

```

1 /*-----Basic Features-----*/
2 #include "contiki.h"
3 #include "dev/light-sensor.h"
4 #include "lib/list.h"
5 #include "lib/memb.h"
6 #include <stdio.h> /* For printf() */
7
8 #define BUFFER_SIZE 12
9 #define LOW_ACTIVITY_THRESHOLD 1000
10 #define HIGH_ACTIVITY_THRESHOLD 2000
11 #define SAX_FRAGMENTS 4
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 /* Use linked list for sensor data */
31 struct sensor_data
32 {
33     struct sensor_data *next;
34     float light;
35 };
36
37 LIST(sensor_list);
38 MEMB(sensor_mem, struct sensor_data, BUFFER_SIZE);
39
40 static void add_sensor_data(float light)
41 {
42     struct sensor_data *new_data;
43
44     if (list_length(sensor_list) >= BUFFER_SIZE)
45     {
46         struct sensor_data *oldest = list_pop(sensor_list);
47         memb_free(&sensor_mem, oldest);
48     }
49
50     new_data = memb_alloc(&sensor_mem);
51     if (new_data == NULL)
52     {
53         printf("Memory allocation failed!\n");
54         return;
55     }
56
57     new_data->light = light;
58     list_add(sensor_list, new_data);
59 }
60
61 /* Calculate average */
62 static float calculate_avg()
63 {
64     struct sensor_data *item;
65     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     printf("]\n");
227 }
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)

```

```

264 {
265     static struct etimer timer;
266     static int sample_counter = 0;
267     static int k = 12; // number of samples before aggregation
268
269     PROCESS_BEGIN();
270     etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second
271
272     SENSORS_ACTIVATE(light_sensor);
273
274     while (1)
275     {
276         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
277
278         float light = read_light_sensor();
279         add_sensor_data(light);
280         sample_counter++;
281
282         if (sample_counter >= k)
283         {
284             aggregate_and_report();
285             sample_counter = 0;
286         }
287
288         etimer_reset(&timer);
289     }
290
291     PROCESS_END();
292 }

```

```

1 /*-----Advanced Features-----*/
2 #include "contiki.h"
3 #include "dev/light-sensor.h"
4 #include "dev/sht11-sensor.h"
5 #include "lib/list.h"
6 #include "lib/memb.h"
7 #include <stdio.h> /* For printf() */
8 #include <math.h>
9
10 #define LN2 0.69314718056
11 #define BUFFER_SIZE 12
12 #define PI 3.141592653589793
13
14 /* Helper functions */
15 static void print_float(float number)
16 {
17     int integer_part = (int)number;
18     int decimal_part = (int)((number - integer_part) * 1000);
19     printf("%d.%02d", integer_part, decimal_part);
20 }
21
22 float read_light_sensor(void)
23 {
24     int lightData = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
25     float V_sensor = 1.5 * lightData / 4096;
26     float I = V_sensor / 100000;
27     float light = 0.625 * 1e6 * I * 1000;
28     return light;
29 }
30
31 float get_temp_sensor(void)
32 {
33     float tempData = sht11_sensor.value(SHT11_SENSOR_TEMP_SKYSIM); // For Cooja Sim
34     float d1 = -39.6;
35     float d2 = 0.04; // Adjust as per sensor calibration
36     float temp = tempData * d2 + d1;
37     return temp;
38 }
39
40 /* Use linked list for sensor data */
41 struct sensor_data
42 {
43     struct sensor_data *next;
44     float value;
45 };
46
47 LIST(light_list);
48 LIST(temp_list);
49 MEMB(light_mem, struct sensor_data, BUFFER_SIZE);
50 MEMB(temp_mem, struct sensor_data, BUFFER_SIZE);
51
52 static void add_sensor_data(float value, list_t lst, struct memb *mem)
53 {
54     struct sensor_data *new_data;
55
56     if (list_length(lst) >= BUFFER_SIZE)
57     {
58         struct sensor_data *oldest = list_pop(lst);
59         memb_free(mem, oldest);
60     }
61
62     new_data = memb_alloc(mem);
63     if (new_data == NULL)
64     {
65         printf("Memory allocation failed!\n");
66         return;
67     }
68
69     new_data->value = value;
70     list_add(lst, new_data);
71 }
72
73 static float calculate_sun/(list_t lst)

```

```

73 static float calculate_avg(list_t lst)
74 {
75     struct sensor_data *item;
76     float sum = 0.0;
77     int count = 0;
78     for (item = list_head(lst); item != NULL; item = list_item_next(item))
79     {
80         sum += item->value;
81         count++;
82     }
83     return (count == 0) ? 0.0 : sum / count;
84 }
85
86 static float calculate_ssd(float avg, list_t lst)
87 {
88     struct sensor_data *item;
89     float ssd = 0.0;
90     for (item = list_head(lst); item != NULL; item = list_item_next(item))
91     {
92         float diff = item->value - avg;
93         ssd += diff * diff;
94     }
95     return ssd;
96 }
97
98 static float sqrt_approx(float ssd)
99 {
100     float error = 0.001; // Error tolerance for Babylonian method
101     float x = ssd; // Initial guess for square root
102     float difference;
103     int i;
104
105     if (ssd == 0)
106     {
107         return 0.0; // No variance
108     }
109
110     for (i = 0; i < 50; i++)
111     {
112         x = 0.5 * (x + ssd / x);
113         difference = x * x - ssd;
114         if (difference < 0)
115         {
116             difference = -difference;
117         }
118         if (difference < error)
119         {
120             break;
121         }
122     }
123     return x;
124 }
125
126 static float calculate_std(list_t lst)
127 {
128     float avg = calculate_avg(lst);
129     float ssd = calculate_ssd(avg, lst);
130     return sqrt_approx(ssd);
131 }
132
133 static float calculate_manhattan_dist(list_t light_list, list_t temp_list)
134 {
135     struct sensor_data *light_item = list_head(light_list);
136     struct sensor_data *temp_item = list_head(temp_list);
137     double dist = 0.0; // Use double for accumulation
138
139     while (light_item != NULL && temp_item != NULL)
140     {
141         double diff = (double)light_item->value - (double)temp_item->value;
142         dist += (diff < 0) ? -diff : diff; // Inline abs for double
143         light_item = list_item_next(light_item);
144         temp_item = list_item_next(temp_item);
145     }
146     return (float)dist;

```

```

147 }
148
149 static float calculate_correlation(list_t light_list, list_t temp_list)
150 {
151     struct sensor_data *light_item = list_head(light_list);
152     struct sensor_data *temp_item = list_head(temp_list);
153
154     float avg_x = calculate_avg(light_list);
155     float avg_y = calculate_avg(temp_list);
156     float std_x = calculate_std(light_list);
157     float std_y = calculate_std(temp_list);
158
159     if (std_x == 0 || std_y == 0)
160     {
161         // correlation is undefined if either std is zero
162         return 0.0;
163     }
164
165     float numerator = 0.0;
166     while (light_item != NULL && temp_item != NULL)
167     {
168         float x = light_item->value;
169         float y = temp_item->value;
170
171         numerator += (x - avg_x) * (y - avg_y);
172
173         light_item = list_item_next(light_item);
174         temp_item = list_item_next(temp_item);
175     }
176     return numerator / (std_x * std_y);
177 }
178
179 // TODO: Implement SFFT
180 typedef struct
181 {
182     float real;
183     float imag;
184 } complex_t;
185
186 // helper functions
187 float list_get(list_t lst, int index)
188 {
189     struct sensor_data *element = (struct sensor_data *)list_head(lst);
190     int curr_index = 0;
191     while (element != NULL && curr_index < index)
192     {
193         element = (struct sensor_data *)list_item_next(element);
194         curr_index++;
195     }
196     if (element == NULL)
197     {
198         printf("Index %d is out of bounds. \n", index);
199         return 0.0;
200     }
201     return element->value;
202 }
203
204 // define memory pool
205 MEMB(chunk_pool, float, 4);
206 MEMB(fft_result_pool, complex_t, 4);
207
208 float sine_approx(float x)
209 {
210     float term = x; // First term
211     float result = term;
212     int sign = -1;
213     int n;
214
215     // Use 5 terms for approximation
216     for (n = 3; n <= 9; n += 2)
217     {
218         term = term * x * x / (n * (n - 1)); // Compute next term
219         result += sign * term;                // Add/subtract the term

```



```

220     sign = -sign;                                // Alternate sign
221 }
222 return result;
223 }
224
225 float cosine_approx(float x)
226 {
227     float term = 1.0f; // First term
228     float result = term;
229     int sign = -1;
230     int n;
231
232     // Use 5 terms for approximation
233     for (n = 2; n <= 8; n += 2)
234     {
235         term = term * x * x / (n * (n - 1)); // Compute next term
236         result += sign * term;                // Add/subtract the term
237         sign = -sign;                         // Alternate sign
238     }
239     return result;
240 }
241 void fft(complex_t *data, int fft_size)
242 {
243     int i, j, len;
244
245     // Bit-reversal permutation
246     for (i = 0, j = 0; i < fft_size; i++)
247     {
248         if (i < j)
249         {
250             // Swap data[i] and data[j]
251             complex_t temp = data[i];
252             data[i] = data[j];
253             data[j] = temp;
254         }
255         int m = fft_size / 2;
256         while (m >= 1 && j >= m)
257         {
258             j -= m;
259             m /= 2;
260         }
261         j += m;
262     }
263
264     // Iterative FFT computation
265     for (len = 2; len <= fft_size; len *= 2)
266     {
267         float angle = -2.0f * PI / len;
268         complex_t wlen = {cosine_approx(angle), sine_approx(angle)};
269
270         for (i = 0; i < fft_size; i += len)
271         {
272             complex_t w = {1.0, 0.0};
273             for (j = 0; j < len / 2; j++)
274             {
275                 complex_t u = data[i + j];
276                 complex_t v = {
277                     w.real * data[i + j + len / 2].real - w.imag * data[i + j + len / 2].imag,
278                     w.real * data[i + j + len / 2].imag + w.imag * data[i + j + len / 2].real};
279
280                 data[i + j].real = u.real + v.real;
281                 data[i + j].imag = u.imag + v.imag;
282                 data[i + j + len / 2].real = u.real - v.real;
283                 data[i + j + len / 2].imag = u.imag - v.imag;
284
285                 // Update twiddle factor
286                 complex_t w_new = {
287                     w.real * wlen.real - w.imag * wlen.imag,
288                     w.real * wlen.imag + w.imag * wlen.real};
289                 w = w_new;
290             }
291         }
292     }
293 }

```

```

293 ,
294
295 // Helper function to print STFT results as a table
296 void print_stft_results(int chunk_index, complex_t *data, int chunk_size)
297 {
298     int j;
299     printf("Chunk %-3d", chunk_index);
300     for (j = 0; j < chunk_size; j++)
301     {
302         print_float(data[j].real);
303         printf(" + ");
304         print_float(data[j].imag);
305         printf("i      ");
306     }
307     printf("\n");
308 }
309
310 static void perform_stft(list_t lst, int chunk_size, int hop_size)
311 {
312     memb_init(&chunk_pool);
313     memb_init(&fft_result_pool);
314
315     int signal_length = list_length(lst);
316     int num_chunks = (signal_length - chunk_size) / hop_size + 1;
317     printf("signal length %d and %d chunks\n", signal_length, num_chunks);
318     int i, j;
319     for (i = 0; i < num_chunks; i++)
320     {
321         float *chunk = (float *)memb_alloc(&chunk_pool);
322         if (chunk == NULL)
323         {
324             printf("Chunk allocation failed\n");
325             return;
326         }
327
328         int start_index = i * hop_size;
329         // printf("Start index: %d\n", start_index);
330
331         for (j = 0; j < chunk_size; j++)
332         {
333             int signal_index = start_index + j;
334             // printf("Signal index: %d\n", signal_index);
335             if (signal_index < signal_length)
336             {
337                 chunk[j] = list_get(lst, signal_index);
338             }
339             else
340             {
341                 chunk[j] = 0.0f; // just in case, not needed in our case.
342             }
343         }
344         complex_t data[chunk_size];
345         for (j = 0; j < chunk_size; j++)
346         {
347             data[j].real = chunk[j];
348             data[j].imag = 0.0f;
349             /*-----DEBUGGING-----*/
350             // printf("CHUNK VALUE: \n");
351             // print_float(data[j].real);
352             // printf("\n");
353             // print_float(data[j].imag);
354             // printf("\n");
355             /*-----DEBUGGING-----*/
356         }
357         fft(data, chunk_size);
358         /*-----DEBUGGING-----*/
359         print_stft_results(i, data, chunk_size);
360         /*-----DEBUGGING-----*/
361         memb_free(&chunk_pool, chunk);
362     }
363 }
364
365
366 void compute_power_spectrum(complex_t *chunk, float *power_spectrum, int chunk_size)

```

```

367 {
368     int k;
369     for (k = 0; k < chunk_size; k++)
370     {
371         power_spectrum[k] = chunk[k].real * chunk[k].real +
372                             chunk[k].imag * chunk[k].imag;
373     }
374 }
375
376 static float compute_spectral_entropy(list_t lst, int chunk_size, int hop_size)
377 {
378     int signal_length = list_length(lst);
379     int num_chunks = (signal_length - chunk_size) / hop_size + 1;
380
381     if (num_chunks <= 0)
382         return 0.0f;
383
384     // Allocate memory for power spectrum calculations
385     float power_spectrum[chunk_size];
386     float avg_power_spectrum[chunk_size];
387     float pdf[chunk_size];
388
389     // Initialize average power spectrum
390     int i;
391     for (i = 0; i < chunk_size; i++)
392     {
393         avg_power_spectrum[i] = 0.0f;
394     }
395
396     // Process each chunk
397     for (i = 0; i < num_chunks; i++)
398     {
399         complex_t chunk_data[chunk_size];
400         int j;
401         for (j = 0; j < chunk_size; j++)
402         {
403             int signal_index = i * hop_size + j;
404             if (signal_index < signal_length)
405             {
406                 chunk_data[j].real = list_get(lst, signal_index);
407                 chunk_data[j].imag = 0.0f;
408             }
409             else
410             {
411                 chunk_data[j].real = 0.0f;
412                 chunk_data[j].imag = 0.0f;
413             }
414         }
415
416         fft(chunk_data, chunk_size);
417
418         // Compute power spectrum for this chunk
419         compute_power_spectrum(chunk_data, power_spectrum, chunk_size);
420
421         // Add to average power spectrum
422         for (j = 0; j < chunk_size; j++)
423         {
424             avg_power_spectrum[j] += power_spectrum[j];
425         }
426     }
427
428     // Compute final average
429     float total_power = 0.0f;
430     for (i = 0; i < chunk_size; i++)
431     {
432         avg_power_spectrum[i] /= num_chunks;
433         total_power += avg_power_spectrum[i];
434     }
435
436     // Compute PDF and entropy
437     float entropy = 0.0f;
438     if (total_power > 0.0f)
439     {

```

```

440         for (i = 0; i < chunk_size; i++)
441         {
442             if (avg_power_spectrum[i] > 0.0f)
443             {
444                 pdf[i] = avg_power_spectrum[i] / total_power;
445                 entropy -= pdf[i] * logf(pdf[i]);
446             }
447         }
448     }
449
450     return entropy;
451 }
452
453 static void aggregate_and_report()
454 {
455     struct sensor_data *light_item = list_head(light_list);
456     struct sensor_data *temp_item = list_head(temp_list);
457     printf("X (Light Sensor Readings) = [");
458     for (light_item = list_head(light_list); light_item != NULL; light_item = list_item_next(light_item))
459     {
460         print_float(light_item->value);
461         if (list_item_next(light_item) != NULL)
462         {
463             printf(", ");
464         }
465     }
466     printf("]\n");
467     printf("Y (Temperature Sensor Readings) = [");
468     for (temp_item = list_head(temp_list); temp_item != NULL; temp_item = list_item_next(temp_item))
469     {
470         print_float(temp_item->value);
471         if (list_item_next(temp_item) != NULL)
472         {
473             printf(", ");
474         }
475     }
476     printf("]\n");
477
478     printf("Manhattan Distance: ");
479     print_float(calculate_manhattan_dist(light_list, temp_list));
480     printf("\n");
481
482     printf("Correlation: ");
483     print_float(calculate_correlation(light_list, temp_list));
484     printf("\n");
485
486     printf("Performing STFT with ");
487     perform_stft(light_list, 4, 2);
488     printf("\n");
489
490     printf("Spectral Entropy: ");
491     print_float(compute_spectral_entropy(light_list, 4, 2));
492     printf("\n");
493 }
494 /*-----*/
495 PROCESS(sensor_reading_process, "Sensor reading process");
496 AUTOSTART_PROCESSES(&sensor_reading_process);
497 /*-----*/
498 PROCESS_THREAD(sensor_reading_process, ev, data)
499 {
500     static struct etimer timer;
501     static int sample_counter = 0;
502     static int k = 12; // number of samples before calculation
503
504     PROCESS_BEGIN();
505     etimer_set(&timer, CLOCK_CONF_SECOND / 2); // two readings per second
506
507     SENSORS_ACTIVATE(light_sensor);
508     SENSORS_ACTIVATE(sht11_sensor);
509
510     while (1)
511     {
512         PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
513

```

```
514     float light = read_light_sensor();
515     float temp = get_temp_sensor();
516     add_sensor_data(light, light_list, &light_mem);
517     add_sensor_data(temp, temp_list, &temp_mem);
518
519     sample_counter++;
520     if (sample_counter >= k)
521     {
522         aggregate_and_report();
523         sample_counter = 0;
524     }
525
526     etimer_reset(&timer);
527 }
528
529 PROCESS_END();
530 }
```