High-precision low-cost micro controlled syringe pump for flow analysis devices

J. I. Otero-Bueno, J. D. Mozo*

Applied Electrochemistry Laboratory. Factulty of Experimental Sciences. University of Huelva. Av. 3 de Marzo, s/n 21071 Huelva. SPAIN

* Corresponding author: Juan Daniel Mozo Llamazares
e-mail: jdaniel.mozo@diq.uhu.es
phone: +34 959 219992
fax #: +34 959 219983

ABSTRACT

The development of a micro-controlled syringe pumping system is presented. We describe the mechanical layout, the electronics for manual control, the micro computer programming and the remote control procedure. A communication protocol is implemented allowing remote control and fully programmable operation: linear flow gradients and complex sequences, control of multiple pumps for continuous flow or mixtures of fluids, and more. All documentation it's fully accessible and both, hardware and software are open code. The final cost is very affordable and will save about 75% on commercially available devices with comparable features.

KEYWORDS

Arduino, flow injection analysis, open code, free software, programmable device

INTRODUCTION

The flow analysis systems have become the most growing technological area of chemical analysis because it is the approach that better permeates the automation and notably increases the performance of procedures respecting to higher analytical throughput or lower measurement dispersions. At present most of the literature on new analytical systems are based on some version of flow systems and there are publications specifically dedicated to this topic.

These flow systems, obviously, require a fluid driving system which must have certain characteristics to be adequate. Among them we must mention the need to maintain a that the flow must be steady flow in time, and such flow was as free as possible of artifacts such as pulsations. It is further desirable that the flow could can be adjusted over a wide range, to drive fluid continuously, and it can be programmed or synchronized to expand the device applicability to different measurement protocols.

Mainly two types of pumping systems are used in flow analyzers: peristaltic pumps and syringe pumps. The first ones consist of a rotor with several rollers pressing a flexible tube, driving the liquid inside through its rotation. They can continuously drive liquid from a reservoir but the shift of a roller to another generates an unwanted variation in the flow that can be registered at the detector as noise. The flow can be adjusted by selecting the flexible tube inner diameter and by controlling the rotational speed of rotor.

The syringe pumps basically consist of a given volume syringe, whose plunger is driven by a linear motor. The total driven volume is only the volume loaded in the syringe and the flow is virtually free of noise. The flow can be adjusted by selecting the syringe size and the motor speed. This type of pumps can be more compact and lightweight, so they are used for medical applications to continuously dispense controlled doses of drugs.

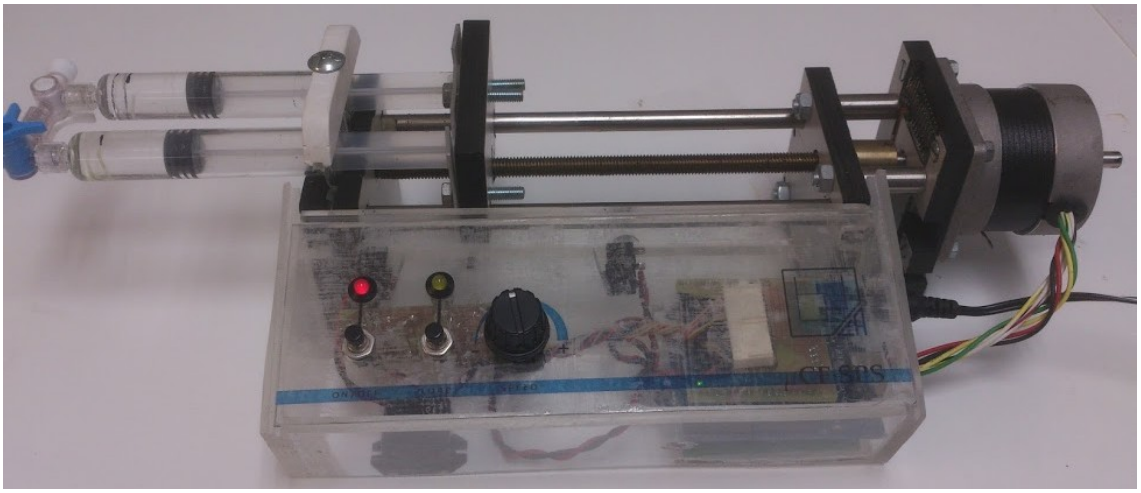
There are a number of manufacturers selling a wide range of syringe pumps depending on the application and user needs. Therefore there are manual controlled versions, OEM, multichannel, programmable, micro-flow, high pressure, and so; prices ranged from US$ 250 to 5000 and more.

The syringe pump structure is very simple and it can be easily constructed with a minimum of work, some manual ability and a small spending on materials. The electronic control of motor operation allows achieving good levels of precision and accuracy when in adjusting the flow and also the micro-controlled operation enables sequence programming and remote/centralized control.

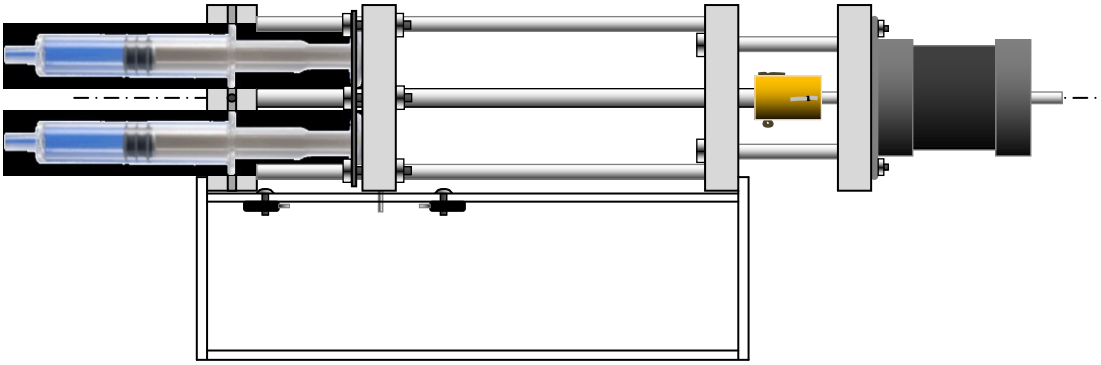
The aim of this work is to illustrate how to make a syringe pump from scratch; . W which technical considerations must be made at each stage, how to build the hardware, which electronics are needed, how to implement a micro-processor, how and why to communicate with a host and how to implement the control software. This can be useful for chemical/clinical labs without resources or with very specific flow control requirements. It could also serve as a comprehensive building project for engineering students, including mechanical and electronic hardware manufacturing, software programming and communications.

derived works, in part or completely, as long as you also adopt the same license and only for non commercial uses.

All programming tools used in this work are Free Software.



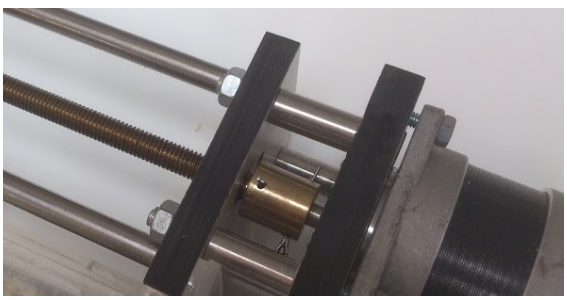
MECHANICAL SETUP



The main component of the mechanical setup is the linear motor, a device capable of moving ~~or move~~ something along a direction. Its choice can determine both the mechanical and electronic setup to ensure the technical specifications required for a syringe pump. A cheaper ~~cheapest~~[M1] option to a linear motor is a rotary motor equipped with a linear actuator to transform the rotary motion to linear. There are several possibilities ~~to choose~~ when choosing the motor type; all of them ~~they~~ allow precise control of its movement both as to its length and its speed: the servo, the DC and the steeper motors.

In DC motors, the cheapest option, the rotation speed is proportional to the supply potential. So, by ~~controlling~~ changing the potential, the motor movement can be controlled. If torque exceeds the motor capacity, information on motion control is lost because motion doesn't follow potential[M2]. To avoid this you need to include some position sensor that allows full control, this is what the servomotors do. The encoder is integrated in the servo and a relatively sophisticated controller is needed, often a dedicated module specifically designed for each servo. All this extra technology represents a significant increase in the final cost.

The stepper motors represents the intermediate solution. We can accurately control the angle and direction of rotation and a generic driver can be used with any of such engines; so, a good enough control is achieved with a reasonable cost. The overload trouble is not completely avoided by working with this type of engines, so it's advisable to design oversized systems capable of pushing liquid even in the worst conditions. We have selected the RS Components part nr. 440-458 (bill of materials' part nr. uCF0001), a hybrid stepper motor type powered with 12Vdc@0.6A. This motor is able to rotate in steps of 1.8 degrees (full step, 257 steps by revolution) or 0.7 degrees (half step, 514 steps by revolution).



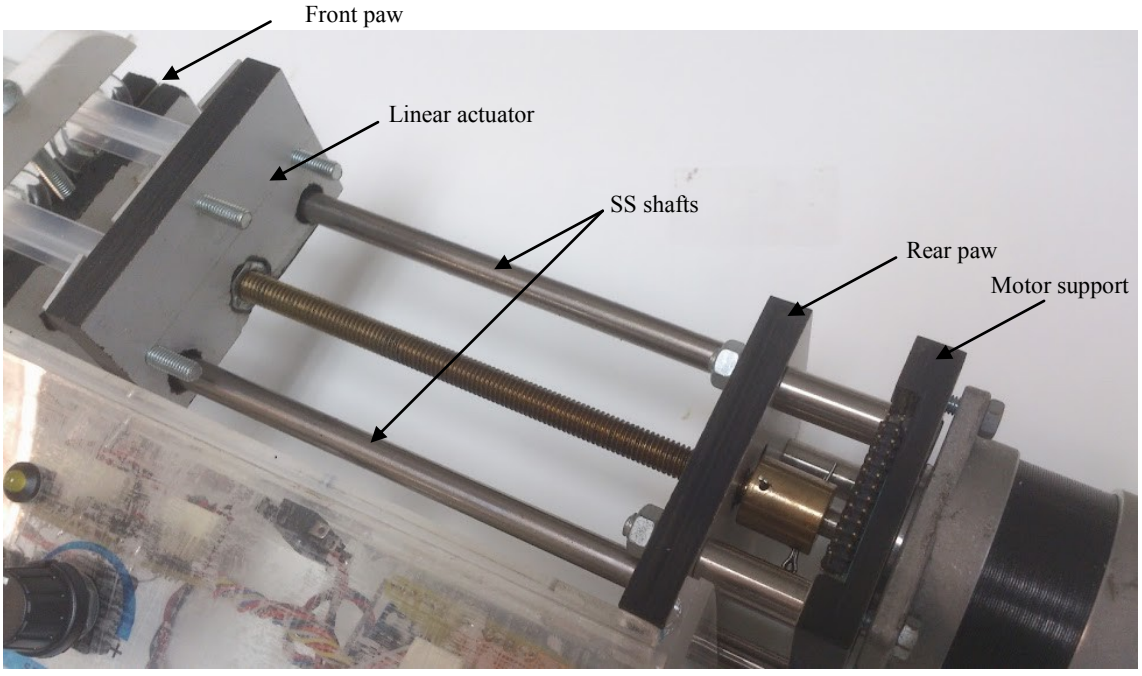
To translate the rotary motion of motor into a linear motion we have designed a linear actuator. The main component is a threaded rod M8 (bill of materials' part nr. uCF0002) attached to the motor shaft with a home-made adapter (bill of materials' part nr. uCF0003). Both adapter and rod were made of brass though other materials can be used. The threaded rod rotates together with the motor shaft and acts as a worm screw. So, as the rod's thread is M8, the thread pitch is 1.25 mm and each motor half-step turn (0.7º) implies a linear

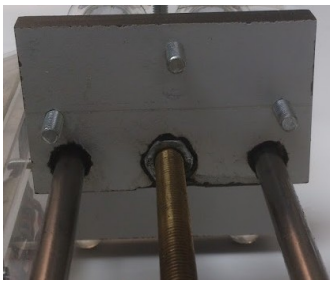
motion of 2.43 microns for a nut (bill of materials' part nr. uCF0004) that is inserted in the screw. Although the threaded rod can be purchased at any hardware store, both ends must be machined for proper placement in the holder and to connect it to motor.

The adapter for attaching the motor shaft and the threaded rod is a cylindrical piece made on a lathe from a brass rod. The piece was drilled along its axis, so that the motor shaft fits into the bore on one side and the rod on the other. To prevent the parts to rotate freely, they have been made two small through-holes in which a split pin is inserted. These cross the adapter from side to side along with the shafts, which have been also drilled perpendicularly. The central bore of the adapter was machined slightly wider than the shafts should host. This clearance allows a small elbow-joint, thus facilitating the placement of the shafts and relaxing their mutual alignment.

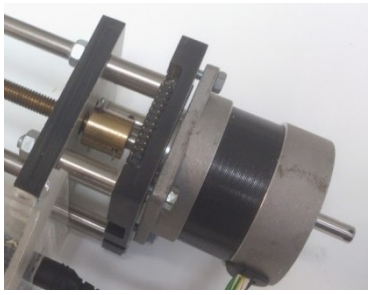
To support the motor/worm screw assembly and lining up both, it has been designed a holder which also facilitates the movement of an actuator attached firmly to the aforementioned nut inserted in the worm screw. The structure consists on a set of pieces of plastic material (bill of materials' parts nr. uCF0006 to uCF0009) that can be cut from a 1 cm thick griddle with an electric jigsaw and a guide. Alternatively they can be printed in 3D from their CAD-CAM models.

Two of pieces are the structure paws and, furthermore the worm, they supports two parallel stainless steel shafts that guide the movement of the linear actuator avoiding its rotation. The linear actuator is driven by the rotary motion of the worm screw, so the screw must be secured to avoid shifting back and forth, but ~~must be~~ still being able to rotate on its axis. The guide shafts are firmly attached to the paws by grub screws and must be accurately aligned. To facilitate this, the mounting holes should be slightly larger than the shaft diameter and small wedges can be used where needed.



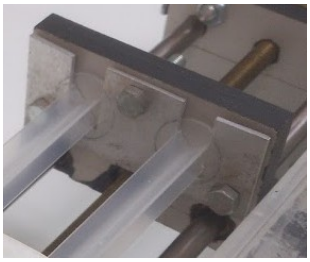


The linear actuator is traversed by both the screw and the shafts. Its movement is driven by the nut inserted in the screw, so the nut should not turn when the screw does. As a result, we must make a hexagonal hole to fit and glue if necessary the nut. The actuator must slide smoothly on the shafts; linear bearings can be used to reduce friction but a cheaper option is a brass/nylon bushing fitted to the shaft diameter and properly lubricated. Both shafts and bushing can be acquired easily and also they can be recovered from the mechanism of an old desktop printer or scanner.



The last plastic piece is to support the steeper[M3] motor. It is attached to the rear paw toward a set of flat head bolts. To make room for the shaft/screw coupler, spacers are used between the rear paw and the motor support. The spacers are some pieces of pipe that are placed between the two parts and through which the bolts are inserted. Finally, motor is mounted by four flat-head bolts.

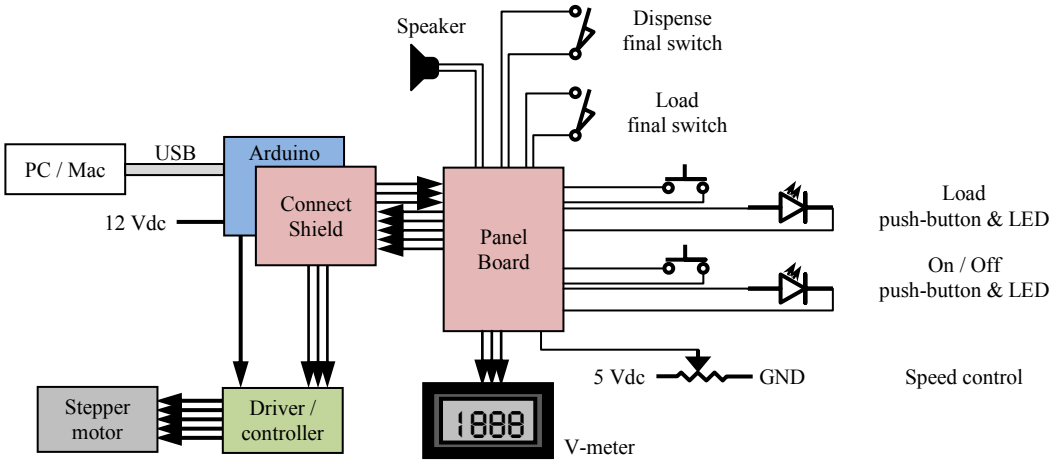
To mount the syringes the front paw is double thick. By this way a groove is machined on top side to hold the barrel collar of syringe. This avoids the back and forth shifts when pushing or pulling the syringe piston. The groove can be machined with an electrical rotary multi-tool and the accessories needed to guide and routing. To complete lock the syringe to paw an upper grip is designed. Actually a double syringe system is designed, so a double grip for two identical syringes is used. Any other design is possible upon demand.



The setup described by now is able to push the syringe plunger and infuse liquids. If we would withdraw fluids we need to pull the plunger without manual intervention, so the plunger must be fixed to the linear actuator. A plastic or metallic piece was attached to the actuator leaving a slot to insert the plunger collar. This way, by ~~reverse~~ reversing the motor rotation, we can pull the syringe plunger at controlled speed.

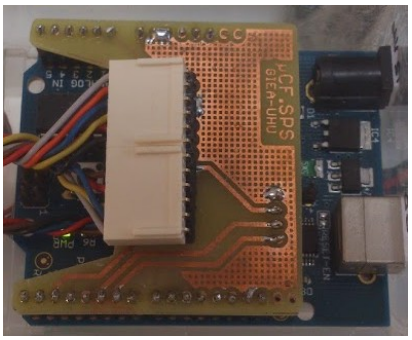
The dimension drawings of all fabricated parts and the assembly drawings are available for free. Also the BoM with indication of where to find sellers for each item are available.

ELECTRONIC SETUP



The stepper motor can drive directly by four digital lines (conveniently boosted) attached to the four motor windings and activated in the proper sequence to generate the rotary motion. This operation can be simplified by using a driver card which, through logical circuitry, generates the appropriate pulse sequence depending on the direction and rotation mode selected. In addition, three control lines (digital outputs) are only needed, not four, and the power stage for windings is included. The RSSM2 driver card was selected (RS Components part nr. 240-7920). The driver has multiple input lines to control the motor in several ways. We have chosen to maintain the possibility to select the width of advance step and direction. Therefore we have to control the state of three lines: half/full step, clockwise/counterclockwise direction and external clock (to control speed).

It is also possible to use a motor shield that you can purchase for a specific microprocessor. They are electronic boards designed to manage the operation of one or many motors from digital ports or by communication ports controlled by the processor. The software can be easily implemented towards dedicated libraries ~~freely accessible~~ with free access. The RS motor driver we have selected is processor-independent and it can be implemented in any platform. It can provide higher currents (up to 2 Amp) and more powerful motors can be used if necessary.



The digital control of motor as well as the management of the manual control panel to operate the syringe pump in local mode is carried out by a micro-controller. We have used an open-code full accessible and very know architecture, the Arduido Duemilanove version. See below to learn about how to program ~~programming~~ the Arduino.

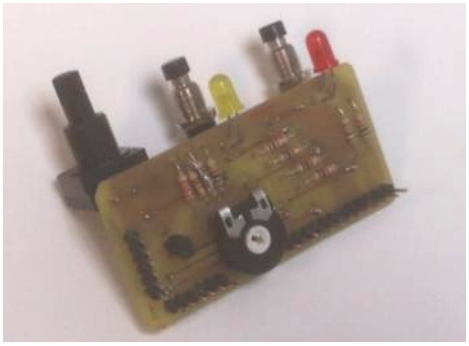
We make a shield board to connect the processor board with both the panel board and the driver motor board.

The control panel for manual operation of syringe pump is as simple as possible. It only has two push buttons with two LED indicators and a potentiometer to speed setting. The potentiometer (linear type) is connected to ground and 5 Vdc and the wiper drive a variable voltage that is read by and analog input in Arduino. It's used to calculate the

frequency of motor steps. One of the pushbuttons is used to start or stop the motor motion in the infusion direction~~:~~, it has an associated LED that is ON when the motor is running (no matter the direction) and OFF when stopped. Finally, the other pushbutton is used to start or stop the Load of syringes~~.~~; ~~T~~ the associated LED is ON when motion is in the withdraw mode and OFF in the infuse mode.

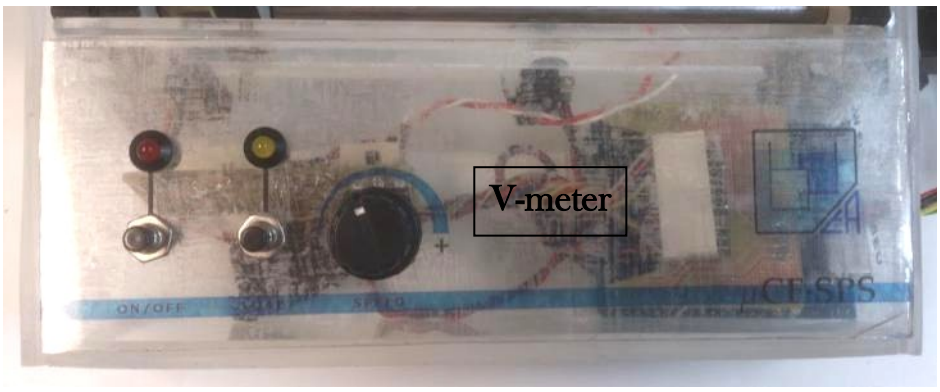
It's a good idea to include a digital voltmeter connected to the analog voltage driven by the potentiometer. ~~By t~~This way it's easy to reproduce the speed at several experiments by setting the same number in the display. This type of voltmeters operates from 0 to 200 mV, so the voltage output from potentiometer must be divided by 25 (with a tunable voltage divider). The SP400 model by Lascar has 3 ½ digits and the adequate dimensions. It can show numbers from 0 to 1999 and it can be powered by 5 Vdc. However, these modules are normally expensive and, as the pump can be controlled without them, they must be considered as an option.

Both pushbuttons are normally open and are connected to respective Arduino digital lines. While they are open a 10 k pull-down resistor drives a Low level~~;~~, when ~~pushes~~they are pushed, they connect 5 V to digital input, driving a High level. The LED indicators have a 1 k resistor connected in series to limiting the current. Both are driven directly from two digital outputs from the Arduino, cathodes are grounded and a High level lights up LEDs.

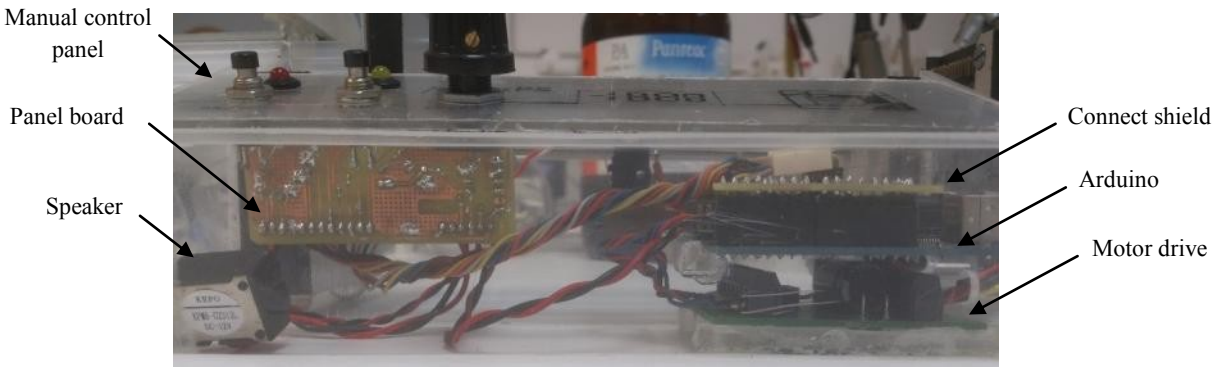


The circuitry associated to LEDs, pushbuttons and potentiometer and the devices themselves are integrated in the named Panel Board. This board also includes circuitry for a speaker to make sound alerts when the syringes are completely empty or filled with fluid. It includes an amplifier stage Vdc power supplied and controlled by an Arduino digital output. To check the limit states of syringes, two limit switches are used, whose circuitry (a pull-down resistor each) is also on the panel board. Both the limit switches and the speaker are cable mounting and are connected to the board by header pins. Header pins are also used to connect ~~with the~~ to the Arduino the digital and analog lines and the power supply and also the optional panel-mount voltmeter.

Schematics and masks for double layer board are available for free. You can make your panel board, mount all components, connect with Arduino and then think about the programming.

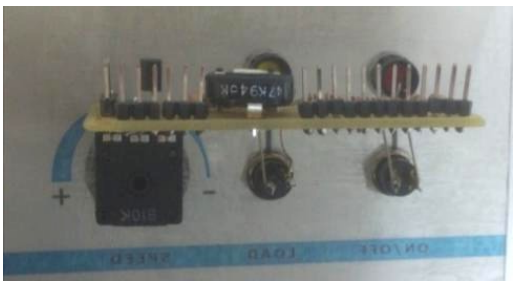


All electronic components are installed inside a cabinet specifically designed to be attached to the mechanical assembly and getting a compact and nice equipment. We have made the cabinet with transparent PMMA to better see the inner for teacher purposes, but other plastic or metallic material can be used. In any case, there are a number of [M4]things to consider in the design to be operational that ~~will list~~ are listed below.



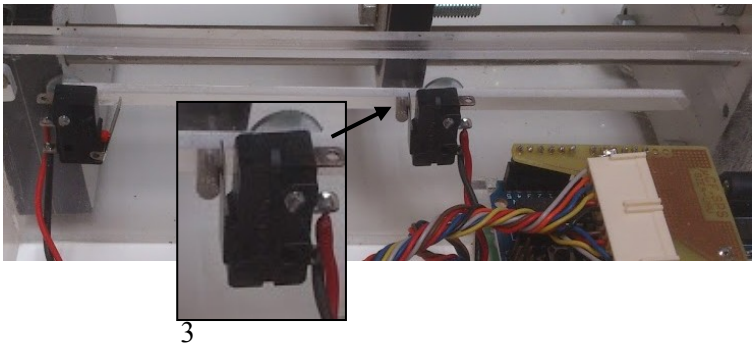
~~It should be accessible the Arduino power and USB connectors~~Arduino power and USB connectors should be accesible from the outside. The power plug must be derived to the motor driver directly (there's the higher power consumption), so we welded two wires to the weld between the connector and the Arduino board and we ~~have~~ connected them to the driver.

The connections of the motor windings must be made through the wall of the cabinet. To prevent accidental spills ~~that~~ to reach the electric circuits, they are made in the far side of the syringes.



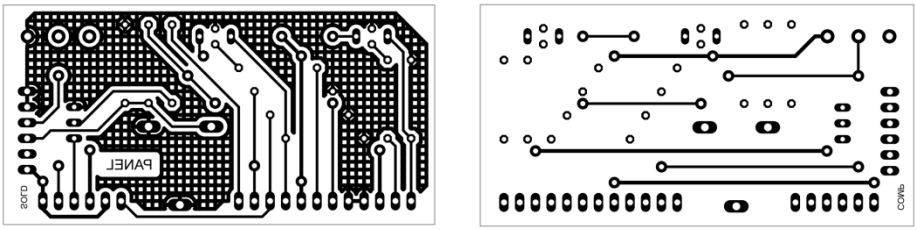
Obviously, the control panel should be placed on top as well as the voltmeter if included. The panel board is attached to the panel by the panel components themselves which are welded to the PCB and fixed to the panel. The speaker is placed anywhere in the cabinet.

The limit switches are mounted on the inside of the back wall, near the mechanical actuator. In that wall a slot has cut for peeking out a small piece that moves

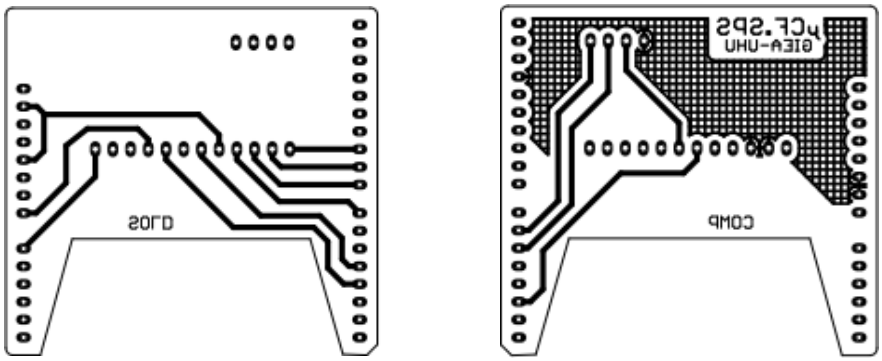


with the actuator and push switches. The same slot is used to install both switches on the right [M5]positions depending on the syringes volume. To avoid the slot, and having a perfect isolation of the electronics, more expensive magnetic field switches could have been used.

In short, we must build two printed circuit boards: the Connect shield for Arduino and the Control panel board. We use the Eagle PCB Design Software from CadSoft to design them. The Eagle Light Edition is a freeware edition for non-profit purposes only. It is fully operative within a few limitations and it can be downloaded for free from the CadSoft web site (http://www.cadsoftusa.com/download-eagle/?language=en). Both PCB designs have two signal layers and both schematics and PCB design are freely available. Both layers masks for components and routes are also available in printable version.



Panel board route masks for top and bottom layers (not to scale)



Connector shield route masks for top and bottom layers (not to scale)

PROGRAMMING ARDUINO

The sketch written for the Arduino microprocessor is developed and uploaded by using the Arduino IDE software. It checks the actuators states and set the indicators or active the stepper motor. Much code is derived from the Arduino code examples published on web, so it's open code.

It starts with the constant declarations and variable assignments:

```
/*
  Remote_control.ino
  To manually control towards front-panel and towards USB remote control the uCF·SPS module,
  a Syringe Pumping System developed by the
  Applied Electrochemistry Research Group on the University of Huelva

  Programmed by Juan Daniel Mozo Llamazares, José Ignacio Otero Bueno and Angel García Barrios
  February, 2014
 */

// pin assignation (constants).
// give it a name:
const int botonONOFF = 2;              // button Run/Stop
const int ledONOFF = 3;                // Red LED Start flag
const int ledLOAD = 4;                 // Yellow LED LOAD flag / R-L motion pin
const int Paso = 5;                    // motor clock pin(one step)
const int pinHalf = 6;                 // full/half step pin
const int botonLOAD = 7;               // LOAD button
const int FinVaciar = 8;               // Dispense limit switch
const int FinLOAD = 9;                 // LOAD limit switch
const int Buzz = 10;                   // buzzer
const int ledPRUEBA = 13;              // Arduino ON BOARD led
const int Veloc = A0;                  // analogical potential read pin (speed)

// other constants
const int Prueba = LOW;                // Demo mode (activa el pin 13 en lugar del motor)
const long debounceDelay = 50;         // stores Debounce time in millisec

// variables
unsigned long previousTime = 0;        // stores previous step time in microsec
unsigned long currentTime = 0;         // stores actual time in microsec
unsigned long interval = 500;          // stores interval-between-steps in microsec
unsigned long previousTimeVolt = 0;    // stores voltmeter' previous refresh time in microsec
long lastDebounceTime = 0;             // stores last LED turnover time
int ledState = LOW;                    // stores led's state
int runMotor = LOW;                    // stores motor's state (HIGH = run)
int runLOAD = LOW;                     // stores running direction (HIGH = LOAD = LEFT)
int botonState = LOW;                  // stores limit switches state
int botonState1 = LOW;                 // stores ONOFF button state
int previousBotonState1 = LOW;         // stores ONOFF button previous state (to Debounce)
int botonState2 = LOW;                 // stores LOAD button state
int previousBotonState2 = LOW;         // stores LOAD button previous state (to Debounce)
int volt = 0;                          // stores analog speed setting (0 - 1024)
char serialData = 0;                   // stores data read from USB (one byte 0 - 255)
int remote = LOW;                      // stores control active mode (to speed's question)
```

A set of remote commands is provided to enable the remote control via USB connection. Almost all are one character commands for easy programming.

```
/* USB command set
fn,n    == Freq (hl = millisec 0 to 1024)    replace manual panel control (hi and low byte required)
g       == Go run                            start motor to difuse
l       == Load                              start motor to withdraw
s       == Stop                              stop motor
r       == Remote                            enable remote control (disable speed reading)
m       == Manual                            disable remote control (enable speed reading)
u       == Up                                increases speed
d       == Down                              decreases speed
*/
```

Then there is the `setup` routine, therein it sets the digital pins as input or output and the serial port is initialized.

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pins as input or output.
  pinMode(ledONOFF, OUTPUT);
  pinMode(ledLOAD, OUTPUT);
  pinMode(Paso, OUTPUT);
  pinMode(pinHalf, OUTPUT);
  pinMode(Buzz, OUTPUT);
  pinMode(ledPRUEBA, OUTPUT);
  pinMode(botonONOFF, INPUT);
  pinMode(botonLOAD, INPUT);
  pinMode(FinVaciar, INPUT);
  pinMode(FinLOAD, INPUT);

  // initialize the serial port
  Serial.begin(9600);
}
```

The `loop` routine is ever run. It starts by reading the actual time (in microseconds) and loading it in the `currentTime` variable. This value is used to compare below with the motor ~~steep's~~ [M6]step's period. As time goes by, the routine is checking several things.

```
// the loop routine runs over and over again forever:
void loop() {
  // first read the actual time                    ------------------------------------------------
  currentTime = micros();
```

First it checks for a command waiting on the serial port. If any, the variables values changes depending on the command received by a switch/case structure. When the command has a valid meaning and has acted accordingly, a handshaking is returned echoed the same command.

```
  // check if data is waiting on serial port       ------------------------------------------------
  while (Serial.available() > 0) {
    serialData = Serial.read();
    switch (serialData) {
      case 's':                                    // stop
        runMotor = LOW;
        runLOAD = LOW;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'm':                                    // manual mode
        remote = LOW;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'r':                                    // remote mode
        remote = HIGH;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'l':                                    // load
        runMotor = HIGH;
        runLOAD = HIGH;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'g':                                    // go run
        runMotor = HIGH;
        runLOAD = LOW;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'u':                                    // up volt (speed)
        volt += 5;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'd':                                    // down volt (speed)
        volt += -5;
        Serial.print("Ok, ");                      // Handshaking Ok
        Serial.println(serialData);
        break;
      case 'f':                                    // frequency (speed)
        int HByteRead = Serial.parseInt();         // read the message completely
```

```
        int LByteRead = Serial.parseInt();
        LByteRead = constrain(LByteRead, 0, 255);   // avoid out-of-range values
        HByteRead = constrain(HByteRead, 0, 3);
        volt = LByteRead + (HByteRead * 256);       // compose two Bytes to calculate volt

        Serial.print("Ok, ");                       // Handshaking Ok
        Serial.print(serialData);
        Serial.print(HByteRead,DEC);
        Serial.print(',');
        Serial.print(LByteRead,DEC);
        Serial.print(',');
        Serial.println(volt,DEC);
        break;
    }
    int serialDatatmp = Serial.read();              // empty serial port (CR)
  }
```

For 'f' command (sets a new value of frequency) it must be included the new setting, replacing the analog read from the 10-bits A/D converter build in the Arduino Duemilanove. A 10-bits number reaches 1024 but serial port support only bytes, so two bytes are needed that must be composed and prevent overflows[M7].

Then it checks if the control panel's buttons are pressed. To prevent unwanted multiple activation, a debouncing procedure is implemented. A single push generates only a toggle in the variables setting. The On/Off button toggles the `runMotor` variable but, if `runMotor` results `LOW` the `runLoad` goes also `LOW` and motor always stops. The Load button toggles the `runLoad` variable but, if `runLoad` results `HIGH` the `runMotor` goes also `HIGH` and motor starts.

```
  // check control panel knobs                    ------------------------------------------------
  // check ONOFF button setting --------------------
  int reading = digitalRead(botonONOFF);
  if (reading != previousBotonState1) {             // with Debounce
    lastDebounceTime = millis(); }
  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != botonState1) {
      botonState1 = reading;
      if (botonState1 == HIGH) {                    // changes the runMotor value if button is pushed
        runMotor = !runMotor; }
      if (runLOAD == HIGH && runMotor == LOW) {     // if loading and run stop all
        runLOAD = LOW; }}}
  previousBotonState1 = reading;
  // check LOAD button setting ---------------------
  reading = digitalRead(botonLOAD);
  if (reading != previousBotonState2) {             // with Debounce
    lastDebounceTime = millis(); }
  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != botonState2) {
      botonState2 = reading;
      if (botonState2 == HIGH) {                    // changes the runLOAD value if button is pushed
        runLOAD = ! runLOAD; }
      if (runLOAD == HIGH) {                        // check if goes LOAD and start the motor too
        runMotor = HIGH; }}}
  previousBotonState2 = reading;
```

Then the micro-switches are checked. If they are pressed a `HIGH` is read and we must stop the motor by setting `LOW` at `runMotor`. If load switch is pressed, `runLoad` goes `LOW` also. An audio alarm is activated to alert than the user attention is required. A different tone is used to distinguish ~~the two~~ both events.

```
  // check the limit switches                     ------------------------------------------------
  // check the DISPENSE limit switch setting -------
  botonState = digitalRead(FinVaciar);
  // motor must be running and dispensing to check the limit switch
  if (botonState == HIGH && runMotor == HIGH && runLOAD == LOW) {
    tone(Buzz,250,500);                             // audio alarm
    runMotor = LOW; }                               // if limit switch is pressed stop the motion


  // check the LOAD limit switch setting -----------
```

```
botonState = digitalRead(FinLOAD);
// motor must be running and loading to check the limit switch
if (botonState == HIGH && runMotor == HIGH && runLOAD == HIGH) {
  tone(Buzz,523,250);                          // audio alarm
  runLOAD = LOW;                               // if limit switch is pressed stops and load
  runMotor = LOW; }
```

Once they have been checked all the actuators, the LED are setting to reflect the variables. Remember that `runLoad` affects to LED and also to the direction setting on motor driver. To fully configure the motor driver the step-mode pin must be sets. We have programmed a half-step mode for infusion motion (more resolution) and a full-step mode for withdraw motion (more speed) so the `pinHalf` always has the inverse setting of the `runLoad` variable and no need to program anything else. As `pinHalf` setting is code-controlled, other configurations are possible but more code is required.

```
digitalWrite(ledONOFF, runMotor);              // write RUN setting on LED
digitalWrite(ledLOAD, runLOAD);                // write LOAD setting on LED and set direction
digitalWrite(pinHalf, !runLOAD);               // sets HALF step to dispense and FULL to load
```

It's time to check if we have to [M8]move the motor. But first we have to calculate the period of motor movement. In remote mode the setting is read from serial port and ~~there is nothing to do~~you don't have to do anything. In manual mode we must read the analog port to check the potentiometer setting. The reading is done twice every second. In addition we write the reading in the serial USB port, so the remote control software knows the data.

In any case, the `interval` between motor steps is calculated from the `volt` variable to infuse fluids. To withdraw, a fixed `interval` is assigned to get a quick filling of syringes.

~~There is provided a Demo mode~~A Demo mode is provided, ~~-~~in which there is full operation but, instead of activating the motor windings, a flash is generated in the Arduino onboard LED, which is associated to pin 13. In that case a bigger interval is needed to ~~correctly~~ see the blink properly. This mode is advisable while debugging routine, eliminating the higher power consumption the electronics can be powered directly from the USB port.

```
// calculate motor rotation speed              ------------------------------------------------
if (remote == LOW) {                           // manual mode
  volt = analogRead(Veloc);                    // read potentiometer value each 500 millisec
  // check if refresh time is passed (500 ms)
  if (currentTime - previousTimeVolt >= 500000L) {
    previousTimeVolt = currentTime;
    Serial.print('v');                         // write in serial USB to PC synchronization
    Serial.println(volt); }}

if (Prueba == LOW) {
  interval = (1025 - volt) * 250L;             // calculates motor steps interval in microsec
  if (runLOAD == HIGH) interval = 3500L; }     // full speed to load
else {                                         // in demo mode calculates bigger intervals (LED)
  interval = (1024 - volt) * 1000L;
  if (runLOAD == HIGH) interval = 25000L; }
```

Once the `interval` has been calculated, it is compared with the time since the last check only if the motor must be activated.

If the interval has passed, the current time is stored in the variable `previousTime` and a pulse is generated on pin `Paso`. The motor driver CLK input is flange active on rising edge, so it is advisable to stay the pin `LOW` and generate an `L-H-L` sequence.

In Demo mode a blink routine is programmed by inverting the `ledState` variable and writing it on pin 13 every `interval`.

```
  // check if elapsed time is bigger than calculated interval to motor step up ----------------------
  if (currentTime - previousTime >= interval && runMotor == HIGH) {
    previousTime = currentTime;                    // stores actual time (reset time)
    if (Prueba == HIGH) {
      ledState = !ledState;                        // turn the LED OFF and ON every interval
      digitalWrite(ledPRUEBA, ledState); }
    else {
      digitalWrite(Paso,HIGH);                     // set a pulse on Paso (Motor goes on)
      delay(1);
      digitalWrite(Paso,LOW); }
  }
}
```

After the last checking the `loop` routine ends and it starts over again.