```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;
```

```
    int count = 0;
    for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
    {
        sum += item->light;
        count++;
    }
    return (count == 0) ? 0.0 : sum / count;
}

static float calculate_ssd(float avg)
{
    struct sensor_data *item;
    float ssd = 0.0;
    for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
    {
        float diff = item->light - avg;
        ssd += diff * diff;
    }
    return ssd;
}

static float sqrt_approx(float ssd)
{
    float error = 0.001; // Error tolerance for Babylonian method
    float x = ssd;       // Initial guess for square root
    float difference;
    int i;

    if (ssd == 0)
    {
        return 0.0; // No variance
    }

    for (i = 0; i < 50; i++)
    { // Babylonian method
        x = 0.5 * (x + ssd / x);
        difference = x * x - ssd;
        if (difference < 0)
        {
            difference = -difference;
        }
        if (difference < error)
        {
            break;
        }
    }
    return x;
}

static float calculate_std()
{
    float avg = calculate_avg();
    float ssd = calculate_ssd(avg);
    return sqrt_approx(ssd);
}

void perform_sax(char sax_output[SAX_FRAGMENTS])
{
    struct sensor_data *item;
    float fragment_means[SAX_FRAGMENTS] = {0};
    int fragment_size = BUFFER_SIZE / SAX_FRAGMENTS;
    int i = 0, count = 0;

    // Compute fragment means
    for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
    {
```

```c
        fragment_means[i] += item->light;
        count++;
        if (count == fragment_size)
        {
            fragment_means[i] /= fragment_size;
            i++;
            count = 0;
        }
    }

    // Normalize fragment means
    float avg = calculate_avg();
    float std = calculate_std();
    char alphabet[4] = {'A', 'B', 'C', 'D'};
    float breakpoints[3] = {-0.67, 0, 0.67};

    // Assign SAX symbols
    for (i = 0; i < SAX_FRAGMENTS; i++)
    {
        float z = (fragment_means[i] - avg) / std;
        if (z <= breakpoints[0])
        {
            sax_output[i] = alphabet[0];
        }
        else if (z <= breakpoints[1])
        {
            sax_output[i] = alphabet[1];
        }
        else if (z <= breakpoints[2])
        {
            sax_output[i] = alphabet[2];
        }
        else
        {
            sax_output[i] = alphabet[3];
        }
    }
}

static void aggregate_and_report()
{
    struct sensor_data *item = list_head(sensor_list);
    float std = calculate_std();
    float avg = calculate_avg();
    char sax_output[SAX_FRAGMENTS];

    // Print the original buffer
    printf("B = [");
    for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
    {
        print_float(item->light);
        if (list_item_next(item) != NULL)
        {
            printf(", ");
        }
    }
    printf("]\n");

    // Print the standard deviation
    printf("StdDev = ");
    print_float(std);
    printf("\n");

    // Determine the activity level and aggregation
    if (std < LOW_ACTIVITY_THRESHOLD)
    {
```

```
        printf("Aggregation = 12-into-1\n");
        printf("X = [");
        print_float(avg);
        printf("]\n");
    }
    else if (std < HIGH_ACTIVITY_THRESHOLD)
    {
        printf("Aggregation = 4-into-1\n");
        printf("X = [");
        int count = 0;
        float sum = 0.0;
        item = list_head(sensor_list);
        while (item != NULL)
        {
            sum += item->light;
            count++;
            if (count == 4)
            {
                print_float(sum / 4);
                sum = 0.0;
                count = 0;
                if (list_item_next(item) != NULL)
                {
                    printf(", ");
                }
            }
            item = list_item_next(item);
        }

        printf("]\n");
    }
    else
    {
        printf("Aggregation = 1-into-1\n");
        printf("X = [");
        for (item = list_head(sensor_list); item != NULL; item = list_item_next(item))
        {
            print_float(item->light);
            if (list_item_next(item) != NULL)
            {
                printf(", ");
            }
        }
        printf("]\n");
    }

    // Perform SAX transformation and print
    perform_sax(sax_output);
    printf("SAX = [");
    int i;
    for (i = 0; i < SAX_FRAGMENTS; i++)
    {
        printf("%c", sax_output[i]);
        if (i < SAX_FRAGMENTS - 1)
        {
            printf(", ");
        }
    }
    printf("]\n");
}

/*---------------------------------------------------------------------------*/
PROCESS(sensor_reading_process, "Sensor reading process");
AUTOSTART_PROCESSES(&sensor_reading_process);
/*---------------------------------------------------------------------------*/
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();
}
```

```c
/*----------Advanced Features----------*/
#include "contiki.h"
#include "dev/light-sensor.h"
#include "dev/sht11-sensor.h"
#include "lib/list.h"
#include "lib/memb.h"
#include <stdio.h> /* For printf() */
#include <math.h>

#define LN2 0.69314718056
#define BUFFER_SIZE 12
#define PI 3.141592653589793

/* 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;
}

float get_temp_sensor(void)
{
    float tempData = sht11_sensor.value(SHT11_SENSOR_TEMP_SKYSIM); // For Cooja Sim
    float d1 = -39.6;
    float d2 = 0.04; // Adjust as per sensor calibration
    float temp = tempData * d2 + d1;
    return temp;
}

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

LIST(light_list);
LIST(temp_list);
MEMB(light_mem, struct sensor_data, BUFFER_SIZE);
MEMB(temp_mem, struct sensor_data, BUFFER_SIZE);

static void add_sensor_data(float value, list_t lst, struct memb *mem)
{
    struct sensor_data *new_data;

    if (list_length(lst) >= BUFFER_SIZE)
    {
        struct sensor_data *oldest = list_pop(lst);
        memb_free(mem, oldest);
    }

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

    new_data->value = value;
    list_add(lst, new_data);
}

static float calculate_avg(list_t lst)
```

```
static float calculate_avg(list_t lst)
{
    struct sensor_data *item;
    float sum = 0.0;
    int count = 0;
    for (item = list_head(lst); item != NULL; item = list_item_next(item))
    {
        sum += item->value;
        count++;
    }
    return (count == 0) ? 0.0 : sum / count;
}

static float calculate_ssd(float avg, list_t lst)
{
    struct sensor_data *item;
    float ssd = 0.0;
    for (item = list_head(lst); item != NULL; item = list_item_next(item))
    {
        float diff = item->value - avg;
        ssd += diff * diff;
    }
    return ssd;
}

static float sqrt_approx(float ssd)
{
    float error = 0.001; // Error tolerance for Babylonian method
    float x = ssd;       // Initial guess for square root
    float difference;
    int i;

    if (ssd == 0)
    {
        return 0.0; // No variance
    }

    for (i = 0; i < 50; i++)
    {
        x = 0.5 * (x + ssd / x);
        difference = x * x - ssd;
        if (difference < 0)
        {
            difference = -difference;
        }
        if (difference < error)
        {
            break;
        }
    }
    return x;
}

static float calculate_std(list_t lst)
{
    float avg = calculate_avg(lst);
    float ssd = calculate_ssd(avg, lst);
    return sqrt_approx(ssd);
}

static float calculate_manhattan_dist(list_t light_list, list_t temp_list)
{
    struct sensor_data *light_item = list_head(light_list);
    struct sensor_data *temp_item = list_head(temp_list);
    double dist = 0.0; // Use double for accumulation

    while (light_item != NULL && temp_item != NULL)
    {
        double diff = (double)light_item->value - (double)temp_item->value;
        dist += (diff < 0) ? -diff : diff; // Inline abs for double
        light_item = list_item_next(light_item);
        temp_item = list_item_next(temp_item);
    }
    return (float)dist;
```

```
}

static float calculate_correlation(list_t light_list, list_t temp_list)
{
    struct sensor_data *light_item = list_head(light_list);
    struct sensor_data *temp_item = list_head(temp_list);

    float avg_x = calculate_avg(light_list);
    float avg_y = calculate_avg(temp_list);
    float std_x = calculate_std(light_list);
    float std_y = calculate_std(temp_list);

    if (std_x == 0 || std_y == 0)
    {
        // correlation is undefined if either std is zero
        return 0.0;
    }

    float numerator = 0.0;
    while (light_item != NULL && temp_item != NULL)
    {
        float x = light_item->value;
        float y = temp_item->value;

        numerator += (x - avg_x) * (y - avg_y);

        light_item = list_item_next(light_item);
        temp_item = list_item_next(temp_item);
    }
    return numerator / (std_x * std_y);
}

// TODO: Implement SFFT
typedef struct
{
    float real;
    float imag;
} complex_t;

// helper functions
float list_get(list_t lst, int index)
{
    struct sensor_data *element = (struct sensor_data *)list_head(lst);
    int curr_index = 0;
    while (element != NULL && curr_index < index)
    {
        element = (struct sensor_data *)list_item_next(element);
        curr_index++;
    }
    if (element == NULL)
    {
        printf("Index %d is out of bounds. \n", index);
        return 0.0;
    }
    return element->value;
}

// define memory pool
MEMB(chunk_pool, float, 4);
MEMB(fft_result_pool, complex_t, 4);

float sine_approx(float x)
{
    float term = x; // First term
    float result = term;
    int sign = -1;
    int n;

    // Use 5 terms for approximation
    for (n = 3; n <= 9; n += 2)
    {
        term = term * x * x / (n * (n - 1)); // Compute next term
        result += sign * term;               // Add/subtract the term
```

```
        sign = -sign;                        // Alternate sign
    }
    return result;
}

float cosine_approx(float x)
{
    float term = 1.0f; // First term
    float result = term;
    int sign = -1;
    int n;

    // Use 5 terms for approximation
    for (n = 2; n <= 8; n += 2)
    {
        term = term * x * x / (n * (n - 1)); // Compute next term
        result += sign * term;               // Add/subtract the term
        sign = -sign;                        // Alternate sign
    }
    return result;
}
void fft(complex_t *data, int fft_size)
{
    int i, j, len;

    // Bit-reversal permutation
    for (i = 0, j = 0; i < fft_size; i++)
    {
        if (i < j)
        {
            // Swap data[i] and data[j]
            complex_t temp = data[i];
            data[i] = data[j];
            data[j] = temp;
        }
        int m = fft_size / 2;
        while (m >= 1 && j >= m)
        {
            j -= m;
            m /= 2;
        }
        j += m;
    }

    // Iterative FFT computation
    for (len = 2; len <= fft_size; len *= 2)
    {
        float angle = -2.0f * PI / len;
        complex_t wlen = {cosine_approx(angle), sine_approx(angle)};

        for (i = 0; i < fft_size; i += len)
        {
            complex_t w = {1.0, 0.0};
            for (j = 0; j < len / 2; j++)
            {
                complex_t u = data[i + j];
                complex_t v = {
                    w.real * data[i + j + len / 2].real - w.imag * data[i + j + len / 2].imag,
                    w.real * data[i + j + len / 2].imag + w.imag * data[i + j + len / 2].real};

                data[i + j].real = u.real + v.real;
                data[i + j].imag = u.imag + v.imag;
                data[i + j + len / 2].real = u.real - v.real;
                data[i + j + len / 2].imag = u.imag - v.imag;

                // Update twiddle factor
                complex_t w_new = {
                    w.real * wlen.real - w.imag * wlen.imag,
                    w.real * wlen.imag + w.imag * wlen.real};
                w = w_new;
            }
        }
    }
}
```

```
}

// Helper function to print STFT results as a table
void print_stft_results(int chunk_index, complex_t *data, int chunk_size)
{
    int j;
    printf("Chunk %-3d", chunk_index);
    for (j = 0; j < chunk_size; j++)
    {
        print_float(data[j].real);
        printf(" + ");
        print_float(data[j].imag);
        printf("i     ");
    }
    printf("\n");
}

static void perform_stft(list_t lst, int chunk_size, int hop_size)
{
    memb_init(&chunk_pool);
    memb_init(&fft_result_pool);

    int signal_length = list_length(lst);
    int num_chunks = (signal_length - chunk_size) / hop_size + 1;
    printf("signal length %d and %d chunks\n", signal_length, num_chunks);
    int i, j;
    for (i = 0; i < num_chunks; i++)
    {
        float *chunk = (float *)memb_alloc(&chunk_pool);
        if (chunk == NULL)
        {
            printf("Chunk allocation failed\n");
            return;
        }

        int start_index = i * hop_size;
        // printf("Start index: %d\n", start_index);
        for (j = 0; j < chunk_size; j++)
        {
            int signal_index = start_index + j;
            // printf("Signal index: %d\n", signal_index);
            if (signal_index < signal_length)
            {
                chunk[j] = list_get(lst, signal_index);
            }
            else
            {
                chunk[j] = 0.0f; // just in case, not needed in our case.
            }
        }
        complex_t data[chunk_size];
        for (j = 0; j < chunk_size; j++)
        {
            data[j].real = chunk[j];
            data[j].imag = 0.0f;
            /*--------DEBUGGING--------*/
            // printf("CHUNK VALUE: \n");
            // print_float(data[j].real);
            // printf("\n");
            // print_float(data[j].imag);
            // printf("\n");
            /*--------DEBUGGING--------*/
        }
        fft(data, chunk_size);

        /*--------DEBUGGING--------*/
        print_stft_results(i, data, chunk_size);
        /*--------DEBUGGING--------*/

        memb_free(&chunk_pool, chunk);
    }
}

void compute_power_spectrum(complex_t *chunk, float *power_spectrum, int chunk_size)
```

```c
{
    int k;
    for (k = 0; k < chunk_size; k++)
    {
        power_spectrum[k] = chunk[k].real * chunk[k].real +
                            chunk[k].imag * chunk[k].imag;
    }
}

static float compute_spectral_entropy(list_t lst, int chunk_size, int hop_size)
{
    int signal_length = list_length(lst);
    int num_chunks = (signal_length - chunk_size) / hop_size + 1;

    if (num_chunks <= 0)
        return 0.0f;

    // Allocate memory for power spectrum calculations
    float power_spectrum[chunk_size];
    float avg_power_spectrum[chunk_size];
    float pdf[chunk_size];

    // Initialize average power spectrum
    int i;
    for (i = 0; i < chunk_size; i++)
    {
        avg_power_spectrum[i] = 0.0f;
    }

    // Process each chunk
    for (i = 0; i < num_chunks; i++)
    {
        complex_t chunk_data[chunk_size];
        int j;
        for (j = 0; j < chunk_size; j++)
        {
            int signal_index = i * hop_size + j;
            if (signal_index < signal_length)
            {
                chunk_data[j].real = list_get(lst, signal_index);
                chunk_data[j].imag = 0.0f;
            }
            else
            {
                chunk_data[j].real = 0.0f;
                chunk_data[j].imag = 0.0f;
            }
        }

        fft(chunk_data, chunk_size);

        // Compute power spectrum for this chunk
        compute_power_spectrum(chunk_data, power_spectrum, chunk_size);

        // Add to average power spectrum
        for (j = 0; j < chunk_size; j++)
        {
            avg_power_spectrum[j] += power_spectrum[j];
        }
    }

    // Compute final average
    float total_power = 0.0f;
    for (i = 0; i < chunk_size; i++)
    {
        avg_power_spectrum[i] /= num_chunks;
        total_power += avg_power_spectrum[i];
    }

    // Compute PDF and entropy
    float entropy = 0.0f;
    if (total_power > 0.0f)
    {
        for (i = 0; i < chunk_size; i++)
```

```
        for (i = 0; i < chunk_size; i++)
        {
            if (avg_power_spectrum[i] > 0.0f)
            {
                pdf[i] = avg_power_spectrum[i] / total_power;
                entropy -= pdf[i] * logf(pdf[i]);
            }
        }
    }

    return entropy;
}

static void aggregate_and_report()
{
    struct sensor_data *light_item = list_head(light_list);
    struct sensor_data *temp_item = list_head(temp_list);
    printf("X (Light Sensor Readings) = [");
    for (light_item = list_head(light_list); light_item != NULL; light_item = list_item_next(light_item))
    {
        print_float(light_item->value);
        if (list_item_next(light_item) != NULL)
        {
            printf(", ");
        }
    }
    printf("]\n");
    printf("Y (Temperature Sensor Readings) = [");
    for (temp_item = list_head(temp_list); temp_item != NULL; temp_item = list_item_next(temp_item))
    {
        print_float(temp_item->value);
        if (list_item_next(temp_item) != NULL)
        {
            printf(", ");
        }
    }
    printf("]\n");

    printf("Manhattan Distance: ");
    print_float(calculate_manhattan_dist(light_list, temp_list));
    printf("\n");

    printf("Correlation: ");
    print_float(calculate_correlation(light_list, temp_list));
    printf("\n");

    printf("Performing STFT with ");
    perform_stft(light_list, 4, 2);
    printf("\n");

    printf("Spectral Entropy: ");
    print_float(compute_spectral_entropy(light_list, 4, 2));
    printf("\n");
}
/*---------------------------------------------------------------------------*/
PROCESS(sensor_reading_process, "Sensor reading process");
AUTOSTART_PROCESSES(&sensor_reading_process);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(sensor_reading_process, ev, data)
{
    static struct etimer timer;
    static int sample_counter = 0;
    static int k = 12; // number of samples before calculation

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

    SENSORS_ACTIVATE(light_sensor);
    SENSORS_ACTIVATE(sht11_sensor);

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

```

```
        float light = read_light_sensor();
        float temp = get_temp_sensor();
        add_sensor_data(light, light_list, &light_mem);
        add_sensor_data(temp, temp_list, &temp_mem);

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

        etimer_reset(&timer);
    }

    PROCESS_END();
}
```