**Background**

Icarus ONE is truly the culmination of all the learning and work I’ve done with microcontrollers, circuit design, and programming over the last 4 years. A year and a half ago, I started my first high-altitude balloon project. I basically completed all that I had intended to accomplish, both hardware and software, but I wasn’t able to launch with the move of my Ph.D. laboratory from Houston, TX to Bloomington, IN at that time. Now, I couldn’t be happier that my first prototype now sits in a storage unit, as this second attempt revealed too many coding blunders and potential hardware problems to count, but hey…I’m self-taught, and that’s part of learning!

**Rationale**

Now, I’m far more prepared to take on this project – probably far too ambitious as well. I don’t in any way recommend taking the “go big or go home” approach I did in this project, but I do hope that people can learn from my mistakes in one or more aspects of their own projects that I’ve included here.

So, as my opus magnum, Icarus ONE includes as many elements of my knowledge as I could integrate (and/or justify including). I’ll try to lay it all out simply and concisely, so others may hop onto the learning curve I was able to navigate.

**Overview**

Any high-altitude balloon has at least two critical elements, GPS location and some form of communicating this to someone tracking it as it falls back to earth with (hopefully) some really awesome media captured in near-space! Icarus contains a UBLOX GPS from this purpose, and I chose to opt for redundancy with communication, including both a 70cm band transmitter for radio teletype (RTTY) and a GPRS shield for SMS messaging. By including SMS capabilities, I was also able to add features like the piezo buzzer for locating Icarus in high-grass or similar situations, which can be activated by sending a text to the device after landing.

Outside of the core functions, all of my other add-ons were optional, but I’m excited about looking at the data afterwards, maybe even gaining a better understanding of the atmosphere (I’m also a weather nerd). I’ve added external environmental sensors, such as temperature, humidity, gas, and atmospheric pressure as well as internal sensors such as the Adafruit 1604 11-DOF (barometer, gyroscope, accelerometer, and magnetometer) and a sensor for battery temperature monitoring.

Some of the devices also provide output. For example, a custom heater board provides a more stable environment for the lithium ion batteries used.

The real payoff for a high-altitude balloon project is with the pictures. I mean, why send something up to around 100,000ft if you can’t see what it looks like up there!? [I’ll post later about the custom rockets I’ve also been working on in parallel with the iterations of Icarus ONE.] Icarus is equipped with three cameras: an up-facing webcam for capturing the balloon and parachute during certain flight phases, a down-facing webcam for capturing the earth from straight overheard, and the PiCam V2 which faces outwards and hopefully will capture the curvature of the earth from the edge of space. As an added bonus, a servo motor has been setup to swing a “selfie” of my girlfriend and me in front of the PiCam near the peak altitude…I’ve always wanted a picture of us in space!

Finally, there were a few personal items sent up, both from me and from a couple others to whom I offered the opportunity to send something. From me, I chose something very personal. Sadly, my father passed away a year and a half ago. He was a flight attendant for forty years and used to fly remote control gliders back in the early eighties, so much of his life was spent in the sky. So, it seemed like a perfect fit to include some of his ashes as well, which will be released as high as I’m able to get them, where our world meets the infinite expanse of space.

So, here we go. Time to get down to the nitty-gritty of Icarus’ design, construction, and programming.

**Hardware (Enclosures & Power)**

**Payload boxes:**

Luckily, biological laboratories have more Styrofoam boxes than anyone knows what to do with, so I had my choice of all shapes, sizes, and wall thicknesses. While Icarus ONE started with a single payload container (the larger box sitting on top), it quickly became necessary to expand this, as all of the things I crammed into this project required a decent amount of power. To provide this, I needed to expand Icarus with a smaller Styrofoam box underneath the main one to hold the batteries necessary for operation. The two boxes are permanently attached to each other, and holes were ported between the two to allow power cords, sensor wires, and the RTTY antenna cable to pass through.

**Lower Container (Power & Environmental Control):**

All of the power sources can be found in the lower container.

1. Main, low-current power for the Raspberry Pi 3, Arduino Mega, and Arduino Uno is provided by two 10,000mAh portable charging packs (Anker brand from Amazon), which have USB ports for convenient powering of the devices.
2. The Arduino Nano (which drives a servo motor) and a few accessories are powered by two sets of 4x NCR18650B lithium ion batteries each in series and linked together with a custom parallel harness made out of parts I used to recharge my drone batteries. [Fun drone posts to come later!] This arrangement creates a “4S2P” lithium ion setup, with a nominal 14.8V and 6800mAh (rated) of power. Each of the lithium ion battery packs was wired with a regulator board that prevents over-charging, excessive current draw, and excessive battery drain.

Looking at the data sheet for 18650 batteries, the minimum temperature under discharge conditions was -20C. The payloads are well insulated, but it’s definitely possible for temperatures to drop below this at high altitudes, potentially even to -40C. My calculations showed more than sufficient battery capacity to handle all of the main functions with significant overhead, so I chose to include a heater from the battery payload bay to keep the lithium ions happy for the journey. To do this, I simply wired up two 10ohm (10W) resistors in series on a custom board mounted in the lower payload container. Also in the container is a DS18B20 temperature sensor, which monitors the battery box and relays the temperature back to the Arduino Mega via the one-wire protocol. When temperatures below 0C are detected, a relay is tripped, allowing 14.8V power from the lithium ions to pass through the resistors, heating them up nicely.

Next up, the guts and brains of Icarus ONE…

**Arduino Mega (The Sensor & Communication Brain):**

For the main data acquisition board, I chose to use an Arduino Mega. I’m sure there was a way to pull all of this off without using the Mega, but it’s just so convenient with four serial ports that eliminate the need for software serial and the multitude of analog inputs that were really useful given the six gas sensors used with two analog pins already used by the I2C devices.

(Question #1: Can I2C analog pins be used as analog input and I2C simultaneously? Never investigated, but I suspected not. Question #2: Is there any SPI, I2C, or other alternate protocol device that can multiplex analog inputs? For example, is there some sort of board that can accept many analog inputs and send them over a couple wires to a device [with limited analog inputs like the Uno] using an alternate protocol?)

The Mega serves at the main input for all of the sensors, internal and external, output and storage of data to an SD card, controls digital output to other devices, and controls media acquisition from the Raspberry Pi over a serial connection.

The SD shield I used was designed for the Arduino Uno, which has a different set of SPI pins than the Arduino Mega. To make everything work, I stacked the SD shield and the relay shield together, separate from the Mega. I then routed pins 4, 5, 6, and 7 (to trigger the four independent relays) to digital pins on the Mega, pin 9 to another digital pin to allow software power-up of the GPRS shield, and I routed the SPI pins to their proper targets as listed on the table below.

|  |  |  |
| --- | --- | --- |
| **SD Shield (Uno Compatible)** | **SPI Pin** | **Arduino Mega** |
| 10 | SS | 53 |
| 11 | MOSI | 51 |
| 12 | MISO | 50 |
| 13 | SCK | 52 |

In addition, I linked the serial jumper pin-outs on the GPRS directly to the Mega via one of its hardware serials.

Internal:

A few sensors are included internally within the main payload package. First, there’s the Adafruit 1604 11-DOF sensor breakout (**INCLUDE LINK**), and it’s fantastic. There are way better applications for this one, but it had the barometer built in, so I included it. On a single board, Adafruit was able to put an accelerometer, gyroscope, and magnetometer, which provides 6-axis data logging and will allow me to model the movement of the payload box during the trip once I (hopefully) recover it after landing. In addition, the BMP180 provides temperature and barometric pressure data that is accurate and sensitive enough to calculate altitude with +/-10cm resolution.

(Question: If a Styrofoam payload is sealed securely on the exterior, not allowing free passage of air, does any/enough air pass through the pores in the Styrofoam to allow proper operation of and altitude calculation by the BMP180? I haven’t found an answer, so I went ahead and ported a few tiny holes in the side of the box using 14Ga magnet wire.)

The other internal “sensor” is the Adafruit Ultimate GPS Breakout V3, which also has a spot to install a coin cell battery on the backside for quicker GPS lock after resets. Fortunately, the GPS breakout also has a U.FL connector where I was able to connect a U.FL to SMA adaptor. I hot-glued the SMA connector to the side of the box and can connect a 28dB active GPS antenna that is fixed to the lid of the box, optimizing reception. Since I used a Mega, I chose to use another hardware serial for communication.

External:

On the lid are six “MQ”-type gas sensors, with two custom boards accommodating three each. The boards allow the sensors to quickly plug in and be swapped out, and also have a 5V voltage regulator circuit onboard, so I can power the current-hogging gas sensors that require a heating element to function properly. I’ve had some experience with these in the past, so I bought a cheap assortment of them from Amazon (**INCLUDE LINK**) and settled on what seemed like the most useful six of the bunch.

MQ-type Sensors:

* MQ-2 🡪 Methane, Butane, Liquid Propane Gas & Smoke
* MQ-4 🡪 Methane & Compressed Natural Gas
* MQ-7 🡪 Carbon Monoxide
* MQ-8 🡪 Hydrogen
* MQ-9 🡪 Carbon Monoxide & Flammable Gasses
* MQ135 🡪 Benzene, Alcohol & Smoke

Also on the lid, under a protective covering made from a pill bottle, I included a DHT22 temperature/humidity sensor. Originally, I had an SHT11…but an unfortunate and rather dumb mistake led to me frying the little guy.

On the side of the main payload box is a second barometer, the MS5607, a functionally equivalent alternative to the BMP180 used on the internal Adafruit board. To protect from light and moisture, as directed by the datasheet, I built a protective hood out of Gorilla tape and hot glue. Really, the inclusion of this component was fairly arbitrary, but it will be interesting to see how it compares with the BMP180 under various harsh conditions on the outside, rather than the comfy interior of the box.

Other than the previously mentioned external antennae for the GPRS and GPS, the final component controlled by the Mega is the piezo buzzer. Actually, it’s a “siren,” which as far as I can tell is a piezo buzzer that oscillates in frequency to create a siren effect. It was the loudest I could find at Radio Shack, so that’s what I chose. The buzzer is activated by triggering a relay on the relay shield, which provides power directly from the lithium ion batteries.

**Raspberry Pi 3 (The Media Brain):**

The Raspberry Pi 3 onboard is responsible for capturing media from all cameras during the flight. This made sense, given the CSI port which accommodates the highly-capable PiCamera, now in version 2, that can capture some truly striking images and integrates seamlessly via a tiny ribbon cable. Also, the four available USB ports on the RPi, alongside the V4L drivers in Raspbian that make it play nicely with most webcams out-of-the-box, created the perfect media acquisition solution. Inputs to the Raspberry Pi 3 are listed below:

|  |  |  |
| --- | --- | --- |
| **Camera (Interface)** | **Capture Angle** | **Resolution (Still Images)** |
| Logitech C270 | Up-facing | 1280 x 720 |
| Logitech C615 | Down-facing | 1920 x 1080 |
| PiCam V2 | Out-facing | 3280 x 2464 |

At regular intervals during the flight, each camera captures media in sequence, with no two cameras capturing simultaneously to prevent excess current draw from the RPi USB ports, which would cause a reboot. The capture parameters for each device are dependent on the phase of flight, so the most important shots will be captured at any given time. Flight phases are triggered by the Arduino Mega based on altitude and/or pressure depending on the phase. The Mega then sends a signal to the RPi via serial over a USB connection. Flight phases and camera functions are listed below, and the functions described are repeated at interval until the next phase is achieved:

1. **Takeoff capture (0 – 5,000ft):** Up-facing photos, Out-facing photos, Down-facing video
2. **Ascent phase (5000 – 80,000ft):** Up-facing photos, Out-facing photos, Down-facing photos
3. **Peak capture (80,000ft –** **Balloon break/Descent):** Up-facing video, Out-facing photos, Down-facing photos
4. **Descent phase (Balloon break/Descent –** **5,000ft):** Up-facing photos, Out-facing photos, Down-facing photos
5. **Landing capture (5,000ft –** **Landing):** Up-facing video (Short), Out-facing photos, Down-facing video (Long)
6. **Landing phase (Until power-off):** Up-facing photos, Out-facing photos, Down-facing photos

I’ll go into more detail later when I write some posts on all of the code, but in brief, all of the camera functions are coordinated by a Python script which calls shell scripts to initiate media capture. This Python script is initiated via crontab on each boot, and the current flight phase is written to a text file each time a new phase is triggered. In this way, I was able to protect against desynchronization of the Arduino Mega and RPi during various phases of the flight in case the RPi was forced to reboot for whatever reason.

The last thing worth mentioning, which you’ll run into later when I explain the function of the Uno, is that I also implemented a hardware watchdog on the RPi, just in case some event causes the device to freeze – a condition that wouldn’t be recognized and corrected by any other method (to my knowledge).

**Arduino Nano (The Servo Controller):**

The Arduino Nano’s sole purpose on Icarus is to control a servo motor. Well, not just any servo motor…this one is strategically designed to extend a “selfie” photo of my girlfriend and me in front of the PiCamera on the end of a long arm. If all goes well, it should look like we took a picture in space! I tested the “selfie” at a few different distances, all of which seemed to capture with decent focus, both on photo and background. Needless to say, I’ve always been impressed with the PiCam’s capabilities given its cost, even more so with version 2. I settled on 12” from photo to PiCam and crafted a lightweight arm out of 14Ga magnet wire coated in a layer of 5 minute epoxy (Gorilla Glue brand) to keep it rigid and prevent it from bouncing around like a spring.

Given the possibility of a quick current spike when using a servo, I chose to power the Nano from the lithium ion batter bank. Because of the lack of common grounds between the Mega and Nano, I made the digital input pin an INPUT\_PULLUP and triggered the servo activation by having the Mega pull its output pin LOW.

***INSERT VIDEO HERE***

Deployment of the “selfie” is controlled by the Arduino Mega. The code on the Nano itself is simply designed to read a digital pin whose level is controlled by the Mega and move the servo from a predefined “retracted” value to “deployed,” which was determined through testing. Servo deployment is triggered from the Mega when the BMP180 (altimeter) detects a pressure below 28hPa (hectopascals). Adafruit’s devices have never failed me, so, while I did have a few options, I thought this to be the best and most reliable indicator of prime territory for a “selfie” in near space. 28.0hPa was determined from a chart found at AVS.org (<https://www.avs.org/AVS/files/c7/c7edaedb-95b2-438f-adfb-36de54f87b9e.pdf>).

Upon deployment, the photo remains in front of the camera, which will capture a handful of images, until a timeout period of 240 seconds expires. I thought that making servo retraction dependent on a fixed timeout interval would be smarter than referencing some other value, which might not trigger if an error occurred, blocking additional photos from the PiCam with the fun, but unnecessary “selfie” photo.

**Arduino Uno (The Heartbeat Monitor):**

Up until about a month ago, I was having a lot of timing issues when resetting the system to ready it for a new “launch” condition. This was specifically caused when EEPROM triggers indicated one flight phase while the Raspberry Pi thought something different, and I have a single tactile switch that manually controls EEPROM reset. Not only that, but the software power-up of the GPRS shield was also inconsistent, and without that, Icarus would be lost for sure. This sounds very confusing, which admittedly it is and that was a fault of my patchwork approach to this hardware element. I’ll explain my use of EEPROM later in the coding section, which I thought I was able to use quite cleverly and effectively.

Anyways, the Uno was included as a hardware watchdog to combat timing issues. In the main sketch running on the Arduino Mega, a “heartbeat” is triggered regularly, which is simply a digital pin pulling high momentarily. The Uno is monitors for this and will pull the Mega’s reset pin low if a heartbeat isn’t detected within a defined interval. As far as I can tell, there are no longer any hang-ups in the main program, but the Uno watchdog seems to only provide an additional level of safety, as it’s powered by a dedicated 10,000mAh portable USB charger of its own via the DC barrel jack and could run into perpetuity.

In the process of troubleshooting this, I also familiarized myself with the somewhat obscure and unknown software watchdog also available on the Arduino, since it is actually native to the backend AVR libraries/functions. I will demonstrate in a later post on the code how to utilize the AVR software watchdog. For good measure, I integrated the AVR watchdog on the Uno heartbeat monitor, with a heartbeat timeout of eight seconds.

Honestly, these watchdogs were really unnecessary, but I’m happy I had the chance to learn about watchdogs in a few different contexts. (A third example is implemented in Linux on the RPi!)

**Power Regulation & Distribution:**

I think managing power regulation and distribution was my main challenge in the process, but provided a fantastic lesson in the subject that I’d often neglected while appreciating the simplicity of my embedded devices. Not only did I have a handful of different devices to power, the power requirements also varied significantly. My struggles can be seen in the unused 5V voltage regulator board made from an LM317, proper resistors and a few filtering capacitors whose value I determined from a simple tutorial (http://www.learningaboutelectronics.com/Articles/How-to-connect-a-voltage-regulator-in-a-circuit). Since this regulator doesn’t have any attached load, it doesn’t draw current, so I chose to leave it where it is.

An almost identical custom board sits next to the 5V board, but the resistors used in it give a fixed output of 12V from the LM317. The voltage regulator boards are powered by the lithium ion batteries, and are used for high-current draw applications such as the gas sensors on the lid, the heater in the batter payload bay, and the piezo siren. Output from the regulator boards is fed directly to components or to the relay junction installed just underneath.

***LM317 CIRCUIT HERE!!!!!!!!***

* 12V Regulator
* 5V Regulator
* 5V Distribution
* Relay Junction

**Communication:**

* RTTY
  + Hardware hack
  + PITS
  + String library
  + UKHAS tutorial
  + 300 baud
  + Custom string
* GPRS/SMS Functions
  + Startup
  + Landing
  + Functions

Icarus ONE has two protocols for communication, with a a GPRS shield to allow for SMS reporting after landing and a 434MHz transmitter providing in-flight data via radio teletype (RTTY).

***PICTURE OF GPRS SHIELD***

The GPRS shield (by Seeed Studio) is stacked directly onto the Arduino Mega, however, stacking was mostly to save space, and the important pins on the GPRS were actually moved to prevent hookup with the Mega. For example, pin 9 on the GPRS was routed to Mega digital pin 22, and this pin is used specifically for software power-on of the device by pulling the pin high momentarily. Also, the GPRS is powered by the lithium ion batter packs, rather than the Mega, because current fluctuations were causing the GPRS to shut-off immediately after power-up for reasons still unknown to me. Providing external power solved this though. A jumper on the GPRS allows one to select software serial communication via pins 7/8 or hardware serial communication via pins 0/1. Instead, I removed the jumpers and wired GPRS serial output directly to one of the extra serial ports available on the Mega.

***PICTURE OF ANTENNA HOOKUP***

***PICTURE OF EXTERNAL GPRS ANTENNA SMA PORT***

***PICTURE OF EXTERNAL GPRS ANTENNA***

The GPRS shield included a U.FL connector, so to decrease the possibility of losing reception, I hooked up a U.FL to SMA adaptor to it and ported the connector to the exterior of the box. On the lid, I placed a GSM antenna, which hooks up to the external connector.

***PICTURE OF UNDERSIDE OF LID WITH GPS WIRE***

***PICTURE OF GPS HOOKUP LOCATION***

***PICTURE OF GPS ANTENNA***

While on the topic of external antennae, it’s worth mentioning that the Adafruit Ultimate GPS Breakout V3 also has a U.FL connector. I setup the GPS using this connector in the same way as I did for the GPRS, with a 28dB active GPS antenna on the lid and the SMA connector ported through to the payload interior. The antenna can be hooked up just before attaching the lid, using an SMA connector fixed to an inside corner.

***PICTURE OF RTTY BOARD***

***PICTURE OF RTTY ANTENNA***

The RTTY communication implemented is critical for tracking in-flight, and will allow me to actively chase the payload and be near to the landing site when it hits the ground. If this weren’t implemented, I would have to rely solely on SMS functions, which won't function at significant altitude.

At this point, I need to give credit and much appreciation to Dave Akerman (<http://www.daveakerman.com/>) for providing the inspiration for this project in its entirety. A few years ago, I purchased one of his “Pi In The Sky” boards (<http://www.pi-in-the-sky.com/>), which stack onto the Raspberry Pi. These boards have a UBLOX GPS onboard, a Radiometrix 434MHz transmitter for RTTY and slow-scan digital video (SSDV) transmission of low-res images in real time. The board has a number of other features and a software package to run with it. When I purchased the board, it was only in version 0.5, now at V2.5 with a couple other add-on boards also available.

***PICTURE OF PITS BOARD***

I never launched using his board, and unfortunately the GPS was damaged at some point during my move to Indiana or while working at some point. I had most of the original features implemented already, and I was about to give up on RTTY altogether (which would have been very dumb), but I gave hardware hacking a shot. So, I desoldered the transmitter from the Pi In The Sky board and transferred it to a custom board that I made, using resistors to create a voltage divider for the modulation pin according to the datasheet. I was convinced that this was a total long shot, but using a tutorial from the absolutely fantastic HAB resource UKHAS (<https://ukhas.org.uk/guides:linkingarduinotontx2>), I was able to get RTTY working at 300 baud from the Arduino Mega.

***Important side note:*** When using the Arduino RTTY code directly from the tutorial, I had no problem getting everything setup. However, through much trial and error, I found out that if you use the “String” library (capital “S”) in Arduino, it will break the code and cause the RTTY to transmit a few times then simply stop. By copying the format used in the RTTY code throughout my entire HAB sketch, I was able to prevent this problem entirely (ex. sprintf, dtostrf, strcat functions native to C).

Much of Dave Akerman's work was done in C and runs on the RPi, and unfortunately I never familiarized myself properly with the details of C programming despite it being quite similar in many ways to Arduino. Because of this, I did not incorporate many of the features that Dave was able to, such as SSDV and a convenient config.txt file for changing settings, but I really couldn't have done any of this without all of the work he did to lay the foundation.

In my implementation, a custom string with a decent amount of information (most importantly altitude and GPS coordinates) is transmitted at interval.

***PICTURE OF SDR***

***PICTURE OF MOBILE ANTENNA***

I won’t go into a lot of detail about receiving and decoding the RTTY, because I was able to follow the exact instructions in a tutorial on UKHAS (<https://ukhas.org.uk/guides:tracking_guide>) to get it working. One element detailed there that I omitted was the setup of a payload document and upload of received RTTY data to the APRS network. I’m definitely going to try and get this working before launch, because the creation of a payload document allows crowd-sourcing of location information on your payload. In other words, anyone actively listening for signals in the band can received your payload data, identify it by your payload document, and upload the information to the network. This is a huge benefit, because even if you are out of signal range, it is very likely that someone else will have received the data and made it available for you. I’ll post again with an update if I integrate this.

**User Input:**

* Sensor mounting
* LED indicators
* Activation switch

**Code**