MobaTools - an Arduino library for model railroaders and model makers

Documentation for Version 2.2

The MobaTools are a compilation of classes that are useful in the model railway environment (but not only there). They facilitate some standard tasks, thus enabling even beginners to write complex Sketches.

Summary of the classes:
In version 2.0 the class names have been changed and unified. The old class names are still valid for compatibility reasons, but should not be used in new sketches. The old class names are no longer marked red in the IDE.

MoToServo	Control of up to 16 servos. The methods are largely compatible with the standard Servolib of the Arduino. However, the speed of movement of the servo can be specified..
MoToStepper	Control of stepper motors. Unlike the standard library 'steppers' the calls are nonblocking. Once a reference point has been established, the stepper motor can be positioned absolutely (in degrees) with nearly the same methods as in the MoToServo Library. The stepper motor driver A4988 (and all drivers that use a dir and a step input) is supported. With this driver bipolar steppers can be used. A ramp for accelerating and decelerating the motor can be defined
MoToSoftLed	Soft fading in and out of Leds. All digital pins are permissible as connection pins.
MoToTimer	With this class it is easy to create time delays without blocking the Sketch.
MoToButtons	Manage up to 32 buttons and switches with debounce and event handling (pressed, released, short press, long press). The buttons/switches are read in via a user callback function. This enables matrix arrangements and e.g. I2C port expander to be used..
MoToPwm	This class exists only for ESP8266. On ESP8266, the integrated tone(), analogWrite() and Servo() functions cannot be used in parallel with MobaTools due to limited timer resources. MoToPwm provides methods to replace tone() and analogWrite()

The MobaTools intensively use the Timer1 of Arduino (AVR). Therefor all other libraries or functions that also use the Timer1 can not be used simultaneously. This concerns, for example, the standard ServoLib and analog write on pins 9 + 10. If timer 3 is available, it is used instead. This applies to Arduino Leonardo, Micro and Mega.

Timer 4 is used on the STM32 platform. The core of RogerClark (https://github.com/rogerclarkmelbourne/Arduino_STM32) must be used.

From version V2.0 onwards, MobaTools can also be run on the ESP8266. Since the timer structure is completely different from the above processors, some concepts had to be changed for this processor. This means that the GPIO16 cannot be used for pulse outputs (Softled, Servo, Step). It can be used as a dir output for stepper if the enable function is not used.

Although there is no longer a hard limit to the number of instantiatable objects for softleds and steppers, the performance limits of the processor used must be taken into account. Both stepper and softleds are managed in one and the same interrupt routine.Softleds only burden the interrupt during fade up and fade down. For steppers, the load depends on the steprate (frequency of the interrupt) and the ramp (during the ramp, the computing load in the interrupt is higher).
With UNO/Nano/Leonardo and Micro the limitation already results from the number of available pins. With a mega, however, this must be taken into account more carefully.
Attempts with 6 steppers and 6 constantly pulsating LEDs were still problem-free. The maximum number of steppers is limited to 6 by a #define in MoBaTools.h. If e.g. no softleds are used, this can be set higher.

Class MoToServo

Creating one Servo object (up to 16 objects are allowed):
```
MoToServo  myServo;
```

Methods:
```
byte myServo.attach( int pin );
```
Assigns a pin where the pulses are output. The pin is switched to output, but no pulses are produced yet.

```
byte myServo.attach( int pin, bool autoOff );
```
If the optional parameter 'AutoOff' with the value 'true' is passed, then the pulse is turned off automatically when the length of the pulse does not change for more than 1sec.

```
byte myServo.attach( int pin, int pos0, int pos180, bool autoOff );
```
With the parameters pos0 and pos180 can be specified, which pulse lengths are assigned to the angles '0' or '180'. (The parameter AutoOff is optional)

```
byte myServo.detach();
```
The assignment of the servos to the pin is removed, it will no longer generate pulses. If this method is called during an active pulse, the pulse end is waited for. This way no cut off / too short pulses occur before switching off

```
void myServo.setSpeed(int Speed);
```
Set moving speed of the servos. 'Speed', is the value (in 0,5µs units) the pulse length changes every 20ms until target length. Speed = 1 means that 40s are needed to change the pulse width from 1ms to 2ms.
Speed = 0 (default value) means direct change of pulselength (like standard servo Library)

With version 0.9 the resolution of speed ist 4 times better. That means Speed is in units of 0.125µs per 20ms. The lib starts in a compatibility mode where it behaves like the former versions. (There is no compatibility mode on ESP8266)

The new resolution is globally enabled with the command:
```
void myServo.setSpeed(int Speed, HIGHRES );
```
When this command is called once, alle servo objects use the higer resolution.

```
void myServo.write(int angle);
```
Specify arget position. The servo moves with the preset speed to that position. With values from 0 ... 180 the value is interpreted as an angle, and accordingly converted into a pulse

length. Values > 180 are interpreted as pulse length in microseconds, where pulse length is limited to the minimum or maximum length.
The write command takes effect immediately, even if the servo has not yet reached its set position of the previous write.
If this is the first 'write' call after an 'attach', the specified pulse length is output immediately (starting position).

```
byte myServo.moving();
```
Information about the state of motion of the servo.
0: The servos rests
> 0: Distance to go until the servo reaches its target position. In% of the total distance since the last 'write'.

```
byte myServo.read();
```
Current position of the servos in degrees.
```
byte myServo.readMicroseconds();
```
Current position of the servo in microseconds (pulse length).
In contrast to standard Lib these values are not necessarily the values passed with the last 'write' call. While the servo is still moving, the actual position is returned.

Although there is no separate 'stop' command, this can be accomplished with the sequence
`myServo.write( myServo.readMicroseconds() );`
Thus the current position is set as the new target position, which leads to the immediate stop of the servo.

```
byte myServo.attached();
```
Returns 'true' if a pin is assigned.

```
void setMinimumPulse( word length);
```
Defines the pulse length for angle '0' . Default is
550 µs for ESP8266
700µs otherwise (must not be set shorter).

```
void setMaximumPulse( word length);
```
Defines the pulse length for angle '180'. Default is
2600 µs for ESP8266
2300µs otherwise (must not be set longer).

Class MoToStepper

Bipolar and unipolar stepper motors can be controlled. In contrast to the Arduino standard library, the calls are nonblocking. The program in the 'loop' continues to run while the stepper motor is turning.
In addition to the step speed, a starting and braking ramp can also be specified. The ramp length is defined by the number of steps. This means that it is indicated how many steps are required to get from standstill to the set speed (and vice versa).
With ramp length 0 (default) the stepper motor behaves as before (before V1. 1).
If the step speed is changed without specifying a new ramp length, the ramp length is adjusted in such a way that the starting behavior remains approximately the same.

Setting up of the stepper motor:

```
Stepper4 myStepper( int steps360, byte mode  );
```

mode specifies whether the engine is activated in `FULLSTEP` or `HALFSTEP` mode. If the parameter is omitted, `HALFSTEP` is assumed. (Except ESP8266, where the only allowed value is A4988)
`mode = A4988` indicates that the motor is connected via a stepper motor driver with step and direction input (for example, the A4988). For ESP8266, this is the only valid mode!

`steps360` is the number of steps the motor takes for one revolution (in the specified Mode. For A4988 the set microstepping must be taken into account)

Methods:

```
byte myStepper.attach( byte spi );
byte myStepper.attach( byte pin1, byte pin2, byte pin3, byte pin4 );
```

Assignment of the output signals. The signals can be assigned either to 4 individual pins, or to the SPI interface. For the SPI interface the values SPI_1, SPI_2, SPI_3 or SPI_4 are permitted. There are always 16 bit shifted out. The parameters indicate the position of the 4 bits of the assigned motor. SPI_4 are the first pushed out bits and are therefore at the end of the shift register chain. If only a 8-bit shift register connected, only the values of SPI_1 and SPI_2 can be used.
The SPI interface is enabled only if at least one of the stepper motor is connected via SPI. An example of the wiring is shown below.

The function value is 0 ('false') if no mapping was possible.
Both unipolar and (via a double H-bridge) bipolar motors can be controlled via the 4 output pins.
These options cannot be used on an ESP8266.

```
byte myStepper.attach( byte pinStep, byte pinDir );
```

Assignment of the outputs ‚step‘ and ‚direction‘ if the A4988 driver is used.

```
void myStepper.attachEnable( uint8_t pinEna, uint16_t delay, bool active );
```

This defines a pin which can be used to switch the motor on or off or to set it to power saving mode. The pin is always active while steps are being output.

`pinEna`: Arduino pin number.
`delay`: Delay between pin switch-on and 1st step in ms, also between last step and pin switch-off.
`active`: Output active HIGH or active LOW.

ESP8266: When using this function, the Dir output cannot be routed to the GPIO16

```
void myStepper.detach();
```

The pin assignment is canceled.

```
void myStepper.setSpeed(int rpm10 );
```

Set up rotation speed (in revolutions / min). The value must be specified in rpm*10 .

```
uint16_t myStepper.setSpeedSteps( uint32_t speed10 ); // ESP8266 only
uint16_t myStepper.setSpeedSteps( uint16_t speed10 ); // AVR + STM
```

Stepping speed of the motor in steps / 10sec. Maximum value for AVR processors is 25000 (2500 Steps/sec), for STM 50000 (5000 Steps/sec) and for ESP8266 80000 (8000 Steps/sec)
If necessary, the ramp length is adjusted to keep the starting behavior approximately the same.
Function value is the new ramp length.

```
uint16_t myStepper.setSpeedSteps( uint32_t speed10, uint16_t rampLen );
uint16_t myStepper.setSpeedSteps( uint16_t speed10, uint16_t rampLen );
```

Stepping speed of the motor in steps / 10sec. (for ESP8266 as unit32_t, because the steprate may become > 65000 see above)
Ramp length in steps.
The permissible ramp length depends on the step speed. The maximum is16000 for high step rates. At step speeds below 2steps / sec, no ramp is possible. If rampLen is outside the permitted range, the value is adjusted.
Function value is the current (possibly adjusted) ramp length.

```
uint16_t myStepper.setRampLen( uint16_t rampLen );
```

Ramp length in steps.
The permissible ramp length depends on the step speed. Maximum is16000 for high step rates. At Stepraten below 2steps / sec, no ramp is possible. If rampLen is outside the permitted range, the value is adjusted.
Function value is the current (possibly adjusted) ramp length.

```
void myStepper.doSteps(long stepcount );
```

Number of steps to be performed by the stepper. The sign indicates the direction of rotation. The reference point for starting the steps is always the current motor position. This is also true if the engine is already rotating at the time of the command. A new target position is calculated from the number of steps, which is then approached (with ramp). This can cause the motor to overshoot the target and then rotate back to the target position.
doSteps (0) e.g. cause the engine to decelerate and turn back to the position at the time of the command,

```
void myStepper.rotate( int direction );
```

The motor rotates up to the next stop command. (1= forward, -1 = backwards)
Rotate (0) stops the motor (with ramp). The motor stops at the end of the braking ramp.

```
void myStepper.stop( );
```

Stops the motor immediately (emergency stop).

```
void myStepper.setZero( );
```

The current position of the motor is taken as a reference point for the 'write' commands with absolute positioning.

```
void myStepper.setZero(long zeroPoint);
```

The new reference point is set '`zeroPoint`' steps away from the current position.

```
void myStepper.write( long angle );
void myStepper.write( long angle, byte factor );
```

The motor moves (measured from SetZero-point) to the specified angle. The passed angle is not limited to 360 °. For example setting angel = 3600 means 10 revolutions from reference point. If, for example, next call angel = -360 is passed, this does not mean 1 turn back, but 11 turns back! (-360 ° from reference point)
The 2nd call allows to specify the angle as a fraction. With factor = 10 set angle is interpreted in 1/10 °.

```
void myStepper.writeSteps( long stepPos );
```

The motor will move to the position that is stepPos steps away from reference point

```
byte myStepper.moving();
```
Information about the state of motion of the stepper.
= 0 when the engine stopped
= 255 when the motor rotates endlessly ('rotate' call)
> 0 ... <= 100 rest of movement to the destination point in% of total distance since last 'write' or 'doSteps' command. This may also include a 'detour'; if the ramp causes the stepper to move beyond the target first and then back again. In this case, values above 100% can occur.

```
long myStepper.stepsToDo();
```
returns the remaining number of steps until reaching the target. This may also include a 'detour'; if the ramp causes the stepper to move beyond the target first and then back again.

```
long myStepper.read();
```
returns to actual position of the stepper in degrees (measured from reference point).

```
long myStepper.readSteps();
```
returns to actual position of the stepper in steps (measured from reference point).

```
int32_t myStepper.getSpeedSteps();    //(ESP8266)
int16_t myStepper.getSpeedSteps();    // AVR+STM
```
always returns the current speed in steps/10sec (also during acceleration/braking)

More Info about the stepper functionality:
Absolute and relative commands can be used mixed. Internally, the steps taken are always counted (in a long variable). As long as this variable does not overflow (happens after about two billion steps in one direction), the controller always knows where the stepper is. 'SetZero' sets this counter to 0.
The maximum speed is related to the IRQ rate, and is designed to operate up to 6 steppers at this speed. In addition, the softleds are also processed in the same interrupt, which also loads this interrupt.
The achievable step rates are therefore also dependent on the target platform. Default values are currently as follows:
AVR 2500 Steps/sec
STM 5000 Steps/sec
ESP 8000 Steps/sec
Taking into account the total load, these values can be slightly optimized (via #defines in MobaTools.h)

Class MoToSoftLed

The class MoToSoftLED contains methods to switch a LED on and off softly.

Create:
```
SoftLed myLed;
```
The number of controllable leds is basically not limited. In practice, the available pins and the performance of the microprocessor (Ram and speed) determine the usable number.
Methods:
```
byte myLed.attach( byte pinNr );
byte myLed.attach( byte pinNr, byte Invert );
```
It is not necessary that pinNo is a PWM capable pin. If the parameter Invert is specified and 'True', the output logic changes: ON is then LOW at output, OFF is HIGH at output.

ESP8266: Gpio16 cannot be used.

`void myLed.riseTime( int Wert );`
The value indicates the time in ms until the Led reaches full brightness or dark.
Since internally only certain steps are possible, the actual time is the best possible approximation to the transferred value.
The maximum value depends on the processor (approx. 5sec for STM, 10s.for AVR, 65s for ESP)

`void myLed.on();`
The LED is turned on.

`void myLed.off();`
The LED is turned off.

`void myLed.on(uint8_t brightness );` (**ESP8266 only**)
`brightness = 0...100` PWM-balue for ‚ON' in %
The LED is switched on and lights up with brightness % of maximum brightness.
The PWM value is stored for subsequent switch-on operations (on/toggle/write).

`void myLed.off(uint8_t brightness );` (**ESP8266 only**)
`brightness = 0...100` PWM-value for OFF in %
The LED is switched off, but still lights up with the PWM value set.
The PWM value is stored for subsequent switch-off operations (off/toggle/write).
The PWM value for off must be lower than that for on.

`void myLed.toggle();`
If the LED is on, it is switched off and vice versa.

`void myLed.write( byte State);`
State = ON oder OFF.

`void myLed.write(byte Zustand, byte Type );`
Type can be one of LINEAR or BULB. This changes the switching characteristic.

Class MoTotimer

This class contains methods for non blocking time delays.
If you want to use this class only, you can also use #include <MoToTimer.h>.

Set up:
`MoToTimer myTimer`

Methods:
`void myTimer.setTime( long Zeit );`
Starts the timer. The value specifies the duration of the delay in ms.
Maximum value is 2 billions ms

`bool myTimer.running();`
'true' while the timer is running, false otherwise .

```
bool myTimer.expired();
```
Event 'Timer expiry'. Function value is 'true' only for the first call after the timer has expired, otherwise it is always 'false'.

```
long myTimer.getTime();
```
returns the remaining time (in ms) or 0 if not running.

```
void myTimer.stop();
```
stops the timer before expiry.

Class MoToButtons

This class contains methods for reading buttons and switches.
By default it is able to manage up to 16 buttons/switches. The default can be changed to save RAM (up to 8 buttons) or to manage up to 32 buttons (with additional RAM consumption).
This can be achieved by inserting '#define MAX32BUTTONS' or '#define MAX8BUTTONS' before the #include <MoBaTools.h>.
If you want to use this class only, you can also use #include <MoToButtons.h>.

Reading the hardware state of the buttons can be done by a usercallback function. This enables designs where the buttons/switches are arranged in a matrix and/or read via a port extender. The return value of this function has to be a 8-Bit, 16-Bit or 32-Bit value according to the maximum manageable number of buttons. Every button/switch is represented by one bit, where '1' means the button is pressed. The button number in the methods corresponds to the bit number

The datatype 'button_t' is automtically set to the correct type (uint8_t, uint16_t or uint32_t) and can be used to define the type of the callback function.

Set up:

```
MoToButtons myButtons( const uint8_t pinNumbers[], const uint8_t pinCnt,
uint8_t debTime, uint16_t pressTime );
```
Par1: Address of an array with the pin numbers of the switches. The switches must be connected against Gnd. (Pressed results in LOW at the pin).
Par2: Number of connected switches (length of the array of Par1).
Par3: Debounce time in ms
Par4: Time (in ms) to distinguish between short an long button press. Maximum is debouncetime * 255

```
MoToButtons myButtons( button_t (*getHWbuttons)(), uint8_t debTime,
uint16_t pressTime );
```
Par1: Address of the function to read the HW.
Par2: Debounce time in ms
Par3: Time (in ms) to distinguish between short an long button press. Maximum is debouncetime * 255

In both cases a further parameter can optionally be specified to change the double click time (default is 300ms).

Methods:

```
void myButtons.processButtons();
```
This method must be called in the loop frequently.If it is called less frequently than the debounce time, the distinction between short and long pressing is inaccurate.

```
bool myButtons.state( uint8_t buttonNbr );
```
Get static state of button ‘buttonNbr’ (debounced).

```
button_t myButtons.allStates();
```
Returns the debounced state of all buttons.

```
button_t myButtons.changed();
```
All bits are set where state has changed since last call of `myButtons.changed()`.

```
bool myButtons.shortPress( uint8_t buttonNbr );
```
True if button ‘buttonNbr’ was pressed short. This event is set when the button is released after a short press and reset after the call of this method or if it is pressed again.

```
bool myButtons.longPress( uint8_t buttonNbr );
```
True if button ‘buttonNbr’ was pressed long. This event is set when the button is released after a long press and reset after the call of this method or if it is pressed again .

```
bool myButtons.pressed( uint8_t buttonNbr );
```
True if button ‘buttonNbr’ was pressed. This event is set when the button was pressed and reset after the call of this method or if it is released.

```
bool myButtons.released( uint8_t buttonNbr );
```
True if button ‘buttonNbr’ was released. This event is set when the button was released and reset after the call of this method or if it is pressed again.

```
void myButtons.forceChanged();
```
The next call of 'changed' or 'pressed'/'released' behaves as if the button had changed its state to the actual position. longPress() and shortPress() are unaffected.

```
void myButtons.resetChanged();
```
All ‘changed’ events are cleared (regarding the call of 'changed' , 'pressed' or 'released' , but longPress() and shortPress() are unaffected.

Klasse MoToPwm (ESP8266 only)

The class MoToPwm contains methods to generate sounds and PWM signals.
The original functions tone() and analogWrite() must not be used.

Set up:
```
MoToPwm myPwm;
```

Methods:
```
byte myPwm.attach( byte pinNr );
```
The function value is '0', if an invalid pin number was specified or the pin is already in use. The pin is switched to output, but no pulses are generated yet.

```
byte myPwm.detach();
```

The pin is released.

```
void myPwm.analogWrite( uint16_t duty1000 );
```
A PWM signal is generated with the specified duty cycle. The value is specified in per mille (0...1000). By default the signal has a frequency of 1kHz

```
void myPwm.setFreq(uint32_t freq);
```
Changes the frequency of the PWM signal for subsequent calls of analogWrite.

```
void myPwm.tone(float freq, uint32_t duration );
```
Generates a sound with the specified frequency and duration. The duration is specified in ms. If duration is 0, an endless tone is generated.

```
void myPwm.stop();
```
Stops the signal output (pwm and tone).

```
void myPwm.setPwm( uint32_t high, uint32_t low );
```
Generates a PWM signal with freely adjustable HIGH and LOW time (in µs),
Minimum time for HIGH and LOW is 40µs.

Appendix:
Example diagram of the connection of a step motor via the SPI interface: