Revision: 1.0

CS4360 High Altitude Balloon

Software Design Document

Robert Susmilch, Zachary Hewitt

October 20, 2016

Contents

1 Introduction

1.1 Purpose

This document will describe the design implementation of a high-altitude balloon (HAB) controller. The controller will facilitate data collection and remote management of the payload for scientific exploration of the Earth's atmosphere in an educational setting.

1.2 Scope

The scope encompasses both software and hardware design:

- The High Altitude Balloon controller requires combining multiple hardware systems in a compact and lightweight payload. This entails a custom motherboard with separate data collection daughter-boards containing sensors, break-out headers and a fail-safe cut-down system.
- Software ties the hardware to the ultimate aim of data collection.
  - ◇ Sensor data is recorded and acted upon in the controller logic.
  - ◇ A radio system to facilitate tracking, command actions and improve payload retrieval after flight termination.[1]
  - ◇ An adjustable cut-down mechanism that releases the payload after a predetermined amount of time or radio signal.[2]

1.3 Audience

The intended audience of this document includes the client and supervising managers. It provides feedback and an overview of the intended system.

[1] To be implemented at a later date.
[2] To be implemented at a later date.

2 Design Overview

2.1 Description of the Problem

The scientific community performs experiments and gathers data from high altitudes, typically from flying a weather balloon. At high enough altitudes, the atmosphere can be termed *near-space,* due to the very low amount at altitude. It is of interest scientifically due to the benefits of low atmospheric disturbances on experiments—such as gathering cosmological data—at a fraction of the cost of a real space launch.

Such near-space flights require the ability to gather and record sensor data, such as temperature and pressure, as well as whatever experiment is primarily of interest on the individual flight. In addition, tracking and recovery of the equipment when a flight has ended is important, in light of the cost and time savings associated with equipment reuse.

Lastly, flights are unpredictable and require fail-safe ways to terminate a misbehaving flight for a variety of reasons:

- Unfavorable flight path characteristics.
- Balloon failing to reach bursting altitude,[1] and thus the payload may not be recoverable.
- Problems with equipment or experiment.

2.2 Assumptions

It is assumed that the primary user will have an understanding of computers and electronics at a daily-life level. The user will understand how to install software, identify and plug in common ports (including battery connections and USB), know basic arithmetic and follow and understand instructions.

It is further assumed that the primary users of this system will include the typical middle school to college level students and teachers. While every effort is made to build a reliable system, the current implementation does not implement various system reliability methods. These additional methods would further complicate and increase the project's cost and time-to-deployment.

2.3 Constraints

The general constraints around HAB flights involve weight, size, cost and ruggedness.

[1] As the balloon climbs, the barometric pressure decreases. This allows the balloon to expand. When the elastic threshold of the balloon material is exceeded, the balloon bursts and the payload parachutes to Earth.

A weather balloon can typically lift a certain amount of weight based on the size of the balloon. Regulatory agencies, such as the FAA, also limit the payload weight. Thus the controller and related systems should be of minimum weight to allow cheaper and light balloons, and allow more free payload capacity for other experiments.

Coupled with weight, as it is a large proportion of it, is battery capacity. Higher capacity batteries increase run-time, but at the expense of weight. Using a lighter, and hence lesser capacity battery, will be beneficial to a HAB flight. It takes no leap of imagination to understand that the conservation of energy is important to allow long flights, more payload capacity or extended payload retrieval once on the ground.

Additionally, the desire for a compact system to allow room for other experiments and data collection technologies limits the controller size. While size is an important consideration, weight is the overriding constraint due to the hard regulatory limit placed on balloon payloads.[2]

Any system is also constrained by budgetary boundaries. This is especially true outside of the government funded studies in academia. Hence, a low cost solution is desirable, this would allow more educational opportunities through continued balloon launches.

Ruggedness of the equipment is also important. The payload is subjected to accelerations and torque upon liftoff, as well as impacts during free-fall and ground landing. This requirement can be fulfilled by a secure means of attachment to both the balloon and the payload container of any hardware.

Lastly, a short time-frame of three weeks between the undertaking of the controller and the scheduled launch is a major constraint on development.

2.4 Goals

The goals of the HAB project are derived from, and similar to, those in the constraints section above (section 2.3 on page 4.)

They consist of:

- Low weight
- Small volume
- Energy efficiency

[2] Cost is another factor. Large balloons require more Helium, which can get quite expensive for educational institutes as a one time use item.

3 Architectural Strategies

3.1 Software Environment

The software used during development will include the standard Arduino IDE[1] with various sensor libraries (open source and typically included with many sensors boards upon purchase). The Arduino framework allows code written in C++ and inline assembly. Custom software will be designed to provide logging and control of the various sensors and subsystems.

Software reuse is an important consideration due to the tight time frame. Many sensors and third party libraries are available to interface with the Arduino software framework. These libraries save time and energy in writing interface code, as well as provide less demand for debugging of software problems. Additionally, future maintenance is enhanced due to the ease of programming and understanding the more simplistic Arduino programming language.[2]

3.2 User Interface

The controller interface is straight forward. During launch the user will apply power by plugging in the battery and observing if a GPS signal is aquired through the built-in LED indicator.

After successful recovery, the user will remove power to the unit.

When logged data is required, the user will reapply power and hold the interface button to exit recovery mode. Upon exiting recovery mode the user will plug in a serial cable to the appropriate on-board header and use a terminal program to send commands to download the raw data from the SDCard.

3.3 Data Storage

Data is stored on an SDCard for ease of availability and large storage capacity. Data integrity is important and is accomplished by detecting and correcting errors during data logging.

Successful detection of errors is done through CRC16 checksums coupled to every SDCard write. Before every potential block of data is written to the card, the card block is first checked if empty. If the SDCard block is empty, data is written to that block and immediately read back into a separate area of memory. This is then compared with the valid sensor data to ensure successful data write and retrieval. If the data is found to not match, the controller will continue to attempt to write and check subsequent blocks. Each block is written in a raw format to guard against file system corruption (if power loss or a watchdog reset occurs.)

[1] The Arduino IDE can scarcely be called an IDE. However, time shortages do not allow migration to a more full featured IDE, or distancing from the Arduino core libraries.

[2] Except for the standard C and C++ libraries, all C++ functionality is available.

If a reset occurs, the controller finds the tail, or end, of the log (where data was last written) and continues to log data. This protects against overwriting of previous data.[3]

3.4 Hardware

The mother and daughter board prototypes are hand made to reduce time to deployment.[4] PCBs are hand laid out and soldered. Custom PCB design allows tight integration with control over all aspects of the design. It also reduces weight and size by eliminating unnecessary components that would come with a conventional Arduino board.

[3]The user must explicitly erase the SDCard by writing zeros to it.
[4]Typical turn-around time for affordable custom PCB boards are 3–4 weeks.

4 System Architecture

4.1 Hardware

The HAB controller will be based on:

- An Atmel ATMEGA2560 microcontroller. This is a small system-on-a-chip (SOC) that allows communication with various sensors and peripherals without needing additional components.
- Various sensors such as,
  - ◇ Magnetometer
  - ◇ Gyroscope
  - ◇ Accelerometer
  - ◇ Geiger counter
  - ◇ Temperature
  - ◇ Barometric pressure
  - ◇ Humidity
  - ◇ Light level
  - ◇ Global Positioning System (GPS)
- Recordable medium such as EEPROM and SDcards.
- Resistive heaters to control sensor and experiment temperatures while in flight.
- Payload retrieval components,
  - ◇ A siren or buzzer to allow auditory location of payload once on the ground.
  - ◇ LEDs to attract attention visually.

The cut-down unit will include,

- A nichrome resistive wire that will cut through the nylon payload supporting cord.
- A microcontroller that will implement a timer and react to barometric pressure.
- A spring loaded housing to provide force to pull the nichrome wire through the payload line when activated.

4.1.1 Schematics

The main control board consists of the ATMEGA2560 microcontroller shown in fig. 1 on the next page.

Additionally, supporting components that are needed are shown in sub-schematics. The auxiliary connections, to control such features as power toggling of sensors, power distribution and filtering, and real time clocks are shown in fig. 2 on page 11.

The control module regulates the temperature of the various subsystems via resistive heaters. This schematic is shown in fig. 3 on page 12.

Lastly, a proposed SDCard backup, consisting of discrete EEPROM chips, is shown in fig. 4 on page 13.

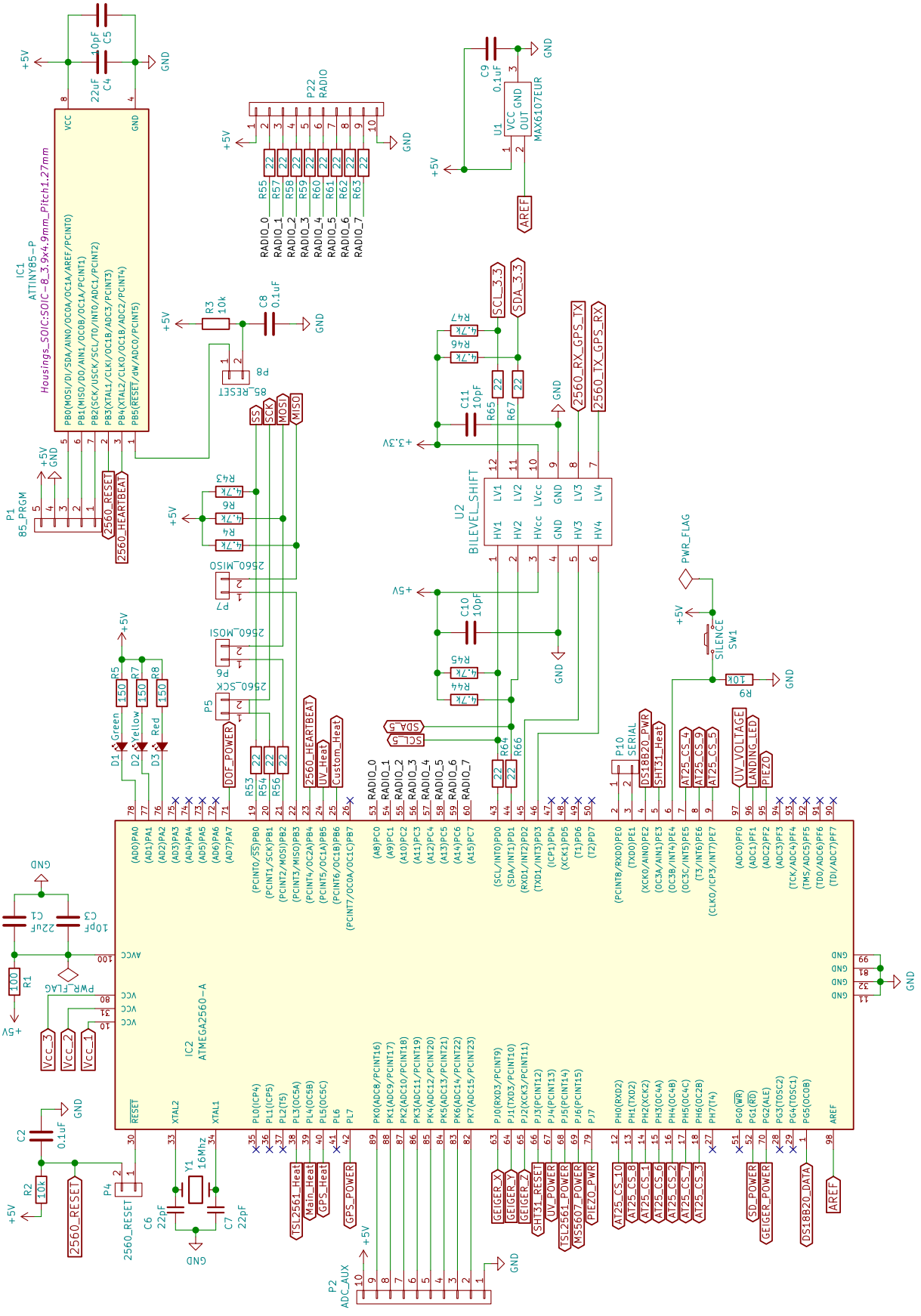

Figure 1: Main HAB controller schematic.

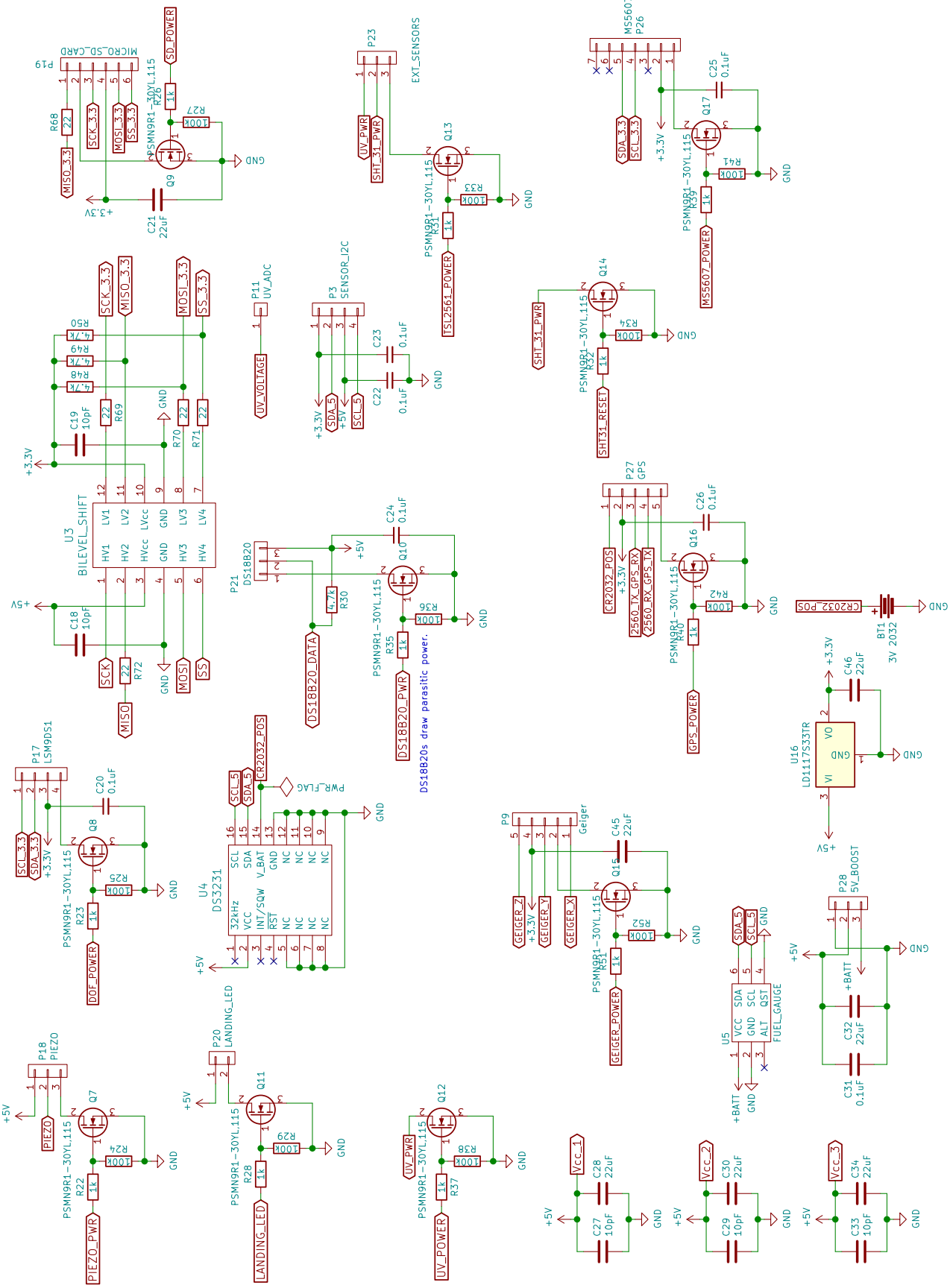

Figure 2: Schematic detailing the auxiliary connections from motherboard. Controls various aspects of the controller such as power and communications.

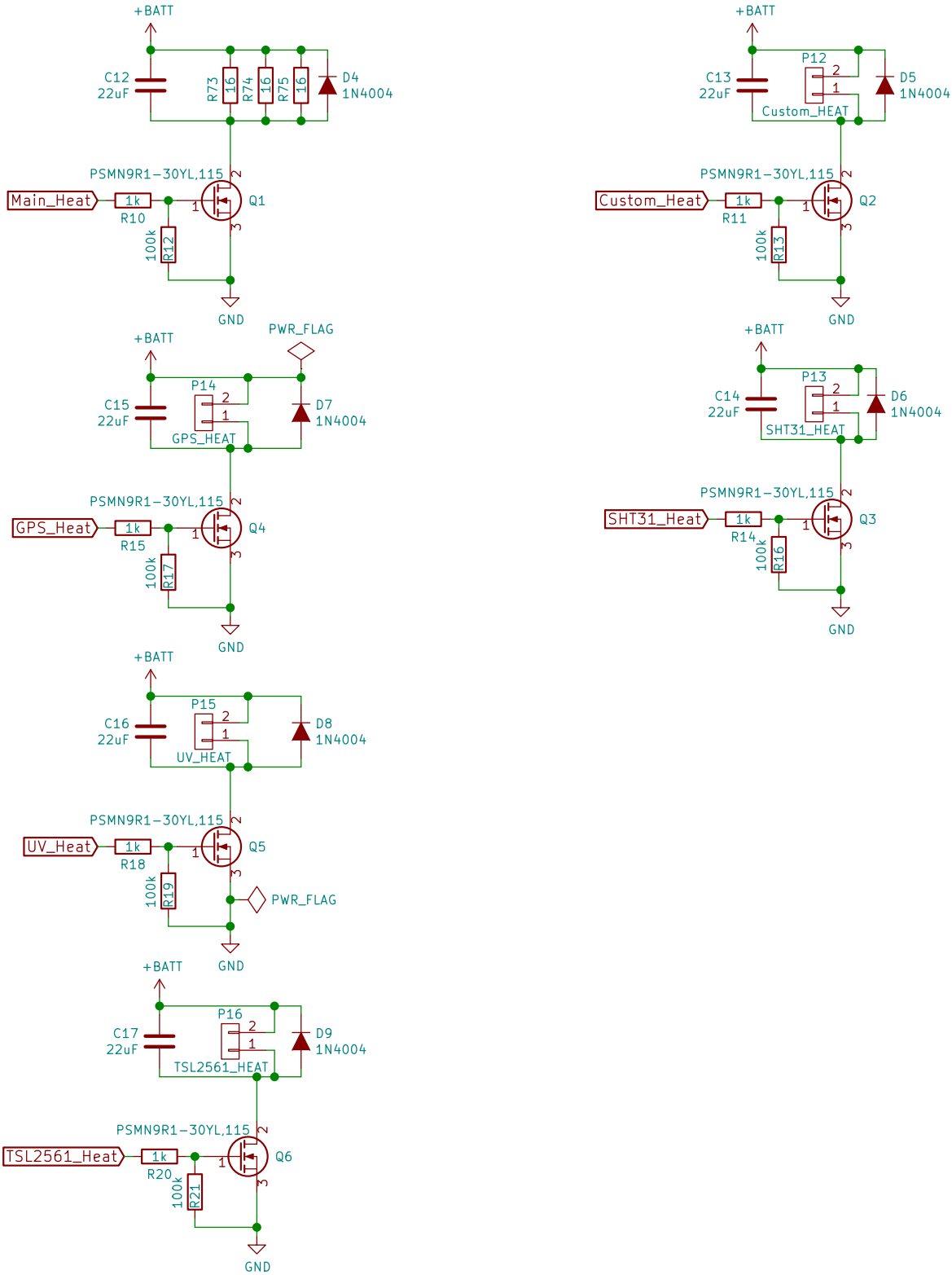

Figure 3: Heat control schematic.

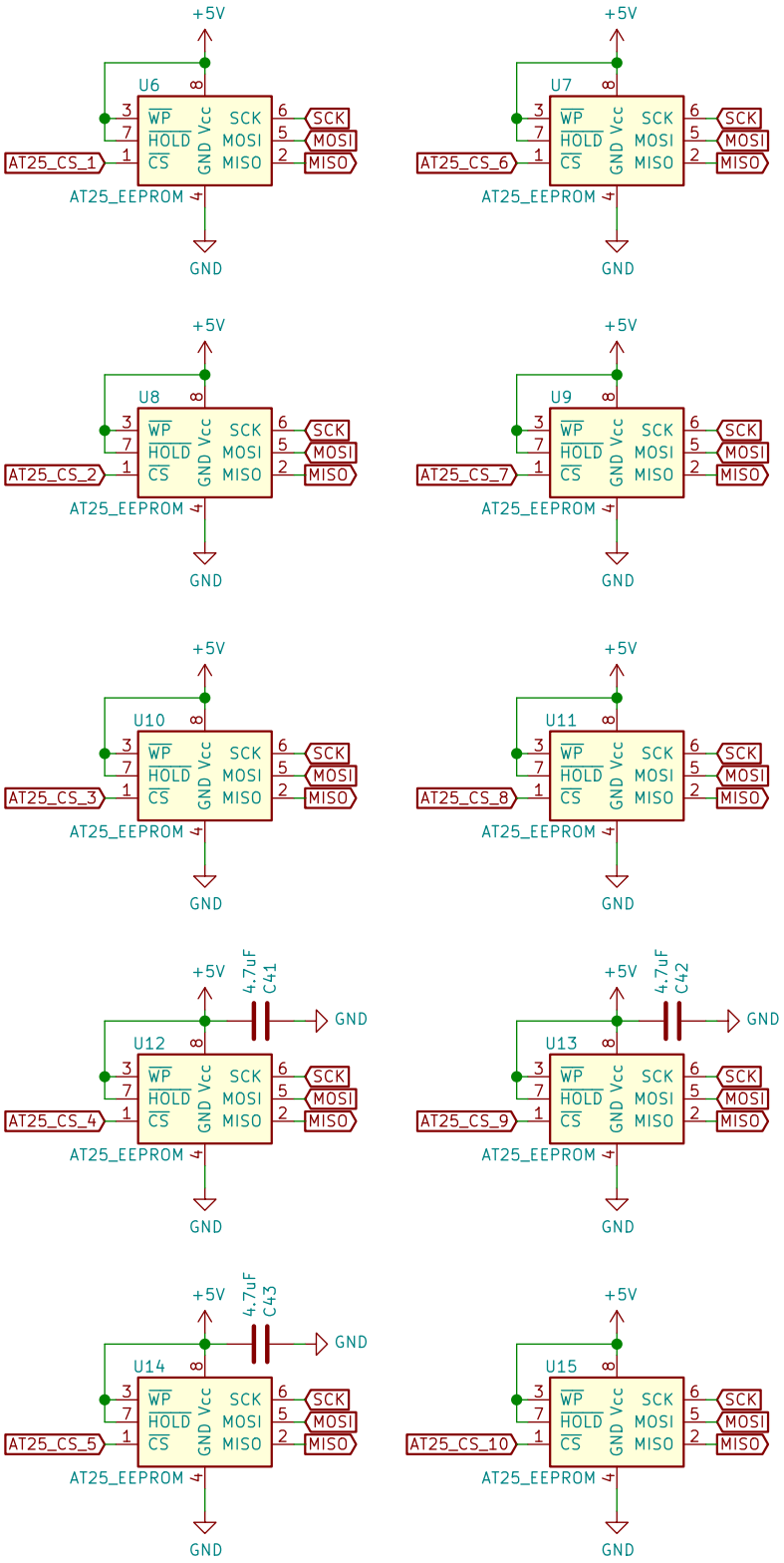

Figure 4: Proposed EEPROM schematic for backup datalogging.

4.1.2 PCB Design

The printed circuit board design in given for the front (or top) of the board in fig. 5 on the next page and back (bottom) in fig. 6 on page 16.

4.2 Software

4.2.1 Overview

The software of the HAB controller is primarily responsible for polling various sensors and recording the data for later analysis.

The controller communicates with the following sensors:

- Temperature sensors
  - Dallas Semiconductor DS18B20s mounted near the following sensors or locations:
    - GPS Receiver
    - ML8511 UV light sensor
    - TSL2561 Lux light sensor
    - SHT31 Humidity and temperature sensor
    - Internal header break-out board
    - Free hanging outside the payload container (external temperature)
  - Additionally, the DS3231 Real time clock contains a temperature sensor that is designated the motherboard temperature for thermostatic regulation.
- Humidity through SHT31
- Light levels
  - TSL2561 Lux and Broadband light level sensor
  - ML8511 UV sensor
- Ionizing radiation through a modified MightyOhm Geiger counter board
- Barometric pressure via a MS5607
- LSM9DS1 Nine degrees of freedom sensor
  - Magnetometer
  - Gyroscopic
  - Accelerometer

The controller also communicates with a microSD flash card to record the relevant data.

Lastly, the software is responsible for flashing LEDs and sounding an acoustic alarm to aid in payload retrieval once it determines it has landed.

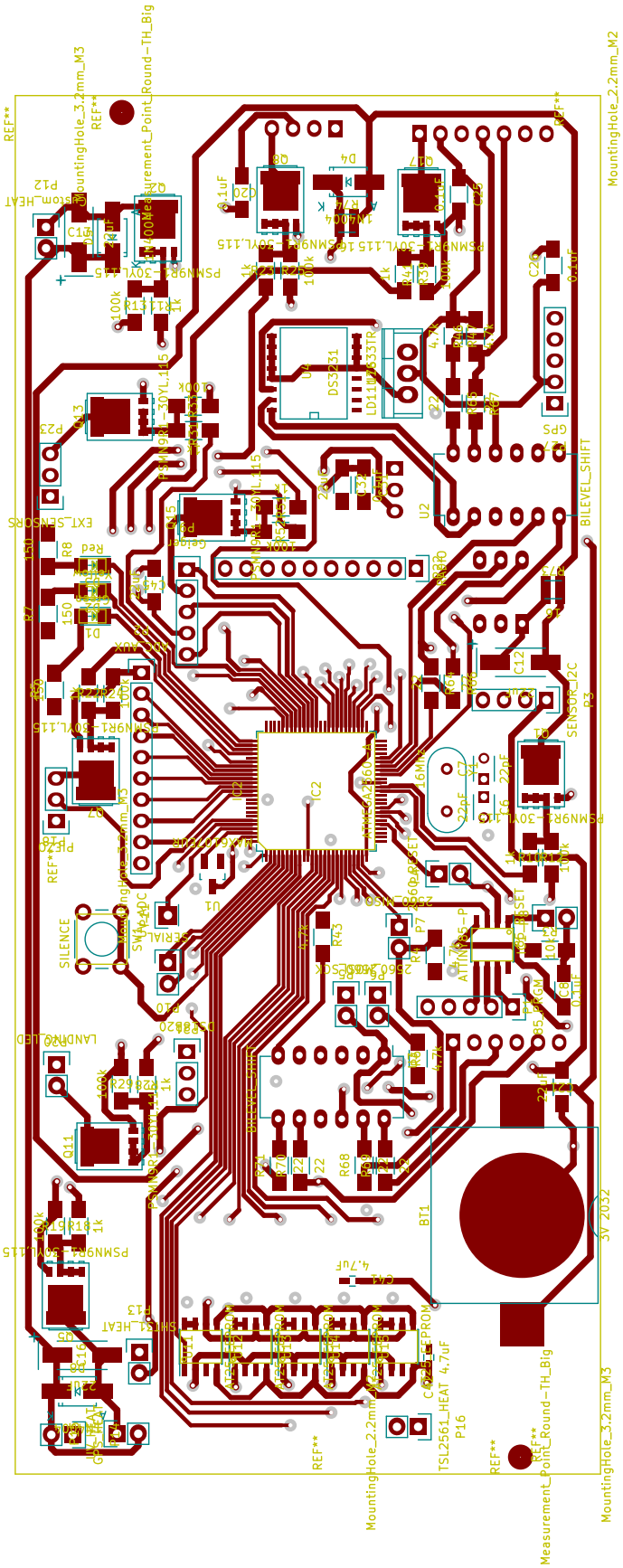

Figure 5: Front PCB routing layout.

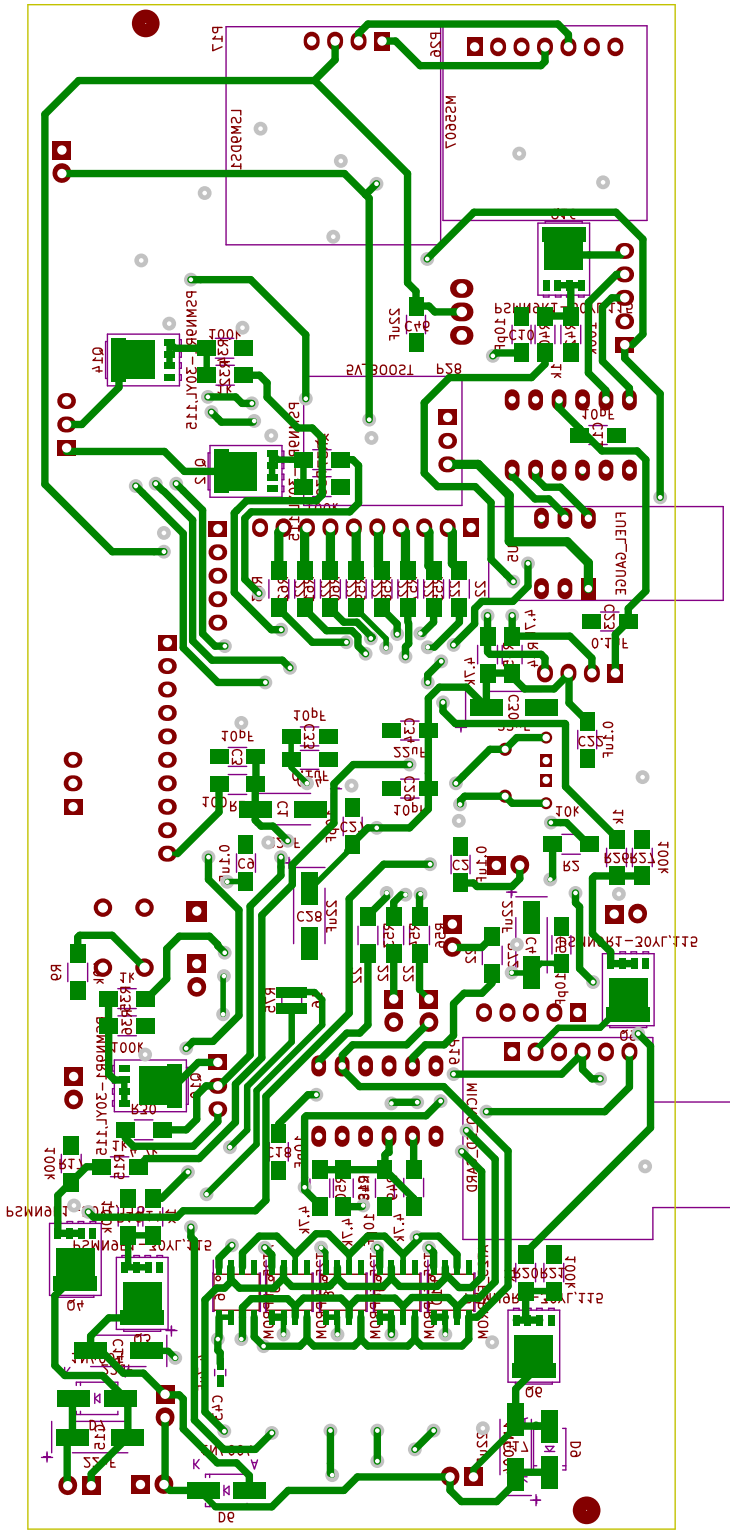

Figure 6: Back PCB routing layout.

5 Implementation

5.1 Overview

The C++ `#include` diagram for the main program code, which shows the various sensor classes and supporting libraries, can be seen in fig. 1 on the next page.[1]

Certain flight parameters are stored in non-volatile EEPROM memory to enable communication across microcontroller restarts. These include flags if the payload has been launched or determined to have landed. Additionally altitude and pressure thresholds for those flags are also stored.

Overall program flow can be seen in the flow chart in fig. 2 on page 19

Program execution begins with initializing sensors and data structures, as well as checking for errors or unintended resets. The function calls from `setup()` are given in fig. 3 on page 21, with a more detailed examination in section 5.2 on page 20.

Control then passes to the main control loop, `loop()`, where the controller does various logging and parameter checking. `loop()` function calls are seen in fig. 6 on page 24 with an overview in section 5.3 on page 23.

[1] Maybe...

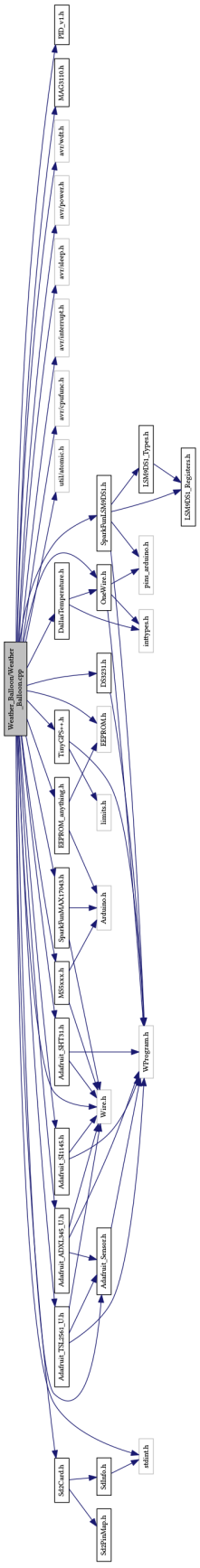

Figure 1: Include files for the HAB controller main program.

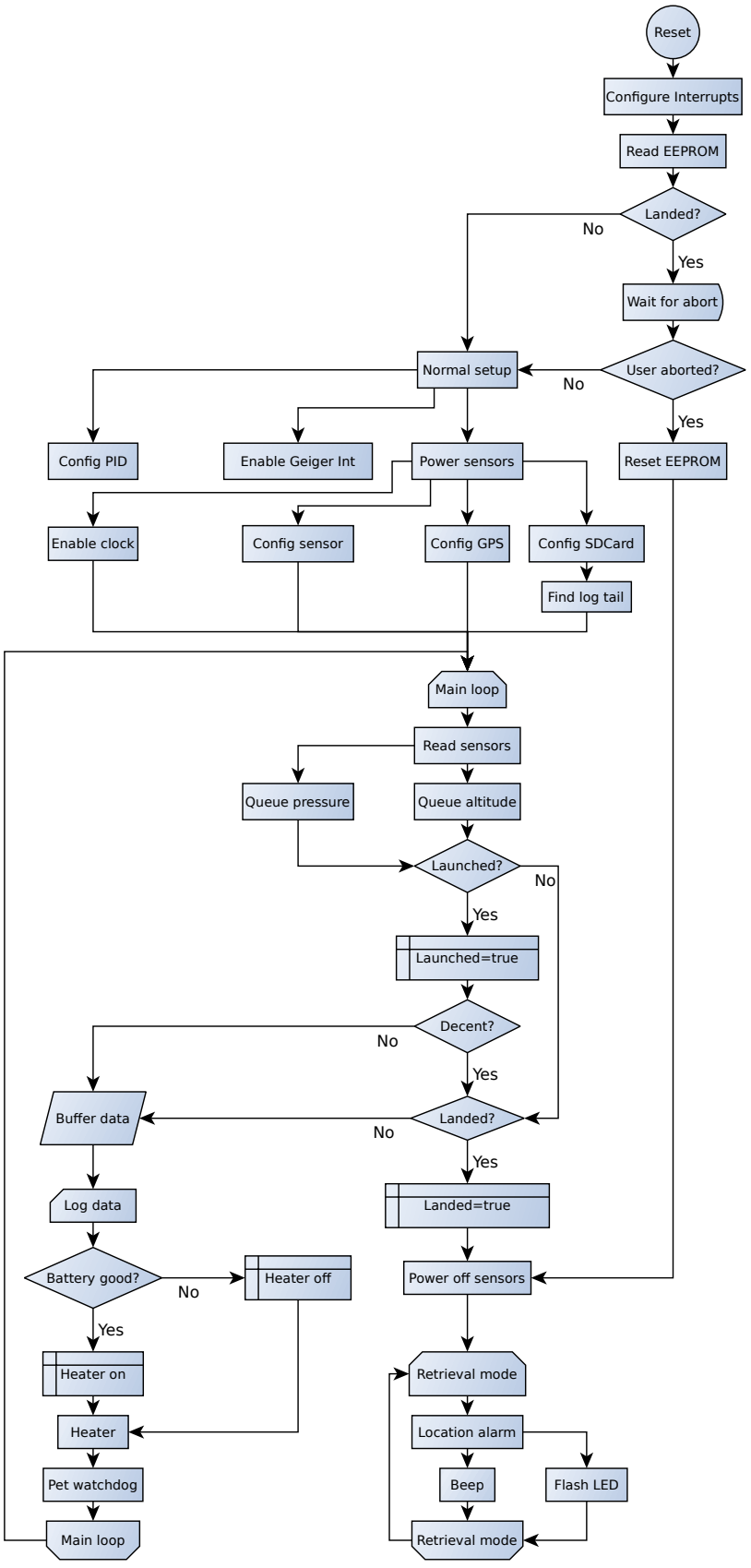

Figure 2: Program execution flow.

5.2 Setup()

The setup function (fig. 3 on the next page) begins by safely conserving power through disabling sensor power (section 5.4 on page 29.) This ensures that a rogue reset does not leave sensors unnecessarily powered.[2]

The controller configures needed pin change interrupts for the user interface, then restores any settings from previous sessions by reading data from the onboard EEPROM.

If it is determined that the payload has landed previously, and would enter the location retrieval phase of deployment, `setup()` calls a user interface function, `count_down_led()`. This function allows the user to exit the low power retrieval mode and reset associated data parameters by a timed press of an on-board button. This insures the user can enter flight mode in the field without programming or other cumbersome equipment.[3]

If the retrieval flag has *not* been set, the execution continues to configure the sensors (such as fig. 4 on page 22 and fig. 5 on page 22), PID heater control, initialize SDCard (section 5.5 on page 29) and determine the end of any previous logged data (section 5.6 on page 30) as well as configure additional interrupts. Control then passes to the main program `loop()` described in section 5.3 on page 23.[4]

However, if the retrieval flag *has* been set, and the user interaction timer is allowed to expire, and control of the program passes to the location retrieval function, `location_alarm()`, detailed in section 5.8 on page 31.

[2] An impact during landing may temporarily remove power to the controller, resetting it. Since the controller does not need sensors after payload cut-down, powered sensors reduce the location retrieval energy budget.

[3] For example, leaving the controller on too long causes it to assume faulty pressure or GPS altitude readings, and that it has launched. A user could reset the controller prior to launch and allow data recording for the upcoming flight.

[4] Arduino IDE deemed a normal `main()` with an embedded `while` loop too complicated for ordinary users. Hence, Arduino's `main()` continuously calls `loop()` for the developer. Local variables can be lost, since the function returns and they go out of scope.

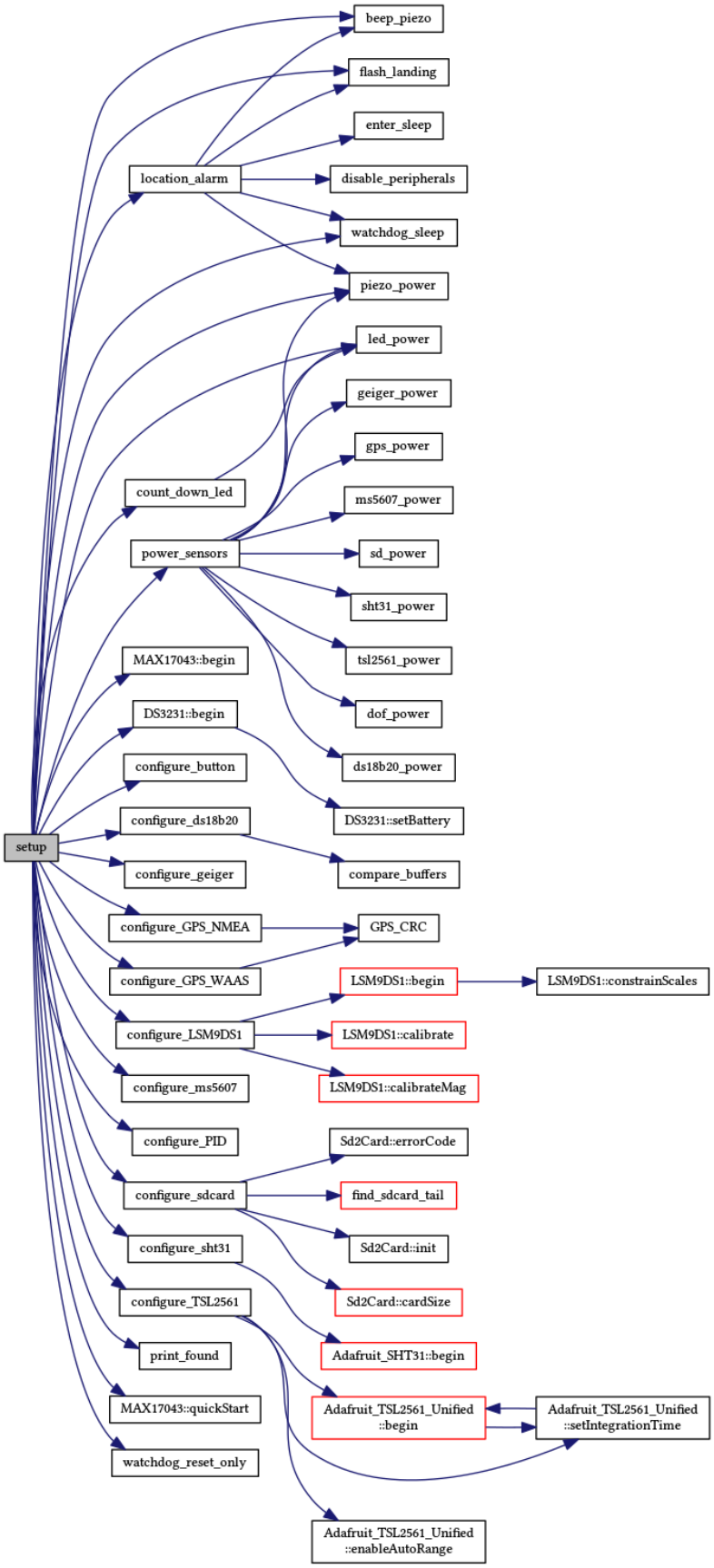

Figure 3: Overview of the function calls from the initial setup function.

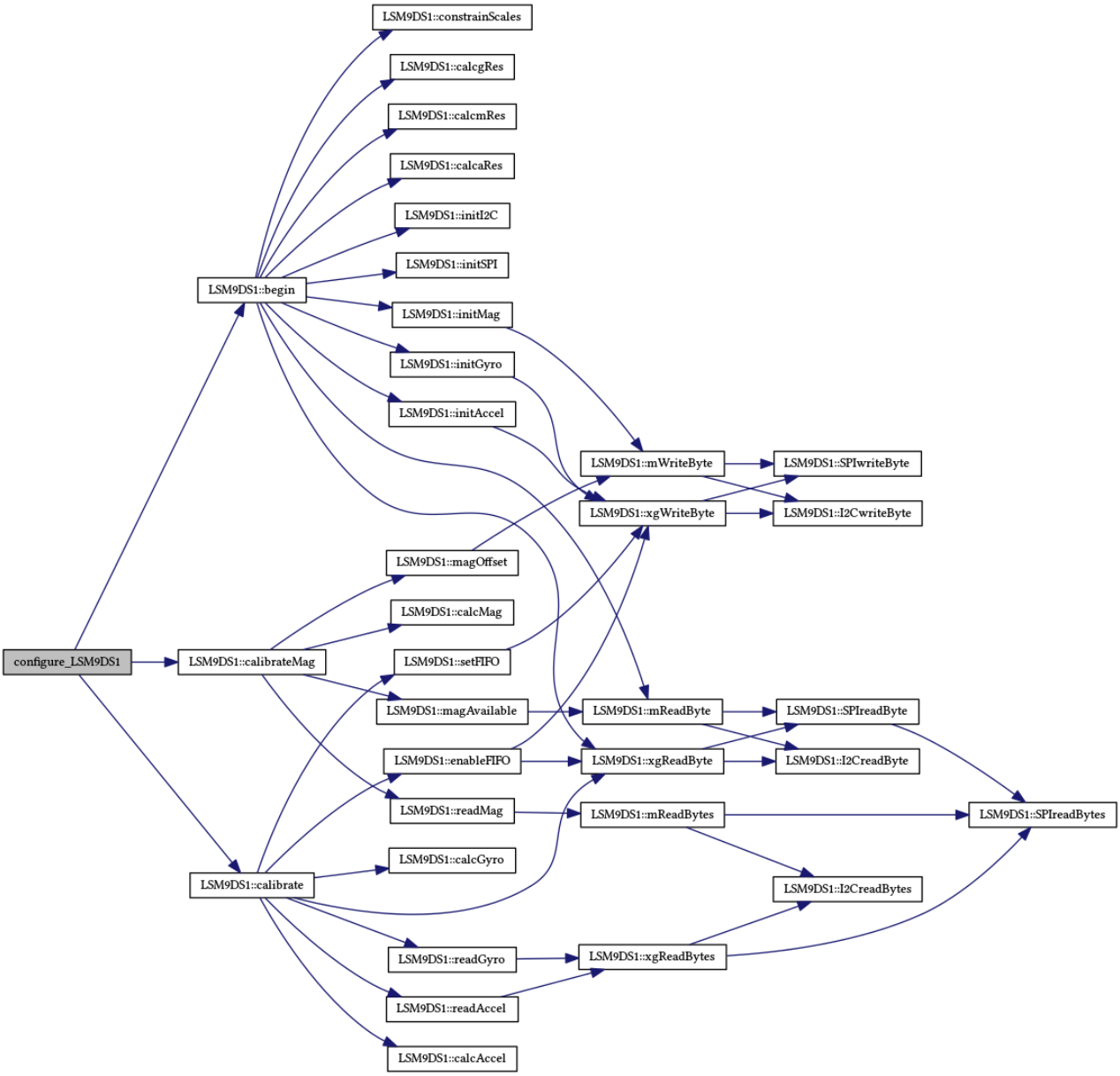

Figure 4: Function calls to configure the LSM9DS1 acceleration/gyroscope/magnetometer sensor.

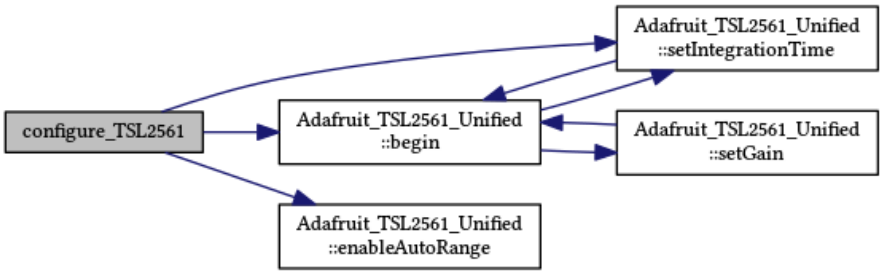

Figure 5: Function calls to configure the TSL2561 light sensor.

5.3 Loop()

The main program logic is contained in function calls from `loop()` (fig. 6 on the next page.)

`loop()` consists of the high level calls to read the sensors (fig. 7 on page 25, fig. 12 on page 28), check for launch and decent events, ration battery energy, call for heater PID updates, and determine if the watchdog should be reset (section 5.9 on page 32.)

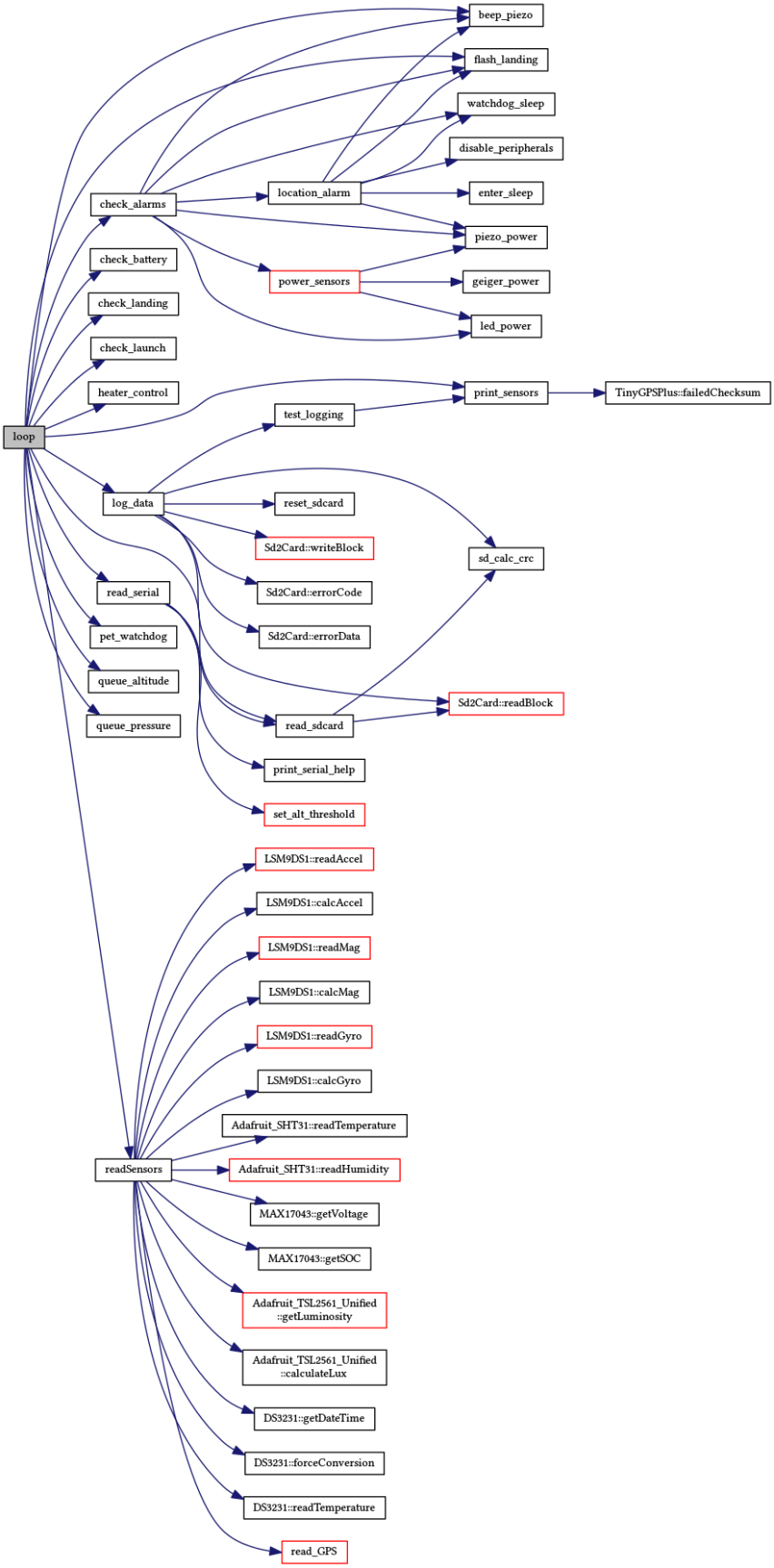

Figure 6: Function calls from the main loop function.

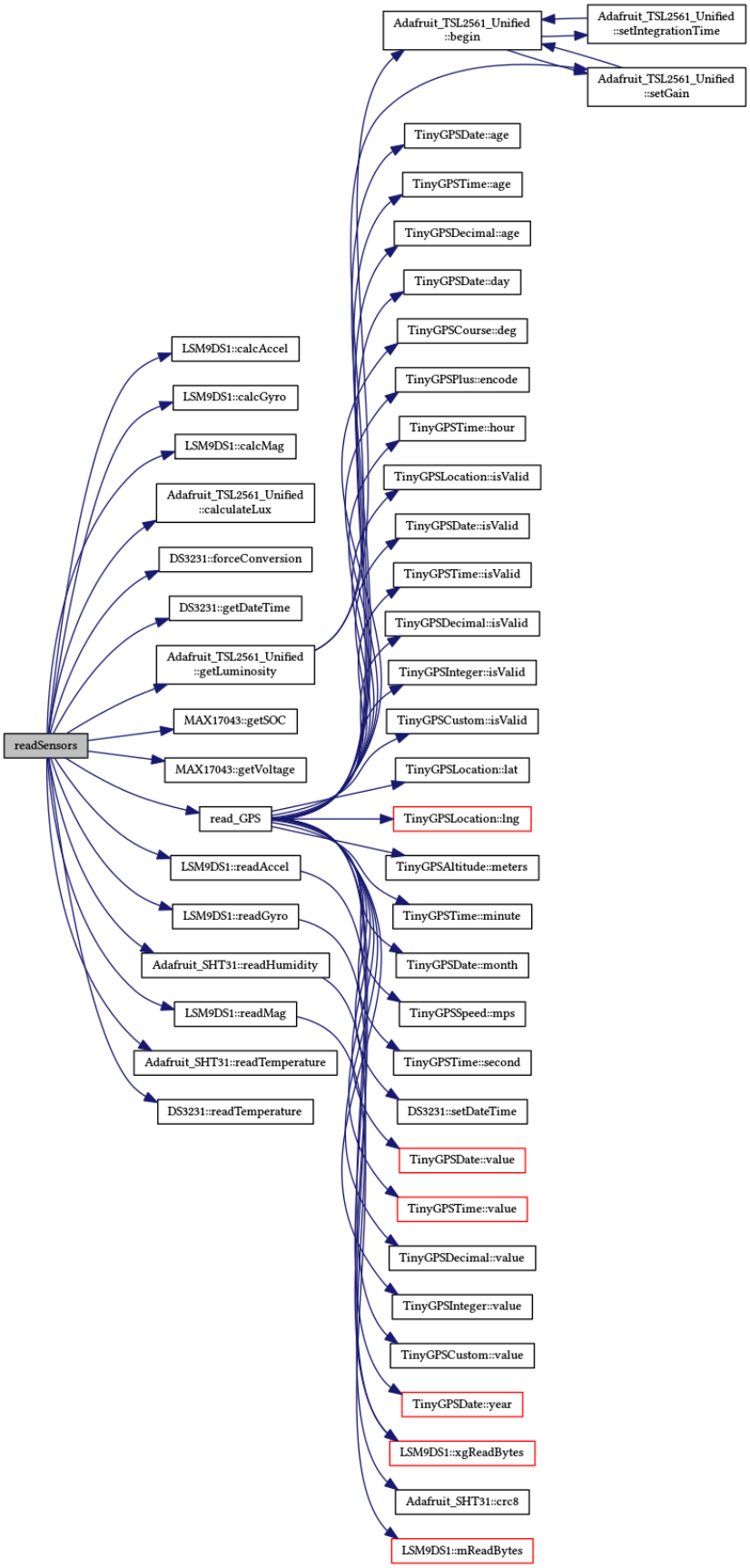

Figure 7: Function calls from the readSensors function.

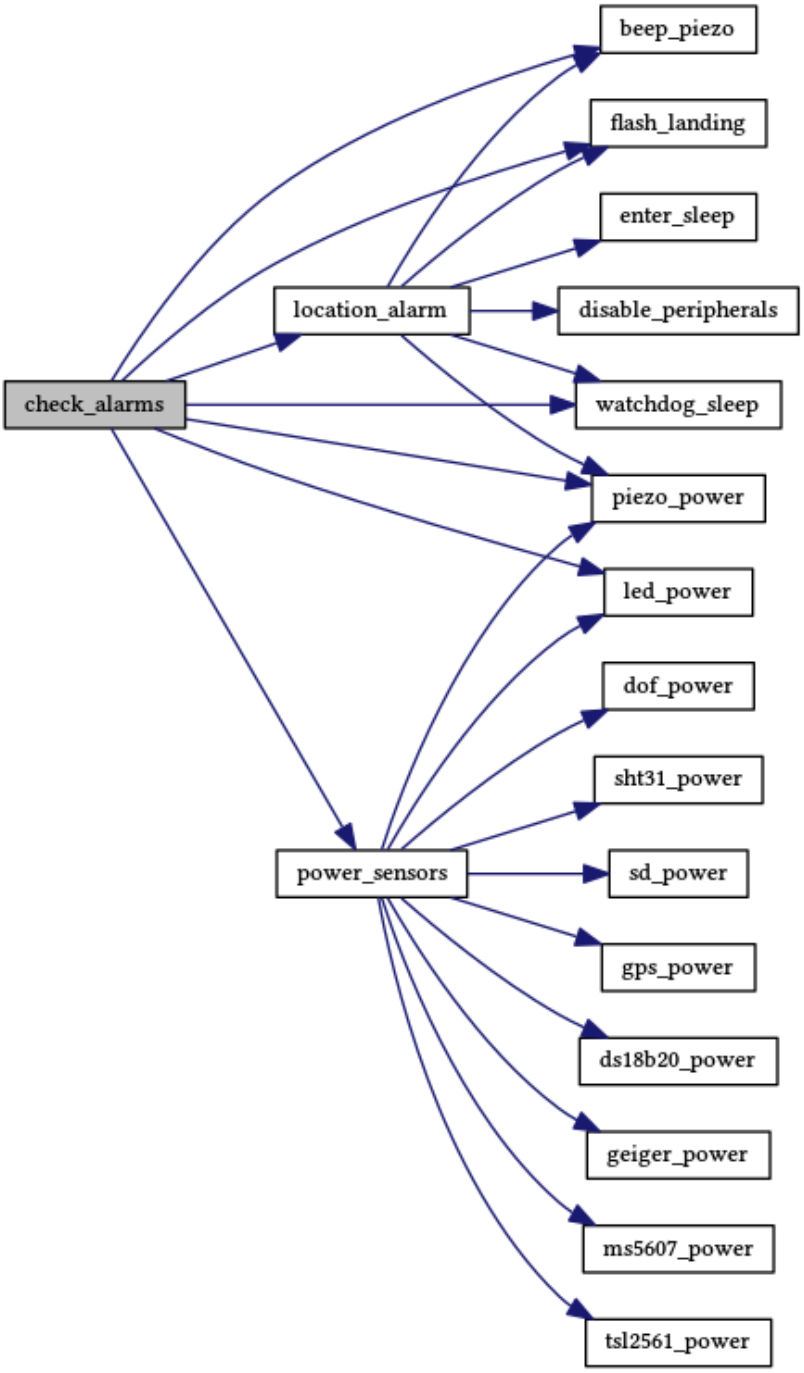

Figure 8: Function calls for the check_alarms function.

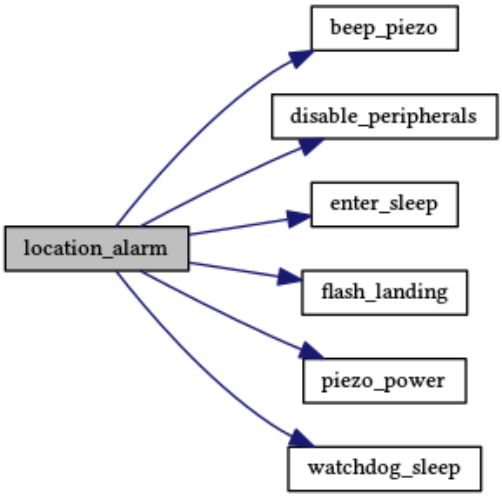

Figure 9: Function calls from the location_alarm function.

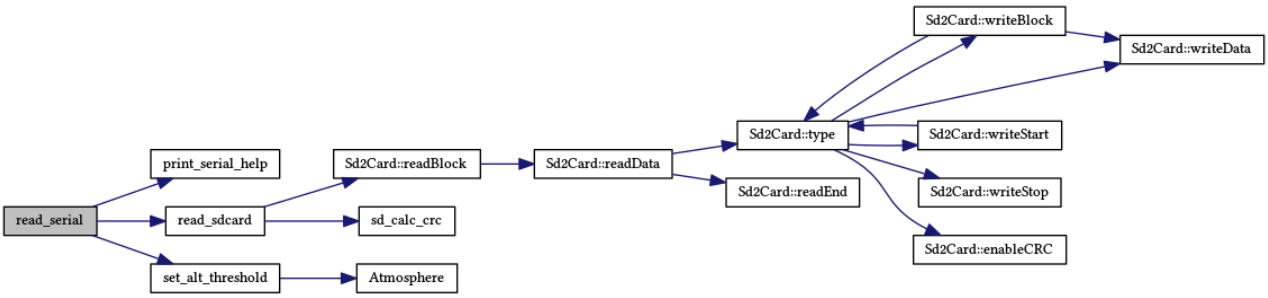

Figure 10: Function calls from the read_serial function.

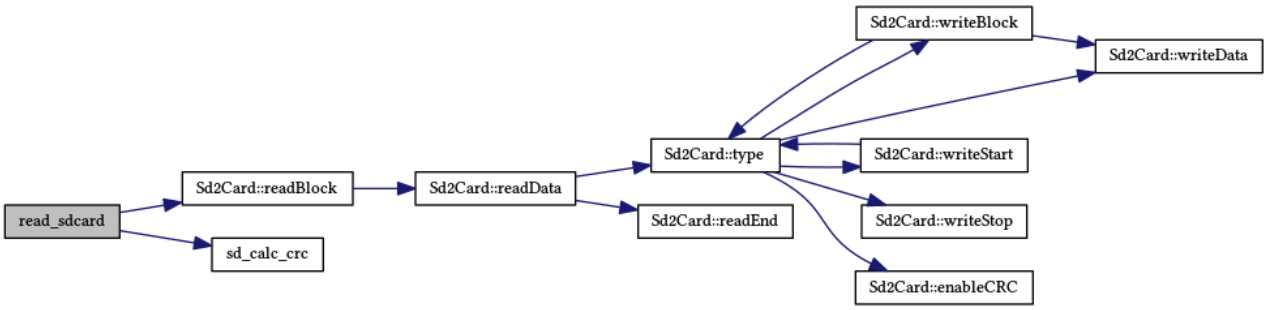

Figure 11: Function calls from the read_sdcard function.

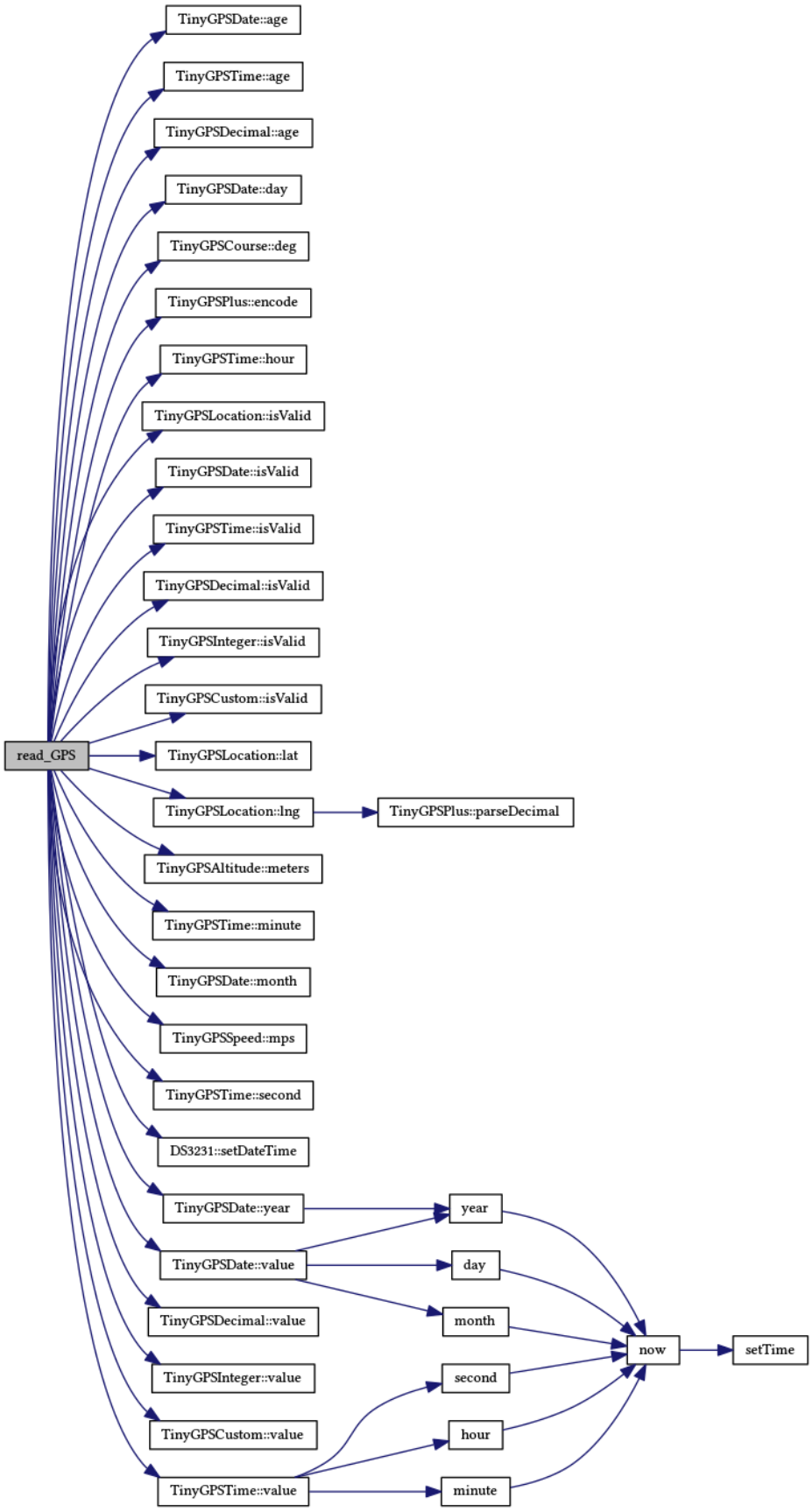

Figure 12: Function calls from the read_GPS function.

5.4 power_sensors()

The `power_sensors()` function passes a given argument to various sub-functions to disable power to all sensors via discrete MOSFET transistor components. This is called to enable or disable all sensors as a group. The rational behind this is to conserve power for later payload retrieval. Low battery levels require conservation of energy so that `location_alarm()` (section 5.8 on page 31) will have sufficient energy to aid in location finding and equipment reclamation.

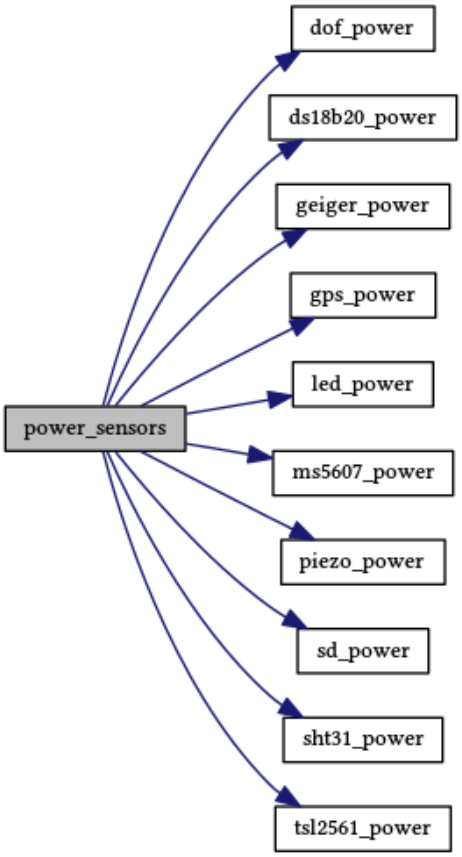

Figure 13: Function calls from the power_sensors function.

5.5 configure_sdcard()

This function (fig. 14 on the following page) initializes the SDCard via each cards built-in SPI bus. Following successful configuration, the tail (or last entry) of the log is determined by calling `find_sdcard_tail()`, shown in section 5.6 on the next page.

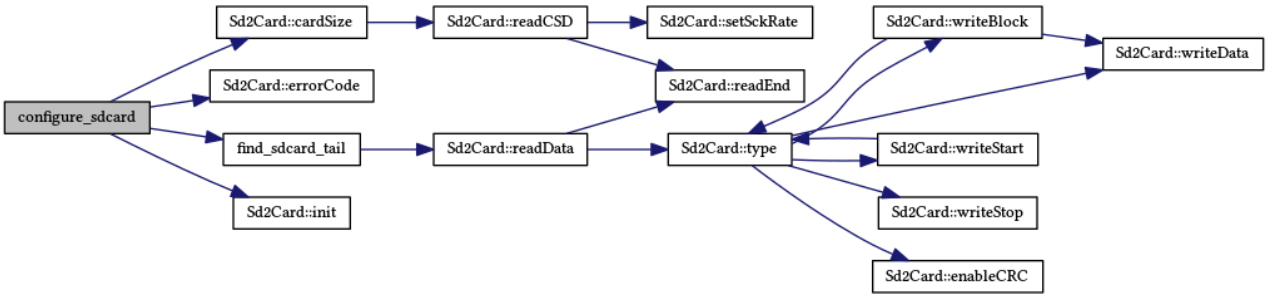

Figure 14: Function calls from the configure_sdcard function.

5.6 find_sdcard_tail()

The smallest size access unit for SDCards is a 512 byte block. This function (fig. 15) utilizes a binary search algorithm to search for the last used SDCard block. The algorithm's focus is for the first completely empty block on the card. This block is assumed to be the tail of the log and set as the next location for logging data.[5]

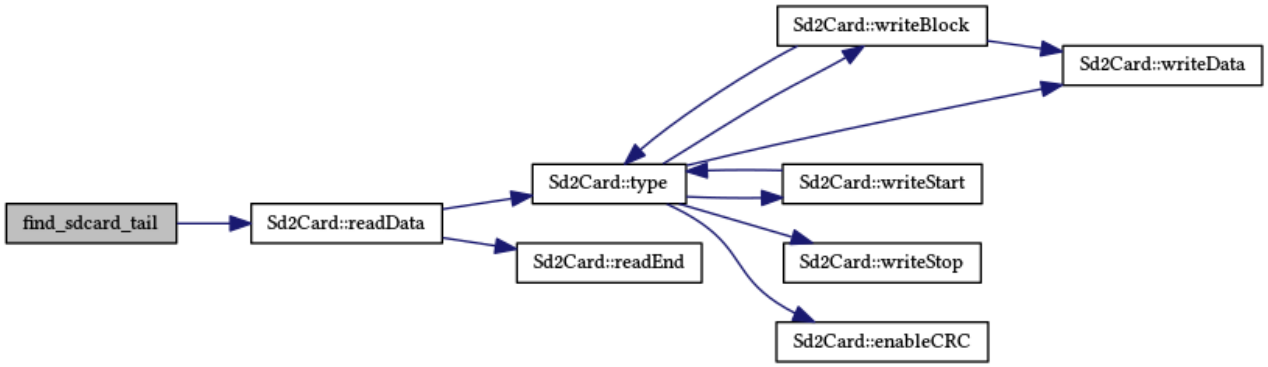

Figure 15: Function calls from the find_sdcard_tail function.

5.7 log_data()

This function (fig. 16 on the next page) packs multiple sensor readings into one 512 byte SDCard block. When enough data readings are accumulated a CRC16 checksum (algorithm 5.1 on the following page) is generated for, and appended to, that block. The function then zeros a temporary buffer and reads the next assigned block from the SDCard. This is to verify if the location is empty, to avoid overwriting previous data.

If the block is *not* empty, the function increments the current block to the next value and repeats the verification. This process is repeated until either the end of the SDCard is encountered, or a valid empty block is found.

[5] Assumed because sequential access to modest size cards would take hours to cycle through (15-20 million blocks). A binary search finds the end of the log in, at most, 24 block reads for an 8 GB SDCard. Worst case access is $O\left(\log n / \log 2\right)$.

Once an empty block is verified, the data is written to the SDCard. The data is then immediately reread into the temporary buffer and another CRC16 checksum is generated for comparison.

If the checksums *do* match, the current block variable is incremented for the next iteration of data logging.

If the checksums do *not* match, the current block variable is incremented and the process of searching and verifying a valid location and successful write begins again. This guards against silent write errors, where the card reports a successful write but has silently corrupted the data.

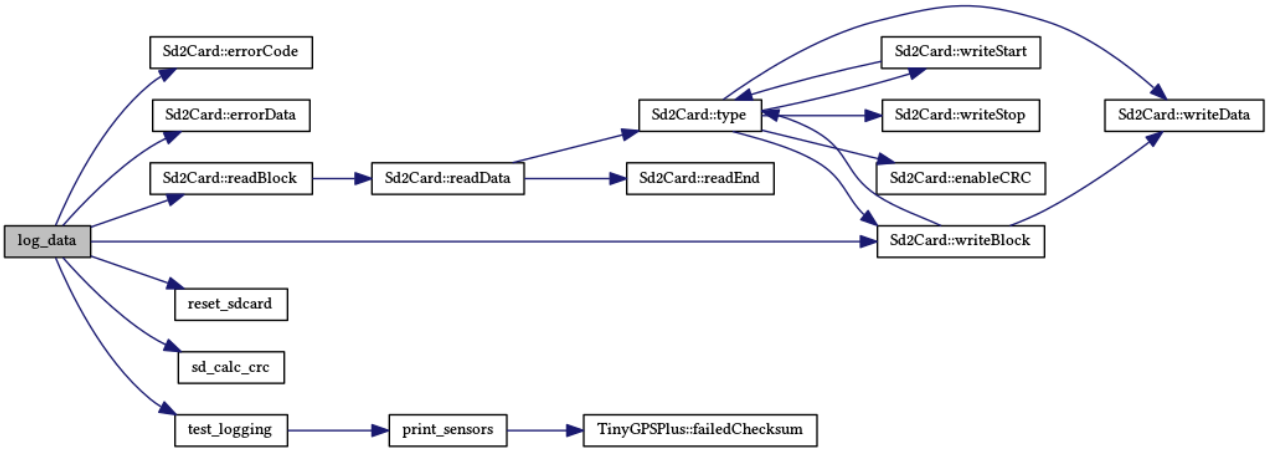

Figure 16: Function calls from the log_data function.

Algorithm 5.1 CRC16 implementation.

```
int16_t sd_calc_crc(const uint8_t* src, const uint8_t
 ↪ block_size) {
  int16_t crc, i, x = 0;

  for(crc = i = 0; i < block_size; i++) {
    x = ((crc » 8) ^ src[i]) & 0xff;
    x ^= x » 4;
    crc = (crc « 8) ^ (x « 12) ^ (x « 5) ^ x;
  }

  return crc;
}
```

5.8 location_alarm()

This function's sole task is to wake up due to a watchdog interrupt, flash the external LEDs and sound a short attention getting siren. It then resets the watchdog and places the microcontroller back into a low-power sleep mode.

5.9 pet_watchdog()

The `pet_watchdog()` function (algorithm 5.2) is designed to ensure all pieces of the control module are functioning correctly.[6] This is accomplished through strategic placement of checks within all functions that should be successful in a correctly running program. When `pet_watchdog()` is called, it checks a volatile watchdog sentinel variable against an expected value. If the value does not match, it is assumed the microcontroller is in an unknown state and the function enters a loop awaiting a watchdog timer system reset.

The watchdog variable is a state variable for a long "chain" of appropriate and successful program execution.

Looking at the chain example of algorithm 5.3 on the following page, upon entering a function, the volatile watchdog sentinel variable is checked against an expected value. The expected value is derived from the previous successful function, which updated the watchdog variable prior to returning (in this example case, successful completion of `pet_watchdog()`.)

As each of the expected values are encountered, the watchdog variable is set to another value for the next appropriate watchdog check.

Upon successful completion of the current function, the watchdog variable is again checked against what it is expected to be at that point of program execution. If it matches, once again it is updated to a unique value for the next function.

This chain of values ensures that corruption of the program counter, buffer overflows, or other unintentional actions, do not place the microcontroller in an unexpected or undefined state. A run-away program counter may jump anywhere within the program space executing random instructions, but it would be very unlikely to set the watchdog variable correctly for a valid watchdog timer reset to occur.

Algorithm 5.2 Watchdog sentinel checking. Only if the value matches what is expected is the watchdog reset. Other values allow the watchdog timer to reset the microcontroller.

```
void pet_watchdog() {
  // Make sure that registers are flushed and
  ↪ optimizations don't fiddle with our watchdog variable.
  _MemoryBarrier();
  if (watchdog == 210) {
    wdt_reset();
    watchdog = 20;
    _MemoryBarrier();
  } else {
    // Wait for reset.
    while(true) {}
  }
}
```

[6] A good primer is found at Designing Watchdog Timers for Embedded Systems (`http://www.ganssle.com/watchdogs.htm`).

Algorithm 5.3 Example of the watchdog chain at the beginning of `loop()`.

```
void loop() {
  if (watchdog == 20) {
    watchdog = 30;
  }

  /* Execute loop body code.
  *  Each called function checks and
  *  updates watchdog along the way */

  if (watchdog == 200) {
    watchdog = 210;
  }
  pet_watchdog();
}
```

5.10 read_sdcard()

This function is initiated by the user. It reads and parses the data off the SDCard, and outputs a comma separated value stream over the serial terminal. The user is expected to save this output to a file. A comma separated value file is an industry standard used by many spreadsheet, graphing and statistical programs.[7]

[7] My give-a-damn ran out.

Nomenclature

Accelerometer	An accelerometer is a device that measures acceleration.
Barometric pressure	The local pressure of the atmosphere as measured by a sensor or instrument.
Checksum	A checksum is a small-sized datum from a block of digital data for the purpose of detecting errors which may have been introduced during its transmission or storage. By themselves, checksums are often used to verify data integrity, but is not relied upon to verify data authenticity.
Controller	Generically, the HAB data collection and payload control system.
CRC16	A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Blocks of data entering these systems get a short check value attached, based on the remainder of a polynomial division of their contents. On retrieval, the calculation is repeated and, in the event the check values do not match, corrective action can be taken against data corruption.
Cut-down	A device that releases the payload from the balloon upon a specific signal. Eg, time, altitude, GPS coordinants, radio signal.
Data collection	Gathering and logging data retrieved from electronic sensors for later retrieval and analysis.
EEPROM	Electrically Erasable Programmable Read-Only Memory and is a type of non-volatile memory used in computers and other electronic devices to store relatively small amounts of data.
Flight termination	Ending of a flight, either naturally through a balloon bursting at high altitude, or through the mechanical means of a cut-down device.
Geiger counter	An instrument used for measuring ionizing radiation.
Global Positioning System	A global navigation satellite system that provides geolocation and time information to a GPS receiver in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.
GPS	Global Positioning System.

Gyroscope	A sensor that measures degree of rotation of a body due to the gyroscopic effect of a spinning mass. Electronic versions are offered today that result in the same function.
HAB	High altitude balloon. A platform tethered to a balloon, designed to achieve a high altitude, allowing scientific experiments and data collection.
Header	A place on an electronic board where related signals are routed to allow a cable to connect, thus interconnecting seperate boards or devices.
IDE	Integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development.
Interrupt	An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing.
LEDs	Light Emitting Diodes. Small electronic components that emit light when receiving electric current.
Magnetometer	An instrument that measures magnetism, such as the strength and direction of the magnetic field at a point in space.
Microcontroller	A small computer; the lower end on the system-on-a-chip spectrum.
MOSFET transistor	The metal–oxide–semiconductor field-effect transistor (MOSFET, MOS-FET, or MOS FET) is a type of transistor used for amplifying or switching electronic signals. The main advantage of a MOSFET over a regular transistor is that it requires very little current to turn on (less than 1mA), while delivering a much higher current to a load (10 to 50A or more.)
Motherboard	The main board in charge of a system. Connects to smaller and more specialized daughter-boards, headers, sensors, etc.
Near-space	Tthe region of Earth's atmosphere that lies between 20 and 100 km (65,000 and 328,000 feet) above sea level, encompassing the stratosphere, mesosphere, and the lower thermosphere.
Nichrome resistance wire	A wire designed to heat up when an electric current is passed through it. Nichrome wire provides oxidation resistance which is important for wire reuse.
Payload	The packages attached to the balloon. Typically does not include items such as the parachute and cut-down unit.
Payload retrieval	Retrieving the payload after it has landed on the ground.

PCBs	Printed Circuit Boards.
PID controller	A proportional–integral–derivative controller (PID controller) is a control loop feedback mechanism (controller) commonly used in industrial control systems. A PID controller continuously calculates an error valu as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms, (sometimes denoted P, I, and D respectively) which give their name to the controller type.
Remote management	Allows communication from a ground station to the HAB controller via radio link. This allows remote control of certain settings and issuance of commands.
SDcard	A Secure Digital card is a small form factor memory device containing a microcontroller and flash memory.
Sensor	An electronic device designed to transform a physical property into an electronic representation to allow recording and analysis.
SOC	System-on-a-chip.
SPI bus	The Serial Peripheral Interface (SPI) bus is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The interface was developed by Motorola in the late eighties and has become a de facto standard. Typical applications include Secure Digital cards and liquid crystal displays.
System-on-a-chip	A single integrated circuit containing a processor core, memory, and programmable input/output peripherals.
Tracking	A HAB flight is typically tracked in real-time so that the payload may be recovered after it parachutes back to Earth. Tracking is typically done with radio and direction finding equipment or transmission of GPS coordinants.
Volatile variable	The volatile keyword indicates that a value may change between different accesses, even if it does not appear to be modified. This keyword prevents an optimizing compiler from optimizing away subsequent reads or writes and thus incorrectly reusing a stale value or omitting writes. Volatile values primarily arise in hardware access (memory-mapped I/O), where reading from or writing to memory is used to communicate with peripheral devices, and in threading, where a different thread may have modified a value.
Watchdog	An electronic timer that is used to detect and recover from computer malfunctions. A timeout is used to initiate corrective action, typically by resetting the system.