A Low-Power Air Quality Data Logger

Yaqub Mahsud and Ethan Greenberg

Introduction

2020 was a particularly bad year for fires across the United States. In addition to destroying millions of acres of land, these wildfires release harmful particles into the atmosphere that can cause dangerous respiratory issues for people, particularly those with underlying respiratory conditions. As such, the team decided to build a device capable of measuring and logging the concentrations of these harmful particles in the air. Referred to by the average particle size in microns, the team will monitor the concentrations of PM1, PM2.5 and PM10 in units of ug/m3. The team desires that the device be powered by a rechargeable battery pack so that it can be moved virtually anywhere. The device will be controlled by an STM32 microcontroller, mounted to a Nucleo F401RE development package. The STM32 will interface with an Amphenol SM-UART-04L air quality sensor to obtain critical measurements of air quality. The STM32 will log these measurements to an SD card so that log files can be opened on a PC and data can be reviewed. Finally, to conserve precious battery power and log as many measurements as possible on a single charge, the device will use a low power mode on the microcontroller to turn off power to as many peripherals as possible during the time between samples. Figure 1 shows a block diagram of the device's functionality.

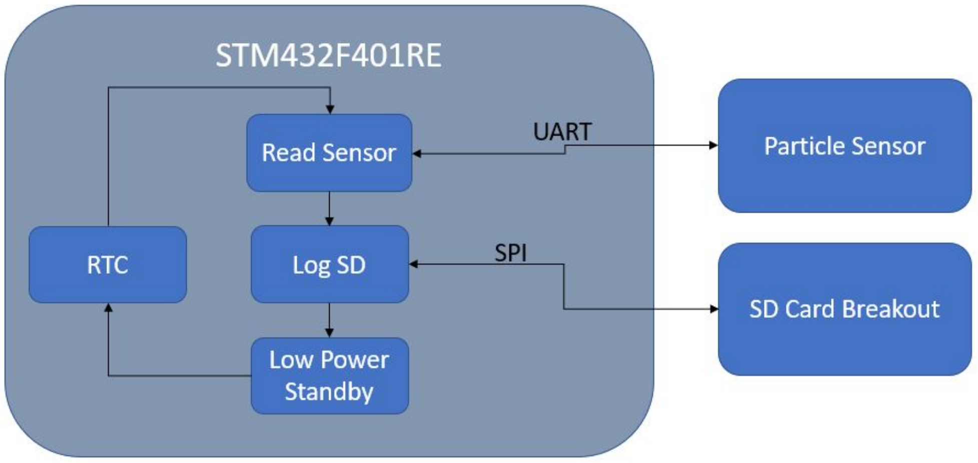


Figure 1: System block diagram

This report will outline the design and implementation of each component of the project, such that individuals may replicate the project in the future.

Hardware

This section will discuss the individual components used in the design.

STM32

The STM32F401RE microcontroller is an ARM Cortex-M4 32-bit processor with a max operating speed of 84MHz. The device has a variety of GPIOs and ADC for interacting with additional components.

Amphenol SM-UART-04L

The Amphenol SM-UART-04L air quality sensor uses a light emitter and detector, positioned specifically within the device, to determine the size and concentration of particles based on the scatter light pattern (**cite datasheet**). The device provides calibrated PM2.5 concentration readings along with estimated PM1 and PM10 measurements. The sensor accepts a 5V supply voltage and typically consumes 60mA during standard operations. The device can also be put in a standby mode by pulling an input pin low. In this standby state, the sensor consumes 5mA. The device communicates measurements and additional information over UART. The UART protocol, as outlined in the datasheet, calls for 9600 bps transfer rate, 8 data bits, no parity and 1 stop bit. The device acts as a slave and waits to accept a command (over UART) from the master before transmitting anything over the line. Valid commands and expected response formats are also outlined in the datasheet, however, the team only uses command “0xE2” to request a measurement update from the device.

The team made a modification to the sensor to make it easier to interface with. The Amphenol sensor has a 10 position, 1.27mm pitch, male connector for interfacing with additional hardware. The team originally ordered a mating female receptacle, but realized that they would not be able to easily interface with the connector without an appropriate PCB or protoboard. Furthermore, the connector is not like standard connectors with the same position configuration and pitch. Considering the hassle and time delay of ordering additional components, the team made the decision to open the sensor and modify the device. The team desoldered the existing male header from the board and instead soldered long wires to the appropriate pins. While this operation was time consuming and stressful, the resulting modifications were successful and eliminated future headaches.

Only a handful of the pins on the device were important for the team. Table 1 shows the pinout for the Amphenol sensor.

Pin #	Function
1	5V power
2	5V power
3	GND
4	GND
5	RESET
7	UART RX
9	UART TX

10	Sleep/Standby Pin

Table 1: Amphenol Sensor Pinout

The team used pins 1, 3, 7 and 9 in their design. Redundant pins were not used. Additionally, the team had no use for the reset or sleep/standby pins or their associated functionality in the design. While the sleep functionality may seem relevant considering the low power objective, the team gains better performance by turning the sensor off entirely. Figure 2 shows the communication connections between the Nucleo and the sensor.

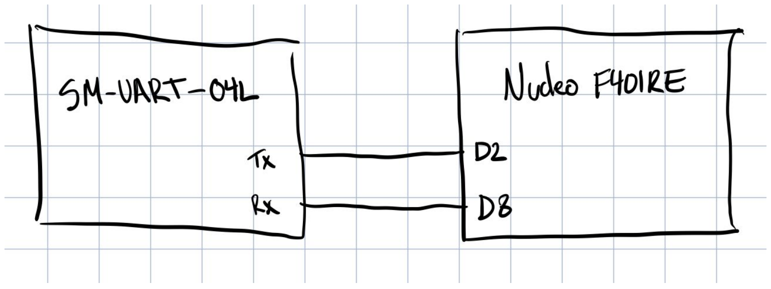


Figure 2: SM-UART-04L and Nucleo connection

SD Card

The team used an Adafruit microSD breakout board configured for SPI functionality. The breakout board also has an on-board level shifter, which shifts 5V logic levels down to 3.3 volts for the card to use. The board has a maximum current draw of 150mA, with a typical draw of 60mA. Figure 3 shows the communication connections between the Nucleo and the SD breakout.

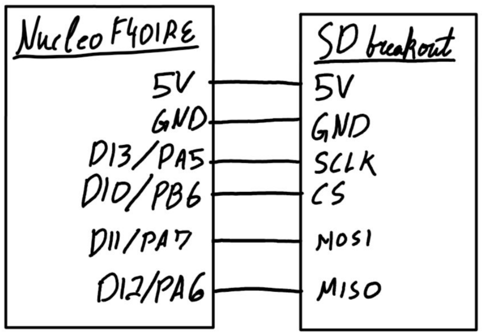


Figure 3: The SD breakout and Nucleo Connection

Battery and Power Components

As previously mentioned, the team is powering the device using an 11.1V, rechargeable, lithium-polymer battery with 2200mAh capacity. The Nucleo board, SD card and Amphenol sensor can be powered with 5V. The SD card peripheral is powered by 3.3V. Conveniently, the Nucleo board has both 5 and 3.3V regulators onboard. The 5V regulator is capable of providing enough current to power the sensor and SD card. To turn off the sensor and SD card power during standby, the team designed 2 low-side NPN transistor switch circuits. Thus, the team will just use two L7805CV linear power regulators in parallel to step the 11.1V battery source down to 5V, which will then power the Nucleo board. The L7805CV regulators take a 7-35V DC input and output 5V, providing up to 1.5A. While the team does not expect the system to consume 1.5A of current, lessening the load on an individual regulator decreases the need for heat dissipation tools. Figure 4 shows how power is supplied to all the individual components.

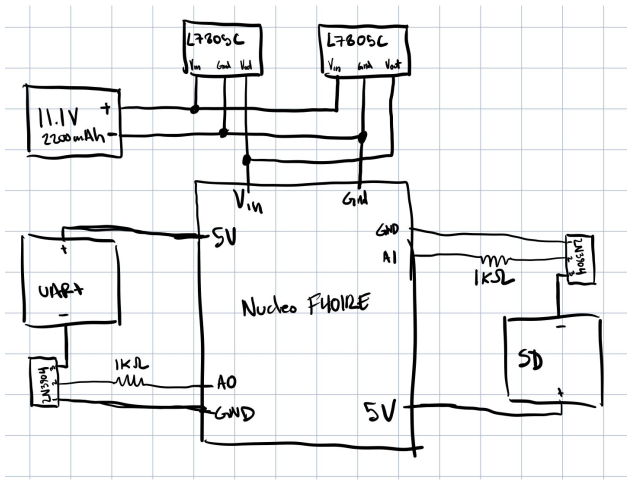


Figure 4: Power connections

To turn off the additional hardware during standby mode, the team used transistor switching circuits. The team used 2, 2N3904 NPN transistors, one to control the SD power and one to control the particle sensor power. These transistors are configured as low-side switches, meaning they are placed after the loads (SD and sensor) but before ground. A GPIO pin connected to each circuit determines whether the "switch" is open or closed. When the GPIO pin goes high, the switch is closed. When the system emerges from low power mode to execute the measurement and logging sequence, the GPIO controlled switches are closed and the sensor and SD card are powered up. Based on the power characteristics of the STM32 and external hardware, the transistor circuits were designed as shown in figure 5.

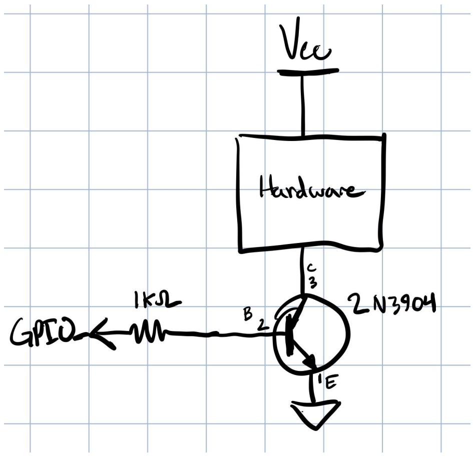


Figure 5: Transistor circuit

The team calculated an estimated system battery life. Table 2 the estimated power consumption of each relevant component of the system.

Component	Worst Case Power (mW)	Best Case Power (mW)
STM32	103.3	0.01518 (standby)
SM-UART-04L	500	0 (off)
SD Card	495	0 (off)
Total	1098.3	0.01518

Table 2: Component power consumptions

As previously mentioned, the team designed the system such that the majority of time is spent in the low power state. The device only consumes full power periodically for short durations to update a measurement and log that measurement to the SD card. The team estimates the awake time to be roughly 30 seconds, which provides enough time for all subsystems to wake up from low power states. The team is considering a variety of sampling periods from 10 minutes to 1 hour. The 11.1V, 2200mAh battery can supply 24.42 watts for one hour. Using this information, the team arrived at the following equation to estimate the battery life of the system on a full charge:

$$battery\ life\ (hours)\ =\ \frac{sample\ period * 24.42W}{1098.3mW * 30s + 0.01518mW * (sample\ period - 30s)}$$

Using the equation, the team estimated roughly 445 hours of life using a 10 minute sample time, and 2660 hours using a 1 hour sample time! It is important to note that the above calculation is somewhat simplified and does not account for all power consumption. First, the rechargeable battery does not output a constant voltage over the duration of a discharge cycle. At some point, the output voltage will drop below 5V and it will no longer power the system. Another important consideration is the power dissipated by the regulators as heat: these devices are not 100% efficient. Performance will also be greatly affected by changes in temperature: lower temperatures will result in a decreased battery output power and life. Regardless, the team is satisfied with the estimated system life, even if it is somewhat overestimated.

Software

The team developed software for the project in C using the Visual Studio Code environment with PlatformIO IDE. PlatformIO is a helpful tool for developing software for a variety of cross-platform embedded targets. It boasts a variety of helpful development tools like an integrated debugger and testing frameworks. The team developed code modularly through the construction of libraries, each containing important functionality.

UART Library

As previously mentioned, the particle sensor uses UART communication to send and receive data. Fortunately, the team was already very familiar with UART communication. Through previous exercises, the team developed a UART library capable of sending and receiving from the STM32. This library consists of two smaller libraries. One library contains functions to initialize one of the STM32's built-in USART peripherals. An initialization function sets up a module with the desired data transfer speed by setting the prescalers to the USART input clock frequency. The initialization function also handles pin mode and alternate function assignments. This library also includes functions to send a byte by writing to the USART data register and functions to receive data. The receive functions work using a ring buffer.

The second small library defines the ring buffer that is used to store received data on the STM32. The ring buffer object includes a buffer (array), head index and tail index. As data is loaded into the buffer, the head is incremented to record the location of the last stored data. Data is loaded into the buffer using hardware interrupts. The hardware is configured such that an interrupt service routine (ISR) function is called when the USART data register receives data. The code inside this ISR takes care of loading the data into the buffer and incrementing the head. The USART ISR located in main simply calls another function defined in the ring buffer library to load newly received data into the buffer. The tail records the index to the last data that was actually read out by the program; each time buffered data is read, the tail is incremented in software. Functions readChar and receiveString, located in the USART library, call functions within the ring buffer library to deal with incrementing the tail appropriately. When the tail and head are equal, then there is no new data to be read. Using a ring buffer and interrupts to receive data is very convenient, especially when one expects to receive tens of bytes in a row. The team did not bother implementing a ring buffer for transmissions because they only need to send a single byte to the sensor.

SD Libraries

Background

Communication with SD cards can be done in one of two modes: SD mode, or SPI mode. With regards to the microcontroller used in this project, SPI mode was selected. This design choice was made because SD mode would have required the development of two of the STM32's peripheral features: DMA and SDIO. With no way to test SDIO on any other devices except for the logic analyzer, and to cut down on development time, the team decided to choose SPI mode.

SPI communication (SPI_SD)

The SPI protocol as defined for SD cards relies on sending commands to the card and receiving responses from it. Valid SD card commands consist of 6 bytes (48 bits) as shown in figure 6, with the most significant bits sent to the card first.

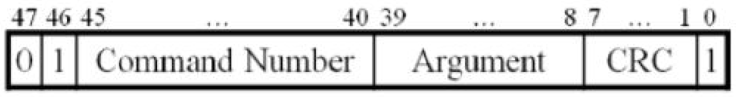


Figure 6: Format of the 48-bit commands [SD association]

The first two bits sent to the SD card are the start bits 0 and 1. These are then followed by a 6-bit command number and a 32-bit argument where additional information (data addresses, configurations, etc.) can be provided. Following the argument is 7 bit containing Cyclic Redundancy Check (CRC) data, and then finally a stop bit. It is worth noting at this point that this is on the MOSI line, and is always kept high.

CRC functionality is only valid in SPI mode when sending the first two commands in the initialization process. These are CMD0 and CMD8. Since SPI mode does not require valid CRC data after these two commands have been sent, the team hardcoded the CRC values when sending these commands. Not using the CRC functionality built into the SPI peripheral on the STM32 was not a design decision made by the team - the CRC functionality in the SPI peripheral is only capable of sending out an entire byte of CRC data, whereas SD mode requires 7 bits followed by a stop bit of one.

SPI communication as implemented by the team was done in accordance with SD specifications [SD association Physical Layer Specification]. All data is transferred in byte oriented serial communication (figure 7). This meant using the 8-bit mode for SPI data transfer on the STM32 microcontroller. Chip select (CS) must be driven low for the entirety of the command-response transaction (including data reads and writes), and since transfers are driven by the master generated clock, the master must continue to send 0xFF until a valid response is received. Clock phase and clock polarity are both 9 when interfacing with SD cards.

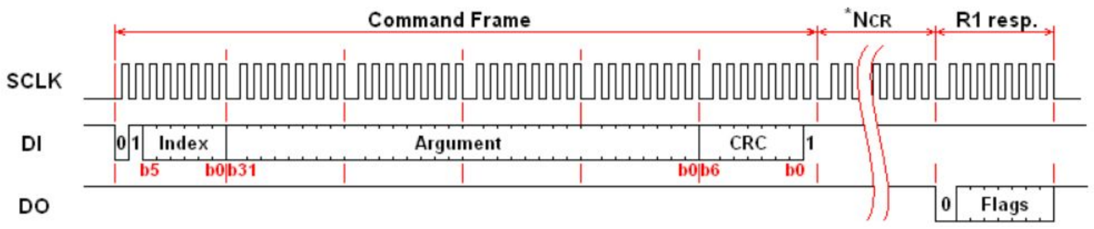


Figure 7: Command and response [Chan SD documentation]

The SPI command set consists of commands with a specific index and argument. Table X in Appendix A lists the commands necessary for generic read/write operations and card initialization. One byte responses of type R1 is the most common response, and for all other types of responses is the first byte returned. Since R1 precedes the other byte(s) in all other responses, every response returns the flags shown in figure 8.

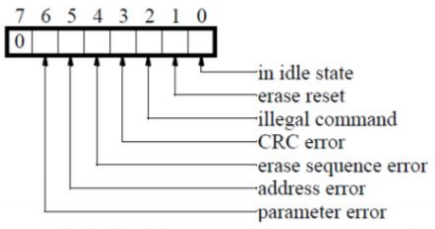


Figure 8: R1 response format

Figure 9 below shows the initialization process for SD cards. Given that the card used by the team was SD Version 2+, the flow followed by the team is as follows:

1. With the card inserted correctly and with CS ,MOSI, and MISO pulled high, send a minimum of 74 clock pulses to the card.
2. Send CMD0 with valid CRC data. The card should return an R1 response of 0x01.
3. Send CMD8 with supply voltage information, check pattern, and valid CRC data.
4. If check patterns are matched on the response from the card, send CMD55, followed by ACMD41.
5. If ACMD41 results in a response of 0x01 (see figure 10), repeat steps 4 and 5 in order.
6. If a response of 0x00 is obtained, send CMD58. The card is now out of idle state and is in data transfer mode.

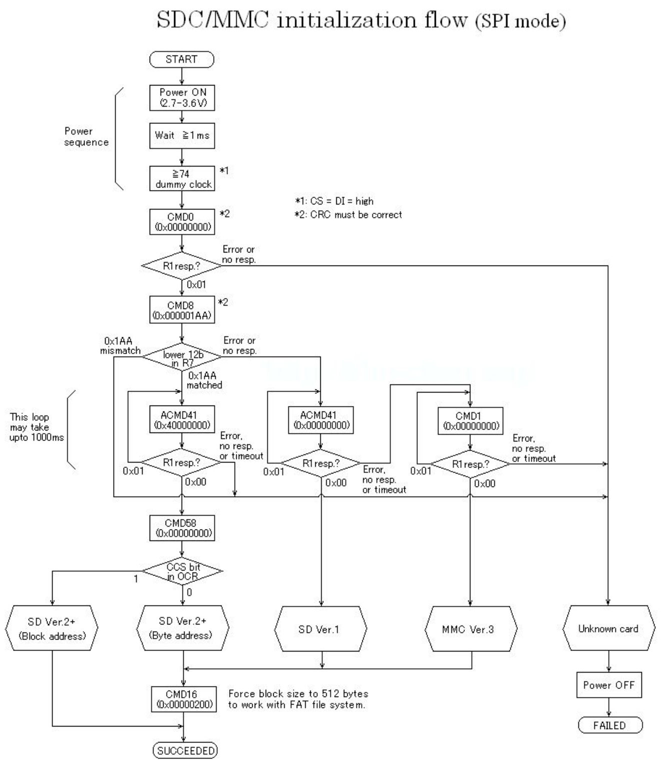


Figure 9: Initialization block diagram [Chan]

Figure 10 shows the first successful initialization of the SD card. The first byte is CMD0, which in hex is 40 00 00 00 00 95 (shown here on the MOSI line). After a delay of one byte, a valid “in idle state” R1 response is received on the MISO line. CMD8 is sent next, corresponding to 48 00 00 01 AA 87. The 5-byte R7 response is received with the correct flags. CMD55 is then sent twice, followed by ACMD41.

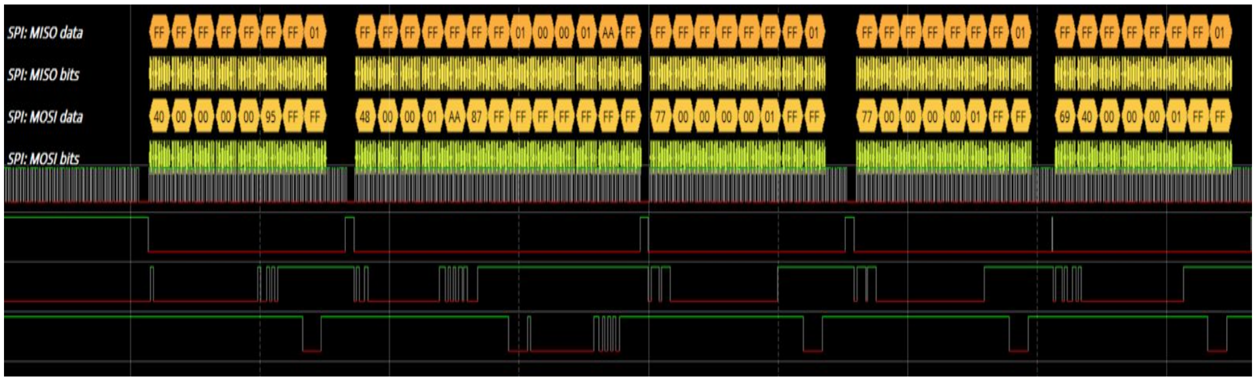


Figure 10: The first successful initialization process (the final ACMD41 with 0x00 response is not shown).

With initialization successful, the team then implemented read and write functionality at the SPI driver level. However, at the application level (main.c), controlling the card and interpreting data so as to remain computer readable and at a higher level of abstraction (e.g. file.open) is the province of the FAT filesystem driver.

The FAT filesystem

All data transfers between modern SD cards are done in units of data blocks, with each data block consisting of 512 bytes. The file system defined for higher capacity SD cards (card having over 4GB of memory) is FAT32. While communicating with an SD card using an SPI driver provides control over raw data.

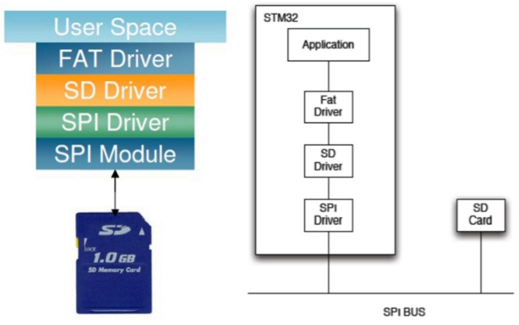


Figure 11: SD card software stack [Chan]

Considering figure 11 above, this project consists of the user space (main.c) where functionality such as opening a file and writing a string to the file is possible. This is built on top of the FAT driver, in this case the open-source FATFs driver (ff.c). This in turn leverages code that needs to be written by users. This is the SD driver, (diskio.c), which converts lower level read, write, and other miscellaneous functions to a form acceptable to the FATFs driver. These lower level functions (SPI_SD.c) are functions such as “sendCommand” or “multiple_block_write” which do exactly what their names imply. These, in turn, are built on the SPI peripheral, initialized in the SPI module (STM32F401RE_SPI.c).

However, while the team initialized the SD card successfully and configured the lower level drivers appropriately, the filesystem functions crashed the microcontroller, leading to no data logging functionality. This is discussed further in the challenges section.

RTC Library

The RTC--or real time clock--is a peripheral on the STM32 that enables a user to keep track of the true time and date as long as the device remains powered. The team was not so much interested in keeping track of the actual time and date, so much as they were interested in using the wakeup event generation feature built into the RTC. The RTC can be configured to generate a variety of time-dependent flags and alarms that can be used to affect the behavior of the STM32. Specifically, the team was interested in using the RTC to generate wakeup events at a determined frequency. These wakeup events can be configured to take the STM32 out of various low power modes, which will be discussed in the following section. Additionally, while the team did not use the real time-keeping features of the RTC, future work on the system could add these features without changing the existing functionality.

The primary components that the team configured on the RTC module were the clock source and prescalers. The team selected the low speed external (LSE) oscillator as the RTC clock input signal. This clock runs at roughly 32kHz and consumes very little power. Setting the prescalers to their max values results in a 1Hz internal clock frequency. In this configuration, wakeup events with a period between 1s and 18hr can be generated, with a 1s resolution, which seemed like an appropriate range for desired sampling rates. The wakeup event frequency is determined by the value in the wakeup timer reload register. Finally, the wakeup timer event generation is enabled. Configuring the RTC can be rather fickle; there are a variety of registers with write protection enabled. To change the values in these registers, keys must be written to other registers to enable write access. This feature prevents buggy user-developed software from accidentally reconfiguring mission-critical modules like the RTC. When a wakeup event is triggered, various flags must be cleared in a specific order to ensure proper functionality. The wakeup events will be discussed more in the following section.

Low Power Mode

As previously mentioned, the team is powering the logging system using a rechargeable, 11.1V, 2200mAh battery pack. This allows the device to be moved virtually anywhere, but makes power usage much more of a concern. Since the team only seeks to log a measurement periodically (anywhere from every 10-60 minutes), the device only needs to function at normal capacity for a fraction of its battery life. The STM32 has a variety of low power modes that can be configured to conserve significant power. As already discussed, the team is using the RTC to generate wakeup events that will take the STM32 out of low power mode. The team decided to use the lowest power option possible: standby mode. In standby mode, the STM32 is practically powered down; only a select few registers retain power and thus maintain their configurations on wakeup. Many of the RTC registers are included in this group, hence why the device is still able to generate wakeup events. Other power saving measures include shutting off nearly all of the voltage regulators on the Nucleo board and thus most of the clocks, too. One exception here is the LSE oscillator, which is very power efficient and can be configured to retain power. With these many power saving considerations, standby mode only consumes roughly 4.6uA, compared to a normal operating power consumption of roughly 31.1mA.

Low power mode is configured through a variety of registers in a few different blocks. The power controller module is, unsurprisingly, the primary source for relevant control registers, but bits must also be set in the system control registers, too. This configuration determines the type of low power behavior exhibited by the system. The system can be put into low power mode by two main commands: wait for interrupt (WFI) or wait for event (WFE). As the team is using the RTC to generate wakeup events, they used WFE to put the system into low power mode. The team also configured the EXTI control registers per the reference manual such that the wakeup event would bring the device out of standby mode.

Main

The developed libraries are brought together and utilized in the main source files. Within these files, a main function is called on startup. Code in the main code is responsible for setting up and initializing all of the relevant hardware using the functions from the developed libraries. Most of this setup happens unconditionally with the exception of the RTC and low power setup. As mentioned in the previous section, the team is using the standby low power mode. When the STM32 wakes from standby mode it largely acts as though it has experienced a power cycle: returning to the start of main. The only notable peripherals that maintain their original state are the RTC and power controller. Upon waking from standby, a flag in the power controller, CWUF, is set to differentiate wakeup from reset after a power cycle. Initialization of the RTC and power controller is safeguarded by a conditional statement and a check to this flag; if the device has returned from standby, these peripherals are not edited. The RTC is left unchanged to maintain the real time held by the clock. This makes it easier for individuals to add functionality that initializes the RTC with the real time.

After initialization is complete, central functionality of the device is executed through a series of function calls. A short delay is called to ensure the particle sensor has adequate time to wake and warm up. Then, a function, “getAmphenol’ is called. This function uses the UART libraries to send the get measurement command to the particle sensor and stores the response. The response, which is initially put into a ring buffer, is then copied into a character array global to the main. As noted in a previous section, the response from the sensor is of a known format. Each particle concentration measurement is stored in two bytes: high_data and low_data. The concentration value is obtained by: high_data*256 + low_data. The three calculated particle concentrations are then stored in global variables.

After updating the current particle sensor measurements, the logging operation begins. This consists of initializing the SD card, file system and file objects, and then opening and writing to a file. After these operations are complete the device re-enters standby mode.

Challenges

The team encountered a variety of challenges throughout the project. First, configuring the RTC wakeup events to bring the device out of low power mode was non-trivial. Unsurprisingly, the debugger is not very effective when the device has turned off nearly all peripherals. As such, the team needed to use visual hardware indicators (like LEDs) or the oscilloscope to indicate the power state. The oscilloscope was difficult to use because it required the team to monitor the scope over long periods of time. Initially, the team tried blinking an LED each time the device exited low power mode. The team saw the LED blink at irregular frequencies and not at the set wakeup event frequency. Ultimately, the team discovered that the delay function was somehow interfering with the RTC.

Debugging the SD functionality was even more challenging. Initially, initializing the SD card failed due to a 2-bit incrementing offset in responses to SD commands. This led to all commands after CMD0 being rejected as illegal commands as the card only saw offset (and thus invalid) commands. This error was due to some clock cycles passing between subroutine calls (e.g send_command and get_response_byte were two separate functions). The solution to this was to condense SPI code into large blocks which ensured that no delays would be possible in between commands, responses, reads, and writes.

The filesystem layer proved to be the proverbial nail for data logging functionality. Despite successfully initializing the SD card, as soon as any file system functionality was called upon, the STM32's SPI functionality ceased to function at all, *even when power cycled and uploaded without file system functions.* This meant that the team could not view any error flags returned by the card on the logic analyzer, as all commands AND responses on the logic analyzer were not actually reflective of any implementation in main.

Works Cited

"SM-UART-04L PM2.5+PM10 Particulate Dust Sensor", Telaire Amphenol Advanced sensors. Available at: https://www.amphenol-sensors.com/en/component/edocman/514-telaire-sm-uart-04l-laser-dust-sensor-application-note/download?Itemid=8488%20%27

"How to use MMC/SDC", Elm Chan, Available at: http://elm-chan.org/docs/mmc/mmc_e.html, last updated December 26, 2019

"FatFS - Generic FAT Filesystem Module", Elm Chan, Available at: http://elm-chan.org/fsw/ff/00index_e.html
"SD Association Physical Layer Simplified Specification", Available at: https://www.sdcard.org/downloads/pls/

"MicroSD card breakout board+", Available at: https://www.adafruit.com/product/254

Appendix A: Main

```
/**
    Main Header: Contains general defines and selected portions of CMSIS
files
    @file main.h
    @author Yaqub Mahsud and Ethan Greenberg
    adapted from Joshua Brake
    @version 2.0 12/1/2020
*/

#ifndef MAIN_H
#define MAIN_H

#include "STM32F401RE.h"

///////////////////////////////////////////////////////////////////////////
/////
// Custom defines
///////////////////////////////////////////////////////////////////////////
/////

// Wifi Network Info (make sure to keep surrounding double quotes)
#define SSID "magoonda"
#define PASSWORD "6654ff6654"

#define NVIC_ISER0 ((uint32_t *) 0xE000E100UL)
#define NVIC_ISER1 ((uint32_t *) 0xE000E104UL)
#define SYSCFG_EXTICR4 ((uint32_t *) (0x40013800UL + 0x14UL))
#define PWR_BASE ((uint32_t *) 0x40007000UL)
#define SCB_SCR ((uint32_t *) 0xE000ED10UL)

typedef struct {
  volatile uint32_t LPDS  :1; //low power deep sleep
  volatile uint32_t PDDS  :1; //power down deepsleep
  volatile uint32_t CWUF  :1;
  volatile uint32_t CSBF  :1;
  volatile uint32_t       :4;
  volatile uint32_t DBP   :1;
  volatile uint32_t       :23;
} PWR_bits;

typedef struct {
  volatile uint32_t WUF   :1; //wakeup flag
  volatile uint32_t SBF   :1; //standby flag: 1 if the device has been in
standby
  volatile uint32_t       :30;
} CSR_bits;

typedef struct {
  volatile PWR_bits PWRR;
  volatile CSR_bits CSR;
} PWR_typedef;

typedef struct {
    volatile uint32_t IMR;
    volatile uint32_t EMR;
    volatile uint32_t RTSR;
    volatile uint32_t FTSR;
    volatile uint32_t SWIER;
    volatile uint32_t PR;
}EXTI_TypeDef;

#define EXTI ((EXTI_TypeDef *) 0x40013C00UL)
#define PWR ((PWR_typedef *) PWR_BASE)

// Request defines
#define REQ_UNKNOWN 0
#define REQ_LED_ON 1
#define REQ_LED_OFF 2

// LED pin
#define LED_PIN 5
// Switch Pin
#define SWITCH 4
//sensor sleep pin
#define AMPHENOL_SLEEP_PIN 8
```

```
//sensor power pin
#define AMPHENOL_POWER 0
//sd power pin
#define SD_POWER 1

#define AMPHENOL_UART_ID USART1_ID
#define TERM_USART_ID USART2_ID
#define DELAY_TIM TIM2
#define CMD_DELAY_MS 30
#define BUFFER_SIZE 2048

///////////////////////////////////////////////////////////////////////////
/////
// IRQn_Type and __NVIC_PRIO_BITS from stm32f401xe.h
///////////////////////////////////////////////////////////////////////////
/////

/**
 * @brief STM32F4XX Interrupt Number Definition, according to the selected
device
 *        in @ref Library_configuration_section
 */
typedef enum
{
/******  Cortex-M4 Processor Exceptions Numbers
****************************************************************/
  NonMaskableInt_IRQn         = -14,    /*!< 2 Non Maskable Interrupt
*/
  MemoryManagement_IRQn       = -12,    /*!< 4 Cortex-M4 Memory Management
Interrupt                            */
  BusFault_IRQn               = -11,    /*!< 5 Cortex-M4 Bus Fault
Interrupt                                    */
  UsageFault_IRQn             = -10,    /*!< 6 Cortex-M4 Usage Fault
Interrupt                                  */
  SVCall_IRQn                 = -5,     /*!< 11 Cortex-M4 SV Call
Interrupt                                     */
  DebugMonitor_IRQn           = -4,     /*!< 12 Cortex-M4 Debug Monitor
Interrupt                               */
  PendSV_IRQn                 = -2,     /*!< 14 Cortex-M4 Pend SV
Interrupt                                     */
```

```
  SysTick_IRQn                = -1,     /*!< 15 Cortex-M4 System Tick
Interrupt                                */
/******  STM32 specific Interrupt Numbers
**********************************************************************/
  WWDG_IRQn                   = 0,      /*!< Window WatchDog Interrupt
*/
  PVD_IRQn                    = 1,      /*!< PVD through EXTI Line
detection Interrupt                         */
  TAMP_STAMP_IRQn             = 2,      /*!< Tamper and TimeStamp
interrupts through the EXTI line             */
  RTC_WKUP_IRQn               = 3,      /*!< RTC Wakeup interrupt through
the EXTI line                         */
  FLASH_IRQn                  = 4,      /*!< FLASH global Interrupt
*/
  RCC_IRQn                    = 5,      /*!< RCC global Interrupt
*/
  EXTI0_IRQn                  = 6,      /*!< EXTI Line0 Interrupt
*/
  EXTI1_IRQn                  = 7,      /*!< EXTI Line1 Interrupt
*/
  EXTI2_IRQn                  = 8,      /*!< EXTI Line2 Interrupt
*/
  EXTI3_IRQn                  = 9,      /*!< EXTI Line3 Interrupt
*/
  EXTI4_IRQn                  = 10,     /*!< EXTI Line4 Interrupt
*/
  DMA1_Stream0_IRQn           = 11,     /*!< DMA1 Stream 0 global
Interrupt                                    */
  DMA1_Stream1_IRQn           = 12,     /*!< DMA1 Stream 1 global
Interrupt                                    */
  DMA1_Stream2_IRQn           = 13,     /*!< DMA1 Stream 2 global
Interrupt                                    */
  DMA1_Stream3_IRQn           = 14,     /*!< DMA1 Stream 3 global
Interrupt                                    */
  DMA1_Stream4_IRQn           = 15,     /*!< DMA1 Stream 4 global
Interrupt                                    */
  DMA1_Stream5_IRQn           = 16,     /*!< DMA1 Stream 5 global
Interrupt                                    */
  DMA1_Stream6_IRQn           = 17,     /*!< DMA1 Stream 6 global
Interrupt                                    */
```

```
  ADC_IRQn                    = 18,     /*!< ADC1, ADC2 and ADC3 global
Interrupts                             */
  EXTI9_5_IRQn                = 23,     /*!< External Line[9:5] Interrupts
*/
  TIM1_BRK_TIM9_IRQn          = 24,     /*!< TIM1 Break interrupt and TIM9
global interrupt                    */
  TIM1_UP_TIM10_IRQn          = 25,     /*!< TIM1 Update Interrupt and
TIM10 global interrupt                  */
  TIM1_TRG_COM_TIM11_IRQn     = 26,     /*!< TIM1 Trigger and Commutation
Interrupt and TIM11 global interrupt */
  TIM1_CC_IRQn                = 27,     /*!< TIM1 Capture Compare
Interrupt                                    */
  TIM2_IRQn                   = 28,     /*!< TIM2 global Interrupt
*/
  TIM3_IRQn                   = 29,     /*!< TIM3 global Interrupt
*/
  TIM4_IRQn                   = 30,     /*!< TIM4 global Interrupt
*/
  I2C1_EV_IRQn                = 31,     /*!< I2C1 Event Interrupt
*/
  I2C1_ER_IRQn                = 32,     /*!< I2C1 Error Interrupt
*/
  I2C2_EV_IRQn                = 33,     /*!< I2C2 Event Interrupt
*/
  I2C2_ER_IRQn                = 34,     /*!< I2C2 Error Interrupt
*/
  SPI1_IRQn                   = 35,     /*!< SPI1 global Interrupt
*/
  SPI2_IRQn                   = 36,     /*!< SPI2 global Interrupt
*/
  USART1_IRQn                 = 37,     /*!< USART1 global Interrupt
*/
  USART2_IRQn                 = 38,     /*!< USART2 global Interrupt
*/
  EXTI15_10_IRQn              = 40,     /*!< External Line[15:10]
Interrupts                                   */
  RTC_Alarm_IRQn              = 41,     /*!< RTC Alarm (A and B) through
EXTI Line Interrupt                   */
  OTG_FS_WKUP_IRQn            = 42,     /*!< USB OTG FS Wakeup through
EXTI line interrupt                     */
```

```
  DMA1_Stream7_IRQn             = 47,     /*!< DMA1 Stream7 Interrupt
*/
  SDIO_IRQn                     = 49,     /*!< SDIO global Interrupt
*/
  TIM5_IRQn                     = 50,     /*!< TIM5 global Interrupt
*/
  SPI3_IRQn                     = 51,     /*!< SPI3 global Interrupt
*/
  DMA2_Stream0_IRQn             = 56,     /*!< DMA2 Stream 0 global
Interrupt                                      */
  DMA2_Stream1_IRQn             = 57,     /*!< DMA2 Stream 1 global
Interrupt                                      */
  DMA2_Stream2_IRQn             = 58,     /*!< DMA2 Stream 2 global
Interrupt                                      */
  DMA2_Stream3_IRQn             = 59,     /*!< DMA2 Stream 3 global
Interrupt                                      */
  DMA2_Stream4_IRQn             = 60,     /*!< DMA2 Stream 4 global
Interrupt                                      */
  OTG_FS_IRQn                   = 67,     /*!< USB OTG FS global Interrupt
*/
  DMA2_Stream5_IRQn             = 68,     /*!< DMA2 Stream 5 global
interrupt                                      */
  DMA2_Stream6_IRQn             = 69,     /*!< DMA2 Stream 6 global
interrupt                                      */
  DMA2_Stream7_IRQn             = 70,     /*!< DMA2 Stream 7 global
interrupt                                      */
  USART6_IRQn                   = 71,     /*!< USART6 global interrupt
*/
  I2C3_EV_IRQn                  = 72,     /*!< I2C3 event interrupt
*/
  I2C3_ER_IRQn                  = 73,     /*!< I2C3 error interrupt
*/
  FPU_IRQn                      = 81,     /*!< FPU global interrupt
*/
  SPI4_IRQn                     = 84      /*!< SPI4 global Interrupt
*/
} IRQn_Type;

#define __NVIC_PRIO_BITS          4U       /*!< STM32F4XX uses 4 Bits for
the Priority Levels */
```

```
#include "cmsis_gcc.h"
#include "core_cm4.h"


#endif // MAIN_H
```

```
/*
Main file for air quality logger project
Yaqub Mahsud and Ethan Greenberg
12/1/20

*/

#include "STM32F401RE.h"
#include "main.h"
#include "UARTRingBuffer.h"
#include <string.h>
#include <stdio.h>
#include "SPI_SD.h"
#include "STM32F401RE_SPI.h"
#include "diskio.h"
#include "ff.h"
#include "ffconf.h"

uint8_t AMPHENOL_RESPONSE[60]; //char array for storing response from
particle sensor
uint8_t *arrayPTR = &AMPHENOL_RESPONSE;
double PM2_5; //reading for pm2.5
double PM10; //reading for pm10
double PM1;

//function to get a measurement from the Amphenol sensor
//updates global variables with the values
void getAmphenol(){
    USART_TypeDef * AMPHENOL_UART = initUSART(AMPHENOL_UART_ID, 9600);
//initialize uart for amphenol on usart1
    memset(AMPHENOL_RESPONSE, 0, 60); //clear the receive array
    sendChar(AMPHENOL_UART, 0xE2); //send the get measurement command
    receiveString(AMPHENOL_UART, &AMPHENOL_RESPONSE); //read the char
array response from the amphenol,
    //now convert/parse response into variable updates
    PM1 = AMPHENOL_RESPONSE[5]*256 + AMPHENOL_RESPONSE[6];
    PM2_5 = AMPHENOL_RESPONSE[7]*256 + AMPHENOL_RESPONSE[8];
    PM10 = AMPHENOL_RESPONSE[9]*256 + AMPHENOL_RESPONSE[10];
}

//calling this function logs the global measurement values to the SD card
void logSD(){
    uint8_t *SD_message[64] = {0};
    sprintf(SD_message, "%d\t%d\t%d\n", PM1, PM2_5, PM10);
    //TODO: SD stuff
}

/** Map USART1 IRQ handler to our custom ISR
 */
void USART1_IRQHandler(){
    USART_TypeDef * AMPHENOL_UART = id2Port(AMPHENOL_UART_ID);
    usart_ISR(AMPHENOL_UART);
}



//page 86 in ref manual, wakeup clear routine
void safeWakup(){
    PWR->PWRR.DBP = 1;
    RTC->WPR.KEY = 0xCA;
    RTC->WPR.KEY = 0x53;
    RTC->CR.WUTIE = 0; //disable wakeup interrupt
    RTC->ISR.WUTF = 0;     //clear rtc wakeup flag
    PWR->PWRR.CWUF = 1;    //clear pwr wakeup flag
    RTC->CR.WUTIE = 1;     //enable wakeup interrupt
    RTC->WPR.KEY = 0xFF;
    PWR->PWRR.DBP = 0;
}

int main(void) {
    configureFlash(); //configure flash controller
    configureClock(); //configure clock 38.4MHz
    // Enable GPIOA clock
    RCC->AHB1ENR.GPIOAEN = 1;
    RCC->AHB1ENR.GPIOBEN = 1;
    RCC->APB1ENR |= (1 << 28); //enable clock to pwr controller
    // Initialize timer
    RCC->APB1ENR |= (1 << 0); // TIM2_EN
    initTIM(DELAY_TIM); //initialize timer

    //set pin modes
    //pinMode(GPIOA, LED_PIN, 1);
    pinMode(GPIOA, SWITCH, GPIO_INPUT);
    pinMode(GPIOA, AMPHENOL_POWER, GPIO_OUTPUT);
    pinMode(GPIOA, SD_POWER, GPIO_OUTPUT);
    //supply power to SD card and sensor
    digitalWrite(GPIOA, AMPHENOL_POWER, 1);
    digitalWrite(GPIOA, SD_POWER, 1);


    USART_TypeDef *AMPHENOL_UART = initUSART(AMPHENOL_UART_ID, 9600);
//initialize uart for amphenol on usart1

    // Configure USART1 interrupt
    // Enable interrupts globally
    __enable_irq();

    // Configure interrupt for USART1
    *NVIC_ISER1 |= (1 << 5);
    AMPHENOL_UART->CR1.RXNEIE = 1;




    // Initialize ring buffer
    init_ring_buffer();
    flush_buffer();
    //--------
    //initialize and set RTC IF we have not been in standby before
    //--------
    if(PWR->PWRR.CWUF == 0){
        PWR->PWRR.DBP = 1; //diable write protection on BDCR
        // initRTC();
        PWR->PWRR.DBP = 0; //enable write protection on BDCR
    }

    // configure EXTI controller for wakeup event
    EXTI->EMR |= (1 << 22); //configure the event mask bit
    EXTI->IMR &= ~(1 << 22); //configure the interrupt mask bit
    EXTI->RTSR |= (1 << 22); //set bit on line 22 rising edge
    EXTI->FTSR &= ~(1 << 22); //clear the falling edge trigger
    __NVIC_EnableIRQ(RTC_WKUP_IRQn); //map the event to the NVIC



    //-------
    //Configure SLEEP
    //--------
    //mode configuration: when the following three are zero, sleep mode
    //when SLEEPDEEP and PDDS, standby mode
    //when SLEEPDEEP, PDDS and LPDS, stop mode
    SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; //this is setting SLEEPDEEP to 1
    PWR->PWRR.PDDS = 1; //setting PDDS to 1
    PWR->PWRR.LPDS = 0;
    delay_millis(TIM2, 1);
    safeWakup();

    initSD_SPI(); //initialize SD card for writing
    getAmphenol(); //update particle sensor measurement
    logSD(); //log to SD card
    while(digitalRead(GPIOA, SWITCH) == 1); //do not go to sleep if the
switch is on
    safeWakup();
    __WFE(); //bedtime


    return 1;
}
```

Appendix B: Lower Level SD Libraries

Appendix C: RTC

```
//STM32 RTC Library
// Yaqub Mahsud and Ethan Greenberg
// 12/1/20

#ifndef STM32F401RE_RTC_H
#define STM32F401RE_RTC_H

#include <stdint.h>

#define __IO volatile

// Base addresses
#define RTC_BASE (0x40002800) // base address of RTC

//time register
typedef struct {
    __IO uint32_t SU        :4; //secs units
    __IO uint32_t ST        :3; //sec tens
    __IO uint32_t           :1;
    __IO uint32_t MNU       :4; //min units
    __IO uint32_t MNT       :3; //min tens
    __IO uint32_t           :1;
    __IO uint32_t HU        :4; //hr units
    __IO uint32_t HT        :2; //hr tens
    __IO uint32_t PM        :1; //AM/PM notation, 0: AM or 24; 1: PM
    __IO uint32_t           :9;

} TR_bits;

//controller register bits
typedef struct {
    __IO uint32_t WUCKSEL        :3; //input selection
    __IO uint32_t TSEDGE        :1;
    __IO uint32_t REFCKON          :1;
    __IO uint32_t BYPSHAD       :1;
    __IO uint32_t FMT       :1;
    __IO uint32_t DCE          :1;
    __IO uint32_t ALRAE          :1;
    __IO uint32_t ALRBE          :1;
    __IO uint32_t WUTE          :1; //wakeup timer enable
    __IO uint32_t TSE          :1;
    __IO uint32_t ALRAIE          :1;
    __IO uint32_t ALRBIE          :1;
    __IO uint32_t WUTIE          :1; //wakeup timer interrupt enable
    __IO uint32_t ADD1H          :1;
    __IO uint32_t SUB1H          :1;
    __IO uint32_t BKP          :1;
    __IO uint32_t COSEL          :1;
    __IO uint32_t POL          :1;
    __IO uint32_t OSEL          :2;
    __IO uint32_t COE          :1;
    __IO uint32_t           :8;
} CRR_bits;

//rtc interrupt register
typedef struct {
    __IO uint32_t ALRAWF        :1;
    __IO uint32_t ALRBWF        :1;
    __IO uint32_t WUTWF          :1; //wakeup timer waiting flag
    __IO uint32_t SHPF       :1;
    __IO uint32_t INITS       :1;
    __IO uint32_t RSF          :1;
    __IO uint32_t INITF          :1;
    __IO uint32_t INIT          :1;
    __IO uint32_t ALRAF          :1;
    __IO uint32_t ALRBF          :1;
    __IO uint32_t WUTF          :1; //wakeup timer flag
    __IO uint32_t TSF          :1;
    __IO uint32_t TSOVF          :1;
    __IO uint32_t TAMP1F          :1;
    __IO uint32_t           :2;
    __IO uint32_t RECALPF          :1;
    __IO uint32_t           :15;
```

```c
} ISR_bits;

//clock predivisor bits
typedef struct {
    __IO uint32_t PREDIV_S      :15;
    __IO uint32_t               :1;
    __IO uint32_t PREDIV_A      :7;
    __IO uint32_t               :9;
} PRER_bits;

//write protection key bits
typedef struct {
    __IO uint32_t KEY       :8;
    __IO uint32_t           :24;
} WPR_bits;

//rtc registers structure
typedef struct {
    __IO TR_bits    TR;
    __IO uint32_t   DR;
    __IO CRR_bits    CR;
    __IO ISR_bits   ISR;
    __IO PRER_bits  PRER;
    __IO uint32_t   WUTR;
    __IO uint32_t   CALIBR;
    __IO uint32_t   ALRMAR;
    __IO uint32_t   ALRMBR;
    __IO WPR_bits   WPR;
} RTC_TYPEDEF;

//function prototypes

//function to initialize the rtc and wakeup event flags
void initRTC();

#define RTC ((RTC_TYPEDEF *) RTC_BASE)

#endif
/*
    Source code for RTC configuration
    Yaqub Mahsud and Ethan Greenberg
    12/1/2020
*/

#include "STM32F401RE_RTC.h"
#include "STM32F401RE_RCC.h"

//function to initialize RTC with wakeup event generation
//initializes wakeup event generation at 10 minute period
void initRTC(){
    //configure clocks
    //LESON

    RCC->BDCR.LSEON = 1; //turn on LSE
    //check if LSERDY
    while(RCC->BDCR.LSERDY == 0);
    RCC->BDCR.RTCSEL = 1; //LSE as rtc source
    RCC->BDCR.RTCEN = 1; //enable rtc

    //disable write protection on RTC
    RTC->WPR.KEY = 0xCA;
    RTC->WPR.KEY = 0x53;
    //clear wakeup timer
    RTC->CR.WUTE = 0;
    //poll WUTWF until it is set
    while(0 == RTC->ISR.WUTWF);
    //set RTC prescalers for sample frequency
    RTC->PRER.PREDIV_S = 256;
    RTC->PRER.PREDIV_A = 128;
    //set the timer count register for wakeup period
    RTC->WUTR = 10*60; //this is 10 mins 60*10
    //set desired clock source
    RTC->CR.WUCKSEL = 0b100;
    //enable wakeup timer
    RTC->CR.WUTE = 1;
    //enable write protection on RTC
    RTC->WPR.KEY = 0xFF;
}
```

Appendix D: FAT File System Libraries

```
/*-----------------------------------------------------------------------/
/  Low level disk interface module include file   (C)ChaN, 2019          /
/-----------------------------------------------------------------------*/

// Starter code written by Elm Chan. Adapted by Ethan Greenberg and Yaqub
Mahsud

#ifndef _DISKIO_DEFINED
#define _DISKIO_DEFINED

#ifdef __cplusplus
extern "C" {
#endif


// includes

#include "integer.h"
#include "ff.h"
#include "SPI_SD.h"

/* Status of Disk Functions */
typedef BYTE    DSTATUS;

/* Results of Disk Functions */
typedef enum {
    RES_OK = 0,      /* 0: Successful */
    RES_ERROR,       /* 1: R/W Error */
    RES_WRPRT,       /* 2: Write Protected */
    RES_NOTRDY,      /* 3: Not Ready */
    RES_PARERR       /* 4: Invalid Parameter */
} DRESULT;


/*---------------------------------------*/
/* Prototypes for disk control functions */
```

```
DSTATUS disk_initialize (BYTE pdrv);
DSTATUS disk_status (BYTE pdrv);
DRESULT disk_read (BYTE pdrv, BYTE* buff, LBA_t sector, UINT count);
DRESULT disk_write (BYTE pdrv, const BYTE* buff, LBA_t sector, UINT
count);
DRESULT disk_ioctl (BYTE pdrv, BYTE cmd, void* buff);

DWORD get_fattime (void);


/* Disk Status Bits (DSTATUS) */

#define STA_NOINIT      0x01    /* Drive not initialized */
#define STA_NODISK      0x02    /* No medium in the drive */
#define STA_PROTECT     0x04    /* Write protected */


/* Command code for disk_ioctrl function */

/* Generic command (Used by FatFs) */
#define CTRL_SYNC           0   /* Complete pending write process (needed
at FF_FS_READONLY == 0) */
#define GET_SECTOR_COUNT    1   /* Get media size (needed at FF_USE_MKFS
== 1) */
#define GET_SECTOR_SIZE     2   /* Get sector size (needed at FF_MAX_SS !=
FF_MIN_SS) */
#define GET_BLOCK_SIZE      3   /* Get erase block size (needed at
FF_USE_MKFS == 1) */
#define CTRL_TRIM           4   /* Inform device that the data on the
block of sectors is no longer used (needed at FF_USE_TRIM == 1) */

/* Generic command (Not used by FatFs) */
#define CTRL_POWER          5   /* Get/Set power status */
#define CTRL_LOCK           6   /* Lock/Unlock media removal */
#define CTRL_EJECT          7   /* Eject media */
#define CTRL_FORMAT         8   /* Create physical format on the media */

/* MMC/SDC specific ioctl command */
#define MMC_GET_TYPE        10  /* Get card type */
```

```
#define MMC_GET_CSD          11  /* Get CSD */
#define MMC_GET_CID          12  /* Get CID */
#define MMC_GET_OCR          13  /* Get OCR */
#define MMC_GET_SDSTAT       14  /* Get SD status */
#define ISDIO_READ           55  /* Read data form SD iSDIO register */
#define ISDIO_WRITE          56  /* Write data to SD iSDIO register */
#define ISDIO_MRITE          57  /* Masked write data to SD iSDIO register
*/

/* ATA/CF specific ioctl command */
#define ATA_GET_REV          20  /* Get F/W revision */
#define ATA_GET_MODEL        21  /* Get model name */
#define ATA_GET_SN           22  /* Get serial number */

#ifdef __cplusplus
}
#endif

#endif
```

```
/*-----------------------------------------------------------------------*
/
/* Low level disk I/O source code for FatFs     (C)ChaN, 2013        */
/*-----------------------------------------------------------------------*
/

// Starter code written by Elm Chan. Adapted by Ethan Greenberg and Yaqub
Mahsud

/* This is a glue function to attach various existing                  */
/* storage control module to the FatFs module with a defined API.
*/
/*-----------------------------------------------------------------------*
/

#include "ff.h"         /* Obtains integer types */
#include "diskio.h"     /* Declarations of disk functions */
```

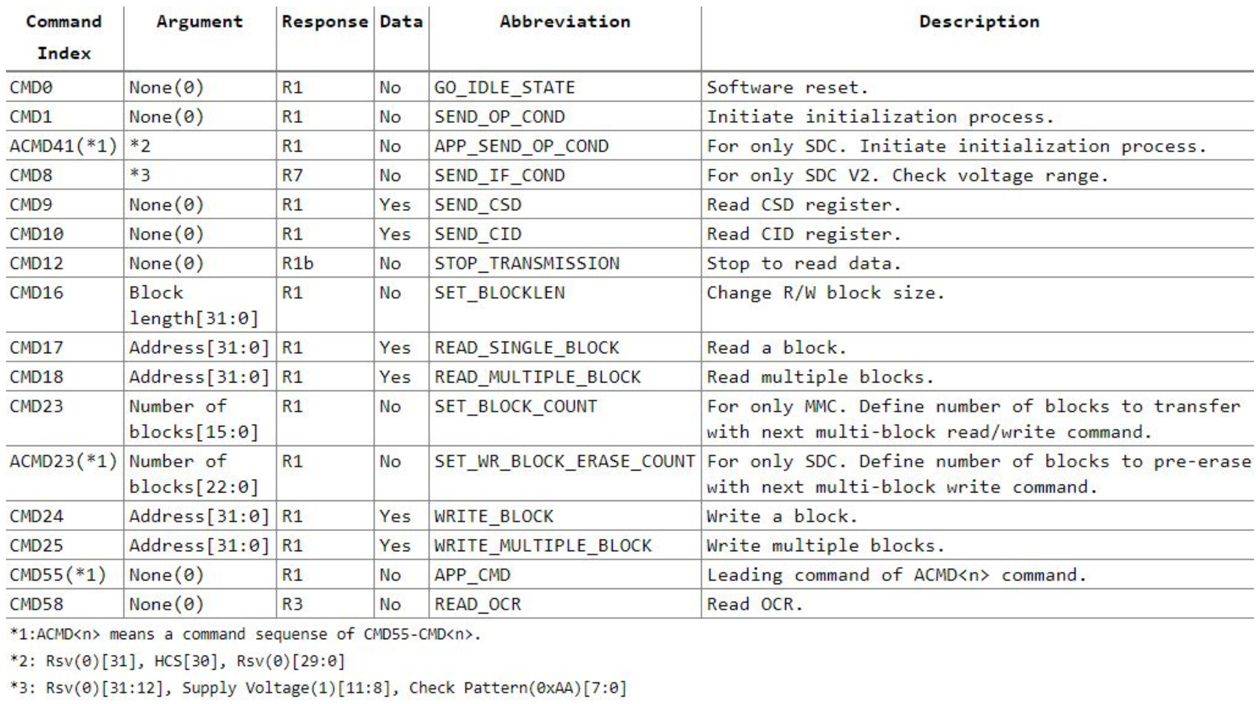
```
// added includes

#include "SPI_SD.h"
#include "STM32F401RE_SPI.h"



/*-----------------------------------------------------------------------*
/
/* Get Drive Status
*/
/*-----------------------------------------------------------------------*
/

DSTATUS disk_status (
    BYTE pdrv       /* Physical drive number to identify the drive */
)
{
    DSTATUS stat;

    uint16_t statusbyte = getStatus();

    if(statusbyte==0){
        stat &= ~STA_NOINIT;
        return stat;
    }
    else{
        return RES_PARERR;
    }

}



/*-----------------------------------------------------------------------*
/
/* Initialize a Drive
*/
/*-----------------------------------------------------------------------*
/
```

```
DSTATUS disk_initialize (
    BYTE pdrv               /* Physical drive number to identify the drive
*/
)
{
    DSTATUS stat;           // this needs to be declared elsewhere
    if (1)  {
        stat  &= ~STA_NOINIT;
        return stat;
    }

    return stat;

}

/*-----------------------------------------------------------------------*
/
/* Read Sector(s)
*/
/*-----------------------------------------------------------------------*
/

DRESULT disk_read (
    BYTE pdrv,      /* Physical drive number to identify the drive */
    BYTE *buff,     /* Data buffer to store read data */
    LBA_t sector,   /* Start sector in LBA */
    UINT count      /* Number of sectors to read */
)
{
    DRESULT res = RES_OK;
    uint8_t result[count][512];

    uint8_t command = CMD18 + 64;

    uint8_t byte1 = (sector & 0xFF000000) >> 24;
    uint8_t byte2 = (sector & 0x00FF0000) >> 16;
    uint8_t byte3 = (sector & 0x0000FF00) >> 8;
    uint8_t byte4 = (sector & 0x000000FF);
    uint8_t cmdarray[6];

    cmdarray[0] = command;
    cmdarray[1] = byte1;
    cmdarray[2] = byte2;
    cmdarray[3] = byte3;
    cmdarray[4] = byte4;
    cmdarray[5] = 1;

    uint8_t data[515*count];
    uint8_t responses[2];

    uint8_t packaged_data[512*count];

    multiple_block_read(cmdarray, count, responses, data);

    // clean out data so only the 512 data bytes are mapped

    for(int j = 0; j < count; j++){
        for(int i = 0; i < 512; i++){
            packaged_data[i+(512*j)] = data[(j*515)+1+i];
        }
    }

    memcpy(buff, packaged_data, 512*count);

    return res;

}


/*-----------------------------------------------------------------------*
/
/* Write Sector(s)
*/
/*-----------------------------------------------------------------------*
/

#if FF_FS_READONLY == 0
```

```
DRESULT disk_write (
    BYTE pdrv,          /* Physical drive number to identify the drive */
    const BYTE *buff,   /* Data to be written */
    LBA_t sector,       /* Start sector in LBA */ // CHECK -> ARE SECTORS
THE SAME AS BLOCKS???
    UINT count          /* Number of sectors to write */
)
{
    DRESULT res = RES_OK;

    uint8_t command = CMD25 + 64;

    uint8_t byte1 = (sector & 0xFF000000) >> 24;
    uint8_t byte2 = (sector & 0x00FF0000) >> 16;
    uint8_t byte3 = (sector & 0x0000FF00) >> 8;
    uint8_t byte4 = (sector & 0x000000FF);

    uint8_t cmdarray[6];

    cmdarray[0] = command;
    cmdarray[1] = byte1;
    cmdarray[2] = byte2;
    cmdarray[3] = byte3;
    cmdarray[4] = byte4;
    cmdarray[5] = 1;

    uint8_t datapackets[515*count];
    uint8_t responses[1+count];
    uint8_t unpackaged_data[512*count];

    memcpy(unpackaged_data, buff, 512*count);

    for(int j = 0; j < count; j++){
        datapackets[j*515] = 0b11111100;
        for(int i = 0; i < 512; i++){
            datapackets[(j*515)+i+1] = unpackaged_data[i+(512*j)];
        }
        datapackets[(j*515)+513] = 0;
        datapackets[(j*515)+514] = 0;
    }
    multiple_block_write(cmdarray, datapackets, count, responses);


    return res;

}

#endif


/*-----------------------------------------------------------------------*
/
/* Miscellaneous Functions
*/
/*-----------------------------------------------------------------------*
/

DRESULT disk_ioctl (
    BYTE pdrv,      /* Physical drive number (0..) */
    BYTE cmd,       /* Control code */
    void *buff      /* Buffer to send/receive control data */
)
{


    DRESULT res = RES_OK;

    return res;
}
```

Appendix E: Additional SD interfacing information

The following is a table of commands and accompanying response types. Note that not all of these commands have to be implemented for effective SD functionality.



Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read OCR.

*1:ACMD<n> means a command sequense of CMD55-CMD<n>.

*2: Rsv(0)[31], HCS[30], Rsv(0)[29:0]

*3: Rsv(0)[31:12], Supply Voltage(1)[11:8], Check Pattern(0xAA)[7:0]

Table E.1: SD SPI mode command list [Chan]

Figures E.1 through E.3 list response formats.

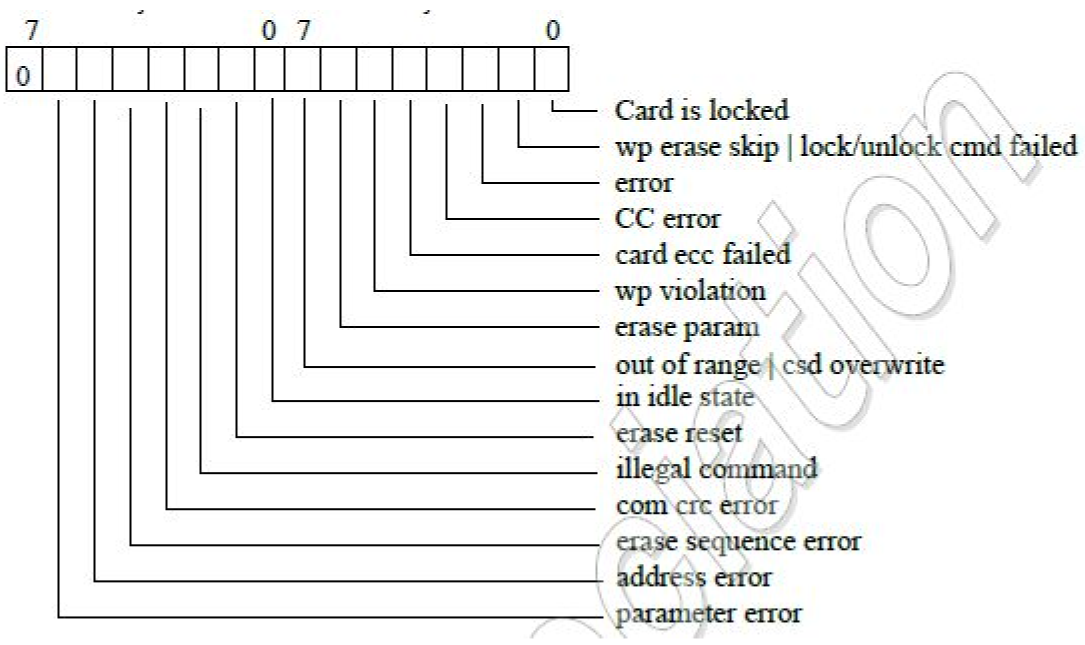


Figure E.1: R2 response format [SD association]

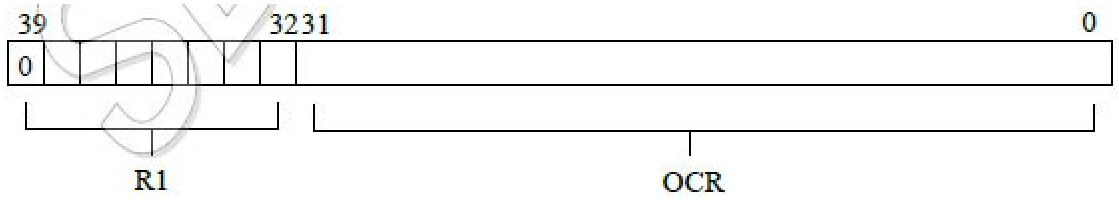


Figure E.2: R3 response format [SD association]

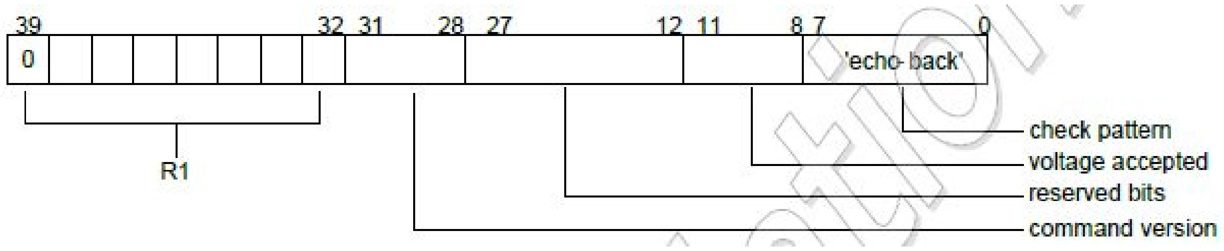


Figure E.3: R7 response format [SD association]

Figures E.4 through E.4 describe read and write operations.

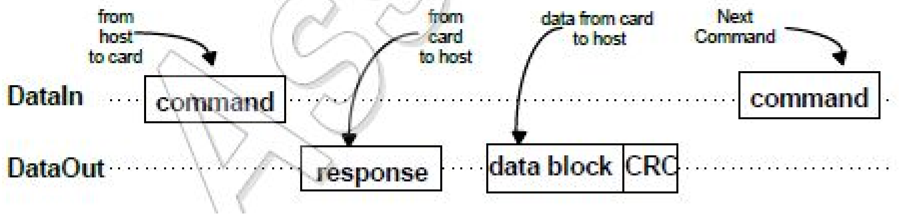


Figure E.4: Single block read operation [SD association]

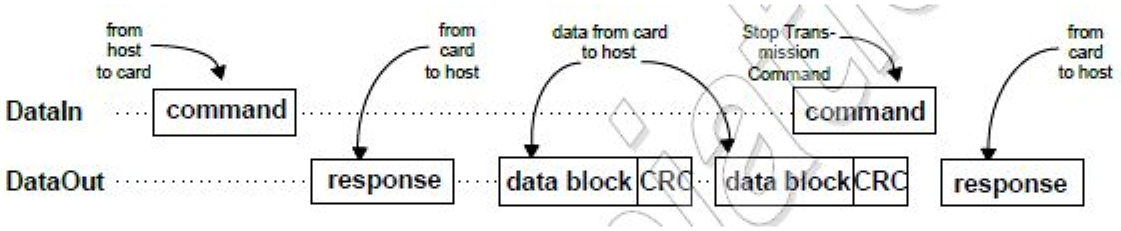


Figure E.5: Multiple block read operation [SD association]

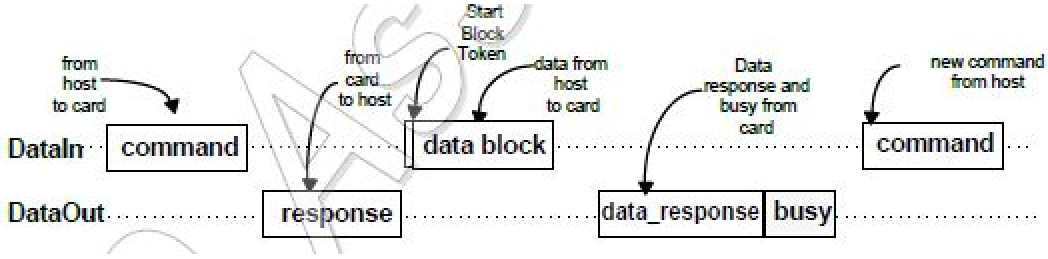


Figure E.6: Single block write operation [SD association]

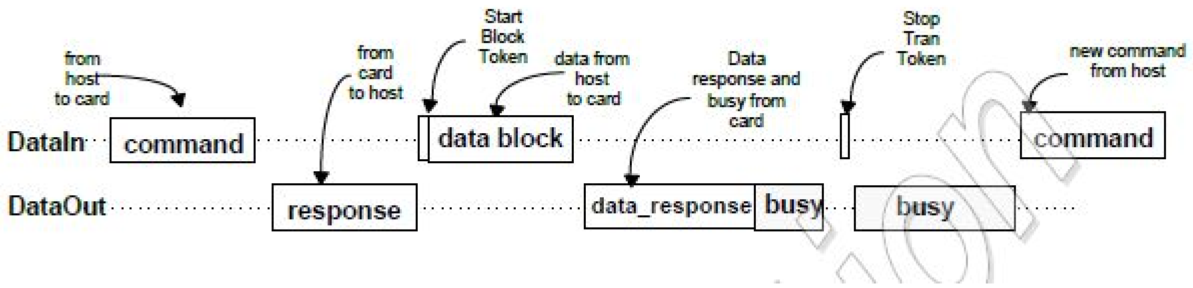


Figure E.7: Multiple block write operation [SD association]