/\* Edge Impulse ingestion SDK

 \* Copyright (c) 2022 EdgeImpulse Inc.

 \*

 \* Licensed under the Apache License, Version 2.0 (the "License");

 \* you may not use this file except in compliance with the License.

 \* You may obtain a copy of the License at

 \* http://www.apache.org/licenses/LICENSE-2.0

 \*

 \* Unless required by applicable law or agreed to in writing, software

 \* distributed under the License is distributed on an "AS IS" BASIS,

 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

 \* See the License for the specific language governing permissions and

 \* limitations under the License.

 \*

 \*/

/\* Includes ---------------------------------------------------------------- \*/

#include <FitTrack\_inferencing.h>

#include <Arduino\_LSM9DS1.h> //Click here to get the library: https://www.arduino.cc/reference/en/libraries/arduino\_lsm9ds1/

#include <Arduino\_LPS22HB.h> //Click here to get the library: https://www.arduino.cc/reference/en/libraries/arduino\_lps22hb/

#include <Arduino\_HTS221.h> //Click here to get the library: https://www.arduino.cc/reference/en/libraries/arduino\_hts221/

#include <Arduino\_APDS9960.h> //Click here to get the library: https://www.arduino.cc/reference/en/libraries/arduino\_apds9960/

enum sensor\_status {

    NOT\_USED = -1,

    NOT\_INIT,

    INIT,

    SAMPLED

};

/\*\* Struct to link sensor axis name to sensor value function \*/

typedef struct{

    const char \*name;

    float \*value;

    uint8\_t (\*poll\_sensor)(void);

    bool (\*init\_sensor)(void);

    sensor\_status status;

} eiSensors;

/\* Constant defines -------------------------------------------------------- \*/

#define CONVERT\_G\_TO\_MS2    9.80665f

#define MAX\_ACCEPTED\_RANGE  2.0f        // starting 03/2022, models are generated setting range to +-2,

                                        // but this example use Arudino library which set range to +-4g.

                                        // If you are using an older model, ignore this value and use 4.0f instead

/\*\* Number sensor axes used \*/

#define N\_SENSORS     18

/\* Forward declarations ------------------------------------------------------- \*/

float ei\_get\_sign(float number);

bool init\_IMU(void);

bool init\_HTS(void);

bool init\_BARO(void);

bool init\_APDS(void);

uint8\_t poll\_acc(void);

uint8\_t poll\_gyr(void);

uint8\_t poll\_mag(void);

uint8\_t poll\_HTS(void);

uint8\_t poll\_BARO(void);

uint8\_t poll\_APDS\_color(void);

uint8\_t poll\_APDS\_proximity(void);

uint8\_t poll\_APDS\_gesture(void);

/\* Private variables ------------------------------------------------------- \*/

static const bool debug\_nn = false; // Set this to true to see e.g. features generated from the raw signal

static float data[N\_SENSORS];

static bool ei\_connect\_fusion\_list(const char \*input\_list);

static int8\_t fusion\_sensors[N\_SENSORS];

static int fusion\_ix = 0;

/\*\* Used sensors value function connected to label name \*/

eiSensors sensors[] =

{

    "accX", &data[0], &poll\_acc, &init\_IMU, NOT\_USED,

    "accY", &data[1], &poll\_acc, &init\_IMU, NOT\_USED,

    "accZ", &data[2], &poll\_acc, &init\_IMU, NOT\_USED,

    "gyrX", &data[3], &poll\_gyr, &init\_IMU, NOT\_USED,

    "gyrY", &data[4], &poll\_gyr, &init\_IMU, NOT\_USED,

    "gyrZ", &data[5], &poll\_gyr, &init\_IMU, NOT\_USED,

    "magX", &data[6], &poll\_mag, &init\_IMU, NOT\_USED,

    "magY", &data[7], &poll\_mag, &init\_IMU, NOT\_USED,

    "magZ", &data[8], &poll\_mag, &init\_IMU, NOT\_USED,

    "temperature", &data[9], &poll\_HTS, &init\_HTS, NOT\_USED,

    "humidity", &data[10], &poll\_HTS, &init\_HTS, NOT\_USED,

    "pressure", &data[11], &poll\_BARO, &init\_BARO, NOT\_USED,

    "red", &data[12], &poll\_APDS\_color, &init\_APDS, NOT\_USED,

    "green", &data[13], &poll\_APDS\_color, &init\_APDS, NOT\_USED,

    "blue", &data[14], &poll\_APDS\_color, &init\_APDS, NOT\_USED,

    "brightness", &data[15], &poll\_APDS\_color, &init\_APDS, NOT\_USED,

    "proximity", &data[16], &poll\_APDS\_proximity, &init\_APDS, NOT\_USED,

    "gesture", &data[17], &poll\_APDS\_gesture,&init\_APDS, NOT\_USED,

};

/\*\*

\* @brief      Arduino setup function

\*/

void setup()

{

    /\* Init serial \*/

    Serial.begin(115200);

    // comment out the below line to cancel the wait for USB connection (needed for native USB)

    while (!Serial);

    Serial.println("Edge Impulse Sensor Fusion Inference\r\n");

    /\* Connect used sensors \*/

    if(ei\_connect\_fusion\_list(EI\_CLASSIFIER\_FUSION\_AXES\_STRING) == false) {

        ei\_printf("ERR: Errors in sensor list detected\r\n");

        return;

    }

    /\* Init & start sensors \*/

    for(int i = 0; i < fusion\_ix; i++) {

        if (sensors[fusion\_sensors[i]].status == NOT\_INIT) {

            sensors[fusion\_sensors[i]].status = (sensor\_status)sensors[fusion\_sensors[i]].init\_sensor();

            if (!sensors[fusion\_sensors[i]].status) {

              ei\_printf("%s axis sensor initialization failed.\r\n", sensors[fusion\_sensors[i]].name);

            }

            else {

              ei\_printf("%s axis sensor initialization successful.\r\n", sensors[fusion\_sensors[i]].name);

            }

        }

    }

}

/\*\*

\* @brief      Get data and run inferencing

\*/

void loop()

{

    ei\_printf("\nStarting inferencing in 2 seconds...\r\n");

    delay(2000);

    if (EI\_CLASSIFIER\_RAW\_SAMPLES\_PER\_FRAME != fusion\_ix) {

        ei\_printf("ERR: Sensors don't match the sensors required in the model\r\n"

        "Following sensors are required: %s\r\n", EI\_CLASSIFIER\_FUSION\_AXES\_STRING);

        return;

    }

    ei\_printf("Sampling...\r\n");

    // Allocate a buffer here for the values we'll read from the sensor

    float buffer[EI\_CLASSIFIER\_DSP\_INPUT\_FRAME\_SIZE] = { 0 };

    for (size\_t ix = 0; ix < EI\_CLASSIFIER\_DSP\_INPUT\_FRAME\_SIZE; ix += EI\_CLASSIFIER\_RAW\_SAMPLES\_PER\_FRAME) {

        // Determine the next tick (and then sleep later)

        int64\_t next\_tick = (int64\_t)micros() + ((int64\_t)EI\_CLASSIFIER\_INTERVAL\_MS \* 1000);

        for(int i = 0; i < fusion\_ix; i++) {

            if (sensors[fusion\_sensors[i]].status == INIT) {

                sensors[fusion\_sensors[i]].poll\_sensor();

                sensors[fusion\_sensors[i]].status = SAMPLED;

            }

            if (sensors[fusion\_sensors[i]].status == SAMPLED) {

                buffer[ix + i] = \*sensors[fusion\_sensors[i]].value;

                sensors[fusion\_sensors[i]].status = INIT;

            }

        }

        int64\_t wait\_time = next\_tick - (int64\_t)micros();

        if(wait\_time > 0) {

            delayMicroseconds(wait\_time);

        }

    }

    // Turn the raw buffer in a signal which we can the classify

    signal\_t signal;

    int err = numpy::signal\_from\_buffer(buffer, EI\_CLASSIFIER\_DSP\_INPUT\_FRAME\_SIZE, &signal);

    if (err != 0) {

        ei\_printf("ERR:(%d)\r\n", err);

        return;

    }

    // Run the classifier

    ei\_impulse\_result\_t result = { 0 };

    err = run\_classifier(&signal, &result, debug\_nn);

    if (err != EI\_IMPULSE\_OK) {

        ei\_printf("ERR:(%d)\r\n", err);

        return;

    }

    // print the predictions

    ei\_printf("Predictions (DSP: %d ms., Classification: %d ms., Anomaly: %d ms.):\r\n",

        result.timing.dsp, result.timing.classification, result.timing.anomaly);

    for (size\_t ix = 0; ix < EI\_CLASSIFIER\_LABEL\_COUNT; ix++) {

        ei\_printf("%s: %.5f\r\n", result.classification[ix].label, result.classification[ix].value);

    }

#if EI\_CLASSIFIER\_HAS\_ANOMALY == 1

    ei\_printf("    anomaly score: %.3f\r\n", result.anomaly);

#endif

}

#if !defined(EI\_CLASSIFIER\_SENSOR) || (EI\_CLASSIFIER\_SENSOR != EI\_CLASSIFIER\_SENSOR\_FUSION && EI\_CLASSIFIER\_SENSOR != EI\_CLASSIFIER\_SENSOR\_ACCELEROMETER)

#error "Invalid model for current sensor"

#endif

/\*\*

 \* @brief Go through sensor list to find matching axis name

 \*

 \* @param axis\_name

 \* @return int8\_t index in sensor list, -1 if axis name is not found

 \*/

static int8\_t ei\_find\_axis(char \*axis\_name)

{

    int ix;

    for(ix = 0; ix < N\_SENSORS; ix++) {

        if(strstr(axis\_name, sensors[ix].name)) {

            return ix;

        }

    }

    return -1;

}

/\*\*

 \* @brief Check if requested input list is valid sensor fusion, create sensor buffer

 \*

 \* @param[in]  input\_list      Axes list to sample (ie. "accX + gyrY + magZ")

 \* @retval  false if invalid sensor\_list

 \*/

static bool ei\_connect\_fusion\_list(const char \*input\_list)

{

    char \*buff;

    bool is\_fusion = false;

    /\* Copy const string in heap mem \*/

    char \*input\_string = (char \*)ei\_malloc(strlen(input\_list) + 1);

    if (input\_string == NULL) {

        return false;

    }

    memset(input\_string, 0, strlen(input\_list) + 1);

    strncpy(input\_string, input\_list, strlen(input\_list));

    /\* Clear fusion sensor list \*/

    memset(fusion\_sensors, 0, N\_SENSORS);

    fusion\_ix = 0;

    buff = strtok(input\_string, "+");

    while (buff != NULL) { /\* Run through buffer \*/

        int8\_t found\_axis = 0;

        is\_fusion = false;

        found\_axis = ei\_find\_axis(buff);

        if(found\_axis >= 0) {

            if(fusion\_ix < N\_SENSORS) {

                fusion\_sensors[fusion\_ix++] = found\_axis;

                sensors[found\_axis].status = NOT\_INIT;

            }

            is\_fusion = true;

        }

        buff = strtok(NULL, "+ ");

    }

    ei\_free(input\_string);

    return is\_fusion;

}

/\*\*

 \* @brief Return the sign of the number

 \*

 \* @param number

 \* @return int 1 if positive (or 0) -1 if negative

 \*/

float ei\_get\_sign(float number) {

    return (number >= 0.0) ? 1.0 : -1.0;

}

bool init\_IMU(void) {

  static bool init\_status = false;

  if (!init\_status) {

    init\_status = IMU.begin();

  }

  return init\_status;

}

bool init\_HTS(void) {

  static bool init\_status = false;

  if (!init\_status) {

    init\_status = HTS.begin();

  }

  return init\_status;

}

bool init\_BARO(void) {

  static bool init\_status = false;

  if (!init\_status) {

    init\_status = BARO.begin();

  }

  return init\_status;

}

bool init\_APDS(void) {

  static bool init\_status = false;

  if (!init\_status) {

    init\_status = APDS.begin();

  }

  return init\_status;

}

uint8\_t poll\_acc(void) {

    if (IMU.accelerationAvailable()) {

    IMU.readAcceleration(data[0], data[1], data[2]);

    for (int i = 0; i < 3; i++) {

        if (fabs(data[i]) > MAX\_ACCEPTED\_RANGE) {

            data[i] = ei\_get\_sign(data[i]) \* MAX\_ACCEPTED\_RANGE;

        }

    }

    data[0] \*= CONVERT\_G\_TO\_MS2;

    data[1] \*= CONVERT\_G\_TO\_MS2;

    data[2] \*= CONVERT\_G\_TO\_MS2;

    }

    return 0;

}

uint8\_t poll\_gyr(void) {

    if (IMU.gyroscopeAvailable()) {

        IMU.readGyroscope(data[3], data[4], data[5]);

    }

    return 0;

}

uint8\_t poll\_mag(void) {

    if (IMU.magneticFieldAvailable()) {

        IMU.readMagneticField(data[6], data[7], data[8]);

    }

    return 0;

}

uint8\_t poll\_HTS(void) {

    data[9] = HTS.readTemperature();

    data[10] = HTS.readHumidity();

    return 0;

}

uint8\_t poll\_BARO(void) {

    data[11] = BARO.readPressure(); // (PSI/MILLIBAR/KILOPASCAL) default kPa

    return 0;

}

uint8\_t poll\_APDS\_color(void) {

    int temp\_data[4];

    if (APDS.colorAvailable()) {

        APDS.readColor(temp\_data[0], temp\_data[1], temp\_data[2], temp\_data[3]);

        data[12] = temp\_data[0];

        data[13] = temp\_data[1];

        data[14] = temp\_data[2];

        data[15] = temp\_data[3];

    }

}

uint8\_t poll\_APDS\_proximity(void) {

    if (APDS.proximityAvailable()) {

        data[16] = (float)APDS.readProximity();

    }

    return 0;

}

uint8\_t poll\_APDS\_gesture(void) {

    if (APDS.gestureAvailable()) {

        data[17] = (float)APDS.readGesture();

    }

    return 0;

}