```c
/* Contiki Header Files */
#include "contiki.h"
#include "dev/light-sensor.h"
#include "dev/sht11-sensor.h"
/* C Standard Header Files */
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
/* Additional C Header Files */
#include <string.h>

/* Circular Buffer Definitions */
#define BUFFER_SIZE 12
#define LOW_LEVEL_ACTIVITY 10
#define MID_LEVEL_ACTIVITY 1000
#define HIGH_LEVEL_ACTIVITY 10000

/* USER OPTIONS */
bool RUN_ADVANCED_FEAT = true;

/* Type Definition for All Circular buffers */
typedef struct {
  float * values; // array as a pointer to be able to put on heap
  int head, tail, num_entries, size;
}
circular_buffer;

/* Measurement Sensor Queues */
static circular_buffer light_q, temp_q;

/*
 * Functions to support the display of Floating-Point numbers
 * Note: this does not support floats with values before the decimal
 * between -32,768 to 32,767 i.e sum of squares calculated in standard deviation
 *
 */

int d1(float f) {
  return ((int) f);
}

unsigned int d2(float f) {
  if (f > 0)
    return (1000 * (f - d1(f)));
  else
    return (1000 * (d1(f) - f));
}

/* Circular Buffer Functions */

void init_circular_buffer(circular_buffer * q, int buffer_size) {
  q -> size = buffer_size, q -> head = 0, q -> tail = 0, q -> num_entries = 0;
  q -> values = malloc(sizeof(float) * q -> size); // dynamic memory allocation of array
}

bool is_circular_buffer_full(circular_buffer * q) {
  return (q -> num_entries == q -> size);
}

bool is_circular_buffer_empty(circular_buffer * q) {
  return (q -> num_entries == 0);
}

bool push_circular_buffer_val(circular_buffer * q, float value) {
  if (is_circular_buffer_full(q)) {
    return false; // no space to push value in
  }

  q -> values[q -> tail] = value;
  q -> tail = (q -> tail + 1) % q -> size; // returns vals 0-11; resets to 0 when tail is 11 (final element)
  q -> num_entries++;
  return true;
}

bool pop_circular_buffer_val(circular_buffer * q) {
  if (is_circular_buffer_empty(q)) {
    return false; // no values left to remove
  }

  q -> head = (q -> head + 1) % q -> size; // returns vals 0-11; resets to 0 when head is 11 (final element)
  q -> num_entries--;
  return true;
}

float * get_circular_buffer_vals(circular_buffer * q) {
  return (q -> values);
}
```

```
88
89  void destroy_circular_buffer(circular_buffer * q) {
90    free(q -> values);
91  }
92
93  void print_circular_buffer(circular_buffer * q, char * value_id) {
94    int i;
95    char begin[] = "= [ ", end[] = "]";
96
97    printf("%s %s", value_id, begin);
98
99    for (i = 0; i < q -> num_entries; i++) {
100     printf("%d.%03u ", d1(q -> values[i]), d2(q -> values[i]));
101   }
102   printf("%s\n", end);
103
104 }
105
106 /* Additional Math Calculation Functions */
107
108 /*
109  * Supported by
110  * https://ourcodeworld.com/articles/read/884/how-to-get-the-square-root-of-a-number-without-using-the-sqrt-function-in-c
111  */
112 float calculate_sqrt(float val) {
113   float temp = 0, sqrt;
114   sqrt = val / 2;
115
116   while (sqrt != temp) {
117     temp = sqrt;
118     sqrt = (val / temp + temp) / 2;
119   }
120   return sqrt;
121
122 }
123
124 float calculate_mean(circular_buffer * q) {
125   float sum_buffer = 0.0;
126   int i;
127   for (i = 0; i < q -> size; i++) {
128     sum_buffer = sum_buffer + q -> values[i];
129   }
130   return (sum_buffer / q -> size);
131 }
132
133 float calculate_square(float val) {
134   return (val * val);
135 }
136
137 float calculate_stdDev(circular_buffer * q) {
138   float mean_buffer = calculate_mean(q), var_buffer_sum = 0.0, temp, diff;
139   int j;
140
141   for (j = 0; j < q -> size; j++) {
142     diff = q -> values[j] - mean_buffer;
143     temp = calculate_square(diff);
144     var_buffer_sum += temp;
145   }
146   return calculate_sqrt(var_buffer_sum / q -> size);
147 }
148
149 /* Aggegation of light measurements in buffer
150  * low level activity - 12 into 1
151  * mid level activity - 12 into 3
152  * high level activity - no aggregation
153  */
154 void aggregate_data(circular_buffer * q, char * agg_level) {
155   printf("Aggregation is %s \n", agg_level);
156
157   if (strcmp(agg_level, "12-to-1") == 0) {
158     int i;
159     float sum, avg;
160
161     // replace measurement values with aggregated value
162     for (i = 0; i < q -> size; i++) {
163       sum = sum + q -> values[i];
164       pop_circular_buffer_val(q);
165     }
166     avg = sum / q -> size;
167     q -> tail = 0;
168     push_circular_buffer_val(q, avg);
169   }
170   if (strcmp(agg_level, "4-to-1") == 0) {
171     float avg_1 = 0, avg_2 = 0, avg_3 = 0, sum_1 = 0, sum_2 = 0, sum_3 = 0;
172     int i, j, k, l;
173
174     for (i = 0; i < 4; i++) {
175       sum_1 = sum_1 + q -> values[i];
```

```
    }
    for (j = 4; j < 8; j++) {
      sum_2 = sum_2 + q -> values[j];
    }
    for (k = 8; k < 12; k++) {
      sum_3 = sum_3 + q -> values[k];
    }

    avg_1 = sum_1 / 4;
    avg_2 = sum_2 / 4;
    avg_3 = sum_3 / 4;

    // replace measurement values with aggregated value
    for (l = 0; l < q -> size; l++) {
      pop_circular_buffer_val(q);
    }

    q -> tail = 0;
    push_circular_buffer_val(q, avg_1);
    push_circular_buffer_val(q, avg_2);
    push_circular_buffer_val(q, avg_3);
  }

}

/* Sensor Readings from Cooja Simulator */

float getLight(void) {
  float V_sensor = 1.5 * light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC) / 4096;
  float I = V_sensor / 100000; // xm1000 uses 100kohm resistor
  float light_lx = 0.625 * 1e6 * I * 1000; // convert from current to light intensity
  return light_lx;
}

float getTemperature(void) {
  int tempADC = sht11_sensor.value(SHT11_SENSOR_TEMP_SKYSIM);
  float temp = 0.04 * tempADC - 39.6; // skymote uses 12-bit ADC, or 0.04 resolution
  return tempADC;
}

/*
 * Continuously reads in measurements into buffer
 * When 12 new readings are collected, calculate standard deviation
 * and aggregate appropriately
 */
int store_light_measurements(int measurement_count_light) {

  float stdDev, light_lx = getLight();

  if (measurement_count_light < 12) {
    printf("Light Readings: %d.%03u lux \n", d1(light_lx), d2(light_lx));
    bool success = push_circular_buffer_val( & light_q, light_lx);
    if (!success) {
      pop_circular_buffer_val( & light_q);
      push_circular_buffer_val( & light_q, light_lx);
    }
    measurement_count_light++;
  } else {
    print_circular_buffer( & light_q, "Light Measurement");
    stdDev = calculate_stdDev( & light_q);
    printf("StdDev = %d.%03u \n", d1(stdDev), d2(stdDev));
    if (stdDev < LOW_LEVEL_ACTIVITY) {
      aggregate_data( & light_q, "12-to-1");
    }
    if (stdDev <= MID_LEVEL_ACTIVITY && stdDev >= LOW_LEVEL_ACTIVITY) {
      aggregate_data( & light_q, "4-to-1");
    }
    if (stdDev > MID_LEVEL_ACTIVITY) {
      aggregate_data( & light_q, "12-to-12");
    }

    print_circular_buffer( & light_q, "Light Aggregation");
    printf("\n\n");
    measurement_count_light = 0;
  }

  return measurement_count_light;

}

/*
 * Continuously reads in measurements into buffer
 * When 12 new readings are collected, print value of buffer
 *
 */
int store_temperature_measurements(int measurement_count_temp) {
  float temp_c = getTemperature();
  printf("Temperature Readings: %d.%03u C \n", d1(temp_c), d2(temp_c));
```

```
264   if (measurement_count_temp < 12) {
265     bool success = push_circular_buffer_val( & temp_q, temp_c);
266     if (!success) {
267       pop_circular_buffer_val( & temp_q);
268       push_circular_buffer_val( & temp_q, temp_c);
269     }
270     measurement_count_temp++;
271
272   } else {
273     print_circular_buffer( & temp_q, "Temperature Measurement");
274     measurement_count_temp = 0;
275   }
276
277   return measurement_count_temp;
278
279 }
280
281 /*---------------------------------------------------------------------------*/
282 PROCESS(sensor_reading_process, "Sensor Measurement Reading and Aggregation");
283 AUTOSTART_PROCESSES( & sensor_reading_process);
284 /*---------------------------------------------------------------------------*/
285
286 PROCESS_THREAD(sensor_reading_process, ev, data) {
287   static int measurement_count_light = 0, measurement_count_temp = 0, count = 0;
288   static struct etimer timer;
289   static float * qlight_vals, * qtemp_vals;
290
291   // Only initialise buffers once during the process
292   if (count == 0) {
293     init_circular_buffer( & light_q, BUFFER_SIZE);
294     init_circular_buffer( & temp_q, BUFFER_SIZE);
295   }
296   count++;
297
298   PROCESS_BEGIN();
299   etimer_set( & timer, CLOCK_SECOND * 0.5); // Set timer to half a second
300
301   SENSORS_ACTIVATE(light_sensor);
302   SENSORS_ACTIVATE(sht11_sensor);
303
304   while (1) {
305     PROCESS_WAIT_EVENT_UNTIL(ev = PROCESS_EVENT_TIMER);
306     measurement_count_light = store_light_measurements(measurement_count_light);
307
308     if (RUN_ADVANCED_FEAT) {
309       measurement_count_temp = store_temperature_measurements(measurement_count_temp);
310     }
311
312     /*
313      * Given both the temperature and light buffers have 12 readings
314      * calculate the euclidean distance, correlation coefficient and linear regression analysis
315      */
316
317     if (measurement_count_light == 12 && measurement_count_temp == 12) {
318
319       /* Euclidean Distance */
320
321       qlight_vals = get_circular_buffer_vals( & light_q);
322       qtemp_vals = get_circular_buffer_vals( & temp_q);
323
324       int i = 0;
325       float diff = 0, sum = 0, euclidean_distance = 0;
326
327       for (i = 0; i < 12; i++) {
328         diff = qlight_vals[i] - qtemp_vals[i];
329         sum += calculate_square(diff); // calculate the distance between the two points sqd
330       }
331
332       euclidean_distance = calculate_sqrt(sum);
333       printf("\n\n");
334       printf("Euclidean distance %d.%03u \n", d1(euclidean_distance), d2(euclidean_distance));
335
336       /* Correlation Coefficient */
337
338       float covariance, mean_lightq, mean_tempq, sum_of_variances, stdDev_light, stdDev_temp, correlation, sum_light_var_sq, sum_temp_var_sq, stdDev_mult;
339
340       mean_lightq = calculate_mean( & light_q);
341       mean_tempq = calculate_mean( & temp_q);
342
343       for (i = 0; i < 12; i++) {
344         sum_of_variances += (qlight_vals[i] - mean_lightq) * (qtemp_vals[i] - mean_tempq); // numerator for covariance
345
346         // calculated for linear regression - light = x variable, temp = y variable
347         sum_light_var_sq += calculate_square(qlight_vals[i] - mean_lightq);
348         sum_temp_var_sq += calculate_square(qtemp_vals[i] - mean_tempq);
349       }
350
351       covariance = sum_of_variances / 11; // divided by length of vector - 1
```

```
    covariance = sum_of_variances / 11; // divided by length of vector - 1

    stdDev_light = calculate_stdDev( & light_q);
    stdDev_temp = calculate_stdDev( & temp_q);
    stdDev_mult = (stdDev_light * stdDev_temp);

    correlation = covariance / stdDev_mult;

    if (stdDev_mult > 0) {
      // Provide support to show negative and positive correlation
      if ((covariance > 0.0 && stdDev_mult < 0.0) || (covariance < 0.0 && stdDev_mult > 0.0)) {
        printf("Correlation Coefficient: - %d.%03u \n", d1(correlation), d2(correlation));
      } else {
        printf("Correlation Coefficient: %d.%03u \n", d1(correlation), d2(correlation));
      }
    } else {
      printf("Correlation Coefficient: 0.000 \n");
    }
    /* Linear Regression Analysis */

    float gradient, intercept;
    gradient = sum_of_variances / sum_light_var_sq; // regression line must go through the mean points of x and y
    intercept = mean_tempq - (gradient * mean_lightq); // using y = mx + c; therefore c = y - mx

    printf(" y(predicted) =  %d.%03ux +  %d.%03u \n", gradient, intercept);

    float y_pred, sum_y_pred_var, r_squared;
    for (i = 0; i < 12; i++) {
      y_pred = (gradient * qlight_vals[i]) + intercept; // predict y (temperature) using above regression line
      sum_y_pred_var += calculate_square(y_pred - mean_tempq); // find the difference between predicted and y avg
    }

    r_squared = sum_y_pred_var / sum_temp_var_sq; // measures the difference between predicted and real values

    if (sum_temp_var_sq > 0) {
      // Provide support to show negative and positive correlation (as for Correlation)
      if ((sum_y_pred_var > 0.0 && sum_temp_var_sq < 0.0) || (sum_y_pred_var < 0.0 && sum_temp_var_sq > 0.0)) {
        printf("R Squared: - %d.%03u \n", d1(r_squared), d2(r_squared));
      } else {
        printf("R Squared: %d.%03u \n", d1(r_squared), d2(r_squared));
      }
    } else {
      printf("R Squared: 0.000 \n");
    }
    printf("\n\n");
  }

  etimer_reset( & timer);
 }

 destroy_circular_buffer( & light_q);
 destroy_circular_buffer( & temp_q);

 PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```