


 frankcholula /
iot



[Code](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

assignment1 [iot / surrey / assignment1 / sensor.c](#)

Go to file

t



frankcholula remove unnecessary comments.

445c84a · 35 minutes ago



Executable File · 291 lines (258 loc) · 6.75 KB

Code Blame



Raw



```
1  #include "contiki.h"
2  #include "dev/light-sensor.h"
3  #include "lib/list.h"
4  #include "lib/memb.h"
5  #include <stdio.h> /* For printf() */
6
7  #define BUFFER_SIZE 12
8  #define LOW_ACTIVITY_THRESHOLD 1000
9  #define HIGH_ACTIVITY_THRESHOLD 5000
10 #define SAX_FRAGMENTS 4
11
12 /* Helper functions */
13 static void print_float(float number)
14 {
15     int integer_part = (int)number;
16     int decimal_part = (int)((number - integer_part) * 1000);
17     printf("%d.%02d", integer_part, decimal_part);
18 }
19
20 float read_light_sensor(void)
21 {
22     int lightData = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
23     float V_sensor = 1.5 * lightData / 4096;
24     float I = V_sensor / 100000;
25     float light = 0.625 * 1e6 * I * 1000;
26     return light;
27 }
28
29 /* Use linked list for sensor data */
30 struct sensor_data
31 {
32     struct sensor_data *next;
33     float light;
34 };
35
36 LIST(sensor_list);
37 MEMB(sensor_mem, struct sensor_data, BUFFER_SIZE);
38
39 static void add_sensor_data(float light)
40 {
41     struct sensor_data *new_data;
42
43     if (list_length(sensor_list) >= BUFFER_SIZE)
44     {
45         struct sensor_data *oldest = list_pop(sensor_list);
46         memb_free(&sensor_mem, oldest);
47     }
48
49     new_data = memb_alloc(&sensor_mem);
50     if (new_data == NULL)
51     {
52         printf("Memory allocation failed!\n");
53         return;
54     }
```

```
55
56     new_data->light = light;
57     list_add(sensor_list, new_data);
58 }
59
60 /* Calculate average */
61 static float calculate_avg()
62 {
63     struct sensor_data *item;
64     float sum = 0.0;
65     int count = 0;
66     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
67     {
68         sum += item->light;
69         count++;
70     }
71     return (count == 0) ? 0.0 : sum / count;
72 }
73
74 static float calculate_ssd(float avg)
75 {
76     struct sensor_data *item;
77     float ssd = 0.0;
78     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
79     {
80         float diff = item->light - avg;
81         ssd += diff * diff;
82     }
83     return ssd;
84 }
85
86 static float sqrt_approx(float ssd)
87 {
88     float error = 0.001; // Error tolerance for Babylonian method
89     float x = ssd;        // Initial guess for square root
90     float difference;
91     int i;
92
93     if (ssd == 0)
94     {
95         return 0.0; // No variance
96     }
97
98     for (i = 0; i < 50; i++)
99     { // Babylonian method
100         x = 0.5 * (x + ssd / x);
101         difference = x * x - ssd;
102         if (difference < 0)
103         {
104             difference = -difference;
105         }
106         if (difference < error)
107         {
108             break;
109         }
110     }
111     return x;
112 }
113
114 static float calculate_std()
115 {
116     float avg = calculate_avg();
117     float ssd = calculate_ssd(avg);
118     return sqrt_approx(ssd);
119 }
120
121 void perform_sax(char sax_output[SAX_FRAGMENTS])
122 {
123     struct sensor_data *item;
124     float fragment_means[SAX_FRAGMENTS] = {0};
```

```

121     struct fragment_means fragment_means[SAX_FRAGMENTS] = {0};
122
123     int fragment_size = BUFFER_SIZE / SAX_FRAGMENTS;
124     int i = 0, count = 0;
125
126
127
128     // Compute fragment means
129     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
130     {
131         fragment_means[i] += item->light;
132         count++;
133         if (count == fragment_size)
134         {
135             fragment_means[i] /= fragment_size;
136             i++;
137             count = 0;
138         }
139     }
140
141     // Normalize fragment means
142     float avg = calculate_avg();
143     float std = calculate_std();
144     char alphabet[4] = {'A', 'B', 'C', 'D'};
145     float breakpoints[3] = {-0.67, 0, 0.67};
146
147     // Assign SAX symbols
148     for (i = 0; i < SAX_FRAGMENTS; i++)
149     {
150         float z = (fragment_means[i] - avg) / std;
151         if (z <= breakpoints[0])
152         {
153             sax_output[i] = alphabet[0];
154         }
155         else if (z <= breakpoints[1])
156         {
157             sax_output[i] = alphabet[1];
158         }
159         else if (z <= breakpoints[2])
160         {
161             sax_output[i] = alphabet[2];
162         }
163         else
164         {
165             sax_output[i] = alphabet[3];
166         }
167     }
168 }
169
170 static void aggregate_and_report()
171 {
172     struct sensor_data *item = list_head(sensor_list);
173     float std = calculate_std();
174     float avg = calculate_avg();
175     char sax_output[SAX_FRAGMENTS];
176
177     // Print the original buffer
178     printf("B = [");
179     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
180     {
181         print_float(item->light);
182         if (list_item_next(item) != NULL)
183         {
184             printf(", ");
185         }
186     }
187     printf("]\n");
188
189     // Print the standard deviation
190     printf("StdDev = ");
191     print_float(std);
192     printf("\n");
193
194     // Determine the activity level and aggregation

```

```

194 // determine the activity level and aggregation
195 if (std < LOW_ACTIVITY_THRESHOLD)
196 {
197     printf("Aggregation = 12-into-1\n");
198     printf("X = [");
199     print_float(avg);
200     printf("]\n");
201 }
202 else if (std < HIGH_ACTIVITY_THRESHOLD)
203 {
204     printf("Aggregation = 4-into-1\n");
205     printf("X = [");
206     int count = 0;
207     float sum = 0.0;
208     item = list_head(sensor_list);
209     while (item != NULL)
210     {
211         sum += item->light;
212         count++;
213         if (count == 4)
214         {
215             print_float(sum / 4);
216             sum = 0.0;
217             count = 0;
218             if (list_item_next(item) != NULL)
219             {
220                 printf(", ");
221             }
222         }
223         item = list_item_next(item);
224     }
225
226     printf("]\n");
227 }
228 else
229 {
230     printf("Aggregation = 1-into-1\n");
231     printf("X = [");
232     for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
233     {
234         print_float(item->light);
235         if (list_item_next(item) != NULL)
236         {
237             printf(", ");
238         }
239     }
240     printf("]\n");
241 }
242
243 // Perform SAX transformation and print
244 perform_sax(sax_output);
245 printf("SAX = [");
246 int i;
247 for (i = 0; i < SAX_FRAGMENTS; i++)
248 {
249     printf("%c", sax_output[i]);
250     if (i < SAX_FRAGMENTS - 1)
251     {
252         printf(", ");
253     }
254 }
255 printf("]\n");
256 }
257
258 /*-----*/
259 PROCESS(sensor_reading_process, "Sensor reading process");
260 AUTOSTART_PROCESSES(&sensor_reading_process);
261 /*-----*/
262 PROCESS_THREAD(sensor_reading_process, ev, data)
263 {
264     // ...

```

```
264     static struct etimer timer;
265     static int sample_counter = 0;
266     static int k = 12; // number of samples before aggregation
267
268     PROCESS_BEGIN();
269     etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second
270
271     SENSORS_ACTIVATE(light_sensor);
272
273     while (1)
274     {
275         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
276
277         float light = read_light_sensor();
278         add_sensor_data(light);
279         sample_counter++;
280
281         if (sample_counter >= k)
282         {
283             aggregate_and_report();
284             sample_counter = 0;
285         }
286
287         etimer_reset(&timer);
288     }
289
290     PROCESS_END();
291 }
```