To mention:

* Fswebcam
* Nohup
* File cleaner
* PiCam (Python)
* Launcher
* Python script to make sure things are running
  + <http://pastebin.com/Y1P0xKfi>
    - Anonymous ☹
* Watchdog
  + <http://blog.ricardoarturocabral.com/2013/01/auto-reboot-hung-raspberry-pi-using-on.html>
    - Referred from Adafruit
  + <http://www.raspberry-pi-geek.com/Archive/2015/09/Making-your-projects-more-reliable>
    - Maybe what I actually used
  + <https://www.raspberrypi.org/forums/viewtopic.php?f=29&t=147501>
    - Now I think this is what I really used in the end…

**DISCLAIMER:** As with the code before, it’s worth mentioning that I’m entirely self-taught, and I’m certain that there are many more elegant approaches to what I’ve done here. Everything works, and I can call that a victory, but I’d love to hear any suggestions or thoughts.

**Raspberry Pi 3 (Python Media Control Code)**

The main Python camera control script only performs a few functions, but is responsible for coordinating most of the actions that the Raspberry Pi performs on Icarus’ journey. There are 4 specific capture modes that are active over the course of the flight, each responsible for controlling photo and video capture from each of the 3 onboard cameras. The parameters for each phase are optimized (by educated guessing) to capture the highest quality media at each flight phase. For example, the down-facing webcam captures video down at the launch site as it ascends, shutting off at 5,000ft – hypothetically, it should look pretty cool watching the launch team get smaller and smaller as Icarus rises.

The libraries used were fairly standard, with a few exceptions. The Python “picamera” library allows easy media capture from the RPI camera, and was a straight-forward solution. Media capture using this library comes directly from the script, with no external handlers necessary. The webcams, however, required slightly more work to integrate. Luckily, the RPi recognizes most mainstream webcams natively, assuming you’re running Raspbian. Thanks to Dave Akerman (as with many other things), I use “fswebcam” as the still image capture solution for the webcams and was able to do so with minimal setup (<http://www.daveakerman.com/?p=592>).

The Python “serial” library is included to handle communication between the RPi and the Mega. This library is well documented and straight-forward to use. The main factor in making Python serial work is in choosing the correct device. I saw tutorials referencing different end-points, but mine happened to be “/dev/ttyACM0.” This can be found by opening a terminal and navigating to the “/dev” folder. When an Arduino is plugged into the RPi via USB, a new object appears in the folder, and this is what needs to be referenced.

When the main script is first run, and before entering the main/permanent loop, a text file in the root folder is referenced (“mode\_file.txt”) to determine the mode in which the program will begin (i.e. takeoff capture, main phase, peak capture, etc.). The functions for photo and video capture are separated into 2 shell scripts, with arguments being called with each to indicate the camera to use for capture and duration, if necessary.

Photo and video capture from the webcams is facilitated by calling shell scripts that invoke the necessary command line functions. These scripts run in the background, allowing the Python script to continue functioning without delay. Because the script continues to run while acquisition is in process, some tricks needed to be implemented to prevent calling a camera while busy or attempting simultaneous video recording.

When video capture is active, booleans (and associated delays) are in place to ensure that no 2 webcams capture video simultaneously nor attempt to capture a photo while currently capturing video, which can cause an over current-based reboot event or an error to be returned, respectively. Captured media is given a file name indicating camera used and time of capture. Media and logs are saved into an organized directory where they may be easily found later.

**Important Note:** When capturing photos from the PiCam or webcams, a short (~2 sec) delay is essential after camera activation to allow the auto-exposure features to properly adjust the image before capture. This is well documented with the PiCam and is visible in the python code between “camera.start\_preview()” and “camera.capture(filename).” For whatever reason, I didn’t come across the same instructions from tutorials of fswebcam use and had to figure this out myself. In fact, I thought that one of my webcams was broken when it kept producing black images. Adding a delay in the form of a command line argument when running fswebcam corrected this issue. Maybe it was just my webcams that were the problem (even though they were Logitech/Microsoft brand initially), but you should test for yourself with your own hardware.

***MAINPYTHONCODE***

**Raspberry Pi 3 (Shell Scripts)**

* Webcam\_photo
  + Fswebcam
* Webcam\_video
  + Avconv
* File\_cleaner

*Photo capture & Shell script arguments.*

In writing this code explanation, I realized that I could have combined the photo and video shell scripts into a single file, using a third command line argument to indicate which capture mode was desired. I may try to do that soon, but for now, the webcam photo and video capture scripts are separated.

When calling the scripts, arguments are passed to determine a few necessary parameters. With the photo capture script, a single argument is passed to choose the camera for media capture (i.e. Up-facing webcam or down-facing webcam). While putting all of this together, I learned a few tricks (probably obvious to the more experience programmers) to create switches for shell script functions. When calling a shell script, any arguments are given incremental variable names that can be used. For example, consider the following call to a shell script:

./test.sh foo bar

This command will execute the script “test.sh,” and the arguments within this script can be utilized simply with some built-in variables. The argument “foo” can be referenced as “$1” while “bar” is defined as “$2.” This is true for as many arguments as you want to pass. While I didn’t use it here, it’s worth mentioning that “$0” becomes the name of the script itself. There are a number of other built-in variables that can be useful, so check out the documentation I used if you’re interested (<https://www.tutorialspoint.com/unix/unix-special-variables.htm>).

The fswebcam command uses a few arguments, which I’ve defined below:

* “--no-banner” 🡪 Removes timestamp and other info from bottom of image, which is the default
* “-d /dev/video$1” 🡪 The “-d” is used to define the capture device. I have 2 webcams, and each is assigned a name (i.e. /dev/video1) based on the USB port used, assuming both are plugged in on boot.
* “-r 1920x1080” 🡪 This is the capture resolution. The default of 1920x1080 I used is the maximum resolution of the 2 cameras I used. I did this because when the lower resolution camera is called at this resolution, it defaults to the next highest resolution of which it is capable.
* “-S 30” 🡪 This defines the number of frames to skip before capturing the final image. The duration that this equates to is dependent on the FPS chosen, but it was suggested that at least 20 skipped frames is sufficient based on a Stack Exchange post (<http://raspberrypi.stackexchange.com/questions/29283/why-is-my-webcam-image-all-black>).
* “$FILE” 🡪 This is the output file name, which I created by combining a prefix to indicate camera used with a time stamp generated earlier in the shell script.

***WEBCAMPHOTOCAPTURECODE***

The webcam video capture script operates in a similar way to that used in photo capture with a few additional elements. As with photos, the first argument defines the capture device to be used. The second arguments defines the length of the video (in seconds) to be recorded. A fixed FPS of 15 is used and a little math on line #5 calculates the number of frames to capture to reach this duration. I considered a few options for video capture, but a nice Stack Exchange post and answer gave me the solution that I ended up using (<http://raspberrypi.stackexchange.com/questions/23953/webcam-capture-into-mp4-or-mov-ffmpeg-is-very-slow-at-this>).

To capture video, in line with the solution from the Stack Exchange post, I used the command line program “streamer.” While this can be used for other purposes, streamer is capable of outputting an avi composed of a rapid series of jpeg captures. The Stack Exchange post also outlines a method of converting this output to a more compatible video compression format using avconv, which I implemented originally. I eliminated this function to prevent problems from extra delays or processor overhead, but it can still be seen in the shell script commented out, and I’ll just convert the files after (hopefully) recovering the payload. The last feature worth mentioning is that streamer is invoked using “nohup” to allow it to run in the background for the duration of capture, which can be fairly lengthy with extended video capture. To prevent the main Python script from waiting for this to finish after invoking it via subprocess, nohup is utilized, outputting output and errors to log files. Booleans and delays are used within the main script to prevent video capture from being reinitialized while capture is still in progress.

***WEBCAMVIDEOCAPTURECODE***

All photos, videos, and logs are stored to dedicated folders that can be cleaned and reset for a new launch by running a Python script (“file\_cleaner.py”). This is more of a helper script than anything, and it’s useful for restoring everything to a fresh state when running the program many times during testing.

***FILECLEANERCODE***

**Raspberry Pi 3 (Crontab, Helper Scripts & Loose Ends)**

A couple other scripts bring everything together and provide a level of reliability of operation by monitoring the state of running scripts and executing them at necessary times. These include a general launcher that executes the main script on boot and one that ensures that the program is always running, both being invoked from crontab.

The launcher script simply navigates to the root directory of Icarus and executes the main script. It’s important that the main script, and any others executed without directly calling Python (i.e. “./hab\_camcontrol.py” as opposed to “python hab\_camcontrol.py”) be given execution permissions by using the command “chmod +x filename” before this is possible. Also, a header must be included so the system knows what program to use to open the file. This is included at the top of the Python files as “#!/usr/bin/env python” and is necessary for the purpose of direct execution from the command line.

***LAUNCHERCODE***

The second helper script serves as a kind of software watchdog. It is run once a minute, and checks that Python is present in the list of currently running processes. This should be the case, as the main script should always be running during the course of the journey, and it is the only Python program that should always be running. If the script finds that Python isn’t running, it executes the main program again to restore function. This was included in case of an accidental exit from the program caused by an error. You can read more about this from the source I used, http://pastebin.com/Y1P0xKfi. Unfortunately, there isn’t any author I can credit for the help.

***PYTHONWATCHDOGCODE***

I chose crontab as the method of calling the persistent programs, “launcher.sh” and “python\_watchdog.sh” given the simplicity of use. By using the “@reboot” prefix, crontabs are run at boot, and this is what I used with the main Python media capture script. The second crontab entry is the shell script designed to ensure that the main Python program is running at all times. By listing all wildcards for the crontab entry (“\* \* \* \* \*”), the script is scheduled to run once a minute.

***CRONTABCODE***

@reboot sh /home/pi/icarus\_one/launcher.sh >/home/pi/icarus\_one/logs/cronlog.txt 2>&1

\* \* \* \* \* sh /home/pi/icarus\_one/python\_watchdog.sh >/home/pi/icarus\_one/logs/pywatchdoglog.txt 2>&1

Last but not least, the final feature I incorporated into the Raspberry Pi portion of Icarus ONE was the software watchdog available natively on the RPi. This triggers the RPi to reboot if something causes the device to hang/freeze. It’s important, however, to make the timeout interval long enough to prevent excessive CPU use and general slow-down trigger reboot. The instructions for setup and use have changed slightly in recent months after a kernel update, which is information I found from the same source I used to setup the watchdog itself (<https://www.raspberrypi.org/forums/viewtopic.php?f=29&t=147501>). A few methods for watchdog activation are listed, but I used the first one which involves adding a line to “/boot/config.txt.”

One interesting caveat to using the RPi watchdog is encountered on shutdown. If the watchdog is enabled and you “sudo halt” to shutdown, the RPi will reboot after the watchdog timeout interval if you don’t also disconnect its power. It’s not something that will affect the function of Icarus, but it’s good to know in other cases where it’s used.

***????WATCHDOGSTUFF????***