Revisión	Fecha	Comentario	Autor
0	22/12/2008		Ulises Bigliati

ÍNDICE

PARTE 1 .... **2**
Introducción **2**
Objetivos **2**
Implementación **2**
Descripción de la aplicación **2**
Interfaz con el usuario 2
Funcionamiento del programa 3
Situaciones de error de memoria de log 3
Precauciones relativas al sistema de archivos 3
Formato de los archivos de log 4
Encabezado de archivo 4
Entradas de log 4
Muestra de un archivo generado por la aplicación 4
Estructura de archivos del proyecto **5**
Archivos del proyecto 5
Archivos fuente del proyecto 5
Nota de fin de sección **5**

PARTE 2 .... **6**
Los módulos de software del proyecto **6**
Módulo principal 6
Módulo flags.lib 9
Módulo logs.lib 11
El proceso de log in 19

PARTE 1

Introducción

Supongamos que tenemos una aplicación que requiere tomar muestras y generar con ellas un registro de unos 100 bytes de longitud por segundo; si contamos con un medio de almacenamiento de 1GB, probablemente reducido en un 10% por la gestión del file system, tendremos:
`10^9 Bytes x 0.90 / 100 Bytes =  9 x 10 ^ 6 registros.`
Los 9 millones de registros, a razón de 1 / seg. significan aprox. 3 meses y tres semanas de operación continua de datalogging. El módulo RCM4300 se orienta a este tipo de aplicaciones, ya que además del soporte para almacenamiento masivo cuenta con conversor AD de 8 canales, además de otros periféricos como las entradas para captura de eventos, detección de cuadratura, RTC, etc.

Objetivos

- Nos proponemos exponer un sencillo ejemplo sobre la utilización del hardware y software para almacenamiento masivo que nos ofrece el módulo Rabbit RCM4300.
- Utilizar el módulo de software `fat.lib` que está integrado en las distribuciones de Dynamic C.
- Usar como soporte físico del sistema de archivos la tarjeta de memoria removible miniSD (128MB - 1GB) presente en el hardware.

Implementación

Para la realización de este trabajo utilizamos el módulo Rabbit RCM4300 y como punto central para este pequeño proyecto nos servimos de las funcionalidades provistas por el módulo de software `fat.lib` mediante el cual realizamos la implementación del sistema de archivos FAT sobre una memoria miniSD.

Para la comunicación con el usuario se utiliza la interfaz *stdio* del entorno de desarrollo Dynamic C, a través de la cual mostramos algunos mensajes referidos a los estados de operación del programa.
Se utilizó además, el soporte de la placa de prototipos incluida en el kit de desarrollo de Rabbit, paricularmente se aprovecharon los LEDs DS2 y DS3 y el switch SW2, utilizados como indicadores de status y como comando de inicio/fin de operación de log respectivamente.

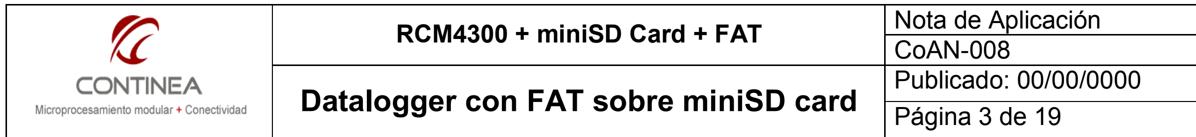
Si bien esta nota es producto de un desarrollo realizado sobre un RCM4300, la misma podría ser implementada con muy pocas variantes sobre un RCM4000 que también dispone de un bloque de memoria para almacenamiento masivo (32MB de NAND flash), de todas formas solo hemos comprobado el funcionamiento del programa sobre el módulo que es objeto de este trabajo.

Descripción de la aplicación

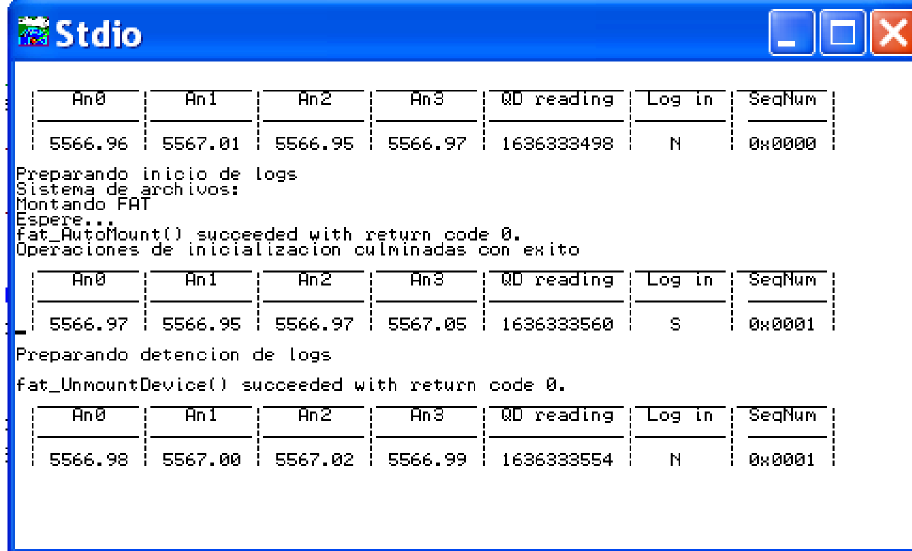
El programa de demostración está escrito en DC10.40 y se trata de la simulación de un sistema de adquisición de datos similar al de la nota CoAN-007, solo que en esta ocasión, los valores registrados se almacenan en archivos ASCII (valores separados por comas) sobre un sistema FAT, utilizando la tarjeta miniSD card como soporte físico.

Interfaz con el usuario

Elegimos el LED DS3 de la placa de desarrollo para indicar mediante su parpadeo que el programa está en operación normal y el LED DS2 lo utilizamos para indicar alguna situación de error en relación con la tarjeta de memoria, particularmente la situación de *memoria llena* o *no insertada*.
Existe un LED adicional que está ubicado en el mismo módulo Rabbit, junto al zócalo de la memoria miniSD, que funciona como indicador de que el sistema de archivos está montado. Este LED es manejado en forma transparente al programador por los drivers de la tarjeta de memoria. Si por alguna razón fuera necesario acceder a este LED, es posible hacerlo API mediante.
El switch S2 lo aprovechamos para iniciar/detener la operación de log manualmente.



Por otra parte, utilizamos la interfaz *stdio* del entorno de desarrollo para visualizar los valores simulados que estarían representando a los canales analógicos del 0 al 3 mas un canal del detector de cuadratura. También aprovechamos la salida estándard del entorno de desarrollo para visualizar algunos mensajes provenientes del sistema de archivos. Puede verse un ejemplo de esto último en la siguiente imagen:



Funcionamiento del programa

El programa tiene la misión de generar series de archivos de log de un tamaño máximo que hemos predefinido, y que son nombrados en base a un número de secuencia expresado en formato hexadecimal que define el nombre de cada archivo de log generado. Los archivos de log, poseen extensión ".log", Ejemplo: A1D.log.
Para esta demo, definimos que cuando una operación de log comienza (presionamos el switch S2), el programa realice el formateo de toda la memoria, el reseteo del numero de secuencia y comience con la generación de los registros en forma continua.
Si por cualquier motivo, el proceso de registro se interrumpe (corte de alimentación, reset, etc.) el equipo guardará su estado, continuando luego de reinicializarse con el proceso que fuera interrupido, es decir, que lo que se pretende, es evitar el formateo correspondiente al procedimiento normal previo a la operación de log.

Situaciones de error de memoria de log

Durante la operación normal del programa se pueden dar distintas situaciones de error relacionadas con la tarjeta de memoria, en esta demo básicamente se han controlado las siguientes excepciones

- Memoria llena:
  Cuando esto ocurre se indica la situación mediante el LED de error DS2.
  El equipo continúa en modo de "log in". La FAT será desmontada, pero será necesario detener la función de log, reponer una memoria con espacio disponible y así continuar con las registraciones.
- Memoria no insertada:
  Si el sistema intenta iniciar la función de log cuando la tarjeta no está insertada, se indicará la situación mediante el LED de error DS2, el sistema continuará en situación de log, y simplemente comenzará a registrar cuando la tarjeta sea insertada.

Precauciones relativas al sistema de archivos

Cuando se encuentra montado el sistema de archivos, el LED anaranjado dentro del módulo se enciende señalando esta condición. Bajo esta circunstancia no se debería apagar o resetear el equipo ni quitarle la tarjeta de memoria, pues esto podría ocasionar la pérdidad de datos.

Si el sistema de archivos está montado y se requiere apagar el equipo o retirarle la tarjeta de memoria será preciso desmontarlo antes. Para esto, según nuestro programa, debemos simplemente

presionar el switch S2 de la placa de prototipos, lo cual dará como resultado el desmonte del sistema de archivos y la conveniente finalización de cualquier operacion en curso pudiendo a continuación apagar el equipo o retirar la tarjeta de memoria con seguridad.

Nota: *durante nuestras pruebas se nos presentaron muchas oportunidades para desestimar las precauciones indicadas arriba, de tal forma que tuvimos la oportunidad de maltratar la tarjetita de memoria, buscando fallos que nunca se produjeron, en virtud de lo cual podemos dar testimonio de la robustez del sistema de archivos que implementa el módulo fat.lib de Rabbit.*

Formato de los archivos de log

Los archivos son de formato ASCII con valores separados por coma (CSV) y se organizan según el siguiente esquema:

Encabezado de archivo

SID	,	NS	,	DT	CR	LF

SID: Leyenda para identificación del sistema. Máximo 20 caractéres.
NS: Número de secuencia del archivo. Formato decimal (long), máximo 10 caractéres.
DT: Marca de tiempo en segundos tipo UNIX. Máximo 10 caracteres.
CR+LF: fin de línea.

Ejemplo: DEMO,426,18/09/2008 10:57:25

Entradas de log

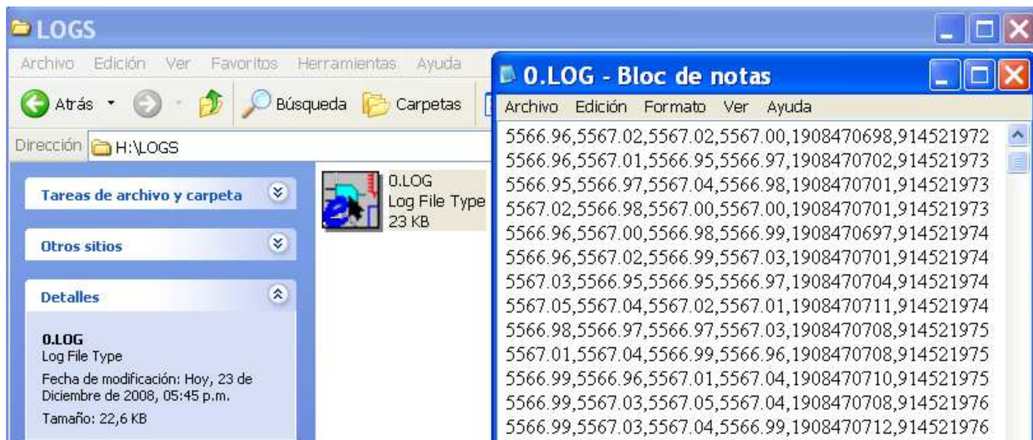
An0	,	An1	,	An2	,	An3	QD	TM	CR	LF

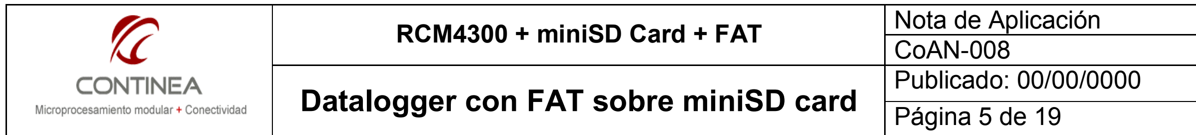
Anx: Valor de tensión en mili voltios del canál analógico (float 4 dígitos + 1 decimal).
QD: lectura del detector de cuadratura. Formato decimal (long), máximo 10 caractéres.
CR+LF: fin de línea.

Ejemplo: 1868.7,2836.1,1423.9,2807.8,50,906202645

Muestra de un archivo generado por la aplicación

A continuación podemos ver en el bloc de notas una fracción de un archivo de log generado por la aplicación que estamos describiendo.
En la barra de dirección puede verse la unidad H:\ que corresponde a la tarjeta de memoria miniSD reconocida como disco extraíble por una computadora PC. Dentro del disco H, se encuentra el directorio *LOGS* creado por nuestra aplicación para contener los archivos *0.LOG,1.LOG,* y siguientes.





Estructura de archivos del proyecto

Físicamente esta demostración está organizada según puede apreciarse en el siguiente esquema. Como de costumbre avisamos que esta es una organización típica para cualquiera de nuestros proyectos en DC, pero que no deja de ser una cuestión de preferencia en cuanto a la forma de organización de los archivos de un proyecto.

```
myProyects___CoAN-008
                    |___CoAN-008.c
                    |___CoAN-008.dcp
                    |___lib.dir
                    |___myLibs
                            |___flags.lib
                            |___logs.lib
```

A continuación se explicará la naturaleza y funciones específicas de cada uno de los elementos del proyecto, para una mejor comprensión se separarán estos elementos en dos categorías:

Archivos del proyecto

Los archivos de proyecto son los siguientes: **myProyects, CoAN-008, myLibs, lib.dir, y CoAN-008.dcp.** No se agregarán comentarios al respecto de estos ya que su uso no difiere de lo ya expuesto en notas anteriores (CoAN-007, CoAN-006, CoAN-005).

Archivos fuente del proyecto

- **logs.lib:**
  Este módulo encapsula todas las funcionalidades propias del manejo de archivos que requerimos para nuestras funciones de log in, a saber: montaje / desmontaje de la FAT, formateo, apertura y cierre de archivos y directorios, escritura/lectura de archivos. También se controlan desde este módulo las excepciones relacionadas con la memoria miniSD.
- **flags.lib:**
  El módulo flags.lib incluye las definiciones correspondientes a los flags de control, señalización y sincronización que se utilizan en el proyecto, como así también una serie de macros que facilitan su manipulación. El curso del programa es determinado prácticamente por el estado de estos flags que son manipulados exclusivamente por el programa principal.
- **CoAN-008.c**:
  Archivo de programa de la aplicación, básicamente consta de un loop principal desde el cual son llamadas las rutinas para la simulación de los valores que luego son accedidos por el módulo *logs.lib* para la escritura de los archivos de log.
  Además se utilizan desde el loop principal las funciones para impresión de la interfaz con el usuario en el estándar output.

Nota de fin de sección

Una vez más dividimos el trabajo en dos secciones, para mayor conveniencia del usuario. En la primera parte se intentó explicar cualitativamente el proyecto y los fundamentos de su realización. Por lo tanto, en este punto, si hicimos bien el trabajo el lector debería estar convenientemente informado sobre el alcance y los objetivos propuestos y si así lo desea, podría recurrir al software que acompaña esta nota, para comprobar sin mas trámites su funcionamiento, y en el mejor de los casos, hasta incorporar alguna parte del código a su propio proyecto.
En cuanto a la segunda parte, se intenta explicar con cierto detalle la estructura y las particularidades del código, a fin de que este sea comprendido y pueda aprovecharse mejor (o quizá desecharse definitivamente), por lo tanto, lo que sigue, podría utilizarse más como material de consulta puntual que como material de lectura.

PARTE 2

Los módulos de software del proyecto

Vamos a pasar ahora, a describir el funcionamiento de los diferentes bloques que integran el proyecto. Por supuesto, conforme al objetivo que se ha declarado para esta nota, nos concentraremos mayormente en los aspectos relacionados con la implementación y uso del sistema de archivos.

Módulo principal

En cuanto al módulo principal intentaremos mostramos aquí a fines de no extendernos demasiado, solo el esqueleto y parte de los comentarios que aluden a una diferenciación de los bloques del programa y los bloque de código de mayor importancia:

```
//my libraries
#use flags.lib
#use logs.lib

//led defines and macros

//sw defines

//Interval sample for log process
#define SAMPLEINTERVAL  300
//variables for measures simulation

/*****************************************
*                  Printer functions
*****************************************/
void print_head(){
}
extern protected unsigned long SeqNumber;
void print_body(){
}

main()
{
   int channel;
   // restore any protected variables
   _sysIsSoftReset();
   // board functions initialization
   brdInit();
   //Flags initialization
   flags_init();
   print_head();

while(1){
      /*******************
       * SIMULATE SAMPLES
       *******************/
      costate
      {  // simulate AD samples for each channel
            for(channel = 0; channel <= 3; channel++)
                  ad_inputs[channel] = (rand()-0.5)/10 + 5567;
         //take QD simulated counter value
            qd_reading += (long)(10*(rand()- 0.5))%10;
            waitfor( DelayMs( 250 ) );
      }
```

Referencias a las libraries que creamos para este proyecto.

Bloque de definiciones y macros para utilizar fácilmente los leds y los switches.

Funciones de impresión en pantalla de los valores simulados y algunos datos de status.

Como estamos usando algunas variables declaradas como “protected” llamamos a esta función del sistema encargada de la recuperación de las variables en caso de fallo.

El resto son funciones de inicialización de los flags, para la placa de prototipos y de interfaz

Solamente a efectos de generar valores simulados para los procesos de log.

```
/*****************
 * DATALOG PROCESS
 *****************/
/**********************************
 * start / stop logging conditions
 **********************************/
 costate
 {
      if (!BitRdPortI(PBDR, S2_BIT) )//wait for switch S2 press
      {
            if( FLGLOGSTARTED ){
              puts("\n\nPreparando detencion de logs\n");
              FLGLOGSTARTED_RST;
            }
            else{
              puts("\n\nPreparando inicio de logs");
              FLGLOGSTARTED_SET;
            }
      }
      waitfor(DelayMs(300));
 }
/*******************************
*  start / stop logging actions
*******************************/
 costate
 { //Logs off -> off
   if( !FLGCHANGED(LOGSTARTED_bit) && !FLGLOGSTARTED )
      abort;
   //Logs on -> on
   if( !FLGCHANGED(LOGSTARTED_bit) && FLGLOGSTARTED )
   {
      waitfor(DelayMs(SAMPLEINTERVAL));
      if( !(FLGMEMNOINSERTED || FLGMEMFULL)  )
      {
            switch( log_insert() ) //New log
            {
                  case LOGMEMAVAILABLE:
                        break;
                  case LOGMEMNOINSERTED:
                        FLGMEMNOINSERTED_SET;
                        break;
                  case LOGMEMFULL:
                        FLGMEMFULL_SET;
                        break;
                  default:
                        break;
            }
      }
      else
      {  //If mem_no_error -> mem_error)
         if ( (FLGCHANGED(MEMNOINSERTED_bit) && FLGMEMNOINSERTED) ||
              (FLGCHANGED(MEMFULL_bit) && FLGMEMFULL)  )
         {  //unmount FAT
            if (log_filesystem_close() == SUCCESS )
                  FLGFILESYSTEMSTARTED_RST;
            //update flags after changed detected
            FLGLASTUPD( MEMNOINSERTED_bit );
            FLGLASTUPD( MEMFULL_bit );
         }//the user must solve the memory problem, and must restart the log
      process.
```

Usamos la detección de la presión del switch S2 para disparar el inicio/fin del proceso de log in.

A continuación, mediante un flag nos damos cuenta si tenemos que iniciar o para la operación.

Controlamos esto cada 300ms.

A partir de acá controlamos cuatro posibles transiciones de estados: off→on, off→off on→off, on→on, en cada uno de esos estados hay que hacer o verificar algo distinto y nos valemos de flags actuales vs. flags anteriores para decidir.

Recordemos que estamos dentro del proceso de log. Entonces on→on significa que estamos en "log in"

Intentamos escribir el archivo de log cada SAMPLEINTERVAL milisegundos

Si no hay errores a la vista...

Intentamos escribir. Luego chequeamos el resultado de la operación.

E informamos con flags según corresponda

Si hay errores de memoria de log..

Básicamente cerramos el filesystem y seteamos flags

```
        }
        abort;
        }
```

Si detectamos la transición off→on montamos el sistema de archivos, asegurandonos de que no haya sido montado ya, y de paso imprimimos el encabezado de la tablita de datos que mostramos en pantalla.

```
    //Logs off -> on
    if( FLGCHANGED(LOGSTARTED_bit) && FLGLOGSTARTED    )
    {  //Mount FAT if is unmounted
       if ( !FLGFILESYSTEMSTARTED )
       {  //if come from an exception, dont format the partition!!!.
             if ( log_filesystem_init( !FLGLOGGINGCURRENTLY ) == SUCCESS ){
                   FLGFILESYSTEMSTARTED_SET;
                   print_head();
             }
       }
```

La función de inicialización del file system lleva un parametro que indica si provenimos de una operación de loggin previa abortada por alguna falla, en ese caso no formatea la memoria durante la inicialización.

Luego de una inicialización correcta aún podemos tener inconvenientes con el sistema de archivos:
-memoria llena.
-memoria extraida
Debemos considerar esos casos y hacemos esto con el switch, en base a la respuesta de la función *log_mem_available*
Luego seteamos flags en consecuencia.

```
       switch( log_mem_available() ){
             case LOGMEMAVAILABLE:
                   FLGLASTUPD( LOGSTARTED_bit );//flag changed->update
                   FLGMEMNOINSERTED_RST;
                   FLGMEMFULL_RST;
                   //Set persistent indication of loggin state preventing
                   //formatting after power cuts
                   cfgSaveLoggingCurrently(1);
             break;
             case LOGMEMNOINSERTED:
                   FLGMEMNOINSERTED_SET;
             break;
             case LOGMEMFULL:
                   // if mem full, then memory is inserted now
                   FLGMEMNOINSERTED_RST;
                   FLGMEMFULL_SET;
                   FLGLASTUPD( LOGSTARTED_bit );//flag changed->update
             break;
          default:
             break;
       }
       abort;
    }
```

Esta función maneja la señalización correspondiente para saber cuando se está volviendo de un proceso de loggin interrumpido por una falla

Para finalizar este costate, controlamos la transición on→off, en la cual simplemente debemos cerrar el file system desmontando la partición FAT, de paso reimprimimos el encabezado en pantalla, lo cual es solo una cuestión estética para la demo.

```
    //Logs  on -> off
    if ( FLGCHANGED(LOGSTARTED_bit) && !FLGLOGSTARTED )
    {  //desmonto la FAT;
       if ( log_filesystem_close() == SUCCESS ){
             FLGFILESYSTEMSTARTED_RST;
             print_head();
       }
       FLGLASTUPD( LOGSTARTED_bit );//flag changed->update
       cfgSaveLoggingCurrently(0);//Reset persistent indication of loggin state
       abort;
    }
}
```

Concluimos guardando el estado de “no loggin”

```
        /***************************
         ALARM LED CONTROL
        ***************************/
        costate
        {
               if( FLGMEMFULL || FLGMEMNOINSERTED){
                      ALARMON;
                      abort;
               }
               else{
                      ALARMOFF;
                      abort;
               }
        }
        /***************************
        STATUS LED CONTROL,
        although, only blinking the led
        ***************************/
        costate
        {
               STATUSON;
               waitfor(DelayMs(750));
               STATUSOFF;
               waitfor(DelayMs(750));
               print_body();
        }

}
}
```

Verificamos si hay alguna alarma y activamos o no el LED asociado

Parpadeo del LED de status

Módulo flags.lib

```
#define LOGGINCURRENTLYMARK      0xAAAA
protected unsigned int loggincurrently;

static unsigned int flags_last;
static unsigned int flags;

#define LOGGINGCURRENTLY_bit          0
#define LOGSTARTED_bit                1
#define FILESYSTEMSTARTED_bit         2
#define MEMFULL_bit                   3
#define MEMNOINSERTED_bit             4
```

Esta es la variable no volátil (declarada como protected) que guarda el estado de log en curso para ofrecer cierta tolerancia a fallos.

El par de variables que contienen los flags

La posición que ocupa cada bit dentro de la variable contenedora

La macro siguiente nos devuelve el valor actual del flag, cero si está reseteado, > 0 si está seteado. Lo usamos de forma genérica `if(FLGnombreDeFlag){}` o `if( ! FLGnombreDeFlag){}`

```
#define FLGLOGGINGCURRENTLY          (flags & (1<<LOGGINGCURRENTLY_bit))
```

Estas macro nos ayudan a setear/resetear un flag con comodidad

```
#define FLGLOGGINGCURRENTLY_SET      (flags |= (1<<LOGGINGCURRENTLY_bit))
#define FLGLOGGINGCURRENTLY_RST      (flags &= (~(1<<LOGGINGCURRENTLY_bit)))
```

Lo mismo que acabamos de explicar mas arriba se aplica a cada uno de los flags que hayamos definido, siguiendo exactaente la misma filosofía:

```
FLGnombreDeFlag          consultar estado
FLGnombreDeFlag_RST      resetear estado
FLGnombreDeFlag_SET      setear estado
```

Y según vermos más abajo:

```
FLGCHANGED(flag_bit)     consultar si el flag cambió
FLGLASTUPD(flag_bit)     actualizar el estado anterior al estado actual
Siendo flag_bit igual a nombreDeFlag_bit
```

```
#define FLGLOGSTARTED               (flags & (1<<LOGSTARTED_bit))
#define FLGLOGSTARTED_SET           (flags |= (1<<LOGSTARTED_bit))
#define FLGLOGSTARTED_RST           (flags &= (~(1<<LOGSTARTED_bit)))

#define FLGFILESYSTEMSTARTED        (flags & (1<<FILESYSTEMSTARTED_bit))
#define FLGFILESYSTEMSTARTED_SET    (flags |= (1<<FILESYSTEMSTARTED_bit))
#define FLGFILESYSTEMSTARTED_RST    (flags &= (~(1<<FILESYSTEMSTARTED_bit)))

#define FLGMEMFULL                  (flags & (1<<MEMFULL_bit))
#define FLGMEMFULL_SET              (flags |= (1<<MEMFULL_bit))
#define FLGMEMFULL_RST              (flags &= (~(1<<MEMFULL_bit)))

#define FLGMEMNOINSERTED            (flags & (1<<MEMNOINSERTED_bit))
#define FLGMEMNOINSERTED_SET        (flags |= (1<<MEMNOINSERTED_bit))
#define FLGMEMNOINSERTED_RST        (flags &= (~(1<<MEMNOINSERTED_bit)))
```

Acá abajo, definimos la macro que nos amablemente nos dice si el flag indicado ha cambiado con respecto a su estado anterior.

```
//say if the flag was changed
#define FLGCHANGED(flag_bit)  ( ( flags_last & (1<<flag_bit) ) ^ ( flags & (1<<flag_bit) ) )
```

La macro que sigue, nos facilita la tarea de actualizar el estado anterior al estado que tiene actualmente el flag en cuestión.

```
//update the flag_last bit at current value
#define FLGLASTUPD(flag_bit)    ( flags_last ^= (1<<flag_bit) )
```

La macros que siguen abajo, son lo mismo que las anteriores, pero definidas de forma que puedan manejar genéricamente cualquier flag de cualquier variable que los contenga.

```
#define FLGCHANGED_GENERIC(bit, flags_last, flags_current)
( ( flags_last & (1<<bit) ) ^ ( flags_current & (1<<bit) ) )
//update the flag_last bit at current value
#define FLGLASTUPD_GENERIC(bit, flags_last)    ( flags_last ^= (1<<bit) )

void flags_init()
{
        flags=0;
        flags_last=0;
        if(loggincurrently == LOGGINCURRENTLYMARK)
                FLGLOGGINGCURRENTLY_SET;
        else
                FLGLOGGINGCURRENTLY_RST;
}
```

Esta es la función de inicialización, tenemos para destacar que durante la inicialización revisamos el estado la variable protegida `loggincurrently` Y establecemos el estado del flag correspondiente en consecuencia

```
void cfgSaveLoggingCurrently(unsigned char enable)
{
        if(enable){
                FLGLOGGINGCURRENTLY_SET;
                loggincurrently = LOGGINCURRENTLYMARK;
          }
          else{
                FLGLOGGINGCURRENTLY_RST;
                loggincurrently = ~LOGGINCURRENTLYMARK;
          }
}
```

Para finalizar con este módulo, miramos dentro de la función que maneja pa persistencia del estado de loggin para prevenir pérdidas de datos ante un fallo. Vemos que además de setear/resetear el flag de estado, también modifica el valor de la variable protegida con una marca especial.

Módulo logs.lib

Llegamos a los comentarios respecto del último módulo que nos queda abordar, y que es el módulo central de esta nota, ya que aquí se resuelve toda la complejidad que se nos pudiera presentar a la hora de manejar nuestro sistema de archivos. Se debe tener presente que cualquier información presente en esta nota puede ampliarse con el manual del módulo fat.lib del fabricante: http://www.rabbit.com/documentation/docs/modules/FileSystem/index.htm

A modo de introducción, podemos destacar que en este módulo realizamos las tareas de montaje y desmontaje de la partición FAT, apertura y cierre de archivos, lectura y escritura de archivos, creación de directorios y formateo de la partición, control de la tarjeta de memoria, etc.
Sin embargo, de acuerdo a las buenas prácticas de programación, solo exponemos al programa principal lo justo y necesario, de tal forma que la interfaz del módulo se reduce a las siguientes funciones que explicaremos con mayor detalle mas adelante:

- `log_filesystem_close`
- `log_filesystem_init`
- `log_insert`
- `log_mem_available`

Hecha la introducción, pasamos a comentar las partes mas importantes de la anatomía de esta library:

```
#define FAT_BLOCK
//#define FAT_DEBUG
#use "fat.lib"

#define SUCCESS 0
```

Definimos si trabajamos en modo de bloqueo o si por el contrario las llamadas a las funciones de la fat library serán no-bloqueantes. En nuestra aplicación elegimos el modo de bloqueo para mayor simplicidad.

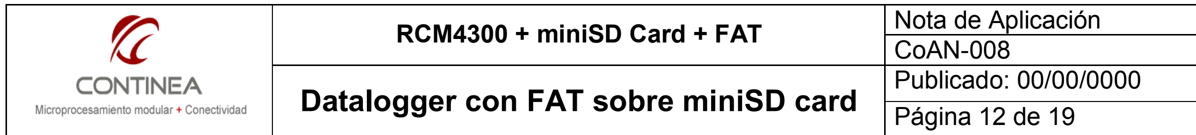
También definimos si habilitan los mensajes de debug para la library fat.lib.
A continuación invocamos la library fat.lib, que es central para el proyecto.

```
#define LOGMEMAVAILABLE       0
#define LOGMEMFULL
#define LOGMEMNOINSERTED
int log_mem_available( );
int log_filesystem_close();
int log_filesystem_init(unsigned
int log_insert();
```

Estos #defines son para que la función *log_mem_available()* Indique el estado de la tarjeta de memoria.

La declaración de las funciones que constituyen la interfaz del módulo

A continuación sigue el bloque de definiciones, que aclaramos que no será expuesto en su totalidad, sino que se explicarán solo las líneas que sean relevantes, por lo tanto a continuación no se reproduce exaustivamente el código presente en el módulo logs.lib al cual podrán encontrar como material adjunto a esta nota.



Las siguientes constantes representan la máxima longitud del encabezado de archivo y de cada línea de log respectivamente. Como vemos los límites están basados en constantes definidas anteriormente.

```
#define LOGHEADMAXLEN   NAMEMAXLEN + SEQMAXLEN +  DATETIMELEN  + SEPLEN*2 +
                        ENDLINELEN
#define LOGENTRYMAXLEN   ANALOGMAXLEN * (4) + QDMAXLEN +  TMMAXLEN +  SEPLEN*5 +
                        ENDLINELEN
```

La definición del tamaño máximo de archivo, seleccionada teniendo cuidado de no desperdiciar espacio de memoria, ya que la mínima parte de memoria asignable a un archivo es un clúster.

```
#define LOGFILEMAXLEN  0xBFFF  //49151  bytes for each log file (6 clusters)
```

La definición de la cantidad máxima de líneas que puede albergar un archivo de log.

```
#define LOGENTRYMAXNUM ( (unsigned int)( (LOGFILEMAXLEN - LOGHEADMAXLEN) /
LOGENTRYMAXLEN ) )
```

La definición del nombre del directorio que contendrá a los archivos de log.

```
#define LOGDIRNAME      "LOGS"
#define LOGDIRNAMELEN   4
```

La siguiente es una macro que provoca la impresión del nombre de archivo basado en el número de secuencia en formato hexadecimal. La impresión se genera en el buffer indicado, en el cual se ha puesto previamente el nombre del directorio que contiene al archivo, si corresponde. A continuación la definición de la extensión del archivo, etc.

```
#define LOGFILENAME(SeqNumber,buf)
      sprintf(&buf[LOGDIRNAMELEN],"\\%x",SeqNumber)
#define LOGFILEEXT      ".log"
#define LOGFILEEXTLEN   4
#define LOGFILENAMEMAXLEN     ( LOGDIRNAMELEN + SEQMAXLEN + LOGFILEEXTLEN )
```

El descriptor de archivos que necesitamos para manejar nuestros archivos, utilizamos un solo descriptor ya que nunca tenemos mas de un archivo abierto en forma simultánea.

```
static FATfile logFile;// When files are accessed, we need a FATfile structure.
```

Al estructura que representa y da soporte a nuestra partición FAT.

```
static fat_part *first_part; // Use the first mounted FAT partition.

protected unsigned long SeqNumber;
extern float ad_inputs[4];
extern struct config_st config;
extern qd_reading;

#define DEFAULT_NAME    "DEMO_ZERO"
```

El número de secuencia de los archivos de log, y a continuación las variables que contienen los valores simulados para registrar mediante las operaciones de log. Mas abajo, la definición del nombre que aparece en el encabezado de los archivos.

A continuación pasamos a comentar las funciones presentes en el módulo, aclarando que en general no se presentará el contenido de estas por razones de simplicidad y para no extender demasiado las explicaciones sobre el código, siendo que en general es más productivo compilar y ejecutar y en todo caso seguirlo en funcionamiento que leer texto explicativo.

```
void get_time( char * strdatetime)
{

}
```

Esta es solo una función accesoria para generar la fecha y hora que se imprimen en el encabezado de cada archivo

```
void log_build_head(char *buffer)
{
}
```

Se encarga de generar en el buffer pasado como parámetro el encabezado correspondiente a la apertura de un nuevo archivo de log.

```
void log_build_entry(char *buffer)
{

}
```

Genera una nueva línea de registro para ser almacenada en un archivo. Para esto accede a `ad_inputs[0-3]`, `qd_reading` y `SEC_TIMER` que es una macro del sistema que entrega la fecha y hora en formato UNIX . El registro lo imprime en el buffer pasado como parámetro

```
void log_file_name_get(char *buffer, unsigned long *SeqNumber)
{
  strcpy(buffer,LOGDIRNAME);
  LOGFILENAME(*SeqNumber, buffer);
  strcat(buffer,LOGFILEEXT);
}
```

Genera el nombre para un nuevo archivo de log, es decir cuando un archivo se llena y se debe cerrar para pasar al siguiente. Nótese como aprovechamos las contantes y macros definidas y explicada más arriba. Todo se devuelve en el buffer pasado como parámetro.

```
int log_fileIsOpen(FATfile *myFile)
{
      if ( myFile->state == FAT_FILESTATE_NOTOPEN )
            return 0;
      else
            return 1;
}
```

Responde si el archivo referenciado por el puntero a una estructura FATfile está i no abierto. Tanto la estructura como la constante utilizada en el if están definidas por el módulo fat.lib

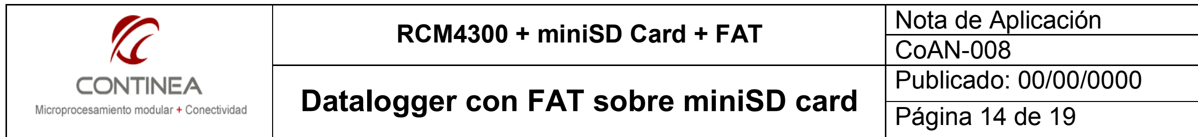
La función siguiente abre el archivo con el nombre especificado (o lo crea si este no existía), siempre que el archivo apuntado por el parámetro `myFile` no estuviera ya abierto. Junto a cada argumento de la función fat_open puede verse su descripción.

```
int log_file_open(char *FileName, FATfile *myFile)
{
      int rc;
      long prealloc;// Used if the file needs to be created.
      prealloc = 0;//Do not pre-allocate any more than the minimum necessary amount of storage.

      //check if file is yet open
      if ( log_fileIsOpen(myFile) )
        return 0;

      // Open (and maybe create) it...
      rc = fat_Open(
                  first_part,  // First partition pointer from fat_AutoMount()
                  FileName,    // Name of file. Always an absolute path name.
                  FAT_FILE,    // Type of object, i.e. a file.
                  FAT_CREATE,  // Create the file if it does not exist.
                  myFile,      // Fill in this structure with file details
                  &prealloc    // Number of bytes to allocate.
                 );

      if (rc < 0) {
      ;//printf("fat_Open() failed with return code %d\n", rc);
       }
      return rc;
}
```



Esta es la función que utilizamos para escribir nuestro archivo. Suministramos el descriptor de archivo y el puntero al buffer de datos al llamarla. Luego, al llamar a fat_write dentro de nuestra función debemos pasarle además la longitud de línea que vamos a escribir.

```
int log_file_write(char *buffer, FATfile *myFile)
{
   int rc;
   rc = fat_Write(
                  myFile,          // File, as set by fat_Open()
                  buffer,          // Some data to write.
                  strlen(buffer)   // Number of characters to write.
                 );
   if (rc < 0)
   {
      printf("fat_Write() failed with return code %d\n", rc);
      //do something with the dead
   }
   return rc;
}
```

La función `log_file_space` verifica el espacio disponible en el archivo conforme al límite que establecimos para este (`LOGFILEMAXLEN`), pues, de no quedar espacio libre (siempre lógicamente hablando) debemos cerrar el archivo en cuestión y abrir el siguiente.

```
#define LOGFILEFULL           0
#define LOGFILEHALFEMPTY      1
#define LOGFILEEMPTY          2
int log_file_space( )
{
    unsigned long length; int ret;

    if ( (ret = fat_FileSize( &logFile, &length ))!= SUCCESS )
      return ret;

    if (length == 0)
      return LOGFILEEMPTY;
    if( (LOGFILEMAXLEN - length) > LOGENTRYMAXLEN    )
      return LOGFILEHALFEMPTY;
    else
      return LOGFILEFULL;
}
```

Estas son las constantes figurativas de retorno.

Aprovechamos la función de la library `fat_FileSize()` que nos devuelve en un puntero a *long* el tamaño del archivo.

Una vez que tenemos el tamaño actual del archivo, comprobamos si cabe una línea más

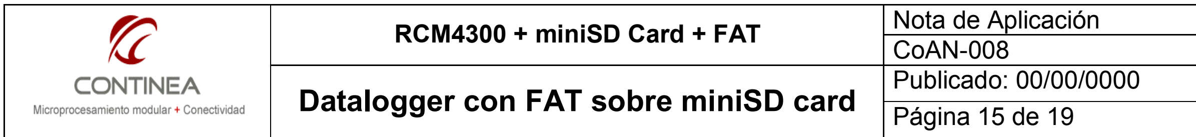
Las constantes y macros que siguen se utilizan exclusivamente por la función`log_mem_available()` y tienen la finalidad de facilitar la comprobación del espacio disponible en bytes dentro de la memoria SD con independencia de la capacidad de la tarjeta de memoria que haya insertado el usuario.

```
#define LOGMEMBYTESBYSECTOR   512

#define LOGMEMRAWSECTORS_128MB          (134217728  / LOGMEMBYTESBYSECTOR )
#define LOGMEMRAWSECTORS_256MB          (268435456  / LOGMEMBYTESBYSECTOR )
#define LOGMEMRAWSECTORS_512MB          (536870912  / LOGMEMBYTESBYSECTOR )
#define LOGMEMRAWSECTORS_1024MB         (1073741824 / LOGMEMBYTESBYSECTOR )

#define LOGMEMSECBYCLUSTER(size)        (size/32)
#define LOGMEMSECBYCLUSTER_DFLT         16

#define LOGMEMBYTESFREE(freeClusters, sec_by_clus)    ( freeClusters *
sec_by_clus * LOGMEMBYTESBYSECTOR )
```



Valiéndose de las constantes y macros que acabamos de comentar, la función que sigue le indica al programa principal es estado físico de la tarjeta de memoria, a saber: disponible, no insertada, o llena.

Función de presente en las API para memoria SD card provista por Rabbit. No indica si la tarjeta está colocada.

Otra función de la API para SD card que nos devuelve la cantidad de sectores totales que contiene nuestra tarjeta.

```
int log_mem_available( )
{
    unsigned long sectors, freeClusters;
    int sec_by_clus;

      if( !sdspi_debounce( &SD[0] ) )
            return LOGMEMNOINSERTED;

      sectors = sdspi_getSectorCount( &SD[0] );

      if ( sectors > 0 && sectors <  LOGMEMRAWSECTORS_128MB )
            sec_by_clus = LOGMEMSECBYCLUSTER(128);
      else if  ( sectors >  LOGMEMRAWSECTORS_128MB && sectors <
            LOGMEMRAWSECTORS_256MB )
            sec_by_clus = LOGMEMSECBYCLUSTER(256);
      else if  ( sectors >  LOGMEMRAWSECTORS_256MB && sectors
            LOGMEMRAWSECTORS_512MB )
            sec_by_clus = LOGMEMSECBYCLUSTER(512);
      else if  ( sectors >  LOGMEMRAWSECTORS_512MB && sectors <
            LOGMEMRAWSECTORS_1024MB )
            sec_by_clus = LOGMEMSECBYCLUSTER(1024);
      else
            sec_by_clus = LOGMEMSECBYCLUSTER_DFLT;
```

En este espantoso if() comparamos la cantidad de sectores totales de la tarjeta insertada con los valores calculados a priori, a fin de determinar que memoria tenemos (128,256,512 o 1GB)

`fat_Free` indica cuantos clústers libres quedan en el sistema de archivos

El tamaño de la memoria va a determinar en forma directamente proporcional la cantidad de sectores por clúster.

```
      freeClusters = fat_Free( first_part );
```

Sabiendo que memoria tenemos, sabemos cuántos sectores hay por cluster. Sabemos a priori cuántos bytes por sector tenemos. Ahora también sabemos cuántos clústers quedan libres
Ya podemos entonces decirle al programa principal si queda o no espacio disponible en la memoria mediante la macro `LOGMEMBYTESFREE`.

```
      if( LOGMEMBYTESFREE(freeClusters, sec_by_clus) > LOGFILEMAXLEN)
        return LOGMEMAVAILABLE;
      else
        return LOGMEMFULL;
}
```

La siguiente función es de vital importancia para nuestra simple aplicación, ya que se encarga de gestionar el esquema de archivos secuenciados y de tamaño fijo, debiendo saber cuándo hace falta estrenar un nuevo archivo, lo cual requiere la construcción de un encabezado, o cuándo el archivo se puede seguir usando. Además, la pobre debe considerar que el tamaño de la memoria SD puede ser variable, y que esta puede no estar presente, o puede estar llena. Como vemos, esta función es central en la library que construimos pero se vale de las funciones precedentes apra realizar todas esats comprobaciones.

```
int log_file_selector( )
{
      char buffer[LOGHEADMAXLEN+1];
      int ret;
      ret=0;
      buffer[0]='\0';
```

Tras haber reservado un buffer lo primero que hacemos es obtener el nombre del archivo vigente, habíamos definido una función para ello.

```
      log_file_name_get(buffer, &SeqNumber);//get the current file name.
```

Debemos abrir el archivo, así que usamos la función que ya habíamos creado, pasamos además del buffer el puntero al descriptor de archivo,(logFile) que en realidad es clobal, pero por generalidad es mejor así.

```
 if ( ( ret=log_file_open(buffer, &logFile) ) == SUCCESS )
 {
```

Ya abrimos el archivo, ahora hay que saber si tenemos espacio dentro. Como fuimos previsores, nos beneficiamos de una función que ya creamos antes. Entonces, si resulta que hay espacio en el archivo, salimos, lo cual indicará a la función que nos llame que puede escribir nomas.

```
    if ( ( ret = log_file_space() ) == LOGFILEHALFEMPTY )
              return SUCCESS;
```

Si el archivo se nos llenó (LOGFILEFULL), esto nos va a dar un poco mas de trabajo: Primero hacemos una sincronización del cahé, para pasar al soporte físico todo lo escrito hasta ahora en el caché.

```
    if ( ret == LOGFILEFULL )
    {  //No more space.
      fat_SyncPartition( first_part )[1];//Cache flushing  data to update device
```

Ahora cerramos el archivo que ya estaba lleno, incrementamos el número de secuencia y generamos nuevamente el nombre a partir de este.

```
      fat_Close( &logFile );            //close old file
      SeqNumber = (SeqNumber == SEQMAXVAL )? 0: (++SeqNumber);//new seq number
      log_file_name_get(buffer, &SeqNumber);     //new file name
```

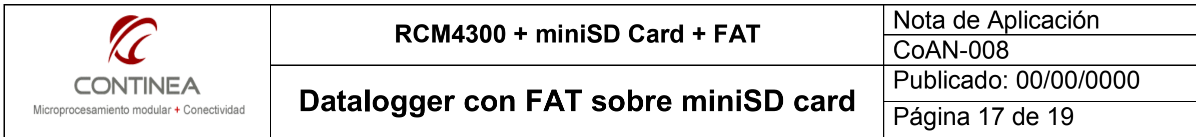
A continuación nos aseguramos de disponer de espacio en la memoria SD, y abrimos el nuevo archivo con su nombre correspondiente.

```
       if ( ( ret=log_mem_available() ) != LOGMEMAVAILABLE )
              return ret;//return becose mem is full or not inserted
       if ( ( ret=log_file_open(buffer, &logFile) ) != SUCCESS )
              return ret; //open new file
    }
```

Ahora que tenemos archivo nuevo, creamos el encabezado, con la función que definimos hace rato para esto y seguidamente lo escribimos, retornando finalmente a la función llamadora.

```
    log_build_head(buffer);                  //buil log file head
    log_file_write(buffer, &logFile);          // and write it.
 }
 else
        return ret;
 return SUCCESS;
}
```

[1] IMPOTANTE: Recordar que normalmente vamos a trabajar por cuestiones de eficiencia con un caché de escritura, lo cual implica que que hasta que no realicemos una sincronización del caché con el medio físico, no tendremos salvados los cambios realizados desde la última sincronización, y estos se perderían en caso de un fallo o cierre insadecuado del sistema.
Las operaciones que efectúan una sincronización del caché son las siguientes:
fat_UnmountPartition, fat_UnmountDevice, fat_Close, fat_SyncFile, fat_SyncPartition



Esta es en realidad la función que más nos interesa cuando vemos a la library desde afuera. Esta función es la punta del iceberg que oculta toda la complejidad que acabamos de describir. Maneja un buffer interno que es pasado como parámetro sucesivamente por el resto de las funciones hasta escribirse finalmente en el archivo.

```
int log_insert()
{
      int ret;
      static char buffer[LOGENTRYMAXLEN+1];
      buffer[0]='\0';

      if( ( ret=log_file_selector() ) != SUCCESS )
            return ret;

      log_build_entry(buffer);//armar la registo
      fat_Seek (&logFile, -1, SEEK_END);//Posicionar cursor al final del archivo
      log_file_write(buffer, &logFile);
      return SUCCESS;
}
```

Si `log_file_selector` devuelve `SUCCESS` se construye la línea de log actual.

Luego buscamos la última posición dentro del archivo

Finalmente, después de tantas vueltas, escribimos nuestra inocente línea de log , retornando con un feliz SUCCESS

Otra función visible desde afuera: la de inicialización del sistema de archivos. Requiere un parámetro utlizado para provocar o evitar el formateo de la memoria durante la inicialización. También el número de archivo es reseteado.

```
int log_filesystem_init(unsigned char clear)
{
      int rc,i;
      word flag;
      if( clear )
      {//if clear is TRUE the partition is formated unconditionally
            flag= (FDDF_UNCOND_DEV_FORMAT | FDDF_UNCOND_PART_FORMAT |
                        FDDF_MOUNT_DEV_0 | FDDF_MOUNT_PART_0);
            SeqNumber=0;//and sequence number is reset
            if(!FLGMEMNOINSERTED)
                  puts("\nSistema de archivos:\nBorrando memoria\nEspere..." );
      }
      else
      {
      //If clear is not activated, the FAT mount normally and sequence number continues
            flag=FDDF_USE_DEFAULT;
            if(!FLGMEMNOINSERTED)
                  puts("\nSistema de archivos:\nMontando FAT\nEspere..." );
      }

      // Auto-mount the FAT file system
      rc = fat_AutoMount(flag);

      // Scan the populated mounted partitions list to find the first mounted partition.
      first_part = NULL;
      for (i = 0; i < num_fat_devices * FAT_MAX_PARTITIONS; ++i) {
            if ((first_part = fat_part_mounted[i]) != NULL) {
                  // found a mounted partition, so use it
                  break;
            }
      }
```

Consecuencia del parámetro `clear` construimos el parámetro `flag` que le indicará a fat_AutoMount si debe formatear o no la memoria SD.

Luego de utilizar `fat_AutoMount` para montar la o las particiones, se selecciona una de ellas, que en nuestro caso es la única para a continuación realizar una serie de chequeos de error:

```
      // Check if a mounted partition was found
      if (first_part == NULL) {
            // No mounted partition found, ensure rc is set to a FAT error code.
            rc = (rc < 0) ? rc : -ENOPART;
      }
      else
      {
            // It is possible that a non-fatal error was encountered and reported,
            // even though fat_AutoMount() succeeded in mounting at least one
            // FAT partition.
            printf("\nfat_AutoMount() succeeded with return code %d.", rc);
            // We found a partition to work with, so ignore other error (if any).
            rc = 0;
      }

      if (rc < 0)
      {
         if(!FLGMEMNOINSERTED){
            if (rc == -EUNFORMAT)
                  printf("\nDevice not Formatted, Please run Fmt_Device.c\n");
            else
                  printf("\nfat_AutoMount() failed with return code %d.\n", rc);
            return rc;
      }
   }
```

Si llegamos acá, es porque todo está en orden, así que lo que hacemos es crear el directorio que contendrá los archivos de log, si es que no existía todavía. A continuación cerramos el directorio abierto.

```
   // Open Log dir(and maybe create) it...
   rc = fat_Open(
                 first_part,  // First partition pointer from fat_AutoMount()
                 LOGDIRNAME,  // Name of file.  Always an absolute path name.
                 FAT_DIR,     // Type of object, i.e. a file.
                 FAT_CREATE,  // Create the file if it does not exist.
                 &logFile,    // Fill in this structure with file details
                 NULL         // Number of bytes to allocate.
                );
  //Close dir recently open.
  rc = fat_Close(&logFile);

  //clear state for logFile
  logFile.state=0;
```

Nos aseguramos de que el estado de nuestro único descriptor de archivo indique que está cerrado

```
  if(!FLGMEMNOINSERTED)
  {
      if (rc < 0)
            printf("\nfat_Open() failed with return code %d", rc);
      else
            puts("\nOperaciones de inicializacion culminadas con exito");
   }
  return rc;
}
int log_filesystem_close()
{
      int rc;
// Since we are using blocking mode, it will not return until it has closed all files and
//unmounted the partition & device.
   rc= fat_UnmountDevice(first_part->dev);
   if(rc==SUCCESS)
         printf("\nfat_UnmountDevice() succeeded with return code %d\n", rc);
   return rc;
}
```

Para finalizar, vemos la última función que forma parte de la interfaz del módulo, la cual simplemente desmonta la partición y esto, realiza un volcado automático del caché, por lo tanto nos aseguramos de realizar un correcto cierre del sistema de archivos.

El proceso de log in

El procedimiento de log-in está resuelto en su totalidad por la función `log_insert()` y esta constituye el proceso principal de la aplicación, ya que este proceso accede a la memoria de almacenamiento masivo mediante las facilidades que nos ofrece el módulo fat.lib y realiza todas las funciones de control necesarias. Sin embargo, es probable que desde la perspectiva del código fuente, pueda parecer algo confuso el modo en que esta tarea se lleva a cabo.
Es por eso que creimos que en algún caso podría ser útil la visualización de un esquema de bloques que revele en forma clara la estructura de la secuencia de llamadas a función del proceso en cuestión: