Capstone Technical Report

Design and Construction of a Kinetic Art Weather Display

By:

Lisa Accolla, Jack Davis, Katherine Ellis, Adam Lenox, Josh Rigby

December 1, 2020

Technical Advisor: Gavin Garner

Table of Contents

PROBLEM DEFINITION

Working in secluded or underground workspaces often causes one to lose track of the time or the weather outside. After working multiple hours on end, one can step outside to find that the weather, temperature, or level of sunlight has changed dramatically from when they first entered the work area or even worse, realize that they have missed an appointment. Especially in the midst of a pandemic and wearing significant personal protective equipment, it is difficult to check one's phone to see how the conditions might have changed.

The proposed solution to this problem is the creation of a kinetic weather display. Through a combination of electronics and mechanical devices, the display will be designed as a window capable of changing weather patterns, brightness, and sun position to match the real-time conditions outside of the work space. The appearance is similar to a window to make it aesthetically appealing to those working in the area while hiding the mechanisms that allow the weather patterns to change. An LED display in the front also provides data on the current temperature and time, thus providing important information in one easy-to-view location.

INITIAL DESIGNS AND CONSIDERATIONS - LINEAR MODELS

Among the most important considerations for this project was the mechanism of changing weather patterns easily and efficiently. Early models of the weather display were linear in nature, meaning that the weather patterns would move in horizontal or vertical directions across the face to indicate the change of weather. One design, shown in *Figure 1*, relied on the use of two rollers with transparent foil that could move up and down tracks mounted to the side of the display to show either rain or snow while conserving space and maximizing the viewing area of the window. However, having a thin foil like this was deemed to be less durable than many alternatives and difficult to change if it would become damaged for some reason during use.

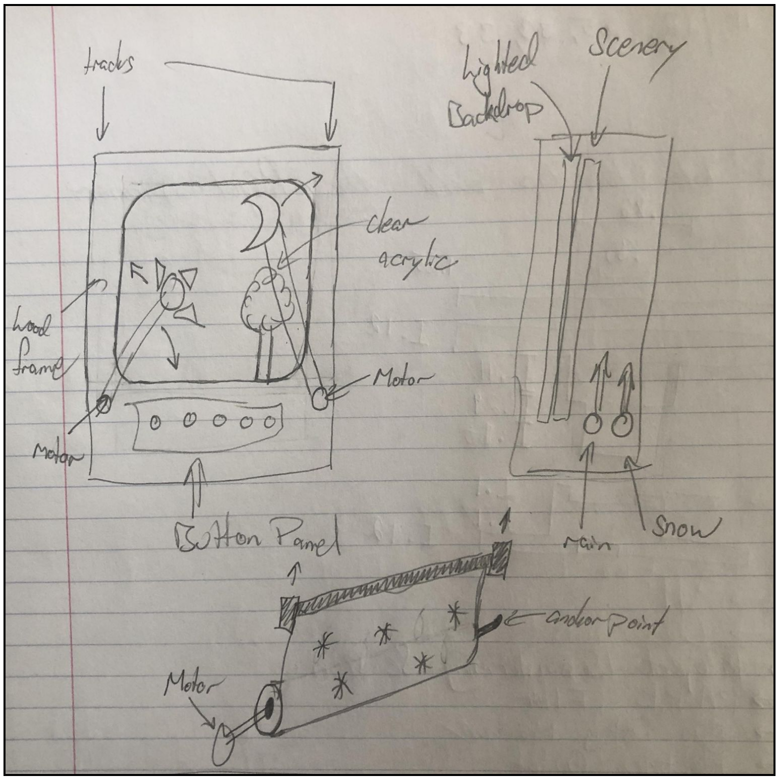


Figure 1: A rectangular design featuring rolling weather patterns and a side-mounted sun and moon

Another linear design, shown in *Figure 2*, resembled a fish tank with a large display on the top and a bottom section where acrylic panes with weather pattern designs could be hidden below the viewing area. Similar to the first design, this display would feature rolling tracks to move the various sheets in and out of view with certain tracks set aside for each location. The difficulty of this design resulted in its bulk when mounted to a wall.

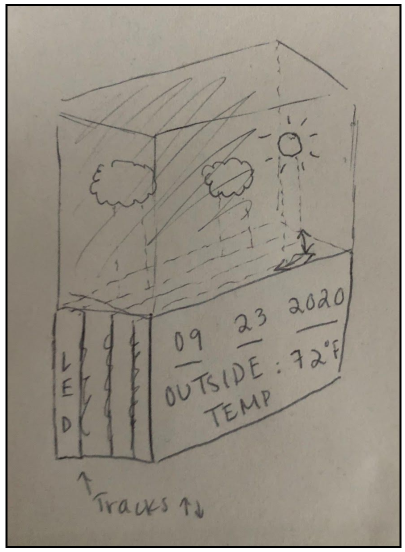


Figure 2: A rectangular design with hidden acrylic sheets displaying weather patterns and an open top

The second major consideration is the movement of the sun and moon. The initial problem definition stated that the desire was to have accurate sun and moon cycles that would reflect the sunset and sunrise times of Charlottesville. With the linear designs shown above, this is difficult to achieve. The first has levered beams that move in and out of view from the sides, which could indicate the day or night time, but would not be accurate to the exact pattern across the sky. The second has similar vertical movements that could move the sun and moon into view, but not easily across the sky in an arc. It is possible to achieve a more two dimensional motion in either case through the use of multiple motors and integrated beams that support the objects, however this could have detracted from the overall aesthetic and artistic nature of the piece.

FINAL DESIGN - THE CIRCULAR MODEL

The final design for the weather display shifted from a linear model to a circular one similar to what is shown in *Figure 3*. The full product is a circular design, where the window display is on the top semicircle while the mechanics and wires are hidden behind a solid face on the bottom semicircle. The advantage to a circular design like this is twofold. Concerning the weather patterns themselves, a large compartment on the bottom makes it possible to hide full sheets of acrylic engraved with snow or rain designs that can rotate into view through the use of hidden motors. These sheets would be similar in strength to the second design, but instead of linear movement, it would be rotational. The second advantage is in the movement of the sun and moon. Having a circular display makes it possible for the two pieces to rotate across the sky and be synced with the sunrise and sunset times that are input into the system. It models the sky much more accurately than the linear model shown in the first two designs. These two functions combined also make the design more compact and prevent it from protruding from the wall too much.

Figure 3: A preliminary sketch of the circular model, which was later adapted to the final model

The LED display is located on the face of the lower hemisphere of the circle, embedded in the front face of the display to make the design more efficient in terms of empty space. LED strips are also inside the display to illuminate weather patterns and simulate snow or rainfall through programmed illumination patterns. A backlight serves as the final layer of the display, just behind the scene of the window. Lights are connected to this in order to adjust the color and brightness of the background, making it possible to form different settings such as the nighttime, daylight, or overcast weather. A preliminary depiction of this is shown in *Figure 4.*

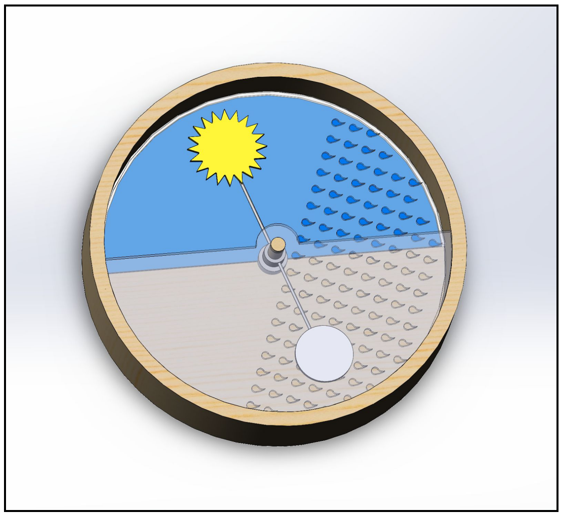


Figure 4: A rough model to conceptualize the final, circular design of the weather display

Information is fed to the display by a Raspberry Pi, a single-board controller capable of extracting weather and time information from the internet using an open-source application programming interface (API). This communicates directly with the Parallax Propeller microcontroller chip. The Propeller chip is a parallel processor with eight "cogs" capable of running simultaneous lines of code to control the movements and actions of the various motors, LEDs and sensors positioned around the display.

PROTOTYPING AND MODELING

Prior to constructing the weather display, several prototypes and models were made to understand the spacing of the various components as well as the size of the device given the restraints imposed by the equipment available for use.

The first model was a proof of concept made out of cardboard and is shown in *Figure 5*. It was capable of rotating panes around a center pin by hand and it was sixteen inches in diameter, which was deemed too small for optimizing visible window space, but it provided a good estimate of how the device could be fitted together.

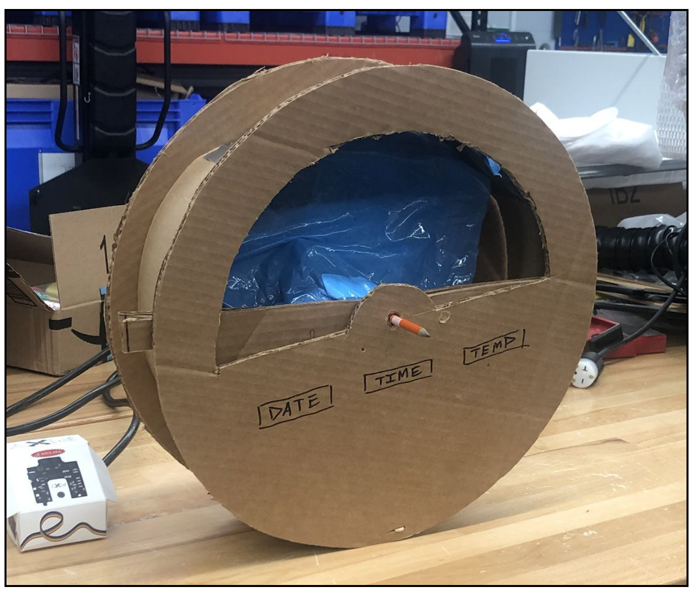


Figure 5: Cardboard model of the circular design with two rotating panels inside

Following the creation of the cardboard model, dimensions were laid out using a measuring tape on a whiteboard in order to better judge the potential size of the display as shown in *Figure 6*. These dimensions assumed a window diameter of 24 inches and 30 inch overall diameter to leave space for covering the mechanisms that would allow the acrylic weather sheets to rotate about the center. *Figure 7* is a sketched section view of how the two acrylic panes may fit together and stagger in size in order to be able to stack motors more effectively.

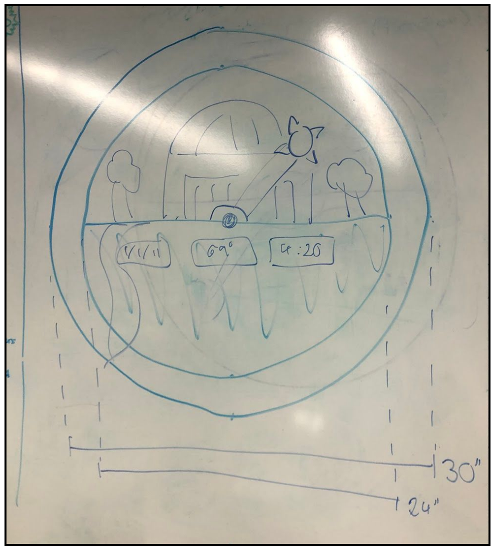


Figure 6: Whiteboard drawing of dimensioned window front face and viewing area

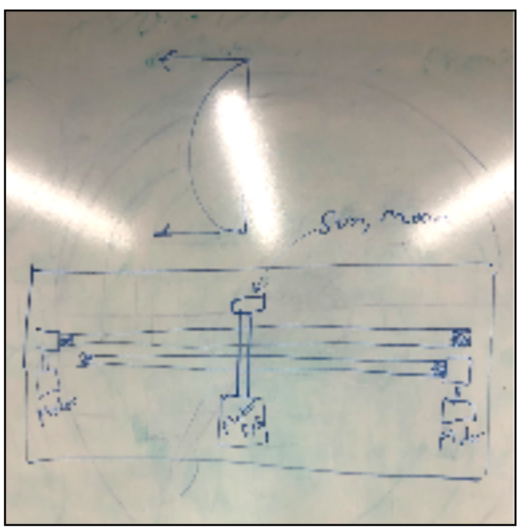


Figure 7: Section view of display depicting how motors for the two weather sheets and the sun and moon could fit together

Following these initial designs, more advanced models were made in SolidWorks, a 3D modeling software capable of forming complex assemblies to scale. It is also useful for exporting laser cutting, 3D printing, and CNC milling models for construction at later stages in the build process. At this stage, the group decided to use lazy susan bearings in order to mount the acrylic sheets and the gears that would turn them. With an outer diameter of 23.63 inches, these bearings became the limiting factor in the size of the model and reduced the diameter from the preliminary drawings and estimates. The finalized window diameter was 20 inches and the outer diameter was 26 inches. It was also decided that the time, date, and temperature displays would be one large piece instead of three separate displays. This decision was made because of the limited number of processors available on the Propeller Chip, as well as a desire to make the display easier to read from farther distances. Finally, in the SolidWorks model, motors were mounted from a center beam as opposed to the back panel. This can be seen in *Figure 8*. They also utilized spur gears mated with internal gears to keep the design compact. Had regular gears been used for the acrylic movement, more space would have been needed to fit the motors on the outside of the bearings, thus resulting in a wide and unattractive frame.

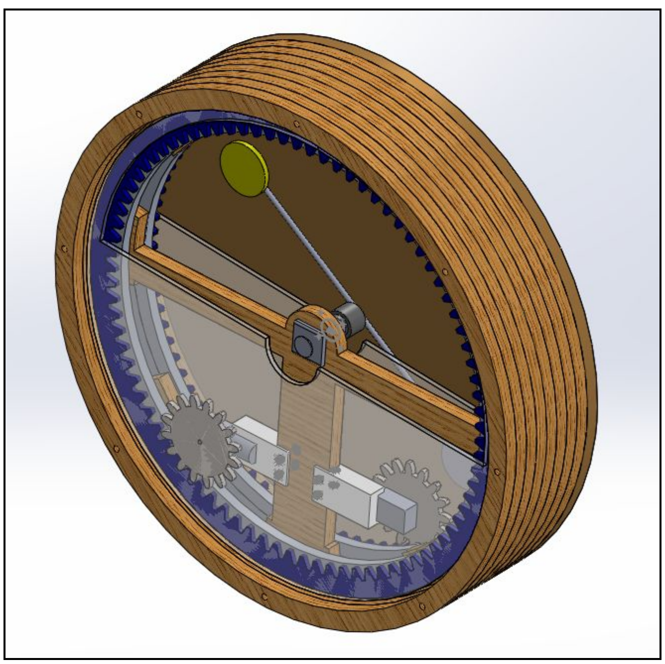


Figure 8: An isometric view of the 3D model without the front face included

The SolidWorks model is particularly useful in studying the spacing of the various components. Because one of the goals was to keep the display compact and as close to the wall as possible to better simulate an actual window, the 3D model was able to layout the wooden frame and all of the internal pieces. That includes the lazy susan bearings, the acrylic weather sheets, the internal gears, the nuts, the bolts, the washers, and the spacers that held them together. *Figure 9* depicts a section view of the model which allowed the group to study how the various pieces could fit together. Following initial placement of the major components, bolts and other connectors were added using models pulled from McMaster Carr to ensure that proper sizes were selected for the final model.

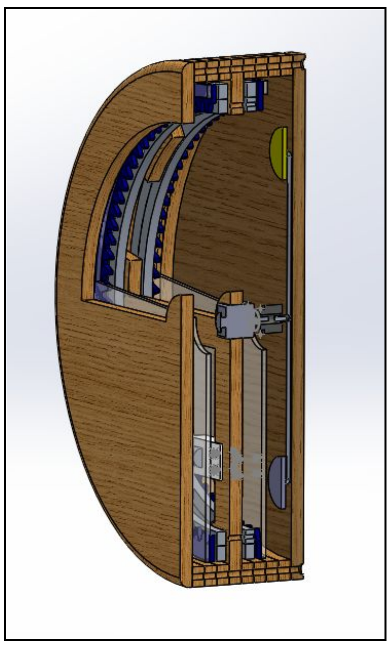


Figure 9: Section view of the weather display that shows spacing of major components in reference to the frame

The benefit of using SolidWorks, as mentioned above, is the ease of exporting components for construction. The gear and half-sheet models for acrylic sheets were easily exported as .dxf files that could be read by the laser cutter without having to adjust size beforehand. It is also possible to modify a part and then update it in the assembly without adjusting the assembly's mates.

SELECTING MOTORS

Among the most important components of the weather display are the motors used to drive the motion of the two acrylic sheets, as well as control the movement of the sun and moon piece. After experimenting with various motors, it was decided that worm gear motors could be used for both.

The first motor tested was a servo motor. The servo motor is a smaller motor that is capable of moving to a precise position based on the duty cycle of an input signal. The duty cycle, as shown in *Figure 10*, is a ratio of high voltage to low voltage over a set period. For instance, a 100% duty cycle would be a constant output of the maximum voltage, while a 0% duty cycle would be a constant 0V output.

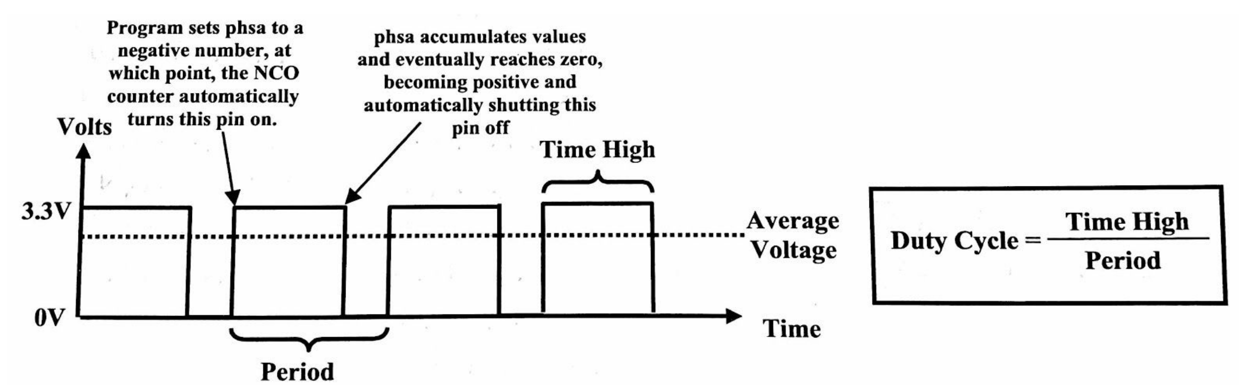


Figure 10: Graphical representation of how duty cycle sets the speed of a motor.

While this type of motor is good at moving to a position and holding it, as well as outputting significant torque, it is not particularly useful in this design. The drawback to the servo motor used in testing was its limited range of motion. The servo used, shown in *Figure 11*, was only capable of rotating 180° before it reached a mechanical stop. It would have been possible to use this motor in moving the acrylic sheets, but the gears mounted to the motor itself would have taken up considerable space in an already compact design. It would not have worked for the sun and moon, as it was not capable of rotating continuously in a full circle.

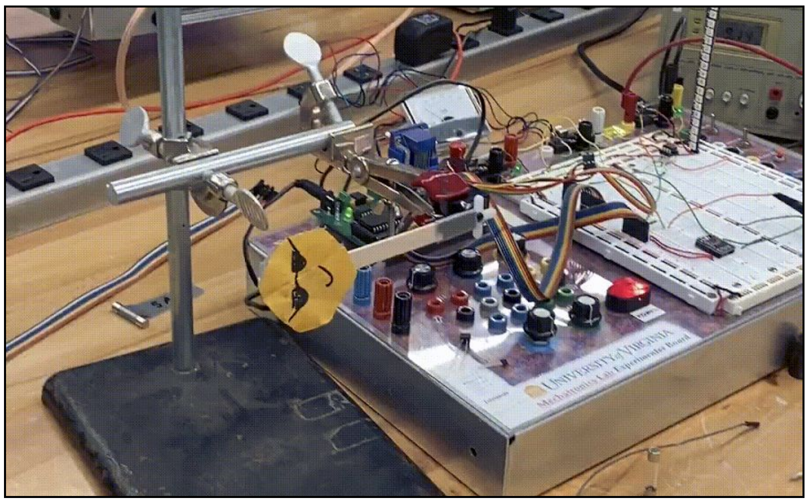


Figure 11: RC servo motor model with a prototype sun arm attached

The second motor tested was a bipolar stepper motor with a microstepper driver, as shown in *Figure 12*. The stepper motor is a type of motor that runs off of internal magnets that turn the motor's head continuously and smoothly between poles, thus making one "step". It is capable of making very small adjustments in order to vary the speed and distance of motion, thus making it ideal for controlling the sun and moon in the display. However, the drawback to this motor was twofold. Firstly, it has some internal resistance to turning when not powered, but it relies solely on the friction of the motor itself when holding a position. This makes it easy for the motor to accidentally lose its set position, especially if the load it carries is not balanced. In other words, the loads at either end of the sun and moon arm would need to be perfectly balanced to ensure there would not be unwanted drift in the motor's positioning. The second problem was that constantly running the motor would cause components to heat up, ultimately resulting in permanent damage to the circuitry. This could be avoided by having the motor make one "step" and then powering itself off to eliminate the heat generation from the current it draws. However, as described above, there are factors that could create error in this movement. It was decided by the group to forego using the stepper motor in favor of one that could better hold position.

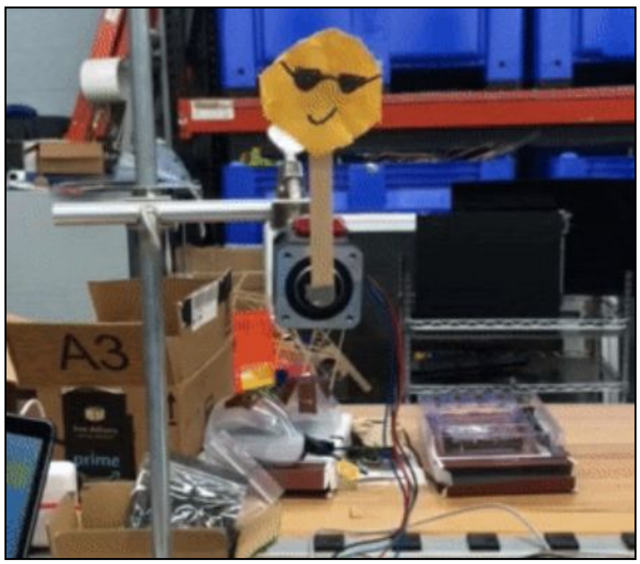


Figure 12: Bipolar stepper motor with microstepper driver with a prototype sun arm attached

Ultimately worm gear motors, as shown in *Figure 13*, were selected for both the sun and moon mechanics, as well as rotating the acrylic sheets. Worm gear motors are a type of DC motor with a gearbox attached that significantly increases the torque output and have such a high mechanical advantage that the output shaft will not rotate when there is no power applied. Like regular DC motors, they run off of pulse width modulation, which allows control of the speed and direction of the motor's movement. The rigidity and strength of these motors is good for ensuring minimal deviation from the programmed path of the sun and moon, as well as little drifting of the acrylic sheets when being used to display a certain weather pattern. The worm gears used in the final model were equipped with quadrature encoders, devices used to track the movement of the motor by emitting a pair of pulses that can be counted to measure the direction of a turn and the total distance traveled.

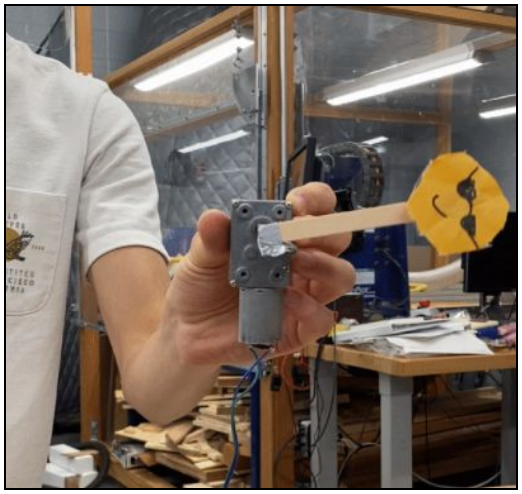


Figure 13: Worm gear motor with prototype sun arm attached

One drawback of using worm gear motors with quadrature encoders is that they lack a built in way to "home" themselves. This means that, should one component be moved by something other than the programming or the power be cut to the display, there would be no way for the motor, and subsequently the Propeller chip, to understand that the motor was no longer in the correct position. In the event of such an error, additional code must be added in order to reset the system before carrying on in its function. To do this, optical limit switches were used. These sensors rely on a thin laser projected between two plates. When an object passes between the plates and breaks the laser's path, a signal is sent to the Propeller chip. Using this device and a small attachment to the weather sheet mechanisms, it is possible to program a reset function that homes all pieces of the display prior to their correct initial positions.

CUTTING ACRYLIC PIECES

A large number of pieces, such as supports for motors, gears, covers for LED displays, and the weather sheets themselves, were made of acrylic. These pieces were cut and designed using a laser cutter. The laser cutter available was capable of performing three functions: through cutting, rastering, and etching. Available for use in the laser cutter were 0.125" and 0.25" continuous cast acrylic sheets. Through cutting was most useful for support pieces and gears. The weather sheets themselves, however, used a combination of through cutting, rastering, and etching in order to make aesthetically appealing weather patterns.

Design of the parts made on the laser cutter was done first in SolidWorks by creating parts with the exact dimensions required for the printed version. After exporting the SolidWorks part as a .dxf file, the program CorelDraw was used to edit the outline that would be cut by the laser cutter. Different colors were used to reference what type of cut to make with red indicating a through cut, black indicating a raster, and blue indicating an etching. It was here that the snow and rain patterns were created and added to the shape of the acrylic sheets in order to make the required patterns. The benefit of the laser cutter is also in its ability to vary the intensity of its beam to make deeper cuts, which were often cleaner and more apparent than a shallow cut. These techniques of increasing the rastering intensity and slowing the speed made bolder patterns as a result.

The limitations of the laser cutter came in its size. The machine itself had a cutting surface 32" by 17.5" in dimension, meaning that all parts required for the display needed to fit on that plane. As such, full circles of weather sheets could not be cut to make the installation process easier, and half sheets were used instead to save material and space in the display. The acrylic sheets in stock were 1' by 2' in dimension, which were just wide enough to fit the diameter of the lazy susan bearings. In some cases, to overcome these limitations, pieces were cut in several segments to conserve acrylic sheets without wasting material. The internal spur gears mounted to the inside of the lazy susans were one instance of this, where the gears themselves were split into four pieces with connecting ends that could be fit and glued together after printing. Smaller pieces, such as the mounts for motors did not have to be broken up in this manner. The weather pattern sheets were cut in full, as breaking them up into smaller pieces would have detracted from the window's view during a rain or snowstorm.

3D PRINTING

Smaller parts for the weather display were created using the 3D printer. Compared to the laser cutter, the 3D printer is capable of making much more detailed parts with features extending in three dimensions as opposed to solely two. This is done by layering strips of hot ABS plastic on top of each other working from the bottom to the top to shape the part according to its specifications. However, the tradeoff for this capability is time, as it usually takes significantly longer for the 3D printer to make a part. Thus, the laser cutter was used as often as possible to make basic parts, and the 3D printer was used for parts that required details in multiple dimensions or parts that were thinner than the acrylic offered in the laboratory.

Similarly to laser cutting, a part that needs to be 3D printed is first built in SolidWorks and then exported as a .STL file. Then it can then be opened in a 3D printing software, CatalystEX in our case, that communicates with the 3D printer. In CatalystEX, it is possible to vary the layer resolution, the model's interior density, and the density of the support material. The denser and smaller the material is, the stronger the part will be as a whole. However, as it also uses more material, the part will be more expensive. The software also has a function that can change the orientation of the part and build layers, making it possible to orient the part such that its strength is optimized. When these details are finalized, the part is added to a “pack” which is the final group of parts that will be printed. Multiple parts can be added to one pack to print them at the same time.

As mentioned prior, the 3D printer is particularly useful for making pieces of unique designs that could not otherwise be created using a laser cutter. Pieces like the mounts for the sun and moon LEDs (*Figure 14*), the mount for the sun and moon motor (*Figure 15*), and fasteners to keep the background LEDs in place (*Figure 16*) were all built in SolidWorks and printed using the 3D printers. These designs are shown below for reference.

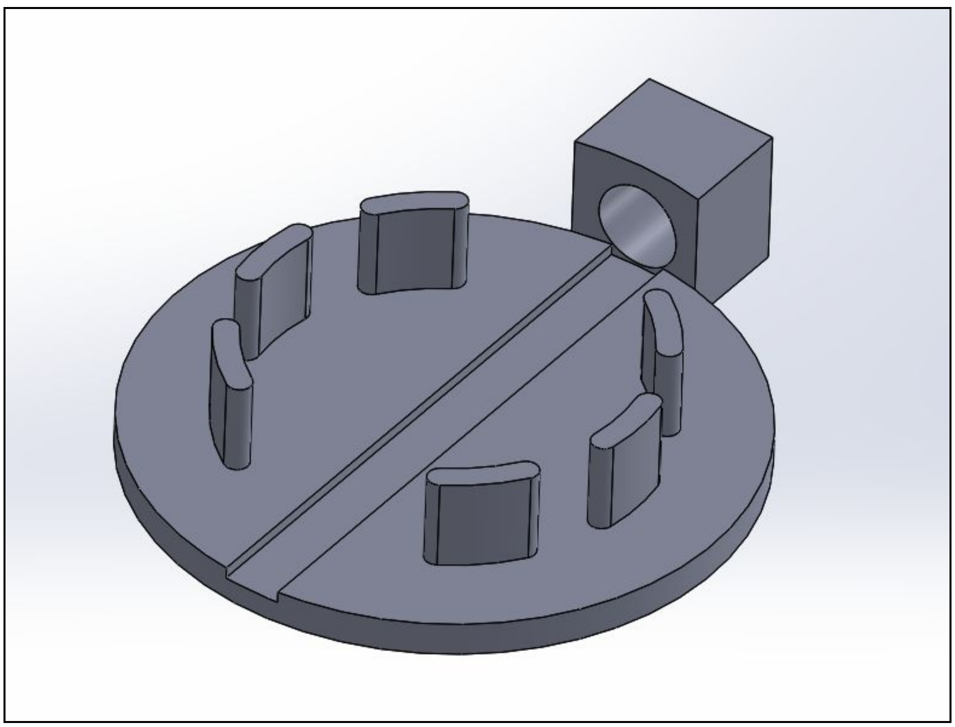


Figure 14: Mount for circular sun and moon LEDs designed for 3D printer

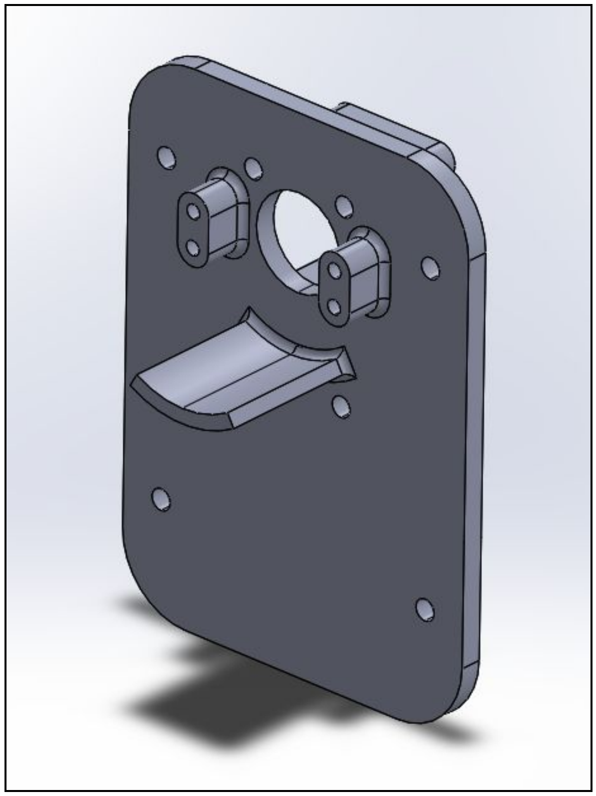


Figure 15: Mount for worm gear motor that controls the sun/moon beam, designed for 3D printer

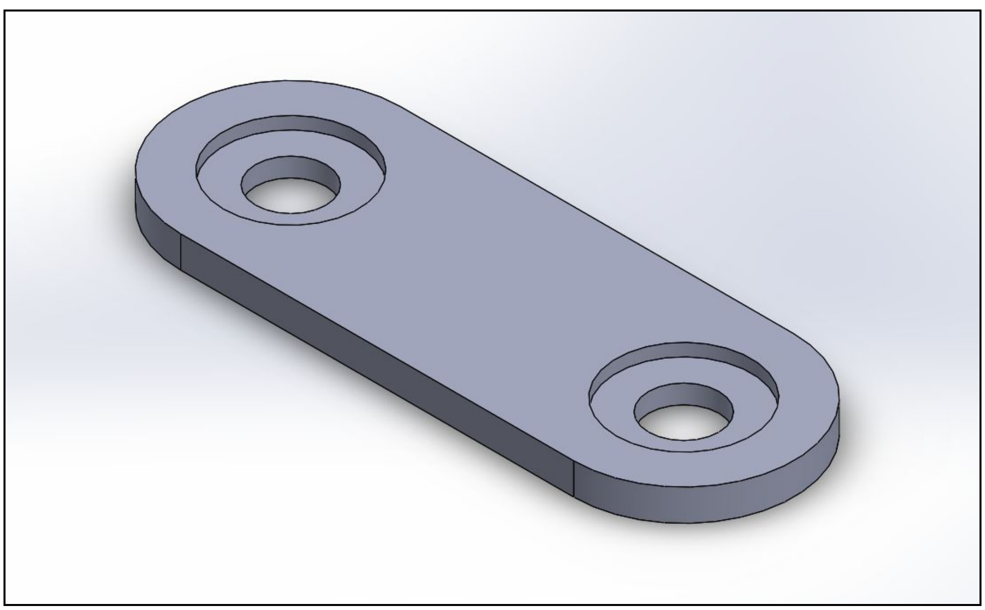


Figure 16: Fastener for holding background and weather sheet LEDs in place designed for 3D printer

CNC MILLING

Large parts for the assembly were cut out of wood using the Shopbot PRSalpha 96-48 Computer Numerical Control (CNC) Router. For this machine in particular, the build area is 105” x 49” x 8”, which was capable of fitting the larger frame pieces for this project. Although this machine is able to move in three dimensions, it is important to consider that the end mill bit has thickness and is fixed vertically which prevents the machine from making certain cuts. For example, inner corners within the part cannot have sharp edges: instead they will always have fillets the same radius of the drill bit. Additionally, curves along the vertical axis can only be created on one side--for example, in order to create a 180° rounded edge, the entire stock material would have to be flipped over to round the other side.

In order to use this machine, each part and the stock material from which it would be cut had to be modeled to scale in SolidWorks. The main purpose of using SolidWorks to prepare for CNC milling is to arrange the part exactly how it will be cut and how it will look in real life. This includes orienting the part exactly, adding tabs to the part which will hold it to the stock while cutting, and using Computer Aided Manufacturing (CAM) and the HSMWorks Plugin within SolidWorks to work out all of the toolpaths that will be used to cut out the part. The preparation in SolidWorks is a tedious process, and it is important to note that it is easy to make a small mistake which ruins the entire part. However, in taking the time to follow all of the preparation steps correctly, CNC machining becomes an extremely powerful tool in successfully creating large parts for an assembly.

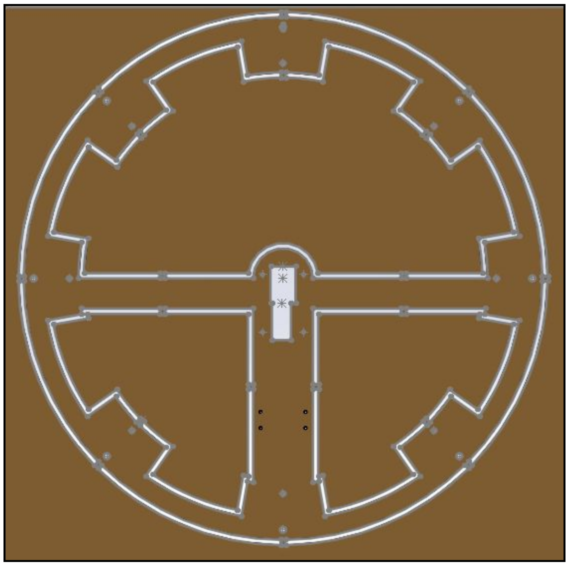
 

Figure 17: Side-by-side images of the center frame piece cut out using the CNC mill. On the left is a model of the part in SolidWorks, and on the right shows an image of the completed part

For this project, the center layer (shown in *Figure 17*), the eight outer frame layers, the front face, and the back layer, were all cut using the CNC machine. Each layer was cut from 0.69" thick plywood. In order to conserve wood, the eight outer frame layers were split into quadrants (shown in *Figure 18*) and later assembled into full rings using wood glue. The quadrants were modeled specifically such that one piece would fit easily into another, sort of like puzzle pieces, helping to align them as precisely as possible while gluing them together.

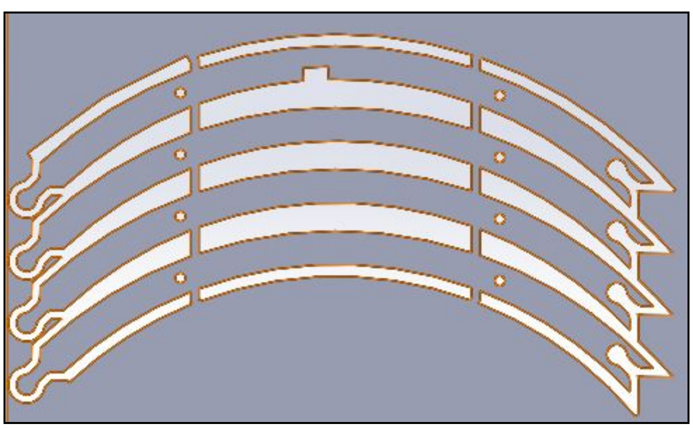
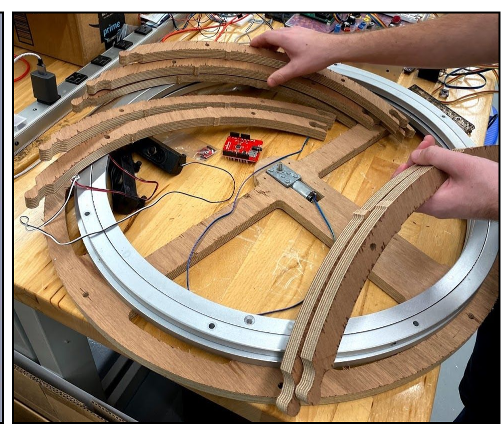
 

Figure 18: The image on the left is taken from SolidWorks of four quadrants to cut one outer frame layer out using the CNC. The image on the right shows the pieces after cutting and before gluing

An important part of CNC machining is choosing the best end mill bit to use when cutting out a part. The smaller the end mill bit, the longer it will take to cut. Furthermore, switching out an end mill bit in the middle of cutting a part is inconvenient and may cause problems, so it is recommended to try to avoid this if possible. Originally, the quadrant frame pieces shown above were designed to have ⅛” inner holes, meaning that it would have to be cut using a ⅛” end mill bit. This would have taken at least a few hours to cut all of the frame layers. However, the design was altered to have ¼” inner holes instead to accommodate stronger bolts, and this ultimately ended up saving a lot of time on the CNC since a ¼” end mill bit was then able to be used.

Another aspect of using the CNC is that the machine works blindly given the toolpath instructions as soon as the process is started. The particular machine used did not have the capability to stop itself if there were any problems in the milling. An example of this occurred when cutting out the outer frame layer quadrants. In the first run of cutting out most of the frame pieces, the end piece of one of the quadrants broke off completely. This was because the plywood had inconsistencies within the material (it is non-isotropic), causing it to break under the stress from the end mill bit. The CNC machine had no way of knowing that this occurred and it continued cutting for the rest of the time as if the piece was still intact.

As stated earlier, it is important to create tabs on your part in SolidWorks which will hold it to the stock material throughout the CNC process. Without these tabs, it would be easy for a part to move or offset throughout the process, causing interference and incorrect cuts. To reiterate, the CNC machine itself does not receive feedback if any parts or materials accidentally offset, so this is why tabs are necessary. Therefore, a part cannot easily “pop out” of the stock material like those created with the laser cutter or 3D printer. Any parts made on the CNC have to be manually cut at the tab sites and then sanded down.

WEATHER PATTERN DESIGNS

When deciding the type of weather patterns to include in the final product, both the typical weather in Charlottesville, Virginia along with the feasibility and team's capability of incorporating each design were considered. Charlottesville has seasonal weather, so it would be necessary to have various weather patterns available for each season. The method of incorporating weather patterns was through "weather sheets" made from laser-cut acrylic working in conjunction with programmed LED strips to mimic the current weather conditions. It was decided that snow and rain weather sheets would be included, as they are the mostly likely weather conditions other than no precipitation. The raindrop and snowflake designs were drawn in SolidWorks then exported as a .dxf file to be used with CorelDraw in conjunction with the laser cutter. Also in SolidWorks, a semicircular shape was made for the weather patterns to be printed on and would be moved by a gear-motor system to cover the display to match the weather conditions outside when appropriate. Four holes large enough for 8-18 bolts were later drilled into each semi-circular acrylic weather sheet after laser-cutting such that each sheet could be mounted to a spur gear. LED strips also lined the inner clock walls parallel to the semi-circular edge of the weather sheets, to add an extra effect and make the patterns more visible to the viewer. The snow and rain weather sheets can be seen in *Figure 19* and *Figure 20*.

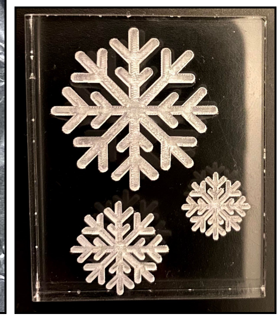


Figure 19: Final acrylic snow sheet implemented into the final assembly, along with the smaller test strip of acrylic used to test etching intensity and snowflake size.

Figure 20: Final acrylic rain sheet implemented into the final assembly, along with the smaller test strip of acrylic used to test pattern, size, and etching intensity.

Other weather conditions such as cloudy weather, thunder storms, and foggy weather could be represented through the use of LED strips along the edges of the clock. Both cloudy and foggy weather conditions could be displayed by using more grayish LED colors. Three LED strips surround the perimeter of the display window at the rain sheet, the snow sheet, and the backlight, creating an opportunity for the colors to cover the display window entirely. For a thunderstorm, one LED strip is programmed to flash white lights through the rain weather sheet to mimic lightning. When these conditions are nonexistent, the backlight LED strip will project blue to show a clear sky.

BACKGROUND DESIGN

Figure 21: Background with the Rotunda

The background is very similar in design to the weather pattern designs. It consisted of a semi-circular, 1/8”, translucent, white acrylic sheet that was slightly smaller in radius than the inner wall of the weather clock frame. Also used were a semi-circular, opaque, white sheet of plastic, 1/4” spacers, and an 1/8” acrylic model of UVA’s Rotunda. The white sheet of plastic was secured directly to the back of the frame and the 1/4” spacers were used to elevate the white acrylic sheet from the white plastic sheet. Like the weather sheets, an LED strip was secured to the inner wall of the clock such that it was parallel to edges of the acrylic and plastic sheets. The LED strip was secured at a height that allowed the LEDS to shine into the 1/4” gap between the white acrylic and white plastic. The idea is that the LEDs will bounce light off of the white plastic to evenly diffuse through the white acrylic, effectively making the white acrylic sheet similar in color to whatever is projected by the LED strip. The Rotunda will be secured on top of the white acrylic sheet for artistic design as shown in *Figure 21*.

ASSEMBLY

Assembly of the weather display begins with the central layer shown in *Figure 22*, which is where all the moving parts are mounted. When the piece was created, inner beams were included to use as mounting points for various motors and features. However, the design of many of these pieces were made after the wood was cut from the CNC mill, and as such, holes for mounting them were drilled in later. The hole in the middle is for the motor that controls the sun and moon arm, which was enlarged to allow the worm gear motor to fit into the gap.

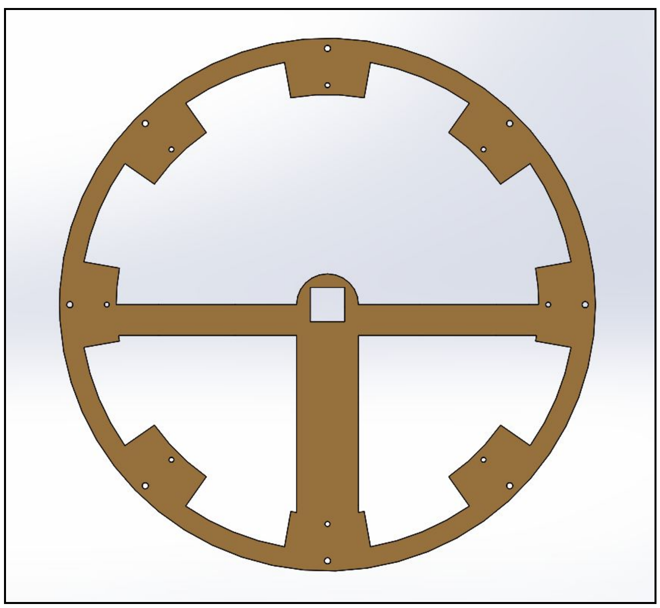


Figure 22: Central layer for mounting mechanisms

The first pieces mounted to the center layer are the two lazy susan bearings. Four 1.75" 8-18 bolts connect the inner ring of the lazy susans to the eight wooden tabs, with four bolts being used on each bearing and alternating tabs selected around the ring. The bolt head touches the wood itself with a 3/16" spacer separating the bearing and wood to allow for free rotation of the bearing's outer ring. A 7/32" tall nut caps the end of the bolt and is tightened to the surface of the lazy susan bearing. Note that the bolts for both lazy susan bearings must be put through the appropriate holes before the lazy susans are attached, as they will not fit through the holes after one bearing has been bolted down.

The next pieces to attach are the internal spur gears and weather panes, which must be attached at the same time. The gears have been marked corresponding to their location on the lazy susan bearing, as the holes on the bearing were not manufactured to be perfectly symmetrical. The holes on the gears must line up with the holes of the bearing. The acrylic weather sheet is placed on top of the gear, again lining up the markings to ensure the proper placement of holes. 3/16" spacers separate the gears and bearings, but not the gear and weather sheet. Four 1.75" 8-18 bolts are used to secure the gear where the weather sheet is stacked on top, and four 1.5" 8-18 bolts for the four holes without the acrylic sheet. 7/32" 8-18 nuts are placed between the outer ring of the lazy susan and the wooden tabs below. Washers are placed on top of the acrylic pieces to prevent the bolt heads from cracking the acrylic. Using an appropriately sized wrench and a phillips head screwdriver, the bolts are tightened into place to secure the weather sheet assemblies. This process is repeated for both sides. Both sheets are oriented so that the etching faces the front window of the display, meaning that the snow sheet (which is the farther back of the pair) will be installed with the etching facing the wooden center layer. This is all shown in *Figure 23.*

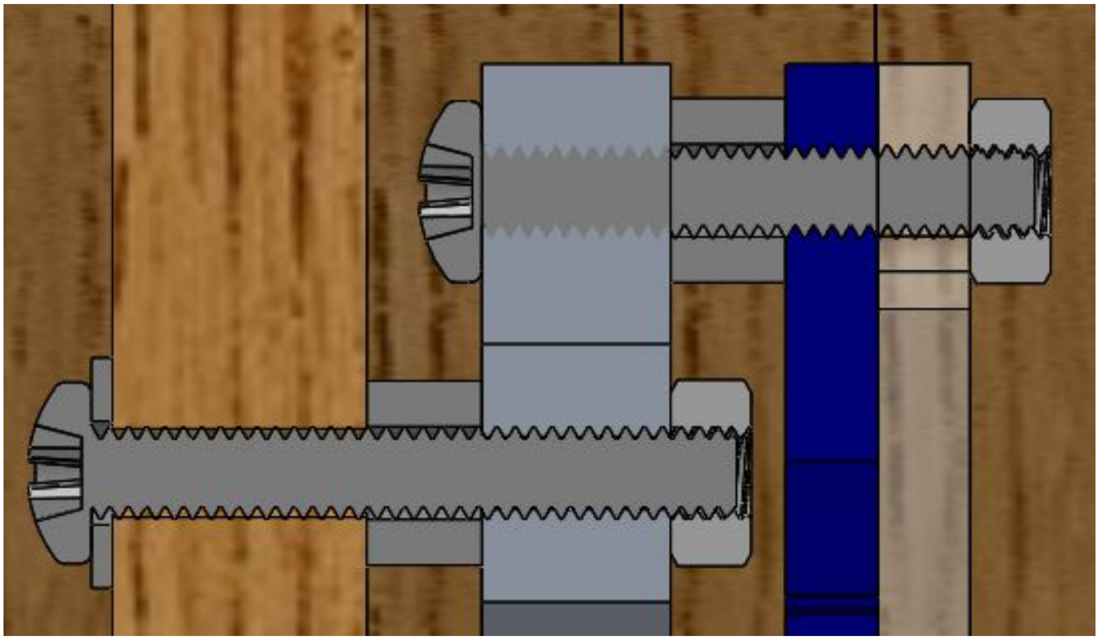


Figure 23: Section view of lazy susan bearing and weather sheet assembly

Once the two weather pieces have been installed, the other smaller features can be attached to the center layer. The first among them is the sun and moon assembly. The 3D printed motor mount is the central piece of this assembly (shown above in *Figure 15*). Prior to attaching it to the motor, the coupler and 8mm shaft must be attached to the end of the motor using set screws. Then, four M3 x 30mm bolts attach the stepper motor to the 3D printed mount. The slip ring sits on the other side, suspended by small supports and resting on the curved lip that extrudes from the surface. Four M2 bolts are used to fix the slip ring to the mount from below. The M3 set screws at the top of the slip ring are tightened to secure the rotating inner ring to the worm gear's extended shaft. The sun and moon shaft is assembled using a 1' long and 3/8" wide hollow rod. 3D printed mounts for the LED arrays are glued so the prongs are facing the mount to the ends. Holes are drilled so that wires can run from the slip ring, through an opening in the center of the rod, and connecting to the two circular light panels. Wires were fed through the metal tube using a string that pulled them through. The central rod is attached to a modified coupler that is capable of fitting over the 8mm rod and locking in place using set screws. The entire sun and moon assembly is mounted in the central gap in the wood, oriented so the motor's shaft points toward the back section of the display as shown in *Figure 24*.

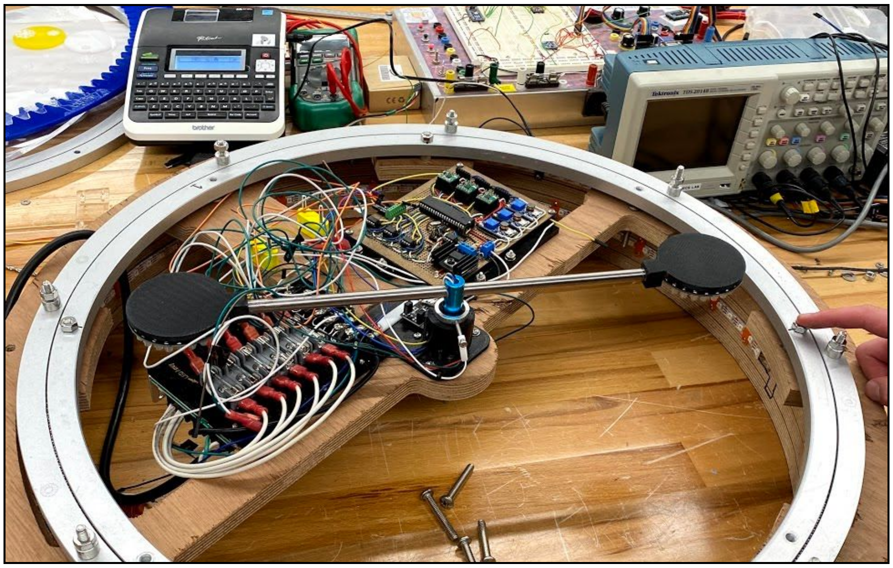


Figure 24: A view of the sun and moon system prior to attaching the weather sheets

Other pieces included on the center sheet are the mounts for the power supply, fuses, Raspberry Pi and circuit board as shown in *Figure 25.* Both of these mounts are acrylic sheets designed with holes that fit into the wooden center layer. One mount holds the fuses on one side and the external 5V, 18A power supply on the other, while the second mount has the Raspberry Pi and Propeller chip on opposite sides. The mounts were fixed to the middle so that the fuses (located on the right of the sun and moon when viewed from the back) and the Propeller chip and circuit board (located on the left) were facing the back, making it easier for any modifications to be made to the display in case they are needed.

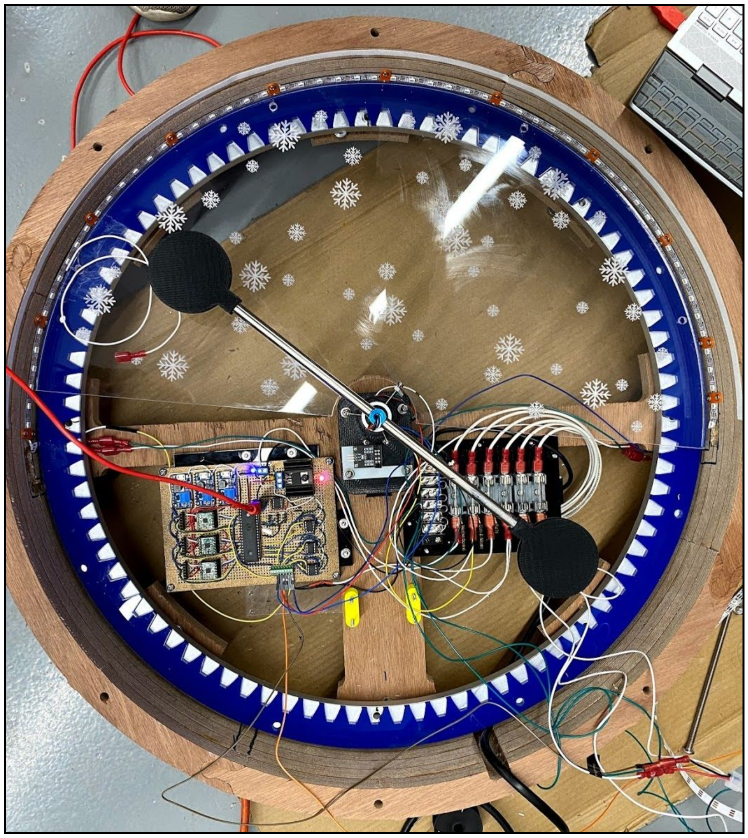


Figure 25: A view from the back of the sun and moon motor and the mounts for the electronics; the circuit board and Propeller chip are mounted on the left, and the fuses and power supply are on the right

The worm gears motors that moved the weather sheets were attached to the central layer using acrylic bars and 3D printed spacers. The spacers were used to ensure that the worm gears would be at the correct height to mate with the internal gears that were attached to the lazy susan bearings. The acrylic bars are what connected to the worm gears to the spacers. The acrylic bars had slots in one end to allow for lateral adjustment of the worm gear position to ensure that they would mate properly to the internal gears.

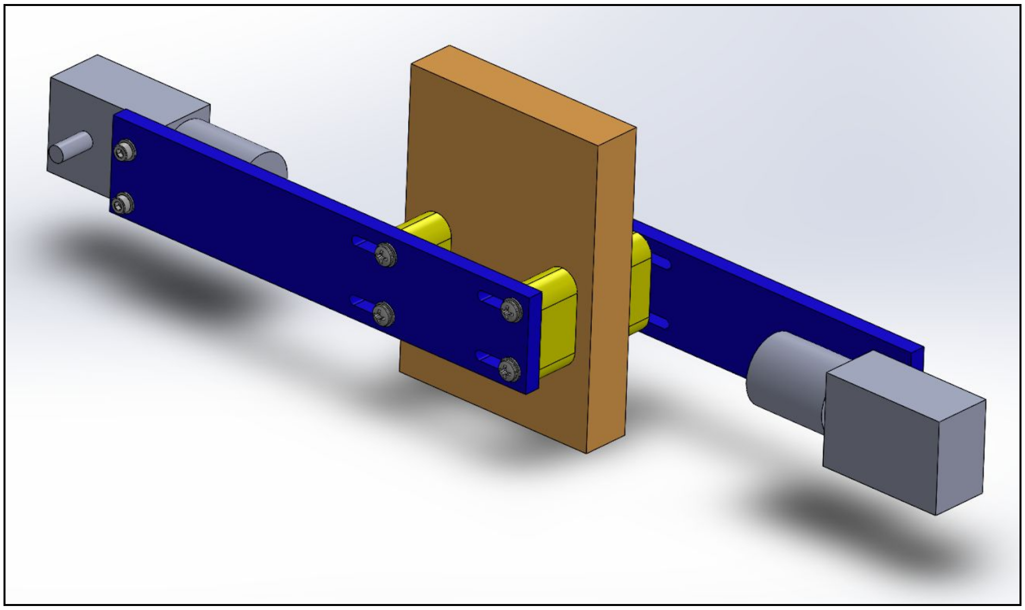


Figure 26: SolidWorks model of the worm gear motor mounts for the acrylic weather sheets, shown from the front side. The acrylic mounts are in blue, and the 3D printed spacers are in yellow. The small gears would be attached to the shafts on the worm gear motors.

Each worm gear motor in *Figure 26* above is held to the ends of the acrylic mounts using M3 x 16mm screws with one washer underneath the screw head. The holes inside the motors are threaded, and in order to position the gears perfectly in place, a nut is used as a spacer between the acrylic and the motor (not visible in the figure above). Without this spacer, the motors would be positioned too far out from the center frame, and the small gears would interfere with the acrylic weather sheets, so it would not be able to turn more than 180°. All four 3D printed spacers and two acrylic mounts are held in place and sandwiching the wooden center beam using four 3” 6-32 screws, with one washer under the screw head and one nut to hold it in place on the back side. These 3” screws have just enough clearance for the acrylic snow sheet to continue to rotate; however, the plan is to eventually replace these with 2.75” 6-32 screws in order to allow for even more clearance. Unfortunately due to time constraints, this length of screw was unable to be purchased and replaced in time.

One of the components of the final assembly which was not directly modeled in SolidWorks prior to implementation was the mounting of the three LED strips, which were used to light up the rain acrylic weather sheet, the snow acrylic weather sheet, and the background, respectively. The LED strips were accounted for when coming up with the final assembly by leaving a small gap between the wooden frame and where the internal components, such as the weather sheets, would lie. Therefore, the plan was to mount the LED strips to the inside ring of the wooden frame and align them correctly with the components that were to be lit up. In order to mount these LED strips properly, the correct position of the LED strips was sketched out on the wooden frame, measured from the distance from the center frame piece. Next, each LED strip was initially fastened with a strip of double-sided tape: this added support to the fasteners that would ultimately hold them in place as well as held the LED strips in place while the fasteners were being screwed in. After each LED strip was taped, holes were marked on either side of the LED strip where the screws would be fastened, and spaced out with 6 LEDs in between each fastener for a total of around 11 fasteners used for each LED strip. A detailed image of the fasteners 3D printed for this process is shown above in Figure 16. Then, the holes were drilled about ½" deep in order to fasten the 3D printed pieces with 4-40 ½" machine screws. The original plan was to drill in wood screws to mount the LEDs, but in testing this beforehand the wood screws caused the frame to split and were therefore replaced with machine screws. Finally, the fasteners were mounted by screwing each one by hand into the pre-drilled holes. Below is an image of two of the mounted LED strips in *Figure 27.*

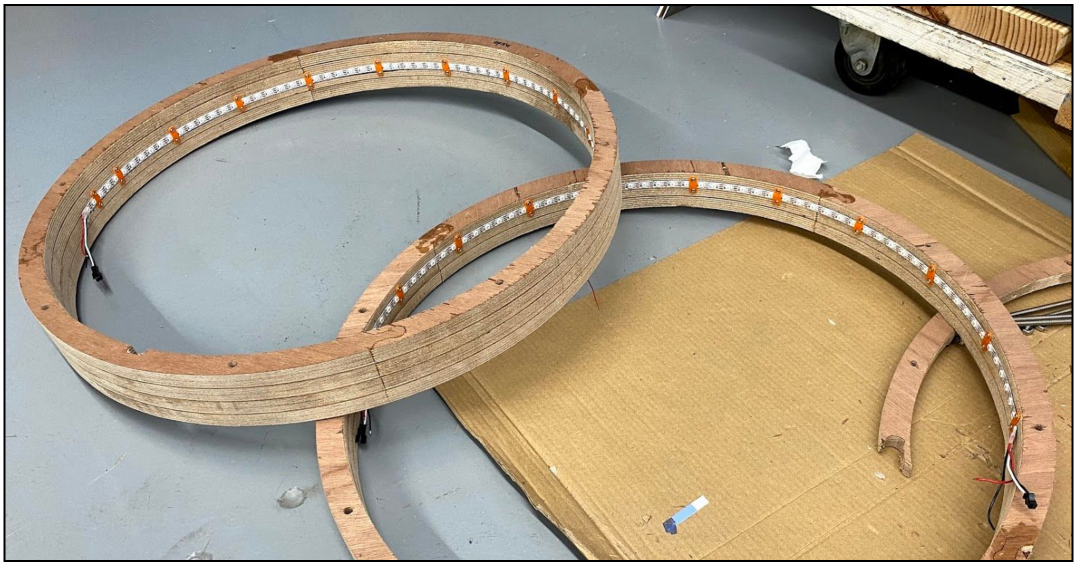


Figure 27: Two of the three mounted LED strips: these were used for the acrylic weather sheets

It is important to note that the background LED strip required the design of a different kind of fastener because it did not have enough room between the LED strip and the back frame piece to fasten with two screws. This new fastener only used one screw to hold in place, and it was also designed with a tab to hold the LED strip in place with friction when screwed in. Other than the new fastener design, the background LED strip was mounted the same way as those used for the weather sheets. A detailed image of this modeled in SolidWorks is provided below in *Figure 28.*

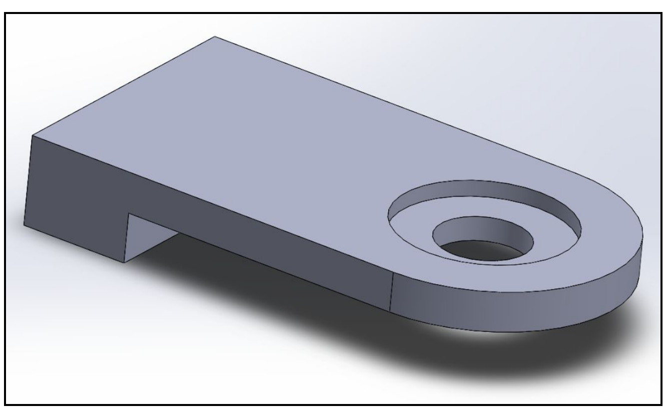


Figure 28: SolidWorks design of the background LED fasteners

The final assembly is composed of four distinct parts that are fastened together. The front piece is comprised of the viewing face, a full circle with an clear acrylic viewing window and an LED display for time and temperature, as well as three hollow wooden rings that are glued to the back of it. In the middle of the glued rings are eight ¼” 8-18 Tee Nuts that were hammered into the wood. This is connected to the center layer such that the rain layer is the closest to the front face. The third section is a stack of five wooden rings that have been glued together and aligned so that the eight holes line up with the other two sections. These fit to the back side of the center layer so that the snow layer lines up with the LED strip bolted to the inside ring. The power cord of the external power supply fits into the divot cut away from the side of the hoop. The backplate is the final layer: a half sheet of wood for the top semicircle and a hollow ring for the bottom. Holes are countersunk into the backplate for the bolt heads to fit in without sticking out from the back. This ensures that the weather display sits flat against the wall. Also on the backplate is a hollowed space where french cleats were attached. These slip over one another on the wall to hold the display in place. A pair of black lines on the bottom of these components line up together to show the proper orientation of the full assembly. When all the holes have lined up, eight 6” 8-18 bolts screw through the full assembly to fasten all four pieces together by locking into the Tee Nuts in the frontmost section. *Figure 29* and *Figure 30* show completed views of the final assembly with and without the front face attached.

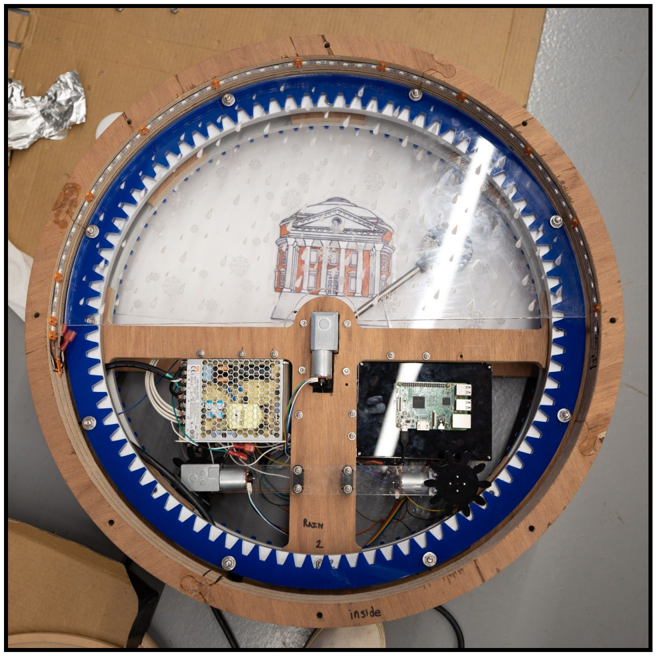


Figure 29: A view from the front showing the full assembly prior to gluing the front face to the front three rings

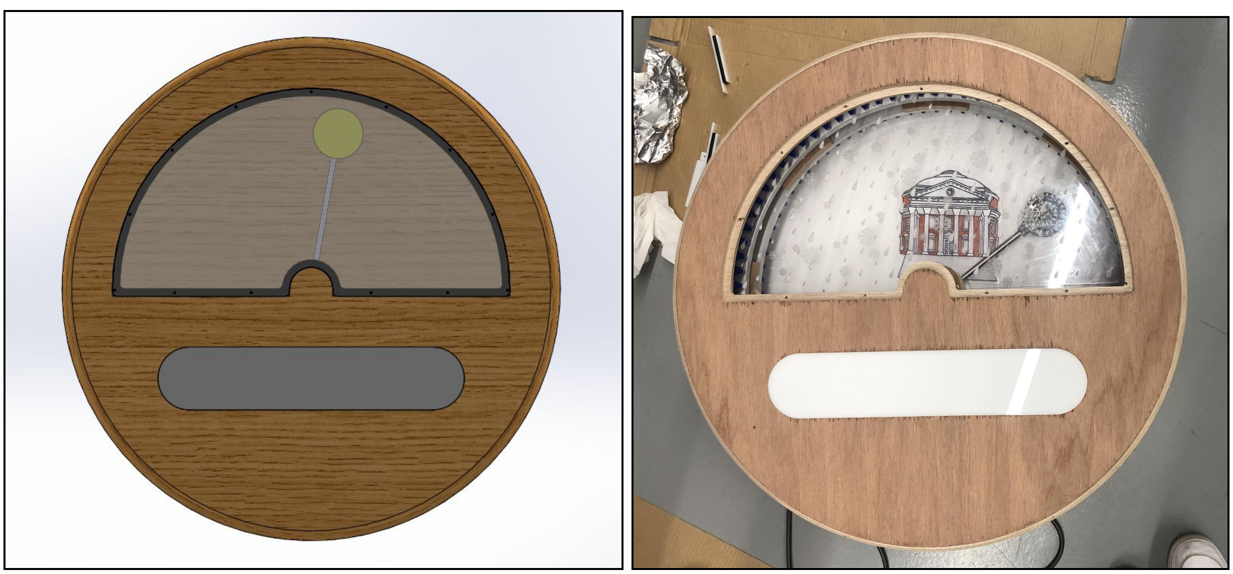


Figure 30: SolidWorks 3D Model (Left) and CNC cut (Right) images of the front face of the display

THE PROPELLER CHIP AND SPIN CODE

At the heart of the weather clock assembly is a Parallax Propeller chip. This microcontroller contains 8 parallel processors (COGs) capable of running methods simultaneously, independent of one another to manage and control 32 input and output pins. This chip was chosen for its simplicity, speed, precision, and familiarity. The Propeller chip controls all processes that occur within the weather clock except for the retrieval of time and weather data, which is completed by the Raspberry Pi and detailed in the following section. The methods described in this section and all line references correspond to the spin code in Appendix A.

The Setup method is the first method run on the Propeller chip (79-89). First, it sets the input/output state of each pin being used. It then sets the LED data pins low on COG 0 to prevent any ambiguity in the output of LED data pins when a COG controlling LEDs is stopped. Next, methods are run to position the acrylic rain and snow weather sheets to their down (not visible) position. Lastly, certain variables are initialized at values that will make every method that updates components run on the first iteration through the main loop.

The main method (90-175) is an infinite loop in charge of calling all necessary methods at appropriate times and retrieving data as needed. At the beginning of each iteration, the main method retrieves the current time from the Raspberry Pi using the GetTime method then updates the LEDs on the LED array to either display the current time or temperature, alternating between the two. Next, the timeTempFlag is flipped between 0 and 1 to indicate if the time or temperature will be shown on the LED array the following time through the loop. An if statement follows this to check if the time has reached a new minute mark, ensuring that all methods within the if statement are only run once per minute. If the minute has not changed, the loop waits 15 seconds and starts over. Every minute, the weather data is retrieved from the Raspberry Pi. Next, a series of if statements check if it is 12:01am, the time of sunrise, or the time of sunset to run methods specific to those times. The following if statements (121 & 123) check if the sun/moon should be rotated. If the variable indicating the current weather (weatherID) has changed, the appropriate

LEDs are set and acrylic weather sheets are positioned according to the type of weather. If the weatherID differs from any of the cases following line 131, the Raspberry Pi is either not powered on or not connected to the internet and the setup method is run, restarting the Propeller chip's code. Lastly, the background LEDs are set to special states for night or twilight if the time is appropriate for these states and the background LEDs are updated if the background has changed.

There are four different methods called within the main loop to rotate the gears that set the positions of the acrylic rain and snow sheets: RainGearUp (196), SnowGearUp (200), RainGearDown (204), and SnowGearDown (208). The RainGearUp method checks if the acrylic rain sheet is in the down position, and—if it is in fact down—runs the moveGear method with the appropriate pins input as parameters. This method simply rotates the rain sheet's worm gear motor 1530°, which corresponds to 180° of rotation for the hollow gear bolted to the rain sheet (see Go method, line 708). The SnowGearUp method is exactly the same as the RainGearUp method, but corresponds to the snow sheet worm gear's pins. The RainGearDown method checks if the acrylic rain sheet is in the up position, and—if it is up—runs the motorUntilSwitch method with the appropriate pins input as parameters. Rather than rotating to a specified angle, this method applies power to the worm gear motor until an infrared sensor switch is activated by a thin piece of metal glued to the acrylic sheet (see GoUntil method, line 701). The SnowGearDown method is the same as the RainGearDown method, but corresponds to the snow sheet worm gear's pins and a separate infrared sensor for the snow sheet.

The sun and moon rotate 20 times per day: 10 times during the day and 10 times during the night. The frequency of rotation is dictated by the sunrise and sunset time so that the sun enters the visible frame at sunrise and exits at sunset. Within the main loop, two if statements check if the current time is between the time of sunrise and sunset (daytime) and if the current time is a multiple of the daytime rotation frequency (123). If so, the moveSunMoon method is run to rotate the worm gear controlling the sun and moon 18°. A separate if statement does the same process for night and the nighttime rotation frequency (121). Once per day, at sunrise, the runMotorUntilSwitch is run on the sun/moon worm gear motor with an optical limit switch to home the sun and moon to a position where the bar to which the sun and moon are attached is horizontal.

All LEDs within the weather clock receive data through WS2812B protocol and are thus controlled using a driver file written by Professor Gavin Garner (object added in line 63, full driver code in Appendix D). This driver requires a new COG to control LEDs. However, because the rain and snow LED strips are the only LEDs that are run and changed at the same time as other processes occurring in the main loop, and because these two LED strips will never run at the same time as each other, they may share one driver and all other LEDs may share another driver. The use of two separate drivers is indicated by the fact that rgb object in line 63 is actually an array of two LED driver objects. If it is raining or snowing, the RainLEDs method (212) or the SnowLEDs method (447) is started on a new COG. This allows the quick changes to different LED patterns and colors to simulate rainfall or snowfall. The RainLEDs method contains 15 different LED pattern sequences depending on the type of rain (light rain, heavy thunderstorm, etc.), and the SnowLEDs method contains eight LED pattern sequences depending on the type of snowfall. The background LED strip, which is changed on COG 0 when the main loop calls the backgroundLEDs method (629) contains LED patterns for 6 different cases: night, twilight, clear/sunny, overcast, partly cloudy, and mostly cloudy.

The LEDs on the Adafruit Triple-Ring boards light up to indicate the sun or moon. At sunrise and sunset, flags indicating the current state of the sun and moon LEDs are set accordingly and the SunLEDs and MoonLEDs methods (176, 186) are run to update the LEDs. These methods either turn all LEDs off, turn the sun LEDs yellow, or turn the moon LEDs white depending on the status of the LED state flag. At sunrise, the moonLEDs method is run first to turn off the moon LEDs before turning on the sun LEDS to prevent both LED boards from being on at the same time. Likewise, the sunLEDs method is run before the moonLEDs method at sunset.

Lines 760 to 928 of the spin code are all methods for the LED array on the front display. These methods draw numbers, colons, and degree symbols based on specified x- and y-coordinates and colors. These methods were purposefully made general so that future work can utilize these methods to potentially make a more animated front display. The LEDArray method (607) starts an LED driver and calls the drawTime or drawTemp methods depending on the aforementioned timeTempFlag before stopping the LED driver. One important challenge in coding the LED array is that, because the LEDs were being updated so often when drawing numbers, the driver sometimes could not keep up and LEDs would not light up as they were supposed to. For this reason, the LED driver was modified to have a separate method to update the LEDs so that all information on which LEDs would light up could be sent to the LEDs less often.

The process of getting data from the Raspberry Pi is detailed in the following section, but one important aspect of the spin code is the way of translating bytes of ASCII characters to integer data. Because all data received is numerical, and the number of digits is always known, the information is first stored byte by byte in an array. This process can be seen in lines 574 and

575. Next, each byte has 48 subtracted from it (numbers start at 48 in ASCII), then each digit is multiplied by a power of 10 corresponding to its location in the array and added to the variable the data should be set to. An example of this process can be seen in lines 578 to 583

Although the eight parallel processors in the Propeller chip is a lot to work with, it is very important that the code does not attempt to use more COGs than are available. For this reason, COGs 5 to 7 are reserved for specific processes and a maximum of two other COGs (LED drivers or the serial reader) will ever be used at once (in addition to COG 0 running the main code). This maximum of 6 COGs being used at once also allows for two more processors to be used should future additions be introduced to the weather clock.

Future work regarding the coding of the weather clock will be taking place over the following months to ensure the code works as it should and all components are able to function together. Code has been and will continue to be commented heavily to ensure future modifications and improvements can be made as effortless as possible.

RASPBERRY PI CODE AND WEATHER API CONNECTION

The Propeller chip used to dictate the mechanical and electrical aspects of the weather clock is very powerful, but lacks the ability to access the internet. However, the weather clock needs a way to access a weather API to retrieve accurate, real time weather data. This need led to the implementation of a Raspberry Pi 3 Model B. Additionally, because the Raspberry Pi will be connected to the internet, it will also be responsible for giving time information to the Propeller chip.

The Raspberry Pi runs Python code that utilizes the built-in “datetime” library for time data and the “requests” library to request and receive a JSON file containing weather data from

OpenWeather's "Current Weather Data" API (OpenWeatherMap, 2020). Once this data is received on the Raspberry Pi, it is transmitted to the Propeller chip using UART protocol, which requires only three pins: TX (data transmission, Raspberry Pi pin 8), RX (data reception, Raspberry Pi pin 10), and a common ground. The Raspberry Pi sends serial data to the Propeller chip using the "write" method which is part of the built-in "serial" library (Appendix E). The Propeller chip uses the "FullDuplexSerial" built-in file as an object to receive and interpret the weather and time data. Rather than using the standard "Rx" method, which waits for data to appear in the queue and returns that data, the "RxTime" method was used. The "RxTime" method differs from the "Rx" method in that it takes in a parameter for the specified amount of time (in milliseconds) that the method should wait for data to appear in the queue. If no data appears in the queue within the allotted time, the method returns -1 and the code moves on. This way, if communication between the Propeller chip and the Raspberry Pi was cut for some reason (such as a power outage), the Propeller's spin code would not get stuck waiting for data to be received.

The Propeller sends data requests to the Raspberry Pi from its TX pin to the RX pin of the Raspberry Pi. These data requests are sent as strings such as "GetTime," "GetSunset," and "GetWeatherData." When the Raspberry Pi receives these requests, it either uses the "datetime" library to get the time or sends a request to the API for weather data and sends this data from the Raspberry Pi TX pin to the Propeller chip RX pin. For the sake of simplicity and so that the number of bytes of data sent to the Propeller is known, all data used is numerical. For example, the weather description is sent as a three digit number (800 is clear, 230 is thunderstorm with light drizzle, etc.).

The OpenWeather API was chosen for two primary reasons: it's expansive and accurate weather database and the fact that it will be free. However, in order to use the API for free, it may only be called 60 times per minute. This frequency of weather data collection is no problem for the weather clock project as long as the API is not overused. To balance the frequency of data requests with the need for the Propeller chip to quickly retrieve up-to-date data, the Python code only sends requests to the API when the Propeller chip requests data from the Raspberry Pi. The communication of requests and data transmission is shown in *Figure 31* below.

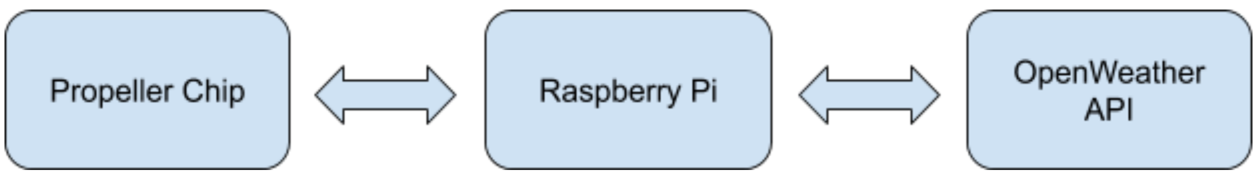


Figure 31: The path of weather data and requests for real time weather data

One of the important elements to implement in the weather communication setup was to make sure that the Raspberry Pi would automatically run the Python code to get the weather data upon starting up. This way, if power ever cut out, the Raspberry Pi would only need to be powered on to restart the code rather than connecting a keyboard, monitor, and mouse to run the script. This was accomplished by editing the rc.local file on the Raspberry Pi and adding a command to run the Python code (Hawkins, 2015).

FINAL PRODUCT

The weather clock, in its entirety, hangs approximately 7.6 inches off a wall, with a diameter of 26 inches. The weather clock is capable of showcasing multiple weather patterns. The Raspberry Pi allows for an API to constantly be referenced and update the clock display according to the input received from OpenWeather. In the center of the display is the sun and moon system. Based on the sunrise and sunset times received from the API, the sun will move across the display accordingly. The moon is on the opposite side of the rod that the sun is on, and will move into the display when the sun is out. Both the sun and moon have LED panels for added effect. If it is raining or snowing, the Propeller will tell the worm gears to move the rain or snow sheet up into the display of the clock. Optical limit switches ensure that the acrylic pieces stop at the correct positions. For thunder, the LED on the edge of the clock will flash lights to mimic a thunderstorm. The LEDs also serve to show what the sky would look like, differentiating between day and night, and also showing if it's cloudy or foggy. Additionally, there is an LED display on the bottom of the clock that shows the current time and temperature using the same API. Lastly, if the power to the clock were to ever go out, there is automatic reset and reboot built into the Propeller.

The current state of the weather clock is the final stages of completion. All the parts and mounts are installed along with the circuit board, power supply, and wiring. Further debugging of the programming running the weather clock needs to occur to ensure that it can operate independently for display in the Mechanical & Aerospace building.

KEY TAKEAWAYS

Throughout the design and implementation process, one of the most important tools was the SolidWorks program. By building the clock first in SolidWorks, the various pieces could be placed in an assembly of the clock to ensure that they would be able to fit or be able to perform any actions required. Many of the issues that came up during the building process stemmed from not having the part in the SolidWorks model. For one, the length of the sun and moon assembly once created outside of the model proved to be too long for the CNC layers that were made to house the mechanics. As a result, another layer had to be cut at the last minute to give enough space for it to fit. Had the assembly been properly modeled originally, this last minute adjustment would not have been necessary. Another important takeaway was to avoid the use of glue if possible, as glue can hinder any necessary re-design, versus using screws which can be easily removed.

Additionally, it was difficult to foresee how heavy the assembly would be through the SolidWorks model. This became an issue in implementing the worm gear motors to move the acrylic weather sheets because the original motors used did not have enough torque to rotate the spur gear. The first attempt to fix this problem was to decrease the size of the smaller gear in order to increase the torque--and in doing so, the mounts for the worm gear motors had to be redesigned in order to move the motors closer to the spur gear. However, this still wasn't enough torque to move the spur gear, so ultimately in order to fix this problem the transmissions on the motors had to be switched. This could have been accounted for by taking a closer look at the SolidWorks model and estimating the potential weight of specific parts of the assembly.

FUTURE CONSIDERATIONS

The creation of this weather clock dives into the custom-design business sector of craftsmanship and interior design. Many company offices as well as homeowners look for custom-made pieces to display in their space. If consumers wanted a similar product with or without their own personalizations, significant internal changes would need to be made to the weather clock design and build for it to be marketable. First, the materials should be reconsidered to minimize cost as well as weight. Bulk ordering of parts would most likely help with cost reductions. The weight of the current design could potentially be addressed by using a less dense wood for the frame and the thickness could be reduced by machining the internal spur gears out of aluminum and then building tracks into the frame for them to rotate on, thus eliminating the need for the lazy susan bearings. More time should also be spent on the build process to make assembling and disassembling easier.

References

Hawkins, M. (2015). How To Autorun A Python Script On Raspberry Pi Boot. *RaspberryPiSpy*. https://www.raspberrypi-spy.co.uk/2015/02/how-to-autorun-a-python-script-on-raspberry-pi-boot/

Garner, G. (2020). *Exploring Mechatronics: Spring 2020*. University of Virginia.

Garner, G. (2020). *Garner's Guide to 3D Printing Parts on the Student UPrints.* University of Virginia.

Garner, G. (2020). *How to Use HSMworks and the Shopbot PRSalpha 96-48 CNC Router.* University of Virginia.

Garner, G. (2020). *The MILL's Laser Cutter Tutorial.* University of Virginia.

Martin, J. (2011). *Propeller Manual Version 1.2*. Parallax Inc.

OpenWeatherMap (2020). Current weather data. https://openweathermap.org/current

Raspberry Pi Documentation. (2020). Retrieved December 01, 2020, from https://www.raspberrypi.org/documentation/

Appendix A: Propeller Chip Spin Code as of 12/01/2020

```
CON
  _xinfreq=6_250_000          'The system clock is set at 100MHz (you need at least a 20MHz system clock)
  _clkmode=xtal1+pll16x

  rotationIncrements=30
  sunMoonLEDnum=44
  LEDArrayNum=256
  TotalLEDs=60
  xLEDs=32
  yLEDs=8
  encoderSteps=14544                 'number of encoder steps for one full rotation for the 15rpm wormgear motors
  sunMoonEncoderSteps=2745           'number of encoder steps for one full rotation for the 100rpm wormgear motor

  weatherLEDCOG=5
  PWMCOG=6
  encoderCOG=7

  rxPin=2
  txPin=3
  sunLEDpin=20
  moonLEDpin=21
  sunMoonDir=4
  sunMoonPWMpin=5
  sunMoonEncA=6
  sunMoonEncB=7
  rainDir=8
  rainPWMpin=9
  rainEncA=10
  rainEncB=11
  snowDir=12
  snowPWMpin=13
  snowEncA=14
  snowEncB=15
  rainLEDpin=16
  snowLEDpin=17
  backgroundLEDpin=18
  LEDArrayPin=19
  sunMoonOptLimSwitchPin=24
  rainOptLimSwitchPin=25
  snowOptLimSwitchPin=26
```

```

  red           = 255<<8                     '%00000000_11111111_00000000
  green         = 255<<16                    '%11111111_00000000_00000000
  blue          = 255                        '%00000000_00000000_11111111
  white         = 255<<16+255<<8+255         '%11111111_11111111_11111111
  cyan          = 255<<16+255                '%11111111_00000000_11111111
  magenta       = 255<<8+255                 '%00000000_11111111_11111111
  yellow        = 255<<16+255<<8             '%11111111_11111111_00000000
  chartreuse    = 255<<16+127<<8             '%11111111_01111111_00000000
  orange        = 60<<16+255<<8              '%10100101_11111111_11010100
  aquamarine    = 255<<16+127<<8+212         '%11111111_11111111_11010100
  pink          = 128<<16+255<<8+128         '%10000000_11111111_10000000
  turquoise     = 224<<16+63<<8+192          '%10000000_00111111_10000000
  realwhite     = 255<<16+200<<8+255         '%11100000_11001000_11000000
  indigo        = 170                        '%00000000_00111111_01111111
  violet        = 51<<16+215<<8+255          '%01111111_10111111_10111111
  grey          = 128<<16+128<<8+128         '%10000000_10000000_10000000
  darkgrey      = 169<<16+169<<8+169         '%
  nightsky      = 12<<16+20<<8+69

OBJ
  ser : "FullDuplexSerial"
  rgb[2]: "WS2812B_RGB_LED_Driver_v2.1"            'rgb[0] is reserved for rain/snow LEDs, rgb[1] is for all other LEDs

VAR
  byte currentWeather[30], rainPosition, snowPosition, info[10], sunLEDstate, moonLEDstate, timeTempFlag
  long weatherID, lastWeatherID, temp, sunrise, sunset, daylight, backgroundID, lastBackgroundID, dayRotFreq, nightRotFreq
  long weatherLEDCOGstack[100], PWMstack[10], encoderStack[100]
  long year, month, day, hour, minute, currentTimeMin, lastMinute
  long DutyCycle, position, target                                       'variables for moving motors

  'backgroundID codes:
  '0: night
  '1: twilight
  '2: clear/sunny
  '3: overcast
  '4: partly cloudy
  '5: mostly cloudy
PUB Setup
  DIRA := %00000000_00001111_00110011_00111110            'set which pins are outputs and which are inputs
```

```
  outa[16..21]~
  RainGearDown                      'start rain and snow sheets positioned down
  SnowGearDown
  backgroundID := 8             'guarantee the BackgroundLEDs method will be run on the first go through
  lastBackgroundID := 8         'start at 8 to ensure lastBackgroundID != backgroundID on first run through Main loop
  lastWeatherID := 0            'start at 0 to ensure lastWeatherID != weatherID on first run through Main loop
  timeTempFlag:=0               '0 indicates time should be displayed, 1 indicated temperature should be displayed
  lastMinute:=-1                'start at -1 to ensure minute != lastMinute on first run through Main loop

PUB Main
  repeat
    GetTime                                                                     'runs about every 15 seconds
    LEDArray
    if timeTempFlag == 0
      timeTempFlag := 1
    else
      timeTempFlag := 0

    if lastMinute <> minute                                                     'only run these methods once per minute
      lastMinute := minute
      GetWeather
      'Daily to-dos
      if hour == 0 AND minute == 1
        GetSunrise
        daylight := sunset - sunrise
        dayRotFreq := daylight / rotationIncrements
        nightRotFreq := (1440 - daylight) / rotationIncrements
      'sunrise to-dos
      if currentTimeMin == sunrise
        sunLEDstate := 1
        moonLEDstate := 0
        MoonLEDs
        SunLEDs
        motorUntilSwitch(sunMoonPWMpin, sunMoonDir, sunMoonOptLimSwitchPin)
      'sunset to-dos
      if currentTimeMin == sunset
        sunLEDstate := 0
        moonLEDstate := 1
        SunLEDs
        MoonLEDs
      if (currentTimeMin <= sunrise OR currentTimeMin > sunset) AND (currentTimeMin//nightRotFreq == 0)
        moveSunMoon(sunMoonPWMpin, sunMoonEncA, sunMoonEncB, sunMoonDir)
      if (currentTimeMin > sunrise AND currentTimeMin <= sunset) AND (currentTimeMin//dayRotFreq == 0)
        moveSunMoon(sunMoonPWMpin, sunMoonEncA, sunMoonEncB, sunMoonDir)

      if weatherID <> lastWeatherID
        if rainPosition == 1 OR snowPosition == 1
          rgb[0].AllOff
          cogstop(weatherLEDCOG)
        lastWeatherID := weatherID
        case weatherID
          200..599:                 '200-299 Thunderstorm, 300-399 Drizzle, 500-599 Rain
            backgroundID:=3
            SnowGearDown
            RainGearUp
            coginit(weatherLEDCOG, RainLEDs, weatherLEDCOGstack)
          600..699:                 'Snow                   Run SnowGear on a COG then stop that COG and run SnowLEDs on it
            backgroundID:=3
            RainGearDown
            SnowGearUp
            coginit(weatherLEDCOG, SnowLEDs, weatherLEDCOGstack)
          700..799:                 'Atmosphere (just assume fog or mist, doubtful we will run into volcanic ash)
            backgroundID:=6         'overcast
            RainGearDown
            SnowGearDown
          800, 801:                 'Clear/ very few clouds
            backgroundID:=2
            RainGearDown
            SnowGearDown
          802:                      'partly cloudy
            backgroundID:=4
            RainGearDown
            SnowGearDown
          803:                      'mostly cloudy
            backgroundID:=5
            RainGearDown
            SnowGearDown
          804:                      'overcast
            backgroundID:=3
            RainGearDown
```

```
            SnowGearDown
          OTHER:
            Setup

      if currentTimeMin < sunrise or currentTimeMin > sunset
        backgroundID:=0                                  'night
      elseif currentTimeMin > (sunset-30)
        backgroundID:=1                                  'twilight

      if backgroundID <> lastBackgroundID
        lastBackgroundID:=backgroundID
        BackgroundLEDs
    else                                                 'if it isn't a new minute, wait 15 seconds and check the time again
      waitcnt(clkfreq*15 + cnt)                          'this is also the amount of time it takes the LEDArray to switch between
                                                         'time and temperature
PUB SunLEDs | i
  rgb[1].start(sunLEDpin, sunMoonLEDnum)
  if sunLEDstate == 1
    repeat i from 0 to (sunMoonLEDnum - 1)
      rgb[1].LED(i, yellow)
    rgb[1].updateLEDs
  else
    rgb[1].AllOff
  rgb[1].stop

PUB MoonLEDs | i
  rgb[1].start(moonLEDpin, sunMoonLEDnum)
  if moonLEDstate == 1
    repeat i from 0 to (sunMoonLEDnum - 1)
      rgb[1].LED(i, realwhite)
    rgb[1].updateLEDs
  else
    rgb[1].AllOff
  rgb[1].stop

PUB RainGearUp
  if rainPosition == 0
    moveGear(rainPWMpin, rainEncA, rainEncB, rainDir)
  rainPosition := 1
PUB SnowGearUp
  if snowPosition == 0
    moveGear(snowPWMpin, snowEncA, snowEncB, snowDir)
  snowPosition := 1
PUB RainGearDown
  if rainPosition == 1
    motorUntilSwitch(rainPWMpin, rainDir, rainOptLimSwitchPin)
  rainPosition := 0
PUB SnowGearDown
  if snowPosition == 1
    motorUntilSwitch(snowPWMpin, snowDir, snowOptLimSwitchPin)
  snowPosition := 0
PUB RainLEDs | i, j
  rgb[0].start(rainLEDpin, TotalLEDs)
  case weatherID
    200, 210:                                  'thunderstorm with light rain
      repeat
        repeat 5
          repeat i from 0 to 4
            rgb[0].AllOff
            repeat j from i to TotalLEDs step 9
              rgb[0].LED(j, blue)
            rgb[0].UpdateLEDs
            waitcnt(clkfreq/3 + cnt)
            rgb[0].AllOff
            repeat j from i to TotalLEDs step 9
              rgb[0].LED(j + 5, blue)
            rgb[0].UpdateLEDs
            waitcnt(clkfreq/3 + cnt)
        lightning

    201, 211:                                  'moderate thunderstorm
      repeat
        repeat 5
          repeat i from 0 to 4
            rgb[0].AllOff
            repeat j from i to TotalLEDs step 7
              rgb[0].LED(j, blue)
            rgb[0].UpdateLEDs
            waitcnt(clkfreq/3 + cnt)
            rgb[0].AllOff
```

```
        repeat j from i to TotalLEDs step 7
          rgb[0].LED(j + 3, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/3 + cnt)
    lightning

202, 212, 221:                                       'heavy thunderstorm
  repeat
    repeat 7
      repeat i from 0 to 2
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 5
          rgb[0].LED(j, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/3 + cnt)
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 5
          rgb[0].LED(j + 3, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/3 + cnt)
    lightning

230:                                                 'thunderstorm with light drizzle
  repeat
    repeat 3
      repeat i from 0 to 4
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 9
          rgb[0].LED(j, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/2 + cnt)
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 9
          rgb[0].LED(j + 5, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/2 + cnt)
    lightning

231:                                                 'thunderstorm with drizzle
  repeat
    repeat 3
      repeat i from 0 to 4
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 7
          rgb[0].LED(j, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/2 + cnt)
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 7
          rgb[0].LED(j + 3, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/2 + cnt)
    lightning

232:                                                 'thunderstorm with heavy drizzle
  repeat
    repeat 5
      repeat i from 0 to 2
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 5
          rgb[0].LED(j, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/2 + cnt)
        rgb[0].AllOff
        repeat j from i to TotalLEDs step 5
          rgb[0].LED(j + 3, blue)
        rgb[0].UpdateLEDs
        waitcnt(clkfreq/2 + cnt)
    lightning

300, 310:                                            'light intensity drizzle
  repeat
    repeat i from 0 to 4
      rgb[0].AllOff
      repeat j from i to TotalLEDs step 9
        rgb[0].LED(j, blue)
      rgb[0].UpdateLEDs
      waitcnt(clkfreq/2 + cnt)
      rgb[0].AllOff
      repeat j from i to TotalLEDs step 9
```

```
            rgb[0].LED(j + 5, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)

    301, 311:                                              'drizzle
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)

    302, 312, 313, 314, 321:                               'heavy intensity drizzle
      repeat
        repeat i from 0 to 2
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)

    500:                                                   'light rain
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/3 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j + 5, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/3 + cnt)

    501, 511:                                              'moderate rain/freezing rain
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/3 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/3 + cnt)

    502..504:                                              'heavy rain
      repeat
        repeat i from 0 to 2
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/3 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/3 + cnt)

    520:                                                   'light intensity shower rain
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j, blue)
```

```
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j + 5, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

    521:                                          'shower rain
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

    522, 531:                                     'heavy shower rain
      repeat
        repeat i from 0 to 2
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

PUB lightning | i
  rgb[0].AllOff
  repeat 2
    repeat i from 0 to TotalLEDs step (TotalLEDs/10)
      rgb[0].LED(i, realwhite)
    rgb[0].UpdateLEDs
    waitcnt(clkfreq/16 + cnt)
    rgb[0].AllOff
    waitcnt(clkfreq/16 + cnt)

PUB SnowLEDs | i, j
  rgb[0].start(snowLEDpin, TotalLEDs)
  case weatherID
    600:                                          'light snow
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j + 5, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)

    601:                                          'snow
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j + 3, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)

    602:                                          'heavy snow
      repeat
        repeat i from 0 to 2
```

```
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j + 3, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/2 + cnt)

    620, 612:                                               'light shower snow/sleet
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j + 5, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

    621, 611:                                               'shower snow/sleet
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j + 3, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

    622, 613:                                               'heavy shower snow/sleet
      repeat
        repeat i from 0 to 2
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 5
            rgb[0].LED(j + 3, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

    615:                                                    'light rain and snow
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 9
            rgb[0].LED(j + 5, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)

    616:                                                    'rain and snow
      repeat
        repeat i from 0 to 4
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j, realwhite)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
          rgb[0].AllOff
          repeat j from i to TotalLEDs step 7
            rgb[0].LED(j + 3, blue)
          rgb[0].UpdateLEDs
          waitcnt(clkfreq/4 + cnt)
```

```
PUB GetWeather | i                                            'sets variable for weatherID
  ser.start(rxPin, txPin, 0, 9600)                            'start serial driver on new cog
  ser.str(String("getID"))                                    'send request to raspberry pi
  repeat i from 0 to 2                                        'grab serial data from the queue and add it to the info array
    info[i] := ser.rxTime(1000)
  weatherID := (100 * (info[0]-48)) + (10 * (info[1]-48)) + info[2] - 48        'turn info array's data into an integer
  ser.stop                                                    'stop the cog

PUB GetSunrise | i, power                                     'gets the sunrise/sunset time for the day in minutes (0-1439)
  'fetch sunrise and sunset time in unix
  ser.start(rxPin, txPin, 0, 9600)                            'start serial driver on new cog
  ser.str(String("getSunrise"))                               'send request to raspberry pi
  repeat i from 0 to 19                                       'grab serial data from the queue and add it to the info array (unix time is 10 digits)
    info[i] := ser.rxTime(1000)
  ser.stop                                                    'stop the cog

  sunrise := 0                                                'turn info array's data into an integer
  repeat i from 0 to 9
    power:=1
    repeat i
      power *= 10
    sunrise += ((info[9-i]-48) * power)

  sunset := 0
  repeat i from 0 to 9
    power:=1
    repeat i
      power *= 10
    sunset += ((info[19-i]-48) * power)

  sunrise:=((sunrise//86400)/60)                              'convert sunrise time from unix to minutes through the day
  sunset:=((sunset//86400)/60)                                'convert sunset time from unix to minutes through the day

PUB GetTime | i
  'fetch unix time
  ser.start(rxPin, txPin, 0, 9600)                            'start serial driver on new cog
  ser.str(String("getTime"))                                  'send request to raspberry pi
  repeat i from 0 to 3                                        'grab serial data from the queue and add it to the info array (unix time is 10 digits)
    info[i] := ser.rxTime(1000)
```

```
  ser.stop                                                    'stop the cog

  hour:= (10*(info[0]-48)) + info[1]-48                       'time hour
  minute:= (10*(info[2]-48)) + info[3]-48                     'time minute
  currentTimeMin:= (60*hour) + minute                         'convert time to minutes through day (0-1439)

PUB LEDArray
  rgb[1].start(LEDArrayPin, LEDArrayNum)
  rgb[1].AllOff
  if timeTempFlag == 0
    if hour > 9
      drawTime(5, 1, red)
    else
      drawTime(10, 1, red)
  else
    if temp < 10
      drawTemp(12, 1, blue)
    elseif temp < 50
      drawTemp(10, 1, blue)
    elseif temp < 76
      drawTemp(10, 1, green)
    elseif temp < 100
      drawTemp(10, 1, orange)
    else
      drawTemp(7, 1, red)
  rgb[1].updateLEDs
  rgb[1].stop

PUB BackgroundLEDs | i
  rgb[1].start(backgroundLEDpin, TotalLEDs)
  'set LEDs based on cases below
  case backgroundID
    0:  'night
      repeat i from 0 to 59
        rgb[1].LEDint(i, nightsky, 180)
      rgb[1].updateLEDs
    1:  'twilight
      rgb[1].LED(0, yellow)
      rgb[1].LED(1, yellow)
      rgb[1].LED(58, yellow)
```

```
      rgb[1].LED(59, yellow)
      repeat i from 2 to 6
        rgb[1].LED(i, orange)
      repeat i from 53 to 57
        rgb[1].LED(i, orange)
      repeat i from 7 to 11 step 2
        rgb[1].LED(i, red)
      repeat i from 8 to 12 step 2
        rgb[1].LED(i, pink)
      repeat i from 48 to 52 step 2
        rgb[1].LED(i, red)
      repeat i from 47 to 51 step 2
        rgb[1].LED(i, pink)
      repeat i from 13 to 20
        rgb[1].LED(i, magenta)
      repeat i from 39 to 46
        rgb[1].LED(i, magenta)
      repeat i from 21 to 38
        rgb[1].LED(i, indigo)
      rgb[1].updateLEDs
    2:  'clear/sunny
      repeat i from 0 to 59
        rgb[1].LED(i, turquoise)
      rgb[1].updateLEDs
    3:  'overcast
      repeat i from 0 to 59
        rgb[1].LEDint(i, grey, 100)
      rgb[1].updateLEDs
    4:  'partly cloudy
      repeat i from 0 to 59
        rgb[1].LEDint(i, turquoise, 150)
      rgb[1].updateLEDs
    5:  'mostly cloudy
      repeat i from 0 to 59
        rgb[1].LEDint(i, turquoise, 80)
      rgb[1].updateLEDs
  rgb[1].stop

................METHODS TO MOVE MOTORS......................................
PUB moveSunMoon(PWMpin, encoderAPin, encoderBPin, dirPin)
  coginit(PWMCOG,PWM(PWMpin),@PWMStack)
  coginit(EncoderCOG,Encoder(encoderAPin, encoderBPin),@EncoderStack)
  target:=(sunMoonEncoderSteps/20)
  Go(dirPin)
  cogstop(PWMCOG)
  cogstop(EncoderCOG)

PUB moveGear(PWMpin, encoderAPin, encoderBPin, dirPin)
  coginit(PWMCOG,PWM(PWMpin),@PWMStack)
  coginit(EncoderCOG,Encoder(encoderAPin, encoderBPin),@EncoderStack)
  target:=(17*encoderSteps/4)
  Go(dirPin)
  cogstop(PWMCOG)
  cogstop(EncoderCOG)

PUB motorUntilSwitch(PWMpin, dirPin, switchPin)
  coginit(PWMCOG,PWM(PWMpin),@PWMStack)
  GoUntil(dirPin, switchPin)
  cogstop(PWMCOG)

PUB GoUntil(dirPin, switchPin)
    outa[dirPin]~~
    repeat until ina[switchPin] == 0
      DutyCycle:=100
    DutyCycle~
    waitcnt(clkfreq/200+cnt)

PUB Go(dirPin)
  repeat 3
    if position=<target
      outa[dirPin]~~
      repeat until position=>target
        DutyCycle:=||(position-target)/8 #>80 <#100
    else
      outa[dirPin]~
      repeat until position=<target
        DutyCycle:=||(position-target)/8 #>80 <#100
    DutyCycle~
    waitcnt(clkfreq/200+cnt)
```

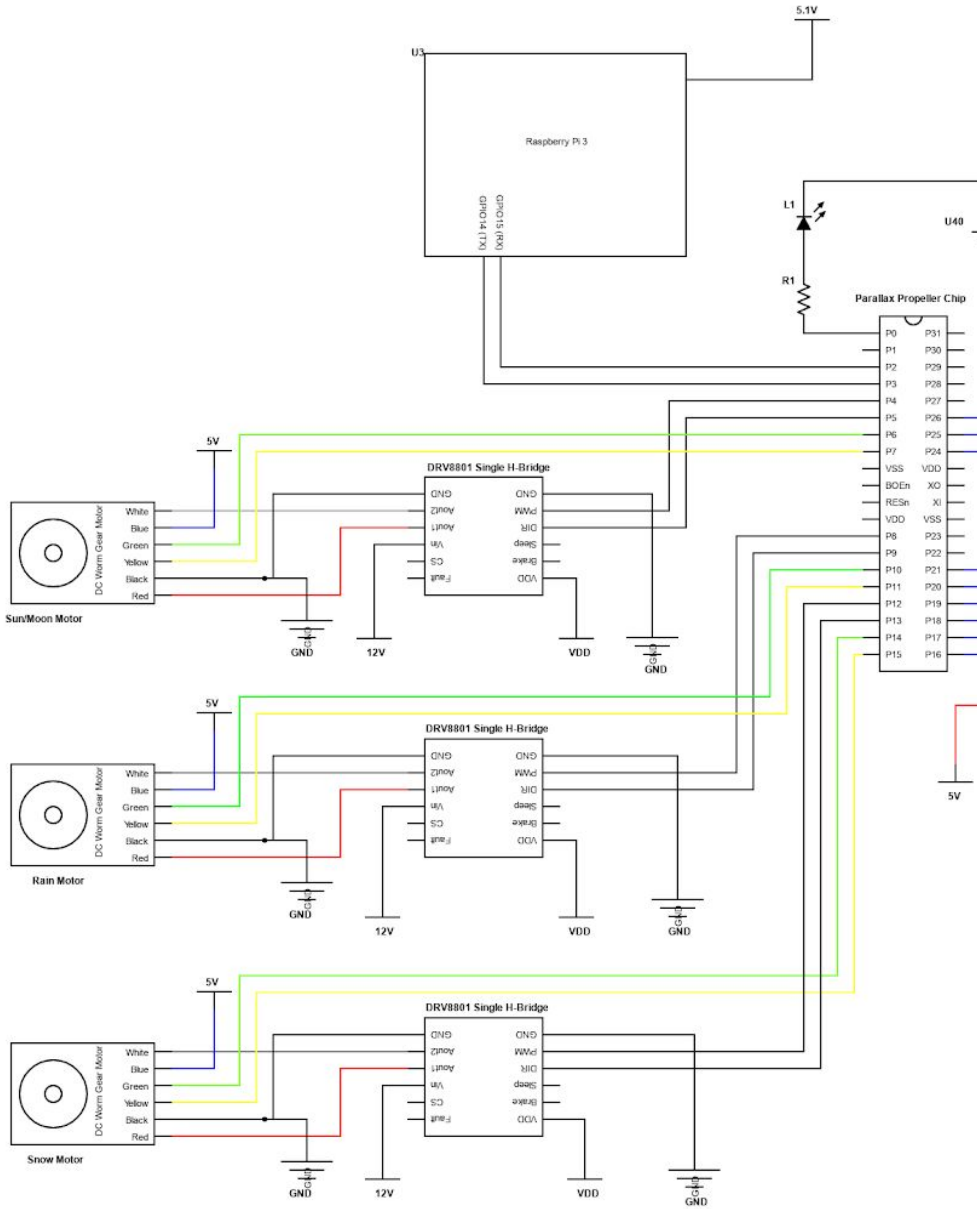
```
PUB Encoder(pinA, pinB)
  position~
  repeat
    case ina[pinA..pinB]
      %00 : repeat until ina[pinA..pinB]<>%00
            if ina[pinA..pinB]==%01
              position++
            if ina[pinA..pinB]==%10
              position--
      %01 : repeat until ina[pinA..pinB]<>%01
            if ina[pinA..pinB]==%11
              position++
            if ina[pinA..pinB]==%00
              position--
      %11 : repeat until ina[pinA..pinB]<>%11
            if ina[pinA..pinB]==%10
              position++
            if ina[pinA..pinB]==%01
              position--
      %10 : repeat until ina[pinA..pinB]<>%10
            if ina[pinA..pinB]==%00
              position++
            if ina[pinA..pinB]==%11
              position--

PUB PWM(pin) | endcnt

  dira[pin]~~
  ctra[5..0]:=pin
  ctra[30..26]:=%00100

  frqa:=1
  endcnt:=cnt
  repeat
    phsa:=-(100*DutyCycle)
    endcnt:=endcnt+10_000
    waitcnt(endcnt)

''''''''''''''''''''''''''ALL METHODS BELOW ARE FOR LED ARRAY''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
PUB drawTime(x, y, color)

    if hour > 9                                           '21 pixels wide
      drawTwoDigitNumber(hour, x, y, color)
      drawColon((x+10), y, color)
      drawTwoDigitNumber(minute, (x+12), y, color)
    else                                                  '16 pixels wide
      drawNumber(hour, x, y, color)
      drawColon((x+5), y, color)
      drawTwoDigitNumber(minute, (x+7), y, color)
PUB drawTemp(x, y, color)
  case temp
    -99..-10:                                             '17 pixels wide
      drawNegative(x, y, color)
      drawTwoDigitNumber(-temp, (x+3), y, color)
      drawDegree((x+13), y, color)
    -9..-1:                                               '12 pixels wide
      drawNegative(x, y, color)
      drawNumber(-temp, (x+3), y, color)
      drawDegree((x+8), y, color)
    0..9:                                                 '8 pixels wide
      drawNumber(temp, x, y, color)
      drawDegree((x+5), y, color)
    10..99:                                               '13 pixels wide
      drawTwoDigitNumber(temp, x, y, color)
      drawDegree((x+10), y, color)
    100..999:                                             '18 pixels wide
      drawThreeDigitNumber(temp, x, y, color)
      drawDegree((x+15), y, color)

PUB drawNumber(num, x, y, color)                          'draws a number from 0 to 9, x and y position indicate top-left corner
  case num
    0:  drawZero(x, y, color)
    1:  drawOne(x, y, color)
    2:  drawTwo(x, y, color)
    3:  drawThree(x, y, color)
    4:  drawFour(x, y, color)
    5:  drawFive(x, y, color)
    6:  drawSix(x, y, color)
    7:  drawSeven(x, y, color)
    8:  drawEight(x, y, color)
    9:  drawNine(x, y, color)
```

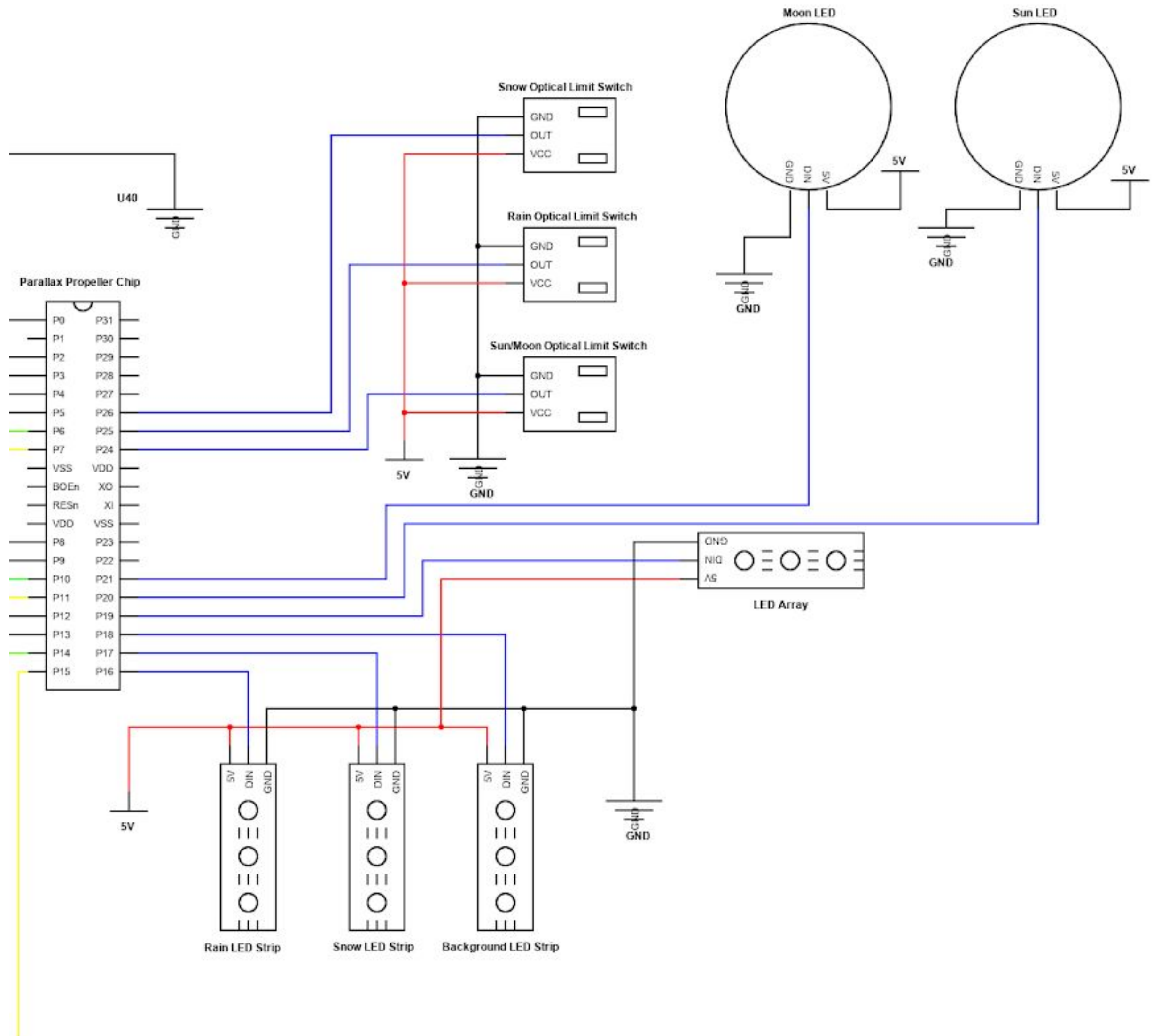
```
PUB drawTwoDigitNumber(num, x, y, color)                          'draws a two-digit number with x and y referencing top left corner
  drawNumber((num/10), x, y, color)                               'draws first digit of number
  drawNumber((num//10), (x + 5), y, color)                        'draws second digit of number 5 pixels to the right

PUB drawThreeDigitNumber(num, x, y, color)                        'draws a three-digit number with x and y referencing top left corner
  drawTwoDigitNumber((num/10), x, y, color)                       'draws first two digits of number
  drawNumber((num//10), (x + 10), y, color)                       'draws third digit of number 10 pixels to the right

PUB drawColon(x, y, color)                                        'y references y-position to be in-line with numbers (1 pixel above top dot)
  rgb[1].LED(convertCoords(x, (y+1)), color)
  rgb[1].LED(convertCoords(x, (y+4)), color)

PUB drawDegree(x, y, color)
  rgb[1].LED(convertCoords((x+1), y), color)
  rgb[1].LED(convertCoords(x, (y+1)), color)
  rgb[1].LED(convertCoords((x+2), (y+1)), color)
  rgb[1].LED(convertCoords((x+1), (y+2)), color)

PUB drawNegative(x, y, color)
  rgb[1].LED(convertCoords(x, (y+2)), color)
  rgb[1].LED(convertCoords((x+1), (y+2)), color)

PUB convertCoords(x, y) : location
  if (x//2) == 0                     'if x is even
    location := (x * yLEDs) + y
  else                               'x is odd
    location := ((x + 1) * (yLEDs)) - 1 - y

PUB drawZero(x, y, color) | i
  repeat i from 1 to 4                                            'column 1
    rgb[1].LED(convertCoords(x, (y+i)), color)
  rgb[1].LED(convertCoords((x+1), (y+5)), color)                  'column 2
  rgb[1].LED(convertCoords((x+1), (y)), color)
  rgb[1].LED(convertCoords((x+2), (y)), color)                    'column 3
  rgb[1].LED(convertCoords((x+2), (y+5)), color)
  repeat i from 1 to 4                                            'column 4
    rgb[1].LED(convertCoords((x+3), (y+i)), color)

PUB drawOne(x, y, color) | i
  rgb[1].LED(convertCoords((x+1), y), color)                      'column 2
  repeat i from 0 to 5
    rgb[1].LED(convertCoords((x+2), (y+i)), color)                'column 3

PUB drawTwo(x, y, color) | i
  rgb[1].LED(convertCoords(x, y), color)                          'column 1
  repeat i from 3 to 5
    rgb[1].LED(convertCoords(x, (y+i)), color)
  repeat i from 1 to 2                                            'columns 2 and 3
    rgb[1].LED(convertCoords((x+i), y), color)
    rgb[1].LED(convertCoords((x+i), (y+2)), color)
    rgb[1].LED(convertCoords((x+i), (y+5)), color)
  rgb[1].LED(convertCoords((x+3), (y+1)), color)                  'column 4
  rgb[1].LED(convertCoords((x+3), (y+5)), color)

PUB drawThree(x, y, color) | i
  rgb[1].LED(convertCoords(x, y), color)                          'column 1
  rgb[1].LED(convertCoords(x, (y+5)), color)
  repeat i from 1 to 2                                            'columns 2 and 3
    rgb[1].LED(convertCoords((x+i), y), color)
    rgb[1].LED(convertCoords((x+i), (y+2)), color)
    rgb[1].LED(convertCoords((x+i), (y+5)), color)
  rgb[1].LED(convertCoords((x+3), (y+1)), color)                  'column 4
  rgb[1].LED(convertCoords((x+3), (y+3)), color)
  rgb[1].LED(convertCoords((x+3), (y+4)), color)

PUB drawFour(x, y, color) | i
  repeat i from 0 to 2                                            'column 1
    rgb[1].LED(convertCoords(x, (y+i)), color)
  rgb[1].LED(convertCoords((x+1), (y+2)), color)                  'column 2
  rgb[1].LED(convertCoords((x+2), (y+2)), color)                  'column 3
  repeat i from 0 to 5                                            'column 4
    rgb[1].LED(convertCoords((x+3), (y+i)), color)

PUB drawFive(x, y, color) | i
  repeat i from 0 to 2                                            'column 1
    rgb[1].LED(convertCoords(x, (y+i)), color)
  rgb[1].LED(convertCoords(x, (y+5)), color)
  repeat i from 1 to 2                                            'columns 2 and 3
    rgb[1].LED(convertCoords((x+i), y), color)
```

```
    rgb[1].LED(convertCoords((x+i), (y+2)), color)
    rgb[1].LED(convertCoords((x+i), (y+5)), color)
  rgb[1].LED(convertCoords((x+3), y), color)              'column 4
  rgb[1].LED(convertCoords((x+3), (y+3)), color)
  rgb[1].LED(convertCoords((x+3), (y+4)), color)

PUB drawSix(x, y, color) | i
  repeat i from 1 to 4                                  'column 1
    rgb[1].LED(convertCoords(x, (y+i)), color)
  repeat i from 1 to 2                                  'columns 2 and 3
    rgb[1].LED(convertCoords((x+i), y), color)
    rgb[1].LED(convertCoords((x+i), (y+2)), color)
    rgb[1].LED(convertCoords((x+i), (y+5)), color)
  rgb[1].LED(convertCoords((x+3), y), color)              'column 4
  rgb[1].LED(convertCoords((x+3), (y+3)), color)
  rgb[1].LED(convertCoords((x+3), (y+4)), color)

PUB drawSeven(x, y, color) | i
  rgb[1].LED(convertCoords(x, y), color)                  'column 1
  rgb[1].LED(convertCoords((x+1), y), color)              'column 2
  repeat i from 3 to 5
    rgb[1].LED(convertCoords((x+1), (y+i)), color)
  rgb[1].LED(convertCoords((x+2), y), color)              'column 3
  rgb[1].LED(convertCoords((x+2), (y+2)), color)
  rgb[1].LED(convertCoords((x+3), y), color)              'column 4
  rgb[1].LED(convertCoords((x+3), (y+1)), color)

PUB drawEight(x, y, color) | i
  rgb[1].LED(convertCoords(x, (y+1)), color)              'column 1
  rgb[1].LED(convertCoords(x, (y+3)), color)
  rgb[1].LED(convertCoords(x, (y+4)), color)
  repeat i from 1 to 2                                  'columns 2 and 3
    rgb[1].LED(convertCoords((x+i), y), color)
    rgb[1].LED(convertCoords((x+i), (y+2)), color)
    rgb[1].LED(convertCoords((x+i), (y+5)), color)
  rgb[1].LED(convertCoords((x+3), (y+1)), color)          'column 4
  rgb[1].LED(convertCoords((x+3), (y+3)), color)
  rgb[1].LED(convertCoords((x+3), (y+4)), color)

PUB drawNine(x, y, color) | i
```

```
  rgb[1].LED(convertCoords(x, (y+1)), color)              'column 1
  rgb[1].LED(convertCoords(x, (y+4)), color)
  repeat i from 1 to 2                                  'columns 2 and 3
    rgb[1].LED(convertCoords((x+i), y), color)
    rgb[1].LED(convertCoords((x+i), (y+2)), color)
    rgb[1].LED(convertCoords((x+i), (y+5)), color)
  repeat i from 1 to 4                                  'column 4
    rgb[1].LED(convertCoords((x+3), (y+i)), color)
```

Appendix B: Circuit Diagram of Kinetic Art Weather Clock




Moon LED
Sun LED
Snow Optical Limit Switch
Rain Optical Limit Switch
Sun/Moon Optical Limit Switch
GND
OUT
VCC
U40
Parallax Propeller Chip
P0
P1
P2
P3
P4
P5
P6
P7
VSS
BOEn
RESn
VDD
P8
P9
P10
P11
P12
P13
P14
P15
P31
P30
P29
P28
P27
P26
P25
P24
VDD
XO
XI
VSS
P23
P22
P21
P20
P19
P18
P17
P16
5V
DIN
LED Array
Rain LED Strip
Snow LED Strip
Background LED Strip

Appendix C: Python Code Run on Raspberry Pi as of 12/01/2020

```python
# Python program to find current
# weather details of any city
# using openweathermap api

# import required modules
import requests, json, serial
from time import sleep
from datetime import datetime

# Enter your API key here
api_key = "bf0d90979461967e0746b9a6e4a4c022"

# base_url variable to store url
base_url = "http://api.openweathermap.org/data/2.5/weather?"

# Give city name
city_name = "Charlottesville"

# complete_url variable to store
# complete url address
complete_url = base_url + "q=" + city_name + "&appid=" + api_key
ser = serial.Serial("/dev/ttyS0", 9600)
while True:
    # get method of requests module
    # return response object
    received_data = ser.read()
    sleep(0.03)
    data_left = ser.inWaiting()
```

```python
    received_data += ser.read(data_left)

    recieved_data = received_data.decode('utf-8')
    #print(received_data)
    if received_data.decode('utf-8') == "getID":
        response = requests.get(complete_url)
        x = response.json()
        # makes sure data can be found (404 = Not Found)
        if x["cod"] != "404":
            z = x["weather"]
            id = str(z[0]["id"])
            ser.write(id.encode('utf-8'))
        else:
            print("Error in the HTTP request")
    elif received_data.decode('utf-8') == "getTemp":
        response = requests.get(complete_url)
        x = response.json()
        # makes sure data can be found (404 = Not Found)
        if x["cod"] != "404":
            y = x["main"]
            current_temperature = y["temp"]
            current_temperature = int((current_temperature-273) * (9/5) + 32)
            if current_temperature > 100:
                current_temperature = str(current_temperature)
            elif current_temperature < 0:
                current_temperature = "000"
            elif current_temperature < 10:
                current_temperature = "00" + str(current_temperature)
```

```python
        else:
            current_temperature = "0" + str(current_temperature)
        ser.write(current_temperature.encode('utf-8'))
    else:
        print("Error in the HTTP request")
elif received_data.decode('utf-8') == "getSunrise":
    response = requests.get(complete_url)
    x = response.json()
    # makes sure data can be found (404 = Not Found)
    if x["cod"] != "404":
        y = x["sys"]
        sunrise = y["sunrise"]
        sunset = y["sunset"]
        timeshift = x["timezone"]
        sunrise += timeshift
        sunset += timeshift
        sunrise = str(sunrise)
        sunset = str(sunset)
        ser.write(sunrise.encode('utf-8'))
        ser.write(sunset.encode('utf-8'))

    else:
        print("Error in the HTTP request")
elif received_data.decode('utf-8') == "getTime":
    current_time = datetime.now().strftime("%H%M")
    ser.write(current_time.encode('utf-8'))
```

Appendix D: LED Driver Spin Code (WS2812B_RGB_LED_Driver_v2.1)

```
'' WS2812B_RGB_LED_Driver
'' by Gavin T. Garner
'' University of Virginia
'' April 20, 2012
{  This object can be used to control a Red-Green-Blue LED light strip (such as the 1m and 2m
   ones available from Pololu.com as parts #2540 and #2541). These strips incorporate TM1804 chips by
   Titan Micro (one for each RGB LED) and 24-bit color data is shifted into them using quick pulses
   (~1300ns=Digital_1 and ~700ns=Digital_0). Because these pulses are so quick, they must be generated
   using PASM code. The advantage to this is that they can be updated and changed much more quickly
   than other types of addressable RGB LED strips. Note that this code will not control RGB LED strips
   that use WS2801 chips (such as the ones currently sold by Sparkfun.com).

                                    Instructions for use:
   Wiring:
         Propeller I/O pin (your choice) <---> IN  (silver wire with white stripe on Pololu Part)
                         Propeller's Vss <---> GND (silver wire with no stripe on Pololu Part)
   NC (both GND terminals are connected) <---> GND (black wire w/dashed white stripe on Pololu Part)
        5V Power Supply (1.25Amps/meter) <---> +VC (black wire with no stripe on Pololu Part)
   Software:
   Insert this RGB_LED_Strip object into your code and call the "start" method. This will
   start the assembly program on a new cog where it will run continuously and take care of
   communication between your spin code and the TM1804 chips. Once this PASM driver is started, you
   can call the methods below such as rgb.ChangeLED(0,255)
   You can also create your own methods, but note that you must set the "update" variable to a
   non-zero value (eg. update:=true) whenever you want the LEDs to change/update
   Note: If you want to control more than 60 LEDs (2 meters), you will need to increase the number
   of longs alotted to the LED variable array below (eg. lights[120] for two 2m strips wired together).
                                        HAVE FUN!!!                                                 }
CON        'Predefined colors that can be accessed from your code using rgb#constant:
                                                   '  green     red      blue
 off            = 0                                '%00000000_00000000_00000000
 red            = 255<<8                           '%00000000_11111111_00000000
 green          = 255<<16                          '%11111111_00000000_00000000
 blue           = 255                              '%00000000_00000000_11111111
 white          = 255<<16+255<<8+255               '%11111111_11111111_11111111
 cyan           = 255<<16+255                      '%11111111_00000000_11111111
 magenta        = 255<<8+255                       '%00000000_11111111_11111111
 yellow         = 255<<16+255<<8                   '%11111111_11111111_00000000
 chartreuse     = 255<<16+127<<8                   '%11111111_01111111_00000000
 orange         = 60<<16+255<<8                    '%10100101_11111111_11010100
 aquamarine     = 255<<16+127<<8+212               '%11111111_11111111_11010100
 pink           = 128<<16+255<<8+128               '%10000000_11111111_10000000
 turquoise      = 224<<16+63<<8+192                '%10000000_00111111_10000000
 realwhite      = 255<<16+200<<8+255               '%11100000_11001000_11000000
 indigo         = 170                              '%00000000_00111111_01111111
 violet         = 51<<16+215<<8+255                '%01111111_10111111_10111111
 grey           = 128<<16+128<<8+128               '%10000000_10000000_10000000
 darkgrey       = 169<<16+169<<8+169               '%10000000_10000000_10000000
 nightsky       = 12<<16+20<<8+69

VAR
  long update            'Controls when LED values are sent (its address gets loaded into Cog 1)
  long maxAddress        'Address of the last LED in the string
  long cog               'Store cog # (so that the cog can be stopped)
  long LEDs              'Stores the total number of addressable LEDs
  long lights[484]       'Reserve a long for each LED address in the string
             ' ↑ THIS WILL NEED TO BE INCREASED IF YOU ARE CONTROLLING MORE THAN 256 LEDs!!!
PUB start(OutputPin,NumberOfLEDs) : okay
'' Starts RGB LED Strip driver on a cog, returns false if no cog available
'' Note: Requires at least a 20MHz system clock
  _pin:=OutputPin
  _LEDs:=NumberOfLEDs
  LEDs:=NumberOfLEDs
  maxAddress:=NumberOfLEDs-1
  _update:=@update

'LED Strip WS2812B chip
  High1:=61   '0.9us
  Low1:=19     '0.35us
  High0:=35   '0.35us
  Low0:=76    '0.9us
  reset:=5000 '50microseconds

  stop                                      'Stop the cog (just in case)
  okay:=cog:=cognew(@RGBdriver,@lights)+1'Start PASM RGB LED Strip driver

PUB stop                                   ''Stops the RGB LED Strip driver and releases the cog
  if cog
    cogstop(cog~ - 1)

```

```
PUB LED(LEDaddress,color)                ''Changes the color of an LED at a specific address
  lights[LEDaddress]:=color


PUB UpdateLEDs
  update:=true

PUB LEDRGB(LEDaddress,_red,_green,_blue) ''Changes RGB values of an LED at a specific address
  lights[LEDaddress]:=_red<<16+_green<<8+_blue
  update:=true

PUB LEDint(LEDaddress,color,intense)               ''Changes the color of an LED at a specific address
  lights[LEDaddress]:=((((color>>16)*intense)/255)<<16) +((((color>>8 & $FF)*intense)/255)<<8)+(((color & $FF)*intense)/255)

PUB Intensity(color,intense) : newvalue              ''Changes the intensity (0-255) of a color
  newvalue:=((((color>>16)*intense)/255)<<16) +((((color>>8 & $FF)*intense)/255)<<8)+(((color & $FF)*intense)/255)

PUB SetAllColors(setcolor) | i           ''Changes the colors of all LEDs to the same color
  longfill(@lights,setcolor,maxAddress+1)
  update:=true

PUB AllOff | i                           ''Turns all of the LEDs off
  longfill(@lights,0,maxAddress+1)
  update:=true
  waitcnt(clkfreq/100+cnt)               'Can't send the next update too soon

PUB SetSection(AddressStart,AddressEnd,setcolor)  ''Changes colors in a section of LEDs to same color
  longfill(@lights[AddressStart],setcolor,AddressEnd-AddressStart+1)'(@lights[AddressEnd]-@lights[AddressStart])/4)
  update:=true

PUB GetColor(address) : color            ''Returns 24-bit RGB value from specified LED's address
  color:=lights[address]

PUB Random(address) | rand,_red,_green,_blue,timer ''Sets LED at specified address to a "random" color
  rand:=?cnt
  _red:=rand>>24
  rand:=?rand
  _green:=rand>>24
  rand:=?rand
  _blue:=rand>>24
```

```
  lights[address]:=_red<<16+_green<<8+_blue
  update:=true

DAT
''This PASM code sends control data to the RGB LEDs on the strip once the "update" variable is set to
'' a value other than 0
              org       0
RGBdriver     mov       pinmask,#1          'Set direction of data pin to be an output
              shl       pinmask,_pin
              mov       dira,pinmask
              mov       index,par           'Set index to LED variable array's base address

StartDataTX   rdlong    check,_update       '
              tjz       check,#StartDataTX  'Wait for Cog 0 to set "update" to true or 1
              mov       count,#0            'Start with "index" count=0

AddressLoop   rdlong    RGBvalue,index      'Fetch RGB[index] value from central Hub RAM
              mov       shift,#23           'Start with shift=23 (shift to MSB of Red value)

BitLoop       mov       outa,pinmask        'Set data pin High
              mov       getbit,RGBvalue     'Store RGBvalue as "getbit"
              shr       getbit,shift        'Shift this RGB value right "shift" # of bits
              and       getbit,#1           'Lop off all bits except LSB
              cmp       getbit,#1      wz   'Check if bit=1, if so, set Z flag
        if_z  jmp       #DigiOne
DigiZero      mov       counter,cnt         'Output a pulse corresponding to a digital 0
              'add       counter,High0

              'waitcnt   counter,Low0        'Wait for 0.7us
              add       counter,Low0
              mov       outa,#0             'Set data pin Low
              waitcnt   counter,#0          'Wait for 1.8us

              tjz       shift,#Increment    'If shift=0, jump down to "Increment"
              sub       shift,#1            'Decrement shift by 1
              jmp       #BitLoop            'Repeat BitLoop if "shift" has not reached 0

DigiOne       mov       counter,cnt         'Output a pulse corresponding to a digital 1
              add       counter,High1
              waitcnt   counter,Low1        'Wait for 1.3us
```

```
              mov      outa,#0                'Set data pin Low
              waitcnt  counter,#0             'Wait for 1.2us
              tjz      shift,#Increment       'If shift=0, jump down to "Increment"
              sub      shift,#1               'Decrement shift by 1

              jmp      #BitLoop               'Repeat BitLoop if "shift" has not reached 0

Increment     add      index,#4               'Increment index by 4 byte addresses (1 long)
              add      count,#1               'Increment count by 1
              cmp      count,_LEDs      wz    'Check to see if all LEDs have been set
        if_nz jmp      #AddressLoop           'If not, repeat AddressLoop for next LED's RGBvalue

              mov      counter,cnt
              add      counter,reset
              waitcnt  counter,#0             'Wait for 24us (reset datastream)
              wrlong   zero,_update           'Set update value to 0, wait for Cog 0 to reset this
              mov      index,par              'Set index to LED variable array's base address
              jmp      #StartDataTX

                                              'Starred values (*) are set before cog is loaded
_update       long     0                      'Hub RAM address of "update" will be stored here*
_pin          long     0                      'Output pin number will be stored here*
_LEDs         long     0                      'Total number of LEDs will be stored here*
High1         long     0                      '~1.3 microseconds(digital 1)*
Low1          long     0                      '~1.2 microseconds*
High0         long     0                      '~0.7 microseconds(digital 0)*
Low0          long     0                      '~1.8 microseconds*
reset         long     0                      '~25 microseconds (the 24us spec doesn't seem to work)*
zero          long     0
pinmask       res
RGBvalue      res
getbit        res
counter       res
count         res
check         res
index         res
shift         res
last          res
              fit

```

Appendix E: FullDuplexSerial Spin Code

```
{{
Object file:    FullDuplexSerial.spin
Version:        1.2.1
Date:           2006 - 2011
Author:         Chip Gracey, Jeff Martin, Daniel Harris
Company:        Parallax Semiconductor
Email:          dharris@parallaxsemiconductor.com
Licensing:      MIT License - see end of file for terms of use.

Description:
This driver, once started, implements a serial port in one cog.

Revision History:
v1.2.1 - 5/1/2011 Added extra comments and demonstration code to bring up
                   to gold standard.
v1.2 - 5/7/2009 Fixed bug in dec method causing largest negative value
                (-2,147,483,648) to be output as -0.
v1.1 - 3/1/2006 First official release.


=============================================
        Connection Diagram
=============================================

        â”Œâ”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
        â”‚         â”‚
        â”‚    rxPinâ”œâ”€â”€â”€ï‚ª TTL level RX line
        â”‚    txPinâ”œâ”€â”€â”€ï‚» TTL level TX line
        â”‚         â”‚
        â””â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜
         Propeller
            MCU
          (P8X32A)

Components:
N/A

=============================================
}}


VAR

  'Global variable declarations

  long  cog           'cog flag/id

  '9 longs, MUST be contiguous
  long  rx_head
  long  rx_tail
  long  tx_head
  long  tx_tail
  long  rx_pin
  long  tx_pin
  long  rxtx_mode
  long  bit_ticks
  long  buffer_ptr

  byte  rx_buffer[16]           'transmit and receive buffers
  byte  tx_buffer[16]           '16 bytes each


PUB Start(rxPin, txPin, mode, baudrate) : okay
{{
   Start serial driver - starts a cog


   Parameters: rxPin    = Propeller pin to set up as RX-ing pin.  Range = 0 - 31
               txPin    = Propeller pin to set up as TX-ing pin.  Range = 0 - 31
               mode     = bitwise mode configuration variable, see mode bit description below.
               baudrate = baud rate to transmit bits at.

   mode bit 0 = invert rx
   mode bit 1 = invert tx
   mode bit 2 = open-drain/source tx
   mode bit 3 = ignore tx echo on rx

   return: Numeric value of the cog(1-8) that was started, false(0) if no cog is available.

   example usage: serial.start(31, 30, %0000, 9_600)

   expected outcome of example usage call: Starts a serial port on Propller pins 30 and 31.
                                           The serial port does not invert the RX and TX data,
                                           no open-drain/source on the TX pin, does not ignore
                                           data echoed on RX pin, at 9,600 baud.
}}

  Stop                                                  'make sure the driver isnt already running
  longfill(@rx_head, 0, 4)                              'zero out the buffer pointers
  longmove(@rx_pin, @rxpin, 3)                          'copy the start parameters to this objects pin variables
  bit_ticks := clkfreq / baudrate                       'number of clock ticks per bit for the desired baudrate
  buffer_ptr := @rx_buffer                              'save the address of the receive buffer
```

```
  okay := cog := cognew(@entry, @rx_head) + 1           'start the new cog now, assembly cog at "entry" label.


PUB Stop
{{
   Stop serial driver if it has already been started - frees the cog

   Parameters: none
   return:     none

   example usage: serial.stop

   expected outcome of example usage call: Stops an already started serial port.
}}

  if cog
    cogstop(cog~ - 1)                                   'if the driver is already running, stop the cog
  longfill(@rx_head, 0, 9)                              'zero out configuration variables


PUB RxFlush
{{
   Continuously pops the head of the receive buffer until no bytes remain.

   Parameters: none
   return:     none

   example usage: serial.RxFlush

   expected outcome of example usage call: Receive bffer will be cleared.
}}

  repeat while RxCheck => 0                             'Call RxCheck until buffer is empty


PUB RxCheck : rxByte
{{
   Check if a byte is waiting in the receive buffer and return the byte if one is there,
   does NOT block (never waits).

   Parameters: none
   return:     If no byte, then return(-1).  If byte, then return(byte).

   example usage: serial.RxCheck

   expected outcome of example usage call: Return a byte if one is available, but dont wait
                                           for a byte to come in.
}}


  rxByte--                                              'make rxbyte = -1
  if rx_tail <> rx_head                                 'if a byte is in the buffer, then
    rxByte := rx_buffer[rx_tail]                        '  grab it and store in rxByte
    rx_tail := (rx_tail + 1) & $F                       '  advance the buffer pointer


PUB RxTime(ms) : rxByte | t
{{
   Wait ms milliseconds for a byte to be received

   Parameters: ms = number of milliseconds to wait for a byte to be received.
   return:     If no byte, then return(-1).  If byte, then return(byte).

   example usage: serial.RxTime(500)

   expected outcome of example usage call: Wait half a second (500 ms) for a byte to be received.
}}

  t := cnt                                              'take note of the current time
  repeat until (rxByte := RxCheck) => 0 or (cnt - t) / (clkfreq / 1000) > ms


PUB Rx : rxByte
{{
   Receive byte (may wait for byte)
   returns $00..$FF

   Parameters: none
   return:     received byte

   example usage: serial.Rx

   expected outcome of example usage call: Wait until a byte has been received, then return that byte.
}}

  repeat while (rxByte := RxCheck) < 0                  'return the byte, wait while the buffer is empty


PUB Tx(txByte)
{{
   Places a byte into the transmit buffer for transmission (may wait for room in buffer).

   Parameters: txByte = the byte to be transmitted
   return:     none
```

```
   example usage: serial.Tx($0D)

   expected outcome of example usage call: Transmits the byte $0D serially on the txPin
}}

  repeat until (tx_tail <> (tx_head + 1) & $F)          'wait until the buffer has room
  tx_buffer[tx_head] := txByte                          'place the byte into the buffer
  tx_head := (tx_head + 1) & $F                         'advance the buffer's pointer

  if rxtx_mode & %1000                                  'if ignoring rx echo
    Rx                                                  '   receive the echoed byte and discard


PUB Str(stringPtr)
{{
   Transmit a string of bytes

   Parameters: stringPtr = the pointer address of the null-terminated string to be sent
   return:     none

   example usage: serial.Str(@test_string)

   expected outcome of example usage call: Transmits each byte of a string at the address some_string.
}}

  repeat strsize(stringPtr)
    Tx(byte[stringPtr++])                                                       'Transmit each byte in the string


PUB Dec(value) | i, x
{{
   Transmit the ASCII string equivalent of a decimal value

   Parameters: dec = the numeric value to be transmitted
   return:     none

   example usage: serial.Dec(-1_234_567_890)

   expected outcome of example usage call: Will print the string "-1234567890" to a listening terminal.
}}

  x := value == NEGX                                    'Check for max negative
  if value < 0
    value := ||(value+x)                                'If negative, make positive; adjust for max negative
    Tx("-")                                             'and output sign

  i := 1_000_000_000                                    'Initialize divisor

  repeat 10                                             'Loop for 10 digits
    if value => i
      Tx(value / i + "0" + x*(i == 1))                  'If non-zero digit, output digit; adjust for max negative
      value //= i                                       'and digit from value
      result~~                                          'flag non-zero found
    elseif result or i == 1
      Tx("0")                                           'If zero digit (or only digit) output it
    i /= 10                                             'Update divisor


PUB Hex(value, digits)
{{
   Transmit the ASCII string equivalent of a hexadecimal number

   Parameters: value = the numeric hex value to be transmitted
               digits = the number of hex digits to print
   return:     none

   example usage: serial.Hex($AA_FF_43_21, 8)

   expected outcome of example usage call: Will print the string "AAFF4321" to a listening terminal.
}}

  value <<= (8 - digits) << 2
  repeat digits                                         'do it for the number of hex digits being transmitted
    Tx(lookupz((value <-= 4) & $F : "0".."9", "A".."F"))'  Transmit the ASCII value of the hex characters


PUB Bin(value, digits)
{{
   Transmit the ASCII string equivalent of a binary number

   Parameters: value = the numeric binary value to be transmitted
               digits = the number of binary digits to print
   return:     none

   example usage: serial.Bin(%1110_0011_0000_1100_1111_1010_0101_1111, 32)

   expected outcome of example usage call: Will print the string "11100011000011001111101001011111" to a listening terminal.
}}

  value <<= 32 - digits
  repeat digits
    Tx((value <-= 1) & 1 + "0")                         'Transmit the ASCII value of each binary digit


DAT
```

```
'***********************************
'* Assembly language serial driver *
'***********************************

                        org
'
'
' Entry
'
entry                   mov     t1,par                'get structure address
                        add     t1,#4 << 2            'skip past heads and tails

                        rdlong  t2,t1                 'get rx_pin
                        mov     rxmask,#1
                        shl     rxmask,t2

                        add     t1,#4                 'get tx_pin
                        rdlong  t2,t1
                        mov     txmask,#1
                        shl     txmask,t2

                        add     t1,#4                 'get rxtx_mode
                        rdlong  rxtxmode,t1

                        add     t1,#4                 'get bit_ticks
                        rdlong  bitticks,t1

                        add     t1,#4                 'get buffer_ptr
                        rdlong  rxbuff,t1
                        mov     txbuff,rxbuff
                        add     txbuff,#16

                        test    rxtxmode,#%100  wz    'init tx pin according to mode
                        test    rxtxmode,#%010  wc
        if_z_ne_c       or      outa,txmask
        if_z            or      dira,txmask

                        mov     txcode,#transmit      'initialize ping-pong multitasking
'
'
' Receive
'
receive                 jmpret  rxcode,txcode         'run a chunk of transmit code, then return

                        test    rxtxmode,#%001  wz    'wait for start bit on rx pin
                        test    rxmask,ina      wc
        if_z_eq_c       jmp     #receive

                        mov     rxbits,#9             'ready to receive byte
                        mov     rxcnt,bitticks
                        shr     rxcnt,#1
                        add     rxcnt,cnt

:bit                    add     rxcnt,bitticks        'ready next bit period

:wait                   jmpret  rxcode,txcode         'run a chuck of transmit code, then return

                        mov     t1,rxcnt              'check if bit receive period done
                        sub     t1,cnt
                        cmps    t1,#0           wc
        if_nc           jmp     #:wait

                        test    rxmask,ina      wc    'receive bit on rx pin
                        rcr     rxdata,#1
                        djnz    rxbits,#:bit

                        shr     rxdata,#32-9          'justify and trim received byte
                        and     rxdata,#$FF
                        test    rxtxmode,#%001  wz    'if rx inverted, invert byte
        if_nz           xor     rxdata,#$FF

                        rdlong  t2,par                'save received byte and inc head
                        add     t2,rxbuff
                        wrbyte  rxdata,t2
                        sub     t2,rxbuff
                        add     t2,#1
                        and     t2,#$0F
                        wrlong  t2,par

                        jmp     #receive              'byte done, receive next byte
'
'
' Transmit
'
transmit                jmpret  txcode,rxcode         'run a chunk of receive code, then return

                        mov     t1,par                'check for head <> tail
                        add     t1,#2 << 2
                        rdlong  t2,t1
                        add     t1,#1 << 2
                        rdlong  t3,t1
                        cmp     t2,t3           wz
        if_z            jmp     #transmit
```

```
                        add     t3,txbuff              'get byte and inc tail
                        rdbyte  txdata,t3
                        sub     t3,txbuff
                        add     t3,#1
                        and     t3,#$0F
                        wrlong  t3,t1

                        or      txdata,#$100           'ready byte to transmit
                        shl     txdata,#2
                        or      txdata,#1
                        mov     txbits,#11
                        mov     txcnt,cnt

:bit                    test    rxtxmode,#%100  wz     'output bit on tx pin according to mode
                        test    rxtxmode,#%010  wc
        if_z_and_c      xor     txdata,#1
                        shr     txdata,#1       wc
        if_z            muxc    outa,txmask
        if_nz           muxnc   dira,txmask
                        add     txcnt,bitticks         'ready next cnt

:wait                   jmpret  txcode,rxcode          'run a chunk of receive code, then return

                        mov     t1,txcnt               'check if bit transmit period done
                        sub     t1,cnt
                        cmps    t1,#0           wc
        if_nc           jmp     #:wait

                        djnz    txbits,#:bit           'another bit to transmit?

                        jmp     #transmit              'byte done, transmit next byte
'
'
' Uninitialized data
'
t1                      res     1
t2                      res     1
t3                      res     1

rxtxmode                res     1
bitticks                res     1

rxmask                  res     1
rxbuff                  res     1
rxdata                  res     1
rxbits                  res     1
rxcnt                   res     1
rxcode                  res     1

txmask                  res     1
txbuff                  res     1
txdata                  res     1
txbits                  res     1
txcnt                   res     1
txcode                  res     1


DAT
{{
```