


 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     float 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 }
```

 frankcholula /
iot



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

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

Go to file

t



frankcholula remove unnecessary comments.

0aa22f5 · 33 minutes ago



Executable File · 529 lines (463 loc) · 13.7 KB

Edit this file

Code

Blame



Raw



```
1  #include "contiki.h"
2  #include "dev/light-sensor.h"
3  #include "dev/sht11-sensor.h"
4  #include "lib/list.h"
5  #include "lib/memb.h"
6  #include <stdio.h> /* For printf() */
7  #include <math.h>
8
9  #define LN2 0.69314718056
10 #define BUFFER_SIZE 12
11 #define PI 3.141592653589793
12
13 /* Helper functions */
14 static void print_float(float number)
15 {
16     int integer_part = (int)number;
17     int decimal_part = (int)((number - integer_part) * 1000);
18     printf("%d.%02d", integer_part, decimal_part);
19 }
20
21 float read_light_sensor(void)
22 {
23     int lightData = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
24     float V_sensor = 1.5 * lightData / 4096;
25     float I = V_sensor / 100000;
26     float light = 0.625 * 1e6 * I * 1000;
27     return light;
28 }
29
30 float get_temp_sensor(void)
31 {
32     float tempData = sht11_sensor.value(SHT11_SENSOR_TEMP_SKYSIM); // For Cooja Sim
33     float d1 = -39.6;
34     float d2 = 0.04; // Adjust as per sensor calibration
35     float temp = tempData * d2 + d1;
36     return temp;
37 }
38
39 /* Use linked list for sensor data */
40 struct sensor_data
41 {
42     struct sensor_data *next;
43     float value;
44 };
45
46 LIST(light_list);
47 LIST(temp_list);
48 MEMB(light_mem, struct sensor_data, BUFFER_SIZE);
49 MEMB(temp_mem, struct sensor_data, BUFFER_SIZE);
50
51 static void add_sensor_data(float value, list_t lst, struct memb *mem)
52 {
53     struct sensor_data *new_data;
```

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

```

125 static float calculate_std(list_t lst)
126 {
127     float avg = calculate_avg(lst);
128     float ssd = calculate_ssd(avg, lst);
129     return sqrt_approx(ssd);
130 }
131
132 static float calculate_manhattan_dist(list_t light_list, list_t temp_list)
133 {
134     struct sensor_data *light_item = list_head(light_list);
135     struct sensor_data *temp_item = list_head(temp_list);
136     double dist = 0.0; // Use double for accumulation
137
138     while (light_item != NULL && temp_item != NULL)
139     {
140         double diff = (double)light_item->value - (double)temp_item->value;
141         dist += (diff < 0) ? -diff : diff; // Inline abs for double
142         light_item = list_item_next(light_item);
143         temp_item = list_item_next(temp_item);
144     }
145     return (float)dist;
146 }
147
148 static float calculate_correlation(list_t light_list, list_t temp_list)
149 {
150     struct sensor_data *light_item = list_head(light_list);
151     struct sensor_data *temp_item = list_head(temp_list);
152
153     float avg_x = calculate_avg(light_list);
154     float avg_y = calculate_avg(temp_list);
155     float std_x = calculate_std(light_list);
156     float std_y = calculate_std(temp_list);
157
158     if (std_x == 0 || std_y == 0)
159     {
160         // correlation is undefined if either std is zero
161         return 0.0;
162     }
163
164     float numerator = 0.0;
165     while (light_item != NULL && temp_item != NULL)
166     {
167         float x = light_item->value;
168         float y = temp_item->value;
169
170         numerator += (x - avg_x) * (y - avg_y);
171
172         light_item = list_item_next(light_item);
173         temp_item = list_item_next(temp_item);
174     }
175     return numerator / (std_x * std_y);
176 }
177
178 // TODO: Implement SFFT
179 typedef struct
180 {
181     float real;
182     float imag;
183 } complex_t;
184
185 // helper functions
186 float list_get(list_t lst, int index)
187 {
188     struct sensor_data *element = (struct sensor_data *)list_head(lst);
189     int curr_index = 0;
190     while (element != NULL && curr_index < index)
191     {
192         element = (struct sensor_data *)list_item_next(element);
193         curr_index++;
194     }

```



```

195     if (element == NULL)
196     {
197         printf("Index %d is out of bounds. \n", index);
198         return 0.0;
199     }
200     return element->value;
201 }
202
203 // define memory pool
204 MEMB(chunk_pool, float, 4);
205 MEMB(fft_result_pool, complex_t, 4);
206
207 float sine_approx(float x)
208 {
209     float term = x; // First term
210     float result = term;
211     int sign = -1;
212     int n;
213
214     // Use 5 terms for approximation
215     for (n = 3; n <= 9; n += 2)
216     {
217         term = term * x * x / (n * (n - 1)); // Compute next term
218         result += sign * term;                // Add/subtract the term
219         sign = -sign;                         // Alternate sign
220     }
221     return result;
222 }
223
224 float cosine_approx(float x)
225 {
226     float term = 1.0f; // First term
227     float result = term;
228     int sign = -1;
229     int n;
230
231     // Use 5 terms for approximation
232     for (n = 2; n <= 8; n += 2)
233     {
234         term = term * x * x / (n * (n - 1)); // Compute next term
235         result += sign * term;                // Add/subtract the term
236         sign = -sign;                         // Alternate sign
237     }
238     return result;
239 }
240 void fft(complex_t *data, int fft_size)
241 {
242     int i, j, len;
243
244     // Bit-reversal permutation
245     for (i = 0, j = 0; i < fft_size; i++)
246     {
247         if (i < j)
248         {
249             // Swap data[i] and data[j]
250             complex_t temp = data[i];
251             data[i] = data[j];
252             data[j] = temp;
253         }
254         int m = fft_size / 2;
255         while (m >= 1 && j >= m)
256         {
257             j -= m;
258             m /= 2;
259         }
260         j += m;
261     }
262
263     // Iterative FFT computation

```

```

264     for (len = 2; len <= fft_size; len *= 2)
265     {
266         float angle = -2.0f * PI / len;
267         complex_t wlen = {cosine_approx(angle), sine_approx(angle)};
268
269         for (i = 0; i < fft_size; i += len)
270         {
271             complex_t w = {1.0, 0.0};
272             for (j = 0; j < len / 2; j++)
273             {
274                 complex_t u = data[i + j];
275                 complex_t v = {
276                     w.real * data[i + j + len / 2].real - w.imag * data[i + j + len / 2].imag,
277                     w.real * data[i + j + len / 2].imag + w.imag * data[i + j + len / 2].real};
278
279                 data[i + j].real = u.real + v.real;
280                 data[i + j].imag = u.imag + v.imag;
281                 data[i + j + len / 2].real = u.real - v.real;
282                 data[i + j + len / 2].imag = u.imag - v.imag;
283
284                 // Update twiddle factor
285                 complex_t w_new = {
286                     w.real * wlen.real - w.imag * wlen.imag,
287                     w.real * wlen.imag + w.imag * wlen.real};
288                 w = w_new;
289             }
290         }
291     }
292 }
293
294 // Helper function to print STFT results as a table
295 void print_stft_results(int chunk_index, complex_t *data, int chunk_size)
296 {
297     int j;
298     printf("Chunk %3d", chunk_index);
299     for (j = 0; j < chunk_size; j++)
300     {
301         print_float(data[j].real);
302         printf(" ");
303         print_float(data[j].imag);
304         printf("i");
305     }
306     printf("\n");
307 }
308
309 static void perform_stft(list_t lst, int chunk_size, int hop_size)
310 {
311     memb_init(&chunk_pool);
312     memb_init(&fft_result_pool);
313
314     int signal_length = list_length(lst);
315     int num_chunks = (signal_length - chunk_size) / hop_size + 1;
316     printf("signal length %d and %d chunks\n", signal_length, num_chunks);
317     int i, j;
318     for (i = 0; i < num_chunks; i++)
319     {
320         float *chunk = (float *)memb_alloc(&chunk_pool);
321         if (chunk == NULL)
322         {
323             printf("Chunk allocation failed\n");
324             return;
325         }
326
327         int start_index = i * hop_size;
328         // printf("Start index: %d\n", start_index);
329         for (j = 0; j < chunk_size; j++)
330         {
331             int signal_index = start_index + j;
332             // printf("Signal index: %d\n", signal_index);
333             if (signal_index < signal_length)

```

```
334         {
335             chunk[j] = list_get(lst, signal_index);
336         }
337         else
338         {
339             chunk[j] = 0.0f; // just in case, not needed in our case.
340         }
341     }
342     complex_t data[chunk_size];
343     for (j = 0; j < chunk_size; j++)
344     {
345         data[j].real = chunk[j];
346         data[j].imag = 0.0f;
347         /*-----DEBUGGING-----*/
348         // printf("CHUNK VALUE: \n");
349         // print_float(data[j].real);
350         // printf("\n");
351         // print_float(data[j].imag);
352         // printf("\n");
353         /*-----DEBUGGING-----*/
354     }
355     fft(data, chunk_size);
356
357     /*-----DEBUGGING-----*/
358     print_stft_results(i, data, chunk_size);
359     /*-----DEBUGGING-----*/
360
361     memb_free(&chunk_pool, chunk);
362 }
363 }
364
365 void compute_power_spectrum(complex_t *chunk, float *power_spectrum, int chunk_size)
366 {
367     int k;
368     for (k = 0; k < chunk_size; k++)
369     {
370         power_spectrum[k] = chunk[k].real * chunk[k].real +
371                             chunk[k].imag * chunk[k].imag;
372     }
373 }
374
375 static float compute_spectral_entropy(list_t lst, int chunk_size, int hop_size)
376 {
377     int signal_length = list_length(lst);
378     int num_chunks = (signal_length - chunk_size) / hop_size + 1;
379
380     if (num_chunks <= 0)
381         return 0.0f;
382
383     // Allocate memory for power spectrum calculations
384     float power_spectrum[chunk_size];
385     float avg_power_spectrum[chunk_size];
386     float pdf[chunk_size];
387
388     // Initialize average power spectrum
389     int i;
390     for (i = 0; i < chunk_size; i++)
391     {
392         avg_power_spectrum[i] = 0.0f;
393     }
394
395     // Process each chunk
396     for (i = 0; i < num_chunks; i++)
397     {
398         complex_t chunk_data[chunk_size];
399         int j;
400         for (j = 0; j < chunk_size; j++)
401         {
402             int signal_index = i * hop_size + j;
403             if (signal_index < signal_length)
```

```
404     {
405         chunk_data[j].real = list_get(lst, signal_index);
406         chunk_data[j].imag = 0.0f;
407     }
408     else
409     {
410         chunk_data[j].real = 0.0f;
411         chunk_data[j].imag = 0.0f;
412     }
413 }
414
415 fft(chunk_data, chunk_size);
416
417 // Compute power spectrum for this chunk
418 compute_power_spectrum(chunk_data, power_spectrum, chunk_size);
419
420 // Add to average power spectrum
421 for (j = 0; j < chunk_size; j++)
422 {
423     avg_power_spectrum[j] += power_spectrum[j];
424 }
425 }
426
427 // Compute final average
428 float total_power = 0.0f;
429 for (i = 0; i < chunk_size; i++)
430 {
431     avg_power_spectrum[i] /= num_chunks;
432     total_power += avg_power_spectrum[i];
433 }
434
435 // Compute PDF and entropy
436 float entropy = 0.0f;
437 if (total_power > 0.0f)
438 {
439     for (i = 0; i < chunk_size; i++)
440     {
441         if (avg_power_spectrum[i] > 0.0f)
442         {
443             pdf[i] = avg_power_spectrum[i] / total_power;
444             entropy -= pdf[i] * logf(pdf[i]);
445         }
446     }
447 }
448
449 return entropy;
450 }
451
452 static void aggregate_and_report()
453 {
454     struct sensor_data *light_item = list_head(light_list);
455     struct sensor_data *temp_item = list_head(temp_list);
456     printf("X (Light Sensor Readings) = [");
457     for (light_item = list_head(light_list); light_item != NULL; light_item = list_item_next(light_item))
458     {
459         print_float(light_item->value);
460         if (list_item_next(light_item) != NULL)
461         {
462             printf(", ");
463         }
464     }
465     printf("]\n");
466     printf("Y (Temperature Sensor Readings) = [");
467     for (temp_item = list_head(temp_list); temp_item != NULL; temp_item = list_item_next(temp_item))
468     {
469         print_float(temp_item->value);
470         if (list_item_next(temp_item) != NULL)
471         {
472             printf(", ");
473         }
474     }
475 }
```

```
474     }
475     printf("]\n");
476
477     printf("Manhattan Distance: ");
478     print_float(calculate_manhattan_dist(light_list, temp_list));
479     printf("\n");
480
481     printf("Correlation: ");
482     print_float(calculate_correlation(light_list, temp_list));
483     printf("\n");
484
485     printf("Performing STFT with ");
486     perform_stft(light_list, 4, 2);
487     printf("\n");
488
489     printf("Spectral Entropy: ");
490     print_float(compute_spectral_entropy(light_list, 4, 2));
491     printf("\n");
492 }
493 /*-----*/
494 PROCESS(sensor_reading_process, "Sensor reading process");
495 AUTOSTART_PROCESSES(&sensor_reading_process);
496 /*-----*/
497 PROCESS_THREAD(sensor_reading_process, ev, data)
498 {
499     static struct etimer timer;
500     static int sample_counter = 0;
501     static int k = 12; // number of samples before calculation
502
503     PROCESS_BEGIN();
504     etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second
505
506     SENSORS_ACTIVATE(light_sensor);
507     SENSORS_ACTIVATE(sht11_sensor);
508
509     while (1)
510     {
511         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
512
513         float light = read_light_sensor();
514         float temp = get_temp_sensor();
515         add_sensor_data(light, light_list, &light_mem);
516         add_sensor_data(temp, temp_list, &temp_mem);
517
518         sample_counter++;
519         if (sample_counter >= k)
520         {
521             aggregate_and_report();
522             sample_counter = 0;
523         }
524
525         etimer_reset(&timer);
526     }
527
528     PROCESS_END();
529 }
```