```c
/*----------Basic Features----------*/
#include "contiki.h"
#include "dev/light-sensor.h"
#include "lib/list.h"
#include "lib/memb.h"
#include <stdio.h> /* For printf() */

#define BUFFER_SIZE 12
#define LOW_ACTIVITY_THRESHOLD 1000
#define HIGH_ACTIVITY_THRESHOLD 2000
#define SAX_FRAGMENTS 4

/* Helper functions */
static void print_float(float number)
{
    int integer_part = (int)number;
    int decimal_part = (int)((number - integer_part) * 1000);
    printf("%d.%02d", integer_part, decimal_part);
}

float read_light_sensor(void)
{
    int lightData = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
    float V_sensor = 1.5 * lightData / 4096;
    float I = V_sensor / 100000;
    float light = 0.625 * 1e6 * I * 1000;
    return light;
}

/* Use linked list for sensor data */
struct sensor_data
{
    struct sensor_data *next;
    float light;
};

LIST(sensor_list);
MEMB(sensor_mem, struct sensor_data, BUFFER_SIZE);

static void add_sensor_data(float light)
{
    struct sensor_data *new_data;

    if (list_length(sensor_list) >= BUFFER_SIZE)
    {
        struct sensor_data *oldest = list_pop(sensor_list);
        memb_free(&sensor_mem, oldest);
    }

    new_data = memb_alloc(&sensor_mem);
    if (new_data == NULL)
    {
        printf("Memory allocation failed!\n");
        return;
    }

    new_data->light = light;
    list_add(sensor_list, new_data);
}

/* Calculate average */
static float calculate_avg()
{
    struct sensor_data *item;
    float sum = 0.0;
```

```
66      int count = 0;
67      for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
68      {
69          sum += item->light;
70          count++;
71      }
72      return (count == 0) ? 0.0 : sum / count;
73  }
74
75  static float calculate_ssd(float avg)
76  {
77      struct sensor_data *item;
78      float ssd = 0.0;
79      for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
80      {
81          float diff = item->light - avg;
82          ssd += diff * diff;
83      }
84      return ssd;
85  }
86
87  static float sqrt_approx(float ssd)
88  {
89      float error = 0.001; // Error tolerance for Babylonian method
90      float x = ssd;       // Initial guess for square root
91      float difference;
92      int i;
93
94      if (ssd == 0)
95      {
96          return 0.0; // No variance
97      }
98
99      for (i = 0; i < 50; i++)
100     { // Babylonian method
101         x = 0.5 * (x + ssd / x);
102         difference = x * x - ssd;
103         if (difference < 0)
104         {
105             difference = -difference;
106         }
107         if (difference < error)
108         {
109             break;
110         }
111     }
112     return x;
113 }
114
115 static float calculate_std()
116 {
117     float avg = calculate_avg();
118     float ssd = calculate_ssd(avg);
119     return sqrt_approx(ssd);
120 }
121
122 void perform_sax(char sax_output[SAX_FRAGMENTS])
123 {
124     struct sensor_data *item;
125     float fragment_means[SAX_FRAGMENTS] = {0};
126     int fragment_size = BUFFER_SIZE / SAX_FRAGMENTS;
127     int i = 0, count = 0;
128
129     // Compute fragment means
130     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
131     {
```

```
132         fragment_means[i] += item->light;
133         count++;
134         if (count == fragment_size)
135         {
136             fragment_means[i] /= fragment_size;
137             i++;
138             count = 0;
139         }
140     }
141 
142     // Normalize fragment means
143     float avg = calculate_avg();
144     float std = calculate_std();
145     char alphabet[4] = {'A', 'B', 'C', 'D'};
146     float breakpoints[3] = {-0.67, 0, 0.67};
147 
148     // Assign SAX symbols
149     for (i = 0; i < SAX_FRAGMENTS; i++)
150     {
151         float z = (fragment_means[i] - avg) / std;
152         if (z <= breakpoints[0])
153         {
154             sax_output[i] = alphabet[0];
155         }
156         else if (z <= breakpoints[1])
157         {
158             sax_output[i] = alphabet[1];
159         }
160         else if (z <= breakpoints[2])
161         {
162             sax_output[i] = alphabet[2];
163         }
164         else
165         {
166             sax_output[i] = alphabet[3];
167         }
168     }
169 }
170 
171 static void aggregate_and_report()
172 {
173     struct sensor_data *item = list_head(sensor_list);
174     float std = calculate_std();
175     float avg = calculate_avg();
176     char sax_output[SAX_FRAGMENTS];
177 
178     // Print the original buffer
179     printf("B = [");
180     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
181     {
182         print_float(item->light);
183         if (list_item_next(item) != NULL)
184         {
185             printf(", ");
186         }
187     }
188     printf("]\n");
189 
190     // Print the standard deviation
191     printf("StdDev = ");
192     print_float(std);
193     printf("\n");
194 
195     // Determine the activity level and aggregation
196     if (std < LOW_ACTIVITY_THRESHOLD)
197     {
```

```
198         printf("Aggregation = 12-into-1\n");
199         printf("X = [");
200         print_float(avg);
201         printf("]\n");
202     }
203     else if (std < HIGH_ACTIVITY_THRESHOLD)
204     {
205         printf("Aggregation = 4-into-1\n");
206         printf("X = [");
207         int count = 0;
208         float sum = 0.0;
209         item = list_head(sensor_list);
210         while (item != NULL)
211         {
212             sum += item->light;
213             count++;
214             if (count == 4)
215             {
216                 print_float(sum / 4);
217                 sum = 0.0;
218                 count = 0;
219                 if (list_item_next(item) != NULL)
220                 {
221                     printf(", ");
222                 }
223             }
224             item = list_item_next(item);
225         }
226 
227         printf("]\n");
228     }
229     else
230     {
231         printf("Aggregation = 1-into-1\n");
232         printf("X = [");
233         for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
234         {
235             print_float(item->light);
236             if (list_item_next(item) != NULL)
237             {
238                 printf(", ");
239             }
240         }
241         printf("]\n");
242     }
243 
244     // Perform SAX transformation and print
245     perform_sax(sax_output);
246     printf("SAX = [");
247     int i;
248     for (i = 0; i < SAX_FRAGMENTS; i++)
249     {
250         printf("%c", sax_output[i]);
251         if (i < SAX_FRAGMENTS - 1)
252         {
253             printf(", ");
254         }
255     }
256     printf("]\n");
257 }
258 
259 /*---------------------------------------------------------------------------*/
260 PROCESS(sensor_reading_process, "Sensor reading process");
261 AUTOSTART_PROCESSES(&sensor_reading_process);
262 /*---------------------------------------------------------------------------*/
263 PROCESS_THREAD(sensor_reading_process, ev, data)
```

```
{
    static struct etimer timer;
    static int sample_counter = 0;
    static int k = 12; // number of samlpes before aggregation

    PROCESS_BEGIN();
    etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second

    SENSORS_ACTIVATE(light_sensor);

    while (1)
    {
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);

        float light = read_light_sensor();
        add_sensor_data(light);
        sample_counter++;

        if (sample_counter >= k)
        {
            aggregate_and_report();
            sample_counter = 0;
        }

        etimer_reset(&timer);
    }

    PROCESS_END();
}
```