Climbing Clock Project Breakdown

A Complete Beginner's Guide and Reference Book

Author: Harrison Outram

Date: Thursday, 16 April 2020

Version History

Version	Author/Editor	Changes Made
0.1	Harrison Outram	Initial version. Missing mechanical/civil designs and Time Scale electronics design.

Table of Contents

List of Figures and Tables

No table of figures entries found.

1. Project Goals and Scope

1.1. Overview

Overall Goal	Create one or more robots that tell the time based on how far they have climbed up (see Figure 1 on next page)
Theme	Tortoise and the Hare.
Functional Requirements	1. Must be able to tell the time by looking at how far robot(s) have climbed up. 2. Must hang from second floor catwalk in building 215.
Non-functional requirements (NFRs)	1. Must be able to tell the time within 5 minutes error from up to 10 metres away with the naked eye 2. Robots must move using internal mechanisms (no pulleys!) 3. Entire system cannot be taller than 2.4 metres. 4. Entire system cannot be wider than 1 metre. 5. Must be a fully autonomous system (no supervision nor human intervention required). 6. Must take no more than 2 hours to assemble or dissemble by one person with appropriate training and toolset. 7. Must take at least 1 hour to maliciously break without project access (e.g. custom made keys for padlocks or keyholes). 8. Noises produced must be quiet and non-distracting from at least 1 metre away. 9. Must take no more than 20 minutes to diagnose any possible fault. 10. Must last at least one semester without maintenance. 11. It has to be interesting to look at (subject to interpretation).
CRoC Resources	*All CRoC resources are available on request, as the club room is closed to non-committee members.* 1. Arduinos and associated electronics (see project inventory on Google Drive) 2. Screwdriver sets 3. 3D printers a. FlashForge Forward (140*140*140 mm, ±0.2 mm) b. FlashForge Creator Pro (227*148*150 mm, ±0.2 mm)

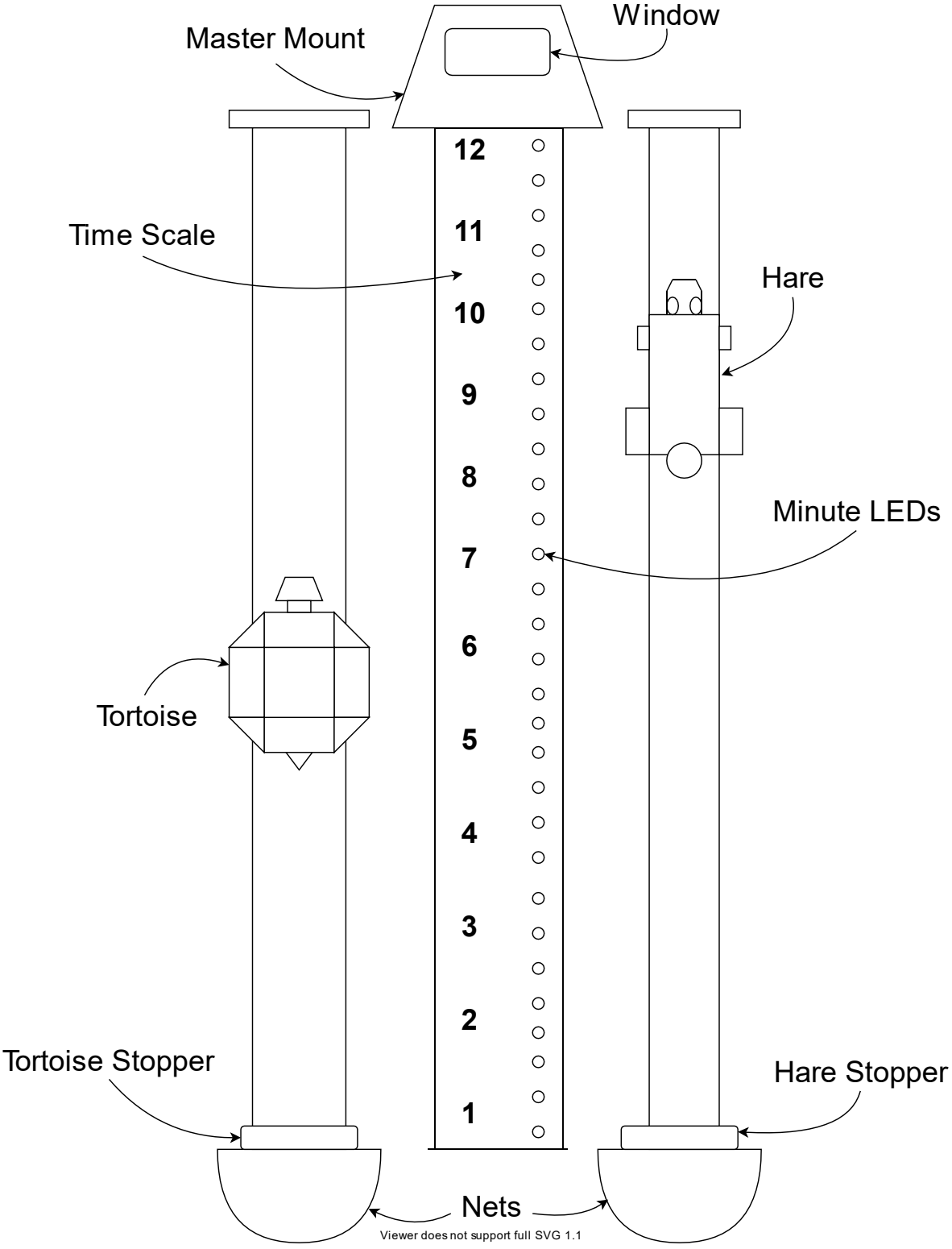

Figure 1: Rough drawing of entire system. Subject to change. Not to scale. Stoppers have not been designed. The window is where the master Arduino will be visible. Made in draw.io (v12.9.3).

1.2. Roles and Specialities

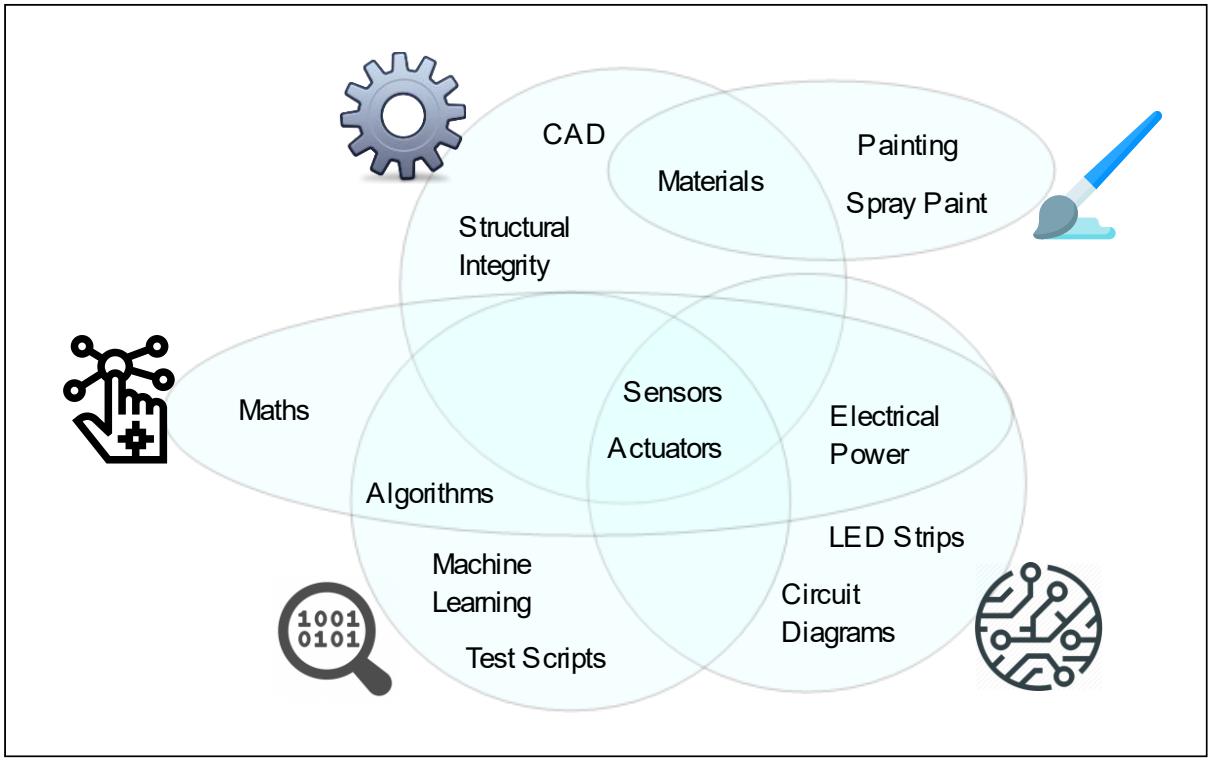

Figure 2: Roles and tasks Venn diagram. The gear icon represents mechanical and civil, the circuit represents electronics, the binary magnifying glass represents programming, the paintbrush represents the aesthetics, and the digital hand touching the network represents simulation. Logistics is not shown since it covers everything. Made in Inkscape (v0.92.4).

2. Team Communication and Management

Communication Methods	1. CRoC Discord (outside build nights) 2. In-person (during build nights) 3. CRoC Facebook page (updates to build night schedule)
File Repositories	1. Generic: Climbing Clock Google Drive (requires Google Drive compatible email)[a] 2. Code: Climbing Clock Github repository.[a]
Progress Tracking	1. Trello Kanban board. [a] 2. Version control spreadsheet (for CAD drawings).
Task Delegation	Trello Kanban board [a] and negotiation with other members.

Note 1: [a]See project lead for access.

3. Programming

3.1. Overview

Duties	1. Design and review code structure (see Figure 3 on next page). 2. Research how to control electronics. 3. Write scripts for controlling different subsystems. 4. For each library, write a test script (see test harness template on Github). 5. Write both internal and external documentation.
Programming Language	Custom C++ compiler made for Arduino IDE. • Does not have libraries in other C++ compilers. • No object oriented error handling: no try, catch, finally, nor throw statements. • Full list of built-in functions: https://www.arduino.cc/reference/en/ • Full list of built-in libraries: https://www.arduino.cc/en/Reference/Libraries
Required Software	1. Arduino IDE (free to download) 2. Github (online git repository)
Recommended Software	1. Visual Studio Code (free, open source, and simple IDE) 2. Git bash (Git and Bash terminal for Windows)

3.2. Code Flow and Overall Structure

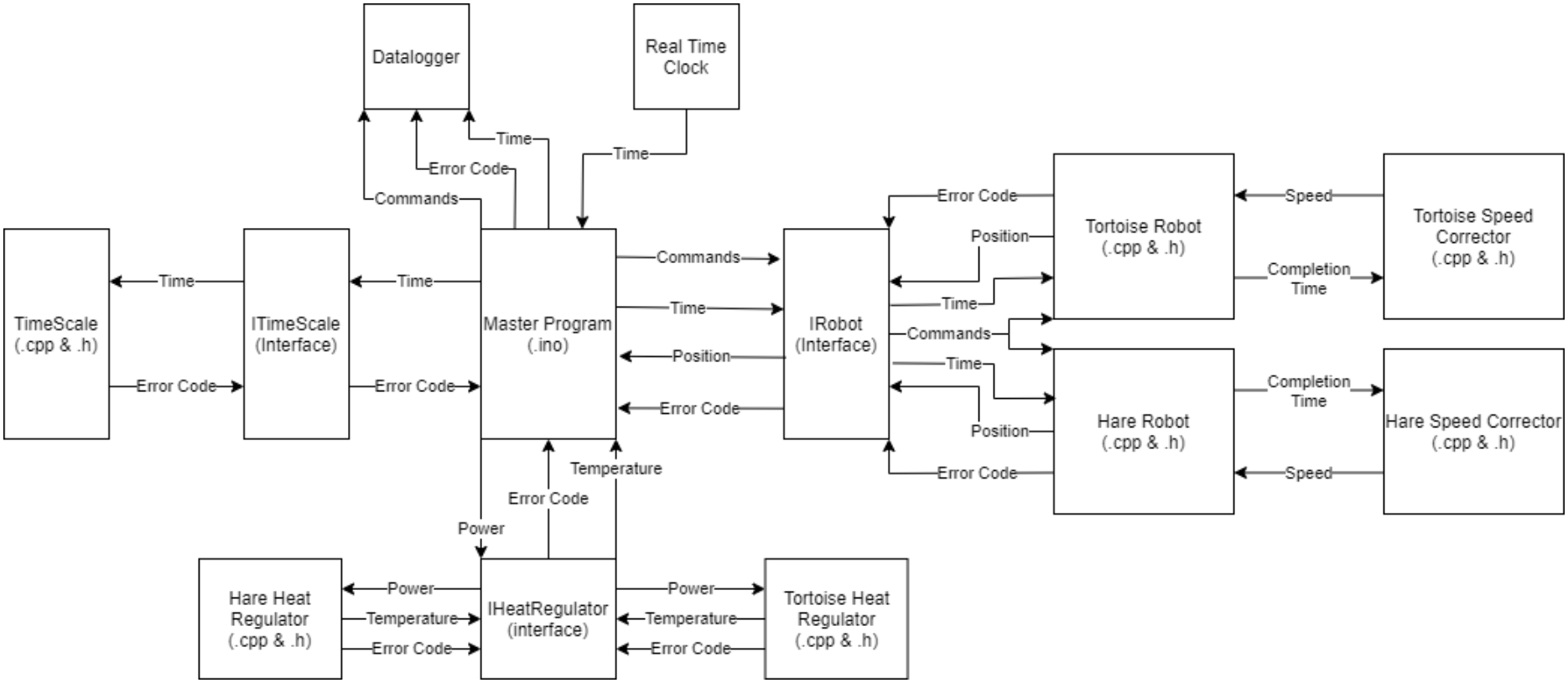

Figure 3: Code flow diagram. Each rectangle represents a file or pair of files, whereas arrows represent the flow of data between files. The “Power” arrows for the heat regulators refer to how high the cooling system is set to. Made in draw.io (v12.9.3).

3.3. Programming Standards

Whether a novice or expert, programming standards must be established to enable consistently formatted and readable code. Appendix A shows acceptable and unacceptable examples of code using the rules below.

1. *Always* follow the design.
    a. If the design can be improved, is impractical/impossible, or contradictory, speak with the project lead to change the design *before* writing code that conflicts with the design.
2. Place comments whenever it is not obvious as to what a block of code's purpose is.
3. All public and protected functions, constants, and global variables must have documentation comments (see Doxygen[1]).
4. All libraries must come with a `README.md` file with a full white-box breakdown of the library,
    a. Must be formatted as per Markdown.[2]
    b. Must include a title, table of contents, project details including `README` version, library purpose, dependencies (both software and hardware), how-to guide with step-by-step instructions of usage, public class constants explanations, public and protected methods explanations, functions purpose, external materials used, and miscellaneous sections.
5. All libraries must come with a `keywords.txt` file, as per the official Arduino instructions.[3]
6. All names should be short (< 20 characters) and descriptive.
    a. Constants should be spelt in UPPERCASE with underscores (_) to separate words.
    b. Classes, structs, custom datatypes (using `typedef`), and namespaces should be spelt in TitleCase.
    c. Variables, object, and functions/methods should be spelt in camelCase or lowercase with underscores (_) separating words.
    d. Data members (what C++ calls the variables of an object) should start with a single underscore.
7. Avoid floats, division, trig functions, and other computationally expensive functions and operations, unless otherwise no better alternative exists.
8. Avoid using `delay()` as the Arduino must control multiple subsystems simultaneously.
9. Use bitwise operators whenever applicable.
10. All operators and operands must have spaces inbetween.
    a. This excludes *some* uses of brackets, as shown in Appendix A.

[1] http://doxygen.nl/
[2] Markdown cheat sheet: https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet
[3] Arduino guide to making a library: https://www.arduino.cc/en/Hacking/LibraryTutorial

11. All functions and methods should use the minimum number of operators required while retaining readability.
    a. If a function/method is too large (> 50 lines) split it into multiple functions/methods.
12. All code written must be reviewed by a non-author programmer.
13. No more than one statement per line, except when using `for` loops.
14. *Absolutely* **never** use the `goto` statement.
15. Always pick the appropriate loop type, e.g. do not treat `for` loops as `while` loops and vice-versa.

This standards will inevitably be biased towards the project lead's personal preferences. To mitigate this, others may challenge the project lead's rules.

3.4. Design Specifics and Justifications

In an attempt to create a clean, maintainable, and readable code base, all code is first derived from the code flow diagram (Figure 3). As per this diagram, the code is split into different files or pairs of files such that each file controls a different subsystem. These subsystems are further detailed below with justifications to their designs. This section does not provide the algorithms nor the source code used as such information can be obtained from reading the source code or `README` files.

3.4.1 Real Time Clock Code Design

The Real Time Clock (RTC) can be controlled through the popular `RTClib` library.[4] Using this library, one does not need to know the registers nor the encoded date sent from the RTC, just the classes and the methods. All the programmer needs to do is create an RTC object based on the RTC being used (DS1307), start the RTC using `rtc.begin()`, set the time using `rtc.adjust(DateTime(F(__DATE__), F(__TIME__)))`, then use `rtc.now()` to get the current date and time.

However, the current version (1.4.1) uses a very inefficient `DateTime` class for setting and getting the date and time. This class stores the date and time as a year, month, day, hour, minutes, and seconds (which shall be referred to as "verbose time") instead of simply using Unix time. Our code will frequently check to see if enough time has passed for the current cycle (12 hours for the tortoise and 60 mins for the hare), which requires the `DateTime` object to convert its time to Unix time, involving numerous modulo and division operations. Additionally, when the robot objects sets the new finish time for the next cycle the `DateTime` constructor used converts a Unix time to verbose time, requiring numerous multiplication and addition operations. To mitigate this inefficiency, a custom version of the library is being made that makes the `UnixTime` class store the date and time in Unix time.

[4] See https://github.com/adafruit/RTClib

Unfortunately, the RTCs themselves store the date and time as verbose time. This is accomplished using Binary Coded Decimal (BCD) for all known RTC devices.[5] Hence, when checking if enough time has passed either a `DateTime` object must be created or a `UnixTime` object must be created by converting the verbose time to a Unix time. To circumvent this, a custom `DateTime` class must be written which compares `DateTime` objects lexicographically: the verbose date times are checked from most significant (the years) to least significant (the seconds) to see where they differ first. Once a difference is found or all data members are the same the Boolean result is known.

3.4.2 Time Scale Interface Justification

The Time Scale, as shown in Figure 1, needs to know the current time of day and return error codes back to the Master program. Since the Time Scale electronics are subject to change, an interface should be used to enforce an implementation agnostic usage and terminology for the Master program to control any arbitrary Time Scale. The methods of which can be found in the README file in the project's Github. Of these methods, one allows the Master program to check if a fault has occurred. If the Time Scale does not have the sensors to detect faults the method will always return a code asserting no fault has occurred.

3.4.3. Robot Interface Justification

There are several ways to design and build climbing clock robots with various actuators and sensors. These could range from simple DC motor and gearbox actuated robots with timing based location detection to jumping hydraulic actuated robots using inverse kinematics, LED displays for different time zones globally, speakers, and a moving head that tracks faces. Obviously, accommodating creativity will result in a large range of complexity, requiring an interface to de-couple the robot objects from the master program.

3.4.4. Speed Corrector Machine Learning Justification

An obvious problem is adjusting the climbing speed of the robots to ensure they reach the top in the correct time. The most reliable and accurate way of adjusting the robot's climbing speed is to use machine learning. Doing so requires the robot to track the time internally (via the RTC), sensors to determine its position (at the bear minimum it needs to know if it's at the top, bottom, or somewhere inbetween), and the current speed. Using this information, the speed can be corrected via determining if the robot should speed up or down based on if it has been climbing too quickly, too slowly, or just right.

[5] Citation pending

The most simple approach is just to change the speed by a constant value if the robot is climbing up too fast or slow. However, this does not consider how close the robot's current speed is to reaching the correct speed. This can result in the robot overcorrecting or taking months to reach the correct speed.

To solve this problem, the robot needs to determine how far it's climbed up against how far it should have climbed up given the time. The ratio between these values is the factor to change the speed by, which when multiplied by the current speed results in the "corrected speed". However, this can allow outliers due to mechanical imperfections (such as motor slip) to cause the next cycle to go at the wrong speed. Solving this problem requires the robot to keep track of a collection of corrected speeds (such as an array of fixed length) so that the next speed will be the result of several corrected speeds. The larger the collection, the less of an impact outliers will cause. Figure 4 visualises this more realistic process. Eventually, the speed corrector will either reach the correct speed or will be the closet integer to the correct speed.

When the robot does not know its exact location (e.g. when enough time has passed that the current cycle has ended but has not reached the top and it doesn't know its exact position), the speed can be changed by a fixed value. Choosing this fixed value is difficult as large values will reach the correct speed faster but can overshoot, whereas small values won't overshoot but can take weeks or months before the correct speed is attained. A better way of dealing with this situation is to start adjusting the speed by a large value that becomes smaller as the value is used, since the robot should become closer and closer to its correct speed. To achieve these goals, all speed corrector libraries should contain a speed corrector class customised for the robot's sensors and actuators.

Climbing clock robots can vary wildly in their designs. Due to this large variance speed correctors should be programmed for each robot class. As shown in Figure 3, the robot objects should be responsible for using the speed correctors, not the Master program. Ergo, it is not possible to design a logical and easy to use interface for speed correctors, hence their absence in Figure 3.

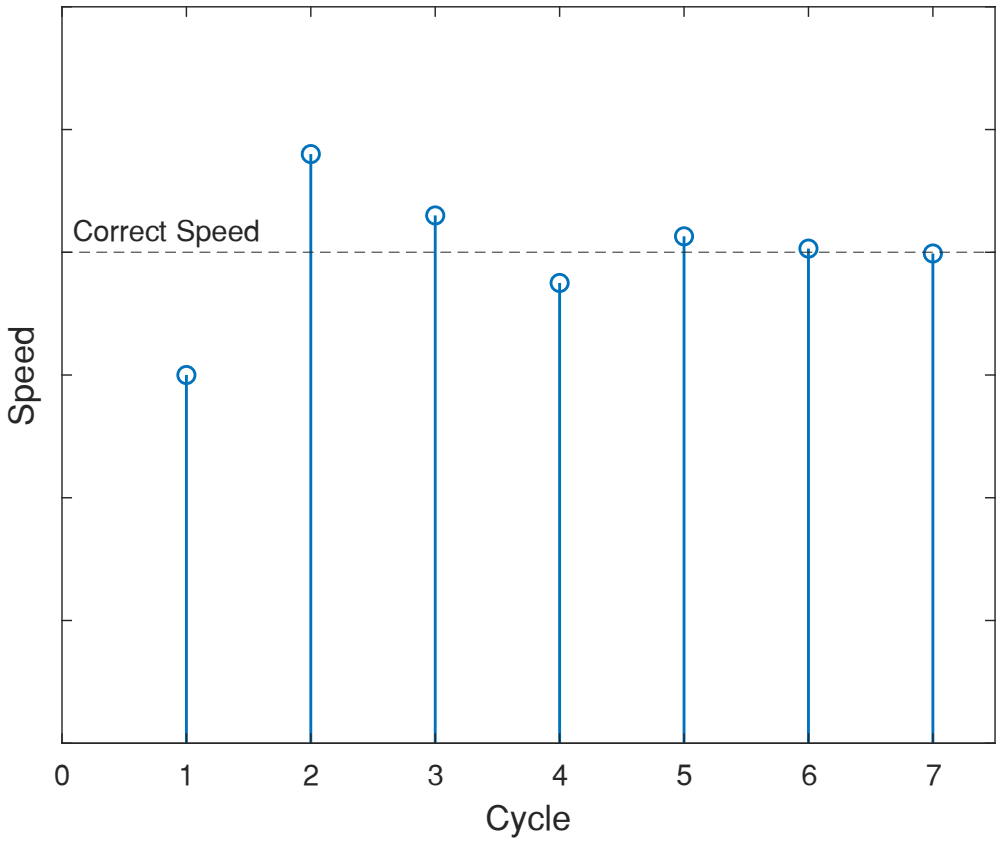

Figure 4: Example of a speed corrector changing the speed closer to the correct speed.

3.4.5. Heat Regulator Interface Justification

There are several ways heat regulators could be designed and implemented. From a simple fan and thermistor to water cooling, heat regulators can vary wildly in their designs and complexities. Fortunately, a white-box approach can be easily established for using an arbitrary heat regulator. This will require a heat regulator class with methods for initialising the hardware, getting the current temperature, getting the cooling setting (or "power") given the temperature, setting the cooling setting, checking to see if a hardware fault has occurred, and an emergency stop method that turns the hardware off. These methods can easily be formalised into an interface, de-coupling the main program from the heat regulators.

3.4.6. Heat Regulator Machine Learning Justification

One of the methods that that the heat regulator interface has is to output the recommended cooling power given the temperature. This begs the question: how does one determine what cooling power should be outputted given the temperature? Obviously, the operating temperature of the components being cooled should dictate the absolute lower and upper limits. Hence, the simplest equation would be to simply scale the power linearly with the temperature, such as

$$P = \min\left\{\max\left\{\frac{P_{\max} - P_{\min}}{T_2 - T_1} \times (T - T_1), P_{\min}\right\}, P_{\max}\right\} \quad (3.1)$$

where P is the cooling power, T is the input temperature, T_1 and T_2 are the minimum and maximum acceptable temperatures, and $P_{\min}$ and $P_{\max}$ are the minimum and maximum power settings ($P_{\min}$ should be the "off" setting). The max function is needed to prevent powers lower than possible for very low temperatures, and the min function is used to prevent power settings above the maximum possible for very high temperatures. However, equation (3.1) does not consider how effective the cooling system would be at reducing or maintaining a given temperature. One could solve this by simply making the slope high so that the cooling system will always keep the robot internals cool. Unfortunately, this will wear out the cooling system faster than necessary. Ergo, a smarter approach than equation (3.1) is required.

One such method would be to use machine learning to allow the Arduino to determine what the appropriate power should be based on data it can collect. As discussed above, this machine learning AI should consider both the temperature increase or decrease required and the wear of the cooling system. I.e. if the temperature is below the acceptable margin turn the cooling system off or keep it very low, if it's within the acceptable margin maintain the temperature, and if it's higher than the acceptable margin make the cooling system lower the temperature. There are several ways of constructing a mathematical model the Arduino could use to learn from; however, one must keep in mind that the Arduinos have extremely limited hardware resources, so a simple model should be used.

3.4.7. Heat Regulator Machine Learning Design

A quadratic model where the coefficient varies as the Arduino collects data is arguably the best model to use as the heat regulator's temperature to power conversion. This would cause the Arduino to set very low powers for low temperatures to increase the temperature when too low, set very high powers to lower the temperature when too high, and keep the temperature roughly the same when in the acceptable margin. Letting the coefficient be ρ and T_0 be the highest temperature at which the power should be set to off (which should be significantly less than the minimum acceptable temperature, T_1), this model can be expressed as equation (3.2) and visualised as per Figure 5:

$$P = \min\{\rho(\max\{T - T_0, 0\})^2 + P_{\min}, P_{\max}\} \quad (3.2)$$

The max function is used to prevent the parabola curving upwards for temperatures less than T_0.

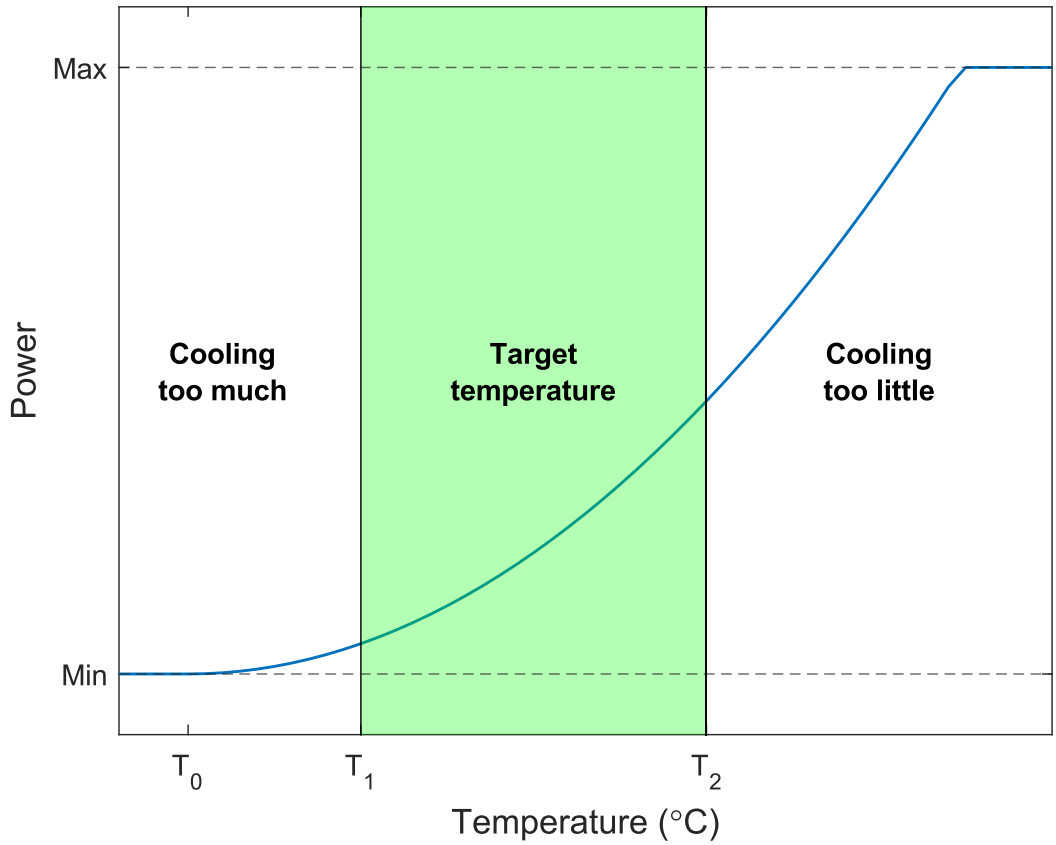

Figure 5: Machine learning model of heat regulator correcting its temperature to power conversion. Based on equation (3.2). Made in MATLAB (R2019b).

The variable ρ can then be changed to ensure the longevity of the cooling system whilst keeping the robot's internals cool. To do so, the heat regulator should track what the previous temperature was so that when the new temperature is found, ρ should be increased if the new temperature is above the acceptable maximum and is greater than the previous temperature, or ρ should be lowered if the new temperature is less than the acceptable minimum and is less than the previous temperature. Caution should be utilised to ensure the inaccuracy of the temperature sensor is considered when evaluating the change in temperature. These rules can be written as

$$\rho' = \begin{cases} \rho + \Delta\rho, & T' - T > \varepsilon_T \wedge T' > T_2 \\ \rho, & T_1 \leq T' \leq T_2 \\ \rho - \Delta\rho, & T' - T < -\varepsilon_T \wedge T' < T_1 \end{cases} \tag{3.3}$$

where ρ' and T' are the new coefficient and temperature values, ρ and T are the previous coefficient and temperature values, $\Delta\rho$ is the amount to change ρ by, and ε_T is the temperature sensor's inaccuracy. Obviously, the value $\Delta\rho$ should be kept small to avoid the machine learning overcorrecting. One such method determining what $\Delta\rho$ should be is to make it large at first, then reduce it every time it is used until it reaches a minimum. Following this model should, at least theoretically, ensure the robot's

internal are kept cool while maximising the lifespan of the cooling system, as per the NFR 10 (section 1.1).

3.4.8. Datalogger Design and Justification

As per NFR 9 (section 1.1), it can take no longer than 20 minutes to diagnose any fault. Considering the high number of components, it can potentially be very difficult to determine what went wrong. Hence, if feasible, sensors should be used where possible to reduce the diagnoses times. The only problem being how does the system output the fault to Climbing Clock personnel?

One such approach is to use a datalogger onto a storage device. Using a SD card, the SD library,[6] and an Arduino compatible board with an SD slot, we can log all commands and error codes to a file. The official Arduino site shows an example of a datalogger logging potentiometer readings to an SD card.[7] This will allow the project to satisfy the 9th NFR and allow Climbing Clock personnel to more quickly get a broken or faulty Climbing Clock up and running.

[6] See https://www.arduino.cc/en/Reference/SD
[7] See https://www.arduino.cc/en/Tutorial/Datalogger

4. Mechanical/Civil

4.1. Overview

Duties	1. Research and test materials 2. Design physical components and subsystems 3. CAD physical components and subsystems 4. Collaborate with electronics team to decide where to fit electronics 5. Research and design movement/locomotion systems. 6. Research and design the implementation of cooling systems. 7. Record CAD file changes in version control spreadsheet.
Required software	1. CAD software capable of making 3D printing compatible files (AutoCAD, Inventor, Fusion 360, SolidWorks, etc.)
Physical Standards	1. Fail safes must be present to prevent injuries and property damage (e.g. nets for catching robots) 2. Components within arm length must be secured from falling off due to being pushed or pulled 3. Components within arm length must have built in security to prevent quick malicious breakage (e.g. padlocks and keyholes) 4. Electronics within arm length must be within containers to prevent ease of access to unauthorised personnel 5. Must be able to assemble or disassemble system into individual parts within two hours by one person.

4.2. Design Specifics and Justifications

4.2.1. Tortoise Design and Justification

To be designed

4.2.2. Hare Design and Justification

To be designed

4.2.3. Time Scale Design and Justification

To be designed

4.2.4. Master Mount Design and Justification

To be designed

4.2.5. Safety Net Design and Justification

To be designed

4.3. Structural Tests

To be designed

5. Electronics

5.1. Overview

Duties	1. Research available electronics and associated documentation. 2. Design Time Scale electronics (e.g. LED strips). 3. Design visible robot electronics (e.g. LED eyes). 4. Design circuit diagrams. 5. Build and test circuits. 6. Collaborate with Mechanical/Civil team to decide where to place electronics.
Required Software	1. Fritzing (free Arduino circuit diagram maker)
Circuit Standards	1. Circuits must be safe (i.e. ensure circuit elements do not exceed power thresholds). 2. Circuit elements must be small enough to fit inside robots. 3. Circuit diagrams must be annotated for circuit element groups and their purposes. 4. Power supply must come from building mains (i.e. no batteries as the main power source).

5.2. Design Specifics and Justifications

5.2.1. Stepper Motor Usage Justification

Even when considering the NFR 2 (section 1.1), there are many conceivable ways to actuate a climbing clock robot. One such method is by using stepper motors rotating a system of gears, resulting in a reaction force pushing the robot upwards. This is appropriate since stepper motors are highly precise, energy efficient (typically 80-90%), and extremely durable. Of these benefits, the durability is most appealing as stepper motors last as long as their internal bearings. This satisfies NFR 10 and reduces ongoing costs, which makes up for the higher initial cost compared to cheaper motors.

5.2.2. Time Scale Electronics Design and Justification

To be designed

5.2.3. Mains Power Source Justification

There are two ways the Climbing Clock could be powered: either through the building's main or from a portable power source. Powering through the building's mains can be achieved via a free power socket next to a touchscreen where the Climbing Clock will be constructed. This will require power cables and a power board connecting the power socket to all the different subsystems. Alternatively, each subsystem could be powered through a portable battery source, such as batteries. Such a power source would impose additional constraints on the physical design to fit the power source, which would have to be designed to ensure ease of replacement. Furthermore, the longevity and of ongoing costs of portable power sources are of great concern, potentially causing the system to fail NFR 10 (section 1.1). Despite the annoyance of managing power cables, it is clear that using the building's mains is the only practical option.

6. Aesthetics

Duties	1. Research and test painting supplies 2. Experiment with art styles, patterns, and colours 3. Collaborate with mechanical/civil team for paintable materials 4. Paint all visible, non-electronic parts
Artistic Standards	1. Art style must be consistent with robotic theme; no photo-realism 2. Colour scheme should be comprised of no more than three colours (not including different shades of the same colour) for each surface 3. Patterns must be consistently applied and low in complexity 4. Multiple patterns on the same surface should be avoided, unless where thematically appropriate 5. Must be able to see patterns from up to 2 metres with 20-20 eyesight

7. Simulation

7.1. Overview

Duties	1. Design block diagram of robots (see Figures 6 and 7) 2. Research mathematical relationships between variables 3. Research and design experiments to determine values of parameters 4. Conduct experiments 5. Program Arduino scripts to measure data 6. Analyse data to determine parameters
Justification	Once a computer model of the robots is complete, we can easily test to see if a design will work without spending months constructing the design.
Required Software	1. Simulink (add-on for MATLAB and free for Curtin students)
Recommended Software	1. MATLAB (for data analysis)
MATLAB Alternatives	1. Excel (free for Curtin students) 2. Google Spreadsheets (online only and free for everyone)
Documentation Standards	1. All experiments designed must have a written (pen & paper or digital) methodology that is easy to understand with appropriate diagrams. 2. All data recorded should be formatted in an unambiguous and legible manner. 3. All code written should be appropriately named, formatted, and commented. 4. All code written should have comment blocks at the top explaining the purpose and experiment name/index. 5. All signals and systems within any block diagram should be appropriately named, including units of measurement. 6. All subsystems should have appropriate parameters for ease of use. 7. No more than ± 5% inaccuracy.

7.2. Block Diagrams

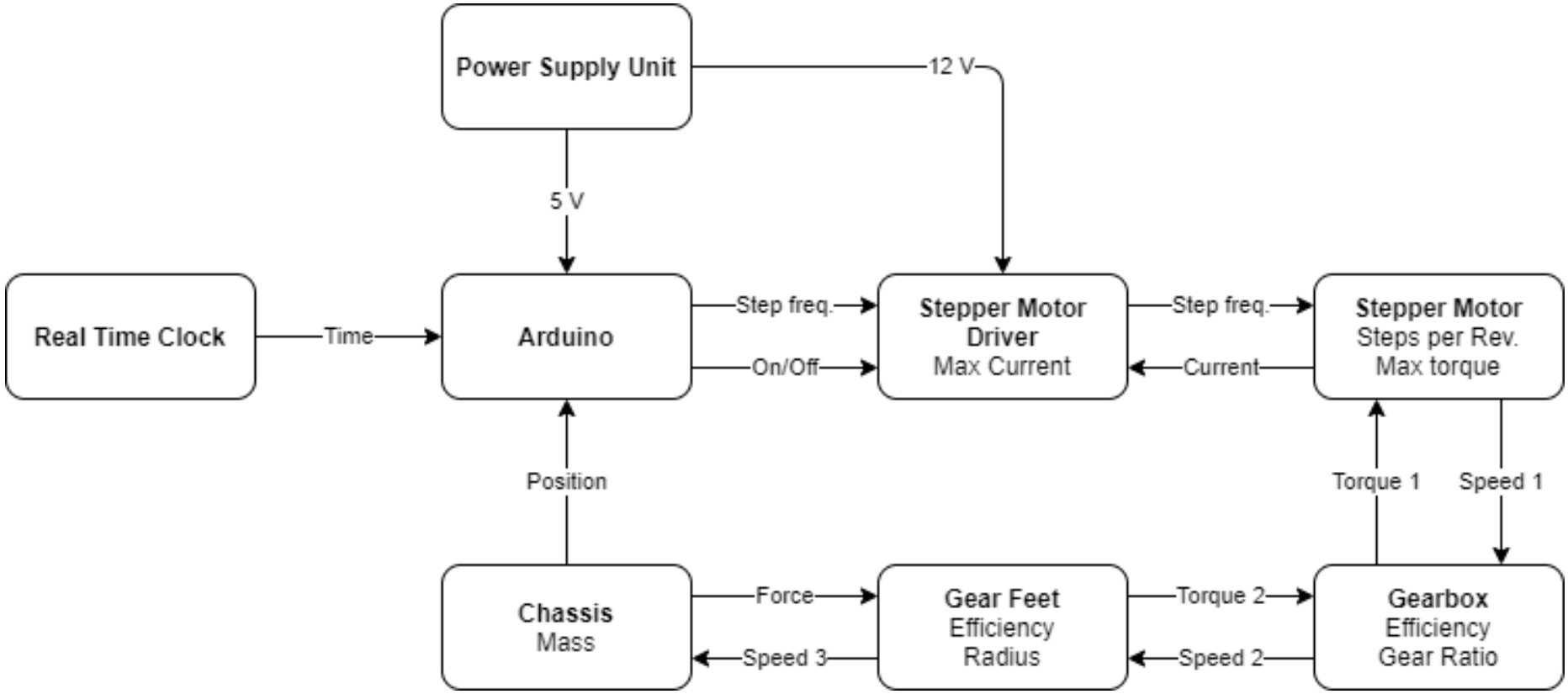

Figure 6: Reference block diagram for a stepper motor actuated robot. Text in **bold** is the name of the subsystem whereas the non-bold text directly below are the subsystem's parameters.

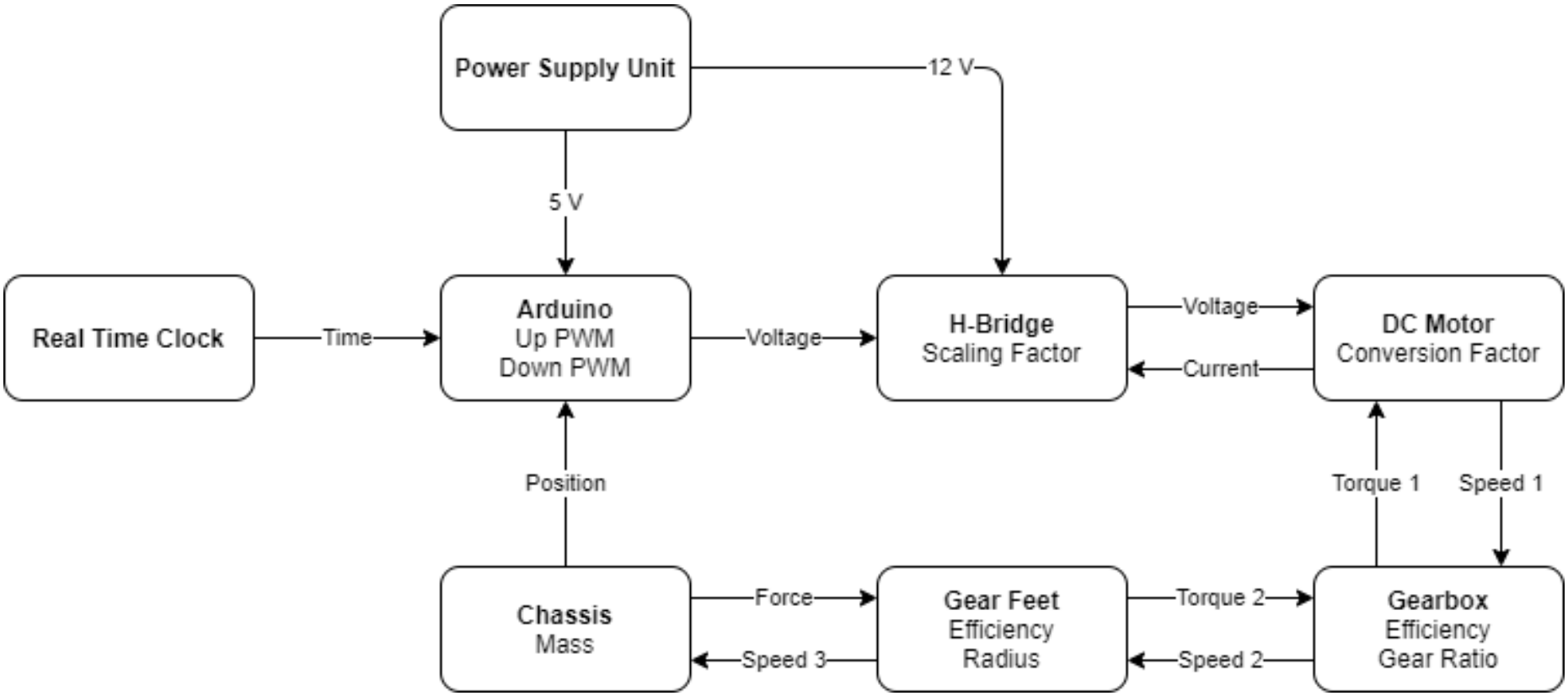

Figure 7: Block diagram of DC motor actuated robot. Text in **bold** is the name of the subsystem whereas the non-bold text directly below are the subsystem's parameters.

7.3. Known Equations

The following is a collection of all equations required for the block diagrams shown above.

7.3.1. Arduino Equations

Notice the following equations ignore the speed corrector discussed in section 3. The following equations use fixed speeds to avoid having to write the same algorithm in multiple different languages. Furthermore, the purpose of the simulation is to see if the robot design will climb at all, not if it can do so at the correct speed. The simulation's speeds should be changed manually if one wishes to correct the speed.

$$v_{\text{Ard}} = \begin{cases} (5\text{ V})d_{\text{up}}, & \text{if going up} \\ 0, & \text{if going down} \end{cases} \tag{7.1}$$

s.t. v_{Ard} is the Arduino's output voltage and d_{up} is the Arduino's up PWM. Used for controlling a DC motor.

$$\omega_{\text{Ard}} = \begin{cases} u, & \text{if going up} \\ 0, & \text{if going down} \end{cases} \tag{7.2}$$

s.t. ω_{Ard} is the output step frequency and u is the chosen step frequency for climbing upwards. Used for controlling a stepper motor. If going down, ensure the "On/Off" signal is set to off and vice versa (see Figure 6).

7.3.2. Stepper Motor Driver Equations

$$\omega_{\text{Ard}} = \omega_{\text{driver}} \tag{7.3}$$

s.t. ω_{Ard} is the Arduino's output step frequency and ω_{driver} is the driver's output step frequency.

$$\text{status} = \begin{cases} \text{Pass}, & i_m \leq i_{\max} \\ \text{Fail}, & i_m > i_{\max} \end{cases} \tag{7.4}$$

s.t. status is whether the simulation has failed or passed, i_m is the stepper motor's current draw, and $i_{\max}$ is the maximum current draw the stepper motor driver can facilitate.

7.3.3. H-Bridge Equations

$$v_{\text{H}} = k_{\text{H}} v_{\text{Ard}} \tag{7.5}$$

s.t. v_{H} is the output voltage of the H-bridge, k_{H} is the scaling factor (which, for Figure 7, will be $12\text{ V}/5\text{ V} = 2.4$), and v_{Ard} is the input voltage from the Arduino.

$$\text{status} = \begin{cases} \text{Pass}, & i_m \leq i_{\max} \\ \text{Fail}, & i_m > i_{\max} \end{cases} \tag{7.6}$$

s.t. status is whether the simulation has failed or passed, i_m is the DC motor's current draw, and $i_{\max}$ is the maximum current draw the H-bridge can facilitate.

7.3.4. Motor Equations

$$\omega_m = \frac{2\pi\omega_{\text{driver}}}{r_m} \tag{7.7}$$

s.t. ω_m is the stepper motor's rotational speed in rad/s, ω_{driver} is the stepper motor's step frequency in steps per second, and r_m is the stepper motor's steps per revolution.

$$\text{status} = \begin{cases} \text{Pass}, & \tau_g \leq \tau_{\max} \\ \text{Fail}, & \tau_g > \tau_{\max} \end{cases} \tag{7.8}$$

s.t. τ_g is the gearbox's required torque and $\tau_{\max}$ is the maximum torque the stepper motor can produce.

$$i_m = k_m \tau_g \tag{7.9}$$

s.t. i_m is the motor's current draw (both stepper and DC), k_m is a conversion factor inherit to the motor, and τ_g is the gearbox's required torque.

$$\omega_m = k_m v_{\text{H}} \tag{7.10}$$

s.t. ω_m is the DC motor's rotational speed in rad/s, v_{H} is the voltage supplied by the H-bridge, and k_m is the same as in equation (7.9).

7.3.5. Gearbox Equations

$$\omega_g = r_g \omega_m \tag{7.11}$$

s.t. ω_g is the gearbox's output rotational speed, r_g is the gearbox's gear ratio from the motor to the gear feet, and ω_m is the motor's rotational speed.

$$\tau_m = \frac{r_g \tau_g}{\mu_g} \tag{7.12}$$

s.t. τ_m is the gearbox's required torque from the motor, τ_g is the feet gear's required torque from the gearbox, and μ_g is the power efficiency of the gearbox.

7.3.6. Gear Feet Equations

$$v_c = r_f \omega_g \tag{7.13}$$

s.t. v_c is the upwards velocity of the chassis, r_f is the radius of the gear feet, and ω_g is the gearbox's output rotational speed.

$$F_f = \mu_f m \left(\frac{\mathrm{d}v_c}{\mathrm{d}t} + g \right) \tag{7.14}$$

s.t. F_f is the force provided by the gear feet, m is the mass of the robot, g is the Earth's gravitational acceleration (~9.8 m/s^2), μ_f is the energy efficiency of the gear feet, and t is time.[8]

7.3.7. Chassis Equations

$$s_c = \int_0^t v_c \, \mathrm{d}t \tag{7.15}$$

s.t. s_c is the chassis' position, where the bottom of the rack is 0 metres.

[8] Derived from a free body diagram showing that $\sum F_y = F_f - mg = ma_y$ where a_y is the upwards acceleration of the robot.

Appendix A: Acceptable Code Examples

While there is a degree of subjectivity involved in programming styles, a standard must be agreed upon to ensure all Climbing Clock programmers can read and understand each other's code. Obviously, the style rules stated here will be biased towards the author's preferences, primarily the project lead. To mitigate these biases, Climbing Clock programmers are encouraged to make their disagreements known so a compromise or understanding can be made.

A.1. One line examples

Example 1: Naming, Spacing, and Unnecessary Operators

Consider the following line: `choco_var=( (*getMass)(blip)*9.8/(dennies) );;`

This line fails multiple standards: it uses poor variable naming, lacks a constant for `9.8`, lacks spaces inbetween operators and operands, has unnecessary characters (the outer parenthesis and the double semi-colon), uses a function pointer when a method is more appropriate, and is inconsistent with its choice of naming format. Fixing it looks like

```
robotPressure = robot.getMass() * GRAV_ACCEL / volume;
```

or

```
robot_pressure = robot.get_mass() * GRAV_ACCEL / volume;
```

Example 2: Simplifying Maths and Bitwise Operators

Supposing that all variables used are integers, consider the line

```
mid_point = (upper - lower) / 2 + lower;
```

This line should be simplified using algebra and bitwise operators to

```
mid_point = (lower + upper) >> 1;
```

A.2. Multi-line examples

Example 1: Boolean Expressions

Consider the method below where the `MyRobot` class has the data member `_climbing`:

```
bool MyRobot::isClimbing(void) {
    if (_climbing == true) {
        return true;
    } else {
        return false;
    }
};
```

This method could be greatly simplified by just returning the value of `_climbing`, as shown below:

```
bool MyRobot::isClimbing(void) { return _climbing; };
```

Example 2: Using the correct loop type

Consider the following usage of a `for` loop:

```
Serial.print("Give me an int: ");

for ( ; ; ) {
    userNum = Serial.parseInt();

    if (userNum < 0)
        break;
    else
        Serial.print("Another! ");
}
```

This `for` loop is used as a `do-while` loop due to no index variable and the exit condition being unknown when entering the loop. It should be written as

```
Serial.print("Give me an int: ");

do {
    userNum = Serial.parseInt();

    if (userNum >= 0)
        Serial.print("Another! ");
} while (userNum >= 0);
```

Example 3: Using typedef

Consider the function below, where cmpFunc returns a non-negative number if the left argument has a lower than or equal precedence than the right argument:

```
int main(void) {
    void* array;
    int (*cmpFunc)(void* left, void* right);

    // get array size and comparison function

    bool ordered = isOrdered(array, len, cmpFunc);
    cout << “Result is “ << ordered << endl;

    return 0;
}

bool isOrdered(void* array, int len, int (*cmpFunc)(void* left, void* right)) {
    ...
}
```

Looking at the first line, it is cumbersome and error prone to have to write the format of cmpFunc multiple times. Instead, a `typedef` should be used:

```
typedef int (*CmpFunc)(void*, void*);

int main(void) {
    void* array;
    CmpFunc cmpFunc;

    // get array size and comparison function

    bool ordered = isOrdered(array, len, cmpFunc);
    cout << “Result is “ << ordered << endl;

    return 0;
}

bool isOrdered(void* array, int len, CmpFunc cmpFunc) {
    ...
}
```

A.3. Commenting Examples

Example 1: Single Line Comments

Consider the following code:

```
PORTA &= ~(1<<PA0); // turn bit PA0 off
PORTA |= (1<<PA1);  // turn bit PA1 on
```

Such code, while efficient, is difficult to understand due to the usage of difficult to remember register names and bitwise operators. The comments provided only explain *what* is happening, not *why*. Hence, such code requires more meaningful comments to understand, as shown below:

```
//set motor direction backwards
PORTA &= ~(1<<PA0);
PORTA |= (1<<PA1);
```

Example 2: Multi Line Comments

Notice how uninformative the block comment is for the method below:

```
/*
 * turnLeft() method for TankRobot class
 */
void TankRobot::turnLeft(int degrees, int speed) {
    setLeftTrackSpeed(-1 * speed);
    setRightTrackSpeed(speed);

    delay(speed * SPEED_TO_DEGREE * degrees);
    setLeftTrackSpeed(0);
    setRightTrackSpeed(0);
};
```

The block comment does not tell readers anything they should already know from the name of the method. Furthermore, the block comment does not warn readers that the method is a blocking function (one that stops the Arduino from doing anything else for a long time) due to the usage of the `delay()` function. A better block comment should contain a Doxygen tags:

```
/*
 * @author Bob Bobsin
 * @since v1.2.0
 * @param degrees How much to rotate by in degrees
 * @param speed How fast to turn
 * @return void
 * @attention WARNING: Returns control only once tank has fully turned!
 * @brief Commands tank to turn left without changing centre position
 */
```

Appendix B: Failed and Ignored Designs

B.1. Tracking the time without a Real Time Clock

A naïve approach went be to just use the `millis()` function available in the Arduino IDE to track how far the robot has climbed given the speed. Unfortunately, this would require the system to be turned on at a precise time (midday or midnight) to ensure the robots start climbing at the correct time and, worse, the function uses a global variable that overflows back to 0 after roughly 50 days.[9] Additionally, some actuated designs may result in unreliable timing based position detection, such as motor slip. Hence, smarter, more reliable, designs must be implemented.

[9] See https://www.arduino.cc/reference/en/language/functions/time/millis/