Datalogger Software Goals

Lowest Layer: Raw SD - SPI Interface

- Optimized SPI Interface to the SD Card, likely using DMA in both foreground (blocking) and background (parallel) modes to achieve maximum SPI throughput.
- This works work raw byte ranges, lengths, and (possibly) sectors.
- Proposed functions:
  - SD SPI Write
    - Writes data to the SD Card. Blocks until completed.
    - Writes are required to be block (usually 512 bytes) aligned.
  - SD SPI Read
    - Reads data from the SD Card. Blocks until completed.
    - Reads are required to be block (usually 512 bytes) aligned.
  - SD SPI Background Write
    - Initiates a DMA request to write data to the SD Card. This returns as soon as the request is initiated, so the CPU can continue with other operations while SPI transfers are happening in the background.
    - It is the responsibility of upper layers to ensure no conflicts happen - so upper layers will probably implement a state machine to keep track of whether transfers over the SPI bus. Behavior is undefined if there is a conflict - but it most likely will result in a very confused SD card.
  - SD SPI Background Read is impractical, since a lot of processing happens between blocks, and in any case, we are mostly concerned with optimizing writes. This may be an area of future research, however.
- The expected performance with DMA will probably be around 1MByte/s. This is assuming one write after the other, which may be possible if background transfers with DMA is used.
- Approximate time: Two days' work - need to work on DMA.

Middle Layer: FAT32 File System Layer

- This abstracts away the SD SPI layer, and allows upper layers to work with files.
- This implements just the FAT32 filesystem. FAT16 may be a later project, but is not currently planned.
- Long file names probably will not be supported. Microsoft apparently has a patent on parts of FAT32, including long file names.
- Proposed serial functions:
  - FAT32 Initialize
    - Initializes data structures for the FAT32 file system - such as getting byte offsets of FAT32 structures like the file table and reading header information. This is more work than it looks like.
  - FAT32 New File (and preallocate)
    - Creates a new file, preallocating a specified number of bytes. This is also a good deal of work.
  - FAT32 Write File (and preallocate as necessary)
    - Writes data to a file. If the data spans past the end of the file, the library will preallocate another block of a specified number of bytes. This avoids hammering the file table each time we write a FAT allocation unit's worth of bytes.
- Approximate time: Several days' work. Perhaps a week?
- Proposed parallel functions:

    - Dual buffering - while a buffer containing data is transferred to the SD Card through SPI, a second buffer is being filled with user data.
    - FAT32 New File, Parallel, Dual-buffer
        - Creates a new file, preallocating a specified number of bytes.
        - Also allocates dual-buffered DMA space on the microcontroller to allow background writes.
    - FAT32 Write File, Parallel, Dual-buffer
        - Writes data to a file using a DMA request. Returns while the rest of the transfer happens in the background (non-blocking code). Also swaps the user buffer with the buffer being shifted onto the SPI line.
- Approximate time: Another two days' work to parallelize it. But unexpected delays are expected, and the main problem would be designing data structures and code to be parallel friendly. Interrupts may be an option to consider for performance, but also add to complexity and correctness issues.
- Proposed advanced functions:
    - Triple buffering - while there is only space for approximately 2 buffers in DMA RAM, there is another 14kB of non-DMA RAM. This could potentially be used as an overflow buffer when both DMA buffers are full. However, to write this data to the SD Card using DMA, this memory must be moved to DMA RAM before the DMA SPI transfer request can happen. Likely, this can be accelerated using 16-bit operations, and the cost is probably 2 cycles per 16 bit move (1 cycle load into register, 1 cycle store into DMA RAM).
- Approximate time: Unknown, not planned.

Higher Layer: "Glue Logic"

- This is the stuff that goes into main()
- This includes any user-interface code through the UART console and reading buttons.
    - Currently, the GPIO expander is acting funny, and buttons are not being read correctly. If there is a run #3, and surface-mount is allowed, moving to a 40-pin PIC device is preferable to completely eliminate the external GPIO expander and move IO functions on-chip. This also saves a considerable amount of processing time, since I2C operations cannot happen over DMA and forces the CPU to wait.
- Specifying a file format. Currently, we are using CSV, which is simple but inefficient (approximately 3 bytes written for every byte received). A move to a binary file format (which thus means work on parser software) would increase efficiency and allow more data to be logged.
- Basically, reading the CAN network, processing the data, and requesting file system writes.
    - Also, writing any other data which we may want logged.
- Detecting power-down conditions (measuring voltage on the +12v bus), and flushing file buffers as necessary to allow for the system to cleanly dismount the SD card and minimize data loss.

Future Work

- Reliability. It's painful, it's confusing, it's spaghetti-code, but it's a must.
- Datalogger on the BRAIN
    - Preliminary assessment: unlikely:
    - A look at the ATMega324p datasheet indicates that there is no FIFO for the SPI bus, and thus strict timing requirements are needed to implement any significant amount of

pipelining. If those requirements are not met, there will be issues with received bytes being discarded.

  - The ATMega324 also does not have DMA, and all SPI operations must happen in the foreground. While there is an allowed delay during a write-wait sequence, logic analyzer output shows that those times are insignificant.
  - The BRAIN uses an external CAN module with only a 2-message-deep receive buffer. When we tested the PIC device (with an internal 28-deep receive buffer) on Sunday, the CANbus at full speed caused significant overflows even in the pipelined code. Background transfers using DMA will probably resolve the issue, but none of those options are available on the ATMega324.
    - Do note that the inefficient .csv file format implemented caused a 3x overhead. Although, in the long run, I don't think eliminating that overhead using a efficient file format will break even with speeds the ATMega324 is capable of.
  - It may be worth a try, but I don't think we are going to get any higher numbers than the benchmarks reported by the Roland-Riegel library. The limited pipelining may improve numbers slightly, but not significantly.

Public Release (timeframe: near summer vacation):

- Chances are, we would publically release the optimized code someday, and someone would find it useful. Sparkfun already sells products based on the Roland-Riegel library. We can achieve much higher data rates, although the techniques used here are not very Arduino-friendly for the reasons discussed above.
- I would propose the FAT32 filesystem be licensed under the 3-clause BSD, while the Datalogger code as a while be licensed under GPLv3 or GPLv2. Yes, open-source is awesome.