Listings

Listing 1: Main coursework program: 2 processes for reading and aggregating data

```
#define READING_INTERVAL 2 //in Hz
#define BUFFER_SIZE 12 // length of buffer to populate

#define SD_THRESHOLD_SOME 400 // some activity, compress above, flatten below
#define SD_THRESHOLD_LOTS 1000 // lots of activity, don't aggregate

#define AGGREGATION_GROUP_SIZE 2 // group size to aggregate (4 in spec)

#define INITIAL_STATE true // whether begins running or not

#define SAX // use sax aggregation and transform instead of simple average aggregation
#define SAX_BREAKPOINTS 4 // number of characters to be used

#include "contiki.h"

#include <stdio.h> /* For printf() */
#include <stdbool.h>

#include "io.h"
#include "util.h" // for print methods
#include "math.h"
#include "buffer.h"
#include "sax.h"

static process_event_t event_buffer_full;

/*---------------------------------------------------------------------------*/
PROCESS(sensing_process, "Sensing process"); // collect data
PROCESS(aggregator_process, "Aggregator process"); // receive full data buffers for
    processing

AUTOSTART_PROCESSES(&sensing_process, &aggregator_process);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(sensing_process, ev, data)
{
    /*INIT*/
    PROCESS_BEGIN();

    static bool isRunning = INITIAL_STATE;
    static struct etimer timer;
    if(isRunning) etimer_set(&timer, CLOCK_SECOND/READING_INTERVAL); // start timer if
        running

    event_buffer_full = process_alloc_event();
    initIO();

    static Buffer buffer;
    buffer = getBuffer(BUFFER_SIZE);
    /*END INIT*/
```

```
    static int counter = 0;
    while(1)
    {
        PROCESS_WAIT_EVENT();

        if (ev == PROCESS_EVENT_TIMER){
            leds_off(LEDS_RED);

            float light_lx = getLight(); // GET

            buffer.items[counter] = light_lx; // STORE

            printf("%2i/%i: ", counter + 1, buffer.length);putFloat(light_lx);putchar('\
                n'); // DISPLAY CURRENT VALUE

            counter++;
            if(counter == buffer.length) // CHECK WHETHER FULL
            {
                process_post(&aggregator_process, event_buffer_full, &buffer); // pass
                    buffer to processing thread
                counter = 0;
                buffer = getBuffer(BUFFER_SIZE); // get new buffer for next data, no
                    freeing in this thread
            }

            etimer_reset(&timer);
        }
        /* BUTTON CLICKED */
        else if (ev == sensors_event && data == &button_sensor)
        {
            isRunning = !isRunning;
            if (isRunning == true)
            {
                printf("Starting...\n");
                etimer_set(&timer, CLOCK_SECOND/READING_INTERVAL);
            }
            else
            {
                printf("Stopping, clearing buffer...\n");
                etimer_stop(&timer);
                counter = 0; // just reset counter, used as index on buffer items, will
                    overwrite
            }
        }
    }

    PROCESS_END();
}
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(aggregator_process, ev, data)
{
    PROCESS_BEGIN();

    while (1)
    {
        PROCESS_WAIT_EVENT_UNTIL(ev == event_buffer_full);
        leds_on(LEDS_RED);
```

```
        Buffer fullBuffer = *(Buffer *)data;
        /**********************/
#ifdef SAX
        handleSAXBufferRotation(&fullBuffer);
#else
        handleSimpleBufferRotation(&fullBuffer); // pass by reference, edited if lots of
            activity
#endif
        freeBuffer(fullBuffer);
        /**********************/
    }

    PROCESS_END();
}
/*---------------------------------------------------------------------------*/
// Buffer filled with readings, process and aggregate
void
handleSimpleBufferRotation(Buffer *inBufferPtr)
{
    printf("Buffer␣full,␣aggregating\n\n");

    Buffer inBuffer = *inBufferPtr;
    Buffer outBuffer; // OUTPUT BUFFER HOLDER
    // above pointer is assigned a buffer in either of the below cases

    Stats sd = calculateStdDev(inBuffer.items, inBuffer.length); // GET BUFFER
        STATISTICS

    /* LOTS OF ACTIVITY - LEAVE */
    if(sd.std > SD_THRESHOLD_LOTS)
    {
        printf("Lots␣of␣activity,␣std.␣dev.:␣");putFloat(sd.std);printf(",␣leaving␣as-is
            \n");

        outBuffer = getBuffer(1); // get a dummy buffer, will swap items for efficiency

        swapBufferMemory(inBufferPtr, &outBuffer); // ensures both are freed but no need
            to copy items

    }
    /* SOME ACTIVITY - AGGREGATE */
    else if(sd.std > SD_THRESHOLD_SOME)
    {
        printf("Some␣activity,␣std.␣dev.:␣");putFloat(sd.std);printf(",␣compressing␣
            buffer\n");

        int outLength = ceil((float)inBuffer.length/AGGREGATION_GROUP_SIZE); //
            CALCULATE NUMBER OF OUTPUT ELEMENTS
        outBuffer = getBuffer(outLength); // CREATE OUTPUT BUFFER

        aggregateBuffer(inBuffer, outBuffer, AGGREGATION_GROUP_SIZE);

    }
    /* NO ACTIVITY - FLATTEN */
    else
    {
        printf("Insignificant␣std.␣dev.:␣");putFloat(sd.std);printf(",␣squashing␣buffer\
            n");
```

```
        outBuffer = getBuffer(1); // CREATE OUTPUT BUFFER

        outBuffer.items[0] = sd.mean;
    }
    outBuffer.stats = sd; // final compressed buffer has stats for uncompressed data in
        case of further interest

    /**********************/
    handleFinalBuffer(outBuffer); // PASS FINAL BUFFER
    freeBuffer(outBuffer); // RELEASE ITEMS
    /**********************/
}

void
handleSAXBufferRotation(Buffer *inBufferPtr)
{
    printf("Buffer␣full,␣SAX-ing\n\n");

    Buffer inBuffer = *inBufferPtr;
    Buffer outBuffer; // OUTPUT BUFFER HOLDER
    // above pointer is assigned a buffer in either of the below cases

    int outLength = ceil((float)inBuffer.length/AGGREGATION_GROUP_SIZE); // CALCULATE
        NUMBER OF OUTPUT ELEMENTS
    outBuffer = getBuffer(outLength); // CREATE OUTPUT BUFFER

    inBuffer.stats = calculateStdDev(inBuffer.items, inBuffer.length); // GET BUFFER
        STATISTICS
    outBuffer.stats = inBuffer.stats;

    normaliseBuffer(inBuffer); // Z NORMALISATION
    aggregateBuffer(inBuffer, outBuffer, AGGREGATION_GROUP_SIZE); // PAA

    /**********************/
    handleFinalBuffer(outBuffer); // PASS FINAL BUFFER
    freeBuffer(outBuffer); // RELEASE ITEMS
    /**********************/
}

// Process final buffer following aggregation
void
handleFinalBuffer(Buffer buffer)
{
    printf("Final␣buffer␣output:␣");
    printBuffer(buffer);putchar('\n');
    printf("Mean:␣");putFloat(buffer.stats.mean);putchar('\n');
    printf("Std␣Dev:␣");putFloat(buffer.stats.std);putchar('\n');putchar('\n');

#ifdef SAX
    char* saxString = stringifyBuffer(buffer);
    printf("SAX:␣%s\n\n", saxString);

    free(saxString);
#endif
}
/*----------------------------------------------------------------------------*/
```

Listing 2: Buffer header file: get, free and manipulate buffers

```
#ifndef _BUFFER_GUARD
#define _BUFFER_GUARD

#include "util.h"
#include "math.h"

typedef struct Buffer {
    float* items;
    int length;
    Stats stats;
} Buffer;

Buffer
getBuffer(int size) // retrieve new buffer with malloc-ed memory space
{
    float* memSpace = (float*) malloc(size * sizeof(float));
    Buffer buffer = {memSpace, size, };
    return buffer;
}

void
freeBuffer(Buffer buffer) // little abstraction function to act as buffer destructor
{
    free(buffer.items);
}

void
swapBufferMemory(Buffer *first, Buffer *second) // swap memspaces between buffers
{
    float* firstItems = first->items; // swap input buffer and output buffer item
       pointers
    first->items = second->items;
    second->items = firstItems;

    int firstLength = first->length; // swap lengths to iterate correctly
    first->length = second->length;
    second->length = firstLength;
}

void // perform aggregation into groupSize (4 in the spec)
aggregateBuffer(Buffer bufferIn, Buffer bufferOut, int groupSize)
{
    int requiredGroups = ceil((float)bufferIn.length/groupSize); // number of groups
    int finalGroupSize = (bufferIn.length % groupSize) * groupSize; // work out length
       of final group if bufferIn not of length that divides nicely

    if(requiredGroups > bufferOut.length) // error check
    {
        putFloat((float)bufferIn.length/groupSize); printf("␣length␣out␣buffer␣required,
           ␣%i␣provided\n", bufferOut.length);
        return;
    }

    int g; // for group number
    float *inputPtr = bufferIn.items; // cursor for full buffer
    float *outputPtr = bufferOut.items; // cursor for output buffer
    for(g = 0; g < requiredGroups; g++)
    {
```

```
        int length = groupSize; // length of this group's size
        if(g == requiredGroups - 1 && finalGroupSize != 0) length = finalGroupSize; //
            shorten if necessary

        *outputPtr = calculateMean(inputPtr, length); // SET OUTPUT VALUE

        inputPtr += length; // increment both cursors
        outputPtr++;
    }
}

void
clearBuffer(Buffer buffer)
{
    int length = buffer.length;
    if(length > 0)
    {
        int i;
        float *bufferPtr = buffer.items;
        for(i = 0; i < length; i++)
        {
            *bufferPtr = 0.0;
            bufferPtr++;
        }
    }
}

void
printBuffer(Buffer buffer)
{
    putchar('[');

    int length = buffer.length;
    if(length > 0)
    {
        int i;
        float *ptr = buffer.items;
        for(i = 0; i < length; i++)
        {
            if(i > 0) printf(",␣");

            putFloat(*ptr);
            ptr++;
        }
    }

    putchar(']');
}

#endif
```

Listing 3: SAX header file: z-normalise and stringify buffers

```
#ifndef _SAX_GUARD
#define _SAX_GUARD

#define SAX_CHAR_START 'a'

#ifndef SAX_BREAKPOINTS
    #define SAX_BREAKPOINTS 4
#endif

// Could have used a 2d array for breakpoints, index by number of breakpoints

// Since the number of boundaries is known at compile-time, save these lookup calls by
// defining as as constant 1D arrays

#if SAX_BREAKPOINTS == 3
    const float breakPoints[] = {-0.43, 0.43};
#elif SAX_BREAKPOINTS == 4
    const float breakPoints[] = {-0.67, 0, 0.67};
#elif SAX_BREAKPOINTS == 5
    const float breakPoints[] = {-0.84, -0.25, 0.25, 0.84};
#elif SAX_BREAKPOINTS == 6
    const float breakPoints[] = {-0.97, -0.43, 0, 0.43, 0.97};
#elif SAX_BREAKPOINTS == 7
    const float breakPoints[] = {-1.07, -0.57, -0.18, 0.18, 0.57, 1.07};
#elif SAX_BREAKPOINTS == 8
    const float breakPoints[] = {-1.15, -0.67, -0.32, 0, 0.32, 0.67, 1.15};
#elif SAX_BREAKPOINTS == 9
    const float breakPoints[] = {-1.22, -0.76, -0.43, -0.14, 0.14, 0.43, 0.76, 1.22};
#elif SAX_BREAKPOINTS == 10
    const float breakPoints[] = {-1.28, -0.84, -0.52, -0.25, 0, 0.25, 0.52, 0.84, 1.28};
#else
    #define SAX_BREAKPOINTS 4
    const float breakPoints[] = {-0.67, 0, 0.67};
#endif

void
normaliseBuffer(Buffer bufferIn) // z normalise buffer for SAX
{
    if(bufferIn.stats.std == 0) // error check, don't divide by 0
    {
        printf("Standard␣deviation␣of␣zero,␣unable␣to␣normalise\n");
        return;
    }

    int i;
    float *inputPtr = bufferIn.items; // cursor
    for(i = 0; i < bufferIn.length; i++)
    {
        *inputPtr = (*inputPtr - bufferIn.stats.mean) / bufferIn.stats.std;

        inputPtr++;
    }
}

char
valueToSAXChar(float inputValue)
{
    int i;
```

```
    for(i = 0; i < SAX_BREAKPOINTS; i++)
    {
        if(i == 0) // first iter, is less than first breakpoint
        {
            if(inputValue < breakPoints[i]) return SAX_CHAR_START + i;
        }
        else if(i == SAX_BREAKPOINTS - 1) // last iter, is more than last breakpoint
        {
            if(breakPoints[i - 1] < inputValue) return SAX_CHAR_START + i;
        }
        else // in between check interval of two breakpoints
        {
            if((breakPoints[i - 1] < inputValue) && (inputValue  < breakPoints[i]))
               return SAX_CHAR_START + i;
        }
    }
    return '0';
}

char* // map buffer of normalised floats into SAX chars
stringifyBuffer(Buffer bufferIn)
{
    char* outputString = (char*) malloc((bufferIn.length + 1) * sizeof(char)); // +1 for
        null terminator

    int i;
    for(i = 0; i < bufferIn.length; i++)
    {
        outputString[i] = valueToSAXChar(bufferIn.items[i]);
    }

    outputString[bufferIn.length] = '\0'; // add null terminator
    return outputString;
}

#endif
```

Listing 4: Math header file: mean, standard deviation, implementations of ceil, sqrt

```
#ifndef _MATH_GUARD
#define _MATH_GUARD

typedef struct Stats {
    float mean;
    float std;
} Stats;

int
ceil(float in) // self-implement ceil func, no math.h
{
    int num = (int) in;
    if(in - num > 0) num++;
    return num;
}

float
sqrt(float in) // self-implement ceil sqrt, no math.h
{
    float sqrt = in/2;
    float temp = 0;

    while(sqrt != temp)
    {
        temp = sqrt;
        sqrt = (in/temp + temp) / 2;
    }
    return sqrt;
}

float
calculateMean(float buffer[], int length)
{
    if(length <= 0)
    {
        printf("%i␣items␣is␣not␣valid␣length\n", length);
        return 0;
    }

    /* SUM */
    float sum = 0;
    int i;
    for(i = 0; i < length; i++)
    {
        sum += buffer[i];
    }

    return sum / length; // DIVIDE ON RETURN
}

Stats
calculateStdDev(float buffer[], int length)
{
    Stats stats;
    if(length <= 0)
    {
        printf("%i␣items␣is␣not␣valid␣length\n", length);
        return stats;
```

```
    }

    stats.mean = calculateMean(buffer, length);

    float sum = 0;
    int i;
    for(i = 0; i < length; i++)
    {
        float diffFromMean = buffer[i] - stats.mean; // (xi - mu)
        sum += diffFromMean*diffFromMean; // Sum(diff squared)
    }

    stats.std = sqrt(sum/length);

    return stats;
}

#endif
```

Listing 5: IO header file: init function for starting sensors

```
#ifndef _IO_GUARD
#define _IO_GUARD

#include "dev/light-sensor.h"
#include "dev/button-sensor.h"
#include "dev/leds.h"

void
initIO()
{
    SENSORS_ACTIVATE(light_sensor);
    SENSORS_ACTIVATE(button_sensor);
    leds_off(LEDS_ALL);
}

// get float from light sensor including transfer function
float
getLight(void)
{
    int lightData = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);

    float V_sensor = 1.5 * lightData / 4096;
    float I = V_sensor/1e5;
    float light = 0.625 * 1e6 * I * 1000;
    return light;
}

#endif
```

Listing 6: Other utilities: short and float printing functions from earlier labs

```
#ifndef _UTIL_GUARD
#define _UTIL_GUARD

typedef unsigned short USHORT;

//print a unsigned short picewise char by char
void
putShort(USHORT in)
{
    // recursively shift each digit of the int to units from most to least significant
    if (in >= 10)
    {
        putShort(in / 10);
    }
    // isolate unit digit from each number by modulo and add '0' char to turn integer
       into corresponding ascii char
    putchar((in % 10) + '0');
}

void
putFloat(float in)
{
    if(in < 0)
    {
        putchar('-'); // print negative sign if required
        in = -in;
    }

    USHORT integerComponent = (USHORT) in; // truncate float to integer
    float fractionComponent =  (in - integerComponent) * 1000; // take fraction only and
        promote to integer
    if (fractionComponent - (USHORT)fractionComponent >= 0.5) fractionComponent++; //
       round

    putShort(integerComponent);
    putchar('.');
    putShort((USHORT) fractionComponent);
}

#endif
```