**ADS task**

**static** **void** **adsTask**(**void** \*pvParameter)

{

ESP\_LOGI(TAG, "[ADS Task] Initializing ADS Task on Core: %d", **xPortGetCoreID**());

**char** WeightString[15];

**double** previous\_weight = 0;

**double** weight\_difference = 0;

**int** adc\_prev\_speed = adc\_speed;

ADS1232(ADS\_PDWN\_PIN, ADS\_SCLK\_PIN, ADS\_DOUT\_PIN, 0, ADS\_SPEED\_PIN, 0,0,0);

begin(0, 128, adc\_speed);

uint32\_t timeout = millis() + 1000;

**while** (timeout < millis())

{

useless\_var = readRaw(1);

//yield();

taskYIELD();

}

calibrateADC();

timeout = millis() + 1000;

**while** (timeout < millis())

{

useless\_var = readRaw(1);

//yield();

taskYIELD();

}

tare\_value = readRaw(10);

ESP\_LOGI("[ADS Task]","ADS Initialization Complete");

ADSInitialized = true;

delay(500);

uint64\_t ads\_start\_time = 0;

uint64\_t ads\_end\_time = 0;

**while** (1)

{

**if** (graphicsInitialized)

{

**if**(adc\_prev\_speed != adc\_speed)

{

setSpeed(adc\_speed, false);

adc\_prev\_speed = adc\_speed;

}

ads\_start\_time = millis();

//printf("\n[ADS Task]: ");

cal\_factor = cal\_factor2;

newvalue = readRaw(ScaleAverage);

//newvalue = readRaw(1);

tared\_value = newvalue - tare\_value;

weight = (**double**) tared\_value / (**double**) cal\_factor;

mlt = **powf**(10.0f, 2);

weight = **roundf**(weight \* mlt) / mlt;

//Stabilizeweight = weight;

weight\_difference = weight - previous\_weight;

**printf**("\n[ADS] Weight: %.2f Difference %.2f", weight, weight\_difference);

previous\_weight = weight;

**if**(calculate\_average)

{

sum\_averages = sum\_averages + weight\_difference;

average\_counter++;

}

oldweight = weight;

// sprintf(WeightString, "%.2f", Stabilizeweight); //CHANGE THIS TO DISPLAY

// lv\_label\_set\_text(ui\_PmWeightLabel, (const char \*) WeightString);

// lv\_label\_set\_text(ui\_SsWeightLabel, (const char \*) WeightString);

}

ads\_end\_time = millis();

//printf("\nADS TASK TIME: %llu \n" , ads\_end\_time-ads\_start\_time );

//ESP\_LOGI(TAG, "[ADS Task] ");

**vTaskDelay**(ads\_task\_delay / portTICK\_PERIOD\_MS);

}

**ADS1232.H (library file)**

**#ifndef** ADS1232\_h

**#define** ADS1232\_h

**#include** "driver/gpio.h"

**#include** "freertos/FreeRTOS.h"

**#include** "freertos/task.h"

//For 80SPS, 20 samples (SPS/4) are enough for impressive +/-0.02g stability (lab conditions) and extremely fast response. From 100g to 0 you will need 20\*12,5ms = 250ms which is barely noticable.

//10SPS is slow by default so be careful when enabling smoothing because you will lose responsiveness in sudden weight changes.

//For example, if you suddenly remove 100g from the scale and you have DATA\_SET\_MIN sampling with 10sps you will need 5 samples to go to 0 = 0.5s which is noticable by the user.

//Please note that the sample size will not compensate for load cell drift.

**#define** DATA\_SET\_MAX 22 // best for 80SPS // 20 samples + 1 ignore low + 1 ignore high

**#define** DATA\_SET\_MIN 5 // best for 10SPS // 3 samples + 1 ignore low + 1 ignore high

////////////////////////////////////////////////////////////////////////////////////////////////////

**#define** pinMode(x,y) gpio\_set\_direction(x,y)

**#define** digitalWrite(x,y) gpio\_set\_level(x, y)

**#define** digitalRead(x) gpio\_get\_level(x)

//#define delayMicroseconds(x) DWT\_Delay(x);

//#define pinMode(x,y)

**#define** HIGH 1

**#define** LOW 0

**#define** byte uint8\_t

**#define** yield() taskYIELD() //vTaskDelay(1000 / portTICK\_PERIOD\_MS);

**#define** delay(x) vTaskDelay(x / portTICK\_PERIOD\_MS);

**#define** millis() (esp\_timer\_get\_time()/1000ULL)

**void** **ADS1232**(uint16\_t pdwn, uint16\_t sclk, uint16\_t dout, uint16\_t a0, uint16\_t spd, uint16\_t gain1, uint16\_t gain0, uint16\_t temp);

**void** **begin**(uint8\_t channel, uint8\_t gain, uint8\_t speed);

**void** **setGain**(uint8\_t gain, **bool** calibrate); // 1/2/64/128 , default 128

**void** **setSpeed**(uint8\_t sps, **bool** calibrate); // 10 or 80 sps , default 10

uint8\_t **getSpeed**(); // 10 or 80 sps , default 10

**void** **setChannel**(uint8\_t channel, **bool** calibrate); // 0 or 1 , default = 0

**void** **powerOn**();

**void** **powerOff**();

**void** **calibrateADC**(); //this is the internal calibration method of the ADC , not the calculation of the calFactor

**void** **tare**(byte type, **bool** calibrate); //type 0 = read one value & tare, 1= tare using last value, 2=read 5 samples and tare and also optionally calibrate

**void** **setCalFactor**(**float** cal);

**void** **setMinDiff**(int32\_t diff, int32\_t threshold); // this is a useful trick to keep the scale stable when around zero (if grams are around 0). This is applied after read raw and you should select proper value for your load cell.

**void** **setSmoothing**(**bool** enable); //if enabled, everything is put in an table and at every read the largest and the lowest value is removed. The "value" returned is then the samples/table size.

**bool** **getSmoothing**();

**void** **setDataSetSize**(uint8\_t datasetsize);

int32\_t **readRaw**(uint8\_t samples); // if smoothing is disabled, returns an average of samples chosen, if smoothing is enabled reads samples chosen and returns the smoothed value.

**double** **readUnits**(uint8\_t samples); // returns a final weight value (tareOffset + calFactor taken into account) after readRaw(samples) with/without smoothing.

// uint8\_t shiftInSlow(uint8\_t dataPin, uint8\_t clockPin, uint8\_t bitOrder);

**void** **initConfig**();

**bool** **isReady**(); // checks the DOUT pin

**bool** **safeWait**(); // returns false if waiting more than 1000 millis

int32\_t **readADC**(); //gets and returns a single ADC value without any processing

**void** **resetSmoothing**(int32\_t value); //replaces all values of the smoothing dataset array with value

int32\_t **getSmoothedValue**();

**#endif**

**ADS1232.c (Library file)**

**#include** <stdio.h>

**#include** "ADS1232.h"

**#include** "driver/gpio.h"

**#include** <stdbool.h>

**#include** "esp\_log.h"

//#include "esp\_timer.h"

**#include** "rom/ets\_sys.h"

//#define ADS\_SERIAL\_DEBUG

**#if** defined ADS\_SERIAL\_DEBUG

**#define** LOG\_PRINT(x) ESP\_LOGI("[ADS\_LOG]", x)

**#define** LOG\_PRINTLN(x) ESP\_LOGI("[ADS\_LOG]", "%s", x)

**#define** DEBUG\_PRINT(x) ESP\_LOGI("[ADS\_DEBUG]", x)

**#define** DEBUG\_PRINTLN(x) ESP\_LOGI("[ADS\_DEBUG]", "%s", x)

**#define** DEBUG\_PRINTV(x) ESP\_LOGI("[ADS\_DEBUG]", "%d", x)

**#define** DEBUG\_PRINTLNV(x) ESP\_LOGI("[ADS\_DEBUG]", "%d", x)

**#else**

**#define** LOG\_PRINT(x)

**#define** LOG\_PRINTLN(x)

**#define** DEBUG\_PRINT(x)

**#define** DEBUG\_PRINTLN(x)

**#define** DEBUG\_PRINTV(x)

**#define** DEBUG\_PRINTLNV(x)

**#endif**

//#define pinMode(x,y) gpio\_set\_direction(x,y)

//#define digitalWrite(x,y) gpio\_set\_level(x,y)

//#define digitalRead(x) gpio\_get\_level(x)

**#define** OUTPUT GPIO\_MODE\_OUTPUT

**#define** INPUT\_PULLUP GPIO\_MODE\_INPUT

**#define** INPUT GPIO\_MODE\_INPUT

/////////////////////////////////////////////////////////////////////////////////////////////////////

/// FUNCTION PROTOTYPES

/////////////////////////////////////////////////////////////////////////////////////////////////////

**void** **ADS1232**(uint16\_t pdwn, uint16\_t sclk, uint16\_t dout, uint16\_t a0, uint16\_t spd, uint16\_t gain1, uint16\_t gain0, uint16\_t temp);

**void** **begin**(uint8\_t channel, uint8\_t gain, uint8\_t speed);

**void** **setGain**(uint8\_t gain, bool calibrate); // 1/2/64/128 , default 128

**void** **setSpeed**(uint8\_t sps, bool calibrate); // 10 or 80 sps , default 10

uint8\_t **getSpeed**(); // 10 or 80 sps , default 10

**void** **setChannel**(uint8\_t channel, bool calibrate); // 0 or 1 , default = 0

**void** **powerOn**();

**void** **powerOff**();

**void** **calibrateADC**(); //this is the internal calibration method of the ADC , not the calculation of the calFactor

**void** **tare**(byte type, bool calibrate); //type 0 = read one value & tare, 1= tare using last value, 2=read 5 samples and tare and also optionally calibrate

**void** **setCalFactor**(**float** cal);

**void** **setMinDiff**(int32\_t diff, int32\_t threshold); // this is a useful trick to keep the scale stable when around zero (if grams are around 0). This is applied after read raw and you should select proper value for your load cell.

**void** **setSmoothing**(bool enable); //if enabled, everything is put in an table and at every read the largest and the lowest value is removed. The "value" returned is then the samples/table size.

bool **getSmoothing**();

**void** **setDataSetSize**(uint8\_t datasetsize);

int32\_t **readRaw**(uint8\_t samples); // if smoothing is disabled, returns an average of samples chosen, if smoothing is enabled reads samples chosen and returns the smoothed value.

**double** **readUnits**(uint8\_t samples); // returns a final weight value (tareOffset + calFactor taken into account) after readRaw(samples) with/without smoothing.

// uint8\_t shiftInSlow(uint8\_t dataPin, uint8\_t clockPin, uint8\_t bitOrder);

**void** **initConfig**();

bool **isReady**(); // checks the DOUT pin

bool **safeWait**(); // returns false if waiting more than 1000 millis

int32\_t **readADC**(); //gets and returns a single ADC value without any processing

**void** **resetSmoothing**(int32\_t value); //replaces all values of the smoothing dataset array with value

int32\_t **getSmoothedValue**();

**void** **delayMicroseconds**(uint64\_t number\_of\_us);

/////////////////////////////////////////////////////////////////////////////////////////////////////

/// GLOBAL VARIABLES

/////////////////////////////////////////////////////////////////////////////////////////////////////

uint16\_t pdwnPin; // keep HIGH for power on, LOW for power off

uint16\_t sclkPin; // not regular SPI, read datasheet

uint16\_t doutPin; // not regular SPI, read datasheet

uint16\_t spdPin; // LOW = 10SPS , HIGH = 80SPS

uint16\_t gain1Pin; // 0|0 = 1 , 0|1 = 2 , 1|0 = 64 , 1|1 = 128

uint16\_t gain0Pin; // in our case, anything lower than 128 is not worth it. Keep both gain0/gain1 HIGH

uint16\_t tempPin; // if HIGH, AINP/AINN can be used to measure temp from internal temp diodes (datasheet, page 13)

uint16\_t a0Pin; //Select output, LOW for AIN1.

uint8\_t adcSpeed;

uint8\_t adcGain;

uint8\_t adcTemp;

uint8\_t adcChannel;

uint8\_t readsPerSecond = 0;

**int** lastRateCheck = 0; //millis before last calculation of readspersecond

int32\_t tareOffset;

bool smoothing;

**int** readIndex = 0;

int32\_t dataSampleSet[DATA\_SET\_MAX+1];

int32\_t lastAdcValue; //this is the absolute last adcvalue we read

**float** ignoreDiff;

**float** ignoreDiffThreshold;

**float** calFactor; // this is the number that will help us translate voltage read from the ADC to units (grams,whatever).

uint8\_t DATA\_SET = DATA\_SET\_MIN; //DATA\_SET can be set at any time up to DATA\_SET\_MAX.

//The actual memory allocation of the array however is static. If you have issues with RAM, change DATA\_SET\_MAX

**int** actualSPS;

/\*

void ADS1232(uint8\_t pdwn, uint8\_t sclk, uint8\_t dout) //constructor

{

pdwnPin = pdwn;

doutPin = dout;

sclkPin = sclk;

a0Pin = 0;

spdPin = 0;

gain1Pin = 0;

gain0Pin = 0;

tempPin = 0;

initConfig();

}

\*/

**void** **ADS1232**(uint16\_t pdwn, uint16\_t sclk, uint16\_t dout, uint16\_t a0, uint16\_t spd, uint16\_t gain1, uint16\_t gain0, uint16\_t temp)

{

pdwnPin = pdwn;

doutPin = dout;

sclkPin = sclk;

spdPin = spd;

gain1Pin = gain1;

gain0Pin = gain0;

a0Pin = a0;

tempPin = temp;

initConfig();

}

**void** **initConfig**()

{

adcSpeed = 80;

adcGain = 128;

adcChannel = 0;

adcTemp = 0;

calFactor = 1400.0;

tareOffset = 0.0;

ignoreDiff = 0.0;

ignoreDiffThreshold = 0.0;

smoothing = true;

}

**void** **begin**(uint8\_t channel, uint8\_t gain, uint8\_t speed)

{

pinMode(pdwnPin, OUTPUT);

pinMode(doutPin, INPUT\_PULLUP); **gpio\_set\_pull\_mode**(doutPin, *GPIO\_PULLUP\_ONLY*);

pinMode(sclkPin, OUTPUT);

powerOn();

**if** (a0Pin >0) {

pinMode(a0Pin, OUTPUT);

}

**if** (spdPin >0) {

pinMode(spdPin, OUTPUT);

}

**if** (gain1Pin >0) {

pinMode(gain1Pin, OUTPUT);

}

**if** (gain0Pin >0) {

pinMode(gain0Pin, OUTPUT);

}

**if** (tempPin >0) {

pinMode(tempPin, OUTPUT);

digitalWrite(tempPin,LOW);

}

setChannel(channel,false);

setGain(gain,false);

setSpeed(speed,false);

delay(250);

tare(false,true);

}

bool **isReady**()

{

**return** digitalRead(doutPin) == LOW;

}

bool **safeWait**()

{

uint32\_t elapsed;

elapsed = millis();

/\*

for(int k = 0; k < 1000; k++)

{

if(isReady())

{

return true;

}

delay(1);

}

return false;

\*/

**while** (!isReady()) {

**if** (millis() > elapsed + 2000) {

//timeout

LOG\_PRINTLN("Error while waiting for ADC");

**return** false;

}

}

**return** true;

}

**void** **powerOn**()

{

digitalWrite(pdwnPin, LOW);

delayMicroseconds(10);

digitalWrite(pdwnPin, HIGH);

digitalWrite(sclkPin, LOW);

// while (!isReady()) { };

**if** (!safeWait()) {

LOG\_PRINTLN("Power on error");

**return**;

}

calibrateADC();

}

**void** **powerOff**()

{

digitalWrite(pdwnPin, LOW);

digitalWrite(sclkPin, HIGH);

}

**void** **setGain**(uint8\_t gain, bool calibrate) // 1/2/64/128 , default 128

{

uint8\_t adcGain1;

uint8\_t adcGain0;

**if**(gain == 1) {

adcGain1 = 0;

adcGain0 = 0;

} **else** **if** (gain == 2) {

adcGain1 = 0;

adcGain0 = 1;

} **else** **if** (gain == 64) {

adcGain1 = 1;

adcGain0 = 0;

} **else** {

//default == 128

adcGain1 = 1;

adcGain0 = 1;

}

**if** (gain0Pin>0 && gain1Pin>0) {

digitalWrite(gain1Pin,adcGain1);

digitalWrite(gain0Pin,adcGain0);

}

**if** (calibrate) { calibrateADC(); }

}

**void** **setSpeed**(uint8\_t sps, bool calibrate) //10 or 80 sps , default 10

{

adcSpeed = sps;

**if** (spdPin > 0 ) {

**if**(adcSpeed == 80) {

digitalWrite(spdPin,HIGH);

} **else** {

digitalWrite(spdPin,LOW);

}

}

**if** (calibrate) { calibrateADC(); }

}

uint8\_t **getSpeed**()

{

**return** adcSpeed;

}

**void** **setChannel**(uint8\_t channel, bool calibrate) //0 or 1

{

**if** (a0Pin > 0 ) {

digitalWrite(a0Pin,channel);

}

**if** (calibrate) { calibrateADC(); }

}

**void** **setDataSetSize**(uint8\_t datasetsize)

{

**if** (datasetsize != DATA\_SET) {

DATA\_SET = datasetsize;

resetSmoothing(0);

DEBUG\_PRINT("dataset size changed to ");DEBUG\_PRINTLNV(DATA\_SET);

DEBUG\_PRINTLN("Zeroing out our dataset");

}

}

**void** **setSmoothing**(bool enable) {

smoothing = enable;

DEBUG\_PRINTLN("Zeroing out our dataset");

resetSmoothing(0);

}

bool **getSmoothing**() {

**return** smoothing;

}

**void** **resetSmoothing**(int32\_t value) {

**if** (DATA\_SET > DATA\_SET\_MAX) {

DATA\_SET = DATA\_SET\_MAX;

} **else** **if** (DATA\_SET < DATA\_SET\_MIN) {

DATA\_SET = DATA\_SET\_MIN;

}

**if** (value == 0) {

//zero is our flag for current value

value = readADC();

}

readIndex = 0;

**for** (uint8\_t i = 0; i < DATA\_SET\_MAX+1; i++) {

dataSampleSet[i]=value;

DEBUG\_PRINTV(dataSampleSet[i]);DEBUG\_PRINT(",");

}

DEBUG\_PRINT("->");DEBUG\_PRINTLNV(value);

}

//void ADS1232::readInternalTemp()

//{

// //Not implemented.

// //If you need to, it is not very hard, check datasheet

//}

**void** **calibrateADC**()

{

DEBUG\_PRINTLN("ADC calibration init...");

readADC();

//readADC returns with 25th pulse completed, so immediately continue with 26th

delayMicroseconds(1);

digitalWrite(sclkPin, HIGH); // 26th pulse

delayMicroseconds(1);

digitalWrite(sclkPin, LOW); // end of 26th

//actual calibration begins... wait for dout = LOW

**if** (!safeWait()) {

//oops...time out !!!

LOG\_PRINTLN("ADC calibration error");

**return**;

}

readADC(); //read once without saving

DEBUG\_PRINTLN("ADC calibration done...");

//all done, ready to read again...

}

**void** **tare**(byte type, bool calibrate)

{

DEBUG\_PRINT("ADC tare with type ");DEBUG\_PRINTLNV(type);

**if** (type == 0) { //quick

tareOffset = readRaw(1);

} **else** **if** (type == 1) { //rapid, not so usefull with buttons, usefull only in many successive tares.

DEBUG\_PRINT("Taring using lastAdcValue= ");DEBUG\_PRINTLNV(lastAdcValue);

tareOffset = lastAdcValue;

} **else** { //full tare + calibrate

**if** (calibrate) {

DEBUG\_PRINTLN("ADC tare + calibrate");

calibrateADC();

}

tareOffset = readRaw(1);

tareOffset = readRaw(1);

tareOffset = readRaw(1);

tareOffset = readRaw(1);

}

DEBUG\_PRINT("New tare offset ");DEBUG\_PRINTLNV(tareOffset);

resetSmoothing(tareOffset);

}

**void** **setCalFactor**(**float** cal)

{

**if** (cal > 0) {

calFactor = cal;

} **else** {

calFactor = 1.0;

}

}

**void** **setMinDiff**(int32\_t diff, int32\_t threshold)

{

//changes less than abs(diff) when value < tareOffset + threshold will not be accounted.

//useful when near 0grams, seems more accurate to the user :D

ignoreDiff = diff;

ignoreDiffThreshold = threshold;

}

**double** **readUnits**(uint8\_t samples) {

**double** unitsvalue = 0.0;

unitsvalue = (**double**)(readRaw(samples)-tareOffset)/(**double**)calFactor;

//Can we go down to 0.001 range or more???

//Sure, I doubt though it can be useful with 3.3V excitation and normal off the shelf load cells.

//In that case, do we really need double here? No. You can change it to float.

//Remember that Serial cannot print more than 2 decimal digits.

//Uncomment the following for testing

//unitsvalue = unitsvalue\*100.0;

//DEBUG\_PRINT("Super duper resolution value\*100 = ");DEBUG\_PRINTLN(unitsvalue);

**if** (unitsvalue == -0.00) { unitsvalue = 0.0; }

**return** unitsvalue;

}

int32\_t **getSmoothedValue**()

{

//simple h/l rejection algorithm from the following library

//https://github.com/olkal/HX711\_ADC

int32\_t data = 0;

int32\_t L = 2147483647;

int32\_t H = -2147483648;

**for** (**int** r = 0; r < DATA\_SET; r++) {

**if** (L > dataSampleSet[r]) L = dataSampleSet[r]; // find lowest value

**if** (H < dataSampleSet[r]) H = dataSampleSet[r]; // find highest value

data += dataSampleSet[r];

}

data -= L; //remove lowest value

data -= H; //remove highest value

//Uncomment the following to debug your load cell

//DEBUG\_PRINT(" L / H "); DEBUG\_PRINT(L);DEBUG\_PRINT("/");DEBUG\_PRINTLN(H);

**return** data/(DATA\_SET-2);

}

int32\_t **readRaw**(uint8\_t samples)

{

int32\_t valuessum=0;

**for** (uint8\_t i=0;i<samples;i++)

{

valuessum += readADC();

}

lastAdcValue = valuessum/samples;

**if** (!smoothing)

{

//lastAdcValue = valuessum/samples;

}

**else**

{

//In any case, put the value to the array

**if** (DATA\_SET > DATA\_SET\_MAX)

{

DEBUG\_PRINTLN("Zeroing out our dataset");

DATA\_SET = DATA\_SET\_MAX;

resetSmoothing(0);

}

**else** **if** (DATA\_SET < DATA\_SET\_MIN)

{

DEBUG\_PRINTLN("Zeroing out our dataset");

DATA\_SET = DATA\_SET\_MIN;

resetSmoothing(0);

}

**if** (readIndex > DATA\_SET - 1)

{

readIndex = 0;

}

**else**

{

readIndex++;

}

dataSampleSet[readIndex] = lastAdcValue;

lastAdcValue = getSmoothedValue();

}

DEBUG\_PRINT("lastAdcValue= ");

DEBUG\_PRINTLNV(lastAdcValue);

readsPerSecond++;

**if** (millis() - lastRateCheck > 1000)

{

actualSPS = readsPerSecond\*1000/(millis()-lastRateCheck);

readsPerSecond = 0;

lastRateCheck = millis();

}

**return** lastAdcValue;

}

int32\_t **readADC**()

{

int32\_t adcvalue = 0;

**if** (!safeWait()) {

LOG\_PRINTLN("ADC wait ERROR");

**return** 0;

}

//ADC ready...begin

//manual 1-bit read

//for esp32 this is a more reliable method than shiftin since we can add a slight delay and avoid spikes in read values

//each pulse must be at least 100ns = 0.1μs. We insert a delay here of 1μs to be sure

//alternatively you can use 8bit shiftin read => adcvalue = shiftIn(doutPin, sclkPin, MSBFIRST); adcvalue <<= 8; , etc...

**int** i=0;

portMUX\_TYPE mux = portMUX\_INITIALIZER\_UNLOCKED;

portENTER\_CRITICAL(&mux);

**for**(i=0; i < 24; i++) { //24 bits => 24 pulses

digitalWrite(sclkPin, HIGH);

delayMicroseconds(1);

adcvalue = (adcvalue << 1) + digitalRead(doutPin);

digitalWrite(sclkPin, LOW);

delayMicroseconds(1);

}

portEXIT\_CRITICAL(&mux);

digitalWrite(sclkPin, HIGH); // keep dout high // 25th pulse

delayMicroseconds(1);

digitalWrite(sclkPin, LOW);

//wait for it to become high actually

**while**(digitalRead(doutPin) != HIGH)

{

yield();

DEBUG\_PRINTLN("waiting for dout");

}

ESP\_LOGI("[ADC Internal]" , "Raw Value: %d ", adcvalue);

adcvalue = (adcvalue << 8) / 256;

**return** adcvalue;

}

**void** **delayMicroseconds**(uint64\_t number\_of\_us)

{

**ets\_delay\_us**(number\_of\_us);

}