Hardware

The hardware has 2 main sections: electronics and housing. The electronics includes the various buttons and switches, the PCB to connect them to the Arduino, and the Arduino itself. The housing includes 2 laser-cut stainless-steel top panels and a 3D printed box.

**Components:**

Here is a list of all the components on the top panel:

* 2 joysticks to control the 3 axes of rotation and throttle
* The 2 buttons on top of the joysticks are to set throttle to 0% or 100%
* Switch requiring key for arming stage button
* Big green stage button
* Big red abort button with a cover to prevent accidental activation
* 10 pushbuttons, 1 for each action group
* 5 more push buttons for quick load, quick save, time warp control and map
* 4 toggle switches for SAS, RCS, gears and lights.
* A toggle switch to switch between plane and rocket mode, where joystick controls are slightly different
* A push button to reset controls whenever the craft is reverted to launch. This syncs up the controller with the situation in-game.
* All buttons and switches have an LED to indicate their state
* The stage section has an RGB led: green to show armed, red to show staging.

Other than these, there are also a few components on the PCB. An Arduino pro micro is used to control the entire controller. We decided to use the pro micro because it is the smallest controller with a MEGA32U4 chip on it, which is required for the Keyboard library to work. A few more logic ICs and passive components are also included to connect the components together.

**PCB:**

All the components are connected directly downwards into a PCB. A technique called wire wrapping is used. This is because wire wrapping is stronger than a breadboard, but also easier and quicker than directly soldering wires. The PCB includes special headers with longer pins designed for wire wrapping.

All the components in the top section are either buttons or switches, they all have the same components connected to them on a PCB. One of their pins is connected to 5V, while the other one is connected to ground through a 10k resistor. The output is normally at 0V because of the 10K resistor, but when it is activated the output is pulled up to 5V. The output is connected to the LED through a 2.2k resistor in series. The resistor is required to prevent too much current flowing through the LED, the high value of 2.2k is chosen to make the LEDs less bright.

The outputs of the toggle switches are connected directly to some pins of the Arduino.

The abort button is connected directly to action group 10, this is because there is no keyboard shortcut for the abort action group, so action group 10 is used instead.

As there are 15 push buttons, there aren’t enough pins on the Arduino for all of them. Instead they are connected, through appropriate diodes, to a 4-bit binary bus. Each button has its own number, and when it is pressed it sends its number onto the binary bus. This reduces the number of pins from 15 to 4. The 4-bit bus is then connected to 4 pins of the Arduino, and code inside the Arduino can decode the binary and figure out which button has been pressed. Note that with this arrangement, a user can’t press more than 1 button at once. This is because it would cause the output to be the “sum” of the two buttons, and the Arduino would think a completely different button is being pressed.

The staging arming switch and stage button are also pulled down with a 10k resistor each. Their outputs are connected to an AND gate, and the AND gate’s output is what tells the Arduino when to stage. This is because staging should only occur when the arming switch is armed, AND when the staging button is pressed. This signal is also connected to the red anode of the common cathode RGB LED. As mentioned above, this RGB LED is green when staging is armed, and red when staging actually happens. The green input is the XOR of the button and the switch, as it should be green only when one of them are on. Therefore, an XOR gate IC is used.

The two buttons on the joysticks are also connected similarly to the rest of the buttons, with a 10k pull down resistor and a 2.2k resistor to its LED. Together with the staging output, these three signals again go into a binary bus, this time only 2-bits. The way this works is essentially identical to the previous diode encoder, and the 2 binary signals get connected to 2 Arduino pins. While this only saves one Arduino pin, it was actually necessary as all pins are used.

Finally, for each axis of the 2 joysticks, there is a potentiometer. +5V is connected to one side and GND to the other, the middle pin is the output. This means the output voltage shows the position of the joystick. The 4 outputs are connected to the 4 analog inputs of the Arduino.

**Housing:**

The housing is made up of 3 sections: the stainless-steel front panels and a 3D printed box. The first panel is positioned flat, while the second one is placed above it at a 40-degree angle. The angle makes it look more like a typical controller and also saves space. All the parts are held together with M6 hex screws. There is a lip along the box where it contacts the top of the angled panel, this allows the top panel to be secured with only 2 screws. The 3D model for these parts are designed in Fusion360, then exported to be 3D printed and laser cut. Text is also engraved on the top panels to indicate the controls. Most components include their own nut so they can hold themselves in place. The joysticks need 4 screws, so there are 4 small holes next to the joysticks.

Software

The code is split up into 6 main sections. Under “a\_global\_controller” all the global variables are declared and libraries are included. Under “b\_setup” various actions are performed to initiate objects and setup variables. Under “c\_loop” the main loop function is included, where actions like checking buttons and actuating joysticks are performed. Under “d\_buttons” all actions related to the buttons and toggle switches are included. Under “e\_read\_joystick” functions to read to position of the joysticks and convert it to a usable value are included. Lastly under “f\_actuate\_directions” a function to perform all the direction actions is included.

**A\_global\_controller:**

Firstly, two libraries are included. Keyboard.h is used to send commands to the computer, allowing the controller to control the game by acting as a keyboard. ezButton.h is used to debounce all inputs from buttons and switches, this avoids unwanted reactivations.

The variable loopDelay specifies the amount of delay after each time void loop() is executed, this delay ensures the “PWM” of the direction control isn’t too fast. I will get onto how this works later.

The next variables set the various default positions of the potentiometers. \*PotMid is where the midpoint of the specified potentiometer is. The Upper and Lower variables specify a range where there should be a “dead zone”, this is to prevent extremely small corrections that aren’t needed. These values are around 500 as the analog input of the Arduino is in 10-bits, meaning the value ranges from 0-1023. 512 is half of 1024, so that’s roughly where the mid points are.

The numerical values for these variables are found experimentally by running the inputs\_test program on the controller after it has been assembled. This program prints out the position of every joystick and switch few hundred ms, so the mid point of each axis of each joystick can be read from the serial port.

The next variables are for the various pins, here’s a list of what each pin is for:

A0: x axis of joystick 1

A1: y axis of joystick 1

A2: x axis of joystick 2 (z axis)

A3: y axis of joystick 2 (w axis)

toggles:

2: rocket/plane mode

3: SAS

4: RCS

5: Gears

6: Lights

Action Groups + abort + control buttons (15 total):

7: AGbin1

8: AGbin2

9: AGbin4

10: AGbin8

joystick buttons + stage (3 total):

14: JBSbin1

15: JBSbin2

16: reset button

The next variables define the various characters that the controller will need to press to control the game. I will first explain the simple variables, then move on to the arrays. “pitchDown” and the next 7 variables are controls specify the key needed to perform the action shown by their respective names. The array controlsAG[20] stores roughly 16 characters, each character is a key for an action that is activated by one of the buttons either in the action group section or the control section. These actions are, in order:

binary values for inputs:

Action group 1 – action group 10: 1-10

load: 11

save: 12

time-: 13

time+: 14

map: 15

Since the keys for load and save (F9 and F5 respectively) aren’t a normal character, in the initial definition of the array they are left as \*, the proper values are filled in later in the setup function. Note the space in the beginning of the char array, this is intentional as the 0th position in an array points to the first value. By leaving a space the 1st position will point to the control for action group 1. This also simplifies the binary decoding later.

controlsJBS[5] is similar, it includes the characters needed to control the following actions:

stage: 1

cut throttle: 2

full throttle: 3

The space after the placeholder \* is intentional, as the space bar is the key for staging.

The functions of the char arrays dirPlane[5] and dirRocket[5] will be explained later in the void loop() section.

Next, the various button objects are created, more information on how to use the ezButton library can be found [here](https://arduinogetstarted.com/tutorials/arduino-button-library).

Next, the variables for the toggle switches are created, I will explain their use later at the switches section.

Finally, the variables for the joysticks are created, I will explain their use later at the joysticks section.

**B\_Setup:**

This section of the code includes the void setup() function that is ran once whenever the Arduino restarts. First, all the pins are set to INPUT pin mode, this is because every single pin on the Arduino is receiving input from the various buttons and switches on the controller.

Next the keyboard is initiated with the begin() function. More information on the Keyboard library can be found [here](https://www.arduino.cc/reference/en/language/functions/usb/keyboard/). The Serial port is also initiated, while it is not used in normal operation, it’s extremely useful when debugging.

Next the debounce time for all the buttons are set to the debounceTime variable.

Finally, the missing keys for the controlsAG array is added. In this case KEY\_F9 and KEY\_F5 is used, for other non-character keys visit the link above.

**C\_loop:**

This section includes the void loop() function, which is executed in a loop after the setup is complete. The loop simply calls all the functions that must be executed over and over, I will explain what each function does as I get onto them. The section also contains another function, but I will explain its use later.

**D\_buttons:**

This section contains all the functions required to make the various buttons and switches work.

The first function is loopAllButtons(), it calls the .loop() action for each button which is required for the debouncing to work. It is called regularly in the loop as that is needed.

The next function checks the values of all the toggle switches (sasSwitchValue) and compares it to the in-game value (sasGameValue). If they are different (!=), that means the toggle switch has been changed, so the key for the action needs to be pressed. The new in-game value is then stored. This is done for all the switches except the mode switch, as mode selection isn’t part of the game but rather a part of the controller; therefore, the switch’s value is simply stored in a variable for later use.

The next function checks all the action group and control buttons, in other words all the buttons connected to the 4-bit binary bus. The first line converts the binary input into a decimal value, this variable shows which button has been pressed. This loop is repeated 15 times, with the value of “i” being 1-15. Because of the way that the controlsAG[] array has been set up, inputting “i” as the position returns the desired control. If “i” is the button that needs to be pressed, it presses it. Otherwise it releases the button.

The next function is very similar, instead there is a 2-bit input. This function checks the two joystick buttons and the stage button.

Finally, the checkReset() function checks the reset button, and resets the value if it is pressed. The reset button is needed because, for example, when a player reverts the game to launch, SAS, RCS and Lights are off. The controller has no way of knowing when the game is reset, so the user must manually tell it. When the button is pressed, all the in-game values are set to the default at the start. This is all that must be done as the checkSwitches() function will see a change has occurred and press the appropriate controls. This is also the first time where the mode matters. For planes the gears are up by default, but for rockets they are down by default. This means depending on the mode the controller must set the game value to different things.

**E\_read\_joystick:**

This section and the following section include functions to control the joysticks. This section takes the analog input and converts it to an actuation value that is easier to work with when actually executing the movement. This section contains 4 functions, but all of them are basically identical and are just for the different axes. I will be explaining the setXActuation() function, but the same logic applies to all other functions.

First the voltage in the analog pin is read and stored in the variable xPotValue. If the value is between the upper and lower mid points, meaning it is in the dead zone, the actuation in both directions is set to 0 since no action is desired.

Otherwise if the value is very big in one direction, the actuation in that direction is set to 100, while the actuation in the other is set to 0.

Next the function checks for the case where the voltage is below the MidLower value, in other words towards the right. In this case the left actuation is set to 0, and the right actuation is calculated with a map function. The map function takes the voltage reading (between 0 and around 512) and converts it to a value between 100 and 0. In this case lower voltages mean higher actuation.

Finally, if none of the above cases are true, the function checks for the case where the voltage is above the MidUpper value, meaning it is tilted to the left. Similar actions are performed, this time setting the right actuation to 0 and setting the left actuation with the map function.

**F\_actuate\_directions:**

Now that there is an actuation value between 0 and 100, it is easier to make the Arduino “PWM” the keys. Let me explain what I mean. Normally the “WASDQE” keys are used to control the direction of the craft in-game. However, on a keyboard the keys are always activated either 0% or 100%, it is impossible to go anywhere in between. This means if a player wants to make a small adjustment to the craft, they must tap the control keys very quickly. Since all that the Arduino can do is press keys, it is also not able to directly control the craft’s controls. The following code essentially taps the key very fast and at a certain duty cycle, I have decided to call it “PWM” because that’s basically how PWM works.

The function to “PWM” the direction control takes 3 inputs: the actuation (between 0-100), the key that needs to be pressed and the last time that the key has been released. These inputs are given in the void loop() function where this function is called. The actuation input is easy as that is what the previous function calculates, the lastReleased variable is also easy as the loop sets the variable to the millis() function every time through the loop; the only tricky input is the key that the function has to press.

This is because the key depends on the mode. In plane mode roll is used much more often than yaw, so it is better if roll is controlled by joystick 1. On the other hand, in rocket mode yaw is more important than roll, so it’s better if yaw is controlled by joystick 1. The dir2key function in the loop section decides which key is needed. It takes an integer input indicating which axis is being executed and returns the appropriate character from the dirPlane or dirRocket array depending on the mode. The way these arrays work is like how the button arrays work, I will not be explaining it again.

Now that the actuateJoystick function has its 3 inputs, it can “PWM” the joystick. First the function checks if the actuation is set to 100. If it is, the key is simply held down. Next the function checks if the actuation is set to 0. If it is, the key is simply released. Lastly, if the actuation is some other value, it performes the “PWM” action.

The amount of time since the last release can be calculated by subtracting millis() (the current time) from lastReleased. The amount of time that the key must be held down will be some multiple of the amount of time that it has been released. This ratio can be calculated by the actuation divided by 100 minus the actuation. I can’t really think of a way to explain why it works, you can just look at it and think it through for yourself. “The proof of this formula is trivial and is left as an exercise to the reader”. Now that the hold time is calculated, the function presses the key, delays for the hold time, then releases the key. One “PWM” cycle has been done.

The void loop() function calls this function for each of the axes. At the end of the loop there is a delay. This is because the game is unable to register tabs faster than about 20 times a second, so to get anything to happen the key presses must be slowed down.