```
#define ADC_pin A9 //waveform input pin

//*****INITIALIZE ALL CONSTANTS*****//
const uint16_t sample_flag_period = 150; //ms, time betweem cycles
const uint16_t sample_T = 100; //ms, sample period
const uint16_t sample_N = 2048; //samples per period
const uint16_t sample_delta = sample_T * 1000 / sample_N; //us, time between samples
const uint32_t sample_T_actual = sample_delta * sample_N; //us, actual sampling period
const uint8_t speed_bin_N = 10; //number of speed bins (same as number of LEDs)
const uint8_t light_offset = 3; //offset number of pins to first light pin
const uint8_t max_speed = 20; //m/s, max bin speed
const double speed_per_bin = max_speed / speed_bin_N; //(m/s)/index
const double speed_per_frequency = 0.02584; //(m/s)/Hz, calculated at 5.8GHz, technically changes at
different channels
const double indexdiff_to_freq = sample_N * 1000000 / (2 * sample_T_actual); //Hz*index, divide by
index difference to find frequency
const double indexdiff_to_speed = indexdiff_to_freq * speed_per_frequency; //(m/s)*index, divide by
index difference to find speed
const uint8_t rms_percentage = 15; //the required deviation from 0 to record new zero crossing, in
percentage of rms value

//*****INITIALIZE ALL VARIABLES*****//
int16_t adc_buffer[sample_N]; // initialize ADC buffer
uint16_t speed_bin[speed_bin_N]; //initialize doppler speed bins
uint16_t current_rms_value = 0; //rms value of input signal, digital scale
uint16_t buffer_index = 0; //index for use in adc_buffer
uint16_t index_delta = 0; //difference between adc_buffer indices
uint8_t bin_index = 0; //index for use in speed_bin
uint8_t active_light = 0; //the pin number for the light which is currently "on" during normal use
uint8_t sample_flag = 1; //if true, allows start of next sampling period
uint8_t calc_flag = 0; //if true, starts calculation loop
uint32_t adc_timer = 0; //us, adc_timer
double freq_sum = 0; //Hz, sum of recorded frequencies, then divided to find average
uint32_t sample_flag_timer = 0; //ms, timer for resetting sample
uint32_t actual_time_0 = 0; //ms, sample-start time
uint32_t actual_time_1 = 0; //ms, sample-end time, for purpose of result verification

void setup() {

  //initialize Serial port
  Serial.begin(9600);
  while (!Serial.available());

  //print sensitive values to ensure inputs are valid
  Serial.println(sample_delta);
  Serial.println(indexdiff_to_freq, 20);
  Serial.println(indexdiff_to_speed, 20);

  //Set pin directions
  pinMode(ADC_pin, INPUT);
  for (active_light = 3; active_light <= 12; active_light++)
  {
    pinMode(active_light, OUTPUT);
    digitalWrite(active_light, HIGH);
  }

  sample_flag_timer = millis(); //preset sample flag timer
}

void loop() {
  if (sample_flag) //if not done sampling
  {
    if (adc_timer <= micros()) //*****************FIX FOR OVERFLOW
    {
      if (buffer_index == 0)
      {
        actual_time_0 = millis();
      }
      adc_timer = adc_timer + sample_delta; //every sample_delta microseconds...
      adc_buffer[buffer_index++] = analogRead(ADC_pin); //record adc values
    }
    if (buffer_index == sample_N)
    {
      actual_time_1 = millis() - actual_time_0;
      Serial.print("Time Elapsed (ms): ");
      Serial.println(actual_time_1);
```

```
      buffer_index = 0; //reset index
      sample_flag = 0; //stop sampling
      calc_flag = 1; //start calculations
    }
  }

  if (!sample_flag && calc_flag && sample_flag_timer < millis()) //if not taking samples and ready to
  calculate
  {
    sample_flag_timer = sample_flag_timer + sample_flag_period; //reset sample timer
    current_rms_value = zero_average_and_rms(adc_buffer); //make samples bipolar, return rms value
    Serial.print("RMS value = ");
    Serial.println(current_rms_value); //print bipolar ADC RMS value
    stitch_sandwich_stacker(adc_buffer); //stack frequency data into bins
    sample_flag = 1; //start sampling
    calc_flag = 0; //stop calcs
  }

  if(adc_timer - sample_delta > micros()) adc_timer = micros() + sample_delta; //timer overflow reset
  -- note that overflow will cause 1-2 false readings
}

uint16_t zero_average_and_rms(int16_t sample[]) //offsets input array to zero-average, calculates and
returns ADC RMS value
{
  uint16_t rms_value;
  int32_t sum = 0;
  int32_t squaresum = 0;
  int16_t diff = 0;
  for (uint16_t j = 0; j < sample_N; j++)
  {
    sum = sum + sample[j];
  }
  diff = sum / sample_N;
  for (int j = 0; j < sample_N; j++)
  {
    adc_buffer[j] = sample[j] - diff;
  }
  for (int j = 0; j < sample_N; j++)
  {
    squaresum = squaresum + (adc_buffer[j] * adc_buffer[j]);
  }

  rms_value = sqrt(squaresum / sample_N); //calculate rms value
  return rms_value;
}

//takes bipolar sample vector
//steps through samples and finds zero crossing indices
//calls bin_increment function to find frequency content
//calculates and prints average frequency and speed
//updates LED indicator
void stitch_sandwich_stacker(int16_t sample[])
{
  uint8_t sign = 0;
  uint8_t old_sign = 0;
  uint8_t old_light = active_light;
  uint16_t new_zero_index = 0;
  uint16_t old_zero_index = 0;
  uint16_t crossing_count = 0;
  double freq_float;
  double speed_float;
  for (bin_index = 0; bin_index < speed_bin_N; bin_index++) //reset bins
  {
    speed_bin[bin_index] = 0;
  }
  for (buffer_index = 0; buffer_index < sample_N; buffer_index++)
  {
    //Serial.println(adc_buffer[buffer_index]); //***TEST CODE***
    //step through samples
    //check if absolute value greater than or equal to XX% rms value
    if ((abs(sample[buffer_index]) * 100) >= (current_rms_value * rms_percentage))
    {
      if (sample[buffer_index] > 0)
      {
        sign = 1;
```

```
      } else {
        sign = 0;
      }
      if (sign != old_sign)
      {
        old_sign = sign;
        //new_zero_index = (buffer_index + old_zero_index) / 2;
        new_zero_index = buffer_index;
        index_delta = new_zero_index - old_zero_index;
        if (old_zero_index != 0)
        {
          crossing_count++;
          bin_increment(index_delta);
        }
        old_zero_index = new_zero_index; //update
      }
    }
    //if so, check if positive/negative
    //record sign, check if different than last sign
    //if so, record a zero crossing index halfway between indices
    //record difference between current and previous crossing index
  }
  freq_float = freq_sum / crossing_count; //find average frequency
  freq_sum = 0; //reset sum
  speed_float = freq_float * speed_per_frequency; //find average speed
  Serial.print("Frequency (Hz): ");
  Serial.println(freq_float, 1); //print average frequency
  Serial.print("Speed (m/s): ");
  Serial.println(speed_float, 1); //print average speed
  active_light = bin_peak_search(speed_bin) + light_offset;
  if (old_light != active_light && active_light >= 3 && active_light <= 12)
  {
    digitalWrite((old_light), HIGH);
    digitalWrite((active_light), LOW);
  }
  buffer_index = 0; //reset adc_buffer index
  adc_timer = micros(); //update adc_timer
}

//takes zero-crossing index delta
//using defined constants, calculates frequency for the given delta
//increments value inside of speed_bin vector
void bin_increment(uint16_t index_difference)
{
  uint8_t bin;
  double freq_float;
  double speed_float;
  freq_float = indexdiff_to_freq / index_difference;// N/T
  freq_sum = freq_sum + freq_float;
  speed_float = indexdiff_to_speed / index_difference; //calculate single-sample speed
  //  Serial.print("speed (m/s) = ");
  //  Serial.println(speed_float);
  //    Serial.print("index difference = ");
  //    Serial.println(index_difference);
  bin = speed_float / speed_per_bin + 0.5; //record rounded bin
  if (bin <= speed_bin_N) //don't write outside of available bins
  {
    speed_bin[bin]++;
  }
}

uint16_t bin_peak_search(uint16_t vector[]) //take vector with unsigned 16-bit values and return the
position of the max value
{
  uint16_t search_index;
  uint16_t peak_index = 0;
  uint16_t current_max = 0;
  for (search_index = 0; search_index < speed_bin_N; search_index++) //for the length of the input vector
  {
    if (vector[search_index] > current_max) //check if value is larger than recorded max
    {
      current_max = vector[search_index]; //update max
      peak_index = search_index; //update index
    }
  }
  return peak_index;
```

```
}
```