Datalogger FAT32 Design Document

The Rantings of a Duck ... on optimized FAT32 ...

Background

The Datalogger FAT32 library is designed to implement background file-oriented read and write functionality for a FAT32 filesystem. FAT16 is possible, but not planned at the moment.
The code will be more optimized for writes, as that is what we expect to do most of the time.

Definitions

FAT (File Allocation Table), **Directory Table** as defined on Wikipedia
Allocation Unit is datalogger-specific - it is the amount of sectors (or clusters) that is preallocated every time a preallocation is done. This will be a decently sized number, my guess is around the cluster size squared.

Write Considerations

A file-oriented write is not as simple as shifting data sequentially to the SD card - space needs to be allocated as the file grows larger, and the FAT and Directory Table needs to be updated every once in a while. Those operations take time for the SD Card, and may cause a backlog (or even overflow) of user data pending writes. The code will attempt to spread out those operations to avoid taking too much CPU time or SD SPI time at once, but no guarantees.
The other option is a completely preallocated disk, but that requires the discipline to run a preallocation program on a computer before inserting the disk in the car - which is not too practical.

Proposed Overview

Buffering

Likely, the code will involve 4 buffers:

- 2 DMA buffers holding user data (512B of user data + 3B for block header)
  - These two buffers alternate between being filled with data by user code and being shifted out through the SPI line using DMA.
- 1 DMA buffer holding file system data (512B of FS data + 3B for block header)
  - This buffer holds things like Directory Table entries or FAT entries which are to be updated for preallocation. The preallocation processing (filling out the buffers) and write (writing to SD card) is spread out between user data writes to minimize user data backlog and the probability of data loss.
- 1 RAM buffer holding backlogged user data (up to 8kB user data in a circular buffer)

User Code

1. The user initializes the SD Card.
2. The user initializes the file system.
3. Optionally, the user looks for and reads a configuration file on the SD card, updating code settings accordingly.
   a. If we really feel like it, it might be possible to make a SD card bootloader, and have the chip flash itself from data on the SD card. But this is completely unnecessary currently.
4. The user creates a file - during file creation, the first Allocation Unit is preallocated. This may either be a foreground or background operation.
   a. By now, it is probably about 5 seconds from startup. Since it is impossible to buffer 5 seconds of heavy data, no attempt will be made to do so - and CAN receiption starts after a file is created and the first Allocation Unit is preallocated.
5. The user repeatedly calls FAT32 Write File and FAT32 Tasks. It is recommended to call FAT32 Tasks at least once per main cycle.

Library Code

- FAT32 tasks mostly deals with pre-allocating the next Allocation Unit.
- The following can happen during a FAT32 Tasks call:

    - Fill the filesystem data buffer with data relevant to preallocation.
    - If the DMA module is not busy and the buffers are relatively empty (and possibly using other heuristics), commit the filesystem data buffer to the SD card in the background.
        - The above may need to be several times during the course of writing an Allocation Unit as there are multiple sectors in the FAT and Directory Table which need to be written.
        - The above may also happen during a FAT32 write call if it reaches the end of an Allocation Unit and the next Allocation Unit has not been pre-allocated. That is only likely in the case of heavy load, and would likely result in the loss of user data.
    - Nothing. In fact, this happens most of the time, and data filling is only done every once in a while to minimize the effect of overhead and spread it out evenly among data writes.
- The following can happen during a FAT32 Write call:
    - Always, the data to be written will be copied over to the active user data buffer.
    - If the active user data buffer fills and the DMA module is ready, that buffer is shifted out to the card through DMA, and the two user data buffers swap roles (between active buffer and data out buffer).
    - If the active user data buffer fills and the DMA module is not ready, the additional data is written to the backlogged user data buffer in RAM. If that fills, an overflow flag is set and the data is lost.
    - If the active user data buffer is relatively empty and the DMA module is ready, the code may begin the process of committing the contents of the file system buffer to disk. This involves:
        - Terminating the active Multiple Block Write operation. This takes several milliseconds, and happens in the background.
        - Committing the data using a Single Block Write operation. This takes several milliseconds, and happens in the background.
        - Restarting the Multiple Block Write operation to resume writing user data.
        - By happening in the background, I mean the function returns before the operation completes, and a state machine keeps track of progress and either uses interrupts to start the next phase (preferable) or waits for the next call to FAT32 Write.
        - The first version will likely use polling instead of interrupts, as it results in code which is more cleanly separated from the lower level SD Card code.

<u>**Proposed Functions**</u>

Above-the-line Abstraction (user-accessible function)

- **FAT32 Initialize**
    - Initializes the data structure for the FAT32 file system - reads the contents of the card to determine byte offsets, FAT pointers, etc...
- **FAT32 Create File**
    - Creates a record in the filesystem for a new file. No space is allocated for the file, but the file is opened and the file data structure is returned.
    - In the sequential optimized version of the code, this code will also preallocate one allocation unit.
- **FAT32 Open File**
    - Opens an existing file - attempts to find the file record in the file table, and if it is there, attempts to create a file data structure so that the file can be accessed.
    - Once the file is open, the file data structure points towards the beginning of the file.
- **FAT32 Seek**
    - Seeks to a position in a file - either an offset from the beginning or an offset from the end.
- **FAT32 Write File**

    - Writes data to a file. More correctly, it writes data to a buffer, and when the buffer is filled it requests the SD SPI code to write that buffer to the SD card in the background.
    - In the first version of the code, this will be heavily optimized towards writing LARGE, sequential files. In a future version for a public library release, that functionality may be moved to a more specific function, say, FAT32 Write Optimized, while this function may be more general.
    - See Write Considerations and Proposed Overview for more information.
- **FAT32 Tasks**
    - This function is called every once in a while and does tasks that FAT32 needs. This function is not supposed to take a ridiculous amount of time, but may take some time. A completely random guess puts the maximum time at 1000-5,000 instructions, which is on the order of hundreds of microseconds. Most of the time, it will be significantly less.
    - Based on current load, does background functions like FAT32 Preallocate Nocommit, FAT32 Preallocate Commit, and FAT32 File Record Commit
    - See Write Considerations and Proposed Overview for more information.
    - This shares a lot of functionality with FAT32 Write File, and may be merged - Tasks may be called with a call to FAT32 Write with null write data.

Below-the-line Abstraction (internal use functions) (see Write Considerations for more information)

- **FAT32 Preallocate Nocommit**
    - Fills out the file system buffer with the contents of the FAT for the next Allocation Unit.
- **FAT32 Preallocate Commit**
    - Commits the file system buffer to disk.
- **FAT32 File Record Nocommit**
    - Fills out the file system buffer with the contents of the Directory Table for the next Allocation Unit.
- **FAT32 File Record Commit**
    - Commits the file system buffer to disk.