**Datalogger v0.2 Architecture and Design Document**

**Overview:**

The Datalogger software consists of these cooperatively multitasked components: FAT32 Library, SD over DMA SPI library, CAN library, and the UI library. The idea is that each one is called during a loop, does what it needs to, and returns so that other components can execute, giving the appearance that everything is happening in parallel.

Note – while the SD Card operations are supposed to happen through DMA, it uses a combination of polling and interrupts to advance the operations since the microcontroller can’t generate an interrupt when certain data is received (such as at the end of a busy signal from the card). Future work might include bypassing this issue by generating an interrupt from a logic level change.

**Components:**

**SD-SPI through DMA Library**

Purpose: Abstract away the details of the SD Card Block Read / Write and Initialization operations. In the future, this should allow multiple cards connected to a single device, although for now the only part of that implemented is having an SDCard data structure storing all card-specific data and which is passed to all functions. Multiple access (in the future) will likely be done through a contention mechanism, where the first to get the SD Card in the idle state gets to go through with the operation.

Rough overview of states:

* Dismounted states:
  + No card
  + Card inserted
  + Card initializing, and related sub-states
* Mounted (initialized) states:
  + Card idle
  + DMA Single Block Read, and related sub-states
  + DMA Single Block Write, and related sub-states
  + DMA Multiple Block Write, and related sub-states

Rough overview of functions:

* SD DMA Get State – Returns the current state.
* SD DMA Initialize – Begin the initialization process through DMA.
* SD DMA Get Initialize Result – Returns the result of the initialize operation, or a busy signal. This should be called periodically to perform tasks blocked by polling.
* SD DMA Read Block – Does a single block read at the specified sector over DMA.
* SD DMA Multiple Block Write (MBW) functions:
  + SD DMA MBW Begin – Begins a multiple block write operation. No blocks are sent, this only sends the command to begin a multiple block write operation. The SD card data structure will change state to indicate this operation has completed and the card is ready to begin accepting block (which is usually when this function returns)
  + SD DMA MBW Send Block – Sends a block (512 bytes) of data to the SD card through DMA. Once the operation is complete, it writes the result into a pointer passed to the function.
  + SD DMA MBW Terminate – Ends the multiple block write operation, committing any blocks buffered into the card to NVM. In most implantations, this operation takes a while and the function will return before it is complete. Once the operation is complete. It writes the result into a pointer passed to the function.
* SD DMA Single Block Write – For a single block write operation, which is preferred when doing random-access writes such as when updating the file table or directory table.
* SD DMA Get \* Result

Rough overview of event functions – these are called when an event happens and the function code is to be modified as the application dictates. In the future, this may be replaced by a hook or callback. The current purpose of these is to blink a status LED.

* SD DMA On Block Read – called whenever a block is read from the SD card
* SD DMA On Block Write – called whenever a block is written from the SD card

**FAT32 Library**

Purpose: Abstraction of FAT32 filesystem operations including initialization and getting filesystem information. All filesystem instance-specific data should be stored into a FAT32 data structure – this also allows multiple filesystems to be open at once.

This (and the Optimized File Library) should completely encapsulate the SD SPI functions.

Rough overview of states:

* Uninitialized
* Initializing
* Initialized

Rough overview of functions:

* FAT32 Get State
* FAT32 Initialize – initializes the FAT32 file system data structure with data from a SD Card. Once this is done, files operations can begin. In the first version, this will likely be a blocking function while reads are happening.
* FAT32 Get Initialization Result – Returns the result of the FAT32 initialization operation, or a busy token.

**Optimized File Library**

Purpose: A FAT32 File operations library optimized for long, contiguous write (or read, in the future) operations. Write operations are buffered – whole blocks are send to the SD card at a time through DMA, and most of the operation happens in the background. All file-specific data (and bookkeeping cache variables) should be stored in an Optimized File data structure. Instead of using states, the file will use bookkeeping variables (such as buffer length, current position, etc) to determine what it does.

This (and the FAT32 Library) should completely encapsulate the SD SPI functions.

Rough overview of functions:

* File Open – “Opens” a file on disk for read or write operations.
* File Create – Creates (and opens) a new file on the disk.
* File Do Atomic Write – Atomically writes (either the entire write succeeds or fails, will not do a partial write) a block of data to the file. Returns the number of byte written (which should be either the input length or 0 on failure). The block of data must be less than 512 bytes in length for now – this allows nice buffering.
  + The definition of a successful write is that the data has been successfully enqueued. Writes to the disk are not guaranteed in case of an SD Card error or unexpected shutdown due to power down or an error condition.
* File Tasks – Called periodically to perform tasks necessary to the file write operation. Likely, this will involve sending blocks to the disk or updating the filesystem.

Statistics functions (future work):

* Get maximum DMA blocks used: Returns the maximum number of full DMA buffer blocks used at a time
* Get maximum overflow buffer length: Returns the highest length of overflow buffer bytes used
* Reset statistics: Duh!

**User Interface**

Purpose: Standard interface and abstraction for the user interface functions, including turning on/off or blinking LEDS and reading switch status. This should be fault tolerant and automatically reset the GPIO Expander on an error.

Rough overview of functions:

* LED On / Off / Blink – Turns a LED on / off or continuously blinks it. Blink rate is adjustable.
* LED Blink Once – Blinks a LED once – pulses it the opposite polarity for a set amount of time. Blink rate is adjustable.
* Read Switch State – Reads the state of a UI switch.
* Update – Update the switch statuses and commits the LEDs to the output where necessary. Note that if the pins are internal to the microcontroller, they should not need updating – instead, they are updated when the functions are called. This function should be called periodically (every 1 – 5 ms) in the main loop to commit changes to the GPIO expander.

**GPIO Expander Hardware**

Purpose: Library and abstraction for the GPIO Expander, allowing read/write status

Rough overview of functions:

* GPIO Expander Write Register – Register-level write operation
* GPIO Expander Read Register – Register-level read operation
* GPIO Expander Write Port Latch – Write the output latch register for the specified port
* GPIO Expander Write Port Direction – Write the direction register for the specified port
* GPIO Expander Read Port Inputs – Read the input register for the specified port.

**CAN Library**

Purpose: Standard interface and abstraction for the CAN functionality, enabling message receive and transmit.

Rough overview of functions – this differs slightly from what is actually implemented since the current implementation is highly coupled to hardware. Future rewrites may fix that. That will likely involve a FIFO stack architecture to enqueue and pop the received frames.

* CAN Transmit – Transmits a CAN frame
* CAN Get Frames Pending – Returns the number of frames pending in the receive buffers
* CAN Get Frame – Returns the CAN frame at the beginning of the stack

**Timing Library**

Purpose: Provides a time-base-independent abstraction for timing functionality.

Architecture overview: The seconds component is stored in memory, and is updated by an interrupt once a second. The milliseconds component is ideally stored in a timer register which fires an interrupt once a second. Some math may be necessary to convert from milliseconds to the hardware time base’s format, but the overhead should be minimal and this allows portability without having to change every time constant written.

Rough overview of functions:

* Elapsed time since start functions:
  + Get Time in Seconds – Returns the number of seconds (rounded down) since the timing library was initialized (essentially when the device started)
  + Get Milliseconds Offset – Returns the milliseconds component of the time since the library started. Accuracy is limited by integer operations and the accuracy of the time base itself.
* Countdown Timer functions:
  + Initialize Countdown Timer – Creates a new countdown timer starting now.
  + Get Timer Expired – Returns whether a countdown timer has expired.
* Count-up Timer functions:
  + Future work

**Code Style Guidelines**

* Microcontroller or hardware dependent code should be decoupled from hardware independent code for better portability
* All functions must have a Doxygen-style comment block stating what the functions does along with description of the arguments taken and values returned where applicable
* Global constant variables and #defines should also be commented
* Code should be written according to team-wide reliability standards, whenever those get released. For now, the guidelines are:
  + Failures should be propagated upwards using return values or some other mechanism. Silent failures should not be allowed.
  + There should be protection against infinite loops caused by external hardware. Therefore, a malfunctioning microcontroller would not be a concern, while a malfunctioning external device should be protected against. Examples: unexpected removal of SD card, I2C clock stretching leading to an infinite loop.
* Use const instead of #define where possible, as this allows some type safety. However, #define may be preferred where it is necessary to generate (or check) numerical constants (such as baud rates) at compile-time.
* Use uint8\_t, etc instead of unsigned char, etc for portability across hardware platforms and compilers.
  + In some cases, you may want to define your own type such as using fs\_length\_t instead of uint32\_t.