Description of the stepper motor software running on an Arduino NANO in slave mode
created: 01.22.2020
modified: 01.17.2023, 05.04.2023

The task specified below is to run a single stepper motor connected to a motor driver in an most optimal way. The output of the stepper software is controlling an output pin of the Arduino that defines the travel direction and a second output pin that provides the travel pulsing. A travel pulse and a corresponding angular travel of the stepper motor´s rotating shaft are correlated via the mechanical properties of the stepper motor. A common value is 1.6 degree per stepper pulse in full step mode.

It is up to the user what mode of operation he prefers: full step mode, half step mode or even micro step mode. In any case the program just produces stepping pulses which have to be mulltiplied according to its mode of operation.

Programming tool is the official Arduino IDE.

There are two modes of stepper operation implemented:
- running continously at a defined speed
- doing a precise travel of defined lenght with a time optimal speed profile. Such a travel consists out of a constant acceleration phase, a constant top speed phase and a breaking phase. The length of a certain travel is defined by the number of steps.
  At the end of the travel the amount of driven steps is equal to the amount of driven steps defined in the travel order that was executed. The motor shaft comes to a rest without any overshoot and signalises: „travel order accomplished, waiting or a new task“ to the main program.

Note that all travelling is relative to the last end position of former travels. No absolute travelling is possible due to the nature of stepper motors.

Way of software implementation:
- all travel activities are to run „in the background“ thus leaving the main program free hands to do whatever is needed to be done in the foreground. In the Arduino´s world the program´s main function is called „loop()“.
- the way of running in the background is implemented by use of an ISR with numberous paths of branching
- such ISR kind of reprograms itself by changing the period within two consequtive calls on the fly.

Brief description of the underlying principle of operation:
Let´s assume the stepper motor is traveling below or at start/stop-speed. The train of pulses can be stopped at any time and the motor shaft will stop accordingly. Up to now everything is simple, yet no anticipatory processing is needed.
But now we start to accelerate the motor to a speed range far beyond its start/stop-speed. Seen out of the programmer´s point of view, this means to produce a number of pulses with the property of making the period between two consecutive pulses diminish at a constant rate. For the benefit of traveling as fast as possible the applied acceleration should be as big as possible. The summation of such pulses tells us the exact breaking distance needed from now on to take the motor to a full stop with fastest possible deceleration applied.
Basically this means now to install a continous comparison of the braking distance with the rest of distance remaining until the travel command is processed fully.
Actually the neccessary continous comparison of the braking distance with the rest of distance has to start even in the acceleration phase of the travel since the travel length might be very short, so the top speed of the constant travel phase was never be applicable.
These requests takes the program to a certain degree of complexity and pushes it close to the margin of comprehensibility even with a high degree of inserted comments left for the interested source code reader.

The comunication between the main program as the ordering client and the ISR processing such tasks, happens via common variables.

The following variables are needed to fully define a travel task:

long int drivedistance; long int target_pos	The target position to drive to in a drive command, relative to the current position. Can be positive thus making the motor shaft run clockwise, or can be negative for a counter clockwise rotation

int st_speed;	Maximum start/stop speed that can be expected to run a stepper motor from a standstill to final speed without loss of steps. Or the highest speed of a stepper motor that allows to stop the motor without overshoot when the stepping pulses are turned off at once Only positive values are allowed Typical values: 20 to 50 steps per second
int ramp;	Positive number of steps for the acceleration or break-down ramp
int top_speed;	Constant speed in between acceleration and break-down travel phases

Motion characteristics:

- positioning range	According to the scope of signed long int (32 bits) data, no checking for overflow
- speed range	10 to 5000 steps per second for a standard stepper motor, no checking for out of range. 9990 is the maximum value, if the motor can handle it.
- speed resolution	10 steps per second with some uncertainty due to hardware restrictions related to timer/counter properties of the Arduino hardware
- ramp lenght	1 to 999 steps
- ramp resolution	1 step
- positioning resolution	1 step
- positioning repeatability	+- 0 step

Timing:

- travel pulse	Defines the binary state of a hardware signal to the stepper controller. For the Arduino UNO this is 1.5us by default. Can be adapted according to the power stage in use.
- travel direction	Hardware related as mentioned before. Should always be set prior to the travel pulse

The main program “loop()“ is responsable for providing plausible travel data. No ISR error messages are returned to the main program. False travel data may cause false travelling.

There are travel orders producing motor motion as well as state reporting orders.

The following travel commands given out by a higher level system or by a “master“ can be processed by the steppermotor controling software on the Arduino NANO, running as slave:

G+/–xxxxxxx	Go to the specified target position.
g(+/–)	Drive the motor indefinitely in the specified direction.
F or in	Feed back the actual status i.e. Ready or Busy concerning a certain travel order, eighter finished progress yet.
Z	Soft stop. Command for an orderly termination of a travel order in progress: The deceleration ramp is driven down before the motor´s shaft comes to full stop. The software keeps track of the reached position.
PO	Enter the programming mode for the execution of a next travel order
I1	Initialize the motor position counter with zero
V1	Read back the current position of the travel in progress
K	Kill a travel order in progress by a hard stop with the result that any information about the present motor position gets lost.

A typical travel order sent by the (Arduino) master contains five data outputs to the Arduino slave. The sixth output starts the motor‘s travel instantly. Each of the six outputs must be terminated by '\r' (carriage return).

So the coded data outputs from the master to the slave, written with the Arduino IDE look like this:

```
Serial.print("PO\r");                          // send the header for the programming mode to enter

sprintf( stringbuf, "S%d\r", st_speed/10 );    //  send the parameter for the start/stop speed
Serial.print( stringbuf );

sprintf( stringbuf, "T%d\r", top_speed/10 );   // send the parameter for the top speed
Serial.print( stringbuf );

sprintf( stringbuf, "R%d\r", ramp );           // send the parameter for the ramp length
Serial.print( stringbuf );

sprintf( stringbuf, "G+%ld\r", my_position );  // send the plus or minus target motor position parameter
Serial.print( stringbuf );                     //  according to LENGTHOFTRAVEL

Serial.print("E\r");                           // send the trailer for a travel order to the slave controller
                                               // in order to start the travel instantly
```

Note the implemented handshake principle: For each output to the slave an acknowledge of the slave is generated and fed back for being processed by the master.

A positive acknowledge is 'Y' whereas a negative acknowledge is 'E' for error.

Below is the circuit diagram showing how the Arduino master and slave are hooked up against each other.