Final Report
Daily Automatic Date Scribe

December 6, 2022

MTE 100 and MTE 121 Course Term Project
Group 4-2

Hannah Lamarche: 20995081
Julia Schirger: 21029808
Leo You: 20998391
Lucas Sadesky: 20957368

Summary

The MTE100 and MTE121 collaborative final project serves as a means for mechatronics engineering students at the University of Waterloo to apply the material learned in class to solve a tangible problem. This report entails an in-depth analysis of the concept drafting, mechanical design, construction, and programming processes of group 4-2, in their journey of building the Daily Automatic Date Scribe (D.A.D.S.). LEGO EV3 kits are used for the mechanical assembly of the robot, and the ROBOTC integrated development environment and C programming language for coding the project. The baseline task for the robot is to write the date on a whiteboard every day.

This paper discusses the group's approach to solving technical problems and making appropriate design decisions. The report begins outlining the purpose of the project and introduces the robot along with its functionality—including relevant constraints and criteria. Then, a more technical breakdown of the mechanics and software of the project is presented. Throughout, a documentation of trials and errors, and methods of testing or verification are incorporated. It is also important to make note of both old and remaining problems faced in the creation of this project, as well as possible solutions.

Acknowledgements

Group 4-2 acknowledges the teaching team for MTE100 and MTE121 for equipping the group with the mechanics and software skills needed to complete the project. Also, group 4-2 acknowledges the teaching team's help on debugging and offering advice for improving the design. Materials are both provided by the University of Waterloo and acquired at personal expense from the WATIMAKE student ideas clinic.

Table of Contents

List of Tables, Figures, and Equations

A. Tables

B. Figures

C. Equations

I. Introduction

The D.A.D.S is a product of a cumulation of ideas. The final project is crafted as a series of design decisions and is significantly different compared to the initial idea. The group's targeted issue begins as cleaning down a vertical pane or board. This design involves pumping soapy water and moving a sponge up and down a heavy aluminum extrusion suspended high on a wall. Safety concerns, as well as the physical limitations of the LEGO yields a need for a downsize in application. The next design idea narrows the scope to the classroom setting—more specifically, for potentially cleaning a whiteboard. Even here, the robot still performs a similar task of scaling a vertical surface. The group's goal, however, remains to design and build a project that brings the best feasible solution to a relevant problem. Writing an important piece of information on the board, is considered by the group to be a more meaningful challenge to solve, compared to simply erasing it. The date, written in the top right corner of the whiteboard, is a staple piece of information in many classrooms. Building a robot that writes the date on the whiteboard is a useful application of the content learned in MTE100 and MTE121. It helps busy teachers worry about one task fewer. Arriving upon this concept marks a major shift in design from cleaning to the board, to writing on it. Specifically, to writing the date. Hence, the development of D.A.D.S. begins. Notice, the moving motions along the board are not vastly dissimilar from the first idea. The erasing aspect, however, is entirely removed, as will be discussed later in the report.

II. Scope

A. Main Functionality

The main functionality of the robot is to automate the process of a person, such as a teacher, writing the date in the upper right corner of a whiteboard. The robot is designed to calibrate itself by moving to the correct position and drawing a box. It then writes the date inside the box and moves to the top-right corner of the whiteboard. After waiting for 24 hours, it repeats the process, with the date incremented by one day. Note D.A.D.S. runs under the assumption that the whiteboard is erased daily.

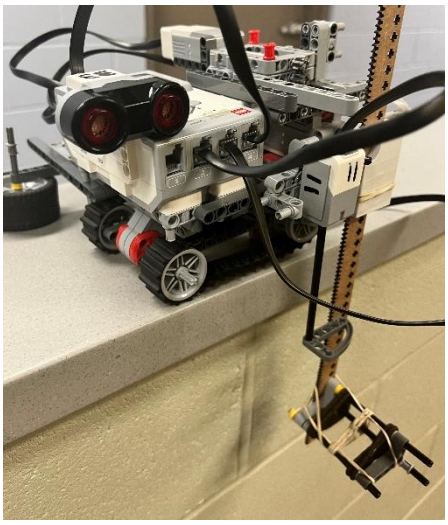


Figure 1. Pictorial view of D.A.D.S.; ultrasonic and writing sensor shown

B. Description of Sensors

The robot has two touch sensors and an ultrasonic sensor. The first touch sensor in position S3 is attached to the front of the robot, facing downwards, so that it drags along the surface behind the whiteboard to detect when it ends. This is important during calibration when the robot drives until it reaches the edge of the ledge. The second touch sensor is placed on the writing side (right side) of the robot and is used to detect when the marker has reached the top of its range of motion (y-position of the origin). The purpose of the ultrasonic sensor is to detect when there is an object such as a hand near the robot to initiate the emergency stop and exit the program to ensure no further movements will harm anyone or anything.

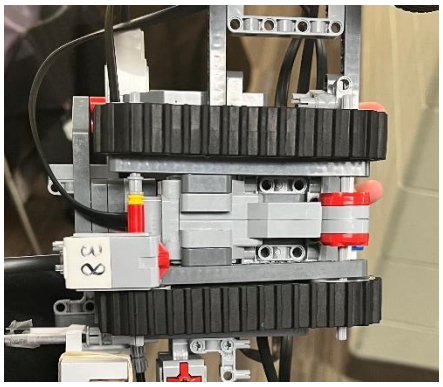


Figure 2. Bottom view; bottom touch sensor shown

C. Description of Interaction with Environment

The robot interacts with its environment using the touch sensor placed on the bottom to find the end of the ledge it is sitting on no matter the distance from the edge. The ultrasonic sensor is used to interact with the environment by sensing objects nearby that may come within 5 cm of the front of the device. There is an adjustable stabiliser on the side opposite to the drawing side to stabilise itself on the ledge and allow the robot to drive straight.

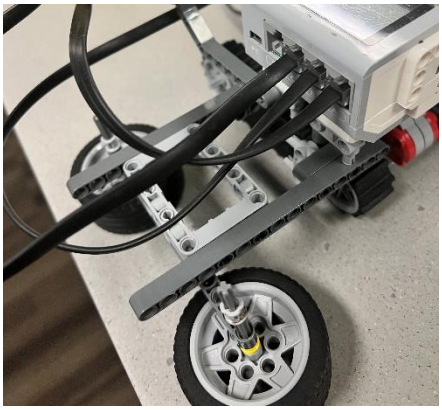


Figure 3. Stabiliser attachment shown

D. Task List and Shutdown Procedure.

Table I. Summary of Task Descriptions

Task	Description
Accept input	Accepts date from user input (selecting year, month, then day) from the arrow and enter buttons.
Calibration	Using the end of the ledge as a reference point, the robot calibrates and draws the box for the date.
Engages marker	The robot presses the marker into and out of the board to be able to draw while moving but also move without drawing.
Draws date	Moves the appropriate distances and engages the marker as needed to draw each numeric or dashed character (YYYY-MM-DD). The date is legible and appropriately clear.
Rests	Once finished writing the date, moves to the top right corner of the board and pulls up the rack.
Waits	Waits the desired time (theoretically 24h, 3 min for demo purposes) before redrawing the date (assuming a person has erased the previously written date).
Shutdown Procedures	Emergency stop: Robot stops when the system detects objects less than 5 cm away from the front of the ultrasonic sensor. Exits the program to prevent harm to objects nearby. Non-emergency stop: While the robot is waiting, the enter button can be pressed to exit the program and prevent the date from being redrawn once the timer elapses.

E. Sensor and Motor Indexes

Table II. Index of Motors

Motor Name	Function
Motor A	Powers the treads. Horizontal movement.
Motor B	Power the pinion gear. Vertical movement.
Motor C	Powers the gearbox motor. Engages and disengages the marker.

Table III. Index of Sensors

Sensor Number	Type and function
Sensor 2	Pinion calibration touch sensor

Sensor 3	Touch sensor to detect if the robot is on the wall
Sensor 4	Ultrasonic sensor for emergency stop

F. Changes in Scope from Earlier Stages of Design

The original design also has an eraser to automate the removal of the previous date. This feature is not implemented in D.A.D.S. since it is not a necessary task for the robot to fulfill. The assumption that a person erases the whole whiteboard, including the date, every day is reasonable.

Previous versions also include a function to check for message obscurity, in the scope of redrawing the date if the color sensor detects that the marker is erased. However, this is also omitted from the final product because of the large error in distinguishing between regular and partially erased marker.

III. Constraints and Criteria

A. Summary of Constraints and Criteria

Table 5.1: Table IV. Summary of Constraints and Criteria

Constraints	Criteria
Collects user date input using console buttons	Automatically increments the date by one day every 24 hours elapsed
Draws numbers using motor encoders	Draws date enclosure (box)
Uses touch sensors to automatically adjust and calibrate to the environment (surrounding surfaces)	Moves to rest position automatically at the start of the program
Detects obstruction and emergency stops	
Redraws date after timer expiration	

B. Adjustment to Constraints and Criteria

The initial constraints and criteria of D.A.D.S. include drawing and erasing the date autonomously, while drawing alphabetic letters as well as periodically checking for erase marks.

The erasing constraint is deliberately removed from the final product; the physical constraints of LEGO in junction with the weak motors prevent the whiteboard eraser from pressing into the board with enough pressure to erase the marker lines completely. Taking this into account, removing the erase constraint allows for a more polished and practical final product.

Moreover, the extended goals of periodically checking for erase marks and drawing messages with letters is also removed. First, the criterion of checking for erase marks and appropriately redrawing the current date is removed after noticing the inaccuracy of the colour sensor readings of the thin and slightly reflective marker lines. Secondly, the criterion of implementing letters is omitted as it does not display any new functionality except being labour intensive to implement. Due to the limitations of ROBOTC and the LEGO EV3, programming the robot to draw numbers is done by manually hardcoding parallel arrays with coordinate instructions.

C. Guiding Principles

The drawing function is the most important constraint that influences the design of the rest of the project. Ensuring that the robot is able to fulfil the base requirement of drawing strokes with reliability and precision is prioritized. Extensive effort is placed on prototyping a robust drawing arm and refining and testing the software.

Conversely, a constraint that is not valuable is the erase function as it is beyond the scope of the project. The mechanics of placing both a marker and its eraser on a single arm and moving at the same time is counterproductive. It makes the design and software considerations overly complicated and stalls progress

on achieving other criteria. Adding a secondary arm is also not elegantly feasible with the rest of the design, so removing it is most favourable in this context.

IV. Mechanical Design and Implementation

A. Design: Description and Philosophy

The mechanical design of the robot is chosen to satisfy the requirement that the robot should be able to freely move its arm while maintaining sufficient fine control so that those motions can be reliable and repeatable.

A rack and pinion setup controls motion along the vertical axis. A small pinion gear is used to decrease the distance that the rack moves per rotation of the motor. This setup optimises the precision of its motion. For the horizontal motion, a simple shaft drive is chosen for its simplicity and reliability. Additionally, it is expected that the result of having two treads driven by one motor through a continuous shaft decreases the possibility for unwanted turns during the driving process. The third axis is controlled by parallel worm gears attached to one axially constrained shaft and one rotationally constrained axle respectively. This gearbox has a very large gear reduction from the motor shaft to the output, decreasing the amount of motor error that is be passed on to the actual motion of the device.

Treads are chosen as the ground interfacing component because they have a greater contact surface with the ground, increasing the traction between the robot and the surface it drives on. They thus decrease the error caused by loss of traction. Given the importance of having a precise interface between the rack and pinion, the decision is made to CAD model and 3D print a rack guide that reduces the relatively large tolerances associated with LEGO. The adjustable, stabilising arm is added to reduce the path deviation that was observed through mechanical testing.

B. Analysis of Specific Sections

1) Parallel worm gearbox

The parallel worm gearbox is an integral component of the design that facilitates the function of the entire device. Due to the critical nature of the component, the design is carefully chosen to satisfy as many of the design criteria as possible. The simplest requirement of the gearbox is to translate the rack perpendicular to the whiteboard between two states, the marker touching the whiteboard and the marker not touching the whiteboard.

The initial design is chosen to prioritise simplicity by attaching the medium LEGO motor to two stacked LEGO Technic Beam Frames and then running a shaft from the output of the motor through to the other side of the Beam Frame. A worm gear and three bushings are then added to the shaft to constrain the worm gear from moving axially as shown in *Figure 4.* A longer axle is added to the lower Beam Frame with another worm gear constrained between a thinner bushing and a bracket, mounted to the motor driving the pinion gear. This prevents the worm gear from moving relative to the axle, but allows the axle, and bracket with connected pinion motor, to slide in the axial direction. By the addition of the bracket to the lower axle, the axle is no longer able to rotate and so any rotational motion translated through the upper worm gear is converted into axial motion through a very large gear reduction.

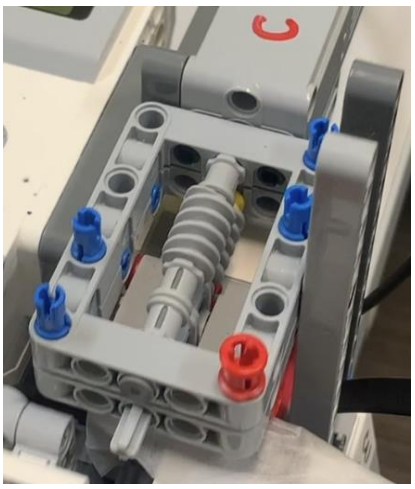


Figure 44. Initial gearbox design

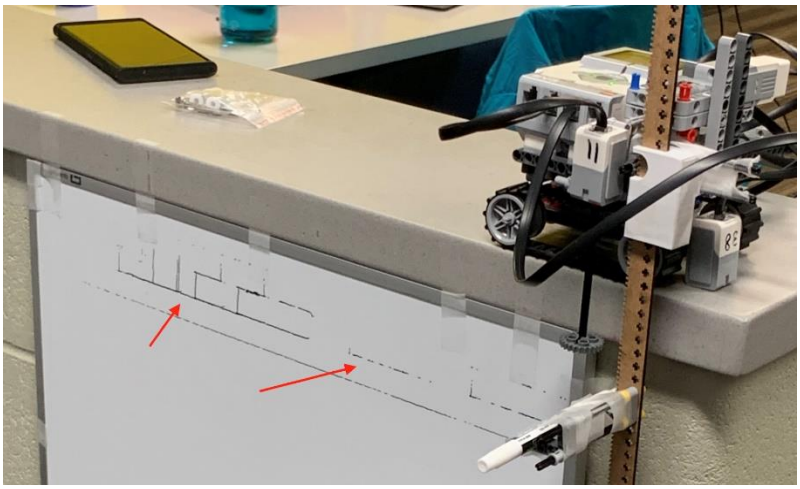


Figure 55. Dotted lines unintentionally drawn due to lack of gearbox range

The initial design has approximately four millimetres of range from fully extended to fully retracted, and due to the bend in the rack. This is insufficient to ensure that the marker does not touch the board throughout the entire vertical range of the rack. This problem becomes apparent in testing when the robot fails to fully raise the marker back from the whiteboard—and instead begins to draw dotted lines in sections where there should have been no markings. Notice this in *Figure 5* This problem motivates the redesign of the gearbox to accommodate a much larger axial movement.

The redesign of the gearbox consists of several significant challenges as the Technic Beam Frames are not long enough to accommodate any more movement compared to the previous design. To resolve this issue, a custom frame is constructed with a greater length. This increase in length requires a repositioning of the console as well as the addition of a third layer to the gearbox as the motor driving the worm gear no longer fits above the sliding assembly below it. The motor driving the worm gear is then moved to the third layer of the gearbox (see *Figure 6*). Two meshed 8-tooth spur gears drive the motion from the motor down to the second layer where the axially constrained worm gear is located (see *Figure 7*). The principle of driving the axial motion of the lowest axle is kept from the initial design. But, with the increased space provided by the frame redesign, a second worm gear is added beside the first. Both worm gears are constrained between a bushing and the bracket connected to the pinion motor, allowing the axle and bracket to slide axially (see *Figure 8*). The redesign increases the range at the gearbox to approximately 10mm which is tested and deemed to be sufficient.

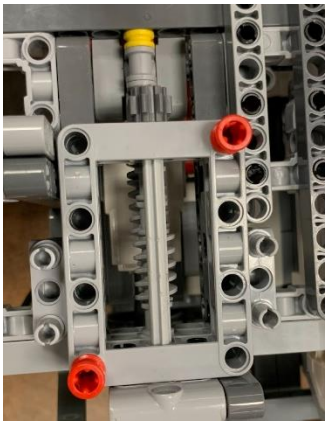


Figure 66. Gearbox top stage

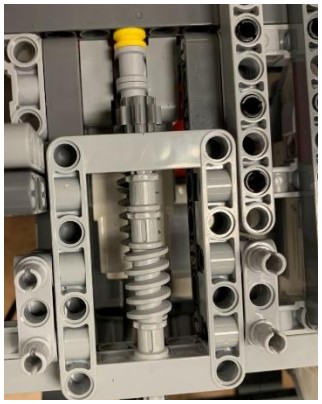


Figure 77. Gearbox middle stage

Figure 88. Gearbox bottom stage

2) Rack and pinion

The vertical axis control is also a critical requirement for the mechanical design of the robot. To minimise the number of components that need to be designed, it is decided to create a design for the vertical actuation that stays in its previous position once the motor is turned off. This means that the resistance to rotation in the motor is sufficient to overcome the force of gravity pulling the pinion and assembly downwards. To increase the likelihood that this resistance is sufficient, a small pinion gear is chosen to decrease the mechanical advantage that the rack has against the pinion. An additional reason that a small pinion is chosen is to increase the precision and decrease the error from the motor.

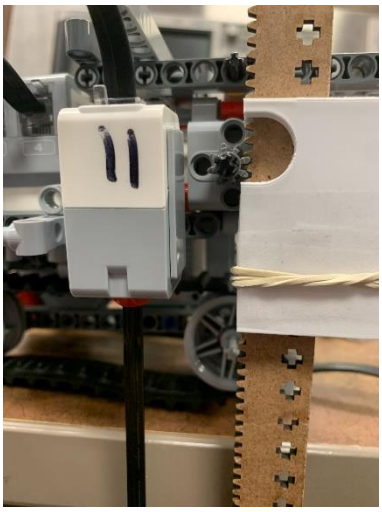


Figure 99. Rack and pinion

3) Rack Guide

Another vital requirement of the assembly is that the rack be very securely interfaced to the pinion gear. Due to the relatively high tolerances of LEGO, it is decided that a custom part is created. The required measurements are taken and then a shaft guide modelled in CAD. The first version features two holes designed to connect to the Lego pins and a slot designed so that the rack could slide through it with very small tolerances (*Figure 10*).

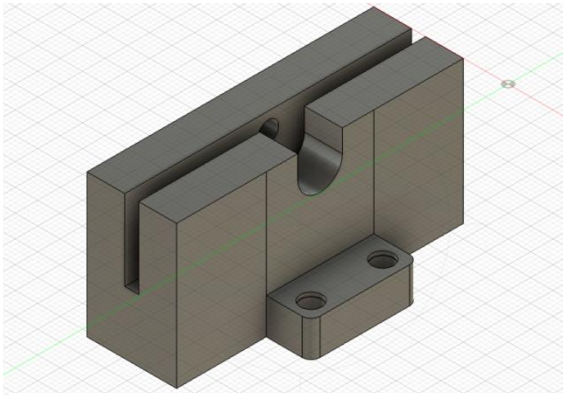


Figure 1010. SOLIDWORKS rack guide model

The first print does not have sufficiently large tolerances for the rack to slide through it. Additionally, because of the orientation of the print is created in, there are many support structures which are difficult to fully remove--resulting in an abrasive finish. A new version is thus created and printed (*Figure 11*). This version features slightly larger tolerances and a reduction of the extruded height of the guide on the top of the part to accommodate the movement of other parts in the robot. The second version is also printed in an alternate orientation, resulting in less issues caused by removing support structures and more precise tolerances.

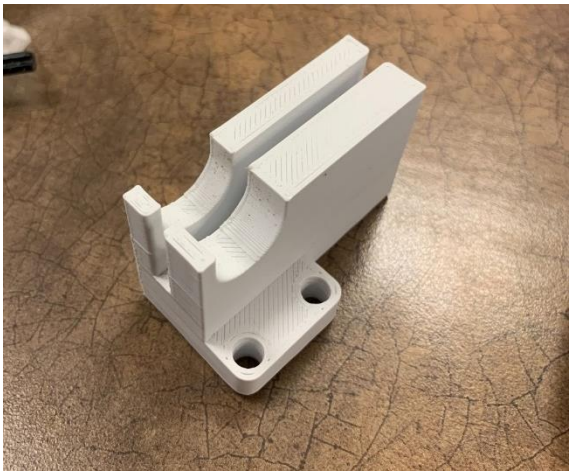


Figure 1111. 3D printed model

4) Stabilising Arm

Initial testing shows that the robot tends to turn while driving when unsupported. These slight turns create an accumulation of error which results in the arm misaligning with the board and in some cases, causing the robot to nearly fall off the side of the board. An adjustable, stabilising arm mitigates this issue. The arm is designed to be easily adjustable so the robot can be used in settings with different wall thicknesses. Large wheels run along the side of the wall opposite to the whiteboard to decrease resistance between the axles and the wheel, better supporting the lopsided weight distribution. Additionally, testing shows that by maximising the distance between the two wheels on the stabilising arm, the torque on the body of the robot introduced by the addition of the wheels is offset and the robot is able to drive in reliably straight lines. The possibility of adding wheels on the whiteboard side of the wall is also explored; however, the one-sided design works sufficiently and reliably.

C. Design Decisions

One trade-off made during prototyping and mechanical design is the removal of the erase function. The original design postulated a drawing arm with an eraser attached directly beside the marker. During implementation, this poses many challenges. Primarily, there is insufficient pressure between the erase and the board compared to the erase and the much sharper marker to remove any significant amount of writing the robot has and is simultaneously writing. This observation, with consideration of the time constraints, scope of the problem, and software concerns—discussed further in section V and VI—the eraser is removed.

V. Software Design and Implementation

A. Software Description

D.A.D.S' software can be broken down into four sets of procedures: core movement functions, date input functions, compound operational functions, and the *task main()* function.

Table V. Blocks of Functions

Core Movement	Date Input	Compound Operational	Main
engageMarker()	*maxDays()*	*drawLineFunction()*	*task main()*
moveEverything()	*updateDate()*	*drawCharPart()*	
sensorIsOn()	*updateDateCharacterArray()*	*calibrationFunction()*	
cmToEncoder()	*inputDate()*		

The software design philosophy is like that of the mechanical design philosophy. By coding complex procedures based on basic functions, D.A.D.S.' software design is highly modular. Effectively containing logical errors to the more logic-based complex procedures and the technical errors to the technical core functions, sources of error are easier to trace and the program is significantly easier to debug.

The set of core movement functions deals with the most basic physical operations of the robot; it is concerned with moving the marker across the whiteboard, engaging and disengaging on command, checking for obstructions and emergency stop conditions, and translating centimetre distances to LEGO EV3 encoder values. These rudimentary functions are not able to intelligently draw characters on their own but form the basis of more complex functions that can call upon them.

The other basic function of D.A.D.S. is the date input and update feature. The set functions concerning date input provides the instructions for collecting user input at the start of the main program and appropriately updates the date and the arrays storing that data after each iteration.

The compound operational functions are sets of higher complexity functions containing the instructions to intelligently draw characters and their appropriate lines as well as perform the calibration of the robot at each iteration. These functions do not hard code exact instructions, instead logically call upon the core movement functions to draw lines and create spaces as necessary.

The main task function contains the complete set of instructions to write the date, from how to write slashes and numbers to the task sequence.

B. Software Task List

Table VI. Software Task List Used for Demo

Task	Criteria for Success
Accept user input	Robot takes user input for the current date using up, down, and enter buttons. Clarifies with the user that their selected date has been input correctly.
Calibration	The robot drives to the edge, pulls up the marker to the top position, backs up the offset distance and the box distance, and draws the box that will surround the date.
Draws date	The robot draws the date in the form YYYY-MM-DD in digital analog numbers.
Move to rest position	The device moves to the top right corner of the board and pulls up the marker.
Wait	Waits for the timer to hit the specified time (theoretically 24 h, approx. 3 min for demo) before repeating the process.
Repeat process	Repeats process and draws an updated date without user input, incrementing the date by one day. Assume someone erased the previously written date.
Emergency stop	Robot stops when the ultrasonic sensor senses an object closer than 5cm. The program ends to prevent further motion.
Non-emergency stop	If the enter button is pushed while the robot is waiting, it should exit the program and not draw the date once the timer has elapsed.

C. Changes from Earlier Task Lists

Originally there was a task to erase the previous date autonomously. However, due to mechanical issues the eraser is removed from the final product. Now the device assumes the previous date had been manually erased at the end of the day before the next date will be written.

There is also an older version which has a software task to check for message obscurity and fix the date if obscurity was detected. This was removed because of measurement error from the sensor and since it goes redundantly beyond the identified problem this project solves.

D. Data Storage

D.A.D.S.' operation does not require very complex data storing. Hence, only a few data structures were implemented. They are described below:

- Character maps are stored as parallel one-dimensional arrays. One of the arrays indicates the lengths of each line segment in line units (defined by a constant). The other array indicates the orientation, direction and whether a line segment is a writing segment or a moving segment.
- The status of the emergency stop condition is constantly being evaluated as Boolean return that passes through its higher-level function it is nested within. This avoids having to store and reassign the state of emergency stop.
- Two arrays store the date. One for the composed date in the format: {YYY,MM,DD} and one decomposed: {Y,Y,Y,Y,M,M,D,D}.

E. Function Descriptions

int cmToEncoder(float distanceCM, float diameterFactor)

Author: Lucas Sadesky
The flowchart for this function can be found in *Appendix B.*
Examples of this function's applications can be found in *moveEverything(): AppendixE* and *calibrationFunction(): Appendix H.*

The *cmToEncoder* function takes in distances in centimetres and outputs an integer encoder value for that distance. This function also takes a parameter for diameter of the wheels or pinion gear creating the displacement. This function is called during the *moveEverything()* and *calibrationFunction().*

bool sensorIsOn()

Author: Julia Schirger
The flowchart for this function can be found in *Appendix C.*
Examples of this function's explicit applications can be found in *moveEverything(): Appendix E, calibrationFunction(): Appendix H, engageMarker(): Appendix D,* and *task main(): Appendix A.*

This function is responsible for emergency stops and does not take in any parameters. It constantly checks for obstructions within a globally defined distance ("SAFE_DIST") using the ultrasonic sensor in every iteration of while loops responsible for moving the drawing arm; upon detecting an obstruction, the function returns the *false* Boolean value. By setting most all function types to bool and placing all functions in if-statements—throughout most all functions, including *task main()*—that only proceed when their outputs are true, this system permeates layers of functions and effectively stops the robot immediately.

bool engageMarker(bool moveMarker, bool markerDown)*

Author: Lucas Sadesky
The flowchart for this function can be found in *Appendix D.*
Examples of this function's applications can be found in *drawLineFunction(): Appendix F*

This function takes in the desired engagement state of the marker as a Boolean vale—*true* for engaged on the whiteboard or *false* for disengaged—as well as the pointer to the present state of the marker as a Boolean value to appropriately engage or disengage the drawing arm. A pointer is used to allow the state of the marker to be dynamically updated throughout the program. If the current and desired state of the marker are not the same, motor C will spin in the appropriate direction by a globally declared number of degrees, that is derived by testing, to change the marker engagement state. As the gearbox engages the arm, the *senorIsOn* function is called to check for obstruction. If the movement proceeds without detecting an obstruction, *encgageMarke*r*()* will then flip the current engagement state of the marker. The return type of this is Boolean, returning *false* if *sensorIsOn()* is confirmed and *true* otherwise.

bool moveEverything(float xDistance, float yDistance)

Author: Lucas Sadesky
The flowchart for this function can be found in *Appendix E.*
Examples of this function's applications can be found in *drawLineFunction(): Appendix F*, *calibrationFunction(): Appendix H*, *drawCharPart(): AppendixG*, and *task main(): Appendix A.*

This function takes in the desired distances in either the x or the y direction (right/down positive, left/up negative conventions) and moves the drawing arm. Referring to the flow chart, this function first converts calls the *cmToEncoder()* function to find the Δx and Δy encoder values, taking into account the diameter of the gear/wheel creating the motion, and stores the current motor encoder values in temporary variables. Due to the reliability limitations of the motor encoders, moving the drawing arm must be done along the x and y-axis consecutively; hence, for a non-zero Δx or Δy encoder value, the complement change will be zero. Using this, the function then uses if-statements to engage the appropriate motor to move horizontally or vertically. The movements proceed while this condition is not satisfied:

Equation 1. General moveEverything() while loop equation

$$abs\ (current\ encoder\ value\text{-}initial\ encoder\ value) < abs(\ encoder\ value)$$

Similar to the *engageMarker* function, the *moveEverything* function also calls *sensorIsOn()* in its while loops and returns Boolean values, *false* if an obstruction is detected and *true* otherwise.

bool drawLineFunction(int xDistance, int yDistance, bool markerDown)*

Author: Lucas Sadesky
The flowchart for this function can be found in *Appendix F.*
Examples of this function's applications can be found in *calibrationFunction(): Appendix H* and *drawCharPart(): Appendix G.*

This function is an extension of the *moveEverything* function. Taking in x and y distance, one required to be zero, and the pointer to the current engagement state of the marker, this function first engages the marker to the board by calling *engageMarker(),* then passes on the desired x and y distances to the *moveEverything* function, and finally disengages the marker by calling *engageMarker()* again. The return

type of this function, Boolean, supports the emergency stop structure of the program. As denoted by the decision modules in the function's flow chart, each step checks for the return value of the called function and immediately returns *false* if any of the functions encounter an obstruction and return *false*.

bool drawCharPart(char direction, int segmentLen, bool markerDown)*

Author: Julia Schirger
The flowchart for this function can be found in *Appendix G*.
Examples of this function's applications can be found in *task main(): Appendix A.*

This function is a compound function responsible for drawing one stroke of a character, taking in a stroke's direction and relative magnitude as well as the pointer for the engagement state of the marker as parameters; the direction and magnitude parameters are drawing directions from the *directions[88]* and *segmentLen[88*] arrays located in *task main().* From the flow chart, there are four if-statements checking if the direction is "h", "v", "H", "V", the "h" and "H" for horizontal and "v" and "V" for vertical with the capitalised directions indicating moving with a line drawn and the uncapitalized indicating moving without. Under the if-statements with uncapitalized directions, *moveEverything()* is called, and under if-statements with capitalised letters, *drawLineFunction()* is called, both with distance parameters appropriately filled and scaled by the globally defined character width.

The return type, Boolean, and internal structure of this function supports the emergency stop feature. Each function is wrapped in an if-statement checking whether the enclosed function returns false, indicating an obstruction is detected and promptly returns false thereafter.

bool calibrationFunction(bool markerDown)*

Author: Hannah Lamarche
The flowchart for this function can be found in *Appendix H.*
Examples of this function's applications can be found in *task main(): Appendix A.*

This function is responsible for the initial setup for D.A.D.S. The function takes in the pointer to the current engagement status of the marker and returns a Boolean to comply with the emergency stop system; if an obstruction is detected in its nested functions, the robot will stop immediately. This feature is expressed, as shown in the code and in the flowchart in the decision blocks.

The function operates as follows:

- Activates primary wheel motors and drives to the right until the touch sensor riding on the surface is no longer activated.
- Activates the pinion and pulls the rack up until the touch sensor is activated by colliding with the drawing arm.
- Lowers the drawing arm slightly by the width of one character unit.
- Drives to the left most corner of the date enclosure box using motor encoders and globally defined dimensions.
- Draws the enclosure by calling *drawLineFunction(),* drawing clockwise from the top left corner.
- Returns true if no obstruction was detected while drawing the enclosure.

int maxDays(int year, int month);

Author: Leo You
The flowchart for this function can be found in *Appendix I.*
Examples of this function's applications can be found in *updateDate(): Appendix L* and *inputDate(): Appendix J.*

This function takes in a specific year and specific month and calculates the maximum number of days in the specified month of that year, returning it as an integer.

From the flowchart, the function deals with the edge cases are first.

- If the month is February, the function calculates the maximum number of days by checking if the year is a leap year and returning 28 or 29 accordingly
- Else, if the month is not February but is the 4th, 6th, 9th, or 11th month, 30 is returned
- Else, 31 is returned

bool inputDate(int dateArray);*

Author: Leo You
The flowchart for this function can be found in *Appendix J.*
Examples of this function's applications can be found in *task main(): Appendix A.*

This function prompts the user for the specified start date using the console buttons. It takes in the pointer to *dateArray[3]*, the array storing the date to be drawn, as a parameter. The return type is a Boolean that is used in *task main()* to indicate whether the date has been correctly inputted.

Per the flowchart, the function has four blocks of code:

- First Block:
  - Displays the current year before selection
  - For-loop:
    - Up/Right buttons: increments the year
    - Down/Left buttons: decrements the year
    - Enter button: breaks the for-loop condition
  - Displays the selected year after selection
- Second Block:
  - Displays the current month before selection
  - For-loop:
    - Checks if the month is between 1 and 12 and resets the month to 1 if needed
    - Up/Right buttons: increments the month
    - Down/Left buttons: decrements the month
    - Enter button: breaks the for-loop condition
  - Displays the selected month after selection
- Third Block:
  - Calculates the maximum number of days in the selected month and year
  - Displays the current date before selection
  - For-loop:
    - Checks if the day is between 1 and the date limit and resets the date to 1 if needed
    - Up/Right buttons: increments the day
    - Down/Left buttons: decrements the day
    - Enter button: breaks the for-loop condition

    - Displays the selected date after selection
- Fourth Block:
    - Displays and confirms the date with user
        - Returns *false* if any button other than the Enter button is pressed
        - Returns *true* if the enter button is pressed and writes the selected year, month, and date to *dateArray[3]*

void updateDateCharacterArray(int dateArray, int* dateCharacterArray);*

Author: Leo You
The flowchart for this function can be found in *Appendix K*.
Examples of this function's applications can be found in *task main(): Appendix A.*

This function takes in the pointer to *dateArray[3]* and *dateCharacterArray[8]* to take the date specified in *dateArray[3]*, parse each digit, and insert them respectively into the consecutive indexes of *dateCharacterArray[8]*. This is done using a series of trivial algebraic operations and returns no value.

void updateDate(int dateArray);*

Author: Leo You
The flowchart for this function can be found in *Appendix L.*
Examples of this function's applications can be found in task main(): *Appendix A.*

This function takes in the pointer of *dateArray[3]* and updates the date based on the year, month, and day value specified in the array. The return type of the function is void as it does not return a value, instead, replaces the specified date in the array.

This function first tracks the maximum number of days in the present month and year. Then it increments the third index of *dateArray[3]*, the index containing the value related to the day of the month and checks and resolves conflicts with the date value being out of range—resetting the day to 1 and incrementing the month if necessary. Next is checks and resolves conflicts with the month—resetting the month to 1 and incrementing the year if necessary.

F. Explanation of Design Choices

The software design of this program is designed to be scalable. The concept is that should this be extended to a more formal product; the software only needs slight adaptations to perform the same functions and any additional features could be reasonably built into the existing framework.

Moreover, this project's software recognises the feasible limits of LEGO EV3 components, particularly that of sensor and motor reliability and aimed to create a polished product within this scope. To this end, certain features have been removed from the final product. Initially, D.A.D.S. was meant to be able to draw diagonal lines with the hope of being able to use them to scale to letter characters; however, the reliability of motor encoders and aligning rates between motors poses a severe spatial and aesthetic problem that undermines the final product's quality. Hence, diagonal lines were removed in place for a slightly more inefficient but overall, more effective solution wherein the robot can only manoeuvre in a grid.

Additionally, as the LEGO is incapable of applying ample pressure to the board to erase marks, the constraints of the project had to be adjusted. While the first iteration of the software contained a fully

developed erase function that could operate the robot completely autonomously, the physical restriction impacts the software design significantly.

Finally, another software design choice was how the emergency stop was implemented. The structure of the program, having functions within *task main()* that call one another, means that emergency conditions have to be communicated through layers of functions immediately. To effectively implement emergency stop, each function is wrapped in an if-statement that only proceeds if a function returns *true* and each function that moves the robot is converted to a Boolean function that returns *true* or *false* based on the detection of an obstruction. Hence, if something is detected, it would return false to its parent function and its parent function to its parent function, moving up in layers until it reached *task main()* in which it would break out and stop immediately as well.

The need for the software to immediately stop and not proceed is intentional. Due to the moving parts of this robot, the main concern is someone getting cinched into the teeth of the rack and pinion or gearbox. Hence, while it was discussed that the robot's emergency procedure should be more complex, a full stop makes more sense when considering the safety concerns as well as the capability of the calibration procedure to reset the robot.

G. Testing Procedure

The program, including the main and all the functions, was tested gradually to sort out where problems occur. The order in which the tasks were tested was chosen based on the complexity and importance of the task. Additionally, some tasks could only be implemented after others were functional such as redrawing the date, which relies on the timer.

Table VII. Testing Procedure

Test	Expectation	Problems	Solutions
Calibration	The robot drives to the edge, back up the desired distance, and draws the box.	Infinitely backs up instead of stopping at the desired distance. Stuck in while loops after entering the *moveEverything* function.	Break up the code into smaller steps Display messages to see run-time progress. Isolate problems to faulty while loop logic. Fix the condition for the while loop.

Drawing Date	Writes the date in analog numbers in the form YYYY-MM-DD.	The marker does not disengage enough causing it to leave marks on the board during non-drawing movements.	Mechanically extend the gear box and edit the code to make the movement larger.
User Input Date	The date will be input by buttons, display the selected date asking for clarification, then wait until the bottom touch sensor is activated to start a two second countdown before moving to calibration.	The date showing current selection is not displaying properly.	Resolve typos in the code. There was an ampersand instead of a percent symbol in the *displayString* function parameters.
Timer	The program waits for the timer to elapse before redrawing the date.	The program was exited instead of starting the timer.	Resolve the syntax error in the code for the timer.
Redrawing the Date	After the timer elapses, the robot redraws the date incremented from the previous day.	No problems occurred.	Not applicable.
Emergency Stop	When an object comes within five centimeters of the ultrasonic sensor, the robot should exit the program.	The initial system of using a variable with pointers passed through ach function does not update immediately.	Converted functions to type bool from void.
Non-emergency Stop	If the enter button is pushed while the program is waiting in the timer, the robot should exit the program instead of redrawing the date once the timer elapses.	No problems occurred.	Not applicable.

H. Significant Problems

1) Emergency stop

The final feature implemented into the project was the emergency stop if a hand, or any object, comes 5cm or closer to the ultrasonic sensor. This is a comparatively complex feature to program, compared to the rest of the code. The objective is for the ultrasonic sensor to be checked at every moment in the program—meaning in every function as well as in every loop. If an object is detected to be too close to the sensor, at any time, the program should terminate. This is a struggle to implement correctly because of the nested functions. It is not sufficient to simply exit a single function if the emergency stopping condition is triggered. The whole program needs to be exited. To solve this problem, every single void function is changed to return a Boolean value, of whether the emergency condition is fulfilled. A trivial Boolean function called *sensorIsOn()* is created with the sole task of returning whether the sensor value of the ultrasonic is less than 5cm. The *sensorIsOn()* function is then called in every other function as well as in the loops. Moreover, every function is called inside the condition of an *if* statement. If at any time that statement yields *false*, the return command is called inside *task main ()* and the program is exited.

2) Erase

As discussed in the mechanical section, having the eraser and the marker on the same arm on our robot does not permit the eraser to rub the marker off with enough force. The current setup does not allow for the insertion of a secondary arm, including another rack and pinion system for the eraser. Thus, the feature is eliminated. A second arm on a larger bodied robot and more powerful motors theoretically permits the addition of this feature. In terms of programming, the function is written nearly identically as when moving the marker. The only change is taking the surface area of the eraser into account when determining the number of swipes needed to cover the entire surface of the box.

3) Engage Marker

At the beginning and end of each line segment that was drawn, *engageMarker()* is called. This menas that *engageMarker()* is by far the most called function that results in a physical action of the robot. Due to the high frequency of this function being called, it is vitally important that deviation from expected behaviour is minimised to not end up with an accumulation of errors later on in the operation of the program. If the error accumulated, the robot would try to move the marker past the mechanical limits of the gearbox resulting in skipping and loosening of the worm gears on the shaft and axle. This problem is mitigated in several ways. First, the range of movement is defined by a number of encoder degrees that the motor should travel from one extreme to the other. This value was determined experimentally, and then once found, was reduced so that there was a small clearance zone between the far ends of the range used and the actual mechanical limits of the gearbox.

4) Diagonal Lines

In the original implementation of the code, the *drawLineFunction()* is designed to be able to draw straight lines at any angle.

This was done by calculating the total displacements in the vertical and horizontal directions and then calculating what fraction of the total movement (hypotenuse) these horizontal and vertical motions represented. Then, when the motors were powered on, the regular power was multiplied by these rates and would both arrive at the desired point at the end of the line at the same time.

In practice, this concept did not work as intended. The robot's motors are not nearly precise enough to make elegant strokes. Hence, the idea was scrapped for a simpler version of the *drawLineFunction*() that used only straight vertical or horizontal lines. One of the implications of this design change was that all the character maps had to be changed from how they were originally designed to use no diagonal lines.

VI. Verification

I. Updated List of Constraints for Demo and their Requirement for Success

Table VIII. Minimum Constraints for Success

Constraint	Description	How it is met
User Input Date	Using the buttons on the EV3 console, the date can be entered and clarified.	The date is input by buttons, console displays the selected date asking for clarification, then once the bottom touch sensor is activated, it waits two seconds allowing the user to move out the way.
Calibration	The robot drives to the edge, backs up the desired distance, and draw the box. This is vital for correct placement of the date.	The robot drives to the edge and backs up the distance of the box and draws the box correctly creating a space for the date.
Drawing the date	Writes the date in analog numbers in the form YYYY-MM-DD.	The robot draws the correct date with very clear, legible numbers on the whiteboard.
Emergency Stop	When an object comes within five centimetres of the ultrasonic sensor, the robot should exit the program.	When a hand is waved in front of the ultrasonic sensor, the robot immediately stops, preventing further motion from causing harm.

J. Unmet Constraints

All constraints included in the updated constraint list that was presented during the formal presentation were met.

Constraints from the initial ideation list that were omitted includes:

- Erasing the board
- Checking for erase marks

Due to a combination of the physical constraints of LEGO in making precise motions as well as assignment time frame, the erase function was omitted; the motors can not localise enough pressure

across a very pliable drawing arm to effectively erase drawing lines. Moreover, this project aims to address inefficiencies when writing the date such as visibility, tediousness, and clarity; erasing falls outside this scope. Hence, there is no design update that addressed this issue. Instead, omitting this constraint directly improves the quality of the final written date by impacting the stability of the drawing arm and simplifying the process of drawing characters.

The criteria of checking for erase marks and redrawing the date is also absent from the final product. This was done for a combination of reasons:

- Accuracy limitations of LEGO EV3 colour sensors tracking and detecting thin marker lines
- Project time constraints during testing and debugging phase
- Relevance of this feature

The prevailing argument for removing this feature was its significant cost, time wise, for its relatively little reward. Overall, this feature is redundant given the quick and easy setup, and unreliable when considering the precision of the sensors. This constraint is not explicitly addressed but is already accounted for by the comprehensive start-up procedure.

VII. Project Plan

A. How Tasks were Split Among Group Members

Table IX. Member Responsibilities

Group Member	Responsibilities
Julia Schirger	Mechanical *Design:* Erase mechanism Software Design: *Draw Character Function, erase* function, Emergency *stop function* Report: *Introduction, Fore Matter, Conclusion, Software problems, Editing*
Hannah Lamarche	Mechanical *Design: Marker attachment* Software Design: *Erase function, Emergency stop function, Calibration* Report: *Scope, Software problems, Contraints, Project plan, Verification*
Leo You	Mechanical *Design: Console and component attachment* Software Design: *Date input, main loop* Report: *Formatting, Editing, Software, Constraints, Verification, Appendix, Back Matter*
Lucas Sadesky	Mechanical Design*: Gearbox, rack and pinion stabilizer, stabilising arm* Software Design: *Core move/draw/engage functions* Report: *Mechanical design, Software problems, Data storage*

While functions and mechanical builds were technically split between members, most tasks were performed collaboratively during this project. When problems occurred, it is useful to get another person's point of view, especially while programming large functions. Sometimes what one member cannot figure out is solved easily with the help of another. Group meetings were frequently planned to split up new tasks, revaluate individual workload based on progress, and help each other out with issues.

B. Revisions to the Project Plan

The main change to the project plan was getting rid of the separation between finishing software and starting testing. It is much more effective to test the software as it is written to ensure issues were found early on and compiling errors are limited.

C. Differences between Project Plan and Actual Project Path

There are very few differences between the project plan and the actual project plan because of realistic planning. Due to demanding schedules, the software was not finished by the preferred deadline, but there was still sufficient time to complete it as there was room for delays in the original schedule.

VIII. Conclusions

Revisiting the objective of this robot project, group 4-2 believes D.A.D.S. not only meets its specific constraints and criteria but fulfills the general purpose of the final assignment for MTE100 and MTE121. Recall, the selected problem resolution is assisting teachers by reducing the number of tasks to do every morning before class. Having an automated process of writing the date on the whiteboard every 24 hours spares the teacher of this chore. Through testing, robot proves itself as functional, from where it can be concluded that it does help solve the intended problem.

The report loosely follows the course of the design cycle. It first outlines the problem, then describes the brainstorming of ideas leading up to the selected design. Once the constraints and criteria are stated, the report steers into an analysis of the mechanical and software systems. These sections include explanations on how intricate parts work together to achieve the desired motion and show reasoning behind design choices.

Arguably the most critical aspect of the analysis lies in the problem discussion and error analysis sections. It is important to acknowledge that the project is not perfect. Keeping record of the solutions to the most difficult problems as well as discussing problems that were never resolved or features that were never implemented all documents the learning experience.

Reflecting on the verification process and looking back to the project plans gives a comprehensive summary of how the various ideas and designs have merged. It also widens the report reader’s horizon on the possible extensions of this simple, yet insightful robot project.

IX. Recommendations

A. Hardware Recommendations

1) Rack and Pinion

The rigidity of the rack and pinion repeatedly caused issues in the design of the robot. For this reason, in a future version, several modifications to the rack and pinion setup are recommended. First, the rigidity of the rack causes a number of issues with precision and was what ultimately led the erasing function to fail. In order to remedy this, a more rigid rack should be implemented, ideally this rack should be chosen to be light as well but rigidity is the more important factor in this case.

2) Rack Guide

Overall, the rack guide was very helpful in securing the interface between the rack and pinion but some modifications could still be made that would be helpful. First, adding a fourth side the rack guide would help to fully constrain the rack in the directions it should not move. Additionally, adding holes for the shaft on which the pinion rotates into the guide would prevent any issues with the rack guide moving farther and farther away from the pinion which were experienced during the project. Adding additional interfacing points with the LEGO would also be helpful to decrease the amount of movement that was possible between the guide and the rest of the LEGO assembly.

3) Stabilisation

The addition of the stabilising arm was crucial to the success of the project but it was felt that the arm could have improved or another solution could have been found. The arm is also dependent on there being an opposite side to the wall on which the robot writes that is in relative close proximity to the whiteboard side. A number of options could be explored to solve this issue in a future iteration of the design. First, the use of a gyro and independent drive motors for each of the treads could be implemented along with an algorithm to ensure that the robot would always drive in a straight line. A touch sensor could also be implemented that would ride along the surface of the whiteboard and the program could adjust the power to each of the motors to ensure that the touch sensor always stayed in contact with the board. Another possibility would be a redesign of the stabilising arm so that it rode along the whiteboard side of the wall. This would be useful in a more general situation because in any application of the robot, there will be a whiteboard wall.

B. Software Recommendations

1) Software Changes

Given more time there would be improvements made to the software: improving the efficiency of the *engageMarker* function and adding slashes to improve the appearance of the date. Currently, the marker disengages every time it finishes drawing a line just to re-engage when it draws the next line. It would be more efficient to not disengage and re-engage the marker between consecutive lines. Moreover, most dates are written with slashes, not dashes. Slashes were included in the original plan but removed due to dashes being simpler and more appropriate for the scope of this assignment. If time was not limited, slashes could be implemented as another character.

2) Industry

Due to the design philosophy of the software design, only a few changes would be made to scale for industry application. Assuming that the robot retains similar mechanical designs, many of the software's core functions would remain; there would be commands for converting distances to appropriate motor rotations, a function to engage and disengage the marker, a function to move the robot, and a function that checks for an emergency condition.

Instead, the bulk of the software changes would be to create a product that is marketable; drawing the date on a whiteboard is a niche market. The first change would be to improve how the program stores the instructions for drawing each character. While parallel arrays were a convenient solution, ideally if the software could use hash maps, the drawing capabilities could be more well-scaled, remaining organised when implementing different symbols and letters to move from drawing just the date to messages.

Accordingly, the software would likely have to account for diagonal and even curved lines. Given access to more complex sensors that are more reliable and accurate, the core functions could be easily modified so that they could calculate rates to synchronously activate and draw a variety of lines in a variety of directions.

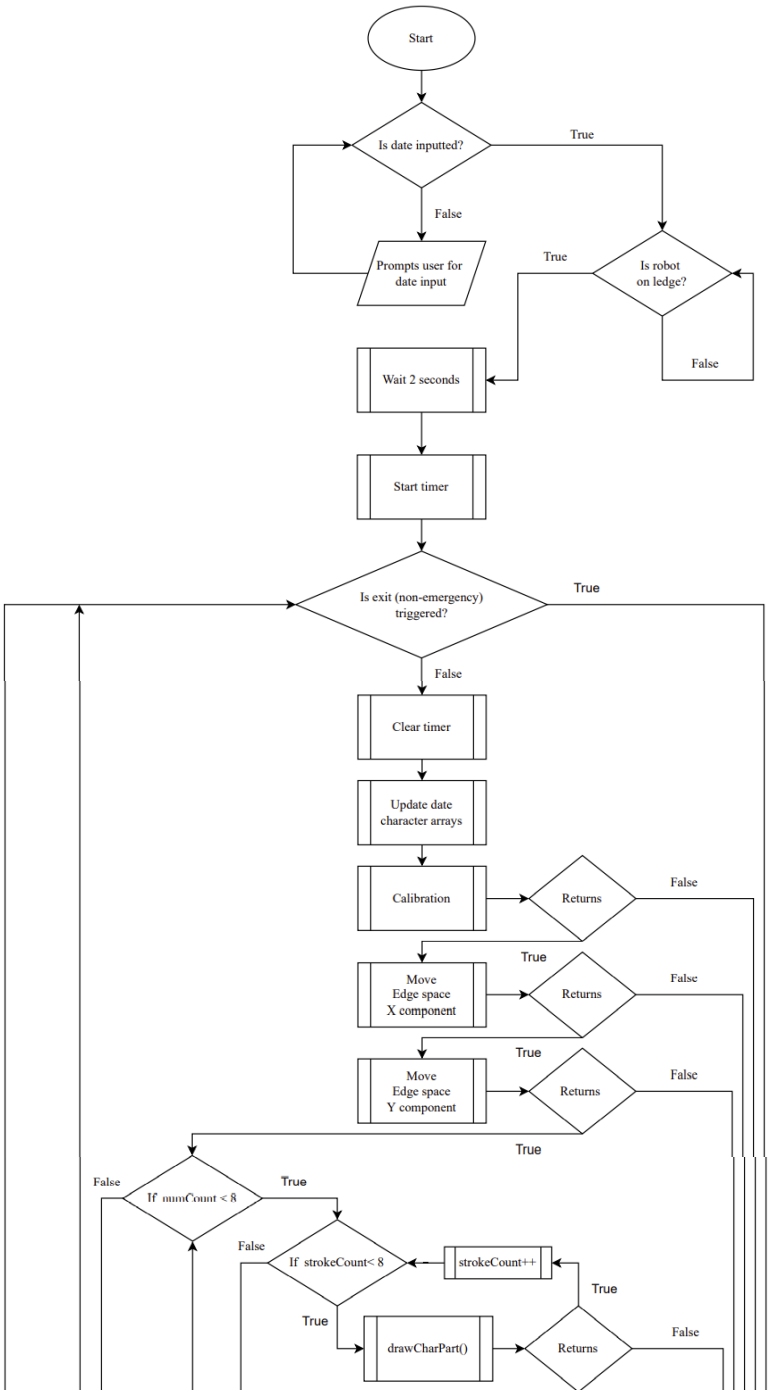
The user interaction would also be retrofitted. Rather than editing the output using the console buttons, to better accommodate a wide range of messages, the robot could be programmed to read in from text files that could be uploaded or using a more complex touch interface on the robot.

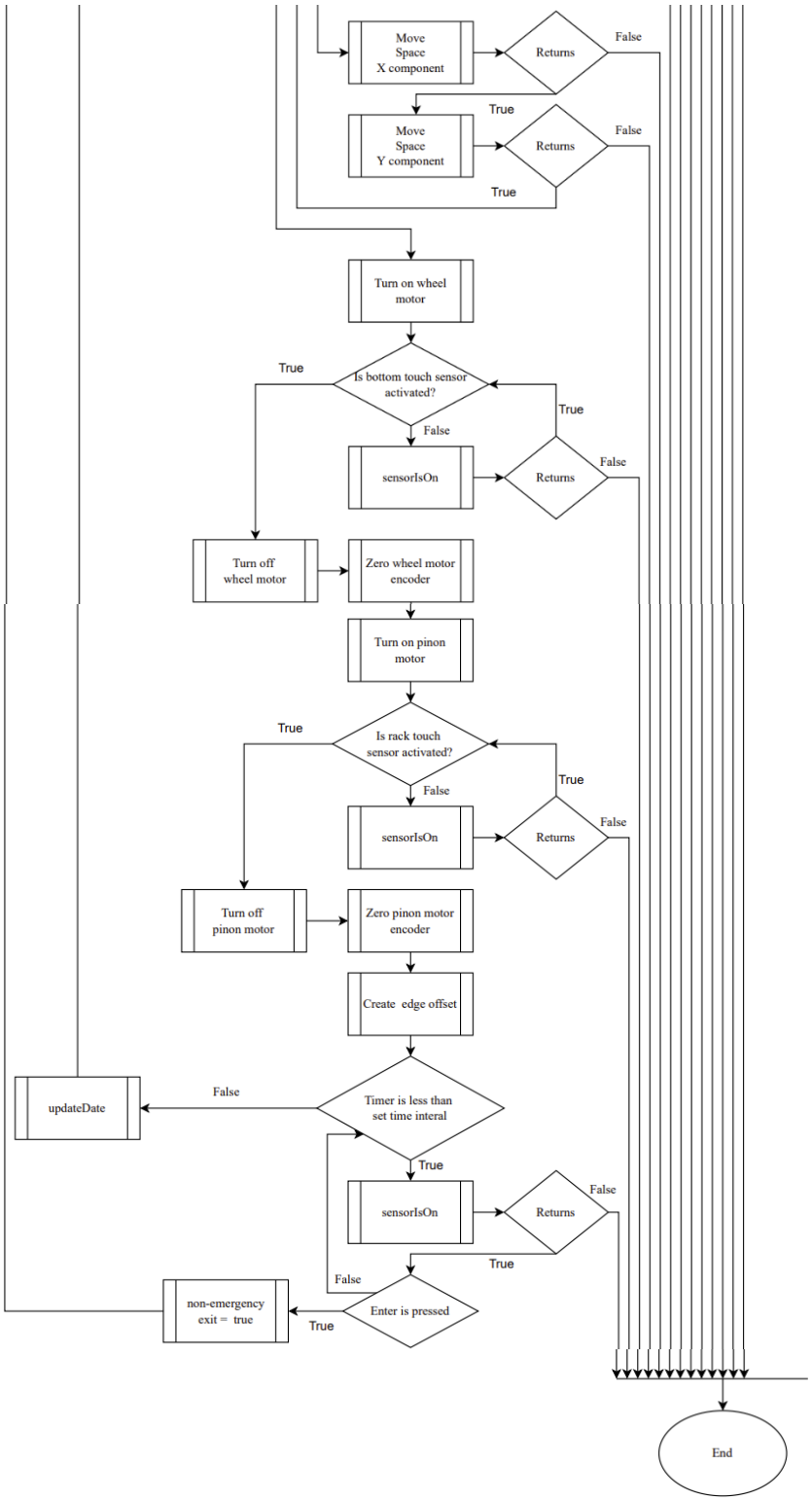
The support of more complex messages and diverse applications also means the method of storing data would need to be updated. Currently, the message length is bound to eight characters by the array length and the enclosure is programmed to specifically accommodate this set length. For varying length messages, a programming language that supports variable length arrays would have to be adopted.

Back Matter

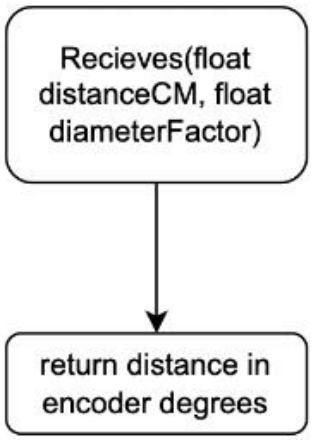
Appendix

Appendix A: task main()

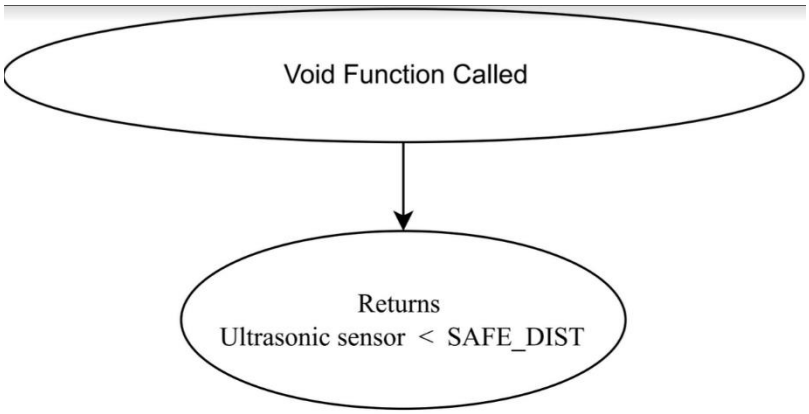



Move Space X component
Returns
False
True
Move Space Y component
Returns
False
True
Turn on wheel motor
Is bottom touch sensor activated?
True
False
True
sensorIsOn
Returns
False
Turn off wheel motor
Zero wheel motor encoder
Turn on pinon motor
Is rack touch sensor activated?
True
False
True
sensorIsOn
Returns
False
Turn off pinon motor
Zero pinon motor encoder
Create edge offset
Timer is less than set time interal
False
updateDate
True
sensorIsOn
Returns
False
True
Enter is pressed
False
True
non-emergency exit = true
End

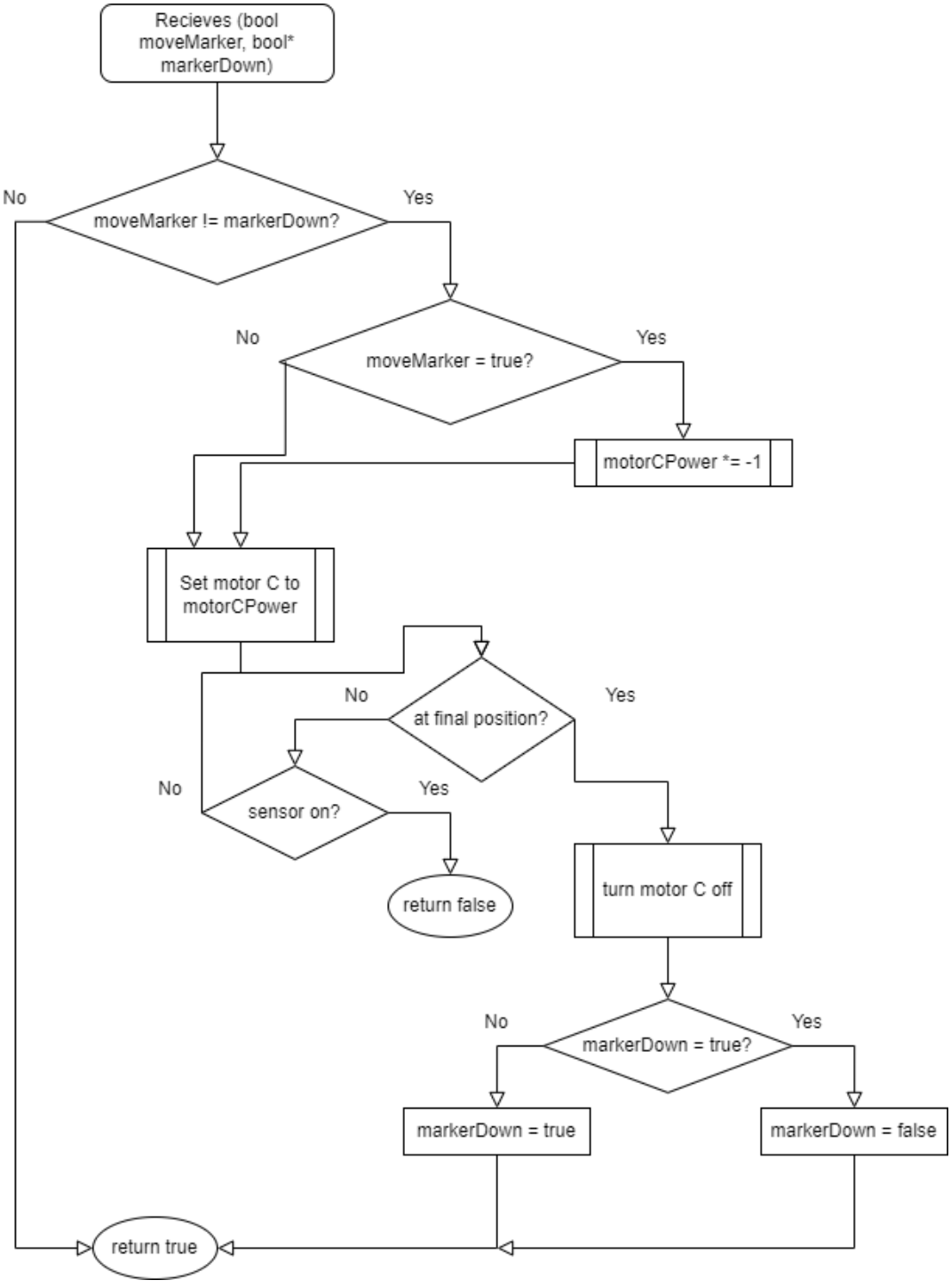
*Appen*dix B: int cmToEncoder()



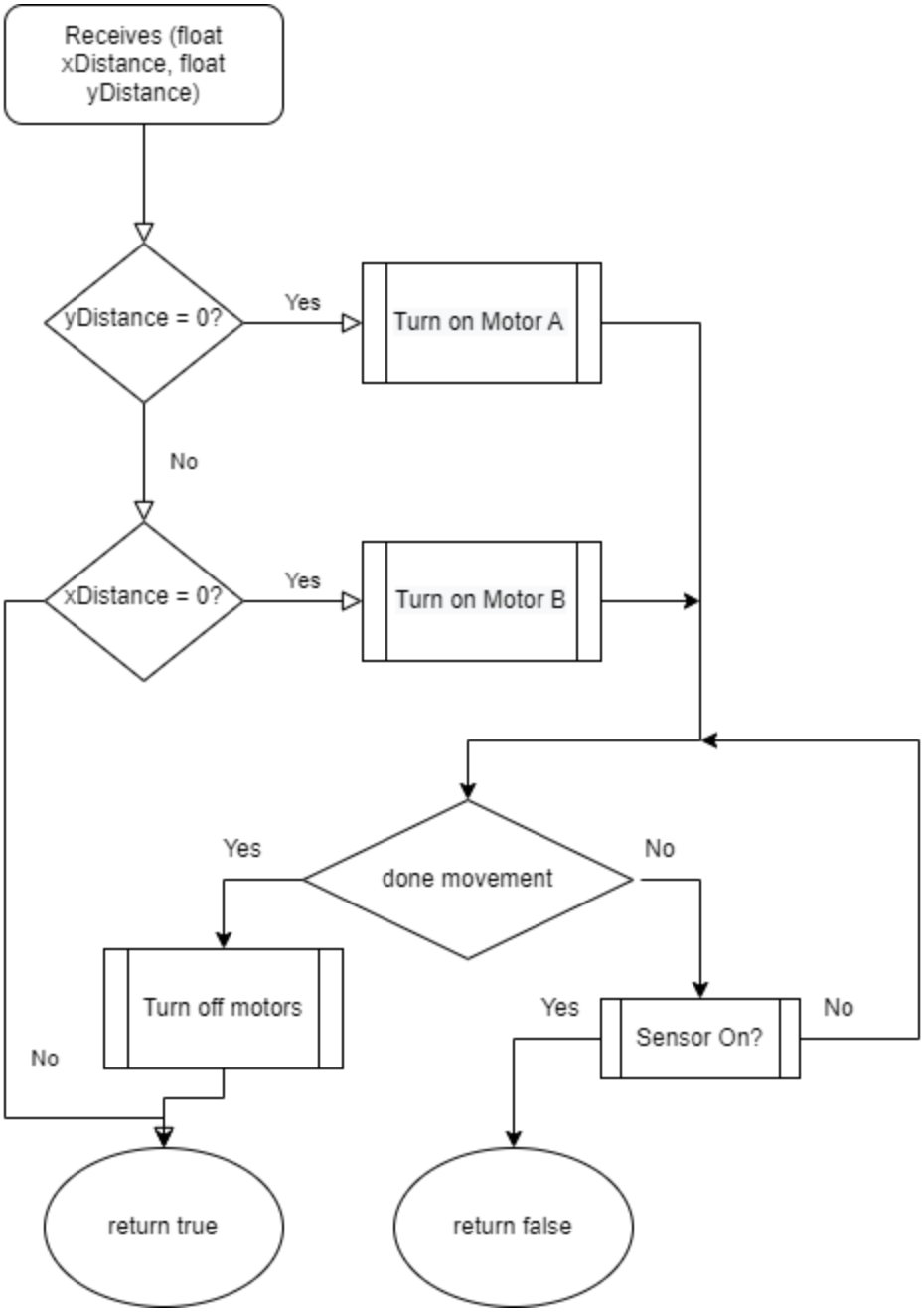
Appendix C: bool sensorIsOn()



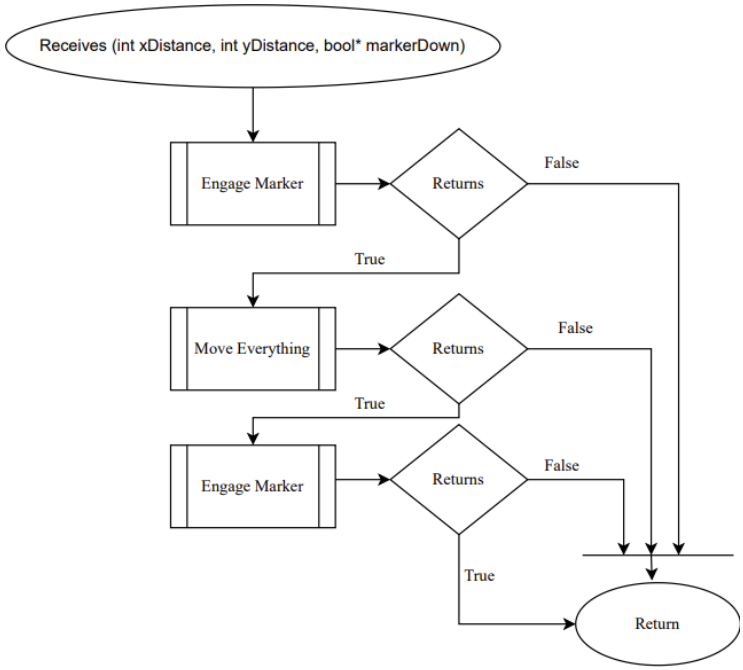
Appendix D: bool engageMarker()



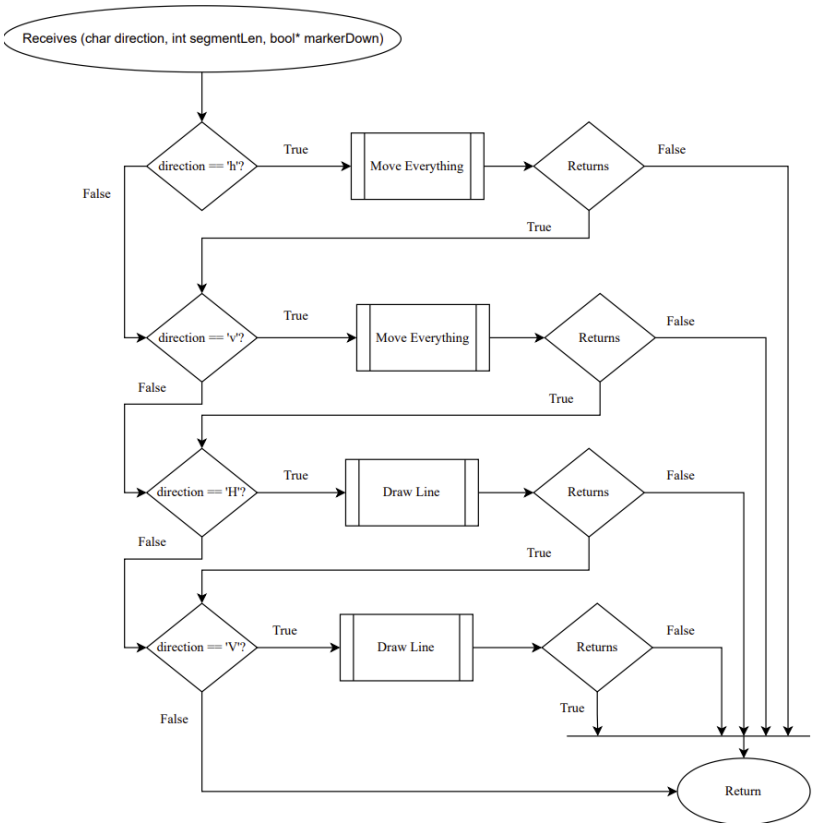
Appendix E: bool moveEverything()



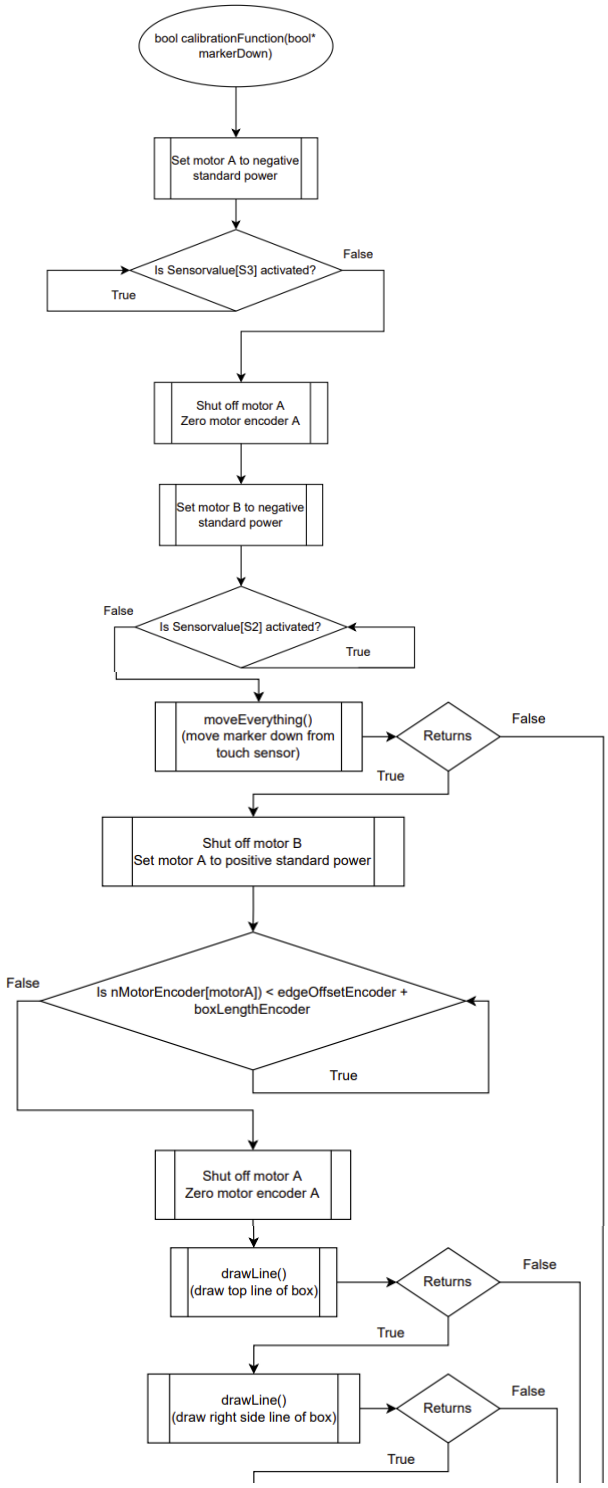
Appendix F: bool drawLineFunction()

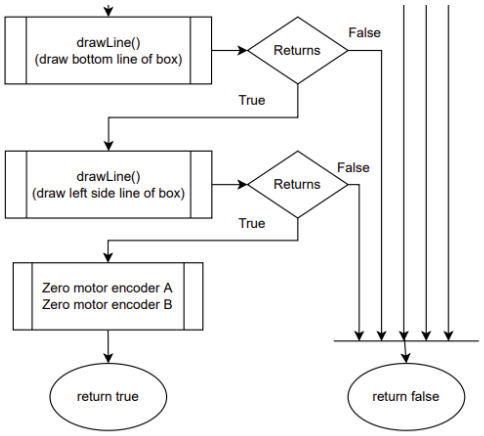


Appendix G: bool drawCharPart()

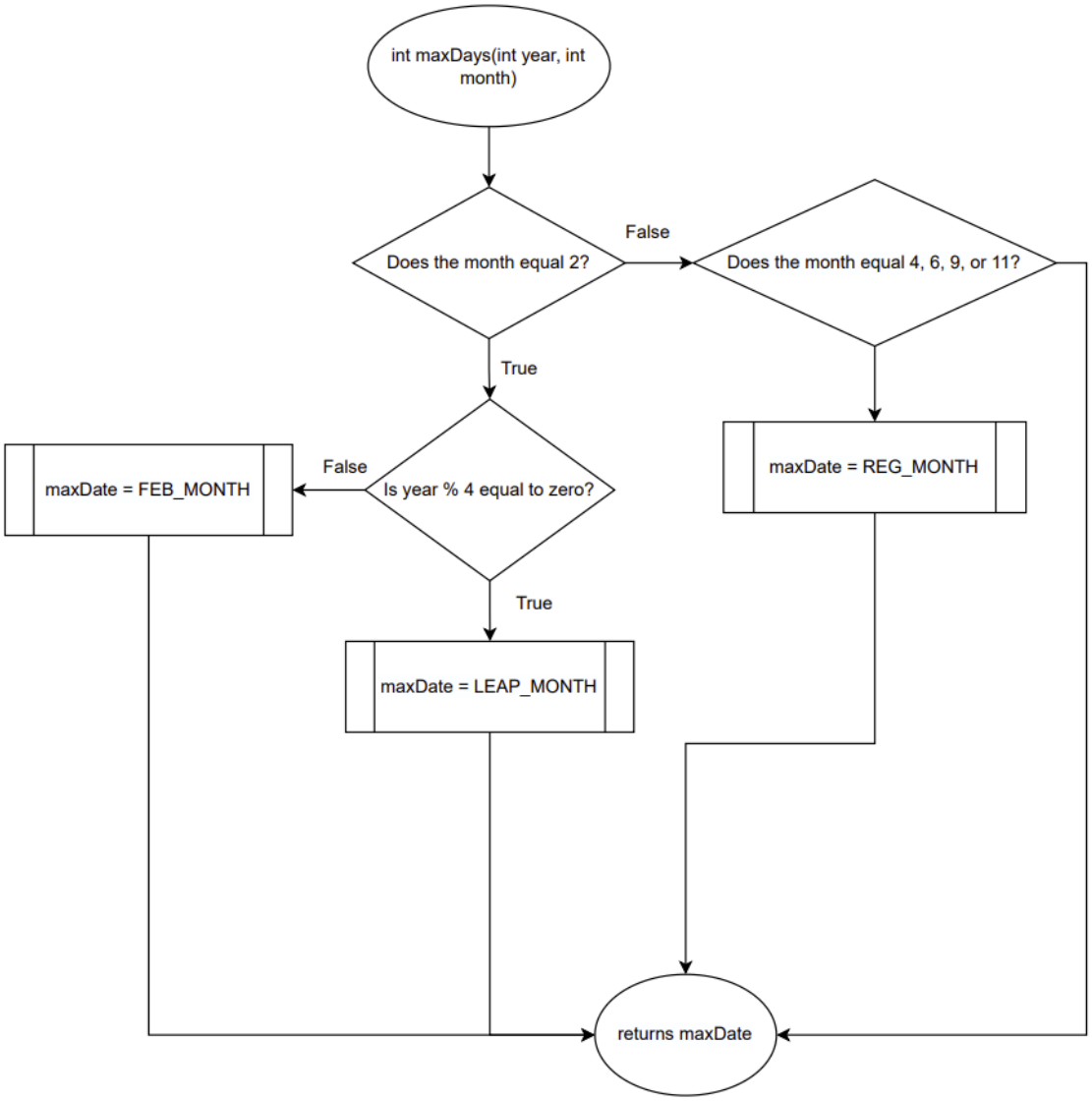


Appendix H: bool calibrationFunction()

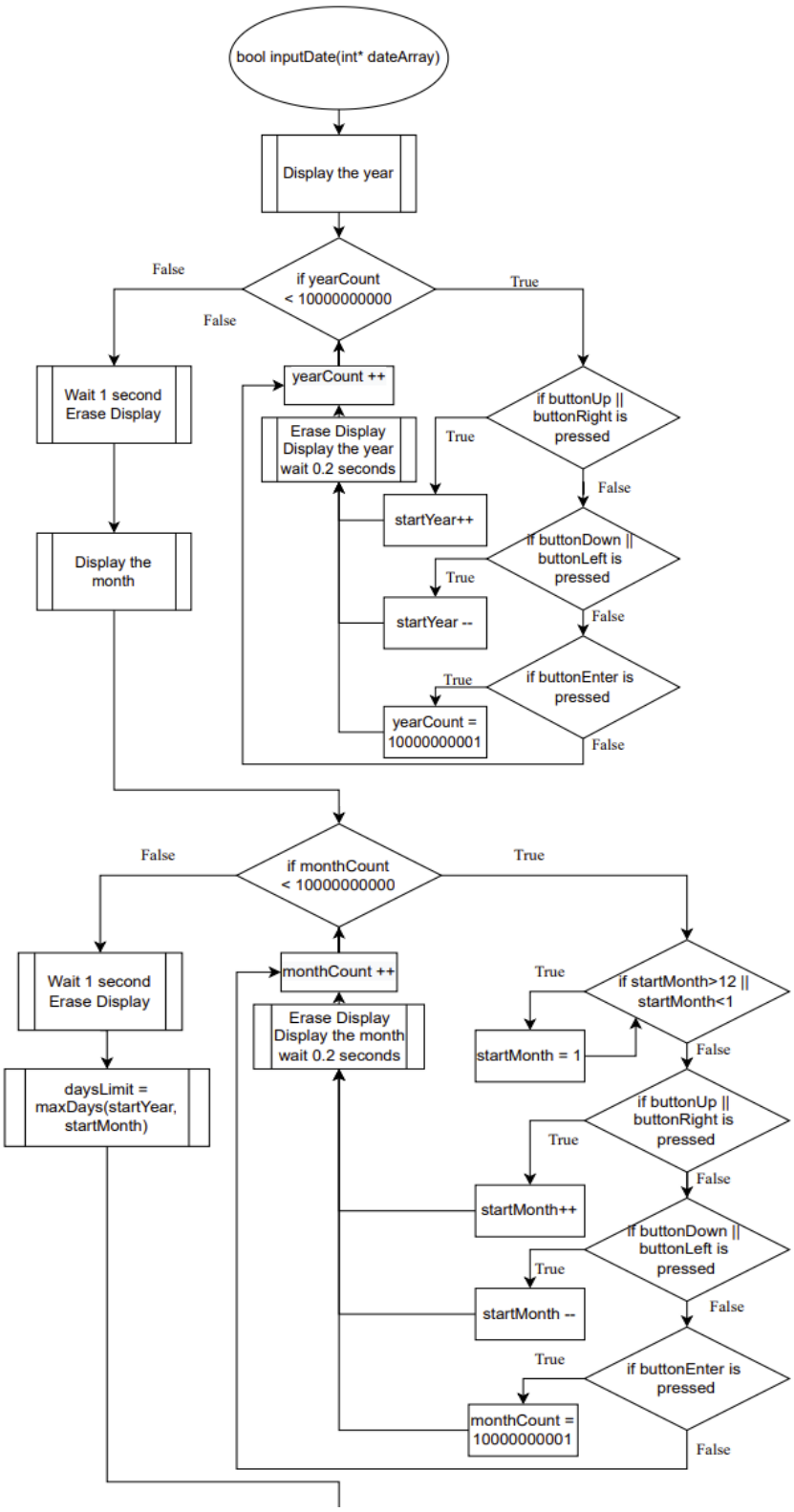


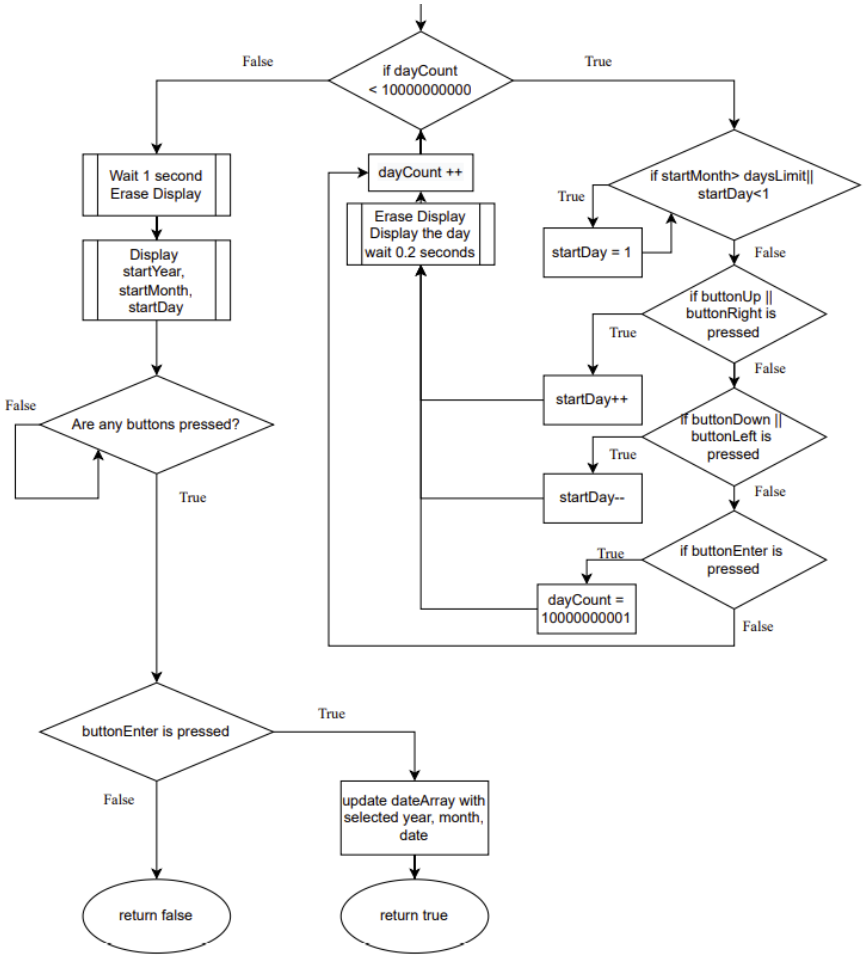


Appendix I: int maxDays()

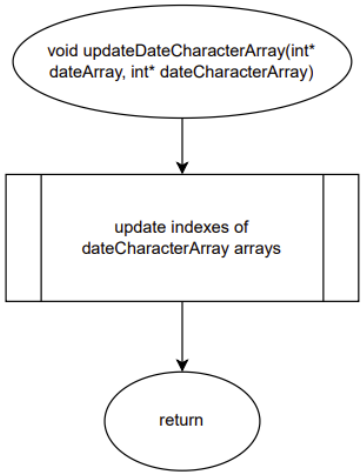


Appendix J: bool inputDate()

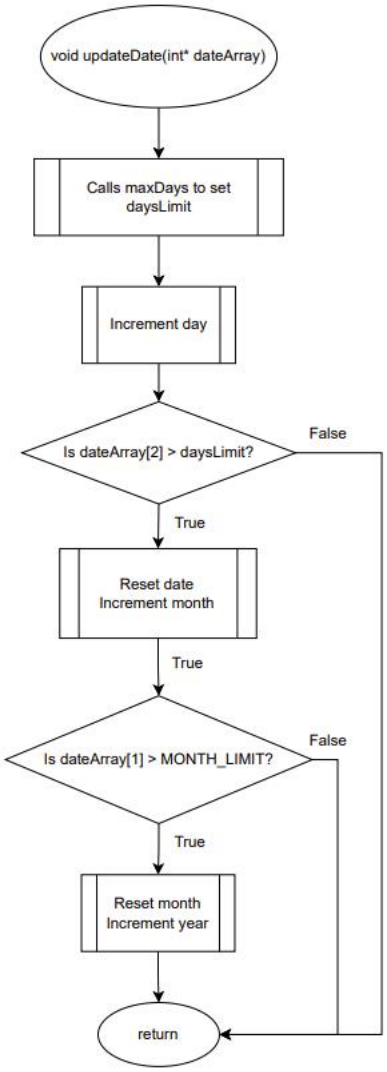




Appendix K: void updateDateCharacterArray()



Appendix L: void updateDate()



Appendix M: RobotC Source Code

```
const int NO_POW = 0, STD_POW = 30;
const float TOTAL_ENCODER_DISTANCE = 1850, WHEEL_DIAMETER_CM = 2.3,
      PINION_DIAMETER_CM = 0.65;
const int BOX_LENGTH_CM = 68, BOX_HEIGHT_CM = 16, SPACE_WIDTH_CM = 2,   CHAR_WIDTH_CM
= 4, SMALL_DIST = 1, EDGE_OFFSET_CM = 10,
SAFE_DIST = 5, TIME_INTERVAL = 864000;

int cmToEncoder(float distanceCM, float diameterFactor);
bool sensorIsOn();
bool engageMarker(bool moveMarker, bool* markerDown);
bool moveEverything(float xDistance, float yDistance);
bool drawLineFunction(int xDistance, int yDistance,
bool* markerDown);
bool drawCharPart(char direction, int segmentLen, bool* markerDown);
bool calibrationFunction(bool* markerDown);

int maxDays(int year, int month);
bool inputDate(int* dateArray);
void updateDateCharacterArray(int* dateArray,
int* dateCharacterArray);
void updateDate(int* dateArray);


task main()
{
      const int dashIndex = 80;
      bool markDown = false;
      bool* markerDown = &markDown;
      bool dateInputted = false;
      bool nonEmergencyExit = false;
      int drawingNum=0, drawingIndex=0;
      int dateArray[3] = {0,0,0};
      int dateCharacterArray[8] = {0,0,0,0,0,0,0,0};

      int segmentLen[88] = {2, 1, -2, -1, 2, 0, 0, 0,
                            2, 0, 0, 0, 0, 0, 0, 0,
                            1, -1, 1, 1, 1, -1, 0, 0,
                            1, -1, 1, 1, -1, 1, 1, -1,
                            1, 1, -1, 1, -1, 1, 0, 0,
                            1, 1, -1, 1, 1, -1, 0, 0,
                            1, 1, -1, 1, 1, -1, -1, 1,
                            1, -1, 2, 0, 0, 0, 0, 0,
                            1, 2, -1, -2, 1, 1, -1, 1,
                            1, 1, -1, -1, 2, 0, 0, 0,
                            1, 1, -1, 1, 0, 0, 0, 0};

      char directions[88] = {'V', 'H', 'V', 'H', 'V', 'n', 'n', 'n',
                             'V', 'n', 'n', 'n', 'n', 'n', 'n', 'n',
                             'H', 'H', 'V', 'H', 'V', 'H', 'n', 'n',
                             'H', 'H', 'V', 'H', 'H', 'V', 'H', 'H',
                             'V', 'H', 'V', 'V', 'H', 'V', 'n', 'n',
                             'H', 'V', 'H', 'V', 'H', 'H', 'n', 'n',
                             'H', 'V', 'H', 'V', 'H', 'V', 'H', 'V',
```

```
                    'H', 'H', 'V', 'n', 'n', 'n', 'n', 'n',
                    'H', 'V', 'H', 'V', 'H', 'V', 'H', 'V',
                    'V', 'H', 'V', 'H', 'V', 'n', 'n', 'n',
                    'h', 'v', 'H', 'v', 'n', 'n', 'n', 'n'};


while(!dateInputted)
{
      dateInputted = inputDate(dateArray);
}

while(!SensorValue(S3))
{}
wait1Msec(2000);

time100(T1) = 0;

while(nonEmergencyExit==false)
{
      clearTimer(T1);
      updateDateCharacterArray(dateArray, dateCharacterArray);

      if (!calibrationFunction(markerDown)) return;
      if (!moveEverything(-(2*CHAR_WIDTH_CM+SMALL_DIST), 0))return;
      if (!moveEverything(0,CHAR_WIDTH_CM)) return;

      for(int numCount=0; numCount < 8; numCount++)
      {
            drawingNum = dateCharacterArray[numCount];
            drawingIndex = dateCharacterArray[numCount]*8;

            for(int strokeCount = 0; strokeCount<8;strokeCount++)
            {
                  if(!drawCharPart(directions[drawingIndex+strokeCount],
                                   segmentLen[drawingIndex+strokeCount],
                                   markerDown)) return;
            }

            if (!moveEverything(-(SPACE_WIDTH_CM+CHAR_WIDTH_CM), 0)) return;
            if (!moveEverything(0,-(2*CHAR_WIDTH_CM))) return;

            if(numCount==3 || numCount==5)
            {
                  for(int dashCount = 0; dashCount<8; dashCount++)
                  {
                        if (!drawCharPart(directions[dashIndex+dashCount],
                        segmentLen[dashIndex+dashCount], markerDown))
                        return;
                  }

                  if (!moveEverything(-(SPACE_WIDTH_CM +CHAR_WIDTH_CM),0))
                  return;
                  if (!moveEverything(0,-(2*CHAR_WIDTH_CM)))return;
            }
      }
```

```
		motor[motorA] = -STD_POW;
		while (SensorValue[S3] == 1)
		{
			if(sensorIsOn()) return;
		}
		motor[motorA] = 0;

		//zeroes encoders
		nMotorEncoder[motorA] = 0;
		motor[motorB] = -STD_POW;
		while (!SensorValue[S2])
		{
			if (sensorIsOn()) return;
		}
		motor[motorB] = 0;

		motor[motorB] = STD_POW;
		wait1Msec(250);
		motor[motorB] = 0;

		motor[motorA] = STD_POW;
		wait1Msec(2000);
		motor[motorA]= 0;

		while(time100[T1] < TIME_INTERVAL)
// wait time is 24 hours
		{
			if (sensorIsOn()) return;
			if(getButtonPress(buttonEnter))
			{
				nonEmergencyExit = true;
			}
		}
		wait1Msec(2000);
		updateDate(dateArray);
	}
}
```

```
bool sensorIsOn()
{
  return SensorValue[S4] < SAFE_DIST;
}

bool engageMarker(bool moveMarker, bool* markerDown)
{
	int motorCPower = -100;
	int encoderInitial = nMotorEncoder[motorC];
	int encoderChange = TOTAL_ENCODER_DISTANCE;

	if (moveMarker != *markerDown)
	{
		if (moveMarker == true)
		{
			motorCPower *= -1;
		}

		motor[motorC] = motorCPower;
		while (abs(nMotorEncoder[motorC]-encoderInitial) < abs(encoderChange))
		{if (sensorIsOn()) return false;}
		motor[motorC] = NO_POW;

		if(*markerDown == true)
		{*markerDown = false;}
		else
		{*markerDown = true;}
	}

return true;
}

bool moveEverything(float xDistance, float yDistance)
{
	float xEncoderChange = cmToEncoder(xDistance, WHEEL_DIAMETER_CM);
	float yEncoderChange = cmToEncoder(yDistance, PINION_DIAMETER_CM);

	float xEncoderInitial = nMotorEncoder[motorA];
	float yEncoderInitial = nMotorEncoder[motorB];

	if (yDistance == 0)
	{
		motor[motorA] = STD_POW *(xDistance/abs(xDistance));
		while(abs(nMotorEncoder[motorA]-xEncoderInitial) < abs(xEncoderChange))
		{if (sensorIsOn()) return false;}
	}

	else if (xDistance == 0)
	{
		motor[motorB] = STD_POW *(yDistance/abs(yDistance));
		while(abs(nMotorEncoder[motorB]-yEncoderInitial) < abs(yEncoderChange))
		{if (sensorIsOn()) return false;}
	}

	motor[motorA] = motor[motorB] = NO_POW;
```

```
return true;
}

bool drawLineFunction(int xDistance, int yDistance, bool* markerDown)
{
	if (!engageMarker(true, markerDown)) return false;
	if (!moveEverything(xDistance, yDistance)) return false;
	if (!engageMarker(false, markerDown)) return false;

	return true;
}

bool drawCharPart(char direction, int segmentLen, bool* markerDown)
{
	//non drawing lines
	if(direction == 'h')
	{
		if (!moveEverything(segmentLen*CHAR_WIDTH_CM, 0)) return false;
	}

	if(direction == 'v')
	{
		if (!moveEverything(0, segmentLen*CHAR_WIDTH_CM)) return false;
	}

	// drawing lines
	if(direction == 'H')
	{
		if (!drawLineFunction(segmentLen*CHAR_WIDTH_CM, 0, markerDown))
		{return false;}
	}

	if(direction == 'V')
	{
		if (!drawLineFunction(0, segmentLen*CHAR_WIDTH_CM,  markerDown))
		{return false;}
	}

	return true;
}

bool calibrationFunction(bool* markerDown)
{
	short edgeOffsetEncoder = cmToEncoder(EDGE_OFFSET_CM, WHEEL_DIAMETER_CM);
	short boxLengthEncoder = cmToEncoder(BOX_LENGTH_CM, WHEEL_DIAMETER_CM);
	//edge check
	motor[motorA] = -STD_POW;
	while (SensorValue[S3] == 1)
	{}
	motor[motorA] = 0;

	//zeroes encoders
	nMotorEncoder[motorA] = 0;
	motor[motorB] = -STD_POW;
```

```
	while (!SensorValue[S2])
	{}
	if (!moveEverything(0,CHAR_WIDTH_CM)) return false;
	motor[motorB] = 0;

	motor[motorA] = STD_POW;
	while (abs(nMotorEncoder[motorA]) < edgeOffsetEncoder + boxLengthEncoder)
	{}
	motor[motorA] = 0;

	nMotorEncoder[motorA] = 0;

	if (!drawLineFunction(-BOX_LENGTH_CM, 0, markerDown)) return false;
//draws top line of box from left to right)
	if (!drawLineFunction(0, BOX_HEIGHT_CM, markerDown)) return false;
//draws right side line from top to bottom)
	if (!drawLineFunction(BOX_LENGTH_CM, 0, markerDown)) return false;
//draws bottom line from right to left)
	if (!drawLineFunction(0, -BOX_HEIGHT_CM, markerDown)) return false;
 //draws left side line from bottom to top)

	nMotorEncoder[motorA] = nMotorEncoder[motorB] = 0;

	return true;
}
```

```
int cmToEncoder(float distanceCM, float diameterFactor)
{
return distanceCM * 180 / (diameterFactor * PI);
}

int maxDays(int year, int month)
{
const int BIG_MONTH = 31, REG_MONTH = 30, LEAP_MONTH = 29,
FEB_MONTH = 28;

int maxDate = BIG_MONTH;

	if (month == 2)
	{
		if (year % 4 == 0)
		{
			maxDate = LEAP_MONTH;
		}

		else
		{
			maxDate = FEB_MONTH;
		}
```

```
	}
	else if (month == 4 || month == 6 || month == 9 || month == 11)
	{
		maxDate = REG_MONTH;
	}

	return maxDate;
}

bool inputDate(int* dateArray)
{
	int startDay = 1, startMonth = 1, startYear = 2022, daysLimit = 0;

	displayString(2, "Enter the year: %d", startYear);
	for(int yearCount = 0; yearCount < 10000000000; yearCount++)
	{
		if (getButtonPress(buttonUp) || getButtonPress(buttonRight))
		{
			startYear += 1;
			eraseDisplay();
			displayString(2, "Enter the year: %d", startYear);
		}
		else if (getButtonPress(buttonDown) || getButtonPress(buttonLeft))
		{
			startYear -= 1;
			eraseDisplay();
			displayString(2, "Enter the year: %d", startYear);
		}
		else if (getButtonPress(buttonEnter))
		{
			eraseDisplay();
			displayString(2, "Selected year: %d", startYear);
			yearCount = 10000000000+1;
		}
		wait1Msec(200);
	}
	wait1Msec(1000);
	eraseDisplay();

	displayString(2, "Enter the month: %d", startMonth);
	for(int monthCount = 0; monthCount < 10000000000; monthCount++)
	{
		if (startMonth > 12 || startMonth < 1)
		{
			startMonth = 1;
			eraseDisplay();
			displayString(2, "Enter the month: %d", startMonth);
		}
		if (getButtonPress(buttonUp) || getButtonPress(buttonRight))
		{
			startMonth += 1;
			eraseDisplay();
			displayString(2, "Enter the month: %d", startMonth);
		}
		else if (getButtonPress(buttonDown) || getButtonPress(buttonLeft))
```

```
	{
		startMonth -= 1;
		eraseDisplay();
		displayString(2, "Enter the month: %d", startMonth);
	}
	else if (getButtonPress(buttonEnter))
	{
		eraseDisplay();
		displayString(2, "Selected month: %d", startMonth);
		monthCount = 10000000000+1;
	}
	wait1Msec(200);
}
wait1Msec(1000);
eraseDisplay();

daysLimit = maxDays(startYear, startMonth);

displayString(2, "Enter the day: %d", startDay);
for(int dayCount = 0; dayCount < 10000000000; dayCount++)
{
	if (startDay > daysLimit || startDay < 1)
	{
		startDay = 1;
		eraseDisplay();
		displayString(2, "Enter the day: %d", startDay);
	}
	if (getButtonPress(buttonUp) || getButtonPress(buttonRight))
	{
		startDay += 1;
		eraseDisplay();
		displayString(2, "Enter the day: %d", startDay);
	}
	else if (getButtonPress(buttonDown) || getButtonPress(buttonLeft))
	{
		startDay -= 1;
		eraseDisplay();
		displayString(2, "Enter the day: %d", startDay);
	}
	else if (getButtonPress(buttonEnter))
	{
		eraseDisplay();
		displayString(2, "Selected day: %d", startDay);
		dayCount = 10000000000+1;
	}
	wait1Msec(200);
}
wait1Msec(1000);
eraseDisplay();

displayString(2, "Is this the correct date?");
displayString(3, "%d %d %d", startYear,startMonth,startDay);
displayString(4, "Press enter to confirm.");
displayString(5, "Press any other button to restart.");
while(!getButtonPress(buttonAny))
```

```
	{}
	eraseDisplay();

	if (getButtonPress(buttonEnter))
	{
		dateArray[0] = startYear;
		dateArray[1] = startMonth;
		dateArray[2] = startDay;

		return true;
	}
	else
	{
		return false;
	}
}

void updateDateCharacterArray(int* dateArray,
int* dateCharacterArray)
{
	const int TEN_FACTOR = 10, HUNDRED_FACTOR = 100,
THOUSAND_FACTOR = 1000;

	dateCharacterArray[0] = dateArray[0] / THOUSAND_FACTOR;
	dateCharacterArray[1] = (dateArray[0] % THOUSAND_FACTOR)/ HUNDRED_FACTOR;
	dateCharacterArray[2] = (dateArray[0] % HUNDRED_FACTOR) / TEN_FACTOR;
	dateCharacterArray[3] = dateArray[0] % TEN_FACTOR;
	dateCharacterArray[4] = dateArray[1] / TEN_FACTOR;
	dateCharacterArray[5] = dateArray[1] % TEN_FACTOR;
	dateCharacterArray[6] = dateArray[2] / TEN_FACTOR;
	dateCharacterArray[7] = dateArray[2] % TEN_FACTOR;
}

void updateDate(int* dateArray)
{
	const int MONTH_LIMIT = 12, RESET = 1;

	int year = dateArray[0];
	int month = dateArray[1];
	int daysLimit = maxDays(year, month);

	dateArray[2]++;

	if (dateArray[2] > daysLimit)
	{
		dateArray[2] = RESET;
		dateArray[1]++;

		if (dateArray[1] > MONTH_LIMIT)
		{
			dateArray[1] = RESET;
			dateArray[0]++;
		}
	}
}
```