando listo para recibir información.

2.3.1. Sistema de archivos FAT

Existen distintos sistemas de archivos y cada uno utiliza métodos diferentes para organizar la información. Se estima que el 90% de las computadoras utiliza el sistema de archivos FAT, siendo el más popular. Para una mejor comprensión del sistema FAT es necesario definir algunos de los términos que éste emplea.

Conceptos básicos Para lograr un uso eficiente y un buen desempeño del dispositivo de almacenamiento, el sistema de archivos FAT divide el espacio de almacenamiento en "pedazos discretos" llamados **clusters**.

Un cluster es un bloque continuo de espacio de almacenamiento que agrupa **sectores**; los sectores en un cluster son continuos.

Un sector es la unidad más pequeña en la que se puede dividir el espacio de almacenamiento, su tamaño es de 512 bytes.

El tamaño de un cluster lo determina el sistema de archivos y depende del espacio de almacenamiento. Volúmenes grandes de almacenamiento determinan tamaños de cluster mayores.

El tamaño de un cluster se expresa generalmente en términos de la cantidad de sectores que contiene, por ejemplo: 4 $\frac{sectores}{cluster}$ (2048 bytes), 64

$\frac{sectores}{cluster}$ (32 768 bytes), 128 $\frac{sectores}{cluster}$ (65 536 bytes). Una vez determinado el tamaño de cluster, éste queda fijo.

Se le llama alojamiento (*allocation*) al proceso en el que los archivos son asignados a los clusters.

Todo archivo debe ser alojado en un número entero de clusters. El cluster es la unidad más pequeña de espacio que se le puede asignar a un archivo; por esta razón, a los clusters se les llaman también unidades de alojamiento (*allocation units*) o unidades de asignación.

Supongamos que tenemos un disco que utiliza clusters de 8192 bytes. Un archivo de 8000 bytes usará un cluster para alojarse, mientras que un archivo de 9000 bytes utilizará dos clusters, es decir, se le asignarán 16 384 bytes de espacio en el disco. Ésta es la razón por la que el tamaño de cluster es importante, los tamaños grandes resultan en mayor espacio de almacenamiento desperdiciado, pues es menos probable que los archivos llenen completamente un número entero de clusters. Los archivos quedan entonces representados por clusters dentro del dispositivo de almacenamiento.

El sistema operativo determina donde se encuentran los clusters que forman a cada archivo guardado en el volumen de almacenamiento, a través del **directorio** y de la tabla de asignación, llamada *File Allocation Table* (**FAT**), del sistema de archivos. El sistema de archivos FAT toma su nombre de esta tabla.

El directorio es una región dentro del dispositivo de almacenamiento en la que se registran características que identifican a cada archivo guardado, como su nombre, extensión, tamaño, cluster de inicio.

En la FAT se guarda información sobre los clusters que se asignan a cada archivo; es decir, la tabla guarda información sobre qué partes del espacio de almacenamiento del dispositivo contienen a qué archivos.

El sistema operativo determina con la información del directorio el primer cluster de cada archivo y recurre a la FAT para conocer los siguientes clusters que lo forman. Los clusters se identifican por un número, ese número le da al cluster una entrada en la FAT.

Cada cluster de cada archivo tiene una entrada en la FAT que describe como se 'arma' ese archivo, la entrada del cluster se refiere a la posición que ocupa ese cluster dentro de la FAT. El valor de la entrada es el contenido de esa posición dentro de la tabla, que es el número de cluster

del siguiente cluster usado por el archivo, es decir, el valor de la entrada indica el número de cluster siguiente del archivo. Esta información la usa el sistema operativo para 'encadenar' clusters y formar archivos.

En la figura 2.16 se muestra la asignación de clusters a dos archivos, MiDoc1.doc de cuatro clusters y Doc.txt de un sólo cluster.

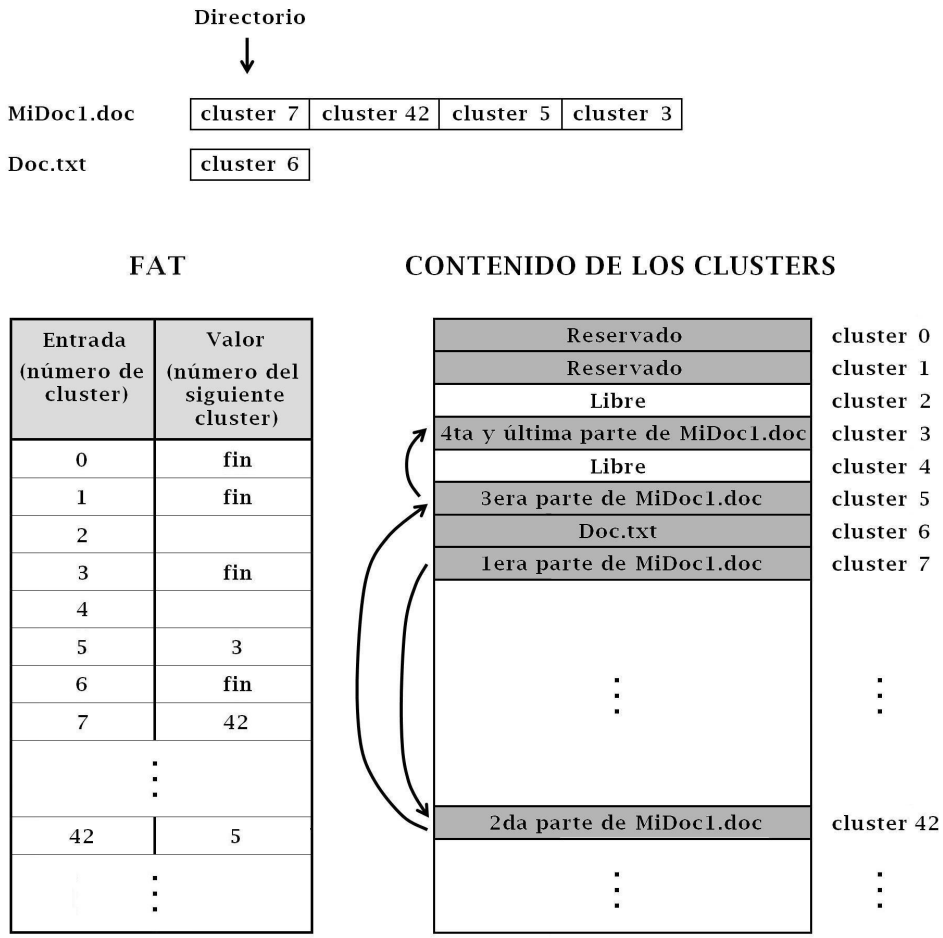


Fig. 2.16. Ejemplo de alojamiento de dos archivos.

En el ejemplo los archivos no están formados por clusters consecutivos pero la asiganción podría cambiarse para que la secuencia de clusters lo fuera. Como lo indica la figura, el número del primer cluster de cada archivo se obtiene del directorio.

Tipos de FAT El tamaño de la FAT limita la cantidad de clusters que un dispositivo de almacenamiento puede tener. Existen tres versiones de esta tabla y con ellas se definen los tres sistemas de archivos de la familia FAT:

- **FAT12**

  Es el tipo de FAT más antiguo. La longitud del valor de entrada (tamaño de la entrada) en la tabla es de 12 bits, es decir, utiliza 12 bits para representar los números de cluster. Un volumen formateado usando el sistema de archivos FAT12 puede tener y por lo tanto direccionar, un máximo de 4086 clusters. El sistema de archivos reserva diez clusters adicionales para uso propio. El FAT12 es apropiado para dispositivos que permiten volumenes pequeños de almacenamiento (menores a 16[MB]).

- **FAT16**

  Es el FAT usado por la mayoría de los sistemas que recientemente se han vuelto antiguos. El tamaño de la entrada en la tabla es de 16 bits, es decir, se representa cada número de cluster por 16 bits. Un volumen que usa FAT16 soporta un máximo de 65526 clusters. El sistema de archivos reserva diez clusters adicionales para uso propio. El FAT16 se usa en dispositivos con necesidades mayores de almacenamiento que abarcan el rango de 16[MB] a 2[GB].

- **FAT32**

  Es el tipo más reciente de la familia FAT. Es compatible con las versiones más recientes de Windows. Este sistema de archivos utiliza números de cluster de 28 bits (no de 32), porque 4 de los 32 bits son reservados. El FAT32 puede, en teoría, manejar volumenes de más de 268 millones de clusters.

En la tabla 2.4 se hace un resumen de las características principales de la familia de archivos FAT.

Atributo	FAT12	FAT16	FAT32
Almacenamiento	Muy pequeño	Moderado	Grande
Tamaño de entrada	12 bits	16 bits	28 bits
Máx. núm. clusters	4086	65526	268435456
Tamaño de cluster	0.5 [kB] – 4 [kB]	2 [kB] – 32 [kB]	4 [kB] – 32 [kB]
Máx. almacenamiento	→ 16[MB]	→ 2[GB]	→ 2[TB]

Tabla 2.4. Sistemas de archivos FAT.

El remplazo del FAT e incompatible con este sistema de archivos es el exFAT.

- **exFAT**

  El sistema de archivos exFAT (*Extended File Allocation Table*) lo introdujo Microsoft a finales del 2006. Está dirigido especialmente a dispositivos de almacenamiento que usan tecnología flash y se pretende reemplace el sistema de archivos FAT que actualmente se usa en estos dispositivos. El volumen de almacenamiento máximo es de 512[TB]. Por el momento no se han hecho disponibles las especificaciones oficiales y por lo mismo la mayoría de los equipos electrónicos no lo soportan, aunque esto podría cambiar con la llegada de las tarjetas SDXC.

Las tarjetas de memoria SD CARD, utilizadas en el desarrollo de este trabajo, soportan los sistemas de archivos FAT16 y FAT32. Estos sistemas de archivos son los que se implementan.

En adelante, en el texto se hará referencia general a ambos, como sistema de archivos FAT, o simplemente FAT[3]. Los nombres explícitos FAT16 y FAT32 se utilizarán cuando se hable en específico de ellos.

Mapas de memoria El volumen de almacenamiento de un dispositivo formateado con el sistema de archivos FAT se compone por cuatro regiones básicas:

[3] No deberá haber confusión entre *el* FAT y *la* FAT. El primero se refiere al sistema de archivos FAT, mientras que el segundo a la tabla de asignación del sistema de archivos FAT.

R0 : Región reservada
R1 : Región FAT
R2 : Región del directorio raíz (no existe para FAT32)
R3 : Región de datos y directorio

El primer sector de la SD CARD es el *Master Boot Record* (MBR), llamado también registro de arranque maestro. El MBR contiene información sobre las diferentes divisiones lógicas de la tarjeta, las particiones.

Las memorias SD CARD tienen una partición activa, es decir, todo el espacio de almacenamiento de la tarjeta está dentro de una sola partición. Cualquier dirección es relativa a esta partición.

La tarjeta tiene entonces dos inicios o 'dos ceros': **el CERO de la tarjeta**, que inicia en el MBR, y **el cero lógico**, que empieza con el inicio de la partición, que es el inicio del sistema de archivos.

En la figura 2.17 se esquematiza la estructura de la tarjeta de memoria bajo los sistemas de archivos FAT16 y FAT32.

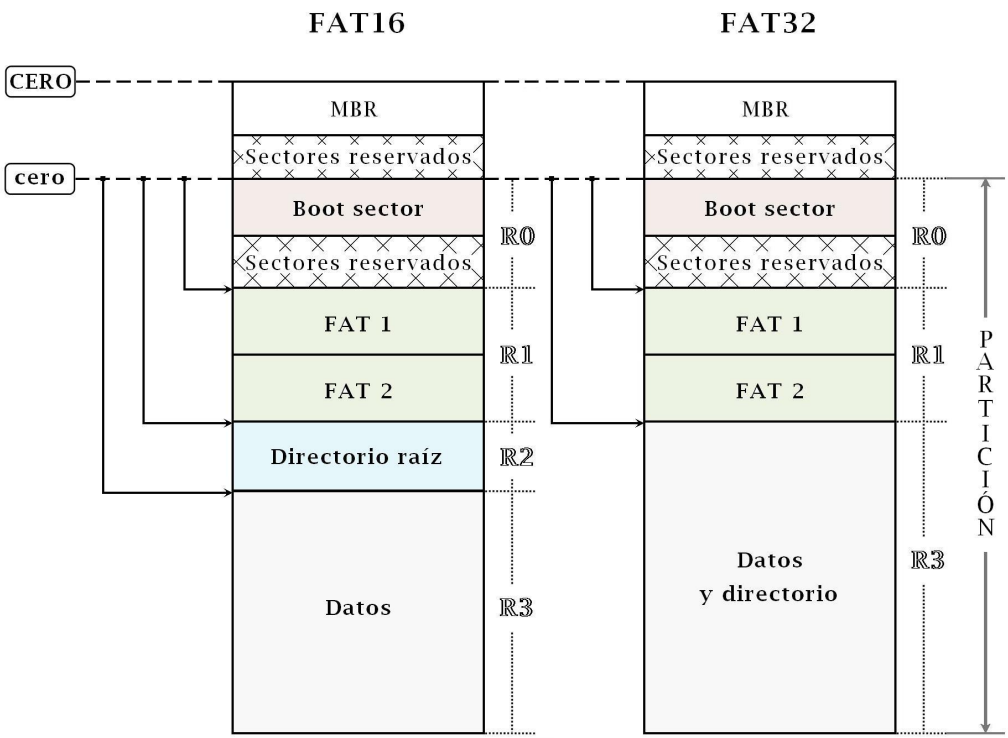


Fig. 2.17. Mapas de memoria.

El **boot sector**, o sector de arranque, es el primer sector de la partición y contiene información básica sobre el sistema de archivos. Esta información la proporciona el sistema operativo cuando se formatea la tarjeta.

La **región FAT** es el mapa de la tarjeta, indica como están distribuídos los clusters en la región de datos. Por lo general hay dos copias de la FAT en esta región, para proveer redundancia en caso de corrupción de una de ellas.

El **directorio raíz** sólo aplica para el sistema de archivos FAT16, esta región sigue a la región FAT. En el sistema de archivos FAT32 el **directorio** no tiene una zona especial, si no que es una cadena de clusters como cualquier archivo y se puede colocar en cualquier lugar dentro de la región de datos.

El directorio guarda información de los datos que almacenamos en la tarjeta; los datos son archivos y subdirectorios que el usuario crea cuando salva su trabajo. La información de estos datos se guarda en *records* o en entradas en el directorio. Ver la figura 2.18.

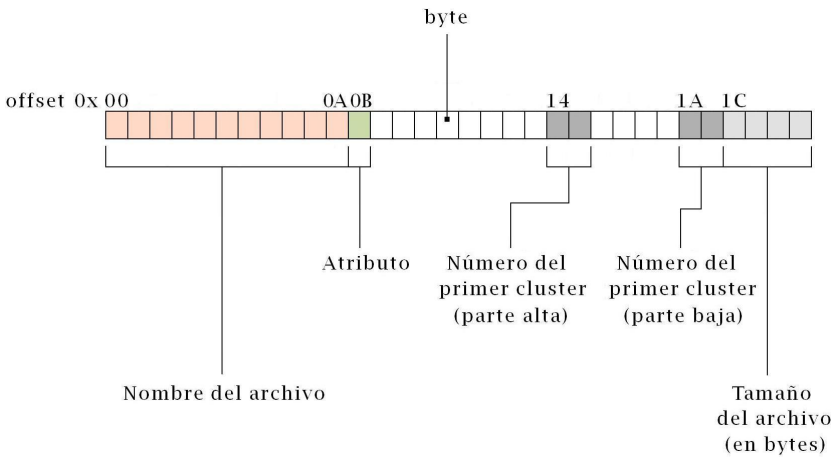


Fig. 2.18. Estructura de una entrada del directorio.

Cada sector del directorio contiene 16 entradas de 32 bytes cada una. Esto aplica para ambos sistemas FAT16 y FAT32.

En términos simples, el directorio sirve para conocer el primer número de cluster del archivo, esto es, el directorio nos dice donde empieza el archivo.

Las regiones R0, R1 y R2 forman el área del sistema. Como se ve, el sistema de archivos reserva un espacio de la partición para guardar datos administrativos del sistema. Es por esto que cuando se formatea a la SD CARD, y en general, a cualquier dispositivo de almacenamiento con este sistema de archivos, el espacio disponible para guardar nuestros datos no es el total de la capacidad de almacenamiento del dispositivo.

La **región de datos** es el espacio de la memoria disponible al usuario, y es el resto del espacio de la partición. En ella se almacenan los archivos y también (para el caso del FAT32) el directorio.

2.3.2. Otras alternativas

Por sus características particulares, las memorias flash van acompañadas de un controlador que realiza funciones de detección de errores y *wear levelling*[4], o de un sistema de archivos específico para esta tecnología, que provee, adicional a estas funciones, un uso adecuado de la tecnología flash.

En la práctica muchos sistemas que integran *memory cards* tienen controladores que llevan a cabo las funciones mencionadas, e implementan sistemas de archivos 'convencionales', como el sistema de archivos FAT.

Los sistemas de archivos convencionales fueron desarollados para usarlos en unidades de almacenamiento antes del estado sólido, como en los discos duros. Estos sistemas convencionales, al ser implementados en las nuevas tecnologías, como en las tarjetas de memoria, emulan una unidad de disco, y por lo mismo, no explotan convenientemente las características de la tecnología flash; sin embargo, permiten compatibilidad universal con cualquier computadora, y proporcionan una opción de bajo costo para administrar el contenido almacenado en la memoria.

Las *Flash Translation Layers* (FTL) son *drivers* que junto con el sistema operativo emulan, a partir de una memoria flash, una unidad de disco común, y la proveen de *wear levelling*; los sectores del dispositivo

[4] Se le llama así a las técnicas que prolongan el tiempo de vida útil de varios tipos de memorias, como las memorias flash, ordenando la información contenida para lograr un desgaste homogéneo en toda la memoria.

emulado se almacenan en localidades que varían y una *Translation Layer* se usa para seguir la pista de la posición actual de cada sector.

Los sistemas de archivos flash (*Flash File Systems*) están especialmente diseñados para almacenar archivos en dispositivos que utilicen memorias flash. El exFAT es un ejemplo de este tipo de sistemas de archivos.

2.4. Comunicaciones serie

En el proyecto se utilizan dos tipos de comunicación serie. Una principal y otra de apoyo. La principal se aplica con la memoria SD CARD, y la comunicación es a través del protocolo SPI. Esta comunicación es suficiente para satisfacer el objetivo del proyecto.

La comunicación de apoyo se realiza con la computadora y se usa como una vía de epuración para el sistema desarrollado. Se realiza a través del protocolo RS-232 y se describe brevemente en la sección 2.4.2.

2.4.1. Protocolo serie síncrono SPI

La comunicación SPI se utiliza para mover datos de forma simple y rápida de un dispositivo a otro. El protocolo permite que un dispositivo 'maestro' inicie comunicación con un dispositivo 'esclavo' y que haya intercambio de datos entre ellos.

El SPI comúnmente está implementado en la mayoría de los microcontroladores como un módulo de *hardware*, como en el microcontrolador utilizado en este proyecto.

Las características principales de la comunicación SPI y cómo la lleva a cabo el microcontrolador, se mencionan a continuación.

Características del protocolo SPI

- La interfaz SPI usa cuatro señales para intercambiar datos de forma serie con otro dispositivo.
  - *Serial Data In* (SDI). Esta línea lleva los datos de entrada al dipositivo.

  - *Serial Data Out* (SDO). Esta línea lleva los datos de salida del dispositivo.
  - *Serial Clock* (SCK). Es la señal de reloj del protocolo de comunicación. Controla el envío y la lectura de los datos y es generado por el maestro.
  - *Slave Select* ($\overline{\text{SS}}$). Cuando esta línea se pone en bajo ('0') la señal es activa y el esclavo atiende al reloj y a las señales de datos. Esta línea controla la conexión y desconexión del esclavo.

- Es un protocolo síncrono.
  - Los datos se envían por el dispositivo maestro junto con la señal de reloj, SCK. Este último lo proporciona el maestro para proveer sincronización (coordinación en tiempo).
  - La señal de reloj controla cuándo los datos cambian y cuándo éstos son válidos para leer.
  - La frecuencia del reloj puede variar sin corromper los datos. La razón de cambio de los datos variará conforme a los cambios en los pulsos del reloj.
- Es un protocolo maestro–esclavo.
  - El maestro controla la señal de reloj.
  - No existe transferencia de datos a menos que la señal de reloj esté presente.
  - El esclavo no pueden manipular la señal de reloj. La configuración del dispositivo determina como responde éste cuando recibe el reloj.
- Es un protocolo de intercambio de datos.
  - Los intercambios se controlan por la línea de reloj.
  - Los datos siempre se intercambian entre el maestro y el esclavo. En la comunicación SPI ningún dispositivo es sólo un transmisor o sólo un receptor. Cada uno tiene dos líneas de datos: una para entrada (SDI) y otra para salida (SDO).

- Mientras el maestro envía datos al esclavo, al mismo tiempo recibe de éste datos de entrada. De la misma forma, mientras el esclavo envía datos al maestro, recibe de éste datos de entrada.

- Los datos de entrada deben ser leídos por el dispositivo antes de intentar una nueva transmisión. Si no se leen serán reescritos por los nuevos datos de entrada recibidos, y el módulo de SPI se deshabilitará como consecuencia. Por ello siempre se deben leer los datos de entrada después de cada transferencia, aún cuando éstos no se utilicen.

- Por regla general, la señal de control $\overline{\text{SS}}$ debe usarse siempre. Esta señal le indica al esclavo que el maestro desea iniciar un intercambio de datos con él.

- La transmisión de datos se configura a través del modo SPI que controla el momento en el que se transmiten los datos.

- Los datos se envían en cada flanco de subida o bajada de la señal SCK según el modo SPI, mientras que en el flanco opuesto al configurado se leen. Si el valor de los datos cambia, lo hace con los flancos de subida o bajada de la transmisión; esta es la forma en la que los datos se sincronizan con la señal de reloj. Por ejemplo, en la figura 2.19 el valor de los datos cambia en cada flanco de bajada y la lectura se hace en cada flanco de subida de la señal SCK.

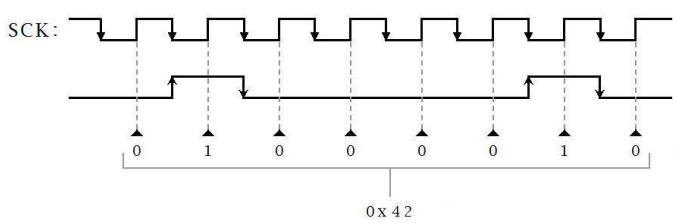


Fig. 2.19. Reloj SPI.

Transferencia de datos

- En la comunicación SPI se crea un lazo de datos entre el maestro y el esclavo. Los datos que entran y salen del maestro los llevan las líneas SDI y SDO de este dispositivo respectivamente. De forma similar ocurre para el dispositivo esclavo.

- La señal de reloj es generada por el maestro y con su envío inicia la comunicación SPI, previa activación del esclavo a través de la línea $\overline{\text{SS}}$. El esclavo espera estas señales y las usa cuando procesa los datos de la comunicación.

- Por cada pulso de reloj un bit es enviado y cada bloque de datos es de ocho bits. El maestro transmite un bloque de datos al esclavo, los que viajan de la terminal SDO del maestro a la terminal SDI del esclavo. Al mismo tiempo, el esclavo manda un bloque de datos al maestro, de la terminal SDO del esclavo a la terminal SDI del maestro. Cuando termina la transferencia los boques de datos respectivos se habrán enviado y ambos dispositivos deberán leer el byte que recibieron de la transferencia. El microcontrolador lo hará a través del registro SSPBUF. El proceso se repite tantas veces como bytes se transmitan durante la comunicación.

- La línea $\overline{\text{SS}}$ es una terminal cualquiera de un puerto del microcontrolador (maestro) y debe ser controlada por el *firmware* implementado en éste. Cuando se termina la transferencia de datos esta línea se pone en alto ('1') y la comunicación SPI termina.

  La figura 2.20 esquematiza la comunicación SPI.

2.4.2. Protocolo serie asíncrono RS-232

Comunicar de forma serie implica el envío y recepción de pulsos digitales a una velocidad de transmisión establecida entre los dispositivos que se comunican.

En la comunicación RS-232 el transmisor envía los datos a través de pulsos, a una razón acordada previamente con el receptor. El receptor por su parte atiende estos pulsos a la misma razón. No existe un reloj común entre los dispositivos, en vez de esto, cada uno tiene su propio

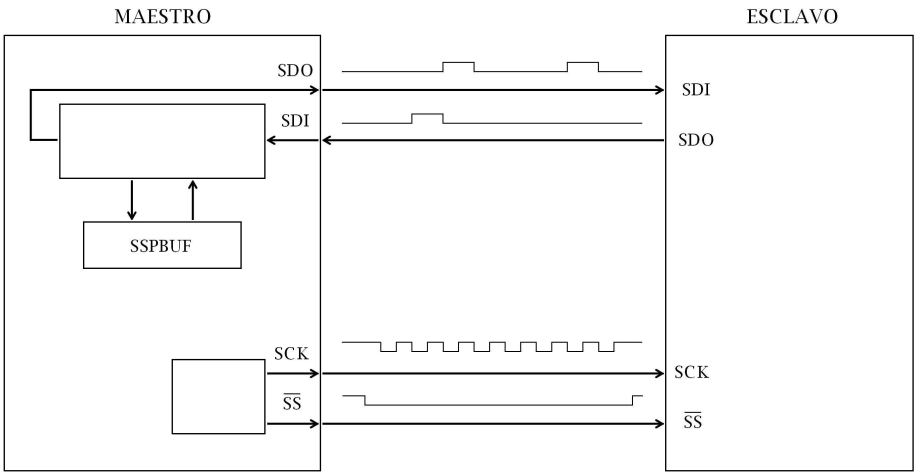


Fig. 2.20. Comunicación SPI.

reloj, y acuerdan una razón para la variación de los datos, para la cual ajustan sus relojes.

Los dipositivos se conectan por tres líneas:

- Tierra común (GND). Ambos comparten el punto de referencia para medir los cambios de tensión.
- Transmisión (TX). El transmisor en turno envía los datos al receptor a través de esta línea.
- Recepción (RX). El receptor en turno recibe los datos del transmisor a través de esta línea.

Comúnmente los datos se envían en grupos de ocho bits. La razón, o tasa de transmisión de datos, se expresa en bits por segundo (bps). Por ejemplo, a una tasa de 38400[*bps*], el receptor leerá continuamente la tensión en la línea en la que el transmisor le envía datos, pero sólo cada $\frac{1}{38400}$ de segundo interpretará el voltaje de la línea como un nuevo bit.

El protocolo agrega a los bits de datos un bit de inicio, bits de fin y un bit de paridad. Estos se pueden configurar.

En el ejemplo mencionado, de los 38400 bits enviados sólo una parte de ellos corresponderán a los bits de datos, el resto son usados por los bits de inicio, fin y paridad.

Cabe comentar que el protocolo RS-232 utiliza lógica invertida y niveles de tensión que pueden ser mayores a los usados por los microcontroladores, -3[V] a -12[V] para el '1' y $+3$[V] a $+12$[V] para el '0'. Debido a lo anterior, y en su caso, la implementación de la comunicación RS-232 se completa con un transceptor, necesario para la inversión y elevación del voltaje entre los dispositivos a comunicarse mediante este protocolo. Los dispositivos involucrados en la comunicación son el microcontrolador y la computadora, a través de su puerto serie, como lo muestra la figura 2.21.

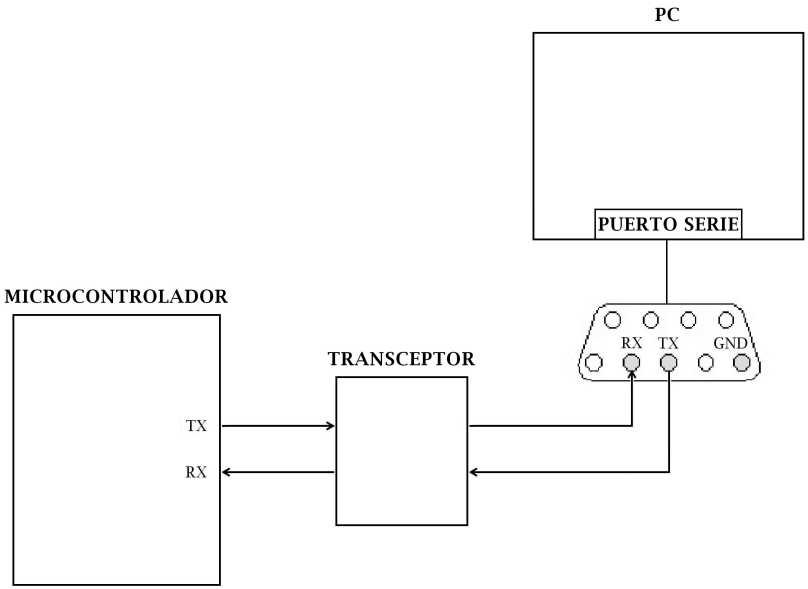


Fig. 2.21. Esquema de la comunicación RS-232.

2.5. Microcontrolador

Un microcontrolador es el núcleo del sistema de registro de datos. Las características de éste atienden a las necesidades que requiere el sistema; en primer lugar, a las especificaciones básicas de la tarjeta de memoria SD CARD, y en segundo, a la implementación del sistema de archivos.

Se estudiaron las características del sistema de registro de datos y se determinó que prácticamente cualquier microcontrolador de propósito general sirve para los fines del proyecto.

El sistema de registro de datos se desarrolló en el Laboratorio de Instrumentación del Instituto de Ingeniería de la UNAM. El laboratorio cuenta con la infraestructura necesaria para desarrollar sistemas empleando microcontroladores de dos de fabricantes, Atmel y Microchip.

Se revisaron características de los microcontroladores de ambos fabricantes y para la selección se tomaron en cuenta dos criterios, la documentación que dispone el fabricante y el costo del componente.

En la época de desarrollo del sistema de registro de datos se podían solicitar muestras de muchos de los microcontroladores de Microchip, los PIC's. Estas muestras se utilizaron para familiarizarse con esta línea de microcontroladores y la decisión para el microcontrolador del sistema se hizo más clara.

El resultado de esta experiencia y la útil documentación facilitada por Microchip determinó la selección del microcontrolador a un PIC y específicamente al PIC18LF452.

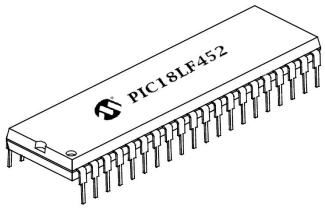


Fig. 2.22. PIC18LF452.

Durante el estudio de los PICS's se encontró conveniente escribir el código del microcontrolador en lenguaje C, y para ello se utilizó la versión estudiantil del compilador que proporciona Microchip para la familia PIC18.

Así, el núcleo del sistema de registro de datos es el microcontrolador PIC18LF452 y su *firmware* se programó usando el lenguaje de programación C.

En la tabla 2.5 se indican las características del microcontrolador seleccionado respecto a las especificaciones del sistema.

ESPECIFICACIÓN	SISTEMA	PIC18LF452
Voltaje de alimentación [V]	2.7 – 3.6	2.0 – 5.5
Comunicación serie: SPI RS-232	 Modo SPI	 Módulo MSSP Módulo USART
Memoria de programa [kB]	15	32
Memoria dato [kB]	1.2	1.5

Tabla 2.5. Características del sistema y del PIC18LF452.

En este capítulo se ha mostrado un panorama general de los conceptos y elementos necesarios para el desarrollo del proyecto. En el siguiente capítulo se presenta de manera formal el desarrollo y la implementación del sistema de registro de datos.

3

Este capítulo se divide en dos partes. En la primera parte se plantea el diseño del sistema de registro de datos y en la segunda su desarrollo. El desarrollo del sistema profundiza lo indicado en la parte de diseño y expresa consideraciones a tomar en cuenta para la implementación de la unidad de registro. Se detalla lo que se considera más importante.

3.1. Diseño del sistema

El sistema de registro de datos está formado por un módulo de almacenamiento y por una unidad de depuración. Ver figura 3.1.

El módulo de almacenamiento comprende una unidad de control y una memoria SD CARD. La unidad de depuración está conectada al módulo de almacenamiento y se utiliza como medio para desplegar bytes de interés, y con ello identificar y corregir errores en el sistema.

El sistema de registro de datos graba información en las tarjetas de memoria de forma tal que una computadora puede identificarlas a través de un lector de memorias SD CARD. Así, el usuario accede al contenido de la tarjeta a través de la computadora.

Antes de que la tarjeta de memoria pueda ser utilizada en el sistema de registro, necesita estar formateada con el sistema de archivos FAT; las memorias SD CARD vienen de fábrica formateadas con este sistema

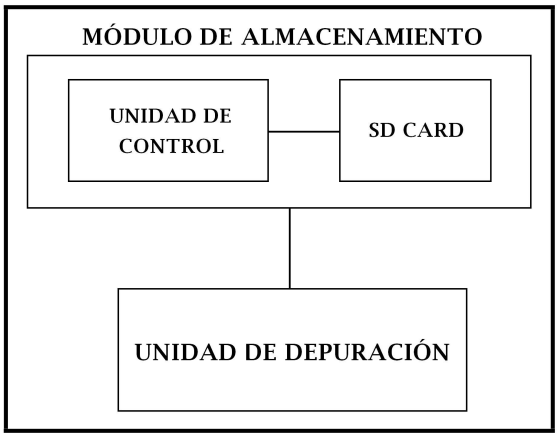


Fig. 3.1. Bloques del sistema de registro de datos.

de archivos, sin embargo, el formato se lo puede dar cualquier PC que soporte el sistema de archivos FAT.

Como se vio en el capítulo anterior, la comunicación con la memoria SD CARD es vía SPI a través de un *host*. El *host* es la unidad de control del sistema de registro.

Antes de iniciar el grabado de datos es obligatorio inicializar la tarjeta de memoria, luego, el envío y recepción de bytes de información puede tomar lugar. Este intercambio de bytes es el resultado de la comunicación entre el *host* y la tarjeta de memoria. De estos bytes, los únicos sobre los que el *host* tiene control son sobre los que envía, pues la tarjeta entrega bytes con un formato esperado pero con un valor desconocido. Se debe entonces analizar los bytes enviados por la SD CARD para conocer la forma en que responde a los comandos.

En términos simples, implementar los sistemas de archivos FAT consiste en leer de la tarjeta, y escribir en ella, bytes específicos en localidades específicas que se determinan a partir de las especificaciones de Microsoft. Los procesos de lectura, cálculo y escritura de estos bytes en la SD CARD los lleva a cabo el *host* del sistema de registro.

Los sistemas de archivos FAT se implementan en las tarjetas de memoria para que el usuario pueda ver su contenido cuando las conecta a una computadora. La implementación consiste en leer de la tarjeta bytes especiales que se encuentran en campos específicos, operarlos y del resultado se determina la estructura interna que le corresponde a cada tarjeta,

de acuerdo con el sistema de archivos con el que fue formateada. A cada tarjeta de memoria se le puede implementar un solo sistema de archivos, es decir, se le implementa el sistema de archivos FAT16 o el sistema de archivos FAT32.

Con la información que proporciona los bytes especiales, se implementan, también a través del *host*, las reglas de las especificaciones para los sistemas de archivos FAT16 y FAT32. Esta información se traslada a las memorias SD CARD, con lo que se puede registrar en ellas datos que cumplen las normas de Microsoft para estos sistemas de archivos.

El sistema de registro de datos establece dos interfaces de comunicación:

El módulo de almacenamiento determina una interfaz entre el *host* y la memoria SD CARD. Como la comunicación entre ellos se lleva a cabo vía SPI, a esta interfaz le llamamos interfaz maestro–esclavo.

La interfaz maestro–esclavo tiene dos funciones principales: comunicarse con la SD CARD para almacenar en ella datos e implementar el sistema de archivos adecuado para cada tarjeta que se conecta al sistema. Ambas funciones requieren del análisis de los bytes intercambiados entre el *host* y la tarjeta de memoria, y para ello, el sistema de registro de datos provee una vía para desplegar estos bytes en la pantalla de una PC, a través de una interfaz que se comunica de forma serie. A esta interfaz le llamamos interfaz de depuración.

La interfaz de depuración la conforma el módulo de almacenamiento y la unidad de depuración. La unidad de depuración participa activamente con el *host* del sistema para que todo byte solicitado dentro de él, tenga salida a la computadora (unidad de depuración), lo que permite analizar los movimientos entre el *host* y la SD CARD.

La figura 3.2 muestra las interfaces del sistema de registro de datos, dentro de la línea punteada está el módulo de almacenamiento (interfaz maestro–esclavo). La interfaz de depuración comunica este módulo con la PC.

El *host* lo conforma el microcontrolador PIC18LF452, que controla todos los procesos del sistema y es el núcleo de éste; al microcontrolador le llamaremos también **PIC**. El bloque SD CARD identifica a la tarjeta de memoria. El sistema soporta una sola tarjeta a la vez y puede ser una memoria SD o SDHC.

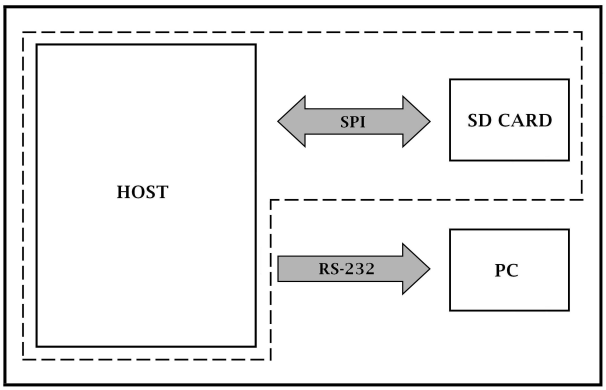


Fig. 3.2. Interfaces del sistema de registro de datos.

Como se observa en la figura 3.2, en la interfaz de depuración la comunicación con la computadora es en un solo sentido. El *host* es el que envía bytes de interés a la PC para que se muestren en el monitor a través de una terminal virtual.

3.2. Desarrollo del sistema

La actividad del sistema de registro de datos es controlada a través del *firmware* almacenado en el microcontrolador, que actúa sobre los circuitos electrónicos conectados al él y cuya conexión depende del protocolo de comunicación utilizado.

Dentro del sistema de registro de datos se tienen dos interfaces y cada una utiliza un protocolo de comunicación. En la interfaz maestro–esclavo el microcontrolador se comunica con la SD CARD vía SPI e implementa el registro de datos en la SD CARD de la forma deseada. En la interfaz de depuración el microcontrolador se comunica con la computadora a través del protocolo serie RS-232 para que la computadora despliegue en una terminal virtual los bytes que recibe del microcontrolador.

Así, cada interfaz del sistema determina una conexión eléctrica entre el microcontrolador y los circuitos electrónicos conectados a él.

3.2.1. Componentes electrónicos

El *hardware* del sistema de registro de datos, atendiendo a las interfaces que lo conforman, la interfaz maestro–esclavo y la interfaz de depuración, presenta las siguientes características:

Interfaz maestro–esclavo En la comunicación SPI el microcontrolador es el maestro y la SD CARD el esclavo.

El microcontrolador establece comunicación con la tarjeta de memoria a través su módulo interno de *hardware* que implementa el protocolo SPI (módulo MSSP) y dedica cuatro terminales eléctricas para este fin: SDI, SCK, SDO y $\overline{\text{SS}}$.

La tarjeta de memoria se comunica con el microcontrolador a través de sus terminales eléctricas para la comunicación SPI. Estas terminales se muestran en la figura 3.3 y en la tabla 3.1 se indica, para el modo SPI, la señal que le corresponde a cada terminal de la figura 3.3 junto con una breve descripción.

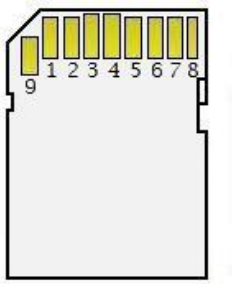


Fig. 3.3. Terminales de la memoria SD CARD.

Terminal	Modo SPI	Descripción
1	$\overline{\text{SS}}$	Recibe señal de control
2	SDI	Recibe señal de datos
3	GND	0[V]
4	V^+	$2.7 - 3.6$ [V]
5	SCK	Recibe reloj SPI
6	GND	0[V]
7	SDO	Envía señal de datos
8	sin uso	–
9	sin uso	–

Tabla 3.1. Terminales de la SD CARD en modo SPI.

La conexión de la interfaz maestro–esclavo se muestra en la figura 3.4.

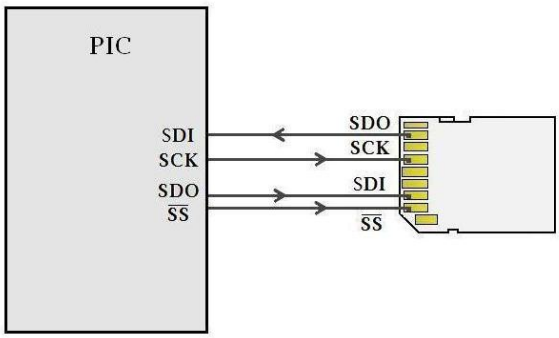


Fig. 3.4. Interfaz maestro–esclavo: PIC–SD CARD.

Interfaz de depuración La conexión de los elementos en la comunicación RS-232 se muestra en la figura 3.5.

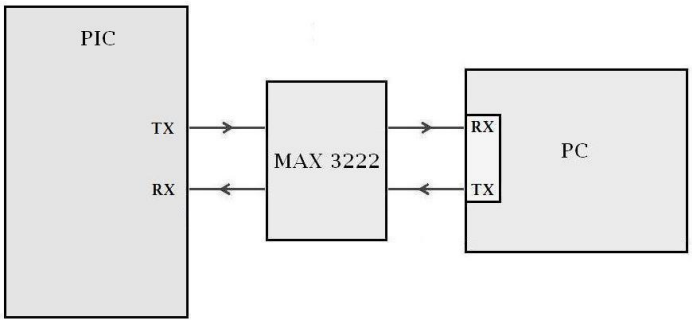


Fig. 3.5. Interfaz de depuración: PIC–PC.

El microcontrolador establece comunicación con el puerto serie de la computadora a través de su módulo interno de *hardware* (módulo USART) y dedica las terminales TX y RX para este fin. El transceptor es el MAX 3222 y conecta al microcontrolador con el puerto serie de la PC como lo muestra la figura 3.5.

3.2.2. Comentarios sobre las memorias Secure Digital

Históricamente las memorias SD CARD tienen su antecedente en las tarjetas MMC. Antes de la versión 2.00 de las especificaciones, las memorias SD CARD no eran sustancialmente distintas a las tarjetas multimedia e incluso un lector de memorias MMC podía leer también las memorias SD.

Por razones de compatibilidad con las tarjetas multimedia, se recomendaba que durante la inicialización de las memorias SD la frecuencia del reloj SPI se mantuviera por debajo de $400[k\text{H}z]$.[5]

Con la introducción de las SDHC en la versión 2.00 de las especificaciones, en el proceso de inicialización de las memorias SD CARD se identifican sólo las tarjetas de memoria Secure Digital, con lo que se hace una distinción clara entre éstas y cualquier otro tipo de memoria. Así, la recomendación de los $400[k\text{H}z]$ ha quedado obsoleta pues desde la versión 2.00 no es posible la compatibilidad con las tarjetas multimedia.

Las especificaciones de la SD CARD no establecen en el modo SPI una frecuencia de reloj recomendada para la comunicación con el *host*, sin embargo, en la secuencia de inicio se impone una restricción en la frecuencia del reloj SPI, como se verá más adelante.

Velocidades de comunicación El microcontrolador utiliza un oscilador basado en un cristal de cuarzo para generar la base de tiempo de su operación interna. Los módulos del microcontrolador dedicados a la comunicación serie utilizan también dicha base de tiempo para ajustar su velocidad de comunicación. Entonces, el valor del cristal debe ser tal que permita generar las velocidades requeridas por las comunicaciones SPI y RS-232.

El desarrollo del sistema de registro de datos inició con el desarrollo independiente de cada interfaz del sistema; se implementó la comunicación serie para cada interfaz, se simuló por *software* su comportamiento y se armó un circuito de prueba para cada interfaz.

Los módulos MSSP y USART del PIC preescalan a través del *firmware* la frecuencia de oscilación del sistema, que es generada por el cristal de cuarzo conectado al microcontrolador. Así se ajusta la velocidad en la comunicación SPI y RS-232.

Utilizando la información brindada en las especificaciones del PIC, se calculó y comprobó un valor de cristal para el que el porcentaje de error en la comunicación RS-232 fuera cero. Como la idea final era que ambas interfaces compartieran un mismo circuito, este cristal se seleccionó como el del sistema.

---

[5] **MAXIM APPLICATION NOTE 3969,** pág. 2.
APPLICATION NOTE: Secure Digital Card Interface for the MSP430, pág. 6.

El cristal del sistema es de 7.3728[MHz]. Se prescala por *firmware* para generar el reloj SPI y la tasa de transmisión de datos para la interfaz de depuración, que es de 38400[bps].

3.2.3. Desarrollo del código del microcontrolador

Como se vio en la sección 3.2.1, el *hardware* del sistema de registro de datos es simple. El peso del proyecto está en el código que se programa en el microcontrolador, el *firmware*, como se verá a lo largo de esta sección.

El *firmware* interactúa con el *hardware* del sistema para llevar a cabo el registro de datos de la forma deseada. Para lograr este objetivo, el código se estructuró en dos partes principales: la comunicación con la SD CARD y la implementación del sistema de archivos.

La comunicación con la SD CARD queda implícita en todos los procesos que el microcontrolador lleva a cabo con la tarjeta de memoria, y por ello, en el *firmware* se programaron librerías propias para implementar las reglas del protocolo para la operación de la SD CARD.

De la misma forma se dedicaron funciones y librerías propias para implementar los sistema de archivos, que programaron las reglas de Microsoft para los sistemas FAT16 y FAT32; la implementación de los sistemas de archivos consume la mayor parte del código del microcontrolador, con un poco más del 80% del total del código.

Todo el programa fue escrito en lenguaje C. Se utilizó la edición estudiantil del compilador MPLAB C18, v3.12 y el ambiente de desarrollo MPLAB IDE v8.00.

Comunicación con la SD CARD

Para que los procesos de lectura y escritura, y en general, para que cualquier operación de interés pueda tener lugar en la SD CARD, es indispensable inicializar la tarjeta. Luego de una inicialización exitosa se puede iniciar inmediatamente cualquier operación con ella a través de sus comandos.

La inicialización deja lista a la SD CARD para que atienda las órdenes del *host* y comprende dos secuencias previas al envío del primer comando

de interés; un comando de interés puede ser por ejemplo la escritura de un bloque de bytes en la tarjeta de memoria.

La inicialización o proceso de inicialización lo forman la secuencia de inicio y la secuencia de comandos de inicialización. En la secuencia de inicio se enciende la tarjeta y se espera a que inicie sus registros internos; con esto la tarjeta se prepara para responder a los comandos de la secuencia de comandos de inicialización. En la secuencia de comandos de inicialización el *host* envía un grupo de comandos siguiendo el algoritmo establecido por las especificaciones, con lo que se saca a la tarjeta del estado *idle*; terminada la secuencia, la SD CARD queda lista para atender los intereses del *host*.

Consideraciones en la comunicación Para llevar a cabo el intercambio de información entre el microcontrolador y la SD CARD se tomaron en cuenta consideraciones al implementar la comunicación. Estas consideraciones se indican a continuación.

- El argumento de cada comando depende del número del comando. Algunos comandos, como el cero, especifican su argumento como 'bits de relleno'. Otros comandos indican en su argumento bits reservados. Las especificaciones recomiendan que los bits de relleno y los bits reservados se escriban como cero, aún cuando su valor es ignorado por la tarjeta de memoria. El sistema de registro de datos se apega a ésta recomendación e implementa como ceros este tipo de bits para los comandos que los requieren.

- Conforme al protocolo SPI, las transacciones son atendidas por la tarjeta siempre que el microcontrolador active la terminal de control, $\overline{\text{SS}}$=0. Así, previo al envío de cada comando la terminal $\overline{\text{SS}}$ se manda a bajo. Por conveniencia, esta línea se mantiene activa desde el inicio del primer comando y hasta que termina de ejecutarse el código del microcontrolador.

- Inmediatamente después del envío de cada comando, el microcontrolador espera respuesta de la SD CARD. Para que ésta pueda ser transmitida por la tarjeta, el microcontrolador debe mantener la señal de reloj activa hasta recibir la respuesta, y para ello, su terminal de salida de datos la mantiene en alto, de acuerdo a la convención usada en las especificaciones para indicar el estado *idle*.

Secuencia de inicio Para encender la SD CARD se debe tomar en cuenta el procedimiento presentado en la figura 3.6, en ella se distinguen dos tiempos importantes: el de encendido y el de rampa.

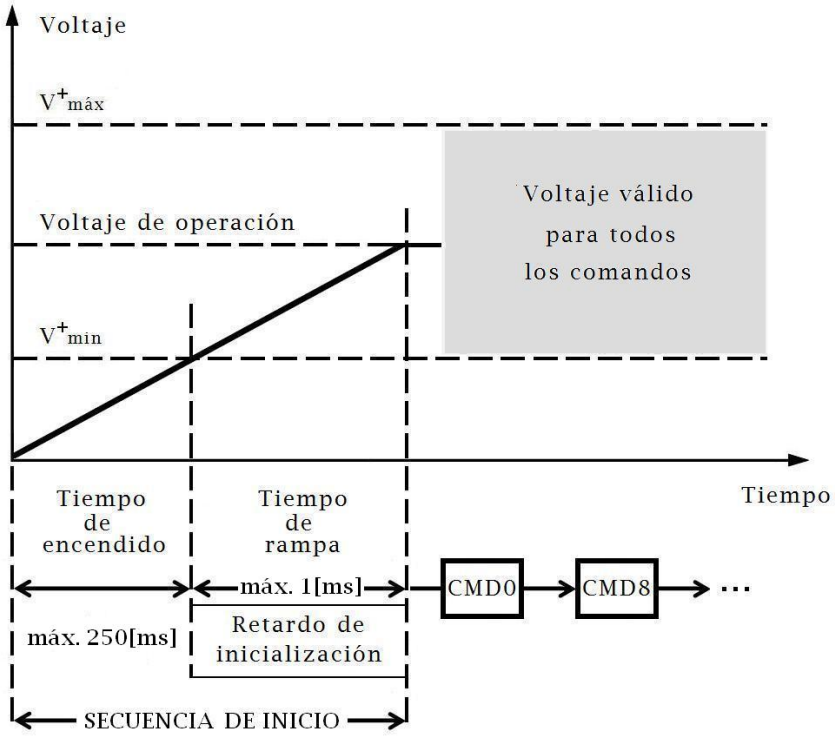


Fig. 3.6. Diagrama de encendido.

Las especificaciones definen el tiempo de encendido como el tiempo en que la tensión de alimentación se eleva de 0[V] a $V^{+}min$ y depende de los parámetros de la aplicación, a saber: de la longitud del *bus*, de las características de la fuente de alimentación y del número de tarjetas que se conectan. El tiempo de rampa se define como el tiempo en que la polarización alcanza el voltaje de operación de la tarjeta, y es el tiempo que la tarjeta debe esperar para recibir el primer comando.

Las especificaciones de la SD CARD son claras en indicar que los tiempos de encendido y de rampa no deberán ser sobrespasados si la tarjeta pretende utilizarse.

De acuerdo con las especificaciones, el *host* debe energizar a la tarjeta para que el voltaje $V^{+}min$ se alcance dentro de los primeros 250$[ms]$

(tiempo de encendido) y a continuación debe enviarle al menos 74 pulsos de reloj, manteniendo $\overline{\text{SS}}$=1, durante el tiempo de rampa (retardo de inicialización). La duración máxima del retardo de inicializacion es de $1[ms]$. Ver figura 3.6.

El microcontrolador enciende la SD CARD a través de un transistor que funciona como interruptor; con esto se comprobó empíricamente que se cumple la especificación para el tiempo de encendido de la tarjeta. Inmediatamente después de encender el transistor se envían los pulsos de reloj para el retardo de inicialización.

En la comunicación SPI el envío de pulsos de reloj implica envío de datos por el canal, los que se transmiten por bytes; entonces los pulsos del reloj SPI se envían en múltiplos de ocho. El microcontrolador enviará 80 pulsos de reloj a la tarjeta de memoria sin exceder $1[ms]$, para cumplir con el retardo de inicialización y para estar conforme con los límites marcados por las transacciones en la comunicación SPI.

La frecuencia de oscilación del sistema, F_{OSC}, se prescala por 64 para generar la frecuencia del reloj SPI, F_{SCK}.

$$F_{\text{SCK}} = \frac{F_{\text{OSC}}}{64} = \frac{7.3728}{64}[\text{MH}z] = 115.2[k\text{H}z]$$

La duración de cada pulso se determina calculando el periodo de la señal de reloj SPI,

$$T_{\text{SCK}} = \frac{1}{F_{\text{SCK}}} = \frac{1}{115.2}[ms] = 8.68[\mu s]$$

así, la duración de cada pulso SPI es $8.68[\mu s]$ y la duración del total de pulsos enviados es

$$8.68[\tfrac{\mu s}{pulso}] \times 80[pulsos] = 694.4[\mu s] < 1[ms]$$

con lo que se satisface la condición para el retardo de inicialización.

La única restricción en la velocidad de comunicación entre el microcontrolador y la SD CARD la impone la secuencia de inicio en el retardo de inicialización. Por conveniencia, la frecuencia del reloj SPI para el retardo de inicialización se mantiene durante la secuencia de comandos de inicialización, y su valor se aumenta terminado el proceso de inicialización.

La frecuencia del reloj SPI se ajusta a $115.2[k\text{H}z]$ para cumplir con la condición del retardo de inicialización. La frecuencia del reloj se aumenta a $1.8432[\text{MH}z]$ terminada la inicialización y se mantiene para el resto de la comunicación.

Secuencia de comandos de inicialización Terminada la secuencia de inicio, la tarjeta "despierta" en estado *idle* en modo SD. Para entrar al modo SPI se le da un *reset* con el comando cero.

Todo el argumento del comando cero son bits de relleno (bits en cero), y el CRC debe ser válido. El valor del CRC para este comando se proporciona en las hojas de especificaciones de las memorias SD CARD.

La figura 3.7 muestra como se compone el comando.

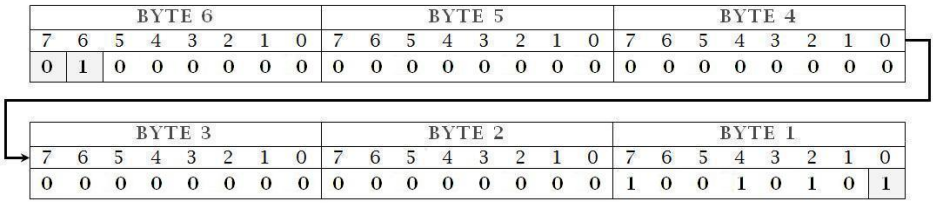


Fig. 3.7. Contenido del comando cero.

Expresado en forma hexadecimal,

0x40 0x00 0x00 0x00 0x00 0x95

y el comando se envía byte a byte empezando por el byte 6. Ver figura 3.8(a).

Enviado el comando, el microcontrolador espera por la respuesta en estado *idle*, manteniendo su terminal de salida de datos en alto (0xFF). Mientras la tarjeta no responda se mantendrá también en *idle*, como se ve en la figura 3.8(b).

Cabe comentar que cada comando tiene asociado un tipo de respuesta, y su valor deberá analizarse para conocer el estado de la comunicación, como lo muestra la figura 3.8(c) para el comando cero.

El comando cero tiene asociada la respuesta tipo R1. Si el *reset* ha sido aceptado por la tarjeta, el valor de la respuesta es 0x01 como se ve en la figura 3.8(d), y se ha entrado al modo SPI.

Una vez enviado el comando cero se procede a enviar el comando ocho (referirse a la figura 3.13). El comando ocho es nuevo en la versión 2.00 de las especificaciones. Este comando pregunta a la tarjeta si puede operar dentro del margen de voltaje que le provee el microcontrolador;

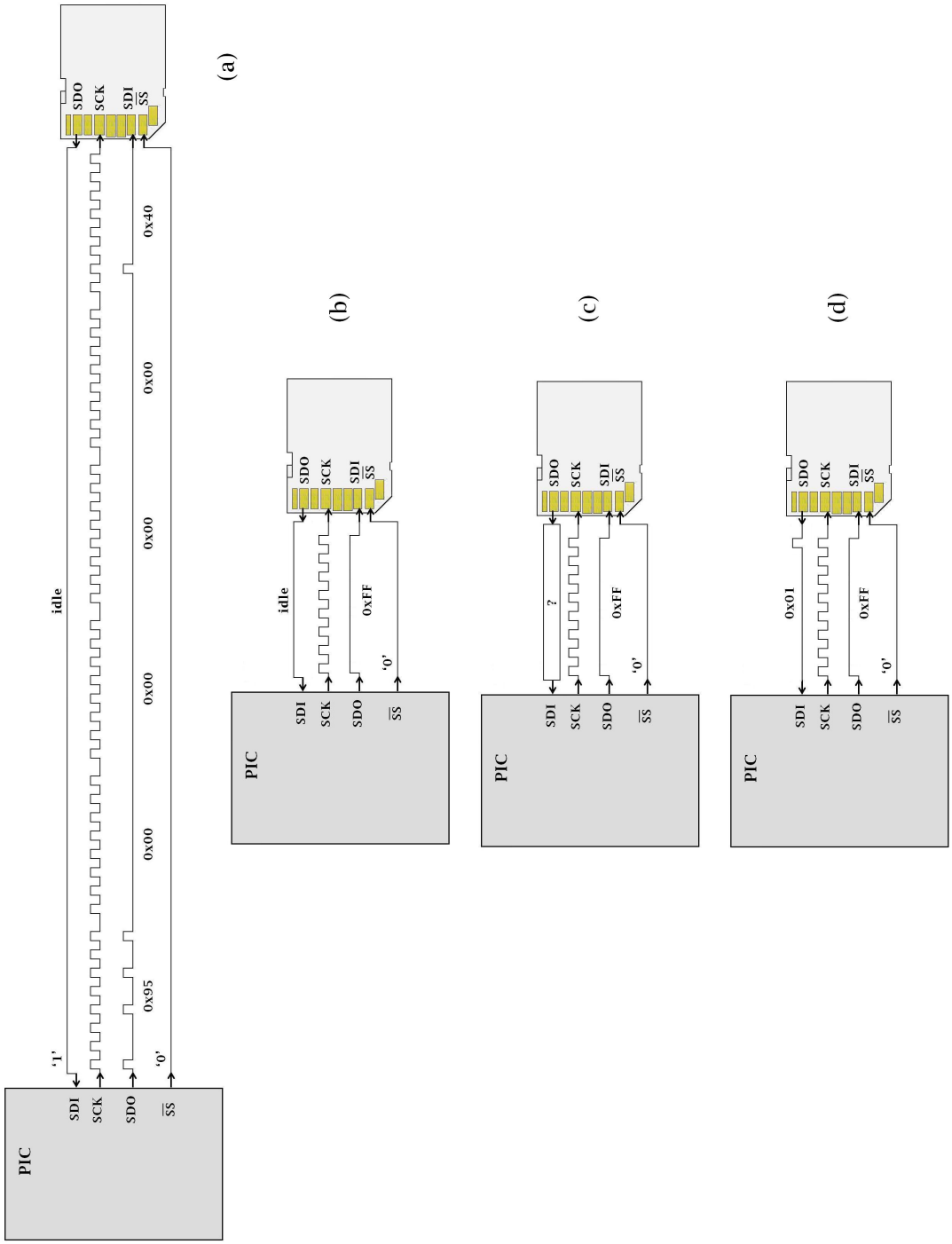


Fig. 3.8. Transacciones del comando cero.

la tarjeta considera que será alimentada con una tensión dentro de este margen. Si la tarjeta acepta el voltaje, contesta con la respuesta R7, de otro modo se mantiene en estado *idle*.

El contenido del comando es de la siguiente forma,

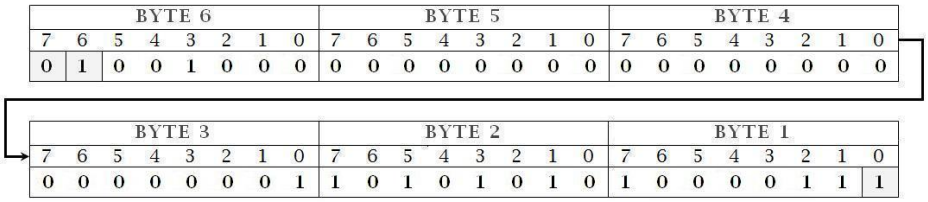


Fig. 3.9. Contenido del comando ocho.

o en hexadecimal,

0x48 0x00 0x00 0x01 0xAA 0x87

son bits reservados (bits en cero) del b7,B5 al b4,B3.

Del b3,B3 al b0,B3 se especifica el voltaje suministrado a la tarjeta de acuerdo con la tabla 3.2.

El byte 2 es el *check pattern*. Este byte puede tener cualquier valor, pero las especificaciones recomiendan que se use 0xAA. El *check pattern* se utiliza para verificar la validez en la comunicación entre el microcontrolador y la tarjeta de memoria.

Y del b7,B1 al b1,B1 se escribe el CRC7.

VOLTAJE SUMINISTRADO	DEFINICIÓN DEL VALOR
0000	No definido
0001	2.7 – 3.6 [V]
0010	Reservado para bajo voltaje
0100	Reservado
1000	Reservado
Otros	No definido

Tabla 3.2. Definición del voltaje suministrado.

Si la tarjeta soporta el comando ocho y puede operar con el voltaje suministrado, la respuesta R7 regresa como aceptados el voltaje de suministro y el *check pattern* enviados en el argumento del comando, como se muestra en la figura 3.10. Como se observa en la misma figura, la versión del comando ocho es cero.

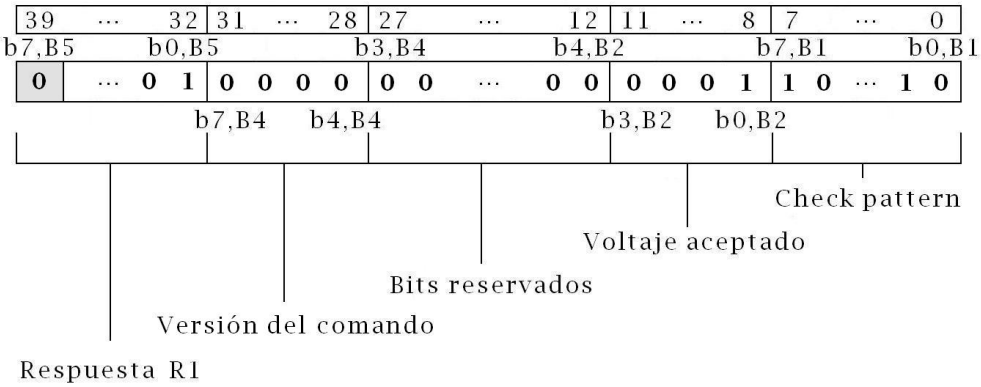


Fig. 3.10. Respuesta al comando ocho.

La respuesta en hexadecimal es,

```
0x01 0x00 0x00 0x01 0xAA
```

Si la tarjeta responde con comando ilegal, enviará sólo el primer byte de R7 con el bit dos encendido, como se mencionó en el capítulo de generalidades (referirse a la figura 2.8). Luego la tarjeta no soporta el comando ocho por ser de una versión anterior a la versión 2.00 de las especificaciones.

Si en la respuesta el campo de voltaje aceptado tiene el mismo valor que el enviado en los bits 3 al 0 del byte 3 del argumento del comando, la tarjeta puede operar con la tensión que se le suministra. Por el contrario, un valor de 0x0000 significa que la tarjeta no puede operar con la tensión suministrada. De la misma forma, si el campo de *check pattern* en la respuesta no tiene el mismo valor que el *check pattern* del argumento del comando, la comunicación con el comando ocho no es válida. En este caso se recomienda repetir su envío.

Las especificaciones establecen que en el modo SPI el CRC no es obligatorio, y de inicio está desactivado. Esto resulta conveniente al no tener

que calcular el CRC para cada comando, simplificando la comunicación, sin embargo, el comando ocho es un caso especial.

Una vez dentro del modo SPI los bits de CRC de cada comando son ignorados por la tarjeta, con excepción de los bits del comando ocho. La tarjeta siempre verifica que este comando tenga un CRC válido y por lo tanto debe calcularse.

En pruebas realizadas a varias memorias SD CARD de distintas capacidades de almacenamiento y de distintos fabricantes, se descubrió que no todas necesitaron un valor de CRC válido para aceptar el comando ocho. Por ello se notó la conveniencia de enviar este comando con un CRC válido.

El algoritmo para generar los bits de CRC7 se implementó, pero sólo al comando ocho se le adjuntó su valor calculado de CRC. Para el resto de los comandos los bits de CRC se envían como unos. En el apartado "Código de detección de error" se completa el tema del CRC.

Una vez enviado el comando ocho y obtenida su respuesta, se envía el comando 58 para obtener el margen de voltaje que soporta la tarjeta (ver figuras 3.13, 3.14 y 3.15). El formato correspondiente a este comando, en hexadecimal es:

0x7A 0x00 0x00 0x00 0x00 0xFF

todo el argumento son bits de relleno y la respuesta asociada al comando es tipo R3.

El comando lee el *Operation Conditions Register* (OCR), que es un registro de 32 bits dentro de la tarjeta de memoria. Los primeros 30 bits forman la ventana de voltajes de la tarjeta de memoria, y los últimos dos son bits de estatus. Ver la tabla 3.3.

La tarjeta envía del b7,B4 al b0,B1 de la respuesta R3, el registro OCR. El bit 7 del byte 4 de R3 corresponde al bit 31 del registro y el bit 0 del byte 1 de R3 corresponde al bit 0 del registro. La tarjeta indica los margenes de voltaje, de la ventana de voltajes, que no soporta con '0' en la respuesta R3.

El bit 31 del registro se pone en '1' cuando el proceso de inicialización ha terminado, mientras se mantiene en *busy* ('0'). El bit 30 se pone en

NÚMERO DE BIT	DEFINICIÓN
0 – 6	Reservado
7	Reservado para margen de bajo voltaje
8 – 14	Reservado
15	2.7 – 2.8
16	2.8 – 2.9
17	2.9 – 3.0
18	3.0 – 3.1
19	3.1 – 3.2
20	3.2 – 3.3
21	3.3 – 3.4
22	3.4 – 3.5
23	3.5 – 3.6
24 – 29	Reservado
30	*Card Capacity Status (CCS)*
31	*Card power up status bit*

Tabla 3.3. Registro OCR.

'1' si la tarjeta es SDHC y en '0' si es SD. Estos valores se leen, respectivamente, del bit 7 del byte 4 y del bit 6 del byte 4 de la respuesta R3.

El bit 30 del OCR será válido cuando el bit 31 esté en alto, es decir, cuando la inicialización se haya completado.

Si el comando ocho es aceptado por la tarjeta el bit 30 se enciende (CCS=1), indicando que se trata de una memoria SDHC. En este punto de la inicialización es de interés el margen de voltaje de la tarjeta, que es lo que deberá verificarse. Si el microcontrolador no acepta como compatible el margen de voltaje enviado por la tarjeta, no deberá proceder con la inicialización.

En la figura 3.11 se muestra la respuesta del comando 58 para una memoria SDHC.

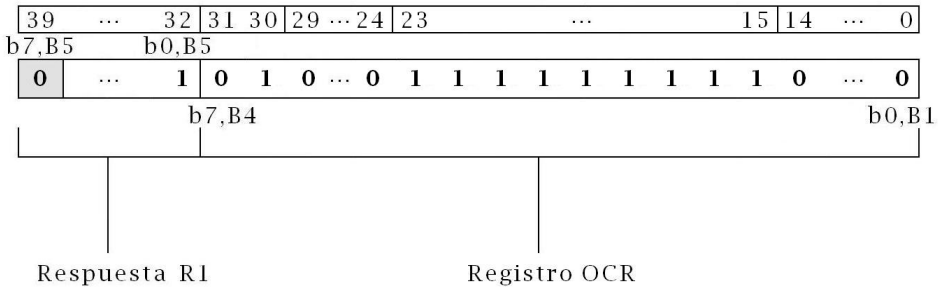


Fig. 3.11. Respuesta del comando 58.

El proceso de inicialización sigue con el envío del comando de aplicación específica 41, previo envío del comando 55.

El argumento del comando 55 son bits de relleno,

```
0x77 0x00 0x00 0x00 0x00 0xFF
```

y su respuesta es de tipo R1 con valor 0x01 como muestra la figura 3.12.

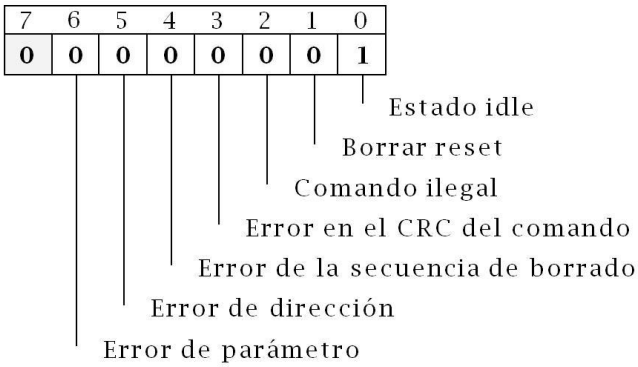


Fig. 3.12. Respuesta del comando 55.

El comando de aplicación específica 41 se utiliza para verificar si la inicialización de la tarjeta se ha completado. El bit 6 del byte 5 del comando corresponde al bit de *High Capacity Support* (HCS).

Con el bit HCS el microcontrolador indica si soporta o no *high capacity*. Si HCS=1 el microcontrolador soporta memorias SDHC, y si HCS=0 sólo soporta memorias SD.

En el argumento del comando de aplicación específica 41 todos los bits son reservados con excepción del bit HCS,

0x69 0x40 0x00 0x00 0x00 0xFF → HCS=1

0x69 0x00 0x00 0x00 0x00 0xFF → HCS=0

y la respuesta es tipo R1.

El comando se envía tantas veces hasta que el bit de estado *idle* en la respuesta R1 se ponga en '0.' Cuando R1=0x00 la secuencia de comandos de inicialización ha terminado.

El envío del comando ocho antes del comando de aplicación específica 41 es obligatorio.

Al recibir el comando ocho la tarjeta expande las funciones del comando 58 y del comando 41 de aplicación específica; CCS en la respuesta del comando 58 y HCS en el argumento del comando 41. El bit HCS lo ignoran las tarjetas que no aceptaron el comando ocho; las memorias SD ignoran este bit.

Completada la inicialización el microcontrolador debe verificar el bit CCS en la respuesta del comando 58, con lo que termina el proceso de inicialización.

Las figuras 3.13, 3.14 y 3.15 esquematizan el algoritmo de inicialización descrito.

Las leyendas Versión 1.X y Versión 2.00 o posterior, se refieren a la versión de las especificaciones, que es también la versión de la tarjeta de memoria.

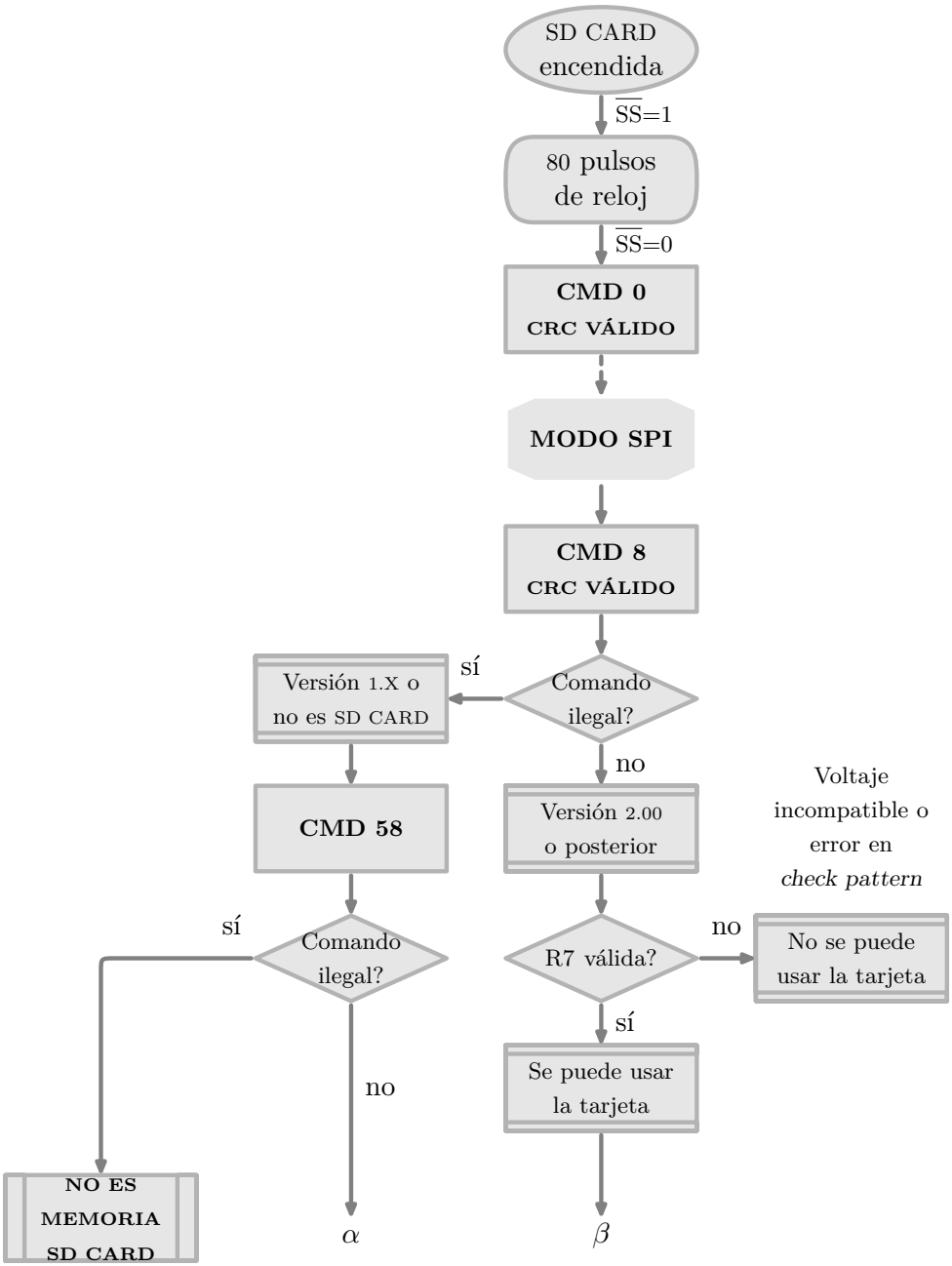


Fig. 3.13. Algoritmo de inicialización (1 de 3).

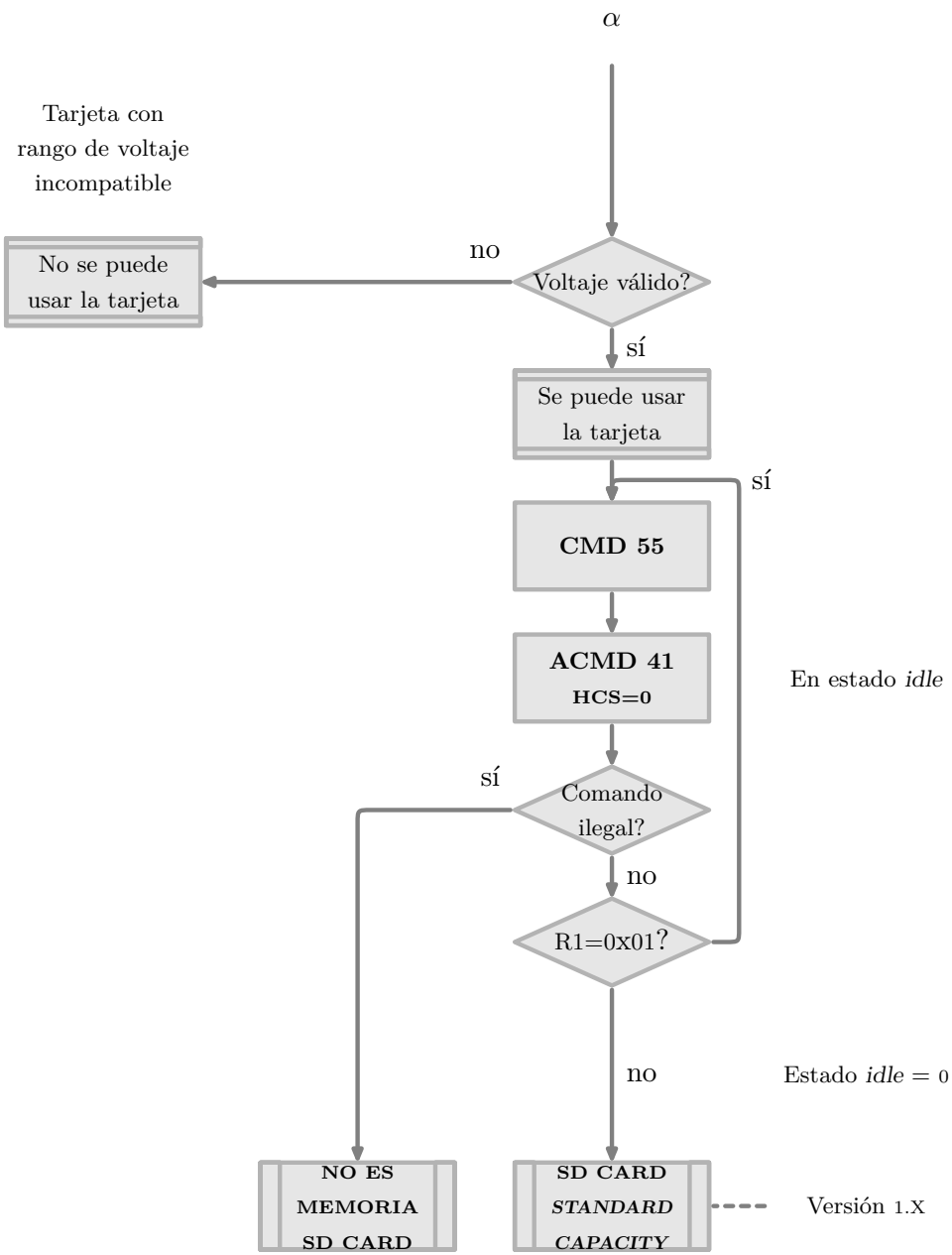


Fig. 3.14. Algoritmo de inicialización (2 de 3).

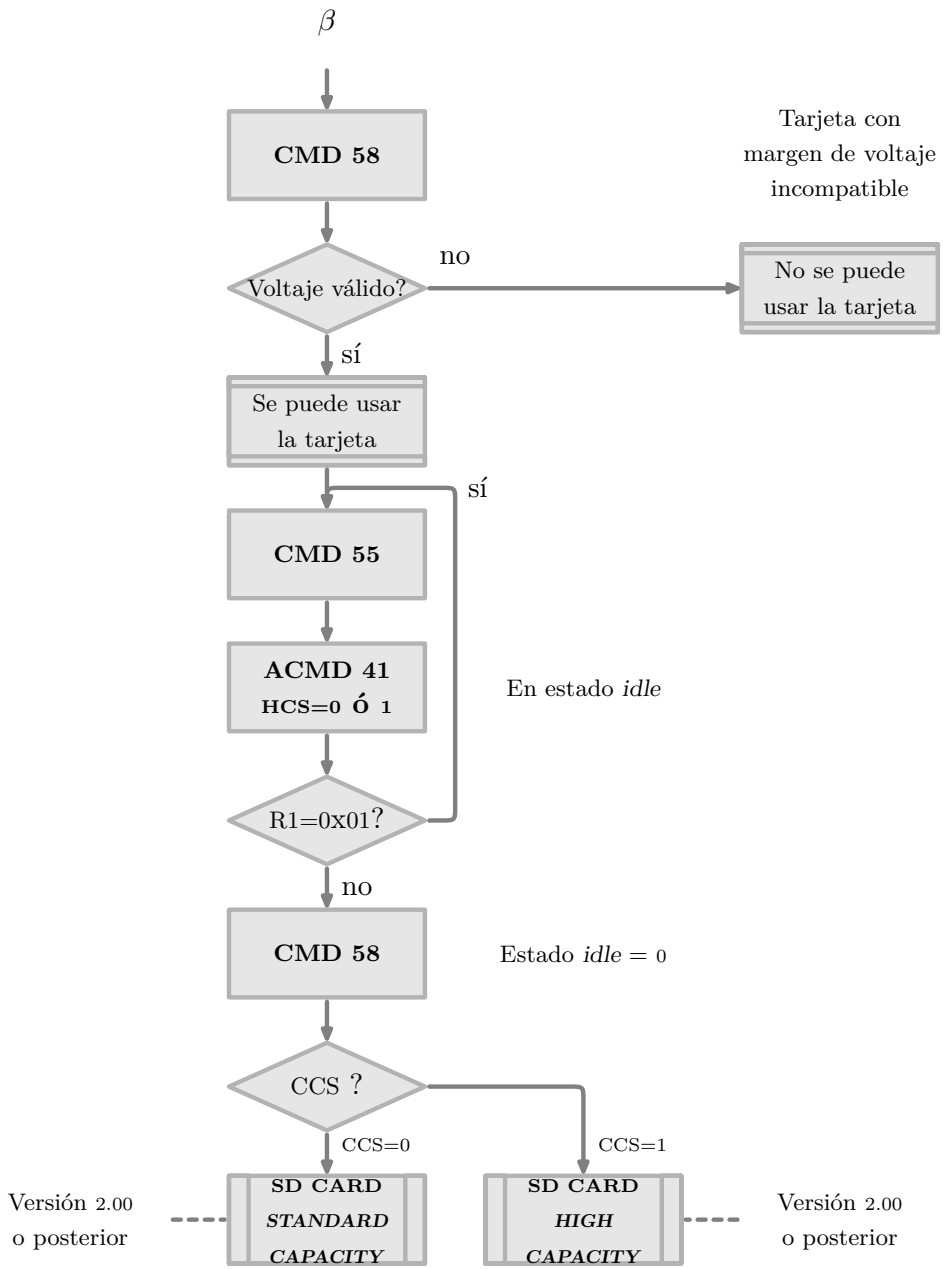


Fig. 3.15. Algoritmo de inicialización (3 de 3).

Lectura y escritura de datos La tarjeta de memoria tiene comandos específicos para la escritura, lectura y borrado de datos; estas acciones se realizan por bloques de bytes.

En el sistema se utilizan comandos para la lectura de un sólo bloque de datos y para la escritura de uno y múltiples bloques. En el siguiente apartado, "Formato de los bloques de datos", se trata sobre los bloques de bytes y se analizan los comandos involucrados en la transferencia de datos.

A continuación se explican los procesos de lectura y escritura.

Lectura de datos

El comando 17 lee un sólo bloque de datos del tamaño especificado por el comando 16.

Si el comando de lectura es válido, la tarjeta envía la respuesta del comando 17 (respuesta tipo R1) y a continuación el *Start Block Token* (SB) seguido por el bloque de bytes de datos solicitado.

El SB marca el inicio de los bytes de datos, su valor es 0xFE. Al final del bloque se envían los dos bytes correspondientes de CRC16.

Terminado el envío el microcontrolador puede transmitir el siguiente comando. Ver la figura 3.16.

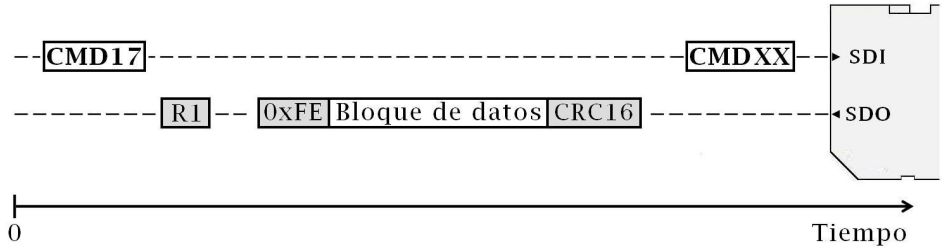


Fig. 3.16. Operación de lectura de un bloque de datos.

En caso de error en el regreso de los datos, la tarjeta no los transmite y por tanto manda un *token* de error, como lo muestra la figura 3.17.

El *token* de error le indica al microcontrolador que la tarjeta no puede proveerle el bloque de datos solicitado; la causa del error se determina analizando este *token* (ver figura 3.18).

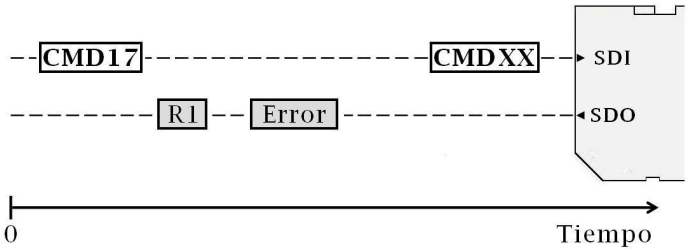


Fig. 3.17. Error en la lectura del bloque de datos.

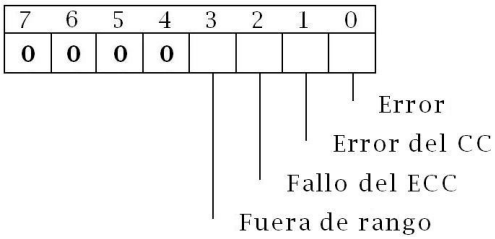


Fig. 3.18. *Token* de error.

'Error del CC' es un error interno del controlador de la tarjeta y 'Fallo del ECC' resulta del intento fallido para corregir los datos luego de que la tarjeta aplicara su código de corrección de error.

Escritura de datos

Para la operación de escritura de un sólo bloque de datos se usa el comando 24 y para la escritura de múltiples bloques, el comando 25.

Si el comando se ha validado en la escritura de un sólo bloque, la tarjeta contesta con la respuesta del comando 24 (respuesta tipo R1) y espera el envío del bloque de bytes precedido por el byte SB. La tarjeta reconoce el bloque recibido con el *Data Response Token* (DRT). Ver figuras 3.19 y 3.20(a).

7	6	5	4	3	2	1	0
X	**X**	**X**	**0**	Estatus			**1**

Fig. 3.19. Byte de DRT.

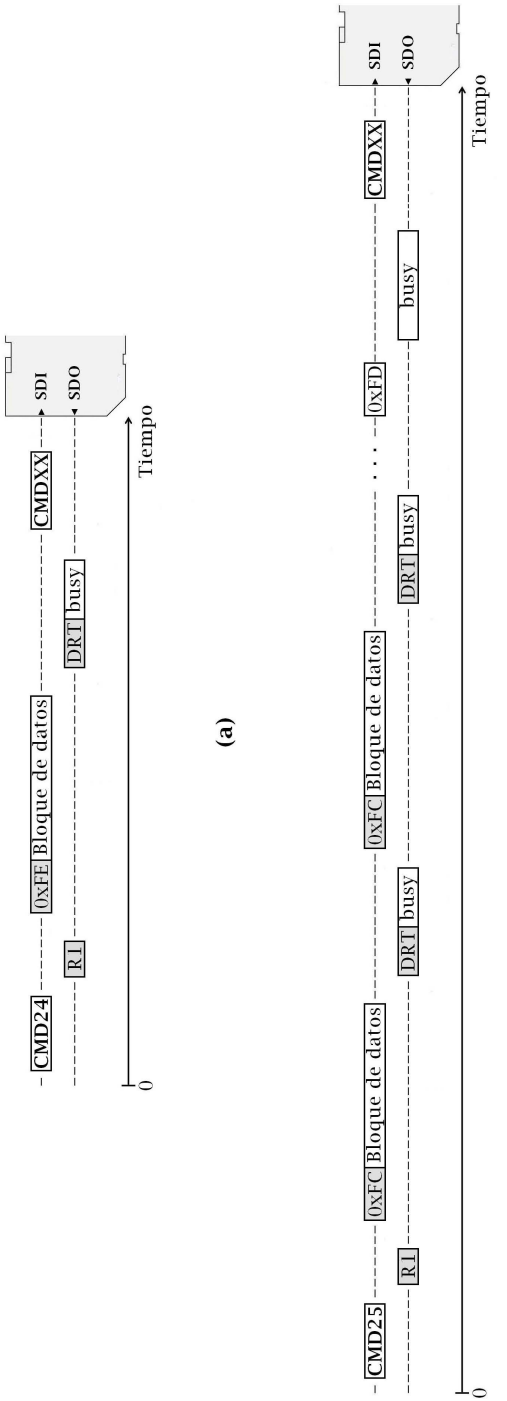


Fig. 3.20. Operación de escritura de datos.

Si la SD CARD recibió el bloque sin errores lo programa, y durante la programación mantendrá su terminal de salida de datos en bajo (estado *busy*). Terminada la programación, el microcontrolador deberá verificar el resultado de la escritura del bloque con el comando 13.

En la operación de múltiples bloques, el fin de la transmisión se indica enviando el *Stop Tran Token* (STT) en vez del *Start Block Token* para múltiples bloques (SBT) al inicio del siguiente bloque, como lo muestra la figura 3.20(b). El valor de STT es 0xFD y de SBT es 0xFC.

Los bits de estatus en el byte DRT se deberán analizar para asegurarse que no haya indicación de error en los procesos de escritura. Ver la tabla 3.4.

ESTATUS	SIGNIFICADO
0 1 0	Datos aceptados
1 0 1	Datos rechazados por error en CRC
1 1 0	Datos rechazados por error en escritura

Tabla 3.4. Bits de estatus en el byte DRT.

Formato de los bloques de datos Las SD CARD utilizan dos formas para direccionar una posición dentro de su área de memoria, este direccionamiento lo determina el tipo de tarjeta.

Las memorias SDHC utilizan direcciones de memoria en formato de bloques o en *block address format*, en estas tarjetas la longitud de los bloques de información que se transfieren en la comunicación es fijo a 512 bytes.

Las memorias SD utilizan direcciones de memoria en formato de bytes o en *byte address format*, en estas tarjetas la longitud de los bloques de información que se transfieren en la comunicación lo define el comando 16.

En el sistema de registro, el tamaño de los bloques de memoria utilizados para ambos tipos de tarjetas es de 512 bytes, es decir, cada bloque de memoria equivale a un sector.

En el argumento del comando 16 se especifica la longitud de los bloques en *byte address format* para las tarjetas SD, la sintaxis del comando para una longitud de 512 bytes se muestra.

0x50 0x00 0x00 0x02 0x00 0xFF

El comando 17 pide la lectura de un sólo bloque de datos. El comando 24 escribe un sólo bloque y el comando 25 escribe múltiples bloques de datos, en el argumento de estos comandos se indica la dirección del bloque de memoria con el formato adecuado según el tipo de tarjeta que se utilice.

La respuesta asociada a los comandos 16, 17, 24 y 25 es tipo R1.

Por ejemplo, si se quisiera leer el bloque de memoria dos (sector 2) de una tarjeta SDHC se enviaría el comando 17 con el siguiente argumento,

0x51 0x00 0x00 0x00 0x02 0xFF

y para leer el mismo sector en una tarjeta SD,

0x51 0x00 0x00 0x04 0x00 0xFF

como el tamaño de cada sector en la SD es de 512 bytes la dirección en el argumento del comando se especifica en múltiplos de 512.

Si se quisiera escribir en ese mismo sector se enviaría el comando 24. La sintaxis del comando para una memoria SDHC

0x58 0x00 0x00 0x00 0x02 0xFF

y para una SD,

0x58 0x00 0x00 0x04 0x00 0xFF

En el argumento del comando 25 se especifica la dirección de inicio de escritura. Mientras la tarjeta no reciba el STT se mantendrá escribiendo en sectores consecutivos.

Continuando con el ejemplo, para escribir múltiples sectores en una memoria SDHC empezando en el sector dos, la sintaxis del comando 25 sería,

0x59 0x00 0x00 0x00 0x02 0xFF

y para una memoria SD,

0x59 0x00 0x00 0x04 0x00 0xFF

Conociendo el número de sector que se quiere leer o escribir, el argumento del comando para las memorias SDHC es inmediato; para las memorias SD se determina calculando una simple regla de tres,

$$\frac{argumento}{512} = \frac{sectorX}{sector1}$$

argumento es el número de byte en el que empieza el sector x que se quiere leer o escribir.

Por ejemplo, si en ambos tipos de SD CARD se quiere escribir en el sector 496, el comando 24 para la SDHC queda,

0x58 0x00 0x00 0x01 0xF0 0xFF

Para la SD,

$$\frac{argumento}{512} = \frac{496}{1} \Rightarrow argumento = 496 \times 512$$

en hexadecimal, $argumento =$ 0x3E000 así,

0x58 0x00 0x03 0xE0 0x00 0xFF

En la figura 3.21 se ilustran los formatos de los bloques de memoria para ambos tipos de tarjetas.

El comando 13 se envía finalizada la operación de escritura para verificar el resultado de la programación de los datos en la tarjeta. Su argumento son bits de relleno,

0x4D 0x00 0x00 0x00 0x00 0xFF

y su respuesta tipo R2, de dos bytes (ver figura 3.22).

Código de detección de error Como se mencionó en la sección 2.2.4, el parámetro principal de cada algoritmo CRC es el polinomio generador. Éste marca la diferencia entre los dos CRC's que implementan las memorias SD CARD.

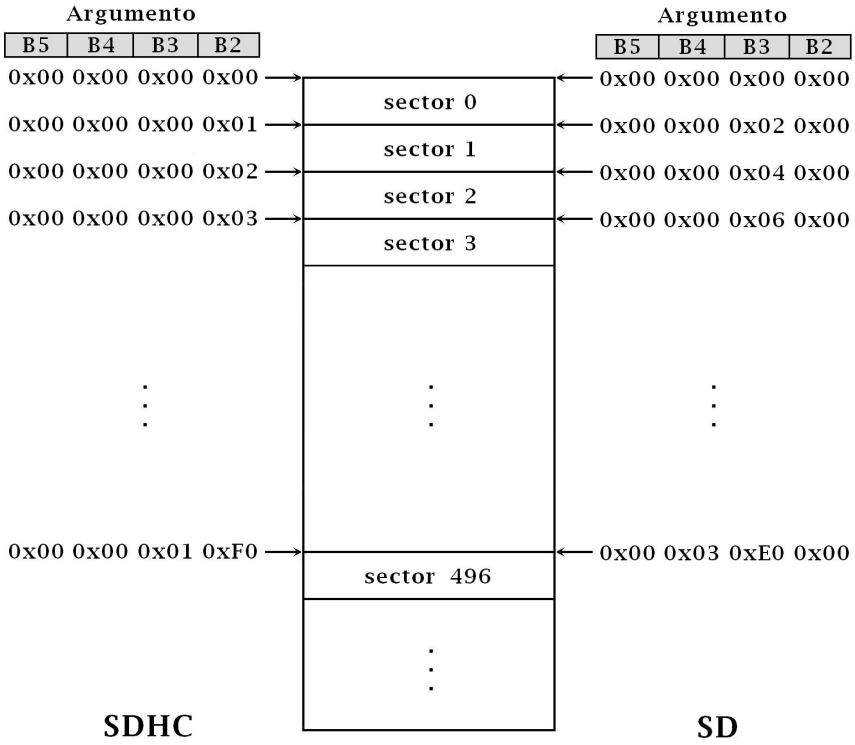


Fig. 3.21. Formatos de los bloques de memoria.

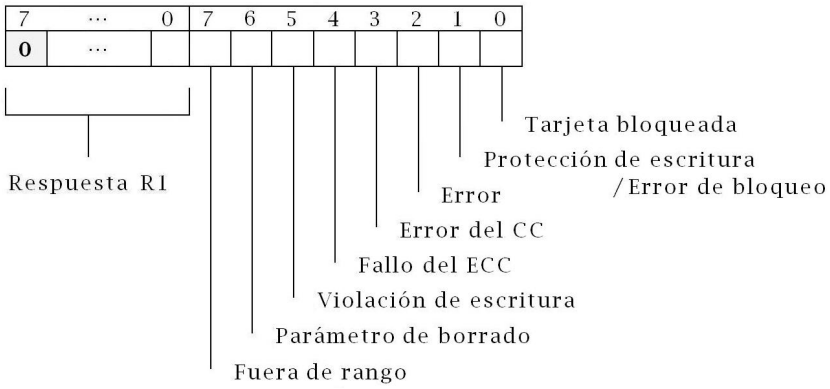


Fig. 3.22. Formato de la respuesta R2.

El CRC7 utiliza un polinomio generador de orden 7 y el CRC16 uno de orden 16. En la tabla 3.5 se muestran los polinomios para cada CRC junto con su representación binaria.

CRC	POLINOMIO GENERADOR
CRC7	$x^7 + x^3 + 1$ 1 0 0 0 1 0 0 1
CRC16	$x^{16} + x^{12} + x^5 + 1$ 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1

Tabla 3.5. Polinomios generadores.

El valor del CRC es el residuo de la división entre el mensaje (con ceros adjuntados) y el polinomio generador correspondiente. Son 7 bits de CRC para el CRC7 y 16 para el CRC16.

En el caso del CRC7 el mensaje está formado por los bits del comando; en el CRC16 el mensaje son los bits del bloque de datos enviado. Entonces hay un solo valor de CRC asociado a cada comando y a cada bloque de datos transferido.

En el sistema de registro de datos se implementó el CRC7, pero sólo se utilizó para obtener los bits de CRC válidos del comando ocho.

Los bits del mensaje abarcan del b6,B6 al b0,B2 del comando; se indican en la figura 3.23. Del b7,B1 al b1,B1 son los bits de CRC7 a calcularse.

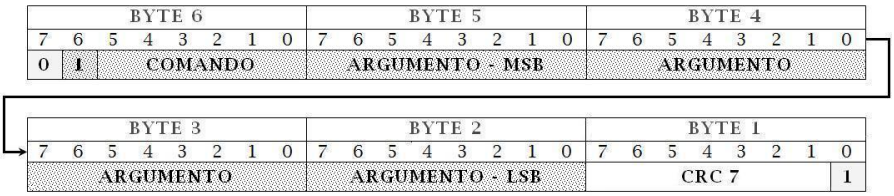


Fig. 3.23. Mensaje para el cálculo del CRC7.

La división queda como muestra la figura 3.24. El mensaje junto con los ceros forman el dividendo de la división. El polinomio generador es el divisor y el residuo es el CRC7.

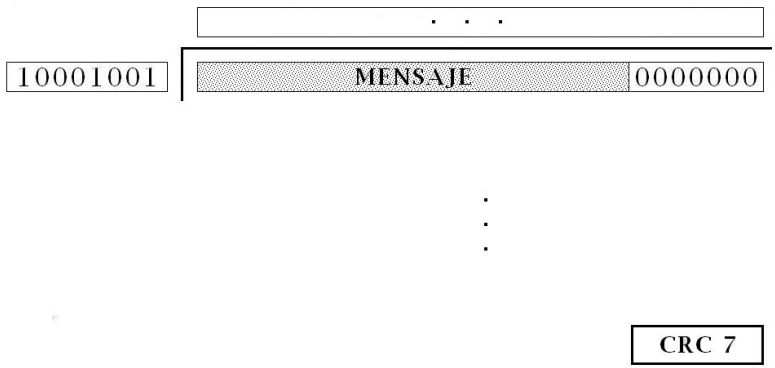


Fig. 3.24. División para el cálculo del CRC7.

La división se implementó utilizando corrimientos y operaciones XOR para la aritmética binaria sin acarreos.

En la tabla 3.6 se muestra el CRC7 generado para algunos comandos del proceso de inicialización.

NÚMERO DE COMANDO	VALOR DE CRC7
CMD 8	0x43
CMD 58	0x7E
CMD 55	0x32
ACMD 41	0x3B

Tabla 3.6. Algunos valores de CRC7.

Implementación del sistema de archivos

La implementación del sistema de archivos en las tarjetas de memoria SD CARD consiste de tres pasos:

- **Determinar el tipo de sistema de archivos de la tarjeta (FAT16 o FAT32)**
- **Calcular la posición de cada región del sistema de archivos dentro de la partición de la tarjeta**
- **Crear un archivo y asignarle clusters**

Estos pasos se discutirán más adelante. La idea tras la implementación es simple, sin embargo, la programación es paso a paso, y por lo mismo, larga y laboriosa.

La implementación correcta del sistema de archivos en cada tarjeta se comprueba fácilmente conectándola a una PC con lector de memorias SD CARD, o a través de un lector externo conectado a la PC. Si el archivo programado en la tarjeta se genera de forma compatible con los formatos leíbles por la computadora, y esta lectura es lograda, el sistema de archivos se ha implementado correctamente, de otro modo deberán revisarse las estructuras del sistema y su contenido, y para ello se utiliza la versión libre del WinHex.

El WinHex es un *software* que permite ver el contenido de las estructuras del sistema de archivos en la tarjeta de memoria, y en general, el contenido de cualquier unidad de almacenamiento que tenga implementado un sistema de archivos que éste soporte. El *software* WinHex se puede descargar de su página de internet[6].

La unidad de registro de datos implementa los sistemas de archivos FAT16 y FAT32.

A cada memoria SD CARD se le implementa un sólo tipo de sistema de archivos y éste dependerá del formato que la tarjeta reciba, previa conexión en la unidad de registro de datos.

La unidad de registro recibe formateada la SD CARD y lee su primer sector lógico para capturar la información del sistema de archivos. Con esta información implementa en la tarjeta el sistema correspondiente, FAT16 o FAT32, conforme con el documento *Microsoft Extensible Firmware Initiative FAT32 File System Specification Version 1.03*, que contiene las especificaciones de Microsoft para los tres sistemas de archivos de la familia FAT.

Formateo La tarjeta de memoria requiere ser formateada para recibir las estructuras del sistema de archivos. Dado que la tarjeta emula a una unidad de disco, el formateo que se le aplica es el correspondiente a un disco común.

[6] http://www.x-ways.net/winhex/

El proceso de formateo consta de tres pasos: el formateo de bajo nivel (*Low-Level Formatting* o *True Formatting*), la partición y el formateo de alto nivel (*High-Level Formatting*).

El formateo de bajo nivel crea las estructuras físicas en una unidad de disco, como los sectores; este proceso se realiza de fábrica. Las tarjetas de memoria no son divididas físicamente como los discos pero están diseñadas para agrupar el volumen en sectores. La partición divide a la unidad en volumenes lógicos y los prepara para dedicarlos a distintos usos. La tarjeta de memoria tiene una sola partición activa.

El formateo de alto nivel escribe en el disco la información del sistema de archivos; se hace después de la partición. Los procesos de partición y formateo de alto nivel son funciones que realiza el sistema operativo en una computadora y es a lo que comunmente le llamamos 'formatear'.

Las tarjetas de memoria se formatean utilizando una PC con sistema operativo Windows y así reciben la información de las estructuras del sistema de archivos FAT.

El formateo de las tarjetas se puede hacer básicamente de dos formas: desde el *shell* (línea de comandos o símbolo del sistema) o desde el ambiente Windows.

En las figuras 3.25 y 3.26 se muestra el formateo de una tarjeta de memoria SD de 1[GB] por ambos métodos.

Fig. 3.25. Formateo de una tarjeta SD desde el *shell*.

Una vez formateada la SD CARD, se lee su primer sector lógico para capturar la información del sistema de archivos que recibió.

Fig. 3.26. Formateo de una tarjeta SD desde el ambiente Windows.

Primer sector lógico El sector lógico cero o primer sector lógico de la única partición activa de la SD CARD es el *boot sector*. En el formateo el sistema operativo escribe en este sector toda la información necesaria para armar las estructuras del sistema de archivos.

En adelante llamaremos a los sectores lógicos simplemente sectores, entendiendo que nos referimos a los sectores dentro de la partición activa de la tarjeta de memoria.

La figura 3.27 muestra una representación del *boot sector*. Cada casilla contiene un byte y su posición se indica a través de un *offset*, acorde con la posición dentro de la memoria; los números dentro de las casillas señalan este *offset*.

El contenido del *boot sector* es distinto para el FAT16 y para el FAT32.

Durante el formateo la computadora determina que tipo de sistema de archivos le corresponde a la SD CARD conectada y, de acuerdo con ello, escribe la información del *boot sector*.

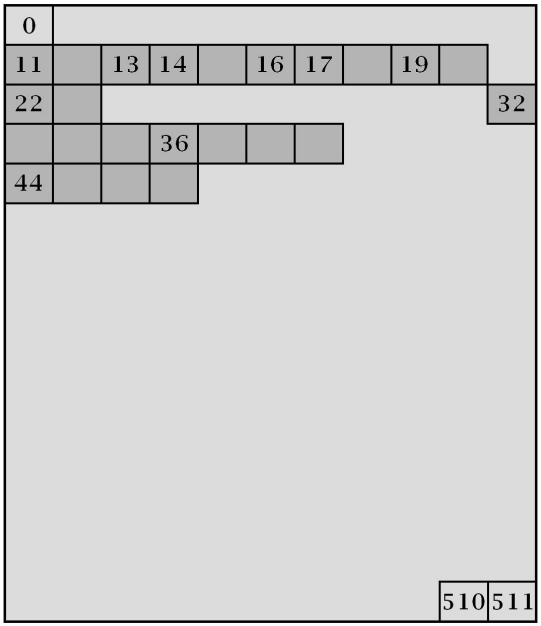


Fig. 3.27. *Boot sector.*

En las tablas 3.7 y 3.8 se muestran los campos capturados del *boot sector* para cada tipo de sistema de archivos, FAT16 y FAT32 respectivamente. Las variables mostradas proporcionan toda la información necesaria para implementar los sistemas de archivos en las tarjetas.

VARIABLE	OFFSET (en bytes)	TAMAÑO (en bytes)
Número de bytes por sector	11	2
Número de sectores por cluster	13	1
Número de sectores reservados	14	2
Número de FATs	16	1
Número de entradas en el directorio raíz	17	2
Cantidad total de sectores en el volumen	19	2
Número de sectores por FAT	22	2

Tabla 3.7. Información del *boot sector* para FAT16.

VARIABLE	OFFSET (en bytes)	TAMAÑO (en bytes)
Número de bytes por sector	11	2
Número de sectores por cluster	13	1
Número de sectores reservados	14	2
Número de FATs	16	1
Cantidad total de sectores en el volumen	32	4
Número de sectores por FAT	36	4
Número de cluster del primer cluster del directorio	44	4

Tabla 3.8. Información del *boot sector* para FAT32.

Al trabajar con los sistemas de archivos FAT debe tomarse en cuenta la manera en que la información se representa en estos sistemas.

La familia FAT se desarrolló originalmente para la arquitecura de las máquinas IBM PC y por ello la información de las estructuras de datos para estos sistemas de archivos está en *little endian*. Esto es importante si el procesador con el que se trabaja es *big endian*, porque se tendrá que hacer la conversión de un formato a otro mientras se lea y escriba en las estructuras del sistema de archivos.

En *little endian* el byte menos significativo de la palabra se escribe primero.

Supongamos que el valor del campo que corresponde a la variable `Número de bytes por sector` es 512. En el *boot sector* se leerá

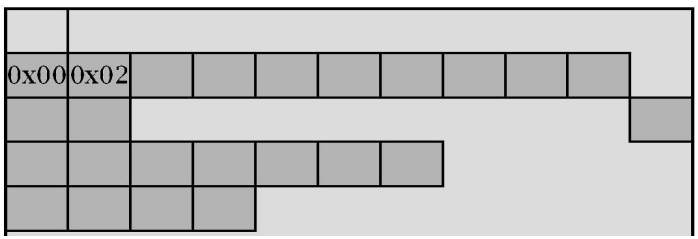


Fig. 3.28. Ejemplo de *little endian* en el *boot sector*.

y el valor se interpretará como **0x02 0x00**.

En este ejemplo el valor capturado nos indica que cada sector de la tarjeta de memoria tiene 512 bytes.

El PIC18LF452 almacena su información en *little endian* por lo que la manipulación de las variables del *boot sector* es inmediata.

Para leer el *boot sector* el argumento del comando de lectura se fija a cero y se sigue el protocolo de la SD CARD para recibir la información del sector. Se ilustra una parte de este proceso en el código a continuación.

```
WriteCMD ( CMD_17 ) ;
Wait_R1 ( idle ) ;

do {
        WriteSPI ( idle ) ;
} while ( SSPBUF != SB ) ;

for ( j = 0 ; j < 11 ; j ++  )  // let go from
{                               // offset 0 to 10
        WriteSPI ( idle ) ;
}

i = 0 ;

do {                        // get BS[0] to BS[9]
        WriteSPI ( idle ) ;
        BS[ i ] = SSPBUF ;
        i ++ ;
} while ( i <= 9 ) ;

        .
        .
        .
```

La función `WriteCMD` se encarga de formar el comando especificado en su argumento y lo envía a la memoria SD CARD a través de la función `WriteSPI`. Esta última es la función que implementa la comunicación SPI en el microcontrolador y la proporciona Microchip. En el código siguiente se esboza la manera en que la función `WriteCMD` se programó.

```
#define CMD_00 0x40
#define CMD_17 0x51
```

```
            .
            .
            .

unsigned char WriteCMD ( unsigned char CMD )
{
   switch ( CMD ) {
       case CMD_00:
              Response = R1 ;
              B5 = 0x00 ;
              B4 = 0x00 ;
              B3 = 0x00 ;
              B2 = 0x00 ;
              B1 = 0x95 ;
              break ;
            .
            .
            .

       case CMD_17:    // read 1 block
              B5 = Byte5__BlockAddr ;
              B4 = Byte4__BlockAddr ;
              B3 = Byte3__BlockAddr ;
              B2 = Byte2__BlockAddr ;
              B1 = 0xff; // ---any CRC as CRC is off
              break ;
            .
            .
            .

   }

   for( i = 6 ; i >= 1 ; i-- ){
       switch ( i ) {
            case 6:
                   ss=0 ;
                   WriteSPI( CMD ) ;
                   break ;
            case 5:
```

```
                WriteSPI( B5 ) ;
                break ;
            case 4:
                WriteSPI( B4 ) ;
                break ;
            case 3:
                WriteSPI( B3 ) ;
                break ;
            case 2:
                WriteSPI( B2 ) ;
                break ;
            case 1:
                WriteSPI( B1 ) ;
                break ;
            default:
                WriteSPI( idle ) ;
                break ;
        }
    }

}
```

El código programado en el microcontrolador utiliza estructuras, definiciones de tipos de datos, enumeraciones, apuntadores, *casting* y librerías propias como medio para ordenar y manipular la información de las variables del *boot sector* que llevarán a la implementación del sistema de archivos.

Con el fin de dar una idea de la forma en que la programación se llevó a cabo, se ilustra a continuación varios de los conceptos mencionados para las variables `Número de bytes por sector` y `Número de sectores por cluster` del *boot sector*.

Los campos de interés del sector se guardan en el arreglo `BS`. El microcontrolador captura siempre 24 campos del *boot sector* de cada tarjeta.

```
for ( i = 0 ; i <= 23 ; i ++ )
{
        BS_Struct ( & BS[ i ] , i ) ;
}
```

La función `BS_Struct` separa el arreglo `BS` para asignar las variables de las tablas 3.7 y 3.8.

Cada variable del *boot sector* se programa como una estructura de datos que guarda el valor y el tamaño en bytes de cada campo.

En el siguiente extracto de código se muestra la definición y asignación de las estructuras de datos correspondientes a las variables `Número de bytes por sector` y `Número de sectores por cluster` del arreglo `BS`.

```
typedef struct {

       BYTE * value ;
       enum sizes cntbytes ;
} BS_1BV ; // Boot Sector 1 Byte Value

typedef struct {

       WORD * value ;
       enum sizes cntbytes ;
} BS_2BV ; // Boot Sector 2 Byte Value

BS_1BV SecsPerClus ;
BS_2BV BytesPerSec ;

char BS_Struct ( BYTE * BSi , int i )  // Boot Sector
{                                       // Structure
       switch ( i ) {

       case 0 :
              BytesPerSec.value = ( WORD * ) BSi ;
              BytesPerSec.cntbytes = two ;
              break;
       case 2 :
              SecsPerClus.value = BSi ;
              SecsPerClus.cntbytes = one ;
              break;
        }
}
```

De esta forma la unidad de registro obtiene las variables del *boot sector* para cada tarjeta que se le conecta. Estas variables las podemos cotejar con las mostradas por el *software* WinHex.

WinHex

Este *software* es un editor hexadecimal particularmente útil en la computación forense y en la recuperación de datos; es una herramienta avanzada para inspeccionar, editar y recuperar información perdida o borrada de unidades de disco duro o de tarjetas de memoria con sistemas de archivos que se han corrompido. Las características del *software* dependen del tipo de licencia con la que se adquiera.

Fig. 3.29. Pantalla de inicio del WinHex.

La versión libre del WinHex permite, en términos generales, ver el contenido de la tarjeta de memoria que está conectada a la computadora donde reside este *software*, y para nuestro objetivo esta función es suficiente.

La figura 3.30 muestra los pasos para abrir una tarjeta de memoria conectada en una PC que corre WinHex; el sistema operativo detecta a la tarjeta del ejemplo como la unidad de disco duro E:

La figura 3.31 muestra el *boot sector* de la tarjeta de memoria y a su lado se muestra el menú de las estructuras de su sistema de archivos. En este menú se selecciona la plantilla mostrada en la figura 3.32.

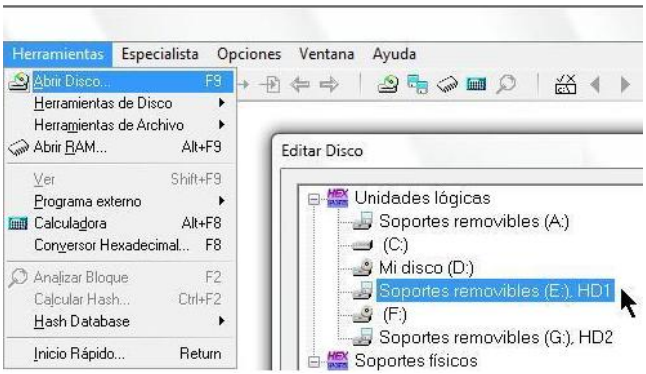


Fig. 3.30. Menú de inicio del WinHex.

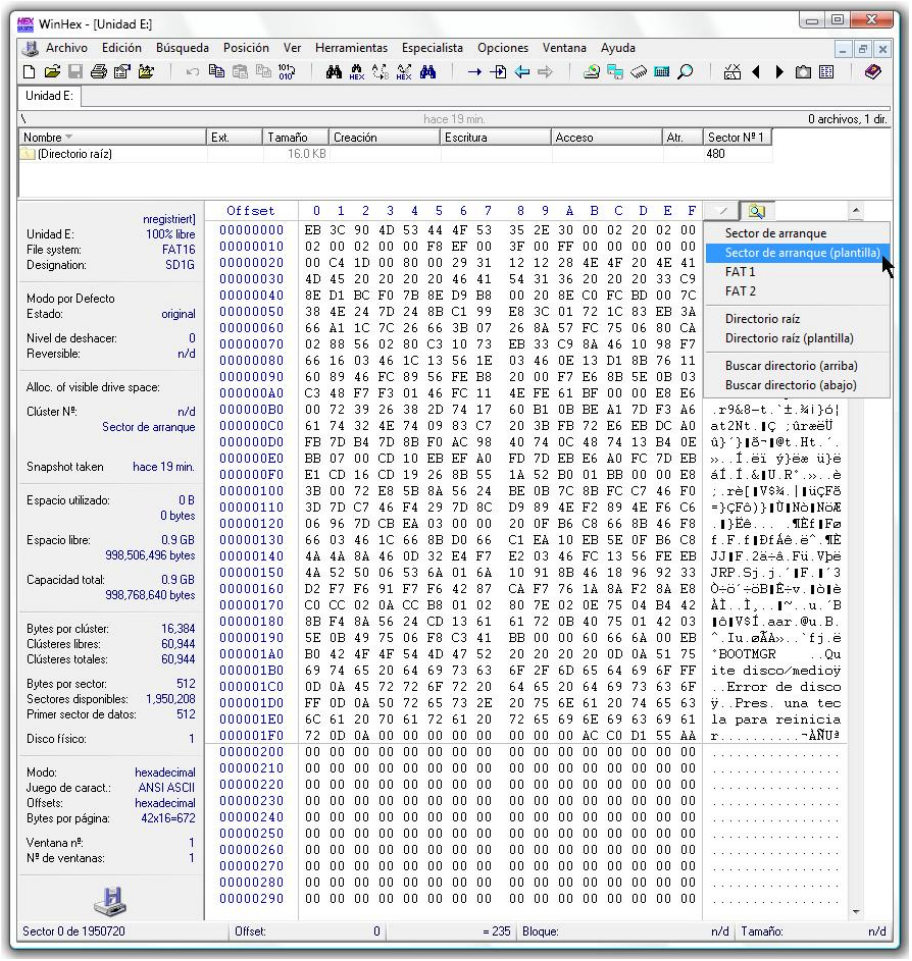


Fig. 3.31. *Boot sector* en WinHex.

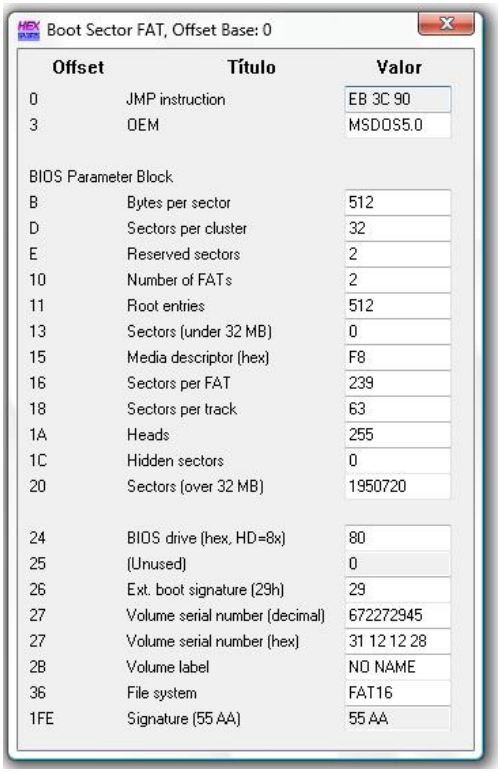


Fig. 3.32. Plantilla del *boot sector*.

Las figuras 3.30, 3.31 y 3.32 corresponden a la tarjeta de memoria SD de 1[GB] de las figuras 3.25 y 3.26.

De la misma forma en que WinHex permite visualizar el contenido del *boot sector* en este ejemplo, también lo permite para cualquier estructura del sistema de archivos de la tarjeta de memoria que se conecte a la PC.

Determinación del tipo de FAT Una vez que la unidad de registro de datos obtiene las variables del *boot sector* las opera para conocer la cantidad de clusters en la región de datos, y con ello determina el tipo de sistema de archivos de la SD CARD conectada. En otras palabras, la unidad de registro calcula a partir de las variables capturadas del *boot sector* cuántos clusters del total de clusters de la partición de la tarjeta corresponden a la región R3.

Para ello se determina primero la cantidad de sectores que hay en cada región, luego se calcula el total de sectores en R3 y este valor se

convierte a clusters. Finalmente se decide a partir de este valor el tipo de FAT.

Para explicar el proceso de determinación del tipo de FAT nos auxiliaremos de fragmentos de código. Comenzaremos explicando la forma en que se determina la cantidad de sectores en el directorio, en la FAT y en el volumen, para luego calcular la cantidad de sectores en la región de datos que nos llevará finalmente a determinar el tipo de FAT.

Sectores del directorio

Primero se determina la cantidad de sectores de la región del directorio para ambos sistemas de archivos.

En el código presentado a continuación, `RootEntCnt_` y `BytesPerSec_` son apuntadores de dos bytes, almacenan en sus direcciones de memoria respectivamente, el número de entradas del directorio raíz y el número de bytes por sector.

```
extern BS_2BV RootEntCnt, BytesPerSec ;
BS_2BV * ptr2B ; // pointer to 2 Byte values
WORD * RootEntCnt_ , * BytesPerSec_ ;

ptr2B = & RootEntCnt ;

RootEntCnt_ = ptr2B -> value ;

ptr2B = & BytesPerSec ;

BytesPerSec_ = ptr2B -> value ;

        if (  ( * RootEntCnt_ ) != 0L  ) {

          dividend = ( ( * RootEntCnt_ ) * 32 ) ;
          divisor =  ( * BytesPerSec_ ) ;

          RootDirSecs = dividend / divisor ;

          res = dividend % divisor ;
```

```
   if ( ( res * 2 ) > divisor )
      RootDirSecs = RootDirSecs + 1 ;

}

else {

   RootDirSecs = 0 ; // for FAT32 only

}
```

Cuando la condición *if* es cierta, existe la región del directorio raíz (R2) y se calcula la cantidad de sectores de esta región redondeando hacia arriba, en caso contrario, no existe R2 y la tarjeta es tipo FAT32. El número de sectores del directorio raíz se guarda en la variable `RootDirSecs`, de cuatro bytes.

Sectores por FAT

`FATSz16_` y `FATSz32_` son apuntadores de dos y cuatro bytes, almacenan en sus direcciones de memoria el número de sectores por FAT para cada sistema de archivos. Su valor se asigna a la variable de cuatro bytes `FATSz` como se muestra.

```
ptr2B = & FATSz16 ;

FATSz16_ = ptr2B -> value ;

if (  ( * FATSz16_ ) != 0 )

        FATSz = * FATSz16_ ; // for FAT16

else {

        ptr4B = & FATSz32 ;
        FATSz32_ = ptr4B -> value ;
        FATSz = * FATSz32_ ; // for FAT32 only
}
```

Cantidad total de sectores en el volumen

La variable de cuatro bytes `TotSec` guarda la cantidad total de sectores en el volumen de acuerdo con la condición mostrada.

```
if ( * ( TotSec16.value ) != 0 )

        TotSec = * ( TotSec16.value ) ;

else

        TotSec = * ( TotSec32.value ) ;
```

`* ( TotSec16.value )` y `* ( TotSec32.value )` regresan el valor al que apuntan los apuntadores `TotSec16` y `TotSec32` de dos y cuatro bytes respectivamente, que es el valor de las variables `Cantidad total de sectores en el volumen` para cada sistema de archivos.

Cantidad de sectores en la región de datos

El total de sectores en la región de datos (R3) se calcula como se muestra a continuación (ver figura 2.17):

```
/*

Total Count of Sectors in
the Cluster Area (Data Region) =

Total Count of Sectors on the Volume -

[ Number of Reserved Sectors in the Reserved Region +

( Number of FATs * Sectors per FAT ) +

Count of sectors in the Root Directory ]

=> R3  =  R0 + R1 + R2 + R3  -  [ R0 + ( R1 ) + R2 ]

*/
```

```
DWORD DataSec;

DataSec = TotSec - ( * ( RsvdSecCnt.value ) +
          ( * ( NumFATs.value ) * FATSz ) + RootDirSecs ) ;
```

`* ( RsvdSecCnt.value )` regresa el número de sectores reservados y `* ( NumFATs.value )` regresa el número de FATs.

`DataSec` es una variable de cuatro bytes que guarda el cálculo de la cantidad total de sectores en la región de datos y este valor se convierte a clusters, como se muestra a continuación:

```
CountOfClusters =
     DataSec / ( * ( SecsPerClus.value ) ) ;
```

Tipo de FAT

Finalmente el tipo de FAT se determina como sigue.

```
       if ( CountOfClusters < 4085 ) {

               /* Volume is FAT12 => NOT SUPPORTED ! */
       }

       else {

               if ( CountOfClusters < 65525 ) {
                       /* Volume is FAT16 */
                       fattype = FAT16 ; // 0x16 ;
               }

               else {
                       /* Volume is FAT32 */
                       fattype = FAT32 ; // 0x32 ;
               }

       }

Display ( fattype ) ;
```

La función `Display ( fattype )` despliega el tipo de FAT a través de la interfaz de depuración de la unidad de registro de datos.

La interfaz de depuración utiliza una terminal virtual en la PC para mostrar los bytes que el microcontrolador de la unidad de registro de datos envía. Esta terminal es especialmente útil porque permite ver los bytes en forma hexadecimal, a diferencia de otras terminales que restringen el despliegue en caracteres ASCII.

La terminal virtual utilizada se llama RealTerm, es un *software* libre y se puede descargar desde su página de internet[7].

La figura 3.33 muestra la pantalla de esta terminal y la figura 3.34 muestra un acercamiento; los valores 32 y 16 son el resultado de la función `Display ( fattype )` para dos tarjetas de memoria que se conectaron en la unidad de registro, una con sistema de archivos FAT32 y otra con sistema FAT16.

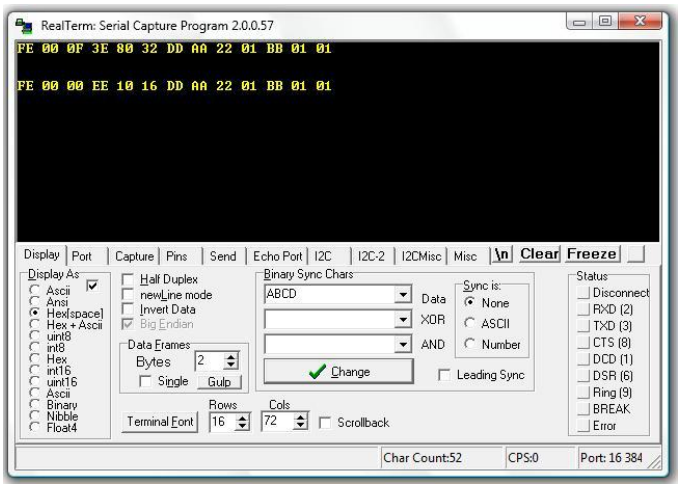


Fig. 3.33. Terminal virtual RealTerm.

Cálculo de la posición de cada región Una vez determinado el tipo de sistema de archivos, se tiene toda la información necesaria para calcular el sector de inicio de cada región de la partición. El número del primer sector de cada región es el sector de inicio de esa región, este sector indica la posición de la región dentro de la tarjeta de memoria.

[7] http://realterm.sourceforge.net/

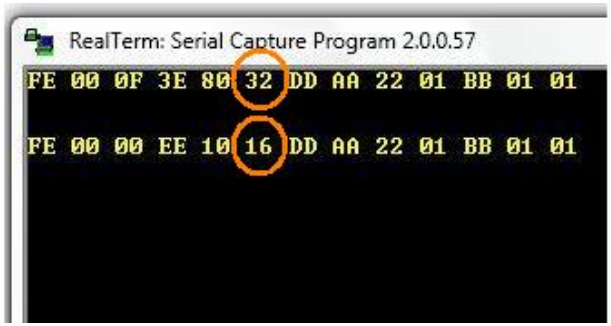


Fig. 3.34. Tipo de FAT en RealTerm.

El sistema de archivos FAT16 contempla las cuatro regiones: R0, R1, R2 y R3, mientras que el sistema FAT32 contempla las regiones: R0, R1 y R3. Las operaciones para calcular los sectores de inicio son las mismas para ambos sistemas de archivos y los operandos se obtienen de la información capturada del *boot sector* para cada sistema. El procedimiento para el cálculo de los sectores de inicio se describe a continuación y para ello nos referiremos a la figura 2.17.

Sectores de inicio

El sector de inicio de cada región corresponde al primer sector de cada estructura del sistema de archivos; por ejemplo, si el número del primer sector de la región dos (sector de inicio de la región dos) es el sector 100, el primer sector de la estructura directorio raíz será el sector 100.

Región cero

El sector de inicio de R0 corresponde al *boot sector* y de forma inmediata se conoce su sector de inicio, el sector cero.

Región uno

El sector de inicio de R1 marca el inicio de la FAT y en esta región hay por lo general dos tablas: la FAT1 y la FAT2. La FAT1 es la tabla de asignación principal y una copia de ella se escribe en la FAT2 como respaldo, por si la FAT1 se corrompe de algún modo.

El sector de inicio de la región FAT es el primer sector de la FAT1. El número de este sector se obtiene directamente de la variable `Número de`

`sectores reservados`, su valor lo regresa `* ( RsvdSecCnt.value )` y se guarda en la variable `FATBegin` de cuatro bytes.

```
FATBegin = * ( RsvdSecCnt.value ) ;
```

Región dos

Esta región sólo existe para FAT16 y su sector de inicio corresponde al primer sector del directorio raíz. El número de sector del primer sector del directorio raíz se calcula como sigue.

```
if ( fattype == FAT16 )

FirstRootDirSecNum =
* ( RsvdSecCnt.value ) + ( * ( NumFATs.value ) * FATSz ) ;
```

`* ( NumFATs.value )` regresa el `Número de FATs` en la región R1 y su valor se multiplica por la variable `FATSz` que guarda el número de sectores por FAT. El resultado se asigna a la variable de cuatro bytes `FirstRootDirSecNum`.

Región tres

El sector de inicio de R3 corresponde al primer sector de la región de datos y se calcula de la siguiente forma.

```
FirstDataSector = * ( RsvdSecCnt.value ) +
                ( * ( NumFATs.value ) * FATSz ) + RootDirSecs ;
```

La variable de cuatro bytes `FirstDataSector` almacena el número del primer sector de la región de datos.

NOTA: Las especificaciones de Microsoft establecen que la región de datos siempre inicia en el cluster número dos, porque el cluster 0 y el cluster 1 son reservados. Entonces, el sector de inicio de la región de datos es el primer sector del cluster 2. Esto deberá tomarse en cuenta cuando se requiera calcular el primer sector de un cluster en la región de datos.

Los sectores de inicio, como cualquier otro sector, se direccionan como lo explica el apartado "Formato de los bloques de datos". De esta forma, en el argumento de los comandos de escritura y lectura se puede especificar cualquier sector dentro de cualquier región de la tarjeta de memoria.

Creación de un archivo y asignación de clusters En este punto se conoce la posición en memoria de cada estructura para ambos sistemas de archivos. Con esta información podemos acceder a sectores de interés (sectores de los que vamos a leer o a los que les vamos a escribir bytes) dentro de cada región; el *boot sector*, por ejemplo, es un sector de interés. Es importante remarcar que la lectura y escritura en la tarjeta de memoria se hace por sectores y la cantidad más pequeña de información que se puede leer o escribir en la SD CARD son 512 bytes.

Los sectores de interés son leídos por el sistema operativo de una computadora para generar el archivo que le indican las estructuras del sistema de archivos, de acuerdo con las especificaciones de Microsoft para los sistemas FAT16 y FAT32. Así, para crear un archivo necesitamos escribir en sectores de interés dentro de las estructuras de los sistemas de archivos.

El sistema de registro de datos crea un sólo archivo y lo hace en tres pasos:

1. Escribe en el directorio las características del archivo.
2. Escribe en la FAT la secuencia de clusters asignados al archivo.
3. Escribe el contenido del archivo en la región de datos.

A continuación se explica cada uno de ellos.

Escritura en el directorio

Como se vio en el capítulo anterior, cada sector del directorio tiene 16 *records* o entradas, y en los 32 bytes de cada una se escriben las características de un sólo archivo, como lo indica la figura 2.18. Esto quiere decir que cada entrada identifica a un archivo, por lo que en un sector del directorio se pueden especificar un máximo de 16 archivos.

La primera entrada del primer sector del directorio identifica el archivo que corresponde a la etiqueta del volumen; esta entrada la proporciona el sistema operativo cuando se formatea la tarjeta de memoria. Este archivo siempre debe estar presente en el directorio y se reconoce porque el valor de su byte de atributo es 0x08.

De acuerdo con lo anterior, la entrada del archivo que vamos a crear no puede ocupar la primera entrada del directorio, pero sí la siguiente, y en general, puede ocupar cualquier otra entrada de cualquier sector del directorio. El sistema de registro de datos escribe las características del archivo a crear en la entrada siguiente a la entrada del archivo de la etiqueta del volumen. Para ello, el microcontrolador lee el primer sector del directorio y captura los 32 bytes del archivo de la etiqueta del volumen, luego, genera un *buffer* de 512 bytes cuyos primeros 32 son los capturados en la lectura y los siguientes 32 bytes corresponden a la entrada del archivo que nos interesa crear; el resto del buffer se llena con ceros. Finalmente, este *buffer* se escribe en el primer sector del directorio y con ello se han escrito las características del archivo a crearse en la segunda entrada de este sector del directorio.

Con la intención de dar una idea de la programación, se presentan fragmentos de código relevantes que llevan a cabo el procedimiento descrito.

La función `WriteRootDir` escribe el primer sector del directorio para cada sistema de achivos a través de la función `WriteSingleBlock`. La función `WriteRootDir` calcula el número del primer sector del directorio para cada sistema y luego llama a la función `WriteSingleBlock`. El argumento de esta última función pide el número del primer sector calculado, la función `ReadRootDir` y el valor de la constante de enumeración `RootDirvalues`.

A continuación se muestra la función `WriteRootDir`.

```
void WriteRootDir ( void )
{
    if ( fattype == FAT32 ) {      /* FAT32 */

        ptrDW =
        Clus2Sec (  ( BYTE * ) ( RootClus.value ) , fattype  ) ;
    }
```

```
    else {                          /* FAT16 */

       ptrDW = & FirstRootDirSecNum ;

       SDSectorAdjust =
       * ( ptrDW ) * (  * ( BytesPerSec.value )  ) ;

       ptrDW = & SDSectorAdjust ;
    }

WriteSingleBlock (  * (  ( BYTE * ) ptrDW + 3  ) ,
                    * (  ( BYTE * ) ptrDW + 2  ) ,
                    * (  ( BYTE * ) ptrDW + 1  ) ,
                    * (  ( BYTE * ) ptrDW  ) ,
                    ReadRootDir ( fattype ) , RootDirvalues  ) ;
}
```

La función `Clus2Sec` calcula, a partir del número de cluster y del tipo de FAT especificado en su argumento, el número del primer sector de ese cluster (ver la nota de la sección anterior). La función regresa el apuntador de cuatro bytes `Clus32ptr` que apunta al valor calculado. La forma en que `Clus2Sec` determina este valor se muestra a continuación.

```
WORD * Clus16ptr ;
DWORD * Clus32ptr , FstSecOfClus ;

// Given any cluster number N, Clus2Sec computes the
// sector number of the first sector of that cluster

DWORD * Clus2Sec ( BYTE * N , char fattype )

{
    switch ( fattype ) {

        case FAT16 :
                        Clus16ptr = ( WORD * ) N ;
                        * Clus32ptr = * ( DWORD * ) Clus16ptr ;
                        break;
```

```
        case FAT32 :
                        Clus32ptr = ( DWORD * ) N ;
                        break;
        }

FstSecOfClus = FirstDataSector +
     (  ( * Clus32ptr - 2 ) * (  * ( SecsPerClus.value )  )  ) ;

Clus32ptr = & FstSecOfClus ;

return ( Clus32ptr ) ;

}
```

La función `WriteSingleBlock` solicita en su argumento a la función `ReadRootDir` y al valor de la constante `RootDirvalues`. La información proporcionada por ambas la utiliza `WriteSingleBlock` para conocer el valor y la posición de cada byte dentro del *buffer* que se va a escribir en el primer sector del directorio.

La función `ReadRootDir` lee el primer sector del directorio y atrapa los 32 bytes que corresponden a la entrada del archivo de la etiqueta del volumen. Su valor de regreso es un apuntador al primer byte de esta entrada.

La constante de enumeración `RootDirvalues` forma parte de la enumeración `SDB`. Cada elemento de la enumeración es una constante simbólica a la que se le asigna un número, como se muestra.

```
typedef enum SingleDataBlock

{
        RootDirvalues = 0 ,
        FATvalues ,
        WriteTestvalues,
        WriteTestMultiple

} SDB ;
```

El número de `RootDirvalues` lo utiliza la función `WriteSingleBlock` para seleccionar la función que generará el *buffer* que se escribirá en el directorio. Esta función es `RootDirValues`, como se muestra.

```
SDB DataBlockValue ;

switch ( DataBlockValue ) {

       case 0 :
              RootDirValues ( FstByteOfData ) ;
              break;
       case 1 :
              FATValues ( FstByteOfData ) ;
              break;
       case 2 :
              WriteTest ( FstByteOfData ) ;
              break;

}
```

`RootDirValues` genera el *buffer* de 512 bytes que se escribirá en el primer sector del directorio. Para ello, la función se apoya de otra función, llamada `DirEntData`.

El contenido de cada entrada del directorio se ordena con la estructura `DirEntry`. La función `DirEntData` fija las características del archivo con esta estructura; se muestra su definición a continuación.

```
typedef struct {

       BYTE * Name ;           // 5 bytes
       BYTE * Ext ;            // 3 bytes
       BYTE Attr ;             // 1 byte
       BYTE * BlankSpace1 ;    // 8 bytes
       BYTE * FrstClusHI ;     // 2 bytes
       BYTE * BlankSpace2 ;    // 4 bytes
       BYTE * FrstClusLO ;     // 2 bytes
       BYTE * Size ;           // 4 bytes

} DirEntry ;
```

El valor que le corresponde a cada miembro de la estructura lo asigna `DirEntData` de la siguiente forma.

```
void DirEntData ( void ) {

DirEntry File, * ptrDir ;

static BYTE FileName [ 8 ] = "FILE1   " ; // 8 bytes
static BYTE FileExt [ 3 ] = "TXT" ; // 3 bytes
static BYTE Attribute = 0x20 ; *** ver NOTA
static BYTE Blank1 [ 8 ] =
{ 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 , 0x00 } ;
static WORD FrstClusH = 0 ;
static BYTE Blank2 [ 4 ] = { 0x00 , 0x00 , 0x00 , 0x00 } ;
static WORD FrstClusL = 3 ;
static DWORD FileSize = 12345678 ; // in bytes

ptrDir = & File ;

        ptrDir -> Name =  ( BYTE * ) & FileName ;
        ptrDir -> Ext = ( BYTE * ) & FileExt ;
        ptrDir -> Attr = Attribute ;
        ptrDir -> BlankSpace1 = ( BYTE * ) & Blank1 ;
        ptrDir -> FrstClusHI = ( BYTE * ) & FrstClusH ;
        ptrDir -> BlankSpace2 = ( BYTE * ) & Blank2 ;
        ptrDir -> FrstClusLO = ( BYTE * ) & FrstClusL ;
        ptrDir -> Size = ( BYTE * ) & FileSize ;

}
```

NOTA: El valor del byte de atributo se debe fijar a 0x20. Este valor le indica al sistema operativo que la entrada corresponde a la de un archivo.

Como se había dicho, la función `RootDirValues` genera byte a byte el *buffer* que se escribirá en el directorio. El *buffer* está formado en sus primeros 32 bytes por la entrada del archivo de la etiqueta del volumen, en los siguiente 32 bytes por la entrada del archivo que estamos creando y el resto de los 512 bytes se rellena con ceros.

La funciones `WriteSingleBlock` y `RootDirValues` trabajan en conjunto para escribir el *buffer* en la tarjeta de memoria.

El código que se mostrará a continuación presenta una parte del proceso que sigue la función `RootDirValues` para generar y escribir los 512 bytes del *buffer* en el primer sector del directorio.

```
j = 0 ;

for ( i = 0 ; i <= 511 ; i ++ ) {

     if ( i <= 31 ) {          // RD[0] to RD[31]
        WriteSPI ( * FstByteOfData1 ) ;
        FstByteOfData1 ++ ;
     }
     else {                    // FILE1 entry
          if (  ( i >= 32 )  &&  ( i < 64 )  ) {

/* File */      if (  ( i >= 32 )  &&  ( i <= 39 )  ) {
/* Name */         WriteSPI (  * ( ( File.Name ) + j )  ) ;
/* 8 bytes */      j ++ ;

                   if ( i == 39 )
                      j = 0 ;
                }

/* File */      if (  ( i >= 40 )  &&  ( i <= 42 )  ) {
/* Extension */    WriteSPI (  * ( ( File.Ext ) + j )  ) ;
/* 3 bytes */      j ++ ;

                   if ( i == 42 )
                      j = 0 ;
                }

/* File */       if (  ( i == 43 )  ) {
/* Attribute */     WriteSPI (  ( File.Attr )  ) ;
/* 1 byte */     }
                                .
                                .
```

```
                                .
        }
        else {  // (  ( i >= 64 )  &&  ( i <= 511 )  )
               WriteSPI ( 0x00 )  ;
        }
    }

}
```

Una vez que la entrada de nuestro archivo se ha escrito en el directorio de la tarjeta de memoria, se puede verificar en el WinHex.

La figura 3.35 presenta el resultado del proceso descrito; la entrada de nuestro archivo es la segunda entrada del primer sector del directorio de una SD CARD de 4[GB].

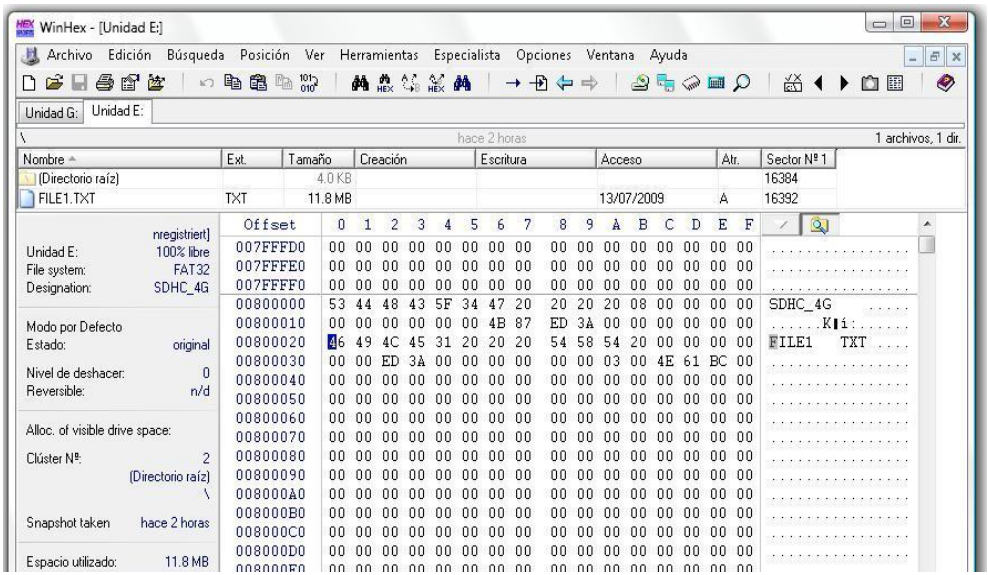


Fig. 3.35. Entrada del archivo en el directorio de una SDHC.

En el WinHex la plantilla del directorio se puede ver para cada entrada. La figura 3.36 muestra la primera entrada del directorio (entrada cero), que es la entrada del archivo de la etiqueta del volumen. La figura 3.37 muestra la siguiente entrada del directorio (entrada uno), que es la entrada de nuestro archivo.

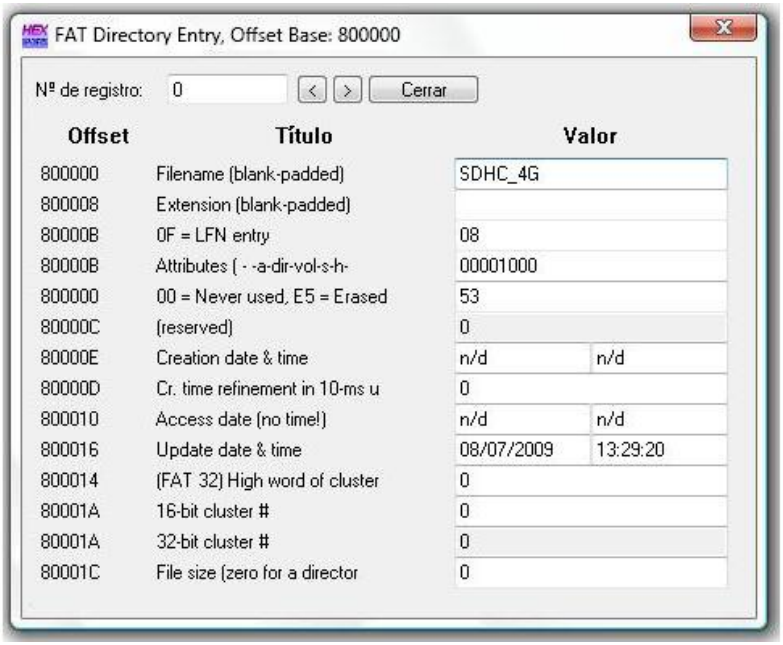


Fig. 3.36. Plantilla de la entrada cero del directorio.

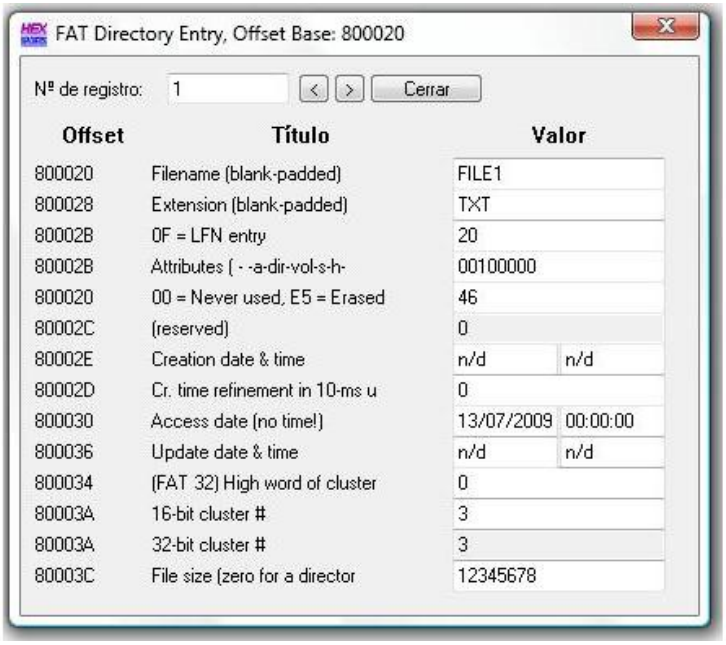


Fig. 3.37. Plantilla de la entrada uno del directorio.

Con la información de la entrada de nuestro archivo, se da inicio al siguiente paso para crearlo.

Escritura en la FAT

Hasta este punto se ha creado la 'identificación' del achivo. Con ella sabemos que se trata de un archivo de texto, de nombre `FILE1`, de `12345678 bytes` y que sus datos empiezan en el cluster número tres (inicio de `FILE1`). Con la información que proporciona la identificación se forma el 'contenedor' del archivo, que es donde los datos se van a 'vertir'. El contenedor se arma escribiendo la secuencia de clusters en la FAT.

Como se mencionó en el capítulo anterior, todo cluster tiene una entrada en la FAT con un valor. La entrada es simplemente la posición que ocupa ese cluster dentro de la FAT y el valor o contenido de esa entrada, indica el número del siguiente cluster de la cadena de clusters que forma al archivo.

La longitud de las entradas en la FAT depende del tipo sistema de archivos. La longitud de cada entrada para las tarjetas con sistema FAT16 es de 2 bytes y para las tarjetas con sistema FAT32 es de 4 bytes. De acuerdo con lo anterior, en la FAT se escribe la secuencia de clusters que se asignan al archivo, es decir, se escribe el valor de la entrada de cada cluster que forma al archivo.

El sistema de registro de datos hace una asignación de clusters consecutiva empezando en el primer sector de la FAT1. Para escribir en la tabla, se ordena primero la información de la entrada del directorio para `FILE1`. Luego, se examina el tamaño de `FILE1` para determinar la cantidad de clusters que le corresponde, y con esto se conoce cuántas entradas tiene el archivo en la FAT. Finalmente, a cada entrada se le da un valor y éste se escribe en la posición que le corresponde en la FAT. Usamos el nombre de FAT para referirnos a la FAT1 pues el sistema no escribe byte alguno en la FAT2.

Se debe recordar que los primeros dos clusters de la región de datos son reservados (cluster 0 y cluster 1) y cada uno tiene su entrada en la FAT. Esto debe tomarse en cuenta para rescatar el valor de sus entradas antes de escribir en la tabla.

A continuación se muestra el proceso descrito con extractos de código relevantes.

La información de la entrada de `FILE1` en el directorio se ordena utilizando la estructura `DirEnt4alloc`.

```
typedef struct {

       WORD * FrstClusHI ; // 2 bytes

       WORD * FrstClusLO ; // 2 bytes

       DWORD * Size ; // 4 bytes

       DWORD * FrstClusNum ; // 4 bytes : get together
                             // FrstClusHI and FrstClusLO

} DirEnt4alloc ; // data Directory Entry for allocation
```

Del *buffer* que se escribió en el primer sector del directorio, se capturan los bytes que corresponden a la parte alta y baja del primer número de cluster de `FILE1` y se forma `FrstClusNumb`. Del *buffer* también se capturan los bytes que corresponden al tamaño del archivo. Esta información se guarda en la estructura `FileAlloc` como se muestra.

```
BYTE * buf_ptr = & buffer [ 0 ] ;
DirEnt4alloc FileAlloc ;

FileAlloc.FrstClusHI = ( WORD * ) & buf_ptr [ 52 ] ;
FileAlloc.FrstClusLO = ( WORD * ) & buf_ptr [ 58 ] ;

                    // 0x00000000 = 0L
FrstClusNumb =
 (  (  * ( FileAlloc.FrstClusHI ) + 0L  ) << 16  ) +
                           (  * ( FileAlloc.FrstClusLO )  ) ;

FileAlloc.FrstClusNum = & FrstClusNumb ;

FileAlloc.Size = ( DWORD * ) & buf_ptr [ 60 ] ;
```

Lo que sigue es determinar la cantidad de clusters que se le van a asignar al archivo. Para ello se programa la lógica que calcula cuántos

sectores utiliza el archivo de acuerdo con su tamaño, y a partir de esto se determina el total de clusters del archivo. Este valor se guarda en la variable `CntOfClus`. Las operaciones que se utilizan en este proceso se muestran a continuación.

```
DWORD CntOfSecs , CntOfClus , SpareSecs ;
WORD SpareBytes ;

CntOfSecs = * ( FileAlloc.Size ) / * ( BytesPerSec.value ) ;

SpareBytes =
(  * ( FileAlloc.Size )  ) %  (  * ( BytesPerSec.value )  ) ;

CntOfClus = CntOfSecs / (  * ( SecsPerClus.value )  ) ;

SpareSecs = CntOfSecs % (  * ( SecsPerClus.value )  ) ;
```

Lo que resta es escribir en la FAT. Para hacerlo, se necesita conocer la entrada y su valor para cada cluster. La función `ClusNum2FAT` lo determina y regresa las variables `ThisFATSecNum` y `ThisFATEntOffset` que guardan, respectivamente, el número del sector donde se encuentra la entrada de ese cluster en la FAT1 y el número de la entrada de ese cluster dentro de la FAT1 (posición dentro de la tabla). La función mediante la cual se realiza esto se muestra a continuación.

```
BYTE ThisFATEntOffset ;
WORD * clusnum ;
DWORD * ClusNum , FATOffset , ThisFATSecNum ;

// Given any valid cluster number N, ClusNum2FAT computes
// where in the FAT1 is the entry for that cluster number

char ClusNum2FAT ( BYTE * N , char fattype )

{
       if ( fattype == FAT16 ) {

               clusnum = ( WORD * ) N ;
               ClusNum = ( DWORD * ) clusnum ;
```

```
                FATOffset = ( * ClusNum ) * 2 ; //  2 bytes
                                                //  per entry
                                                //  (in any
                                                //  sector of
                                                //  the FAT)
                                                //  for FAT16
        }
        else {

                ClusNum = ( DWORD * ) N ;
                FATOffset = ( * ClusNum ) * 4 ; //  4 bytes
                                                //  per entry
                                                //  (in any
                                                //  sector of
                                                //  the FAT)
                                                //  for FAT32
        }

                        /* FATBegin */
ThisFATSecNum = * ( RsvdSecCnt.value ) +
                          ( FATOffset / * ( BytesPerSec.value ) );

ThisFATEntOffset = FATOffset % (  * ( BytesPerSec.value )  ) ;

}
```

Esta función se aplica al número del primer cluster de FILE1 para saber en dónde se empezarán a escribir los valores de las entradas de los clusters del archivo.

```
ClusNum2FAT ( ( BYTE * ) ( FileAlloc.FrstClusNum ) , fattype ) ;

// output : ThisFATSecNum and ThisFATEntOffset
```

Como se había comentado, se debe capturar el valor de las entradas de los clusters 0 y 1 en la FAT. Este procedimiento es similar al que se implementó para capturar los bytes de la entrada del archivo de la etiqueta del volumen. Para conocer el número del sector que debe leerse,

se aplica la función `ClusNum2FAT` al cluster 0 y al cluster 1. Las entradas de ambos clusters se encuentran en el mismo sector, el que se lee para capturar el valor de las entradas.

Debe tomarse en cuenta que la cantidad de clusters del archivo determina el número de entradas que se escriben en la FAT. Así, es posible que el archivo necesite más de un sector de la FAT para escribir el valor de todas sus entradas, y si esto aplica, debe determinarse cuántos sectores de la FAT utiliza el archivo. El sistema de registro de datos implementa este análisis llevando la cuenta de las entradas por sector de la FAT, y decide, para el archivo especificado en la entrada uno del directorio, si más de un sector es necesario; si más de uno se requiere, calcula cuántos sectores más se necesitan.

Los valores de las entradas de los clusters del archivo son números consecutivos de clusters, de 2 bytes para el sistema FAT16, y de 4 bytes para el sistema FAT32.

Con los valores capturados de las entradas de los clusters 0 y 1, y conociendo la posición en la FAT donde se van a empezar a escribir los valores de las entradas, se procede a escribir en la FAT. A cada sector de la FAT se le escribe el *buffer* con los valores de las entradas de los clusters del archivo que le corresponden a cada sector. El *buffer* se reusa para escribir cada sector de la FAT, y el valor y la posición de cada byte dentro del *buffer* se genera para cada sector que se escribe; se debe tomar en cuenta que cada valor de entrada debe tener el fomato *little endian*.

Para generar el *buffer* se utiliza la estructura `Next` de tipo `NextValues`, que se define de la siguiente forma.

```
typedef struct {

       DWORD * ClusNum ;

} NextValues ;

NextValues Next ;
```

El valor de `Next.ClusNum` se va modificando para generar cada byte del valor de cada entrada y se inicializa con el número del cluster de inicio del archivo mas uno, como se muestra.

```
FrstClusNumPLUS1 = * ( FileAlloc.FrstClusNum ) + 1  ;

Next.ClusNum = & FrstClusNumPLUS1 ;
```

El sistema de registro de datos implementa la lógica para generar el *buffer* que se va a escribir en cada sector de la FAT.

Las siguientes lineas de código tienen la intención de dar una idea de la forma en que esta lógica se programó.

```
#define FAT32value 4 // 4-byte value in the FAT per
                     // entry for FAT32
#define FAT16value 2 // 2-byte value in the FAT per
                     // entry for FAT16
              .
              .
              .

if ( fattype == FAT32 ) {

    FATSizeOfVal = FAT32value ;
}

else {

    FATSizeOfVal = FAT16value ;
}

for ( j = 1 ; j <= CntOfClus ; j ++ ) {

    offset = ThisFATEntOffset + (  ( j - 1 ) * FATSizeOfVal  ) ;

    if ( j != CntOfClus ) {

       for ( i = 0 ; i < FATSizeOfVal ; i ++ ) {

               buffer [ offset + i ] =
               * (  ( BYTE * ) ( Next.ClusNum ) + i  ) ;
       }
```

```
    * ( Next.ClusNum ) += 1 ;

    }

    else {  // last cluster of the file

       if ( FATSizeOfVal == FAT32value ) { // 4 bytes

               * ( Next.ClusNum ) = 0x0FFFFFFF;

       }

       else { // FAT16value : 2 bytes

               * ( Next.ClusNum ) = 0xFFFF;

       }

       for ( i = 0 ; i < FATSizeOfVal ; i ++ ) {

               buffer [ offset + i ] =
                * (  ( BYTE * ) ( Next.ClusNum ) + i  ) ;

       }

    }

}
```

En el código anterior se puede notar que el valor de la última entrada del archivo tiene un valor especial. Este valor le indica al sistema operativo que es la última entrada del archivo en la FAT.

Los valores que se pueden escribir en la FAT se muestran en la tabla 3.9.

Para el archivo `FILE1` se propuso el cluster 3 como su cluster de inicio, pero se pudo haber propuesto cualquier otro. En el ejemplo que estamos desarrollando, el archivo se crea en una memoria SDHC, por lo

FAT16	FAT32	DESCRIPCIÓN
0x0000	0x00000000	Cluster disponible para usarse
0x0001	0x00000001	Cluster reservado
0x0002 – 0xFFEF	0x00000002 – 0x0FFFFFEF	Apunta al siguiente cluster del archivo
0xFFF0 – 0xFFF6	0x0FFFFFF0 – 0x0FFFFFF6	Cluster reservado
0xFFF7	0x0FFFFFF7	Cluster dañado
0xFFF8 – 0xFFFF	0x0FFFFFF8 – 0x0FFFFFFF	Último cluster del archivo

Tabla 3.9. Valores permitidos en la FAT.

que el sistema de archivos es FAT32. En este sistema, el directorio empieza siempre en el cluster 2 de la región de datos. Esta es la razón por la que el cluster 3 se eligió como el cluster de inicio de `FILE1`.

El resultado de escribir en la FAT la secuencia de clusters del archivo `FILE1` se muestra en la figura 3.38. En la figura, el cursor indica la posición de inicio de la primera entrada del archivo. El valor de esta entrada es 0x00000004 y significa que el cluster 4 es el cluster que sigue al cluster 3 del archivo. Se debe notar que las primeras tres entradas corresponden a las de los clusters 0, 1 y 2.

La figura 3.39 muestra el último sector escrito en la FAT y el cursor indica la posición de la última entrada del archivo, con valor 0x0FFFFFFF.

Con el contenedor del archivo listo, lo que resta es escribir los bytes de datos en los clusters correspondientes, y así se inicia el último paso en la creación del archivo.

Escritura del contenido del archivo

La información que al usuario le interesa almacenar es a lo que nos referimos como contenido del archivo, y esta información se representa como

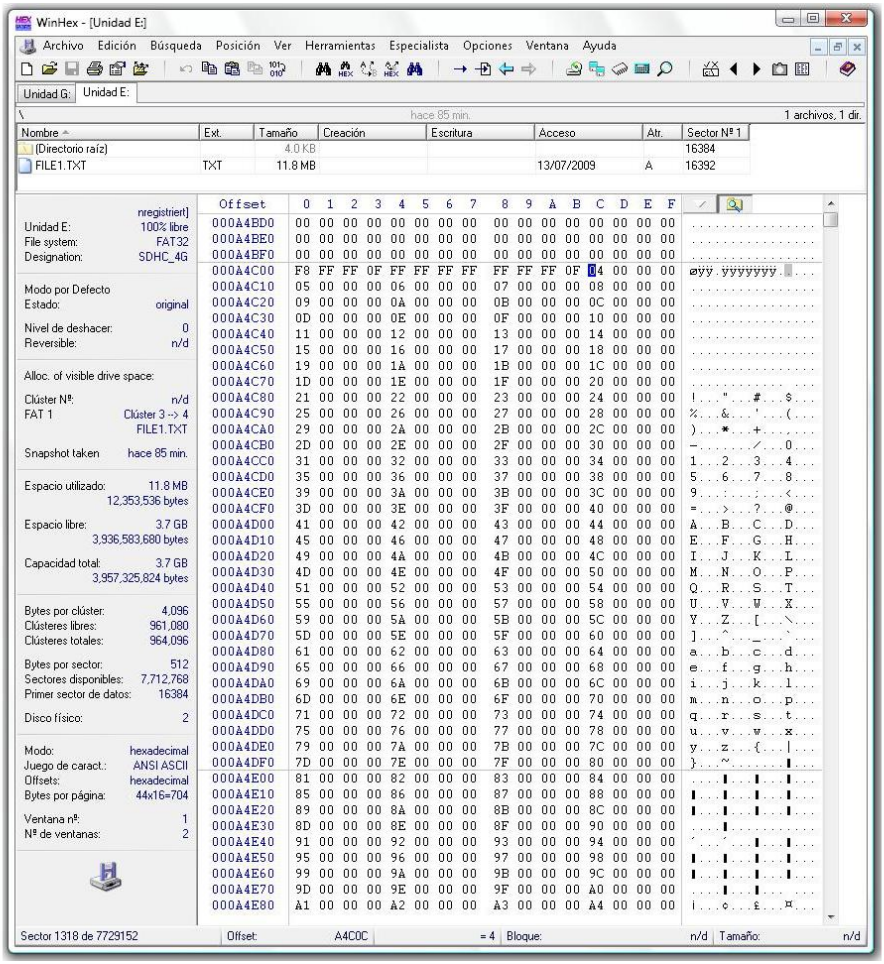


Fig. 3.38. Primer sector escrito en la FAT para el archivo `FILE1`.

un arreglo de bytes de datos para el sistema de registro. Los bytes de datos aparecerán como la información contenida en el archivo siempre que estén escritos en cada sector de cada cluster asignado al archivo.

De acuerdo con lo anterior, para escribir lo bytes de datos se necesita conocer el primero y último sector del archivo, lo que se determina a partir de su primer y último cluster. Este cálculo lo hace el sistema de registro de datos con la función `SetFILEsBoundaries`, en cuyo argumento se especifica el primer número de cluster y la cantidad de clusters del archivo, como se muestra en su prototipo de función.

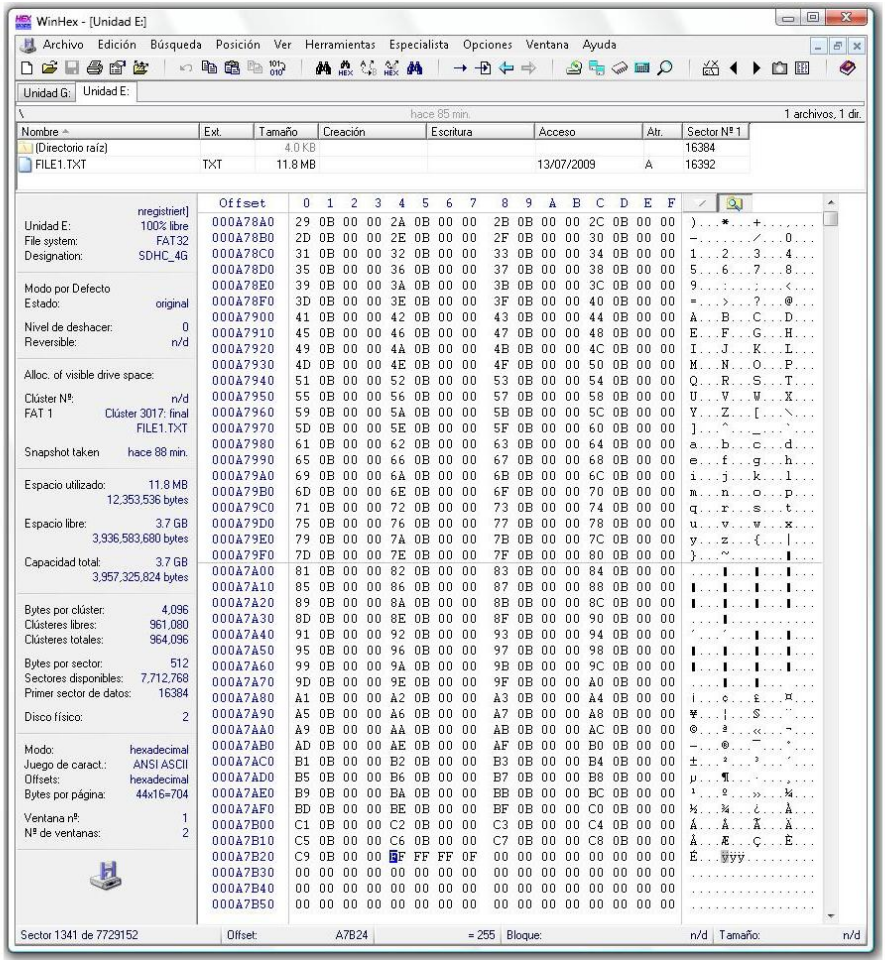


Fig. 3.39. Último sector escrito en la FAT para el archivo FILE1.

```
SetFILEsBoundaries ( DWORD * frstClusNum , DWORD * CntofClus ) ;
// output: FILE_FirstSecNum and FILE_LastSecNum
```

Las variables de salida de esta función son FILE_FirstSecNum y FILE_LastSecNum, ambas de cuatro bytes. FILE_FirstSecNum guarda el número del primer sector del archivo y FILE_LastSecNum el número del último sector. La función SetFILEsBoundaries aplicada a la información de FILE1 se muestra a continuación.

```
SetFILEsBoundaries ( & FrstClusNumb , & CntOfClus ) ;
```

Conociendo los sectores límite del archivo, se procede a escribir en ellos y para esto se utilizan la función `FillClusNum`. El sistema de registro de datos indicará a través de esta función si la escritura de los datos fue exitosa, enviando 0x01 a la terminal virtual o 0x00 en caso contrario. La función `FillClusNum` se muestra a continuación.

```
void FillClusNum ( void ) {

GetArgRWOneSec ( & FILE_FirstSecNum , Sector_Num , 'M' ) ;

        if (  WriteMultBlocks (  & FILE_FirstSecNum ,
                                 & FILE_LastSecNum ,
                                 WriteTestMultiple  )  )
       {
                Display ( TRUE ) ; // 0x01
       }

       else {
                Display( FALSE ) ; // 0x00
       }

}
```

La función `GetArgRWOneSec` forma el argumento del comando de escritura; la `'M'` indica que se escribirán múltiples sectores en la tarjeta de memoria. La función `WriteMultBlocks` lleva a cabo la transmisión del comando y los bytes de datos.

La constante de enumeración `WriteTestMultiple`, dentro del argumento de la función `WriteMultBlocks`, forma parte de la enumeración `SDB` definida con anterioridad. El valor de esta constante le indica a la función `WriteMultBlocks` el *buffer* que debe generar y enviar a la tarjeta de memoria. Este *buffer* contiene los bytes de datos.

Para el sistema de registro, el valor de los bytes de datos del archivo no es importante. Así, el sistema genera los bytes del *buffer* que se escriben en los sectores del archivo `FILE1`, como se muestra. Este *buffer* se escribe repetidamente en cada sector de `FILE1`.

```
for ( i = 0 ; i <= 511 ; i ++ ) {

        buffer [ i ] = 'a' ;
        buffer [ i + 1 ] = 'n' ;
        buffer [ i + 2 ] = 'a' ;
        buffer [ i + 3 ] = ':' ;
        buffer [ i + 4 ] = ')' ;
        i += 4 ;

}
```

Con la escritura exitosa de los bytes de datos en los sectores de `FILE1`, se concluye la creación del archivo.

El resultado del proceso descrito se puede ver en la figura 3.40. En la figura se muestran únicamente el primer sector y una parte del siguiente del archivo `FILE1`.

Como se ha visto en esta sección, el código que se programa en el microcontrolador implementa el registro de datos en las tarjetas de la manera deseada. Este código utiliza espacio de memoria dentro del microcontrolador, y para dar una idea de la extensión de este espacio, se muestra la figura 3.41. La figura indica el espacio de memoria de programa y memoria dato utilizada en el microcontrolador.

A lo largo de esta sección se ha cubierto la idea general de la implementación del sistema de registro de datos, haciendo énfasis en el *firmware* del microcontrolador. En la siguiente sección se completa este tema presentando la implementación del *hardware* del sistema de registro de datos.

3.2.4. Circuito del sistema

El circuito del sistema de registro de datos tiene como núcleo el PIC18LF452. A él se conecta la memoria SD CARD a través del *bus* SPI y la PC a través del transceptor MAX 3222. La conexión del transceptor es la configuración típica para este circuito integrado de acuerdo con sus hojas de especificaciones.

Como se mencionó en el apartado "Secuencia de inicio", la tarjeta de memoria debe ser energizada por el microcontrolador y cumplir con la

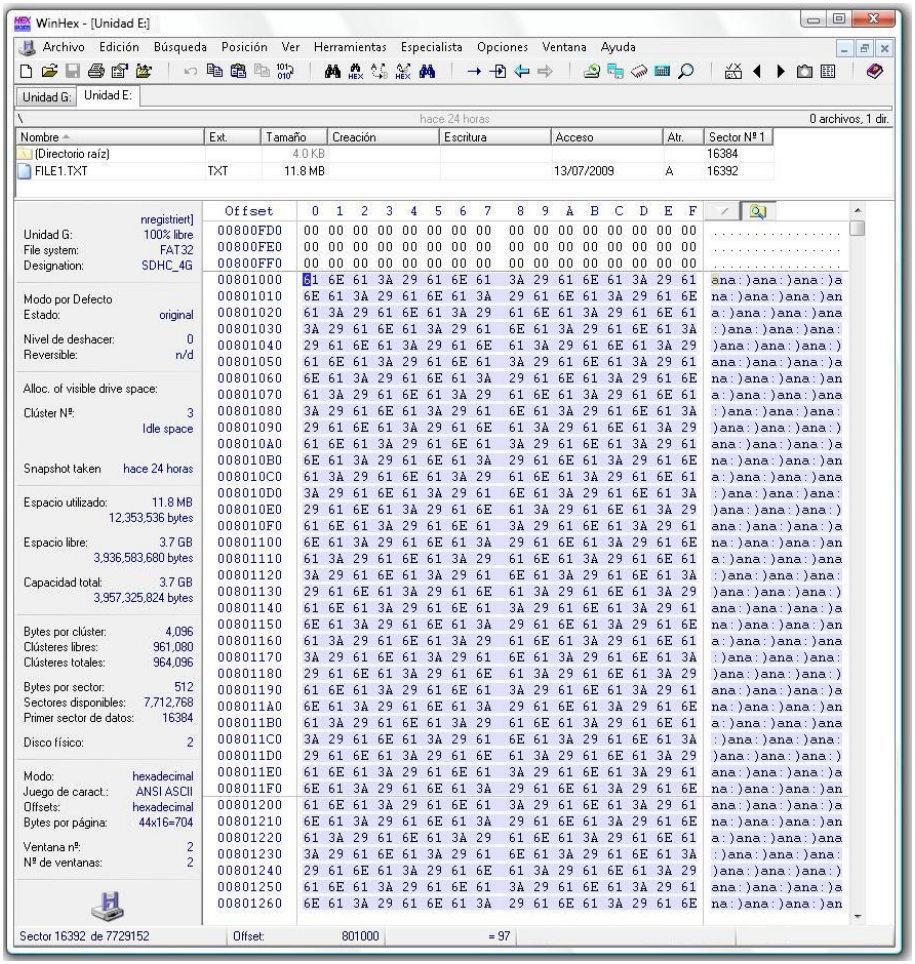


Fig. 3.40. Primeros sectores del contenido de `FILE1`.

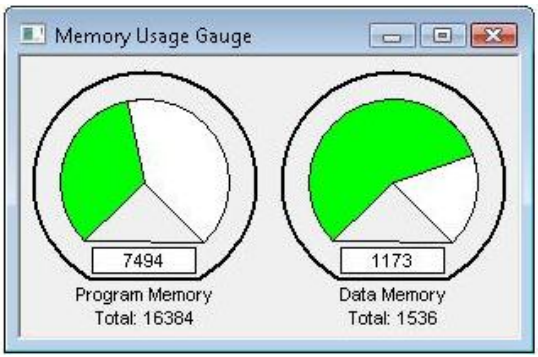


Fig. 3.41. Espacio de memoria usado en el PIC.

especificación del tiempo de encendido. Se descubrió empíricamente que este tiempo se satisface al encender la tarjeta a través de un transistor que funciona como interruptor y que está controlado por el PIC. Cualquier transistor de propósito general se puede utilizar para encender a la tarjeta. En el sistema se utilizó el transistor BC557 y se eligió una resistor de 620[Ω] para polarizarlo. La base del transistor se conectó a través del resistor de 620[Ω] a la terminal cuatro del puerto B del microcontrolador y esta terminal se configuró como salida; se pudo haber selecionado otra terminal del puerto. El emisor se conectó a la tensión de alimentación del sistema y el colector a la terminal V^+ de la SD CARD. El microcontrolador enciende al transistor poniendo en '0' su terminal conectada a la base del transistor y con ello se enciende la SD CARD.

Adicionalmente dos terminales del microcontrolador se utilizaron para indicar el estatus del sistema y a cada una se le conectó un LED. Un LED ultrabrillante indica el encendido del sistema de registro de datos y un LED miniatura indica la presencia de algún error en la comunicación. El LED ultrabrillante se eligió sobre un LED convencional porque al ser encendido el observador lo detecta inmediatamente, en comparación con los LEDs convencionales. El LED ultrabrillante se enciende con un transistor BC557 de forma similar como se encendió la SD CARD con el transistor y una resistor de 56[Ω] se utilizó para limitar el paso de corriente a través de él. El LED miniatura, o LED de error, ayudó a depurar el *firmware* del microcontrolador. Este tipo de LED puede conectarse directamente a la terminal del microcontrolador sin dañarse y por esta razón se eligió. La conexión del LED miniatura se omite en el circuito del sistema.

En la figura 3.42 se muestra el circuito del sistema de registro de datos y en la figura 3.43 se observa una fotografía del sistema.

De esta forma se concluye la implementación del sistema de registro de datos. En el capítulo siguiente se muestran algunas pruebas realizadas al sistema completo.

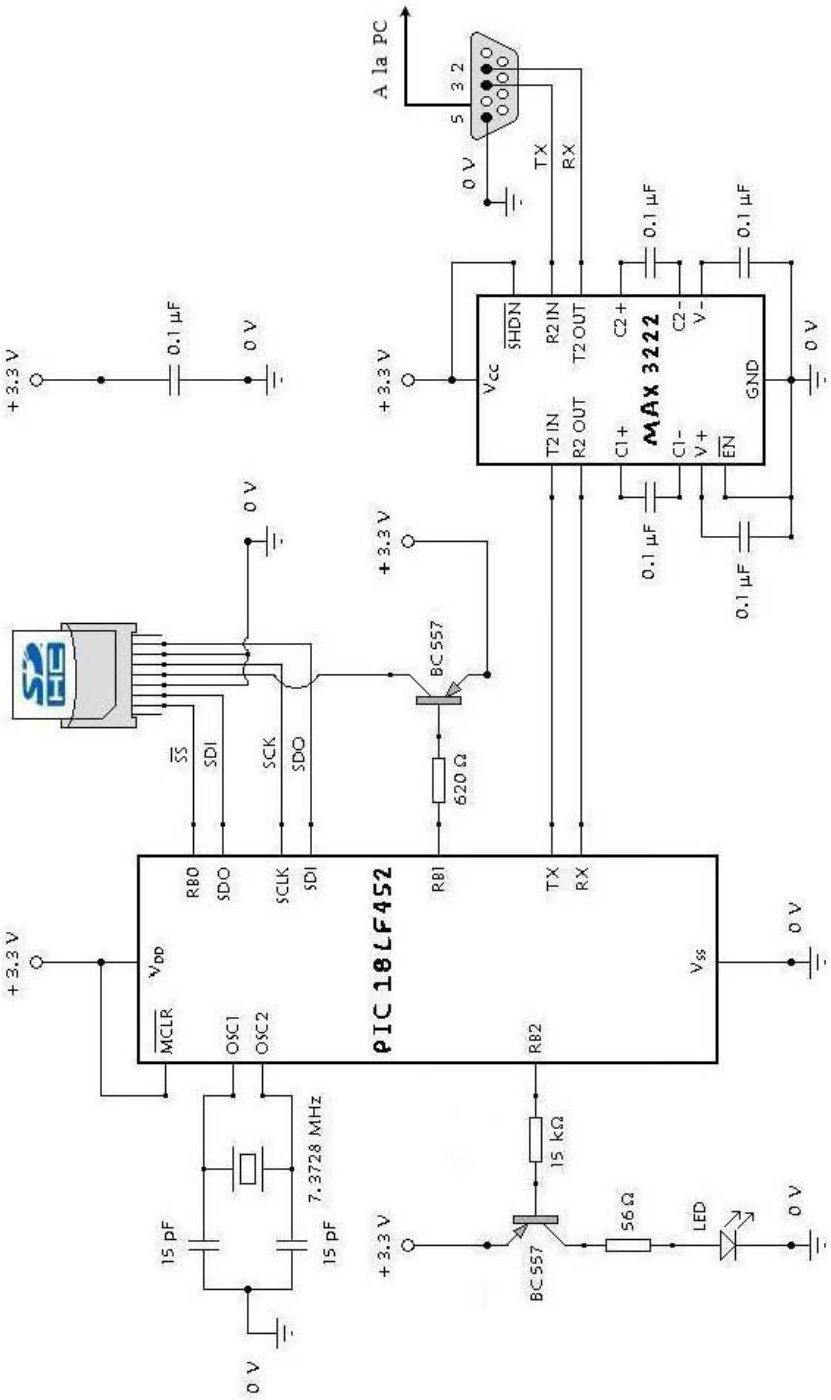


Fig. 3.42. Circuito del sistema de registro de datos.

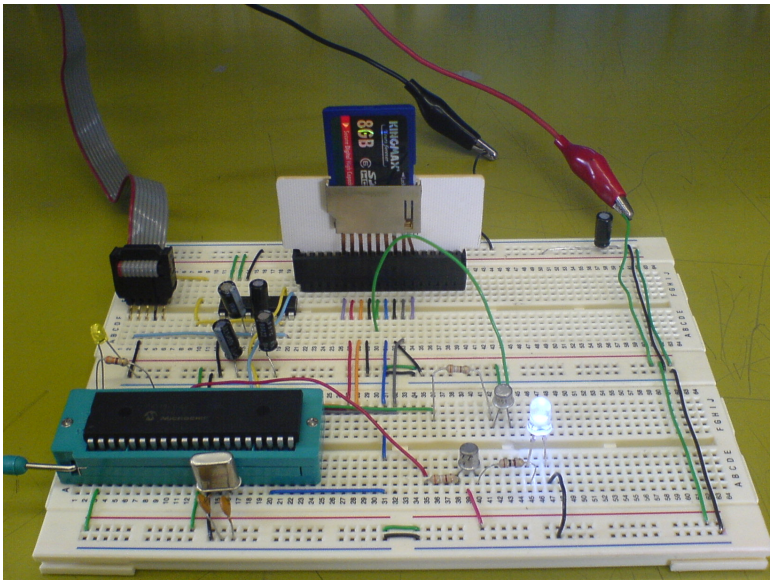


Fig. 3.43. Foto del sistema de registro de datos.

4

En este capítulo se explican las pruebas realizadas al sistema y los resultados obtenidos de ellas. Para grabar información en ambos tipos de memorias SD CARD, el sistema de registro de datos se sometió a pruebas durante su implementación y a pruebas finales para comprobar el registro correcto de los datos.

4.1. Pruebas al sistema de registro de datos

Como se ha comprobado en los capítulos anteriores, la implementación del sistema de registro consta en su mayor parte de la programación del código del microcontrolador. Al código se le hicieron pruebas conforme se fue generando para verificar la programación correcta de la lógica, de la comunicación con la tarjeta y en general para verificar la operación correcta del sistema en cada etapa del desarrollo. Herramientas como WinHex y la terminal RealTerm se utilizaron para conocer el resultado de estas pruebas, y del análisis de sus resultados, se hicieron ajustes al código del microcontrolador.

Las pruebas que se presentan muestran el grabado de datos en las tarjetas de memoria utilizando el sistema de registro desarrollado. Con las pruebas presentadas se verifica:

- Que el sistema de archivos se haya implementado en una memoria SD CARD.
- El grabado de un archivo de datos en la memoria SD CARD.

Y estos objetivos se comprueban utilizando una computadora que soporte los sistemas de archivos FAT.

Antes de iniciar cada prueba, la tarjeta que se va a conectar al sistema de registro debe estar formateada. Cada prueba inicia especificando en el código las características del archivo a crearse (información del directorio). El *firmware* actualizado se programa en el microcontrolador, y a continuación se energiza el sistema de registro. Terminada la ejecución del código la operación del sistema ha finalizado y la tarjeta de memoria se retira. Esta tarjeta se inserta en un lector de memorias SD CARD conectado a una computadora y se observa la interpretación que el sistema operativo le da a la información que lee de la tarjeta. Con esto se obtiene el resultado de la prueba.

A continuación se muestran los resultados de cuatro pruebas realizadas al sistema de registro de datos. La primera prueba presenta algunos errores comúnes en la implementación del sistema de archivos. La segunda prueba muestra a detalle el resultado para un archivo de 1000 bytes en una memoria SD y en una SDHC. Y en las últimas dos pruebas se muestra el resultado para un archivo de 3102009 bytes en una memoria SD y para el archivo de 12345678 bytes explicado en el capítulo 3.

4.1.1. Prueba 1 : Error en la implementación

En el proceso de implementar el sistema de archivos, es común escribir bytes en regiones de memoria incorrectas o escribir valores equivocados en espacios incorrectos. Estos son los principales errores por los que la implementación del sistema de archivos falla. La SD CARD permite el grabado de información en cualquier posición de memoria, y es nuestra responsabilidad escribir esta información en los lugares correctos, para que, cuando la tarjeta de memoria se conecte a una computadora, el sistema operativo pueda leerla de forma adecuada.

En la figura 4.1 se muestra el resultado de una escritura incorrecta en las regiones del sistema de archivos.

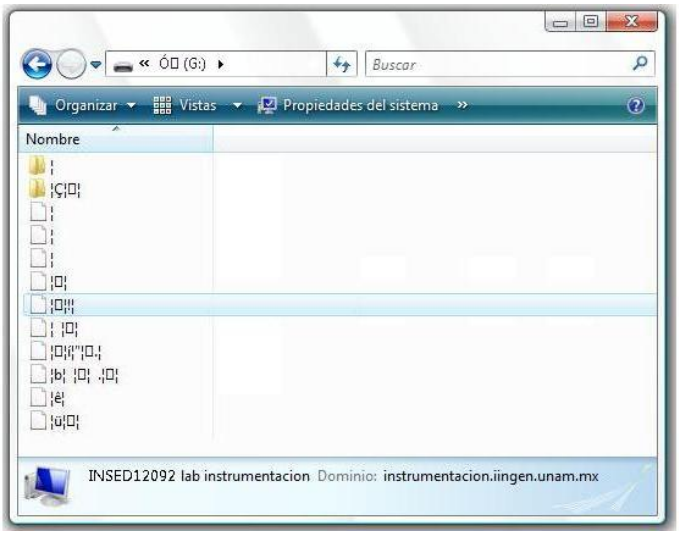


Fig. 4.1. Programación incorrecta del sistema de archivos.

Al formatear esta tarjeta, el sistema operativo detecta el sistema de archivos presente en ella como desconocido y lo identifica como RAW (sin sistema de archivos reconocible), como lo muestra la figura 4.2.

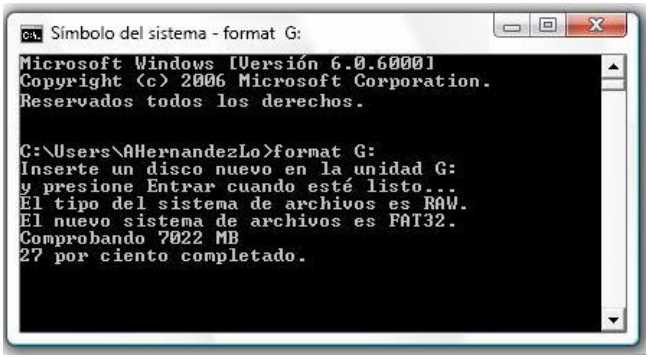


Fig. 4.2. Sistema de archivos RAW.

Un error común ocurre al escribir incorrectamente los valores de las entradas en la FAT. En la figura 4.3 se muestra el resultado de un error de este tipo, cuando en la FAT no se ha escrito el valor de la entrada del último cluster asignado al archivo. Como se observa en la figura, el archivo se crea pero su contenido no aparece.

Fig. 4.3. Resultado en Windows de un error de escritura en la FAT.

El peor caso de una implementación incorrecta en la tarjeta provoca un error fatal en el sistema operativo, lo que resulta en el bloqueo del ambiente Windows. Cuando esto ocurre un *reset* es necesario para reestablecer las funciones de la computadora. Implementaciones incorrectas del sistema de archivos que pueden provocar esto son: reescribir el *boot sector*, escribir en regiones reservadas y escribir valores no permitidos o incorrectos en la FAT.

4.1.2. Prueba 2 : Archivo de 1000 bytes

En esta prueba se genera un archivo de 1000 bytes en una memoria SD y en una SDHC. El resultado de la prueba se muestra de dos formas:

utilizando las herramientas que proporciona el ambiente Windows y utilizando el WinHex para ver los bytes escritos en las regiones del sistema de archivos de cada tarjeta de memoria.

Uso del ambiente Windows Antes de conectar cada tarjeta al sistema de registro, se formatean desde el ambiente Windows y para ello el sistema operativo utiliza por *default* los valores predeterminados para el sistema de archivos que le corresponde a cada tarjeta, como se muestra en la figura 4.4.

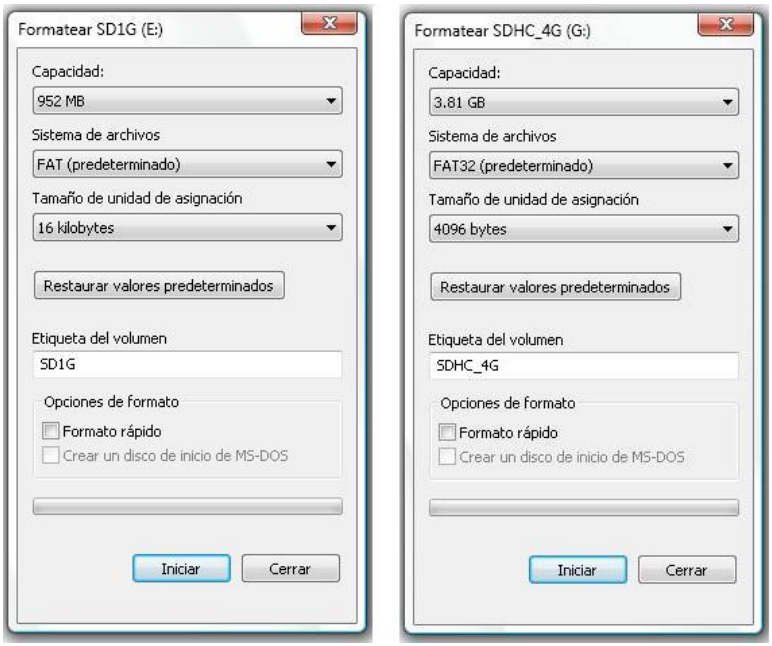


Fig. 4.4. Formateo de dos tarjetas SD CARD.

Como se observa en la figura, la primera SD CARD es de 1[GB] y la segunda de 4[GB]. El "tamaño de unidad de asignación" se refiere al tamaño de cluster y este valor se puede modificar. En esta prueba se mantiene el valor predeterminado del tamaño de unidad de asignación, de forma que la memoria identificada como SD1G tendrá clusters de 16[kB] y la memoria SDHC_4G tendrá clusters de 4[kB].

Cada tarjeta formateada queda lista para conectarse al sistema de registro. A continuación, se especifica el tamaño del archivo en el código del microcontrolador y para ello se utiliza en ambiente de desarrollo

MPLAB IDE como se muestra en la figura 4.5. En la figura, la flecha indica el campo en el que se especifica el tamaño del archivo; en todas las pruebas realizadas se mantuvo el nombre del archivo de texto generado como `FILE1` y el cluster 3 como su cluster de inicio, pero estos pueden modificarse.

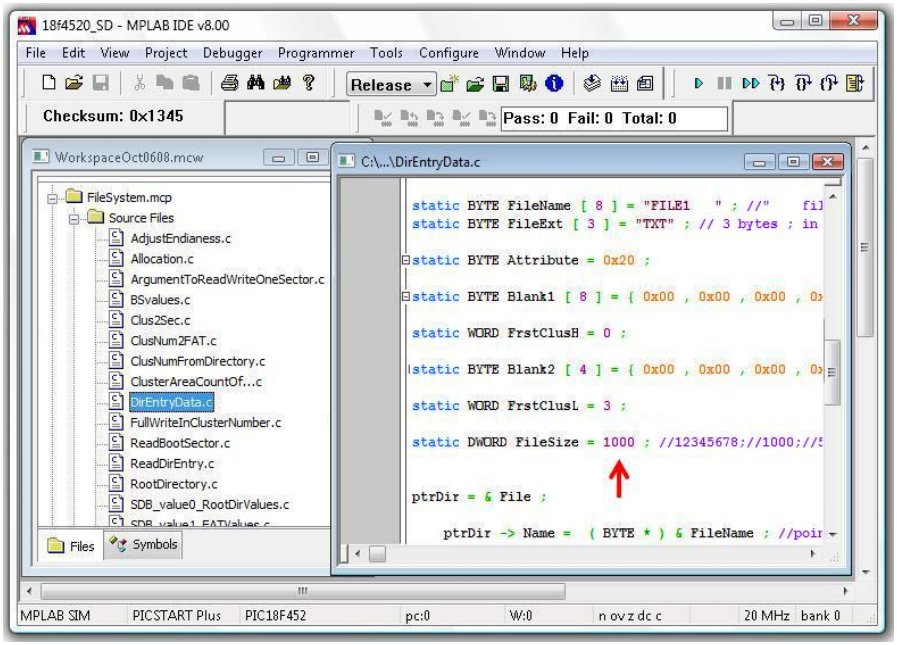


Fig. 4.5. Especificación del tamaño del archivo en el código.

Una vez realizado el proceso descrito, se ejecuta el código del microcontrolador en el sistema de registro con cada tarjeta formateada. Terminada la ejecución, las tarjetas se conectan a la computadora y el resultado se muestra en la figura 4.6.

Para examinar las propiedades de cada memoria, desde su menú se selecciona "Propiedades" como lo muestra la figura 4.7 para la memoria de 1[GB].

La ventana de propiedades para cada tarjeta se muestra en la figura 4.8. Como se observa en esta figura, el espacio de disco usado por la tarjeta de 1[GB] es un cluster, aún cuando el archivo creado es de 1000 bytes, pues la unidad más pequeña de espacio de almacenamiento que se le puede asignar a un archivo es un cluster. En la ventana de propiedades de la tarjeta de 4[GB] se observa que el espacio de disco usado son 8[kB],

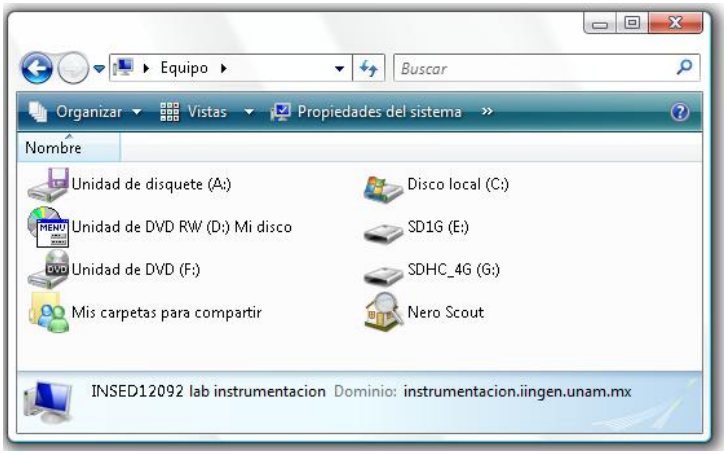


Fig. 4.6. Unidades de memoria SD1G y SDHC_4G.

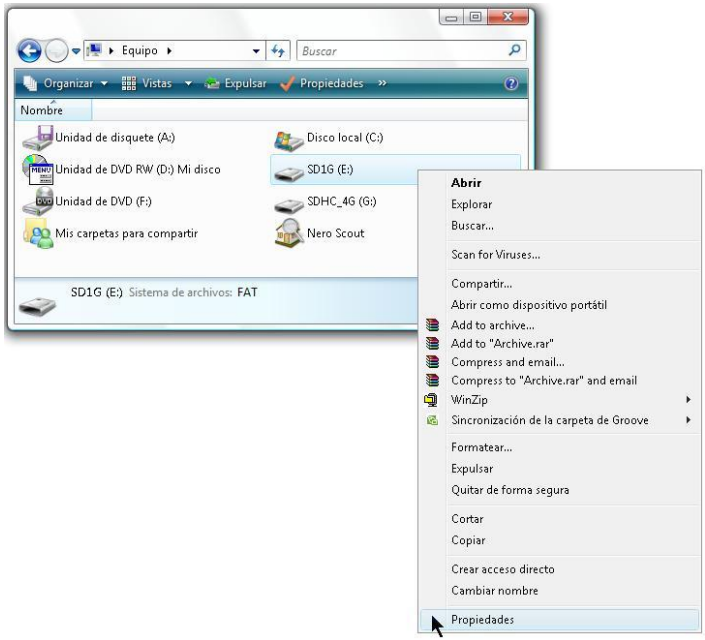
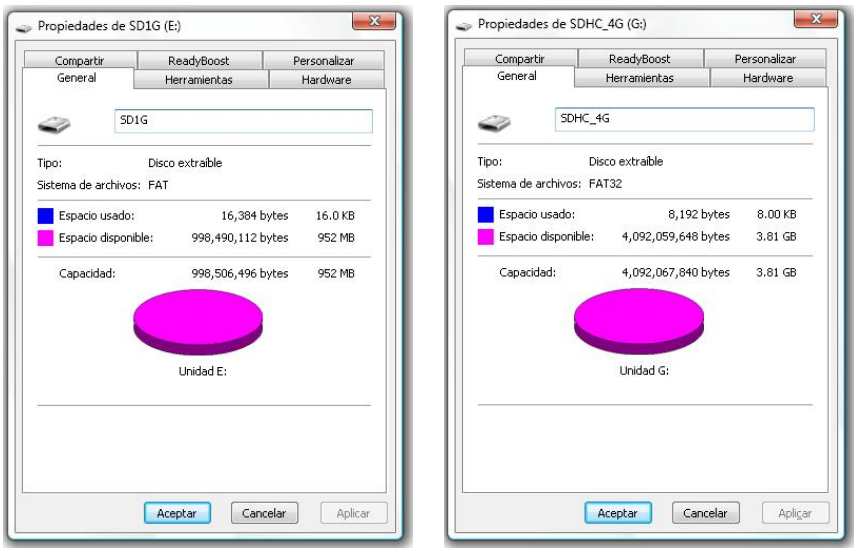


Fig. 4.7. Menú de la memoria SD1G.

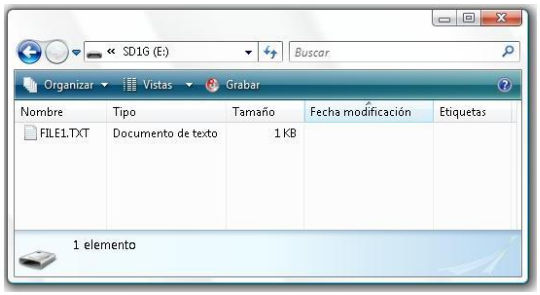
es decir, dos clusters, pues uno corresponde al del directorio y el otro es el cluster del archivo.

El contenido de las unidades SD1G y SDHC_4G se muestra en la figura 4.9. Se observa que el archivo creado se representa por el icono para un archivo de texto.



SD1G. SDHC_4G.

Fig. 4.8. Propiedades de las memorias SD1G y SDHC_4G.



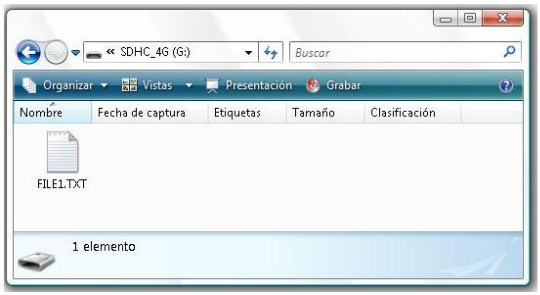


Fig. 4.9. Archivo `FILE1` en las memorias SD1G y SDHC_4G.

Para ver las propiedades del archivo, desde su menú se selecciona "Propiedades" como lo muestra la figura 4.10 para la memoria de 1[GB].

La ventana de propiedades para el archivo de cada tarjeta se muestra en la figura 4.11. En ella se observa el tamaño del archivo y el tamaño en disco utilizado en cada tarjeta de memoria.

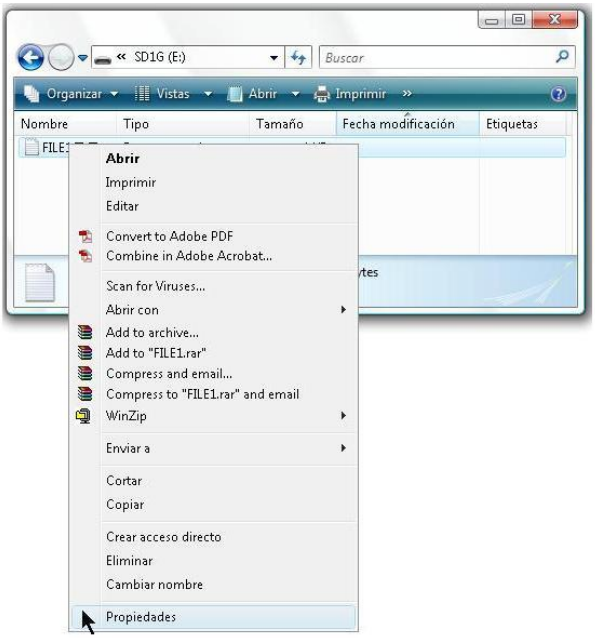


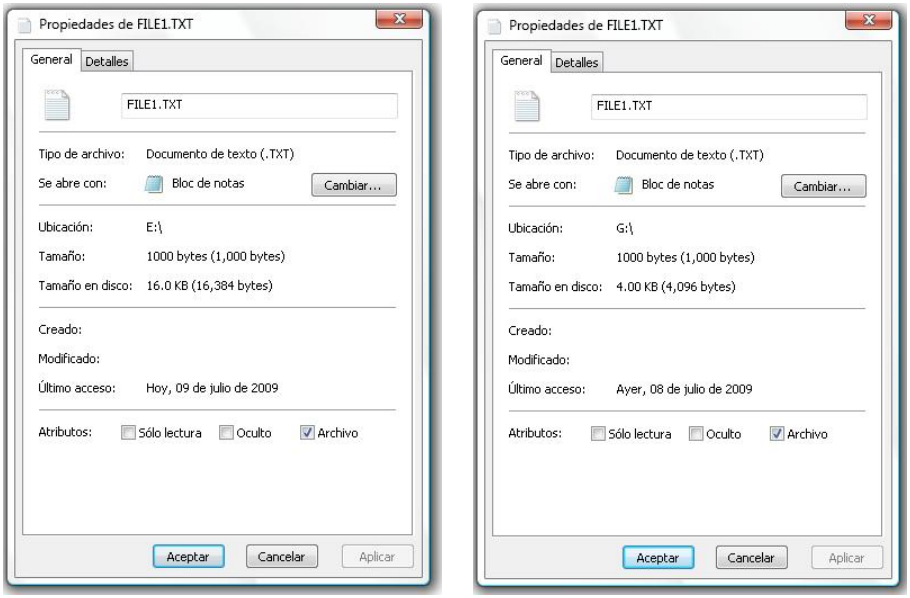
Fig. 4.10. Menú de `FILE1` en la memoria SD1G.

En la figura 4.11 se comprueba la asignación de un cluster completo al archivo de 1000 bytes y se nota que con un tamaño de cluster pequeño habrá menos espacio en disco desperdiciado.

Lo que resta es verificar que el archivo `FILE1` contenga los datos almacenados. La figura 4.12 muestra el contenido de este archivo.

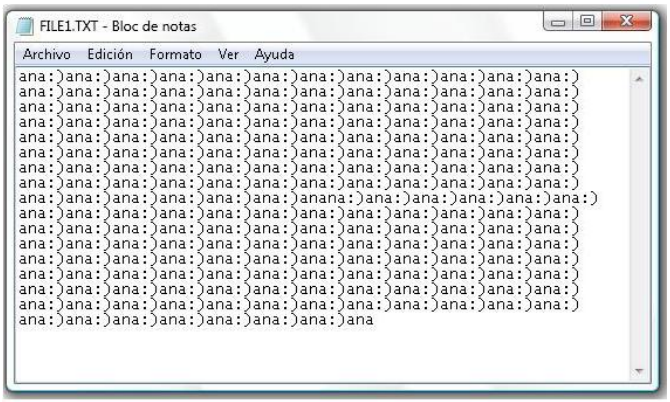
Con MS Word se puede conocer fácilmente la cantidad de páginas de `FILE1`, abriendo el archivo con este procesador de texto y utilizando su contador de palabras. El contador indica también el número de caracteres del documento, como se muestra en la figura 4.13.

Inspección de las regiones Resulta también de interés conocer cómo quedó escrita la información en el interior de cada tarjeta de memoria



SD1G. SDHC_4G.

Fig. 4.11. Propiedades de `FILE1`.


FILE1.TXT - Bloc de notas
Archivo Edición Formato Ver Ayuda
ana:)ana:)ana:)ana:)ana:)ana:)ana:)ana:)ana:)ana:)ana:)ana:)ana:)

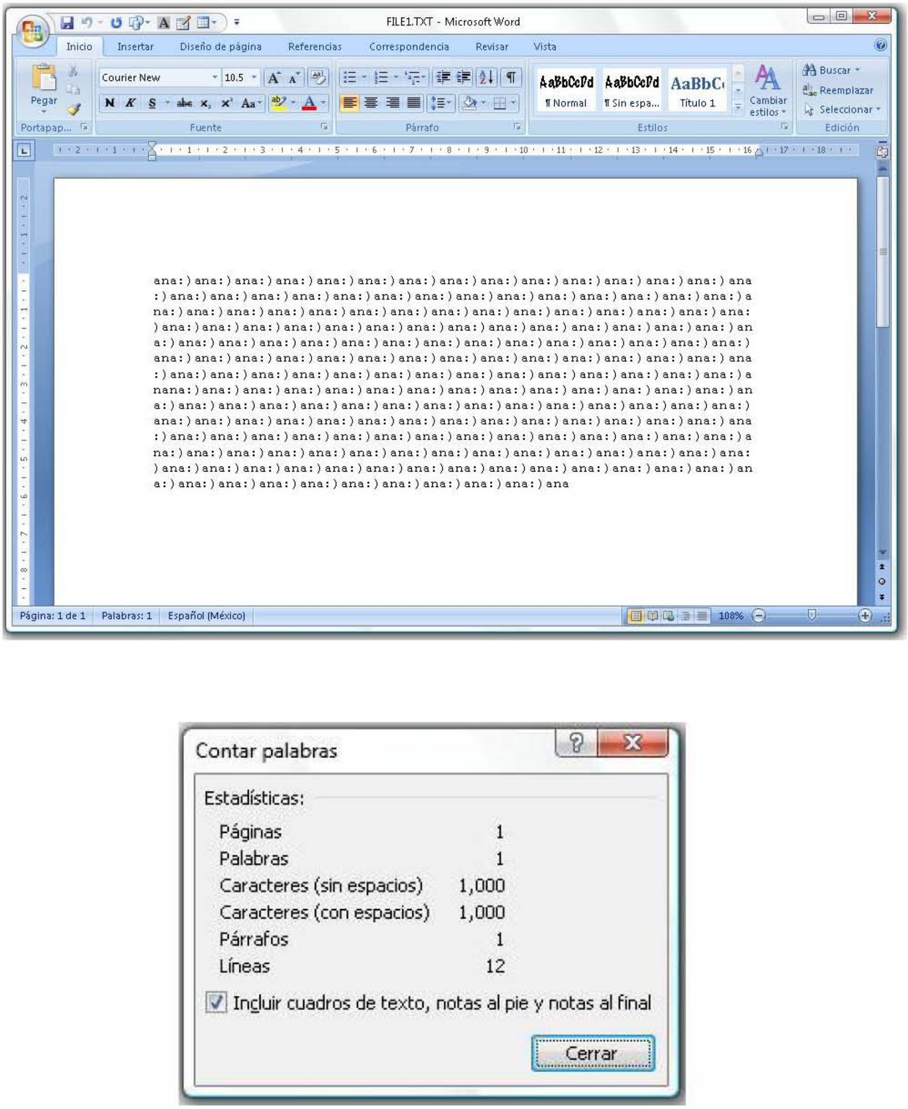


Fig. 4.13. Contador de palabras en MS Word para `FILE1`.

y utilizando el WinHex podemos examinar cada región del sistema de archivos.

La región del directorio y de la FAT para las tarjetas de memoria de 1[GB] y 4[GB] se muestran de la figura 4.14 a la figura 4.17.

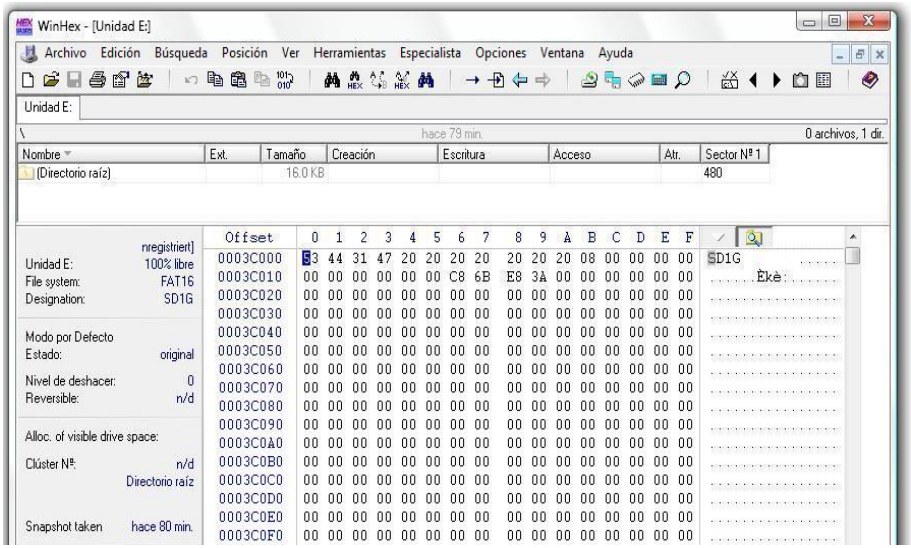
En las figuras 4.14 y 4.15 se observa la región del directorio antes y después de escribir la entrada del archivo.

En las figuras 4.16 y 4.17 se muestra la región FAT antes y después de escribir en la tabla la secuencia de clusters del archivo. El archivo `FILE1` tiene asignado un sólo cluster, es por ello que su valor de entrada en la FAT indica que ese cluster es el último del archivo.

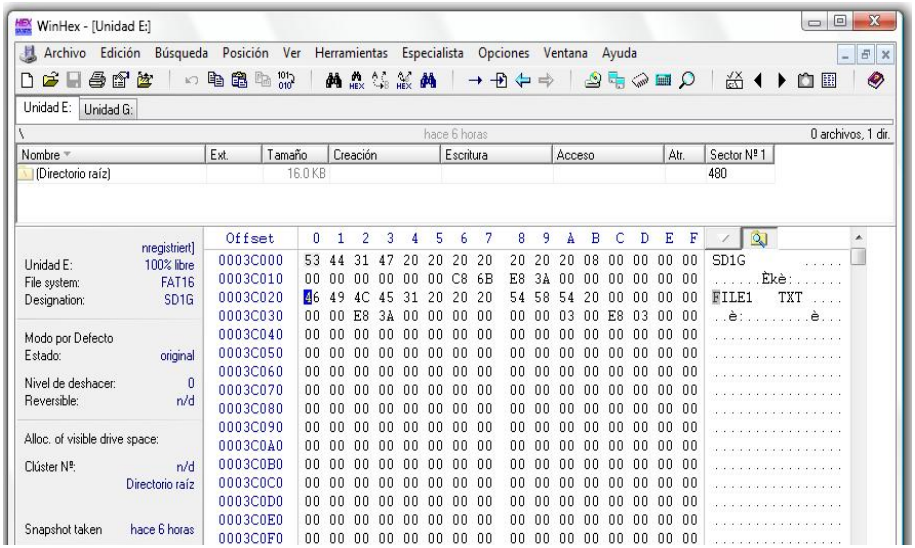
El sistema de registro de datos está programado para determinar, a partir de la información del directorio para `FILE1`, la cantidad de clusters que le corresponden al archivo, y escribe en estos clusters el contenido de `FILE1` en la región de datos. El sistema de registro realiza la escritura de los datos del primer sector del primer cluster al último sector del último cluster del archivo. Dado que a `FILE1` se le asigna un sólo cluster, el sistema de registro escribe los bytes de datos del primer sector al último sector de ese cluster. El tamaño de `FILE1` se especifica en su entrada en el directorio y este valor le indica al sistema operativo que sólo los primeros 1000 bytes del único cluster asignado a `FILE1` (cluster 3) son datos del archivo. Esto quiere decir que no importa si todo el cluster 3 tiene información escrita, pues sólo los primeros 1000 bytes serán considerados por el sistema operativo como los datos del achivo `FILE1`.

El área sombreada en las figuras 4.18 y 4.19 indica los bytes de datos del archivo `FILE1` para las memorias de 1[GB] y 4[GB] respectivamente. Las figuras 4.18(a) y 4.19(a) muestran el inicio de la región de datos y las figuras 4.18(b) y 4.19(b) muestran el final del área sombreada. El último sector escrito del cluster 3 se muestra en las figuras 4.20 y 4.21 para cada tarjeta de memoria.

Como se comentó en el apartado anterior, mientras más grande sea el tamaño de cluster, más espacio de memoria se desperdicia al hacer la asignación de clusters a un archivo. Esto lo podemos comprobar para nuestro archivo de 1000 bytes. El tamaño de cluster para la tarjeta de 1[GB] es de 16[kB], lo que significa que para alojar los 1000 bytes de datos de `FILE1` se utiliza aproximadamente $\frac{1}{16}$ de ese cluster. Para la tarjeta

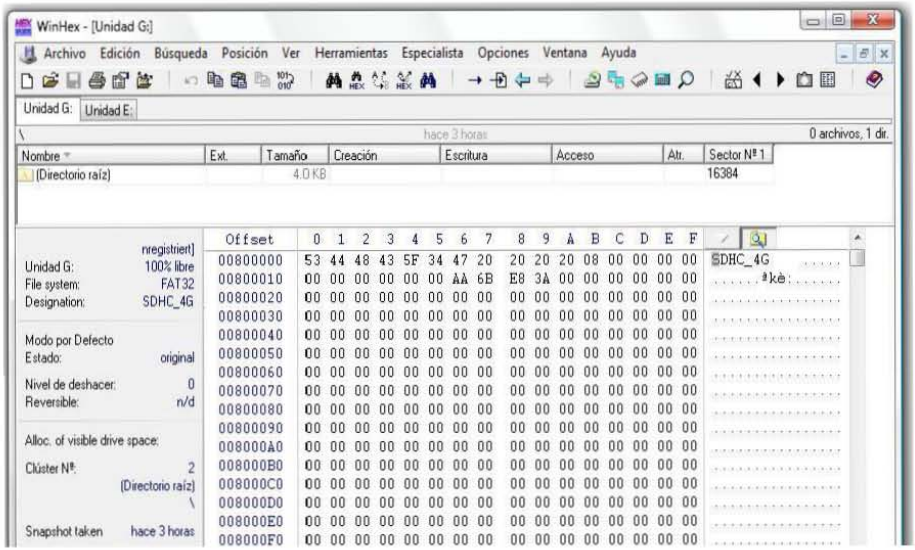


SD1G formateada.

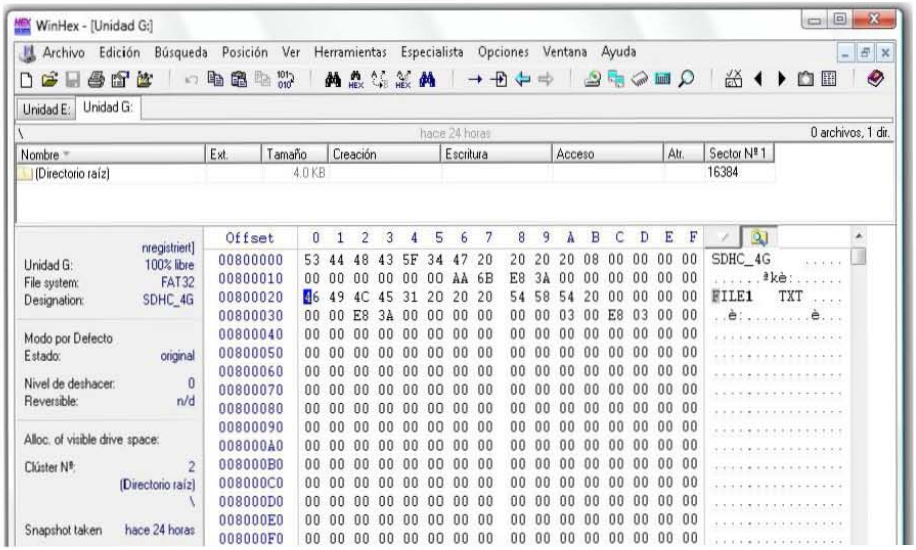


SD1G con el archivo de 1000 bytes.

Fig. 4.14. Región del directorio raíz de la memoria SD1G.

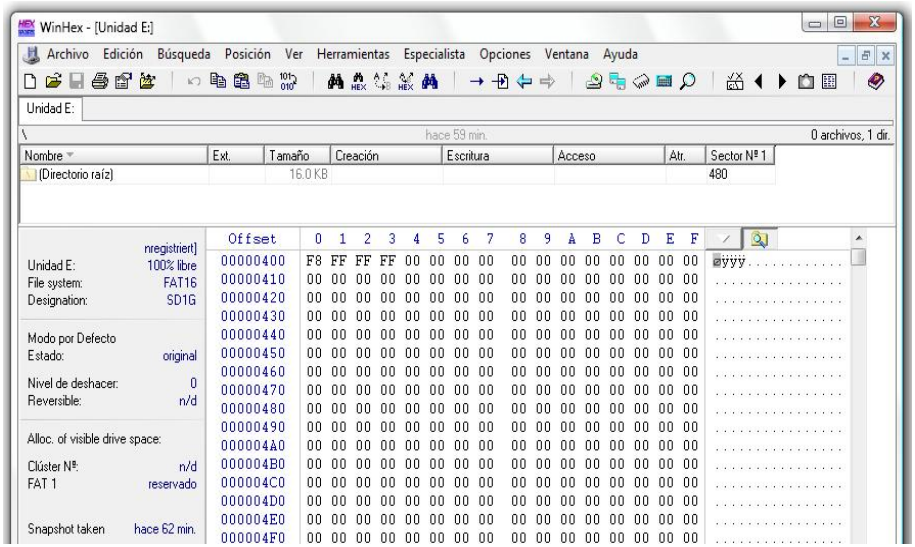


SDHC_4G formateada.

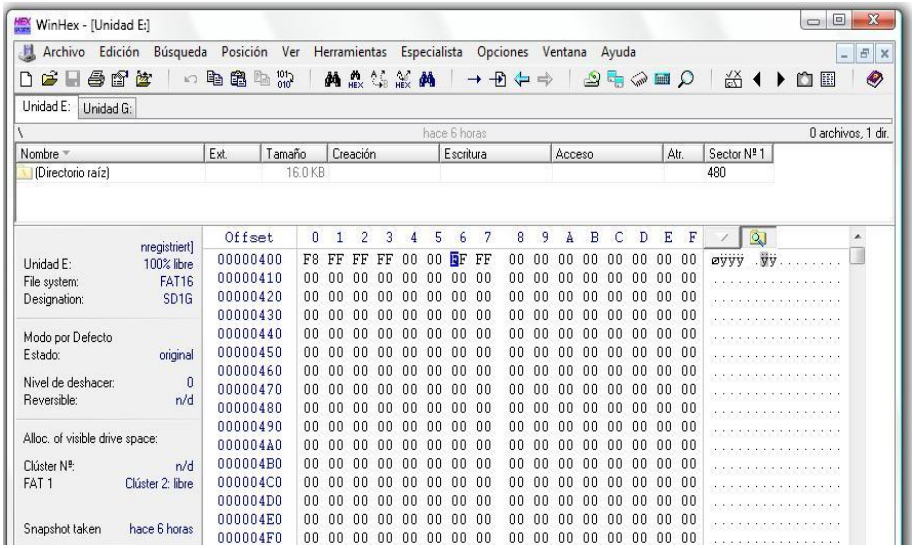


SDHC_4G con el archivo de 1000 bytes.

Fig. 4.15. Región del directorio de la memoria SDHC_4G.

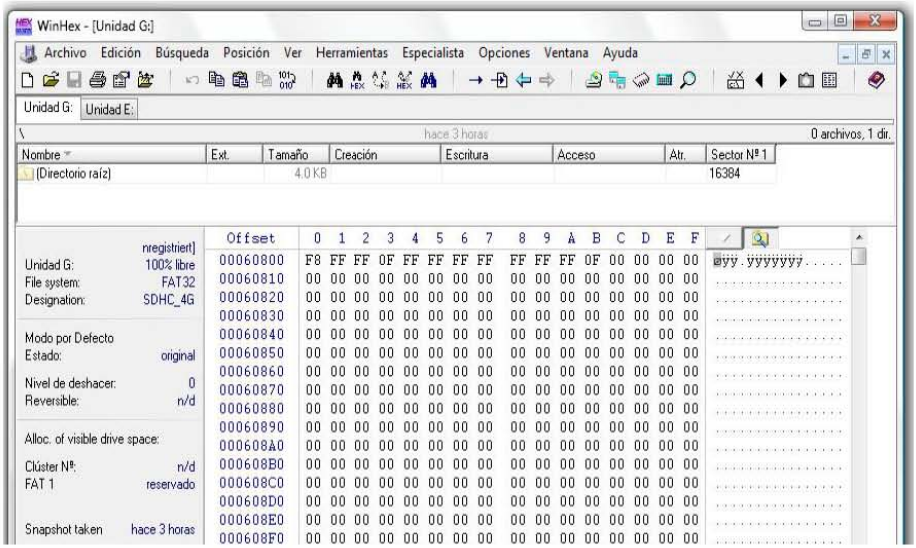


SD1G formateada.

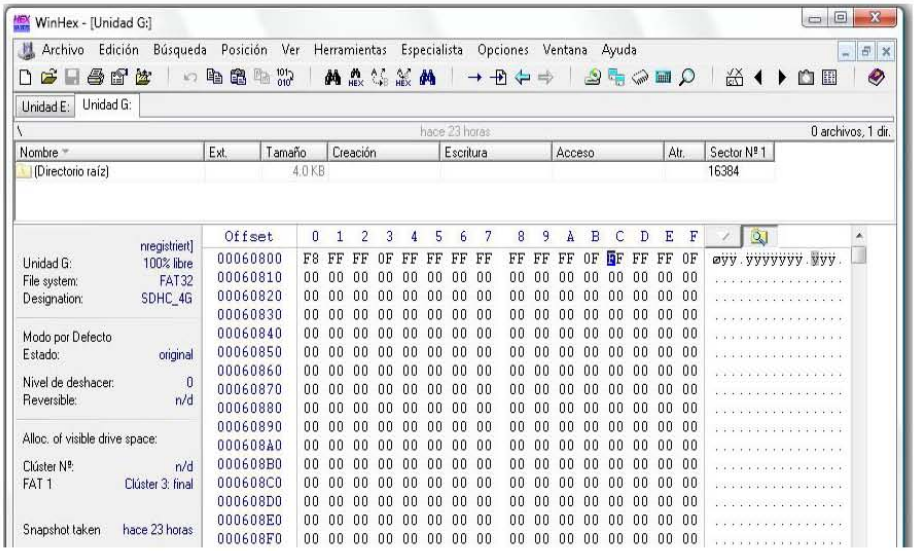


SD1G con el archivo de 1000 bytes.

Fig. 4.16. Región FAT de la memoria SD1G.

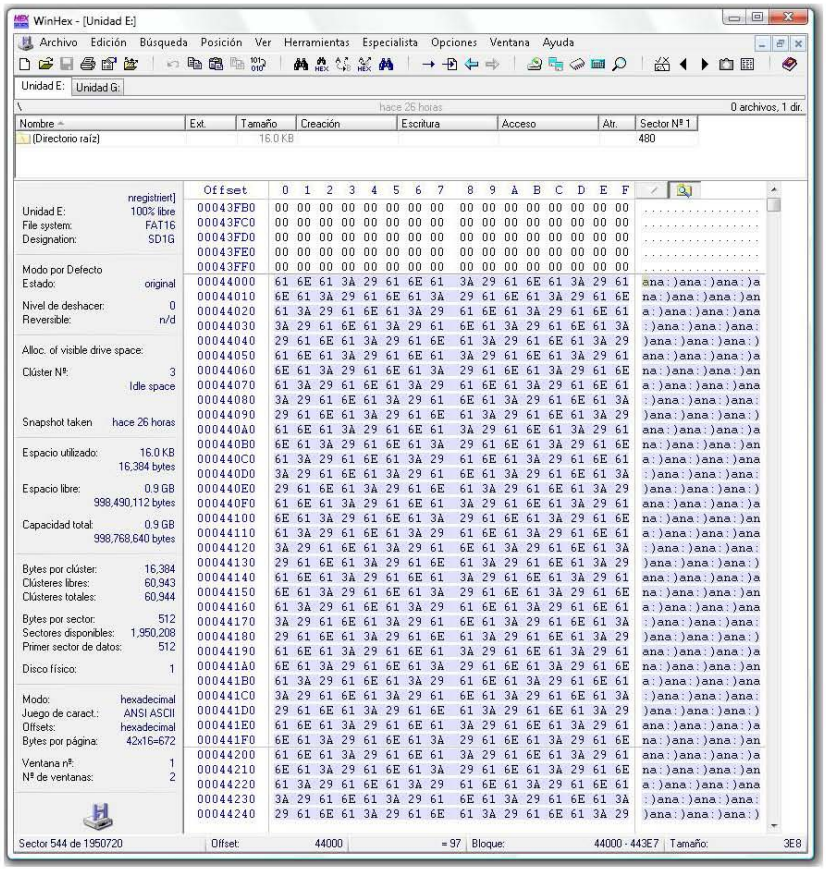


SDHC_4G formateada.

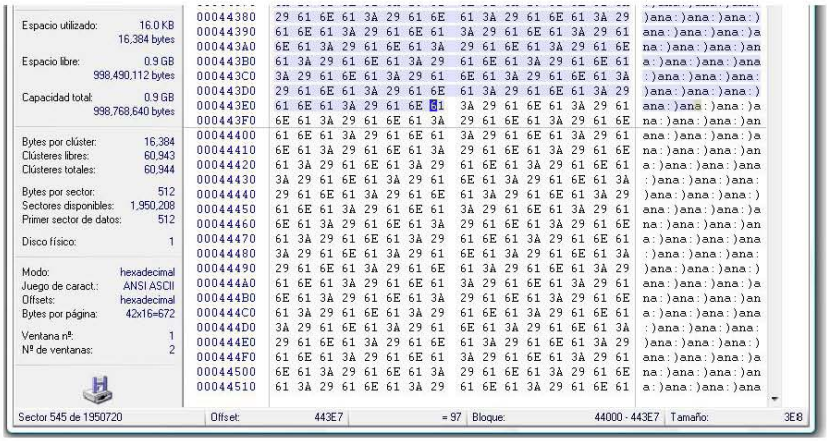


SDHC_4G con el archivo de 1000 bytes.

Fig. 4.17. Región FAT de la memoria SDHC_4G.

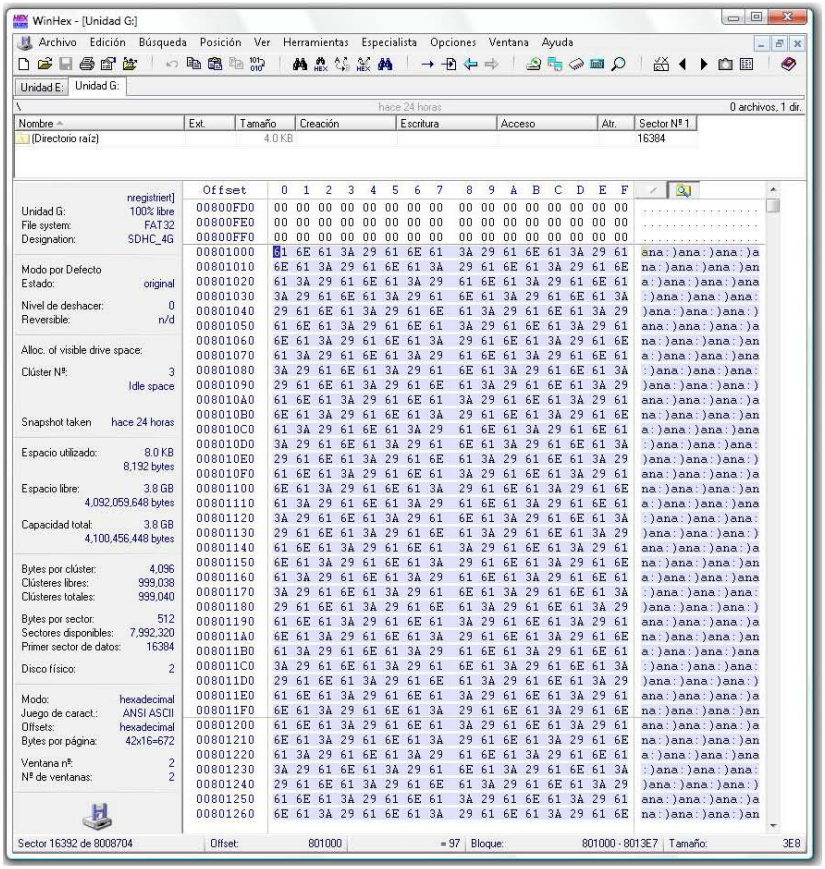


(a)

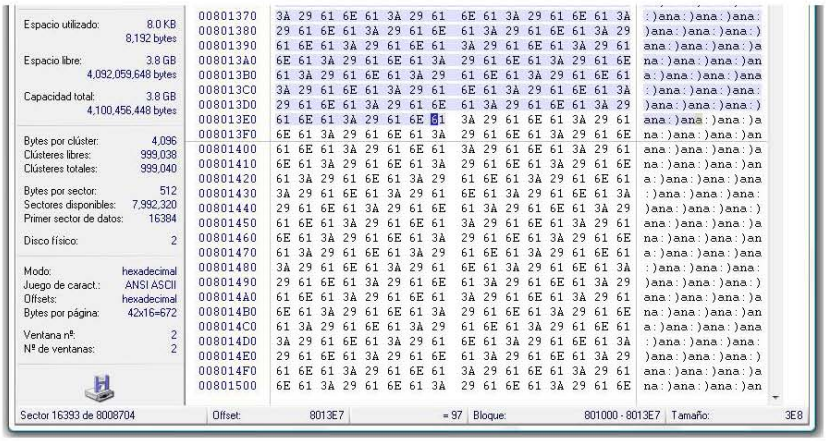


(b)

Fig. 4.18. Región de datos de la memoria SD1G.



(a)



(b)

Fig. 4.19. Región de datos de la memoria SDHC_4G.

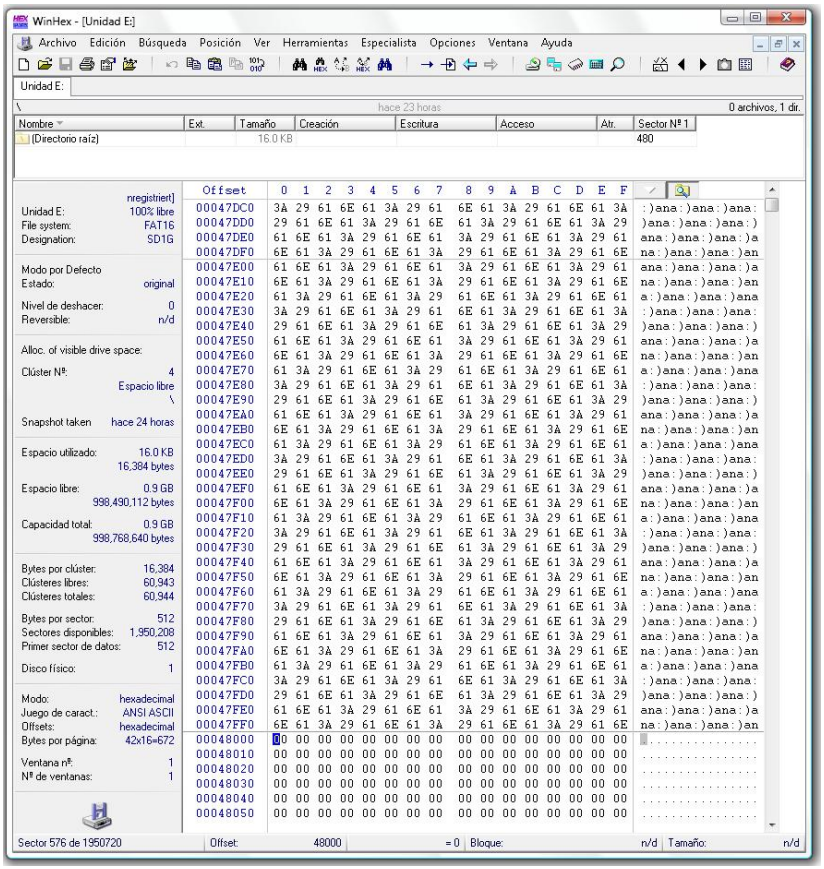


Fig. 4.20. Último sector del cluster 3 de la memoria SD1G.

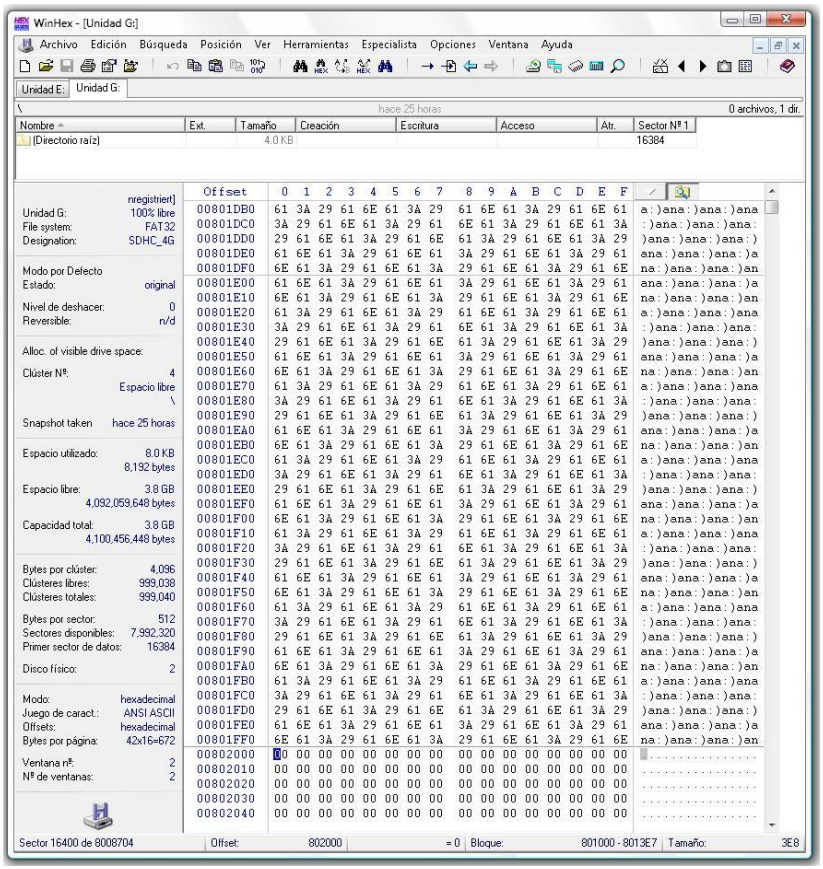


Fig. 4.21. Último sector del cluster 3 de la memoria SDHC_4G.

de 4[GB], el tamaño de cluster es de 4[kB] por lo que los 1000 bytes del archivo utilizarán aproximadamente $\frac{1}{4}$ de ese cluster.

4.1.3. Prueba 3 : Archivo de 3102009 bytes

El objetivo de esta prueba es utilizar la información del archivo creado para efectuar operaciones que nos lleven a comprobar los resultados mostrados en el WinHex para este archivo.

El archivo para esta prueba se crea en una memoria SD de 1[GB]. La tarjeta de memoria está formateada con los valores predeterminados de Windows que se mostraron en la prueba anterior para la memoria SD1G. El archivo creado se nombra `FILE1`, es de 3102009 bytes y su cluster de inicio es el cluster 3.

En la figura 4.22 se muestra el primer sector del cluster 3 de `FILE1` y en la columna de la izquierda se señala información relevante del archivo.

A continuación se calcula el número del último sector del último cluster asignado al archivo. En el proceso se efectúan operaciones cuyos resultados se pueden comprobar en las capturas de pantalla del WinHex, mostradas en las figuras siguientes.

Primero se calcula el número de clusters que el archivo necesita. Se sabe que el tamaño de cluster es de 16384 bytes, de forma que:

$$16384\frac{[bytes]}{[cluster]} \times \frac{1[sector]}{512[bytes]} = 32\frac{[sectores]}{[cluster]}$$

por lo que cada cluster tendrá 32 sectores. Así, la cantidad de clusters de `FILE1` se calcula como sigue:

$$3102009[bytes] \times \frac{1[sector]}{512[bytes]} \times \frac{1[cluster]}{32[sectores]} = 189.331[clusters]$$

y el resultado se redondea al entero más próximo. Entonces, el archivo `FILE1` necesita 190[$clusters$]. La cantidad de bytes escritos en estos clusters se calcula de la siguiente forma:

$$190[clusters] \times \frac{32[sector]}{1[cluster]} \times \frac{512[bytes]}{1[sector]} = 3112960[bytes]$$

y estos bytes se escriben desde el primer sector del cluster 3 y hasta el último sector del cluster 192. El espacio de memoria utilizado se señala en la columna izquierda de la figura 4.22.

Como se mencionó con anterioridad, no todos los bytes escritos en los clusters asignados al archivo corresponden a sus bytes de datos.

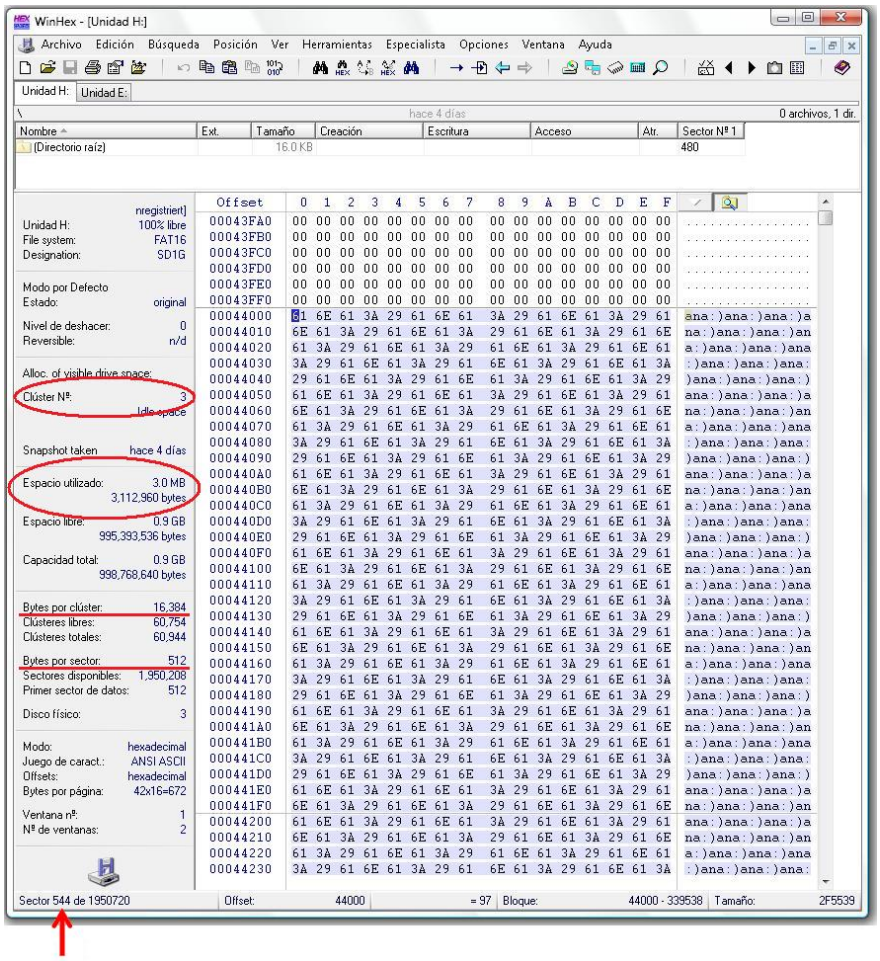


Fig. 4.22. Primer sector de datos del archivo.

Así, de los 3112960 bytes escritos en los 190 clusters, sólo los primeros 3102009 bytes corresponden a bytes de datos de `FILE1`. De manera que se necesita calcular hasta dónde se consideran los bytes escritos como bytes se datos, es decir, se debe calcular en qué sector del último cluster del archivo (cluster 192) terminan los bytes de datos. Esto se obtiene de la siguiente forma:

$$3112960[bytes] - 3102009[bytes] = 10951[bytes]$$

$$10951[bytes] \times \frac{1[sector]}{512[bytes]} = 21.388[sectores]$$

es decir, los últimos 21.388 sectores del último cluster del archivo (cluster 192) no contienen bytes de datos. Entonces, estos deben restarse del total de sectores del cluster 192 que sí tienen bytes de datos.

$$32[sectores] - 21.388[sectores] = 10.611[sectores]$$

Con este resultado se sabe que los primeros 10 sectores del cluster 192 guardan bytes de datos. Con este valor y con el número del primer sector del cluster 3, señalado por la flecha en la figura 4.22, se puede determinar el número del último sector con bytes de datos del archivo de la siguiente forma.

$$544[sectores] + 32\frac{[sectores]}{[cluster]} \times 189[clusters] + 10[sectores]$$

El primer sumando es el número del primer sector del cluster 3. El segundo sumando resulta de multiplicar el número de sectores en cada cluster por los 189 clusters completos que abarca `FILE1`, y el tercer sumando corresponde a los últimos 10 sectores de `FILE1` que guardan bytes de datos. El resultado de esta operación es $6602[sectores]$, de manera que el sector 6602 es el último sector de `FILE1` que contiene bytes de datos. Este sector se muestra en la figura 4.23, el cursor indica su inicio. En la columna de la izquierda se señala el cluster que lo contiene y el número de este sector; en la esquina inferior derecha de esta figura se señala el tamaño del archivo (valor en hexadecimal). El último sector del cluster 192 se muestra en la figura 4.24. El cursor indica el inicio del siguiente cluster de la región de datos.

La secuencia de clusters asignados al archivo `FILE1` se muestra en la figura 4.25. Se observa que se señala el penúltimo valor de entrada de la secuencia. Este valor indica que el cluster 192 es el último cluster asignado al archivo.

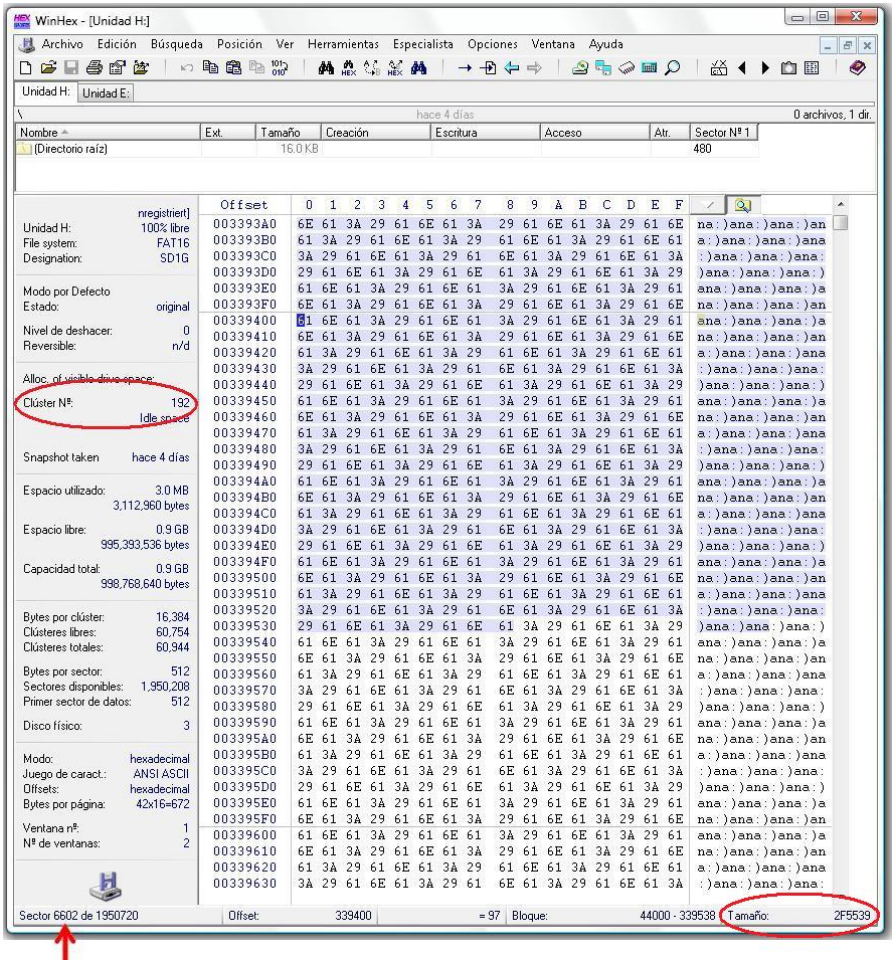


Fig. 4.23. Último sector de datos del archivo.

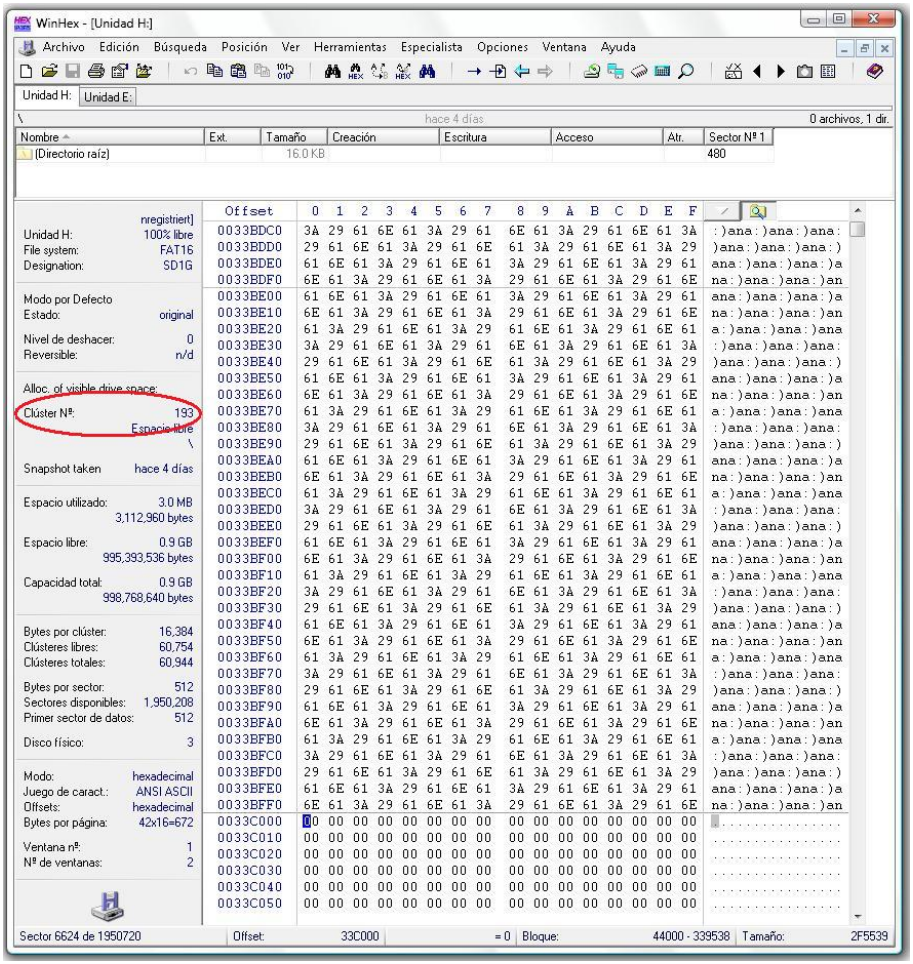


Fig. 4.24. Sector final del último cluster del archivo.

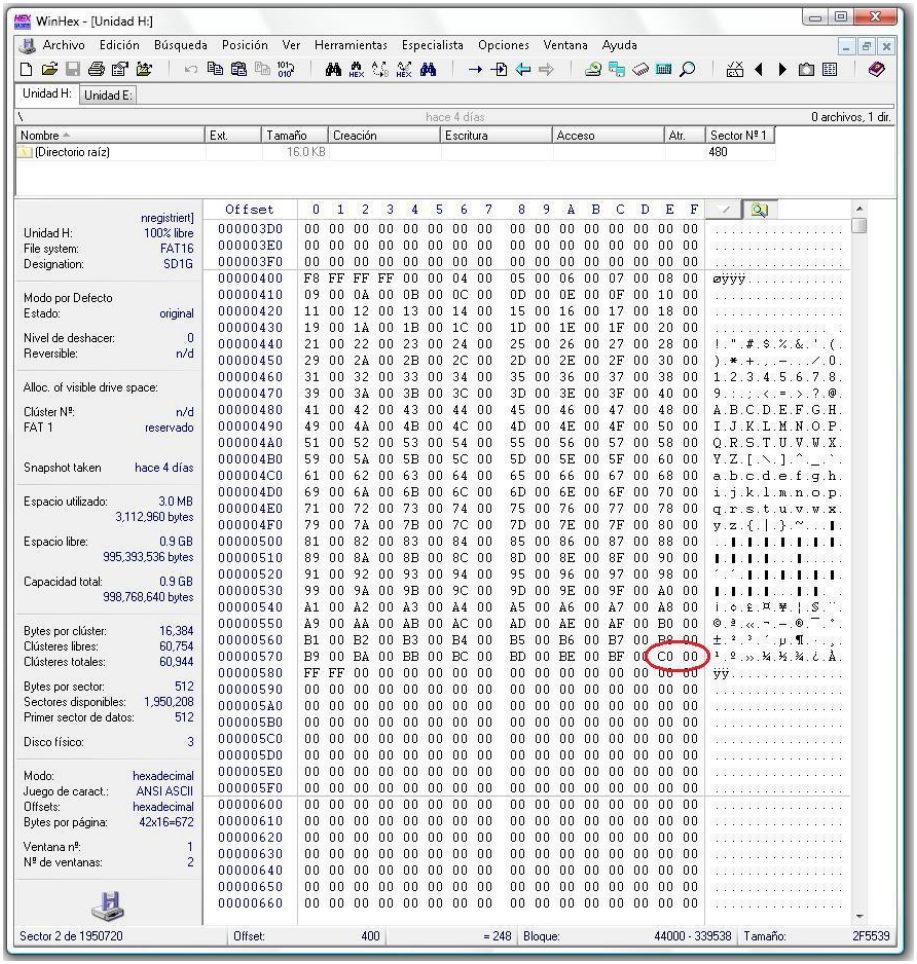


Fig. 4.25. Entradas del archivo en la FAT.

4.1.4. Prueba 4 : Archivo de 12345678 bytes

En esta última prueba se verá el resultado en Windows del archivo de 12345678 bytes creado en el capítulo 3.

La información con la que se formatea la tarjeta se muestra en la figura 4.26(a). En la figura 4.26(b) se observa la unidad de memoria conectada a la computadora luego de que se le ha grabado el archivo y en la figura 4.26(c) se muestran las propiedades de la memoria.

El contenido de la tarjeta de memoria se muestra en la figura 4.27 junto con las propiedades del archivo.

Se debe observar que los valores señalados en las figuras 4.26(c) y 4.27 no coinciden. Se supondría que el "espacio usado" de la ventana de propiedades de la memoria debería ser el mismo que el "tamaño en disco" de la ventana de propiedades del archivo, y sería cierto si la tarjeta soportara el sistema FAT16. Como lo muestra la figura 4.26(a), el sistema de archivos de la tarjeta utilizada es FAT32 y en este sistema se dedica al menos un cluster en la región de datos para el directorio. Por esta razón los valores señalados en las figuras no coinciden, y se puede probar fácilmente que difieren en exactamente un cluster:

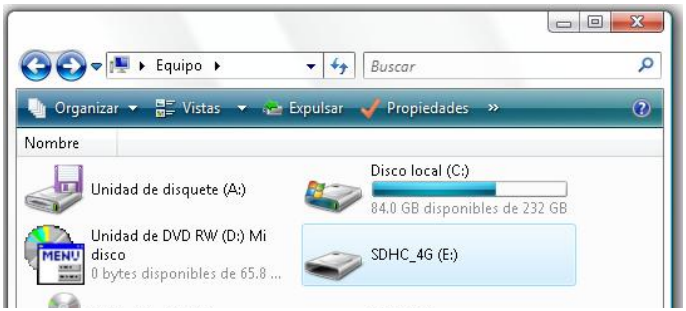
$$12353536[bytes] - 12349440[bytes] = 4096[bytes]$$

y de la figura 4.26(a) se sabe que $4096\frac{[bytes]}{[cluster]}$, de forma que estos bytes de diferencia corresponden al cluster del directorio.

Como se comentaba en la prueba 2, una forma práctica para conocer el número de páginas del archivo creado es con el contador de palabras de MS Word. En la figura 4.28 se muestra que los 12345678 bytes del archivo equivalen a 3132 páginas escritas.

Con esta prueba se concluyen las pruebas al sistema de registro de datos. En el siguiente capítulo da por terminado el presente trabajo presentando las conclusiones del proyecto.

(a)



(b)

(c)

Fig. 4.26. Memoria SDHC_4G.

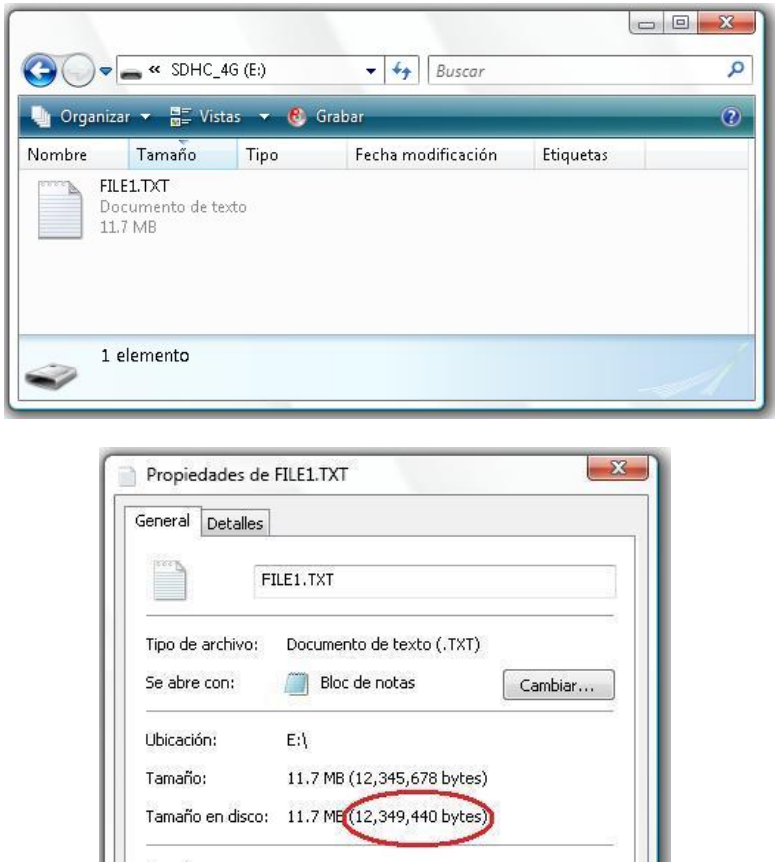


Fig. 4.27. Archivo `FILE1` en la memoria SDHC_4G.

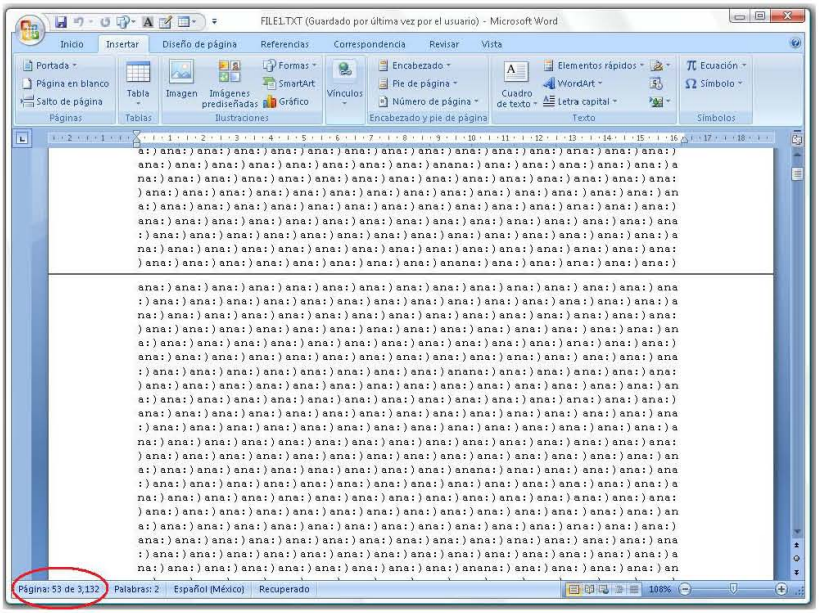


Fig. 4.28. Contador de palabras en MS Word para `FILE1`.

5

Resultados y Conclusiones

El presente trabajo ha tenido por objetivo el diseño y la implementación de un módulo de registro de datos en memorias SD CARD, de forma que los datos almacenados se recuperen de manera convencional utilizando un lector para este tipo de tarjetas de memoria.

La propuesta para alcanzar este objetivo consistió en un sistema basado en un microcontrolador, en el que se implementó el protocolo de comunicación de las memorias SD CARD, así como las reglas de los sistemas de archivos FAT para la compatibilidad de las tarjetas con el sistema operativo Windows. El resultado fue un sistema de registro de datos que cumple totalmente el objetivo planteado.

Un microcontrolador como núcleo del sistema de registro fue una consecuencia natural luego de estudiar las especificaciones de las tarjetas de memoria y la decisión de un PIC fue conveniente no sólo por el bajo costo que resultó desarrollar con esta tecnología, si no también por la excelente información y atención brindada por Microchip al desarrollador.

La implementación del protocolo de la SD CARD es relativamente simple, pero su especificación es un documento extenso. En contra parte, la implementación del sistema de archivos es meticulosa y laboriosa, pero la especificación de Microsoft es breve. La extensión del código del microcontrolador es considerable y la optimización de funciones es una buena recomendación.

El sistema de registro de datos es susceptible de mejoras en varios aspectos:

- Implementación de *wear levelling*. Dado que el grabado de datos se hace en memorias flash, es deseable implementar técnicas para prolongar el tiempo de vida útil de la tarjeta de memoria.

- Optimización del registro de datos. Al ajustar el registro de datos a los espacios de memoria necesarios por el tamaño del archivo.

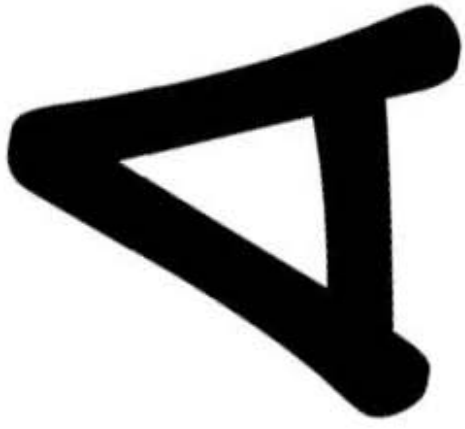
- Generar más de un archivo. En aplicaciones prácticas resultará conveniente registrar datos en más de un archivo e incluso utilizar carpetas que ordenen la información grabada en las tarjetas.

- Implementación de controles externos y alertas. Será deseable que a través de controles externos el usuario pueda configurar las características del registro y que con alertas se pueda conocer el estado del grabado.

- Implementación de bajo consumo. Será conveniente implementar en el microcontrolador algún modo de bajo consumo de energía.

De forma personal, este proyecto me ha aportado conocimiento y satisfacciones de muchas maneras. Ha sido grato complementar mi formación profesional con un proyecto de este tipo y he reafirmado lo que un día descubrí: no siempre lo que parece simple de implementar lo es.

Hojas de especificaciones



MICROCHIP

PIC18FXX2

28/40-pin High Performance, Enhanced FLASH Microcontrollers with 10-Bit A/D

High Performance RISC CPU:

- C compiler optimized architecture/instruction set
  - Source code compatible with the PIC16 and PIC17 instruction sets
- Linear program memory addressing to 32 Kbytes
- Linear data memory addressing to 1.5 Kbytes

Device	On-Chip Program Memory		On-Chip RAM (bytes)	Data EEPROM (bytes)
	FLASH (bytes)	# Single Word Instructions		
PIC18F242	16K	8192	768	256
PIC18F252	32K	16384	1536	256
PIC18F442	16K	8192	768	256
PIC18F452	32K	16384	1536	256

- Up to 10 MIPs operation:
  - DC - 40 MHz osc./clock input
  - 4 MHz - 10 MHz osc./clock input with PLL active
- 16-bit wide instructions, 8-bit wide data path
- Priority levels for interrupts
- 8 x 8 Single Cycle Hardware Multiplier

Peripheral Features:

- High current sink/source 25 mA/25 mA
- Three external interrupt pins
- Timer0 module: 8-bit/16-bit timer/counter with 8-bit programmable prescaler
- Timer1 module: 16-bit timer/counter
- Timer2 module: 8-bit timer/counter with 8-bit period register (time-base for PWM)
- Timer3 module: 16-bit timer/counter
- Secondary oscillator clock option - Timer1/Timer3
- Two Capture/Compare/PWM (CCP) modules. CCP pins that can be configured as:
  - Capture input: capture is 16-bit, max. resolution 6.25 ns (TCY/16)
  - Compare is 16-bit, max. resolution 100 ns (TCY)
  - PWM output: PWM resolution is 1- to 10-bit, max. PWM freq. @: 8-bit resolution = 156 kHz, 10-bit resolution = 39 kHz
- Master Synchronous Serial Port (MSSP) module, Two modes of operation:
  - 3-wire SPI™ (supports all 4 SPI modes)
  - I^2C™ Master and Slave mode

Peripheral Features (Continued):

- Addressable USART module:
  - Supports RS-485 and RS-232
- Parallel Slave Port (PSP) module

Analog Features:

- Compatible 10-bit Analog-to-Digital Converter module (A/D) with:
  - Fast sampling rate
  - Conversion available during SLEEP
  - Linearity ≤ 1 LSb
- Programmable Low Voltage Detection (PLVD)
  - Supports interrupt on-Low Voltage Detection
- Programmable Brown-out Reset (BOR)

Special Microcontroller Features:

- 100,000 erase/write cycle Enhanced FLASH program memory typical
- 1,000,000 erase/write cycle Data EEPROM memory
- FLASH/Data EEPROM Retention: > 40 years
- Self-reprogrammable under software control
- Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
- Programmable code protection
- Power saving SLEEP mode
- Selectable oscillator options including:
  - 4X Phase Lock Loop (of primary oscillator)
  - Secondary Oscillator (32 kHz) clock input
- Single supply 5V In-Circuit Serial Programming™ (ICSP™) via two pins
- In-Circuit Debug (ICD) via two pins

CMOS Technology:

- Low power, high speed FLASH/EEPROM technology
- Fully static design
- Wide operating voltage range (2.0V to 5.5V)
- Industrial and Extended temperature ranges
- Low power consumption:
  - < 1.6 mA typical @ 5V, 4 MHz
  - 25 μA typical @ 3V, 32 kHz
  - < 0.2 μA typical standby current

SD Specifications

Part 1

Physical Layer

Simplified Specification

Version 2.00

September 25, 2006

SD Group
Matsushita Electric Industrial Co., Ltd. (Panasonic)
SanDisk Corporation
Toshiba Corporation

Technical Committee
SD Card Association

Microsoft

Hardware White Paper

Designing Hardware for Microsoft® Operating Systems

Microsoft Extensible Firmware Initiative FAT32 File System Specification

FAT: General Overview of On-Disk Format

Version 1.03, December 6, 2000
Microsoft Corporation

The FAT (File Allocation Table) file system has its origins in the late 1970s and early1980s and was the file system supported by the Microsoft® MS-DOS® operating system. It was originally developed as a simple file system suitable for floppy disk drives less than 500K in size. Over time it has been enhanced to support larger and larger media. Currently there are three FAT file system types: FAT12, FAT16 and FAT32. The basic difference in these FAT sub types, and the reason for the names, is the size, in bits, of the entries in the actual FAT structure on the disk. There are 12 bits in a FAT12 FAT entry, 16 bits in a FAT16 FAT entry and 32 bits in a FAT32 FAT entry.

Contents

19-0273; Rev 7; 1/07

MAXIM

3.0V to 5.5V, Low-Power, up to 1Mbps, True RS-232 Transceivers Using Four 0.1µF External Capacitors

General Description

The MAX3222/MAX3232/MAX3237/MAX3241 transceivers have a proprietary low-dropout transmitter output stage enabling true RS-232 performance from a 3.0V to 5.5V supply with a dual charge pump. The devices require only four small 0.1µF external charge-pump capacitors. The MAX3222, MAX3232, and MAX3241 are guaranteed to run at data rates of 120kbps while maintaining RS-232 output levels. The MAX3237 is guaranteed to run at data rates of 250kbps in the normal operating mode and 1Mbps in the MegaBaud™ operating mode, while maintaining RS-232 output levels.

The MAX3222/MAX3232 have 2 receivers and 2 drivers. The MAX3222 features a 1µA shutdown mode that reduces power consumption and extends battery life in portable systems. Its receivers remain active in shutdown mode, allowing external devices such as modems to be monitored using only 1µA supply current. The MAX3222 and MAX3232 are pin, package, and functionally compatible with the industry-standard MAX242 and MAX232, respectively.

The MAX3241 is a complete serial port (3 drivers/5 receivers) designed for notebook and subnotebook computers. The MAX3237 (5 drivers/3 receivers) is ideal for fast modem applications. Both these devices feature a shutdown mode in which all receivers can remain active while using only 1µA supply current. Receivers R1 (MAX3237/MAX3241) and R2 (MAX3241) have extra outputs in addition to their standard outputs. These extra outputs are always active, allowing external devices such as a modem to be monitored without forward biasing the protection diodes in circuitry that may have VCC completely removed.

The MAX3222, MAX3237, and MAX3241 are available in space-saving TSSOP and SSOP packages.

Applications

Notebook, Subnotebook, and Palmtop Computers

High-Speed Modems

Battery-Powered Equipment

Hand-Held Equipment

Peripherals

Printers

Typical Operating Circuits appear at end of data sheet.

Next Generation Device Features

- **For Smaller Packaging: MAX3228E/MAX3229E: +2.5V to +5.5V RS-232 Transceivers in UCSP™**
- **For Integrated ESD Protection: MAX3222E/MAX3232E/MAX3237E/MAX3241E*/ MAX3246E: ±15kV ESD-Protected, Down to 10nA, 3.0V to 5.5V, Up to 1Mbps, True RS-232 Transceivers**
- **For Low-Voltage or Data Cable Applications: MAX3380E/MAX3381E: +2.35V to +5.5V, 1µA, 2 Tx/2 Rx RS-232 Transceivers with ±15kV ESD-Protected I/O and Logic Pins**

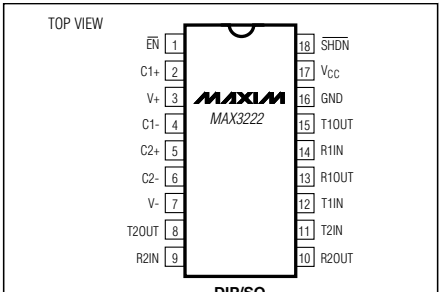
Ordering Information

PART	TEMP RANGE	PIN-PACKAGE	PKG CODE
MAX3222CUP+	0°C to +70°C	20 TSSOP	U20+2
MAX3222CAP+	0°C to +70°C	20 SSOP	A20+1
MAX3222CWN+	0°C to +70°C	18 SO	W18+1
MAX3222CPN+	0°C to +70°C	18 Plastic Dip	P18+5

+Denotes lead-free package.

Ordering Information continued at end of data sheet.

Pin Configurations



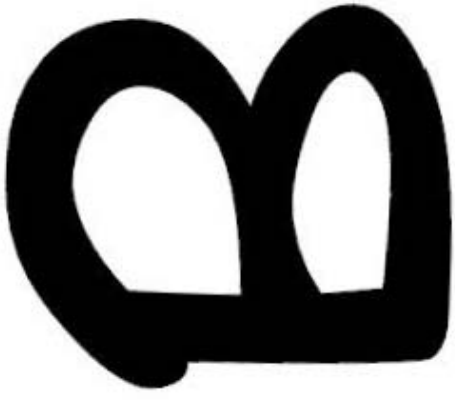
Pin Configurations continued at end of data sheet.

MegaBaud and UCSP are trademarks of Maxim Integrated Products, Inc.

**Covered by U.S. Patent numbers 4,636,930; 4,679,134; 4,777,577; 4,797,899; 4,809,152; 4,897,774; 4,999,761; and other patents pending.*

For pricing, delivery, and ordering information, please contact Maxim/Dallas Direct! at 1-888-629-4642, or visit Maxim's website at www.maxim-ic.com.

Glosario de términos



ACMD *Application specific Command*. Nomenclatura para representar a los comandos que van precedidos por el comando estándar 55.

ASCII *American Standard Code for Information Interchange*. Esquema de codificación de caracteres para representar texto en las computadoras.

BIOS *Basic Input/Output System*. Es el *firmware* o código de inicio que ejecuta una computadora al ser encendida.

CCS *Card Capacity Status*. Bit de estatus del OCR que indica si la SD CARD es tipo SD o SDHC.

CF *Compact Flash*. Estándar de tarjetas de memoria flash basado en el formato producido por SanDisk.

CMD *Command*. Nomenclatura para representar a los comandos estándar.

CRC *Cyclic Redundancy Code*. Código de detección de error que protege contra errores de transmisión entre la tarjeta y el *host*.

DRT *Data Response Token*. Byte enviado por la tarjeta para reconocer la recepción de un bloque de datos.

EEPROM *Electrically Erasable Programmable Read Only Memory*. Es el tipo más flexible de memoria ROM pues puede borrarse a través de *software*.

FAT *File Allocation Table*. Familia de sistemas de archivos de Microsoft que utiliza la tabla de asignación FAT.

FAT12 Sistema de archivos de la familia FAT que utiliza un tamaño de entrada en la tabla de asignación de 12 bits.

FAT16 Sistema de archivos de la familia FAT que utiliza un tamaño de entrada en la tabla de asignación de 16 bits.

FAT32 Sistema de archivos de la familia FAT que utiliza un tamaño de entrada en la tabla de asignación de 28 bits.

firmware Término usado para denotar el código fijo que permite que un sistema electrónico realice sus funciones básicas de operación.

HCS *High Capacity Support*. Bit fijado por el *host* para indicar si acepta o no las memorias SDHC.

IEEE *Institute of Electrical and Electronics Engineers*. Organización internacional para el progreso tecnológico en las áreas eléctrica, electrónica y tecnología de la información.

MMC *MultiMedia Card*. Estándar de tarjetas de memoria flash desarrollado por Siemens AG y SanDisk.

MS	*Memory Stick*. Estándar de tarjetas de memoria flash desarrollado por Sony.
NVRAM	*Non-Volatile Random Access Memory*. Nombre usado para describir cualquier tipo de RAM no volátil.
OCR	*Operation Conditions Register*. Registro de 32 bits dentro de la SD CARD que contiene la ventana de voltajes que acepta la tarjeta y dos bits de estatus.
PC Card	*Personal Computer Card*. Interfaz periférica diseñada para las computadoras portátiles.
PCMCIA	*Personal Computer Memory Card International Association*. Organización internacional que desarrolla y promueve el estándar de la PC CARD.
PROM	*Programmable ROM*. Es un tipo de ROM que puede ser programada una sola vez.
ROM	*Read Only Memory*. Tipo de memoria no volátil utilizada para almacenar programas del sistema que no se modifican normalmente y que se requieren tener disponibles en todo momento.
SB	*Start Block Token*. Byte con valor 0xFE que marca el inicio de los bytes de datos en una transmisión de un sólo bloque de datos.
SBT	*Start Block Token* (para múltiples bloques). Byte con valor 0xFC que antecede a cada bloque de datos en una transmisión de múltiples bloques.
SD	*Secure Digital*. Nomenclatura para hacer referencia a las memorias SD CARD con capacidades de hasta 2[GB].
SD CARD	*Secure Digital Card*. Estándar de tarjetas de memoria flash basado en el formato de las memorias MMC desarrollado por Panasonic, SanDisk y Toshiba.
SDHC	*Secure Digital High Capacity*. Nomenclatura para hacer referencia a las memorias SD CARD con capacidades de hasta 32[GB].
SDRAM	*Synchronous Dynamic Random Access Memory*. Tipo de DRAM que espera una señal de reloj antes de responder a los cambios en las entradas de control (interfaz síncrona).
SDXC	*Secure Digital eXtreme Capacity*. Es el tipo más nuevo de memorias *Secure Digital* con capacidades de hasta 2[TB].
SM	*Smart Media*. Estándar de tarjetas de memoria flash desarrollado por Toshiba.

SPI *Serial Peripheral Interface*. Protocolo de comunicación serie entre un dispositivo maestro y un dispositivo esclavo que utiliza cuatro señales para transmitir información entre ellos.

SRAM *Static Random Access Memory*. Es un tipo de memoria RAM que mantiene los datos almacenados en ella mientras se mantenga energizada, en contraste con la DRAM (*Dynamic Random Access Memory*) que requiere refrescarse varias veces por segundo para mantener los datos almacenados.

STT *Stop Tran Token* (para múltiples bloques). Byte con valor 0xFD que indica el fin de la transmisión de múltiples bloques.

USB *Universal Serial Bus*. Protocolo para transferir datos entre dispositivos digitales.

Bibliografía

[1]**LETTER SYMBOLS TO BE USED IN ELECTRICAL TECHNOLOGY - PART 2: TELECOMMUNICATIONS AND ELECTRONICS.** IEC 60027-2, Second edition, 2000-11.

[2]**PHYSICAL LAYER SIMPLIFIED SPECIFICATION VERSION 2.00.** SD group, 2006.

MICROSOFT EXTENSIBLE FIRMWARE INITIATIVE FAT32 FILE SYSTEM SPECIFICATION VERSION 1.03. Microsoft, 2000.

Deitel H.M y Deitel P.J. **COMO PROGRAMAR EN C/C++.** Estados Unidos. Ed. Prentice Hall, 1994.

Foust F. **APPLICATION NOTE:** Secure Digital Card Interface for the MSP430.

MAXIM APPLICATION NOTE 3969: SD Media Format Expands the MAXQ2000's Space for Nonvolatile Data Storage.

NXP APPLICATION NOTE: Interfacing a MultiMedia Card to the LH79520 System-On-Chip.

MICROCHIP APPLICATION NOTE AN647: Overview and Use of the PICmicro Serial Peripheral Interface.

MICROCHIP APPLICATION NOTE AN1045: Implementing File I/O Functions Using Microchip's Memory Disk Drive File System Library.

http://www.electronics-manufacturers.com/info/circuits-and-processors

http://www.tigoe.net/pcomp/code/serial-communication

http://www.pcguide.com/ref/ram/types_ROM.htm

http://en.wikipedia.org/wiki/Flash_memory

http://electronics.howstuffworks.com/flash-memory.htm

http://sourceware.org/jffs2/jffs2-html/node1.html

http://www.ross.net/crc/download/crc_v3.txt

http://www.pjrc.com/tech/8051/ide/fat32.html

http://augustcouncil.com/~tgibson/tutorial/ptr.html