

## MobaTools – eine Arduino Bibliothek für Modellbahn und Modellbau

### Inhaltsverzeichnis

1 Übersicht.....	1
1.1 Unterschiede zur V2.3.....	2
2 Unterstützte Prozessoren.....	2
2.1 AVR ( Atmega, Attiny ).....	2
2.2 STM32F103.....	3
2.3 ESP32.....	3
2.4 ESP8266.....	3
2.5 Allgemeine Hinweise.....	3
3 Klassenbeschreibungen.....	4
3.1 MoToServo.....	4
3.1.1 Einrichten des Servos:.....	4
3.1.2 Methoden:.....	4
3.2 MoToStepper.....	6
3.2.1 Einrichten des Schrittmotors:.....	6
3.2.2 Methoden:.....	6
3.2.3 Zusatzinfos zum Stepper:.....	9
3.3 MoToSoftLed.....	11
3.3.1 Einrichten:.....	11
3.3.2 Methoden:.....	11
3.4 MoToTimer.....	13
3.4.1 Einrichten:.....	13
3.4.2 Methoden:.....	13
3.5 MoToTimebase.....	13
3.5.1 Einrichten:.....	14
3.5.2 Methoden:.....	14
3.6 MoToButtons.....	15
3.6.1 Einrichten:.....	15
3.6.2 Methoden:.....	16
3.7 MoToPwm ( nur auf ESP8266 ).....	18
4 Anhang:.....	19

## 1 Übersicht

Die MobaTools sind eine Zusammenstellung von Funktionen, die speziell im Modellbahnumfeld ( aber nicht nur dort ) oft gebraucht werden können. Sie erleichtern einige Standard-Aufgaben und ermöglichen so auch Einsteigern komplexe Sketches zu schreiben.

Zusammenfassung der Klassen:

Ab der Version 2.0 wurden die Klassennamen verändert und vereinheitlicht. Die alten Klassennamen sind aus Kompatibilitätsgründen noch gültig, sollten aber in neuen Sketchen nicht mehr verwendet werden. Die alten Klassennamen werden in der IDE nicht mehr rot markiert.

MoToServo	Ansteuerung von bis zu 16 Servos an beliebigen Pins. Die Aufrufe sind weitgehend kompatibel zur Standard Servolib des Arduino. Allerdings kann die
-----------	--

Geschwindigkeit, mit der sich das Servo bewegt, vorgegeben werden.

MoToStepper	Ansteuerung von Schrittmotoren. Im Gegensatz zur Standard Library 'Stepper' sind die Aufrufe nicht blockierend. Nachdem ein Referenzpunkt festgelegt wurde, lässt sich der Schrittmotor mit nahezu den gleichen Aufrufen wie in der MoTServo – Library auch absolut positionieren (in Winkelgraden). Der Schrittmotortreiber A4988 ( und alle Treiber mit Step/Dir Eingang ) werden unterstützt. Damit können bipolare Schrittmotoren (auch mit Microstepping) angesteuert werden. Eine Rampe für Anfahren und Bremsen kann festgelegt werden.
MoToSoftLed	Weiches auf- und Abblenden von Leds. Als Anschlußpins sind alle Digitalpins zulässig,.
MoToTimer	'Kurzzeitwecker' der einfach Zeitverzögerungen ohne Blockierung des Sketchablaufes im loop ermöglicht.
MoToTimebase	zur Auslösung von Aktionen in regelmäßigen Abständen.
MoToButtons	Verwalten von bis zu 32 Tasten und Schaltern pro Instanz mit Entprellen und Ereignisbehandlung (gedrückt, losgelassen, kurzes Drücken, langes Drücken, einfach-klick, doppel-klick). Die Tasten/Schalter können über eine User-Callback-Funktion eingelesen werden. Dies ermöglicht die Verwendung von Matrix-Anordnungen und z.B. I2C-Port-Expandern.
MoToPwm	Diese Klasse existiert nur für ESP8266. Auf dem ESP8266 können wegen der begrenzten Timerressourcen die intergrierten Funktionen tone(), analogWrite() und Servo() nicht parallel zu den MobaTools genutzt werden. MoToPwm stellt Methoden zur Verfügung, um tone() und analogWrite() zu ersetzen.

## 1.1 Unterschiede zur V2.3

Mit der Version V2.4 werden die Prozessoren ESP32 und Attiny unterstützt. Bei Attiny ist allerdings Voraussetzung, dass sie einen Timer1 mit 16 Bit und eine SPI oder USI Hardware beinhalten. Sie sollten mit mindestens 8MHz laufen und mehr als 4k Flash besitzen.

Die Unterstützung von 32-Bit Prozessoren wurde allgemein verbessert. Dadurch sind bei STM32F103 und ESP32 deutlich höhere Stepraten möglich. Aufgrund der nur sehr eingeschränkten HW-Peripherie ( Timer ) gilt dies allerdings nicht für den ESP8266.

Bei allen 32-Bit Prozessoren sind bei Softled für die Zustände EIN und AUS auch gedimmte Werte möglich ( Bei AVR nur voll AN oder ganz AUS ).

## 2 Unterstützte Prozessoren

### 2.1 AVR ( Atmega, Attiny )

Die MobaTools verwenden intensiv den Timer1 des Arduino (**AVR**). D.h. Alle anderen Libraries oder Funktionen, die ebenfalls den Timer1 verwenden, können nicht gleichzeitig genutzt werden. Dies betrifft z.B. die Standard-ServoLib und analogWrite an den Pins 9+10. Ist der Timer 3 vorhanden, so wird er anstelle des Timer 1 verwendet. Das betrifft z.B. Arduino Leonardo, Micro und Mega.

Nur Attiny die ebenfalls den Timer 1 enthalten, werden unterstützt. Zusätzlich benötigen die Attiny

eine SPI oder USI Hardware, um mit den MobaTool kompatibel zu sein.

## 2.2 STM32F103

Auf der **STM32F103**-Plattform wird der Timer 4 verwendet. Es muss der Core von RogerClark ( [https://github.com/rogerclarkmelbourne/Arduino\\_STM32](https://github.com/rogerclarkmelbourne/Arduino_STM32) ) verwendet werden.

Zur Installation kann diese URL benutzt werden:

[http://dan.drown.org/stm32duino/package\\_STM32duino\\_index.json](http://dan.drown.org/stm32duino/package_STM32duino_index.json)

Die URL bei den ‚Voreinstellungen‘ zu den Boardverwalter-URL’s hinzufügen, dann kann der Core über den Boardverwalter installiert werden.

## 2.3 ESP32

Auf der **ESP32** Plattform werden die Impulse für Servos und Led’s mit der LED\_PWM-Hardware des ESP erzeugt. Dies führt zu sehr stabilen Impulsen, was speziell bei den Servos zu einem ruhigen Stand des Servo auch ohne Abschalten der Impulse führt. Aufgrund der HW sind beim ESP insgesamt maximal 16 Instanzen für Servos und Softleds möglich. Wird neben den MobaTools auch die AnalogWrite Funktion des ESP genutzt, kann es zu Konflikten kommen, da diese ebenfalls die LED\_PWM-Hardware nutzt.

## 2.4 ESP8266

Da auf dem **ESP8266** die Timerstruktur vollkommen anders ist, als bei den obigen Prozessoren mussten für diese Plattform einige Konzepte geändert werden. Dies führt dazu, dass der GPIO16 nicht für Pulsausgänge ( Softled, Servo, Step ) genutzt werden kann. Als Dir-Ausgang beim Stepper kann er genutzt werden, wenn die enable-Funktion nicht verwendet wird.

## 2.5 Allgemeine Hinweise

Obwohl es für Softleds und Stepper keine harte Grenze bezüglich der Zahl der instanziierten Objekte mehr gibt, sind die Leistungsgrenzen des verwendeten Prozessors zu beachten. Sowohl Stepper als auch Softleds werden in ein und derselben Interruptroutine verwaltet. Softleds belasten den Interrupt nur während des Auf- und Abblendens. Bei Steppern ist die Belastung abhängig von der Steprate (Häufigkeit des Interrupts) und der Rampe ( Während der Rampe ist die Rechenbelastung im Interrupt höher ).

Bei UNO/Nano/Leonardo und Micro ergibt sich die Begrenzung bereits aus der Zahl der verfügbaren Pins. Bei einem Mega ist aber verstärkt darauf zu achten.

Versuche mit 6 Steppern und 6 ständig pulsierenden Leds waren noch problemlos. Die maximale Zahl der Stepper ist per #define in MoBaTools.h auf 6 begrenzt. Wenn z.B. keine Softleds verwendet werden, kann dies auch hochgesetzt werden.

## 3 Klassenbeschreibungen

### 3.1 MoToServo

Die Servo Funktionen erzeugen die Impulse verschachtelt innerhalb eines 20ms Zyklus. Da die 20ms fest sind, müssen innerhalb dieser Zeit alle Impulse erzeugbar sein. Die Zahl der erzeugbaren Impulse hängt von der Maximallänge und – da die Impulse überlappend erzeugt werden – auch von der Minimallänge der Impulse ab. Standardmäßig liegen die Grenzen bei 0,7ms (Min) und 2.3ms maximal. Daraus ergibt sich die Zahl von 16 anschließbaren Servos.

#### 3.1.1 Einrichten des Servos:

```
MoToServo myServo;
```

Es werden keine Parameter benötigt.

#### 3.1.2 Methoden:

```
byte myServo.attach( int pin );
```

Ordnet dem Servo einen pin zu, an dem die Impulse ausgegeben werden. Der Pin wird auf Ausgang geschaltet, es werden aber noch keine Impulse erzeugt.

Rückgabewert ist 0, wenn keine Zuordnung möglich war, sonst 1. Auf dem ESP32 ist der Rückgabewert die pwm-Kanalnummer.

```
byte myServo.attach( int pin, bool autoOff );
```

Wird der optionale Parameter 'autoOff' mit dem Wert 'true' übergeben, so wird der Impuls automatisch abgeschaltet, wenn sich die Länge für mehr als 1sec nicht ändert.

```
byte myServo.attach( int pin, int pos0, int pos180, bool autoOff );
```

Mit den Parametern pos0 und pos180 kann vorgegeben werden, welche Impulslängen den Winkeln '0' bzw. '180' zugeordnet werden. ( der Parameter autoOff kann auch hier entfallen)

```
byte myServo.detach();
```

Die Zuordnung des Servos zum Pin wird aufgehoben, es werden keine Impulse mehr erzeugt. Wenn diese Methode während eines aktiven Pulses aufgerufen wird, wird das Pulsende abgewartet. So entstehen keine abgeschnittenen Pulse.

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

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

Bewegungsgeschwindigkeit des Servos vorgeben. 'Speed' ist der Wert um den sich die Impulslänge ( als Mittelwert ) alle 20ms verändert, wenn sich das Servo bewegt.

Auf AVR und STM32 wird Speed in 0,5µs Einheiten angegeben ( Kompatibilität zu älteren Versionen ) . Auf den ESP Prozessoren sind es 0,125µs Einheiten. Wird einmal der optionale Parameter ‚HIGHRES‘ angegeben, sind es auch auf AVR und STM generell 0,125µs ( für alle Instanzen ).

Da die interne Auflösung der Impulsbreite – je nach Zielplattform – größer ist als 0,125µs kann dies bedeuten, dass sich die Pulslänge gegebenenfalls nicht bei jedem Impuls ändert.

Speed=20 bedeutet z.B., dass 8s gebraucht werden, um die Impulslänge von 1ms auf 2ms zu ändern.

Speed=0 ( default Wert) bedeutet direkte Impulsänderung (wie Standard-Servo Library )

```
void myServo.write(int angle);
```

Zielposition vorgeben. Der Servo bewegt sich mit der vorgegebenen Geschwindigkeit zu dieser Position. Bei Werten von 0...180 Wird der Wert als Winkel interpretiert, und entsprechend in eine Impulslänge umgerechnet. Werte > 180 werden als Impulslänge in  $\mu\text{s}$  interpretiert, wobei auf die minimale ( $700\mu\text{s}$ ) bzw. maximale ( $2300\mu\text{s}$ ) Impulslänge begrenzt wird.

Der write-Befehl wirkt sofort, auch wenn das Servo seine beim vorhergehenden write gesetzte Position noch nicht erreicht hat. Gegebenenfalls ändert das Servo auch sofort die Drehrichtung.

Ist dies der erste 'write' Aufruf nach einem 'attach', so wird die vorgegebene Impulslänge sofort ausgegeben (Startposition).

```
byte myServo.moving();
```

Information über den Bewegungszustand des Servo.

0: Das Servo steht still

>0: Restweg, den das Servo bis zum Ziel noch hat. In % vom Gesamtweg seit dem letzten 'write'.

```
byte myServo.read();
```

Momentane Position des Servos in Winkelgraden.

```
byte myServo.readMicroseconds();
```

Momentane Position des Servos in  $\mu\text{s}$  (Impulslänge).

Im Gegensatz zur Standard-Lib entsprechen diese Werte nicht unbedingt dem mit dem letzten 'write' übergebenen Wert. Während sich das Servo noch bewegt, wird die tatsächliche Position zurückgegeben.

Obwohl es keinen eigenen 'stop' Befehl gibt, lässt sich dies mit der Sequenz

```
myServo.write( myServo.readMicroseconds() );
```

erreichen. Damit wird die Momentanposition als neue Zielposition vorgegeben, was zum sofortigen Stop des Servos führt.

```
byte myServo.attached();
```

Gibt 'true' zurück, wenn ein Pin zugeordnet ist

```
void setMinimumPulse( word length);
```

Ordnet dem Winkel '0' eine Impulslänge zu, Standard ist

$550\mu\text{s}$  für ESP8266

$500\mu\text{s}$  für ESP32

$700\mu\text{s}$  sonst ( darf nicht kleiner gemacht werden )

```
void setMaximumPulse( word length);
```

Ordnet dem Winkel '180' eine Impulslänge zu, Standard ist

$2600\mu\text{s}$  für ESP8266

$2300\mu\text{s}$  sonst ( darf nicht länger gemacht werden )

## 3.2 MoToStepper

Damit können bipolare und unipolare Schrittmotore angesteuert werden. Im Gegensatz zur Arduino Standard Library sind die Aufrufe nicht blockierend. Das Programm im 'loop' läuft weiter, während sich der Schrittmotor dreht.

Neben der Schrittgeschwindigkeit kann auch einen Anfahr- und Bremsrampe vorgegeben werden. Die Rampenlänge wird über die Schrittzahl definiert. D.h. es wird angegeben, wie viele Schritte benötigt werden, um vom Stillstand auf die eingestellte Geschwindigkeit zu kommen.

Bei Rampenlänge 0 (default) verhält sich der Schrittmotor wie bisher ( vor V1.1 ).

Wird die Schrittgeschwindigkeit verändert ohne eine neue Rampenlänge vorzugeben, so wird die Rampenlänge so angepasst, dass das Anfahrverhalten etwa gleich bleibt.

Die MobaTools verwalten jederzeit die Position des Schrittmotors. Die Position ist der Abstand zu einem vorgebbaren Referenzpunkt in Schritten gemessen. Beim Starten des Arduino wird die momentane Stellung des Schrittmotors als Referenzpunkt genommen. Der Referenzpunkt kann jederzeit verändert werden.

Der Schrittmotor kann in Absolutwerten zu einer Zielposition gefahren werden, die einen entsprechenden Abstand zum Referenzpunkt hat. Diese Zielposition kann wahlweise in Winkelgraden oder in Schritten vorgegeben werden. Werden Winkelgrade verwendet, ist es wichtig, dass die Zahl der Schritte/Umdrehung beim Einrichten des Schrittmotors korrekt angegeben wird. Abhängig von der aktuellen Position des Schrittmotors wird Drehrichtung und die Zahl der notwendigen Schritt bis zur Zielposition berechnet.

Alternativ können auch relative Schritte vorgegeben werden. Dann beziehen sich die Schritte auf die momentane Position des Schrittmotors. Die Drehrichtung wird in diesem Fall durch das Vorzeichen bestimmt. Intern wird aus der aktuellen Position und den vorgegebenen relativen Schritten eine neue Zielposition berechnet, die dann angefahren wird.

In beiden Fällen bleibt die interne Positionsverfolgung des Motors korrekt. Relative und absolute Steuerung des Motors kann beliebig gemischt genutzt werden.

### 3.2.1 Einrichten des Schrittmotors:

```
MoToStepper myStepper( long steps360, byte mode );
```

mode gibt an, ob der Motor im FULLSTEP oder HALFSTEP Modus angesteuert wird. Wird der Parameter weggelassen, wird HALFSTEP angenommen. ( Ausser bei ESP8266, da wird der einzig zugelassene Wert STEPDIR gesetzt )

mode=STEPDIR ( vormals A4988 ) gibt an, dass der Motor über einen Schrittmotortreiber mit Step- und Direction Eingang ( z.B. der A4988 ) angeschlossen ist. Beim ESP8266 ist dies der einzig gültige Mode!

steps360 ist die Zahl der Schritte, die der Motor für eine Umdrehung braucht ( im jeweils angegebenen Mode, bei STEPDIR ist das eingestellte Microstepping zu berücksichtigen)

### 3.2.2 Methoden:

```
byte myStepper.attach( byte spi );
```

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

Zuordnung der Ausgangssignale. Die Signale können entweder an 4 einzelnen Pins, oder er die SPI-Schnittstelle ausgegeben werden. Für den Parameter `spi` sind die Werte

SPI\_1, SPI\_2, SPI\_3 oder SPI\_4 zulässig. Es werden immer 16 Bit herausgeschoben. Die Parameter geben an, an welcher Position die 4er Gruppe steht. SPI\_4 sind die zuerst herausgeschobenen Bits und stehen deshalb am Ende der Schieberegisterkette. Wird nur ein 8-Bit Schieberegister angeschlossen, können nur die Werte SPI\_1 und SPI\_2 genutzt werden.

Die SPI-Schnittstelle wird nur aktiviert, wenn mindestens ein Schrittmotor über SPI angeschlossen wird. Ein Beispiel zur Beschaltung ist im Anhang zu sehen.

Funktionswert ist 0 ('false') wenn keine Zuordnung möglich war.

Über die 4 Ausgangspins können sowohl unipolare als auch ( über eine doppelte H-Brücke ) bipolare Motoren angesteuert werden.

Diese Varianten können auf einem ESP8266 nicht genutzt werden.

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

Bei der Ansteuerung über den A4988 ( oder kompatiblen ) Treiber werden hier die Ports für ‚Step‘ und ‚Direction‘ angegeben.

Funktionswert ist 0 ('false') wenn keine Zuordnung möglich war.

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

Hiermit wird ein Pin definiert, über den der Motor ein- bzw ausgeschaltet oder in einen Stromsparmodus gebracht werden kann. Der Pin ist immer aktiv, während Steps ausgegeben werden.

pinEna: die Arduino Pinnummer

delay: Verzögerung zwischen Pin-Einschalten und 1. Step in ms, Ebenso zwischen letztem Step und Pin Ausschalten.

active: Ausgang HIGH aktiv oder LOW aktiv

**ESP8266:** Wird diese Funktion genutzt, kann der Dir-Ausgang nicht auf dem GPIO16 gelegt werden

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

Die Pinzuordnung wird aufgehoben.

```
uint16_t myStepper.setSpeed(int rpm10 );
```

Rotationsgeschwindigkeit ( in Umdrehungen / min ) einstellen. Der Wert muss in rpm\*10 angegeben werden.

Funktionswert ist die aktuelle Rampenlänge

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

```
uint16_t myStepper.setSpeedSteps( uint16_t speed10 ); // AVR
```

- Schrittgeschwindigkeit des Motors in Steps/10sec.

Die Rampenlänge wird gegebenenfalls angepasst, um das Anfahrverhalten in etwa gleich zu halten.

Funktionswert ist die aktuelle Rampenlänge.

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

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

- Schrittgeschwindigkeit des Motors in Steps/10sec. ( bei 32-Bit Prozessoren als uint32\_t, da die Steprate > 65000 werden darf )

- Rampenlänge in Steps.

Funktionswert ist die aktuelle (gegebenenfalls angepasste) Rampenlänge.

>>Weitere Informationen zur Steprate und Rampenlänge in Kap 3.2.3 ( Seite 9)

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

Rampenlänge in Steps. Die zulässige Rampenlänge ist von der Schrittgeschwindigkeit abhängig, und maximal 16000 für hohe Stepraten. Bei Stepraten unter 2Steps/sec ist keine Rampe mehr möglich. Liegt rampLen außerhalb des zulässigen Bereiches, wird der Wert angepasst.

Funktionswert ist die aktuelle (gegebenenfalls angepasste) Rampenlänge.

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

Zahl der Schritte, die der Motor ausführen soll. Das Vorzeichen gibt die Drehrichtung an.

Referenzpunkt für den Start der Schritte ist immer die momentane Motorposition. Und zwar auch dann, wenn sich der Motor zum Zeitpunkt des Befehls bereits dreht. Es wird dabei aus der Zahl der Schritte eine neue Zielposition errechnet, die dann angefahren wird ( mit Rampe ). Dies kann dazu führen, dass der Motor erst über das Ziel hinausfährt, und sich dann zurück zur Zielposition dreht.

doSteps(0) führt z.B. dazu, dass der Motor abbremst und zu der Position zum Zeitpunkt des Befehls zurückdreht,

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

Der Motor dreht bis zum nächsten Stop-Befehl.( 1= forwards, -1 = rückwärts )

Rotate(0) hält den Motor an (mit Rampe). Der Motor bleibt am Ende der Bremsrampe stehen.

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

Hält den Motor sofort an (Notstop).

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

Die aktuelle Position des Motors wird als Referenzpunkt für den 'write Befehl mit absoluter Positionierung genommen.

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

Der neue Referenzpunkt wird ‚zeroPoint‘ steps entfernt von der aktuellen Position gesetzt.

```
void myStepper.setZero(long zeroPoint, long steps360);
```

Der neue Referenzpunkt wird ‚zeroPoint‘ steps entfernt von der aktuellen Position gesetzt.

Zusätzlich wird die Zahl der Steps für eine Umdrehung neu gesetzt. Dies wirkt sich erst beim nächsten ‚setSpeed‘ Kommando aus.

```
void myStepper.write( long angle );
```

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

Der Motor bewegt sich zum angegebenen Winkel ( gemessen von setZero-Punkt ). Der vorgebbare Winkel ist nicht auf 360° beschränkt. Z.B. bedeutet angle=3600 10 Umdrehungen vom setZero Punkt. Wird danach z.B. angle= -360 übergeben, bedeutet das nicht 1 Umdrehung zurück, sondern 11 Umdrehungen zurück! (-360° vom setZero Punkt)

Der 2. Aufruf erlaubt es, den Winkel als Bruch anzugeben. Wird factor = 10 gesetzt, wird angle in 1/10° interpretiert.

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

Der Motor bewegt sich zu der Position, die stepPos Schritte vom setZero Punkt entfernt ist



```
byte myStepper.moving();
    =0 wenn der Motor steht
    >0...<=100 Restlicher Weg bis zum Zielpunkt in % vom Gesamtweg seit letztem 'write' oder
    'doSteps' Befehl. Dies beinhaltet auch gegebenenfalls einen ,Umweg', wenn aufgrund der
    Rampe der Stepper erst über das Ziel hinaus, und dann wieder zurückbewegt wird. In diesem
    Fall können Werte über 100% entstehen.

long myStepper.stepsToDo();
    gibt zurück wie viel Schritte noch bis zum Ziel ausgeführt werden müssen. Dies beinhaltet
    auch gegebenenfalls einen ,Umweg', wenn aufgrund der Rampe der Stepper erst über das Ziel
    hinaus, und dann wieder zurückbewegt wird.

long myStepper.read();
    gibt zurück an welchem Winkel sich der Motor befindet ( gemessen vom setZero Punkt).

long myStepper.readSteps();
    gibt zurück wie viel Schritte sich der Motor vom setZero Punkt entfernt befindet.

int32_t myStepper.getSpeedSteps();    // (ESP8266)
int16_t myStepper.getSpeedSteps();    // AVR+STM
    gibt immer die aktuelle Geschwindigkeit in Steps/10sec zurück ( auch während
    Beschleunigen/Bremsen )
```

### 3.2.3 Zusatzinfos zum Stepper:

Absolute und relative Befehle können gemischt verwendet werden. Intern werden die ausgeführten Schritte immer mitgezählt ( in einer long Variablen ). Solange diese Variable nicht überläuft ( passiert nach gut 2Mrd Steps in eine Richtung ), weiß die Steuerung immer, wo sich der Motor befindet. 'setZero' setzt diesen Zähler zu 0.

Die maximale Geschwindigkeit hängt mit der IRQ-Rate zusammen und ist dafür ausgelegt, dass auch bis zu 6 Stepper mit dieser Geschwindigkeit betrieben werden können. Außerdem werden auch die Softleds im gleichen Interrupt bearbeitet, was diesen ebenfalls belastet. Die erreichbaren Stepraten sind daher auch von der Zielplattform abhängig. Default sind derzeit folgende Werte:

AVR	2500 Steps/sec	
ESP8266	6250 Steps/sec	
STM32F103	20000 Steps/sec	bei schwacher Belastung des Prozessors auch mehr
ESP32	30000 Steps/sec	bei schwacher Belastung des Prozessors auch mehr

Die zulässige Rampenlänge ist von der Schrittgeschwindigkeit abhängig, und maximal 16000(AVR) bzw. 160000(32-Bit) für hohe Stepraten. Bei Stepraten unter 2Steps/sec ist keine Rampe mehr möglich. Liegt rampen außerhalb des zulässigen Bereiches, wird der Wert angepasst.

Unter Berücksichtigung der Gesamtbelastung kann die maximale Steprate für AVR und ESP8266 ( über #defines in MobaTools.h ) auch noch etwas ,getuned' werden.

Auch die maximale Zahl der Stepper (default is max 6 Stepper ) lässt sich über einen #define in der Datei MobaTools.h hochsetzen.

### Rotate:

Effektiv ist dies kein Endlosdrehen. Vielmehr wird der Zielpunkt auf den weitestmöglichen Wert in

plus bzw minus Richtung gesetzt ( ca. 2Mrd Steps vom Nullpunkt ). Da auch bei rotate die Position des Steppers intern verfolgt wird, kann auch jederzeit einer der Befehle `doSteps` / `write` / oder `writeSteps` abgesetzt werden, um ein neues Ziel vorzugeben.

Soll tatsächlich über diesen Endpunkt hinaus gedreht werden, dann muss rechtzeitig der Referenzpunkt zur aktuellen Position ‚nachgezogen‘ , und dann `rotate` erneut aufgerufen werden. Allerdings wird – ausgehend vom Referenzpunkt - dieser Endpunkt auch bei der höchstmöglichen Steprate des ESP32 erst nach gut einem Tag ununterbrochenen Drehens erreicht.

### 3.3 MoToSoftLed

Die Klasse MoToSoftLed enthält Methoden, um eine Led 'weich' ein- und auszuschalten. Die PWM-Impulse während des Auf/Abblendens werden per Software erzeugt. Daher funktioniert dies an allen digitalen Ausgängen.

#### 3.3.1 Einrichten:

```
SoftLed myLed;
```

Die Zahl der ansteuerbaren Leds ist grundsätzlich nicht begrenzt. In der Praxis bestimmen die vorhanden Pins und die Leistungsfähigkeit des Microprozessors (Ram und Geschwindigkeit) die sinnvoll nutzbare Anzahl.

#### 3.3.2 Methoden:

```
byte myLed.attach( byte pinNr );
```

```
byte myLed.attach( byte pinNr, byte Invert );
```

Der Digitalausgang muss nicht PWM-fähig sein. Ist der Parameter Invert angegeben und ‚True‘, so ändert sich die Ausgangslogik: ON ist dann LOW am Ausgang, OFF ist HIGH am Ausgang.

ESP8266: Gpio16 kann nicht verwendet werden.

Auf dem ESP32 ist der Rückgabewert die pwm-Kanalnummer ( >0 ) und 0 wenn kein attach möglich war. Auf den anderen Plattformen wird immer 1 zurückgegeben.

```
void myLed.riseTime( uint16_t Wert );
```

Der Wert gibt die Zeit in ms an, bis die Led ihre volle Helligkeit erreicht hat bzw. dunkel ist. Da intern nur bestimmte Stufen möglich sind, ist die tatsächliche Zeit eine möglichst gute Annäherung an den übergebenen Wert.

Der Maximalwert ist prozessorabhängig ( ca 5sec bei STM, 10s.bei AVR, 65s bei ESP )

```
void myLed.on();
```

Die Led wird eingeschaltet.

```
void myLed.off();
```

Die Led wird ausgeschaltet.

```
void myLed.on(uint8_t brightness ); ( nicht bei AVR-Prozessoren )
```

brightness = 0...100 PWM-Wert für ‚ON‘ in %

Die Led wird eingeschaltet und leuchtet dann mit brightness % der maximalen Helligkeit.

Der PWM-Wert wird für nachfolgende Einschaltvorgänge mit on/toggle/write gespeichert.

Aus Kompatibilitätsgründen kann die Methode auch bei AVR-Prozessoren aufgerufen werden, der übergebene Parameter wird allerdings ignoriert.

```
void myLed.off(uint8_t brightness ); ( nicht bei AVR-Prozessoren )
```

brightness = 0...100 PWM-Wert für OFF in %

Die Led wird ausgeschaltet, leuchtet aber noch mit dem eingestellten PWM-Wert.

Der PWM-Wert wird für nachfolgende Ausschaltvorgänge mit off/toggle/write gespeichert.

Der PWM-Wert für off muss kleiner sein als der für on.

Aus Kompatibilitätsgründen kann die Methode auch bei AVR-Prozessoren aufgerufen werden, der übergebene Parameter wird allerdings ignoriert.

```
void myLed.toggle();
```

Die Led wird umgeschaltet.

```
void myLed.write( byte Zustand);
```

Zustand = ON oder OFF. Die Led wird entsprechend ‚Zustand‘ ein- oder ausgeschaltet.

```
void myLed.write(byte Zustand, byte Type );
```

mit Type = LINEAR oder BULB kann die Charakteristik des Auf- Abblendens verändert werden.

### 3.4 MoToTimer

Die Klasse MoToTimer enthält Methoden für einfache Zeitverzögerungen.

#### 3.4.1 Einrichten:

```
MoToTimer myTimer
```

#### 3.4.2 Methoden:

```
void myTimer.setTime( unsigned long Zeit );
```

Startet den Timer. Der Wert gibt die Laufzeit des Timers in ms an.

Wird 0 übergeben wird der Timer nicht gestartet und stoppt falls er gerade läuft .

```
void myTimer.restart();
```

Der Timer wird mit der zuletzt per setTime übergebenen Zeit neu gestartet. Die Zeit startet auch dann wieder von vorn, wenn der Timer derzeit bereits läuft.

```
bool myTimer.running();
```

'true' während der Timer läuft, 'false' sonst..

```
bool myTimer.expired();
```

Ereignis ,Timerablauf'. Funktionswert ist ,true' nur beim ersten Aufruf nach Timerablauf, sonst immer ,false'.

```
unsigned long myTimer.getElapsed();
```

gibt die bisherige Laufzeit des Timers zurück ( in ms ). Läuft der Timer nicht mehr, wird die zuletzt mit setTime gesetzte Zeit zurückgegeben ( ist dann identisch zu runTime() )

```
unsigned long myTimer.getRemain();
```

gibt die Restlaufzeit des Timers zurück ( in ms ). Läuft der Timer nicht ( mehr ), wird immer 0 zurückgegeben. Alternativ ist auch noch der bisherige Methodename getTime() gültig, sollte aber bei neuen Programmen nicht mehr verwendet werden.

```
unsigned long myTimer.getRuntime();
```

gibt den letzten per setTime() gesetzten Laufzeitwert zurück.

```
unsigned long myTimer.getTime();
```

sollte nicht mehr verwendet werden. Neu ist getRemain() .

Gibt die Restlaufzeit des Timers zurück ( in ms ). Läuft der Timer nicht ( mehr ), wird immer 0 zurückgegeben.

```
void myTimer.stop();
```

den Timer vorzeitig anhalten. Es entsteht kein ,expired' Ereignis.

### 3.5 MoToTimebase

Mit der Klasse MoToTimebase lassen sich Aktionen in regelmäßigen Abständen auslösen (Taktgeber). Die Abstände werden genauer eingehalten, als wenn man dies mit der Klasse MoToTimer nachbildet.

### 3.5.1 Einrichten:

```
MoToTimebase myBase;
```

Unmittelbar nach dem Einrichten ist der Taktgeber noch inaktiv.

### 3.5.2 Methoden:

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

Zeitintervall setzen ( in ms ). Wird die Zeit negativ angegeben, so wird das Intervall zwar gesetzt, der Taktgeber aber noch nicht gestartet.

Wird 0 als Intervall übergeben, so wird der Taktgeber inaktiv, und kann nicht gestartet werden.

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

Startet den Taktgeber, wenn er noch nicht läuft, und ein Intervall gesetzt ist.

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

Stoppt den Taktgeber ( es werden keine Ticks mehr ausgelöst ). Der Wert des Intervalls bleibt erhalten. Der Taktgeber kann jederzeit wieder gestartet werden.

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

Gibt ‚true‘ zurück, wenn der Taktgeber gestartet ist und die Intervallzeit abgelaufen ist.

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

‚true‘, wenn der Taktgeber aktiv und gestartet ist.

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

‚true‘ wenn keine Intervallzeit gesetzt ist.

### 3.6 MoToButtons

Diese Klasse enthält Methoden zum Verwalten von Tastern und Schaltern.

Standardmäßig können bis zu 16 Taster/Schalter verwaltet werden. Diese Voreinstellung kann geändert werden, um RAM zu sparen (bis zu 8 Taster) oder um bis zu 32 Taster zu verwalten (mit zusätzlichem RAM-Verbrauch).

Dies kann durch Einfügen von '#define MAX32BUTTONS' oder '#define MAX8BUTTONS' vor dem #include <MoBaTools.h> erreicht werden.

Soll nur diese Klasse verwenden wollen, kann auch #include <MoToButtons.h> verwendet werden.

Der Zustand der Tasten/Schalter kann über eine User-Callback-Funktion gelesen werden. Dies ermöglicht Designs, bei denen die Tasten/Schalter in einer Matrix angeordnet sind und/oder über einen Port-Extender gelesen werden. Der Rückgabewert dieser Funktion muss ein 8-Bit-, 16-Bit- oder 32-Bit-Wert sein, entsprechend der maximal verwaltbaren Anzahl von Schaltern. Jeder Taster/Schalter wird durch ein Bit repräsentiert, wobei '1' bedeutet, dass der Taster gedrückt ist. Die Tasternummer in den Methoden entspricht der Bitnummer im Rückgabewert der Einlesefunktion.

Der Datentyp 'button\_t' wird automatisch auf den richtigen Typ gesetzt ( uint8\_t, uint16\_t oder uint32\_t ) und kann zur Definition des Typs der Callback-Funktion verwendet werden.

#### 3.6.1 Einrichten:

##### Variante mit direktem Einlesen von Pins:

```
MoToButtons myButtons( const uint8_t pinNumbers[], const uint8_t pinCnt,
uint8_t debTime, uint16_t pressTime );
```

- Par1: Adresse eines Array mit den Pinnummern der Schalter. Die Schalter müssen gegen Gnd angeschlossen sein ( Gedrückt ergibt LOW am Pin ).  
Das Einrichten der Pins ( pinMode ) wird in der Lib gemacht. Es ist nicht notwendig, dies im setup() zu erledigen.
- Par2: Zahl der angeschlossenen Schalter ( Länge des Arrays von Par1 )
- Par3: Entprellzeit in ms
- Par4: Zeit ( in ms ) um zwischne kurzen und langen Tastendrücken zu unterscheiden. Der Maximalwert ist Entprellzeit \* 255

##### Variante mit Einlesen der Taster über eine Callback-Funktion:

```
MoToButtons myButtons( button_t (*getHWbuttons)(), uint8_t debTime,
uint16_t pressTime );
```

- Par1: Adresse der Callbackfunktion für das Einlesen der Taster/Schalter.  
Bei dieser Variante wird von MotoButtons keinerlei Initiierung vorgenommen ( z.B. pinMode ), da die Taster mit der Callback-Funktion auf beliebigen Wegen – auch z.B. über I2C oder SPI – eingelesen werden können.
- Par2: Entprellzeit in ms
- Par3: Zeit ( in ms ) um zwischne kurzen und langen Tastendrücken zu unterscheiden. Der Maximalwert ist Entprellzeit \* 255

In beiden Fällen kann optional ein weiterer Parameter angegeben werden, um die Doppeklickzeit zu ändern ( default ist 300ms ).

### 3.6.2 Methoden:

`void myButtons.processButtons();`

Diese Methode muss möglichst häufig im `loop()` aufgerufen werden. Wenn sie weniger häufig als die Entprellzeit aufgerufen wird, ist die Unterscheidung zwischen kurzem und langem Drücken ungenau.

`bool myButtons.state( uint8_t buttonNbr );`

Gibt den entprellten Zustand der Taste 'buttonNbr' zurück.

`button_t myButtons.allStates();`

Gibt den entprellten Zustand aller Taster/Schalter zurück ( bitcodiert wie im Returnwert der Lesefunktion ).

`button_t myButtons.changed();`

Alle Bits bei denen sich der Zustand seit dem letzten Aufruf von `myButtons.changed()` geändert hat, sind gesetzt.

`bool myButtons.shortPress( uint8_t buttonNbr );`

True, wenn die Taste 'buttonNbr' kurz gedrückt wurde. Dieses Ereignis wird gesetzt, wenn die Taste nach einem kurzen Druck losgelassen wurde und zurückgesetzt nach dem Aufruf dieser Methode oder wenn die Taste erneut gedrückt wird.

`bool myButtons.longPress( uint8_t buttonNbr );`

True, wenn die Taste 'buttonNbr' lang gedrückt wurde. Dieses Ereignis wird gesetzt, wenn nach dem Drücken und Halten der Taste die Zeit für einen langen Druck abgelaufen ist und zurückgesetzt nach dem Aufruf dieser Methode oder wenn die Taste erneut gedrückt wird.

`bool myButtons.pressed( uint8_t buttonNbr );`

True, wenn die Taste 'buttonNbr' gedrückt wurde. Dieses Ereignis wird gesetzt, wenn die Taste gedrückt wird und zurückgesetzt nach dem Aufruf dieser Methode oder wenn die Taste losgelassen wird.

`bool myButtons.released( uint8_t buttonNbr );`

True, wenn die Taste 'buttonNbr' losgelassen wurde. Dieses Ereignis wird gesetzt, wenn die Taste losgelassen wird und zurückgesetzt nach dem Aufruf dieser Methode oder wenn die Taste erneut gedrückt wird.

`void myButtons.forceChanged();`

Der nächsten Aufrufe von 'changed()' oder 'pressed()'/released()' verhalten sich so, als ob der Taster/Schalter seinen Zustand in die aktuelle Position geändert hätte. Die Aufrufe von `longPress()` und `shortPress()` sind davon nicht betroffen.

`void myButtons.resetChanged();`

Alle 'geändert' Ereignisse werden gelöscht( bezüglich des Aufrufs von 'changed()', 'pressed()' oder 'released()' ). Die Aufrufe von `longPress()` und `shortPress()` sind davon nicht betroffen.

`uint8_t myButtons.clicked( uint8_t buttonNbr );`

Gibt zurück, ob der Taster einfach oder doppelt geklickt wurde. Der Rückgabewert ist:

<code>NOCLICK (=0)</code>	wenn nichts geklickt wurde.
<code>SINGLECLICK (=1)</code>	wenn einfach geklickt wurde.
<code>DOUBLECLICK (=2)</code>	wenn doppelt geklickt wurde.



Nach dem Aufruf der Methode wird das Ereignis gelöscht, d.h. weitere Aufrufe geben wieder NOCLICK zurück. Das Ereignis wird auch gelöscht, wenn der Taster erneut gedrückt wurde, bevor die Methode aufgerufen wurde.

Zu beachten ist, dass ein Klick auch als ‚shortPress‘ interpretiert wird. Bei einem Klick wird also auch ein ‚shortPress‘ Ereignis ( s.o. ) generiert. In der Regel sollte also entweder auf ‚clicked‘ oder auf ‚shortPress‘ abgefragt werden.

Wird vor dem #include <MobaTools.h> eine Zeile mit #define CLICK\_NO\_SHORT eingefügt, wird auch das ‚shortPress‘ Ereignis gelöscht, wenn beim Aufruf von ‚clicked‘ ein Einfach- oder Doppel-Klick erkannt wurde. ( siehe auch das Beispiel ‚Button\_I2C‘ )

### 3.7 MoToPwm ( nur auf ESP8266 )

Die Klasse MoToPwm enthält Methoden um Töne und PWM-Signale zu erzeugen. Die originalen Funktionen tone() und analogWrite() dürfen nicht genutzt werden

Einrichten:

```
MoToPwm myPwm;
```

Methoden:

```
byte myPwm.attach( byte pinNr );
```

Der Funktionswert ist ,0‘, wenn eine ungültige Pinnr. angegeben wurde, oder der Pin schon genutzt wird. Der Pin wird auf Ausgang geschaltet, aber noch keine Pulse erzeugt.

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

Der Pin wird wieder freigegeben.

```
void myPwm.analogWrite( uint16_t duty1000 );
```

Es wird ein PWM Signal mit dem angegebenen Dutycycle erzeugt. Der Wert wird in Promille angegeben ( 0...1000 ). Nach dem attach hat das Signal eine Frequenz von 1kHz

```
void myPwm.setFreq(uint32_t freq);
```

Ändert die Frequenz des PWM-Signals für folgende Aufrufe von analogWrite.

```
void myPwm.tone(float freq, uint32_t duration );
```

Erzeugt einen Ton mit der angegebenen Frequenz und Dauer. Die Dauer wird in ms angegeben. Ist Dauer 0, so wird ein endloser Ton erzeugt.

```
void myPwm.stop();
```

Stoppt die Signalausgabe ( pwm und Ton ).

```
void myPwm.setPwm( uint32_t high, uint32_t low );
```

Erzeugt ein PWM-Signal mit frei einstellbarer HIGH und LOW Zeit ( in  $\mu$ s ), Mindestzeit für HIGH und LOW ist 40 $\mu$ s.

## 4 Anhang:

Beispielschema des Anschlusses eines Schrittmotors über die SPI-Schnittstelle:

