Simple and sliding average

This documentation is for beginners. The goal is to make clear the usefulness and ease of use of the sliding/moving average. Glis.h "library" is easy to understand and includes the functions to handle two streams of 16 bits values. It can be modified easily for 8 bit data and is a good preparation to program more complex averaging solutions.

Simple averaging

The measurement of a sensor is always noisy. A simple solution is to group several measures and calculate the average. The duration between the results is increased.
For example, we receive 8-bit values from a sensor and accumulate them. Every 4 values, the average is transmitted.
The blocking function that average 4 equidistant measurements is simple. The measure is made every **pp** millisecond and the measures are summed and divided by for (and not divided by 4 and summed). This mean the total may need more bits than the measures.

```
uint16_t Average (uint16_t pp, uint16_t mes) {
  uint16_t total;
  for (uint8_t i=0; i<4; i++) { // four <14-bits measures
     total  += mes;
     delay(pp);
  }
  return (total/4);
}
```

The total must not exceed 16 bits. If you add 10-bit words (analogRead() results), you can average up to 64 values.
A division by 4 is clearly faster than a division by 3 or 5, because it is just a shift. We will always average on 2, 4, 8,16 values to save memory and calculated time.

The test program depends on what you read. If a potentiometer is connected to A0, the values can be displayed on the serial terminal.

```
// TestSimpleAverage.ino
void setup() {
  }
int Average (int pp,int mes) { // 4 measures
  int total;
  for (byte i=0; i<4; i++) {
     total += mes;
     delay(pp);
  }
  return (total/4);
}
byte light;
void loop() {
  light = Average(10,analogRead(A0));
//...  if (light > MaxLight) {...}
// Update every 4x10 ms
}
```

1. Average values of a table

We have accumulated the values in a table and we need the average. We also need to round the result, that is if the rest of the division is higher than 0.5, add one to the integer part.

```
void setup()   {
  Serial.begin(9600);
}
byte table[] = {23,43,66,51,11,24,92,7,33,15};
```

```
lgTable = sizeof(table);
uint16_t total = 0;
uint16_t avera
void loop(){
  for (int i=0; i<lgTable; i++) {
    total += table [i] ;
  }
  avera = total/lgTable ;
  if (total%lgTable) >= lgTable/2) { moyenne++}
  Serial.print (avera);
  while (1) {}
}
```

We need the length of the tables in several instructions:
- in the definitions of the table [] (the hook can remain empty, the C can count the declared elements),
- in the number of iterations in the for loop,
- in the division.
- in the calculation of the rounding

Thanks to the **sizeof**() operator, the compiler transfers the length of the table into a variable, and the program does not depends on the initial length of the table. .

This example also makes it possible to mention a functionality of the C called type conversion (casting type): In the total statement += table [i]; an 8-bit word is added to a 16-bit word. The compiler must convert the 8-bit word table [i] into a 16-bit word so that it can be added to total. In this case, it does it alone, but sometimes, especially with floating numbers, it must be said that there is a change of type. In our case, some compilers may require to write
total += (int) table [i];

3 Non-blocking average

The normal case is that the function that reads the sensor gives an averaged value. The function is called by synchronous interrupt to read the sensor, but the value of the result of the measurement is updated only every 4 interrupts.

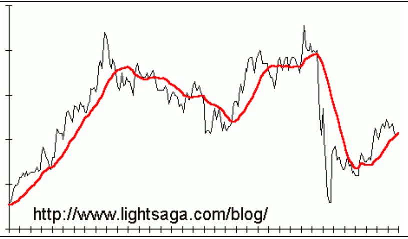
```
// TestAverageByInterrupt.ino
volatile int MoyCaptA0;  // Global var
int total; byte i=0;
void GetMoyCaptA0 () { // 4 measures
  if (i++ < 4) {total += analogRead(A0)}
  else {i=0; MoyCaptA0 = total/4;}
}
void setup() {
  ISR (TIMER2_OVF_vect) {
    TCNT2 = 141;  //  58 us (256-141=58x2)
    GetMoyCaptA0();
  }
}

void loop() {
//...  if (MoyCaptA0 > MaxLight) ...
}
```

In summary, every 60 microseconds (or another period), we read the sensor, and every 240 microseconds, the MoyCaptA0 value is changed. But Maxlight will be updated according to the main loop duration.

4 Moving average

The sliding or moving average (running average) is much more interesting. We average the last 4 or 8 bars. The smoothing is improved, as shown in the curve opposite. The filtered signal naturally has a delay, less annoying than the unfiltered sudden changes or the stairs of a simple average.



Obviously, at the first measurement, the average can not be calculated on previous data. The setup must fill the table with a first set of measurement of the sensor. If you initialize the table with zeros, which the compiler does by default, you have to wait 4 measures before the average is usable. What must be done is to initialize the table with the first value read.

4.1 Algorithm

The algorithm save the data on a **`taGlis[Lglis];`** array; the variable **`total`** totalize the contents of the table. For each new measurement, one replace the oldest measurement in the table and correct the total. The **`ptTaglis`** pointer is a circular pointer. The total is divided by **`Lglis`** to get the average value to be returned.

```
byte mesure;  // sensor value
#define Lglis 8
byte taGlis[ptTaGlis];
byte ptTaGlis; int total;
void SetupGlis (byte dd) {
  for (byte i=0; i<Lglis; i++) {
    taGlis[i]= dd;
    total += dd;
  }
}
byte Glis (byte dd) {  // rend la moyenne (8 bits) après nouvelle donnée dd
  total -= taGlis[ptTaGlis];  // on soustrait la plus ancienne valeur
  total += dd;                // on ajoute la nouvelle
  taGlis[ptTaGlis] = dd;      // table à jour
  if (++ptTaGlis==Lglis) {ptTaGlis=0;}  // incrémentation circulaire
  return (toTable/Lglis);
}
```

`IniGlis(firstVal);` is the function that just fill the table with the first valuer read.

4.2 Test with the serial terminal

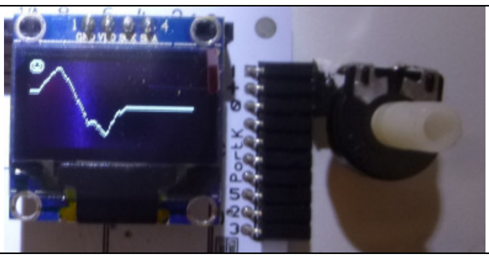
This test program helps to understand the algorithm by displaying the contents of the table on the terminal. 0 to 9 digits are entered in the top line and terminated by a CR.

```
//TestMoyGlis.ino avec terminal série
//Taper un chiffre de 0 - 7 dans la ligne du haut suivi d'un Enter
#define Lglis 8   // debut du fichier MoyGlis.h
byte taGlis[Lglis];
byte ptTaGlis; int toTable;
void SetupGlis () {
  for (byte i=0; i<Lglis; i++)  { taGlis[i]=0; }
  toTable=0;
}
byte Glis (byte dd) {
  toTable -= taGlis[ptTaGlis];
  toTable += dd;
  taGlis[ptTaGlis] = dd;
  if (++ptTaGlis==Lglis) {ptTaGlis=0;}
  return (toTable/Lglis);
}
void setup(void) {
  Serial.begin(9600);
  SetupGlis();
}
// Le terminal rend des car Ascii.
byte moy;
void loop () {
  if (Serial.available() > 0) {
    byte car = (Serial.read ())&0x07;   //0-7
    moy = Glis (car);
    Serial.print (car);Serial.print (" ");
    Serial.print (moy);Serial.print (" ");
    Serial.print (ptTaGlis);Serial.print (" tot ");
    Serial.println (toTable,HEX);
  }
}
Modify the program to type 8-bit number
```

4.3 Glis.h library

The file Glis.h is for 16 bits variables.
The number of measurements is declared at the beginning of **`Glis.h`**:
`moy16=Glis16(val16);` make the average.
`Glis16(first16);` initialize the table at setup.

4.4 Example with a potentiometer

Glis16 are tested with a pot on A0, with display on Oled (see www.didel.com/Oled.pdf). By acting on the delay, we see the average catch up value read. The program, like the other programs in this documentation, can be loaded from www.didel.com/Moyennes.zip.	

```
//TestOledPotGlis  170509 Lit un pot et affiche la moyenne
//\image:OledPot.bmp
typedef uint8_t byte;
#include "MoyGlis.h"
void setup(){
  SetupOledPix();
  SetupMoyGlis16();
  // pas besoin de setup pour le pot
}
int moy, pot; // variables globales
void loop(){
  pot = analogRead(A0);
  moy = Glis16 (pot);

  delay(100);
}
```

With a delay of 0.1s and a depth of 8, we see that the abrupt variations of the potentiometer are taken into account with a delay of 0.5s

5 Timer interrupt

The sensor (here an analog signal on A0) is read regularly by a timer interrupt. The rolling average is calculated each time and the global variable moyA0 is updated. The main program uses this variable, without knowing how it was made.
The principle of synchronous programming as "LibX" (www.didel.com/LibX.pdf) is easy to understand:
Working with interrupts is like having an extra processor with its setup and loop that decides which tasks to execute at each interrupt. The included file "Inter2.h" contains the SetupInter2(); function. ISR (TIMER2_OVF_vect) {} lists the tasks to be executed. These tasks must all run in less than 10 microseconds, but are called every 60 microseconds to get a good "real-time" response.
Our example reads a potentiometer. Arduino installs it by default, there is no explicit setup. This potentiometer is read by interruption, and averaged in parallel. The file InterGetA0.h is therefore the following:

```
//InterGetA0.h
void  SetupInter(){ // initialise the timer
  TCCR2A = 0; //default
  TCCR2B = 0b00000010;  // clk/8
  TIMSK2 = 0b00000001;  // TOIE2
  sei();
}
// déclarer les variables locales "volatile"
ISR (TIMER2_OVF_vect) {
   TCNT2 = 141;  //  58 us (256-141=58x2)
   GetMoyA0();
}
```

The inserted file for one variable is

```
// Glis.h
#define Lglis 8
uint16_t taGlis[Lglis];
uint16_t ptTaGlis;
uint32_t total;
void SetupGlis (dd) {
  for (uint8_t i=0;i<Lglis;i++) {taGlis[i]=dd;}
}
int16_t Glis(int16_t dd) {
  total -= taGlis[ptTaGlis];  // on soustait la plus ancienne valeur
  total += dd;          //total à jour
  taGlis[ptTaGlis] = dd;  // table à jour
  if (++ptTaGlis==Lglis) {ptTaGlis=0;}
  return (total/Lglis);
}

void SetupGlis() {
  for (uint8_t i=0; i<Lglis; i++) {taGlis16[i]=0;}
  toTa32=0;
}
int Glis(int dd) {
  toTa32 -= taGlis16[ptTaGlis];  // on soustait la plus ancienne valeur
  toTa32 += dd;          //total à jour
  taGlis16[ptTaGlis] = dd;  // table à jour
  if (++ptTaGlis==Lglis) {ptTaGlis=0;}
  return (toTa32/Lglis);
}
```

By having prepared and understood these libraries, the program is very simple. The problem we sometimes have when calling included files is to respect the order the compiler needs. A function or variable must be defined before being used. Global variables are declared at top.

```
//TestMoyenneGlissanteInter.ino
uint_16 MoyA0;  //Variables globales
#include "Inter2.h
#include "Glis.h"
#include "OledPix.h"

void setup() {
  SetupMoyGlis();
  SetupInter();
    // A0 ne doit pas être initialisé
}
uint8_t x,y1,y2; // on veut le graphique de la valeur et sa moyenne
void loop() {
  uint16_t val=GetMoy16(analogRead(A0));  <----- ko
  delay(100);
}
```

Note that if the sliding average of an ultrasonic sensor is to be used, we can not use the Arduino Pulsein () function which is blocking. The GetUson () function of the X library must be used, and is easily supplemented by a sliding average that filters reading anomalies (see www.didel.com/OledUson.pdf).

Application for the Gy521 accelerometer and sensor.
See www.didel.com/WittyGyro.pdf

jdn 170527/190512