Introduction

For this Arduino internship, we decided to challenge ourselves by attempting to conceive, design, build, and program a complex tool from scratch. Having both taken the Astrophysics course at Dawson College this semester, in addition to Engineering Physics, we thought of an interesting idea that would bring what we learned in those classes together while giving us the opportunity to improve on our robotics and DIY skills in general; we decided to build what we dubbed the Celestial Browser, to give an original name to the project, but is commonly referred to as a star tracker in astronomy fields. This piece of machinery which essentially consists of a self-adjusting telescope (or at least a small eyepiece) would require in-depth coding and calculations as well as implementing of basic electronics and 3D printing skills, a challenge we gladly took on and did our best to bring as close to completion as we could over the past few months.

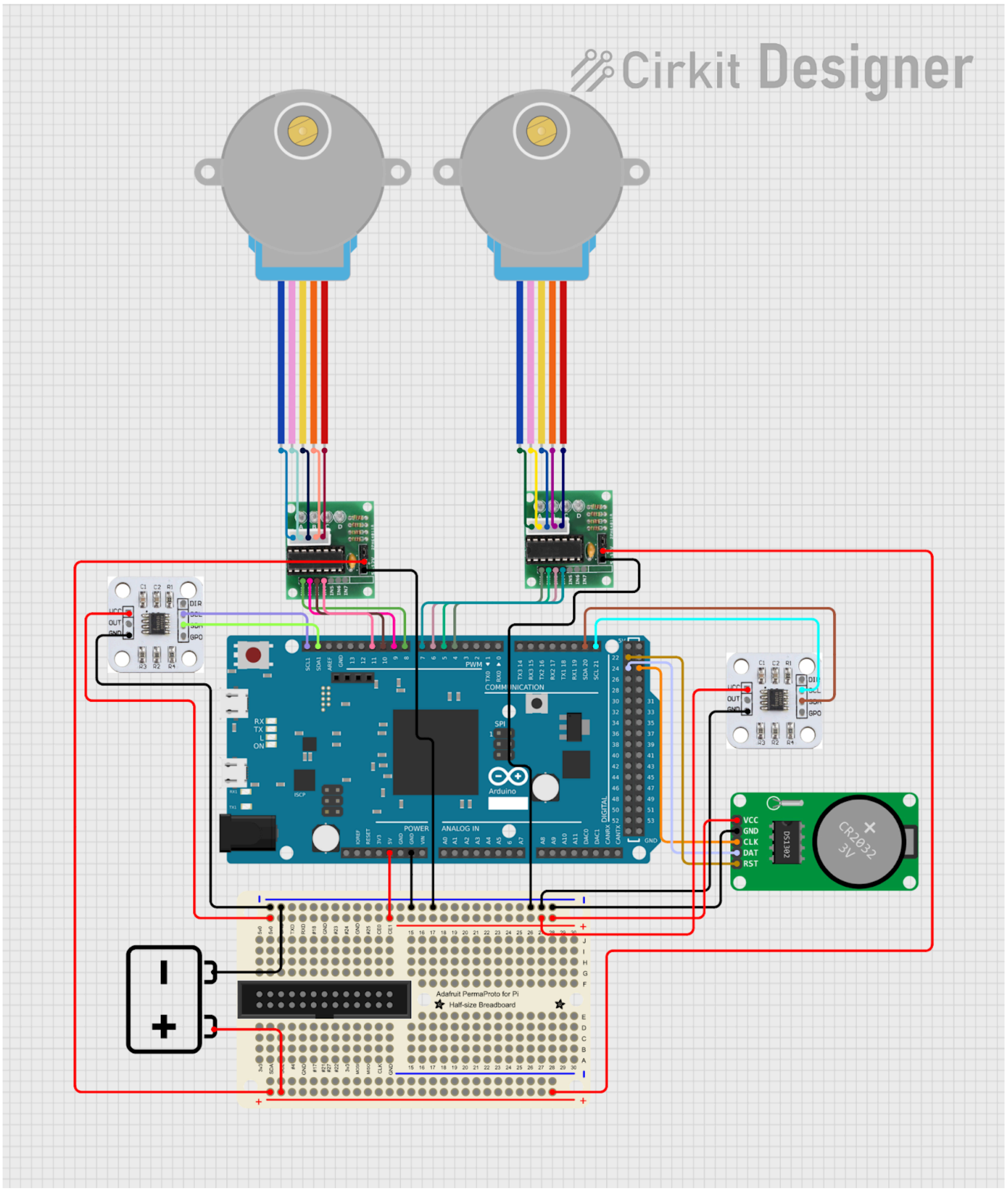
Project Description

- **Objective:**
  - The objective of a star tracker is fairly straightforward. Its purpose is, as its name entails, not only to track stars, but to locate them first in the night sky (which is why we opted for the name *Celestial Browser* rather than the generic alternative). Feed it the coordinates of the object you wish to see, and it will point its scope to the sky, directly at the desired celestial object, be it faraway star, galaxy, or other, so long as it sits still on the celestial sphere.
- **Inspiration:**
  - As was mentioned earlier in the introduction, we both studied Astrophysics this semester, which sparked a certain interest in the study of what lays beyond our solar system. Additionally, both of us are planning to complete an Honors Physics degree at McGill University in the next couple of years, and Astronomy is a major branch of this field. It's no wonder then that we would make up our minds on this particular project as soon as Mikael came up with the idea. As for finding inspiration in other projects, we did take a look at a few similar builds on Instructables, but there was one in particular, built by a user named "Gorkem," from which we got a lot of ideas. That page will be listed in the references.
- **Design and Planning:**
  - We very early on started separating and distributing our respective tasks for the project. I (Michael) took care of pretty much the whole material aspect of the build while Mikael wrote the series of codes that would produce the desired angle values to orient our machine. Because of the motorized aspect of the project, a sort of overall frame, including gears and axes, was needed, so I drew early designs on paper before eventually turning to 3D-modeling for the final model, which I repolished multiple times and designed with scale to fit the motors perfectly. As for the general plan for the functionment of the machine, using stepper motors and rotary encoders (one of each per axis) was really a no-brainer once we figured out how those chips worked.

Implementation

- **Hardware Setup:**
  - 1x Arduino Due

- 2x 28BYJ-48 stepper motor
- 2x ULN2003A driver board
- 2x AS5600A magnetic rotary encoder
- 1x Real Time Clock DS1302
- Breadboard and jumper wires
- 1x thrust bearing + 3x simple ball bearings
- PLA 3D-printed frame
- Eyepiece



- **Software Development:**
    - Discuss the process of writing the code.

      The first part of the code I (Michael) wrote consists of the interaction between the user and the serial monitor. The goal is to obtain the four values that must be obtained through the manual input of the user: first, his own coordinates (latitude and longitude), which will not be asked for again unless the code is reset, and then the *permanent* celestial coordinates of the object he wishes to observe (its right ascension and declination), which the user can change at any time, simply by inputting new ones. To achieve this, I used a system of step integers within the loop, by way of if-statements corresponding to specific values of the integer, values which would change at the end of the first step to advance to the next. Within those steps, there are first demands made by way of serial printing, and then a string function destined to read the required coordinates in a specific format, each corresponding to a set of integers later converted to a float variable (for example, the earth coordinates would come in the format `Lat=deg,arcmin,arcsec,N/S Long=deg,arcmin,arcsec,E/W` and would then be converted to two floats Latitude and Longitude, with the overall value in degrees being positive if North/East and negative otherwise). Here is, as an example, the first step of the code within the loop, responsible for the input of Earth coordinates:

```
//within void setup; initial instructions printed on serial monitor

Serial.println("Enter your current latitude and longitude in the format:");

Serial.println("Lat=deg,arcmin,arcsec,N/S Long=deg,arcmin,arcsec,E/W");

-------------------------------------------------------------------------------------------------------------------------------------------

//within loop

while (step == 1) {          //1st iteration within loop (step=1 from setup) : get latitude and longitude

   if (Serial.available()) {              //if there is input to be read from serial monitor, proceed

     String input = Serial.readStringUntil('\n');       //read input

     input.trim();


     if (sscanf(input.c_str(),           //if follows correct format, set integer values for lat and long

           "Lat=%d,%d,%d,%c Long=%d,%d,%d,%c",          // desired format of input

           &latDeg, &latMin, &latSec, &latDir,                    //set int values and char values for direction

           &longDeg, &longMin, &longSec, &longDir)

       == 8) {
        //set final float values for lat and long from integer subvalues and sign from direction chars

        latitude = latDeg + latMin / 60.0 + latSec / 3600.0;

        longitude = longDeg + longMin / 60.0 + longSec / 3600.0;

        if (latDir == 'S' || latDir == 's') latitude = -latitude;

        if (longDir == 'W' || longDir == 'w') longitude = -longitude;


        Serial.println("Current coordinates:");          //print results

        Serial.printf("Lat: %d°%d'%d\" %c\n", latDeg, latMin, latSec, latDir);

        Serial.printf("Long: %d°%d'%d\" %c\n", longDeg, longMin, longSec, longDir);


        Serial.println("Enter the celestial coordinates of the object:");          //instructions for next step

        Serial.println("RA=deg,arcmin,arcsec Dec=hour,min,sec");


        step = 2;                 //send to next step in code (celestial coordinates)

      } else {                    //error message for invalid format

        Serial.println("Invalid format. Try: Lat=deg,arcmin,arcsec,N/S Long=deg,arcmin,arcsec,E/W");

      }

    }

  }
```

Once the user inputs the coordinate values, the second part of the code would take the current time using the Real Time Clock module. This part of the code is very short as it does not require many functions. It was written separately from the previous first part of the code and for now only prints the time it gets just for verification. Normally, these time values would’ve been sent to the third part of the code (in Python) for conversion.

```
// CLK pin = 25

// DAT pin = 24
// RST pin = 22


ThreeWire wire(24,25,22); // wire(DAT, CLK, RST)

RtcDS1302<ThreeWire> rtc(wire);  //create rtc object


void setup() {

  Serial.begin(9600);

  delay(1000);


  rtc.Begin();


  if (!rtc.GetIsRunning()) {

    Serial.println("Clock not running. Starting now...");

    rtc.SetIsRunning(true);

    delay(1000);

  } else {

    Serial.println("Clock is running...");

    delay(1000);

  }


  if (rtc.GetIsWriteProtected()) {

    Serial.println("Enabling writing...");

    delay(1000);

  }
  // Serial.println(__DATE__);

  // Serial.println(__TIME__);


  RtcDateTime compiled = RtcDateTime(__DATE__, __TIME__);

  if (!rtc.IsDateTimeValid()) {

    Serial.println("Updating Date & Time... (invalid)");

    rtc.SetDateTime(compiled);

    delay(1000);

  }


  RtcDateTime live = rtc.GetDateTime();

  if (live < compiled) {  // rtc is older (not updated to current time)

    Serial.println("Updating Date & Time... (older)");

    rtc.SetDateTime(compiled);

    delay(1000);

  }

}


void loop() {


  RtcDateTime live = rtc.GetDateTime();

  double year = live.Year();

  Serial.print("Year: ");

  Serial.println(year);

  double month = live.Month();
  Serial.print("Month: ");

  Serial.println(month);

  double day = live.Day();

  Serial.print("Day: ");

  Serial.println(day);

  double hour = live.Hour();

  Serial.print("Hour: ");

  Serial.println(hour);

  double min = live.Minute();

  Serial.print("Minute: ");

  Serial.println(min);

  double sec = live.Second();

  Serial.print("Second: ");

  Serial.println(sec);


  while (1);

}
```

The third part of the code is mostly math and this was written in python for a better accuracy of the numbers. The Arduino float caps at 6-7 digits of precision, but the calculations needed more, and so this is why the code for conversion was written in python so we could use the long data type for better precision. However, we did not know how to send the previously obtained variables of time and coordinates to this python code, which is why it is only a test code for now (using test variables).

```
year = month = day = hour = min = sec = 0.0;

Jdate = Gyear = Gmonth = Gday = Uhour = Umin = Usec = 0.0;

Gst = Lst = 0.0;

RAhour = RAmin = RAsec = DECdeg = DECmin = DECsec = Ha = 0.0;

timezone = 0;
```

```
longitude = 0.0;

latitude = 0.0;


def trunc(x):

    x = math.trunc(x);

    return x;


def getJD(y, m, d):

    A = B = C = D = 0.0;


    if m == 1 or m == 2:

        y -= 1;

        m += 12;


    if y > 1582:

        A = trunc(y/100);

        B = 2 - A + trunc(A/4);

    elif y == 1582:

        if m > 10:

            A = trunc(y/100);

            B = 2 - A + trunc(A/4);

        elif m == 10:

            if d >= 15:

                A = trunc(y/100);

                B = 2 - A + trunc(A/4);
    if y < 0:
        C = trunc((365.25 * y) - 0.75);
    else:
        C = trunc(365.25 * y);

    D = trunc(30.6001 * (m + 1));

    Jdate = B + C + D + d + 1720994.5;

    return Jdate;

def ungetJD(jd):
    A = B = C = D = E = G = I = F = 0.0;
    jd += 0.5;

    I = trunc(jd);
    F = jd - trunc(jd);

    print(I);
    print(F);

    if I >= 2299160:
        A = trunc((I - 1867216.25)/36524.25);
        B = I + A - trunc(A/4) + 1;
    else:
        B = I;

    C = B + 1524;
    D = trunc((C - 122.1)/365.25);
    E = trunc(365.25 * D);
    G = trunc((C - E)/30.6001);

    Gday = C - E + F - trunc(30.6001 * G);

    if G > 13.5:
        Gmonth = G - 13;
    else:
        Gmonth = G - 1;

    if Gmonth > 2.5:
        Gyear = D - 4716;
    else:
        Gyear = D - 4715;

    return (Gday, Gmonth, Gyear);

def getGST(jd, ut):
    S = T = 0.0;
    S = jd - 2451545.0;

    T = S/36525.0;


    T = 6.697374558 + (2400.051336 * T) + (0.000025862 * math.pow(T, 2));


    while (T < 0):

        T += 24;


    while (T - 24 > 0):

        T -= 24;


    print(T);


    ut *= 1.002737909;


    Gst = ut + T;


    if Gst >= 24:

        Gst -= 24;

    elif Gst < 0:

        Gst += 24;


    return Gst;


def getLST(gst, long):
    x = long[0];


    if "W" in long:
        x *= -1;


    x /= 15;


    Lst = gst + x;


    if Lst < 0:
        Lst += 24;
    elif Lst - 24 > 0:
        Lst -= 24;


    return Lst;


# test e.x.
timezone = -4;
year = 1980;
month = 4;
day = 22;
hour = 14;
min = 36;
sec = 51.67;
longitude = (64, "W");
```

```
RAhour = 18;

RAmin = 32;

RAsec = 21;


""" if (month >= 3 and month <= 10):

   hour -= 1; """


hour = hour + (min/60) + (sec/3600);


Uhour = hour - timezone;


Gday = day + (Uhour/24);


Jdate = getJD(year, month, Gday);


Gday, Gmonth, Gyear = ungetJD(Jdate);


Uhour = (Gday - trunc(Gday)) * 24; # UT hours in decimals


Gday = trunc(Gday); # remove decimal hours


Jdate = getJD(Gyear, Gmonth, Gday); # find Jdate at 0h on Greenwhich calendar date


Gst = getGST(Jdate, Uhour); # find GST in decimal hours
```

```
Lst = getLST(Gst, longitude); # find LST in decimal hours

RAhour = RAhour + (RAmin/60) + (RAsec/3600); # RA in decimal hours

Ha = Lst - RAhour;

if Ha < 0:
   Ha += 24;

print(Uhour);
print(Jdate);
print(Gday);
print(Gst);
print(Lst);
print(Ha);
```

The third part stops here, as for further conversion, we needed to obtain the values of coordinates to finalize the conversion.

Once the final Azimuth and Altitude values are obtained, it would be time for the robot to move so as to point towards them. This is where the current major lack of our project most shows. We do not account for calibration. Instead, the code simply assumes that the machine is already calibrated and that the (0°, 0°) coordinates correspond to 0° altitude, full north. It then uses a function called "MoveToAngle" created earlier that simply moves the stepper until the encoder reaches the desired angle value, on both axes.

```
 while (coostatus == 1) {            //coordinates are obtained - move the robot (end of the loop step)
   // Here would go the coordinate conversion code
   float targetAzimuth = XX;         //only an example, values are obtained from the above missing code
   float targetAltitude = XX;
   Serial.println("Moving to target position...");
   //activate movetoangle function twice, each wrt specific stepper, encoder, target angle, and axis enum
   moveToAngle(stepperazi, Wire, targetAzimuth, AZIMUTH);
   moveToAngle(stepperalt, Wire, targetAltitude, ALTITUDE);


   Serial.println("Target acquired.");        //success message
   coostatus = 0;             //end iteration - will reset only if step 2 is run again; only if new values are input
  }
```

```
//individually created function outside of loop & setup - is specific to one stepper, one encoder (wire),
one desired angle, one axis enum - will be run for each axis setup this way
void moveToAngle(Stepper &stepper, TwoWire &bus, float targetAngle, AxisType axis) {
  while (true) {                //run until objective achieved
//define angular position from respective encoder reading
   float currentAngle = readAngle(bus) * 360.0 / 4096.0;
   float error = targetAngle - currentAngle;          //find difference with target angle
   if (error > 180) error -= 360;
   if (error < -180) error += 360;
//when aligned (i.e. difference is within acceptable error range), stop code
   if (abs(error) <= angleTolerance) break;


   int direction = (error > 0) ? 1 : -1;         //define direction of motion
   stepper.step(direction);           //move stepper in direction of motion (continues until break above)
   delay(5);
  }
```

```
 if (axis == AZIMUTH)     //use enum to identify and print on serial monitor axis aligned when code ends

  Serial.println("Azimuth aligned");

 else

  Serial.println("Altitude aligned");

}
```

- Libraries :
    - **Stepper** (needed to control stepper motors)
    - **Wire** (enables use of the I^2C buses)
    - **Arduino SAM Boards** (needed to use the Arduino Due)
    - **RtcDS1302** (to use the RTC module of this specific model)
    - **ThreeWire** (for communication between the RTC and the Arduino)

- **Sensor Documentation:**
    - For this project, we used only one type of sensor; the AS5600A magnetic encoder. We used one for each axis and they were connected to the two I^2C buses on the Arduino Due. The function to read the angle was defined as so:

```
//function to read “raw” angle value (0-4095) with reference to one of two wires (Wire1 & Wire) each
corresponding to a specific encoder

uint16_t readAngle(TwoWire &bus) {

 bus.beginTransmission(AS5600_ADDR);

 bus.write(0x0E);

 bus.endTransmission();

 bus.requestFrom(AS5600_ADDR, 2);

 uint8_t msb = bus.read();

 uint8_t lsb = bus.read();

 return ((msb & 0x0F) << 8) | lsb;

}
```

Further on, within the created “MoveToAngle”, the above function would be used with respect to a specific wire (and thus encoder), and the resulting angle, serving to track the angular position of an axis of the robot, would be converted to degree units, like follows:

```
float currentAngle = readAngle(bus) * 360.0 / 4096.0;
```

Results

There was no collection of data involved in our project; thus, no results to present.

Analysis:

Again, no data, no results, no analysis. We simply sought to create a functioning tool destined for individual use, with no sort of data tracking/collecting.

Discussion and Conclusion

The project was difficult and we encountered many obstacles. At first, the hardware we had didn't fit our needs; as an example, DC motors couldn't stay in a fixed position, and so we had to change to stepper motors. Moreover, it was late until we figured that the power required for the motors was bigger than what we had. We had many problems with our code (most of it due to our inexperience), such as the lack of precision of the Arduino, which we fixed by doing part of the code in python. However, the python code went well, and the math worked successfully. In fact, the 3D design of our project was also successful; the printed pieces fitted perfectly and we didn't need to use the printer twice.

Initially, the purpose of the project was to have a star pointer (or "tracker") such that we could find any star we wanted in the night sky. This would've helped us study the night sky and understand it more thanks to a concrete, real, and visual approach such as this project. We had a lot of inspiration from our astrophysics class with Rim Dib. And so we knew we needed two motors, with two encoders, so that we could have our pointer orient itself in a three axis system. Then, we "invented" the design entirely by ourselves (with maybe some inspiration from Gorkem - *see references*), as we tried to find the simplest way to create that "3d system" for our pointer. And from that, only the code part of our project remained, with the mathematical conversion of coordinates and the control of the stepper motors. Without being able to finalize this project in time, there are no comments to make on results, as there are none.

However, we find many possible improvements/extensions to this project. In addition to completing it, there is a possibility for us to automatically track the pointed star by regularly updating the coordinates of it (like in a loop) so that the pointer follows its path. The addition of a self-calibration system involving a compass chip and a leveling system would complete our project perfectly as well. All in all, despite the challenges encountered along the way, this has been a great learning experience and will surely help us in our further academic pathways.

References

Gorkem. "Star Track." *Full DIY Guide*, gorkem.cc/projects/StarTrack/. Accessed 28 Feb. 2025.

“Inter-Integrated Circuit (I2C) Protocol.” *Docs.Arduino.Cc*, docs.arduino.cc/learn/communication/wire/. Accessed 30 Apr. 2025.

Duffett-Smith, Peter, and Jonathan Zwart. *Practical Astronomy with Your Calculator or Spreadsheet*. 4th ed., Cambridge University Press, 2011.

Makuna. *RTC Library Wiki*. GitHub, https://github.com/Makuna/Rtc/wiki.

Acknowledgements

Obviously, we’d both like to thank Joel for directing the project, providing us what we needed and giving help when required, so a big thanks for that. Also, we should give thanks to Dawson College for funding the project. Finally, I (Michael) also want to thank my dad for helping me out a lot with the assembly of our overall product.