**How Otto moves ?**

**J. Lemaire**

**(Pierrefeu May 2021)**

# Motivations

Some time ago, my granddaughter (8 years old) expressed her enthusiasm, following an initiation to robotics that she had just taken at school. This prompted me to give her an OttoDIY+[[1]](#footnote-2) robot, to let her know a little more... and possibly wow the friends !

But, while helping to assemble it, I admit I was totally amazed by this little Robot, really brilliant by its simplicity and its possibilities, in particular by its approach using only 4 micro servos.

Having been a teacher in a computer environment[[2]](#footnote-3) in another lifetime (I'm now retired), I got caught up in the game and tried to understand how it works.

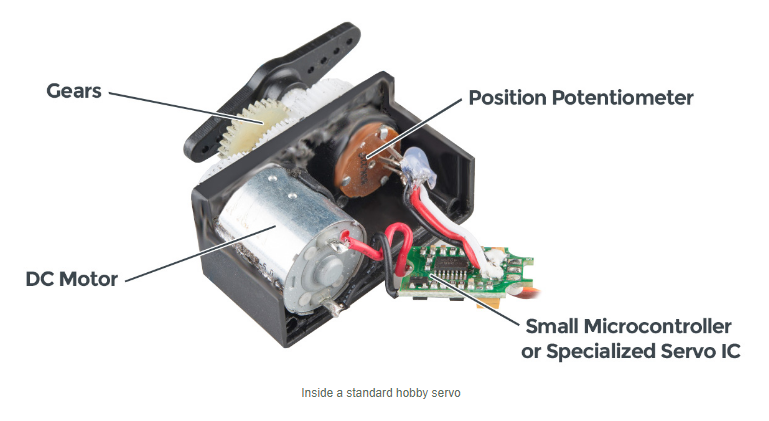
Moreover, having observed like other users that the firmware proposed to control OttoDIY+ in Bluetooth was extreme for an Arduino Nano which only has 2K of SRAM memory, I wondered if it was not possible to optimize a bit of the C ++ code provided, limiting myself here to the only problem of basic Otto movements, for simplicity.

This document exposes this little work. But before going into the details, I want to make it clear that I am not trying here to criticize the remarkable work of the designers of Otto, but just to understand its  locomotor functioning, to evoke the questions that I asked myself occasionally, and very modestly, to present the provided answers. In doing so, I am well aware of the very limited nature of this study in view of all of Otto's body language !

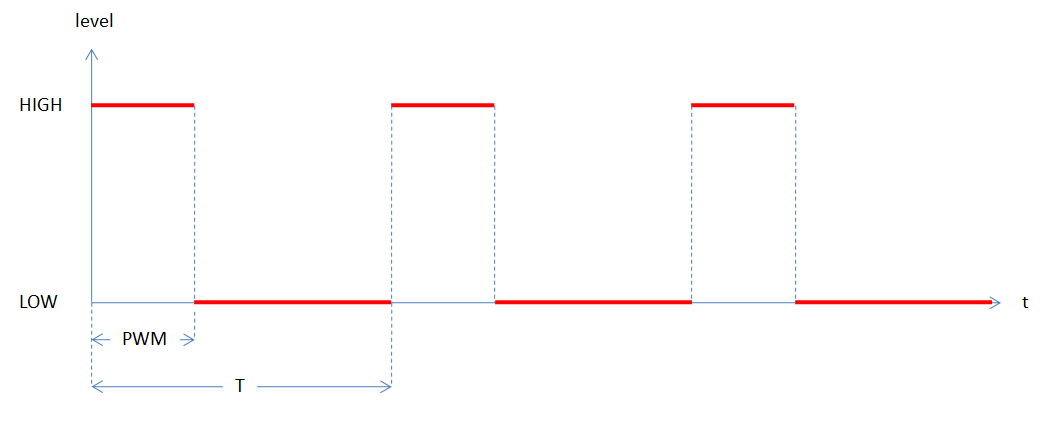
# Servos

Otto's movements are carried out by 4 micro-servos of the MG90S type[[3]](#footnote-4).

They each consist of a small DC motor that, through a series of gears, can rotate an axis through an angle of approximately 0-180 °. An integrated circuit controls the motor, the angle being controlled by a potentiometer[[4]](#footnote-5):



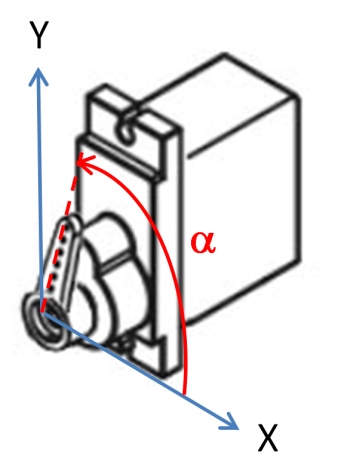
This integrated circuit can be controlled by a 50Hz frequency PWM[[5]](#footnote-6) signal (period T = 20ms), the value of which, also denoted PWM for simplicity, will be defined by the duration of the HIGH state over each period :



Default for a servo :

* PWMmin = 544µs => angle ≅ 0°
* PWMmax = 2400µs => angle ≅ 180°

The angle **α** of rotation, also called the **position** of the servo, will be measured as follows, the half-arm being mounted so that this angle can vary between 0 and 180 °:



To control a servo with an Arduino and on any digital pin, just use the **Servo** library made by Michael Margolis[[6]](#footnote-7), which defines the **Servo** class:

class Servo

**{**

public**:**

Servo**();**

uint8\_t attach**(**int pin**);** // attach the given pin to the next free channel, sets pinMode, returns channel number or 0 if failure

uint8\_t attach**(**int pin**,** int min**,** int max**);** // as above but also sets min and max values for writes.

void detach**();**

void write**(**int value**);** // if value is < 200 its treated as an angle, otherwise as pulse width in microseconds

void writeMicroseconds**(**int value**);** // Write pulse width in microseconds

int read**();** // returns current pulse width as an angle between 0 and 180 degrees

int readMicroseconds**();** // returns current pulse width in microseconds for this servo (was read\_us() in first release)

bool attached**();** // return true if this servo is attached, otherwise false

private**:**

uint8\_t servoIndex**;** // index into the channel data for this servo

int8\_t min**;** // minimum is this value times 4 added to MIN\_PULSE\_WIDTH

int8\_t max**;** // maximum is this value times 4 added to MAX\_PULSE\_WIDTH

**};**

Here is an example of use in an Arduino sketch, for a servo connected to pin 2 :

#include <Servo.h>

Servo s**;** // Servo object

void setup**()**

**{**

s**.**attach**(**2**);** // Attaches the servo to pin 2

s**.**write**(**0**);** // Turns 0°

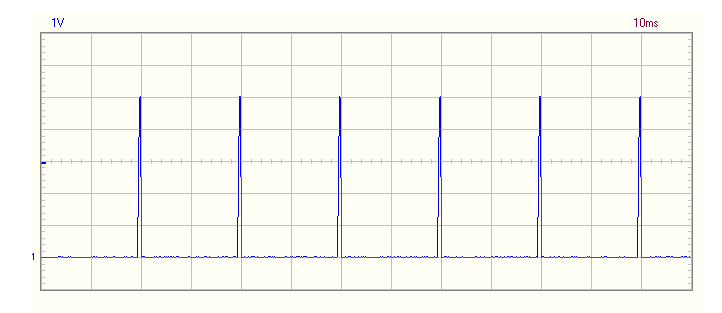
**}**

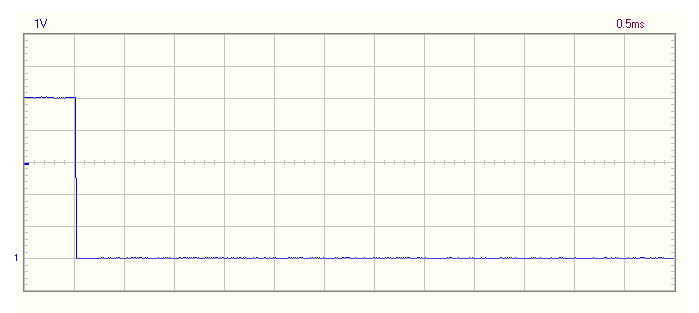
void loop**()**

**{**

**}**

In the case of an Arduino Nano, 12 servos can be used in parallel and the implementation of this library takes advantage of **Timer1** to generate such signals, in interrupt mode[[7]](#footnote-8). Here are the signals observed with an oscilloscope[[8]](#footnote-9) on pin 2, when we run the previous sketch:





This interrupt control mode allows a **50Hz** PWM signal to be sent to any digital output pin and to run servos in parallel, without blocking the microcontroller. But, the **Servo** library does not allow to control their rotation speed[[9]](#footnote-10) or even to stop them on a precise angle[[10]](#footnote-11). Finally, note that a servo is not as precise as a stepping motor : it is therefore often necessary to trim it, to make the effective angle of the arm coincide with the angle transmitted to the **write** method, for example at 90° ; this possibility is also ignored in the **Servo** library. On the other hand, one can redefine the values ​​PWMmin and PWMmax, which is another way, fairer but not always possible, to solve the problem.

The following sketch shows that the read method returns the last angle passed to **write** and not the current position of the servo:

#include <Servo.h>

Servo s1**,**s2**;**

void setup**()**

**{**

s1**.**attach**(**2**);**

s2**.**attach**(**3**);**

s1**.**write**(**0**);**

s2**.**write**(**0**);**

Serial**.**begin**(**115200**);**

delay**(**1000**);**

Serial**.**println**(**"start"**);**

Serial**.**println**(**millis**());**

Serial**.**print**(**"s1="**);** Serial**.**print**(**s1**.**read**());**

Serial**.**print**(**" s2="**);** Serial**.**println**(**s2**.**read**());**

s1**.**write**(**180**);**

s2**.**write**(**180**);**

**}**

void loop**()**

**{**

Serial**.**println**(**millis**());**

Serial**.**print**(**"s1="**);** Serial**.**print**(**s1**.**read**());**

Serial**.**print**(**" s2="**);** Serial**.**println**(**s2**.**read**());**

delay**(**10**);**

**}**

Display on the serial monitor :

start

999

s1=0 s2=0

1000

s1=180 s2=180

1010

s1=180 s2=180

1021

s1=180 s2=180

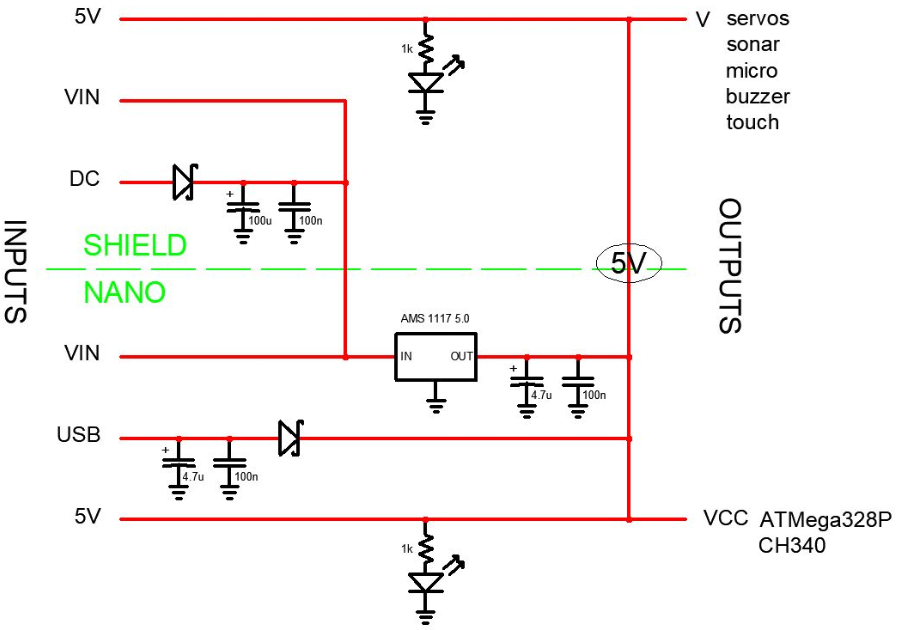
1032

s1=180 s2=180

…

To end this short presentation, it should be remembered that a servo, even micro, can generate a significant current : this can exceed 200mA, when operating under constraints. In the case of Otto, it is therefore strongly recommended to supply its 4 micro-servos separately, and not only via a type 2 USB serial port, which cannot supply more than 500mA. This will also protect the Arduino Nano microcontroller and other robot components against current variations, when powered through its VIN input, even if its 5V regulator (AMS 1117 5.0) here seems quite tolerant and responsive.

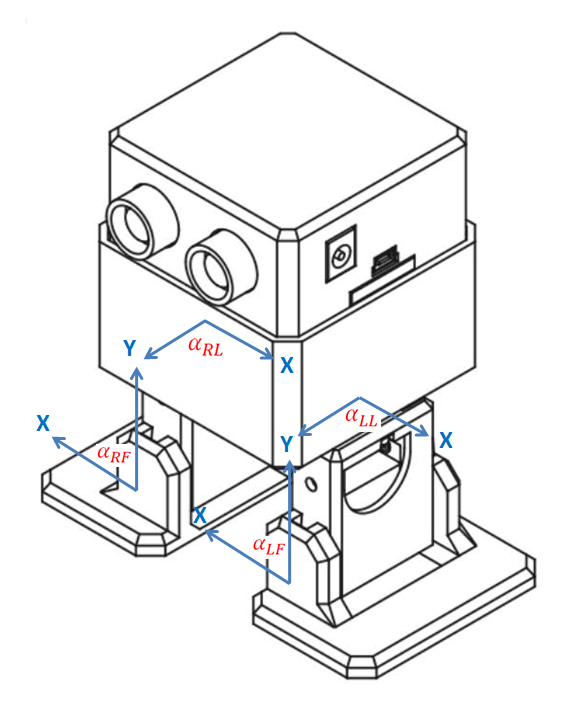
To fully understand and implement an effective solution, here is Otto's default power circuit diagram:



# The solution adopted in Otto

The previous convention of measuring the angle of rotation of a servo leads to this, for all 4 Otto servos, mounted according to the assembly instructions :

* => left leg
* => right leg
* => left foot
* => right foot

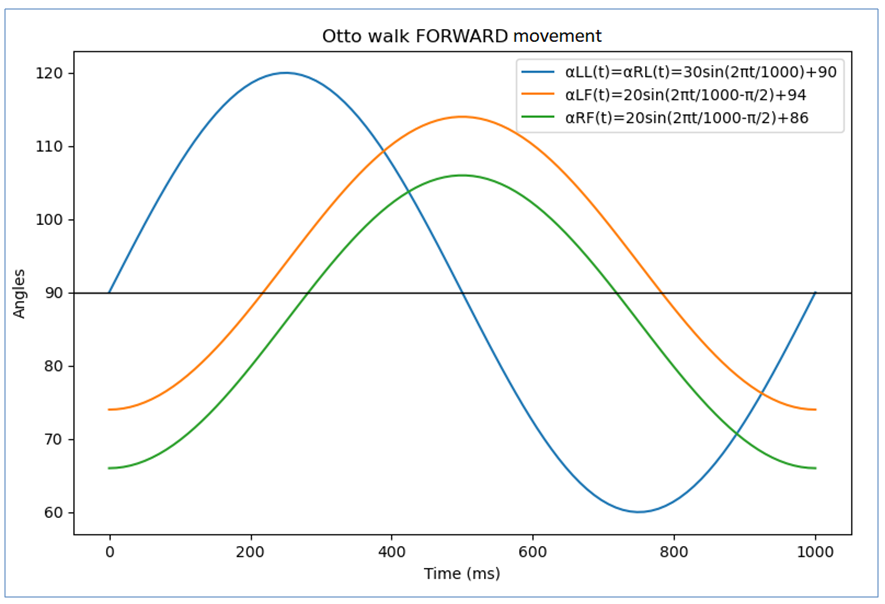


Then :

* or ⬄ legs turned to the right
* ⬄ left foot lowered (outward)
* ⬄ right foot lowered (outward)

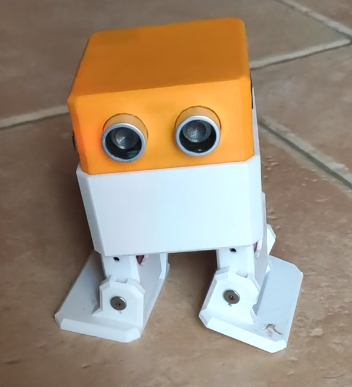
An Otto movement for a duration *T* ms will be specified by the data of 4 angular functions, again denoted and for simplicity, defined on the interval [0, *T*] and with values ​​in [0.180].

For example, here are the functions called by the **walk (1, 1000, FORWARD)** method, to move Otto forward for 1000ms, with a step to the left then a step to the right:

[[11]](#footnote-12)

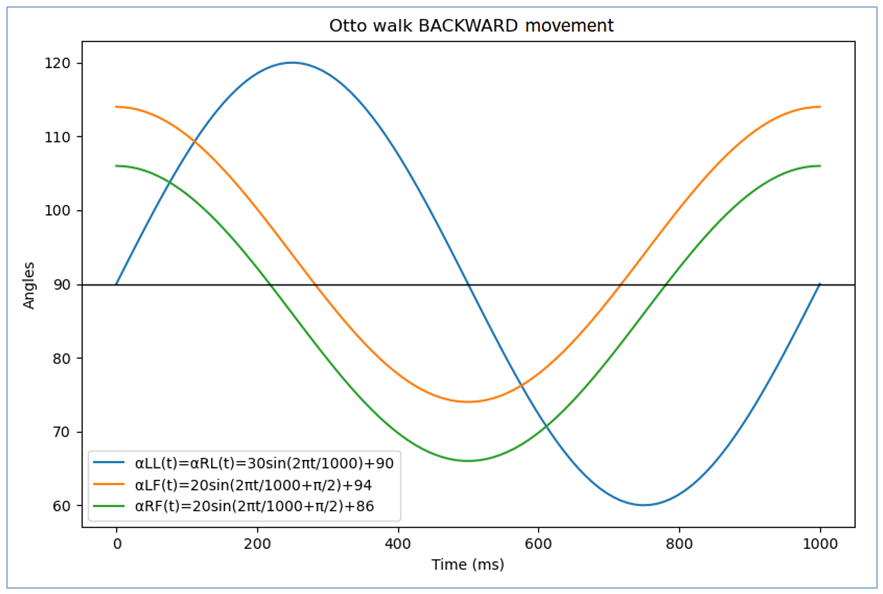
When  :

These cause the robot to advance with a fluid swing where it leans successively on its left leg then on its right leg, while making them turn to the right, then to the left (this is the resistance of the ground under the flat foot which allows this) :

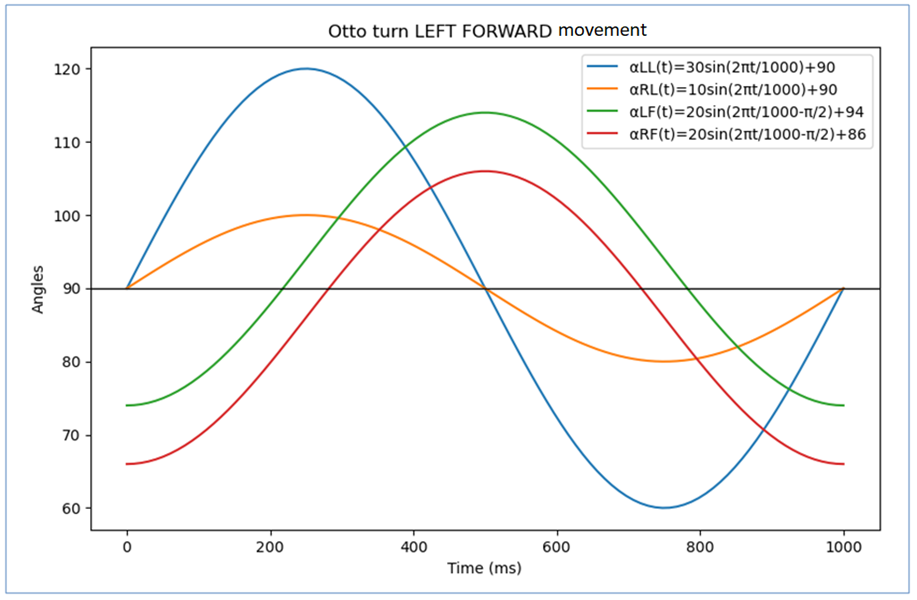
[](Otto.mp4)

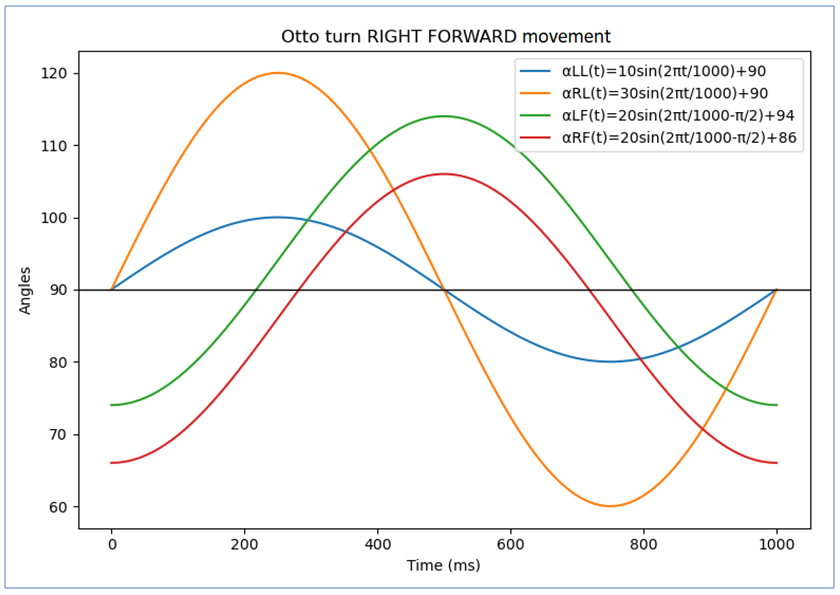
*(L-clic to see the video)*

To move it back, simply reverse the phase shifts of the feet :

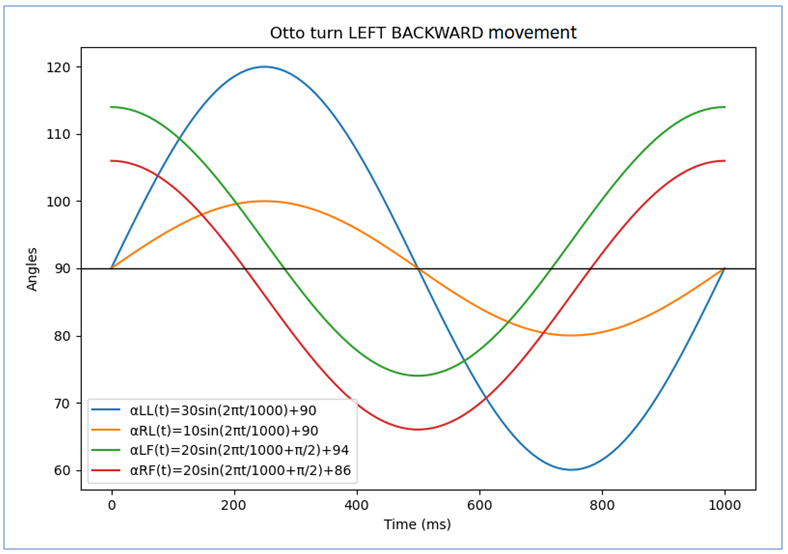


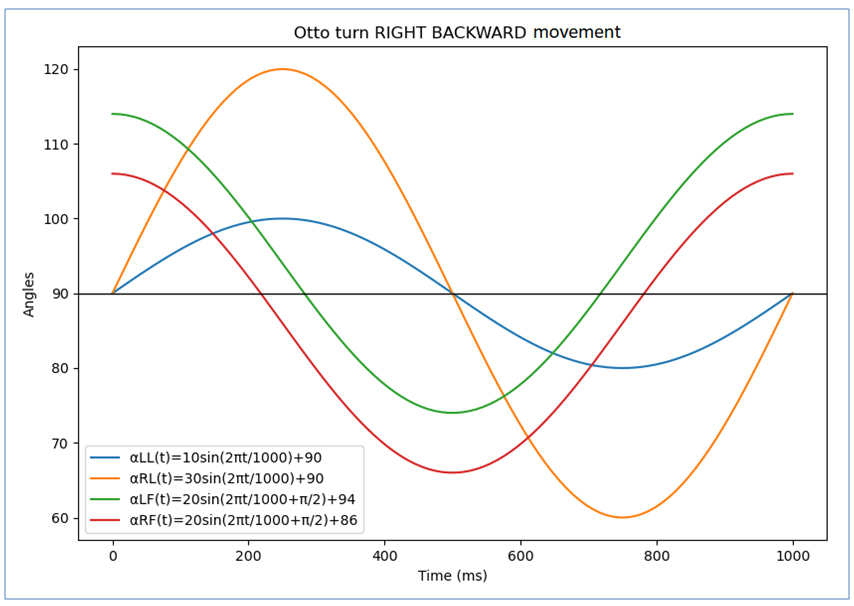
To rotate it while moving forward, all you have to do is reduce the amplitude of the rotation of the leg on the opposite side to which you want to turn ; for example :



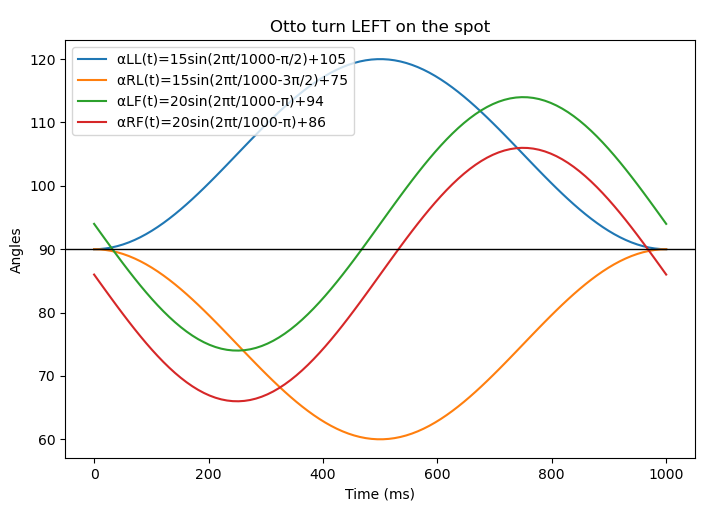


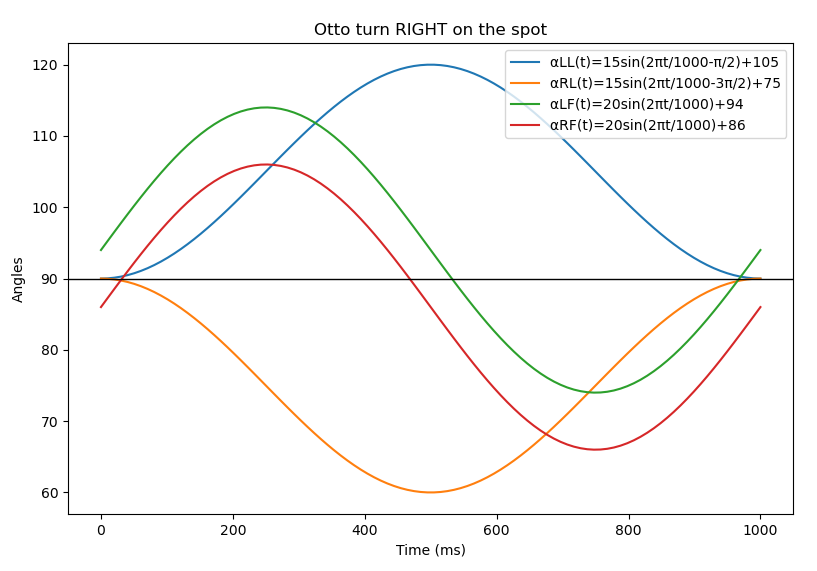
We can therefore imagine how to make it turn, moving backward this time:





Finally, if you want to make it turn on the spot, it suffices to generate a contradictory movement of the 2 legs, as for a mechanical showel :





All these angular functions are of the form :

With

* = amplitude (°)
* = offset (°)
* = phase (radians)
* = duration or period (ms)

Having noticed that several other movements could also be carried out with such functions (**turn**, **updown**, **swing**, **tiptoeSwing**, **jitter**, **ascendingTurn**, **moonwalker**, **crusaito**, **flapping**) [[12]](#footnote-13), the designers of Otto used « oscillator » objects, instances of the **Oscillator**[[13]](#footnote-14) class, almost identical to the **ServoOsc** class defined in the library of the same name[[14]](#footnote-15). It was initially developed by **Juan González Gómez**, as part of his Ph.D. thesis in robotics, **Robótica Modular y Locomoción: Aplicación a Robots Ápodos (2008)**[[15]](#footnote-16), who clearly demonstrated the importance of this concept, with an Arduino implementation[[16]](#footnote-17), used in early versions of Otto[[17]](#footnote-18):

class Oscillator

**{**

public**:**

Oscillator**(**int trim**=**0**)** **{**\_trim**=**trim**;};**

void attach**(**int pin**,** bool rev **=false);**

void detach**();**

void SetA**(**unsigned int A**)** **{**\_A**=**A**;};**

void SetO**(**unsigned int O**)** **{**\_O**=**O**;};**

void SetPh**(**double Ph**)** **{**\_phase0**=**Ph**;};**

void SetT**(**unsigned int T**);**

void SetTrim**(**int trim**){**\_trim**=**trim**;};**

int getTrim**()** **{return** \_trim**;};**

void SetPosition**(**int position**);**

void Stop**()** **{**\_stop**=true;};**

void Play**()** **{**\_stop**=false;};**

void Reset**()** **{**\_phase**=**0**;};**

void refresh**();**

private**:**

bool next\_sample**();**

private**:**

//-- Servo that is attached to the oscillator

Servo \_servo**;**

//-- Oscillators parameters

unsigned int \_A**;** //-- Amplitude (degrees)

unsigned int \_O**;** //-- Offset (degrees)

unsigned int \_T**;** //-- Period (miliseconds)

double \_phase0**;** //-- Phase (radians)

//-- Internal variables

int \_pos**;** //-- Current servo pos

int \_trim**;** //-- Calibration offset

double \_phase**;** //-- Current phase

double \_inc**;** //-- Increment of phase

double \_N**;** //-- Number of samples

unsigned int \_TS**;** //-- sampling period (ms)

long \_previousMillis**;**

long \_currentMillis**;**

//-- Oscillation mode. If true, the servo is stopped

bool \_stop**;**

//-- Reverse mode

bool \_rev**;**

**};**

This **Oscillator** class is basically associated with a servo that it can « oscillate ». It allows to define the parameters , , and of a function *α* of the previous form, by calling the methods **SetA**, **SetO**, **SetPh** and **SetT**. The **refresh** method causes the update of the movement obtained by applying this function ; it suffices to call it periodically in the **loop** section and this call is not-blocking. You can also trim the servo by calling the **SetTrim[[18]](#footnote-19)** method, stop it at any time by calling the **Stop** method and reduce its consumption by calling the **detach** method.

From the point of view of its movements, an Otto object, here instance of the **Otto9[[19]](#footnote-20)** class, is therefore essentially defined by the 4 oscillators stored in the array :

Oscillator servo**[**4**];**

All the movement functions ultimately call[[20]](#footnote-21) the **oscillateServos** method, passing its arrays containing the appropriate values ​​of the previous parameters ; after having taken them into account, the latter uses the **refresh** method of the 4 oscillators[[21]](#footnote-22):

void Otto9**::**oscillateServos**(**int A**[**4**],** int O**[**4**],** int T**,** double phase\_diff**[**4**],** float cycle**=**1**) {**

**for** **(**int i**=**0**;** i**<**4**;** i**++)** **{**

servo**[**i**].**SetO**(**O**[**i**]);**

servo**[**i**].**SetA**(**A**[**i**]);**

servo**[**i**].**SetT**(**T**);**

servo**[**i**].**SetPh**(**phase\_diff**[**i**]);**

**}**

double ref**=**millis**();**

**for** **(**double x**=**ref**;** x**<=**T**\***cycle**+**ref**;** x**=**millis**()) {**

**for** **(**int i**=**0**;** i**<**4**;** i**++) {**

servo**[**i**].**refresh**();**

**}**

**}**

**}**

All this therefore very clearly implements this notion of oscillation, central in the thesis of **Juan González Gómez**.

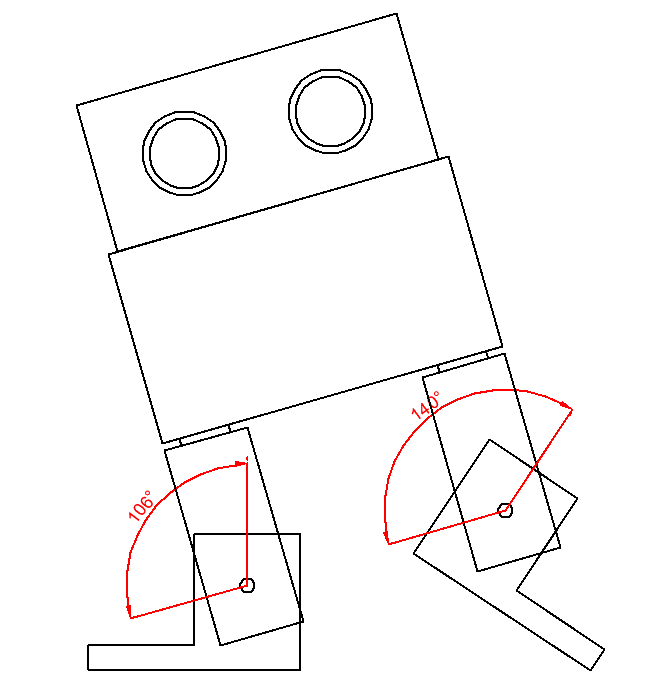
Without wishing to claim to correct this implementation, it is nevertheless tempting to better distribute the different functionalities considered here :

* The notion of movement first, which could use functions other than oscillations, with possibilities of sampling, interrupting and resuming, all in a non-blocking operation mode ;
* Its application to a servo, by inheriting the functionality of the **Servo** class, with a possibility of trim, as in **OttoDIYLib** ;
* The management of several servos in parallel ;
* The implementation of this in Otto, taking into account all the movements mentioned above, at different speeds and in a fluid manner ;
* Control of Otto via a serial port and in particular the one created by its Bluetooth module.

This is what we will attempt to do in the next paragraph.

But before doing this, we would like to ask the designers of Otto a question: How were the parameters of the various oscillatory movements determined, and in particular those of the feet ?

For example, if we consider the walk method where the legs are always parallel, we can calculate the angle of inclination of one foot so that the other is flat and this can be done graphically with a drawing software (Autocad here) and a scale representation of Otto:



Thus, when the right foot is raised with an angle of **106**° (when the 2 legs have an angle of 90 ° and are therefore separated as far as possible), the left foot should be lowered with an angle of **140°** so that the right foot either flat. However, the latter is set at **114°** in the walk oscillation. Why ?

# Classes Movement, ServoE and TServoE<N>

## Movement function

To vary the angle of a servo over time, we have seen that it suffices to define a function *α* over an interval [0, *T*] ; if the movement starts at time, the angle will be equal to at time .

This function *α* will generally be characterized by **parameters** ; this is the case, for example, for an oscillation, specified by the parameters and :

with

In C ++ , if the times are evaluated in milliseconds, we can therefore naturally consider the type **<double (double)>** for a movement function.

If we have a version at least equal to 11 of this language, there is a very direct way to implement such a function, with parameters : it suffices to consider a **Lambda** function whose parameters of the capture list (between []) correspond to the aforementioned parameters, cast with **std::function** to ultimately obtain a function of the type sought ; for example, for an oscillation :

std**::**function**<**double**(**double**)>** **[**a**,**b**,**phi**,**T**](**double t**){return** a**\***sin**(**2**\***PI**\***t**/**T**+**phi**)+**b**;};**

Unfortunately this powerful mechanism does not seem possible with an Arduino Uno or Nano, for which the Arduino IDE, which uses the avr-G ++ 7.3.0 compiler, does not know about **std::function**. This restriction is explained by the very low SRAM memory available on their **ATMega328P** processor : 2KB only ![[22]](#footnote-23)

We therefore implemented another technique, less compact but much more economical in processor resources, based on a class of parametric functions and a pointer to one of them. These functions, although classic, look like Lambda functions.

*Class* ***Movement****(files* ***Movement.h*** *and* ***Movement.cpp****) :*

#define TS 20 // Sampling period(ms) to update movements

enum StateMovement **{**active**,** stopped**,** inactive**};**

class Movement

**{**

**typedef** double **(**Movement**::\***PtrMovementF**)(**double**);**

// Type of pointer on a movement function

public**:**

// Constructor

Movement**();**

// Selection methods (NB : speed (unit/s) >0)

void selAff**(**double a**,** double b**,** double T**,** unsigned nRep **=** 0**);**

void selOsc**(**double a**,** double b**,** double phi**,** double T**,** unsigned nRep **=** 0**);**

// Access to private fields

double T**()** const **{return** \_T**;}**

void setT**(**double T**);**

StateMovement stateMov**()** const **{return** \_stateMov**;}**

void setStateMov**(**StateMovement stateMov**)** **{**\_stateMov **=** stateMov**;}**

double **operator()(**double t**)** **{return** **((\*this).\***\_pMovF**)(**t**);}** // Selected function

// Sampling with TS period

void start**();**

bool update**();**

void stop**();**

double val**()** const **{return** \_val**;}**

private**:**

// Movement functions

double Aff**(**double t**)** **{return** \_a**\*(**1**-**t**/**\_T**)+**\_b**\***t**/**\_T**;}** // Affine

double Osc**(**double t**)** **{return** \_a**\***sin**(**2**\***PI**\***t**/**\_T**+**\_phi**)+**\_b**;}** // Oscillation

// Parameters

double \_a**;**

double \_b**;**

double \_phi**;** // (radians)

double \_T**;** // Period(ms) > 0 (no control)

// Other fields

PtrMovementF \_pMovF**;** // Pointer on the selected function

StateMovement \_stateMov**;** // State movement

unsigned int \_nRep**;** // Number of repetitions

unsigned int \_iUse**;** // Movement use number (0, 1, ..., \_nUse)

unsigned long \_startTime**,** \_endTime**,** \_previousTime**,** \_stopTime**;**

double \_val**;** // Sampled value

**};**

It only implements 2 movement functions here, provided as an example, both of which will be used later:

* (**affine**)
* (**oscillation**)

But it's very easy to add more!

To select one of them, simply call the corresponding **Sel...** method, passing it its parameters, as well as a possible repetition factor[[23]](#footnote-24).

Therefore, by using the **operator()**, we can obtain the values ​​of this function for , or beyond if it is repeated.

This class also makes it possible to sample a movement with a period **TS**, fixed here at 20ms : for that, it suffices to call initially the **start** method, then **update** in the loop ; the latter is non- blocking and returns **true** when a new sample is available. In both cases, the **val** method provides the value of the movement.

You can also interrupt the movement by calling **stop**, then resume it by calling **start** again. Note that the update method returns **true** during the interrupt and that val does not change.

*Sketch* ***TestMovement.ino****:*

// TestMovement.ino

#include <Movement.h>

Movement m**;**

unsigned long oldTime**,** newTime**;**

byte state**;**

void print**(**double x**)**

**{**

Serial**.**print**(**x**);** Serial**.**print**(**'\r'**);**

**}**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

m**.**selOsc**(**2**,** 4**,** 0**,** 1000**,** 2**);** // Osc[2,4,0,1000] 3 times

m**.**start**();**

print**(**m**.**val**());**

oldTime **=** millis**();**

state **=** 0**;**

**}**

void loop**()**

**{**

**if** **(**m**.**update**())** print**(**m**.**val**());**

newTime **=** millis**();**

**if** **(**newTime**-**oldTime **>=** 500**)**

**{**

oldTime **=** newTime**;**

**if** **(**state**==**0**)** **{**m**.**setT**(**500**);** state**++;}** // Period/2

**else** **if** **(**state**==**1**)** **{**m**.**stop**();** state**++;}** // Stop

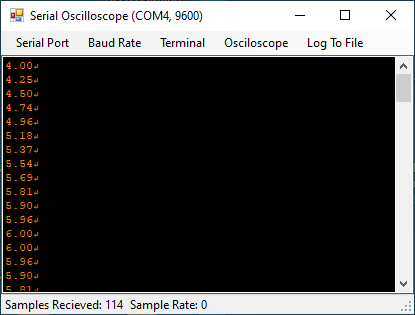
**else** **if** **(**state**==**2**)** **{**m**.**start**();** state**++;}** // Restart

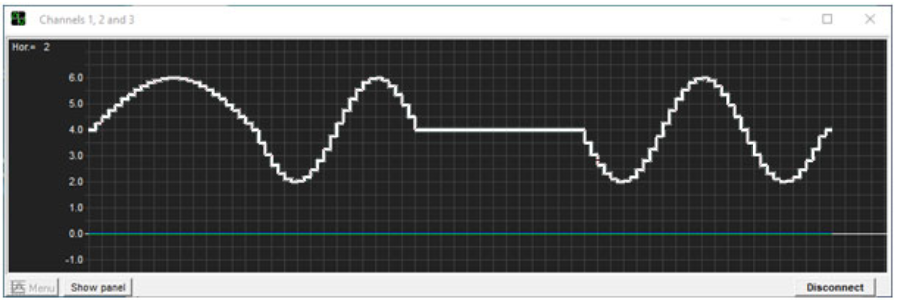
**}**

**}**

We consider here an oscillation, Osc[2, 4, 0, 1000], 3 times repeated, and sampled at 20ms. We start by dividing by 2 the period of 1000ms after 500ms, then we stop the movement after 500ms, and we finally resume it after 500ms

*Display of the results with* ***Serial Oscilloscope***[[24]](#footnote-25) :





*(NB : horizontal unit = 40ms)*

We can therefore very easily select, sample, interrupt and resume a movement, without having to manage the sampling.

## Extension of the Servo class

We are going to extend, by inheritance, the possibilities of the Servo class by offering it to be able to control the movement of a servo over time, with one of the parametric functions previously defined, and this in the least blocking way possible.

However, proceeding in this way has a small drawback : that of not knowing the exact position (= angle) of the servo, if the movement is interrupted, and this can be useful in certain movement functions. We will therefore permanently store this position in the **\_pos** field and implement a **move** method similar to **Servo::write**, to update it. This will also allow us to integrate the **trim** correction :

void move**(**double pos**);**

Initially, we wanted to introduce a systematic waiting period at the end of this method, to ensure that the target position is achieved. But we gave it up, because it disrupted the sampling. It will therefore be advisable to assess the risk of this situation occurring, and if necessary call on **delay** immediately afterwards, when it is called directly.

For this reason, it will be better to use instead :

void move**(**double pos**,** double T**);**

which will control the movement with a duration equal to **T** ms, using **Aff[\_pos, pos, T]** as the movement function. You will have to choose **T** large enough so that for each sampling (all **TS** = 20ms), we have plenty of time to complete the rotation of the arm, knowing that in 20ms we can rotate a maximum of 12°.

Obviously, as this class will be able to use the **Aff** and **Osc** parametric functions of the **Movement** class to control the movements of the servo over time. This can be done in a non-blocking way with the following methods which overload the methods of the same name of the **Movement** class and can be used identically :

void start**(**);

bool update**();**

void stop**();**

*Class* ***ServoE****(files* ***ServoE.h*** *and* ***ServoE.cpp****) :*

class ServoE **:** public Servo**,** public Movement

**{**

public**:**

// Constructors

ServoE**();**

ServoE**(**unsigned pin**,** double trim **=** 0.**);**

// Access to private fields (only necessary)

void setPin**(**unsigned pin**)** **{**\_pin **=** pin**;};**

void setTrim**(**double trim**)** **{**\_trim **=** trim**;};**

void setPos**(**double pos**)** **{**\_pos **=** pos**;};**

double pos**()** const **{return** \_pos**;};**

// Methods

void attach**();**

void detach**();**

// Blocking movement

void move**(**double pos**);** // Uncontrolled

void move**(**double pos**,** double T**);** // Controlled

// Non-blocking controlled movement

void start**();**

bool update**();**

void stop**();**

bool isInactive**();**

private**:**

unsigned \_pin**;** // Arduino pin where the servo is connected

double \_trim**;** // Trim value on writing (°)

double \_pos**;** // Current position (°)

**};**

*Sketch* ***TestServoE.ino****:*

// TestServoE.ino

#include <ServoE.h>

ServoE s**(**3**,-**5**);** // Servo on pin 3 with trim -5

unsigned long oldTime**,** newTime**;**

unsigned sta**;**

void print**(**double x**)**

**{**

Serial**.**print**(**x**);** Serial**.**print**(**'\r'**);**

**}**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

s**.**attach**();**

// Oscillation (x4) with 3000ms period

s**.**selOsc**(**90**,** 90**,** 0**,** 3000**,** 3**);**

s**.**start**();**

print**(**s**.**pos**());**

oldTime **=** millis**();**

sta **=** 0**;**

**}**

void loop**()**

**{**

**if** **(**s**.**update**())** print**(**s**.**pos**());**

newTime **=** millis**();**

**if** **(**newTime**-**oldTime **>=** 2000**)**

**{**

oldTime **=** newTime**;**

**if** **(**sta **==** 0**)** **{**s**.**setT**(**1000**);** sta**++;}** // Period/3 at 2s

**else** **if** **(**sta **==** 1**)** **{**s**.**stop**();** sta**++;}** // Stop at 4s

**else** **if** **(**sta **==** 2**)** **{**s**.**start**();** sta**++;}** // Restart at 6s

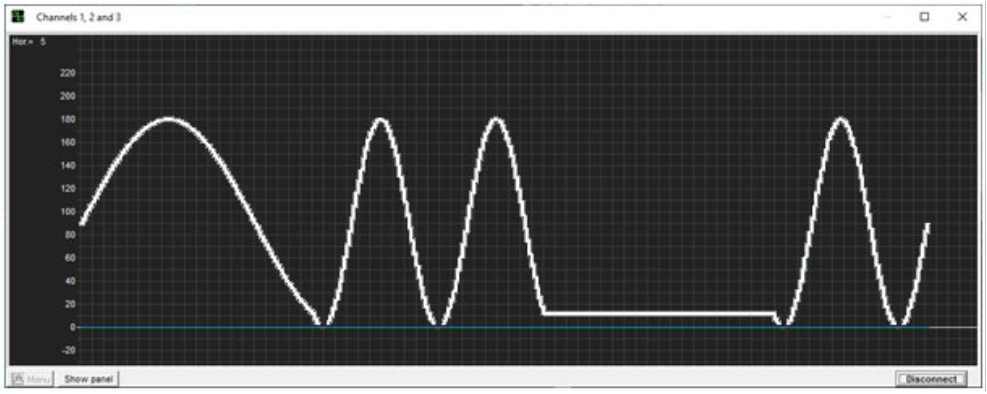
**}**

**if** **(**s**.**isInactive**())** s**.**detach**();**

**}**

We consider a servo on pin **3**, trimmed at **-5°**. Its movement is an oscillation defined by **Osc [90, 90, 0, 3000]**, repeated **3** times and sampled at **20ms**. We start by dividing its period at 2s by 3, then we stop the movement at 4s, then we resume it at 6s.

*Display of the results with* ***Serial Oscilloscope****:*



*(NB : horizontal unit = 100ms)*

## Movement of several servos in parallel

The **TServoE<N>** class responds to this problem by defining classical methods on an array of **N** servos, all of type **ServoE.**

*Class* ***TServoE<N>****(file* ***TServoE.hpp****)*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class TServoE<N>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Table of N extended servos (template)

Public methods :

TServoE() : constructor

TServoE(unsigned tPin[], double tTrim[] = NULL) : constructor

ServoE &operator[](unsigned i) : access to servos

void attach() : attach

void detach() : detach

void move(double tPos[], double T) : blocking controlled parallel movements

void start() : start or restart non-blocking controlled parallel movements

bool update() : update non-blocking controlled parallel movements

void stop() : stop non-blocking controlled parallel movements

bool isInactive() : test if all the servos are inactive

\*/

Since this is a **template** class, it is declared and implemented in a single file, **TServoE.hpp**, of which we only provide comments here.

The **operator[]** gives a reference on each servo in the table. By using it, you can for example select a movement function for it.

All the movement methods make the **N** servos turn in parallel by controlling them and there again, we have **start**, **update** and **stop** methods, to do it in a non-blocking way.

*Sketch* ***TestTServoE.ino****:*

// TestTServoE.ino

#include <TServoE.hpp>

unsigned tPin**[]** **=** **{**2**,**3**};**

double tTrim**[]** **=** **{-**12**,-**5**};**

TServoE**<**2**>** tS**(**tPin**,** tTrim**);**

unsigned long oldTime**,** newTime**;**

unsigned sta**;**

void print**(**double x**,** double y**)**

**{**

Serial**.**print**(**x**);** Serial**.**print**(**','**);** Serial**.**print**(**y**);** Serial**.**print**(**'\r'**);**

**}**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

tS**.**attach**();**

// Blocking controlled parallel movements to start positions

double tPos**[]** **=** **{**45**,** 180**};**

tS**.**move**(**tPos**,** 500**);**

// Non-blocking controlled parallel movements

tS**[**0**].**selOsc**(**45**,** 45**,** 0**,** 2000**,** 3**);**

tS**[**1**].**selOsc**(**45**,** 135**,** PI**/**2**,** 3000**,** 3**);**

tS**.**start**();**

print**(**tS**[**0**].**pos**(),** tS**[**1**].**pos**());**

oldTime **=** millis**();**

sta **=** 0**;**

**}**

void loop**()**

**{**

**if** **(**tS**.**update**())** print**(**tS**[**0**].**pos**(),** tS**[**1**].**pos**());**

newTime **=** millis**();**

**if** **(**newTime**-**oldTime **>=** 2000**)**

**{**

oldTime **=** newTime**;**

**if** **(**sta **==** 0**)** **{**tS**[**1**].**setT**(**1000**);** sta**++;}** // Period/3 at 2s

**else** **if** **(**sta **==** 1**)** **{**tS**[**1**].**stop**();** sta**++;}** // Stop at 4s

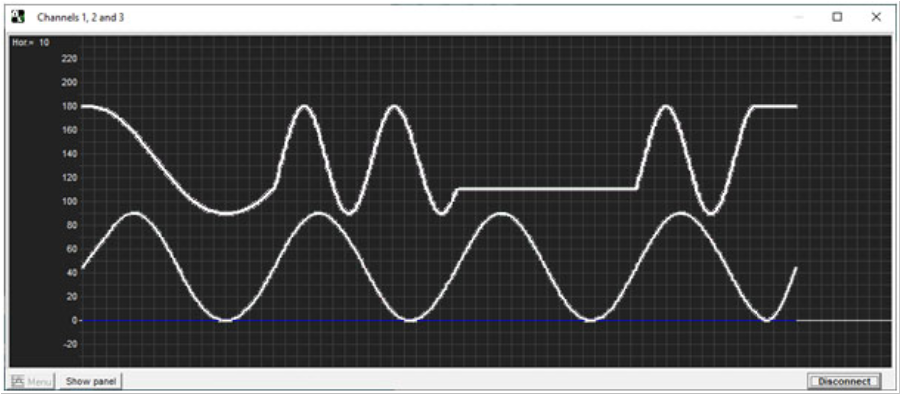
**else** **if** **(**sta **==** 2**)** **{**tS**[**1**].**start**();** sta**++;}** // Restart at 6s

**}**

**if** **(**tS**.**isInactive**())** tS**.**detach**();**

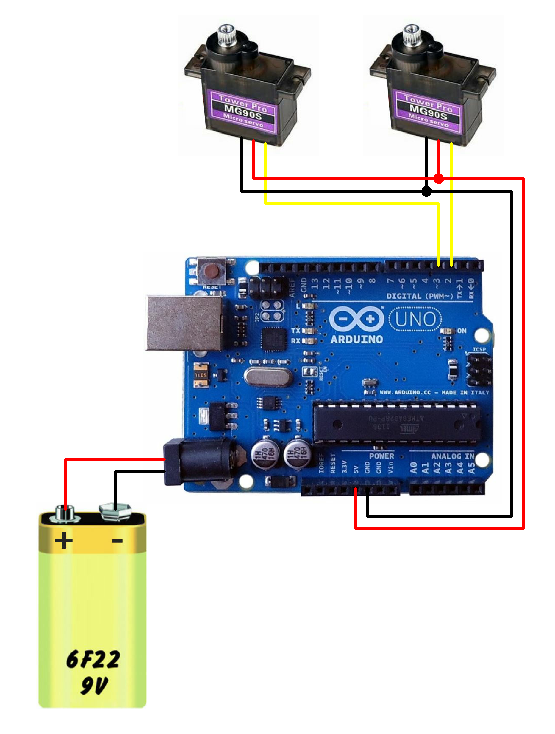
**}**

*Display of the results with* ***Serial Oscilloscope****:*

****

*(NB : horizontal unit = 200ms)*

*Connection diagram with an* ***Arduino Uno*** :



# Implementation with Otto

With these 3 classes **Movement**, **ServoE** and **TServoE<N>**, it is then very easy to define a Robot class to model Otto's movements : you just have to make it inherit from **TServoE<4>** !

*Class* ***Robot****(files* ***Robot.h*** *and* ***Robot.cpp****) :*

enum Part **{**LL**,** RL**,** LF**,** RF**};**

enum Direction **{**FORWARD**,** BACKWARD**};**

enum Side **{**LEFT**,** RIGHT**};**

enum Speed **{**SLOW**,** NORMAL**,** RAPID**};**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class Robot

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Robot class for OttoDIY with only legs and feet movement functionnalities

\*/

class Robot **:** public TServoE**<**4**>**

**{**

public**:**

Robot**(**unsigned pinLL**,** unsigned pinRL**,** unsigned pinLF**,** unsigned pinRF**,** bool calibrated **=** **true);** // Constructor

void setSpeed**(**Speed spd**);** // To change the movement speed

void move**(**double tPos**[]);** // Blocking controlled parallel movements to tPos

void home**();** // Blocking controlled parallel movements to {90,90,90,90}

void walk**(**Direction dir**,** unsigned int nRep **=** 0**);** // To start a walk non-blocking movement

void turn**(**Side sid**,** unsigned int nRep **=** 0**);** // To start a turn non-blocking movement

void walk\_turn**(**Direction dir**,** Side sid**,** unsigned int nRep **=** 0**);** // To start a walk\_turn non-blocking movement

private**:**

Speed \_spd**;**

void setMov**(**double tA**[],** double tB**[],** double tPhi**[],** unsigned int nRep**);**

**};**

Otto having 4 servos, we defined a constructor to which it is necessary to indicate the numbers of the used pins : 2, 3, 4, 5 by default. As in **OttoDIYLib**, its last argument, **calibrated**, indicates whether we can read the trim values ​​from EEPROM.

3 movement methods are provided as an example :

* The **walk** and **walk\_turn** methods are almost identical to the **walk** and **turn** methods of the **Otto9** class, with the addition of the possibility of turning backwards for the second ;
* The **turn** method corresponds here to a rotation of the robot in place, to the left or to the right.

Their implementations are analogous to those of the **Otto9** class ; for example :

void Robot**::**turn**(**Side sid**,** unsigned int nRep **=** 0**)**

**{**

double phi**;**

**switch** **(**sid**)**

**{**

**case** LEFT**:** phi **=** **-**PI**;** **break;**

**case** RIGHT**:** phi **=** 0**;** **break;**

**}**

float tA**[**4**]** **=** **{**15**,** 15**,** 20**,** 20**};**

float tB**[**4**]** **=** **{**105**,** 75**,** 94**,** 86**};**

float tPhi**[**4**]** **=** **{-**PI**/**2**,** **-**3**\***PI**/**2**,** phi**,** phi**};**

setMov**(**tA**,** tB**,** tPhi**,** nRep**);**

**}**

with :

void Robot**::**setMov**(**double tA**[],** double tB**[],** double tPhi**[],** unsigned int nRep**)**

**{**

double tPos**[**4**];**

**for** **(**unsigned i**=**0**;** i**<**4**;** i**++)** tPos**[**i**]** **=** tA**[**i**]\***sin**(**tPhi**[**i**])+**tB**[**i**];**

move**(**tPos**);**

double x **=** 30.**;**

double T **=** 4000**\***x**/**tSpeed**[**\_spd**];**

**for** **(**unsigned i**=**0**;** i**<**4**;** i**++)** **(\*this)[**i**].**selOsc**(**tA**[**i**],** tB**[**i**],** tPhi**[**i**],** T**,** nRep**);**

start**();**

**}**

It should be noted that such movements therefore break down into :

* An affine blocking movement to rotate the servos to the starting positions of the oscillatory movement, in order to increase the fluidity of the movement ;
* A non-blocking oscillatory movement, with the appropriate parameters.

These 2 movements are controlled by a speed parameter **\_spd**, from which we deduce the periods **T**, common for the 4 servos of the robot, which are therefore well synchronized. The **setSpeed** ​​method allows you to change it « hot ».

*Sketch* ***TestRobot.ino****:*

// TestRobot.ino

#include <Robot.h>

Robot r**(**2**,** 3**,** 4**,** 5**);**

void setup**()**

**{**

r**.**attach**();**

r**.**home**();**

r**.**walk**(**FORWARD**,** 5**);**

**}**

void loop**()**

**{**

r**.**update**();**

**if** **(**r**.**isInactive**())**

**{**

r**.**home**();**

r**.**detach**();**

**}**

**}**

# Controlling the robot via a Bluetooth connection

## Classes SerialCommand and RobotSC

The essential thing to understand here is that the problem is to control Otto via a TTL serial link[[25]](#footnote-26), for example the one established with an USB connection, using the **Serial** object, instance of the **HardwareSerial** class.

Indeed, Otto's Bluetooth module is connected to pins 12 (RX) and 11 (TX) of the Arduino Nano, with establishment of a TTL serial link on them, using an instance of the **SoftwareSerial[[26]](#footnote-27)** class.

However, if we look at the definitions of these 2 classes[[27]](#footnote-28) :

class HardwareSerial **:** public Stream

class SoftwareSerial **:** public Stream

we see that they both inherit from the **Stream** class and notably implement its virtual methods :

virtual int available**()** **=** 0**;**

virtual int read**()** **=** 0**;**

Thanks to the polymorphism of C ++, we can treat the two cases in a generic way, by considering an instance (in fact a reference to avoid an unnecessary copy) of the **Stream** class, which can therefore accept an instance object of the **HardwareSerial** class as well as **Serial**, as an instance object of the **SoftwareSerial** class (see below), and of course call the previous methods. This is what we will do in the **SerialCommand** class defined as follows.

*Class* ***SerialCommand*** *(files* ***SerialCommand.h*** *and* ***SerialCommand.cpp****) :*

class SerialCommand

**{**

public**:**

SerialCommand**(**Stream**&** serial**,** int nCmd**,** Command tCmd**[]);**

char typ**()** const **{return** \_typ**;}**

unsigned opt**()** const **{return** \_opt**;}**

void resetParse**();**

bool parseCommand**();**

private**:**

Stream**&** \_serial**;** // Serial port used for commands

unsigned \_nCmd**;** // Number of acceptable commands

Command **\***\_tCmd**;** // Array of acceptable commands (as reference)

bool \_okCommand**;** // True if a command as been correctly parsed

char \_typ**;** // Type of this command

unsigned \_opt**;** // Option of this command

unsigned \_nParse**;** // Number of parsed characters

**};**

The purpose of this class is to analyze the commands received on the **serial** Stream passed to its constructor. These must respect the following syntax and semantics:

Syntax :

<command> = <typ><opt>#

<typ> = <letter>

<letter> = A|B|...|Z

<opt> = <digit>|<digit><digit>

<digit> = 0|...|9

Semantic :

(<typ>,<opt>) in tCmd

The **tCmd** array of acceptable commands, a parameter of the constructor, is here defined by :

Command tCmd**[]** **=** // Acceptable commands

**{**

**{**'C'**,**2**,{**1**,**0**}},** // Contact : 1=>on, 0=>off

**{**'M'**,**9**,{**0**,**1**,**2**,**3**,**4**,**13**,**14**,**23**,**24**}},**

// Movement : 0=>home, 1/2=>walk FORWARD/BACKWARD, 3/4=>turn LEFT/RIGHT, 13=>walk FORWARD and turn LEFT, etc.

**{**'I'**,**2**,{**1**,**0**}},** // Interruption : 1=>on, 0=>off

**{**'S'**,**3**,{**1**,**2**,**3**}}** // Speed : 1=>SLOW, 2=>NORMAL, 3=>RAPID

**};**

each being of type **Command[[28]](#footnote-29)** :

struct Command **{**char typ**;** unsigned nOpt**;** unsigned tOpt**[**MAX\_OPTIONS**];};** // Acceptable command

For instance :

* M0# => home
* M1# => walk forward
* M3# => left turn (at the place)
* M13# => walk forward and turn left
* Etc.

As we do not always control the production of ‘\ n’ (line feed) or ‘\ r’ (carriage return), these characters will be ignored and we preferred to use ‘**#**’ to indicate the end of a command.

The main method of this class is responsible for decoding the commands received :

bool parseCommand**();**

Its implementation is that of a classic interpreter, which returns **true** on receiving one of them ; all you have to do is call the **typ** and **opt** methods to find out their type and option.

And that's what we do in the unique **processCommand** method of the **RobotSC** class, which inherits from the **Robot** class and the previous class.

*Classe* ***RobotSC*** *(files* ***RobotSC.h*** *and* ***RobotSC.cpp****) :*

class RobotSC **:** public Robot**,** SerialCommand

**{**

public**:**

RobotSC**(**int pinLL**,** int pinRL**,** int pinLF**,** int pinRF**,** Stream**&** serial**,** bool calibrated **=** **true);**

void processCommand**();**

**};**

void RobotSC**::**processCommand**()**

**{**

**if** **(**parseCommand**())**

**{**

resetParse**();**

unsigned \_opt **=** opt**();**

**switch** **(**typ**())**

**{**

**case** 'C'**:**

**if** **(**\_opt**==**1**)** attach**();**

**else** **if** **(**\_opt**==**0**)** detach**();**

**break;**

**case** 'M'**:**

**if** **(**\_opt**==**0**)** home**();**

**else** **if** **(**\_opt**==**1**)** walk**(**FORWARD**,** REP**);**

**else** **if** **(**\_opt**==**2**)** walk**(**BACKWARD**,** REP**);**

**else** **if** **(**\_opt**==**3**)** turn**(**LEFT**,** REP**);**

**else** **if** **(**\_opt**==**4**)** turn**(**RIGHT**,** REP**);**

**else** **if** **(**\_opt**==**13**)** walk\_turn**(**FORWARD**,** LEFT**,** REP**);**

**else** **if** **(**\_opt**==**14**)** walk\_turn**(**FORWARD**,** RIGHT**,** REP**);**

**else** **if** **(**\_opt**==**23**)** walk\_turn**(**BACKWARD**,** LEFT**,** REP**);**

**else** **if** **(**\_opt**==**24**)** walk\_turn**(**BACKWARD**,** RIGHT**,** REP**);**

**break;**

**case** 'I'**:**

**if** **(**\_opt**==**1**)** stop**();**

**else** **if** **(**\_opt**==**0**)** start**();**

**break;**

**case** 'S'**:**

**if** **(**\_opt**==**1**)** setSpeed**(**LOW**);**

**else** **if** **(**\_opt**==**2**)** setSpeed**(**NORMAL**);**

**else** **if** **(**\_opt**==**3**)** setSpeed**(**RAPID**);**

**break;**

**}**

**}**

**}**

Note here the **resetParse** call to reset the interpreter, as well as the constant value of the **REP** parameter, set to 100 to simplify the sending of commands.

## Controlling the Robot with a serial monitor, via an USB link

*Sketch* ***TestRoboSC.ino****:*

// TestRobotSC.ino

#include <RobotSC.h>

RobotSC r**(**2**,** 3**,** 4**,** 5**,** Serial**);**

void setup**()**

**{**

Serial**.**begin**(**9600**);**

delay**(**100**);**

**}**

void loop**()**

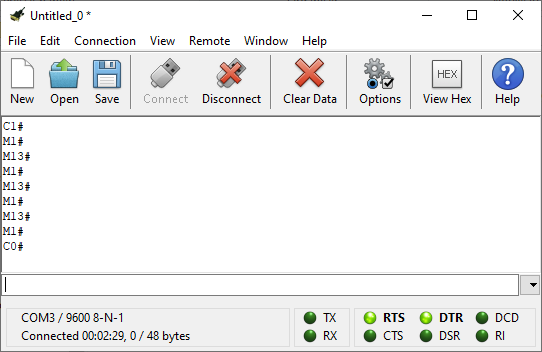
**{**

r**.**processCommand**();**

r**.**update**();**

**}**

Here is an example of commands transmitted with the **Coolterm**[[29]](#footnote-30) serial monitor, to rotate the robot around a cube :



## Controlling the Robot with a Smartphone, via a Bluetooth link

For the Arduino sketch, as announced, you just need to replace **Serial** with an instance, **BTSerial** here, of the **SoftwareSerial** class.

*Sketch* ***TestRobotBT.ino****:*

// TestRobotBT.ino

#include <RobotSC.h>

#include <SoftwareSerial.h>

SoftwareSerial BTSerial**(**11**,**12**);**

RobotSC r**(**2**,** 3**,** 4**,** 5**,** BTSerial**);**

void setup**()**

**{**

BTSerial**.**begin **(**9600**);**

delay**(**100**);**

**}**

void loop**()**

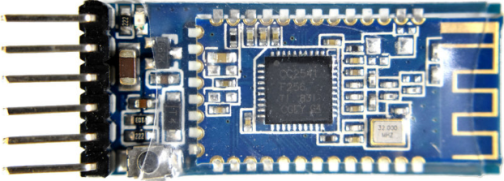
**{**

r**.**processCommand**();**

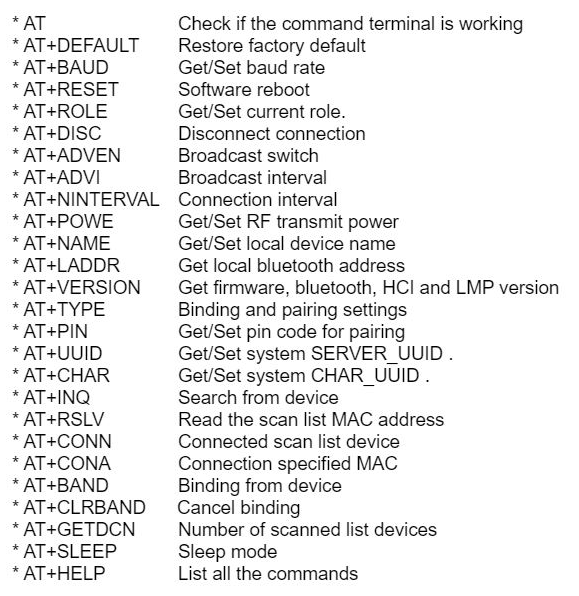
r**.**update**();**

**}**

A Bluetooth connection must then be established between the module provided here and the Smartphone.



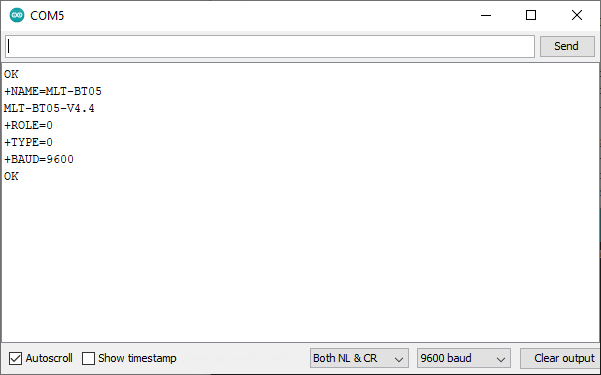
This « Low Energy 4.0 » Bluetooth module is compatible with an **MLT-BT05** and built around a **CC2541** component from Texas Instrument (or clone ...). It can be configured with **AT** commands[[30]](#footnote-31) [[31]](#footnote-32).



The easiest way to set it up is to use the Arduino IDE serial monitor, connected at 9600 baud, and the **Otto\_BlueTest.ino** sketch, provided in **OttoDIYLib**.

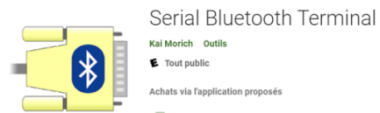
Here are the responses obtained by sending the following AT commands (with the factory settings, only the serial port speed needs to be changed):

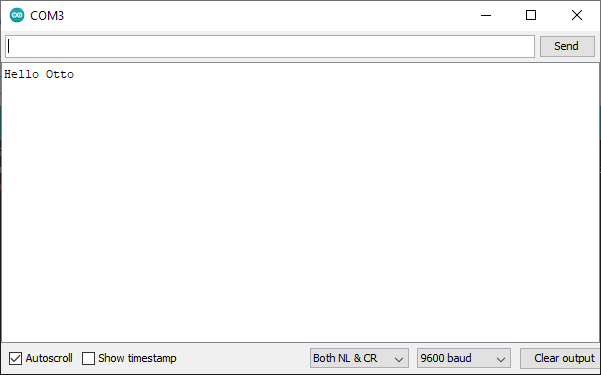
* **AT+DEFAULT** // Restore factory settings
* **AT+NAME** // Display name
* **AT+VERSION** // Display version
* **AT+ROLE0** // Set role (NB : 0 => Slave)
* **AT+TYPE0** // Set type (NB : 0 => no password)
* **AT+BAUD4** // Set serial port speed (NB : 4 => 9600 bauds)

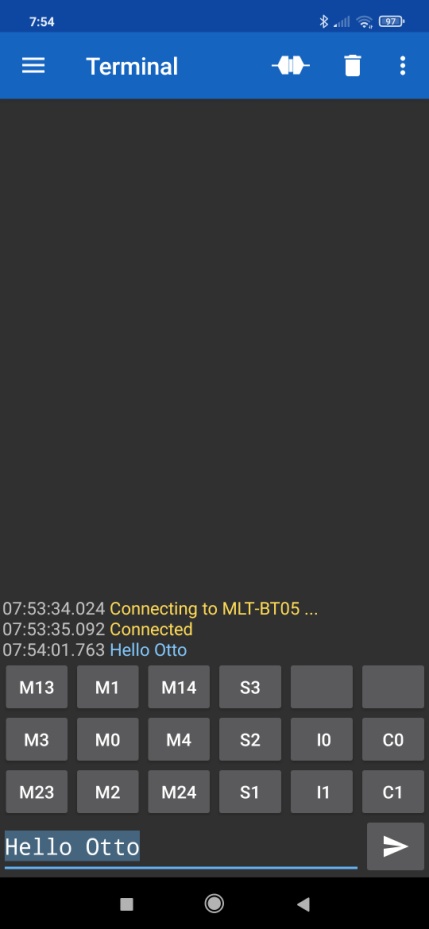


The module can then be recognized on a Smartphone[[32]](#footnote-33) accepting the Bluetooth Low Energy 4.0 protocol.

To be convinced, after activating this protocol, all you have to do is install the Android **Serial Bluetooth Terminal** application, available on Google Play[[33]](#footnote-34):

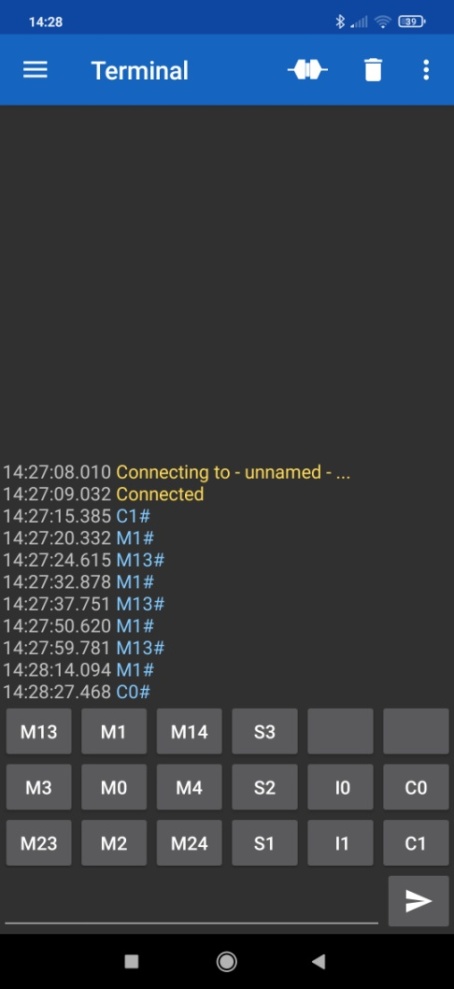
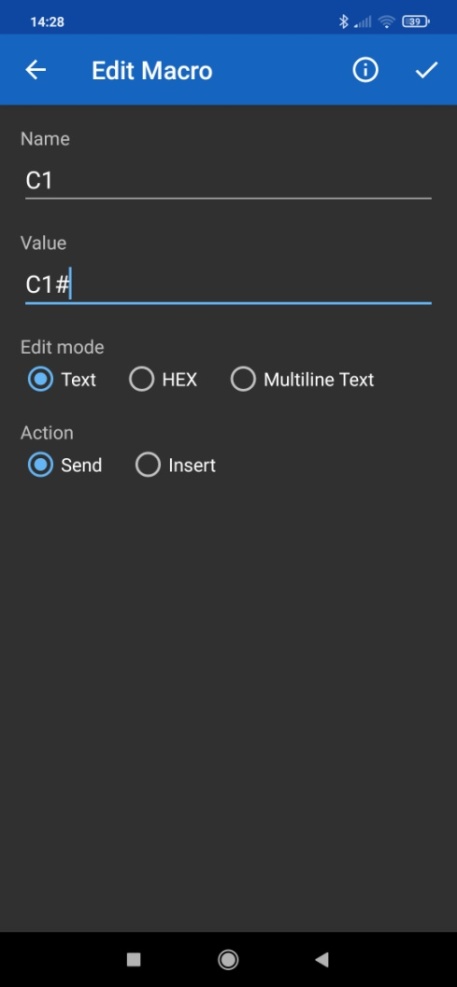




If we now install the **TestRobotBT.ino** sketch in the robot, we can control it as we did previously with a serial monitor.

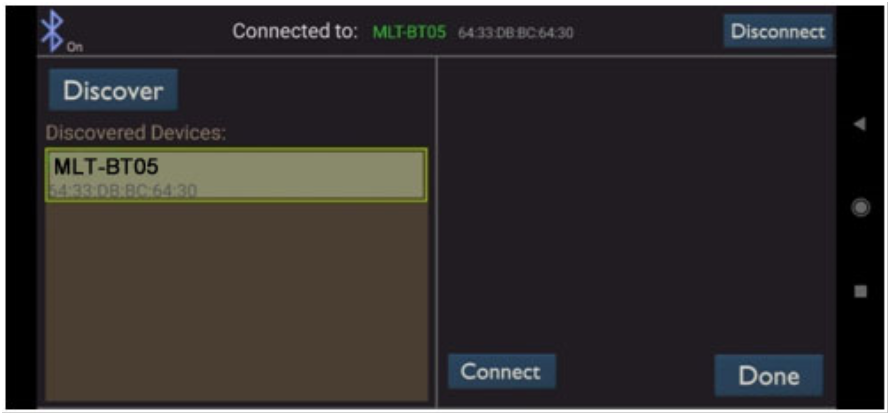
To make it easier to send commands, you can assign macros and labels to application buttons.

If you want a more « fun » interface, you can use the **Bluetooth Electronics[[34]](#footnote-35)** application, and build the interface by dragging and dropping its components :

****

Here again, the Bluetooth module is recognized when you activate Bluetooth in the Smartphone[[35]](#footnote-36):



And then all you have to do is to compose a screen in « Edit » mode where, again, you can assign macros to the elements of the interface :



then switch to « Run » mode to control the robot.

Note, however, a few bugs, especially if we use ‘\ n’ as end of line : this is why we preferred ‘#’.

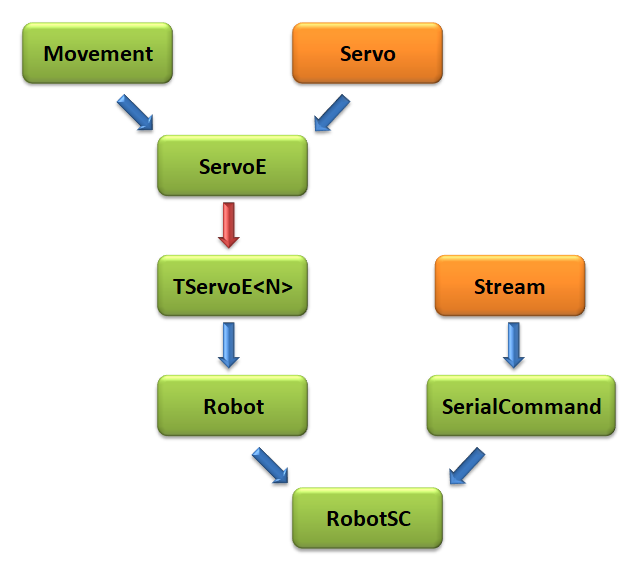
***Remarks***

* The executables produced with the 2 previous sketches are largely compatible with the memory possibilities of an Arduino Nano: they only occupy about 30% of the flash memory and the available SRAM memory ;
* There is therefore room to take into account other sensors : microphone, capacitive sensor (sensitive to touch), distance sensor (sonar), etc. They can be integrated as we just did with a serial port, since the latter is used here as a character « sensor ».
* This functionality can also be extended to the case of a text file or a network flow of commands, since this would still amount to implementing class instance objects inheriting from the **Stream** class. But here, the **Blockly** approach proposed with **Otto** is much more attractive and it would not be very difficult to adapt it to produce code compatible with our **Robot** class.
* The simplicity of this implementation is only possible with non-blocking processes, both at the level of the servos and at the level of the sensors. Otherwise, we would have to resort to interrupt processing, which we wanted to avoid here so as not to disturb the proper functioning of the timers and in particular **timer1**, used in the **Servos** library.

# Conclusion

This study allowed us to understand how Otto moves and in particular the importance of the oscillatory movements studied by **Juan González Gómez**.

To meet the objectives stated at the end of paragraph 3, we have defined an Arduino library of C ++ classes (in green below), with the following dependencies, our primary concern always being to implement the targeted functionalities as soon as possible, to so as to simplify the dependent classes as much and in the most general way possible.



In this simplified diagram[[36]](#footnote-37), we can see the importance of the **Movement** class on which all other classes depend. It generically defines the notion of movement, without specifying the specific function chosen, and it encapsulates the entire process of sampling a movement, a key notion to then achieve controlled, non-blocking and parallel movements on several servos.

We have used all the power of C ++[[37]](#footnote-38) and object-oriented programming (inheritance, composition, polymorphism, genericity, pointer to a class method, operators, template) to simply implement the targeted functionalities. This allowed us to produce very light executables, well suited to the low memory capacities of the Arduino Nano contained in Otto.

Tools such as **WinPython** and **Serial oscilloscope** have been used to produce graphics : they deserve to be known !

As our goals were only for Otto's leg and foot movements, we didn't want to take into account all of Otto's features and especially those related to its sensors. But, with the concepts implemented, this would not be difficult.

Finally, we would like to thank all of Otto's designers for inspiring us with this work and in particular **Camilo Parra Palacio** for the information he kindly transmitted to us.

1. Named only Otto, to simplify. [↑](#footnote-ref-2)
2. Département Informatique - Institut Universitaire de Technologie de Nice-Côte d’Azur, especially. [↑](#footnote-ref-3)
3. [Micro Servo Motor MG90S - Tower Pro (electronicoscaldas.com)](http://www.electronicoscaldas.com/datasheet/MG90S_Tower-Pro.pdf) [↑](#footnote-ref-4)
4. [Servos Explained - SparkFun Electronics](https://www.sparkfun.com/servos) [↑](#footnote-ref-5)
5. [Pulse-width modulation - Wikipedia](https://en.wikipedia.org/wiki/Pulse-width_modulation) [↑](#footnote-ref-6)
6. [GitHub - arduino-libraries/Servo: Servo Library for Arduino](https://github.com/arduino-libraries/Servo) [↑](#footnote-ref-7)
7. An Arduino Nano, with an **ATMega328P**, has 3 programmable timers, **timer0** (8bits), **timer1** (16bits) and **timer2** (8bits), which by default control the PWM outputs on pins 5/6 (with **timer0** to 980Hz), 9/10 (with **timer1** at 490Hz) and 3/11 (with **timer2** at 490Hz), these can be generated with the **analogWrite (…)** method. [↑](#footnote-ref-8)
8. With time zoom. [↑](#footnote-ref-9)
9. Approximatively 300ms for 180° when 5V powered. [↑](#footnote-ref-10)
10. **detach()** stops the rotation, without maintaining an angle. [↑](#footnote-ref-11)
11. Graphic made with **WinPython** ([WinPython download | SourceForge.net](https://sourceforge.net/projects/winpython/)) and the program :

    # Plot.py

    # Libraries

    **from** numpy **import** pi**,** sin**,** linspace

    **import** matplotlib**.**pyplot **as** plt

    # Time values

    t **=** linspace**(**0**,** 1000**,** 100**)**

    #Plots

    plt**.**plot**(**t**,** 30**\***sin**(**2**\***pi**\***t**/**1000**)+**90**,** label**=**"αLL(t)=αRL(t)=30sin⁡(2πt/1000)+90"**)**

    plt**.**plot**(**t**,** 20**\***sin**(**2**\***pi**\***t**/**1000**-**pi**/**2**)+**94**,** label**=**"αLF(t)=20sin⁡(2πt/1000-π/2)+94"**)**

    plt**.**plot**(**t**,** 20**\***sin**(**2**\***pi**\***t**/**1000**-**pi**/**2**)+**86**,** label**=**"αRF(t)=20sin⁡(2πt/1000-π/2)+86"**)**

    plt**.**axhline**(**y**=**90**,** linewidth**=**1**,** color**=**'k'**)**

    plt**.**xlabel**(**"Time (ms)"**)**

    plt**.**ylabel**(**"Angles"**)**

    plt**.**title**(**"Otto walk FORWARD movement"**)**

    plt**.**legend**()**

    plt**.**show**()** [↑](#footnote-ref-12)
12. Cf. **Otto9.h** and **Otto9.cpp** in [GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots](https://github.com/OttoDIY/OttoDIYLib) [↑](#footnote-ref-13)
13. Cf. **Oscillator.h** and **Oscillator.cpp** in [GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots](https://github.com/OttoDIY/OttoDIYLib) [↑](#footnote-ref-14)
14. [GitHub - fitzterra/ServoOsc: Servo oscillator library for Arduino](https://github.com/fitzterra/ServoOsc) [↑](#footnote-ref-15)
15. [Juan Gonzalez:Tesis - WikiRobotics (iearobotics.com)](http://www.iearobotics.com/wiki/index.php?title=Juan_Gonzalez:Tesis#Experimentos_con_robots_reales) [↑](#footnote-ref-16)
16. [GitHub - Obijuan/ArduSnake: Arduino modular snake robots library. Generate the locomovement of snake robots easily!](https://github.com/Obijuan/ArduSnake) [↑](#footnote-ref-17)
17. [GitHub - bqlabs/zowi: An open-source and fully hackable biped robot.](https://github.com/bqlabs/zowi) [↑](#footnote-ref-18)
18. We will pass it the value of the angle (read in the EEPROM) resulting from the robot calibration procedure.  [↑](#footnote-ref-19)
19. Cf. **Otto9.h** in [GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots](https://github.com/OttoDIY/OttoDIYLib) [↑](#footnote-ref-20)
20. Via the **\_execute** method. [↑](#footnote-ref-21)
21. Cf. **Otto9.cpp** in [GitHub - OttoDIY/OttoDIYLib: Latest Libraries for Otto DIY robots](https://github.com/OttoDIY/OttoDIYLib) [↑](#footnote-ref-22)
22. Arduino STL libraries, such as **ArduinoSTL** and **StandardCplusplus**, also do not support the **std::function**. Note however the existence of an STL library which implements **std::function** ([GitHub - fopeczek/arduino-function\_objects](https://github.com/fopeczek/arduino-function_objects)), but for **platformIO**. Preferring to stay within the framework of the **Arduino IDE**, we have not tested it. [↑](#footnote-ref-23)
23. NB : 0 repetition => the movement is used 1 time. [↑](#footnote-ref-24)
24. <https://x-io.co.uk/serial-oscilloscope/> (NB : the graphic color has been changed to make it more readable). [↑](#footnote-ref-25)
25. Here logic level = 5V for the Arduino Nano, the line being idle in the LOW state (0V). [↑](#footnote-ref-26)
26. An Arduino Uno or Nano has only one hardware serial port, connected to pins 0 (RXD) and 1 (TXD) and used during a USB connection. It is for this reason that these pins are not connected to a component of the Robot. However, the **SoftwareSerial** library ([Arduino - SoftwareSerial](https://www.arduino.cc/en/Reference/SoftwareSerial)) allows this functionality to be extended by software by creating other serial ports on other pins, but with restrictions. [↑](#footnote-ref-27)
27. In the Arduino IDE installation folder:

    <Arduino>/hardware/arduino/avr/cores/arduino/HardwareSerial.h

    <Arduino>/hardware/arduino/avr/libraries/SoftwareSerial/src/HardwareSerial.h [↑](#footnote-ref-28)
28. Here MAX\_OPTIONS = 9. [↑](#footnote-ref-29)
29. <http://freeware.the-meiers.org/> with **Line Mode** and **Local Echo** as options. [↑](#footnote-ref-30)
30. For instance : [Bluetooth Module with Arduino (AT-09, MLT-BT05, HM-10) — Maker Portal (makersportal.com)](https://makersportal.com/blog/2019/10/14/bluetooth-module-with-arduino) [↑](#footnote-ref-31)
31. [MLT-BT05-AT-commands-TRANSLATED.pdf (wlu.ca)](http://denethor.wlu.ca/arduino/MLT-BT05-AT-commands-TRANSLATED.pdf) [↑](#footnote-ref-32)
32. Xiaomi MI9 SE with Android 10 (MIUI 12.0.4) here. [↑](#footnote-ref-33)
33. [Serial Bluetooth Terminal – Applications on Google Play](https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal&hl=fr&gl=US) [↑](#footnote-ref-34)
34. [Bluetooth Electronics (keuwl.com)](https://www.keuwl.com/apps/bluetoothelectronics/) [↑](#footnote-ref-35)
35. With Localisation… [↑](#footnote-ref-36)
36. Not UML ! Blue arrows => inheritance and red arrow => composition. [↑](#footnote-ref-37)
37. But limited to those taken into account by the Arduino IDE compiler for an Arduino Nano. [↑](#footnote-ref-38)