```c
#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))
    {
        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();
}
```