Arduino Lab 4

Knut Ola Nøsen

***Abstract*—The project explores the mechanics of line-following as well as task scheduling without a proper threading system.**

***Index Terms*—Arduino, LED, RAMP, Sequential, Multithreading, Zumo, PID, Regulator**

I. Theory

THE the Zumo platform contains a suite of sensors and actuators. In this project, we mostly focus on the light-sensors used To detect guide-lines below the robot, the bi-directional motors driving the belts of the Zumo, as well as the lcd display attached to the top of the board.

A. Software

Lets start by creating some configs to keep our code free of "magic constants". By keeping these configs in separate files, we avoid cluttering our logic in the main.cpp file.

Listing 1. Range.h

```
#ifndef Range_h
#define Range_h

struct Range
{
   int minValue;
   int maxValue;
   Range(int minValue, int maxValue) :
       minValue(minValue),
       maxValue(maxValue)
   {
   }

   int getMidValue() const
   {
      return (minValue + maxValue) / 2;
   }

   int getWidth() const
   {
      return maxValue - minValue;
   }
};

#endif
```

Listing 2. PidControllerConfig.h

```
#ifndef PidControllerConfig_h
#define PidControllerConfig_h
#include <Scaling.h>
#include <Range.h>

struct PidControllerConfig
{
   float kp = 0;
   float ki = 0;
   float kd = 0;
   Range inputRange;
   Range outputRange;
   // Max amount of time between
       updates for the update to be valid
   long updateTimeout = 20;

   PidControllerConfig(
      float kp,
      float ki,
      float kd,
      Range inputRange,
      Range outputRange)
      : kp(kp),
        ki(ki),
        kd(kd),
        inputRange(inputRange),
        outputRange(outputRange)
   {
   }
};

#endif
```

Listing 3. ApplicationConfig.h

```
#include <PidControllerConfig.h>
#include <Range.h>

// NUM_LINE_SENSORS must be defined as a const separately
// of appConfig because it is needed when declaring an array
#define APP_CONFIG_NUM_LINE_SENSORS 5

struct ApplicationConfig
{
   const int baudRate = 9600;
   const int lcdWidth = 8;

   // Speed
   const int targetSpeed = 200;
   const int calibrationSpeed = 200;
   const int maxSpeed = 300;

   // All LEDS will flash x times before the motors start
   const int numSafetyFlashesBeforeStart = 5;
   const int safetyFlashDurationMs = 400;

   // Pid
   PidControllerConfig linePidConfig =
      PidControllerConfig(
         2.0, //                kp
         0.0, //                ki
         0.0, //                kd
         Range(0, 4000), //     inputRange
         Range(-maxSpeed, maxSpeed) // outputRange
      );
} const appConfig;
```

This gives us a global const appConfig when we include the ApplicationConfig.h file in our main sketch.

Listing 4. Timer.h

```
#ifndef Timer_h
#define Timer_h
#include <Arduino.h>

class Timer
{
private:
   long startTime;

public:
   Timer()
   {
      reset();
   }
   void reset()
   {
      startTime = millis();
   }

   unsigned long getElapsedTime()
   {
      return millis() - startTime;
   }

   /*
      Will check if the time since last reset()
      call is greater than the given time
      Note that while loopWait() automatically resets after
      the given time, this function does not
     */
   bool isFinished(const unsigned long durationMs)
   {
      return getElapsedTime() >= durationMs;
   }
   /*
   Will return false until the given time has passed
   Then it will return true and start counting down the same amount again
   Example:
      Timer timer;
      while(true) {
         if(timer.isTimePassed(1000)) {
            // Will run every 1000ms
         }
      }
   */
   bool loopWait(const unsigned long durationMs)
   {
      if (isFinished(durationMs))
      {
         reset();
         return true;
      }
      return false;
   }
};

#endif
```

We also create a Timer class to make time tracking a bit easier and more intuitive.

Listing 5. Scaling.h

```
#ifndef Scaling_h
#define Scaling_h
#include <Range.h>
namespace Scaling
{

   int clamp(const int value, Range range)
   {
      if (value < range.minValue)
      {
         return range.minValue;
      }
      else if (value > range.maxValue)
      {
         return range.maxValue;
      }
      else
      {
         return value;
      }
   }

   // Returns the amount that was left out of a range
   // Example:
   // remainderFromClamp(-1, Range(0, 10)) == -1
   // remainderFromClamp(0, Range(0, 10)) == 0
   // remainderFromClamp(1, Range(0, 10)) == 0
   // remainderFromClamp(9, Range(0, 10)) == 0
   // remainderFromClamp(10, Range(0, 10)) == 0
   // remainderFromClamp(11, Range(0, 10)) == 1
   int remainderFromClamp(const int value, Range range)
   {
      return value - clamp(value, range);
   }

   // Arduino.h
   long map(long x, long in_min, long in_max, long out_min, long out_max)
   {
      return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
   }

   long mapToRange(const long value, Range inputRange, Range outputRange)
   {
      return map(
         value,
         inputRange.minValue,
         inputRange.maxValue,
         outputRange.minValue,
         outputRange.maxValue);
   }

}

#endif
```

This project calls for much manipulation of IO-Ranges. We create an own file to keep this reusable and well-documented. We also copy the implementation of map from Arduino.h, to make such fundemendal tasks less platform-dependent.

Listing 6. PidController.h

```
#include <Scaling.h>
#include <PidControllerConfig.h>
#include <Range.h>
#include <Arduino.h>

class PidController
{

public:
   PidControllerConfig pidConfig;

private:
   double area = 0;
   double lastError = 0;
   long previousUpdateTime = millis();
   double previousUpdateOutput = 0;

public:
   // Set kp, ki and/or kd to 0.0 to disable the respective mode
   PidController(PidControllerConfig pidConfig) : pidConfig(pidConfig)
   {
   }

   double update(double value, double target, bool clampOutput)
   {
      double error = value - target;

      long deltaTime = millis() - previousUpdateTime;
      if (deltaTime > pidConfig.updateTimeout)
      {
         // Ignore this update if it is too old
         previousUpdateTime = millis();
         return previousUpdateOutput;
      }

      double deltaError = error - lastError;

      double centerPointError = (error + lastError) / 2;
      area += centerPointError * deltaTime;

      float kp = pidConfig.kp;
      float ki = pidConfig.ki;
      float kd = pidConfig.kd;

      double output = kp * error + ki * area + kd * deltaError;

      lastError = error;
      previousUpdateTime = millis();

      int inputRangeWidth = pidConfig.inputRange.getWidth();
      double scaledOutput = Scaling::mapToRange(
         output,
         Range(-inputRangeWidth, inputRangeWidth),
         pidConfig.outputRange);

      previousUpdateOutput =
         clampOutput
            ? Scaling::clamp(scaledOutput, pidConfig.outputRange)
            : scaledOutput;

      return previousUpdateOutput;
   }
};
```

To abstract the complex inner workings of the PidController we create an own class to encapsulate this logic.

Finally for our last external file we create a state-machine to handle task scheduling for our sequential code.

Listing 7. Sequence.h

```
#ifndef Sequence_h
#define Sequence_h
#include <Arduino.h>
#include <Timer.h>

/*
Warnings:
The & symbol inside the capture-clause [&]
of the lambda will cause all variables that
are used in the given lambda to be referenced
rather than copied. Not doing this could cause
funky issues that are next to impossible to debug.

In short:
Always start your lambdas with [&] when using Sequence!

About the empty comments in the example below:
The // comments at the end of each .then({ //
are there to make the formatter snap codelines
to a new line. If you do not include this the
code becomes ugly and hard to read.

Example usage:

struct Sequences {
   Sequence red;
   Sequence yellow;
} sequences;

void loop()
{
   sequences.red
      .then([&] { //
         ledRed(true);
      })
      .delay(2000)
      .then([&] { //
         ledRed(false);
      })
      .delay(2000)
      .loop()
      .endOfSequence();

   sequences.yellow
      .then([&] { //
         ledYellow(true);
      })
      .delay(200)
      .then([&] { //
         ledYellow(false);
      })
      .delay(200)
      .loop()
      .endOfSequence();
}
*/
class Sequence
{
private:
   int timesPreviousStepLooped = 0;

   int timesLooped = 0;
   int timesToLoop = 0;

   // Keeps track of what step we should be executing
   int sequenceStep = 0;
   // Keeps track of how many steps we have registered
   int checkedSteps = 0;
   // Keeps track of how many steps we registered last time
   // we ran the sequence (the last known value of checkedSteps)
   // Used to determine how many steps are in the sequence
   // and by extension, if the sequence execution has finished executing.
   int numSteps = 0;
   // Will be true after the last step has executed and during the execution
   // of the first step after a sequence restart.
   bool sequenceHasFinished = false;
   bool shouldLoop = false;
   bool endSequenceShouldReturnTrueBetweenLoops = true;
   bool paused = false;

public:
   Timer timer;
   void moveSteps(int deltaSteps)
   {
      sequenceStep += deltaSteps;
      timer.reset();
   }
   // Will run the code provided and go to next sequence step
   // if the function returns true
   template <typename F>
   Sequence &thenWhenReturnsTrue(F callback)
   {
      if (paused)
      {
         return *this;
      }

      if (checkedSteps == sequenceStep)
      {
         if (callback())
         {
            moveSteps(1);
         }
      }
      checkedSteps += 1;
      return *this;
   }

   // Will run the code provided and go to next sequence step
   template <typename F>
   Sequence &then(F callback)
   {
      if (paused)
      {
         return *this;
      }
      if (checkedSteps == sequenceStep)
      {
         callback();
         moveSteps(1);
      }
      checkedSteps += 1;
      return *this;
   }

   // Will reset the sequence to the first step
   void reset()
   {
      checkedSteps = 0;
      sequenceStep = 0;
      timesLooped = 0;
      timesPreviousStepLooped = 0;
      paused = false;
      sequenceHasFinished = false;
   }
```

```
void start()
{
   reset();
}

void pause()
{
   paused = true;
}

void resume()
{
   paused = false;
}

int getTimesSequenceLooped()
{
   return timesLooped;
}

int getTimesPreviousStepLooped()
{
   return timesPreviousStepLooped;
}

/**
 * @brief Repeats n previous steps until callback returns true
 *
 * @param steps How many steps to go back each time
 * @param callback Function to evaluate whether (true) or not (false) to go back
 * @return Sequence& returns the sequence so chaining can continue
 */
template <typename F>
Sequence &repeatPreviousStepsUntil(int steps, F callback)
{
   if (paused)
   {
      return *this;
   }
   if (checkedSteps == sequenceStep)
   {
      // SHould run before callback is checked
      timesPreviousStepLooped += 1;
      if (callback())
      {
         moveSteps(1);
         timesPreviousStepLooped = 0;
      }
      else
      {
         moveSteps(-steps);
      }
   }
   checkedSteps += 1;
   return *this;
}

/**
 * @brief Repeats n previous steps n times
 *
 * @param steps How many steps to go back each time
 * @param times How many times to go back
 * @return Sequence& returns the sequence so chaining can continue
 */
Sequence &repeatPreviousStepsTimes(int steps, int times)
{
   if (paused)
   {
```

```
            return *this;
        }

        return repeatPreviousStepsUntil(steps, [&]() { //
            return timesPreviousStepLooped >= times;
        });
    }

    // Will run the code provided for the specified number of milliseconds
    // and then continue to the next step
    template <typename F>
    Sequence &thenRunFor(long durationMs, F callback)
    {
        if (paused)
        {
            return *this;
        }

        if (checkedSteps == sequenceStep)
        {
            if (timer.isFinished(durationMs))
            {
                moveSteps(1);
            }
            else
            {
                callback();
            }
        }
        checkedSteps += 1;
        return *this;
    }

    // Will wait until the specified time has passed and then go to next sequence
        step
    Sequence &delay(long delayMs)
    {
        if (paused)
        {
            return *this;
        }
        return thenRunFor(delayMs, [&] {});
    }

    // Restart sequence from beginning when sequence has finished executing
    Sequence &loop()
    {
        if (paused)
        {
            return *this;
        }

        if (checkedSteps == sequenceStep)
        {
            endSequenceShouldReturnTrueBetweenLoops = true;
            shouldLoop = true;
            sequenceStep += 1;
        }

        checkedSteps += 1;
        return *this;
    }

    // Restart sequence from beginning when sequence has finished executing
    // until the callback returns true
    template <typename F>
    Sequence &loopUntil(F callback)
    {
        if (paused)
        {
            return *this;
        }

        if (checkedSteps == sequenceStep)
        {
            endSequenceShouldReturnTrueBetweenLoops = false;
            shouldLoop = !callback();
            sequenceStep += 1;
        }

        checkedSteps += 1;
        return *this;
    }

    /**
     * Will prevent endSequence from returning true until the
     * specified number of times has been looped
     * When the specified number of times has been looped,
     * the sequence will start executing from the first step
     * just like with a normal loop
     * @param restartAfterCompletion true
     * Will restart the sequence from the first step
     * after the specified number of times has been looped.
     * During the last update after the sequence has run the
     * specified number of times, endOfSequence will return true
     * before the sequence restarts and counts from zero.
     * Leave this as true if you do not intend to manually reset the
     * sequence between uses.
     */
    Sequence &loopTimes(int times, bool restartAfterCompletion)
    {
        if (paused)
        {
            return *this;
        }

        if (checkedSteps == sequenceStep)
        {
            endSequenceShouldReturnTrueBetweenLoops = false;
            shouldLoop = true;
            sequenceStep += 1;
            timesToLoop = times;
            // timesLooped should be incremented before the check
            timesLooped += 1;
            if (timesLooped == timesToLoop)
            {
                endSequenceShouldReturnTrueBetweenLoops = true;
                // This will allow the sequence to restart from the first step
                // and simply return true after the specified number of times has been
                     looped
                // This way there is a queue for when the fixed number of loops has
                     been completed
                // and the sequence can be restarted from the first step just like
                     with a normal
                // loop. This results in reusable sequences that do not require a
                     manual restart.
                shouldLoop = restartAfterCompletion;
                timesToLoop = 0;
                timesLooped = 0;
            }
        }

        checkedSteps += 1;
        return *this;
    }

    // Returns true if the sequence has finished executing and should be restarted
    // to continue from the first step
```

```
   bool hasFinished()
   {
      return sequenceHasFinished;
   }

   // Mark end of sequence and return true if sequence has finished executing
   bool endOfSequence()
   {
      if (paused)
      {
         return true;
      }

      if (numSteps < checkedSteps)
      {
         numSteps = checkedSteps;
      }
      checkedSteps = 0;

      sequenceHasFinished = sequenceStep == numSteps;

      if (sequenceHasFinished)
      {
         if (shouldLoop)
         {
            sequenceStep = 0;
            return endSequenceShouldReturnTrueBetweenLoops;
         }
         else
         {
            // The sequence execution
            // is complete and there is
            // no looping. This block will
            // run until the sequence is
            // restarted manually by the user
            return true;
         }
      }

      return false;
   }
};

#endif
```

Now that all our external files are set up, we can start looking at the main.cpp file.

Listing 8. main.cpp

```
#include <Arduino.h>
#include <ApplicationConfig.h>
#include <Zumo32U4.h>
#include <Timer.h>
#include <Range.h>
#include <Sequence.h>
#include <PidController.h>
#include <Scaling.h>

Zumo32U4ButtonA buttonA;
Zumo32U4ButtonB buttonB;
Zumo32U4Motors motors;
Zumo32U4LineSensors lineSensors;
Zumo32U4LCD lcd;

struct Sequences
{
   // Exercises
   Sequence exercise1;
   Sequence exercise2;
   Sequence exercise3;
   Sequence exercise4;

   // Functions
   Sequence moveEight;
   Sequence flashAllLeds;
   Sequence calibration;
   Sequence printPosition;
   Sequence lcdScroll;
   Sequence loop;

   // Groups
   struct FollowLineSequences
   {
      Sequence followLine;
      Sequence printSpeed;
   } followLineGroup;
} sequences;

struct ApplicationState
{
   PidController linePidController = PidController(appConfig.linePidConfig);

   // Set by printMessage functions
   // to make the background lcd scroll
   // sequence start/stop scrolling
   bool scrollLcd = false;

   // TODO: Implement EEPROM
   // Changing the shape of this struct will change the way
   // the data is stored in EEPROM.
   // Any existing EEPROM will no longer be usable.
   struct SensorData
   {
      unsigned int lineSensorValues[APP_CONFIG_NUM_LINE_SENSORS];
   } memorizedValues;
} state;

void blockingCountdown(const String message, const int durationMillis)
{
   for (int i = durationMillis; i > 0; i--)
   {
      lcd.clear();
      lcd.gotoXY(0, 0);
      lcd.print(message);
      lcd.gotoXY(0, 1);
      lcd.print(i);
      delay(1000);
   }
   lcd.clear();
}

void setup()
{
   Serial.begin(9600);
   lineSensors.initFiveSensors();
   blockingCountdown("Starting ...", 3);
}

void exercise1()
{
   // By adding || exercise1Sequence.hasFinished()
   // we ensure that the sequence will execute in its entirety
   // even after the button is released. This ensures that
   // the led will have the same state as it appears
   // at the end of the sequence.
   if (buttonA.isPressed() || sequences.exercise1.hasFinished())
   {
      sequences.exercise1
         .then([&] { //
            ledGreen(true);
         })
         .delay(100)
         .then([&] { //
            ledGreen(false);
         })
         .delay(100)
         .loop()
         .endOfSequence();
   }
}

/**
 * @return true when sequence is finished
 */
bool moveEight(int speed)
{
   return sequences.moveEight
      .then([&] { //
         motors.setSpeeds(0, speed);
      })
      .delay(2000)
      .then([&] { //
         motors.setSpeeds(speed, speed);
      })
      .delay(1000)
      .then([&] { //
         motors.setSpeeds(speed, 0);
      })
      .delay(2000)
      .loop()
      .endOfSequence();
}

void moveCircle(int speed, bool direction)
{
   int slowSpeed = speed / 3;
   if (direction)
   {
      motors.setSpeeds(slowSpeed, speed);
   }
   else
   {
      motors.setSpeeds(slowSpeed, speed);
   }
```

```
}

void exercise2()
{
   sequences.exercise2
      .thenWhenReturnsTrue([&] { //
         // moveEight returns true when its sequence has finished
         // this function will continue to the next step when
         // its return value is true.
         // That means that this sequence will continue
         // after the moveEight sequence finishes.
         return moveEight(appConfig.targetSpeed);
      })
      .thenWhenReturnsTrue([&] { //
         return moveEight(appConfig.targetSpeed);
      })
      .then([&] { //
         moveCircle(appConfig.targetSpeed, true);
      })
      .delay(2000)
      .then([&] { //
         moveCircle(appConfig.targetSpeed, false);
      })
      .delay(2000)
      .loop()
      .endOfSequence();
}

void printMessage(const String message, const String message2)
{
   lcd.clear();
   lcd.gotoXY(0, 0);
   lcd.print(message);
   lcd.gotoXY(0, 1);
   lcd.print(message2);
   state.scrollLcd = false;
}

void printValues(const float message, const float message2)
{
   printMessage(String(message, 2), String(message2, 2));
}

void printMessage(const String message)
{
   printMessage(message, "");
}

void printScrollMessage(const String message, const String message2)
{
   printMessage(message, message2);
   state.scrollLcd = true;
}

/**
 * @return true when sequence is finished
 */
bool flashAllLeds(const int numFlashes, const int flashDuration)
{
   return sequences.flashAllLeds
      .then([&] { //
         ledGreen(true);
         ledRed(true);
         ledYellow(true);
      })
      .delay(flashDuration)
      .then([&] { //
         ledGreen(false);
         ledRed(false);
```

```
            ledYellow(false);
        })
        .delay(flashDuration)
        .loopTimes(numFlashes, true)
        .endOfSequence();
}

/**
 * @return true when sequence is finished
 */
bool calibrate(const int speed)
{
    // This if will run in parallell with the sequence
    if (buttonA.getSingleDebouncedPress())
    {
        motors.setSpeeds(0, 0);
        sequences.calibration.reset();
        return true;
    }
    return sequences.calibration
        .then([&] { //
            printMessage("Press A", "to cancel");
        })
        .then([&] { //
            printMessage("Starting in", "3");
        })
        .delay(1000)
        .then([&] { //
            printMessage("Starting in", "2");
        })
        .delay(1000)
        .then([&] { //
            printMessage("Starting in", "1");
        })
        .delay(1000)
        .then([&] { //
            printMessage("Calibrating", "...");
        })
        .thenWhenReturnsTrue([&] { //
            return flashAllLeds(
                appConfig.numSafetyFlashesBeforeStart,
                appConfig.safetyFlashDurationMs);
        })
        .thenRunFor(2000, [&] { //
            motors.setSpeeds(-speed, speed);
            lineSensors.calibrate();
        })
        .thenRunFor(2000, [&] { //
            motors.setSpeeds(speed, -speed);
            lineSensors.calibrate();
        })
        .repeatPreviousStepsTimes(2, 2)
        .then([&] { //
            motors.setSpeeds(0, 0);
            printMessage("Done!");
        })
        .delay(1000)
        // By using loop this function can be used multiple times without
        // having to reset the sequence.
        .loop()
        .endOfSequence();
}

int getLineSensorValue()
{
    // NB! The variable passed to readLine is a reference that will be
    // updated by the function. Make sure you pass the ACTUAL
    // variable that should receive the changes.
    // In short, you can NOT make a variable to store the value of
```

```
   // state.memorizedValues.lineSensorValues
   // and then pass this to readLine.
   return lineSensors.readLine(state.memorizedValues.lineSensorValues);
}

void printValue(const String message, const String value)
{
   lcd.clear();
   lcd.gotoXY(0, 0);
   lcd.print(message);
   lcd.gotoXY(0, 1);
   lcd.print(value);
}

void printValue(const String message, const int value)
{
   printValue(message, String(value, 2));
}

/**
 * @return true if the sequence is finished
 */
bool printPosition()
{
   return sequences.printPosition
      .then([&] { //
         printMessage("Press A", "to cancel");
      })
      .delay(3000)
      .thenWhenReturnsTrue([&] { //
         const int position = getLineSensorValue();
         printValue("Position: ", position);
         return buttonA.getSingleDebouncedPress();
      })
      .loop()
      .endOfSequence();
}

String getProgressBar(const int value, const Range range)
{
   const int numDots = Scaling::mapToRange(
      value, range, Range(0, appConfig.lcdWidth));

   String bar = "";
   for (int i = 0; i < numDots; i++)
   {
      bar += "|";
   }
   return bar;
}

/**
 * @return true if the sequence is finished
 */
bool followLine()
{
   // This if will run in parallell with the sequence
   if (buttonA.getSingleDebouncedPress())
   {
      motors.setSpeeds(0, 0);
      sequences.followLineGroup.followLine.reset();
      return true;
   }
   return sequences.followLineGroup.followLine
      .then([&] { //
         printMessage("Press A", "to cancel");
      })
      .delay(1000)
      .then([&] { //
         printMessage("Press B to", "show sp/pos");
      })
      .delay(1000)
      .then([&] { //
         printMessage("Starting in", "3");
      })
      .delay(1000)
      .then([&] { //
         printMessage("Starting in", "2");
      })
      .delay(1000)
      .then([&] { //
         printMessage("Starting in", "1");
      })
      .delay(1000)
      .then([&] { //
         printMessage("Following", "line ...");
      })
      .thenWhenReturnsTrue([&] { //
         return flashAllLeds(
            appConfig.numSafetyFlashesBeforeStart,
            appConfig.safetyFlashDurationMs);
      })
      .thenWhenReturnsTrue([&] { //
         Range outputRange = appConfig.linePidConfig.outputRange;
         Range inputRange = appConfig.linePidConfig.inputRange;
         // const int sensorValue = currentPosition;
         const int sensorValue = getLineSensorValue();

         const int output = state.linePidController.update(sensorValue, 2000,
            true);

         // Assign the output to the left
         const int leftSpeed = appConfig.targetSpeed + output;
         // Take what ever is left over and assign it to the right along with the
            output
         const int unusedLeftSpeed = Scaling::remainderFromClamp(leftSpeed,
            outputRange);
         const int rightSpeed = appConfig.targetSpeed - output - unusedLeftSpeed;
         // Take what ever is left over and assign/re-assign it to the left
         const int unusedRightSpeed = Scaling::remainderFromClamp(rightSpeed,
            outputRange);
         const int rightCompensatedLeftSpeed = leftSpeed - unusedRightSpeed;
         // Clamp the distributed speeds to the output range
         const int clampedLeftSpeed = Scaling::clamp(rightCompensatedLeftSpeed,
            outputRange);
         const int clampedRightSpeed = Scaling::clamp(rightSpeed, outputRange);

         sequences.followLineGroup.printSpeed
            .thenWhenReturnsTrue([&] { //
               printMessage(
                  "Position",
                  getProgressBar(sensorValue, inputRange));

               return buttonB.getSingleDebouncedPress();
            })
            .thenWhenReturnsTrue([&] { //
               printValues(sensorValue, output);
               return buttonB.getSingleDebouncedPress();
            })
            .thenWhenReturnsTrue([&] { //
               printValues(clampedLeftSpeed, clampedRightSpeed);
               return buttonB.getSingleDebouncedPress();
            })
            .loop()
            .endOfSequence();

         motors.setSpeeds(clampedLeftSpeed, clampedRightSpeed);
         return buttonA.getSingleDebouncedPress();
```

```
      })
      .then([&] { //
         // Kill the motors after the user stops the program
         motors.setSpeeds(0, 0);
      })
      .loop()
      .endOfSequence();
}

void exercise3()
{
   sequences.exercise3
      .then([&] { //
         printMessage("Press A to", "calibrate");
      })
      .thenWhenReturnsTrue([&] { //
         return buttonA.getSingleDebouncedRelease();
      })
      .thenWhenReturnsTrue([&] { //
         return calibrate(appConfig.calibrationSpeed);
      })
      .thenWhenReturnsTrue([&] { //
         return printPosition();
      })
      .loop()
      .endOfSequence();
}

void exercise4()
{
   sequences.exercise4
      .then([&] { //
         printMessage("Press A to", "calibrate");
      })
      .thenWhenReturnsTrue([&] { //
         return buttonA.getSingleDebouncedRelease();
      })
      .thenWhenReturnsTrue([&] { //
         return calibrate(appConfig.calibrationSpeed);
      })
      .then([&] { //
         printMessage("Press A to", "follow line");
      })
      .thenWhenReturnsTrue([&] { //
         return buttonA.getSingleDebouncedRelease();
      })
      .thenWhenReturnsTrue([&] { //
         return followLine();
      })
      .loop()
      .endOfSequence();
}

// Keeps the LCD scrolling constantly
// so that this does not have to be implemented
// elsewhere in the code.
void updateLcdScroll()
{
   if (false)
   {
      sequences.lcdScroll
         .then([&] { //
            lcd.scrollDisplayLeft();
         })
         .delay(300)
         .repeatPreviousStepsTimes(2, 20)
         .then([&] { //
            lcd.scrollDisplayRight();
         })
         .repeatPreviousStepsTimes(1, 20)
         .delay(1500)
         .loop()
         .endOfSequence();
   }
}

void updateMenuControls()
{
   sequences.loop
      .then([&] { //
         printScrollMessage("A: Calibrate", "B: Follow Line");
         sequences.calibration.reset();
         sequences.followLineGroup.followLine.reset();
         sequences.calibration.pause();
         sequences.followLineGroup.followLine.pause();
      })
      .thenWhenReturnsTrue([&] { //
         if (buttonA.getSingleDebouncedPress())
         {
            sequences.calibration.start();
            return true;
         }
         else if (buttonB.getSingleDebouncedPress())
         {
            sequences.calibration.start();
            sequences.followLineGroup.followLine.start();
            return true;
         }
         return false;
      })
      // Only one of these two next sequences should be running at a time.
      // because we reset them in the steps above
      //.thenWhenReturnsTrue([&] { //
      .thenWhenReturnsTrue([&] { //
         // Will continue if the sequence is not running
         return calibrate(appConfig.calibrationSpeed);
      })
      .thenWhenReturnsTrue([&] { //
         // Will continue if the sequence is not running
         return followLine();
      })
      .loop()
      .endOfSequence();
}

void loop()
{
   updateLcdScroll();
   updateMenuControls();
   // Delay the entire loop for one millisecond
   // since our code is now blazingly fast
   delay(1);
}
```

II. Discussion

In the previous two labs, my discussion contained the following:

The current code works excellently, however it does have a weakness. While the nested nature of this code makes for very few instances of ”state” and globals, it does prevent us from running continous updates on anything while a piece of code is executing. In a larger project, i believe it would be beneficial to avoid while and for-loops, in favour of a more flat architecture with state machines. That way, the project remains scalable, and we can easily add continous checks without risking spagheti code and human errors due to forgetting to call an updater during a special loop.

Here, we have finally addressed this problem by writing the Scaling.h library. This is reusable and should cover our needs for the forseable future in regards to getting rid of state-machines.