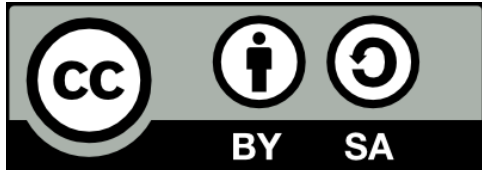
The Olympus Project

Building the Olympus Model Rocket with an Arduino Electronic Payload

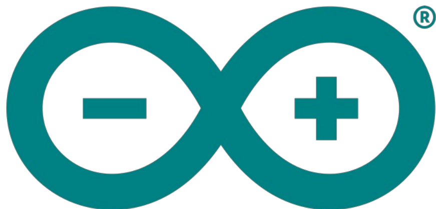
Robert W. Austin
NAR 47533

Copyrighted 2023



The Olympus Project © 2023
by Robert W. Austin is licensed under CC BY-SA 4.0.
To view a copy of this license, visit
http://creativecommons.org/licenses/by-sa/4.0/

This Project Manual was created using LibreOffice Writer Version 7.5. The LibreOffice logo was created by Christoph Noack - CC BY-SA 3.0, https://wiki.documentfoundation.org/File:LibreOffice-Initial-Artwork-Logo.svg, https://commons.wikimedia.org/w/index.php?curid=34369081



The Arduino code was created using the Arduino IDE

A Quick Note about the Products Mentioned

Any product that you see mentioned in this manual is listed because I bought it, use it, and found it did the job I asked of it. No person or company sent me anything. I do not receive any type of compensation if you buy anything I mention here.

The Austin Aerospace Educational Network

The Austin Aerospace Educational Network (AAEN) is a network of sites that is designed to provide you with the resources you need to perform a wide variety of projects using model rocketry. Model rocketry is a wonderful hobby, a great educational tool and just a world of fun. Model rocketry can provide a window that allows you to looking towards the future, be active in present day events and peer back into history and learn from those who have gone before us.

If you like working with technology, model rocketry and computers are made for each other. You can use software to design your own model rocket and make sure it is stable so it will fly straight and true. You can create simulations of how your model rocket will perform using different motor or fin configurations. You can conduct real research projects.

How about electronics? Thanks to small micro-controllers like the Arduino and ESP 32 as well as single board computers like the Raspberry Pi, anyone can add real electronics and avionics to their rocketry projects.

You can incorporate 3D printing to create parts that currently don't exist or to bring old designs long since removed from store shelves back to life. This offers the rocketeer a way to create components for their rockets and support systems that couldn't have been imagined even 10 years ago.

But perhaps the best news is that we have only begun to scratch the surface of the hobby, as there is so much more that you can do with model rocketry. Our intention is to provide you with a full array of information on the wide and wonderful world of model rocketry, what I consider the most fascinating hobby on the planet!

The Rocketry Research Journal

Our main site is the *Rocketry Research Journal*. This blog and web site can be found at https://rocketryjournal.wordpress.com. Below we list what you can expect to find on this site. There is no charge for any of the information or software you find on the site. Please feel free to download and share our reports, software, technical manuals, etc.

Here's What Is on the Site

The web site provides a portal to a number of the resources we have available. They include:

- The *Rocketry Research Journal* blog features articles on recent projects, news from the world of rocketry (both full size and miniature) and more. Check back frequently for the latest updates.
- View our Tech Reports. At the time of this report there a total of eleven reports available. They cover the basics of model rocketry, an introduction to doing research, single station altitude tracking, two station altitude tracking, how to adjust your electronic altimeter to account for temperature changes, how to use a spreadsheet to calculate altitude and tips on getting started using an Arduino micro-controller, plus a while lot more.

- We have a section that focuses on the Arduino micro-controller and how it can be used in model rocketry.
- We have a section set aside for 3D printing. Currently we have an article on using 3D printing to build a Dyna-Soar Titan II model rocket.
- There is a page for Model Plans. There are two plans currently available, but more are on the way.
- The Austin Aerospace Education Network (AAEN) has been developing the open source *Flight Logs Database Program*. The software can track your rockets from initial construction, then track all flights and record any maintenance needed or performed. It can calculate altitude, record any 3D prints used on the model, store the plans and even report CATOs to the MESS (Malfunctioning Engine Statistical Survey) site. If you are a NAR member and looking at completing your NARTREK submissions for the Bronze, Silver or Gold levels, it can help with that as well. There's even more the software can do for you. Read more about it on the Flight Logs Software page.
- Our other major software project is the *Rocketry Research Assistant* (RRA). This database is designed to assist you when using rocketry in research and engineering projects. The RRA is being developed using LibreOffice Base and the HSQL database engine. This means that the program will work on a number of computer systems including Windows, Mac, and Linux - including the Raspberry Pi. This also will make it easier for you, if desired, to modify the software to meet your specific needs. The software is in its early development phase, so look for a lot more updates and expansions in the future.

Our Sister Sites

We have a number of other sites that you can visit for specific rocketry projects or activities.

Source Forge Open Source Software Repository

- A Listing of All of Our Software
  https://sourceforge.net/u/austinaerospace

3D Printing File Repositories

- Thingiverse
  https://www.thingiverse.com/austin_aerospace_education/designs

CAD Files

- TinkerCAD
  https://www.tinkercad.com/users/kGt9Dmmc88b

Project Instruction/Tutorials

- Instructables
  https://www.instructables.com/member/Austin_Aerospace_Education

YouTube

- Rocketry Research Journal Video Channel
  https://www.youtube.com/@AustinAerospace

Other Project Manuals Available from AAEN

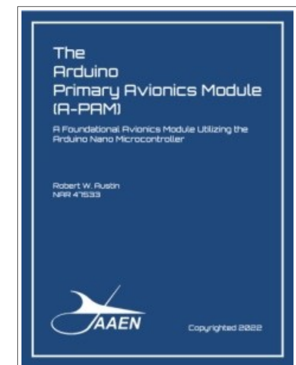
The Arduino Launch Control System

Our first manual is on our Arduino Launch Control System. This manual offers an introduction to the Arduino micro-controller and electronics. The manual takes you step-by-step through the process of design, breadboard, code creation and making the physical system. In addition to the instructional component the appendix contains drawings and schematics of the system, the Nano pin assignments, the complete source code listing and a listing of the parts needed to create the project.

164 pages. Adobe Acrobat (pdf) format.

Arduino-Primary Avionics Module

The A-PAM is designed as a foundational component of an overall avionics system. As such, the A-PAM by itself really doesn't do much. It needs to be connected to another payload module or sensor array to have data to collect. What the A-PAM does is supply power for your payload system, provide a micro-controller to conduct sensor readings and other task, utilizes a microSD card to record the data, and incorporates a RGB LED lamp that can be used to provide status messages in the field, when not connected to a computer. With the foundation provided for an electronic avionics system, the rest is left to your imagination.

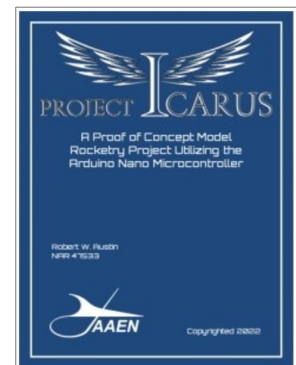


66 pages. Adobe Acrobat (pdf) format.

Project: Icarus

Project: Icarus is a proof of concept that brings several projects together into a single model rocketry research launch vehicle. This specific project includes the A-PAM avionics payload (listed above), temperature sensors along the body, and a video camera.

The sensors record the heat inside the body tube as the solid rocket motor burns and fires the ejection charge. They provide data on the temperature outside the motor mount, the temperature above the motor mount but below the flameproof recovery wadding and the area where the parachute is housed, just above the wadding.



112 pages. Adobe Acrobat (pdf) format.

Rocketry Research Assistant-Part 1

The *Rocketry Research Assistant* Database Project is designed to get you started in understanding, designing and creating databases. To make you aware of how databases work. To help you understand how they can be helpful in your research projects. To let you know that you can develop basic database programs and skills that can be immensely beneficial to you and your team.

With this release you will see the start of building the foundation for this database. It includes three basic forms (Projects, Team Members and Tasks) and the initial foundational tables. Our SourceForge download includes a copy of our Tech Report TR-11 "Introduction to Database Design" along with the accompanying Project Manual, "Creating the Rocketry Research Assistant-Part 1". The Project Manual takes you step-by-step through the design and creation of the database. The Project Manual has a number of screen shots and a detailed appendix. If you have an interest in database development this is a great introductory package.

79 pages. Adobe Acrobat (pdf) format.

Table of Contents

01 Introduction....12
The Olympus Project....13
The Rocket Kit....14
The Electronic Payload....14
The Rocketry Research Assistant....14
Our Goal....14
Additional Resources....15
Project Manual Layout....15
02 Designing the Project....16
The Rocketry Research Assistant....16
Getting Setup....16
Setting Up a Project....17
03 Olympus Construction....19
Construction....19
Motor Mount....19
Fins....19
Shock Cord Mount....20
Installing the Sub-assemblies....20
Installing the Motor Mount Assembly....21
Attaching the Fins....21
Installing the Shock Cord Mount....22
Installing the Motor Retaining Ring....23
Gluing the Launch Lug....23
The Payload Bay....24
Recovery System....24
Finishing....25
Prepping the Model....25
Filling Fins and Body Tube Spirals....25
Sanding the Fins and Body Tube....27
Prepping Plastic Parts....27
Primer....28

The Paint Scheme..29
04 Designing the Payload..33
The Arduino Primary Avionics Module..33
Arduino Uno..33
Arduino Nano...34
microSD Card Module...34
Battery..34
RGB LED..34
220 Ohm Resistors...34
The Payload Sensors..35
BMP180 Pressure Sensor...35
MPU-6050 IMU..35
Initial Design Planning...35
Fritzing Drawings..35
06 Start Coding...37
Documenting the Code...37
Layout..37
Title Block..38
Libraries..40
Declarations...40
07 Setup and Loop Code..43
The setup() Function..43
Initializing LED Pins..43
Initializing Serial Port – For Development & Testing Only..44
Calling Setup Functions..44
The loop() Function..45
08 Setup Functions...47
Splash Screen..47
The microSD Card Module..48
Setup_microSD_Card function...48
Setup microSD Card File Name..50
Pressure Sensor BMP180...51
SetupBMP180 Function..51
Baseline Pressure..52

Inertial Measurement Unit MPU6050....52
SetupMPU Function....53
Sensor Calibration....53
Setup Complete....54
09 Loop Functions....55
Pressure Sensor Readings....55
Checking the Sensor....56
Pressure Sensor Calculations....58
IMU Sensor Readings....59
Rotational Readings....59
Acceleration (G-force) Readings....60
Write Data to microSD Card....60
Header and Local Variables....60
Writing Data to the Card....62
The RGB_LED_Lamp tab....63
Header and Local Variables....63
Color Functions....63
Steady or Flashing....64
Using the Status Lamp in the Program....65
Code Review Conclusion....65
10 Building the Avionics Bay....66
Designing the A-PAM Housing....66
Tinkercad....66
Iterative Design....67
The Olympus Avionics Bay....68
A-PAM Housing....68
Sensor Module Housing....69
Payload Base....69
3D Prints....69
A-PAM Housing....69
Sensor Housing....70
Payload Base....70
The Completed Avionics Bay....71
Printing the Housings....71

Installing the Electronics....72
Wiring the Components....72
Final Assembly....76
11 Vent Holes and Temperatures....78
Are Vent Holes Needed?....78
Research Using Simulations....78
Research Under Flight Conditions....79
Calculating Vent Holes....80
Rules of Thumb....81
Calculations that are a Bit More Specific....81
Calculating Vent Holes for the Olympus Payload Bay....81
Locating the Vent Holes....83
Other Considerations....83
One More Thought on Temperature....84
Summer Example....84
Winter Example....84
12 Using the Olympus Electronic Payload....85
Preflight....85
At the Launch Pad....85
During Flight....86
Post Flight....86
13 Tactics to Improve the Olympus Project....87
Launch Vehicle....87
Electronics....87
Software....89
14 Conclusion....90
A1 Olympus Avionics System Drawings....92
A2 Pin Assignments....95
Nano Pin Assignments....95
A3 Complete Code Listing....96
Olympus_Avionics_V1.0.ino Tab....97
Calculation_BMP180.ino Tab....103
RGB_LED_Lamp.ino Tab....104
Sensor_BMP180.ino Tab....107

Sensor_MPU.ino Tab....110
Serial_Monitor_Splash_Screen.ino Tab....112
Setup_BMP180.ino Tab....113
Setup_MPU.ino Tab....115
Setup_microSD_Card.ino Tab....116
Write_Data_To_SD_Card.ino Tab....118
A4 Parts Listing....120
A5 The Scientific Method and the Engineering Process....121
The Scientific Method....121
The Engineering Process....123
A6 Engineering Notebook....125
Purpose....125
What to Track....125
Notebook Standards....126
Electronic Notebooks....127
A7 3D Printed Display Stand....128
A8 References....130
Arduino Programming....131

01 Introduction

If you do a search for model rocket electronic payload, you will see a lot of entries for medium to high power, large diameter rockets. Joe Barnard of BPS space has been doing some really fantastic stuff with electronics and rocketry – including thrust vector control and powered landings just like a Falcon 9 rocket. You can see videos of his accomplishments on his YouTube channel at https://www.youtube.com/c/BPSspace.

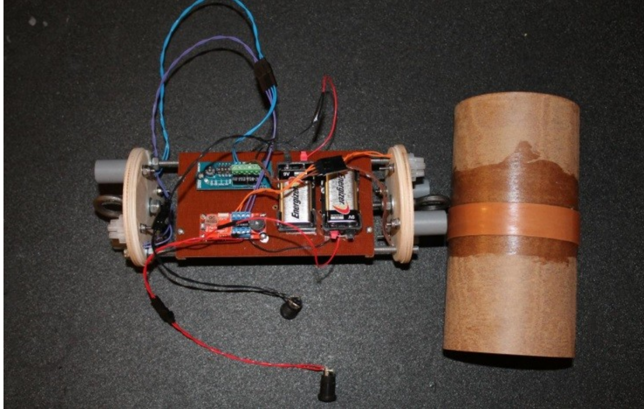
A high-powered rocket launch in the Black Rock Desert By Steve Jurvetson from Los Altos, USA - Launch of the Red Mongoose rocket to Mach 1.4, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=126487381

These types of rockets are large, typically 80mm (3.1 inches) in diameter or larger. You may see videos of these rockets where an Arduino Uno board is contained inside. These rockets are flown in large open areas, including deserts and dry lake beds. These larger rockets also use larger propellants, typically in the High Power Rocketry (HPR) range of "H" and above impulse level. Not only do these motors require you to be certified in HPR before you can buy them, the motors themselves tend to be rather expensive.

Now imagine the high school student interested in conducting a rocketry related electronics project. It can be very discouraging to watch these videos and realize that you simply can't do what they are doing because you don't have the large open space to launch the rocket, you need a HPR certificate, it's too expensive, or any number of barriers. I wanted to change that.

Another common factor among custom made electronic payloads is that often the payload is designed for a specific mission. This means that each time a new mission is developed a new specialty payload is needed.

I have been involved in rocketry since the mid 1970s, but electronics is completely new to me. However, microcontrollers manufactured by companies such as Arduino have made modern electronics readily accessible. After playing with the Arduino Uno, Mega2560 and the Nano, I realized that these can be used in much smaller rockets, using much smaller motors.



A typical high power altimeter-based deployment bay.
Public Domain, https://commons.wikimedia.org/w/index.php?curid=108282729

My goal was to develop a system that could be used by high school students. It would involve proven rocket designs that could be flown on school yards. That meant it needed to be small, much smaller than what you typically see. That meant fitting everything into a body tube with a diameter of about 42mm (1.65 inches), or about half the size you typically see. This would allow it to use "D" or "E" powered black powder motors. No special HPR certification is required to purchase or use these motors. No large dry lake bed required to fly these rockets.

I also wanted to create a system that could be reused, thereby reducing the need to constantly recreate new electronic payload designs. It would need to be flexible and upgradable. I wanted the design to be a *starting* point – one that students could take and modify to allow them to conduct real science. It also had to be inexpensive.

The Arduino Primary Avionics Module (A-PAM)

With all of this in mind I began designing the *Arduino Primary Avionics Module* (A-PAM). The module provides a microcontroller, a data recorder and a status lamp. It is housed in an avionics bay that is printed on a 3D printer. The CAD drawings for the original A-PAM are readily available on TinkerCAD (https://www.tinkercad.com/things/7pel9mfXqzs) This allows anyone to take the design and make it theirs. To see an example of how it can be used, look at our Instrucable on "Project: Icarus" (https://www.instructables.com/Project-Icarus-a-Temperature-Sensor-Model-Rocket/).

With the *Olympus Project* I will show you how we had to modify our original A-PAM housing to fit in the slightly larger payload bay. The CAD drawings are available for download and modification at (https://www.tinkercad.com/things/iY6COkklnYd). The construction of the A-PAM electronics here is the same as the original A-PAM as only the housing is changed.

The Olympus Project

This project is a multipart engineering project that is designed to introduce the concept of building a payload model rocket kit, creating a custom designed electronic payload, and tracking your progress using the *Rocketry Research Assistant* software. We want to show that anyone can create an interesting rocketry science/engineering project using common rocketry kits and electronic components that are readily available.

The Rocket Kit

The model rocket kit we are using is the Estes Olympus payload model. It is sold through Hobby Lobby stores or through their web site (https://www.hobbylobby.com/Crafts-Hobbies/Hobbies-Collecting/Rockets/Olympus-Flying-Model-Rocket-Kit/p/80971713). It retails for $16.99. The model is just over 29-inches tall and 1.75-inches in diameter. It can fly on a variety of motors including D12-3, E12-4, D12-5, Or E12-6.

The Electronic Payload

The electronic payload that we will be developing in this project is an altimeter with a motion sensor. We will create our payload using the *Arduino Primary Avionics Module* (A-PAM) and then adding the necessary components for the altimeter.

We will be using an Arduino Nano as the primary microcontroller. The Arduino Nano makes a great entry level board and it is readily available from a variety of sources. It has a large support network behind it including numerous web sites that offer advice, sample code, insight into the inner workings of the board and more.

The Arduino IDE is the programming software that we will use to develop our code. It is quick to learn and easy to use for the person that is just starting out doing microcontroller programming.

The Nano coupled with the Arduino IDE makes it an ideal platform for someone that is new to using microcontrollers and building electronic payloads. While there are other controllers that are more capable, chances are if you are aware of these other microcontrollers you are beyond the scope of this project manual anyway.

The Rocketry Research Assistant

At the time of this writing the *Rocketry Research Assistant* is only at Version 0.1.0. Despite this early release version we are going to incorporate it into this project. This will allow us to track the progress of the project as we work through it. It will also let us define our objectives and tasks that need to be performed as part of this project.

The other advantage of incorporating the *Rocketry Research Assistant* at this stage is that it allows us to test and update the software. We can see issues that may arise while using the software. It also helps provide us with ideas of what should be incorporated into the next version of the software, and if our development road map is working as it should.

Our Goal

The overall goal is to develop a working payload launch vehicle that can be incorporated into a science or engineering project. By incorporating the RRA we begin the Project Management process that is important to any research project.

This project should be thought of as just the beginning. While you will have a working payload model when finished, you should also be thinking about how the project can be improved and modified to meet specific research project needs.

Additional Resources

In addition to this project manual there are a number of resources that are available to help you along the process. This includes

- Videos of the construction process
  Visit our YouTube site for videos on building and finishing the Olympus, as well as construction the electronic payload
- Payload Avionics Mount
  The design of the avionics bay can be found on Tinkercad. The STL files are available through Thingiverse
- Arduino Software
  The software used to make the electronics come to life can be found on our SourceForge page
- Rocketry Research Assistant
  The latest version of the RRA can be found on our SourceForge

Project Manual Layout

While you can read the manual from cover to cover, you don't have to. By looking at the various chapters in the manual you can go directly to the area you are interested in.

02
Designing the Project

As with most research projects you start by sitting down and figuring out exactly what the project should do. What is the end goal or primary objective? What tasks need to be performed for the project to be completed?

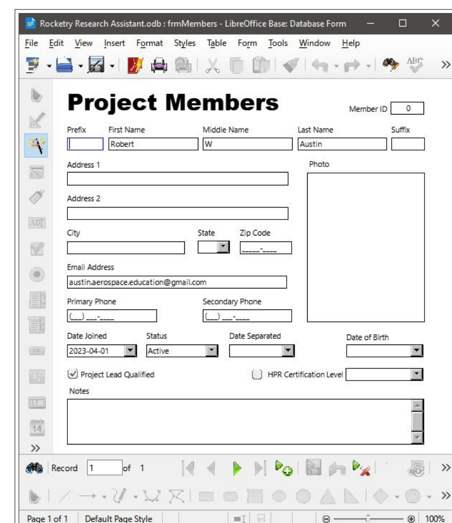
The *Olympus Project* is an engineering project, much like Project: Icarus. Where *Project: Icarus* was designed to test the temperatures inside a model rocket, the *Olympus Project* is being designed to test electronic payload attachments along with the *Arduino Primary Avionics Module* (A-PAM). In this case, we are going to be testing two specific sensors; a pressure sensor to detect altitude and an Inertial Measurement Unit to test things like g-forces and rocket spin rates.

There are several ways to track this. You can keep a written notebook and write down your ideas, the tasks that need to be accomplished, etc. You could also keep the same information on your computer. You might use a spreadsheet or document to keep track of things. Another option would be to use a Project Manager program. In *Appendix 5* you can find general guidelines on creating an engineering notebook. We have one other option and that is the use of the *Rocketry Research Assistant* (RRA). With this project, we will be using a combination of these options.

The Rocketry Research Assistant

We will start by using the RRA. At the time of this writing the first version (0.1) has been released and it is limited in scope. However, by using the software we can test it to see how well it works even at this early stage. It will help us find issues with the software, or areas where the software feels clunky. It will also provide us with ideas on how we can improve the software.

Because the RRA is limited, we will keep track of the project in two separate notebooks. One notebook will be used to track the construction of the rocket. A second notebook will be used to track the progress on the electronic payload. We will also be keeping a third notebook that is tracking the development of the RRA. Lessons learned from this project will be used to improve future projects and the RRA software.



Getting Setup

Before we can use the RRA to track our Olympus project, we need to do some preliminary setup. The first thing is to input at least one member in the Project Members form. This needs to be done so that an individual can be assigned as a Project Lead on a project, as well as being assigned to the various tasks that need to be accomplished.

In my Project Member's form I only have one name, as I am working on the project alone. This will likely be common for those working by themselves on projects. However, if you are working with a team on a project, all members of the team should be entered into the database. How much information you enter is up to you. As you can see in the screenshot above, only a few fields have been entered. Once the members have been entered, we can begin setting up our project.

Setting Up a Project

The next step is to set up our project in the Projects form. Start by opening up the Projects form in Base. Once it is open, we need to enter the name of our project, in this case "Project: Olympus".

Next is a drop-down box for the type of project. A number of options are available, including:

- Avionics
- Database
- Engineering
- Ground Support
- Launch Vehicle
- Report
- Research Payload

For this project we have selected "Engineering".

The next entry allows us to provide a simple overview or description of the project. For this project we entered, "Creating a payload model rocket that can test various electronic payloads." The hope is that the *Olympus Project* will be used as a test bed for various electronic sensors and, later, cameras. This is why we have labeled this an Engineering project. Its purpose is to test other systems.

The next series of entries deal with the progress of the project as well as important dates. This includes:

- Priority (1-4)
- Current status of the project
- The date the project is due to be completed
- The date the project started
- The date the project ended (this is not the same as due date. The end date may be the same as the due date, but may be sooner or later)

The final item in this section is the project lead. This is the person responsible for the management of the project.

> *Note: Future versions of the Rocketry Research Assistant will include the option to list all the members of the project team.*

The middle section of the form allows you to enter a number of objectives. Here you can enter a description of the objective and it's priority. For the *Olympus Project* we have included a number of objectives. Some of these include:

- Create an electronic payload that records acceleration
- Create an electronic payload that records altitude
- Create an electronic payload that records roll rate
- Create an electronic sensor payload package using the A-PAM

These types of objectives help us make decisions concerning the design and construction of the vehicle. By including the A-PAM device it lets us know the minimum diameter of the payload bay to accommodate the A-PAM. The objectives on acceleration, altitude and roll rate can help guide us in selecting the electronic sensors that we will need to use.

The bottom part of the form contains a Notes section. This can be used to capture any information about the project that is not collected elsewhere. You can see where we have entered the purchase of the Olympus payload model and are looking at a BMP180 to measure altitude and a MPU6050 to measure roll and acceleration.

This type of record keeping is important for any research or engineering project. It can help keep a project on track by clearly stating what are the objectives of the project and the date they have to be completed. It also identifies who has the overall responsibility for the project.

At this stage of the process the basic planning of the project is complete, the premise of the project is outlined, and the objectives of the project are written down. We can now move forward with the construction of the project.

03
Olympus Construction

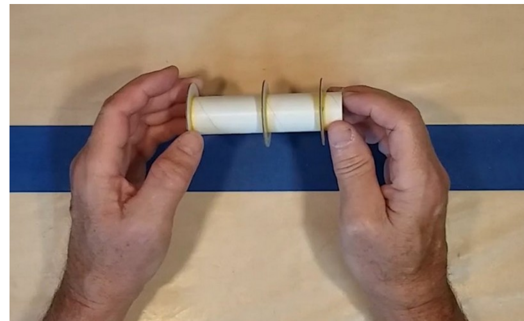
In this project we chose to use an “off the shelf” model rocket kit instead of building one from scratch. The rocket selected was the Estes Olympus. The model is 29 ¼-inches in length with a diameter of 1 ¾-inches. This makes it is slightly larger than the standard Estes BT-60 body tube. It features a clear payload bay that is 5-inches in length. The rocket is rated for the following motors:

- D12-3
- D12-5
- E12-4
- E12-6

Construction

The rocket was built using most of the materials supplied in the kit. This included retaining the balsa fins and clear payload section. One change made to the materials was the substitution of ¼-inch elastic band for the kit supplied rubber band shock cord.

Motor Mount



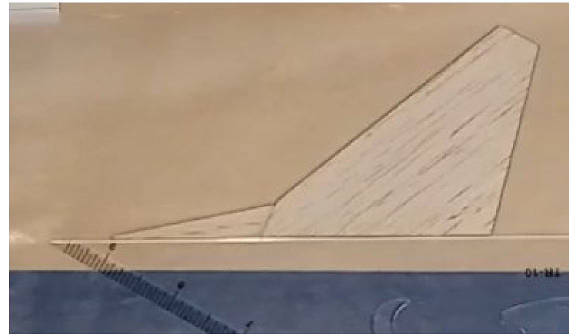
The motor mount is a typical Estes design consisting of three centering rings and a motor block. The centering rings are laser cut, but you still need to cut them away from their attachment points. Once you have them cut out, take some sandpaper (220 grit works well) and sand the outer ring and the inner ring smooth. Test fit the motor mount tube inside the centering rings. They should fit snug and not be either too tight or too loose.

Before you glue the rings to the motor tube, consider take a small bit of sandpaper and sanding off the shine of the motor tube where the rings will attach. This will help you get a better glue joint as you are gluing directly to the paper fibers and not the finish. Make sure that the rings are perpendicular to the motor tube when you glue them on. Once the centering rings are glued in place, give each ring a good set of fillets on both side of the ring. Don’t forget to add a fillet to the motor block at the top of the motor tube.

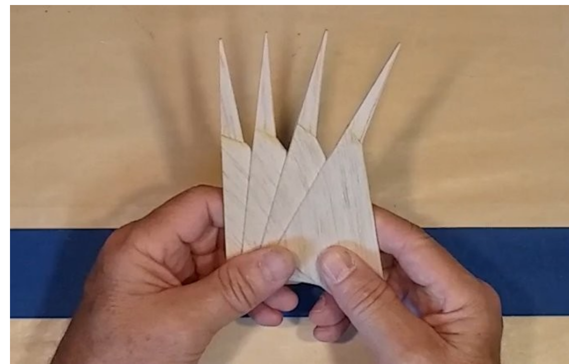
Fins

The fins are made from 3/32-inch balsa. They are laser cut and you will need to carefully cut the small attachment points to remove the fins from the balsa sheet. Once all eight pieces are removed from the sheet, stack like pieces together and sand each edge flat. This will help make all of the fins the same size. Once both sets of fin pieces have been sanded flat, we can glue them together.

Each fin consist of two pieces – a main fin and a dorsal fin. Lay your fins out on a flat surface that is covered in wax paper (this will stop the fin from sticking to the build surface). Apply a bit of glue to the dorsal fin, attach the dorsal fin to the main fin, and then remove it. Wait about 45-60 seconds before reattaching it. Lay both pieces down on the wax paper. Use a straight edge (such as a ruler or triangle) along the root edge to keep it straight. Let this assembly dry. Once the fin assemblies are dry, apply a thin layer of glue on each side of the joint to help strengthen it.



Once everything is dry, give each side of the fin a light sanding to smooth everything. Next we want to make sure that the root edge is straight. Sand this as necessary to get a nice straight edge. Finally, you want to sand the leading edges round. This will allow the air to flow around the fin smoothly, reducing the amount of drag experienced by the model. Take your time when sanding the leading edge of the dorsal fin, as it is easy to break it away from the main fin.



Shock Cord Mount

The shock cord mount used on the Olympus is the same style Estes has been using for years. The mount is printed on the instruction sheet so we recommend that you make a photocopy of the mount and use the copy instead.

The kit includes a ¼-inch wide rubber band to use as a shock cord. We would recommend that you replace the rubber band with 6mm wide fabric elastic strap. You can find the elastic straps in the sewing section of most big box stores or craft and fabric shops.

The other change we would recommend is to extend the length of the shock cord. A good rule of thumb is to make the shock cord twice as long as the rocket. The Olympus is just under 30-inches, so doubling that is 60-inches or 5-feet.

When you glue the shock cord into the mount, place the cord at an angle for the initial fold. Then angle the cord back to the center of the mount. This angle helps to lock the cord into the mount. When you have completed gluing the mount, add a slight curve to the mount. This will allow it to fit snugly against the inside of the body tube. Let that entire assembly dry.

Installing the Sub-assemblies

At this stage of construction we have finished putting together all of the sub-assemblies needed for the Olympus. Before moving on with installing these assemblies, make sure that that their joints are strong and that all of the glue is dry.

Installing the Motor Mount Assembly

Test fit the motor mount assembly before you glue it into position. Like the centering rings it should be snug but not tight. If it is too tight you will not be able to insert the mount fully inside the body tube before the glue grabs. Lightly sand the rings if the fit is binding.

You can use a small diameter dowel to lay in a bead of glue where the motor assembly is being mounted. Measure how far in the glue needs to go, mark your dowel, and then place the glue up to that mark.

When you insert the motor mount assembly, make sure you continue to slide the mount inside the body tube until both the bottom of the tube and the bottom of the mount are even. Do this in a single smooth motion. Don’t Stop! Once you stop the glue will grab and the mount will not go any farther in either direction. Some folks have indicated that they use epoxy for the motor mount as it is not only stronger than glue, but it doesn’t grab like glue does. This would provide you some play to move the tube back and forth as well as some extra working time. We haven’t tried this yet, but intend to in a future build. The choice is yours which method you use.

Set this aside to dry in an upright position. The glue that is pushed forward during the installation of the mount will act as a fillet at the top of the mount. If you lay the rocket down, this glue will move to one side of the body tube and the added reinforcement of the fillet will be lost.

With the motor mount in place you want to make sure the bottom of the mount is secure. Once the glue at the top of the mount has dried, apply glue around the bottom centering ring where the ring meets the tube. The motor tube is taking up a lot of space, you may not be able to use your finger to create a nice smooth fillet. Instead use a cotton swab to create the fillet. Let this dry with the rocket in an upright position, with the motor tube at the top.

Attaching the Fins

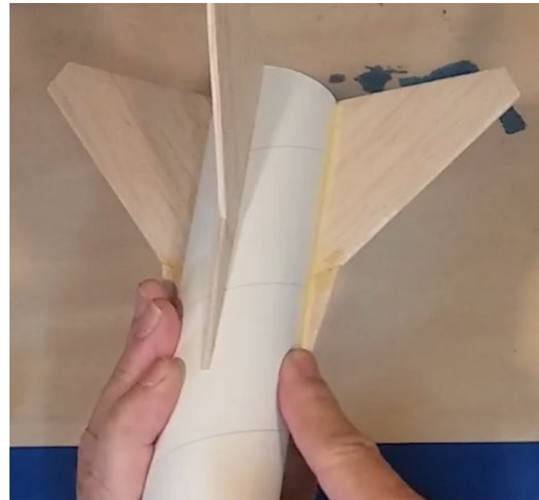
There are several methods of attaching the fins to the rocket. In our case we are going to use the Estes Fin Alignment Guide.

The original Fin Kwik alignment guide first appeared in the Estes 1975 catalog. This was an all metal adjustable guide. The plastic Fin Alignment Guide first appeared in the 1980 Estes Catalog at a price of $6.00. Today there are a number of fin alignment guides from a variety of sources. You can even 3D print your own guides if you want to. If you aren’t currently using an alignment guide of some type I would highly encourage you to get one. It

will reduce the amount of time you spend attaching your fins (you don't have to wait for each fin to dry before moving on to the next fin) and they will be in perfect alignment with the rocket.

When you glue your fins to the rocket, use the same procedure we used when attaching the centering rings. Sand a small area on the body tube where the fin will attach to remove the glassine finish and expose the paper fibers. Then use the same technique to glue the fins to the rocket that we used when constructing the fins. Apply glue to the fin, attach it to the rocket and then remove it. Wait 45-60 seconds and then reattach it.

Make sure that the rocket is positioned 90º to the base of the fin guide. If the body is leaning, your fins will not be properly aligned. Let the fins dry completely before removing them from the guide. Use care when you do remove the rocket, as you may have some areas where the glue is sticking to the guide.



After you have removed the rocket from the guide, take a look at the fins and make sure they are attached in the proper place (for the Olympus they should be even with the bottom of the body tube). If they are off, carefully remove the fin and then reattach it in the correct position.

With everything dry and aligned properly, you want to apply fillets to the fins. These glue fillets will strengthen the joint between the rocket body and the fin. It will also improve the aerodynamics of the fin section. To create the fillets, run a bead of glue along the joint between the fin and the body tube. Then take your finger and smooth the glue joint. Make sure the rocket is sitting level (so the glue doesn't run) and let this dry.

Installing the Shock Cord Mount

Our next step is to glue the shock cord mount into the body tube. You want to press the mount as far down the inside of the tube as you can. This will prevent it from interfering with the nose cone/payload adapter. Lay down a section of glue inside the body tube. Then insert the mount, pressing it down against the body tube (if you have curled it earlier, it should fit snugly against the tube). Once it is in place, coat the mount with glue. This helps cover the sharp corners of the mount that can snag a shroud line or the parachute itself.

Installing the Motor Retaining Ring

When installing the motor retaining ring, we will be using 5-minute epoxy to secure the ring in place. Before mixing your epoxy, take a piece of sandpaper and sand the glassine off of the end of the motor tube. Also sand the inside of the retainer ring. This will rough up the ring and give the epoxy something to grab in to.

With both the motor tube and ring prepped, mix up a small amount of epoxy (follow the manufacturer's recommendations on the mixture). It doesn't take much to secure the ring in place. When working with epoxy, make sure you use gloves to protect your hands. With the epoxy thoroughly mixed, apply a thin layer of epoxy on the outside of the motor tube. Now place the ring on the motor tube, pressing it all the way in until it stops against the motor mount centering ring. I also twist the retaining ring at least a quarter turn to help spread the epoxy evenly around the joint. Also, make sure that the epoxy does not get onto the threads of the retaining ring. This can prevent the motor retaining cover from fully tightening down and securing the motor.

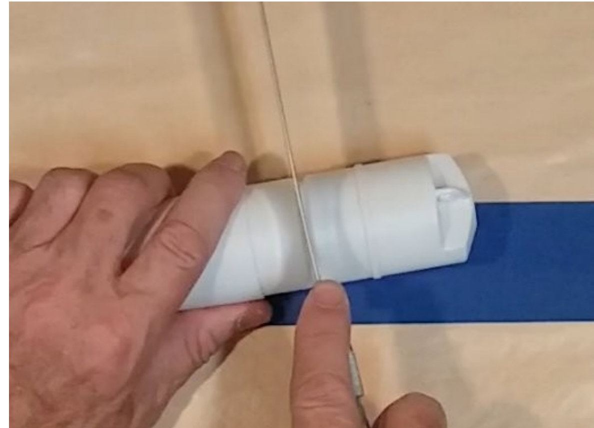
Gluing the Launch Lug

The launch lug is glued between the fins and parallel to the body tube. As we have done before, once you know where the launch lug will be attached, sand off the glassine on the body tube where it is attached and sand the launch lug where it will attach. Run a bead of glue along this area of the launch lug and attach it to the body tube. As before, remove it and wait 45-60 seconds, then reattach it.

You also want to apply a set of fillets on each side of the launch lug. We do this for the same reasons as we did it to the fins – aerodynamics and strength. The area between the launch lug and the body tube is much smaller than where the fins attached to the body tube so you probably can't get a finger in there to smooth everything down. Instead use a toothpick to add some glue to open areas and to smooth the fillet. You might also consider using a cotton swab here as well.

The Payload Bay

The next section to construct on our Olympus rocket is the payload bay. In the kit the nosecone and payload adapter are molded together. They need to be cut apart in order to be used. You can use a heavy modeling knife or a razor saw. We used the razor saw, constantly rotating the nose cone as we saw through the plastic. This resulted in a nice straight cut. With the two pieces now separate, any extra plastic can be cleaned off and sand around the cut edges to smooth the ends of both pieces.



On the payload adapter is the hook that will hold the parachute and shock cord. This may have some molded plastic inside the hook area. If it does, carefully cut this out using a sharp knife. You don't want to cut into the hook as it will weaken it and make it unusable. Once it is cut away, clean the inside hook area of any burs or sharp edges. With both pieces now separated, you can slide the clear plastic payload tube in place.

Recovery System

The final stage of construction involves attaching the recovery system to the payload adapter. Here we use snap swivels to attach both the parachute shroud lines and the shock cord to the payload adapter. Make sure you find the center of the shroud lines and pinch the lines at the point when inserting them into the snap swivel.

The same technique is used on the shock cord. Thread the cord through the eye of the snap swivel and tie it off. You might consider a drop or two of glue added to knot to prevent it from coming apart in flight.

Attach the open end of the each snap swivel to the hook on the payload adapter. You want to secure the snap swivel so it doesn't come apart in flight.

At this stage the construction of the model is complete. You could go and fly the model at this time if you wanted to. However, we are going to apply a nice finish to the rocket before it ever takes flight.

Finishing

How you finish a model is really an individual decision. Some modelers don't apply any finish at all and simply fly the model "as is" once construction is complete. Others will apply a coat of color to the model, going for the "5-foot finish" look (if the model looks good from 5-feet away, they're good). Others will spend a large amount of time prepping, painting and polishing their model. While I tend not to go to that extreme, I do spend some time trying to obtain a nice finish that looks good close up.

From a performance standpoint there are good reasons to apply a nice finish to the model. A model that is not painted will likely have increased drag from the irregular surface of the bare wood fins and the unfinished body tube. A model that has a smooth finish will have less drag, and so will obtain better performance, typically resulting in higher altitudes. However, the painted model will have additional weight from the filler, primer and color coats. That weight can reduce the performance of the model as there is now more weight the motor has to overcome. As with all things, there are pros and cons to the finish debate and you must decide which route you want to take.

In the following sections I'll describe the process I typically use on my models. It is the same process I used on my Olympus model and achieved pretty good results.

Prepping the Model

Before we do anything to the model we need to prep it. This is the process of going over the model and looking for any defects that can be corrected. If there is a gouge in the balsa fin we want to fill it and correct it before we start the finishing process.

Once all of the defects are corrected it is time to lightly sand the model. This should remove any bumps or fuzz on the model. These types of imperfections will be magnified once paint is applied. Run your fingers over the model feeling for any defects. Once you are satisfied, it is time to move on to the next step – filling the fins and body tube spirals

Filling Fins and Body Tube Spirals

At this stage we want to fill the balsa grain in the fins. This will provide a smooth surface for the primer and paint to stick to. By filling the grain in the wood we reduce the drag caused by the wood grain.

The same is true of the spirals that go along the entire length of the body tube. By filling these we create a smoother body tube and reduce the amount of drag. Both of these also result in a much nicer looking finish.

There are any number of methods that you can use to fill the fins and spirals. A simple Google search will bring up hundreds of articles dealing with the subject. The process I will describe here is the one I use and have had very good results. It involves the use of wood filler thinned down with water.

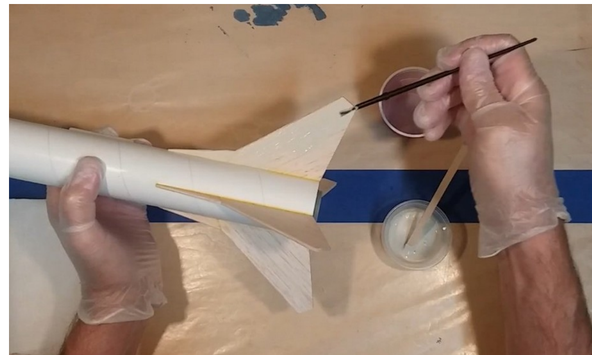
Creating the Filler

For the Olympus I created a filler using MinWax wood filler and water. You may find others around the web who use other types of wood filler. I started using the MinWax filler back during the COVID shutdowns as it was the only filler I could find. I was able to obtain good results with it so I kept using it.

Mix the filler in a small container with a little bit of water. It doesn't take much water to get the filler to the consistency of a thick paint. Make sure you stir the filler and water well as you want a nice smooth liquid without any lumps.

Filling the Fins

To start filling the fins simply brush the filler on the fins. If the filler doesn't go on smooth you may need to thin it a bit more with water. The smoother you can brush on the filler the less sanding you will need to do when the filler dries.



As you work on the fins you may find that the filler in your brush starts to get thick and harden. If this happens simply clean the brush off with a rag and some water. If the filler in the container starts to get too thick, just add a few drops of water and stir it in to get the filler back to the consistency you need.

Pay attention to the fins that you are working on. Make sure that you cover all of the bare wood with filler. This includes the leading and training edges as well as the fin tips. After the filler is dry if you find you missed a spot, simply go back and apply some filler to the area.

Filling the Spirals

If you look at the body tube you will see that there is a spiral gap that runs up the entire length of the tube. Like our fins this can be filled using the wood filler as well. You don't need a lot of filler here. I typically use a brush edge or a toothpick. Others may use an old knife blade to help get the filler into the seams. I find that if the filler is a bit thicker it fills the spirals better.

While you are filling the spirals on the body tube, don't forget to fill in the spirals on the

launch lug. It is made just like a body tube and has the same spirals. Filling in these spirals will help you get a nice smooth overall finish.

Sanding the Fins and Body Tube

Before you start sanding, make sure the filler is completely dry. Moving on to sanding the fins before the filler has dried will result in a gummed up piece of sandpaper.

I start with 220 grit sandpaper. This should help get the fins down to a fairly smooth finish. Don't forget to sand the leading and trailing edges. You want to sand down any filler that is on the body tube where the fin meets the body tube. Check your work and make sure that the fin is smooth and doesn't have bumps or ridges.

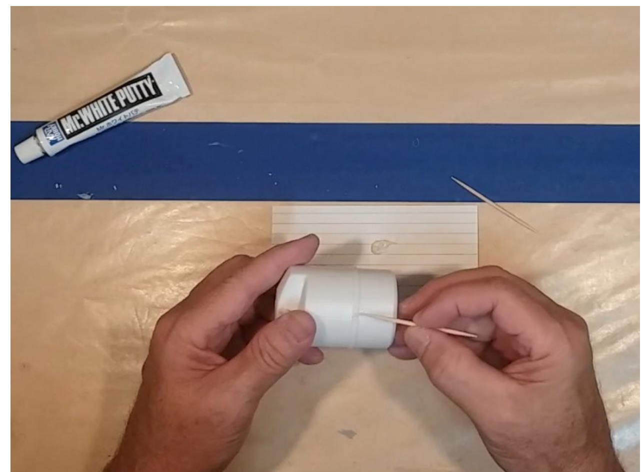
You want to sand down the body tube spirals the same way you did the fins. Take your time as you only need to sand down the excess filler, not the entire body tube. Don't forget to sand down the launch lug as well. When you are done, everything should fill smooth to the touch.

Once you have everything feeling good and smooth with the 220 grit paper, go ahead and sand down the entire rocket with 320 grit paper. This will help get rid of the small gouges in the fins and body tube caused by the 220 grit sandpaper. It doesn't take a lot of sanding with the 320 grit paper to get a really smooth surface.

Prepping Plastic Parts

The Olympus uses a plastic nose cone and payload adapter. These need to be prepped a bit differently from the wood fins and paper body tube.

Payload Adapter



Our adapter has two small depressions on the outer ring, right along the mold line. To give us a nice smooth appearance we want to fill in these depressions. Here we will use White Putty. Use just a small amount of putty on a toothpick to fill the area. You will find that you don't have a lot of working time with the White Putty. Just a couple of minutes. Once the area is filled set it aside to dry. Don't try to use wood filler in this situation as it simply will not stick.

Once the putty is dry, sand down the ring with 220-grit sandpaper. The ring should appear smooth with no raised or depressed areas. Finish the sanding with 320 grit paper.

An important note about the white putty. Depending on the manufacturer and type of putty you purchase, you may find it has a strong odor. Make sure to use this in a well ventilated area. This is another substance where you want to make sure you are wearing gloves.

Nose Cone

Our nose cone showed the opposite issue from the payload adapter. Instead of small depressions our nose cone had a raised edge right along the mold line. To get rid of this mold line I start by scrapping the raised edge with a used knife blade, not a new blade. Here we are scrapping, not cutting, so the blade that is slightly dull seems to work much better. Scrap the plastic ridge down until it feels even with the rest of the nose cone.

With the raised edge now even with the rest of the nose cone, it is time to start sanding. Start again with the 200-grit paper, sanding the entire nose cone (you don’t need to sand the shoulder). Look at the nose cone from various angles. The nose cone should appear the same no mater what angle you view it. Once the nose cone is looking good with the 220-grit sanding, switch to 320 grit paper to smooth out the nose and get rid of any gouges left in the plastic.

Primer

With all of our prep sanding completed, we can look to applying primer to the model. First we need to protect a couple areas from paint and primer before we start spraying the parts.

We are going to apply some blue painters tape to the top and bottom shoulder areas of payload adapter and to the shoulder area of the nose cone. Leave a slight gap between the edge of the tape and the end of the shoulder. This will allow some color to show through if either part is not fully seated in the body tube/payload tube. It just looks better.

The other area that we will need to mask off is the threaded motor retainer. You don’t want paint getting into this area and messing up the threads. That can result in a motor retention cover that never seats properly.

The primer I used on this model is the Rustoleum “2-in-1 Filler and Sandable Primer” that comes in gray. This is something you likely won’t find in the spray can aisle with all of the other colors and brands of spray paint. Instead, you will need to look in the automotive body care section. This is a primer used on cars to help fill in small scratches – the exact same thing we want to do as well.

Applying the Primer

With the Olympus I used a ½-inch diameter dowel with a motor spacing tube attached to it to hold the rocket. This dowel allows me to move the rocket around while applying primer or paint, as well as rotate the rocket into various positions. It is a great way to maneuver the rocket to ensure you cover the entire model.

I used small dowels with alligator clips to hod the nose cone and payload adapter. These are commonly found in stores that sell plastic model kits, tools and supplies. The dowels come with a stand, so after painting you can place the dowel in the stand to let it dry.

You want to make sure that the primer is completely dry before moving on to sanding. If it is not, it will gum up on the sandpaper. It will also cause small scratches or even gouges to be seen in the primer – something we are trying very hard to avoid.

Sanding the Primer

With the primer coat completely dry, it is time to start sanding. Unlike our previous sanding efforts that used dry sanding techniques, this time we are going to be wet sanding. We want to provide a very smooth surface for the color coat to lay on. The smoother the primer surface, the better the color surface will appear.

I used 320-grit sandpaper for this part of the process. Drip the paper in water and then sand as usual. You will both see and feel the difference in the finish as you wet sand. Using just the 320-grit paper, we can get a slight shine on our flat gray primer. This surface doesn't need to be like a glass polished surface, so one round of the 320-grit is all we did.

Because you are using water to assist with the sanding, make sure you wipe it off occasionally as you work. Our rockets are still made of paper and balsa and we don't want the water to seep into either of those materials. Light sanding and wiping down the excess water will provide you with that smooth surface you desire without messing up the materials under the primer.

The Paint Scheme

I frequent a YouTube channel called *Max's Models* (https://www.youtube.com/@maxsmodels). The channel motto is "Make it what *You* choose it to be" and that is exactly what I decided to do. The model is not painted as depicted on the kit's cover card. I was building this rocket as a test platform for an Arduino electronic payload. To me, this was like a smaller version of the sounding rockets that are used to test much larger and more powerful electronic payloads. So I decided I wanted this Olympus to look like a sounding rocket.

I decided to base the color scheme on the very distinctive Canadian Black Brant III. This color scheme consist of a red body tube with a white stripe along with a single white fin. The nose cone and payload bay is painted silver. I planned on making my Olympus look very similar.

Painting Order

As a general rule, when painting anything you want to start with the lightest color and work your way progressively to the darkest color.

> ***Note:*** *The general exceptions to this are the colors silver and gold. Here you often want a base color of black to help bring out the vibrancy in the gold and silver.*

As the rocket body was to be painted red and white, it meant starting with white and then overlaying the red. For the nose cone and payload adapter I decided to just spray the silver over the primer gray. I actually wanted a slightly dull, non-polished aluminum look, so I could forgo the black undercoat. The model would be painted using Rustoleum 2X rattle can paint.

Painting the Base Coats

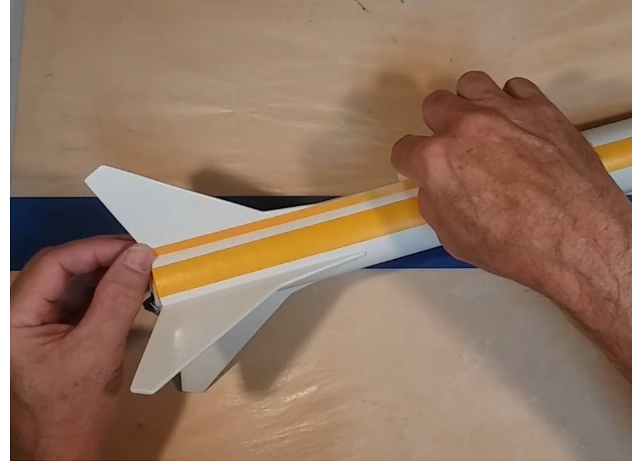
Using the same dowel to hold the rocket I began by laying down a thin coat of gloss white on the rocket. After letting this dry for about 15 minutes, I came back and hit it with a second coat. This gave the rocket an overall white appearance, but I wanted a slightly glossier look. I let the white paint dry overnight, and came back the next day applying a nice even coat over the entire rocket. This gave me the gloss look I was hoping for.

Next came the nose cone and payload adapter. Using the same short dowels and clips as when the primer was applied, they held the pieces while I painted on the silver. Here a light first coat was allowed to dry for 15 minutes just like I did with the white. Then a second coat was applied to give it a smooth even finish. Both parts were set aside to dry.

Before moving on to the masking and the red color coat I let the white cure for several days. The humidity is high where I live and this can slow down the cure time of the paint. This is a time for patience if you really want the model to come out looking nice. Start handling the model too soon and you will find your finger prints are permanently embedded in the finish.

Masking the Stripes

With the white base coat completely dry it was time to move on to masking the rocket for the red finish. I started off by using the ¾-inch wide yellow modeling tape available at Hobby Lobby for the stripe. This tape was wide enough to give me the size stripe I wanted, even leaving a little bit of red on each side. This was applied using nothing but my old eyeballs.



Next up was to mask off the one white fin. I wanted it to be one of the fins next to the white stripe. This time I used the ¼-inch wide yellow modeling tape to mask off the area where the fin meets the body tube. This narrow tape makes it easier to work the tape to the edge. Once the thin tape was in place, I next laid down a strip of the wide tape, slightly overlapping the narrow tape. The remainder of the fin was covered with blue painters tape.

The last thing to be masked off was a white ring at the top of the body tube. Here again I used the wide yellow tape even with the end of the body tube. I actually used two bands of the wide tape that are overlapped.

One thing you want to do is smooth down the edges of the tape. Take your fingernail (or other smooth object) and 'burnish' the tape edges. This will help seal the edge of the tape and prevent the red from seeping under the tape.

Painting the Upper Coats

Like we did before, we sprayed a light coat of red on the rocket and let it dry for about 15 minutes. Then a second coat was applied to smooth out the finish. Leave this to dry for several days just like we did with the white coat.

When the paint is dry it is time to remove the tape. When removing the tape that meets up against the color, I tend to pull the tape back against itself. Again, take your time so that the lines come out crisp.

Decals and Trim

The decals I used in this kit came from spares I had in my scrap decal box. Again, this isn't a scale model, so you can use whatever you want.

The other part of the trim is stick-on vinyl. This is the same vinyl that folks use in their Cricut machines. I used two strips of silver vinyl at the top and bottom on the clear payload bay. This

helps hide the shoulders of the nose cone and the payload adapter. I also added two vinyl black boxes at the top of the body tube.

The last change to the model included the 3D printing of the motor retaining ring in black plastic, to replace the kit's gray ring. It just looks better.

This finishes the Olympus kit build. In the image below you can see the finished Olympus rocket alongside the Black Brant that provided the inspiration for this model.

04
Designing the Payload

The initial step in developing the electronic payload is to determine what we need the payload to accomplish. This goes back to the original objectives we set down in Chapter 2. The objectives indicate that we want to track altitude, spin rate and G-forces on the rocket. This helps us determine what sensors will be needed to meet the objectives. In this case a pressure sensor and an IMU. Knowing the type of sensor needed, we can select the specific sensor based on criteria such as the ability of the sensor to do the job, the availability of the sensor and the cost involved.

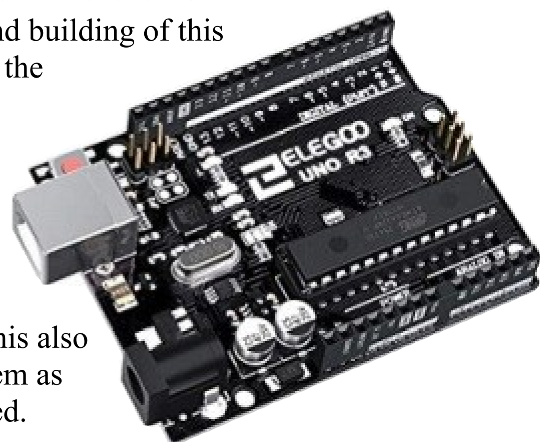
The foundation for the electronic payload on the *Olympus Project* is the *Arduino Primary Avionics Module* or A-PAM. It provides the foundational component of the overall avionics system. The A-PAM module provides power for the system, a microcontroller to conduct sensor readings and other tasks, a microSD card to record the data, and incorporates a RGB LED lamp that can be used to provide status messages in the field, when not connected to a computer.

The A-PAM is designed to be connected to a payload module/sensor array. For the *Olympus Project* that sensor module will consist of a pressure sensor that is used as an altimeter and an Inertial Measurement Unit (IMU) to record roll, pitch and yaw rates of the rocket as well as acceleration on all three axis.

The first set of components that will be describe are for the A-PAM. Afterwards we will list the components for the payload module sensors.

The Arduino Primary Avionics Module

I used two Arduino boards during the development and building of this project. This included clones of the Arduino Uno and the Arduino Nano.

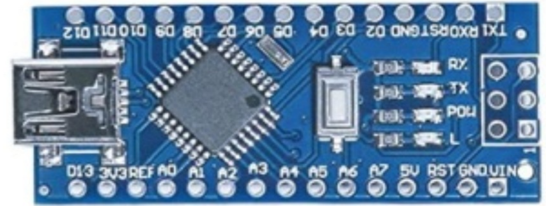


Arduino Uno

The Uno board was used during the initial development of the system. This board makes it easy to attach jumper wires between the Uno board, a breadboard, and the various electrical components. This also helps with rapid testing of various aspects of the system as well as moving connections and components as needed.

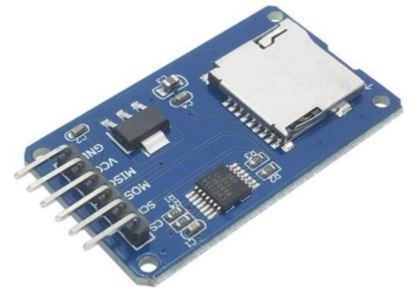
Arduino Nano

The Nano board is used during the actual construction of the A-PAM. The Nano has all of the same pins as the Uno (It actually has two additional pins that we don't use) and the code is interchangeable between the two devices (provided the two additional pins on the Nano are not referenced). The Nano board is considerably smaller than the Uno board.



microSD Card Module

The microSD Card Adapter Reader Module is used to record the data from our sensors to a microSD card. The module we use has a 6-pin SPI Interface (GND, VCC, MISO, MOSI, SCK, CS). The module that holds the microSD card is recessed slightly from the edge of the circuit board. This can cause a bit of a headache in inserting and retrieving the microSD card, but we have tried to make the task significantly easier with some design changes to the A-PAM housing.



Battery

I tried a number of power options for our system and in the end settled on a 7.4volt, 200mAh LiPO Battery. We found that this style of battery provided enough power to reliably power our system. The use of the JST Plug makes it easy to connect to the Nano board. The ability to use a USB Charger to recharge the battery is an added advantage. The battery is also small enough to fit inside the bay (37 x 19 x 11mm) and weighs 14 grams.

RGB LED

There is a need to be able to tell the status of the A-PAM and any attached payload without connecting it to a computer. When the rocket is on the pad, a quick and simple method of determining if everything is "GO" is essential. To meet this need we use a 5mm RGB Multicolor LED Lamp. The use of a programmable RGB lamp allows us to write code that can change the color of the bulb depending on the status of the avionics. We can also add flash patterns to provide additional information. This single bulb can provide a wealth of information when programmed properly.

220 Ohm Resistors

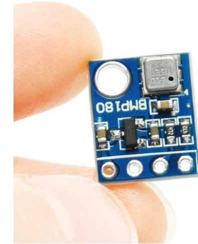
For the RGB LED to function properly, we would need to add three 220Ω resistors to the bulb. Each resistor attaches to a leg of the bulb (one each on the red, green and blue legs).

This completes the list of electronic components that make up the A-PAM. The next set of components are the sensors specific to this payload module.

The Payload Sensors

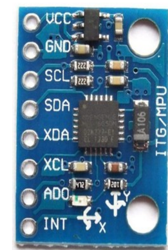
There are two sensors used in the payload bay. It consists of an altimeter and IMU.

BMP180 Pressure Sensor



BMP180 is a sensor designed to measure Barometric Pressure or Atmospheric pressure. The BMP180 senses that pressure and provides that information in digital output to our Nano. Temperature will affect the pressure and needs to be taken into account. This sensor uses I2C to communicate with the Nano. The BMP180 includes a temperature sensor as well. This sensor uses 3.3V instead of the typical 5V used with all of our other components. We must make sure that only 3.3V is supplied to the sensor, as 5V will ruin it.

MPU-6050 IMU



The MPU-6050 IMU (Inertial Measurement Unit) is a 3-axis accelerometer and 3-axis gyroscope sensor. The accelerometer measures the gravitational acceleration, and the gyroscope measures the rotational velocity. Additionally, this module also measures temperature. This sensor is ideal for determining the orientation of a moving object. Like the BMP180 this sensor also uses I2C to communicate. Unlike the BMP180 this sensor uses 5V logic.

Initial Design Planning

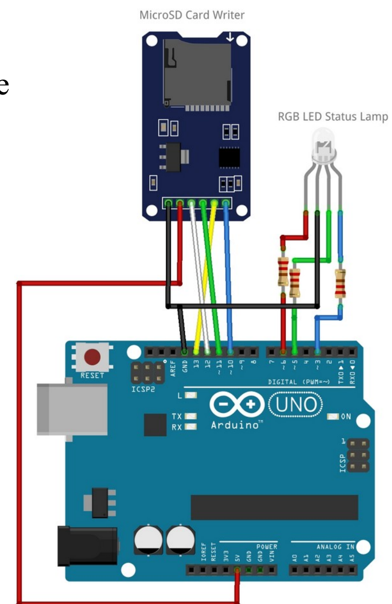
With all of the electronic components identified to be used in the Olympus payload, the next step is to design the complete avionics system. The A-PAM will provide the working base – power, data storage, sensor processing capabilities. The payload sensors need to be added into the system to complete the avionics package.

Fritzing Drawings

Before we start putting the pieces together, we draw out the connections using an open source program called Fritzing (https://fritzing.org). The software allows us to draw out the components and the connections. By drawing out the system first we can make sure that everything fits the available number of pins and that the components integrate with the Arduino.

The Original A-PAM

During development we use the Arduino Uno board and so our drawings reflect that decision. This is the same process that was followed when the A-PAM was originally developed. Because we have that as a foundation to work from, we can start developing our new payload by using the original A-PAM drawings and simply add the new sensors being used in this project.

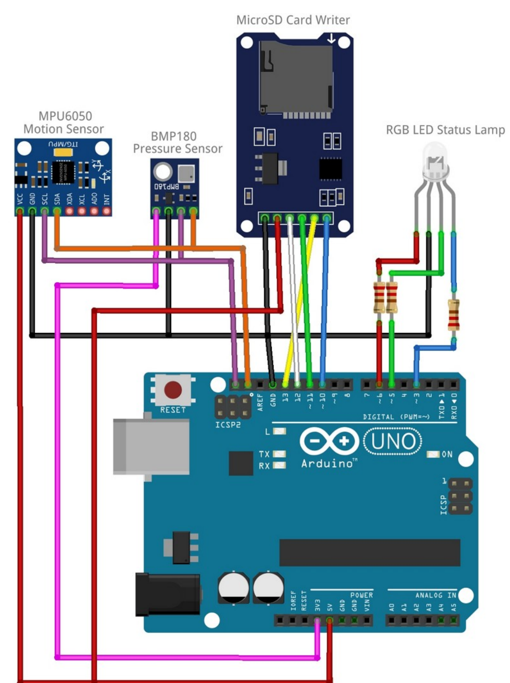


The drawing at the upper right depicts the A-PAM in its early design phase. The color lines each represent wires going to specific pins on the Uno board. You can see the RGB Status Lamp, the microSD Card module and the Uno board. We start with this baseline and begin to add the new sensors into the mix.

At this stage we are not looking at exactly how the components will be put together within the payload bay. This step in the process is to simply make sure the parts/sensors will work from a development viewpoint.

The Expanded Sensor Package

If you look at the graphic in the lower right, you can see the direct connection back to the A-PAM development drawing. The RGB status lamp and microSD card module along with the Arduino Uno are all present. The wiring is also consistent between the two drawings, further reinforcing the notion of the A-PAM as the foundation of the electronic payload.



What is new are the two sensors that are added to the drawing. The first new sensor is the MPU6050 IMU motion sensor, with the second sensor being the BMP180. The drawing shows a couple of interesting things about these sensors.

The first thing you may notice is that the BMP180 is using 3.3V while the MPU6050 is using 5V. The BMP180 cannot use 5V and will burn up if it is plugged into a 5V system. So we have to make sure that we keep it separate.

The second thing you may notice is that both the BMP180 and the MPU6050 are tied together on two lines. One line is marked "SCL" and the other is marked "SDA. These lines go to similar marked pins on the Uno. SDA and SCL are the ports used as part of the I2C (pronounced "eye-squared-sea") communications protocol. This is how these two sensors will "talk" to the Uno. This is different from the Serial Peripheral Interface (SPI) protocol used by the microSD card module to write the sensor data to a microSD card.

06
Start Coding

After putting the electrical components together, the next step is to begin writing the code to make everything work. I knew I could use most of the routines that I had written earlier as part of *Project: Icarus* and the *A-PAM Project*. I also knew I would need to create new routines for the IMU and pressure sensors as well as test routines to make sure the code worked properly. These test routines would make use of the Serial Monitor – something that you can't do during flight. Some test results would need to be converted into a signal that used the RGB LED routines.

Documenting the Code

I needed the code to "make sense" so that I could come back later (days, months, maybe years later) and understand what I wrote. This is important as most projects will change over time. There may be new code you wish to add. The code may become part of a different project later on. This is why "documenting the code" as you create it is so important. You just don't know how or when you may need to use it again.

A major reason for documenting your project is to record what the code is supposed to accomplish. This is a habit I learned long ago when I first started writing BASIC programs on the Commodore 64. It is important that you include enough notes in the code that it describes what the code is doing and why. If you come back a year from now and look at the code, the notes in the code should be adequate to tell you what you were doing at the time. If you are asking yourself "Should I add a note here?" go ahead and add it. I have never heard another programmer complain that the code they were reviewing had too many comments. The common complaint is that the previous programmer didn't add enough comments. This results in spending a lot of time trying to figure out what the previous programmer was trying to accomplish.

Layout

As you start to write your code you should develop a particular style. Even if you are just writing code for yourself, you will find that there are certain things that you will do for each project. Once you have a style you like, it can be useful to create a template. A template allows you to document each project in pretty much the same way each time.

> ***Note****: The Arduino IDE that you are using will dictate how you can use the template in your production environment. In the classic Arduino IDE (I am currently using Version 1.8), you can create your own template and save it.*
>
> ***Windows:*** *Open up the file manager and go to "C:\Program Files (x86)\Arduino\examples\01.Basics\BareMinimum". Rename file BareMinimum.ino. to something else (I use the name BareMinimum.org). Copy and paste your new template in this directory. Change the name to BareMinimum.ino. Now when you are in the classic Arduino IDE, you can click on File > New and your template will be displayed.*

Linux: (I'm using Q4OS - a Debian derivative - create your template and save it in the Home folder as BareMinimum.ino. Open up your file manager (I'm using Krusader) with root *access. Go to the "usr/share/arduino/examples/01.Basics/BareMinimum" folder. Rename the file BareMinimum.ino to something else (I use the name BareMinimum.org). Now copy your BareMinimum.ino from your Home folder to this directory. Now when you are in the classic Arduino IDE, you can click on File > New and your template will be displayed.*

With the release of the update Arduino IDE 2.0, this procedure no longer works. Their latest version (V2.2.1) as of this writing does not have a built in procedure for adding or using templates. As a work-around, you could create a Sketchbook that contains any number of templates. That sketchbook could then be accessed in the File > Sketchbook option. You have to remember to rename the file after you open it.

The remainder of this chapter will be a description of how I laid out my code in this project. Feel free to use anything here if you think it will help in your projects.

Title Block

I start my code with a title section. This provides the name of the project, a brief description, and the date of the last update. Since I release my code as open source projects I decided the license I will use is GPL-3.

```
/*********************************************************************
 *********************************************************************
 *
 * Project: Olympus Avionics Package
 * Version: 1.0
 * Description: This project is designed to use the A-PAM as the
 *              baseline avionics package. This includes an
 *              Arduino Nano, a microSD card module, and a RGB
 *              LED to show avionics status. The system is powered
 *              by a 7.4-volt LiPo battery.
 *
 *              The Olympus Project adds two sensors to the package
 *               - MPU6050 IMU Sensor
 *               - BMP180 Pressure Sensor
 *
 *              All code listed as Serial is for testing purposes.
 *              This code is commented out for use in the flight
 *              version of the code and is marked as "USE FOR
 *              GROUND TESTING ONLY”. Uncomment the code to use
 *              for testing.
 *
 * Created: 29 June 2023
 * Updated: 11 August 2023
 * Author: Robert W. Austin
 * (C) Austin Aerospace Education Network
 * License: GPL-3.0
 *
 * ==================================================================
```

The next section is a list of any coding examples I use or modify in the project. I also provide a link to the original code if it is on the internet. This does several things. First, it recognizes the programmers who came before me and generously donated their expertise so that I could learn from them. Secondly, if I run into an issue with a piece of code and can't figure out what I did wrong, it can often help to go back and look at the original code.

```
* ==================================================================
* Based on the following coding examples:
*  SdFat SD Card Code
*  https://github.com/greiman/SdFat
*
*  Altduino Altimeter
*  Code to auto-increment SD card files
*  https://www.hackster.io/M4K3R_4_M4R5/altduino-c0ac61
*
*  BMP180 Pressure Sensor
*  https://learn.sparkfun.com/tutorials/bmp180-barometric-pressure-sensor-
*
*  Royrobotiks Rocket Altimeter
*  https://github.com/royrobotiks/RocketAltimeter
*
*  MPU6050
*  https://www.jarzebski.pl/arduino/czujniki-i-sensory/3-osiowy-zyroskop-i-
*  https://github.com/jarzebski/Arduino-MPU6050
*  Examples -> MPU6050 -> MPU6050_gyro_simple
*  Examples -> MPU6050 -> MPU6050_accel_simple
*
* ==================================================================
```

The third section is a list of components and pin configurations. This identifies what is being used and where components are being plugged in. I also include the name of the board being used. This is important as different boards have different pin configurations.

```
* ==================================================================
* Pin configuration - for Nano board
*
* RGB LED lamp
*   Red   3 (D3)
*   Green 5 (D5)
*   Blue  6 (D6)
*
* microSD Card
*   CS   10 (D10)
*   SCK  13 (D13)
*   MOSI 11 (D11)
*   MISO 12 (D12)
*   VCC  5V
*   GND  GND
*
* BMP180
*   SDA  A4
*   SCL  A5
*   VCC  3.3V
*   GND  GND
```

```
 * MPU6050
 *   SDA  A4
 *   SCL  A5
 *   VCC  5V
 *   GND  GND
 *
 ****************************************************************/
```

Libraries

The next section of my code is dedicated to libraries. Libraries are pieces of code that will help make writing your code easier. There are literally thousands of libraries available for the Arduino. You can look at the Arduino Library Reference (https://www.arduino.cc/reference/en/libraries) for more in-depth information.

Any library I use is listed in this section. I also indicate what the library is being used for. Sometimes it is for a component (such as the microSD card module) and other times it is a library for a sensor (such as the MPU6050 library). Each library is separated by a series of lines. This lets me know exactly what libraries are used for specific components or functions.

```
/****************************************************************
 ****************************************************************
 *                                                              *
 *                         LIBRARIES                            *
 *                                                              *
 ****************************************************************
 ****************************************************************/

// ==============================================================
// library required for I2C communications
   #include <Wire.h>

// ==============================================================
// libraries required for the microSD Card Reader/Writer
   #include <SPI.h>
   #include <SdFat.h>  // replaced SD.h

// ==============================================================
// library required for the BMP180 Sensor
   #include <SFE_BMP180.h>

// ==============================================================
// library required for the MPU6050
   #include <MPU6050.h>
```

Declarations

This section contains all the global variables that are used in the program. As before, each declaration is segmented for a particular component or function. This can be a rather lengthy section of your code. The assigning of variables allows you to write code that can be easily modified later. The variables listed here are used in multiple areas of the program. These are different from local variables. Local variables are only used within a specific function. They cannot be referenced outside of that function.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                        DECLARATIONS                           *
 *                                                               *
 *****************************************************************
 *****************************************************************/

// ================================================================
// declarations for the microSD card
   SdFat SD;
   const int SD_FAT_TYPE = 3; // 3 = FAT16/FAT32 and exFAT file system
   const int pinSDcard = 10;  // pin number for CS on Nano board (D10)
   char fileName[] = "FLTLOG00.CSV";  //file name for flight data

// ================================================================
// declarations for time
   unsigned long timeStamp;

// ================================================================
// Define Pins for RGB LED lamp
   const int RED = 3;
   const int GREEN = 5;
   const int BLUE = 6;

// ================================================================
// declarations required for the BMP-180
// create an SFE_BMP180 object, identified as "baroPressure"
   SFE_BMP180 baroPressure;

// variables for sensor readings
// baroBaseline = baseline barometric pressure at launch site
// currentTemp =  local current temperature
// absolutePress = local barometric pressure
// altitudeCalc = calculated altitude
   double baroBaseline;
   double currentTemp;
   double absolutePress;
   double altitudeCalc;

// ================================================================
// declarations for MPU6050
// create a MPU6050 object
   MPU6050 mpu;

// Pitch, Roll and Yaw values
   double pitch;
   double roll;
   double yaw;
   double accelX;
   double accelY;
   double accelZ;
```

There are a couple of things I want you to notice in the microSD card section above.

- The first is the use of the SdFat library for writing to the microSD card. This library was used as it is not as memory intensive as the 'SD' library that was used in the original A-PAM code. This change was incorporated as memory issues began to appear.
- Notice the variable `char fileName[] = "FLTLOG00.CSV";` This is the file name that will be used to store data on the microSD card. FLTLOG00 is a generic name that we have supplied. You should change this to meet your needs. The two zeros (00) in the file name will be incremented for each new file. The "csv" file extension indicates it is a "coma separated variable" file. Keep in mind that it must use the "8.3" format of 8 letters, a dot, and a 3 letter extension. You need to keep the two zeros to allow the auto file naming routine to work. Also keep the "csv" extension to allow for easier importing of the data into a spreadsheet or database.
- The declaration for time contains the statement "unsigned long timeStamp;" This variable contains the time in milliseconds since the Nano has been powered up. This number will overflow (go back to zero), after approximately 50 days. Your rocket flights really shouldn't take that long.
- As we discussed earlier, there are a number of comments in this section that identify what the variable represents. We also tried to use variable names that make sense.

This completes the first part of the coding on the tab, "Olympus_Avionics_V1.0". The next chapter will look at the `void setup()` and `void loop()` code that make up the rest of the code in this tab.

Setup and Loop Code

In the previous chapter we discussed the start of writing the code for the Olympus Avionics Module. This included listing the libraries required, declaring variables and assigning pin definitions. Now we will look at the next section of code, `void setup()`.

The setup() Function

This section of code is included in every Arduino program. It is where we start setting up the Arduino so that it can begin to do things. This will include initializing pins on the board, turning on sensors, setting up modules, etc. The `setup()` section is run only once, at the start of each Arduino program.

Initializing LED Pins

The first part of our setup code is dedicated to the RGB LED lamp. You will notice that the line immediately after the void setup() statement is an opening curly bracket "{". Everything within the setup section will be contained between this opening curly bracket and a closing curly bracket "}" at the end of the section. You must include these curly brackets in your code.

This code is used to tell the Nano that the pinMode for each lamp color is an output. We will send power out through the pin to the LED. These pins are a bit different from the other pins on the Arduino. Most pins are either "on" or "off" but the pins used by the LED can very the amount of power being sent. As these pins can vary the power being sent out (or detected) they are called PWM or Pulse Width Modulation. This is what gives us the ability to vary the colors displayed by the LED lamp. On the Nano pins 3, 5, 6, 9, and 11 are capable of PWM. That is why in the declarations section we used pins 3, 5 and 6 for our lamp.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                            SETUP                              *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setup()
{
  // ==================================================================
  // Setup for RGB LED
     pinMode(RED, OUTPUT);
     pinMode(GREEN, OUTPUT);
     pinMode(BLUE, OUTPUT);
```

Initializing Serial Port – For Development & Testing Only

The next part of the setup routine we typically use to initialize the serial port. To use the Serial Monitor we first need to initialize it. We have chosen to set the rate to 115200 baud. This must match the baud rate on the serial monitor. If the Nano and the Serial Monitor are at two different rates, you will likely get nothing but garbage characters on your serial monitor screen or you may get nothing at all.

```
// ================================================================
  // Initialize the serial port.
     // Serial.begin(115200); // USE FOR GROUND TESTING ONLY
```

However, with this project we began to run into memory issues and had to cut back where we could. The serial monitor is used primarily for testing during the development process and tracking down bugs later on. It is not needed for flight. You can't hook up a USB cable to the avionics package and test it while the rocket is traveling through the air.

With your code functioning properly, you need to decide what you want to do with any code that you used during the testing process, including the code that posts data to the serial monitor. You can remove it and this may be necessary if program storage space is becoming an issue. The other option is to leave it in, but comment out these sections, making them invisible to the compiler.

For me, I tend to leave it in unless there is an issue with storage. Typically this code isn't going to hurt anything. By commenting out the code you don't have to be concerned about memory issues. This excludes it from the compiled version, but would still be there if you needed to do additional testing at a later time. The Olympus Avionics software includes a number of code sections that are commented out for just this reason. Where testing code has been commented out that was used as part of the development and testing process, you will see the comment "`USE FOR GROUND TESTING ONLY`".

Calling Setup Functions

It is with this part of the setup routine that we begin to do things a bit differently from a regular Arduino program. In most of the Arduino programs that you will find on the internet, all of the program is on a single screen. You follow the code from the top to the bottom. Yet in our code there is just a single line, `splashScreenSerialMonitor();` (it is currently commented out as I only use it for testing) but this command does something rather remarkable. That line calls another function within the program, on another tab.

Instead of everything being on a single screen in the Arduino IDE, we incorporate the use of multiple functions, with each function in its own separate tab. If you have downloaded any of our other Arduino software projects you will have seen this before. I do this to keep the code better organized and easier to use – especially when chasing down bugs. If you look at the rest of the Setup section, you will see where it contains commands to access various setup functions

```
  // ================================================================
  // Show Splash Screen
     // splashScreenSerialMonitor(); // USE FOR GROUND TESTING ONLY

  // ================================================================
  // Setup for microSD card
     setupmicroSDCard();

  // ================================================================
  // Setup for BMP180 Pressure Sensor
     setupBMP180();

  // ================================================================
  // Setup for MPU6050 IMU
     setupMPU();
```

As the program works through the Setup function, it will go to the `setupmicroSDCard();` function. Once it completes that function, it returns back to the Setup function and executes the next line. That line has the software jumping to the `setupBMP180();` function. The same thing happens – the code in that function is executed and then returns back to Setup. It then encounters the last instruction in the Setup function, a call to go to the `setupMPU();` function. As before, the code in that function is executed and then the program returns. With all of the code having been executed in the Setup function it then continues on the `loop()` function.

The loop() Function

The `loop()` function is where the Arduino software spends the bulk of its time. As indicated by the name of the function, "`loop()`" the software stays here, continuously running through the code listed in this function. When it reaches the last command in this function, it doesn't stop but returns back to the beginning of the loop function and will do it all over again. This type of layout works very well when you want to collect data from sensors on a regular basis.

The `loop()` function starts off with a header to easily identify this function. It then gets the current elapsed time in milliseconds and assigns that to the variable `timeStamp`

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                      MAIN PROGRAM LOOP                        *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void loop()
{
  timeStamp = millis();

  // get pressure readings
     readingsBMP180();
```

```
  // calculate altitude
     calculationsBMP180();

  // get roll, pitch yaw sensor readings
     readingsMPU();

  // write data to the microSD card
     writeDataSDCard();
}
```

The remainder of the loop() function is nothing but calls to four different functions;

- `readingsBMP180();`
  This function gets the temperature and pressure readings from the BMP180 sensor

- `calculationsBMP180();`
  This function takes the sensor readings from the previous function and converts them in standard measurements

- `readingsMPU();`
  This function gets the readings from the MPU6050 Inertial Measurement Unit and returns those readings in a standard measurement

- `writeDataSDCard();`
  This function takes all of the sensor readings and the time stamp and writes that data to a microSD card utilizing the Comma Separated Value (CSV) file format.

Once all of the four functions have been completed the program returns to the top of the loop and repeats this ad infinitum.

08
Setup Functions

In the last chapter we discussed how in the `setup()` function we simply called on other functions to be executed. In this chapter we will look at the four functions that are used in the setup routine.

Splash Screen

Note: *This code is only executed if the comment marks are removed.*
The purpose of this function is to display the name of the program and version number. This can be beneficial during development and testing, or when trying to trace down issues with the software.

If you look at the code in the Arduino IDE, there is a tab titled “Serial_Monitor_Splash_Screen.ino”. The code in that tab is below.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                SPLASH SCREEN ON SERIAL MONITOR                *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void splashScreenSerialMonitor(void)
{
  /*
  // ============================================================
  // USE FOR GROUND TESTING ONLY - Display Header on the Serial Monitor
     Serial.println(F("Austin Aerospace Educational Network Project"));
     Serial.println(F("Olympus Project Avionics"));
     Serial.print(F("Version: 1.0.0"));
     Serial.println(F("https://rocketryjournal.wordpress.com"));
     Serial.println();
     Serial.println(F("=================================================="));
     Serial.println(F("Serial Monitor Version for Ground Testing"));
     Serial.println(F("Comment Out Serial Monitor Coding for Use In "));
     Serial.println(F("Actual Flight and Data Collection"));
     Serial.println(F("=================================================="));
     Serial.println();
     Serial.println(F("=================================================="));
     Serial.println(F("Begin Initilization Process"));
     Serial.println();
  */
}
```

The Splash Screen is something that I use mainly during development/testing. It is displayed on the Serial Monitor and shows the version of the program that I am working on. Sometimes I would print out the screen for a current version and compare it to previous versions. This helps me keep the different versions separate but still easy to compare.

In reviewing the program we can see that at the very top is the comment header like we have seen earlier that helps identify the section. Directly after that is the statement `void splashScreenSerialMonitor(void)`. This is the name of the function and this is the function the program will jump to when it sees the line `splashScreenSerialMonitor();` in the `setup()` function. Once the code has been executed, it will return back to the `setup()` function and execute the next line in the code.

The microSD Card Module

The next function called from the `setup()` section is used to initialize and setup the microSD card module. Like the splash screen, there is a single line that calls the `setupMicroSDCard()` function. When the program carries out this command it will jump to that function and execute the code contained in that function.

Setup_microSD_Card function

As we have seen with our previous functions, the `Setup_microSD_Card` function is located in a separate tab. At the top of the function is a header that clearly identifies the function and contains the name of the function, "`setupMicroSDCard`". It then sets up a local variable that is used to collect the data that is written to the card.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                      SETUP MICRO SD CARD                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupMicroSDCard(void)
 {
  // ===============================================================
  // make a string for assembling the data to the log
     String dataString = "";
```

Next we have a section of code that is only used for ground testing. If needed, the comment marks in front of the Serial statement can be removed and it will send a message to the serial monitor that the program will now attempt to initialize the microSD card module.

```
  // USE FOR GROUND TESTING ONLY
  // Serial.println(F("Initializing Micro SD card. Standby..."));
```

To initialize the card module the software will perform a test to see if a microSD card has been inserted into the module or if that card is defective.

```
  // see if the card is present and can be initialized:
     if (!SD.begin(pinSDcard))
```

If the card does not initialize, the program will notify the user via a flashing red LED.

```
     {
       // card did not initialize
          ledRed();
          ledFlashingLamp();
```

If the program is being tested, removing the comment symbols (/* and */) will allow the error message to be displayed on the serial monitor

```
       /*
       // USE FOR GROUND TESTING ONLY
          Serial.println(F("Card initialization failed, or not present.
          Replace card and reset system."));
          Serial.println(F("==============================================="));
       */
```

With the error messages displayed (LED lamp and serial monitor) the software stops and does not go any further.

```
       // stop program at this point:
        while (1);
  }
```

The program stops at this point because the error needs to be corrected and the system reset. The reason for the shutdown of the software is that there is no alternative method of getting the flight data if the microSD card is not functioning. There is no radio telemetry to capture the data. If the purpose of the flight is to get the performance data, it needs the microSD card to be functioning properly. Otherwise, it is just another sport flight with a lot of expensive nose weight!

If the card does initialize, then that information needs to be displayed as well. In this case the lamp burns a steady green

```
// card initialized properly and is ready to start writing data
   ledGreen();
   ledSteadyLamp();
```

If the program is being tested, remove the comment symbols (/* and */) and the success message will be displayed on the serial monitor. Remember, when in the development stage or tracking down issues with the payload it is just as important to know what is working as well as what is not working.

```
/*
// USE FOR GROUND TESTING ONLY
   Serial.println(F("Micro SD card initialization successful."));
   Serial.println();
*/
```

This completes the card initialization process, but not the function. The second part of this function is designed to set up a file on the card and assign a unique file name for each specific flight.

Setup microSD Card File Name

We knew that the data from the payload sensors would need to be stored and a microSD card fit this requirement very well. But we also wanted to be able to store multiple flights on a single card. We did not want the program to simply overwrite a previous file. We found a nice little routine that does this in the Altduino project. The original code can be found at https://www.hackster.io/M4K3R_4_M4R5/altduino-c0ac61.

The routine starts by creating the filename where the last two characters are 00. The routine checks to see if there is a file on the microSD card that matches the file name. If it finds it the loop is repeated and increases the number by one. It checks again and continues this process until it finds a filename that is not on the card or until it reaches 99. For this project we assigned the variable `fileName` as "`FLTLOG00.CSV`" back in the Declarations section.

```
// ================================================================
// Setup Flight Data Log file name
// create new file name - routine from altduino.ino
// this loops through the numbers 00 to 99 to append to name
// of the CSV file. This prevents overwriting previous data
   for (uint8_t i = 0; i < 100; i++)
   {
    fileName[6] = i/10 + '0';
    fileName[7] = i%10 + '0';
    if (! SD.exists(fileName))
     {
      break; // leave the loop!
     }
   }
```

Once the file is created it will open the file and write the headers for the CSV data file. It then closes the file.

```
// write headers to CSV file
File dataFile = SD.open(fileName, FILE_WRITE);

// This writes column headers in the CSV file
dataString = "Time Stamp (milliseconds),Altitude (meters),Absolute Pressure
(hPa),Pitch-Y (rad/s),Roll-X (rad/s),Yaw-Z (rad/s),Acceleration X
(m/s^2),Acceleration Y (m/s^2),Acceleration Z (m/s^2),Temperature (C), ";
dataFile.println(dataString);
dataFile.close();
}
```

With the closing of the file it completes the `Setup_microSD_Card` module setup function. The program now returns back to the `setup()` function.

Pressure Sensor BMP180

The next function call in the `Setup()` function is for the BMP180 pressure sensor. The pressure sensor is used to determine the altitude of the rocket. The sensor obtains two data points – the barometric pressure and the temperature. From these two data points it is able to calculate the altitude of the rocket.

SetupBMP180 Function

It starts like all of our functions with a header identifying the function. The first command in the function is commented out. If the comment is removed for testing it will print that the program is going to attempt to initialize the sensor to the serial monitor.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                  SETUP BMP180 PRESSURE SENSOR                 *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupBMP180(void)
{
  // ================================================================
  // USE FOR GROUND TESTING ONLY
     // Serial.println(F("Initializing BMP180 pressure sensor. Standby..."));
```

The next step is to initialize the sensor. If the sensor initializes, the software turns the RGB LED lamp a steady green. It gets the current time and assigns it to the variable `timeStamp`. If testing, a success message is printed to the serial monitor with the time stamp.

```
  // Initialize the sensor
  // (it is important to get calibration values stored on the device).
     if (baroPressure.begin())
     {
      // Sensor initiated
      timeStamp = millis();

      /*
      // USE FOR GROUND TESTING ONLY
      Serial.print(F("BMP180 sensor initialization successful at "));
      Serial.println (timeStamp);
      */

      ledGreen();
      ledSteadyLamp();
     }
```

If the sensor doesn't initialize the RGB LED turns blue and begins flashing. If testing, a failure message is sent to the serial monitor with instructions to check sensor connections. The program stops working at this point

```
     else
     {
```

```
// Something went wrong, this is usually a connection problem
   ledBlue();
   ledFlashingLamp();

/*
// USE FOR GROUND TESTING ONLY
timeStamp = millis();
Serial.print(F("BMP180 sensor initialization failed at ")); // USE FOR
GROUND TESTING ONLY
Serial.println(timeStamp);
Serial.println(F("Check connections and reset system."));
*/
// Stop program at this point
   while(1);
```

Baseline Pressure

With the sensor initialized, the next step is to obtain a baseline pressure reading. It does this by calling the function `readingsBMP180()` and storing the data in the variable `baroBaseline`. If you are performing testing the baseline pressure is printed to the serial monitor.

```
 // Get the baseline pressure:
       baroBaseline = readingsBMP180();

  /*
  // USE FOR GROUND TESTING ONLY
     Serial.print(F("baseline pressure: "));
     Serial.print(baroBaseline);
     Serial.println(F(" mb"));
     Serial.println(F("==================================================="));
     Serial.println();
  */
}
```

This completes the setup of the pressure sensor. The program returns back to the `Setup()` function where it finds the next command that sends the program to the Inertial Measurement Unit setup function.

Inertial Measurement Unit MPU6050

The final sensor to be initialized in the setup function is the Inertial Measurement Unit. This sensor will be used to determine the g-force load in all three axis of the rocket. It is also used to get the rotation rate in all three axis as well.

SetupMPU Function

The function starts off with a header and then attempts to initialize the tri-Axis angular rate sensor (gyro) at a setting of ±2000dps as well as the tri-axis accelerometer at a setting of ±16g.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                  SETUP MPU6050 IMU SENSOR                     *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupMPU(void)
{
  // ================================================================
  // Initialize MPU6050
     while(!mpu.begin(MPU6050_SCALE_2000DPS, MPU6050_RANGE_16G))
```

If the sensor cannot be found it will display a fail message on the serial monitor if in testing mode, along with a suggestion to check wiring. It then turns the RGB LED to yellow and it starts flashing. The program stops at this point.

```
      {
        // USE FOR GROUND TESTING ONLY
        // Serial.println("Could not find a valid MPU6050 sensor, check
            wiring!");

        ledYellow();
        ledFlashingLamp();

        // Stop program at this point
        while(1);
      }
```

Sensor Calibration

If the sensor is located it proceeds through the calibration process. The software also sets the sensitivity of the sensor to its default value of 3. If testing, the serial monitor will display that the sensor is set up and the calibration is complete. There is no need to change the RGB LED at this stage as it will still be green from earlier successful sensor initializations.

```
  // Calibrate gyroscope. The calibration must be at rest.
     mpu.calibrateGyro();

  // Set threshold sensitivity. Default is 3.
     mpu.setThreshold(3);

  // USE FOR GROUND TESTING ONLY
  //   Serial.print(F("MPU Setup & Calibration Complete "));

}
```

This completes the setup of the IMU. At this point the program returns back to the Setup function.

Setup Complete

With the completion of the IMU setup routine, all of the individual functions in the main setup routine have been completed. The code in these functions are only performed once. With no more commands in the `setup()` function the software continues on the `loop()` function.

09
Loop Functions

The `loop()` function is where the Arduino software spends the bulk of its time. As indicated by the name of the function – `loop()` – the software stays here running through the code listed in this function. When it reaches the end, it doesn't stop but returns back to the beginning of the loop function to do it all over again. This type of layout works very well when you want to collect data from sensors on a regular basis.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                      MAIN PROGRAM LOOP                        *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void loop()
{
  timeStamp = millis();

  // get pressure readings
     readingsBMP180();

  // calculate altitude
     calculationsBMP180();

  // get roll, pitch yaw sensor readings
     readingsMPU();

  // write data to the microSD card
     writeDataSDCard();
}
```

Here we see that this function starts off as our others have. We include a header section to identify the function, the function name and then an opening curly bracket. The next thing the software does is get the current time stamp. This is the number of milliseconds that have passed since the Arduino was turned on.

The remainder of the loop is just calls to other functions. We will look at each function and what it performs.

Pressure Sensor Readings

The first function that is called in the `loop()` function is `readingsBMP180()`. This function is used to get the barometric pressure readings and the temperature. This information will be used to calculate the altitude of the rocket, however, that is not performed in this function.

Checking the Sensor

At the start of the function we find the usual header that identifies the function. It then sets up a local variable that will be used to record the current status of the sensor. The program then proceeds to get the status of the sensor by attempting to obtain a current temperature. If working, the sensor will then return a number. This number is the amount of milliseconds that the sensor must wait before getting the temperature.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *               PRESSURE SENSOR READINGS                        *
 *                                                               *
 *****************************************************************
 *****************************************************************/

double readingsBMP180(void)
{
// =================================================================
// local variable to report on status of sensor
   char bmp180Status;

// =================================================================
// Get a temperature measurement first before perform a pressure
// reading.

  // Start a temperature measurement:
  // If request is successful, the number of ms to wait is returned.
  // If request is unsuccessful, 0 is returned.

  bmp180Status = baroPressure.startTemperature();
  if (bmp180Status != 0)
```

If the number is 0 (zero) the sensor is not functioning properly. The program will skip this section (and subsequent sections). If the unit is in testing mode the serial monitor will display an error message.

If any number other than 0 (zero) is returned, the program will delay for that specific time period. After the wait period it retrieves the temperature measurement.

```
  {
    // Wait for the measurement to complete:

    delay(bmp180Status);

    // Retrieve the completed temperature measurement:

    bmp180Status = baroPressure.getTemperature(currentTemp);
```

If testing, it will display the result on the serial monitor.

```
    // USE FOR GROUND TESTING ONLY
    // Serial.print(F("Current Temp: "));
    // Serial.println(currentTemp);
```

```
    // Serial.print(F("Sensor Status: "));
    // Serial.println(bmp180Status);
```

Having retrieved the temperature, the next step is to obtain the current pressure. Once again we check the status of the sensor and if the number is 0 (zero) the sensor is not functioning properly. The program will skip this section (and subsequent sections).

If any number other than 0 (zero) is returned, the program will delay for that specific time period. We also set the sensor's resolution to the highest setting which is three. After the wait period it retrieves the pressure measurement.

```
// ================================================================
// Start a pressure measurement:
    if (bmp180Status != 0)

    {
      // The parameter is the oversampling setting, from 0 to 3 (highest res,
         longest wait).
      // If request is successful, the number of ms to wait is returned.
      // If request is unsuccessful, 0 is returned.

      bmp180Status = baroPressure.startPressure(3);
      if (bmp180Status != 0)
      {
        // Wait for the measurement to complete:
        delay(bmp180Status);

        // Retrieve the completed pressure measurement:
        // Note that the measurement is stored in the variable absolutePress.
        // Note also that the function requires the previous temperature
           measurement (currentTemp).
        // (If temperature is stable, you can do one temperature measurement
           for a number of pressure measurements.)
        // Function returns 1 if successful, 0 if failure.

        bmp180Status = baroPressure.getPressure(absolutePress,currentTemp);

            /*
            // USE FOR GROUND TESTING ONLY
               Serial.print(F("Current Temp: "));
               Serial.println(currentTemp);
               Serial.print(F("Absolute Pressure: "));
               Serial.println(absolutePress);
            */

        if (bmp180Status != 0)
        {
          return(absolutePress);
        }
        // else Serial.println(F("error retrieving pressure measurement")); //
USE FOR GROUND TESTING ONLY
      }
      // else Serial.println(F("error starting pressure measurement")); // USE
FOR GROUND TESTING ONLY
    }
```

```
    // else Serial.println(F("error retrieving temperature measurement")); //
USE FOR GROUND TESTING ONLY
  }
  // else Serial.println(F("error starting temperature measurement")); // USE
FOR GROUND TESTING ONLY
}
```

At this point in the program the function is complete and it will return back to the main loop. We have retrieved the data from the sensor, but we now need to calculate the altitude. This will be performed by the next function called in the main loop.

Pressure Sensor Calculations

From the main loop, the next function call is "`calculationsBMP180(void)`" and this function will provide us with the altitude of the rocket.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                PRESSURE SENSOR READINGS                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void calculationsBMP180(void)
{
  // ================================================================
  // Get a new pressure reading:
     absolutePress = readingsBMP180();

  // Show the relative altitude difference between
  // the new reading and the baseline reading:
     altitudeCalc = baroPressure.altitude(absolutePress,baroBaseline);
```

The heavy lifting of calculating the altitude is actually performed in the library. This makes our programming significantly easier as we only have two lines of code to write to obtain the altitude.

The code section below is for ground testing and will display the altitude in meters on the serial monitor. Then the program returns to the main loop.

```
  /*
  // ================================================================
  // USE FOR GROUND TESTING ONLY
  // print results - for ground testing only
     Serial.print(F("Relative altitude: "));

    if (altitudeCalc >= 0.0) Serial.print(F(" ")); // add a space for positive
      numbers
      Serial.print(altitudeCalc,2);
      Serial.print(F(" meters, "));
   */
  }
```

IMU Sensor Readings

The MPU6050 is our IMU sensor that will provide us with the data on g-force and roll rates. The MPU6050 library that we are using (https://github.com/jarzebski/Arduino-MPU6050) will perform most of the heavy lifting for us as far as the sensor calculations.

Rotational Readings

This tab starts off like all of the others with a header block to identify the purpose of this section of code.

```
/****************************************************************
 ****************************************************************
 *                                                              *
 *                MPU6050 SENSOR READINGS                       *
 *                                                              *
 ****************************************************************
 ****************************************************************/

void readingsMPU(void)
{
```

The next line of code allow us to get the reading from the gyroscope.

```
  // ==============================================================
  // Gyroscope/rotation readings in radians per second

  // Get reading
     Vector normGyro = mpu.readNormalizeGyro();
```

Once we have those readings they are assigned to the variables `roll` (for the X axis), `pitch` (for the Y axis) and `yaw` (for the Z axis)

```
  // Assign values
     roll = normGyro.XAxis;
     pitch = normGyro.YAxis;
     yaw = normGyro.ZAxis;
```

We finish this section with the code for ground testing this section. Currently commented out, when used in testing it displays the values of each axis on the Serial Monitor.

```
  /*
  // USE FOR GROUND TESTING ONLY
     Serial.println("Rotation");
     Serial.print(" Xnorm = ");
     Serial.print(normGyro.XAxis);
     Serial.print(" Ynorm = ");
     Serial.print(normGyro.YAxis);
     Serial.print(" Znorm = ");
     Serial.println(normGyro.ZAxis);
     Serial.println();
  */
```

Acceleration (G-force) Readings

The second part of this function returns the g-force values experienced by the rocket in all three axis. As with our roll rate readings, we first get the sensor readings

```
// ================================================================
// Acceleration readings in meters per second
  // Get reading
     Vector normAccel = mpu.readNormalizeAccel();
```

These reading are then assigned to the variables for the X, Y and Z axis

```
  // Assign values
     accelX = normAccel.XAxis;
     accelY = normAccel.YAxis;
     accelZ = normAccel.ZAxis;
```

We finish this section out in the same way as the one above. This is the testing code that will display these values on the Serial Monitor.

```
  /*
  // USE FOR GROUND TESTING ONLY
     Serial.println("Acceleration");
     Serial.print(" Xnorm = ");
     Serial.print(normAccel.XAxis);
     Serial.print(" Ynorm = ");
     Serial.print(normAccel.YAxis);
     Serial.print(" Znorm = ");
     Serial.println(normAccel.ZAxis);
     Serial.println();

     delay(100);
  */
```

This completes the reading of the sensors from the IMU. The program will now return to the main loop and execute the next command listed there.

Write Data to microSD Card

Each time you collect data from your sensors you want to save that information. We are going to save that information to a microSD card. To save the data from each round of sensor readings, a file on the microSD card must be opened, have the data written to it, and then closed.

Header and Local Variables

The function starts off with the typical header at the top and then a lengthy comment section on what this function will be doing. This provides helpful information for someone who is attempting to figure out what the code is executing instead of just reviewing the program code and trying to figure it out.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    WRITE DATA TO SD CARD                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void writeDataSDCard(void)
{
  // ==================================================================
  // Main Loop for microSD Card

  // Open the file. Note that only one file can be open at a time,
  // so you have to close this one before opening another.

  // CSV stands for "Comma Separated Values". It is a plain text file
  // format where each value is separated by a coma. It can be read by
  // nearly all spreadsheet and database programs.
```

Next is a local variable declaration section. By declaring the variables locally (as they are only used within this function) we can help save on memory space. If these variables were declared globally, we would loose a significant amount of memory.

```
  // make a string for assembling the data to write to the log
  String writeDataString = "";

  // local variables to write data to sd card
  String fileTimeStamp = "";
  String fileAltitudeCalc = "";
  String fileAbsolutePress = "";
  String filePitch = "";
  String fileRoll = "";
  String fileYaw = "";
  String fileAccelX = "";
  String fileAccelY = "";
  String fileAccelZ = "";
  String fileCurrentTemp;
```

The next section begins by collecting the sensor data that will be written to the file, beginning with the time stamp. Next it collects the data from the various sensors functions we executed earlier in the program and assigns each one to a variable.

```
  // assign data from sensor readings to variables for CSV string
  fileTimeStamp = String(timeStamp);
  fileAltitudeCalc = String(altitudeCalc);
  fileAbsolutePress = String(absolutePress);
  filePitch = String(pitch);
  fileRoll = String(roll);
  fileYaw = String(yaw);
  fileAccelX = String(accelX);
  fileAccelY = String(accelY);
  fileAccelZ = String(accelZ);
  fileCurrentTemp = String(currentTemp);
```

Now we combine all of these data fields into a single data string. Each data point is separated by a coma. We need to make sure that the data points are in the same order as the header that was created in the `setupMicroSDCard(void)` function.

```
// create data string
writeDataString = fileTimeStamp + "," + fileAltitudeCalc + "," +
fileAbsolutePress + "," + filePitch + "," + fileRoll + "," + fileYaw + "," +
fileAccelX + "," + fileAccelY + "," + fileAccelZ + "," + fileCurrentTemp;
```

Writing Data to the Card

We are now ready to write the data to the microSD card. We start by opening the file name that was created in the `setupmicroSDCard(void)` function and setting it to write data to the card

```
// open sd card to write
File dataFile = SD.open(fileName, FILE_WRITE);
```

Next the program attempts to write data to the card. If it is successful the data is written and the RGB LED status lamp turns green. If it cannot write the data to the card the RGB LED lamp turns blue. It is possible that if you have a loose connection or other issues with the microSD card or module that you may see the lamp flicker back and forth from green to blue.

```
// attempt to write to sdcard
if (dataFile) {
  //write to sd card
  dataFile.println(writeDataString);
  dataFile.close();
  ledGreen();
  ledSteadyLamp();
} else  // if the file isn't open, pop up an error:
{
  ledBlue();
  ledSteadyLamp();
```

If you are performing ground testing you can use this next line to display the error message on the Serial Monitor

```
  // Serial.println(F("Error opening data log...")); // USE FOR GROUND
     TESTING ONLY
```

The following line resets the microSD card module when an error occurs. When an error occurs this reset allows the program to attempt to write the next round sensor data to the card. This prevents a data write error from locking up the card due to a simple error and losing all of the sensor data.

```
  // reset the SD card to try and write data again
  SD.begin(pinSDcard);
}
```

Lastly we add a small delay of 1 millisecond to allow the system to reset and be ready for the next round of sensor data.

```
  delay(1);  // Pause for 1 milliseconds.
```

When this is complete, the software takes a 1 millisecond delay to allow the Arduino time to reset. The microSD card write function is now complete and will return to the main `loop()`.

The RGB_LED_Lamp tab

The last set of functions we are going to discuss are located on the `RGB_LED_Lamp.ino` tab. The LED lamp is used to provide status information on the avionics and the payload. There are a number of functions included on this tab that will provide those messages.

Header and Local Variables

This section is the same as all of the others as it includes a header identifying the section and three local variables. These variables are all integers which will carry a value of between 0 and 255 for each color Red, Green and Blue. By combining these values we can create a wide assortment of colors.

```
 *****************************************************************
 *****************************************************************
 *                                                               *
 *                    RGB LED Lamp Settings                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

// ==================================================================
// Local variables
   // define variables
   int redValue = 0;
   int greenValue = 0;
   int blueValue = 0;

   // choose a value between 0 and 255 on each variable to change the color.
```

Color Functions

The next series of functions create various colors that we can call from anywhere in the program. For this project we used only four colors; Red, Green, Blue and Yellow.

```
// ==================================================================
// Red lamp
   void ledRed(void)
    {
      redValue = 255;
      greenValue = 0;
      blueValue = 0;
    }
```

```
// ================================================================
// Green lamp
   void ledGreen(void)
    {
      redValue = 0;
      greenValue = 255;
      blueValue = 0;
    }

// ================================================================
// Blue lamp
   void ledBlue(void)
    {
      redValue = 0;
      greenValue = 0;
      blueValue = 255;
    }

// ================================================================
// Yellow lamp
   void ledYellow(void)
    {
      redValue = 255;
      greenValue = 50;
      blueValue = 0;
    }
```

Steady or Flashing

The last two functions on this tab contain the code that determines how the color will be displayed; either as a steady lamp or as a flashing lamp.

```
// ================================================================
// Solid lamp
   void ledSteadyLamp(void)
    {
      analogWrite(RED, redValue);
      analogWrite(GREEN, greenValue);
      analogWrite(BLUE, blueValue);
    }


// ================================================================
// Flashing lamp
   void ledFlashingLamp(void)
    {
      while (1)
      {
        analogWrite(RED, redValue);
        analogWrite(GREEN, greenValue);
        analogWrite(BLUE, blueValue);

        delay(100);
```

```
        analogWrite(RED, 0);
        analogWrite(GREEN, 0);
        analogWrite(BLUE, 0);

        delay(100);
      }
    }
```

Using the Status Lamp in the Program

These RGB LED lamp functions are used to provide status messages through the LED lamp. For example, both the color and state of the RGB LED can tell us if the microSD card has successfully written the data to the card or if the write attempt was unsuccessful. The code below is from the section of code where the program attempts to write the data to the card. The RGB LED function calls are highlighted in bold print

```
  // attempt to write to sdcard
  if (dataFile) {
    //write to sd card
    dataFile.println(writeDataString);
    dataFile.close();
    ledGreen();
    ledSteadyLamp();
  } else  // if the file isn't open, pop up an error:
  {
    ledBlue();
    ledSteadyLamp();
```

If the write function is successful, the first function call is to set the color to green. Next it calls the function for the steady LED lamp state. If the write attempt is unsuccessful it calls for the color to be set to blue and the lamp to be in a steady state. Every time we need to display a status through the RGB LED lamp we follow the same process; first set the color and second, set the lamp state to steady or flashing.

These are just some examples of what can be accomplished with the RGB LED lamp. By adjusting the values for the red, green and blue components of the bulb, various other colors can be created. You can also create different flash patterns by varying the lengths and number of flashes. For example, you may have a minor sensor warning that you want to be made aware of. You could change the color to yellow and flash the bulb 3 times, pause, then flash once and repeat the pattern. Using the combinations of colors and flashing patterns you could create an almost unlimited number of arrangements.

Code Review Conclusion

This concludes the review of the code for the avionics section of the Olympus Project. I hope this has helped explain what the code performs and why. Hopefully I’ve also been able to provide some ideas that will help you expand this project or provide inspiration for your own avionics projects.

10
Building the Avionics Bay

The avionics bay is a combination of electronics, sensors, and three 3D printed housings. They are combined together and fit inside the clear payload bay of the Olympus rocket.

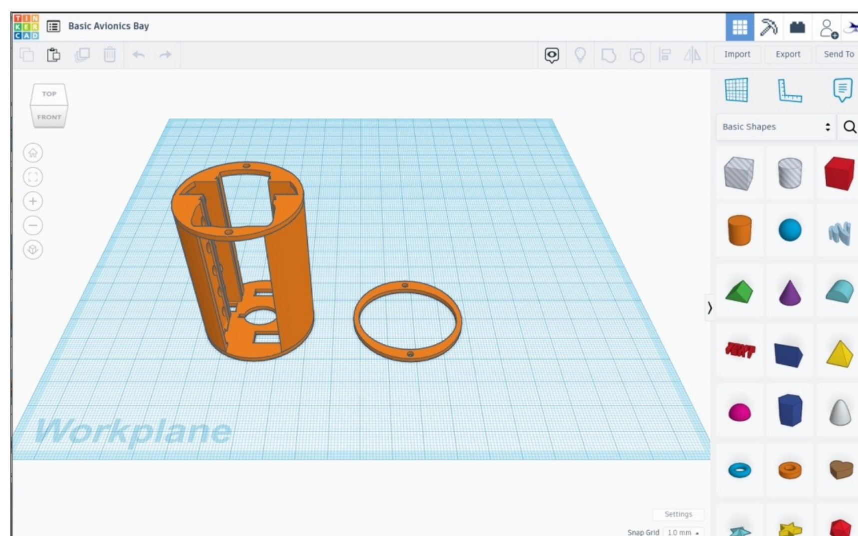
Designing the A-PAM Housing

The A-PAM avionics bay went through a number of iterations before we settled on the A-PAM design used in this project and others. When we first started out we were using a simple piece of foam board with the various components hot glued to the board. A 3-D printed base was added to secure the foam board inside the payload bay – but it didn't work well.

These designs really didn't give us what we were looking for. The Arduino Nano and the microSD card module ended up on the same side of the board, which meant you had to take out the entire payload and disassemble it to get the microSD card in and out of the module. The same was true of the USB connection on the Nano if you wanted to update or test the code. I simply needed a better design that didn't require the entire payload assembly to be taken apart after each flight.

Tinkercad

This led me to look at Tinkercad (https://www.tinkercad.com). It is a relatively simple CAD (Computer Aided Design) program and new users can get up to speed pretty quickly. Tinkercad uses basic shapes that are moved onto a workplane. The shapes can be adjusted in a number of ways. Different shapes can be merged together and you can take those same shapes and convert



them into "holes." On the previous page is a screenshot of the original A-PAM in the Tinkercad workspace

Once the project is complete you can export the project (either in parts or as a whole) in stl, obj, gltf (glb) format for 3D printing. You can also export it in svg format for use in laser cutting, and if you have an iPad you can export it in usdz format.

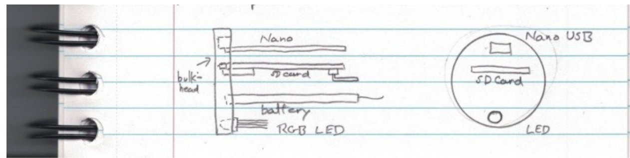
When you create your designs you can decide if you want to keep them private or make them public. The AAEN design gallery is located at https://www.tinkercad.com/users/kGt9Dmmc88b. Even when they are made public, you still need a Tinkercad account to view and download the designs. You can also keep a design semi-private by sending specific users a link to a design. These designs do not appear in your public gallery.

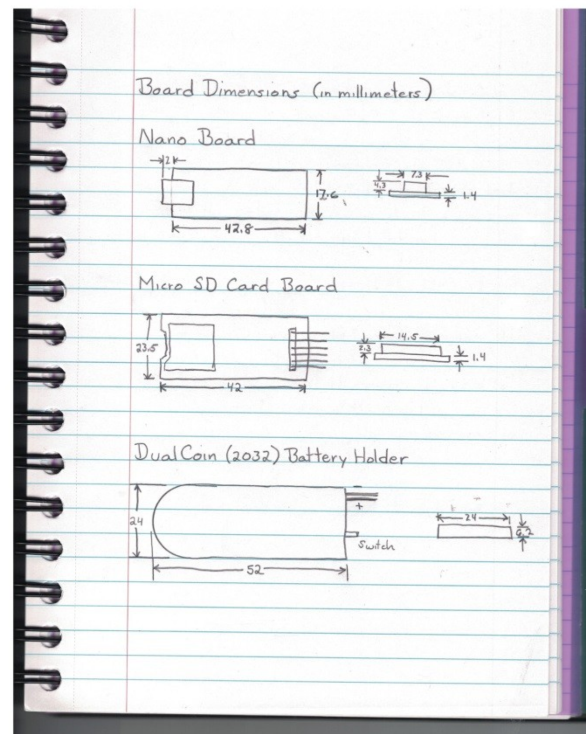
One of the downsides of Tinkercad is that it is a web-based program only. You have to use your browser to access and work on your designs. There is no off-line option available. It helps to have a robust Internet connection as you do send a lot of design information back and forth between your computer and the Tinkercad servers. That being said, I have been able to do somewhat moderate designs using my shop laptop with a32-bit AMD Athlon dual core and 4GB of memory running Q4OS Linux. So you don't need a state of the art machine to use Tinkercad.

Iterative Design

When designing the original avionics bay we started out by getting some basic measurements. We determined that the smallest diameter payload bay we could use is the BT-60. A quick search revealed the inside diameter of the payload bay (40.513mm). This matched our actual measurements of the BT-60 tubing. We also needed to know the dimensions of the main modules so I could start figuring out how to lay them out in a design that made sense.

The initial design sketches in my notebook got me started on the design process. The original layout was similar, but not exactly, as the final design.





With these basic sketches we started working on the initial design in Tinkercad. We found the design process went fairly smoothly, and it took considerably less time to make corrections and export a working and accurate stl file for printing. It took a series of design changes and prints before we settled on the final design now available on Tinkercad as the A-PAM project (https://www.tinkercad.com/things/7pel9mfXqzs). In the end the Nano stayed consistent from

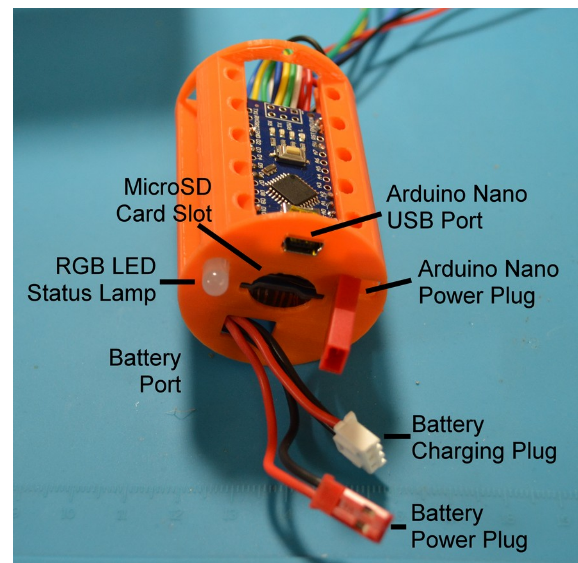
sketch to final print. The microSD card module was originally inverted, but in the final design was placed upright. The original coin battery in the sketch was replaced with a larger LiPo battery. With this change, the LED lamp would no longer fit at the bottom and was moved to the side. The original sketch did not include openings for the two power cables or the circular opening to assist in removing the microSD card.

Project: Icarus used the original A-PAM along with several temperature sensors placed along the body tube to measure temperature changes inside the body tube. For the *Olympus Project*, we wanted to use the same A-PAM setup, but knew we would need to add a sensor bay. We also weren't satisfied with the original attachment point for the recovery system, so we wanted to go back and take a look at that as well. Finally, the Olympus rocket was slightly larger than the original BT-60 used for the A-PAM design, so we needed to figure out how best to adapt to the larger payload bay.

The Olympus Avionics Bay

The Olympus Avionics Bay consist of three components – The A-PAM housing, the Payload Sensor housing, and the payload adapter base.

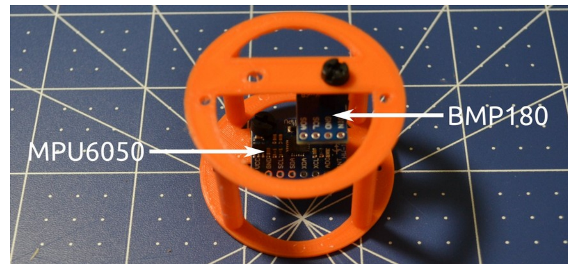
A-PAM Housing



The housing for the A-PAM components is designed to hold all four modules and it fits snugly into a BT-60 body tube. Since the Olympus is slight larger than a BT-60 tube (41.6 mm versus 44 mm), the top and base of the A-PAM housing was increased to make up the extra space. Looking at the graphic on the right of the original A-PAM we can see that the top bulkhead of the bay includes:

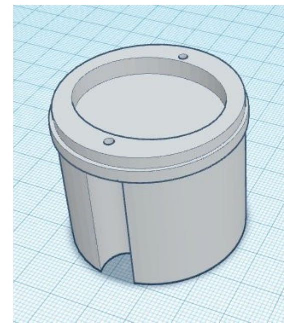
- Openings for the USB connection on the Nano.
- Directly underneath the Nano is the microSD card module. Like the Nano it too has a slot to allow you to insert and remove a microSD card.
- At the bottom is the LiPo battery holder with a rectangular opening that allows both the power plug and charging cord to exit the housing.
- Off to the left side is the RGB LED status lamp, while the right side opening allows the Nano power plug to exit the housing. When the battery and Nano power plugs are connected it turns the system on and provides power to the entire avionics system. No need for a switch.

Sensor Module Housing



The sensor module housing only needs to hold two sensors; the BMP180 and the MPU6050. It also needs to be able to attach to the A-PAM housing. That attachment is accomplished through the use of two 2.5m screws. The picture at the right shows a test fit of the two sensors in an early development print of the sensor payload housing. The BMP180 is at the top of mount while the MPU 6050 is at the bottom (this would be flipped in the final design). The mount was designed to be open enough to allow any wiring to pass between the sensors and the A-PAM. The final design needed to be expanded to allow enough room for the wiring around the Arduino Nano to fit into place.

Payload Base

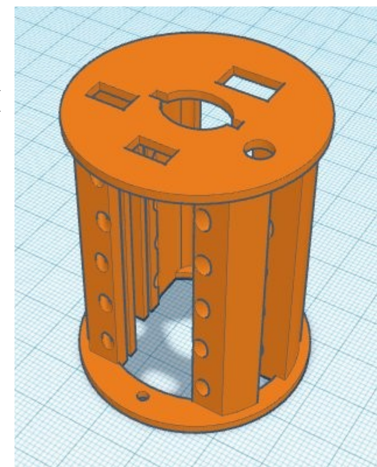


The third and final part of the avionics bay is the payload base. This base is attached to the sensor and replaces the original base that comes with the Olympus kit. The base is 3D printed and allows for the sensor bay to be screwed to the base. When all three pieces (the A-PAM, the Sensor Housing, and the Payload Base) are assembled together it makes up the complete Olympus avionics bay.

3D Prints

Both of the housings and the payload base were designed using Tinkercad (https://www.tinkercad.com/things/iY6COkklnYd). The designs are based on the original A-PAM drawings which are designed around the inner diameter of the BT60 body tube (40.5mm). The Olympus has a slightly larger interior diameter (44mm). To allow the original A-PAM to fit snugly inside the Olympus payload bay, the front and rear rings on the A-PAM were simply increased in size to accommodate the larger tube.

A-PAM Housing



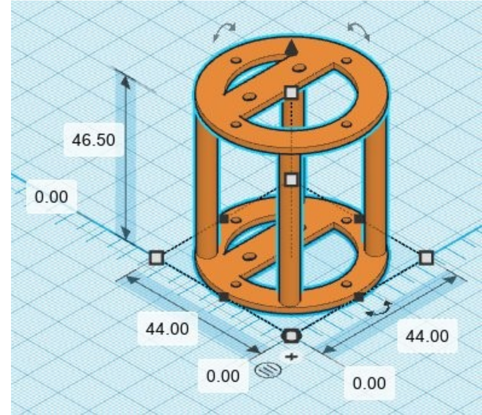
In the A-PAM we have provided openings in the housing wherever possible. The rails of the housing have multiple lightening holes. These can be used to route wiring if desired. For the basic module I am able to put the entire package together before inserting it into the housing.

The slots should be deep enough to hold the Nano board, yet not obstruct the wiring holes down each side of the board. While the two boards and the LED are held in place by friction, the battery has the greatest potential to move about. You should consider securing the battery in place using glue or other adhesive. If you find that the boards or the LED are not tight enough and prone to movement, consider securing them in place with a bit of adhesive as well.

At the rear of the housing is a ring with two small holes on opposite sides of the ring. These holes are used to attach any the sensor housing to the A-PAM. Use M2 nylon screws, washers and nuts to attach the sensor housing to the A-PAM housing.

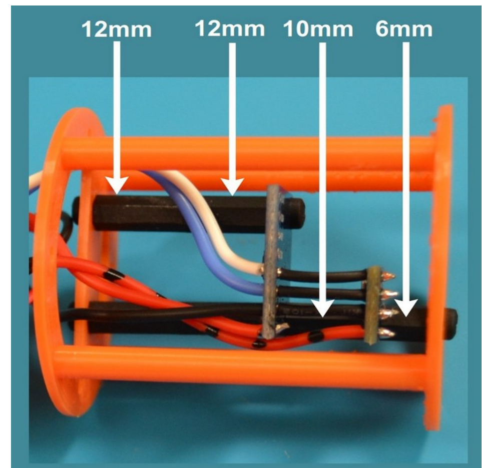
Sensor Housing

We start our design of the sensor payload housing with the adapter ring. This is just a copy of the ring that makes up the bottom of the A-PAM. By using this ring as the basis for our design, we can be assured that the sensor housing (or any payload housing you design) will line up and mate with the A-PAM housing. The Tinkercad drawing includes the adapter ring as a separate drawing.



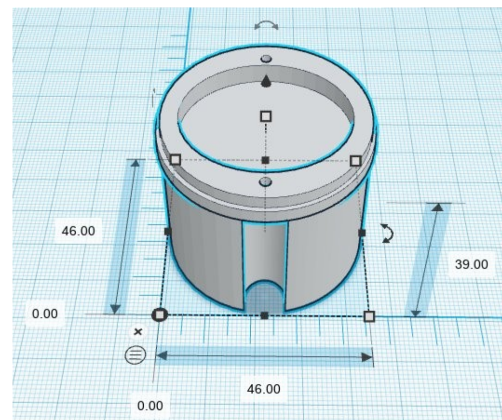
The sensor housing consist of two rings separated by four posts equally spaced around the ring. Each ring has an offset mount that allows a sensor to be screwed into place. The offset allows the MPU6050 to be centered in the housing. Both the top and bottom rings have the offset and two mounting holes to simply make installing the sensors easier.

The sensors are held in place by M2.5 screws. A series of spacers are used to allow for the wiring to flow from the sensor housing into the A-PAM. As shown in the graphic on the right there are two 12mm spacers located between the mounting bracket and the MPU6050. Between the BMP180 and the MPU6050 is a 10mm spacer. Finally there is a 6mm spacer between the mounting bracket and the BMP180. Check the fit of your sensor boards along with the spacers prior to finalizing the sensor housing. You may need to adjust the spacers or the housing to allow everything to fit properly.



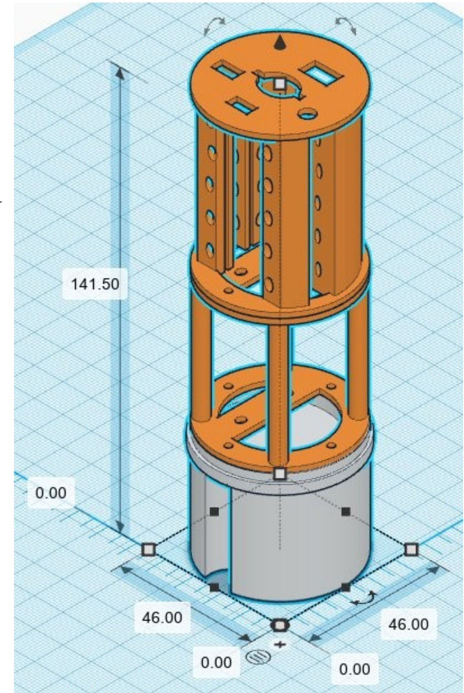
Payload Base

It was quickly realized that the payload base that comes with the Olympus model would not work with the electronic payload that was being designed. The standard base did not allow for any method of securing the payload in the payload bay. This resulted in a new base being designed and printed.



The payload base is designed to match up to the bottom ring of either the A-PAM or the sensor housing. The payload base includes the mounting point for the recovery system. The base also secures the clear payload section to the rocket.

The Completed Avionics Bay



When the three pieces (A-PAM, sensors and base) are stacked together they form the completed avionics bay. Housings are attached using M2 screws. This keeps the screws within the clear payload housing. The sensors are attached with M2.5 screws and spacers. The recessed area in the top of the payload base provide room for the screws heads.

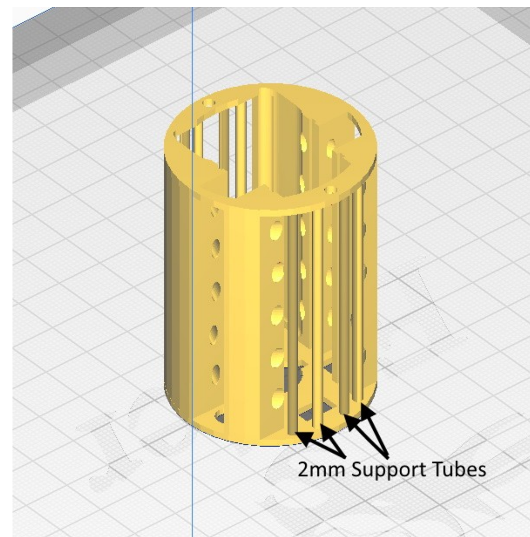
The payload base doesn't need the large surface area to hold the clear payload tube in place. This is because the sensor housing and the A-PAM housing all work together to secure the clear tube to the rocket.

As noted earlier the recovery system is attached to the payload base. As the base is part of the entire avionics stack, it helps ensure that the electronic payload section will stay attached to the parachute – even if it breaks away from the rest of the Olympus rocket.

Printing the Housings

The stl files were exported individually from Tinkercad and then imported in Cura 5.2.1. When printing these parts I used a 0.2mm layer height and 2-wall construction. Supports were necessary for the sensor housing and the A-PAM housing. Supports are optional on the base. Infill was set to 12% for all three parts.

Printing the Housings



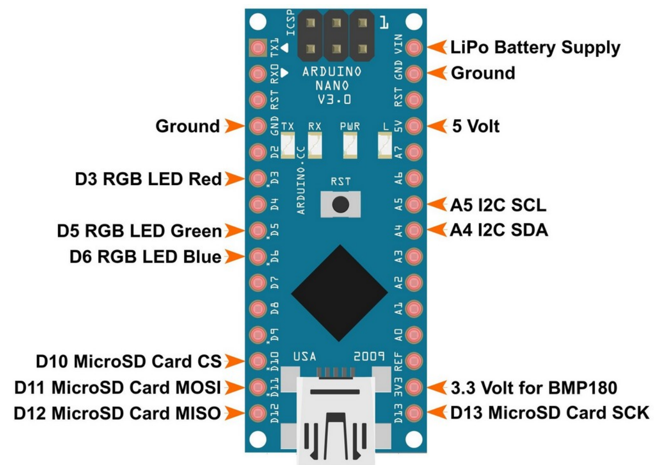
Both housings and the payload base was printed on an End 3 V2 printer. I have been using Hatchbox 1.75mm filament and it has worked well for me. The A-PAM and Sensor housings were printed in a bright orange color for high visibility in case it separates from the rocket. The payload base was printed in silver to match the rocket color scheme

I did add four custom support cylinders to help bridge the gap in the A-PAM and Sensor housings. Each support tube was just 2mm in diameter. For me, this arrangement seemed to work slightly better than using just two 4mm supports on each side. We also used the support tubes on the payload base were the screw holes are located. As with most projects like this, you may wish to try both. Results can vary from printer to printer so use the settings that work best for you.

> ***Note:*** *Since this project was completed, Cura has released an updated version that now includes tree supports. Although we haven't tried this support setting on the A-PAM, it might end up working better than the support tubes.*

Installing the Electronics

With the avionics and sensor housings printed, it is time to wire the components together. This takes some planning and patience as you are working in tight quarters. However, it is possible to get everything installed with room to spare.



The graphic on the right shows the connections needed for the avionics package. The LED lamp requires four connections, while the microSD card module requires six (including a common ground with the LED and card module). The two sensors both require I2C connections and the BMP180 needs a 3.3 volt power supply. To complete the wiring is the VIN and ground connection coming from the battery to the Nano.

I was able to determine pretty quickly that soldering the wires to the top of the Nano was going to cause issues. Instead, the wires come up from the bottom and are soldered into place.

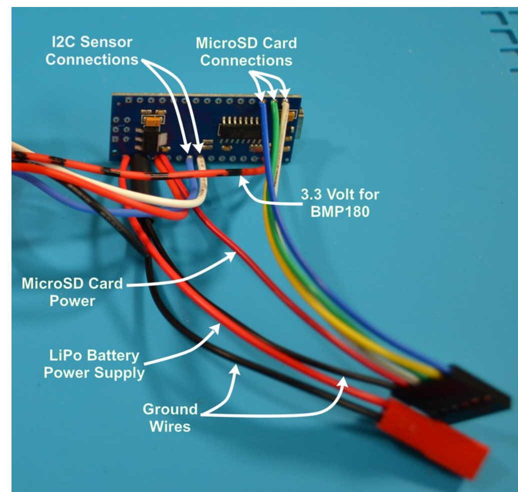
The next item that became readily apparent is that the components would need to be soldered in place first and then installed in the avionics bay. Trying to solder next to the plastic avionics bay was not going to end well.

Wiring the Components

There is no single method to installing the connections on the avionics package. You can do it in any order you think will work best for you. The order that I present here is just one method, but not the only method.

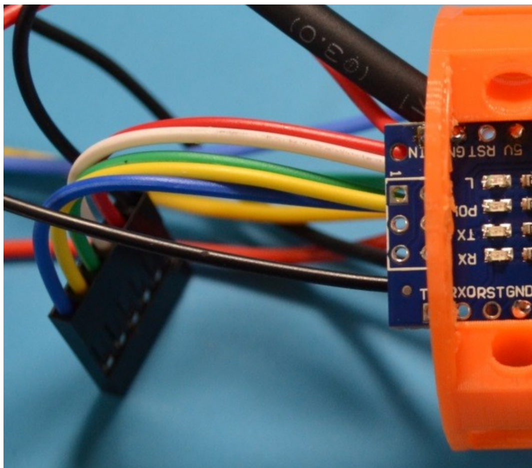
Installing the MicroSD Card Module

I gave considerable thought on how to attach the microSD card module to the Nano. The connections on the Nano are along the sides of the board, while the card module has pin connections that are at the rear of the board. While I did give some thought to removing the pins and soldering the wires directly to the board, in the end I decided to make use of the pins. The use of the pins actually seems to make the assembly easier.



In the picture on the right you can see five of the six microSD card module wires routed to the rear of the Nano (the others wires seen include the I2C connections, power and ground wires and the 3.3 volt connection for the BMP180).

Note: To help identify the 3.3 volt power line, it was marked with black stripes using a permanent marker.



The microSD card connection wires use a single six slot DuPont connector at the end. The wires are long enough to curl around and reach the pins of the card module.

The picture on the left shows the card module wires inserted into a six slot DuPont connector. The other end of the wires are soldered to the underside of the Nano and exit to the rear.

Before you install the Nano board into the avionics housing, you need to complete all of the soldering connections.

Power Connection

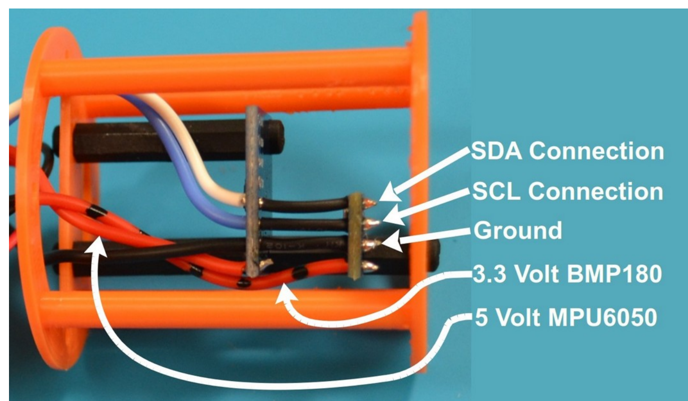
After trying several different power supplies I finally settled on a 7.4 volt, 200 mAh, LiPo battery. The battery has two connections;

- A 2-pin male JST plug for power
- A 3-pin plug for a USB charger

I soldered a 2-pin female JST connector to the Nano. This was made long enough to travel the entire length of the Nano board and exit out of the forward bulkhead. The male connection from the battery would also exit the forward bulkhead. When the two wires are plugged in it turns on the Nano and the avionics system.

Installing the Sensor Package

The two sensors share three common connections; SCL, SDA and Ground. This made the soldering of the three components a bit easier.

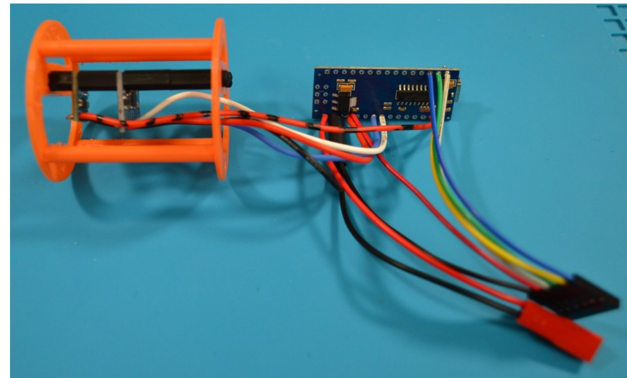


I started by soldering the 5 volt power supply on the MPU6050. Next I soldered the ground wire, but I made the ground wire long enough to extend from the MPU6050 to the BMP180. I did the same thing with the SCL and SDA wires. This finished up the MPU6050.

The next step was to solder the connections to the BMP180. First I soldered the 3.3 volt power supply (this is the red wire with the black stripes). Next I attached the 10mm spacer that connects the BMP180 to the MPU6050. I had three bare wires that would be going from the BMP180 to the MPU6050. I used small sections of heat shrink tubing to protect these wires. Next attach the

BMP180 to the spacer using a screw. Make sure to thread the correct wires through the appropriate pins on the BMP180. The two sensors end up with the connections directly in line with each other. I now had three wires through the pins on the BMP180 that I could solder in place. Afterwards the excess wire was trimmed off. With the sensors soldered together they were mounted in the sensor housing.

The next step was to solder the sensor wires to the Nano. As with the microSD card module, this was done before the Nano is installed in the housing. The picture on the right shows the sensors and sensor housing, with the wired connections going to the Nano. You should allow some extra wire to give both parts the ability to move while being assembled. However, too much excess wire will make the assembly much harder than it should be.

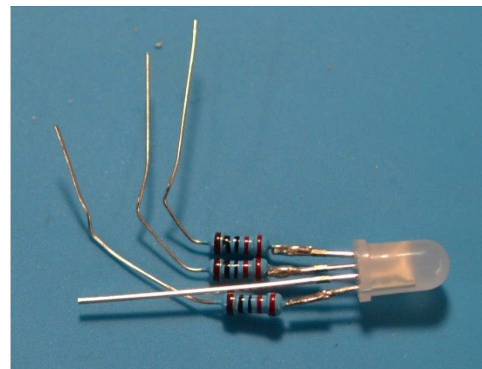


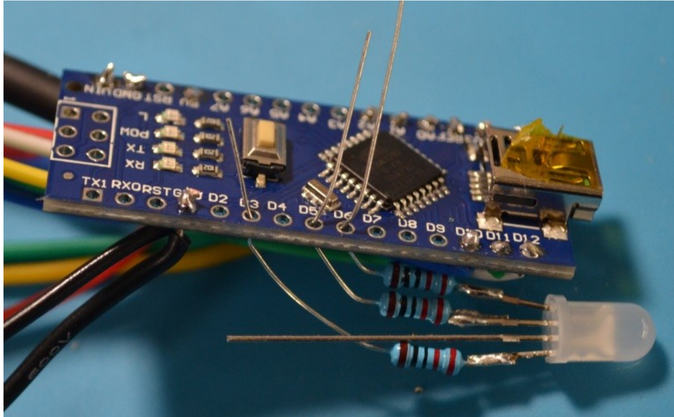
With all of the connections soldered in place and the sensors mounted in their housing, it was time to install the status lamp.

Installing the LED Status Lamp

The last component to install is the LED status lamp. Fortunately, all of the pins used for the lamp are on the same side of the Nano board and pretty close together. Unfortunately the lamp is located on the side between the two plastic mounting rails for the Nano and the microSD card. This does not leave much working room.

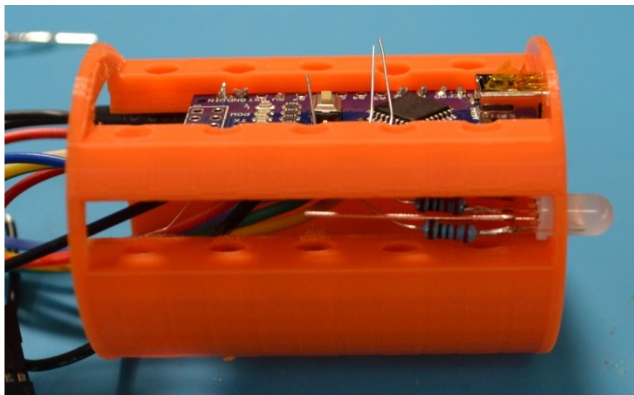
I knew early on that we wouldn't be using any extra wire for these connections. Except for the wire that connects to the common ground, this is true. The first thing we did was solder the 220Ω resistors to each of the red, blue and green pins on the LED. I cut the pins on the LED short, as I knew there was a limited amount of space to work with. I also want the wires from the resistors to fold back to reach the connections on the Nano. The picture on the right shows this part of the assembly. The resistors are soldered in place and the wires are bent into the approximate position on the Nano. The ground wire has not been soldered into place at this point.

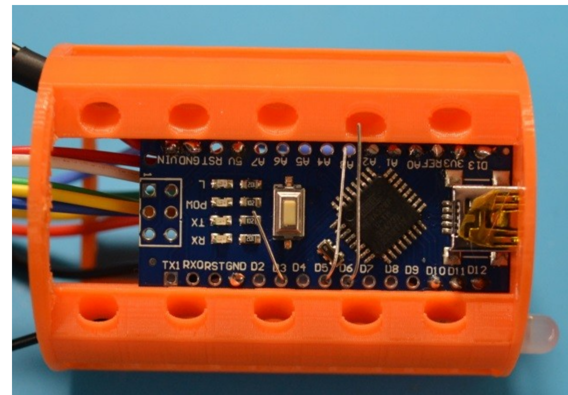


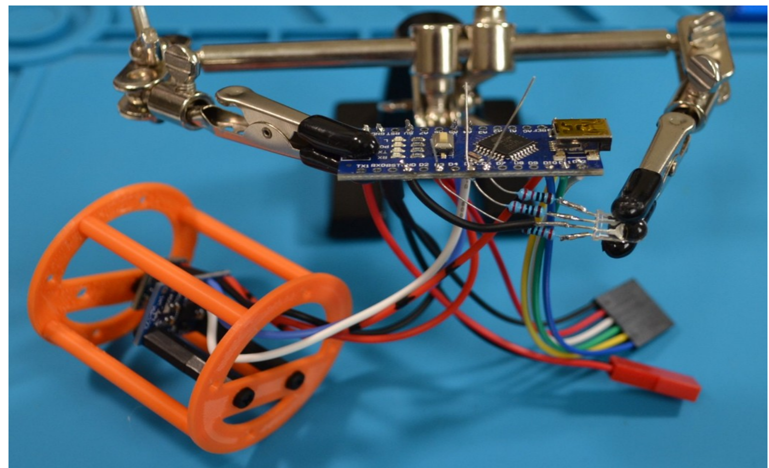


When the Nano board is inserted into the avionics bay, the board comes all the way to the forward bulkhead. I knew that the lip around the LED lamp would end at the bulkhead as well. Line up the LED lip with the end of the Nano board and then routed the resistor wires to the appropriate connections on the Nano. However, at this stage I was not ready to solder these wires in place. I needed to make sure the assembly would fit inside the avionics bay.

The two pictures below show the side and top views of the Nano and the LED status lamp placed in position in the avionics bay. With both pieces in place, the resistor wires were bent over to hold them in place. Both pieces were carefully removed and the wires soldered in place.
The picture on the right shows the completed LED lamp assembly. The common ground wire has been soldered into place and covered with heat shrink tubing.







In the picture on the right you can see the other connections are already in place. With the clamps holding everything in place we can solder the status lamp to the Nano and then trim off the excess wire.

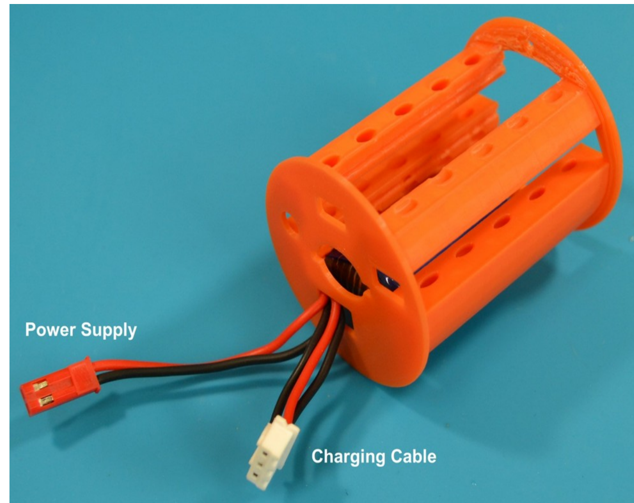
Before you move on to the final assembly, check your system to make sure everything is working as expected. It is important to test your components and assemblies as you go along. It makes it much easier to detect and troubleshoot issues when they occur.

Final Assembly

With everything soldered in place, it is time to start the final assembly of the avionics bay.

The Battery

The first component to be installed is the battery. The battery has two sets of wires coming from it. The charging cable has a 3-pin connector and is attached to the shorter wire set, while the power cable has a 2-pin connector but contains the longer set of wires.

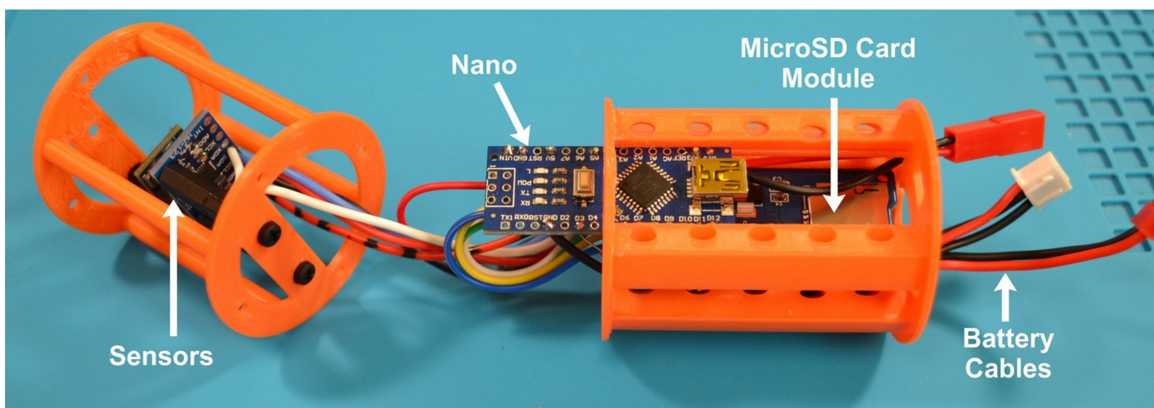


When inserting the battery, both connectors will not fit through the opening at the same time. Begin by inserting the charging cable and make sure it is completely through the opening.

Next insert the power cable through the opening. Once both cables and connectors are through the opening, push the battery forward into the avionics bay.

Nano & microSD Card Module Placement

The final assembly process begins by connecting the microSD card module to the Nano. With the two connected, begin to insert the entire assembly into the A-PAM housing. In the picture below you can see where the microSD module has been slid forward, with the Nano about halfway into the A-PAM housing. The sensor housing is to the rear as the cable lengths allow it to move out of the way while working on the A-PAM components. You can also see the battery cables are extended fully through the front of the housing.



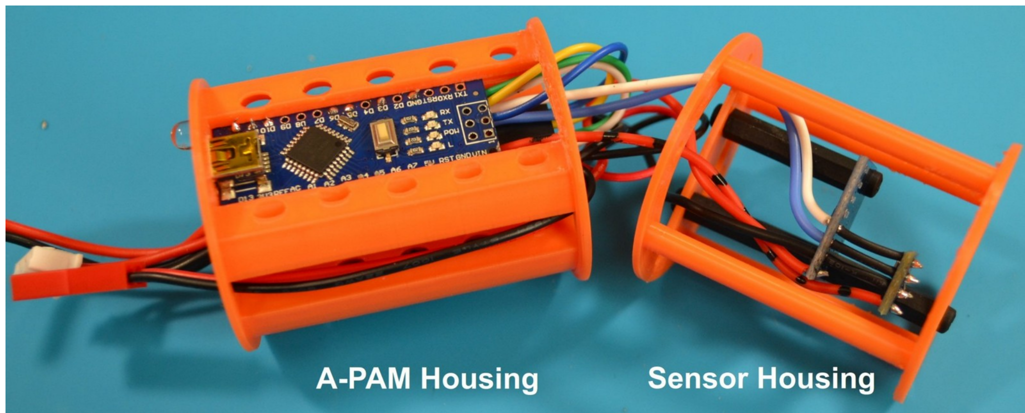
Continue to move the Nano and the microSD card module forward until the USB connection of the Nano is completely within the opening. The front of the microSD board should be flush against the inside of the forward bulkhead. The LED status lamp should slid into position along with the Nano. When moving this assembly take your time, making sure the LED lamp and the Nano both move forward at the same time.

The picture on the left, taken from the rear and above the avionics bay, shows both the Nano board and the microSD card module. The wires from the Nano to the DuPont connector are curled around to allow the connection.

Also clearly seen in this picture are the top and bottom screw holes in the rear bulkhead. The Sensor Housing will attach to the avionics bay using these two holes and a 2M nylon screw.

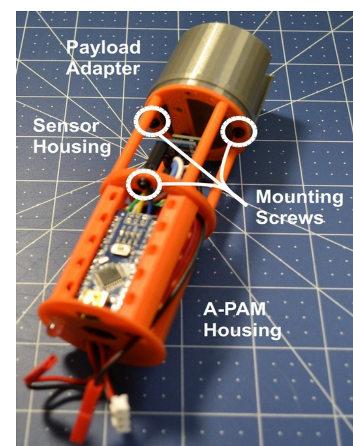
Attaching the Sensor Housing

With all of the components secured in the A-PAM housing, it is time to attach the sensor package. The picture below shows the two components being lined up. You will need to work the wires into the sensor housing as you bring the two housing together. Once lined up insert two screws to secure the two housing together.



Attach the Payload Adapter

The final part that needs to be attached is the Payload Adapter. There are two screws that attach it to sensor housing. Make sure that these screws are secure, as this is what attaches the entire payload assembly to the recovery system. The picture to the right shows the completed assembly.



This completes the assembly of the avionics package for the Olympus Project.

11
Vent Holes and Temperatures

The *Olympus Project* uses the BMP180 barometric pressure sensor to determine the altitude of the rocket. The BMP180 is much like many other altitude sensors in that it uses changes in barometric pressure and temperature to calculate altitude.

Are Vent Holes Needed?

Prior to July 1, 2017 if you were flying in an altitude competition sponsored by the National Association of Rocketry, the US Model Rocket Sporting Code required that the rocket have vent holes in the altimeter payload bay. Now, vent holes are no longer required for competition flying and some of those flying in altitude competitions no longer use vent holes. So are they really needed?

Research Using Simulations

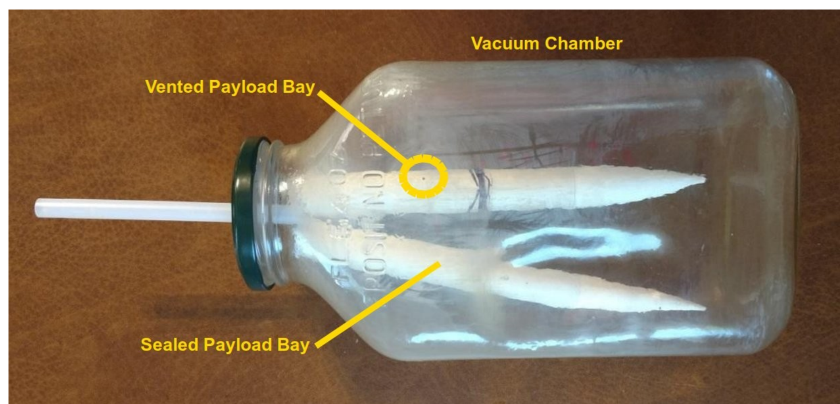
For the Research & Development competition during NARAM 59 (National Association of Rocketry Annual Meet), Chad Ring submitted the research project "*To Vent or Not to Vent, That is the Question*" which looked at the need to provide vents for rocketry altimeters. In his project he created two payload sections; one had small vent holes that are typically found on a competition rocket, and one that was sealed.

Three pairs of altimeters were tested, for an overall total of six altimeters. They included:

- Micro Peak – two each
- Pnut – two each
- Firefly – two each

The altimeters were tested in similar pairs, with one altimeter placed in the vented compartment while the second altimeter was placed in the sealed compartment.

To simulate flight a small vacuum chamber was constructed that would allow both payload sections to be tested at the same time. By drawing a vacuum this would simulate the less dense atmosphere of higher altitudes. By having both altimeters exposed to the same conditions it was easier to compare the results.



Note: *The graphic above is from the NARAM 59 R&D report, with labels added for clarity*

In 24 out of 25 sample flights, all but one showed higher altitudes when using a vented versus a sealed payload bay. The overall average of all 25 flights with the 3 different altimeters showed the following:

	Vented	Sealed	Difference	Percentage
Micro Peak	325.8	239.56	86.24	30.51%
Pnut	463.76	293.08	170.68	45.10%
Firefly	463.80	292.68	171.12	45.24%

During NARAM 60 a followup study was performed by Mr. Ring. This study included the Adrel ALT-BMP altimeter as it was claimed that this unit did not need any vent holes. Using the same methodology as his previous study he compared the Micro Peak altimeter in the vented and sealed payload bays to the AdrelALT-BMP in the sealed payload bay. After 20 simulated flights the Adrel ALT-BMP altimeter in the sealed payload bay performed nearly identical to the Micro Peak in the sealed bay. These were both significantly less than the Micro Peak altimeter in the vented payload bay.

The chart below shows the average after 20 simulated flights

Micro Peak Vented	Micro Peak Sealed	Adrel Sealed	Adrel Vs Micro Peak Vented	Percentage	Adrel Vs Micro Peak Sealed	Percentage
358.75	290.8	290.75	-68	20.94%	-0.05	0.02%

The simulated flights showed a significant difference in the altitudes recorded when using a vented payload bay versus a sealed bay. However, as Mr. Ring noted at the end of his two reports, "The next logical step is to still do actual flight tests with these units." The question is would real world conditions reveal a difference.

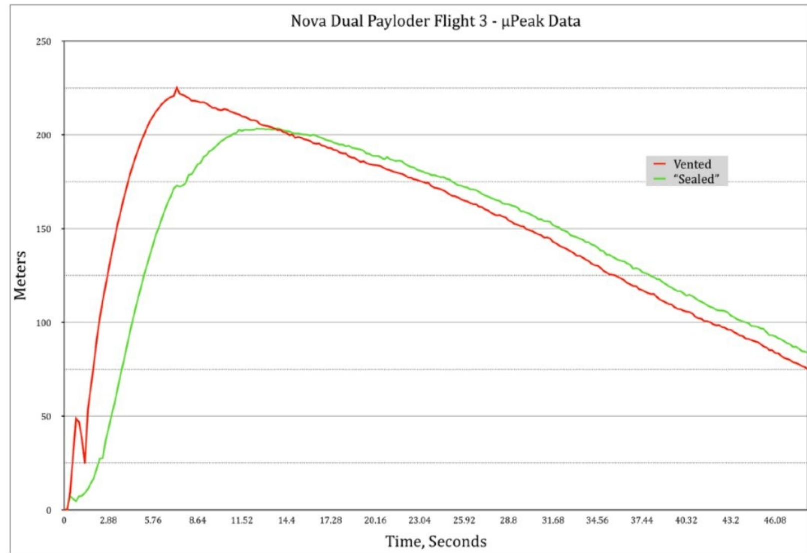
Research Under Flight Conditions

NARAM 61 would see a third research report looking at vent holes and altimeters. This report was written by Bernard Cawley and had a title of, "A*ltimeter Venting - Do I REALLY Have To? OR: If you fly nine altimeters in one rocket, do you really know how high it went?*"

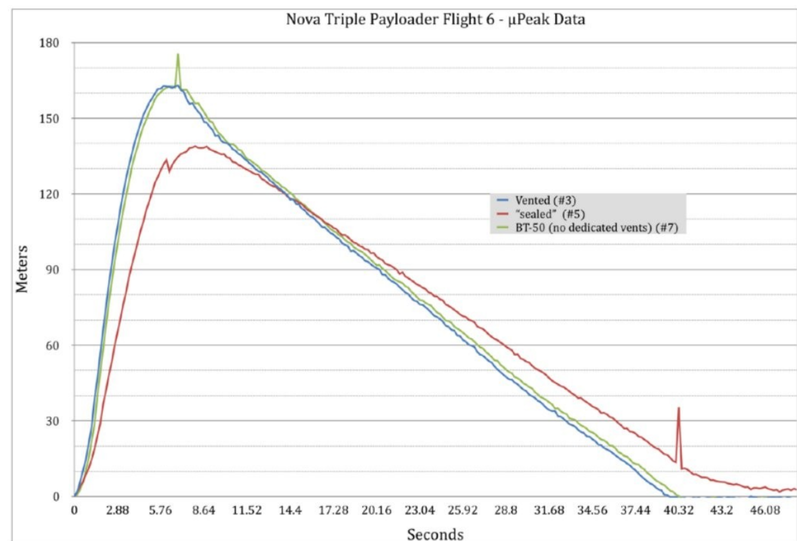
In this study Mr. Cawley used three different altimeters; Micro Peak, Adrel and Firefly. Using an Estes Nova Payload as a launch vehicle, a series of flights were conducted using a sealed payload and vented payload as done in the previous two studies. Additionally, he tested a payload compartment that had large vent openings and a payload compartment that did not have any vents but was not sealed either. This payload used "natural" venting, where a balsa block was used in combination with a nose cone where neither one was sealed to the tubing.

The flight data supported the previous studies that showed a sealed compartment results in lower altitude readings compared to a vented compartment.

Additionally it also showed that the altimeter in the sealed compartment reported information later in the time line as opposed to the vented compartment (see graph on the right from the study).



A new test performed by Mr. Cawley saw the use of a payload tube that was neither sealed nor vented. It simply used the natural openings around the payload tube to provide passive venting. This was found to be nearly as effective as including vent holes in the payload tube. In the graph on the right the two lines for the vented and natural vented are nearly identical. The sealed payload bay still shows a lower altitude and a delay in reporting.



These studies show that there needs to be some method for the atmospheric pressure to equalize within the payload tube. Failure to provide that venting (either through vent holes or relying on natural ventilation) will result in accurate altitude data.

Calculating Vent Holes

In order to accurately sense the changes in atmospheric pressure, the payload bay must be exposed to the outer atmosphere. If we decide to use vent holes that are added to the payload bay to allow the pressure inside the bay to equalize with the pressure outside the bay, how do we determine the size of the vent holes or where they are located? We are going to look at exactly how we calculate the size of our vents as well as some common misconceptions concerning vent holes.

Rules of Thumb

One of the more common "Rules of Thumb" you will hear when calculating vent holes is that you need one ¼-inch hole for every 100 cubic inches of payload interior. This is the same calculation that is used in the Rocketry Workbench module in FreeCAD.

Another rule you may hear is that for low or mid power rockets you should use 3/32-inch holes in models up to 1.6 inches in diameter, and 1/8-inch holes for larger diameter models. This is basically a catch-all to try and make sure that some type of venting occurs.

Calculations that are a Bit More Specific

David Walker of Cambridge University Space Flight (https://www.cusf.co.uk/) presents a calculation (https://www.cusf.co.uk/2016/09/sizing-static-ports-of-altimeter-sampling/) based on Bernoulli's Principle. From the web page;

> *First decide on the maximum allowable error in the altitude, say 10m. This provides the pressure difference between the altimeter bay and the free stream. Next find the maximum flight speed, this gives the maximum rate of change of pressure. Modeling the altimeter bay as a volume V with an orifice area A and pressure drop delta p this gives all the information needed based on Bernoulli's equation for the flow through the orifice and mass conservation.*

In his example he uses a 10-meter error rate and a maximum velocity of Mach 2. The payload bay is 108mm in diameter and 620mm in length, with four vent holes. His calculation results in a 0.176″ hole for every 100 cubic inches. This is significantly less than the rule of thumb discussed above.

Finally, Gary Stroick in March of 2020 published the report, "HPR Research - Static Port Holes from Nescience to Science" (https://documents.pub/document/hpr-research-static-port-holes-from-nescience-to-port-holespdf-hpr-research-.html?page=1) where he looks at four of these equations and breaks them down according to their reliability. Instead of using a "Rule of Thumb" to determine port size, he turned to the specialty of fluid dynamics to determine proper vent hole sizing. We won't go into the equations here, but instead direct you to a spreadsheet that Mr. Stroick has made available for free at https://www.offwegorocketry.com/userfiles/file/Calculators/Static%20Port%20Holes.xls.

Calculating Vent Holes for the Olympus Payload Bay

Using Mr. Stroick's spreadsheet, we entered the following values into the spreadsheet

- Highest Launch Temperature = 38ºc
- Lowest Launch Altitude = 198 m (the elevation of our town)
- Maximum Velocity = 106 m/s (from the OpenRocket simulation)
- Altimeter Bay Radius = 2.2 cm
- Altimeter Bay Length = 10.5 cm
- Number of Ports = 4

The result of the calculations (shown in the screen shot on the next page) shows that each vent port in the Olympus should be 0.5mm in diameter. That is four rather small vent holes.

Constants				
	Imperial		Metric	
Variables	Values	Units	Values	Units
Air Specific Heat Ratio	1.401			
Discharge Coefficient	0.62			
g	32.174048556	ft s^{-2}	9.80665	m s^{-2}
Gas Constant	89494.596	lbm ft^{2} lb-mol^{-1} K^{-1} s^{-2}	8.31432	N m mol^{-1} K^{-1}
Lapse Rate	-0.0019812	°K ft^{-1}	-0.0065	°K m^{-1}
One Atmosphere	29.92126	inHg	101325	Pa
One Mole Air Density	28.9645	lbm lb-mol^{-1}	0.0289645	kg mol^{-1}
Conversion Factors				
	Values	Units		
inHgtoPSF	70.7261873	lbf ft^{-2}		
JouletoLAtm	0.0098692	J L^{-1} Atm^{-1}		
LitertoCubicInches	61.023744	in^{3}		
MeterstoInches	39.37007874	in		
MeterstoFeet	3.280839895	ft		
Enter Data Values into Turquoise Cells				
Measurement System	**Metric**			
Anticipated Launch Parameters (Optional)	**Values**	**Units**		
Highest Launch Temperature	**38**	°C	9.5	°C
Lowest Launch Altitude	**198**	m	253	m
Maximum Velocity	**106**	m/s	122	m/s
Avionics Bay Parameters (Required)	**Values**	**Units**	Values	Units
Altimeter Bay Radius	**2.2**	cm	7.4	cm
Altimeter Bay Length	**10.5**	cm	29.2	cm
Number of Ports	**4**		4	
Atmospheric Pressure at Launch Altitude	99141.517244	Pa	98264.644	Pa
Air Density (Dry Air)	1.1100053091	kg/m3	1.21112117	kg/m3
Atmospheric Pressure Change	0.01102%		0.00012157	
Pressure Equalization Rate	0.0094339623	s	0.00819672	s
Choked Threshold	1.8935326534		1.89353265	
Altimeter Bay Volume	159.65573866	cm3	5023.38152	cm3
Air Mass	0.0001772187	kg	0.00608392	kg
Mass Flow Rate	2.07103E-06	kg/s	9.0233E-05	kg/s
Size of Ports	**0.5**	**mm**		

While our use of the altimeter in the Olympus is simply to obtain altitude data, altimeters in larger, high power rockets are often used to deploy recovery systems. At the time of his report, 73% of all flight failures are due to recovery system failures. Of those 31.5% are due to failure of the motor ejection or electronic ejection systems. Proper use of altimeters, when part of the recovery system deployment, is crucial to a safe launch and recovery of the rocket. That is why it is important to understand why the proper venting of the altimeter payload pay is important.

Locating the Vent Holes

Now that we know how big our vent holes need to be and how many we are going to use, the next question we need to answer is where the holes need to be located. They need to be in the area of the payload bay, but is one location better than another?

First off, the vent holes do not need to be located next to the altimeter. If the altimeter is just one sensor within a much larger bay (such as occurs with our Olympus payload bay) then the vent holes will help equalize the pressure throughout the entire bay, regardless of where the altimeter is actually located.

Second, you want to select an area that is relatively smooth. You don't want anything sticking into the airflow either beside or below the vent hole. Such protrusions will cause turbulent air flow and can interfere with the proper operation of the altimeter. This can become an issue if you have a camera in a housing that is attached to the side of the payload bay, or a launch lug located near the altimeter bay. You should also avoid locating the vent holes near changes in the airframe of the rocket, such as adapters/boat tails. These can also result in turbulent airflow around the vent holes.

On our Olympus model we will locate them near the middle of the payload bay. This is a smooth area and is a significant distance from the nose cone/payload bay junction. Once the holes are opened, they need to be smooth on the outside of the airframe. With the plastic payload bay this is not difficult to accomplish. With a paper payload tube you might consider applying some CA glue to the edges of the vent hole to strengthen it and sand down any rough edges.

Other Considerations

There are several other things to keep in mind when using altimeters. They include:

- *Keep it cool*
  An accurate temperature reading is needed to measure the pressure so try not to expose the device to rapid temperature changes. Also keep it away from any other parts that generate heat.
- *Keep it dry*
  The BMP180 is sensitive to moisture. Don't allow it to contact liquid water. Rainy days are not friends for your BMP180 sensor
- *Don't blind it*
  The silicon within the BMP180 is sensitive to light. For maximum accuracy, shield the chip from ambient light as much as possible. This can be difficult when using a clear plastic payload bay.
- *Keep it away from ejection charge gases*
  The gases generated by the ejection charge are corrosive. This can wreak havoc on the electronics. The Olympus uses a dedicated payload pay that separates it from the ejection charge. If your altimeter is not isolated (such as a stand-alone altimeter attached to the parachute/shock cord) you can try to lightly wrap recovery wadding around the device. Do not wrap it so tightly as to block off the vent holes in the device.
- *Handle gently*
  The tiny pressure sensors can be affected by violent motion. This can cause, at best, false

readings, or at worst, activate a recovery device or other device based on errant senor readings.

One More Thought on Temperature

The pressure sensor used by the BMP180 uses the change in atmospheric pressure as well as the temperature recorded at the sensor to calculate altitude. However, it is important to understand that this temperature is not the same as ambient air temperature. The temperature inside the payload bay can be substantially different from the air temperature outside the payload bay. Consider the temperature inside a payload bay that is painted black on the outside on a nice hot summer's day. It could be as much as 20º C warmer inside the payload bay versus outside the bay.

All commercial model rocket altimeters assume an ambient air temperature of 15º C when calculating altitude. This is based on the *International Standard Atmosphere* model. Yet to obtain an accurate altimeter reading, we need to know the ambient air temperature and take that into account. The formula we use is;

$$Altitude = \left(\frac{273.15 + T}{288.15}\right) \times Altimeter$$

where *T* is the ambient air temperature at launch and *Altimeter* is the altimeter reading.

How big a difference can air temperature make? Here are two examples.

Summer Example

Our rocket is flown on a summer day with an air temperature around 29º C (84º F). The altimeter reports an altitude of 345 meters. To determine our actual altitude:

Altitude = ((273.15 + 29)/288.15) x 345
Altitude = (302.15/288.15) x 345
Altitude = 1.048585806 x 345
Altitude = 361.76210307
Altitude = 361.76 meters

Winter Example

Now lets look at a winter flight where the temperature is around 2º C (35.6º F). The altimeter reports an altitude of 345 meters. To determine our actual altitude:

Altitude = ((273.15 + 2)/288.15) x 345
Altitude = (275.15/288.15) x 345
Altitude = 0.9548846087 x 345
Altitude = 329.4351900015
Altitude = 329.44 meters

Despite the altimeter recording the same altitude for each flight, there was actually a difference of 32.32 meters between the two flights.

12 Using the Olympus Electronic Payload

When you are ready to use your altimeter in an Olympus flight, the following steps are recommended.

Preflight

- Determine which microSD card will be used for flight. It is recommended that a blank card be used and that you format the card prior to use. Place it into a computer to check for errors.
- You may wish to rename the card to something that make sense for the mission. This may include a flight date, project name, session number, etc.
- Charge the battery. This will help ensure that you have plenty of battery power should you find your rocket and payload sitting on the pad for an extended period of time.
- Do a preflight test on the payload. Insert a microSD card and turn on the system. Move the payload bay around and then turn the system off. Remove the card and insert it into a computer and read the data. This will help ensure everything is working properly before heading to the pad.
- Place clear payload bay over the electronic payload. Secure to payload base using tape. Do not cover vent holes located near bottom of the payload bay.

At the Launch Pad

- Insert the flight ready microSD card into the payload bay.
- Connect the two battery clips to power on the system. The status lamp should turn green to indicate the system is functioning as expected. If you get a color other than green, it indicates an error in the system. Error messages include:
    - Flashing Red – microSD Card Module error.
      Check that a card has been inserted into the holder. If a card is present remove and reinsert. If you still get a flashing red lamp, try a different card. If still getting a flashing red lamp, check for loose connections
    - Flashing Yellow – MPU6050 error.
      Check for loose connections
    - Flashing Blue – BMP180 error.
      Check for loose connections
    - Steady Blue – File Write error
      This indicates that there was an issue while attempting to write data to the microSD card. This error will attempt to self-correct by resetting the microSD card module. If it does not reset check the microSD card and replace.

- If payload appears to be working properly, attach nose cone to payload bay and secure with tape.

During Flight

- Observe the flight of the rocket so that it can be recovered successfully
  - You may want to use multiple observers. If the payload bay were to separate from the rocket one observer can track the payload bay while the second observer tracks the rocket.
- Track payload bay to landing site. Upon arriving on the landing site, remove the tape from the nose cone and remove the nose cone from the payload bay.
- Disconnect the battery from the payload to turn off the payload
- Observe the overall condition of the payload. It anything appears missing, you may want to search the local area.

Post Flight

Upon returning to the flight preparation area

- Remove the payload bay tubing from the payload
- Remove the microSD card from the payload.
- Insert the microSD card into a computer. Confirm that there are data files on the card. Copy data files from microSD card onto the local computer
  - During the appropriate time, perform analysis on the data collected
- Check the rocket and payload. Make note of any damage or missing components. Depending on any damage that has occurred, make a determination
  - on whether it is safe to continue flying in the current condition
  - to repair the rocket on site and attempt to fly again
  - if the rocket will need extensive repairs offsite in order to fly again
  - if the rocket has sustained major damage and is no longer in a repairable condition
- Make any notes and observations that may be appropriate for the mission/project.

13 Tactics to Improve the Olympus Project

Any time you create a project you should always go back and see how it might be improved. Such "lessons learned" will help you improve projects that you create later. It can also be beneficial to try some of the ideas on the current project. Improvement through iteration is a valid engineering process. Companies like SpaceX have made great strides in rocket performance by trying new things, and being willing to break stuff to learn in the process. With that background, let's take a look at some of the changes, updates and improvements that can be made to this project.

Launch Vehicle

In this project we decided to use an 'off the shelf' model rocket. Despite being a manufactured rocket kit, it has an odd-size, non-standard diameter. This rocket is slightly larger in diameter (44mm inside diameter) than the standard Estes BT60 body tubes (40mm inside diameter). The larger tube does allow for the rocket to be able to use a raw Grade A egg as a payload. It also meant that we had to modify the A-PAM housing and sensor housing to fit the larger payload bay.

The clear payload housing does allow for the electronics to be visible, but it also allows sunlight to enter the payload bay. This can cause havoc with the BMP180 sensor. Due to the odd diameter of the bay, there was no readily available body tube we could use to substitute for the payload housing. The other option would be to cover or paint the clear housing.

The rocket was built mostly according to the instructions. There was a modification in replacing the rubber band shock cord with an elastic strap. Future versions may look at using Kevlar cord attached to the motor mount as an attachment for the shock cord. One needs to be careful as these Kevlar cords have been known to cause 'zippering' down the side of the rocket during the ejection charge.

Another option that could be looked at in future versions is to use a rail launcher instead of a launch rod. The use of a rail launcher would eliminate the need for a launch lug and so the performance of the rocket should improve with the cleaner airframe.

Electronics

The project made use of the original A-PAM. The Arduino Nano was chosen for its size, flexibility and ease of use. However, using the basic Nano resulted in an issue with memory space, although the available program space was beginning to decrease to the point it might have

become an issue as well. Fortunately there are other members of the Nano microcontroller family available that can expand the use of the system.

There are several members of the Nano family that come equipped with WiFi and/or Bluetooth. Having a board that can communicate wirelessly allows you to check the status of the payload and the avionics while you are located a safe distance from the pad. You could even make updates to the onboard coding if something doesn't look right. Having the ability to communicate with your avionics is a big step forward.

There are also Nanos that have built in pressure sensors and IMUs. While this looks promising, further investigation reveals that the IMUs are restricted to no more than 4Gs of force. This appears to be a software library restriction as the onboard chip is rated to 16Gs. However, until that restriction is lifted, the presence of an onboard IMU does not offer any advantage.

The most recently addition to the Nano family is the one that probably holds the greatest interest for improving the A-PAM, and that is the Nano ESP32 board. There are a number of improvements in this board above the standard Nano board. It includes WiFi and Bluetooth connectivity. It also includes a faster processor and more memory. The only downside that we currently see is that the board only has 3.3V output, no 5V. That would mean that we would need to find a compatible IMU.

The Nano ESP32 allows you to write software using the Arduino IDE or you can utilize MicroPython. This is the first Arduino Nano that will natively support MicroPython.

The NanoESP32 is built on the same board size as the standard Nano. This means that there is no need to modify the A-PAM housing for the new board. Also, the pin assignments are the same, so any code written for the Nano should easily transfer to the Nano ESP32 (we haven't tried this yet, but likely will in our next payload build).

Another area that can be looked at is the power source. The current design uses a 7.4 volt, 200mAh rechargeable LiPo battery. However, depending on the project you are designing, you may have different power requirements. That can lead to a change in the battery you use. Some of your components may need a steady power stream so a voltage regulator might need to be added.

You should keep in mind that you want to use a battery that will provide power for the entire mission and not just the flight itself. There can be delays in the lift-off or the rocket may stay airborne longer than anticipated. If being used as part of a recovery system you may need the power to last for a significantly longer time to allow you to find the rocket and payload.

As you make hardware changes, you must remain cognizant of changes in component sizes and weight. Is there a change to the weight of the rocket and how might that affect its performance? Before actually flying the rocket, all of these questions (and more) should be thought out and if possible, flight simulated to see what the impact will be.

Software

Software changes are easy to implement and easy to revert back if things don't go as planned. It is one of the big advantages of software based electronic systems. The *Olympus Project* could also benefit from such improvements. Here are a few things that you might consider as you begin you're Olympus project.

> ***Note:*** *One of the issues that we ran into with the Nano was that we quickly ran out of memory and were running out of storage space. This resulted in having to comment out code that used the Serial Monitor to keep us within working memory limitations. In order to accomplish any of the changes listed here, an upgrade in the Nano to increase program storage space and memory space will be needed.*

One item that would result in a big improvement is the addition of a clock synchronization function. The current software is based on a time stamp that starts when the Arduino is turned on. While this will work with the sensors attached to the Nano, it can become more difficult to synchronize with other components. This could include a video camera that uses the current date and time to synchronize video. By developing a routine to synchronize the avionics and other components to the same clock (such as the clock on a local laptop), it can be easier to sync the data from multiple components.

Most of time this is accomplished through the use of a Real Time Clock (RTC), but you do not have to use one to post the actual date and time to the Arduino. The addition of a RTC would add unnecessary hardware, weight and complexity to the avionics that is not needed.

One of the fun things about the Arduino is that not only can it be programmed, it can talk to other computer programs. We see that through the use of the Serial Monitor and the Serial Plotter. The use of WiFi, Bluetooth and other communication systems allows the microcontroller to talk to a local computer or network.

The use of these types of communication systems would require additional coding to allow the communication to take place, but there are already libraries and tutorials that can help you develop that code. Once the communication is established and data is flowing, the code can be written to allow displays of telemetry on a laptop. Data could be seen and saved in near real time. You wouldn't have to wait for the recovery of the rocket and retrieval of the microSD card to get started analyzing your data.

Use your imagination and think of what you would like to do – what you want to accomplish. Then set out to do it!

14
Conclusion

This brings to a close our walk through of the Olympus Project. I wanted you to see what we did for several reasons:

- To allow any beginner to observe what I did and why I did it. When I made mistakes, I pointed them out so you won't need to make the same ones. Remember, this is all new to me too!
- I tried to explain why I made certain decisions during the development process, whether that involved electronic hardware, 3D printing, or software development.
- To get folks excited about bringing electronics into their rocketry hobby – especially if they have never had it as part of their process. For me, it has added a whole new level of excitement. It also gives me a feeling of accomplishment to be able to create something like the avionics module.
- I am excited to see what other rocketeers create. Please let me know how you have built your own Olympus Project, or if you used any of the ideas from this project in *your* projects.

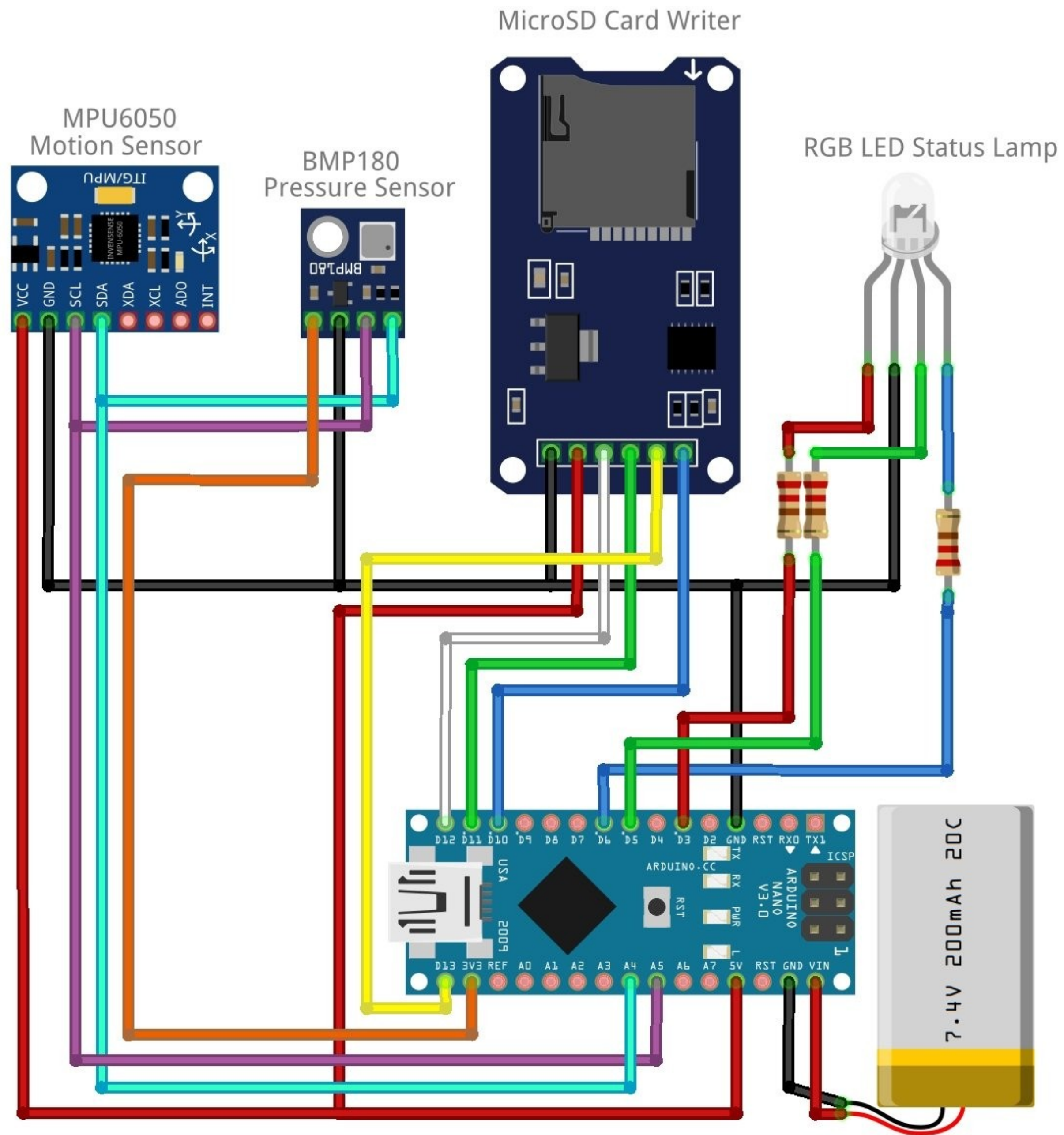
There is just so much that rocketeers of all ages can accomplish with model rocketry. The addition of small, single board computers and microcontrollers has added a new dimension to what you can do with a model rocket. The aspect of 3D printing and readily available CAD programs allow you to design the supporting hardware needed for your project. It is indeed an exciting time for rocketeers of all ages.

We hope that you found this Program Manual useful, and that it helps spur new ideas for new projects in model rocketry.

Appendix

A1
Olympus Avionics System Drawings

These drawings show the final setup of the avionics module. The first drawing is the setup as a breadboard drawing. The second is the schematic drawing.


MicroSD Card Writer
MPU6050
Motion Sensor
BMP180
Pressure Sensor
RGB LED Status Lamp
VCC GND SCL SDA XDA XCL ADO INT
7.4V 200mAh 20C

Battery

MPU6050 Sensor

MicroSD Card Module

MOSI CS Vcc Gnd MISO SCK

Arduino Nano (Rev3.0)

VIN 5V 3V3

RESET AREF A0 A1 A2 A3 A4 A5 A6 A7

D13/SCK D12/MISO D11/MOSI D10 D9 D8 D7 D6 D5 D4 D3 D2 D1/TX D0/RX

GND

BMP180 Barometer Breakout

Vin SCL SDA Gnd

R1 0.25 220Ω

R2 0.25 220Ω

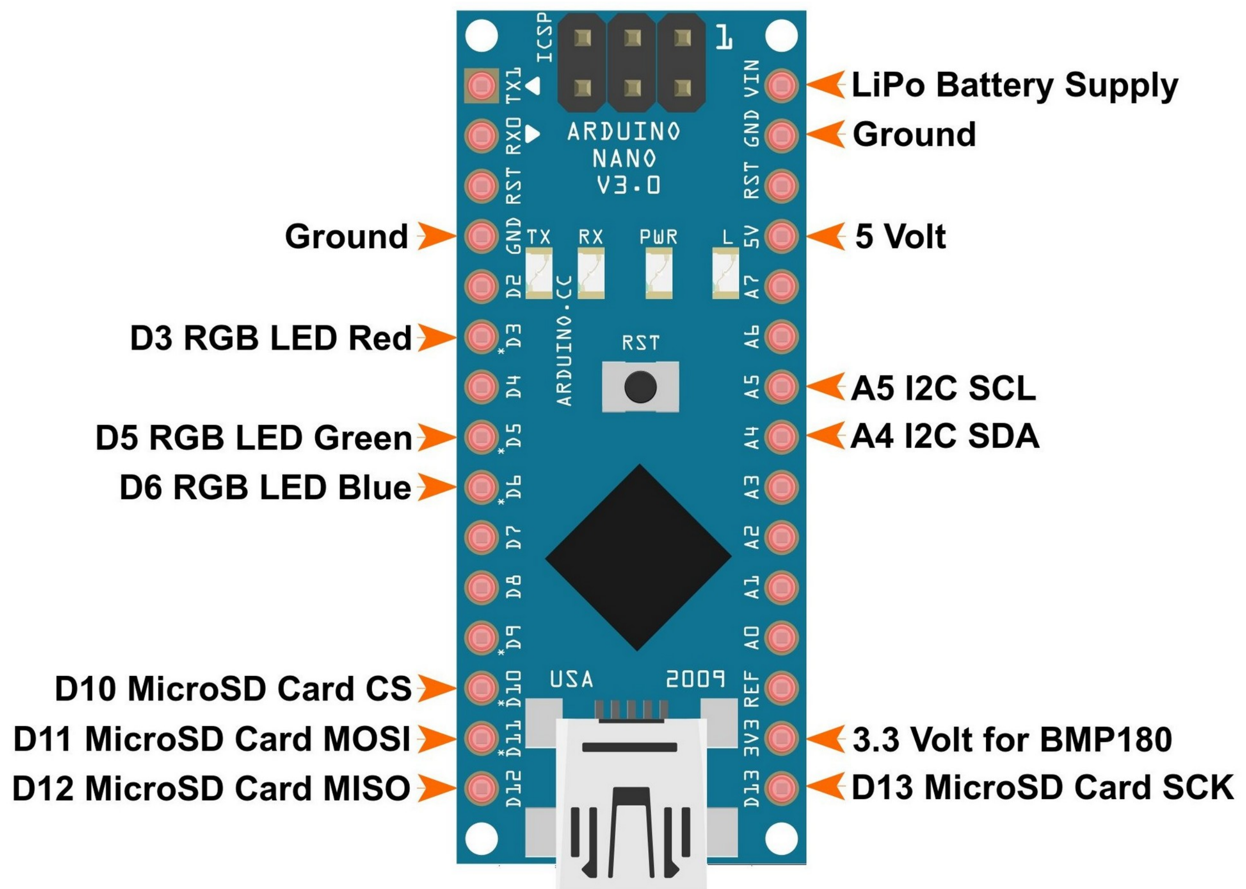
R3 0.25 220Ω

RGB LED

fritzing

A2
Pin Assignments

Nano Pin Assignments



A3
Complete Code Listing

On the following pages is the complete code listing for Version 1.0 of the *Arduino Primary Avionics Module*. The subroutines are saved as individual ".ino" files, with the file name displayed in the tab. The file/tab name is displayed at the beginning of each subroutine.

The complete code can be downloaded from our SourceForge repository

Olympus_Avionics_V1.0.ino Tab

```
/*********************************************************************
 *********************************************************************
 *
 * Project: Olympus Avionics Package
 * Version: 1.0
 * Description: This project is designed to use the A-PAM as the
 *              baseline avionics package. This includes an
 *              Arduino Nano, a microSD card module, and a RGB
 *              LED to show avionics status. The system is powered
 *              by a 7.4-volt LiPo battery.
 *
 *              The Olympus Project adds two sensors to the package
 *               - MPU6050 IMU Sensor
 *               - BMP180 Pressure Sensor
 *
 *              All code listed as Serial is for testing purposes.
 *              This code is commented out for in the flight
 *              version of the code and is marked as "USE FOR
 *              GROUND TESTING ONLY". Uncomment the code to use
 *              for testing.
 *
 * Created: 29 June 2023
 * Updated: 11 August 2023
 * Author: Robert W. Austin
 * (C) Austin Aerospace Education Network
 * License: GPL-3.0
 *
 * ==================================================================
 * Based on the following coding examples:
 *
 *  SdFat SD Card Code
 *  https://github.com/greiman/SdFat
 *
 *  Altduino Altimeter
 *  Code to auto-increment SD card files
 *  https://www.hackster.io/M4K3R_4_M4R5/altduino-c0ac61
 *
```

```
*  BMP180 Pressure Sensor
*  https://learn.sparkfun.com/tutorials/bmp180-barometric-pressure-sensor-hookup-/all
*
*  Royrobotiks Rocket Altimeter
*  https://github.com/royrobotiks/RocketAltimeter
*
*  MPU6050
*  https://www.jarzebski.pl/arduino/czujniki-i-sensory/3-osiowy-zyroskop-i-akcelerometr-mpu6050.html
*  https://github.com/jarzebski/Arduino-MPU6050
*  Examples -> MPU6050 -> MPU6050_gyro_simple
*  Examples -> MPU6050 -> MPU6050_accel_simple
*
* ================================================================
* Pin configuration - for Nano board
*
* RGB LED lamp
*   Red   3 (D3)
*   Green 5 (D5)
*   Blue  6 (D6)
*
* microSD Card
*   CS   10 (D10)
*   SCK  13 (D13)
*   MOSI 11 (D11)
*   MISO 12 (D12)
*   VCC  5V
*   GND  GND
*
* BMP180
*   SDA  A4
*   SCL  A5
*   VCC  3.3V
*   GND  GND
*
* MPU6050
*   SDA  A4
*   SCL  A5
*   VCC  5V
*   GND  GND
*
*****************************************************************
```

```
 *****************************************************************/


/******************************************************************
 ******************************************************************
 *                                                                *
 *                        LIBRARIES                               *
 *                                                                *
 ******************************************************************
 *****************************************************************/

// ================================================================
// library required for I2C communications
   #include <Wire.h>

// ================================================================
// libraries required for the microSD Card Reader/Writer
   #include <SPI.h>
   #include <SdFat.h>  // replaced SD.h

// ================================================================
// library required for the BMP180 Sensor
   #include <SFE_BMP180.h>

// ================================================================
// library required for the MPU6050
   #include <MPU6050.h>

/******************************************************************
 ******************************************************************
 *                                                                *
 *                        DECLARATIONS                            *
 *                                                                *
 ******************************************************************
 *****************************************************************/

// ================================================================
// declarations for the microSD card
   SdFat SD;
   const int SD_FAT_TYPE = 3; // 3 = FAT16/FAT32 and exFAT file system
   const int pinSDcard = 10;  // pin number for CS on Nano board (D10)
```

```
   char fileName[] = "FLTLOG00.CSV";  //file name for flight data

// ================================================================
// declarations for time
   unsigned long timeStamp;

// ================================================================
// Define Pins for RGB LED lamp
   const int RED = 3;
   const int GREEN = 5;
   const int BLUE = 6;

// ================================================================
// declarations required for the BMP-180
// create an SFE_BMP180 object, identified as "baroPressure"
   SFE_BMP180 baroPressure;

// variables for sensor readings
// baroBaseline = baseline barometric pressure at launch site
// currentTemp =  local current temperature
// absolutePress = local barometric pressure
// altitudeCalc = calculated altitude
   double baroBaseline;
   double currentTemp;
   double absolutePress;
   double altitudeCalc;

// ================================================================
// declarations for MPU6050
// create a MPU6050 object
   MPU6050 mpu;

// Pitch, Roll and Yaw values
   double pitch;
   double roll;
   double yaw;
   double accelX;
   double accelY;
   double accelZ;
```

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                            SETUP                              *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setup()
{
  // ================================================================
  // Setup for RGB LED
     pinMode(RED, OUTPUT);
     pinMode(GREEN, OUTPUT);
     pinMode(BLUE, OUTPUT);

  // ================================================================
  // Initialize the serial port.
     // Serial.begin(115200); // USE FOR GROUND TESTING ONLY

  // ================================================================
  // Show Splash Screen
     // splashScreenSerialMonitor(); // USE FOR GROUND TESTING ONLY

  // ================================================================
  // Setup for microSD card
     setupmicroSDCard();

  // ================================================================
  // Setup for BMP180 Pressure Sensor
     setupBMP180();

  // ================================================================
  // Setup for MPU6050 IMU
     setupMPU();
}
```

```
/*******************************************************************
 *******************************************************************
 *                                                                 *
 *                      MAIN PROGRAM LOOP                          *
 *                                                                 *
 *******************************************************************
 *******************************************************************/

void loop()
{
  timeStamp = millis();

  // get pressure readings
     readingsBMP180();

  // calculate altitude
     calculationsBMP180();

  // get roll, pitch yaw sensor readings
     readingsMPU();

  // write data to the microSD card
     writeDataSDCard();
}
```

Calculation_BMP180.ino Tab

```
/****************************************************************
 ****************************************************************
 *                                                              *
 *              PRESSURE SENSOR CALCULATIONS                    *
 *                                                              *
 ****************************************************************
 ****************************************************************/

void calculationsBMP180(void)
{
  // ================================================================
  // Get a new pressure reading:
     absolutePress = readingsBMP180();

  // Show the relative altitude difference between
  // the new reading and the baseline reading:
     altitudeCalc = baroPressure.altitude(absolutePress,baroBaseline);

  /*
  // ================================================================
  // USE FOR GROUND TESTING ONLY
  // print results - for ground testing only
     Serial.print(F("Relative altitude: "));

    if (altitudeCalc >= 0.0) Serial.print(F(" ")); // add a space for positive numbers
      Serial.print(altitudeCalc,2);
      Serial.print(F(" meters, "));

  */
}
```

RGB_LED_Lamp.ino Tab

```
/****************************************************************
 ****************************************************************
 *                                                              *
 *                     RGB LED Lamp Settings                    *
 *                                                              *
 ****************************************************************
 ****************************************************************/

// ==============================================================
// Local variables
   // define variables
   int redValue = 0;
   int greenValue = 0;
   int blueValue = 0;

   // choose a value between 0 and 255 on each variable to change the color.

// ==============================================================
// Red lamp
   void ledRed(void)
    {
      redValue = 255;
      greenValue = 0;
      blueValue = 0;
    }

// ==============================================================
// Green lamp
   void ledGreen(void)
    {
      redValue = 0;
      greenValue = 255;
      blueValue = 0;
    }
```

```
// ================================================================
// Blue lamp
   void ledBlue(void)
    {
      redValue = 0;
      greenValue = 0;
      blueValue = 255;
    }

// ================================================================
// Yellow lamp
   void ledYellow(void)
    {
      redValue = 255;
      greenValue = 50;
      blueValue = 0;
    }


// ================================================================
// Solid lamp
   void ledSteadyLamp(void)
    {
      analogWrite(RED, redValue);
      analogWrite(GREEN, greenValue);
      analogWrite(BLUE, blueValue);
    }

// ================================================================
// Flashing lamp
   void ledFlashingLamp(void)
    {
      while (1)
      {
        analogWrite(RED, redValue);
        analogWrite(GREEN, greenValue);
        analogWrite(BLUE, blueValue);

        delay(100);
```

```
    analogWrite(RED, 0);
    analogWrite(GREEN, 0);
    analogWrite(BLUE, 0);

    delay(100);
  }
}
```

Sensor_BMP180.ino Tab

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                PRESSURE SENSOR READINGS                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

double readingsBMP180(void)
{
// ===============================================================
// local variable to report on status of sensor
   char bmp180Status;

// ===============================================================
// Get a temperature measurement first before perform a pressure
// reading.

  // Start a temperature measurement:
  // If request is successful, the number of ms to wait is returned.
  // If request is unsuccessful, 0 is returned.

  bmp180Status = baroPressure.startTemperature();
  if (bmp180Status != 0)
  {
    // Wait for the measurement to complete:

    delay(bmp180Status);

    // Retrieve the completed temperature measurement:
    // Note that the measurement is stored in the variable currentTemp.
    // Use '&currentTemp' to provide the address of currentTemp to the function.
    // Function returns 1 if successful, 0 if failure.

    bmp180Status = baroPressure.getTemperature(currentTemp);
```

```
    // USE FOR GROUND TESTING ONLY
    // Serial.print(F("Current Temp: "));
    // Serial.println(currentTemp);
    // Serial.print(F("Sensor Status: "));
    // Serial.println(bmp180Status);

// ================================================================
// Start a pressure measurement:
    if (bmp180Status != 0)
    {
      // The parameter is the oversampling setting, from 0 to 3 (highest res, longest wait).
      // If request is successful, the number of ms to wait is returned.
      // If request is unsuccessful, 0 is returned.

      bmp180Status = baroPressure.startPressure(3);
      if (bmp180Status != 0)
      {
        // Wait for the measurement to complete:
        delay(bmp180Status);

        // Retrieve the completed pressure measurement:
        // Note that the measurement is stored in the variable absolutePress.
        // Use '&absolutePress' to provide the address of absolutePress.
        // Note also that the function requires the previous temperature measurement (currentTemp).
        // (If temperature is stable, you can do one temperature measurement for a number of pressure
            measurements.)
        // Function returns 1 if successful, 0 if failure.

        bmp180Status = baroPressure.getPressure(absolutePress,currentTemp);
            /*
            // USE FOR GROUND TESTING ONLY
               Serial.print(F("Current Temp: "));
               Serial.println(currentTemp);
               Serial.print(F("Absolute Pressure: "));
               Serial.println(absolutePress);
            */

        if (bmp180Status != 0)
        {
          return(absolutePress);
        }
```

```
        // else Serial.println(F("error retrieving pressure measurement")); // USE FOR GROUND TESTING ONLY
      }
      // else Serial.println(F("error starting pressure measurement")); // USE FOR GROUND TESTING ONLY
    }
    // else Serial.println(F("error retrieving temperature measurement")); // USE FOR GROUND TESTING ONLY
  }
  // else Serial.println(F("error starting temperature measurement")); // USE FOR GROUND TESTING ONLY
}
```

Sensor_MPU.ino Tab

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                 MPU6050 SENSOR READINGS                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void readingsMPU(void)
{
  // ===============================================================
  // Gyroscope/rotation readings in radians per second

  // Get reading
     Vector normGyro = mpu.readNormalizeGyro();

  // Assign values
     roll = normGyro.XAxis;
     pitch = normGyro.YAxis;
     yaw = normGyro.ZAxis;

  /*
  // USE FOR GROUND TESTING ONLY
     Serial.println("Rotation");
     Serial.print(" Xnorm = ");
     Serial.print(normGyro.XAxis);
     Serial.print(" Ynorm = ");
     Serial.print(normGyro.YAxis);
     Serial.print(" Znorm = ");
     Serial.println(normGyro.ZAxis);
     Serial.println();
  */
```

```
  // ================================================================
  // Acceleration readings in meters per second

    // Get reading
       Vector normAccel = mpu.readNormalizeAccel();

    // Assign values
       accelX = normAccel.XAxis;
       accelY = normAccel.YAxis;
       accelZ = normAccel.ZAxis;

    /*
    // USE FOR GROUND TESTING ONLY
       Serial.println("Acceleration");
       Serial.print(" Xnorm = ");
       Serial.print(normAccel.XAxis);
       Serial.print(" Ynorm = ");
       Serial.print(normAccel.YAxis);
       Serial.print(" Znorm = ");
       Serial.println(normAccel.ZAxis);
       Serial.println();

       delay(100);
    */
}
```

Serial_Monitor_Splash_Screen.ino Tab

```
/******************************************************************
 ******************************************************************
 *                                                                *
 *              SPLASH SCREEN ON SERIAL MONITOR                   *
 *                                                                *
 ******************************************************************
 ******************************************************************/

void splashScreenSerialMonitor(void)
{
  /*
  // ================================================================
  // USE FOR GROUND TESTING ONLY - Display Header on the Serial Monitor
     Serial.println(F("Austin Aerospace Educational Network Project"));
     Serial.println(F("Olympus Project Avionics"));
     Serial.print(F("Version: 1.0.0"));
     Serial.println(F("https://rocketryjournal.wordpress.com"));
     Serial.println();
     Serial.println(F("=================================================="));
     Serial.println(F("Serial Monitor Version for Ground Testing"));
     Serial.println(F("Comment Out Serial Monitor Coding for Use In "));
     Serial.println(F("Actual Flight and Data Collection"));
     Serial.println(F("=================================================="));
     Serial.println();
     Serial.println(F("=================================================="));
     Serial.println(F("Begin Initilization Process"));
     Serial.println();
  */
}
```

Setup_BMP180.ino Tab

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                SETUP BMP180 PRESSURE SENSOR                   *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupBMP180(void)
{
  // =================================================================
  // USE FOR GROUND TESTING ONLY
     // Serial.println(F("Initializing BMP180 pressure sensor. Standby..."));

  // Initialize the sensor
  // (it is important to get calibration values stored on the device).
     if (baroPressure.begin())
     {
      // Sensor initiated
      timeStamp = millis();

      /*
      // USE FOR GROUND TESTING ONLY
      Serial.print(F("BMP180 sensor initialization successful at ")); // for ground testing only
      Serial.println (timeStamp);
      */

      ledGreen();
      ledSteadyLamp();
     }
     else
     {
      // Something went wrong, this is usually a connection problem
         ledBlue();
         ledFlashingLamp();

      /*
```

```
      // USE FOR GROUND TESTING ONLY
      timeStamp = millis();
      Serial.print(F("BMP180 sensor initialization failed at ")); // for ground testing only
      Serial.println(timeStamp);
      Serial.println(F("Check connections and reset system."));
      */
      // Stop program at this point
         while(1);
     }

 // Get the baseline pressure:
        baroBaseline = readingsBMP180();

  /*
  // USE FOR GROUND TESTING ONLY
     Serial.print(F("baseline pressure: "));
     Serial.print(baroBaseline);
     Serial.println(F(" mb"));
     Serial.println(F("=================================================="));
     Serial.println();
  */
}
```

Setup_MPU.ino Tab

```
/****************************************************************
 ****************************************************************
 *                                                              *
 *                  SETUP MPU6050 IMU SENSOR                    *
 *                                                              *
 ****************************************************************
 ****************************************************************/

void setupMPU(void)
{
  // ================================================================
  // Initialize MPU6050
     while(!mpu.begin(MPU6050_SCALE_2000DPS, MPU6050_RANGE_16G))
      {
        // USE FOR GROUND TESTING ONLY
        // Serial.println("Could not find a valid MPU6050 sensor, check wiring!");

        ledYellow();
        ledFlashingLamp();

        // Stop program at this point
        while(1);
      }

  // Calibrate gyroscope. The calibration must be at rest.
     mpu.calibrateGyro();

  // Set threshold sensivty. Default is 3.
     mpu.setThreshold(3);

  // USE FOR GROUND TESTING ONLY
  //   Serial.print(F("MPU Setup & Calibration Complete "));

}
```

Setup_microSD_Card.ino Tab

```
/******************************************************************
 ******************************************************************
 *                                                                *
 *                      SETUP MICRO SD CARD                       *
 *                                                                *
 ******************************************************************
 ******************************************************************/

void setupmicroSDCard(void)
{
  // =================================================================
  // make a string for assembling the data to the log
     String dataString = "";

  // USE FOR GROUND TESTING ONLY
  // Serial.println(F("Initializing Micro SD card. Standby..."));

     // see if the card is present and can be initialized:
     if (!SD.begin(pinSDcard))
     {
        // card did not initialize
           ledRed();
           ledFlashingLamp();

        /*
        // USE FOR GROUND TESTING ONLY
        Serial.println(F("Card initialization failed, or not present. Replace card and reset system."));
        Serial.println(F("=================================================="));
        */

        // stop program at this point:
        while (1);
     }

     // card initialized properly and is ready to start writing data
        ledGreen();
        ledSteadyLamp();
```

```
      /*
      // USE FOR GROUND TESTING ONLY
         Serial.println(F("Micro SD card initialization successful."));
         Serial.println();
      */

    // ================================================================
    // Setup Flight Data Log file name
      // create new file name - routine from altduino.ino
      // this loops through the numbers 00 to 99 to append to name
      // of the CSV file. This prevents overwriting previous data
         for (uint8_t i = 0; i < 100; i++)
         {
           fileName[6] = i/10 + '0';
           fileName[7] = i%10 + '0';
           if (! SD.exists(fileName))
           {
             break;  // leave the loop!
           }
         }

   // write headers to CSV file
      File dataFile = SD.open(fileName, FILE_WRITE);

   // This writes column headers in the CSV file
      dataString = "Time Stamp (milliseconds),Altitude (meters),Absolute Pressure (hPa),Pitch-Y (rad/s),Roll-
X (rad/s),Yaw-Z (rad/s),Acceleration X (m/s^2),Acceleration Y (m/s^2),Acceleration Z (m/s^2),Temperature
(C), ";
      dataFile.println(dataString);
      dataFile.close();

}
```

Write_Data_To_SD_Card.ino Tab

```
/****************************************************************
 ****************************************************************
 *                                                              *
 *                    WRITE DATA TO SD CARD                     *
 *                                                              *
 ****************************************************************
 ****************************************************************/

void writeDataSDCard(void)
{
  // ==================================================================
  // Main Loop for microSD Card

  // Open the file. Note that only one file can be open at a time,
  // so you have to close this one before opening another.

  // CSV stands for "Comma Separated Values". It is a plain text file
  // format where each value is separated by a coma. It can be read by
  // nearly all spreadsheet and database programs.

  // make a string for assembling the data to write to the log
   String writeDataString = "";

  // local variables to write data to sd card
  String fileTimeStamp = "";
  String fileAltitudeCalc = "";
  String fileAbsolutePress = "";
  String filePitch = "";
  String fileRoll = "";
  String fileYaw = "";
  String fileAccelX = "";
  String fileAccelY = "";
  String fileAccelZ = "";
  String fileCurrentTemp;
```

```
  // assign data from sensor readings to variables for CSV string
  fileTimeStamp = String(timeStamp);
  fileAltitudeCalc = String(altitudeCalc);
  fileAbsolutePress = String(absolutePress);
  filePitch = String(pitch);
  fileRoll = String(roll);
  fileYaw = String(yaw);
  fileAccelX = String(accelX);
  fileAccelY = String(accelY);
  fileAccelZ = String(accelZ);
  fileCurrentTemp = String(currentTemp);

  // create data string
  writeDataString = fileTimeStamp + "," + fileAltitudeCalc + "," + fileAbsolutePress + "," + filePitch + ","
+ fileRoll + "," + fileYaw + "," + fileAccelX + "," + fileAccelY + "," + fileAccelZ + "," + fileCurrentTemp;

  // open sd card to write
  File dataFile = SD.open(fileName, FILE_WRITE);

  // attempt to write to sdcard
  if (dataFile) {
    //write to sd card
    dataFile.println(writeDataString);
    dataFile.close();
    ledGreen();
    ledSteadyLamp();
  } else  // if the file isn't open, pop up an error:
  {
    ledBlue();
    ledSteadyLamp();

    // Serial.println(F("Error opening data log...")); // USE FOR GROUND TESTING ONLY

    // reset the SD card to try and write data again
    SD.begin(pinSDcard);
  }

  delay(1);  // Pause for 1 milliseconds.
}
```

A4
Parts Listing

Item	Vendor	URL	Price
ELEGOO Nano Board CH 340/ATmega+328P	Amazon	https://www.amazon.com/gp/product/B0713XK923/	3 for $18.99
HiLetgo Micro SD TF Card Adapter Reader Module 6Pin SPI Interface	Amazon	https://www.amazon.com/gp/product/B07BJ2P6X6	5 for $6.99
7.4V 2S 200mAh 20C LiPO Battery JST Plug and USB Charger	Amazon	https://www.amazon.com/gp/product/B07MW2L96L	3 for $29.99
CHANZON 5mm RGB Multicolor LED Diode LightsThrough-Hole Resistors - 220 ohm 5% 1/4W - Pack of 25	Amazon	https://www.amazon.com/Tricolor-Multicolor-Lighting-Electronics-Components/dp/B01C19ENDM	100 for $8.99
220Ω Resistors (Need 3)Through-Hole Resistors - 220 ohm 5% 1/4W - Pack of 25	Adafruit	https://www.adafruit.com/product/2780	25 for $0.75
Gikfun GY-68 BMP180 Barometric Pressure Temperature Sensor Module	Amazon	https://www.amazon.com/gp/product/B07Q3PQ81R	3 for $12.28
HiLetgo 3pcs GY-521 MPU-6050 MPU6050 3 Axis Accelerometer Gyroscope Module	Amazon	https://www.amazon.com/gp/product/B00LP25V1A	3 for $9.99
HATCHBOX 1.75mm Orange PLA 3D Printer Filament-Orange	Amazon	https://www.amazon.com/gp/product/B00J0EE1D4	1kg reel for $24.99
LitOrange 320PCS M2 Male Female Nylon Hex Spacer Standoff Screw Nut Assorted Assortment Kit	Amazon	https://www.amazon.com/gp/product/B07D78PFQL	Set for $14.88
Black Nylon Machine Screw and Stand-off Set – M2.5 Thread	Adafruit	https://www.adafruit.com/product/3299	Set for $16.95
Black Nylon Machine Screw and Stand-off Set – M3 Thread	Adafruit	https://www.adafruit.com/product/4685	Set for $16.95

A5
The Scientific Method and the Engineering Process

The *Scientific Method* is a cornerstone of research and crosses all scientific disciplines. The *Engineering Method* is similar to the *Scientific Method*, but is different in several important ways. In this section we provide a brief overview of each so that you can determine which method is most appropriate for your project.

It should be noted that the two techniques work hand in hand. A research project will work under the *Scientific Method* as the researchers try to answer a specific question. The engineers who are tasked to create the equipment used in the research project will use the *Engineering Process* to design, build and test the equipment.

Consider the OSIRIS-REx (Origins, Spectral Interpretation, Resource Identification, and Security-Regolith Explorer) Project. The researchers are working to understand whether asteroids colliding with Earth billions of years ago brought water and other key ingredients for life. They will be using the *Scientific Method* to guide the research aspect of this project.

The OSIRIS-Rex spacecraft was developed to help the researchers. The spacecraft was equipped with a number of sensors, including a robotic arm that collected samples from the asteroid Bennu's rocky surface. The engineers building the spacecraft would have used the *Engineering Process* to guide the development and construction of the spacecraft and the various systems housed within it.

The Scientific Method

Chances are you have heard or read about the *Scientific Method*. The *Scientific Method* is a set of procedures that allows you to explore knowledge, answer questions, and be able to provide answers that will stand up to review by others. The scientific method involves the following steps:

1. *Observation that results in a question*
   As you look at the world, what have you seen that raises a question. For example, in model rocketry, maybe you have observed that different models react to different types of parachutes, and that brings up the question of what type of parachute is best suited for competition flying.

2. *Conduct background research*
   Once you have a question you want answered, you need to conduct research into what has been done previously. What can you learn from others who have been researching the same question that you are interested in. This helps you better define the question you want to answer and helps you to avoid mistakes that others have made.

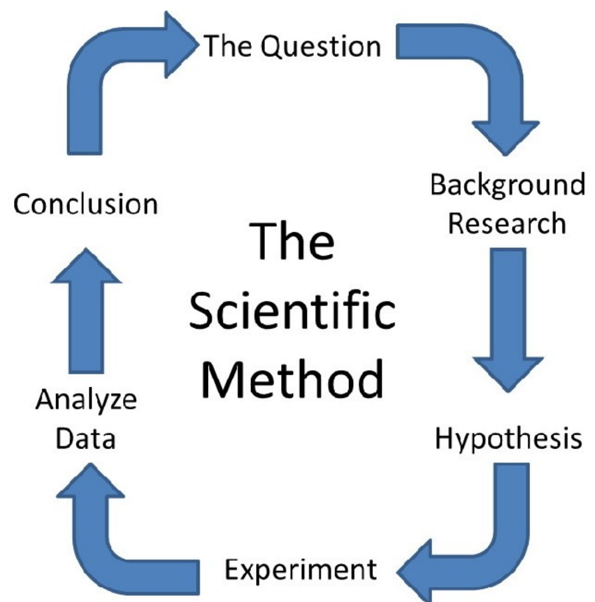
3. *Development of a hypothesis*
   Your hypothesis is a prediction of what you think will happen. This prediction is based on your question and your research.

4. *Conduct an experiment to test the hypothesis*
   During this stage you conduct an experiment of your own design and collect data from the experiment. The more times you can repeat your experiment, the more data you will collect, and the more reliable your data. This leads to a better exploration of the question and more meaningful your results.

5. *Analyze the data collected during the experiment*
   Use the data you have collected to evaluate the results. Determine if your hypothesis was correct or not, and explain the results you obtained – especially if your hypothesis was incorrect.

6. *Use the results to make new hypotheses and predictions*
   Based on what you learned during the experiment and data collection, develop new questions and predictions. This leads you back to step 1 where you start the process over, based on everything you have learned.

The core of the scientific method is used in all of the sciences. Using the scientific method allows others to review your work and your data. It allows others to duplicate your work to either confirm or dispute your results. Throughout the entire scientific process you continue to learn more about the subject, even when your experiment results don't match what you thought would happen. In fact, we often learn significantly more about the subject area when things don't go as expected.



The other important part of the scientific method is that it is an ongoing process. Each new hypothesis, experiment, data collection and analysis leads to another hypothesis, experiment, data collection and analysis. It is a circular process, and not a linear activity.

The Engineering Process

The engineering process differs slightly from the scientific method as it seeks to solve a problem instead of answering a question. Despite this different focus there are still many items in common between the two processes. There are times when the Scientific Method and the Engineering Process intersect and support each other. The engineering process is realized through the design and building of of a device or system.

Consider one of the projects undertaken by NASA, the James Webb Space Telescope (JWST). However, the JWST doesn't just happen. There are numerous engineering problems that must be solved to make the JWST a reality. How do we design a craft that is a large as the JWST fit into a launch vehicle? How do we make the JWST strong enough to withstand the forces of launch and the hostile environment of space? These are all engineering problems that must be solved before the craft is ever launched into space. The Engineering Process was used to work through these issues.

Once the engineering problems are solved, the spacecraft was launched and successfully deployed around the second Sun-Earth Lagrange point (L2), approximately 1½ million kilometers (1 million miles) from Earth. Now the JWST can be used by astronomers around the world to study the stars, to help answer questions about the beginnings of our universe. All of these astronomy projects will use the scientific method to study the universe through the JWST.

The engineers work with the scientists to develop solutions for the research projects. The scientist explains the goals of their project, and what they need to conduct their research. The engineers use this as criteria to begin designing the products needed by the scientists. They two groups will work together to create what is needed.

The steps involved in the engineering process look very similar to those of the scientific method. They include:

1. *Define a need or problem*
   Here you determine what needs to be accomplished. In model rocketry, this could be developing a more dependable helicopter deployment system.

2. *Develop the design criteria*
   What are the constraints of the project? Does it need to be a certain weight, or fit within a specific container? Is there a restriction on the materials that can be used? The design criteria provides the factors you will be working within.

3. *Conduct background research*
   Just like the scientific method, once you understand the problem to be solved and the constraints you must work within, you need to conduct research into what has been done before. Again we are learning from others who have been researching the same or similar problem that you are working on. It also helps you to avoid mistakes that others have made, just like in the scientific method.

4. *Prepare a preliminary design*
   In this stage you would create an initial design based on the design criteria and the research you conducted. With the software available today, you may actually be able to test your design within a computer simulation. Model rocketry programs (such as OpenRocket) offer flight simulations of various designs created within the program. They can test for stability, typical flight, altitude attained, duration, etc.

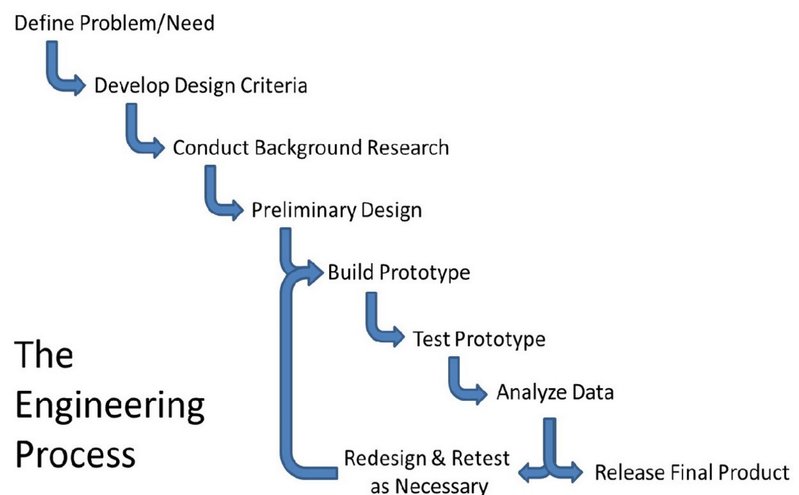
5. *Build the prototype*
   Once the design is finalized, a prototype is created to test the design. The prototype may only be a small model that can be tested under various conditions (such as in a wind tunnel) or it could be a full scale version of the design.

6. *Test the prototype*
   This is the equivalent of the experiment in the scientific method. While computer simulations are very good, there is still the need to test the design under real world conditions. As with the scientific method, data is collected during the testing phase to be analyzed later.

7. *Analyze the Data*
   The data gathered from the prototype testing is analyzed. Based on the data analysis, did the prototype perform as expected, or according to specification? Did it exceed design expectations, or did it fall short? The analysis of the data will help guide you through this process. There is also the question of do you have enough data, or do you need to do some more testing to obtain the data that you need.

8. *Retest and redesign as necessary*
   The prototype works as expected if it meets the design goals as set out in the beginning of this process, and you have sufficient data to support this conclusion. Then you can move on to production. However, it is also possible that the prototype didn't work as expected, and the analysis of the data indicates certain changes need to be made. When this happens, you need to review the data and and update the design the prototype. Testing of the new prototype is done to determine if the changes provide the desired result.

One of the significant differences between the *Engineering Process* and the *Scientific Method* is that the *Engineering Process* is generally not an ongoing process. With the engineering process you are seeking to solve a problem, and once the problem has been solved and the project works as expected, you are done. It is more a linear process, and not a circular activity.

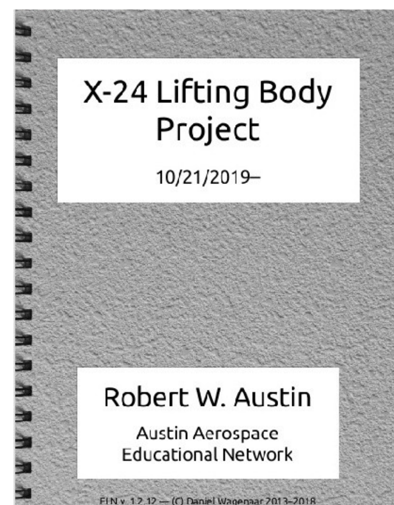


A6 Engineering Notebook

In chapter 2 we mentioned the need to keep an engineering notebook. In this appendix we are going to provide some general guidelines for creating and maintaining an engineering notebook.

Purpose

The primary purpose of an engineering notebook is to document your project. The notebook starts with defining the need or the problem you intend to solve. It documents the research you performed, the design process, the building and testing process, your thoughts, ideas, and notes. It should include the decisions you have made about the project and why. You should sketch (or draw) your ideas in the notebook. You put your data in the notebook. The engineering notebook should be like a diary that covers every aspect of the project.



It should also be neat and organized. It should be thorough. Your notebook should be detailed and complete, so that anyone could come in at any time during the project, and your notebook would allow them to continue the project without any additional information.

An engineering notebook can also be a legal document. It can help prove the origin of your idea and how the idea was turned into a workable solution. This can be extremely important if the project you are working on leads to submission of a patent or a copyright claim. An engineer's notebook is considered to be Proprietary Information.

What to Track

So what should be included in your notebook? Everything. If it is not documented, it did not happen. If you write it the next day, it did not happen. The following list should help you determine what to include.

- Identifying the problem
  - Research on the problem
- Expert input
  - Interview information (who was contacted, why, when, and what was discussed or learned)
  - Include their names, positions, contact info and details of conversations
- Design Process

- Brainstorming
  - Sketches with labels and descriptions
  - Digital technical drawings
  - Pictures
- Your daily thoughts and ideas
  - These should be stated or presented in a way that can be clearly understood
- Description of your current status
  - List of present team members
  - List of objectives for the day, and objectives for the week
  - Important dates for concepts, calculations, test results, improvements, and project completion.
  - Briefly explain the importance of any performed activities.
- Work session and meeting summaries
- Research findings
  - Test procedures, results, and conclusions
  - Calculations
  - Reference information for all sources
- Design modifications
  - Descriptions of changes
  - Rationales of changes

This notebook is a daily record of your project , you should write everything down. If you make a mistake, cross out the word(s), sketch, note, etc. Use a single line to cross out the error along with the initial of the person who crossed out the item. Do not erase anything. The notebook should help you remember any issues you have previously had during the project. This can help you avoid those issues in the future resulting in an improved design. Keeping notebooks on file of previous projects can be a source for research and problem solving when similar issues occur in newer projects.

Notebook Standards

When keeping an engineering notebook, you should adhere to the local standards as required by your organization. If your organization has not formalized any standards, here are some basic guidelines that should be followed (and can serve as a starting point for developing your organization's own standards).

- Write legibly.
- The Notebook must be bound. Pages cannot be added or removed without disrupting the binding.
- Pages are sequentially numbered in permanent ink at the top corner of each page.
- All work is in pen.
- Clearly indicate the date before or above each new entry.
  - Always be consistent, accurate, and chronological when recording entries in your notebook.
- All figures and calculations should be clearly labeled and printed in permanent ink.
- Entries start at the top of the page, working left-to-right and top-to-bottom.
- Entries should include enough information so someone else could successfully duplicate your work.
- Use complete sentences.

- Label figures and sketches. Keep sketches up-to-date.
- Inserted items are permanently attached
  - Glue is preferred
  - No loose leaf items
- Do not leave open or unused space.
- Simply draw lines through blank or unused space as entries are made.
  - Never leave blank spaces - simply "X" out any blank spots.
- Markers that can bleed through the paper are not to be used. (Permanent ink entries are preferred)
- Permanently attach inserted items with tape or a glue stick. Loose items do not belong in the notebook.
- No data recording pages are to be removed from the notebook for any reason.
- Sign and date each completed entry page before you begin the next page.
  - International Standard ISO 8601 specifies numeric representations of date and time.
  - The international standard date notation is "YYYY-MM-DD" where YYYY is the year in the usual Gregorian calendar, MM is the month of the year between 01 (January) and 12 (December), and DD is the day of the month between 01 and 31.
    - Year-Month-Day Example: 2019 October 15
- A colleague should review the entries on each page and sign/date as the witness in the designated boxes.
  - Excluding the witness, no one other than yourself should write in your notebook.
- Your notebook should be secured in a safe location when not in use.
- When the notebook is full, begin a new one that picks up where the other ended.

Be Organized: Use a systematic approach – be brief, neat, structured, sequential, and consistent.

Electronic Notebooks

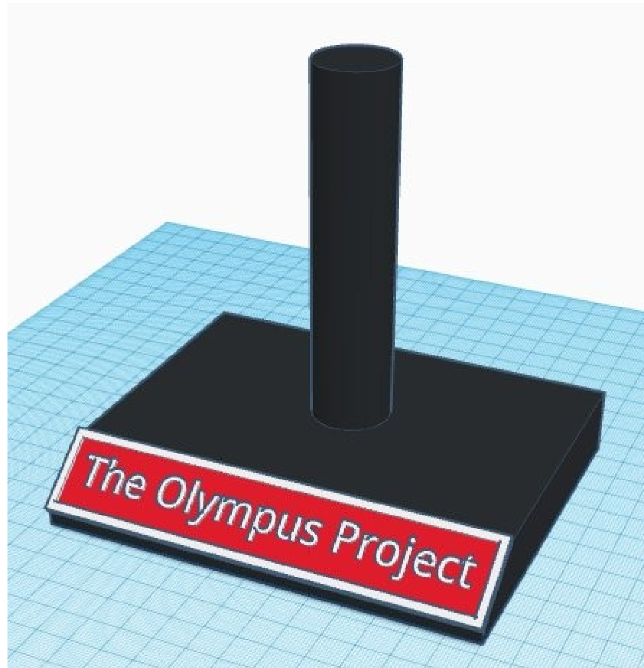
Most of the information in this appendix has dealt with paper based engineering notebooks. Today, there are a number of electronic notebooks that a person can use instead of the traditional paper notebook. As with most things, there are advantages and disadvantages to both.

Electronic notebooks are great at handling text based inputs. A number of people find it as easy to type in their notes, thoughts, ideas, etc into an electronic notebook as opposed to writing it out by hand in a paper notebook. Electronic notebooks are also good at attaching items to the notebook. This may include research documents, web links, emails, etc. By adding tags to your entries, the electronic notebook can help you organize your notes when it comes time to compile and write the research paper.

Electronic notebooks are not very good at adding sketches. Some laptops and tablets come with a stylus that may help with creating sketches on a screen. They are also not adept to writing out complex equations. Again, those that make use of a stylus can help bridge this gap, but it is not yet as easy as writing them on paper.

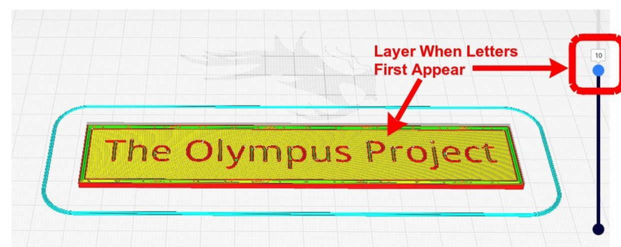
A7
3D Printed Display Stand

Although it is not required for the project, we did design a 3D printed desktop stand. The stand is divided into two parts, with the base of the stand consisting of the bulk of the print and the name plate that is glued onto the front.



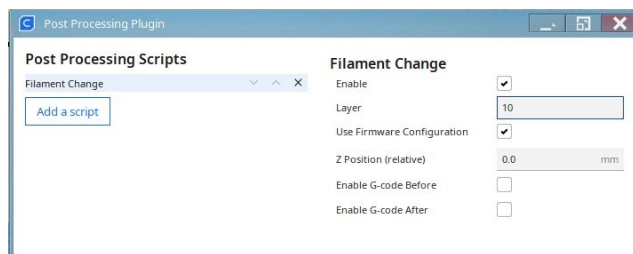
When printing the name plate we used Cura as our slicing program and used the "Filament Change" post processing script. This allows us to print the base of the name plate in one color (we used red), stop the print and change the filament to a different color (we used white), and then continue printing.

To perform two color printing on the name, begin by importing the plate into Cura and slice it as you normally would. When it is done slicing, go into preview and lower the layers bar until the first layer with letters appears. In the graphic on the right, you can see where the letters appear on layer 10. Remember this number as you will need it when we add the extension.



We now want to add the Filament Change script to our G-code. To access this script, start at the main menu in Cura, click on Extensions > Post Processing > Modify G-code. A dialog box will appear. Click on the "Add a script" button. A listing of scripts will be displayed. Select "Filament Change" from the list.

Enter the layer number where you want the color change to take place. Next, click on the option "Use Firmware Configuration". Now click on the "Close" option in the bottom right corner of the dialog box.



When you return back to the main screen you will notice that there is a red circle with the number 1 next to the Slice button. This indicates that the Filament Change script is now active. To add it to your G-code, click on the “Slice” button again. Cura will reslice the name plate and this time will add the filament change to the G-code. Save this G-code file as you normally would and load the file into your printer.

Start printing the name plate just like any other 3D print. When the printer gets to the layer you specified, it will stop printing and move to a neutral parking position. The stepper motors are locked in place to allow you to remove the old filament and insert the new filament color. Hit the continue button on the printer and the printer will heat up the new filament and purge the old filament from the print head. It will then continue printing using the new filament color.

I used plastic model glue to attach the name plate to the base.

A8 References

- Angra, Aakanksha and Greenwod, Jannetta. "*How to Organize a Lab Notebook.*" Georgia Institute of Technology. 2012.
- Barber, Trip. "*Model Rocketry In Science Fairs.*" National Association of Rocketry, Marion, Iowa. March, 2011.
- Cawley, Bernard. "*Altimeter Venting.*" Apogee Peak of Flight Newsletter Issue 543. March 16, 2021 https://www.apogeerockets.com/education/downloads/Newsletter543.pdf
- Cawley, Bernard"*Altimeter Venting - Do I REALLY Have To? OR: If you fly nine altimeters in one rocket, do you really know how high it went?*" NARAM-61 R&D report. 2019
- Cawley, Bernard. "*How High Does It Go? Electronic Altimeters.*" NAR Member Guidebook, Page 40-45. January 2022 edition.
- Haskins, Leon. "*Engineering Design Notebook Template.*" MESA, Tacoma/South Puget Sound, Washington. October, 2013.
- Lafferty, Kenneth. "*Teacher's Guide to Science Projects*." Version 1. Hess Family Charitable Foundation. October 2007.
- Martin-Myers, Karen; Stephen, Mary Ellen; Young, Mary. "*Student Guide: How to Do a Science Fair Project*." Massachusetts State Science & Engineering Fair. March, 2012.
- Michielssen, Chris. "*Altimeter Hole Sizes? TIP.*" Model Rocket Building blog. December 29, 2014. https://modelrocketbuilding.blogspot.com/2014/12/altimeter-hole-sizes-tip.html
- Patterson, Steve. "*Engineering Design Notebooks*." University of North Carolina, Charlotte. Revision B. September 29, 2009.
- Ring, Chad. "*To Vent, Or Not To Vent, That Is The Question.*" NARAM-59 R&D report. July 2017.
- Ring, Chad. "*Vented Vs Non-Vented: Is The Adrel Immune*?" NARAM-60 R&D report. July 2018.
- Salem, Dr. Mohamed Labib. "*Keeping a Laboratory Notebook*." Tanta University, Egypt. March 2011.
- Stroick, Gary. "*HPR Research - Static Port Holes from Nescience to Science*" Off We Go Rocketry. March 14, 2020. https://www.offwegorocketry.com/userfiles/file/Documents/Static%20Port%20Holes.pdf
- Sumner, Anna and Shabram, David. "*Engineering Notebook.*" Westside Middle School. Omaha, Nebraska and the Peter Kiewit Institute, University of Nebraska.
- Thang, Tran. "*A Student's Guide to the Science Fair.*" NIWA Wellington Regional Science Fair. Wellington, New Zealand. 2009.
- Van Milligan, Tim. "*How To Select An Altimeter.*" Apogee Peak of Flight Newsletter, Issue 445. June 13, 2017. https://www.apogeerockets.com/education/downloads/Newsletter445.pdf
- Van Milligan. Tim. "*Pressure Relief Holes.*" Apogee Peak of Flight Newsletter, Issue 68. https://www.apogeerockets.com/education/downloads/Newsletter68.pdf

- Walker, David. "*Sizing Static Ports For Altimeter Sampling*." Cambridge University Spaceflight. September 2016. https://www.cusf.co.uk/2016/09/sizing-static-ports-of-altimeter-sampling
- Wolf, Dan. "*The Care and Feeding of Altimeters for NAR Competition*." Sport Rocketry Magazine, January/February 2019.
- "*Altimeters*". American Rocketry Challenge 2024 Team Handbook, Appendix 6. Version 24.1, July 20, 2023. https://www.rocketcontest.org
- "*Engineering Notebook*." PowerPoint Presentation. Project Lead the Way. Indianapolis, Indiana. 2012.
- "*Gateway to Discovery: A Science Fair Guide to Get You Started*." Home Science Tools. St Billings, Montanna. 2007.
- "*Rocket Vent Hole Size Calculator*". FreeCAD Wiki. https://wiki.freecad.org/Rocket_Vent_Hole_Size_Calculator
- "*Science Fair Experimental Research Project-Student Guidebook*." General Ray Davis Middle School, Stockbridge, GA. 2014.
- "*Science Fair Project Guidebook*." South Carolina Department of Health and Environmental Control. July 2013.
- "*The Scientific Method*." Google Science Fair. 2012.

Arduino Programming

- SdFat SD Card Code
  https://github.com/greiman/SdFat
- Altduino Altimeter
  Code to auto-increment SD card files
  https://www.hackster.io/M4K3R_4_M4R5/altduino-c0ac61
- BMP180 Pressure Sensor
  https://learn.sparkfun.com/tutorials/bmp180-barometric-pressure-sensor-hookup-/all
- Royrobotiks Rocket Altimeter
  https://github.com/royrobotiks/RocketAltimeter
- MPU6050
  https://www.jarzebski.pl/arduino/czujniki-i-sensory/3-osiowy-zyroskop-i-akcelerometr-mpu6050.html
  https://github.com/jarzebski/Arduino-MPU6050
  Examples -> MPU6050 -> MPU6050_gyro_simple
  Examples -> MPU6050 -> MPU6050_accel_simple

If You Enjoy Rocketry, Consider Joining the NAR

If you enjoy model rocketry and projects such as the Arduino Launch Control System, Project:Icarus, The Dyna-Soar and others, then consider joining the National Association of Rocketry (NAR). The NAR is all about having fun and learning more with and about model rockets. It is the oldest and largest sport rocketry organization in the world. Since 1957, over 80,000 serious sport rocket modelers have joined the NAR to take advantage of the fun and excitement of organized rocketry.

The NAR is your gateway to rocket launches, clubs, contests, and more. Members receive the bi-monthly magazine “Sport Rocketry” and the digital NAR Member Guidebook—a 290 page how-to book on all aspects of rocketry. Members are granted access to the “Member Resources” website which includes NAR technical reports, high-power certification, and more. Finally each member of the NAR is cover by $5 million rocket flight liability insurance.

For more information, visit their web site at https://www.nar.org/