# Arduino Lab 1

Knut Ola Nøsen, Kristian Hansen,

***Abstract*—The project centers around an ultrasonic proximity sensor, and an OLED Display that shows a revolving set of sensor readings in real-time. The purpose of this project is to demonstrate a situation where many isolated components can function in conjunction, while still not interfering with each others operation. Furthermore, the project explores the use of Software-Libraries to improve the quality of the final product, while reducing TTC (Time To Completion). The contents of this paper reflects the aspects most in focus during the execution of the lab. That being my research into best-practices in the IoT industry, rather than the project itself.**

***Index Terms*—Arduino, OLED, C++, PlatformIO, Code-Library, Header File, Test Driven Development, Unit Test, ASSERT, HE-SR04**

## I. Theory

THE HE-SR04 ultrasonic sensor is a low-cost Arduino component for measuring short distances. It gives a relatively inaccurate reading, and out of the box is subject to signal noise. For this reason, a Median-Filter has been implemented through software, to remove the more extreme spikes in telemetry. The inner workings of the ultrasonic sensor are based on ultrasound, commonly found in animals like bats and whales. The following code shows a crude implementation of this sensor:

```
float getDistance(int trigger, int echo) {
   // Disable speaker
   digitalWrite(trigger, LOW);
   // Wait until any existing sound waves
       have been dissipated
   delayMicroseconds(2);
   // Enable the speaker
   digitalWrite(trigger, HIGH);
   // Keep it on long enough for a proper
       signal to be transmitted
   delayMicroseconds(10);
   // Disabe the speaker
   digitalWrite(trigger, LOW);
   // Measure the time in microseconds for
       the audio pulse to return
   long duration = pulseIn(echo, HIGH);
   // Calculate the distance based on the
       speed of sound (343m/s), and divide it
       by two (because the total duration
       includes travel time in both
       directions)
   float distance = duration * 0.0343 / 2;
   return distance;
}
```

There are many more aspects to consider when writing proper code for this sensor, such as temperature changing the speed of sound in the given medium. In order to maintain our sanity, as well as avoid unnecessary mistakes and debugging, it is considered a good practice to avoid writing your own code. For this project we used the HCSR04 Arduino library.

## II. Methods

### A. Hardware

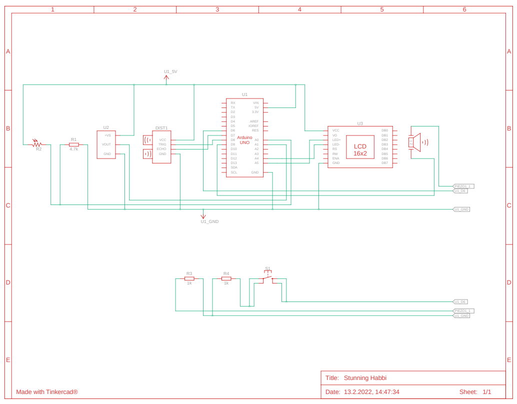

Fig. 1. Wiring Diagram

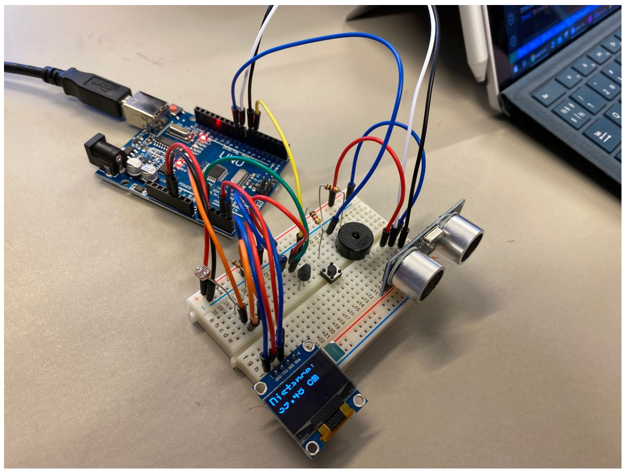

Fig. 2. Lab1 Circuit

## B. Software

We start by defining our enums. To keep the code more readable, we split this code into multiple files

```
enum class AlarmState
{
   OFF,
   ON,
   OVERRIDE
};
```

For the OledState we also have a function for getting the next OledState value. This means we also need a header file. This is a language requirement in C++.

```
#ifndef OledState_h
#define OledState_h

enum class OledState
{
   DISTANCE,
   TEMPERATURE,
   LIGHT
};

const int OLED_STATE_LENGTH = 3;

OledState getNextOledState(OledState oled);

#endif
```

In the OledState.cpp file, we can now import the header file and give getNextOledState an implementation.

```
#include "OledState.h"

OledState getNextOledState(OledState oled)
{
   // use static_cast to convert from enum to
       int
   int currentStateIndex =
       static_cast<int>(oled);
   if (currentStateIndex < (OLED_STATE_LENGTH
       - 1))
   {
      int nextStateIndex = currentStateIndex
          + 1;
      // use static_cast to convert from int
          to enum
      return
          static_cast<OledState>(nextStateIndex);
   }
   int firstStateIndex = 0;
   // use static_cast to convert from int to
       enum
   return
       static_cast<OledState>(firstStateIndex);
}
```

While splitting the code up like this makes for more files and more code to write over all, it does make for more scalable code. PlatformIO also provides us with a lib folder where we can put such files, to keep the code more portable, as well as providing a standardised system for code-splitting.

Now that we have defined our external files, it is time to give the project a configuration.

```
; PlatformIO Project Configuration File
;
; Build options: build flags, source filter
; Upload options: custom upload port, speed
   and extra flags
; Library options: dependencies, extra
   library storages
; Advanced options: extra scripting
;
; Please visit documentation for the other
   options and examples
;
   https://docs.platformio.org/page/projectconf.html

[platformio]
default_envs = uno

[env]
lib_deps =
  martinsos/HCSR04@^2.0.0
  robtillaart/RunningMedian@^0.3.4
  adafruit/Adafruit GFX Library@^1.10.13
  adafruit/Adafruit SSD1306@^2.5.1
  bxparks/AceButton@^1.9.1
  adafruit/Adafruit BusIO@^1.11.0
  bblanchon/ArduinoJson@^6.19.1

[env:uno]
platform = atmelavr
board = uno
framework = arduino
test_ignore = test_desktop
lib_deps = siteswapjuggler/Ramp@^0.6.0

[env:native]
platform = native
debug_test = test_desktop
lib_deps = siteswapjuggler/Ramp@^0.6.0

[env:CI]
platform = native
debug_test = test_desktop
lib_deps = siteswapjuggler/Ramp@^0.6.0
```

Under env we specify lib_deps which is a list of the Code Libraries this project uses. This is where PlatformIO shines compared to the ArduinoIDE, as anyone who opens this project with a PlatformIO compatible IDE (In our case VSCode) will have all dependencies automatically downloaded with the correct version.

We have defined three environments: uno, native and CI. The uno environment is for compiling the project to the arduino uno. It defines the board and framework, which is used by PlatformIO behind the scenes to present us with proper compilation warnings, as well as automatically selecting the proper procedures for uploading the project.

The native environment is intended for testing. Test driven development is a good practice, and PlatformIO presents us with a solid foundation to do this without needing the physical board.

Finally we have the CI environment. CI stands for Continuous Integration, and refers to a automated build system in github. When code in this project is pushed, a linux server will automatically run the native tests, and tell the developer if the code is working and safe to merge into the master branch. This is useful for code developed in teams, so each team member does not need to run the tests manually to ensure nothing was broken by their changes. Here is the CI script:

```yaml
name: Run all native tests
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1
      - name: Set up Python
        uses: actions/setup-python@v1
      - name: Install PlatformIO
        run: |
          python -m pip install --upgrade pip
          pip install platformio
      - name: Run tests in lab1
        working-directory: ./lab1
        run: platformio test -e CI
```

Now that the CI is set up, lets take a look at our Unit Tests:

```cpp
#include <OledState.h>

const OledState last = OledState::LIGHT;

void test_function_getNextOledState_findsNext(void)
{
   const OledState current = OledState::DISTANCE;
   const OledState next = OledState::TEMPERATURE;
   TEST_ASSERT_EQUAL(next, getNextOledState(current));
}

void test_function_getNextOledState_cylesOnLast(void)
{
   const OledState first = static_cast<OledState>(0);
   TEST_ASSERT_EQUAL(first, getNextOledState(last));
}

// Ensures that the OLED_STATE_LENGTH variable is always up to date
void test_function_OledState_lengthVariableMatchesEnumLength(void)
{
   const int lastIndex = static_cast<int>(last);
   TEST_ASSERT_EQUAL(OLED_STATE_LENGTH, lastIndex + 1);
   //"Please update the OLED_STATE_LENGTH variable in OledState.h to represent the number of
       values in the enum";
}

// https://docs.platformio.org/en/latest/plus/unit-testing.html
int main(int argc, char **argv)
{
   UNITY_BEGIN();
   RUN_TEST(test_function_getNextOledState_findsNext);
   RUN_TEST(test_function_getNextOledState_cylesOnLast);
   RUN_TEST(test_function_OledState_lengthVariableMatchesEnumLength);
   UNITY_END();

   return 0;
}
```

The purpose of this code is to check/ASSERT that the code related to the OledState is working properly. We do this by writing test code that uses the production code, and then checks that the outputs are what we expect.

Now, lets look at the main.cpp file:

```
#include "AlarmState.h"
#include <OledState.h>

//
    https://github.com/RobTillaart/RunningMedian/blob/master/examples/RunningMedian/RunningMedian.ino
#include <RunningMedian.h>

// https://github.com/Martinsos/arduino-lib-hc-sr04
#include <HCSR04.h>

// https://github.com/bxparks/AceButton
#include <AceButton.h>
using namespace ace_button;

#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

struct PinConfig
{
   const byte light = A0;
   const byte temp = A1;
   const byte button = 6;
   const byte sonic_echo = 7;
   const byte sonic_trigger = 8;
   const byte buzzer = 9;
} const pinConfig;

struct ApplicationConfig
{
   const int alarmDistanceCmTreshold = 5;
   const int medianFilterWindowSize = 5;
   const float referenceVoltage = 5.0;

   // OLED Display
   const int screenWidth = 128; // OLED display width, in pixels
   const int screenHeight = 64; // OLED display height, in pixels

   // Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
   // The pins for I2C are defined by the Wire-library.
   // On an arduino UNO: A4(SDA), A5(SCL)
   // On an arduino MEGA 2560: 20(SDA), 21(SCL)
   // On an arduino LEONARDO: 2(SDA), 3(SCL), ...
   const byte oledReset = -1; // Reset pin # (or -1 if sharing Arduino reset pin)
   const byte screenAddress = 0x3C; ///< See datasheet for Address; 0x3D for 128x64, 0x3C for
       128x32
} const appConfig;

// Median filters to remove spikes from sensor readings
struct Filters
{
   // Keep the window filter size small to avoid slow response times
   RunningMedian temp = RunningMedian(appConfig.medianFilterWindowSize);
   RunningMedian distance = RunningMedian(appConfig.medianFilterWindowSize);
   RunningMedian light = RunningMedian(appConfig.medianFilterWindowSize);
};

struct ApplicationState
{
   OledState oled = OledState::DISTANCE;
   AlarmState alarm = AlarmState::OFF;
   Filters filters;
} state;

// https://github.com/Martinsos/arduino-lib-hc-sr04
UltraSonicDistanceSensor distanceSensor = UltraSonicDistanceSensor(pinConfig.sonic_trigger,
    pinConfig.sonic_echo);
// Initialize in void setup()
AceButton button(pinConfig.button, HIGH);
// OLED Display Setup
Adafruit_SSD1306 display(appConfig.screenWidth, appConfig.screenHeight, &Wire,
    appConfig.oledReset);

void onClick(AceButton *button, uint8_t eventType, uint8_t buttonState)
{
   if (eventType == AceButton::kEventPressed)
   {
      if (state.alarm == AlarmState::ON)
      {
         state.alarm = AlarmState::OVERRIDE;
      }
      else
      {
         state.oled = getNextOledState(state.oled);
      }
   }
}

void setupDisplay()
{
   // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
   if (!display.begin(SSD1306_SWITCHCAPVCC, appConfig.screenAddress))
   {
      Serial.println(F("SSD1306 allocation failed"));
      for (;;)
         ; // Don't proceed, loop forever
   }
}

void setup()
{
   Serial.begin(9600);
   pinMode(pinConfig.light, INPUT);
   pinMode(button.getPin(), INPUT_PULLUP);
   button.setEventHandler(onClick);
   setupDisplay();
}

double getVoltageFor(int pin)
{
   int value = analogRead(pin);
   return (value * appConfig.referenceVoltage) / 1023;
}

double getTemperatureCelsiusFor(int pin)
{
   const float voltageAtZeroDeg = 0.5;
   const int tempScaleFactor = 100;
   const auto voltage = getVoltageFor(pin);
   return (voltage - voltageAtZeroDeg) * tempScaleFactor;
}

void updateAlarmState()
{
   const auto distanceCm = state.filters.distance.getMedian();
   // Activate alarm if within treshold
   if (distanceCm < appConfig.alarmDistanceCmTreshold)
   {
      // Unless manually overridden
      if (state.alarm != AlarmState::OVERRIDE)
      {
         state.alarm = AlarmState::ON;
      }
   }
   else
   {
      // And remove the ON/OVERRIDE when no longer within the treshold
      state.alarm = AlarmState::OFF;
   }
}

void updateFilters()
{
   state.filters.light.add(getVoltageFor(pinConfig.light));
   // Must run before distance.add() to provide initial temp value
   state.filters.temp.add(getTemperatureCelsiusFor(pinConfig.temp));
   state.filters.distance.add(distanceSensor.measureDistanceCm(state.filters.temp.getMedian()));
}

void printSensorValue(String label, float value, String unit)
{
   display.print(label);
   display.print(": ");
   display.print(value);
   display.print(" ");
   display.println(unit);
}

void updateDisplay()
{

   const auto photoresistorVoltage = state.filters.light.getMedian();
   const auto distanceCm = state.filters.distance.getMedian();
   const auto tempCelsius = state.filters.temp.getMedian();

   display.clearDisplay();
   display.setTextSize(2);
   display.setTextColor(WHITE);
   display.setCursor(0, 0);

   if (state.alarm == AlarmState::ON)
   {
      display.println("ALARM");
   }
   else
   {
      switch (state.oled)
      {
      case OledState::DISTANCE:
         printSensorValue("Distance", distanceCm, "cm");
         break;
      case OledState::TEMPERATURE:
         printSensorValue("Temp", tempCelsius, "C");
         break;
      case OledState::LIGHT:
         printSensorValue("Light", photoresistorVoltage, "V");
         break;
      }
   }
   display.display();
}

void updateBuzzer()
{
   if (state.alarm == AlarmState::ON)
   {
      tone(pinConfig.buzzer, 1000);
   }
   else
   {
      noTone(pinConfig.buzzer);
   }
}
```

```
void loop()
{
   delay(10);
   button.check();
   updateFilters();
   updateAlarmState();
   updateDisplay();
   updateBuzzer();
}
```

By using the median of the last 5 values, we are able to remove high spikes, that would cause a large error in the distance measurement. We could use a running-average, however such a filter would still have a short-term error. The median filter eliminates this issue all together. It is worth noting however, that larger windows in the median filter could cause the data to lag behind the real values. In a realtime application, this could potentially be unacceptable, and other methods for filtering should be considered.

The code uses the AceButton library to detect button presses. Not only does this lib handle debouncing for us, but it also gives us a event based architecture to handle clicks. This makes for more readable code.

## III. Discussion

Given my previous experience in the subject, my focus was directed away from the circuit and code, and more towards researching best-practices in the IoT industry. After many hours, i found PlatformIO, which has presented the most benefits compared to the alternatives. With a builtin package manager, as well as its design with modern code-practices such as Unit Testing in mind, it stands out as an exceptional tool. Furthermore, i took the time to explore more advanced C++ features such as structs, allowing me to create a more scalable design for global variables. Currently, all configurations such as pins and constants are centralised in the PinConfig and ApplicationConfig. This way, the tendency of arduino projects reaching houndreds of globals can be avoided.

I think the way i spent my time on this project prooved productive, and the fruits of may labour will likely be reflected in future projects.

## Acknowledgment

While Kristian Hansen is not a co-author on this paper, he did create the circuit diagram in Figure 1, while i was working on the code. Figure 2 is a picture of the circuit used in this project. It was created by me and photographed by Kristian.