MobaTools - an Arduino library (not only) for model railroaders and model makers

Table of contents

1 Overview

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.
MoToTimebase	to trigger actions at regular intervals.
MoToButtons	Manage up to 32 buttons and switches per instance with debounce and event handling (pressed, released, short press, long press , single click, double click). The buttons/switches can be 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()

1.1 Differences to V2.3

Starting with version V2.4 the processors ESP32 and ATtiny are supported. For ATtiny it is a prerequisite that they contain a timer1 with 16 bit and a SPI or USI hardware. They should run with at least 8MHz and have more than 4k Flash. As of version 2.5, the Mega4809 (Arduino Nano Every) is also supported, from version 2.6 on also the UNO R4.
The support of 32-bit processors has been improved in general. This allows much higher steprates for STM32F103 and ESP32. Due to the very limited HW ressources (timer) this does not apply to the ESP8266 though.
With all 32-bit processors, dimmed values are also possible with Softled for the ON and OFF states (with AVR only fully ON or fully OFF is possible).

2 Supported processors

2.1 AVR

2.1.1 ATmega328P, ATMega2560, ATmega32u4, ATtiny

Boards: UNO Rev3, Nano classic, Mega2560, Leonardo, Micro etc.

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.
Only Attiny which also contain the Timer 1 are supported. Additionally the Attiny need a SPI or USI hardware to be compatible with the MobaTool.

2.1.2 Atmega 4809

Boards: Nano Every, UNO WiFi Rev2 .

As of V2.5 the Atmega4809 is supported too. The timer TCA0 is used with this processor. The millis() function of the megaAVR core uses timer TCB3. Unfortunately, the prescaler of TCA0 is also used for millis(). Therefore, this must not be changed in order not to influence the millis() function. This leads to the fact that the physical resolution of the pulse width for the servos is only 4µs (<= 1µs for all other processors).

2.2 Renesas RA4M1

Boards UNO R4 Minima, UNO R4 WiFi

From version V2.6 the new UNO R4 versions Minima and WiFi are supported. The timer used (GPT timer) is determined via the timer management of the Renesas core and is not absolutely fixed.

2.3 Raspberry Pi Pico (Rp2040 / RP2350)

Boards Pi Pico, Pi Pico W, Pi Pico 2, Pi Pico 2 W, Nano RP2040 Connect. Other boards may also work, but have not been tested.

As of version V2.7, the Rasperry Pi processors RP2040 and RP2350 are supported. Since MobaTools uses the Pico SDK, the core from Earle Philhower must be used. The Mbed OS core does not work.
The servo pulses are generated with the PWM HW and are therefore very stable. As the assignment of the PWM HW to the IOs is fixed and not every IO has its own PWM HW, not all combinations of IO assignment are possible (see chapter 3.1.3 about MoToServo).

2.4 STM32F103

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

For installation this URL can be used:
http://dan.drown.org/stm32duino/package_STM32duino_index.json
Open ‚preferences' and add this URL to ‚Additional Boards Manager URL's. Then the core can be installed via the boards manager.

2.5 ESP32

On the ESP32 platform the pulses for servos and leds are generated with the LED_PWM hardware of the ESP. This leads to very stable pulses, which especially with the servos leads to a jitter free stand of the servo even without switching off the pulses. Due to the HW, a maximum of 16 instances for servos and softleds are possible with the ESP. If the AnalogWrite function of the ESP is also used in addition to the MobaTools, conflicts can occur because this also uses the LED_PWM hardware.
As of MobaTools version 2.6.2, the 2.x version and the 3.x version of the ESP32 board manager are supported (tested with 3.0.3). However, the new HW versions of the ESP32 (S2,S3,C3) are not supported.

2.6 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.

2.7 General notes

Although there is no 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.

3 Class descriptions

3.1 MoToServo

The servo functions generate the pulses interleaved within a 20 ms cycle. As the 20ms are fixed, it must be possible to generate all pulses within this time. The number of pulses that can be generated depends on the maximum length and - as the pulses are generated overlapping - also on the minimum length of the pulses. By default, the limits are 0.7ms (min) and 2.3ms maximum. This results in the number of max 16 servos.
The above does not apply to the ESP and RP2040 versions. In this case, the limitation is due to the hardware (see also chapter 2.5 and 3.1.3).

3.1.1 Creating one Servo object

```
MoToServo  myServo;
```

Up to 16 objects are allowed.

3.1.2 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.
Return value is 0 if no assignment was possible, otherwise 1. On the ESP32 the return value is the pwm channel number (>0).

```
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);
void myServo.setSpeed(int Speed, HIGHRES );
```

Set the speed of movement of the servo. Speed' is the value by which the pulse length (as average value) changes every 20ms when the servo moves.
On AVR Speed is given in 0.5µs units (compatibility to older versions). On all other processors it is 0.125µs units. If the optional parameter 'HIGHRES' is specified once, it is also on AVR generally 0.125µs (for all instances).

Since the internal resolution of the pulse width - depending on the target platform - is greater than 0.125µs, it is possible that the pulse length may not change with each pulse.

For example, Speed=20 means that 8 seconds are needed to change the pulse length from 1ms to 2ms. Due to internal rounding errors, these times are not always fully accurate.

Speed=0 (default value) means direct pulse change (like standard servo library).

`void myServo.setSpeedTime(int Time);`

Time (in ms) the servo needs to move from 0° to 180°. This also applies if the values for 0° or 180° are changed. This means that these changes also influence the servo speed.
The maximum value is 30000 (30 seconds for the full movement). Because of internal roundings, the value is not reached exactly. Especially with large values (= small increments from pulse to pulse) the error can become larger.
If the `setSpeedTime` function is used, the high resolution is generally also used on AVR with `setSpeed` (no more compatibility mode).

`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.

`uint16_t 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/ESP32/RP2040
700µs otherwise (must not be set shorter).

`void setMaximumPulse( word length);`

Defines the pulse length for angle '180'. Default is
2600 µs for ESP8266/ESP32/RP2040
2300µs otherwise (must not be set longer).

3.1.3 IO Pins and RP2040

The integrated PWM hardware is used for the servo signals on the RP2040. GPIO 0 ... 15 each have their own HW, so that up to 16 servos can be used independently on these GPIOs. However, beginning witth GPIO 16, the same HW is used as from GPIO 0. These GPIOs can therefore only be used alternately to GPIO 0...15:

GP0 or GP16,

GP1 or GP17,

GP2 or GP18,

…

There are similar HW restrictions when using the servo class and analogWrite at the same time. In each of the pairs GP0/GP1,GP2/GP3 ... GP14/GP15 both pins can only be used either for analogWrite or for the servo class.

3.2 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.

The MobaTools manage the position of the stepper motor at any time. The position is the distance to a presettable reference point measured in steps. When the Arduino is started, the current position of the stepper motor is taken as the reference point. The reference point can be changed at any time.

The stepper motor can be moved in absolute values to a target position that has a corresponding distance to the reference point. This target position can be specified either in angular degrees or in steps. If angular degrees are used, it is important that the number of steps/revolution is specified correctly when setting up the stepper motor.
Depending on the current position of the stepper motor, the direction of rotation and the number of necessary steps to the target position are calculated.

Alternatively, relative steps can also be specified. In this case, the steps refer to the current position of the stepper motor. In this case, the direction of rotation is determined by the sign. Internally, a new target position is calculated from the current position and the specified relative steps, which is then approached.

In both cases, the internal position tracking of the motor remains correct. Relative and absolute control of the motor can be used mixed as desired.

3.2.1 Setting up of the stepper motor.

```
MoToStepper myStepper( long 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 STEPDIR)
`mode = STEPDIR ( formerly 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 STEPDIR the set microstepping must be taken into account)

3.2.2 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 a driver with step and dir input (mode STEPDIR) 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.attachEnable( uint16_t delay );
```
In FULLSTEP or HALFSTEP mode with unipolar steppers, they can be switched off without an additional enable pin. The stepper is switched off by switching off all 4 stepper pins. This also works if the stepper pins are controlled via SPI.

`delay`: Delay between switching pins on and 1st step in ms, also between last step and switching pins off.

ESP8266: This function cannot be used because the ESP8266 does not support FULLSTEP and HALFSTEP modes.

```
bool myStepper.autoEnable( bool isActive );
```
If the function to switch off the motor at standstill was activated with `attachEnable()`, it can be switched off/on again here during operation
`isActive`: true/false to switch the function on/off

Return value is the current state (true/false) . Without a previous `attachEnable()` the return value is always false.

```
bool myStepper.autoEnable(  );
```
Return value is the current state of autoenable (true/false) . Without a previous `attachEnable()` the return value is always false.

```
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 .

Internally, the rotation speed specified here is converted into the step rate and then `setSpeedSteps` is called. The limitations with regard to the maximum step rate (see chapter 3.2.3) therefore apply here in exactly the same way. A prerequisite for the correct calculation is that the steps/revs were specified correctly when setting up the stepper.

Function value is the new ramp length (as returned by the internal call of `setSpeedSteps)`

```
uint32_t myStepper.setSpeedSteps( uint32_t speed10 ); // 32-bit boards
uint16_t myStepper.setSpeedSteps( uint16_t speed10 ); // AVR
```

- Stepping speed of the motor in steps / 10sec.

If necessary, the ramp length is adjusted to keep the starting behavior approximately the same.

Function value is the new ramp length.

```
uint32_t myStepper.setMaxSpeed( uint32_t speed ); // 32-Bit boards
uint16_t myStepper.setMaxSpeed( uint16_t speed ); // AVR
```

- Stepping speed of the motor in steps / sec.

Otherwise like setSpeedSteps(speed10).

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

- Stepping speed of the motor in steps / 10sec. (for 32-Bit pr ocessors as unit32_t, because the steprate may become > 65000 see above)
- Ramp length in steps.

Function value is the current (possibly adjusted) ramp length.

General info about setSpeed:
As of version 2.5, it is allowed to set speed to 0. In this case, the motor will decelerate to a standstill, regardless of the set target position. The previously set target position remains stored and is approached again as soon as the speed is set greater than 0 again. The target position can also be changed during standstill. If the enable function is active, the stepper is switched off during standstill.

>>Further information on the step rate and ramp length in chap. 3.2.3 (Page 12)

```
uint32_t myStepper.setRampLen( uint32_t rampLen ); // 32-Bit boards
uint16_t myStepper.setRampLen( uint16_t rampLen ); // AVR
```

- Ramp length in steps. The permissible ramp length depends on the step rate, and a maximum of 16000 for high step rates. For step rates below 2steps/sec, ramping is no longer possible. If ramplen is outside the permissible range, the value is adjusted.

Function value is the current (possibly adjusted) ramp length.

```
void myStepper.doSteps(long stepcount );
void myStepper.move(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.setZero(long zeroPoint, long steps360);
```
The new reference point is set '`zeroPoint`' steps away from the current position.
Additionally, the number of steps for one revolution is set to ‚steps360'. This will take effect with the next 'setSpeed' command.

```
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 );
void myStepper.moveTo( 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 is 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();
long myStepper.distanceToGo();
```
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();
long myStepper.read(byte factor);
```
returns to actual position of the stepper in degrees (measured from reference point). If the argument 'factor' is specified, the angle is output in corresponding fractions, which increases

the accuracy of the position value. With factor=10, for example, the position is returned in 1/10°.
Be careful with large values of factor and positions far away from the reference point: As the angle values can then become much larger than the position values in steps, an overflow may occur.

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

```
int32_t myStepper.getSpeedSteps();
```
always returns the current speed in steps/10sec (also during accelerating/decelerating). The sign indicates the current direction of rotation.

3.2.3 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, (not on ESP) 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	
ESP8266	6250 Steps/sec	
UNO R4	12000 Steps/sec	
STM32F103	20000 Steps/sec	absolute maximum is about 40000 Steps/sec
RP2040	25000 Steps/sec	
ESP32	30000 Steps/sec	theoretical maximum is 50000 Steps/sec

The permissible ramp length depends on the step speed. Maximum is 16000 (AVR) or 160000(32-bit) for high step rates. At Steprates below 2steps / sec, no ramp is possible. If rampLen is outside the permitted range, the value is adjusted.

Taking into account the total load, the max steprates for AVR and ESP8266 can be slightly optimized (via #defines in MobaTools.h)

Also the maximum number of steppers (default is max 6 steppers) can be increased by a #define in the MobaTools.h file.

Rotate:
This is not really an endless rotation. Rather, the target point is set to the furthest possible value in the plus or minus direction (approx. 2 billion steps from the zero point). As the position of the stepper is also tracked internally with rotate, one of the commands doSteps / write / or writeSteps can be issued at any time to specify a new target.
If you actually want to rotate beyond this end point, the reference point must be set to the current position befor reaching the target and then rotate must be called again. However, starting from the reference point, this end point is only reached after more than 19 hr of continuous rotation, even with the highest possible step rate of the ESP32.

3.3 MoToSyncStepper

Class to synchronize the movement of up to MAXSTEPPER steppers. The speed of each stepper is set in such a way, that all steppers reach their target nearly at the same time.
IMPORTANT: ramp length is set to 0 when moving the steppers!! During the synced movement no changes to the stepper objects are allowed.
The functionality is derived from the MultiStepper class of AccelStepper

You should not control a single stepper of the group while the group is moving.

3.3.1 Creating an instance

```
MoToSyncStepper mySteppers;
```
Create the object. Steppers must be added later.

3.3.2 Methods

```
bool mySteppers.addStepper(MoToStepper& stepper);
```
Add a stepper to a group of synced steppers. The return value is false if there are too many steppers.

```
void mySteppers.setMaxSpeedSteps( uint32_t speed10 ); // 32-Bit boards
void mySteppers.setMaxSpeedSteps( uint16_t speed10 ); // AVR
```
Set the speed for the stepper that has the longest distance to the target (which will be the stepper with the highest speed). Value is in steps/10sec.

```
void mySteppers.moveTo( long *absTarget )
```
Parameter is the address of an array with the targets of all steppers. The order of the steppers is the same order as they were added to the object.

```
bool mySteppers.moving()
```
true as long as the steppers are moving.

3.4 MoToSoftLed

The class MoToSoftLed contains methods to switch a LED on and off softly (bulb simulation).

3.4.1 Create an instance

```
MoToSoftLed 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.

3.4.2 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.
On the ESP32 platform the return value is the pwm channel number (>0) and 0 if no attach was possible. On the other platforms always 1 is returned.

```
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 ); ( not with AVR processors )
```

`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 ); ( not with AVR processors )
```

`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.

3.5 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>.

3.5.1 Set up

`MoToTimer myTimer`
Creates an object, but doesn't start the timer

`MoToTimer myTimer( unsigned long time );`
Creates an object and immediately starts the timer if time is >0

3.5.2 Methods

`void myTimer.setTime( unsigned long time );`
Starts the timer. The value specifies the duration of the delay in ms.
If 0 is passed the timer is not started and stops if it is running.

`void myTimer.restart();`
The timer is restarted with the last time passed via setTime. The time will restart from the beginning even if the timer is currently already running.

`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'.

`unsigend long myTimer.getElapsed();`
Returns the current runtime of the timer (in ms) since last setTime or restart. If the timer is no longer running, the time last set with setTime is returned (it is then identical to runTime()).

`unsigend long myTimer.getRemain();`
Returns the remaining time (in ms) or 0 if not running. Same as the deprecated method getTime.

`unsigend long myTimer.getRuntime();`
Returns the last runtime value set by setTime().

`void myTimer.stop();`
Stops the timer before expiry. No 'expired' event is generated.

`unsigend long myTimer.getTime();`
Returns the remaining time (in ms) or 0 if not running. Deprecated, schould not be used in new Sketches.

3.6 MoToTimebase

A time base to trigger actions in regular time intervals The distances are kept more exactly than if you do this with the class MoToTimer.

3.6.1 Set up

```
MoToTimebase myBase;
```

After setup the time base is inactive.

3.6.2 Methods

```
void myBase.setBasetime( long baseTime );
```

Set time interval (in ms). If the time is specified negatively, the interval is set but the time base is not yet started.

If 0 is transferred as the interval, the clock becomes inactive and cannot be started.

```
void myBase.start();
```

Starts the time base if it is not yet running and a time interval is already set.

```
void myBase.stop();
```

Stops the time base (no more ticks are triggered). The value of the interval remains unchanged. The time base can be started again at any time.

```
bool myBase.tick();
```

Returns 'true' when the time base has been started and a time interval has expired.

```
bool myBase.running();
```

‚true‘, wenn the time base has been started and is running

```
bool myBase.inactive();
```

‚true‘ if the interval time is 0 (time base is inactive).

3.7 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).
The setting of the pins (pinMode) is done in the Lib. It is not necessary to do this in the setup().
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)

not pressed: 0
pressed: The time the button is already pressed in debounce time units (max 255)

`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 time for a long press has elapsed after pressing and holding the key and is reset after calling this method or when the key 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.

`uint8_t myButtons.released( uint8_t buttonNbr );`
>0 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.
The value returned is the duration of the press in `debTime` (debounce time) units. The maximum value is 255, even if the key was pressed longer.

`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.

`uint8_t myButtons.clicked( uint8_t buttonNbr );`
Returns whether the button was single or double clicked. The return value is:
`NOCLICK` (=0) if nothing has been clicked.
`SINGLECLICK` (=1) when a single click is detected.
`DOUBLECLICK` (=2) when a double click is detected.

After calling the method, the event is deleted, i.e. further calls return NOCLICK. The event is also deleted if the pushbutton was pressed again before the method was called.

Note that a click is also interpreted as a 'shortPress'. So a click will also generate a 'shortPress' event (see above). As a rule, you should either query on 'clicked' or on 'shortPress'.

If a line with #define CLICK_NO_SHORT is inserted before the #include <MobaTools.h>, the 'shortPress' event is also deleted if a single or double click was detected when calling 'clicked'. (see also the example 'Button_I2C')

3.8 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.

4 Appendix:

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