Project: Vulcan

A Model Rocketry Test Stand Utilizing the Arduino Mega2560 Microcontroller

Robert W. Austin
NAR 47533

AAEN

This Project Manual was created using LibreOffice Writer Version 24.8. 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

The cover drawing is a graphical representation of the Vulcan statue located in Birmingham, Alabama. You can find additional information on Vulcan at https://en.wikipedia.org/wiki/Vulcan_statue

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 there. 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

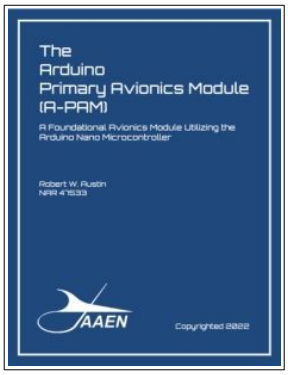
Other Project Manuals Available from AAEN

The Arduino Launch Control System

Our first Project 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.

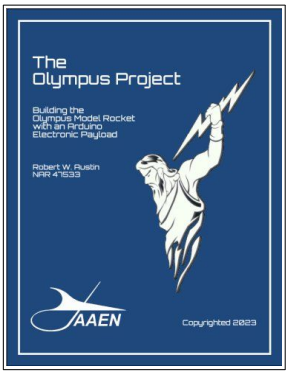
Rocketry Research Assistant-Part 1

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 and 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.

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 off the shelf rocketry kits and electronic components that are readily available.

The goal is to develop a system that can be used by high school students. It involves proven rocket designs that can be flown on school yards. It needs to be small, much smaller than the high power rockets that you typically see. It meant fitting everything into a body tube with a diameter of about 42mm (1.65 inches). This allows the use of "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.

132 pages. Adobe Acrobat (pdf) format.

Table of Contents

01 Introduction....15
What is a Test Stand?....15
The Basic Layout of the Project....15
Looking at What Already Exists....16
Before we start our project....17
We Are All Learning....17
02 Rocket Motor Basics....18
Typical Engine Sizes....18
The Engine Coding System....19
Total Impulse....19
Average Thrust....20
Delay Time....20
Thrust Curves....20
Rocket Motor Design....23
03 Rocket Motor Test Stands....25
Our Test Stand....26
04 Designing the Test Stand....28
Research....28
Initial Designs....29
05 Components....32
The Test Stand Electrical Components....32
Elegoo Mega2560....32
Load Cell & Amplifier....33
MicroSD Card Module....33
LEDs....33
Piezo Buzzer....34
BME280 Environmental Sensor....34
Real Time Clock....34
Firing Relay....34
Resistors....34
AA Battery Holder....34

The Remote Head Electrical Components....35
LED Clock....35
LCD Screen....35
Function Buttons....35
DB15 Connectors and Cable....36
Other Hardware....36
Tools and Supplies....37
06 Writing Code....38
Documenting the Code....38
Layout....38
Title Block....39
Header Blocks....41
Subsection Dividers....41
Comments....42
07 Libraries, Declarations, and Setup Code....43
Libraries....43
Declarations....44
void setup()....45
Using Functions....45
Initializing Serial Port....46
Splash Screen....47
Setup Display Screens....48
LED & Buzzer Pins....48
Setting Up the Real Time Clock (RTC)....48
Checking and Adjusting the Date & Time....48
Setup the MicroSD Card Module....49
Initialization of the BME280 Weather Sensor....50
Testing the HX711 Load Cell Amplifier....51
Initialization of the Fire Control System....51
All Systems Initialized....51
08 The Loop() Function....52
void loop() function....52
09 LEDs, Buzzers, Clocks and Displays....54
The RGB_LED_Lamp_Settings Tab....54

RGB Colors....54
LED Lamp States....54
10 Writing Data to a SD Card....56
Information Log....56
Opening and Writing the File....56
Error Reporting....56
Motor Data Log....57
System Log....58
11 Motor Prep Sequences....59
Getting Started....59
Test Preparation....59
Motor Preparation Information....59
Motor Casing Information....61
Impulse Information....62
Average Thrust....64
Delay Time....64
Propellant Data....64
Ignition Time....65
Calculating Data Collection Time....65
12 Motor Load Sequence....67
Getting Started....67
Motor Load Checklist Function....67
Motor Mount Checklist Function....68
Load Cell Calibration Decision....68
Clear Test Area....69
Setting Up The Load Cell....69
13 Fire Sequence....70
Transition to the Fire Sequence....70
Fire Control Sequence....70
Waiting for the Fire Button to be Pressed....70
Fire Button Pressed....71
ABORT....72
Abort Recycle....72
Shutdown Period....72

Wait Period....72
All Clear....73
Post Test Data....74
Followup Testing....74
14 Test Data....75
Data Collection....75
Data Collection Time....76
Code Review Conclusion....76
15 Building the Test Stand Base....77
Base....78
Warning Light Housings....78
Wiring the Lamps....79
Adjustable Legs....79
Load Cell....80
Load Cell Location & installation....80
Motor Mount....81
Base Complete....82
16 Building the Motor Mounts....83
Printed Motor Mount Centering Rings....83
Assembling the Motor Mounts....84
17 Designing and Building the Electronics Housing....86
A Step at a Time....86
Individual Component Mounts....87
The Jigsaw Puzzle Design....87
Back to Tinkercad....88
3D Printing....89
Installing the Electrical Components....90
Wiring....90
Battery Pack and Continuity Lamp....91
Warning Light Wiring....92
Platform Levels....92
18 Building the Remote Head....93
Version 1....93
Versions 2 and 3....93

Version 4 94
Printing the Remote Head 94
Wiring the Remote Head 95
The DB15 Connector 95
The Cover 95
Connecting the Cover and Base 96
19 Assembling the Test Stand 97
Setup Area 97
Connections 97
Motors and Motor Mounts 98
Accessories 98
20 Conducting a Motor Test 99
Mega2560 Boot 99
Data Entry Process 99
Pre-fire Process 101
Motor Loading Checklist 101
Motor Mount Checklist 101
Test Stand Calibration 101
Clear Test Stand Area 102
Motor Test Fire Process 102
Abort 102
Motor Firing 103
Post Motor Test Firing 103
CATO 103
Case Mass 103
Comments 103
Test Complete 103
21 Tactics to Improve the Test Stand 105
Hardware Updates 105
Software Updates and Changes 105
Make The Project Yours 106
22 Conclusion 107
A1 System Drawings 110
A2 MEGA 2560 Pin Assignments 113

A3 Complete Code Listing.....114
Test_Stand_V1.0.ino.....114
Buzzer_Tones.ino.....120
Calibrate_Load_Cell.ino.....121
Clock_LED.ino.....125
Fire_Abort_Recycle.ino.....125
Fire_Abort_Sequence.ino.....126
Fire_Control_Sequence.ino.....127
Fire_Data_Collection.ino.....130
Fire_Shutdown_Period.ino.....132
Fire_Weather.ino.....134
Initialization_Pass.ino.....135
LCD_Date_Time.ino.....136
Motor_Load_Sequence.ino.....137
Motor_Mount_Sequence.ino.....138
Motor_Prep_And_Test.ino.....139
Motor_Prep_Avg_Thrust.ino.....140
Motor_Prep_Calc_Time.ino.....141
Motor_Prep_Casing.ino.....141
Motor_Prep_Delay_Time.ino.....144
Motor_Prep_Ignition_Time.ino.....145
Motor_Prep_Info.ino.....145
Motor_Prep_Propellant.ino.....148
Motor_Prep_Scale.ino.....150
Motor_Prep_Total_Impulse.ino.....150
Motor_Recalibration_Sequence.ino.....152
Motor_Test_Clear_Area.ino.....153
Motor_Test_Tare.ino.....154
Post_Test_Data_Entry.ino.....155
RGB_LED_Lamp_Settings.ino.....157
Sensor_Data_BME280.ino.....159
Serial_Monitor_Date_Time.ino.....159
Serial_Monitor_Splash_Screen.ino.....159
Setup_BME280_Sensor.ino.....160

Setup_Date_Time_Check.ino....161
Setup_Date_Time_Entry.ino....163
Setup_Fire_Control_System.ino....166
Setup_HX711.ino....167
Setup_LCD_I2C.ino....168
Setup_LED_Display.ino....169
Setup_MicroSD_Card.ino....169
Setup_Real_Time_Clock.ino....172
Strobe_LED_Bulb.ino....173
Write_Info_Data_To_SD_Card.ino....173
Write_Motor_Data_To_SD_Card.ino....174
Write_Sys_Data_To_SD_Card.ino....175
Compilation Notes - Arduino Mega2560....176
A4 Parts Listing....177
Electronics....177
A5 References....180
A6 Project Links....182

01 Introduction

Our first project, the Arduino Launch Control System, was a part of the ground support equipment needed for launching model rockets. The next two projects focused on rocket avionics systems, with Project: Icarus looking at temperature sensors and The Olympus Project working with altimeter and roll rate/g-force sensors. All of these projects made use of the Arduino Nano microcontroller.

This project will be a bit different. First, we are going back to the area of ground support equipment, this time building a model rocket motor test stand. However, we will not be using the Arduino Nano but instead will build it around the Arduino Mega2560 board.

What is a Test Stand?

So what is a model rocket motor test stand? Well, at its core it is a measuring device. It simply measures how much thrust a motor produces. It does this through the use of a load cell – the same type of electronic device that is in your weight scale. With this test stand the motor is placed in a downward position (with the motor exhaust coming out of the top). When the motor is ignited it presses down on the load-cell. That creates a small change in the electrical current and that change is measured and recorded. The load-cell does the same thing when you step on a scale to weigh yourself. There is a change in the current and a calculation is performed to display your weight.

For this project our load-cell can measure up to 5 kilograms of force. This will be more than adequate for our Estes black powder A-E motors. For more powerful motors you will need a larger capacity load-cell.

The Basic Layout of the Project

The project uses an Arduino Mega2560 board to process everything, and there is a lot going on. This Test Stand does a lot more than just provide the data to create a thrust curve. Since I always try to look at these projects from an educational/research perspective, we have added some additional components into the system. These items were added based on previous test stand projects and research papers that we examined. We also looked at the National Association of Rocketry's (NAR) Standards and Testing (S&T) process as outlined on the NAR web site (https://www.nar.org/nar-motor-testing). The result is the following items included in this project:

- Environmental Sensor
  Detects temperature, humidity and barometric pressure
- Data Storage
  Three separate data files are stored on a MicroSD card. They include;
  - an Information Log that records the motor and test location information

  - a System Log that records the status of the test stand system, including time stamps when various activities occurred.
  - a Motor Log that provides a readout of the load cell in Newtons, total impulse in Newtons, and each entry is timed stamped.
- Use of the Serial Monitor to both monitor the test stand systems and to input log information.
- The software begins data collection on ignition and will continue recording data until one second past the end of the delay charge.

We also wanted the system to have some "coolness" to it. In that respect, it includes the following:

- 7-segment LED clock
- 5-second countdown
- Warning lamps and buzzer
- LCD screen with general messaging

Looking at What Already Exists

As noted above I looked at the NAR S&T procedures and requirements as I developed this test stand. There is a major differences between the two stands and that involves the number of samples the test stand processes per second. This stand uses a common load-cell I purchased off of Amazon, with an amplifier from Sparkfun. This combination can record 80 samples per second. The NAR test stand can perform 1000 samples per second (or more) for motors with a burn time of less than 1-second, and at least 500 samples per second for all other motors[1].

Another difference is that the test stands used by the NAR can place the motor in a horizontal position or a vertical position. My test stand has the motor in a vertical position thrusting downward. Later I'll explain why the position makes a difference.

So is our test stand worthless? Absolutely not! The NAR Test Stand is used to validate and certify motors. As certification is the primary purpose, all the motors are tested under identical conditions. The S&T manual states, "Motors, fuel grains, oxidizer tanks etc. shall be kept within the environmental conditions specified by NFPA 1125 for testing. For motors, these conditions are a temperature of 20°C + 5°C."[2]

It is the certified motors that we intend to test in our stand. The 80 samples per second collection is more than adequate to help determine accurate thrust curves for simulations based on local conditions. If conducting research our test stand can be used to see how various temperature and humidity changes affect the motors. There are a number of ways you can use this stand to help create a more accurate estimate of exactly how the motor will perform, resulting in more accurate simulations. This additional data can help you obtain better performance from your rocket.

1 NAR Standards and Testing Committee Motor Testing Manual Version 1.5. July 1, 2011. Section 8.5 "Test Stand Requirements", item 8.5.2.

2 NAR Standards and Testing Committee Motor Testing Manual Version 1.5. July 1, 2011. Section 8.3 "Motor Testing Process", item 8.3.3.

Finally, this project builds on two of our previous projects. Many of the components and code from the *Arduino Launch Control System* were either utilized in the test stand or were upgraded from their original use as outlined in the LCS Project Manual. From *Project: Icarus* I reused the code for the RGB LED bulbs and the MicroSD card module.

Before we start our project

Before I get into the building of the Test Stand, I want to provide you with some basic information. I'll discuss the basics of model rocket motors and how they work, what a thrust curve is and how it is created, and a bit more detail about rocket motor test stands and how they are used for sounding rockets and space launch vehicles.

This manual does not require the user to have read or created any of the previous projects. Each step of the project will be explained so that the user can successfully create the test stand project.

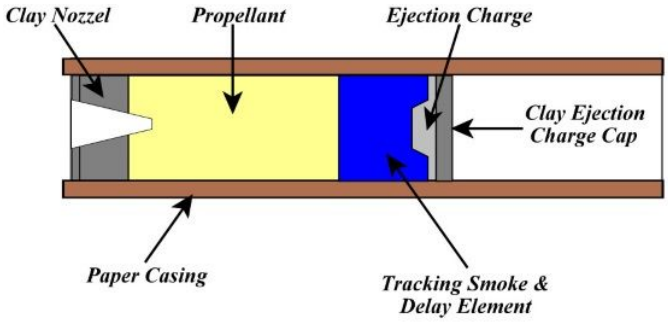
Because of this there is some overlap between this project and our other project manuals that I have released. Therefore if you have built or read our other manuals, you may find that some material is duplicated. When you encounter these areas, you can simply ignore the duplication as long as you feel confident in the information being presented.

We Are All Learning

If you have read any of our previous Project Manuals, you will know that I am learning this "new-to-me" aspect of the hobby called electronics. I am creating these manuals so that you can learn from me, as I make mistakes and then add changes during the development of the test stand. The more experienced electronic experts reading this may cringe at some of things I am doing, as they know better, easier or more robust ways of accomplishing the same goal. That’s fine, because the goal of these Project Manuals is *not* to create a perfect project, but rather to *learn* new skills and increase our knowledge in the topic of electronics. We want to have a working project when we are done, but we also want this to be fun and enjoyable. If its not, you won’t learn anything.

02
Rocket Motor Basics

The basic black powder model rocket engine is composed of five sections. These are identified in the graphic below;



The typical model rocket motor is made of a paper casing with clay caps on each end of the motor. The rear of the motor has a nozzle molded from clay. The nozzle opens into the propellant. The electrical igniter is placed into the nozzle and pushed forward until it touches the propellant.

The most popular and inexpensive propellant on the market today is black powder. When this propellant is ignited, it burns from the back of the engine and moves to the front, expelling the hot gases out through the nozzle and thrusting the model into the air.

The next section in the motor is the tracking smoke delay element. When the propellant has finished burning, the model is still moving rapidly through the air. The delay element allows the model to continue gaining altitude and allows the aerodynamic force of drag to slow the model down. During this time, tracking smoke is released through the nozzle, making it easier to see exactly where the model is located in the sky.

When the tracking smoke has been totally used up, a small charge called an ejection charge is fired, pushing off the clay end cap. The hot gases from this charge enter the body tube of the model and expand rapidly. This rapid expansion of the ejection charge gases pushes off the nose cone of the model, deploying the recovery device.

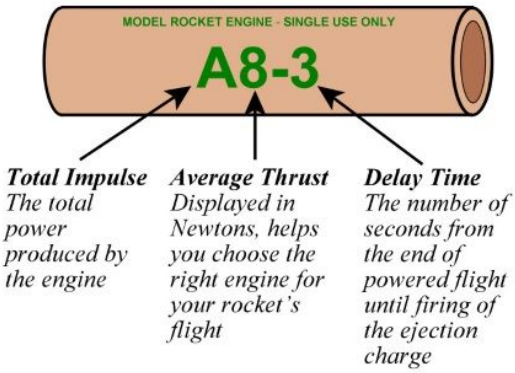
Typical Engine Sizes

Solid model rocket motors come in a number of sizes. As a general rule the larger the diameter/length of the motor, the more thrust the rocket will be able to produce. The chart below shows some common diameters, lengths and general power range.

Diameter	Length	Type
13mm	44mm	Mini Motors
18mm	70mm	Standard Motors
24mm	70mm and 95mm	D and E Motors
29mm and 38mm	114mm +	Mid-power Motors

The Engine Coding System

All model rocket engines, regardless of the manufacturer, utilize the same coding system. This allows you to know the basic performance capabilities of the engine. That coding system is shown in the graphic on the right.



Total Impulse

The following chart shows the total impulse classification for each letter code.

Code	Newton-Seconds
1/4A	0.000 - 0.625
1/2A	0.625 – 1.250
A	1.250 – 2.500
B	2.250 – 5.000
C	5.000 – 10.000
D	10.000 – 20.000
E	20.000 – 40.000
F	40.000 - 80.000
G	80.000 – 160.000

You will notice two things in this chart:

- Each time you increase the letter, the maximum total impulse limit of the engine doubles. This means that a 'C' engine has twice the total impulse of a 'B' engine; a 'B' engine has twice the total impulse of an 'A' engine, and so on.
- The total impulse is given as a range. Looking at the sample motor in the graphic above, an "A" engine has a total impulse of between 1.250 to 2.500 Newton-seconds. One of the benefits of a test stand is that you can test several motors from the same production lot and see how closely they actually perform to the listed classification. This can help make your simulations more accurate as opposed to using the standard figures given for a motor.

What's a Newton?

When you talk about the thrust of a model rocket motor, it is usually reported in "Newtons" (you may also see it referred to as "pounds force", especially in older documents). The definition of a Newton is the force needed to accelerate one kilogram of mass at the rate of one meter per second squared. In the formula below N = newton, kg = kilogram, m = meter, and s = second.

$$1\,N = 1\,\frac{kg \times m}{s^2}$$

You will also see the term "Newton-seconds" (the other older term is "pound-seconds"). This is a standard unit of impulse and is simply a one-newton force applied for one second. The newton is named after Sir Isaac Newton in recognition of his work on classical mechanics.

Average Thrust

The first number following the letter displays the average thrust in newtons. Average thrust is determined by dividing the total thrust by the thrust's duration. The formula below shows how to calculate average thrust where T_a = average thrust, T_d = thrust duration (in seconds), and T_t = total thrust.

$$T_a = \frac{T_t}{T_d}$$

This becomes important for you to be able to determine what motor you want to use in your model. For example both the C5-3 and the C11-3 have the same total impulse. However, the C11 has a higher average thrust, indicating that it has a stronger thrust than the C5-3. However, the C11 has a much shorter burn time than the C5. Based on the rocket you are looking to fly, you need to decide if it is better to use the higher thrust/shorter duration motor, or the lower thrust/longer duration motor.

Delay Time

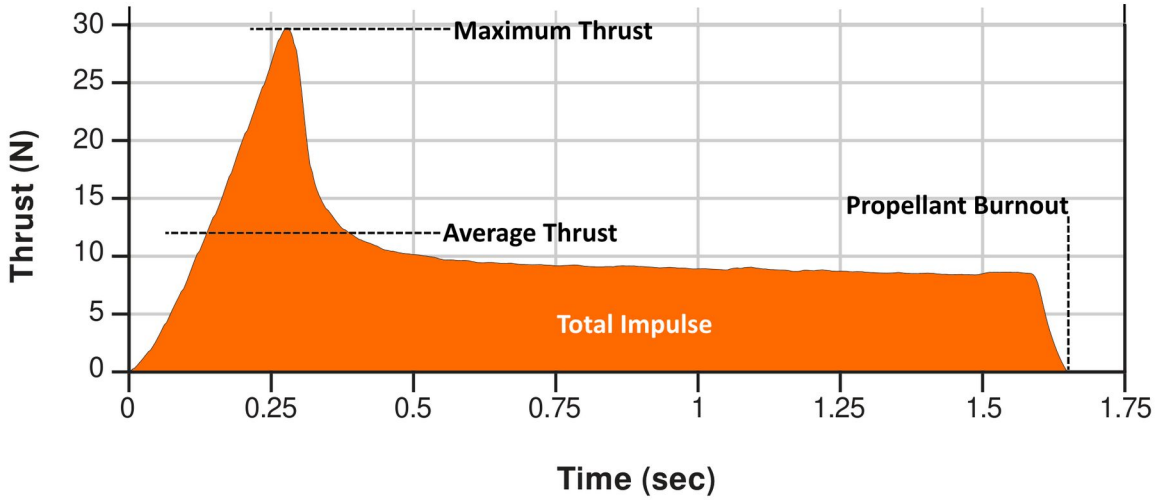
The last number on the engine casing is the delay time in seconds. This time starts at the completion of the propellant burn and continues until the ejection charge fires. This delay allows the rocket to continue to coast upward towards apogee. The end of the delay time should ideally occur at apogee. If the delay time is too short, the recovery system will deploy too soon, while the rocket is moving at speed, and can damage the recovery system. If the delay is too long, the rocket can arc over and pick up speed as it heads back to earth. The result is the same problem; the recovery system deploys at a greater speed than intended. Another outcome may be that the rocket is too close to the ground to allow the recovery system to fully deploy. The worst case occurs when the delay is too long and the rocket impacts the ground prior to the ejection charge being fired.

Some motors will have a "0" for their delay time. This indicates that the motor has no delay or ejection charge. These are booster motors and are designed to burn through at the top of the motor, sending hot propellant fragments into the stage above igniting the engine in the next stage. These motors should never be used in a single stage rocket, but only in booster stages.

Thrust Curves

A thrust curve is a visual representation of the motor burn. The X axis displays the time, while the Y axis displays the thrust in newtons. Below is the thrust curve of the Estes D12 motor. When using the test stand, the sensor results will allow you to produce a graph similar to the one below.

This is done by importing the numbers into a spreadsheet and creating a line graph. In the graph below, several items have been labeled. This includes total impulse, maximum thrust, average thrust, and propellant burnout.

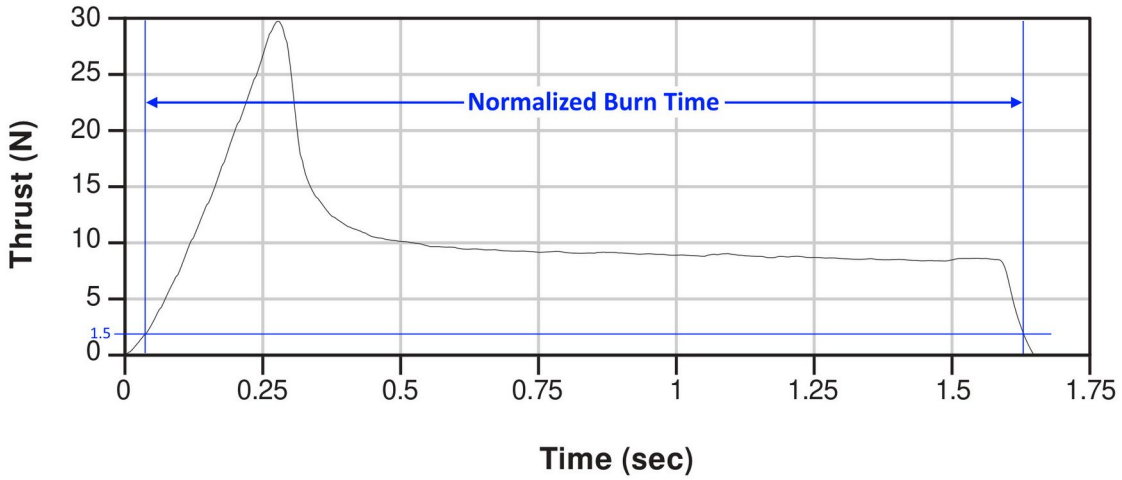


The total impulse is total amount of thrust produced by the motor during the total duration of the burn. This total impulse is used to determine the class of the motor. For this reason, the letter designation is a range for the total impulse of the motor and not a specific number of newtons For a D motor, the total impulse will be between 10.001–20.000 newtons.

The maximum thrust is just under 30 newtons. This is the maximum amount of force produced by the motor. In the thrust curve for D12 motor above, you can see where this peak occurs early in the burn then drops and levels off for the remainder of the burn. While this is typical of most model rocket motors, it is not true for all motors. Different curves are generated based on the motor design and the propellant being used.

While we discussed average thrust above, the average thrust that is displayed on your solid rocket motor has a slight variation from what you may expect. Here it is the average amount of thrust that is produced during a *normalized* burn time, not the entire burn time. The usefulness of the average thrust figure can vary depending on the thrust curve of the motor.

A normalized burn time is based on the NFPA (National Fire Protection Agency) Standard 1125. In a normalized burn time you first need to determine 5% of the maximum thrust. In our D12 motor, the maximum thrust is about 30 newtons, so 5% of 30 newtons is 1.5 newtons. Looking at the thrust curve, for a normalized burn time we don't start the clock until the thrust exceeds 1.5 newtons. We then stop the clock as soon as the thrust falls below 1.5 newtons. By using this method we can get a better indication of useful thrust. It is important to remember that when you look at the certification markings on a solid rocket motor, the average thrust is calculated based on the normalized burn time, while the total impulse is based on the entire burn time.



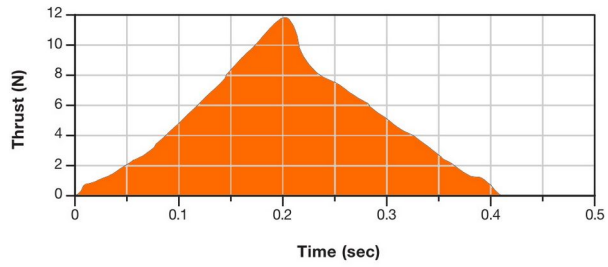
There are two other measurements that can be very useful when determining if a motor will meet the needs of the rocket and its mission. These include *initial thrust* and *specific impulse. Initial thrust* is defined as the average amount of thrust the motor develops for the first ½ second of the burn. This can be helpful in determining just how much weight the motor can lift. This measurement is not something that you will typically find in any of the motor documentation, but it is something that you can calculate once you do your own tests on the test stand.

Specific Impulse (I_{SP}) is a measurement of the motor's efficiency. It is calculated by dividing the total impulse by the weight of the propellant. The higher the specific impulse, the more impulse you get for the propellant weight. If you have two motors that develop the same total impulse, but with the first motor the weight of the propellant is half that of the opposing motor, the lighter weight motor can lift more total weight. Consider the following as a simplified example:

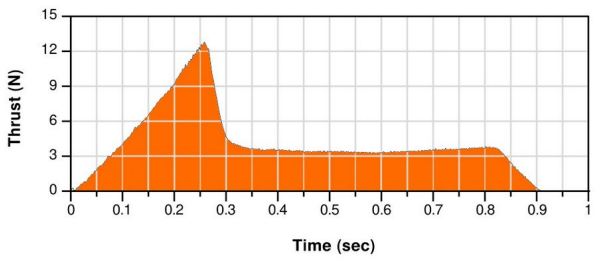
> Two motors have a total impulse of 20 newton-seconds and can lift a maximum weight of 400 grams. Both are used in the same rocket that weighs 350 grams not including motor and payload.
>
> - Motor A has a propellant weight of 20 grams, resulting in an I_{SP} of 1 second.
> - Motor B has a propellant weight of 30 grams, resulting in an I_{SP} of 0.667 seconds.
> - Due to the lighter weight of Motor A, the payload can weigh up to 30 grams.
> - Due to the heavier weight of Motor B, the payload can not weigh more than 20 grams.

Rocket Motor Design

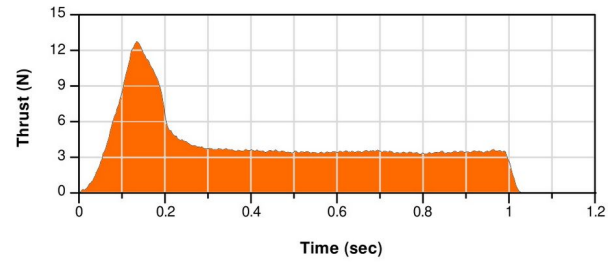
In the example above, we looked at the thrust curve for an Estes D12 motor. Below are four additional thrust curves from four different motors.



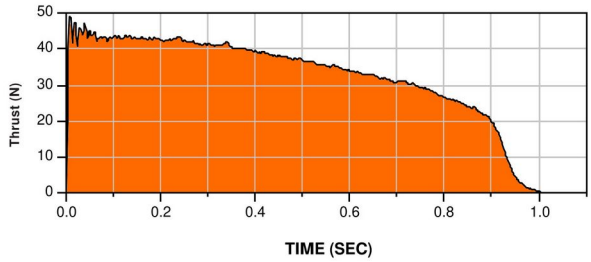
Quest A4



Quest B4



Estes B4



Estes E30

Looking at the five thrust curves, you can see that they are all unique in their shape. Even the two B4 motors have very different thrust curves. The reason behind such variations and the cause of these differences all comes down to motor design.

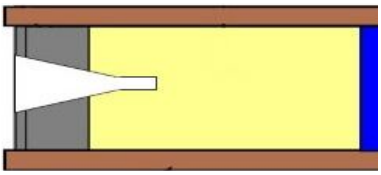
Rocket motors, whether on full size rockets like the Falcon 9, or model rockets like the Alpha, are all designed and made to fit a specific need or requirement. Consider a motor like the B14 that has a huge kick that is delivered nearly all at once. Such a motor would be good for booster stages and high acceleration studies.

So how do they get the different thrust patterns? For model rockets it all comes down to the design of the propellant.

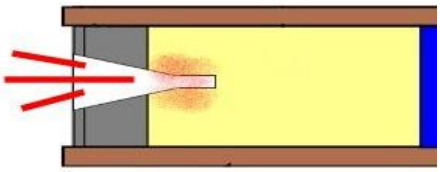
Most model rockets are using black powder (as you get into mid and high power rocketry that changes, but let's stick with black powder for now). Most are using the same nozzle design. So what changes is the pattern or geometry of the propellant grain.

To get the propellant to burn faster, you need more surface area exposed. In our model rocket motors this is done by creating a core hole in the center of the propellant. On most model rocket motors this core doesn't go very deep. The faster burning around the core gives the motor its

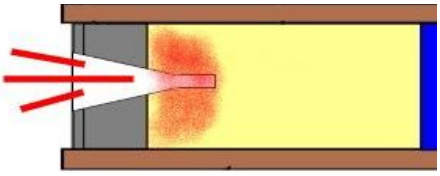
initial kick. As the propellant continues to burn the core burns away and transitions to a flat, end burning operation. This does not burn as fast, so the thrust drops and gives you the longer, flat portion of the thrust curve



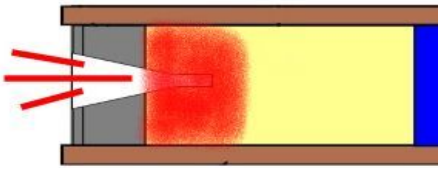
Motor awaiting ignition



Motor ignited, begins burn around central core



Motor continues to burn around central core, begins transition to flat bottom burn



Central core burned away, now solely burning from flat bottom

03
Rocket Motor Test Stands

Rocket motor test stands have been around almost as long as rocket motors. As stated in the very name, a test stand is used to test the performance of the rocket motor. Test stands used by NASA are typically very large as they are testing very powerful rocket engines. In the picture below you can see the A-1 Test Stand at the Stennis Space Center testing one of the Space Launch System RS-25 rocket engines.

Not all rockets need the large test stands. Smaller rocket engines can make use of smaller stands, some of which are portable.

Test stands can perform a number of tests of a rocket engine. They can measure the performance of the engine, how well it reacts to commands, the temperatures inside the engine components while the engine is firing, length of time the engine is firing and more.

The stands also need to be able to absorb the massive amount of energy released by the motor. Most of these motors are destined to be launched into space and so develop a lot energy. The test stand needs to be constructed to contain that energy. Even more important is the ability of the test stand to minimize damage in the event of the failure of a motor. Should an engine experience an anomaly you can have a sudden release of all of that energy in a very uncontrolled manner.

There are basically two types of test stands; horizontal and vertical. The test stand at Stennis pictured above is a vertical test stand. Vertical test stands are very good at testing the rocket in the same orientation that it will be flying in. This allows for testing of the fuel tanks, pumps and other hardware in the same attitude that the rocket will be in during flight. Vertical test stands tend to be more expensive to build and maintain as they're very tall. You will probably need a flame trench as these rockets fire in a downward direction, the same as during flight.

The test stand at the bottom of the previous page is the J-1 horizontal test stand at the Glenn Research Center. Horizontal test stands are typically used just to test engines themselves and not the associated hardware. These stands are usually cheaper to build and maintain, and depending on the size of the stand, can even be made portable. Because the engine exhaust does not point towards the ground, there is no need for a flame diverter or trench. Horizontal test stands can often be seen in university settings as they tend to use less powerful motors and are cheaper to build and maintain.

Our Test Stand

The test stand we are going to build has many of the same characteristics of the larger test stands. It places the solid rocket motors in the vertical position. However, instead of the motor exhaust facing down it will face up. This stand is designed to test commercial solid rocket motors where the specific impulse and thrust curve is published by the manufacturer and have been tested and certified by the NAR. The thrust curves created by this test stand can be compared to the NAR developed thrust curves. This can be valuable when testing under various environmental conditions (such as cold winter or hot summer temperatures) and using the updated data to more accurately simulate rocket performance.

The test stand is also our next step in learning about electronic design and fabrication, as well as programming while using an Arduino microcontroller. Our past projects have used the Arduino Nano, either as the main board for an electronic payload or to control the Arduino Launch Control System. This project utilizes the Arduino Mega2560 board, which is more powerful and has more pin options available.

The test stand makes use of a remote unit, tethered to the main test stand by a 10-foot DB15 cable. This is a new challenge to be able to control the electronics located on the stand from a safe distance away.

This also challenged me to expand my ability to design and print 3D enclosures for the electronics in the stand and the remote unit. The design needs to include attachment points for the electronics and access to the various connections, such as USB and the DB-15 cable. I needed to make sure that the mounting holes were located properly and that the various boards are positioned in logical locations.

Lastly, I needed to do more programming than any previous project. The test stand software has three separate logs for each test. The first log records basic information about the test (such as location, motor tested, etc), the second log records the functions of the board and when they occur (such as re-calibration of the load cell, when the fire button was pressed, environmental factors just prior to ignition, etc) and finally, the log of the test firing so that thrust curves can be created.

I have seen other projects where the electronics are out in the open, often using a breadboard to create the project. There is no doubt that these projects work and I have found several of these provided inspiration for this project. However, I wanted a test stand that had the look and feel of a working piece of test equipment. I wanted a test stand that would look ‘finished’ and could be

placed on a display of model rocketry objects and look like it was an actual research tool – because it is. That is why I designed the 3D enclosure and remote head.

This project presents a number of challenges, and like our previous projects I worked through each one until I had a functioning test stand. I hope that you will try this project and enjoy building it as much as I did. Remember, I am learning this new (to me) aspect of the hobby, so I talk about what went right and what went wrong. After all, it is the things that go wrong that teach us the most.

04
Designing the Test Stand

As with any science or engineering project, one of the first things that needs to be done is background research. What have others done, how well did it work, what improvements can be made? What can we learn from their experience and incorporate into our designs? I needed to determine what specific tasks I wanted this test stand to perform. My research took me down two specific paths – the first concerning the motors themselves, and the second path concerned the construction and layout of the test stand.

Research

The National Association of Rocketry (NAR) offers its members access to the Research and Development (R&D) reports library. These reports that have been submitted over the years for the R&D event at the National Associations of Rocketry's Annual Meet (NARAM). These reports covered a wide range of topics, with a number of reports looking at the performance of black powder engines. Some reports looked at the physical characteristics of the motors prior to use and correlated that to the performance during flight. Other papers reviewed how external factors can impact the performance of the motor.

Another important part of the research was reviewing the documentation of the NAR Standards and Testing (S&T) Committee. This committee performs the testing for all model rocket motors certified by the NAR. They have specific requirements and procedures that they use when conducting testing. While the S&T committee looks at performance, they are looking for consistency in performance and that the labeling of the motor by the manufacturer equates to the actual performance of the motor.

There are also a number of documents, tutorials and videos on constructing model rocket test stands. Some of these are very simple and basic in construction and operation, while other stands are designed to conduct testing of high performance, high power rocket motors.

I looked at the documentation on the test stands used by NASA, SpaceX and other rocket companies. I wanted to understand how they use their test stands, what procedures are used during a test, what data they collect, etc.

The one thing that had an impact in the design and construction of the test stand was the need to make it a useful educational and research tool. Many of the NAR R&D motor research documents needed a test stand as part of their research. I read an article on how a high school was using a test stand to help teach calculus[3]. With TARC competition being ever more popular I was sure they would be conducting testing of the motors the teams plan to use. I wanted to create a test stand that would be meet all of these challenges.

3 "Using a Model Rocket-Engine Test Stand in a Calculus Course" - https://pubs.nctm.org/view/journals/mt/95/7/article-p516.xml

Based on this research, I developed a list of specifications of what I wanted to be able to accomplish with the test stand. I decided early on that the stand will be designed for low to mid power black powder motors. It needs to be flexible and have the ability to test a variety of motors various diameters and lengths. To accomplish this it meant designing mounts that had to be interchangeable.

I wanted the stand to look more like a "real" piece of testing equipment. A number of test stands that you find on the web are made using just breadboards with all of the wiring exposed. I wanted something that had a more finished look to it. I also saw on other systems where there was a lot of 'residue' from the black powder motors that would get on the test stand's electronics, and sometime cause issues. This meant that the electronics would need to be protected in some fashion.

Looking at the requirements made it easy to determine what electronics would be needed in this project. A load cell is necessary to perform the actual thrust measurements. Here I decided to use a 5kg cell, as this would be adequate for the motors I envisioned being tested.

From the research I had performed I also knew that environmental conditions (such as temperature, humidity and barometric pressure) can impact motor performance. This necessitated an environmental sensor be incorporated.

The S&T Committee records the temperatures on the outside of the motor casing to ensure it doesn't rise above 200^{o} C. This sensor would have to wait till later.

I needed a way to store all of the data from each test. For that I would decided on a MicroSD Card module solution.

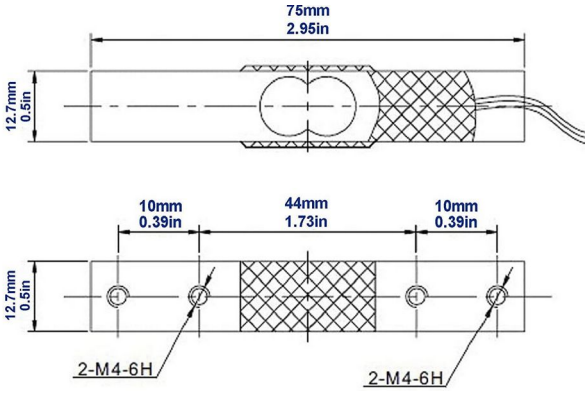
There are other items that I want to include. A warning system needs to be included. A simple solution is to use a piezo buzzer and warning lamps. A clock which could provide basic info to the user along with an LCD screen for instructions.

Finally, there would need to be an interface between the user and the test stand. Much like a launch controller, the interface (or remote head) would need to be about 10-15 feet from the stand. I also saw the need to have a laptop be an integral part of the system. The laptop would be used to monitor the system as well as allow the user to provide input. If things went well, I was hoping to incorporate a Raspberry Pi into the system (that didn't happen – at least not yet). With the research complete and these specifications in hand, it was time to start work on designing both the hardware and software for the test stand.

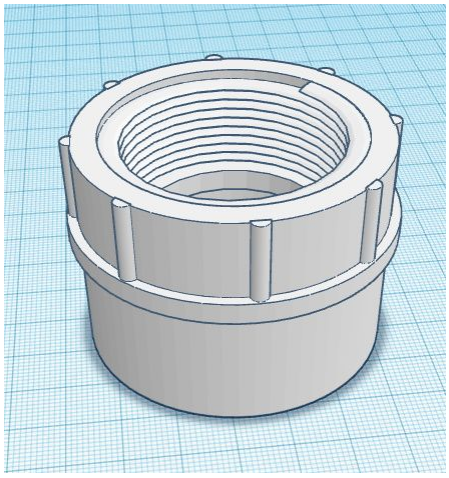
Initial Designs

I wanted to have a good idea of how the test stand was going to look. When I created Project: Icarus it was the first time I had used Tinkercad to create parts for a project. I decided to use it again, this time from the beginning of the project, to develop an idea for how the test stand would be built.

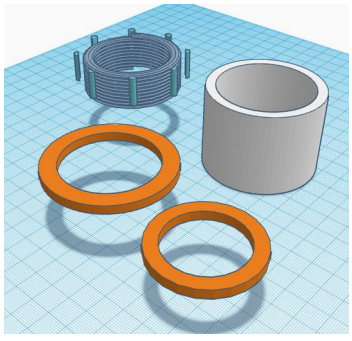
For the interchangeable motor mounts I had decided to utilize PVC pipe. They would be attached to a load cell, which is then mounted on a nearly square piece of particle board (actually a piece of scrap from our counter top from a bathroom remodeling job). While I could create the test stand base as a simple cube, there is no load cell nor PVC pipe fittings in Tinkercad. Before I could design the test stand I needed to create the parts that would make up the individual components of the test stand.



The first item I tackled was the load cell. A set of dimensioned drawings was available so it was a fairly simple matter to recreate it in Tinkercad. With the load cell created as a part, it was ready to be inserted into the drawing. However, there were no drawings of M4 screws or nuts available. This became the next step. Once these items were completed, I was able to add the load cell to the base of the test stand.



Next on the list was the motor mounts and attachment plate. The attachment plate is an oval electrical panel cover. It is not a perfect oval or square, but rather a combination of both. After measuring an actual cover, one was created and it was attached to the load cell.
I was now at the point where I needed to create the PVC connection that attaches to the plate on the load cell. I planned on using a 1.5-inch female fitting on the plate. This would allow the use of both 1.5-inch and 1.25-inch fittings. Taking careful measurements of the actual fitting, a reasonable replica was created in Tinkercad.



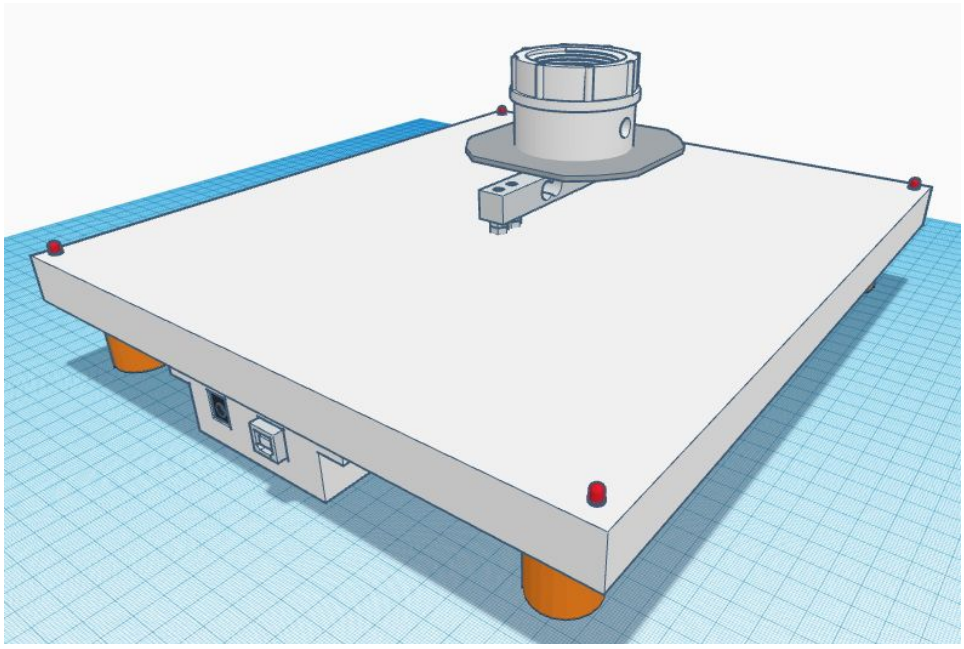
Looking at the exploded view on the right shows that the bulk of the design is just round tubes. A helix is used to simulate the threads inside the fitting. A series of "round roof" components were used to simulate the raised edges. When combined together, the result was a satisfying facsimile of a 1.5-inch PVC fitting.

The PVC fitting was centered on the load cell platform. I knew that I would need to vent any ejection charge gases so a hole is created in the fitting, perpendicular to the load cell.

The next step in the design process is to determine where the Arduino Mega2560 board will be located. I wanted to keep the board and any other electronics away from the exhaust and other particulate matter of a burning motor. I also knew that I have to keep the load cell amplifier close to load cell and the Arduino board. The best (and easiest) solution is to locate the board under the test stand. Furthermore, by

enclosing the board and any other electronic components in a protective housing it helps keep the components free from any debris.

Tinkercad has a number of small electronic boards, including the Arduino Nano. However they do not have a Mega2560 board. Hoping I would not need to create an Arduino board in Tinkercad I did a search and found a nice ready made drawing. This was copied and imported into the test stand drawing. My initial plan was to create a simple box that would cover all of the electronics located at the test stand. I even added some warning lamps to each corner of the stand. Finally, a series of small round legs were created to raise the test stand up off the ground. Thus the first design version of the test stand is complete.



Even though I now had an initial design drawing there were still a large number of unknowns with this project. In order to develop the code required to run the test stand, I would need to have the load cell attached to the base. Still I was pretty confident that I could use this drawing to start building the physical test stand.

The stand was laid out as designed and holes are drilled for the load cell mounting points. An additional hole is drilled to allow the electrical wiring to pass through the stand and remain underneath. Instead of creating the legs at this stage, I simply use small painters tripods to hold it up off the work table.

With this stage of the design and construction complete, it was time to gather the electronic components I need to start writing the code to bring the test stand to life.

05 Components

A Quick Note about the Products Mentioned

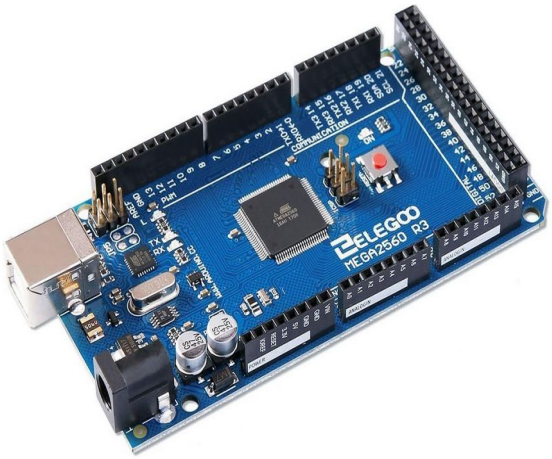
Any product that you see mentioned in this manual is listed because I bought it, use it, and found it did the job I asked of it. No person or company sent me anything. I do not receive any type of compensation if you buy anything I mention here.

In the introduction I outlined what I want the test stand to record (engine force and environmental conditions) along with how to input, display and store that data (the Serial Monitor and MicroSD card). Based on these requirements, I have a pretty good idea of the electronic components I need. I also know this could change as I get into the actual development process.

Another aspect of this project that is different from our other projects is that it is divided into two separate components. The bulk of the electronics and sensors are located under the test stand itself, while a remote head is used as the interface to interact with the stand. In this section I will list all of the electronics that I used in final project. They are broken down into two areas; *Test Stand Components* (the parts under the stand) and *Remote Head Components.*

The Test Stand Electrical Components

Elegoo Mega2560

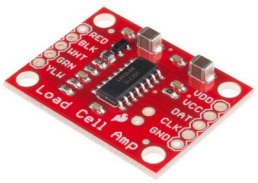


The Mega2560 board is the one I chose for this project. It is a clone of the Arduino Mega2560 board. This board has additional memory and pins slots. The setup of the board makes it easy to attach jumper wires between the board, a breadboard, and the various sensors and components. It also helps with rapid testing of various aspects of the system as well as moving connections and components when needed.

It becomes a bit more of a challenge when moving from testing to actual product. The fact that the board is installed in the test stand and in an inverted position provides an additional challenge.

Load Cell & Amplifier

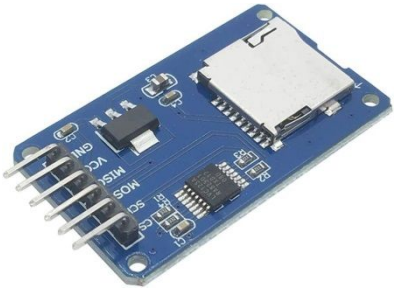
The Load Cell is used to record the amount of thrust generated by the rocket motor. I purchased a 5kg cell which will be plenty for the motors we are testing.



The load cell electrical signals by themselves are too weak to register so an amplifier is required. While the load cell did come with an amplifier I decided to go with the Sparkfun HZ711 amplifier instead. Most of the research done on the amplifier indicated that it was more accurate and stable than others, so I spent the extra money for the upgraded board. The choice is yours if you want to do this.

MicroSD Card Module

The MicroSD Card Adapter Reader Module is used to record the motor performance data, the system performance data, and the basic test information to a MicroSD card. The module I chose has a 6-pin SPI Interface, consists of GND, VCC, MISO, MOSI, SCK, and CS pins. The module that holds the MicroSD card is recessed slightly from the edge of the circuit board. While this caused a bit of a headache in inserting and retrieving the MicroSD card, a simple solution made the task easier.



LEDs

There is a need to be able to tell the status of the test stand at any point during the test. We wanted a visual indicator when it was safe to approach the test stand, and when to stay away. To meet this need I use a combination of 10mm RGB Multicolor LED lamps and 5mm white/clear lamps. A pair of these LEDs are located on each corner of the test stand. Besides acting as a warning, the lamps look really cool!

RGB LEDs

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 stand. I use the three basic colors of Green (OK to approach the stand), Yellow (a motor is on the test stand and will soon be fired) and Red (the countdown to firing the motor has commenced or we are in the post fire stage).

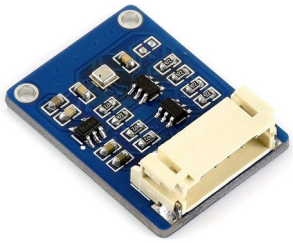
Red and White/Clear LEDs

A 5mm red LED is used to show that the ignition system has continuity. We use the 5mm clear LEDs to add a flashing strobe pattern once the test stand is in either a “Yellow” or “Red” condition. This is an additional warning to supplement the color RGB LEDs.

Piezo Buzzer

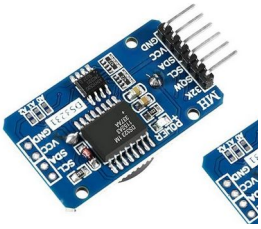
The piezo buzzer is used to help draw your attention to certain activities occurring on the system. It will chirp when the system successfully initializes, it provides warnings that a countdown is underway on each second as well as a long continuous tone while power is being supplied to the test stand. It also chirps rapidly and repeatedly when a launch is aborted.

BME280 Environmental Sensor



The BME280 is used to record temperature, humidity and barometric pressure. It uses the I2C communications protocol, so it shares these pins with the Real Time Clock and the LCD screen.

Real Time Clock



A Real Time Clock (RTC) provides an accurate date and time stamp for the data. It has its own battery which keeps the clock running even when power to the system is turned off. Like the BME280, the RTC uses I2C and so it shares the use of these pins.

Firing Relay

The firing relay takes the signal from the Arduino and allows power to continue to the motor test stand to ignite the motor. It is the gate keeper for the test stand power. It uses a single pin on the Arduino (D49) to send the signal to the relay when to open.

On the opposite side of the relay is the positive power coming off of the batteries. This is attached to the Common (middle) connection. A second line comes off the relay on the Normally Open (NO) connection and goes to the igniter clip at the stand.

Resistors

A number of resistors are required for this project. They are used for the RGB LEDs and the strobe lamp LEDs. 220 Ohm resistors are for the RGB LED to function properly. I 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). I also use a 1KΩ resistor for the continuity circuit.

AA Battery Holder

This is used to provide power to the igniter. I used a four battery holder on my version of the test stand. You can use whatever size battery pack you desire. Just make sure you adjust the resistance for the continuity lamp for the power supply you select.

The Remote Head Electrical Components

The remote head is the hardware interface between the user and the test stand. It is what allows the user to operate the test stand at a safe distance.

LED Clock

I wanted to add an LED clock which also acts as the 5-second countdown clock and to display other simple messages. I selected a TM1637 display. This is a 4-digit, 7-segment display. It uses two of the digital pins on the Arduino Mega2560 (D12 and D13) and requires a +5V and ground connection. The LED clock gets its information from the Mega2560 board, but it is located in the remote head.

LCD Screen

The LCD screen is used to display the current date and time, as well as provide updates on the system. The LCD screen utilizes the I2C interface, along with the BME280 and the Real Time Clock.

Function Buttons

Four buttons are located on the remote head that control the functions of the test stand. They include a Reset button, a Cal(ibrate) button, a Start button and a Fire button.

Reset Button

This is a momentary push button that when pushed will reset the system. It will cause the system to reboot and go through the initialization process. It is connected directly to the RST (reset) pin on the Arduino. This button uses the white cap. The system is reset prior to each use.

Start Button

When pushed, this momentary button starts the actual test process. It requires the user to begin entering information in the computer through the Serial Monitor screen. It is connected to pin D9 on our Arduino and uses the green cap.

Calibration Button

This is another momentary push button that is used to calibrate the load cell prior to use. It is connected to pin D8 on our Arduino. This button uses the blue cap.

Fire Button

This button is used to start the 5-second countdown to the firing of the motor. If the button is released during this time, the firing of the motor is aborted. It is connected to pin D7 on the Arduino.

DB15 Connectors and Cable

We needed the ability to fire the rocket motors a safe distance from the test stand. We use a USB cable to connect the Arduino to the laptop and a DB15 cable to connect the electronic connections from the test stand to our remote head. This required the use of two female DB15 connectors (one attached to the test stand and the other attached to the remote head) and a 10-foot DB15 cable.

Other Hardware

While the electronics are the primary focus of this manual, there are other parts that we need to complete the test stand. Here are the rest of the materials we used in the making of the test stand.

- Wood Base
  Approximately 1-foot by 1-foot
- PLA+ filament
  I used black and orange, but feel free to use any colors you like
- Wood screws
  To secure the legs and wiring conduits
- 1.25-inch PVC pipe
  This will be used to create the motor mounts
- 18mm and 24mm motor mount tubes
  I used standard Estes motor mount tubes and motor blocks
- Electrical box cover
  This is a standard 4-inch square steel box cover that is used to attach the PVC mount to the load cell
- Alligator clips
  These are used to attach the power cables to the igniter
- Heat shrink tubing
  This is used to protect and cover solder connections in the electronics housing and remote head
- Heat set inserts (M3)
  These are used in the electronics housing and remote head to attach the covers to the base
- Stainless steel screws (M3)
  These are used with the heat set insert

- Nylon screws and nuts (M2 and M2.5)
  These are used to attach the electrical components to the electronics housing and the prototype boards
- 2.5M steel nuts and bolts
  These attach the load cell to the test stand platform. Use a length that will work with the thickness of your platform
- Protective Wire Wrap (3/8-inch)
  This will cover the wires from the warning lamps on the underside of the test stand platform
- Felt feet
  These are added to the bottom of the adjustable legs on the test stand
- Rubber feet
  These are added to the bottom of the remote head to stop it from sliding around
- RV camper levels
  These come in pairs and are attached to two sides of the Test Stand platform to allow you to level the platform

Tools and Supplies

- Soldering iron and solder
- Heat insert tool
- Screwdriver
- 3D printer
- 5-minute epoxy
- Blue painters tape
- Helping hands clamps
- Label maker
- Stickon Vinyl
- Orange and black paint
- Ruler, pencil
- drill and drill bits
- PVC adhesive
- Wood glue
- Clamps

06
Writing Code

This is the fifth rocketry project that I've created using an Arduino board. I knew I would be able to reuse some code functions from our previous projects. This includes:

- The firing sequence, countdown timer and LED clock routines would come from the Arduino Launch Control System (LCS) project.
- To store the data that would be collected would require the use of a MicroSD card. For this I could use the code that I developed for both Project: Icarus and The Olympus Project.
- These two projects would also provide the base code for controlling the LED lamps located on the test stand.

The code that records the load cell data would have to be new. I knew that the code from the BMP180 sensor would not transition to the BME280 sensor. I would also need to write code to allow the user to interface with the stand via the Serial Monitor on a computer.

There would be a need to create test routines in the code. Some of these would be permanent while others would be temporary. New in this project would make use of the Serial Monitor as an input device and not just as a monitoring tool. I would need to write code to allow the user to interface with the stand via the Serial Monitor on a computer. I also wanted the code to "make sense" so that I could come back later (days, months, maybe years later) and understand what I wrote.

Documenting the Code

A major reason for documenting your project is to record what the code you write is supposed to perform. 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.

Another aspect of documentation is to keep a journal of notes that tracks what you wrote. This can be used to make comments on issues you ran into and how you solved them. These notes will likely be more in-depth than the notes in the code itself.

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 you can create your template and save it. Next if using a Windows computer, open up the file manager and go to "C:\Program Files (x86)\Arduino\examples\01.Basics\BareMinimum". You will see the file BareMinimum.ino. Rename the file 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.*
>
> *With the release of the update Arduino IDE 2.0, a different procedure must be used. The following is from the Arduino web site at https://forum.arduino.cc/t/arduino-ide-2-0-1-is-now-available/1046764*
>
> When a new sketch is created, Arduino IDE populates it with the bare minimum empty setup and loop functions.
>
> Some users may wish to adjust this code. That is now possible by specifying the path to a file containing the custom content in the advanced settings:
>
> 1. Press the **Ctrl**+**Shift**+**P** keyboard shortcut (**Command**+**Shift**+**P** for macOS users) to open the "**Command Palette**".
> 2. Select the "**Preferences: Open Settings (UI)**" command from the menu.
> 3. A "**Preferences**" tab will now open in the IDE. In the "**Search Settings**" field, type arduino.sketch.inoBlueprint
> 4. Add the path to the file containing your custom new sketch content in the field under the "**Sketch: Ino Blueprint**" setting.
> 5. Click the **X** icon on the "**Preferences**" tab.
> 6. Select **File > Quit** from the Arduino IDE menus.
> 7. Start the Arduino IDE.

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 is a series of comment blocks that provide information on the project. The first section gives the name of the project, the version number, a brief description of what the project does, 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: Model Rocket Motor Test Stand
 * Version: 1.0.0
 * Description: This test stand is designed for black powder model
 *              rocket motors from mini "T" motors through "F"
 *              power motors. Test stand utilizes a single load cell
 *              and a HX711 amplifier.
 *
 *              Test stand includes a BME280 temperature, humidity and
 *              barometric pressure sensor which is displayed on an LCD
 *              screen.
 *
 *              Incorporates a RTC to display time on a 4-digit
 *              7-segment LED.
 *
 *              It maintains three logs during the test
 *               - Basic test information
 *               - Test stand system status
 *               - Load cell data during test firing
 *
 * Created: 05 January 2023
 * Updated: 28 September 2024
 *
 * 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 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. It allows you as to go and see how the original code was written. Finally, 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:
 *
 *  TM1637 Clock Example
 *  https://www.makerguides.com/tm1637-arduino-tutorial/
 *
 *  Arduino - LCD I2C Tutorial
 *  https://arduinogetstarted.com/tutorials/arduino-lcd-i2c
 *
 *  Read Load Cell (Examples -> (Examples from Custom Library) HX711_ADC ->
 *  Read 1x Load Cell
 *
 * ================================================================
```

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 Mega 2560 board
 *
 * Peizo Buzzer
 *   SIG    A0
 *
 * Strobe LED Lamp
 *   SIG    2   (D2)
 *
 * RGB LED lamp
 *   Red    3   (D3)
 *   Blue   4   (D4)
 *   Green  5   (D5)
```

This completes the Title Block of my code. All of this information has been nothing but comments - there is no actual, executable code contained here. However, when I return to the code 6 months, a year from now, or longer, this section of code will provide the basic information I need to understand what the project hardware is designed to do.

Header Blocks

There are other items I use while writing code besides in the very beginning. One item I use frequently is header blocks. Every time I start a new function, I add a header block. They are placed just above the start of the function, as seen in the example below:

```
/*****************************************************************
******************************************************************
*                                                                *
*                  FIRE DATA COLLECTION SEQUENCE                 *
*                                                                *
******************************************************************
*****************************************************************/

void fireDataCollection(void)
```

This header block lets me know what the function is designed to perform. If needed, I can add additional comments to explain the function in greater detail. Typically, you will find these at the top of a new tab, as I tend to put each function in its own tab. However, if I have multiple functions in the same tab, each function on that tab gets a header block.

Subsection Dividers

These are contained within each function. These are used to identify what each part of the function is designed to accomplish. A sample is seen below:

```
// ============================================================
// Ready for Test Firing
```

If needed, additional comments are added for clarity.

Comments

Finally, as we noted previously, make use of comments. The can be a great help when trying to understand what the code is trying to do.

```
// Write the following in this section
//  - time stamp in milliseconds
//  - data from load cell
//  - call to write data to SD card
```

07
Libraries, Declarations, and Setup Code

With the Title Block created, along with the pin assignments and code references, it is time to start adding in the first parts of actual code. First we must identify any libraries that will be used in our program. This is followed by the declarations for any global variables that will be used in the software. Once these items are entered we can proceed to the setup() function.

Libraries

Libraries are pieces of code that help make writing your program 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 on the types of libraries that have been made available.

Any library I use is listed in this section. I also use comments to indicate what the library is being used for. Sometimes it is for a component (such as the MicroSD card module) while other times it is a functional library (such as the timer library used for the strobe lamps). Each library is separated by a series of lines. This lets me know exactly what libraries are used for specific components/functions. A sample is seen below.

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

// ===============================================================
// library required for the Real Time Clock DS1307
   #include <RTClib.h>

// ===============================================================
// library required for the TM1637 display
   #include <TM1637Display.h>

// ===============================================================
// library required for flashing strobe lights
// regardless of other activities
   #include <MsTimer2.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.

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

// ===============================================================
// declarations for Program Version
   const int prgMajor = 1;
   const int prgMinor = 0;
   const int prgPatch = 0;
```

The Declarations section is rather lengthy in this project. The declarations made here are used in multiple functions throughout the program. If a variable is only used within a single function, it is declared within that function and is not listed here. Now let's take a closer look at the MicroSD card declaration section, shown below.

```
// ===============================================================
// declarations for the MicroSD card
   float timeDataCollection;
   String dataString = "";   // make a string for assembling the data
                             // to the log

   char fileMotorLog[] = "MTRLOG00.CSV";
   String sdImpulse = "";
   String sdTotalImpulse = "";
   String sdTimeStamp = "";

   char fileSystemLog[] = "SYSLOG00.CSV";
   String sdRTCStamp = "";
   String sdMessage = "";
   String sdResult = "";

   char fileInfoLog[] = "INFLOG00.TXT";
   String sdInfo = "";
   String sdDate = "";
   String sdImpulseTotal = "";
   String sdThrustAvg = "";
   String sdDelay = "";
```

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

- There are declarations for three different logs:
  - `MTRLOG00.csv` (Motor Log)
  - `SYSLOG00.csv` (System Log)

  - `INFLOG00.csv` (Information Log)

  Each log tracks a different component of the test stand, yet all three logs will have the same number attached.
- The first is the variable `char fileMotorLog[] = "MTRLOG00.CSV";` This is the file name that will be stored on the MicroSD card. 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. The csv extension for import into a spreadsheet or database.
- 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.
- The variable `sdTimeStamp = ""` attaches the elapsed time stamp to the file. This is the amount of milliseconds that have elapsed between the time the Mega2560 was powered up and the data was collected.

Next we will look at the `void setup()` and `void loop()` code that make up the rest of the code in this tab.

void setup()

This section of code is included in every Arduino program. It is where we start setting up the Arduino board 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.

There are a number of things going on here, and they are performed in a specific order. You should also notice that most of the items in `setup()` are actually calling other functions located elsewhere in the program.

Using Functions

If you have looked at most Arduino code published on the Internet, you will typically find everything written out in the `setup()` function (and the `loop()` section as well). This is fine for small programs, but it can quickly get out of hand when writing larger programs such as this. To make things easier to code, to debug and to ultimately understand what the software is doing, I use functions to perform very specific actions. Let's take a look at part of the code in the setup() to see how this works. Take a look at the code snippet below:

```
// ================================================================
// Setup for Real Time Clock
   setupRTC();

// ================================================================
// Check Date and Time
   setupDateTimeCheck();

// ================================================================
// Setup for MicroSD card
   setupMicroSDCard();
```

```
// ===================================================================
// Setup for BME280 Sensor
   setupBmeSensor();

// ===================================================================
// Setup HX711 Load Cell
   setupHX711();

// ===================================================================
// Setup Fire Control System
   setupFireControl();
```

Even if you don't know anything about programming or the C language, it is still pretty easy to figure out what the program is doing. Each function call describes what you expect to have happen, whether its setting up the MicroSD card, activating the BME 280 sensor, or getting the Fire Control System ready for use. The function names are descriptive enough to figure out what you expect them to do that you really don't need the comments - but put the comments in anyway! There are a number of activities being performed in `setup()`. However, by using functions, we make it easier to really understand what is happening during the setup process.

This also helps with debugging. When you discover something not working the way you expect, having the program broken into separate functions makes it easier to troubleshoot. Suppose you have problems with initializing the MicroSD card and you think the problem is in setup(). By commenting out the function call, you can bypass that function. If the program works as expected, you now where the problem likely located. If the program still fails at the same point, then you know that the function you bypassed isn't the cause.

I would encourage you to use functions in your own programs, especially as they become more expansive and involved. You will probably find it makes life much easier than putting everything on a single tab.

In the rest of this chapter we will take a closer look at what is actually occurring in the `setup()` function.

Initializing Serial Port

The first item in the setup routine is to initialize the serial port. This is performed first as it allows us to see what our program is doing through the Serial Monitor. When we run into issues we can write test code and see the results on the Serial Monitor. With the Test Stand program, we use the Serial Monitor to input data. If we can't get the Serial Monitor initialized, we simply can't use the software.

The rate is set to 115200 baud and this must match the rate on the serial monitor. If the Mega2560 and the Serial Monitor are at two different rates, you will likely get nothing but garbage characters on your serial monitor screen. You might not get nothing at all.

Splash Screen

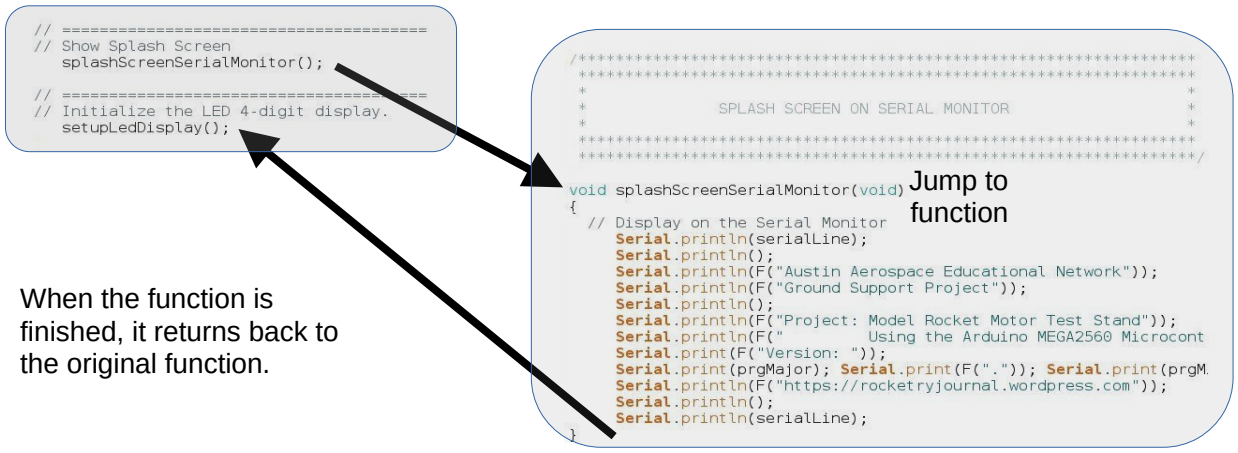
The line `splashScreenSerialMonitor();` is used to call the first function of setup. If you look at the code in the Arduino IDE, there is a tab titled "Serial_Monitor_Splash_Screen". The code in that tab is below.

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

void splashScreenSerialMonitor(void)
{
  // Display on the Serial Monitor
     Serial.println(serialLine);
     Serial.println();
     Serial.println(F("Austin Aerospace Educational Network"));
     Serial.println(F("Ground Support Project"));
     Serial.println();
     Serial.println(F("Project: Model Rocket Motor Test Stand"));
     Serial.println(F("         Using the Arduino Mega2560 Microcontroller"));
     Serial.print(F("Version: "));
     Serial.print(prgMajor); Serial.print(F(".")); Serial.print(prgMinor);
     Serial.print(F(".")); Serial.println(prgPatch);
     Serial.println(F("https://rocketryjournal.wordpress.com"));
     Serial.println();
     Serial.println(serialLine);
}
```

At the very top is the header that we described earlier in this section. Directly after that is the statement `void splashScreenSerialMonitor(void).` This is the name of the function and this is where the program will jump to when it sees the line `splashScreenSerialMonitor();` in `setup()`.

The code displays a splash screen on the serial monitor. It identifies the name of the program, the version number and the address to our web site. Once the code has been executed, it will return back to the `setup()` function and execute the next line in the code (see graphic below).



Setup Display Screens

The next two sections setup both display screens, the 16 x 2 LCD screen and the 4-digit 7-segment display. These are setup early to allow us to see additional information on these screens. The LCD screen lets us know that the system is initializing while the LED display shows "BOOT". As each item comes on line, we will see this displayed on the LCD screen.

The setupLedDisplay() is also the first time we run into locally declared variables. In this case the variables are used to show "BOOT" on the display. "BOOT" is only displayed in this function, so by putting the variable declaration locally in the function the memory used to hold these variables are released once the function is complete. By using local variables instead of global variables where possible, we help preserve memory for use by other functions.

LED & Buzzer Pins

Next, the pins for the LEDs are setup. This includes both the RGB LEDs and the white LED that will act as a strobe. This is followed by the buzzer pin initialization.

Setting Up the Real Time Clock (RTC)

Just about everything performed with the test stand has a time attached to it. Sometimes the time is reported in milliseconds since the Arduino was activated. Other times it is reported as standard time in a 24-hour format, and sometimes both times are reported. The Real Time Clock not only keeps the local time, but contains a battery that keeps the clock running even when the power is turned off to the Arduino.

In this function the programs looks to see if the RTC is present and running. If it can't find the clock it reports the error on all three displays, turns the LEDs to red, and stops the program from progressing any further. If the clock isn't running, it won't make sense to continue. If the program is able to find the clock, it reports that RTC has been initialized.

Next it checks to see if the RTC has lost power. If the clock has lost power (such as a bad battery) it will reset the time to the date and time the program was compiled. This will provide a date and time, but it may be off by several minutes to several years! (We will deal with this in the next function call) If no loss of power is detected this is reported via the serial monitor, and then displays the current date and time. This information is also recorded to the System Log on the MicroSD card (more on this shortly).

Checking and Adjusting the Date & Time

As noted above, the RTC may display an incorrect date/time if power has been lost. You may want to have the date/time match what is on your computer. This function allows you to adjust the date and time.

When called the function displays the current date and time, and then asks the user if this is correct. If it is correct, the user enters a 'Y' into the serial monitor and the program continues. If the User enters 'N' then the program will ask the user to input the correct date and time. When the user enters the current minute and hits the enter button, that date and time is sent to the RTC.

The results of this function is reported to the System Log. It records whether the time was accurate or if it needed to be changed.

Setup the MicroSD Card Module

The next part of the `setup()` section is used to initialize and setup the MicroSD card module. This is another critical component of the Test Stand. All of the data is recorded on the MicroSD card. This allows you to import the motor test data into a spreadsheet or database. It also records system events and information on the motor being tested. Therefore it is very important that this component is working properly.

Check the Status of the Card

In the first part of this function we see the declaration of a local variable. In this case it is the chipset for the Mega2560 board. This is the only function that uses this variable, so we declare the variable locally instead of globally.

```
// ================================================================
// variable declaration
   const int chipSelect = 53;  //pin number for CS on Mega2560 board (D53)
```

Next the software performs a test to see if a MicroSD card has been inserted into the module or if that card is defective.

```
// ================================================================
// initialize Micro SD card
   Serial.println();
   Serial.println(F("Initializing Micro SD card. Standby..."));

  // see if the card is present and can be initialized:
     if (!SD.begin(chipSelect))
     {
      Serial.println(F("Card initialization failed, or not present. Replace
                        card and reset system."));
      Serial.println(serialLine);
```

If the card is missing or corrupt, the software will notify you two ways.

- A message will appear on the serial monitor indicating the failure with instructions to replace the card and restart the system (as seen above).
- The LED display will show "FAIL"
- The LCD display display an error message
- The LED lamp will turn red

The program stops execution at this point.

If the card module is initialized successfully, the serial monitor and the LCD screen will both display a successful initialization message. The program will then continue to the next phase of the MicroSD card setup process.

Setup the Logs on the Card

With the successful initialization of the MicroSD card, the software will begin to prepare the card to receive data. First it looks for the file "`fileMotorLog`" along with the last two number of the file name.

```
// create new file name - from altduino.ino
// create a new file
   for (uint8_t i = 0; i < 100; i++)
   {
    fileMotorLog[6] = i/10 + '0';
    fileMotorLog[7] = i%10 + '0';

    if (! SD.exists(fileMotorLog))
    {
      break;  // leave the loop!
    }
```

The first thing it does is start a loop creating a new file name. It does this by replacing the last two digits with numbers between 00 and 99. Once it finds it has created a file name that does not exist, it exits the loop and continues on.

The next step is to write the initial data to the card. It starts by opening the file and writing the first row of the CSV file. This row contains the headers for each column. These are constants that do not change. It does this by first assigning the names of each column header to the variable `dataString`. Each column header is listed in order with a comma between each header name. In the sample code we start with the Time Stamp and then we have header names of "Load Cell Reading (n)" and "Total Impulse (n)". The `dataString` is then written to the card and the Motor Log file is closed.

The same process is then followed for the System Log and the Information Log. Both logs use the same file number as the Motor Log, making easier to link the 3 files for a single test.

Finally, the program has been holding on to information about the Real Time Clock, the Time Sync information and even the MicroSD card that need to be added to the System Log. This information is now recorded to the MicroSD card. Finally, the software reports on the successful initialization of the MicroSD card component.

When the code is finished in this function it encounters the closed brace and returns back to the `setup()` function. This completes the setup function and the software continues on the `loop()` function.

Initialization of the BME280 Weather Sensor

The software is now ready to start initializing the various sensors on the test stand, with the first being the BME280 weather sensor. We know that the performance of a model rocket motor can be influenced by the weather. This is one reason why the NAR Standards requires all rocket motors being certified are performed in a stable consistent environment. But we often fly our rockets in weather that is not the same as that during certification testing. This sensor reads

barometric pressure, temperature, and humidity. This is recorded at the time of ignition and allows the user to identify changes in performance.

During the initialization process the software checks for the presence of the sensor. If it doesn't find the sensor it displays errors on the three displays that are available. If it finds the sensor it reports that the sensor has been initialized. The results are reported in the system log.

Testing the HX711 Load Cell Amplifier

The Load Cell requires an amplifier to make the pressure readings on the load cell visible. The HX711 load cell fills this role. During setup, the amplifier is tested for proper function. If it completes the process successfully, this is reported on all three screens. If it is not successful, the errors are reported. The results of the test are reported in the system log.

Initialization of the Fire Control System

Unlike the other setup functions, this function does not involve a sensor. Instead it is setting up the Arduino pins to watch for the pressing of various buttons on the Remote Head. This is accomplished by setting the pins for the Start button and the Fire button. The pin for the Fire relay (located at the test stand) is set up as well. Once this is completed it is reported on the screens and the data is reported to the System Log.

All Systems Initialized

At this stage of the process all of the systems have been initialized and are ready for use. This function reports the success on all screens and makes a notation in the system log. It also lets the user know that they can now press the green button on the remote head to start the process for motor test.

This completes the `setup()` function. Next is the `loop()` function which will we cover in the next chapter.

08
The Loop() Function

The `loop()` function is where the typical Arduino program 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.

void loop() function

Below is the entire `loop()` function for the Test Stand.

```
void loop()
{
  // ====================================================================
  // Show time on the LED display
     ledClock();

  // Clock updates display once every second to reduce flicker
     currentMillis = millis();
     if (currentMillis - lastExecutedMillis >= 1000)
     {
      lastExecutedMillis = currentMillis; // save the last executed time
      lcdDateAndTime();
     }

     // Check for button selection
       if (digitalRead(pinCalibrate) == LOW) loadCellCalibrate();
       if (digitalRead(pinStartButton) == LOW) motorPrepAndTest();
}
```

The loop() function starts by calling the function `ledClock()`. This function displays the current time on the 4-digit, 7-segment LED on the remote head.

The next section updates the LCD screen once a second. It performs this function by getting the current time in milliseconds. If less than 1000 milliseconds (1 second) has passed, it exits this section. If however, more than 1000 milliseconds has past, it will update the current time on the LCD screen. The current time is saved in the variable `lastExecutedMillis` and now this time is checked against the current time to see if more than 1000 milliseconds has past. This routine helps to reduce the flickering of the LCD screen.

The bottom section checks to see if the Calibrate button or the Start button has been pushed. When the calibration button that is selected, the program will jump to the calibration function and take the user step-by-step through the process of calibrating the load cell.

When the Start button is pushed, it begins a series of function calls that takes the user step by step through the test process. This includes entering data about the motor, the actual firing of the motor and collecting data, along with results data.

Until one of these two buttons is pressed, the loop function will continue on, updating the time in both clocks.

09
LEDs, Buzzers, Clocks and Displays

Before we drill down into the functions that allow the test stand to function, we want to look at the supporting functions. These functions (LEDs, buzzer, etc) are used by a number of other functions throughout the program.

The RGB_LED_Lamp_Settings Tab

The first set of functions I am going to discuss are located on the RGB_LED_Lamp_Settings tab. These lamps are used to provide both information and warnings about what is occurring around the test stand. Unlike most of our other tabs that contain just a single function, this tab contains a number of functions. However, all of these functions relate to the display of the LED lamps on the test stand.

RGB Colors

The first four functions determine the color of the RGB LED bulbs. The last three functions determine the state of the bulbs. The code snippet below is for a single color RGB LED bulb

```
void rgbRed(void)
//Solid red lamp
{
  valueRed = 255;
  valueGreen = 0;
  valueBlue = 0;
}
```

The color displayed by the bulb is determine by the amount of light emitted by the three bulbs within the RGB LED. In the snippet below we see a similar set of codes, this time mixing red and green to create yellow.

```
  valueRed = 255;
  valueGreen = 70;
  valueBlue = 0;
```

LED Lamp States

The next two functions create either a steady burning lamp, `rgbSteadyLamp()`, or a flashing lamp, `rgbFlashLamp()`. These functions are used to get the user's attention through the LED lamp. For example, if the MicroSD card fails to initialize, the following code for LED color and status is executed;

```
// Show steady red lamp
   rgbRed();
   rgbSteadyLamp();
```

The function call `rgbRed();` provides the values for the RGB LED bulb to display a red color. Then the function call `rgbSteadyLamp();` executes the code for the lamp to remain on in a steady fashion.

The final lamp state is used to add a strobe effect to the RGB LED bulbs. This mimics the warning strobes that are familiar around hazardous environments. In this case the function is used to get the user's attention when the test stand enters a 'yellow' or 'red' condition.

```
void rgbStrobeLamp(void)
{
  for (int x = 1; x < 50; x++)
  {
    analogWrite(RED, valueRed);
    analogWrite(GREEN, valueGreen);
    analogWrite(BLUE, valueBlue);

    delay(15);

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

    delay(500);
  }
}
```

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 can create an almost unlimited number of arrangements.

10
Writing Data to a SD Card

If you have read any of our previous Project Manuals where we used the A-PAM (Arduino Primary Avionics Module) this chapter will be familiar to you. However, what sets this project apart from the others is that here we are writing to three different files rather than just one. First I will look at the Information Data Log. Then I'll discuss the Motor Log and the System Log, how they are similar and how they differ.

Information Log

The Information Log is designed to record the responses the user provides to the questions asked during the preparation and shutdown phases of the test. This includes things like the location of the test, the size and mass of the motor, if a CATO occurred, etc. The data is stored as "plain text" which can be read by any text editor/word processor on any computer system.

Each time the user responds to a question you want to save that information. The method I use is saving it to a MicroSD card as a plain text file. Each time a new piece of information is provided the Information Log file on the MicroSD card must be opened, written, and then closed.

Opening and Writing the File

The first line of code opens the file and sets it up to have data written to it.

```
File fileInfoData = SD.open(fileInfoLog, FILE_WRITE);
```

If the file can be opened, it will continue to next step of writing the data.

```
if (fileInfoData)
```

The function where the question was asked has taken the response and assigned it to the variable dataString. The program will write the new data into the file. It then closes the file.

```
{
  dataString = sdInfo;
  fileInfoData.println(dataString);
  fileInfoData.close();
}
```

Error Reporting

If there is an issue and the file can’t be opened, several items will occur.

- The software will attempt to open the System Log on the MicroSD card
  `File fileSystemData = SD.open(fileSystemLog, FILE_WRITE);`

- It obtains the current time.
```
sdTimeStamp = String(timeMillis,10);
```

- If records the error to the System Log and closes the file.
```
// write error to sd card if possible
   sdMessage = "Info Data Log";
   sdResult = "Card Write Error";
   dataString = sdRTCStamp + "," + sdTimeStamp + "," + sdMessage + ","
                + sdResult;
   fileSystemData.println(dataString);
   fileSystemData.close();
```

- It indicates an error was encountered on the Serial Monitor.
```
// Show error on Serial Monitor
   Serial.println(F("Error opening Info data log..."));
```

- The 4-digit, 7-segment LED screen displays "FAIL"
```
// show FAIL on LED
   clockDisplay.setSegments(SEG_FAIL);
```

- The LCD screen displays the error message
```
// show error on LCD
   lcd.setCursor(0,0);
   lcd.print(F("Card Write Error"));
   lcd.setCursor(0,1);
   lcd.print(F("Opening Info Log"));
```

Regardless of whether the program was able to successfully write the data to the MicroSD card, the software takes a 1 millisecond delay to allow the Arduino time to reset. The function then returns to the function that made the call to write the data.

One thing that does not occur with this error is that the program does not stop. It records the error, but then continues on. It is possible that despite having an error writing to the MicroSD card, the software may be able to write the other data to the MicroSD card without issue.

Motor Data Log

This log records the force information of the motor while it is firing. The information is stored in a type of plain text file called "CSV" or "Comma Separated Values". In this format, each data point is separated by a comma. This is a common file format that can be read by just about any spreadsheet or database program on any computer system.

The function starts off in a similar fashion as the Information Log, by opening the file to the MicroSD card for writing. If the file opens and can be written to, the function deviates from the Information Log function.

In the Information Log, the data point was ready to be written to the MicroSD card. Here, the function must first get the data that from previous functions. This data has been stored in global variables:

```
sdTimeStamp = String(timeMillis,10);
sdImpulse = String(scaleNewtons,10);
sdTotalImpulse = String(impulseTotal,10);
```

Once it has the data points, it then concatenates them into a single variable called dataString. Now like in the Information Log, the data is written to the MicroSD card:

```
dataString = sdTimeStamp + "," + sdImpulse + "," + sdTotalImpulse;
fileMotorData.println(dataString);
fileMotorData.close();
```

If the file on the MicroSD card cannot be opened or written to, the software follows the same error routine we saw in the Information Log. The error is displayed on the LCD screen, in the Serial Monitor and written to the System Log. The software then continues as normal.

Lastly, as we saw in the Information Log, there is a 1 millisecond delay to allow for a reset.

System Log

The System Log collects data on the operations of the test stand system. The function for recording these data points operates in a similar fashion to the Motor Log. Both are CSV files. Both need to get the data points from other functions within the program and concatenate them into a single data point.

The System Log uses the same coding for error reporting that we saw used in both the Information Log and the Motor Log. The coding uses the same 1 millisecond reset pause and the software continues running after the error is reported.

Despite using three different logs to record various aspects of the testing process, the basic coding for all three functions is pretty much the same. The use of plain text for all three files allows the data to be highly portable. It can be read by numerous programs on all types of systems.

11
Motor Prep Sequences

When you press the green 'Start' button, it begins a series of function calls that ultimately end with a completed motor test. Along the way the user will enter information about the motor, the test site and more.

Getting Started

Once the start button has been pressed, the first thing the software does is check to make sure it is an intentional press of the start button. It does this by checking to see if the pin for the start button is still low. If instead it finds the pin has returned to HIGH, it assumes it was an errant signal and returns back to the main loop. Otherwise it continues on with the current function.

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *              MOTOR PREPARATION AND TEST SEQUENCE              *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepAndTest(void)
{
  // if Start button is HIGH assume rogue button push
     if (digitalRead(pinStartButton) == HIGH)
     {
       return;
     }
```

Test Preparation

The next section is a series of function calls that are all part of getting prepared for the motor test. Much like the `setup()` and `loop()` functions, this section conducts eight function calls. The majority of these function calls are used to get information about the test from the user.

Motor Preparation Information

The first function we jump to is `motorPrepInfo()`. The program then lets the user know what is going on. It does this through the LED segments, the LCD screen and the Serial Monitor.

At the beginning of the is local variables to spell out "PREP" on the LED screen and then it displays that on the LED segments. Next it displays the Motor Preparation notice on the LCD screen. Once this has been performed, it is time to start getting information from the user.

Before we collect any inputs from the user, we clear the serial buffer. We don't want any errant data to mess up our data collection.

```
// clear serial buffer
     while(Serial.available())
     {
      char getData = Serial.read();
     }
```

With the serial buffer clear we can now ask for information. This is performed through the Serial Monitor. It will display a series of questions on the Serial Monitor using the `Print()` statement. The first question being asked is the location of the test.

```
// Display instructions on Serial Monitor
   Serial.println();
   Serial.println(F("Motor Data Entry: Location Information"));
   Serial.println();
   Serial.println(F("Enter the location where the test is being
                  conducted."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();
```

You might have noticed the use of (F) in the print statements above. This is the "F macro" and on a basic level, it is used to help save memory. When you only have small amounts of memory to work with, any methods used to save some is sought after. The F macro you can only be used on strings, and it can only be used on strings that do not change - there cannot be any variables in the string.

Once the question has been displayed on the screen, we need to get the input from the user. The next section of code performs this task.

```
// Get location for the motor test
   resumeProcedure = false;

   while (resumeProcedure == false)
     {
      if (Serial.available() > 0)
       {
        sdInfo = Serial.readString();
         {
          if (sdInfo != "")
           {
            Serial.print(F("Location entered: "));
            Serial.println(sdInfo);
            Serial.println();

            // Info Log for Test Area Cleared
               sdInfo = ("Test Location: ") + sdInfo;
               writeInfoDataToCard();

            resumeProcedure = true;
```

```
            sdInfo="";
          }
        }
      }
```

We need to wait for the user to enter some data and hit the enter key. We do this by setting the variable `resumeProcedure = false`. As long as this variable is false, it will not go past the 'while' loop.

Next we check to see if anything has been entered into the serial monitor. If data is there, `Serial.available` will be greater than 0 and we will read what has been entered and place this in the variable `sdInfo`. We then check to make sure that there is something there to read `(sdInfo != "")`, and if data is there is we display back to the user what they entered. The software then writes this data to the MicroSD card by calling the function. With this question answered and the data stored on the MicroSD card, we can let the program continue to the next section by setting `resumeProcedure` equal to true. We also clear the variable `sdInfo`.

The remaining questions in this section follow the same basic procedure

- Display the question on the serial monitor
- Set up the 'While' loop
- Get the information from the Serial Monitor input
- Display what was entered
- Write the data to the MicroSD card
- Exit the 'While' loop and clear all variables

Using this basic setup, this section will ask the following two questions:

- Test stand elevation
- Name of manufacturer of the motor

That completes this section and the program returns back to the `motorPrepAndTest()` function.

Motor Casing Information

The next function call is to `motorCasingInfo()` which will obtain data on the motor casing. If you look at the coding in this function, you will see that it follows the same pattern that we saw in the Motor Preparation function. As before, the responses to these questions are written to the MicroSD card.

In this function, the following motor data is collected:

- Case length
- Case diameter
- Condition of the case
- Mass of the case

Once this function is complete, it returns back to the `motorPrepAndTest()` function. From there it jumps to the next function which looks at the rated impulse of the rocket.

Impulse Information

This function is designed to perform a single task - to obtain the total impulse rating stamped on the motor casing. Our other functions allowed for any text entry to be submitted, but here we require a specific entry. The code is going to look at what the user entered and decide if it meets our criteria. If it doesn't, it will wait until a proper entry is made. Once a valid entry is made, it will finish the function.

When the function starts, it looks very much like our previous data entry functions

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                MOTOR PREP TOTAL IMPULSE SEQUENCE              *
 *                                                               *
 *****************************************************************
 *****************************************************************/
void motorPrepTotalImpulse(void)
{
```

One of the first differences you will likely notice is that the instructions advise the user to enter a specific character depending on the total rated impulse of the motor. The user is restricted to the numbers 2 and 4, as well as the letters A, B, C, D, E, F, G

```
// ===============================================================
// Get motor total impulse information

  // Display Total Impulse entry instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Total Impulse"));
     Serial.println();
     Serial.println(F("Enter the letter for the total impulse of the motor
                       being tested."));
     Serial.println(F("  - For 1/4A motors, enter '4'."));
     Serial.println(F("  - For 1/2A motors, enter '2'."));
     Serial.println(F("  - For all other motors (A-G), enter the letter
                       designation."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();
```

Now we are back to familiar territory. We see the `resumeProcedure` in use with a 'while' loop being set up. We also see the familiar code to see if anything is available from the serial monitor, and if data is present it is read.

```
  // Get total impulse of the motor being tested
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
```

```
        {
          charInput = Serial.read();
```

The next part of the code is completely different from anything we have seen thus far. What follows is a series of '`if/else if`' statements

```
if (charInput == '4')
{
  motorTotalImpulse = 0.625;
  sdImpulseTotal = "1/4A";
}
else if (charInput == '2')
{
  motorTotalImpulse = 1.25;
  sdImpulseTotal = "1/2A";
}
else if (charInput == 'a' || charInput == 'A')
{
  motorTotalImpulse = 2.5;
  sdImpulseTotal = "A";
}
else if (charInput == 'b' || charInput == 'B')
{
  motorTotalImpulse = 5;
  sdImpulseTotal = "B";
}
else if (charInput == 'c' || charInput == 'C')
{
  motorTotalImpulse = 10;
  sdImpulseTotal = "C";
}
else if (charInput == 'd' || charInput == 'D')
{
  motorTotalImpulse = 20;
  sdImpulseTotal = "D";
}
else if (charInput == 'e' || charInput == 'E')
{
  motorTotalImpulse = 40;
  sdImpulseTotal = "E";
}
else if (charInput == 'f' || charInput == 'F')
{
  motorTotalImpulse = 80;
  sdImpulseTotal = "F";
}
else if (charInput == 'g' || charInput == 'G')
{
  motorTotalImpulse = 160;
  sdImpulseTotal = "G";
}
```

This 'while' loop uses a series of 'if/else if' to see if the user has entered a valid character. If no valid character is entered, the program stays in the loop waiting for a valid entry.

Once a valid character has been entered, we see values assigned to two different variables. The first variable, `motorTotalImpulse`, stores the rated total impulse value as a number. This number will be used later to calculate total burn time.

The second variable, `sdImpulseTotal`, stores the rated total impulse as a character. It will be used later to create the total ratings for the motor.

To finish this function, the user data is displayed on the Serial Monitor as we have seen before. The function being complete it returns back to the `motorPrepAndTest()` function, where it will jump to the next function in the list.

Average Thrust

The next data point we need to obtain is the average thrust for the motor. This is printed on the side of the motor itself. This data is collected through the `motorPrepAvgThrust()` function. This function gets the user input using the same technique we have seen previously. Once the data is entered, it is stored in two variables - `motorAverageThrust` is the data entry variable and `sdThrustAvg` is used to write the data to the MicroSD card. The data entry is displayed and the program returns to the `motorPrepAndTest()` function.

Delay Time

The next data item is also printed on the motor casing and it is the delay time. The function motorPrepDelayTime() is used to collect that data using the same procedures that were used earlier. However, near the end of the function we have a line of code that concatenates the Total Impulse, the Average Thrust and the Delay Time into a single data point. This matches the same format that is imprinted on the motor itself.

```
// Now that we have the full motor code we can write it to the Information
   text file
     sdInfo = sdImpulseTotal + sdThrustAvg + "-" + sdDelay;
```

With this new data point now created, it is written to the MicroSD card.

```
// Info Log for Test Area Cleared
   sdInfo = ("Motor Classification: ") + sdInfo;
   writeInfoDataToCard();
```

The function being complete, it returns back to the `motorPrepAndTest()` function.

Propellant Data

The next set of data points all deal with the propellant being used and it is collected in the `motorPrepPropellant()` function. Here we collect data on the

- Type of propellant used
- Propellant mass
- Motor date or lot code
- Type of igniter used

Lot codes are typically stamped on the motor. The type of propellant used in the motor and the propellant mass will be found in the documentation from the motor manufacturer. This data is collected just like the in the previous functions, so nothing new or unusual here.

Ignition Time

Following the data entry for the propellant, the next item deals with the length of time to keep the relay open that sends current to the igniter. This is located in the `motorPrepIgnitionTime()` function. The user is asked to specify a time limit between 3 and 10 seconds. The program uses the same procedure as before to get the user's input.

Once the user enters a number, the software does something a bit different from the other inputs. First it looks to see if the value entered is less than 3 seconds or if it is greater than 10 seconds. If the input is not within that range it sets the ignition time to 5 seconds.

```
if ((timeFireRelay < 3) || (timeFireRelay >10))
timeFireRelay = 5;
```

Following this it displays the entry as expected. It does not report this to the MicroSD card. Now finished it returns back to the `motorPrepAndTest()` function.

Calculating Data Collection Time

The `motorPrepCalcDataTime()` completes the last function call as part of the motor preparation section. This function will determine length of time that data should be collected. We don't need the system to be collecting data while the motor is sitting on the test stand and not doing anything. This function determines the length of time needed to collect data based on the burn time of the motor, the delay time of the motor and the relay time.

The function starts by declaring two local variables

- `timeBurn`
- `timeMotor`

First the program calculates the burn time of the motor by dividing the Total Impulse by the Average Thrust. This is stored in the variable `timeBurn`.

```
timeBurn = (motorTotalImpulse/motorAverageThrust);
```

Next it adds together the motor's Delay Time and the Firing Time.

```
timeMotor = (motorDelayTime + timeFireRelay);
```

To determine the total time, the program first adds `timeBurn` to `timeMotor`. This is multiplied by 1000 to to derive a total time in milliseconds. Next an additional 1,000 milliseconds (or 1 second) is added to the total.

```
timeDataCollection = (((timeBurn + timeMotor) * 1000) + 1000);
```

This becomes the total amount of time we will collect data in milliseconds and is stored in the global variable `timeDataCollection`. The clock starts the instant the relay opens and power flows to the igniter. It will continue to collect data until the time period specified in the `timeDataCollection` variable is reached.

As before, the data collection time is displayed and then the software returns to the `motorPrepAndTest()` function.

12
Motor Load Sequence

The next set of functions deal with the loading of the motor on to the test stand, as well as the pre-fire process. These functions are all called from the `motorPrepAndTest()` function. This is a continuation of the function calls that we looked at in the previous chapter.

Getting Started

In the `motorPrepAndTest()` function we transition from the preparation phase to the loading phase. This is seen in the code as a new header is displayed on the Serial Monitor.

```
// ============================================================
// Show header on Serial Monitor
   Serial.println();
   Serial.println(serialLine);
   Serial.println(F("Begin Motor Loading and Pre-Fire Process"));
   Serial.println(serialLine);
```

The program then calls for the next function, `motorLoad()`.

Motor Load Checklist Function

The first thing part of this function is used to display the updated status. We see the declaring of local variables that are used only within this function. These variables are used to spell out "LOAD" on the 4-digit, 7-segment display on the remote head.

The next action that occurs is changing the color of the test stand's RGB LEDs. They are now turned to yellow and will glow steadily.

```
  // Turn test stand LEDs to Yellow
     rgbYellow();
     rgbSteadyLamp();
```

Next we see that the clear white LEDs are turned on as strobes. We use the library MsTimer2 to provide continuous flashing of the strobes, despite whatever else the code may be doing. This allows us to set the timing and forget it until it needs to be changed.

```
  // Turn on white strobe lamps
     MsTimer2::set(500, ledWhiteStrobeLamp); // 500ms period
     MsTimer2::start();
```

The 4-digit 7-segment display displays "LOAD" using the variables declared above

```
// show "LOAD" on clock
   clockDisplay.setSegments(SEG_LOAD);
```

The last display to be updated is the LCD screen. This is used to indicate that the motor loading process is now underway

```
// show loading sequence in progress on LCD
   lcd.setCursor(0,0);
   lcd.print(F(" Motor Loading  "));
   lcd.setCursor(0,1);
   lcd.print(F("  In Progress   "));
```

The serial monitor is used to display a motor loading checklist. It uses the same procedure that we have seen elsewhere in previous functions to display the information on the screen. At the end of the checklist, it ask the user to enter the letter 'V' to verify that the checklist has been completed. The program will wait until the letter 'v' is entered and it will accept either an upper case or lower case 'v'.

```
if (charInput == 'v' || charInput == 'V')
```

Once the verification has been enter the time of the verification is displayed on the serial monitor and it is written to the System Log on the MicroSD card. Once again the program returns to the `motorPrepAndTest()` function to get the next instruction.

Motor Mount Checklist Function

The motor mount checklist function is nearly identical to the last part of the motor load checklist function. Both display a checklist to the user, and both ask the user to enter a 'v' into the serial monitor to verify the checklist is complete. Both also write to the System Log when the checklist verification is complete.

Load Cell Calibration Decision

The next function (`motorRecalibrate()`) looks at whether the load cell needs to be calibrated prior to the test firing. There are two ways to calibrate the load cell - the first way is to hit the 'CAL' (calibrate) button on the remote head prior to starting the test and the second method is to perform the calibration just prior to the test firing.

Using the same procedures as before, the user is asked if they want to calibrate the load cell (hit 'c' and Enter) or bypass the calibration process (hit 'b' and Enter). The user input is recorded to the System Log on the MicroSD card. If the user decides to calibrate the load cell, the program will jump to the function `loadCellCalibrate()`.

Load Cell Calibration

The function `loadCellCalibrate()` performs the calibration of the load cell and offers the option to store that data in the EEPROM. It begins will declaring the variables to display CAL on the 4-digit 7-segment LED display, as well as variables used to the calibration calculations.

```
// declarations for calibration
   float knownMass;
   float newCalibrationValue;
```

```
boolean performTare;
unsigned long timeStabilizing;
```

The function will lead the user through the calibration process. If provides instructions through the Serial Monitor and LCD screen as we have seen done previously. The software also records the time the calibration process is started in the System Log and stored on the MicroSD card. Once the tare is complete, it needs the weight (in grams) of a known object that is placed on the load cell. This is also recorded in the System Log and stored on the MicroSD card. With the data entered, the load cell library is able to calculate a new calibration value. Once again, this is recorded in the System Log and stored on the MicroSD card.

The user is then asked if the new value should be saved to the EEPROM (Electrically Erasable Programmable Read-Only Memory). This is memory that is stored and stays on the chip even when power to the Arduino is turned off. If the user enters 'y' then the new value is saved and used during subsequent tests. If the user answers 'n' then the currently stored value in the EEPROM is used. This decision (and calibration value) is another data point that is stored in the System Log.

When this has completed, the user is advised to remove the weight from the load cell. The program returns back to the `motorRecalibrate()` function, and then back to the `motorPrepAndTest()` function.

Clear Test Area

The `motorClearArea()` is the next to last function prior to starting the Fire sequence. It is very similar to the other functions we discussed previously.

It begins with the variable declarations for the 4-digit 7-segment LED to display "TEST". The RGB lamps are changed to red (the strobe is still flashing from earlier). The LCD screen warns to clear the test stand area. The checklist for clearing the area is displayed on the serial monitor, and the user is prompted to enter 'V' (for verified) when the checklist is complete. Once entered, the time is written to the MicroSD card and the program returns to the `motorPrepAndTest()` function.

Setting Up The Load Cell

The very last function in the preparation phase is the `motorScalePreparation()` function. Here the load cell is prepared to collect data. The serial connection is cleared, the load cell is started, the calibration value is accessed from the EEPROM and the load cell is updated with that value.

This concludes the preparation phase of the test. The program will now transition into the firing sequence.

13
Fire Sequence

The Fire Sequence controls the actual firing of the solid rocket motor. It also uses other functions to collect data on the test, the post firing sequence and what to do when an abort occurs. Before the actual fire sequence begins we need to return one last time to the `motorPrepAndTest()` function.

Transition to the Fire Sequence

When the program finishes the `motorScalePreparation()` function it returns to the `motorPrepAndTest()` function. Here is where the transition begins. It starts by displaying a notification on the Serial Monitor that the Motor Test Fire Process is beginning. It then calls the `fireSequence()` function to start the test fire phase.

Fire Control Sequence

The program enters the `fireSequence()` function and it starts by declaring the local variables to display the 5-second countdown in the 4-digit 7-segment LED screen. These are declared locally as they are only used in this function, and this helps save memory.

The software begins by getting the current time that the Fire Control Sequence is started and writes this information to the System Log. The LCD screen on the remote head indicates that we are standing by for the Fire Sequence (countdown) to begin.

On the serial monitor, a short description is provided of the three columns of data that will be presented as the test firing occurs. This is followed by a second set of instructions on performing the actual firing and how to abort the test.

Waiting for the Fire Button to be Pressed

The program then enters a 'while' loop. We have seen this used previously where we waited for a user to input information into the Serial Monitor. This time, we are waiting for the user to press the Fire button. We do this by checking the status of the `pinFireButton`, which is Pin 7 on the Mega2560 board. The `pinFireButton` was set to the builtin pull-up resistor on the Mega2560 board during the `setupFireControl()` function.

```
// Setup Fire pin for interrupt
   pinMode(pinFireButton, INPUT_PULLUP);
```

When the Fire button is pressed, it will return a 'LOW' reading. Once the software reads the change in reading, it will exit the 'while' loop and continue.

Fire Button Pressed

With the button pressed, the software notes the time and displays that the countdown has been initiated on the Serial Monitor. It also records this information to the System Log. Finally the LCD screen displays that the countdown is started.

The software now enters an if/else if loop to display the countdown. During each segment of the loop the countdown is displayed in the 4-digit 7-segment LED and a short tone is sounded from the piezo buzzer.

During each part of the loop the software checks to see if the Fire button has been released. It does this by looking at the `pinFireButton` and determining if the pin is showing '`LOW`' (the button is still pushed down) or '`HIGH`' (the button has been released). If it reads '`HIGH`' the countdown sequence is stopped and the program jumps to the `abortTest()` function (more on this later in this chapter).

```
// Check for ABORT
// Fire button must remain depressed during countdown otherwise an ABORT is
   called
     if (digitalRead(pinFireButton) == HIGH)
       {
       // jump to Fire Abort Sequence
          abortTest();
```

Weather Data

When the countdown reaches 0 (zero), a number of activities occur. The first activity is a call to the function `fireWeather()`. This obtains several environmental data points just prior to the firing of the motor. This includes the air temperature, barometric pressure and humidity. These three data points, along with the time, are then written to the system log. The program then returns to the `fireSequence()` function.

Warning Displays

The 4-digit, 7-segment LED display will change to "FIRE". The LCD indicates that power is flowing to the test stand. The serial monitor displays a similar message and includes the time. The same data point is also written to the system log. Finally, the piezo buzzer is turned on to provide an audible warning.

Supplying Power to the Test Stand

It is now time to supply power to the test stand. This is done by setting the relay pin to HIGH. This allows power to flow from the battery pack up to the igniter, igniting the solid rocket motor.

Once the relay has been opened and power is flowing, the program calls on the function `fireDataCollection()`. This will get the data from load cell. We'll take a closer look at the data collection process in the next chapter.

ABORT

One scenario that can occur during a test is an abort. This may be accidental (where the user's finger slipped off the button) or intentional (someone walked into the stay clear zone). When an abort is detected, the program will jump to the `abortTest()` function.

The primary focus of this function is to advise the user of the abort through messaging on the Serial Monitor, the LCD screen and the 4-digit 7-segment LED screen. The piezo buzzer will also sound to identify the abort. The time of the abort is recorded in the System Log.

It should be noted that an abort does not indicate that the test stand is safe. The motor is still in a state where it can be fired. For this reason the LED lamps still glow red and the strobes continue to flash. Because the motor is still in a condition where it can be fired, the software will jump to the `abortRecycle()` function, where the user can try again to perform the test.

Abort Recycle

This function is run right after the abort has taken place. It advises the user that the test can be restarted when they are ready. The recycle will return to the Clear Test Stand Area function so that the user can verify the area is clear before firing the motor. It assumes all of the data entered previously is correct and nothing has occurred during the abort that would require any changes to the data. This allows the user to continue the test without having to start over from the beginning.

Shutdown Period

When the data collection period has concluded the program calls the `fireShutdown()` function. This provides a one minute time period for users to maintain their distance from the stand. There are two reasons for this period:

- If there has been a misfire, you should wait one minute before approaching the test stand.
- Once the motor has stopped firing it is still hot and should be allowed to cool down before touching and removing the motor casing

The function starts off by assigning the variables for the 4-digit, 7-segment LED. Here we see the terms "STAY", "BACK", and "SAFE". These will be used later in this function.

The next bit of code turns the relay off. This is a safety in case the relay failed to turned off earlier. This is an important safety factor in the event of a misfire. You want to make sure power is no longer being sent to the igniter, and this extra line of code performs that function. The buzzer is also turned off at this time.

Wait Period

We now enter the section of code that runs the 60-second wait period. It starts by turning the RGB lamps to Yellow. Next it displays on the Serial Monitor that power has stopped flowing to the pad, and this is also recorded to the System Log.

Next, the user is informed via the Serial Monitor that a one minute post test safety period is beginning and everyone should stay clear of the test stand until the "All Clear" is given. This warning is also written to the Safety Log.

Wait Period Countdown

We now start two "for" loops that display the time left during this safety period. This "loop within a loop" allows the 4-digit 7-segment LED screen to alternate the display of "STAY" and "BACK", while the LCD screen displays time left in 5-second increments.

The first part of the loop uses the "`safety`" variable to countdown the overall time period of 60 seconds.

```
for (int safety = 60; safety > 0; safety -=10)
```

This lines starts with `safety` being equal to 60. As long as `safety` is greater than `0`, each time the loop comes back to the top it will subtract 10 from `safety`. Notice that no actual time is looked at in this loop.

Next we have two loops that work within the loop identified above. The first loop displays "`STAY`" in the 4-digit, 7-segment LED screen, while the LCD screens reports the time left in seconds based on the value of the variable "`safety`". This loop counts the variable "`x`" for 1 through 5, with a delay of 1000 milliseconds (or 1 second) between each increment.

```
// show data while looping for 5 seconds
   for (x = 1; x <= 5 ;x++)
   {
     delay (1000);
   }
```

When this loop is complete, it moves on to the next loop. This loop is very similar to the one described above, except it displays a "`BACK`" in the 4-digit, 7-segment LED screen. The LCD screens reports the time left in seconds based on the value of the variable "`safety`" with 5 seconds subtracted. This accounts for the 5-second loop executed above. Once again we have the same"`x`" for 1 through 5 loop.

With both of these loops now completed, it returns to the top of the primary loop. The variable `safety` will have 10 subtracted from its value, and these loops will continue until `safety is` equal to 0. At that point the countdown loop will exit and the code will continue.

All Clear

With the conclusion of the loops above, the wait period is over and the "All Clear" is signaled. The first thing we see is that the lights on the test stand are changed to green and the strobes are turned off. On the Serial Monitor the "All Clear" is displayed and the System Log records that the Post Test Safety Period is finished. The 4-digit, 7-segment LED displays the word "SAFE" and the LCD screen indicates it is safe to approach the test stand. Finally, a short buzzer tone is played to announce the updated status.

With the shutdown complete, the program returns to the `fireSequence()` function, where it continues to the next section of the code and calls the `postTestDataEntry()` function.

Post Test Data

With the bulk of testing complete, there are a few data points that still need to be collected. The `postTestDataEntry()` function uses the same method we have seen used earlier to solicit input from the user.

The first question asked is if a CATO occurred during the test. This would be a catastrophic failure of the motor. The user responds with a Y or N response. Next they are asked to enter in the mass of the empty engine casing following the burn. Finally, the user can enter any comments that they may have about the test. This is limited to a single line of text due to the limitations of the Serial Monitor. The response to these three questions is noted in the Information Log.

With these last questions answered, the testing session is complete. This is noted on the Serial Monitor and is recorded in both the System Log and Information Log.

The program now returns back to the loop() function and will display the date and time on the LCD screen and the 4-digit, 7-segment LED screen.

Followup Testing

So what if you want to perform another test of a similar or even different motor. You have two options:

- Press the Start Button and the program will begin asking the initial questions about the new motor and will repeat the testing process
- Press the Reset Button to reset the Arduino and start the process over, including the initialization of all of the sensors.

Our recommendation is to use the Reset option. This allows you to have a record showing that the system's components were tested and passed prior to testing, and all the data is kept in an organized fashion.

14
Test Data

The primary objective of the test stand is to collect data on the performance of the rocket motor. This data collection is performed in the function called (appropriately enough) `fireDataCollection()`. However, the function does more than collect the performance data.

The function starts with a lengthy comment section that identifies what the function will perform. It provides the formulas that are used. This information would be of help to other programmers who may want to update or modify the code.

Data Collection

The function `fireDataCollection()` is designed to primarily record the data from the load cell during the firing of the rocket motor. The function pulls two data points:

- The time stamp when the force data was collected
  ```
  timeMillis = micros()/1000000.f;
  ```

- The amount of force exerted on the load cell in Newtons. The force is reported in grams, and the software then converts grams to Newtons
  ```
  scaleNewtons = loadCell.getData() * 0.009806f;
  ```

It then does several calculations before it gets the next load cell force data point:

- Elapsed time since last data collection
  ```
  timeElapsed = timeMillis – timeLast;
  ```

- Current impulse in Newtons
  ```
  impulseSingle = scaleNewtons * timeElapsed;
  ```

- Total impulse in Newtons
  ```
  impulseTotal = impulseTotal + impulseSingle;
  ```

Next, it displays that data on the serial monitor and writes that data to the Motor Log on the MicroSD card.

The program then checks to see if the relay supplying power to the igniter needs to be turned off. If the elapsed time has exceeded the relay time period, the relay is closed by setting the relay pin to '`LOW`'. The buzzer is turned off by setting that piezo pin to '`LOW`' as well. This is recorded on the System Log and displayed on the Serial Monitor.

Finally, the software checks to see if the elapsed time to collect data has been exceeded. If the time has been exceeded it will display this on the Serial Monitor.

Data Collection Time

There is a lot going on in this function. It is one of the busiest functions in the entire program. Yet in testing we found that average time between load cell readings was 0.090 milliseconds. While this is fairly fast, you may want to have shorter times between readings. To accomplish this you can try several things.

- Eliminate any calculations during the data collection period.
  With this, all that is done is to collect the load cell reading in the native gram format along with the time stamp in milliseconds. After this raw data is collected it is imported into a spreadsheet and the calculations are performed there. This would include:
  - Convert grams to Newtons
  - Elapsed time
  - Single point impulse in Newtons
  - Total impulse

- Eliminate data display on the Serial Monitor
  Here you would no longer display the results of each data point collection on the Serial Monitor screen. Any data collected would only be available on the MicroSD card.

There are a couple of things that you can't eliminate. The first is writing the data to the MicroSD card. If you eliminate the Serial Monitor display only the MicroSD card has the data.

The next item you shouldn't eliminate is turning off the relay. This prevents damage to the relay. In testing there was no appreciable delay when the software was calculating the relay cutoff time.

Finally, the calculation to determine when to stop collecting data should remain intact. If the test stand is constantly collecting data even when the motor isn't firing it makes it much more difficult to determine when the actual test started and ended.

Code Review Conclusion

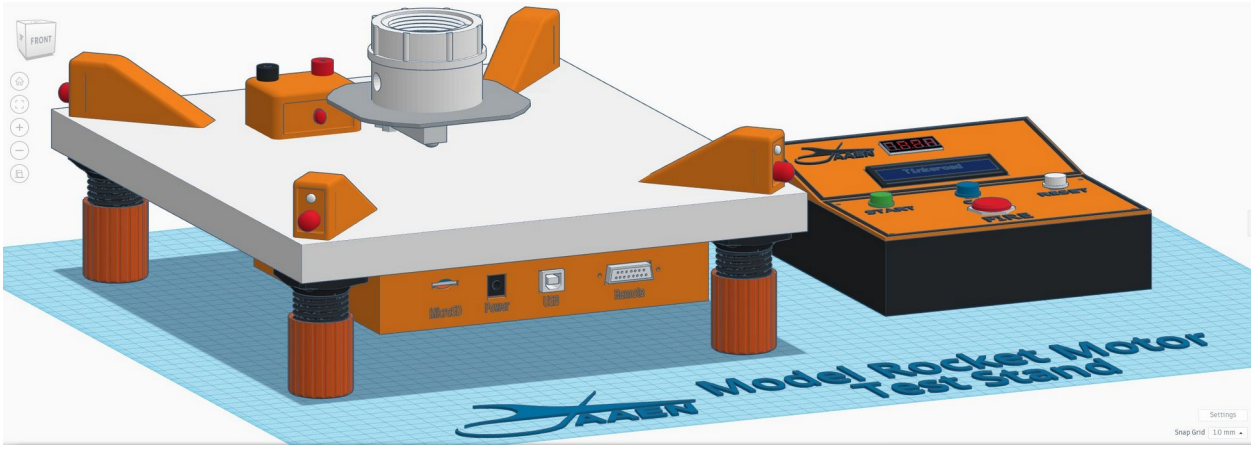
This concludes the review of the code for the Test Stand. I hope this has helped explain what the code does and why we made certain decisions. Hopefully I’ve been able to provide some ideas that will help with your own projects.

15
Building the Test Stand Base

With a working system using the breadboard, it was time to start designing the actual test stand. We had a general idea of what we wanted, and now it was time to go from concept to hardware.

Using the original drawing as a baseline, we were already sure of several things:

- The test stand will be basically about 12-inches square
- The load sensor will be mounted near the center of the test stand
- We will use PVC pipe to create the mounts for the motors, and these would be able to be exchanged using a threaded PVC pipe connection
- The majority of the electronics would be housed under the test stand and inside a 3D printed housing. This will help keep the electronics free from the debris released by a burning motor
- We will need a remote head to perform the actual firing duties to keep everyone a safe distance away from the test stand.
- We will need to interface with a computer to power the electronics and to perform the pre- and post-fire data collection.
- We will need a second power supply to provide the power to igniter the motor on the test stand



It is important to understand that the Test Stand is not a single unit, but actually several subassemblies. Each subassembly is built as a separate unit and then later incorporated into the system. In this chapter we will look at the construction of the platform used by test stand. In later chapters we will look at the motor mounts, the electronics located at the test stand and then the remote head. Finally we will look at how they all fit together.

Base

The base of our test stand was constructed from scrap countertop. During a renovation of our bathroom, we saved the circular scrap pieces cut out for the sink. This oval was squared off and become the platform of our test stand.

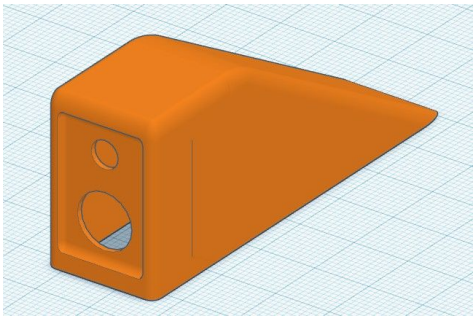
For your platform, you can use any type of material you wish. When selecting a material make sure it is sturdy enough to handle the demands you will be placing on it. Also, use a materials that you find easy to work with.

The most common is probably going to be plywood or MDF. When using wood or a wood type product, consider applying some type of protectant to the top and sides of the wood. Ours had a laminate on top, and then we filled in the sides with wood filler and painted them orange.

Spend some time on the base, as a number of items will be mounted here, and this is where the bulk of the abuse from the testing will take place.

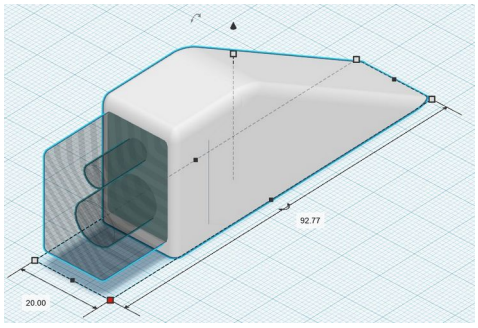
Warning Light Housings

There are four Warning Lamp Housings, with one located on each corner of the base. The housings contain a 10mm RGB LED bulb and a 5mm white LED bulb.

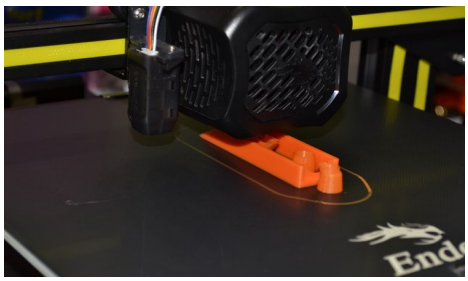


The lamp housings were designed using Tinkercad. I chose to place the white LED (which will act as a strobe lamp) above the larger RGB LED lamp. The inside of the housing was hollowed out using the 'hole' function.

I didn't want just a plain, flat face on the housing. I decided to add a bit of flair by indenting the face of the housing slightly. This was done using the "hole" function too. A box was created slightly smaller than the face. The ends of the box were rounded and then inserted slightly into the face. When everything was merged, we had the stylish indent we were looking for. In fact, we liked the way this came out so well we used the same technique on the continuity lamp housing.

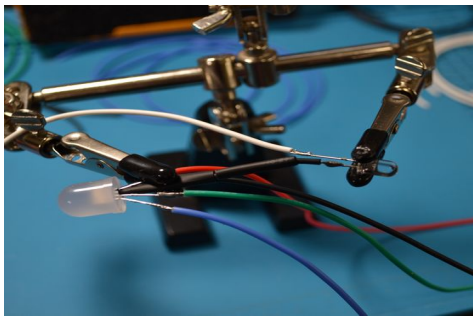


The finished design was exported as an STL file from Tinkercad. This was imported into Cura for slicing. For this print I use a layer height of 0.12mm (Super Quality) as I wanted a really nice smooth print. I chose to print it in a normal orientation. This requires that supports be provided inside the housing. I used tree supports and this worked really well.



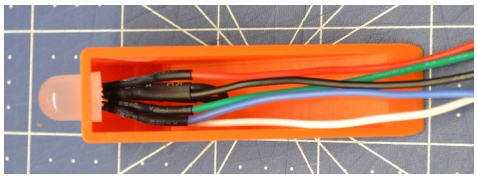
The housings were printed using PLA+ filament in an orange color. We printed the housings one at a time and when complete I was very pleased with the look and finish.

Wiring the Lamps



The LED bulbs needed to be wired within the housing before the housings could be attached to the base. This involved soldering a red, green and blue wire to the corresponding leg of the RGB LED bulb. A white wire was soldered to the white LED leg, and a black wire was soldered to the common ground on both bulbs.

With the bulbs soldered, we attached them to the breadboard and did a quick test to make sure all the connections were good. With that confirmed, heat-shrink tubing was applied over each connection.

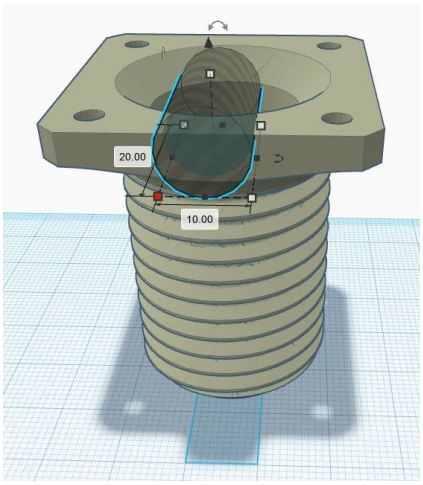


Next, the lamps were inserted into the housing. The strobe lamp is at the top, with the larger RGB LED underneath. The common ground wiring between the two bulbs must be curled around. Use care as you perform this to prevent breaking the connection. With the bulbs in place in the housing, they were tested again. With all the bulbs working, a small amount of glue is applied to the inside of the housing to hold the bulbs in place. Finally, 100Ω resistors are soldered onto each RGB LED wire and to the white LED bulb. No need for a resistor on the ground wire. Make sure you leave plenty of wire coming off these lampsas it has to stretch down the sides and then into the electronics housing.

Adjustable Legs

The legs I used are 3-printed. The basic design for the legs came from the post "Adjustable leg Furniture" by makarov_dimas on Thingiverse (https://www.thingiverse.com/thing:6450953). He has two separate designs, as well as multiple lengths.

I initially chose the 50mm Cap and Support design (the Cap and Support is on the left in the picture). The 50mm height was selected to allow plenty of clearance for the electronics housing that would be underneath the platform. I printed one leg to test the size and fit. Given my rough estimate of the electronic housing size, I decided to reduce the size of the leg and cap to 80%. This made the attachment base of the leg small enough to fit on each corner of the stand and not interfere with the electronics housing. The height was reduced to about 46mm which was still tall enough to allow plenty of clearance for the electronics box.



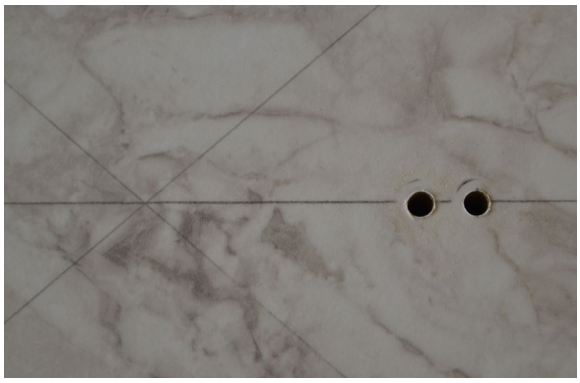
The final issue with the legs involved the position of the light housings and the wiring for the lamps. The opening for the light wires would be directly over the center of the adjustable leg base. The base already had an opening at the top, but there was no where for the wire to exit. To solve this issue, the STL file was uploaded into Tinkercad, and a small round 10mm diameter opening was added. This would be large enough to allow the 5 wires to pass through.

With this last design change, we imported the modified STL into Cura. Quality was set to a layer height of 0.16mm (Dynamic), with no need for supports. Four legs and caps were printed using PLA+ filament. We printed the legs in black, with the caps in orange.

Load Cell

The final piece to be attached to the base is the Load Cell. The Load Cell will sense the amount of force placed on it when the rocket motor fires. I wanted to make sure that the load cell assembly was centered on the base, was mounted solidly, and could use different motor mounts to test different size motors.

Load Cell Location & installation

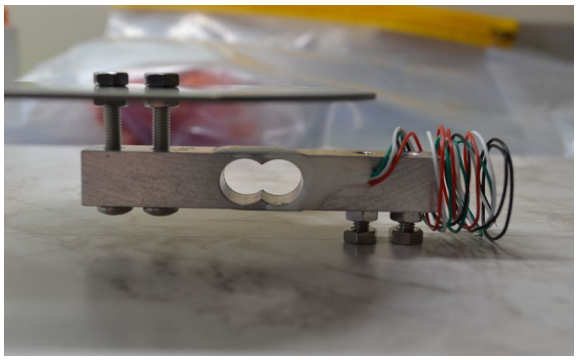


To locate the center of the platform I drew and 'X' from the opposite corners. Where they meet will be the center of the board. Next I drew a line from center to the edge using an 'L' square.

There are two threaded screw holes on each end of the load cell. One set is hear where the wires are attached. This end will be secured to the platform. The other two ends are the floating part of the load cell. This is where I will attach the motor mount. Using this end of the load cell, I placed the X between the two screw holes. I then marked on the platform where the opposite end screw holes were located. This is where I will drill the holes for the mounting screws.

I had decided that my motor mounts would be created using PVC pipe. I had scrap pipe available and the use of the screw-on ends would make changing motor mounts fairly easy.

I then needed a way to mount the female threaded PVC pipe to the load cell. I need something that is strong and that will hold up to the ejection charge of the motor. I decided to use a standard electrical box cover. This would be attached to the load cell using screws. I found the center of the cover and then marked where the mounting holes would be drilled.

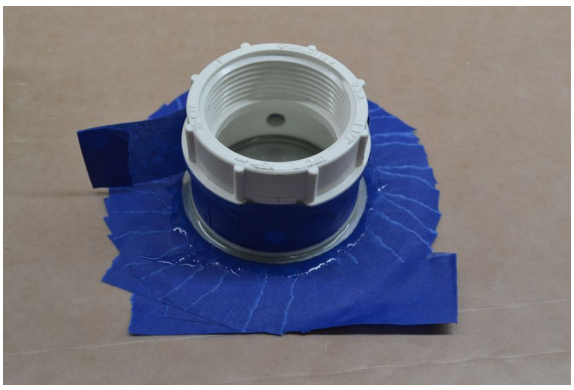


With the mounting holes drilled in the platform and the motor mount base, I put the two together. The screws for the motor mount platform were inserted from the bottom of the load cell (the load cell will have an arrow showing the direction of the force). With these screwed in, I can bring in the base screws from under the base and loosely attached nuts to hold the screws in place. A second set of nuts is put into place, and then the screws are inserted into the load cell. Once in position, use the nuts to secure the load cell.

In a similar fashion nuts are added to the screws for the motor mount. These are use to support the motor mount platform. A second set of nuts is attached to the end of the screws to hold the platform in place.

Motor Mount

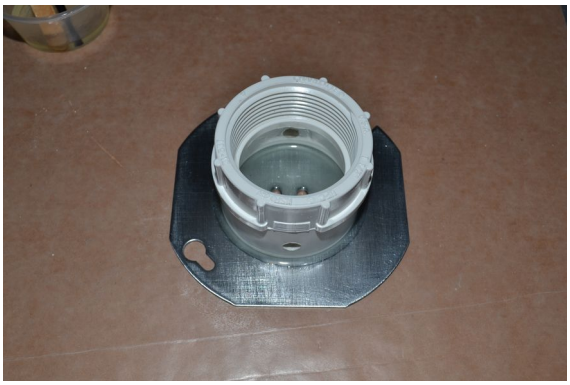
Before mounting the PVC pipe connector to the metal plate, drill a hole on opposite sides near the bottom. These will be used to allow any ejection gases to escape.



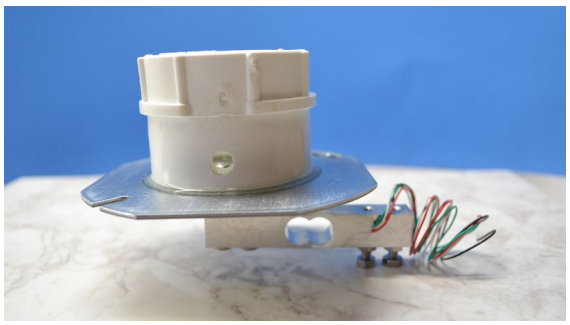
Place the PVC connector centered on the plate. Use a pencil to mark where the outside diameter of the PVC. Remove the connector.

To permanently attach the connector to the plate, I used 30 minute epoxy. This would give me plenty of working time and provide a strong bond. Begin by using sandpaper to rough up the metal plate. This will give the epoxy something to grip in to.

Next, using the pencil mark as a guide, apply small strips of painters tape about 1/8-inch away from the line. Do this around the entire circumference of the connector. Now apply a strip of tape to the bottom of the connector, leaving about a 1/8-inch gap at the bottom.



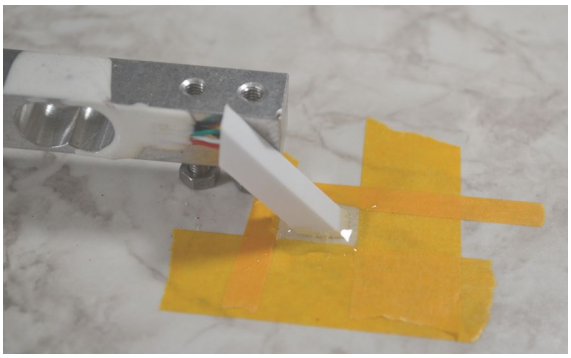
Mix your 30-minute epoxy according to the manufacturer's instructions. Apply and even coat to the entire exposed area of metal plate (this extra coat of epoxy will help protect the plate from the ejection charge). Once you have this area covered, press the PVC connector into the epoxy, centering it in the opening. Once in place, remove the blue painters tape. Let the epoxy harden completely.



Once the epoxy has cured the motor mount platform can be attached to the load cell. Use the two nuts to secure the platform to the load cell.

The last item was the routing of the wires. I didn't want them out in the open. These wires are very thin and can easily be damaged.

I drilled a hole near the end of the load cell, along the same side as the load cell wires. This would be used to route the wires to the electronics housing.



To provide some protection for the wires, I cut a small piece or rectangular plastic tubing. It was cut at a 45-degree angle on each end. The wires are threaded through the tube and into the hole in the base.

To secure the plastic tube, I use 30 minute epoxy. Modeling tape is used to clean up the edges of the epoxy. Apply the epoxy around the base, making sure not to get any on the wires or into the hole. Remember to remove the tape before the epoxy sets.

Base Complete

This completes the primary base assembly. At this point you have what looks a bit like a test stand, with a bunch of wires hanging out. Set this assembly aside as next we begin construction of the motor mounts.

16
Building the Motor Mounts

With the basics of the test stand base completed it is time to start work on assembling the motor mount components. The motor mounts use PVC pipe to allow the various size motors to be used interchangeably on the test stand. If you have built a model rocket in the past and glued together a motor mount, much of this will be familiar. The main difference is the use of 3D printed centering rings instead of cardboard or wood rings, and the use of PVC pipe instead of a paper body tube.

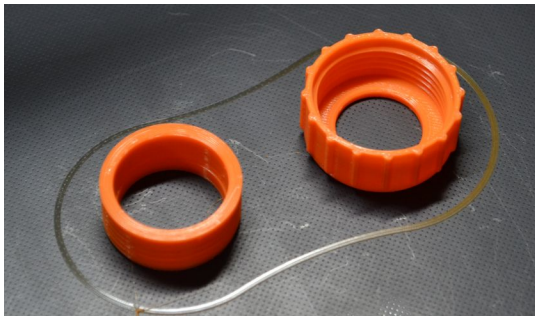
Printed Motor Mount Centering Rings

The next task was to develop the motor mounts in Tinkercad. I would need at least five mounts to accommodate the different size motors that could be tested

- 13mm diameter by 44 mm length motors (mini motors)
- 18mm diameter by 70 mm length motors (standard motors)
- 24mm diameter by 70 mm length motors (large motors C11 & D12)
- 24mm diameter by 95 mm length motors (large motors E12)
- 29mm diameter by 114 mm length motors (extra large motors)

The motor mounts are created using a combination of PVC pipe, standard Estes motor mount tubes and engine blocks, 3D printed centering rings and motor retainers.

I began with the motor retainers by performing a quick search on Thingiverse. I made the decision early on was to not use motor hooks to secure the motors in the tube. Instead I will use the screw-on retainers. This also makes it easier to adapt the 3D printed rings to the PVC tubing. I found a number of screw-on motor retainers. The three files I used were



- 18 mm motor retainer by Owen1975 https://www.thingiverse.com/thing:4346576
- 24mm motor retainer - part of the RTS Rocket by jgutz20 https://www.thingiverse.com/thing:5324868
- 29mm motor retainer by JMillsCabrilloHS https://www.thingiverse.com/thing:882815

I was unable to find a 13mm motor retainer so instead I took the 24mm retainer and reduce it in size to 57% in my slicer to fit the 13mm motor mount tube. The motor retainers were sliced in Cura using a 0.20mm layer height and 100% infill.

The centering rings were created in Tinkercad, using measurements from the inside diameter of the PVC pipe and the outside diameter of the corresponding motor mount tube. The centering rings were sliced using a 0.16mm layer height and 20% infill. Like the retainers they were printed using PLA+ filament.

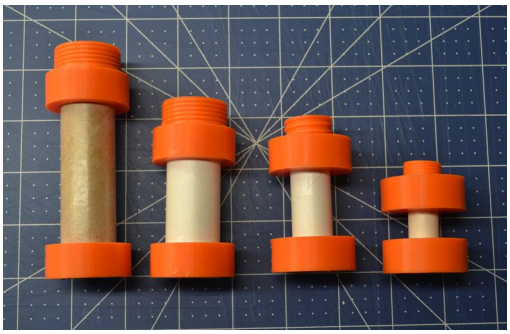
Assembling the Motor Mounts

To create the motor mounts I used standard Estes tubing and engine blocks. We are going to make four different motor mounts for our test stand; mini motors (13 mm x 44 mm); standard motors (18 mm x 70 mm); 'D' size motors (24 mm x 70 mm) and mid-power motors (24 mm x 90 mm). Each of these will be interchangeable with the mount on the test stand.

> ***Note**: While I did design a 29mm mount, I never did get around to actually building one, but included it in the drawings and instructions should you be interested in creating one*

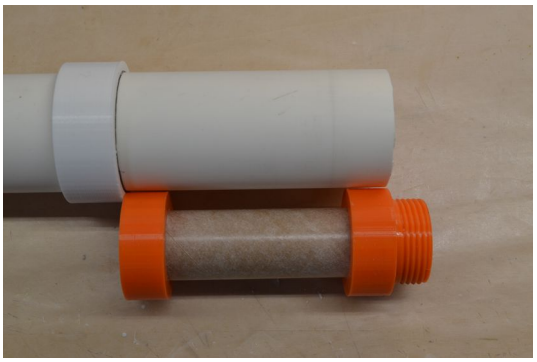
Each motor mount has an engine block to keep the motor in place. This is glued into the engine tube using wood glue, the same way I would do it in a flying rocket. Once the block has dried in place, a fillet of glue is applied around the top of the tube and then set aside to dry.

The 3D printed centering rings are epoxied to each motor mount tube, as is the threaded sleeve for the screw on retainer. For this step we used 30-minute epoxy. This gave us plenty of working time to get all of the pieces in place before the epoxy cured.



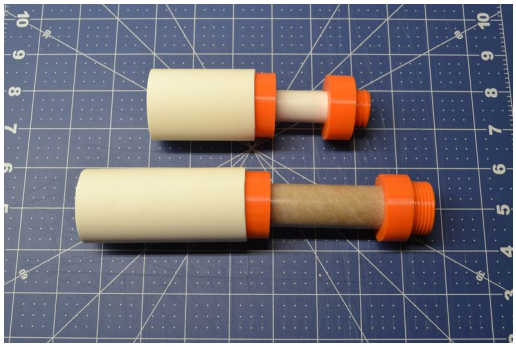
- First epoxy the threaded sleeve in place. This goes on the opposite end from the motor block.
- Before the epoxy cures, install the centering ring that is placed directly against the sleeve. Secure this ring with epoxy.
- The second centering ring is epoxied into place at the other end of the motor mount tube, flush with the end of the tube.
- This is done for each tube in the set.
- Set aside and allowed to fully cure.

Once the motor mounts have completely cured I proceeded to cut the PVC pipe to size for each motor mounts. To help mark the pipe as well as provide a guide for cutting the pipe, I printed out the 1.25-inch PVC straight edge.



To determine the length of the PVC pipe for each motor mount, I began by laying the pipe next to the motor mount. The PVC Straight Edge tool was slid onto the pipe and lined up to the edge of the motor mount. A line was drawn around the PVC pipe. Using a saw the PVC pipe was cut to size. Rough edges were sanded smooth.

A test fit of the motor mounts was conducted to confirm the fit and size of the PVC tube. You might find that some of the PVC pipe is not perfectly round on the inside and this can cause fit issues. You may need to sand the interior of the pipe and/or the centering rings to obtain a good fit.



To secure the motor mounts in the PVC pipe 30 minute epoxy is used. Begin by inserting the motor mount partially into the PVC, with one of the centering rings about 1/3 of the way in. Now apply epoxy to the inside of both ends of the PVC pipe. Slide the motor mount the rest of the way into the PVC pipe. Rotate the motor mount as you feel it entering the epoxy. This will help to evenly spread out the epoxy. Continue pushing the motor mount into the PVC tube until the centering rings are even with the ends of the tube. Wipe up any excess epoxy that is squeezed from the tube. Let this cure completely.

The final step is to epoxy the PVC motor mount tube assembly into the male PVC fitting. This allows you to easily change out different size motor mounts on the test stand. Before putting the two parts together, sand the forward part of the motor mount assembly tube and the inside of the PVC fitting. This will help the epoxy stick better.

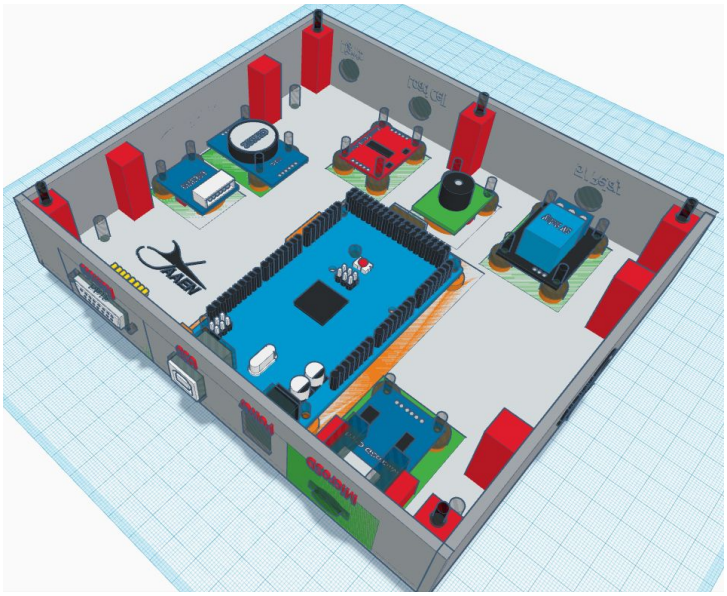
Once again I use 30-minute epoxy and place it inside the PVC fitting. The motor mount assembly is pushed into the PVC fitting, again rotating the assembly to help evenly spread the epoxy. The motor mount assembly should bottom out on the fitting. Wipe away any excess epoxy that is squeezed out. Let this cure completely. After the entire assembly is cured I used a label maker to mark the size of the motor mount tube.

This completes the motor mount tube assembly. Our next task is to design and assemble the Test Stand Electronics Housing.

17 Designing and Building the Electronics Housing

Designing the electronic housing was the most difficult part of this project for me. Being able to put all the electronics together on a breadboard is one thing. Thinking about how to create a housing and layout the parts in a logical manner is something completely different.

I had to think about how to layout the components so that the wiring layout would make sense. I needed to have access to the USB port of the MEGA 2560. I needed easy access to the MicroSD card. I needed to have access to the Relay, the HZ711 amplifier and the wiring for the warning lamps



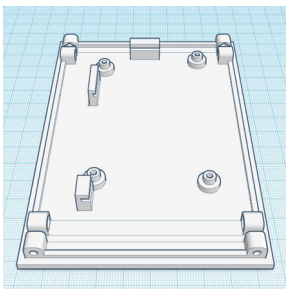
The housing would need to protect the components inside. Some of this protection would be provided by locating the housing under the platform of the test stand.

Finally I wanted the housing to be flexible enough that I could expand the functionality of the test stand if I desired. The Mega2560 has a large number of pins available and the code is using less than 1/3 of available memory and only 26% of available storage space.

The other decision I needed to make was to use some type of ready made enclosure (like we did when we used a toolbox in the Arduino Launch Control System) or should I try to build or 3D print an enclosure. In the end I would go the 3D printing route.

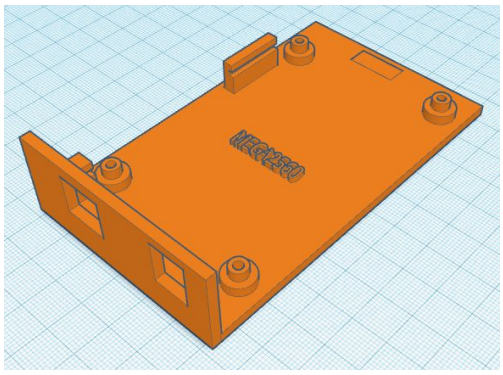
A Step at a Time

I started with an STL file of an Arduino mount that I then imported into Tinkercad. This became the basis for all of the other mounts that were created for the housing.



> ***Note:*** *If you recognize what project this file comes from, please let me know so I can give credit to the original designer. It has been a couple of years since I started this project and I no longer have the original project file. I have looked around the web but haven't found it thus far.*

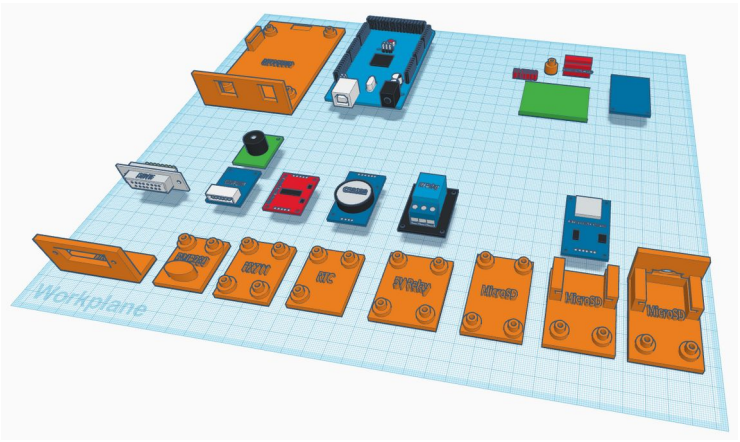
Using the tools available in Tinkercad allowed me to modify the original STL. The edges of the mount were removed and it was slimmed down. This also got rid of the round hinge points and the slotted holder at the rear was removed as well.



I needed to add a 'wall' to the front that would allow access to the USB port. This also meant I had to allow access for the power port even though it is not used with the test stand. This part went through 5 design changes and updates until we had a part that would work.

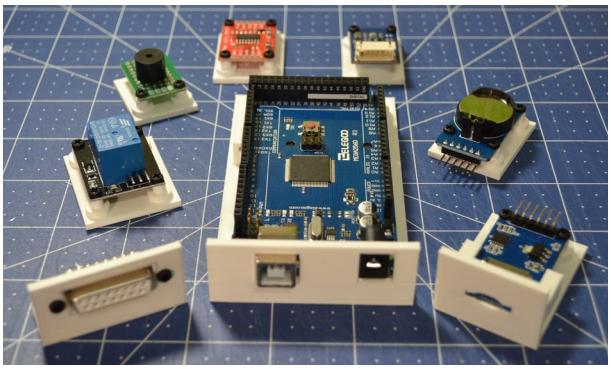
Individual Component Mounts

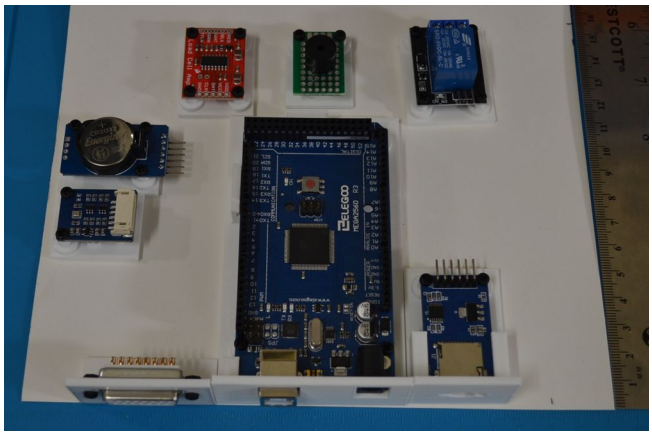
With the Arduino mount design finalized, I was able to use it as a basis to create a series of mounts for each component. Some of the electronic components were already available through Tinkercad, while others we had to create. Some of the mounts went through several iterations (such as the MicroSD card mount) while other mounts would work for different components (the BME280 mount worked for the BME280 sensor and the piezo buzzer. To keep straight which mounts went with which components, the names were incorporated into each mount. Once the designs were completed they were 3D printed and checked for accuracy. Some mounts were fine with the initial design, while others needed one or two modifications.



The Jigsaw Puzzle Design

At this stage of the design process I had a number of individual pieces, but still no coordinated overall design. I basically had a number of jigsaw puzzle pieces. Now I needed to figure out how to put them together. As stated in the initial concept, the design needed to make sense from both a wiring and overall usage view point.





To start determining the layout I got out a square piece of poster board to use as a base. Then I began to lay out the different components on the poster board. After moving the pieces around to see where each would fit and work best, I finally had a basic design that would work.

I cut the poster board to size to make sure it would fit under the platform and between the legs - especially at the front of the platform. The housing would be recessed from the front of the platform for protection, but the user still needs to gain access to the front to attach cables and use the MicroSD card. Once we knew the housing would fit, I marked the location of each component on the poster board.

Back to Tinkercad

With the poster board layout in hand, along with a ruler, I began to transfer the individual components into Tinkercad. First I created a 'floor' the size of the poster board and same thickness as the component mounts. I added in 5 holes that would allow wood screws to hold the housing to the platform. Next came the placement of the components on the floor, based on the measurements taken earlier.

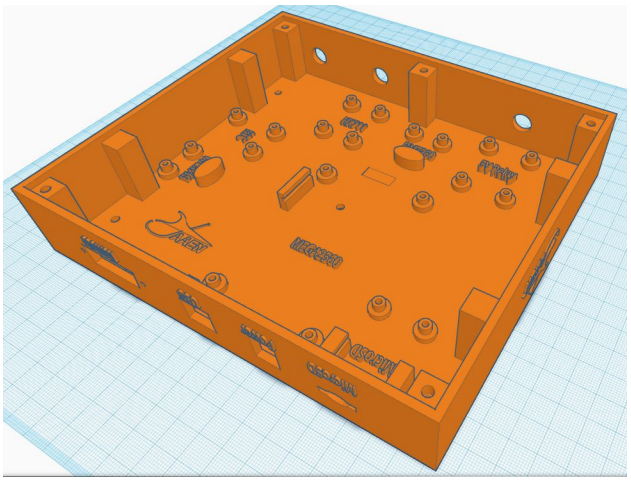
At this point I had the layout I was looking for, but I wasn't nearly done. I still needed "walls" as I needed to enclose the housing to protect the electronic components.

I needed access openings for wires to run through at the rear of the housing. I had some rubber grommets that I could use to protect the wires from the sharp plastic edges, and so the access holes were sized to fit the grommets.

The housing would need a cover to finish protecting the components. This is simply a copy of the 'floor' of the housing, trimmed to fit inside the four walls. I also added an opening over top of the piezo buzzer to allow the sound to escape out of the housing.

Now I needed a to add supports for the cover and attachment points. I decided to use heat inserts to allow the cover to be screwed down on to the housing. Additional supports were added around the side walls to support the entire structure.

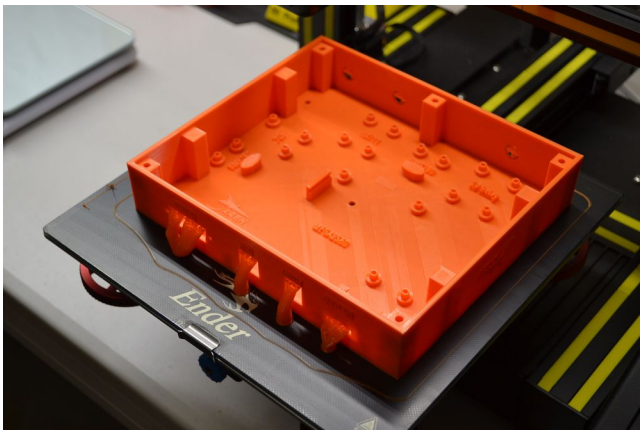
I wanted to add labels to the outside walls of the housing that would identify what each access port was for. However, the housing was to be mounted under the test stand platform. Before I could add to labels, the entire housing had to be flipped 180-degrees.



The final design addition was to add the AAEN logo to the side walls of the housing, as well as on the floor of the housing. With the housing design complete, all of the components are grouped together in Tinkercad resulting in a completed housing. This is then exported as an STL file.

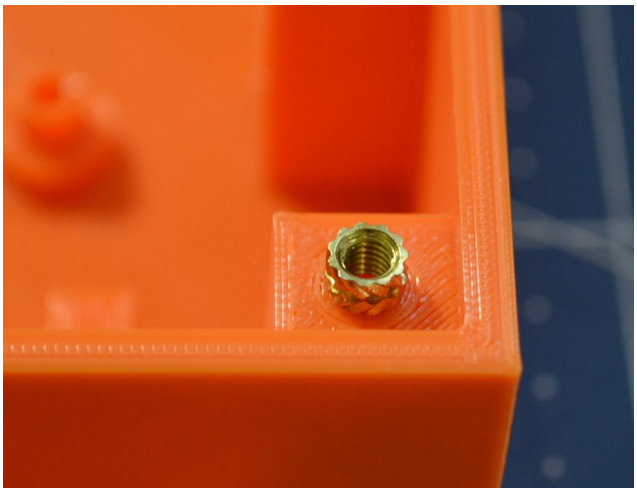
3D Printing

The housing was printed at 0.20mm layer height. I also used tree supports around the openings in the side walls. It was printed using orange PLA+ filament. Using an Ender 3 V2 it took over 18 hours to print.



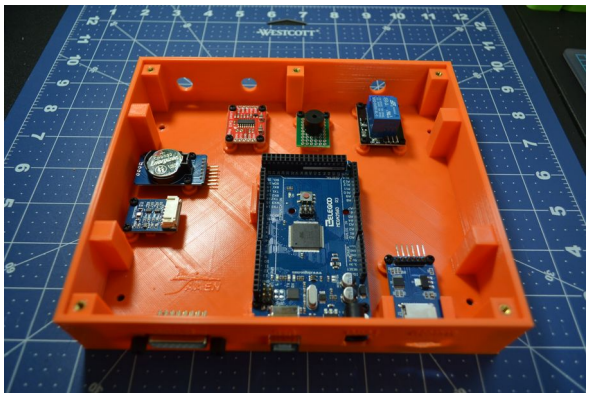
Once the housing finished printing it was time to print the cover. It was also sliced at 0.20mm layer height. Like the housing I used orange PLA+ filament. The cover took over 7 hours to print.

Once the print was finished I added 5 heat set inserts into the four corners and center back support. These were inserted using an insert tool attached to a soldering iron. When using these inserts you don't have to 'push' the inserts into the plastic. Instead let the heat melt the plastic and the weight of the tool will push the insert in place.



Once the insert is in, take a flat head screwdriver and using the flat side of the screwdriver push and hold the insert down flush in the plastic. Once it cools the insert will stay flush with the surrounding plastic instead of slightly bubbling back up.

Installing the Electrical Components

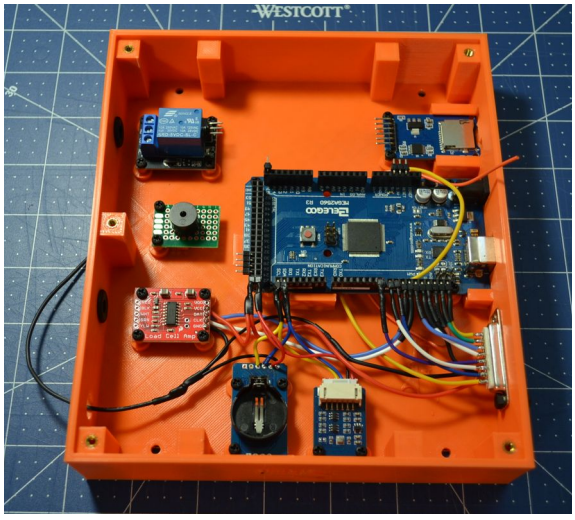


With the printing complete it was time to start installing the electrical components. These are secured to the housing using 2.5M nylon screws.

Even though I had the locations of the components mapped out on paper, it was really handy to see each component labeled inside the housing. This made the process of installing each component easier and quicker.

Once the components are in place it is time to start the wiring process. Before I started wiring anything, I made sure to remove the battery from the Real Time Clock.

Wiring



I knew some of the toughest wiring (at least for me and my big old fingers) would be soldering the connections for the DB15 connector. Additionally, the majority of the connections would be located on that same side of the Arduino as the DB15 connector.

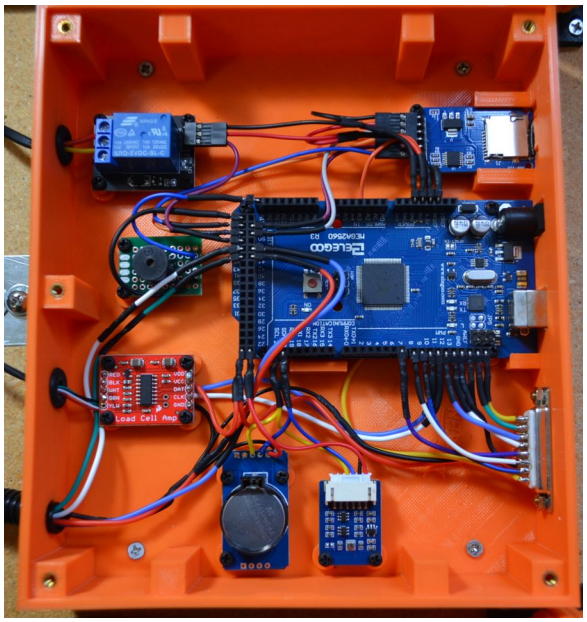
In the picture to the right you can see the wiring for the connector as well as the BME280 and the Real Time Clock. The wiring is soldered to 90-degree connectors and then inserted into the appropriate pins on the MEGA 2560. You can also see in the picture where I was getting ready to make a mistake in my wiring.

Changing Pins

As I began to wire the various components I realized that some of the wires would not use the same pins that we used on the breadboard. In most cases this didn't pose a major problem, I just had to make sure that I made the appropriate changes to pin assignments in the code. For example, the Relay pin was originally digital pin 6. However, based on the location of the relay, it made more sense to move it to digital pin 49 at the rear of the Mega2560 board.

The same thing occurred with the warning lamps. The original RGB pins were digital pins 3, 4 and 5, with the strobe light using digital pin 2. Since wiring for the lamps would be coming in towards the rear of the Mega2560, I wanted to move their pin assignments as well. In the picture above you can see the 90-degree pins plugged into pins 23, 25, 27 and 29, which would operate the R, G and B connections as well as the strobe.

It wasn't until testing that I realized my mistake. At first everything seemed to work fine. The strobes flashed, the red and green illuminated just fine. But I could not get the yellow to work. No matter how I changed the color coding in the software the lamp would only show red or green. It took a while before I realized my mistake.



The RGB lamps need a special type of pin, one that does pulse width modulation (PWM). Unlike the standard digital pins that are either 'on' or 'off' PWM pins allow you to adjust the amount of voltage going through the pin. That lets us adjust the amount of red or green being displayed so we can create a yellow lamp. The pins I had plugged into were standard digital pins. I needed to find the PWM pins located at the rear of the MEGA 2560. It turns out that the PWM pins are located at the opposite end of the MEGA 2560, in pins 44, 45 and 46. In this case we had to extend the wiring to get to the pins we needed.

When you find yourself in this type of situation, make sure you double check that the pins that you are looking at using will do the job. It would have taken me less than 3 minutes to look up the pin configuration on the Arduino. It took a lot longer to troubleshoot and fix.

Battery Pack and Continuity Lamp

A battery pack is used to provide power to the igniter. This power system is separate from the rest of the Test Stand. I chose to use a 4 AA battery pack. The system needs a continuity check to verify that the electrical connections are good.

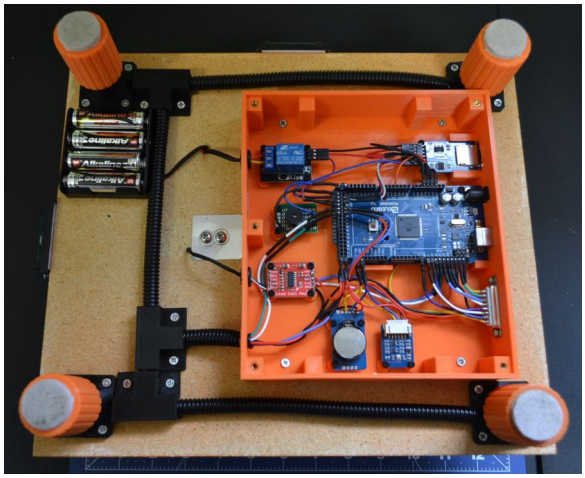
The continuity lamp is a 5mm red LED lamp with a 1KΩ resistor. This is adequate for the Estes igniters that I tend to use. If you are using a different igniter, you will need to make sure that the resistor you use is adequate to prevent the igniter from lighting when power is applied.

The continuity lamp needs to be visible to the user. This meant it needed to be located on top of the Test Stand platform. The wires going to the igniter also need to be on top as well. It was decided to combine the two into a single housing.

A lamp housing was designed in Tinkercad that matched the look of the Warning Lamp housings. The wiring for the continuity lamp and main power for ignition was run into the housing. A set of banana clips was installed on the top of the housing for the igniter wires.

Warning Light Wiring

The warning lights are not difficult to connect but the long runs of wire around the edge of the test stand platform create their own unique challenge. The biggest challenge is how to keep the wires organized and contained. I ended up using 3/8-inch diameter split plastic protective wire wrap. I also used 3D printed conduit T fittings.

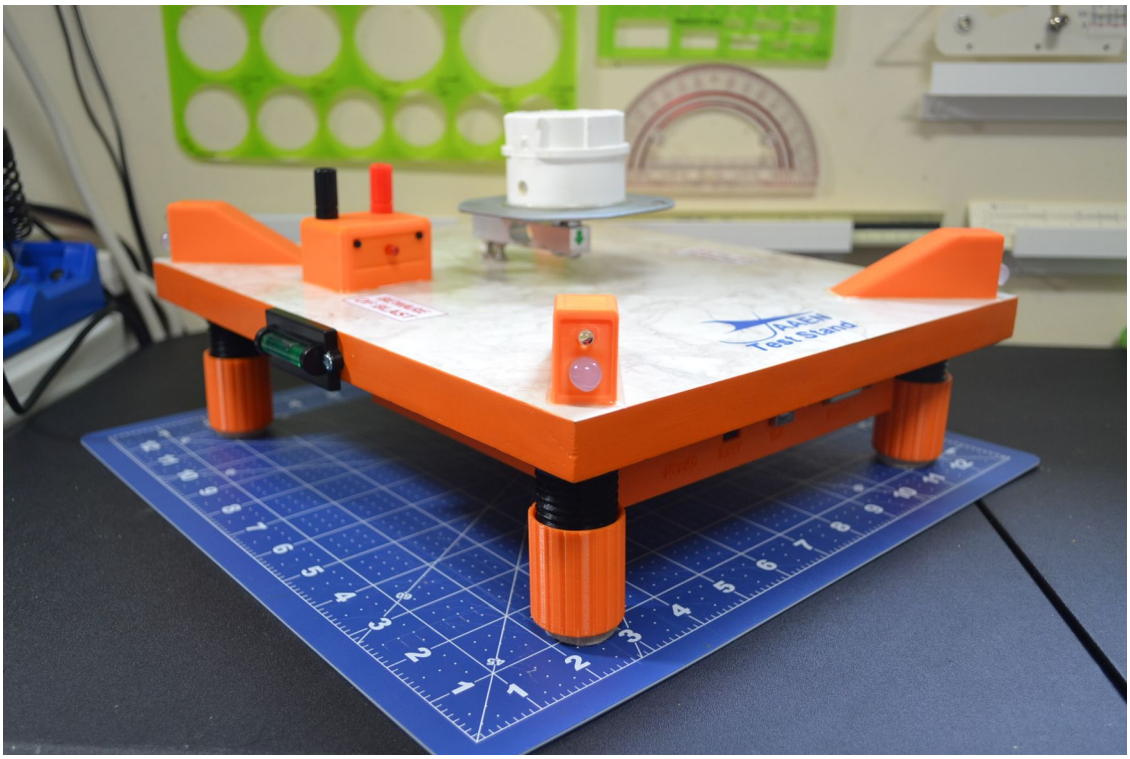


The 3/8-inch diameter split plastic protective wire wrap was purchased at Harbor Freight and was cut to length. Use the split in the plastic to insert the wires into the wrap. The conduit T fittings were used to connect the protective wraps.

Lastly, any other exposed wiring (such as the load cell wires and the igniter wires) were covered with heat shrink tubing to help protect them as well.

Platform Levels

The last thing I added to the platform is a set of RV camper levels to the side and rear of the test stand. Using the levels and the adjustable legs I can make sure that the test stand is sitting level. If the test stand is not level, the data gathered from the test firing may be skewed, as the force from the motor exhaust will not be pushing directly down on the load cell, but instead hitting it at an angle.



18
Building the Remote Head

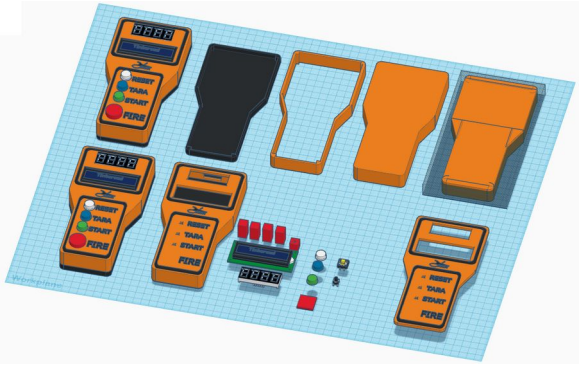
The remote head went through several design changes before I settled on the one that was fairly simple and easy to construct. I'll quickly review each design iteration, what I was trying to accomplish, and why I moved on from that design.

Version 1

The first version that I designed was a simple rectangular box. My initial thought was to create a handheld unit. In this design the bottom is a thin cover that would screw onto the main unit.

The problem I ran into was the size of the LCD screen and LED numbers were too small. Once I up-scaled everything to adjust for size, it became too large to hold in your hand. So we packed this design away and tried a modified version.

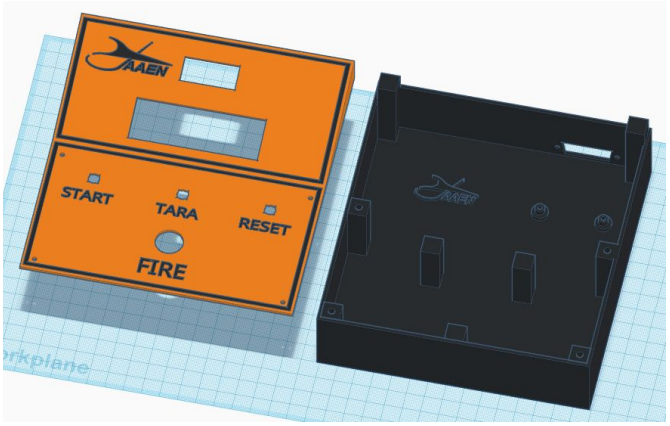
Versions 2 and 3



I spent a lot of time on Versions 2 and 3. The top of the handset was extended out to accommodate the wider screens. Version 2 had the angled top screen, like in Version 1. However, the angled top resulted in a much larger head to allow room for the electronics under the display. To keep the handheld from being unwieldy I decided to ditch the angled top and keep everything flat.

The flat top design became Version 3. While I liked the design, I began to realize that once I got all of the attachment points in place, to mount the electronics and to hold the device together, there was no room inside the handheld. If I were to increase the size to make it easier to install the electronics, it would be too big to hold in your hand. It was at this point I dropped the handheld concept and started looking at a 'desktop' style remote head that I could sit next to the laptop. That design would become Version 4.

Version 4

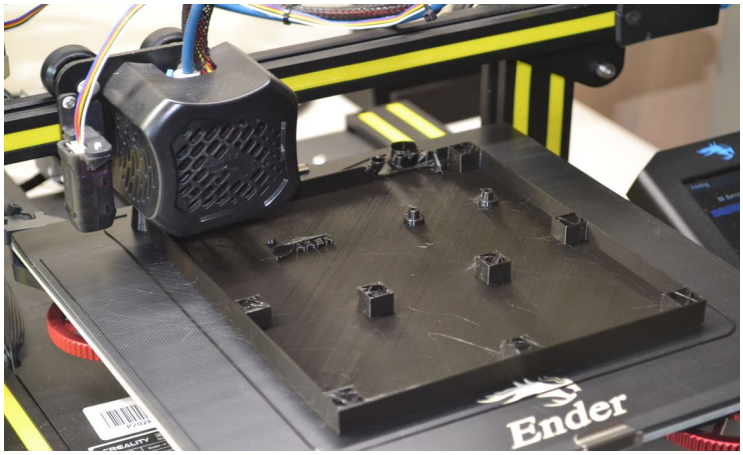


I basically started over with the Version 4 redesign. I decided to go with a simple box layout as my thought was the box would sit next to the laptop that is being used to enter information and record the data. I also brought back the angled view top to improve the visibility of screens when the box is sitting on a table or bench. This version incorporated the lessons learned in the previous designs.

The larger size makes it easier to wire everything inside the box. You don't feel like you are short on space. The top panel is where most all of the electronics in the remote head will be mounted. I opted to not include the mounting holes for these parts in the 3D print, instead deciding to use these parts to mark mounting points and drilling out the holes. This also makes the 3D print more flexible in case someone is using a different LCD or LED screen from the ones I used.

The first print of the box provided insight into some areas that needed improvement. I also added support columns in the area of the buttons. This provided the rigidity I needed without having to print out a thick top panel. We added a couple of additional mounting points to help improve the rigidity of the remote head.

Printing the Remote Head



The design for the remote head is now finalized and is ready to be printed. The design was exported from Tinkercad and imported in Cura. The settings I use included a standard quality setting of 0.2mm layer height and an infill of 15%. On the top you will need to add support. My first print I used tree support and the results were not what I had hoped for. For my second print I used the 'Normal' support structure setting. After slicing the software indicated that the top print would take nearly 9 hours. The base structure showed a print time of nearly 16 hours.

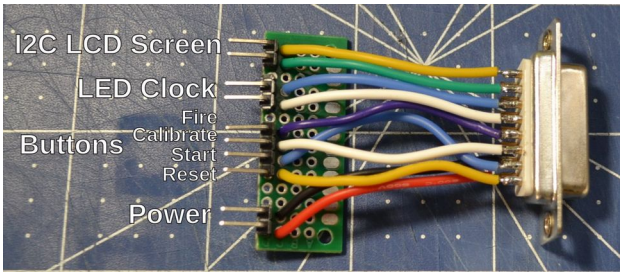
I used PLA+ filament as this filament temperatures were increased to 215 C at the nozzle, and 60 on the bed. The PLA+ filament delivers nearly the same performance of ABS, but with the ease of regular PLA. Like the drawing, orange filament was used for the top and black was used for the base. Once the print was complete a black paint marker was used to highlight the raise sections of the top.

Wiring the Remote Head

Compared to the test stand housing, wiring the remote head is much easier. For me, the hardest part was soldering the DB15 connector.

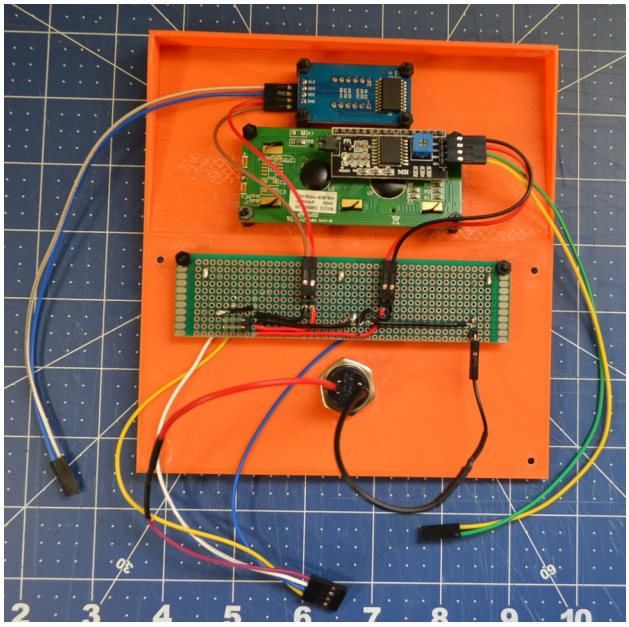
The DB15 Connector

The wires from the DB15 connector are soldered to a small section of prototype board. Opposite the connections I soldered the 90-degree pins. The pins are organized according to their purpose. The prototype board is attached to the base of the remote head.



The Cover

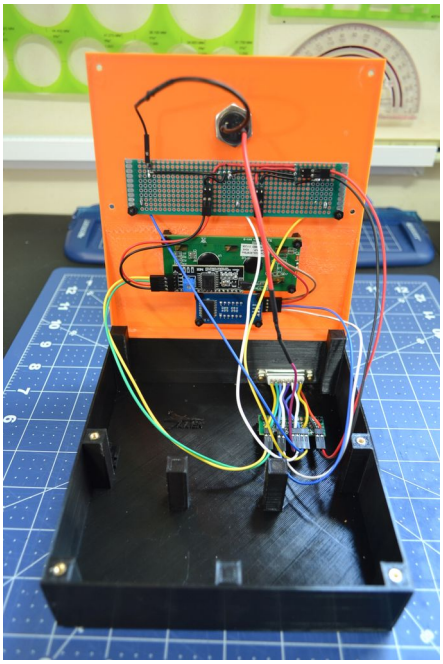
The top cover of the remote head contains the 4-digit, 7-segment LED clock, the 16 x 2 LCD screen, and the four buttons that operate the test stand.



The LED clock and LCD screen are attached to the cover using nylon screws and spacers. The Fire Button is attached to the cover using the supplied screw on nut and washer.

The three remaining buttons are attached to a prototype board. Additionally connectors for power are soldered to the board and run to their respective nodes for the LED clock and LCD screen, as well as the ground connection for all of the buttons. A set of cables is created using jumper wires with female DuPont connectors. These connect the LED clock, LCD screen and all four buttons. The opposite ends will connect to the pins on the DB15 connector.

Connecting the Cover and Base



With the connector cables created, it is a simple matter to plug in the cables to the proper pins on the DB15 board. There is plenty of room in the base of the remote head to work with the cables. No need to have short cables and make things harder.

With all the wire bundles connected, set the cover down on the base and align the screw holes with the inserts. I used 2.5M stainless screws to attach the cover to the base. Make sure you don't pinch any wires between the cover and the base walls.

19 Assembling the Test Stand

With all of the sub assemblies complete, it is time to put our test stand together. The main components are the test stand platform, the remote head, the motor mounts and a computer. You will probably want to use a laptop instead of a desktop computer.

Setup Area

When you are setting the system up, the first thing you want to do is put in a fresh set of batteries. This will allow you to conduct a number of test back-to-back if desired. Fresh batteries also tend to ignite quicker.

Next, look for an area that is fairly level. Use the adjustable legs and the platform levels to get the platform in a stable and level position. If you are in your yard, a concrete pad is a good spot. If you are in a field, try to find a level spot free of rocks and other debris.

You also want to make sure there are no flammable materials around. If you are in a field area, make sure you clear away any dry grass or leaves from around the test stand. If you are in your yard, make sure there is nothing around that can catch fire. With that being said, you should still have a fire extinguisher close by should something catch fire unexpectedly.

Connections

Once you have the test stand setup, level and stable, insert a MicroSD card into the reader. The card should already be formatted. You should consider using a blank card, transferring the data off the card after each test onto the local computer.

You will need to connect the DB15 cable to the test stand, along with the USB cable. These cables should be at least 10-feet in length for motors up through 'C'. Use longer cables for the more powerful motors. Based the cable distance on the launch distance for the size motor you are using.

The opposite end of the DB15 cable is connected to the Remote Head. Make sure you have a good connection on both ends of the cable so that you don't have issues when trying to fire the motor.

The USB cable will attach to your computer. When setting up your computer, make sure you don't block any cooling vents. If you are outside on a hot summer day and block your cooling vents, you could burn up your computer by overheating. Keep things cool.

Motors and Motor Mounts

Test day is very much like a flight day. You may want to have your motors that you intend to test prepped, just like you would if you were going flying. Having the motors prepped will make it quicker to change out motors between tests.

If you are planning on testing more than one size motor, make sure you bring the appropriate motor mounts. Again, this will help testing proceed faster.

Accessories

It is not uncommon to video your motor tests. During flights we barely see the motor as the rocket flies off the pad. During testing you can see the motor's performance during the entire burn. You may learn even more about the motor performance by watching video of the test in slow motion. This may reveal things like huffing or uneven burning.

Because the motor doesn't move from the platform, you can setup your camera on a tripod so you get a nice steady picture. This also means you can get the camera as close to the pad as you want. In addition to the tripod you may consider placing a piece of clear plastic in front of the the camera. This will help protect the camera lens.

Similarly, if you plan to live stream your test, you may need an additional computer to handle the extra cameras and the streaming software.

20
Conducting a Motor Test

With the Test Stand assembled, you can begin booting the system to conduct the test.

Start by booting the computer and opening the Arduino IDE. At the test stand, insert a MicroSD card into the electronics housing on the Test Stand. Now plug in the USB cable to the computer to start the boot process of the Arduino Mega2560. With the IDE up and running, open the Serial Monitor. You computer is now ready.

Mega2560 Boot

As the Mega2560 initializes, the Serial Monitor will display the current status of each sensor as it comes on line. The first few lines displayed on the Serial Monitor show the name of the software and the version number.

The next component to be initialized is the Real Time Clock. It will display the current date and time on the Serial Monitor. If needed you can adjust the date and time to match the clock on your computer.

The next component to be brought on line is the MicroSD card reader/writer. This is followed by the BME280 environmental sensor and the Load Cell with the HX711 amplifier. The last components to be brought on line is the Fire Control System. With the successful initialization of all of these systems, the LED lamps on the Test Stand will turn green. When you are ready, press the green button to start the test procedure.

If any of these systems fail to initialize, the system will show the failure on the Serial Monitor as well as on the Remote Head. All of the systems must be functioning in order to conduct a test.

Data Entry Process

Once the green 'Start' button is pressed, the Serial Monitor will begin to ask a series of questions. They include:

- The location where the test is being conducted
- The elevation of that location
- The name of the manufacturer of the motor
- The length of the motor casing in millimeters
- The diameter of the motor casing in millimeters
- The condition of the case
- The mass of the motor in grams

All of these questions can be answer in different ways. For example, you may answer the name of the town where the test is being conducted. You might add a GPS location, or perhaps an address. Or you might use something like "Football Field" or "Backyard". The choice is yours.

What I would encourage is that you be consistent in your answers. For instance, if you are testing in different locations that are at different elevations, I would try to be as specific about the location as possible. If all of your tests are in your backyard, then indicating that should be adequate.

The next three questions relate to the motor designation. this includes:

- Total Impulse
- Average Thrust
- Delay Time

If you are not familiar with how model rocket motors are designated, please refer back to Chapter 2 on Model Rocket Motor Basics.

This figures, along with the ignition time entry, will determine how long data is collected on your motor. Therefore it is important to answer these questions accurately.

The next three questions are asking about the motor composition. This includes:

- The type of propellant
- The mass of the propellant
- The lot number or manufacturer date

This information can be used when comparing different types of propellants for efficiency. The mass of the propellant can affect the overall performance of the rocket, so two motors rated the same but with different mass will have an impact on the how well the rocket performs. The lot number or date code can be used to test the variances in the same motor classification manufactured at different times. This information can also be helpful if there is a CATO and you need to complete a Malfunctioning Engine Statistical Survey (MESS) report.

The last two questions deal with the igniter. They are:

- Type of igniter used
- Ignition time period

There are a number of different types of igniters available to the rocketeer from a wide variety of manufacturers. There are also those rocketeers who like to modify existing igniters or make their own.

Finally you are asked how long should power be sent to the igniter. You should enter a number between 3 and 10 seconds. If you enter a number outside this range, the program will default to 5 seconds. As noted earlier this is one of the inputs that will determine how long data is collected on your motor. Therefore it is important to answer this question accurately.

With the information entered, the program will display on the serial monitor the length of time that data will be collected in millisecond

All of this is used to help catalog the various motors that you are testing. While the thrust curve is the main information most people are looking for from the test stand, a more complete picture of motor performance is obtained by entering in all of the data requested.

Pre-fire Process

You are now entering the pre-fire process. Instead of inputting information about the motor, you will be given a checklist and asked to verify that the checklist is complete.

Motor Loading Checklist

This checklist is the first one to be displayed. Here it is confirming that a motor has been installed in a motor mount, that it is secured in place with a retaining ring, and that the igniter is installed.

Once this checklist is complete, you would enter a 'V' (for "verified") into the Serial Monitor and press ENTER. The date and time is then displayed on the Serial Monitor and in the System Log.

Motor Mount Checklist

With the motor secured in the mount, it is time to secure it to the test stand. The first step in this process is to look at the threaded adapter ring attached to the load cell. This is where the motor mount will be installed. Make sure that it is relatively clean and free of debris. As part of this check the openings that the exhaust gases will escape through are clear. If these are not clear, it is possible that you could have a pressure buildup in this area.

Next you want to screw the motor mount into the threaded adapter. You should make sure that the mount is secure and will not become disengaged from the mount.

Next, we need to attach the micro clips to the igniter. Neither the clips nor the igniter leads should touch each other. With a good connection to the igniter, the continuity lamp should now glow red.

With the checklist is complete, enter a 'V' into the Serial Monitor and press ENTER. The date are time are again displayed on the Serial Monitor and recorded in the System Log.

Test Stand Calibration

It is now time to calibrate the test stand. If you have calibrated the test stand earlier using the same motor test setup, you can bypass the calibration process. However, it is recommended that the calibration be conducted for each test to ensure accurate test results.

Tare

The first part of the calibration process is to get the weight of the motor setup on the test stand. The weight of the motor mount assembly will be removed from the calculations so that only the force of the motor will be recorded, and not the force of the motor mount weight and thrust combined.

Calibration Weight

The next step is to place a known weight on the test stand. The weight needs to be placed on the load cell itself. Depending on the weight, you may be able to place it on top of the motor mount or perhaps along the metal platform that attaches the adapter ring to the load cell. You will need to enter the weight of this object in grams and hit the Enter key.

Calibration Value

The software will calculate the new calibration value and display it on the Serial Monitor. The software will ask if you want to save this value to EEPROM address 0. If you enter 'Y' (for Yes) the value will be written to memory. This makes it available for later tests if desired. You are reminded to remove the calibration weight. The information is recorded to the System log.

Clear Test Stand Area

This is the last checklist prior to the test firing of the motor. You are asked to verify four things in this checklist:

- Make sure everyone is a safe distance from the test stand
- Confirm that nothing is on the test stand (like your calibration weight you forgot to remove)
- There should not be anything under the load cell. Any item under the load cell may interfere with the movement of the load cell and result in bad data
- The vent ports on the adapter ring must be clear. This is one last check to make sure any pressure from an ejection charge or other event doesn't result in an over-pressure event but instead the pressure can escape through the vents.

Once these items have been verified, enter 'V' into the Serial Monitor and hit the ENTER button.

Motor Test Fire Process

You are now ready to conduct a test fire. On the Serial Monitor it will display an explanation of the data that will be displayed in the three columns while the engine is firing. It will also display the instructions on pressing and holding the red fire button through the 5-second countdown.

When you first press the Fire button down, the date and time is noted in the System Log. The buzzer will sound on each second of the countdown. The LED clock will show the countdown as t-5 through t-0.

Abort

If you release the Fire button before the countdown reaches 0 an abort will occur. This stops the firing process and no power will be sent to the igniter. The date and time of the abort will be noted in the System Log.

When an abort occurs you have the option to recycle the system and continue the countdown. When you recycle the system will return to the Clear Test Stand Area section. You will need to verify that the test area is clear and then you can go through the countdown again to fire the motor. No data is lost during this process.

Motor Firing

If you press the Fire Button and keep it pressed down through the countdown at t-0 the relay is opened and power will begin flowing to the igniter to ignite the motor. This is noted in the System Log and the data from the load cell will begin to appear on the Serial Monitor. During this time the Serial Monitor will display a notice once the relay is closed and power stops flowing to the igniter.

Data will continue to be displayed on the Serial Monitor. This data is also recorded to the Motor Log as a CSV (Coma Separated Values) file. Once the time expires for the data collection period this will be displayed on the Serial Monitor and the System Log.

At this point in the test process, the motor firing is complete.

Post Motor Test Firing

With the firing of the motor complete, the program enters a 1-minute post test safety period. During this time no one should approach the test stand. This period would also be used if there was a misfire. You would wait the 1-minute safety period before removing the bad igniter and installing a new one.

CATO

Following the 1-minute safety period, the program will ask if the motor suffered a CATO (catastrophic failure). The majority of the time, you will answer 'No' as the motor will burn as expected.

If you do experience a CATO, it should be reported to motorcato.org. This is the official NAR site to file a MESS (Malfunctioning Engine Statistical Survey) Report. This helps the NAR track any issues that may be affecting a specific series of lot of motors. Most of the information needed about the motor you can have readily available as you have entered it as part of the testing process.

Case Mass

Here you weigh the empty case and report that mass into the program. This data should be reported in grams as were the previous reports.

Comments

The last data entry point is for comments. Because you are entering this information through the Serial Monitor, you only have a single line to write your comments. You should make them short and concise. You can always expand on the comments later when you write your report.

Test Complete

The test is now complete. At the bottom of the Serial Monitor it provides a statement indicating the test is concluded and the date and time. This information is also recorded in the System Log.

If you wish to conduct another test, it is recommended that you push the 'Reset' button on the remote head. This will start the entire process over again. It will also verify that the sensors are still on line and functioning properly.

21
Tactics to Improve the Test Stand

If you have followed along from the beginning of the manual, you should now have a working Model Rocket Motor Test Stand. You have seen how we designed, developed and built this system. However, like any project, it is far from perfect, and there are things that, in hind sight, we could have done better. Let's take a look at some of the changes, updates and improvements that can be made to the system.

Hardware Updates

Hardware updates are almost always harder to incorporate into a finished project than software updates. There are the questions of where will the hardware be mounted, are there available pins on the Arduino that can be used, will any of the current hardware need to be moved, removed, or replaced?

As we look at potential updates to the stand, consider adding them early in the build. If something needs to be moved or added, it is easier to go into Tinkercad and make the new mounts before the item is 3D printed, instead of trying to figure out how to add it afterwards.

With those thoughts in mind, here are a few things we thought of that could be added to the test stand:

- Add temperature sensor for outside case
- Add temperature sensor for exhaust gases
- Add radio link instead of DB15 cable
- Incorporate a Raspberry Pi

Software Updates and Changes

Software changes are usually easy to implement and easy to revert back if things don't go as planned (as long as you make the changes to a ***copy*** of your program). It is one of the big advantages of software based electronic systems.

You still have to be careful with software updates. Will your change create a conflict with another section of code or piece of hardware? Is there enough memory or storage space for the updates you are considering? Will the changes result in a slow down of sensor readings, resulting in a reduction in the amount of data you can collect?

These are just a few of the issues that you can run into when updating or changing the code. That doesn't mean you shouldn't try, it just means you need to be aware of potential conflicts and

issues. Besides, you still have the original version of the code available (you were making the changes to a copy, weren't you?).

Let's look at some of the changes you might consider incorporating into the test stand software.

- Replace the serial monitor with a stand alone program (python??)
- Add option for time zone (such as UTC or EDT) with the RTC
- Confirm data inputs from user are correct, and allow the user to reenter the information if needed
- Allow adjustment of brightness on LCD and LED screens
- Allow load cell to be used for weighing motors

Make The Project Yours

This project will be much more valuable to you if you try to make updates and changes to what we have presented here. Perhaps you realize there is a better way to collect the data. Maybe you find a better algorithm to process the motor data. Try it, and see how it works. If you have something that works better, let me know about it so I can incorporate it into later versions of the project.

22 Conclusion

This brings to a close our Model Rocket Motor Test Stand build. This build took significantly longer than I expected, mostly because 'life' got in the way. Other things came along that took priority and there were times when the project set on the shelf for several months, collecting dust. Now that we have finished the build, I am very happy with how it turned out.

Throughout this build I have tried to let you to see what we did for several reasons:

- Electronics is a new hobby to me. The use of Arduino has made creating electronic projects significantly easier, at least for me. I hope that this project helps you feel that you can do a project like this as well.
- I wanted to you to see how the design process, especially the remote head, changed over time. We included the initial design of the test stand, and showed had it changed over time.
- For me, transitioning from the breadboard to the electronics housing proved to be one of the more challenging aspects of this project. Building on a breadboard allows you to move things wherever you want, while working within the confines of the electronic housing forces you to be much more disciplined, and that is a good thing.
- This project expanded our use of Tinkercad and 3D printing. I had to learn how to create parts (like the PVC connector) so that we could create an accurate representation of the test stand. I had to think differently about to design the electronic housing, prior to soldering the first wire connection.
- This was my first use of heat set inserts. This meant not just using inserts, but we ended up making a stand for the tool that we used.
- Creating a test stand, like the avionics projects I built previously, opened another area of rocketry that I couldn't participate in before. This is one of the reasons why I really enjoy what the Arduino has been able to open up for folks like myself.

After each project I find myself looking back and seeing things I could change to make the project better. That's one reason why I include a chapter on potential hardware and software improvements. If you are looking at making this project, keep these updates in mind as it will help make your project better than the one I built - and that's a very good thing!

I can see where my CAD skills are improving while using Tinkercad, but I still have a number of areas where I can improve. One major area that I need to improve upon is the design of multipart components and how they are attached. Sometimes, the accuracy of my screw holes left much to be desired.

I am also looking at changing CAD programs. While I have been able to create some decent drawings, especially with this project, I can't get a true CAD drawing using Tinkercad. That means I need to be looking at different CAD software. Being a proponent of open source software, that will likely mean learning FreeCAD.

One of the nice things about Tinkercad was that I was able to make the drawings available. The problem is you can only get the drawings through the Tinkercad web site. If Tinkercad goes down, the drawings cease to exist. By transitioning to FreeCAD, we can make the CAD drawings available and will no longer be dependent on a company maintaining a server.

Another area that I am working on is my wiring layouts. As I alluded to above, working on the breadboard and routing wires to the various components is pretty easy. You can put stuff anywhere and as long as the right wire attaches to the correct pin of the component the system works. This changes dramatically when we start running wires inside of boxes. How the wires are run, where the components are placed, how to we take things apart, all becomes much more important. I worked a bit harder on this project to make the wiring layout make sense. Could I have done it differently or better? I'm sure I could, as hindsight helps us see these issues. Hopefully this project will change how I approach the next one.

This is now the fifth model rocketry related electronics project I have documented. Looking back at my previous projects I can see where I have gotten better in some things. My soldering has improved and I am trying to think more about how the project goes together. My programming in Arduino also seems to be improving. I still have a long way to go, and I will hopefully continue to improve in these areas and others.

In one of the previous Project Manuals I talked about what I am looking at creating in the future. That response is still a good one:

> *"One project I hope to do in the near future with the Arduino is to try some basic telemetry during flight. I am excited about the thought of seeing flight data transmitted to a ground station, see the data displayed on the screen in near real time. This presents all sorts of challenges, from the type of radio system to use, to the type of Nano board to use, to programming the ground station to display all of that data. This type of project will no doubt expand my electronics and radio knowledge, but I think it might have me looking into programming use Python and some of the add-ons to allow a good looking display of the data. Can you see the challenges involved?*
>
> *"The other thing I am looking at is going back to one of the first projects I built, the Launch Control System. We have learned a lot since that first project, and I would like to create LCS 2.0, incorporating many of the lessons learned on the original project and incorporating new stuff we have learned since then. Who knows, maybe we can even combine a couple of projects together for a full fledged 'Mission Control' project that pulls all the pieces together."*

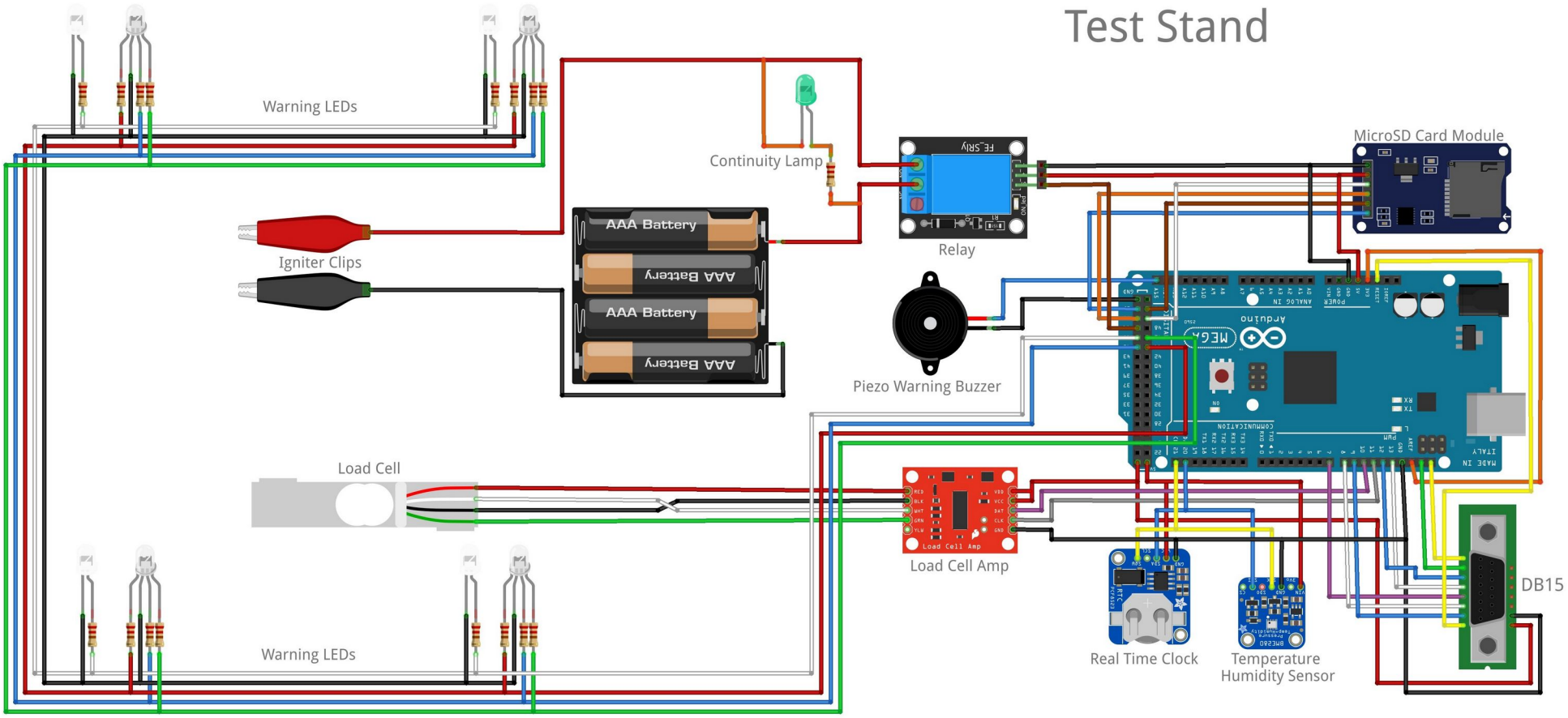
The purpose of this project manual is to help you, the reader, create new and better model rocketry projects. I hope it gets you to thinking about other projects that you can create, and not just in rocketry. Use your skills you developed in rocketry to create other things that may not have anything to do with model rockets. When that happens, you are really expanding your talents, and that is a good thing.

Appendix

A1
System Drawings

These drawings show the final setup of the avionics module. The first drawing shows the connections on the Test Stand platform. The second is the Remote Head.

Test Stand



Top of Test Stand

Underneath Test Stand

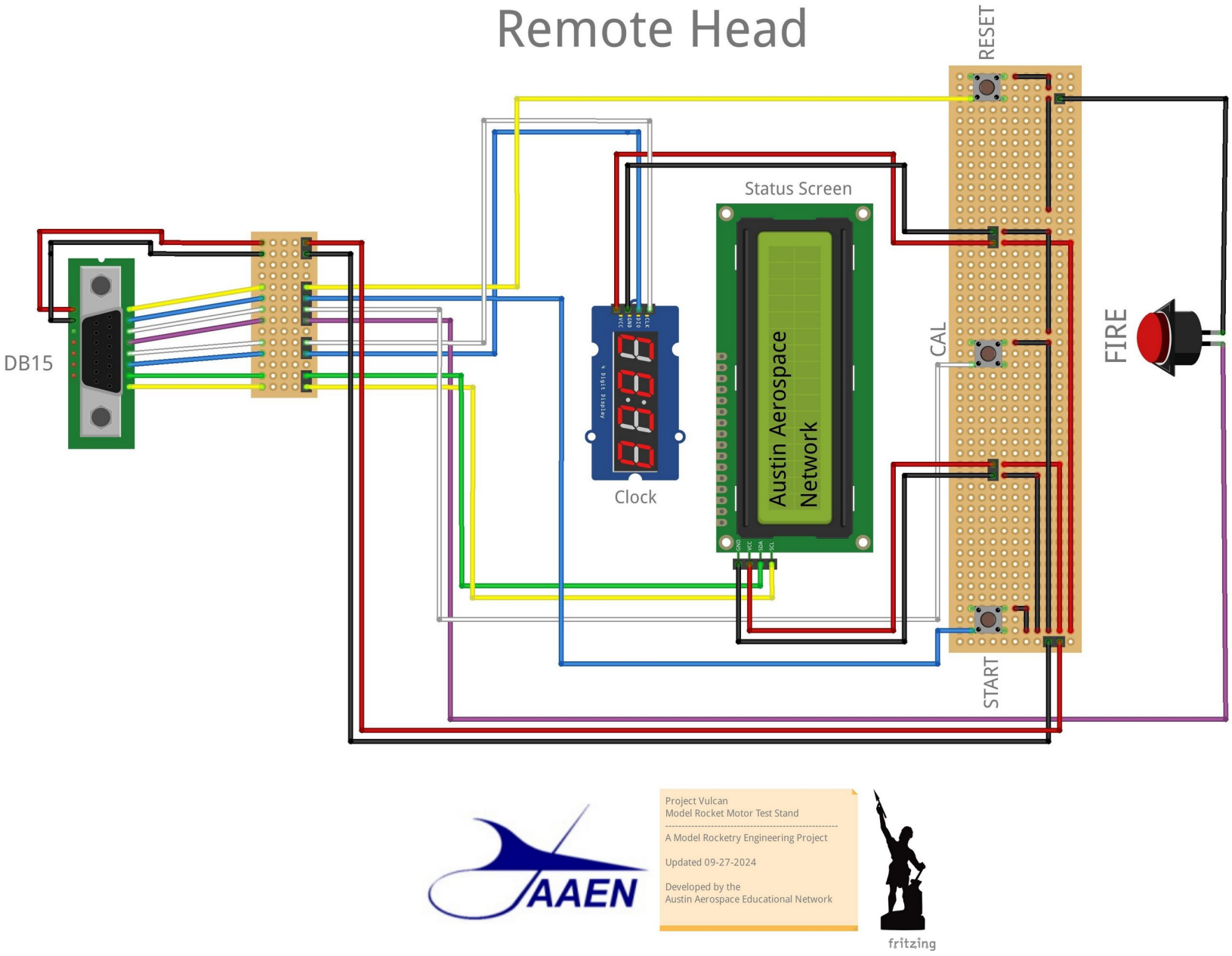
Project Vulcan
Model Rocket Motor Test Stand

A Model Rocketry Engineering Project

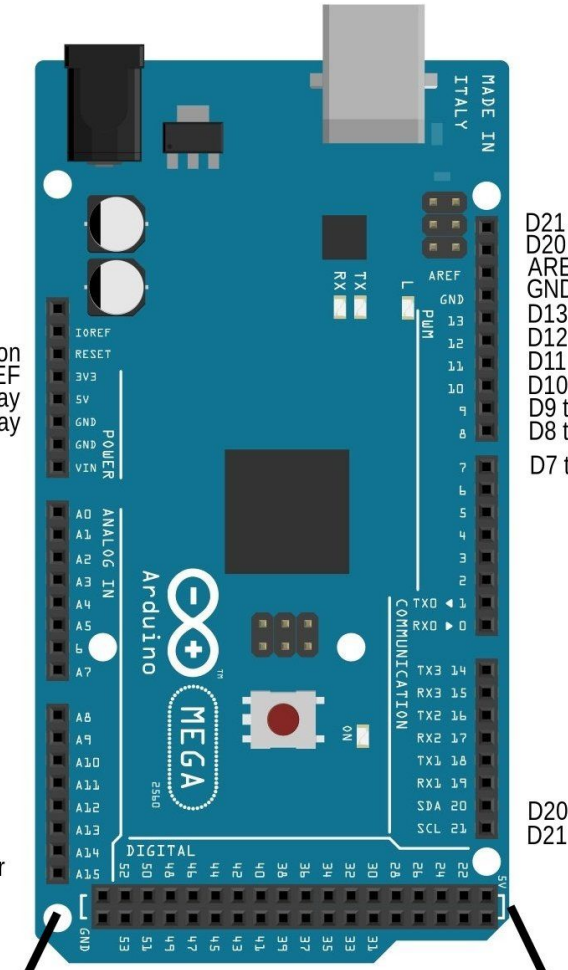
Updated 09-27-2024

Developed by the
Austin Aerospace Educational Network

fritzing


Remote Head
RESET
Status Screen
DB15
Clock
Austin Aerospace
Network
CAL
FIRE
START
Project Vulcan
Model Rocket Motor Test Stand
A Model Rocketry Engineering Project
Updated 09-27-2024
Developed by the
Austin Aerospace Educational Network
AAEN
fritzing

A2
MEGA 2560
Pin Assignments



D21 SCL to DB15/LCD Screen
D20 SDA to DB15/LCD Screen
AREF to 3V3
GND to HX711/BME280/RTC/LEDs/DB15
D13 to DB15/TM1637 DIO
D12 to DB15/TM1637 CLK
D11 HX711 SCK
D10 HX711 DOUT
D9 to DB15/Start Button
D8 to DB15/Calibrate Button

D7 to DB15/Fire Button

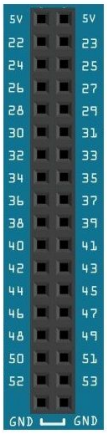
RESET to DB15/Reset Button
3V3 to AREF
5V to SD Card/5V Relay
GND to SD Card/5V Relay

D20 RTC/BME280 SDA
D21 RTC/BME280 SCL

A15 Piezo Buzzer

5V to RTC/BME280

5V to HX711/DB15



D44 RGB LED Red
D46 RGB LED Green

D50 SD Card MISO
D52 SD Card SCK

D45 RGB LED Blue
D47 LED Strobe
D49 5V Relay
D51 SD Card MOSI
D53 SD Card CS
GND Piezo Buzzer

A3
Complete Code Listing

On the following pages is the complete code listing for Version 1.0 of the Model Rocket Motor Test Stand. 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

Test_Stand_V1.0.ino

```
/* ********************************************************************
 * Project: Model Rocket Motor Test Stand
 * Version: 1.0.0
 * Description: This test stand is designed for black powder model
 *              rocket motors from mini "T" motors through "F"
 *              power motors. Test stand utilizes a single load cell
 *              and a HX711 amplifer.
 *
 *              Test stand includes a BMP280 temperature, humidity and
 *              barometric pressure sensor which is displayed on an LCD
 *              screen.
 *
 *              Incorporates a RTC to display time on a 4-digit
 *              7-segment LED.
 *
 * Created: 05 January 2023
 * Updated: 28 September 2024
 *
 * Author: Robert W. Austin
 * (C) Austin Aerospace Education Network
 * License: GPL-3.0
 *
 * ====================================================================
 * Based on the following coding examples:
 *
 *  TM1637 Clock Example
 *  https://www.makerguides.com/tm1637-arduino-tutorial/
 *
 *  Arduino - LCD I2C Tutorial
 *  https://arduinogetstarted.com/tutorials/arduino-lcd-i2c
 *
 *  BME280 Environmental Sensor
 *  https://www.waveshare.com/wiki/BME280_Environmental_Sensor
 *
 *  NM Rocketry Reviews Test Stand
 *  https://github.com/daniel360kim/motorteststand
 *
 *
 *  Datalogger (Examples -> (Examples for Any Board) SD -> Datalogger
 *
 *  Calibration (Examples -> (Examples from Custom Library) HX711_ADC -> Calibration
 *
 *  Read Load Cell (Examples -> (Examples from Custom Library) HX711_ADC -> Read 1x Load Cell
 *
 * ====================================================================
 * Pin configuration - for Mega 2560 board
 *
 * Peizo Buzzer
```

```
 *   SIG    A15
 *
 * Fire Button
 *   Red    7   (D7)
 *
 * Load Cell Calibration Button
 *   Cali   8   (D8) Calibrate
 *
 * Start Test Button
 *   Start  9   (D9)
 *
 * HX711
 *   DOUT   10  (D10)
 *   SCK    11  (D11)
 *   VIN    5V
 *
 * TM1637 4-Digit 7-Segment LCD Display
 *   CLK    12 (D12)
 *   DIO    13 (D13)
 *   VIN    5V
 *
 * RTC Real Time Clock
 *   SDA    20 (D20)
 *   SCL    21 (D21)
 *   VCC    5V
 *   Address  0x57
 *
 * BME280 Environmental Sensor
 *   SDA    20 (D20)
 *   SCL    21 (D21)
 *   VCC    5V
 *   Address  0x77
 *
 * LCD 16x2 Screen I2C
 *   SDA    20 (D20)
 *   SCL    21 (D21)
 *   VCC    5V
 *   Address  0x27
 *
 * RGB LED lamp
 *   Red    44   (D23)
 *   Blue   45   (D27)
 *   Green  46   (D25)
 *
 * Strobe LED Lamp
 *   SIG    47   (D29)

 * Fire Relay
 *   SIG    49   (D49)
 *   VIN    5V
 *
 * SD Card
 *   MISO   50 (D50)
 *   MOSI   51 (D51)
 *   SCK    52 (D52)
 *   CS     53 (D53)
 *   VCC    5V
 *
 ********************************************************************/


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

// ====================================================================
// library required for the Real Time Clock DS1307
```

```
   #include <RTClib.h>

// ========================================================================
// library required for the TM1637 display
   #include <TM1637Display.h>

// ========================================================================
// library required for the 16 x 2 LCD display
   #include <LiquidCrystal_I2C.h>

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

// ========================================================================
// libraries required for the BME280 Environmental Sensor
   #include <Wire.h>
   #include <Adafruit_Sensor.h>
   #include <Adafruit_BME280.h>

// ========================================================================
// libraries required for the Load Cell and HX711
   #include <HX711_ADC.h>
   #include <EEPROM.h>

// ========================================================================
// library required for flashing strobe lights
// regardless of other activities
   #include <MsTimer2.h>


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

// ========================================================================
// declarations for Program Version
   const int prgMajor = 1;
   const int prgMinor = 0;
   const int prgPatch = 0;

// ================================================================
// declaration for serial lines
   const String serialLine =
"================================================================================";
   const String serialLine2 = "---------------------------------";

// ========================================================================
// declaration for loop counter
   int x;

// ========================================================================
// declarations required for the Real Time Clock DS1307
   RTC_DS3231 rtc;

// variiables for rtc - to hold data while SD card is iniated
   String rtcRTCStamp = "";
   String rtcTimeStamp = "";
   String rtcMessage = "";
   String rtcResult = "";

// variiables for clock sync - to hold data while SD card is iniated
   String syncRTCStamp = "";
   String syncTimeStamp = "";
   String syncMessage = "";
   String syncResult = "";
```

```
// variable for serial monitor date and time (sprintf)
   char bufferDate[12];
   char bufferTime[12];

// used for LCD clock refresh
   DateTime now;
   unsigned long currentMillis = millis();
   unsigned long lastExecutedMillis = 0;

// ========================================================================
// declarations required for the 16x2 LCD I2C
// I2C address 0x27
   LiquidCrystal_I2C lcd(0x27, 16, 2);

// ========================================================================
// declarations for the MicroSD card
   float timeDataCollection;
   String dataString = "";   // make a string for assembling the data
                             // to the log

   char fileMotorLog[] = "MTRLOG00.CSV";
   String sdImpulse = "";
   String sdTotalImpulse = "";
   String sdTimeStamp = "";

   char fileSystemLog[] = "SYSLOG00.CSV";
   String sdRTCStamp = "";
   String sdMessage = "";
   String sdResult = "";

   char fileInfoLog[] = "INFLOG00.TXT";
   String sdInfo = "";
   String sdDate = "";
   String sdImpulseTotal = "";
   String sdThrustAvg = "";
   String sdDelay = "";
   //String sdData = "";

// ========================================================================
// declarations for BME280 environmental sensor
   Adafruit_BME280 sensorEnvironment;
   String sdTemperature = "";
   String sdPressure = "";
   String sdHumidity = "";

// ========================================================================
// declarations for LED bulbs
   const int ledStrobe = 47;

// ========================================================================
// declarations for RGB LED lamp pin numbers
   #define RED 44
   #define GREEN 45
   #define BLUE 46

// ========================================================================
// declarations required for Fire and Safety Switch
   const int pinFireRelay = 49;
   const int pinFireButton = 7;
   const int pinStartButton = 9;

   float timeFireRelay;
   boolean skip = false;

// ========================================================================
// declarations for pins to call calibrate and tare load cell
   const int pinCalibrate = 8;

// ========================================================================
// declarations for HX711
```

```
   const int serialDigitalOut = 10;
   const int powerDownSerialClock = 11;
   const int eepromAdressCalibrationValue = 0;

// declarations for calibration
   float knownMass;
   float newCalibrationValue;

   boolean performTare;
   boolean resumeProcedure;

   unsigned long timeStabilizing;
   char charInput;

// declarations for getting thrust data
   float timeMillis = micros()/1000000.f;
   float timeElapsed;
   float timeLast;
   float scaleNewtons;
   float impulseSingle;
   float impulseTotal;
   float timeData;
   float timeRelay;
   float timeNow;
   static boolean newDataReady = 0;

   HX711_ADC loadCell(serialDigitalOut, powerDownSerialClock);

// ========================================================================
// declarations required for the TM1637 4-digit display
   const int CLK = 12;
   const int DIO = 13;

   TM1637Display clockDisplay = TM1637Display(CLK, DIO);

// ========================================================================
// declarations for peizo buzzer
   const int buzzer = A15; // buzzer pin number

// ========================================================================
// declarations for motor specifications
   float motorTotalImpulse;
   int motorAverageThrust;
   int motorDelayTime;
// ========================================================================
// Constants for 4-Digit 7-segment display
   const uint8_t blank[] = {0x00, 0x00, 0x00, 0x00};       //Clear display

// Constants for FAIL
   const uint8_t SEG_FAIL[] =
   {
   SEG_A | SEG_E | SEG_F | SEG_G,                  // F
   SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,  // A
   SEG_E | SEG_F,                                  // I
   SEG_D | SEG_E | SEG_F                           // L
   };

/*****************************************************************
 *****************************************************************
 *                                                               *
 *                            SETUP                              *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setup()
{
  // ======================================================================
  // Initialize the serial port.
     Serial.begin(115200);
```

```
  // ========================================================================
  // Show Splash Screen
     splashScreenSerialMonitor();

  // ========================================================================
  // Initialize the LED 4-digit display.
     setupLedDisplay();

  // ========================================================================
  // Initialize the LCD 16x2 display.
     setupLcdDisplay();

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

  // ========================================================================
  // initialize digital pin 2 as an output.
     pinMode(ledStrobe, OUTPUT);

  // ========================================================================
  // Initialize peizo buzzer
     pinMode(buzzer,OUTPUT);

  // ========================================================================
  // Setup for Real Time Clock
     setupRTC();

  // ========================================================================
  // Check Date and Time
     setupDateTimeCheck();

  // ========================================================================
  // Setup for MicroSD card
     setupMicroSDCard();

  // ========================================================================
  // Setup for BME280 Sensor
     setupBmeSensor();

  // ========================================================================
  // Setup HX711 Load Cell
     setupHX711();

  // ========================================================================
  // Setup Fire Control System
     setupFireControl();

  // ========================================================================
  // Initialization All Sensors Passed
     initializationPass();
}

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

void loop()
{
  // ========================================================================
  // Show date and time on both displays
     ledClock();

  // Clock updates display once every second to reduce flicker
     currentMillis = millis();
```

```
     if (currentMillis - lastExecutedMillis >= 1000)
     {
      lastExecutedMillis = currentMillis; // save the last executed time
      lcdDateAndTime();
     }

     // Check for button selection
       if (digitalRead(pinCalibrate) == LOW) loadCellCalibrate();
       if (digitalRead(pinStartButton) == LOW) motorPrepAndTest();

}
```

Buzzer_Tones.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                          Buzzer                               *
 *                                                               *
 *****************************************************************
 *****************************************************************/


void buzzerWarningTone1(void)
// ===========================================================================
// A short series of tones
{
   int i;
     {
     for (i=0; i<50; i++)
      {
        digitalWrite(buzzer,HIGH);
        delay(50);
        digitalWrite(buzzer,LOW);
        delay(50);
      }
     }
}

void buzzerShortTone(void)
// ===========================================================================
// A short tone
{
   // quick chirp
      digitalWrite(buzzer,HIGH);
      delay(50);
      digitalWrite(buzzer,LOW);
}
```

Calibrate_Load_Cell.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    CALIBRATE LOAD CELL                        *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void loadCellCalibrate()
{
  // ===========================================================================
  // Constants for CAL for LED Clock
    const uint8_t SEG_CAL[] =
    {
    SEG_A | SEG_D | SEG_E | SEG_F,                  // C
    SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,  // A
    SEG_D | SEG_E | SEG_F                           // L
    };

  // declarations for calibration
    float knownMass;
    float newCalibrationValue;

    boolean performTare;

    unsigned long timeStabilizing;


  // ===========================================================================
  // Prerequisites for calibration

  // Preciscion right after power-up can be improved by adding a
  // few seconds of stabilizing time
     timeStabilizing = 2000;

  // Set this to false if you don't want tare to be performed
  // in the next step
     performTare = true;

     loadCell.start(timeStabilizing, performTare);

  // ===========================================================================
  // Start calibration procedure

  // show calibration in progress on LCD
     lcd.setCursor(0,0);
     lcd.print(F(" Calibration in "));
     lcd.setCursor(0,1);
     lcd.print(F("    Progress    "));

  // start time for system log
     //DateTime
     now = rtc.now();
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
     timeMillis = micros()/1000000.f;

  // System Log for start of calibration
     sdRTCStamp = bufferTime;
     sdTimeStamp = timeMillis;
     sdMessage = (F("Load Cell Calibration Process"));
     sdResult = (F("Start"));
     writeSysDataToCard();

  // show "CAL" on clock
     clockDisplay.setSegments(blank);
     clockDisplay.setSegments(SEG_CAL);
```

```
// Zero out (tare) the scale
   Serial.println(serialLine2);
   Serial.println(F("Start Calibration Process"));
   Serial.println();
   Serial.println(F("Place the Test Stand an a level stable surface."));
   Serial.println(F("Remove any load (weight) applied to the load cell."));
   Serial.println(F("Hit 't' followed by the 'ENTER' key from the Serial"));
   Serial.println(F("Monitor to set the tare offset."));
   Serial.println();

   resumeProcedure = false;
   while (resumeProcedure == false)
    {
      loadCell.update();
      if (Serial.available() > 0)
      {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          if (charInput == 't' || charInput == 'T') loadCell.tareNoDelay();
        }
      }
      if (loadCell.getTareStatus() == true)
      {
        Serial.println(F("Tare complete"));
        Serial.println();

        //DateTime
          timeMillis = micros()/1000000.f;
          now = rtc.now();
          sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());

        // System Log for Load Cell
           sdRTCStamp = bufferTime;
           sdTimeStamp = timeMillis;
           sdMessage = (F("Load Cell Tare"));
           sdResult = (F("Complete"));
           writeSysDataToCard();

        resumeProcedure = true;
      }
     }

// ========================================================================
// Calibrate load cell using an object of a known weight
   Serial.println(serialLine2);
   Serial.println(F("Next, place a calibration weight on the load cell."));
   Serial.println(F("Then enter the weight of this mass (i.e. 100.0) in "));
   Serial.println(F("grams using the serial monitor. Now hit the 'ENTER' key."));
   Serial.println();

   knownMass = 0.0;
   resumeProcedure = false;

   while (resumeProcedure == false)
    {
      loadCell.update();
      if (Serial.available() > 0)
      {
        knownMass = Serial.parseFloat();
        if (knownMass != 0)
        {
          Serial.print(F("Known mass is: "));
          Serial.print(knownMass);
          Serial.println(F(" grams"));
          Serial.println();

          //DateTime
          timeMillis = micros()/1000000.f;
          now = rtc.now();
          sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
```

```
          // System Log for Load Cell
          sdRTCStamp = bufferTime;
          sdTimeStamp = timeMillis;
          sdMessage = (F("Load Cell Calibration Weight (grams)"));
          sdResult = (knownMass);
          writeSysDataToCard();

          resumeProcedure = true;
        }
      }
    }

// ====================================================================
// Calculate the new calibration value
// refresh the dataset to be sure that the known mass is measured correct
   loadCell.refreshDataSet();

// get the new calibration value
   newCalibrationValue = loadCell.getNewCalibration(knownMass);

   Serial.println(serialLine2);
   Serial.print(F("New calibration value has been set to: "));
   Serial.println(newCalibrationValue);

   //DateTime
   timeMillis = micros()/1000000.f;
   now = rtc.now();
   sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());

   sdRTCStamp = bufferTime;
   sdTimeStamp = timeMillis;
   sdMessage = (F("Load Cell Calibration Value"));
   sdResult = (newCalibrationValue);
   writeSysDataToCard();

   //DateTime
   now = rtc.now();
   sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
   timeMillis = micros()/1000000.f;

   sdRTCStamp = bufferTime;
   sdTimeStamp = timeMillis;
   sdMessage = (F("Load Cell Calibration Calculation"));
   sdResult = (F("Complete"));
   writeSysDataToCard();

// ====================================================================
// Save new value to EEPROM
   Serial.print(F("Do you want to save the new calibration value to EEPROM adress "));
   Serial.print(eepromAdressCalibrationValue);
   Serial.println(F("? y/n"));
   Serial.println();

   resumeProcedure = false;
   while (resumeProcedure == false)
     {
      if (Serial.available() > 0)
      {
        charInput = Serial.read();
        if (charInput == 'y' || charInput == 'Y')
        {
          EEPROM.put(eepromAdressCalibrationValue, newCalibrationValue);

          EEPROM.get(eepromAdressCalibrationValue, newCalibrationValue);
          Serial.print(F("Value "));
          Serial.print(newCalibrationValue);
          Serial.print(F(" has been saved at EEPROM address: "));
          Serial.println(eepromAdressCalibrationValue);
          Serial.println();
```

```
            //DateTime
            now = rtc.now();
            sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
            timeMillis = micros()/1000000.f;

            // System Log for Load Cell
               sdRTCStamp = bufferTime;
               sdTimeStamp = timeMillis;
               sdMessage = (F("Load Cell Calibration Stored to EEPROM"));
               sdResult = (F("Yes"));
               writeSysDataToCard();

               sdRTCStamp = bufferTime;
               sdTimeStamp = timeMillis;
               sdMessage = (F("Load Cell Calibration EEPROM Address"));
               sdResult = (eepromAdressCalibrationValue);
               writeSysDataToCard();

            resumeProcedure = true;
          }
          else if (charInput == 'n' || charInput == 'N')
          {
            Serial.println(F("New calibration value was not saved to the EEPROM"));

            //DateTime
            now = rtc.now();
            sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
            timeMillis = micros()/1000000.f;

            // System Log for Load Cell
               sdRTCStamp = bufferTime;
               sdTimeStamp = timeMillis;
               sdMessage = (F("Load Cell Calibration Stored to EEPROM"));
               sdResult = (F("No"));
               writeSysDataToCard();

            resumeProcedure = true;
          }
        }
      }

  // ====================================================================
  // Calibration procedure complete
     Serial.println(F("Remove the calibration weight from the test stand"));
     Serial.println();
     Serial.println(serialLine);
     Serial.print(F("This concludes the calibration procedure at "));
     dateTimeSerialMonitor();
     sdTimeStamp = millis();
     Serial.println(serialLine);
     Serial.println();

  // show calibration completed on LCD
     lcd.setCursor(0,0);
     lcd.print(F("  Calibration  "));
     lcd.setCursor(0,1);
     lcd.print(F("   Completed   "));

  // System Log for calibration success
     //DateTime
     now = rtc.now();
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
     timeMillis = micros()/1000000.f;

     sdRTCStamp = bufferTime;
     sdMessage = ("Load Cell Calibration Process");
     sdResult = ("Complete");
     writeSysDataToCard();

     delay(2000);
}
```

Clock_LED.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                       LED CLOCK DISPLAY                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void ledClock(void)
{
   // declare local variable
      int displayTime;

   // Get current date and time
       now = rtc.now();

  // Create time format to display
     displayTime = (now.hour() * 100) + now.minute();

  // Display the current time in 24 hour format
  // with leading zeros enabled and a center colon:
     clockDisplay.showNumberDecEx(displayTime, 0b11100000, true);
}
```

Fire_Abort_Recycle.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    FIRE ABORT RECYCLE                         *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void abortRecycle(void)
{
  // Show recycle message and instructions
     Serial.println();
     Serial.println(F("Recycle from Abort"));
     Serial.println();
     Serial.println(F("When conditions are favorable to reinitiate the test"));
     Serial.println(F("you can choose to recycle the aborted test. During a "));
     Serial.println(F("recycle the program will return to the Clear Test"));
     Serial.println(F("Stand section. The test can be restarted from that point"));
     Serial.println(F("using the test data that has already been entered"));
     Serial.println();
     Serial.println(F("When ready to recycle the test,"));
     Serial.println(F("hit 'r' followed by the 'ENTER' key."));
     Serial.println();

     // Get recycle input
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          {
            if (charInput == 'r' || charInput == 'R')
            {
```

```
              Serial.println(F("Recycle of aborted test initiated at"));
              dateTimeSerialMonitor();
              timeMillis = micros()/1000000.f;
              Serial.println();
              Serial.println();

             // System Log for Test Area Cleared
                sdRTCStamp = bufferTime;
                sdTimeStamp = timeMillis;
                sdMessage = (F("Recycle from Abort"));
                sdResult = (F("Initiated"));
                writeSysDataToCard();

              resumeProcedure = true;
            }
          }
        }
      }

  // Recycle to clear test stand function
     motorClearArea();

  // ================================================================
  // Show header on Serial Monitor
     Serial.println(serialLine);
     Serial.println(F("Begin Motor Test Fire Process"));
     Serial.println(serialLine);
     Serial.println();

  // Fire Sequence
     fireSequence();
}
```

Fire_Abort_Sequence.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                   FIRE ABORT SEQUENCE                         *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void abortTest(void)
{
  // ================================================================
  // Abort Sequence
  // If the Fire button reports pin as HIGH
  // (button was released) assume abort

  // Print abort on serial monitor
     Serial.println();
     Serial.println(serialLine);
     Serial.println(F("             * * * * * *   A B O R T   * * * * * *"));
     Serial.print(F("          Test aborted at "));
     dateTimeSerialMonitor();
     timeMillis = micros()/1000000.f;
     Serial.println(serialLine);

  // Print abort on LCD
     lcd.setCursor(0,0);
     lcd.print(F(" *    TEST    * "));
     lcd.setCursor(0,1);
     lcd.print(F(" *  ABORTED!  * "));

  // show FAIL on LED
```

```
    clockDisplay.setSegments(SEG_FAIL);

  // sound buzzer
    buzzerWarningTone1();

  // System Log for Abort Fire
    sdRTCStamp = bufferTime;
    sdTimeStamp = timeMillis;
    sdMessage = (F("Fire Control Sequence"));
    sdResult = (F("ABORT"));
    writeSysDataToCard();

  // Go to Recycle
    abortRecycle();
}
```

Fire_Control_Sequence.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    FIRE CONTROL SEQUENCE                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void fireSequence(void)
{
  // Constants for countdown for LED clock
    const uint8_t SEG_05[] =
    {
    SEG_D | SEG_E | SEG_F | SEG_G,                          // t
    SEG_G,                                                  // -
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,          // O
    SEG_A | SEG_C | SEG_D | SEG_F | SEG_G                   // 5
    };

    const uint8_t SEG_04[] =
    {
    SEG_D | SEG_E | SEG_F | SEG_G,                          // t
    SEG_G,                                                  // -
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,          // O
    SEG_B | SEG_C | SEG_F | SEG_G                           // 4
    };

    const uint8_t SEG_03[] =
    {
    SEG_D | SEG_E | SEG_F | SEG_G,                          // t
    SEG_G,                                                  // -
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,          // O
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_G                   // 3
    };

    const uint8_t SEG_02[] =
    {
    SEG_D | SEG_E | SEG_F | SEG_G,                          // t
    SEG_G,                                                  // -
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,          // O
    SEG_A | SEG_B | SEG_D | SEG_E | SEG_G                   // 2
    };

    const uint8_t SEG_01[] =
    {
    SEG_D | SEG_E | SEG_F | SEG_G,                          // t
    SEG_G,                                                  // -
```

```
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,           // 0
    SEG_B | SEG_C                                           // 1
    };

  // Constants for FIRE
    const uint8_t SEG_FIRE[] =
    {
    SEG_A | SEG_E | SEG_F | SEG_G,                           // F
    SEG_E | SEG_F,                                           // I
    SEG_E | SEG_G,                                           // R
    SEG_A | SEG_D | SEG_E | SEG_F | SEG_G                    // E
    };

  // ===================================================================
  // Ready for Test Firing
  // time
     now = rtc.now();
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
     timeMillis = micros()/1000000.f;

  // System Log for Fire Sequence
     sdTimeStamp = timeMillis;
     sdRTCStamp = bufferTime;
     sdMessage = (F("Fire Control Sequence"));
     sdResult = (F("Ready"));
     writeSysDataToCard();

  // Standby firing sequence on LCD screen
     lcd.setCursor(0,0);
     lcd.print(F("STANDING BY FOR "));
     lcd.setCursor(0,1);
     lcd.print(F(" FIRE SEQUENCE  "));

  // column headers
     Serial.println(F("Column Header Explanation"));
     Serial.println();
     Serial.println(F("During the test, three columns will be displayed"));
     Serial.println(F("The data displayed is as follows:"));
     Serial.println(F("\t Column One - Elasped Time (in milliseconds)"));
     Serial.println(F("\t Column Two - Thrust (measured in Newtons)"));
     Serial.println(F("\t Column Three - Total Impulse (measured in Newtons)"));
     Serial.println();

  // Display delay time entry instructions on Serial Monitor
     Serial.println(serialLine);
     Serial.println(F("==
=="));
     Serial.println(F("==                      Ready to Test Fire
=="));
     Serial.println(F("==           The system is now ready for a test firing.
=="));
     Serial.println(F("==
=="));
     Serial.println(F("==  Press and hold the red 'Fire' button to proceed through the five
=="));
     Serial.println(F("== second countdown. Release the red 'Fire' button to ABORT the test.
=="));
     Serial.println(F("==
=="));
     Serial.println(serialLine);
     Serial.println();

  // Fire button sequence
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
      if (digitalRead(pinFireButton) == LOW)
       resumeProcedure = true;
     }
```

```
// show Fire button pressed on screen
   Serial.println(serialLine);
   Serial.print(F("Ignition Sequence Started at "));
   dateTimeSerialMonitor();
   timeMillis = micros()/1000000.f;
   Serial.println(serialLine);

// System Log for Fire Sequence
   sdRTCStamp = bufferTime;
   sdTimeStamp = timeMillis;
   sdMessage = (F("Fire Control Sequence"));
   sdResult = (F("Countdown Initiated"));
   writeSysDataToCard();

// Display launch sequence on LCD screen
   lcd.setCursor(0,0);
   lcd.print(F("Fire Sequence   "));
   lcd.setCursor(0,1);
   lcd.print(F("Initiated       "));

// Start loop LED countdown clock from t-5
   for (int countdown = 5; countdown > 0; countdown --)
   {
    if (countdown == 5)
      {
      clockDisplay.setSegments(SEG_05);
      buzzerShortTone();
      delay(1000);
      }
    else if (countdown == 4)
      {
      clockDisplay.setSegments(SEG_04);
      buzzerShortTone();
      delay(1000);
      }
    else if (countdown == 3)
      {
      clockDisplay.setSegments(SEG_03);
      buzzerShortTone();
      delay(1000);
      }
    else if (countdown == 2)
      {
      clockDisplay.setSegments(SEG_02);
      buzzerShortTone();
      delay(1000);
      }
    else if (countdown == 1)
      {
      clockDisplay.setSegments(SEG_01);
      buzzerShortTone();
      delay(1000);
      }

    // Check for ABORT
    // Fire button must remain depressed during countdown otherwise an ABORT is called
       if (digitalRead(pinFireButton) == HIGH)
        {
          // jump to Fire Abort Sequence
             abortTest();

          // return to loop()
             return;
        }
   }

// ====================================================================
// Start test stand power sequence

// Weather data
   fireWeather();
```

```
  // show FIRE on LED
     clockDisplay.setSegments(SEG_FIRE);

  // Display power flowing to stand on LCD screen
     lcd.setCursor(0,0);
     lcd.print(F("Power Flowing to"));
     lcd.setCursor(0,1);
     lcd.print(F("   Test Stand   "));

  // show power is flowing to test stand on Serial Monitor
     Serial.println();
     Serial.print(F("Power flowing to the Test Stand at "));
     dateTimeSerialMonitor();
     timeMillis = micros()/1000000.f;
     Serial.println(serialLine2);

  // System Log for Fire Sequence
     sdTimeStamp = timeMillis;
     sdRTCStamp = bufferTime;
     sdMessage = (F("Fire Control Sequence"));
     sdResult = (F("Ignition"));
     writeSysDataToCard();

  // start buzzer
     digitalWrite(buzzer,HIGH);

  // open relay to send power to igniter
     digitalWrite(pinFireRelay, HIGH);

  // Data collection routine
     fireDataCollection();

  // ========================================================================
  // Show Post Test Header on Serial Monitor
     Serial.println();
     Serial.println(serialLine);
     Serial.println(F("Begin Motor Post Test Fire and Data Entry Process"));
     Serial.println(serialLine);

  // Post fire shutdown safety period
     fireShutdown();

  // Post fire shutdown data entry
     postTestDataEntry();
}
```

Fire_Data_Collection.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                FIRE DATA COLLECTION SEQUENCE                  *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void fireDataCollection(void)
{
  // ========================================================================
  // Write the following in this section
  //  - time stamp in milliseconds
  //  - data from load cell
  //  - call to write data to SD card

  /*
```

```
  This routine collects data from the load cell and converts it into Newtons
  It uses that data to determine impulse for each sensor reading and to
  calculate overall inpulse

  timeMillis;     Current time in microseconds, divided by 1,000,000 to
                  convert to milliseconds
  scaleNewtons;   scale.get_units()*0.009806f; [scale units(in grams)
                  multiplied by 0.009806 to convert to Newtons. Is used
                  to convert grams to newtons where 1 gram = 0.0098067 newtons]
  timeElapsed;    Elapsed time [In setup is 1 divided by 80 as the scale gets
                  80 samples per second - 80Hz]
                  timeMillis-timeLast; [In main loop,it is the elapsed time,
                  calculated by subtracting the current time from the last time
                  (at end of loop)
  impulseSingle;  scaleNewtons*timeElapsed;  [Provides impulse for a single
                  sensor reading]
  impulseTotal;   impulse + impulseSingle; [Adds previous impulse to current
                  impulse to provide total impulse at each sensor reading]
  timeLast;       timeLast = timeMillis; [Gets time at end of loop in microseconds}
  timeData;       timeMillis + 10.f; [In setup, takes the current time and adds 10
                  seconds to it]
*/

// ========================================================================
// Get data during burn
   resumeProcedure = false;
   skip = false;

   timeNow = millis();
   timeData = timeDataCollection + timeNow;
   timeRelay = (timeFireRelay * 1000) + timeNow;

// loop to get data from load cell
   while(resumeProcedure == false)
     {
     // check for new data/start next conversion:
        if (loadCell.update()) newDataReady = true;
        {
         // collect data
            if (newDataReady)
            {
             // get motor thrust data
             timeMillis = micros()/1000000.f;
             scaleNewtons = loadCell.getData() * 0.009806f;
             timeElapsed = timeMillis - timeLast;
             impulseSingle = scaleNewtons * timeElapsed;
             impulseTotal = impulseTotal + impulseSingle;
             timeLast = timeMillis;
             newDataReady = 0;

          // print results to serial monitor
             Serial.print(timeMillis,10);
             Serial.print("\t");
             Serial.print(scaleNewtons,10);
             Serial.print("\t");
             Serial.print(impulseTotal,10);
             Serial.println("\t");

          // write motor data to micro sd card
             writeMotorDataToSDCard();
            }
        }

         // relay shutoff after selected amount of time
            if (skip == false)
            {
             if (millis() > (timeRelay))
              {
                // Turn off power to pad
                // reset relay to off
                   digitalWrite(pinFireRelay, LOW);
```

```
                // stop buzzer
                   digitalWrite(buzzer,LOW);

                   Serial.println();
                   Serial.println(F("Power has stopped flowing to the Test Stand"));
                   dateTimeSerialMonitor();
                   Serial.println();

                // write to system log
                   timeMillis = micros()/1000000.f;
                   sdRTCStamp = bufferTime;
                   sdTimeStamp = timeMillis;
                   sdMessage = (F("Power to Igniter"));
                   sdResult = (F("Off"));
                   writeSysDataToCard();

                  skip = true;
               }
            }

          // check to see if data collection time has been exceeded
              if (millis() > (timeData))
               {
                 Serial.println();
                 Serial.println(F("Test Data Collection Concluded"));
                 dateTimeSerialMonitor();
                 Serial.println();

                 resumeProcedure = true;
               }
      }
}
```

Fire_Shutdown_Period.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                POST FIRE SHUTDOWN SAFETY PERIOD               *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void fireShutdown(void)
{
  // ===========================================================================
  // Constants for LED clock

    // Constants for STAY
      const uint8_t SEG_STAY[] =
      {
      SEG_A | SEG_C | SEG_D | SEG_F | SEG_G,                  // S
      SEG_D | SEG_E | SEG_F | SEG_G,                          // t
      SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,          // A
      SEG_B | SEG_C | SEG_D | SEG_F | SEG_G                   // Y
      };

    // Constants for BACK
      const uint8_t SEG_BACK[] =
      {
      SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G,  // B
      SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,          // A
      SEG_A | SEG_D | SEG_E | SEG_F,                          // C
      SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,                  // K
      };

    // Constants for SAFE
```

```
    const uint8_t SEG_SAFE[] =
    {
    SEG_A | SEG_C | SEG_D | SEG_F | SEG_G,                  // S
    SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,          // A
    SEG_A | SEG_E | SEG_F | SEG_G,                          // F
    SEG_A | SEG_D | SEG_E | SEG_F | SEG_G                   // E
    };

// ====================================================================
// Backup to turn off power to pad in case it was not done earlier
  // reset relay to off
     digitalWrite(pinFireRelay, LOW);

  // stop buzzer
     digitalWrite(buzzer,LOW);

// ====================================================================
// Start 60-second wait period before approaching pad

// Yellow lamps on test stand
   rgbYellow();
   rgbSteadyLamp();

// show power has stopped flowing on screen if not done earlier
   if (skip == false)
   {
    Serial.println();
    Serial.println(F("Power has stopped flowing to the Test Stand"));
    dateTimeSerialMonitor();
    timeMillis = micros()/1000000.f;

    // System Log for Post Fire Shutdown
       sdTimeStamp = timeMillis;
       sdRTCStamp = bufferTime;
       sdMessage = (F("Post Ignition Shutdown"));
       sdResult = (F("Power Off"));
       writeSysDataToCard();
   }

// One minute "Stay Away" post firing safety period
   Serial.println();
   Serial.println(F("Begin 1-minute post test safety period."));
   Serial.println(F("Stay Clear of the Test Stand till 'All Clear' is given"));
   dateTimeSerialMonitor();
   timeMillis = micros()/1000000.f;

// System Log for Post Fire Shutdown
   sdTimeStamp = timeMillis;
   sdRTCStamp = bufferTime;
   sdMessage = (F("Post Test Safety Period"));
   sdResult = (F("Start"));
   writeSysDataToCard();

// Start loop for pad safety countdown
   for (int safety = 60; safety > 0; safety -=10)
   {
    clockDisplay.setSegments(SEG_STAY);
    lcd.setCursor(0,0);
    lcd.print(F("Keep Stand Clear"));
    lcd.setCursor(0,1);
    lcd.print(F(" for "));
    lcd.print(safety);
    lcd.print(F("-seconds "));

    // show data while looping for 5 seconds
    for (x = 1; x <= 5 ;x++)
    {
      delay (1000);
    }

    clockDisplay.setSegments(SEG_BACK);
```

```
      lcd.setCursor(0,1);
      lcd.print(F(" for "));
      lcd.print(safety-5);
      lcd.print(F("-seconds "));

      // show data while looping for 5 seconds
      for (x = 1; x <= 5 ;x++)
      {
        delay (1000);
      }

     }

  // show green lamp on test stand
     rgbGreen();
     rgbSteadyLamp();

  // turn off strobes
     MsTimer2::stop();

  // return to normal operations
     Serial.println();
     Serial.println(F("All Clear Announced"));
     dateTimeSerialMonitor();
     Serial.println();

     timeMillis = micros()/1000000.f;

  // System Log for Post Fire Shutdown
     sdTimeStamp = timeMillis;
     sdRTCStamp = bufferTime;
     sdMessage = (F("Post Test Safety Period"));
     sdResult = (F("Finished"));
     writeSysDataToCard();

  // Display "SAFE" LED countdown clock
     clockDisplay.setSegments(SEG_SAFE);

  // Indicate safe on LCD
     lcd.setCursor(0,0);
     lcd.print(F("It is safe to   "));
     lcd.setCursor(0,1);
     lcd.print(F("approach stand  "));

  // quick chirp
     buzzerShortTone();
}
```

Fire_Weather.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                FIRE WEATHER DATA COLLECTION                   *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void fireWeather(void)
{
  // ===========================================================================
  // get weather data
     sensorEnviroValues();

  // time of sensor readings
     now = rtc.now();
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
```

```
    timeMillis = micros()/1000000.f;

  // System Log for BME 280 Sensor Readings - Temperature
    sdTimeStamp = timeMillis;
    sdRTCStamp = bufferTime;
    sdMessage = (F("Air Temperature"));
    sdResult = (sdTemperature + "* C");
    writeSysDataToCard();

  // System Log for BME 280 Sensor Readings - Barometric Pressure
    sdMessage = (F("Barometric Pressure"));
    sdResult = (sdPressure + " hPa");
    writeSysDataToCard();

  // System Log for BME 280 Sensor Readings - Humidity
    sdMessage = (F("Humidity"));
    sdResult = (sdHumidity + "%");
    writeSysDataToCard();
}
```

Initialization_Pass.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *              INITIALIZATION PASS SEQUENCE                     *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void initializationPass(void)
{
  // Constants for PASS for LED clock
    const uint8_t SEG_PASS[] =
    {
    SEG_A | SEG_B | SEG_E | SEG_F | SEG_G,                  // P
    SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,          // A
    SEG_A | SEG_C | SEG_D | SEG_F | SEG_G,                  // S
    SEG_A | SEG_C | SEG_D | SEG_F | SEG_G                   // S
    };

  // ======================================================================
  // show PASS on LED
    clockDisplay.setSegments(SEG_PASS);
  // ======================================================================
  // Show all sensors initialized on LCD screen
    lcd.setCursor(0,0);
    lcd.print(F("  All Systems   "));
    lcd.setCursor(0,1);
    lcd.print(F("  Initialized   "));

  // card initialized properly and is ready to start writing data
    Serial.println(serialLine);
    Serial.print(F("All sensors and systems initialized at "));

  // print date and time of completion
    dateTimeSerialMonitor();
    timeMillis = micros()/1000000.f;
    Serial.println(serialLine);

  // System Log for Initialization Complete
    sdRTCStamp = bufferTime;
    sdTimeStamp = timeMillis;
    sdMessage = (F("Initialization of All Sensors and Systems"));
    sdResult = (F("Successful"));
    writeSysDataToCard();
```

```
  // show green LED lamp
     rgbGreen();
     rgbSteadyLamp();

  // sound buzzer
     buzzerShortTone();

  // delay for 3 seconds
     delay(3000);

  // print instructions to start testing
     Serial.println();
     Serial.println();
     Serial.println(serialLine);
     Serial.println(F("==
=="));
     Serial.println(F("==             Press Green Button to Start Test Procedures
=="));
     Serial.println(F("==
=="));
     Serial.println(serialLine);
     Serial.println();
}
```

LCD_Date_Time.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *               DATE AND TIME DISPLAYED ON LCD                  *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void lcdDateAndTime(void)
{
  // Get Date & Time Data
     //DateTime
     now = rtc.now();

  // Display Date
     lcd.setCursor(0,0);
     lcd.print(F("Date:           "));
     lcd.setCursor(6,0);

lcd.print(now.month());lcd.print(F("/"));lcd.print(now.day());lcd.print(F("/"));lcd.print(now.yea
r());

  // Display Time
     lcd.setCursor(0,1);
     lcd.print(F("Time:           "));
     lcd.setCursor(6,1);

     //add leading zeros if needed
     sprintf(bufferDate, "%02u:%02u:%02u ", now.hour(), now.minute(), now.second());
     lcd.print(bufferDate);
}
```

Motor_Load_Sequence.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                       MOTOR LOAD SEQUENCE                     *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorLoad(void)
{
  // Constants for LOAD for LED clock
    const uint8_t SEG_LOAD[] =
    {
    SEG_D | SEG_E | SEG_F,                          // L
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,  // O
    SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,  // A
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F   // D
    };

  // ===================================================================
  // motor loading sequence

  // Turn test stand LEDs to Yellow
     rgbYellow();
     rgbSteadyLamp();

  // Turn on white strobe lamps
     MsTimer2::set(500, ledWhiteStrobeLamp); // 500ms period
     MsTimer2::start();

  // show "LOAD" on clock
     clockDisplay.setSegments(SEG_LOAD);

  // show loading sequence in progress on LCD
     lcd.setCursor(0,0);
     lcd.print(F(" Motor Loading  "));
     lcd.setCursor(0,1);
     lcd.print(F("  In Progress   "));

  // ===================================================================
  // Motor loading checklist

  // Display Total Impulse entry instructions on Serial Monitor
     Serial.println();
     Serial.println(F("Motor Loading Checklist"));
     Serial.println();
     Serial.println(F("Make sure the following tasks have been completed:"));
     Serial.println(F("  - Motor has igniter installed."));
     Serial.println(F("  - Motor is installed in appropriate motor mount."));
     Serial.println(F("  - The motor retaining ring is secured in place."));
     Serial.println();
     Serial.println(F("When these tasks have been completed and verified,"));
     Serial.println(F("hit 'v' followed by the 'ENTER' key."));
     Serial.println();

  // Get total impulse of the motor being tested
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          {
            if (charInput == 'v' || charInput == 'V')
            {
              Serial.println(F("Motor load complete and verified"));
              dateTimeSerialMonitor();
```

```
              timeMillis = micros()/1000000.f;
              Serial.println();

              // System Log for Motor Loading
                 sdRTCStamp = bufferTime;
                 sdTimeStamp = timeMillis;
                 sdMessage = (F("Motor Loading Checklist"));
                 sdResult = (F("Verified"));
                 writeSysDataToCard();

              resumeProcedure = true;
            }
          }
        }
      }
}
```

Motor_Mount_Sequence.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                      MOTOR MOUNT SEQUENCE                     *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorMount(void)
{
  // ===========================================================================
  // Motor mount checklist

  // Display motor mount instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Mount Checklist"));
     Serial.println();
     Serial.println(F("Make sure the following tasks have been completed:"));
     Serial.println(F("  - Test stand threaded adapter is clean and free of debris."));
     Serial.println(F("  - Motor mount is installed in test stand threaded adapter."));
     Serial.println(F("  - Motor mount is secured in test stand threaded adapter."));
     Serial.println(F("  - Micro clips are attached to igniter and are not touching."));
     Serial.println();
     Serial.println(F("When these tasks have been completed and verified,"));
     Serial.println(F("hit 'v' followed by the 'ENTER' key."));
     Serial.println();

  // Get verification of the motor mount checklist
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          {
            if (charInput == 'v' || charInput == 'V')
            {
              Serial.println(F("Motor mounting procedure complete and verified"));
              dateTimeSerialMonitor();
              timeMillis = micros()/1000000.f;
              Serial.println();

              // System Log for Motor Mount
                 sdRTCStamp = bufferTime;
                 sdTimeStamp = timeMillis;
                 sdMessage = (F("Motor Mount Checklist"));
```

```
                sdResult = (F("Verified"));
                writeSysDataToCard();

              resumeProcedure = true;
            }
          }
        }
      }
}
```

Motor_Prep_And_Test.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *             MOTOR PREPARATION AND TEST SEQUENCE               *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepAndTest(void)
{
  // if Start button is HIGH assume rogue button push
     if (digitalRead(pinStartButton) == HIGH)
       {
        return;
       }

  // ===============================================================================
  // Test Preparation

  // Show header on Serial Monitor
     Serial.println();
     Serial.println(serialLine);
     Serial.println(F("Begin Test Preparation and Data Entry Process"));
     Serial.println(serialLine);

  // get prep information
     motorPrepInfo();

  // get casing information
     motorCasingInfo();

  // get Total Impulse letter rating
     motorPrepTotalImpulse();

  // get Average Thrust rating
     motorPrepAvgThrust();

  // get Delay Time rating
     motorPrepDelayTime();

  // get propellant information
     motorPrepPropellant();

  // get time for relay to fire
     motorPrepIgnitionTime();

  // calculate total data collection time
     motorPrepCalcDataTime();

  // ===============================================================================
  // Show header on Serial Monitor
     Serial.println();
     Serial.println(serialLine);
     Serial.println(F("Begin Motor Loading and Pre-Fire Process"));
```

```
     Serial.println(serialLine);

  // Load rocket motor into mount
     motorLoad();

  // Insert mount into test stand threaded adapter
     motorMount();

  // Calibrate stand if not done earlier
     motorRecalibrate();

  // Clear the Area
     motorClearArea();

  // Prep scale for test
     motorScalePreparation();

  // ========================================================================
  // Show header on Serial Monitor
     Serial.println(serialLine);
     Serial.println(F("Begin Motor Test Fire Process"));
     Serial.println(serialLine);
     Serial.println();

  // Fire Sequence
     fireSequence();
}
```

Motor_Prep_Avg_Thrust.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *             MOTOR PREP AVERAGE THRUST SEQUENCE                *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepAvgThrust(void)
{
  // ============================================================
  // Get motor average thrust information

  // Display Average Thrust entry instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Average Thrust"));
     Serial.println();
     Serial.println(F("Enter the number for the average thrust of the motor being tested."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get average thrust of the motor being tested
     motorAverageThrust = 0;
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
          {
           motorAverageThrust = Serial.parseInt();
           if (motorAverageThrust != 0)
            {
            // Display average thrust entered
               Serial.print(F("Average thrust entered: "));
               Serial.println(motorAverageThrust);
               Serial.println();
```

```
                sdThrustAvg = motorAverageThrust;
                resumeProcedure = true;
            }
          }
     }
}
```

Motor_Prep_Calc_Time.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *       MOTOR PREP CALCULATE DATA COLLECTION TIME SEQUENCE      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepCalcDataTime(void)
{
/*****************************************************************
 *   Determine length of time that data should be collected. This is
 *   calculated by dividing the Total Impulse by the Avergae Thrust.
 *   The Delay Time is added and then the Firing Time is added. One
 *   additional second is added to the total. The total is multipled
 *   by 1000 to to derive a total time in milliseconds
 *****************************************************************/
  float timeBurn;
  float timeMotor;

  timeBurn = (motorTotalImpulse/motorAverageThrust);
  timeMotor = (motorDelayTime + timeFireRelay);

  timeDataCollection = (((timeBurn + timeMotor) * 1000) + 1000);

  Serial.print(serialLine2);
  Serial.println(serialLine2);
  Serial.print(F("Total data collection time: "));
  Serial.print(timeDataCollection);
  Serial.println(F(" milliseconds"));
  Serial.print(serialLine2);
  Serial.println(serialLine2);
}
```

Motor_Prep_Casing.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    MOTOR CASING INFORMATION                   *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorCasingInfo(void)
{
// clear serial buffer
       while(Serial.available())
       {
        char getData = Serial.read();
       }
```

```
// ========================================================================
// Get motor casing dimensions - length
// Display instructions on Serial Monitor
   Serial.println(serialLine2);
   Serial.println(F("Motor Data Entry: Case Length"));
   Serial.println();
   Serial.println(F("Enter the length of the case (in millimeters)"));
   Serial.println(F("of the motor being tested."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get casing length
   resumeProcedure = false;
   sdInfo = "";
   while (resumeProcedure == false)
   {
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            sdInfo = ("Case Length: ") + sdInfo + (" mm");
            Serial.println(sdInfo);
            Serial.println();

           // Info Log for casing length
              writeInfoDataToCard();

            resumeProcedure = true;
          }
        }
      }
   }

// ========================================================================
// Get casing dimensions - diameter
// Display instructions on Serial Monitor
   Serial.println(serialLine2);
   Serial.println(F("Motor Data Entry: Case Diameter"));
   Serial.println();
   Serial.println(F("Enter the exterior diameter of the case (in millimeters)"));
   Serial.println(F("of the motor being tested."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get casing diameter
   resumeProcedure = false;
   sdInfo = "";
   while (resumeProcedure == false)
   {
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            Serial.print(F("Case Diameter: "));
            Serial.print(sdInfo);
            Serial.println(F(" mm"));
            Serial.println();

           // Info Log for Test Area Cleared
              sdInfo = ("Case Diameter: ") + sdInfo + (" mm");
              writeInfoDataToCard();

            resumeProcedure = true;
          }
        }
      }
```

```
    }

  // ================================================================
  // Get casing condition

  // Display instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Case Condition"));
     Serial.println();
     Serial.println(F("Describe the overall condition of the case of the motor being tested."));
     Serial.println(F("Include any apparent damage or swelling that is noted."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get casing condition
     resumeProcedure = false;
     sdInfo = "";
     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          sdInfo = Serial.readString();
          {
            if (sdInfo != "")
            {
              Serial.print(F("Case Condition: "));
              Serial.println(sdInfo);
              Serial.println();

             // Info Log for Case Condition
                sdInfo = ("Case Condition: ") + sdInfo;
                writeInfoDataToCard();

              resumeProcedure = true;
            }
          }
        }
     }

  // ================================================================
  // Get casing mass

  // Display instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Case Mass"));
     Serial.println();
     Serial.println(F("Enter the total mass of the motor (in grams) that is being tested."));
     Serial.println(F("(Only enter the mass of the motor. Do not include the igniter or any other
additional items.)"));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get casing mass
     resumeProcedure = false;
     sdInfo = "";
     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          sdInfo = Serial.readString();
          {
            if (sdInfo != "")
            {
              Serial.print(F("Case Mass: "));
              Serial.print(sdInfo);
              Serial.println(F(" grams"));
              Serial.println();

             // Info Log for Case Condition
                sdInfo = ("Case Mass: ") + sdInfo + (" grams");
                writeInfoDataToCard();
```

```
              resumeProcedure = true;
            }
          }
        }
    }
}
```

Motor_Prep_Delay_Time.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                MOTOR PREP DELAY TIME SEQUENCE                 *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepDelayTime(void)
{
  // ===========================================================================
  // Get motor delay time information

  // Display delay time entry instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Delay Time"));
     Serial.println();
     Serial.println(F("Enter the delay time of the motor being tested."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get delay time of the motor being tested
     motorDelayTime = 0;
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          motorDelayTime = Serial.parseInt();
          if (motorDelayTime != 0)
          {
          // Display delay time entered
             Serial.print(F("Delay time entered: "));
             Serial.println(motorDelayTime);
             Serial.println();

             sdDelay = motorDelayTime;
             resumeProcedure = true;
         }
       }
     }

  // Now that we have the full motor code we can write it to the Information text file
     sdInfo = sdImpulseTotal + sdThrustAvg + "-" + sdDelay;

  // Info Log for Test Area Cleared
     sdInfo = ("Motor Classification: ") + sdInfo;
     writeInfoDataToCard();
}
```

Motor_Prep_Ignition_Time.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *              MOTOR PREP IGNITION TIME SEQUENCE                *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepIgnitionTime(void)
{
  // ===========================================================================
  // Get motor ignition time information

  // Display delay time entry instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Test Stand Entry: Ignition Time"));
     Serial.println();
     Serial.println(F("Enter the length of time (in seconds) between 3 and 10 that power"));
     Serial.println(F("should be applied to the igniter of the motor being tested."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get length of time for igniter of the motor being tested
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          timeFireRelay = Serial.parseInt();
          if (timeFireRelay != 0)
          {
            if ((timeFireRelay < 3) || (timeFireRelay >10))
            timeFireRelay = 5;

          // Display average thrust entered
             Serial.print(F("Ignition time length entered: "));
             Serial.println(timeFireRelay);
             Serial.println();

             resumeProcedure = true;
          }
        }
      }
}
```

Motor_Prep_Info.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                MOTOR PREPARATION INFORMATION                  *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepInfo(void)
{
  // Constants for PREP for LED clock
    const uint8_t SEG_PREP[] =
    {
    SEG_A | SEG_B | SEG_E | SEG_F | SEG_G,                  // P
    SEG_A | SEG_B | SEG_C | SEG_E | SEG_F | SEG_G,          // R
```

```
  SEG_A | SEG_D | SEG_E | SEG_F | SEG_G,                  // E
  SEG_A | SEG_B | SEG_E | SEG_F | SEG_G                   // P
  };

// ========================================================================
// motor test preparation information

// show "PREP" on clock
   clockDisplay.setSegments(SEG_PREP);

// show test preparation in progress on LCD
   lcd.setCursor(0,0);
   lcd.print(F("Motor Preparatin"));
   lcd.setCursor(0,1);
   lcd.print(F("  In Progress   "));

// clear serial buffer
   while(Serial.available())
   {
    char getData = Serial.read();
   }

// ========================================================================
// Get location information

// Display instructions on Serial Monitor
   Serial.println();
   Serial.println(F("Motor Data Entry: Location Information"));
   Serial.println();
   Serial.println(F("Enter the location where the test is being conducted."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get location for the motor test
   resumeProcedure = false;

   while (resumeProcedure == false)
   {
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            Serial.print(F("Location entered: "));
            Serial.println(sdInfo);
            Serial.println();

           // Info Log for Test Area Cleared
              sdInfo = ("Test Location: ") + sdInfo;
              writeInfoDataToCard();

            resumeProcedure = true;
            sdInfo="";
          }
        }
      }
   }

// ========================================================================
// Get location elevation
   Serial.println(serialLine2);
   Serial.println(F("Motor Data Entry: Location Elevation"));
   Serial.println();
   Serial.println(F("Enter the elevation (in meters) above mean sea level"));
   Serial.println(F("where the test is being conducted."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get elevation for the motor being tested
   resumeProcedure = false;
```

```
    while (resumeProcedure == false)
    {
       if (Serial.available() > 0)
       {
         sdInfo = Serial.readString();
         {
           if (sdInfo != "")
           {
             Serial.print(F("Elevation entered: "));
             Serial.print(sdInfo);
             Serial.println(F(" meters"));
             Serial.println();

            // Info Log for Test Area Cleared
               sdInfo = ("Test Location Elevation: ") + sdInfo + (" meters");
               writeInfoDataToCard();

             resumeProcedure = true;
             sdInfo="";
           }
         }
       }
    }

// ====================================================================
  // Get manufacturer name

  // Display instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Manufacturer Name"));
     Serial.println();
     Serial.println(F("Enter the name of the manufacturer of the motor being tested."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get manufacturer name for the motor being tested
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          sdInfo = Serial.readString();
          {
            if (sdInfo != "")
            {
              Serial.print(F("Manufacturer name entered: "));
              Serial.println(sdInfo);
              Serial.println();

             // Info Log for Test Area Cleared
                sdInfo = ("Manufacturer Name: ") + sdInfo;
                writeInfoDataToCard();

              resumeProcedure = true;
              sdInfo="";
            }
          }
        }
     }
}
```

Motor_Prep_Propellant.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                MOTOR PREPARATION PROPELLANT                   *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorPrepPropellant(void)
{
  // clear serial buffer
       while(Serial.available())
       {
        char getData = Serial.read();
       }

  // ================================================================================
  // Get propellant type

  // Display instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Propellant Type"));
     Serial.println();
     Serial.println(F("Enter the type of propellant being used by the motor being tested."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get propellant mass
     resumeProcedure = false;
     sdInfo = "";
     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          sdInfo = Serial.readString();
          {
            if (sdInfo != "")
            {
              Serial.print(F("Propellant type entered: "));
              Serial.println(sdInfo);
              Serial.println();

             // Info Log for Test Area Cleared
                sdInfo = ("Propellant Type: ") + sdInfo;
                writeInfoDataToCard();

              resumeProcedure = true;
            }
          }
        }
     }

  // ================================================================
  // Get propellant mass

  // Display instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Propellant Mass"));
     Serial.println();
     Serial.println(F("Enter the mass of propellant being used by the motor being tested."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get propellant mass
     resumeProcedure = false;
     sdInfo = "";
     while (resumeProcedure == false)
     {
```

```
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            Serial.print(F("Propellant mass entered: "));
            Serial.print(sdInfo);
            Serial.println(F(" grams"));
            Serial.println();

           // Info Log for propellant mass
              sdInfo = ("Propellant Mass: ") + sdInfo + (" grams");
              writeInfoDataToCard();

            resumeProcedure = true;
          }
        }
      }
   }

// ================================================================
// Get motor date/lot code

// Display instructions on Serial Monitor
   Serial.println(serialLine2);
   Serial.println(F("Motor Data Entry: Motor Date or Lot Code"));
   Serial.println();
   Serial.println(F("Enter either the Date of Manufacture or the Lot Number"));
   Serial.println(F("of the motor being tested."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get lot code
   resumeProcedure = false;
   sdInfo = "";
   while (resumeProcedure == false)
   {
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            Serial.print(F("Date/Lot Code: "));
            Serial.println(sdInfo);
            Serial.println();

           // Info Log for lot code
              sdInfo = ("Date/Lot Code: ") + sdInfo;
              writeInfoDataToCard();

            resumeProcedure = true;
          }
        }
      }
   }

// ================================================================
// Get motor igniter

// Display instructions on Serial Monitor
   Serial.println(serialLine2);
   Serial.println(F("Motor Data Entry: Igniter"));
   Serial.println();
   Serial.println(F("Enter the type of igniter being used on the motor being tested."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get igniter info
   resumeProcedure = false;
```

```
    sdInfo = "";
    while (resumeProcedure == false)
    {
       if (Serial.available() > 0)
       {
         sdInfo = Serial.readString();
         {
           if (sdInfo != "")
           {
             Serial.print(F("Igniter: "));
             Serial.println(sdInfo);
             Serial.println();

            // Info Log for igniter type
               sdInfo = ("Igniter: ") + sdInfo;
               writeInfoDataToCard();

             resumeProcedure = true;
           }
         }
       }
```

Motor_Prep_Scale.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *               Motor Scale Preparation Sequence                *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorScalePreparation(void)
{
  // ===========================================================================
  // Final preparations for data collection

  // Show column headers
     resumeProcedure = false;

  // Clear Serial Buffer
     while(Serial.available() > 0)
     {
     byte dummyread = Serial.read();
     }

  // Get calibration data from EEPROM
     loadCell.start(serialDigitalOut, powerDownSerialClock);
     EEPROM.get(eepromAdressCalibrationValue, newCalibrationValue);
     loadCell.setCalFactor(newCalibrationValue);
}
```

Motor_Prep_Total_Impulse.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *             MOTOR PREP TOTAL IMPULSE SEQUENCE                 *
 *                                                               *
 *****************************************************************
 *****************************************************************/
```

```
void motorPrepTotalImpulse(void)
{
  // ========================================================================
  // Get motor total impulse information

  // Display Total Impulse entry instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Motor Data Entry: Total Impulse"));
     Serial.println();
     Serial.println(F("Enter the letter for the total impulse of the motor being tested."));
     Serial.println(F("  - For 1/4A motors, enter '4'."));
     Serial.println(F("  - For 1/2A motors, enter '2'."));
     Serial.println(F("  - For all other motors (A-G), enter the letter designation."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // Get total impulse of the motor being tested
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          {
            if (charInput == '4')
            {
              motorTotalImpulse = 0.625;
              sdImpulseTotal = "1/4A";
            }
            else if (charInput == '2')
            {
              motorTotalImpulse = 1.25;
              sdImpulseTotal = "1/2A";
            }
            else if (charInput == 'a' || charInput == 'A')
            {
              motorTotalImpulse = 2.5;
              sdImpulseTotal = "A";
            }
            else if (charInput == 'b' || charInput == 'B')
            {
              motorTotalImpulse = 5;
              sdImpulseTotal = "B";
            }
            else if (charInput == 'c' || charInput == 'C')
            {
              motorTotalImpulse = 10;
              sdImpulseTotal = "C";
            }
            else if (charInput == 'd' || charInput == 'D')
            {
              motorTotalImpulse = 20;
              sdImpulseTotal = "D";
            }
            else if (charInput == 'e' || charInput == 'E')
            {
              motorTotalImpulse = 40;
              sdImpulseTotal = "E";
            }
            else if (charInput == 'f' || charInput == 'F')
            {
              motorTotalImpulse = 80;
              sdImpulseTotal = "F";
            }
            else if (charInput == 'g' || charInput == 'G')
            {
              motorTotalImpulse = 160;
              sdImpulseTotal = "G";
            }
```

```
          }

    // Display total impulse entered
       Serial.print(F("Total Impulse entered: "));
       Serial.print(sdImpulseTotal);
       Serial.print(" - ");
       Serial.print(motorTotalImpulse, 3);
       Serial.println(F(" Newtons"));
       Serial.println();

       resumeProcedure = true;
       }
     }
}
```

Motor_Recalibration_Sequence.ino

```
*****************************************************************
 *****************************************************************
 *                                                               *
 *                  MOTOR RECALIBRATION SEQUENCE                 *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorRecalibrate(void)
{
  // ===========================================================================
  // Calibrate decision

  // Display Test Stand calibration entry instructions on Serial Monitor
     Serial.println(serialLine);
     Serial.println(F("Test Stand Calibration"));
     Serial.println();
     Serial.println(F("The test stand should be calibrated prior to testing."));
     Serial.println(F("If you have previously calibrated the test stand you"));
     Serial.println(F("can use that calibration if you have saved it."));
     Serial.println();
     Serial.println(F("If you did not save a previous calibration, if a "));
     Serial.println(F("calibration has never been performed, or if you wish"));
     Serial.println(F("to perform a new calibration, hit 'c' followed by the"));
     Serial.println(F("'ENTER' key."));
     Serial.println();
     Serial.println(F("If you wish to bypass the calibration process hit 'b'"));
     Serial.println(F("followed by the 'ENTER' key."));
     Serial.println();

  // Get calibration decision
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          {
            if (charInput == 'b' || charInput == 'B')
            {
              Serial.println(F("Calibration Bypassed"));
              dateTimeSerialMonitor();
              Serial.println();

              // System Log for Calibration Bypass
                 timeMillis = micros()/1000000.f;
                 sdTimeStamp = timeMillis;
                 sdRTCStamp = bufferTime;
```

```
                sdMessage = (F("Recalibrate Load Cell"));
                sdResult = (F("Bypassed"));
                writeSysDataToCard();

             resumeProcedure = true;
            }
            else if (charInput == 'c' || charInput == 'C')
            {

             // System Log for Recalibraion
                timeMillis = micros()/1000000.f;
                sdTimeStamp = timeMillis;
                sdRTCStamp = bufferTime;
                sdMessage = (F("Recalibrate Load Cell"));
                sdResult = (F("Yes"));
                writeSysDataToCard();

             // Go to calibraion function
                loadCellCalibrate();

             resumeProcedure = true;
            }
          }
        }
    }
}
```

Motor_Test_Clear_Area.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                   MOTOR TEST CLEAR AREA                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorClearArea(void)
{
  // Constants for TEST for LED clock
    const uint8_t SEG_TEST[] =
    {
    SEG_D | SEG_E | SEG_F | SEG_G,                  // t
    SEG_A | SEG_D | SEG_E | SEG_F | SEG_G,          // E
    SEG_A | SEG_C | SEG_D | SEG_F | SEG_G,          // S
    SEG_D | SEG_E | SEG_F | SEG_G                   // t
    };

  // ===================================================================
  // Clear Test Area Sequence

  // Turn test stand LEDs to Red
     rgbRed();
     rgbSteadyLamp();

  // show "TEST" on clock
     clockDisplay.setSegments(SEG_TEST);

  // Display Clear Area on LCD display
     lcd.setCursor(0,0);
     lcd.print(F("CLEAR TEST STAND"));
     lcd.setCursor(0,1);
     lcd.print(F("      AREA      "));

  // Display clear area instructions on Serial Monitor
     Serial.println(serialLine2);
```

```
    Serial.println(F("Clear Test Stand Checklist"));
    Serial.println();
    Serial.println(F("  - Make sure that everyone is a safe distance"));
    Serial.println(F("    from the Test Stand."));
    Serial.println(F("  - Nothing should be on the Test Stand."));
    Serial.println(F("  - Nothing should be under the load cell."));
    Serial.println(F("  - The ejection charge vent ports must be clear."));
    Serial.println();
    Serial.println(F("When these tasks have been completed and verified,"));
    Serial.println(F("hit 'v' followed by the 'ENTER' key."));
    Serial.println();

  // Get verificaton test area is clear
    resumeProcedure = false;

    while (resumeProcedure == false)
    {
       if (Serial.available() > 0)
       {
         charInput = Serial.read();
         {
           if (charInput == 'v' || charInput == 'V')
           {
             Serial.println(F("Clear Test Stand and Area checklist complete and verified"));
             dateTimeSerialMonitor();
             timeMillis = micros()/1000000.f;
             Serial.println();
             Serial.println();

            // System Log for Test Area Cleared
               sdRTCStamp = bufferTime;
               sdTimeStamp = timeMillis;
               sdMessage = (F("Test Stand Area Cleared"));
               sdResult = (F("Verified"));
               writeSysDataToCard();

             resumeProcedure = true;
           }
         }
       }
    }
}
```

Motor_Test_Tare.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                       MOTOR TEST TARE                         *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void motorTare(void)
{
  // ===========================================================================
  // Perform tare on test stand
     loadCell.update();
     loadCell.tareNoDelay();

     if (loadCell.getTareStatus() == true)
        {
          Serial.println(F("Tare complete"));
          Serial.println();

          resumeProcedure = true;
```

```
        }
}
```

Post_Test_Data_Entry.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                POST FIRE SHUTDOWN SAFETY PERIOD               *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void postTestDataEntry(void)
{
// clear serial buffer
       while(Serial.available())
       {
        char getData = Serial.read();
       }

  // ===============================================================================
  // Determine if CATO

  // Display CATO entry instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Post Test Data Entry: CATO"));
     Serial.println();
     Serial.println(F("Did the motor suffer a catostrophic failure?"));
     Serial.println(F("If YES hit 'y' followed by the 'ENTER' key."));
     Serial.println(F("If NO hit 'n' followed by the 'ENTER' key."));
     Serial.println();

  // Get CATO info
     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();
          {
            if (charInput == 'n' || charInput == 'N')
            {
             // Info Log for CATO Reporting
                sdInfo = ("No catostrophic failure reported");
                writeInfoDataToCard();

                Serial.println(F("No catostrophic failure reported"));
                Serial.println();

                resumeProcedure = true;
            }
            else
            {
             // System Log for Motor Loading
                sdInfo = ("Catostrophic failure of the motor occured during this test");
                writeInfoDataToCard();

                Serial.println(F("Catostrophic failure of the motor occured during this test"));
                Serial.println(F("Prepare a MESS Report through https://www.motorcato.org"));
                Serial.println();

                resumeProcedure = true;
             }
          }
```

```
      }
   }

// clear serial buffer
     while(Serial.available())
     {
      char getData = Serial.read();
     }

// ========================================================================
// Get empty casing mass info post test

// Display instructions on Serial Monitor
   Serial.println(serialLine2);
   Serial.println(F("Post Test Data Entry: Case Mass"));
   Serial.println();
   Serial.println(F("Enter the post test mass of the case of the motor that was tested."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get empty mass
   resumeProcedure = false;
   sdInfo = "";
   while (resumeProcedure == false)
   {
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            Serial.print(F("Post test case mass: "));
            Serial.println(sdInfo);
            Serial.println();
            //Serial.println();

           // Info Log for Test Area Cleared
              sdInfo = ("Post test case mass: ") + sdInfo;
              writeInfoDataToCard();

            resumeProcedure = true;
          }
        }
      }
   }

// ========================================================================
// Get casing condition

// Display instructions on Serial Monitor
   Serial.println(serialLine2);
   Serial.println(F("Post Test Data Entry: Comments"));
   Serial.println();
   Serial.println(F("Enter any comments about the test that were not covered"));
   Serial.println(F("in any of the data collection methods used in this test."));
   Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
   Serial.println();

// Get casing condition
   resumeProcedure = false;
   sdInfo = "";
   while (resumeProcedure == false)
   {
      if (Serial.available() > 0)
      {
        sdInfo = Serial.readString();
        {
          if (sdInfo != "")
          {
            Serial.print(F("Overall Test Comments: "));
            Serial.println(sdInfo);
```

```
              Serial.println();
              Serial.println();

             // Info Log for Case Condition
                sdInfo = ("Overall Test Comments: ") + sdInfo;
                writeInfoDataToCard();

              resumeProcedure = true;
            }
          }
        }
      }

  // ====================================================================
  // System Log for Test Complete
    // get date and time
     now = rtc.now();
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
     timeMillis = micros()/1000000.f;

     sdTimeStamp = timeMillis;
     sdRTCStamp = bufferTime;
     sdMessage = (F("Motor Test"));
     sdResult = (F("Concluded"));
     writeSysDataToCard();

  // Info Log for Test Concluded
     sdInfo = ("Testing session concluded at ") + sdRTCStamp;
     writeInfoDataToCard();

  // Info Log for testing complete
     Serial.println(serialLine);
     Serial.print(F("Testing session concluded at "));
     dateTimeSerialMonitor();
     Serial.println(serialLine);
}
```

RGB_LED_Lamp_Settings.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    RGB LED LAMP SETTINGS                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

// define variables
   int valueRed = 0;
   int valueGreen = 0;
   int valueBlue = 0;

// choose a value between 0 and 255 on each variable to change the color.
void rgbRed(void)
//Solid red lamp
{
  valueRed = 255;
  valueGreen = 0;
  valueBlue = 0;
}

void rgbGreen(void)
//Solid green lamp
{
  valueRed = 0;
  valueGreen = 255;
```

```
  valueBlue = 0;
}

void rgbBlue(void)
//Solid blue lamp
{
  valueRed = 0;
  valueGreen = 0;
  valueBlue = 255;
}

void rgbYellow(void)
//Solid yellow lamp
{
  valueRed = 255;
  valueGreen = 70;
  valueBlue = 0;
}

void rgbSteadyLamp(void)
{
  analogWrite(RED, valueRed);
  analogWrite(GREEN, valueGreen);
  analogWrite(BLUE, valueBlue);
}

void rgbFlashLamp(void)
{
  for (int x = 1; x < 50; x++)
  {
    analogWrite(RED, valueRed);
    analogWrite(GREEN, valueGreen);
    analogWrite(BLUE, valueBlue);

    delay(250);

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

    delay(250);
  }
}

void rgbStrobeLamp(void)
{
  for (int x = 1; x < 50; x++)
  {
    analogWrite(RED, valueRed);
    analogWrite(GREEN, valueGreen);
    analogWrite(BLUE, valueBlue);

    delay(15);

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

    delay(500);
  }
}
```

Sensor_Data_BME280.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                   Environmental Sensors                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void sensorEnviroValues(void)
{
  // ===================================================================
  // get environmental temperature, pressure and humdity just prior to ignition
     sdTemperature = (sensorEnvironment.readTemperature());
     sdPressure = (sensorEnvironment.readPressure()/100.0F);
     sdHumidity = (sensorEnvironment.readHumidity());
}
```

Serial_Monitor_Date_Time.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *            DATE AND TIME DISPLAYED ON SERIAL MONITOR          *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void dateTimeSerialMonitor(void)
{
  // ===================================================================
  // get date and time
     now = rtc.now();

     //Time
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
     Serial.print(bufferTime);
     Serial.print(F("hrs - "));

     //Date
     sprintf(bufferDate,"%02u/%02u/%4u ",now.month(),now.day(),now.year());
     Serial.println(bufferDate);
     sdDate = bufferDate;
}
```

Serial_Monitor_Splash_Screen.ino

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

void splashScreenSerialMonitor(void)
{
  // ===================================================================
  // Display Program Info on the Serial Monitor
```

```
     Serial.println(serialLine);
     Serial.println();
     Serial.println(F("Austin Aerospace Educational Network"));
     Serial.println(F("Ground Support Project"));
     Serial.println();
     Serial.println(F("Project: Model Rocket Motor Test Stand"));
     Serial.println(F("         Using the Arduino Mega2560 Microcontroller"));
     Serial.print(F("Version: "));
     Serial.print(prgMajor); Serial.print(F(".")); Serial.print(prgMinor); Serial.print(F("."));
Serial.println(prgPatch);
     Serial.println();
     Serial.println(F("https://rocketryjournal.wordpress.com"));
     Serial.println();
     Serial.println(serialLine);
}
```

Setup_BME280_Sensor.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                      SETUP BME280 SENSOR                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupBmeSensor(void)
{
  // ===========================================================================
  // Setup for BME280 Sensor
     bool result;

     Serial.println(F("Initializing BME280 Environmental Sensor. Standby..."));

     result = sensorEnvironment.begin();
     if (!result)
     {
        Serial.println(F("Environmental sensor initialization failed."));
        Serial.println(F("Please check sensor's address and/or connection."));
        Serial.println(serialLine);

        // DateTime of failure
           now = rtc.now();
           sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
           timeMillis = micros()/1000000.f;

        // show FAIL on LED
           clockDisplay.setSegments(SEG_FAIL);

        // show failure on LCD
           lcd.setCursor(0,0);
           lcd.print(F("Envir Sensr Fail"));
           lcd.setCursor(0,1);
           lcd.print(F("Check Connection"));

        // show steady red LED lamp
           rgbRed();
           rgbSteadyLamp();

        // System Log for BME 280 Sensor
           sdRTCStamp = bufferTime;
           sdTimeStamp = timeMillis;
           sdMessage = (F("Initializing BME280 environmental sensor"));
           sdResult = (F("Failed"));
           writeSysDataToCard();
```

```
        // pause forever
           while (1);
     }

     // card initialized properly and is ready to start writing data
        Serial.println(F("Environmental sensor initialization successful."));
        Serial.println();

     // DateTime of success
        now = rtc.now();
        sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
        timeMillis = micros()/1000000.f;

     // show success on LCD
        lcd.setCursor(0,0);
        lcd.print(F("Enviro Sensor     "));
        lcd.setCursor(0,1);
        lcd.print(F("Initialized       "));
        delay(2000);

    // System Log for BME 280 Sensor
       sdRTCStamp = bufferTime;
       sdTimeStamp = timeMillis;
       sdMessage = (F("Initializing BME280 environmental sensor"));
       sdResult = (F("Successful"));
       writeSysDataToCard();
}
```

Setup_Date_Time_Check.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    SETUP CHECK DATE AND TIME                  *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupDateTimeCheck(void)
{
  // ============================================================
  // This function displays the current date and time
  // If incorrect, the User can enter the correct date and/or time

  // get date and time
  // now = rtc.now();

  // show test preparation in progress on LCD
     lcd.setCursor(0,0);
     lcd.print(F("Date Time Check "));
     lcd.setCursor(0,1);
     lcd.print(F("  In Progress   "));

  // ============================================================
  // Show Date/Time information
  // Display instructions on Serial Monitor

  // get user input
     Serial.println();
     Serial.println(F("If this is correct, press Y for 'Yes'."));
     Serial.println(F("If this is incorrect, press N for 'No'."));
     Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
     Serial.println();

  // clear serial buffer
     while(Serial.available())
```

```
     {
      char charInput = Serial.read();
     }

  // Determine if user answered Yes or No
      resumeProcedure = false;

          while (resumeProcedure == false)
          {
              if (Serial.available() > 0)
              {
                charInput = Serial.read();
                {
                  if (charInput == 'n' || charInput == 'N')
                  {
                  // Display time is incorrect, prompt for correct data
                      Serial.println(serialLine);
                      Serial.println(F("Date and/or time is incorrect"));
                      Serial.println(F("You will be prompted to enter the correct date and
time"));
                      Serial.println();

                      // call date time entry
                      // add function to input date and time
                         setupDateTimeEntry();

                      resumeProcedure = true;
                  }
                  else if (charInput == 'y' || charInput == 'Y')
                  {
                  // Display time is accurate and continue
                      Serial.println(F("Date and time are accurate"));
                      Serial.println();
                      Serial.println(serialLine2);

                  // System Log
                     //DateTime
                       now = rtc.now();
                       sprintf(bufferTime,"%02u:%02u:%02u
",now.hour(),now.minute(),now.second());
                       timeMillis = micros()/1000000.f;

                      syncRTCStamp = bufferTime;
                      syncTimeStamp = timeMillis;
                      syncMessage = (F("Computer/Real Time Clock Synchronization"));
                      syncResult = (F("In Sync"));

                      resumeProcedure = true;

                    // display delay
                       delay(2000);
                  }
                }
              }
            }


}
```

Setup_Date_Time_Entry.ino

```
*****************************************************************
 *****************************************************************
 *                                                               *
 *                    SETUP CHECK DATE AND TIME                  *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupDateTimeEntry(void)
{
  // ===================================================================
  // This function allows the user to enter the current date and time

  // ===================================================================
  // date variables
     int entryMonth = "";
     int entryDay = "";
     int entryYear = "";
     int entryHour="";
     int entryMinute = "";
     //String entrySync = "";

  // ===================================================================
  // Enter new date

      // Display Month entry instructions on Serial Monitor
          Serial.println(serialLine2);
          Serial.println(F("Date Entry: Month"));
          Serial.println();
          Serial.println(F("Enter the number for the current month."));
          Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
          Serial.println();

          resumeProcedure = false;

          while (resumeProcedure == false)
          {
              if (Serial.available() > 0)
                {
                entryMonth = Serial.parseInt();
                if (entryMonth != 0)
                  {
                    // check to see month entry is between 1 and 12
                    if ((entryMonth >= 1) and (entryMonth <=12))
                      // Display month entered
                      Serial.print(F("Month entered: "));
                      Serial.println(entryMonth);
                      Serial.println();
                      resumeProcedure = true;
                  }
                }
          }

      // Display Day entry instructions on Serial Monitor
          Serial.println(serialLine2);
          Serial.println(F("Date Entry: Day"));
          Serial.println();
          Serial.println(F("Enter the number for the current day."));
          Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
          Serial.println();

          resumeProcedure = false;

          while (resumeProcedure == false)
          {
              if (Serial.available() > 0)
                {
                entryDay = Serial.parseInt();
```

```
            if (entryDay != 0)
              {
                // check to see day entry is between 1 and 31
                if ((entryDay >= 1) and (entryDay <=31))
                  // Display day entered
                  Serial.print(F("Day entered: "));
                  Serial.println(entryDay);
                  Serial.println();
                  resumeProcedure = true;
              }
            }
      }

    // Display Year entry instructions on Serial Monitor
        Serial.println(serialLine2);
        Serial.println(F("Date Entry: Year"));
        Serial.println();
        Serial.println(F("Enter the number for the current day."));
        Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
        Serial.println();

      resumeProcedure = false;

      while (resumeProcedure == false)
      {
          if (Serial.available() > 0)
            {
            entryYear = Serial.parseInt();
            if (entryYear != 0)
              {
                // check to see year entry is 2024 or greater
                if (entryYear >= 2024)
                  // Display year entered
                  Serial.print(F("Year entered: "));
                  Serial.println(entryYear);
                  Serial.println();
                  resumeProcedure = true;
              }
            }
      }

// ================================================================
// Enter new time

  // Display hour entry instructions on Serial Monitor
      Serial.println(serialLine2);
      Serial.println(F("Time Entry: Hour"));
      Serial.println();
      Serial.println(F("Enter the number for the current hour in 24-hour time."));
      Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
      Serial.println();

      resumeProcedure = false;

      while (resumeProcedure == false)
      {
          if (Serial.available() > 0)
            {
            entryHour = Serial.parseInt();
            if (entryHour != 0)
              {
                // check to see hour entry is between 0 and 23
                if ((entryHour >= 0) and (entryHour <=23))
                  // Display month entered
                  Serial.print(F("Hour entered: "));
                  Serial.println(entryHour);
                  Serial.println();
                  resumeProcedure = true;
              }
            }
      }
```

```
     // Display minute entry instructions on Serial Monitor
         Serial.println(serialLine2);
         Serial.println(F("Time Entry: Minute"));
         Serial.println();
         Serial.println(F("Enter the number for the current minute."));
         Serial.println(F("Then hit the 'ENTER' key from the Serial Monitor"));
         Serial.println();

         resumeProcedure = false;

         while (resumeProcedure == false)
         {
             if (Serial.available() > 0)
               {
               entryMinute = Serial.parseInt();
               if (entryMinute != -1)
                 {
                   // check to see minute entry is between 0 and 59
                   if ((entryMinute >= 0) and (entryMinute <=59))
                     // Display minute entered
                     Serial.print(F("Minute entered: "));
                     Serial.println(entryMinute);
                     Serial.println();
                     resumeProcedure = true;
                 }
               }
         }

  // ========================================================================
  // Display sync instructions on Serial Monitor
     Serial.println(serialLine2);
     Serial.println(F("Time Entry: Sync to Computer"));
     Serial.println();
     Serial.println(F("Enter 's' the Serial Monitor then hit the'ENTER' key"));
     Serial.println(F("When the entered minute and computer minute align."));
     Serial.println();

     resumeProcedure = false;

     while (resumeProcedure == false)
     {
        if (Serial.available() > 0)
        {
          charInput = Serial.read();

           if (charInput == 's' || charInput == 'S')
              {
                // assign new date and time to RTC
                   rtc.adjust(DateTime(entryYear, entryMonth, entryDay, entryHour, entryMinute,
0));

                // System Log
                //DateTime
                   now = rtc.now();
                   sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
                   timeMillis = micros()/1000000.f;

                   syncRTCStamp = bufferTime;
                   syncTimeStamp = timeMillis;
                   syncMessage = (F("Computer/Real Time Clock Update"));
                   syncResult = (F("Clocks now synchronized"));

                // display sync completed
                   Serial.println(serialLine);
                   Serial.println(F("Sync Completed"));
                   Serial.println(serialLine);
                   resumeProcedure = true;

                // display delay
                   delay(2000);
```

```
            }
        }
    }

}
```

Setup_Fire_Control_System.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                  SETUP FIRE CONTROL SYSTEM                    *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupFireControl(void)
{
  // ===========================================================================
  // Setup Fire Control System
  // Initialize Fire Control Button
     Serial.println(F("Initializing Fire Control System. Standby..."));

  // Setup Start pin for interrupt
     pinMode(pinStartButton, INPUT_PULLUP);

  // Setup Fire pin for interrupt
     pinMode(pinFireButton, INPUT_PULLUP);

  // Setup fire relay pin
     pinMode(pinFireRelay, OUTPUT);

  // Fire Control System now active
     Serial.println(F("Fire Control System initialization successful."));
     Serial.println();

  // DateTime of success
     now = rtc.now();
     sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
     timeMillis = micros()/1000000.f;

  // Show success on LCD screen
     lcd.setCursor(0,0);
     lcd.print(F("Fire Ctrl System"));
     lcd.setCursor(0,1);
     lcd.print(F("Initialized     "));

  // System Log for Fire Control System
     sdRTCStamp = bufferTime;
     sdTimeStamp = timeMillis;
     sdMessage = (F("Initializing Fire Control System"));
     sdResult = (F("Successful"));
     writeSysDataToCard();

     delay(2000);
}
```

Setup_HX711.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                      SETUP HX711 LOAD CELL                    *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupHX711(void)
{
  // ===================================================================
  // Setup for HX711
     Serial.println(F("Initializing Load Cell and HX711. Standby..."));

  // set up Calibration buttons
     pinMode(pinCalibrate, INPUT_PULLUP);

  // start load cell and HX711 test
     loadCell.begin();

     timeStabilizing = 2000;               // precision right after power-up can be improved
                                           // by adding a few seconds of stabilizing time
     performTare = true;                   // set this to false if you don't want tare to
                                           // be performed in the next step

     loadCell.start(timeStabilizing, performTare);

     if (loadCell.getTareTimeoutFlag() || loadCell.getSignalTimeoutFlag())  // added signal time
out again
     {
        Serial.println(F("Load Cell/HX711 initialization failed."));
        Serial.println(F("Please check sensor's connections and pin designations."));
        Serial.println(serialLine);

        //DateTime of failure
           now = rtc.now();
           sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
           timeMillis = micros()/1000000.f;

        // show FAIL on LED
           clockDisplay.setSegments(SEG_FAIL);

        // show failure on LCD
           lcd.setCursor(0,0);
           lcd.print(F("Load Sensor Fail"));
           lcd.setCursor(0,1);
           lcd.print(F("Check Connection"));

        // show steady red LED lamp
           rgbRed();
           rgbSteadyLamp();

        // System Log for HX711 Sensor
           sdRTCStamp = bufferTime;
           sdTimeStamp = timeMillis;
           sdMessage = (F("Initializing Load Cell/HX711 sensor"));
           sdResult = (F("Failed"));
           writeSysDataToCard();

        // pause forever
           while (1);
     }

     // card initialized properly and is ready to start writing data
        Serial.println(F("Load Sensor/HX711 initialization successful."));
        Serial.println();

     // time of success
```

```
        now = rtc.now();
        sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
        timeMillis = micros()/1000000.f;

     // show success on LCD
        lcd.setCursor(0,0);
        lcd.print(F("Load Sensor     "));
        lcd.setCursor(0,1);
        lcd.print(F("Initialized     "));

     // System Log for HX711 Sensor
        sdRTCStamp = bufferTime;
        sdTimeStamp = timeMillis;
        sdMessage = (F("Initializing Load Cell/HX711 sensor"));
        sdResult = (F("Successful"));
        writeSysDataToCard();

        delay(2000);
}
```

Setup_LCD_I2C.ino

```
/*********************************************************************
 *********************************************************************
 *                                                                   *
 *                      SETUP 16x2 LCD I2C                           *
 *                                                                   *
 *********************************************************************
 *********************************************************************/

void setupLcdDisplay(void)
{
  // ====================================================================
  // Setup for LCD display
     lcd.init(); //initialize the lcd
     lcd.backlight(); //open the backlight

  // Display splash screen on LCD screen
     lcd.setCursor(0,0);
     lcd.print(F("Austin Aerospace"));
     lcd.setCursor(0,1);
     lcd.print(F("    Network     "));

     delay(2500);

  // Display splash screen on LCD screen
     lcd.setCursor(0,0);
     lcd.print(F("  Begin System  "));
     lcd.setCursor(0,1);
     lcd.print(F(" Initialization "));

  // display same on Serial monitor
     Serial.println(F("Begin Test Stand System and Sensor Initilization Process"));

     delay(2500);
}
```

Setup_LED_Display.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                     SETUP LED DISPLAY                         *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupLedDisplay(void)
{
  // Constants for BOOT for LED clock
    const uint8_t SEG_BOOT[] =
    {
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G,  // B
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,          // O
    SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F,          // O
    SEG_D | SEG_E | SEG_F | SEG_G                           // t
    };

  // ====================================================================
  // Setup for 4-Digit LED display
  // Set the display brightness (0-7):
     clockDisplay.setBrightness(5);

  // Clear the display:
     clockDisplay.clear();

  // show BOOT on LED
     clockDisplay.setSegments(SEG_BOOT);
}
```

Setup_MicroSD_Card.ino

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

void setupMicroSDCard(void)
{
    // ================================================================
    // variable declaration
       const int chipSelect = 53;  //pin number for CS on Mega2560 board (D53)

    // ================================================================
    // initialize Micro SD card
       Serial.println();
       Serial.println(F("Initializing Micro SD card. Standby..."));

    // see if the card is present and can be initialized:
       if (!SD.begin(chipSelect))
       {
          Serial.println(F("Card initialization failed, or not present. Replace card and reset
system."));
          Serial.println(serialLine);

          // show FAIL on LED
             clockDisplay.setSegments(SEG_FAIL);

          // show failure on LCD
```

```
            lcd.setCursor(0,0);
            lcd.print(F("  SD Card Fail  "));
            lcd.setCursor(0,1);
            lcd.print(F("Check Card-Reset"));

         // sound warning tone and flash rgb red
         //   rgbRedWarningBuzzerFlash();

         // Show steady redlamp
            rgbRed();
            rgbSteadyLamp();

         // don't do anything more:
            while (1);
      }

   // card initialized properly and is ready to start writing data
      Serial.println(F("Micro SD card initialization successful."));
      Serial.println();

   // show success on LCD
      lcd.setCursor(0,0);
      lcd.print(F("MicroSD Card              "));
      lcd.setCursor(0,1);
      lcd.print(F("Initialized               "));
      delay(2000);

// ================================================================
// Setup Motor Log file name

   // create new file name - from altduino.ino
   // create a new file
  for (uint8_t i = 0; i < 100; i++)
  {
    fileMotorLog[6] = i/10 + '0';
    fileMotorLog[7] = i%10 + '0';

    if (! SD.exists(fileMotorLog))
    {
      break;  // leave the loop!
    }
  }

// write motor headers to CSV file
   File fileMotorData = SD.open(fileMotorLog, FILE_WRITE);

// Data to write to SD card
   dataString = "";
   dataString = "Time Stamp,Load Cell Reading (n),Total Impulse (n)";
   fileMotorData.println(dataString);
   fileMotorData.close();

// ================================================================
// Setup System Log file name
// Use same log number as motor file
      fileSystemLog[6] = fileMotorLog[6];
      fileSystemLog[7] = fileMotorLog[7];


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

// Log header to write to SD card
   dataString = "";
   dataString = "RTC Stamp,Microsecond Time Stamp,Message,Result";
   fileSystemData.println(dataString);
   // fileSystemData.close();

   dataString = "-,0,Test Stand Software Boot,Initialization";
   fileSystemData.println(dataString);
   fileSystemData.close();
```

```
  // ================================================================
  // Setup Information Log file name
  // Use same log number as motor file
        fileInfoLog[6] = fileMotorLog[6];
        fileInfoLog[7] = fileMotorLog[7];

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

  // Log header to write to SD card
     fileInfoData.println(F("Austin Aerospace Educational Network"));
     fileInfoData.println(F("Arduino Ground Support Project"));
     fileInfoData.println();
     fileInfoData.println(F("Project: Model Rocket Motor Test Stand"));
     fileInfoData.print(F("Version: "));
     fileInfoData.print(prgMajor); fileInfoData.print(F(".")); fileInfoData.print(prgMinor);
fileInfoData.print(F(".")); fileInfoData.println(prgPatch);
     fileInfoData.println();
     fileInfoData.println(F("https://rocketryjournal.wordpress.com"));
     fileInfoData.println();
     fileInfoData.println(serialLine);

     // get current date and time
        now = rtc.now();
        sprintf(bufferDate,"%02u/%02u/%4u ",now.month(),now.day(),now.year());
        sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());

     fileInfoData.print(F("Motor Test Report for "));
     fileInfoData.print(bufferDate);
     fileInfoData.print(F("beginning at "));
     fileInfoData.print(bufferTime);
     fileInfoData.println(F("hrs."));

     fileInfoData.println(serialLine);
     fileInfoData.close();
     writeInfoDataToCard();

  // ================================================================
  // System log for RTC
     sdRTCStamp = rtcRTCStamp;
     sdTimeStamp = rtcTimeStamp;
     sdMessage = rtcMessage;
     sdResult = rtcResult;
     writeSysDataToCard();

  // System log for Time Sync
     sdTimeStamp= "";
     sdRTCStamp = syncRTCStamp;
     sdTimeStamp = syncTimeStamp;
     sdMessage = syncMessage;
     sdResult = syncResult;
     writeSysDataToCard();

 // System Log for MicroSD Card
     timeMillis = micros()/1000000.f;
     sdRTCStamp = bufferTime;
     sdTimeStamp = timeMillis;
     sdMessage = (F("Initializing MicroSD Card"));
     sdResult = (F("Successful"));
     writeSysDataToCard();
}
```

Setup_Real_Time_Clock.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                    SETUP REAL TIME CLOCK                      *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void setupRTC(void)
{
  // ============================================================
  // Setup for RTC
     Serial.println(serialLine);
     Serial.println();
     Serial.println(F("Initializing Real Time Clock (RTC) Module. Standby..."));

  // Check if RTC is connected correctly
     if (! rtc.begin())
     {
      // show error in Serial Monitor
         Serial.println(F("Unable to find Real Time Clock"));

      // show error on LCD
         lcd.setCursor(0,0);
         lcd.print(F("Unable to find  "));
         lcd.setCursor(0,1);
         lcd.print(F("Real Time Clock "));

      // show FAIL on LED
         clockDisplay.setSegments(SEG_FAIL);

      // show steady red LED lamp
         rgbRed();
         rgbSteadyLamp();

      // pause program
         while (1);
     }
     else
     {
      //show success in Serial Monitor
        Serial.println(F("Real Time Clock initialized"));

      // show success on LCD
         lcd.setCursor(0,0);
         lcd.print(F("Real Time Clock "));
         lcd.setCursor(0,1);
         lcd.print(F("Initialized     "));
         delay(2000);
     }

  // Check if the RTC lost power and if so, set the time
     if (rtc.lostPower())
     {
      Serial.println(F("It appears the Real Time Clock lost power."));
      Serial.println(F("Adjusting date and time to equal sketch compile"));
      Serial.println();

      // show power loss on LCD
         lcd.setCursor(0,0);
         lcd.print(F(" RTC Power Loss "));
         lcd.setCursor(0,1);
         lcd.print(F("  Adjusting...  "));
         delay(2000);

      // The following line sets the RTC to the date & time this sketch
      // was compiled
      rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
```

```
    }
    else
    {
     Serial.println(F("No indication of RTC power loss"));
     Serial.println();
    }

  // Show Date and Time
    Serial.print(F("The current time and date is "));
    dateTimeSerialMonitor();
    timeMillis = micros()/1000000.f;

  // System Log
    now = rtc.now();
    sprintf(bufferTime,"%02u:%02u:%02u ",now.hour(),now.minute(),now.second());
    timeMillis = micros()/1000000.f;

    rtcRTCStamp = bufferTime;
    rtcTimeStamp = timeMillis;
    rtcMessage = (F("Initializing Real Time Clock"));
    rtcResult = (F("Successful"));

    delay(1000);
}
```

Strobe_LED_Bulb.ino

```
/*****************************************************************
 *****************************************************************
 *                                                               *
 *                   STROBE WHITE LED BULB                       *
 *                                                               *
 *****************************************************************
 *****************************************************************/

void ledWhiteStrobeLamp(void)
{
  digitalWrite(ledStrobe, HIGH);  // turn the LED on (HIGH is the voltage level)
  delay(15);

  digitalWrite(ledStrobe, LOW);  // turn the LED off by making the voltage LOW
  delay(500);

}
```

Write_Info_Data_To_SD_Card.ino

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

 void writeInfoDataToCard(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.

  // TXT stands for "Text". It is a plain text file format. It can be
  // read by any text or word processor program.

     File fileInfoData = SD.open(fileInfoLog, FILE_WRITE);

  // if the file is available, write to it:
     if (fileInfoData)
     {
      dataString = sdInfo;
      fileInfoData.println(dataString);
      fileInfoData.close();
     }
     else // if the file isn't open, pop up an error:
     {
      File fileSystemData = SD.open(fileSystemLog, FILE_WRITE);

      sdTimeStamp = String(timeMillis,10);

      // write error to sd card if possible
         sdMessage = "Info Data Log";
         sdResult = "Card Write Error";
         dataString = sdRTCStamp + "," + sdTimeStamp + "," + sdMessage + "," + sdResult;
         fileSystemData.println(dataString);
         fileSystemData.close();

      // Show error on Serial Monitor
         Serial.println(F("Error opening Info data log..."));

      // show FAIL on LED
         clockDisplay.setSegments(SEG_FAIL);

      // show error on LCD
         lcd.setCursor(0,0);
         lcd.print(F("Card Write Error"));
         lcd.setCursor(0,1);
         lcd.print(F("Opening Info Log"));

     }

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

Write_Motor_Data_To_SD_Card.ino

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

 void writeMotorDataToSDCard(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.
```

```
    File fileMotorData = SD.open(fileMotorLog, FILE_WRITE);

 // if the file is available, write to it:
    if (fileMotorData)
    {
       // create coma separated data string from sensor readings
       sdTimeStamp = String(timeMillis,10);
       sdImpulse = String(scaleNewtons,10);
       sdTotalImpulse = String(impulseTotal,10);

       dataString = sdTimeStamp + "," + sdImpulse + "," + sdTotalImpulse;
       fileMotorData.println(dataString);
       fileMotorData.close();
    }
    else // if the file isn't open, pop up an error:
    {
     File fileSystemData = SD.open(fileSystemLog, FILE_WRITE);

     sdTimeStamp = String(timeMillis,10);

     // write error to sd card if possible
        sdMessage = "Motor Data Log";
        sdResult = "Card Write Error";
        dataString = sdRTCStamp + "," + sdTimeStamp + "," + sdMessage + "," + sdResult;
        fileSystemData.println(dataString);
        fileSystemData.close();

     // Show error on Serial Monitor
        Serial.println(F("Error opening Motor data log..."));

     // show FAIL on LED
        clockDisplay.setSegments(SEG_FAIL);

     // show error on LCD
        lcd.setCursor(0,0);
        lcd.print(F("Card Write Error"));
        lcd.setCursor(0,1);
        lcd.print(F(" Motor Data Log "));

    }

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

Write_Sys_Data_To_SD_Card.ino

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

 void writeSysDataToCard(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.
```

```
    File fileSystemData = SD.open(fileSystemLog, FILE_WRITE);

 // if the file is available, write to it:
    if (fileSystemData)
    {
     dataString = sdRTCStamp + "," + sdTimeStamp + "," + sdMessage + "," + sdResult;
     fileSystemData.println(dataString);
     fileSystemData.close();
    }
    else // if the file isn't open, pop up an error:
    {
     sdTimeStamp = String(timeMillis,10);

     // write error to sd card if possible
        sdMessage = "SYS Data Log";
        sdResult = "Card Write Error";
        dataString = sdRTCStamp + "," + sdTimeStamp + "," + sdMessage + "," + sdResult;
        fileSystemData.println(dataString);
        fileSystemData.close();

     // Show error on Serial Monitor
        Serial.println(F("Error opening SYS data log..."));

     // show FAIL on LED
        clockDisplay.setSegments(SEG_FAIL);

     // show error on LCD
        lcd.setCursor(0,0);
        lcd.print(F("Card Write Error"));
        lcd.setCursor(0,1);
        lcd.print(F("  SYS Data Log  "));
    }
 delay(1);  // Pause for 1 milliseconds.
}
```

Compilation Notes - Arduino Mega2560

```
Sketch uses 67988 bytes (26%) of program storage space. Maximum is 253952 bytes.
Global variables use 2477 bytes (30%) of dynamic memory, leaving 5715 bytes for local variables.
Maximum is 8192 bytes.
```

A4
Parts Listing

Electronics

Test Stand Platform Electronics Housing			
Item	**Vendor**	**URL**	**Price**
ELEGOO MEGA R3 Board ATmega 2560	Amazon	https://www.amazon.com/dp/B01H4ZLZLQ	$20.99
Digital Load Cell Weight Sensor	Amazon	https://www.amazon.com/gp/product/B08KRWY43Y	$7.59
SparkFun Load Cell Amplifier - HX711	Amazon	https://www.amazon.com/gp/product/B079LVMC6X	$10.95
HiLetgo Micro SD TF Card Adater Reader Module 6Pin SPI Interface	Amazon	https://www.amazon.com/gp/product/B07BJ2P6X6	5 for $6.99
BME280 Environmental Sensor	Amazon	https://www.amazon.com/gp/product/B07P4CWGGK	$15.85
DS3231 AT24C32 IIC Real Time Clock	Amazon	https://www.amazon.com/gp/product/B07Q7NZTQS	2 for $5.99
5V 1 Channel Relay Module	Amazon	https://www.amazon.com/gp/product/B07874KSLY	5 for $7.99
Micro SD TF Card Adater Reader	Amazon	https://www.amazon.com/gp/product/B07BJ2P6X6	5 for $6.99
Mini Active Piezo Buzzers	Amazon	https://www.amazon.com/gp/product/B07VK1GJ9X	10 for $6.99
10mm RGB Multicolor LED Diode Lights	Amazon	https://www.amazon.com/gp/product/B01CI6EWHK	50 for $9.99
5mm White (4 needed) and red (1 needed) LED *	Amazon	https://www.amazon.com/ELEGOO-Diffused-Assorted-Colors-Arduino/dp/B0739RYXVC	100 for $11.99
220Ω Resistors (Need 16) *	Amazon	https://www.amazon.com/EDGELEC-Resistor-Tolerance-Multiple-Resistance/dp/B07QK9ZBVZ	100 for $5.99
1KΩ Resistor (Need 1) *	Amazon	https://www.amazon.com/EDGELEC-Resistor-Tolerance-Resistance-Optional/dp/B07HDDWFDD	100 for $5.49
Banana Clips (2 pair)	Amazon	https://www.amazon.com/gp/product/B07KG11GL3/	20 pair for $12.89

Item	Vendor	URL	Price
Alligator Clips	Harbor Freight	https://www.harborfreight.com/18-inch-low-voltage-multi-colored-test-leads-66717.html	5 pait for $3.59
4 AA Battery Holder with Leads	Amazon	https://www.amazon.com/LAMPVPATH-Battery-Holder-Leads-Wires/dp/B07T7MTRZX	2 for $5.98
Protective Wire Wrap	Harbor Freight	https://www.harborfreight.com/electrical/electrician-s-tools/wire-running/wire-wrap/3-8-eighth-inch-x-10-ft-protective-wire-wrap-66987.html	10-feet for $3.49
DB15 RS232 Serial D-SUB Solder Cup Connectors	Amazon	https://www.amazon.com/gp/product/B09VG9CNS7	5 pair for $9.99
DB15 Extension Cable Double Shielded, DB15 Male to Male Cable 10FT	Amazon	https://www.amazon.com/gp/product/B093P7T2J7	$14.99
USB-A to USB-B 2.0 Cable, 10 foot	Amazon	https://www.amazon.com/dp/B00NH13DV2	$9.89

Test Stand Remote Head			
Item	**Vendor**	**URL**	**Price**
TM1637 4-Digit 7-Segment LED Display	Amazon	https://www.amazon.com/gp/product/B01DKISMXK	2 for $6.99
I2C 1602 LCD Display Module	Amazon	https://www.amazon.com/gp/product/B07S7PJYM6	2 for $9.99
Tactile switches with caps (3 needed) *	Amazon	https://www.amazon.com/Gikfun-12x12x7-3-Tactile-Momentary-Arduino/dp/B01E38OS7K	25 pieces for $8.68
SPST Momentary Push Button Switch	Amazon	https://www.amazon.com/gp/product/B08JHSG717	6 for $8.99
PCB Prototype Board *	Amazon	https://www.amazon.com/ELEGOO-Prototype-Soldering-Compatible-Arduino/dp/B072Z7Y19F	32 pieces for $9.99

* Can obtain all of these components in a single set. Set contains PCB Boards, Header Connectors, 600 Assorted Resistors (from 10Ω to 1MΩ), Screw Terminal Block, 50 LED diodes in 5 colors & 12 Tactile Cap Switch with multi-color covers - $16.99 at Amazon https://www.amazon.com/gp/product/B07QC5X21L - $16.99

A5
References

- "3D Printed Rocket Test Stand" by -Zander. Instructables, https://www.instructables.com/3D-Printed-Rocket-Test-Stand
- "Arduino - LCD I2C" Arduino Getting Started, https://arduinogetstarted.com/tutorials/arduino-lcd-i2c
- "Arduino with Load Cell and HX711 Amplifier (Digital Scale)" Random Nerd Tutorials, https://randomnerdtutorials.com/arduino-load-cell-hx711/
- "Conducting a Test" Glenn Research Center, https://www1.grc.nasa.gov/historic-facilities/rocket-engine-test-facility/conducting-a-test/
- “Datalogger” (From within the Arduino IDE) File > Examples > SD > Datalogger
- "Design and Characterization of a Lab-Scale Hybrid Rocket Test Stand" by James C. Thomas, Jacob M. Stahl, Gordon R. Morrow, Eric L. Petersen, Texas A&M University, College Station, Texas. July 2016. American Institute of Aeronautics and Astronautics.
- "Effect of Altitude on Rocket Engine Performance" by Ellis Langford. R&D Report for NARAM 42.
- "Effect of Extreme Cold on Model Rocket Motors" by Ric Gaff. August 16, 1984. R&D Report for NARAM-26
- "Effect of Humidity on Model Rocket Motors" by Caroline Steele. August 2004. R&D Report for NARAM-46
- "Getting Started with Load Cells" by Sarah Al-Mutlaq. Sparkfun, https://learn.sparkfun.com/tutorials/getting-started-with-load-cells
- “Hobby Rocket Motor Data”. ThrustCurve. https://www.thrustcurve.org/info/motorstats.html
- "How Do You Measure the Thrust of a Rocket Engine?" April 18, 2022. National Institute of Standards and Technology (NIST),https://www.nist.gov/how-do-you-measure-it/how-do-you-measure-thrust-rocket-engine
- “Model Rocket Engines” Estes Tech Note 1. 1972

- “Model Rocket Engines” by William Simon (Revised by Thomas Beach and Joyce Guzik). Estes Model Rocketry Tech Manual, pg 14. 1993
- “Model Rocket Engine Performance.” Estes Tech Note 2 by Edwin D. Brown.
- "Model Rocket Motor Dynamometer (Arduino Uno)" by nightmare.on.scam.street. Instructables, https://www.instructables.com/Model-Rocket-Motor-Dynamometer-Arduino-Uno/
- "Model Rocket Motor Test Stand" by NM Rocketry. September 6, 2020. Arduino Project Hub, https://projecthub.arduino.cc/nmrsthrust/model-rocket-motor-test-stand-f8a42f
- NAR Standards and Testing Committee Motor Testing Manual Version 1.5. July 1, 2011.
- "Rocket Motor Static Testing" by Richard Nakka. May 14, 2020.Richard Nakka's Experimental Rocketry Web Site, https://www.nakka-rocketry.net/static.html
- "Standards & Testing" by the National Association of Rocketry Standards and Testing Committee. Sport Rocketry Magazine, November/December 2010, Pg 42-47.
- "Static Rocket Fire Test rig" byzuegnull. September 25, 2021. Thingiverse, https://www.thingiverse.com/thing:4974745
- "Stennis Space Center: NASA's Largest Rocket Testing Site" by Nola Taylor Tillman. January 25, 2018. Space.com, https://www.space.com/39498-stennis-space-center.html
- "Test Your Engine" Lesson Plans. Estes Industries, https://edu.estesrockets.com/products/test-your-engines-lesson-plan
- "Using a Model Rocket-Engine Test Stand in a Calculus Course" - https://pubs.nctm.org/view/journals/mt/95/7/article-p516.xml
- “What’s in a Rocket Motor?” NAR Member Guidebook, 2022 Volume 14. pg 12.

A6
Project Links

Tinkercad Drawings: There are a total of six Tinkercad drawings available for this project. They are:

- Overall Design - https://www.tinkercad.com/things/5PkXrbvQFgC-aaen-rocket-motor-test-stand
- Electronics Housing - https://www.tinkercad.com/things/jjsRjP7aWB7-test-stand-individual-and-assembled-electronic-component-mounts
- Remote Head - https://www.tinkercad.com/things/gyea9P4lKON-test-stand-remote-head
- Motor Mounts - https://www.tinkercad.com/things/iWNT7Z2hdte-test-stand-motor-mount-components
- Small Parts - https://www.tinkercad.com/things/4o926bDtEKG-test-stand-small-parts
- Design Elements - https://www.tinkercad.com/things/2oOwyYvD2nV-test-stand-design-elements

SourceForge: To download the Arduino code as a single zip file, visit our SourceForge project page at https://sourceforge.net/p/project-vulcan/

Thingiverse: You can download all of the STL files for this project from our Thingiverse page at https://www.thingiverse.com/thing:6792050