<script class="brush:arduino;first-line:5" type="syntaxhighlighter"><![CDATA[

]]></script>

**The Code (Background)**

**DISCLAIMER:** Any and all programming skills I have are entirely self-taught. I’ve slowly picked up better, cleaner, and more accepted conventions in my program structure, etc., but there are almost certainly many better approaches to the elements I’ve included in this project. If anyone has suggestions, criticisms, or input, I’d be more than happy to hear your thoughts, so I can make improvements in future iterations.

I’ll try to explain the code as simply as I can, without getting too wordy with all of the unimportant details. With me, that’s more easily said than done, but here goes nothing…

**Approach, Challenges & Lessons Learned:**

Within the entire body of code of Icarus, only a few programming languages are used (those that I’m comfortable with): Arduino, Python, and shell scripting. While limited in range of programming approaches, I tried to utilize the languages I know as efficiently as possible. On individual devices, things seemed to run smoothing, and it turned out to be the communication between devices that was the most difficult challenge. Icarus was all about bringing together everything that I’ve learned over the past few years, so it served its purpose. But the difficulties I encountered only serve to highlight the moral of the story that when designing an embedded system-based device with elements that really can’t fail, ***less is more***. I knew this in idea before, but working on Icarus really drove this idea home for me.

**Main Elements & Auxillary Helpers:**

Summarizing everything as simply as possible, the Arduinos run “Arduino” (processing/Wiring) [obviously], communicating with each other and the RPi by digital I/O and serial, respectively. The Raspberry Pi 3 uses crontab to coordinate automatic launch of the main media capture script written in Python, which in turn performs various actions globally by executing shell scripts.

There are 3 lines of communication between the devices:

1. Arduino Mega 🡪 RPi (Serial Communication)
   1. Triggers phase change to alter media capture parameters
2. Arduino Mega 🡪 Arduino Nano (Digital Output)
   1. Triggers deployment of servo for “selfie” photo at peak altitude
3. Arduino Mega 🡪 Arduino Uno (Digital Output)
   1. Heartbeat monitor serving as hardware watchdog

**Arduino Mega & The Core HAB Code**

There’s a lot going on throughout the main Arduino sketch, and I could really write a book on it, but I’ll try to just break down the main functional elements. If you would like to dig deeper into the code, you can always find everything on my GitHub, and I’m happy to answer any questions that anyone may have.

**Libraries:**

The libraries used will look familiar to those who have worked with Adafruit products, GPS, or any of the standard communication protocols. As always, Adafruit has come up clutch with the release of a new unified sensor library that encompasses many of its products and allows simple integration of multiple devices. For GPS, I went with TinyGPS++**(LINK)**. I’ve used Mikal Hart’s TinyGPS many times in the past with great success, and decided to upgrade to his new version in case I chose to utilize some of the cool new features. I chose to use the SdFat library**(LINK)** instead of the stock SD library as I usually do because of its reliability and high performance. The other libraries are pretty self-explanatory with the exception of “util/crc16.h” which is just used to calculate checksum for RTTY transmissions.

***LIBRARIES***

**Debug Switches:**

A few debug boolean switches are included for debugging purposes:

* debugMode 🡪 Turns on/off verbose serial output to console
* debugSmsOff 🡪 Turns off requirement for SMS ready command to be sent
* debugHeaterOff 🡪 Disables battery payload heater

***DEBUG SWITCHES***

**EEPROM Switches:**

These booleans are set from EEPROM and mark the exact flight phase that is active. These are included to allow the program to resume where it left off in case of accidental reboot.

***EEPROM SWITCHES***

**Setup [void setup()]:**

I’ve included all of the functions called during setup above the setup function itself to make things easier to find. I’ll walk through setup operations now.

Setup begins in the usual fashion, setting pin modes, writing digital pins to their proper states, and opening serial connections.

Following pin assignments, the first operation is to check the state of a small toggle switch inside the payload. If set to the start position, all EEPROM values are cleared to reset the program to the initial launch state. All of the accessory sensors are initialized as well as the SD card.

If setup is running for the first time, the GPS is then initialized first. This involves reading the PPS output to confirm that power is present at the device, then the program waits for PPS to indicate that a GPS fix has been attained. Next, GPS data is read until a sufficient horizontal dilution of precision (HDOP) is attained**(LINK)**. This ensures that the GPS data is consistently accurate before proceeding. I’ve found that a value of HDOP=125 provides sufficient accuracy. Once this is achieved, the launch site coordinates are set and stored in EEPROM to be used later for distance calculations.

The GPRS shield is then powered on through software by pulling the shield’s pin 9 high and then low again. I don’t use the pin to power off the device at any point, but it might be helpful for some to look at the code to see how this is done. I’ve had to troubleshoot this before and found that the duration of the pin 9 HIGH state is critical for proper power on/off function. For example, if the delay is 1000ms, this will turn on the device, but also turn it off if accidentally executed again. Making the delay 500ms turns on the device but isn’t sufficient to turn off the device if run a second time.

***GPRS POWER FUNCTION***

An SMS is then sent to the user asking for a reply of “Ready” to allow the program to proceed. This was implemented to ensure that the GPRS is powered on and functional and prevent a tragic loss of Icarus if the landing coordinates could never be sent.

Next, some altitude measurements are made from the Adafruit 10-DOF, averaged, and set as a baseline offset. This allows the ground at launch to be marked as 0ft, providing an altitude change from ground to be calculated that will trigger various flight phases.

A warm-up period for the MQ gas sensors then follows, which is required for proper operation. It’s also worth noting that these sensors require a burn-in time (5V applied to heating element pin) of at least 48 hours. I’m not sure of the exact details on why this is the case, but it’s standard practice and has something to do with the electrolytic element for gas detection.

After this, a read of the same toggle switch from the beginning of the program is made, and the program will wait for this to be toggled before proceeding. I usually go ahead and toggle it, seal the lid, and hook-up the gas senors via the terminal blocks after received the SMS ready request, so I don’t have to do anything at this point.

Finally, a start signal of “$0” is sent to the RPi, which begins the Python program/media acquisition, sending a “$0” back to the Mega when it starts running. This is the only time after waiting for the “Ready” SMS response that I included a mandatory hold in the main HAB sketch, because I wanted to make sure that the devices were communicating before launch and prevent it from hanging at any point after.

**Main Loop [void loop()]:**

All reads of sensor data are broken out into unique functions that are called within the main loop. These functions are return booleans, which are “true” when valid data is read and “false” when some error has occurred causing retrieval of invalid data. My approach to this was based on each sensor, some of which had obvious ways of validating data, while others needed to be defined directly (i.e. “if (value != 0.0)”).

To control timing, all of the sensor reads, state change checks, and output functions are nested within 2 for-loops with calls to major, non-time-sensitive functions after breaking from the for-loops and before beginning the main “loop()” from the beginning again. SD data logging is handled within each loop, as variables would be prematurely overwritten if not. A quick digital pin HIGH “heartbeat” output to the Uno also executes at the end of every loop to prevent timeout/reset. Logging is also performed at the same times as the heartbeat, with a separate log file dedicated to each set of data (AHRS, secondary [“aux”] sensors, GPS, and debugging [errors]).

The inner-most loop reads data the most quickly, the second loop processes after completion of each rapid inner-loop read, and the remaining checks/functions are called after the completion of both. The only included delay (some extra unused delays can be seen in the “#define” constants at the top of the sketch) is within the inner loop and determines the total loop time by default, as it determines how quickly each for-loop is broken on the way towards the outside.

Inner For-Loop (Runs 10x with a 500ms delay):

* Adafruit 10-DOF [I2C]
  + Accelerometer Data [Raw]
  + Gyroscope Data [Raw]
  + Magnetometer Data [Raw]
  + Attitude and Heading Reference System Data (AHRS) [Calculated]
    - Represents roll, pitch, and heading of an airborne vehicle**(LINK)**

Outer For-Loop (Runs 3x after completion of each set of 10 inner for-loop executions):

* MS5607 [I2C]
  + Pressure
  + Temperature
* Gas Sensors [Analog] (If payload is still rising in altitude)
* DHT22 [One-wire]
  + Temperature
  + Humidity
* DS18B20 [One-wire]
  + Temperature
* Radiometrix NTX2B-FA [Digital (Timing-dependent modulation)]
  + RTTY data transmission

End of Main Loop (Runs once at the end after all for-loops complete):

* GPS [Serial]
  + X
  + Y
  + Z
* GPRS [Serial]
  + Check for new incoming SMS messages
* Check for changes in altitude, etc. [void checkChange()]
  + Explained in next section…