Siggis Winde

Inhalt

Siggis Winde

Einleitung

Bei der Entwicklung der Windensteuerung haben wir entscheidend vom Windenbauer Peter profitiert und viele Dinge übernommen.

https://www.rc-network.de/threads/hochstartwinde-brushless-regler.714510/

Die Winde besteht ausfolgenden Komponenten:

- Brushless Motor
- LiPo mit 6 Zellen
- Halbbrücke
- MOSFETLeistungsteil
- Umkehrspindel für Schleppseil
- Drehzahl Messung mit 2 Hallsensoren
- Messung Seilspannung mit Federstahl und Poti
- Funk Fernsteuerung über LoRa
- Touch Screen
- Arduino Mega
- Sensoren zur Überwachung von Strom und Spannung
- Mechanik

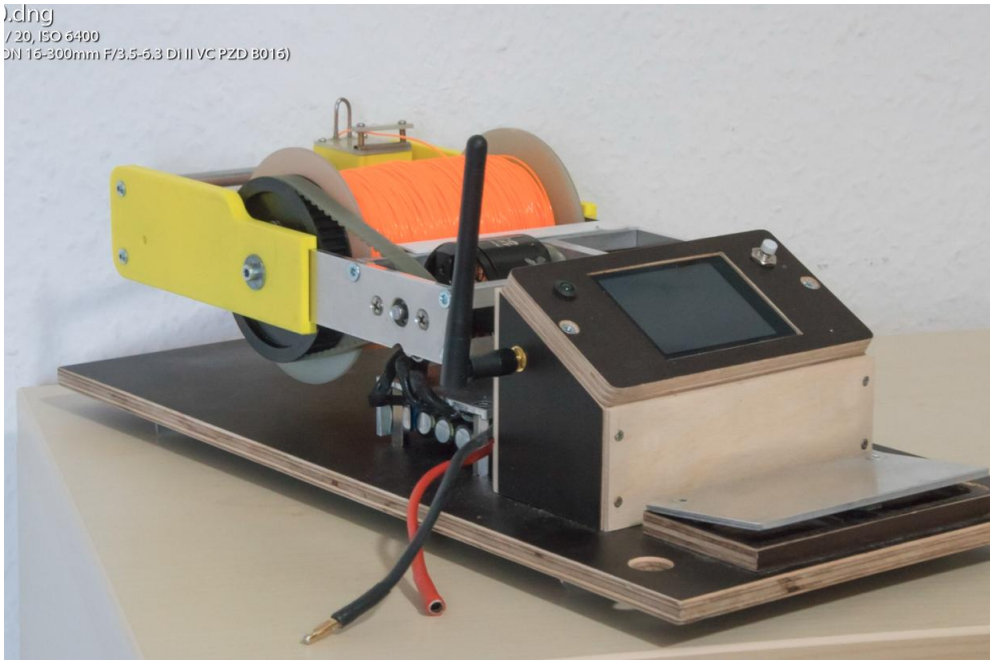


Abbildung 1 Windensteuerung Server

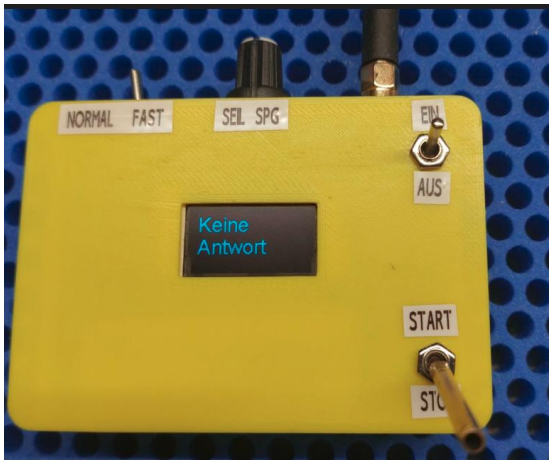


Abbildung 2 Windensteuerung Client

Bedienung

Die Bedienung der Windensteuerung erfolgt über ein Touchscreen Display und über eine Fußtaste. Alternativ kann die Winde über eine Fernbedienung via LoRa gestartet werden.

Windenstart

Die Einstellungen der Seilspannung bzw. der Drehzahl erfolgt über das Touch Screen Display. Die gewünschte Funktion wird über das Hauptmenü ausgewählt.

Die Funktionen

- Start
- Start fast
- Seil holen
- Info
- Konfig

Startmenü

Im Startmenü muß die Seilspannung eingestellt werden. Danach kann der Fußschalter betätigt werden. Zunächst erfolgt eine zweistufige Anlaufphase mit konstantem PWM Signal, danach wird die Seilspannung geregelt.

Abbildung 3 Start Einstellung

Siggis Winde

Start mit Fernbedienung

Die Winde kann alternativ zum Fußschalter mittels Fernbedienung gestartet werden. Die Kommunikation mit der Winde erfolgt über LoRa. Die Seilspannung kann an der Fernbedienung eingestellt werden.

Seil Holen

Im Mode ***Seil holen*** wird nicht die Seilspannung, sondern die Drehzahl geregelt. Die Drehzahl kann im Dialog ausgewählt werden. Der PID Regler regelt dann die eingestellte Drehzahl. Beim ***Seil holen*** wird Drehzahl über eine Sollwertfunktion langsam bis zur Solldrehzahl erhöht.

Abbildung 4 Drehzahl Einstellung

Konfig

Im Konfig Dialog kann der Duty Cycle für beide Anlaufphase Phasen eingestellt werden. Damit kann die Anlaufphase an die Windsituation

angepaßt werden. Siehe : Einfluß der WindgeschwindigkeitEinfluß der Windgeschwindigkeit

Info

Im Info Dialog werden folgende Informationen angezeigt:

- Anzahl Starts
- Verbrauch mAH
- Maximaler Strom
- Letzte Schleppzeit
- Standardabweichung der PID Regelung
- Start- und aktuelle Seillänge

Abbildung 5 Winden Informationen

Diagnose

Falls beim Einschalten der Winde der Fußschalter bereits gedrückt ist, wird das Diagnose Programm aufgerufen.

Mit dem Diagnose Programm können alle Sensoren analysiert werden und der Motor über das Poti an der Seite der Winde gesteuert werden. (PWM IN und SD)

Für die Werte:

Spannung, Strom und Seilspannung wird der analoge Wert und der berechnete Wert angezeigt. Das erleichtert die Kalibrierung.

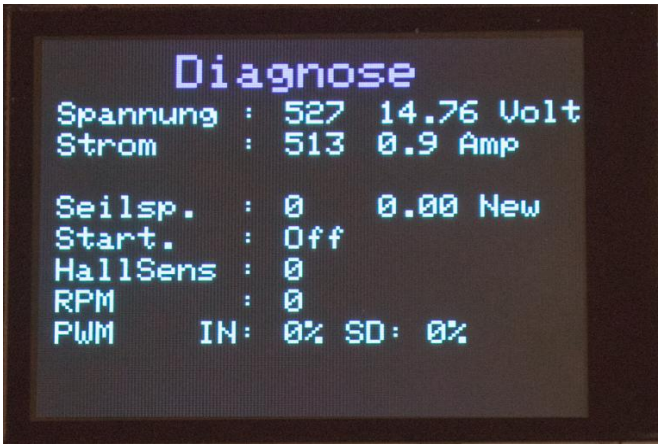


Abbildung 6Diagnose Dialog

Motor Regelung mit Halbbrücke

Die Steuerung des Motors erfolgt über die Halbbrücke mittels PWM Signal. Über das invertierte Shut Down Signal (SD) wird die Regelung gestartet. Wenn das Shut Down Signal HIGH ist kann mit einem PWM Signal am IN Eingang der Halbbrücke der Motor geregelt werden. Der Duty Cycle des PWM Signals bestimmt die Drehzahl des Motors. Duty Cycle von 100% bedeutet Vollgas und bei einem Duty Cycle von 0% steht der Motor. Wenn das SD Signal LOW ist werden beide MOSFETS hochohmig und der Motor befindet sich im Leerlauf.

Für die Windensteuerung wird eine Regelbereich von Vollgas über Bremsen bis zum Leerlauf benötigt. Dies wird erreicht indem das SD Signal ebenfalls

mit einem PWM Signal versorgt wird. Dies hatte ich zunächst falsch interpretiert und wurde nach Infos von Peter korrigiert.

Wenn dem Motor Strom zugeführt wird, beginnt sich Der Motor zu drehen. Eine rotierende Spule in einem Magnetfeld führt jedoch zu einem induzierten Strom. Nach der Lenz'schen Regel muß der induzierte Strom in der Spule eines Motors in die entgegengesetzte Richtung des Stroms fließen, den wir einspeisen. Hierbei entsteht die „Gegen-EMK“ die dem Einspeise Strom entgegenwirkt. Wenn sich der Motor schneller dreht, nimmt die Gegen-EMK des Motors zu, wodurch der Strom im Stromkreis sinkt, bis die Gegen-EMK ungefähr der Spannung entspricht, mit der der Motor versorgt wird. Der Motor dreht sich dann mit konstanter Geschwindigkeit. Wenn der Motor belastet wird, dreht sich der Motor mit einer geringeren Geschwindigkeit da ein Teil des Stroms nun vom Motor verwendet wird, um ein Drehmoment auszuüben.

*Signale der Halbbrücke **IR2104S**:*

- **IN:** Steuersignal (PWM) → bestimmt, ob High- oder Low-MOSFET aktiv ist.
- **SD:** Abschaltsignal → aktiv LOW. Wenn SD = LOW → beide MOSFETs aus.

Der mögliche Regelungsbereich geht von Vollgas über Bremsen bis Leerlauf.

- Vollgas PWM IN = 99 PWM SD = 99
- Bremsen Max. PWM IN = 0 PWM SD = 99
- Leerlauf PWM IN = 0 PWM SD = 0

Regelung der Winde

Bei dem Start der Winde sind drei Phasen zu unterscheiden:

- Anlauf, Start
- Seglerschlepp
- Endphase

Anlauf - Start Phase

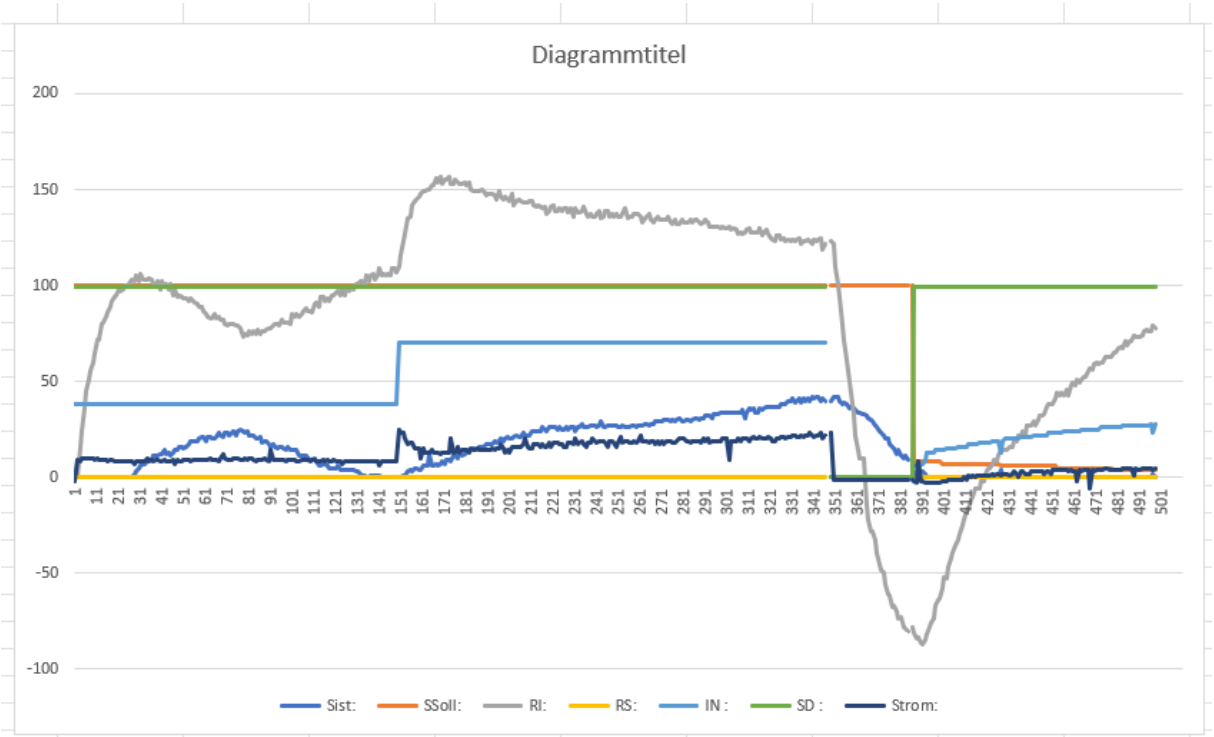
Der Motor ist eine Regelstrecke mit Totzeit. Ein System das mit Totzeit behaftet ist, wird stets verspätet reagieren. Standard PID-Regler versagen oder müssen derart passiv eingestellt werden, daß am Ende von Prozeß-Dynamik keine Rede mehr sein kann. Die Problematik zeigte sich durch eine Überschwingen der Drehzahl beim Start mittels Drehzahl Regelung.

Aus diesem Grund haben wird die Startphase in 2 Stufen mit festen Duty Cycle realisiert. Dabei wurde für Stufe 1 ein Duty Cycle gewählt, bei dem der Motor langsam anläuft und sich das Schleppseil strafft. Die Phase wird nach kurzer Zeit beendet.

Anschließend wird die Stufe 2 gestartet. In dieser Phase wird ein höherer Duty Cycle vorgegeben. Die Phase wird beendet, wenn die Seilspannung ca. 100 % von der Sollseilspannung erreicht hat. Beide Phasen werden sicherheitshalber über einen Timer überwacht, der die jeweilige Phase nach einer einstellbaren Zeit beendet. Nach Beendigung der Start Phase 2 erfolgt die Regelung der Seilspannung.

Einfluß der Windgeschwindigkeit

Die Startphase mit 2 stufigem PWM Signal (Duty Cycle) hat bei normalem und starkem Wind gut funktioniert. Bei Starts an einem windstillen Tage wurde die Soll Seilspannung nicht erreicht und der Start abgebrochen.



Das Verhalten ist beim Windenstart mit konstantem PWM-Signal hängt stark vom Wind und der Fluggeschwindigkeit relativ zur Luft ab. Die Seilspannung entsteht nicht direkt durch die Winde, sondern primär durch den aerodynamischen Widerstand des Flugzeugs, das am Seil hängt.

Der aerodynamische Widerstand steigt ungefähr mit dem Quadrat der Geschwindigkeit der anströmenden Luft.

Warum es bei mittlerem/starkem Wind funktioniert

Bei Gegenwind passiert Folgendes:

- Das Modell hat mehr Luftgeschwindigkeit, obwohl die Bodengeschwindigkeit gleich bleibt.
- Dadurch steigt der aerodynamische Widerstand stark an.
- Das Seil wird stärker belastet → Seilspannung erreicht den gewünschten Wert.

Warum es bei schwachem Wind nicht reicht

Bei fast keinem Wind:

- Luftgeschwindigkeit ≈ Bodengeschwindigkeit

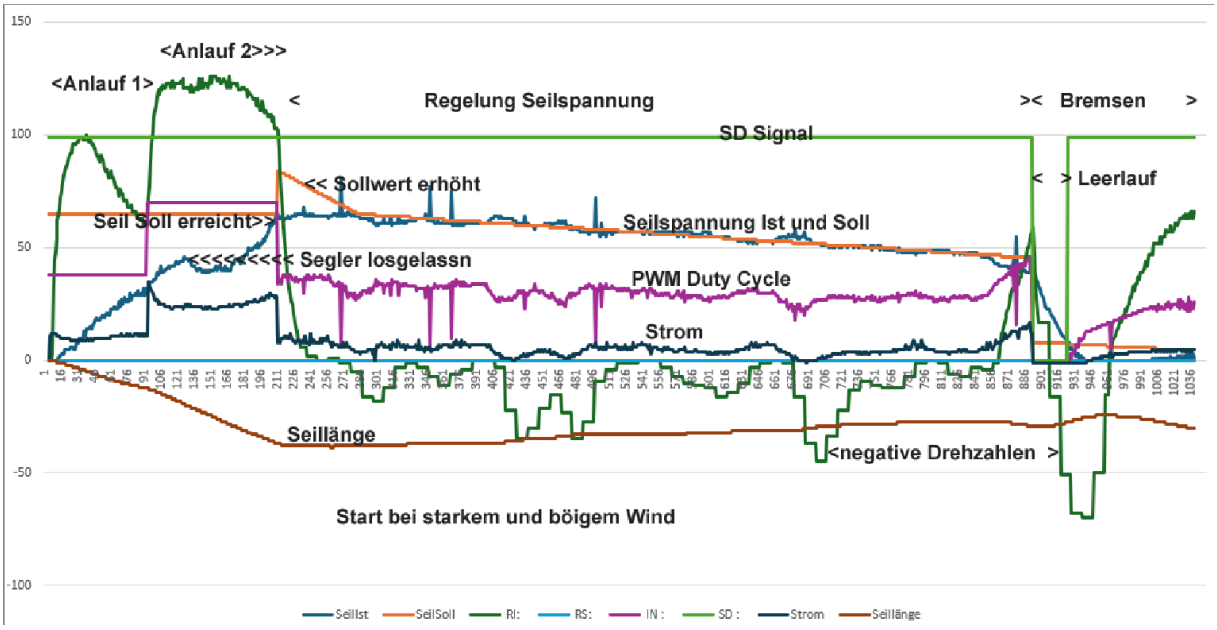
- Widerstand ist deutlich geringer
- Das Flugzeug „zieht“ weniger am Seil und die Seilspannung erreicht nicht den eingestellten Wert

Konsequenz ganz einfach. Sollseilspannung reduzieren:

Wechsel von Anlaufphase zur Schlepppphase

Der Wechsel von der Anlaufphase zur Schlepppphase war zunächst unbefriedigend. Wenn die Soll Seilspannung erreicht ist, wird auf die Regelung Seilspannung umgeschaltet. Das Schleppseil steht unter hoher Spannung und entspannt sich schnell. Die Drehzahl fällt stark ab. Die Seilspannungsregelung startet jedoch mit einem geringen Unterschied zwischen Soll und Ist. Dadurch fällt die Seilspannung zunächst weiter ab, bis die Regelung greift. Aus diesem Grund haben wir den Übergang von der Startphase zur Regelungsphase noch einmal geändert.

Nach Beendigung der Anlaufphase beginnt die Regelung der Seilspannung mit einem um ca. 30% erhöhten Sollwert, der in 1,5 Sekunden auf den Sollwert abfällt. Damit ergibt sich zu Beginn der Regelungsphase eine deutliche Differenz zwischen Soll und Istwert. Damit kann der PID Regler das Absinken der Seilspannung verhindern. Das Flugbild ist jetzt in Ordnung.



Die Sollwert Verlaufs Kurven sind in einem Array definiert . Wobei die erste Spalte die Zeit in Millisekunden definiert und in der zweiten Spalte der Faktor mit dem der Sollwert multipliziert wird.

```
#define AnzValuesRunNormal 5
  struct SollwertFunktion StartNormal[AnzValuesRunNormal] = {
    { 0, 0.0 },
    { 1, 1.3 },       // in 1 msec auf 130 % Seil Soll
    { 1500, 1.0 },    // nach 1501 msec reduziert auf 100%
    { 14000, 0.70 },  // in 15001 msec 70% reduzieren
    { 99999, 0.0 },   //99999 bedeutet Ende Kennung
  };
```

Seglerschlepp Phase

In der eigentlichen Schleppphase wird die Soll Seilspannung über eine einstellbare Sollwert Verlaufs Kurve geregelt. Im *Start Mode* fällte die Seilspannung nach einer einstellbaren Zeit langsam wieder ab. Im *Start fast Mode* bleibt die Seilspannung nach Erreichen des Sollwerts konstant.

Wenn der Segler ausklinkt sinkt die Seilspannung schlagartig. Das passiert auch bei einem Seilriß. Deshalb wird die Seilspannung kontinuierlich überwacht.

Die Überwachung der Seilspannung erfolgt allerdings erst 5 Sekunden nach dem Start um ein vorzeitiges Auslösen des Bremsvorganges zu vermeiden. Außerdem muß die Seilspannung 10 Mal hintereinander den Mindestwert unterschreiten (ca. 200 msec) bevor der Bremsvorgang eingeleitet wird. Falls die Bedingungen erfüllt sind wird die Seglerschlepp Phase beendet und die Bremsphase gestartet. Wenn der Fußschalter gelöst wird erfolgt ebenfalls der Aufruf der Bremsphase.

End Phase, Bremsen

Nach dem Ausklinken ist im Schleppseil noch eine hohe Energie gespeichert. Wenn man den Motor einfach abschaltet (Leerlauf) wird durch die Seilspannung (wie bei einem Gummiseil) das Schleppseil abgewickelt, d.h. die Winde dreht rückwärts. Dadurch baut sich in der Winde durch die Drehzahl Energie auf. Wenn die Seilspannung abgebaut ist wird die Winde durch die gespeicherte Energie in der Winde weiter rückwärts drehen und es entsteht Seilsalat. Hier hatten wir zunächst nach Unterschreitung einer Seilspannung von 8 Newton eine Seilspannungregelung realisiert.

Wenn der Wind den Fallschirm entlastet oder belastet, dann sinkt die Seilspannung mal schneller, mal langsamer. Eine reine

Seilspannungsregelung versucht dann, die „zu kleine“ Spannung wieder anzuheben — und das führt dazu, dass die **Drehzahl unnötig ansteigt**.

Am Ende darf die Seilspannung nicht mehr die führende Regelgröße sein, weil sie dann stark vom Bremsfallschirm und Wind abhängt und nicht mehr zuverlässig den Bewegungszustand der Trommel beschreibt.

Aus diesem Grund muss das Bremsen in zwei Phasen erfolgen.

- Hohe Seilspannung => Leerlaufphase
- **Auslaufphase => Drehzahl regeln**

Warum das besser ist

Am Ende bestimmt nicht mehr das gespannte Seil das System, sondern eher:

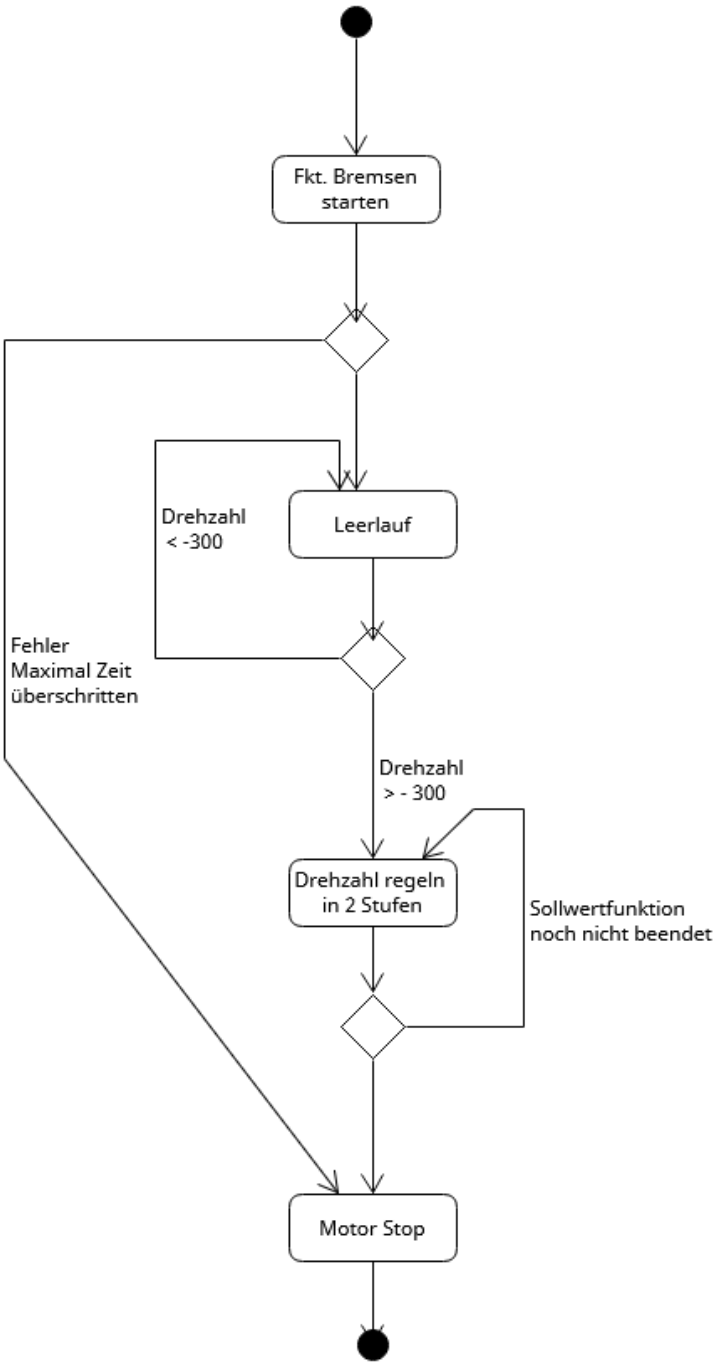
- Luftwiderstand des Fallschirms
- Windböen
- Trommelträgheit
- Reibung

Die Seilspannung wird dann zu einer **Störgröße**.
Die Drehzahl ist in dieser Phase die viel bessere Führungsgröße, die folgendes verhindern soll:

- unnötiges Beschleunigen
- starkes Nachlaufen
- instabiles Verhalten bei Wind

Realisierung Bremsen

Nach Beendigung der Seglerschlepp Phase wird der Motor auf Leerlauf geschaltet (PWM IN = 0 und PWM SD = 0). Die Drehzahl wird negativ, das Seil entspannt sich. Sobald die Drehzahl sich Null nähert (z.B. -300 RPM) wird die die Drehzahlregelung gestartet. Die Regelung der Drehzahl erfolgt über eine Sollwertfunktion. Um einen harten Übergang zu vermeiden, wird der Sollwert zunächst von Null auf die *Brems Solldrehzahl* geregelt und wird danach langsam bis auf eine *Restdrehzahl* herunter geregelt.



Bremsen mit Seil einholen

Nicht mehr aktiviert

Eine weitere Variante zur Beendigung des Schleppvorgangs ist das *Bremsen mit Seil einholen*.

Mit den zwei Hallsensoren wird kontinuierlich die Seillänge ermittelt. In dem Konfiguration Dialog kann die Länge zum Einholen des Schleppseils eingestellt werden. Falls ein Wert größer Null eingestellt ist, erfolgt das *Bremsen mit Seil einholen*.

Zunächst erfolgt ebenfalls die Leerlaufphase. Danach wird das Seil um den eingestellten Betrag eingeholt. Dabei wird das Einholen des Seils mittels Seilspannung geregelt. Beide Phasen werden über einen Timer überwacht. Für den Fall, das sich das Seil (der Schirm) verhakt, wird bei überschreiten einer bestimmten Seilspannung das Bremsen beendet.

Seil einholen

Im Mode ***Seil holen*** wird nicht die Seilspannung, sondern die Drehzahl geregelt. Die Drehzahl kann im Dialog ausgewählt werden. Der PID Regler regelt dann die eingestellte Drehzahl. Beim ***Seil holen*** wird Drehzahl über eine Sollwertfunktion langsam bis zur Solldrehzahl erhöht.

In einem Test wurde beim Einholen des Seils die Seilspannung sehr hoch (Kartoffeln). Dann hat Siggi die Fußtaste gelöst, dadurch wurden die MosFETS hochohmig, der Motor war dadurch im Freilauf. Die Seilspannung baute sich schlagartig ab, indem das Seil abgewickelt wurde. Durch die hohe Drehzahl trat eine hohe Gegen EMK auf, die leider die Elektronik zerstört hat.
Bei der Funktion *Seil einholen* wird jetzt die Seilspannung überwacht und bei zu hoher Seilspannung der Motor abgeschaltet.

Start mit LoRa

Die Winde kann alternativ zum Fußschalter mittels Fernbedienung gestartet werden. Die Kommunikation mit der Winde erfolgt über LoRa. Die Seilspannung kann an der Fernbedienung eingestellt werden.

Abbildung 7 Fernbedienung via LoRa

Die Winde erkennt automatisch die Fernbedienung, sobald diese eingeschaltet wird und schaltet in den LoRa Mode um. Auf dem Display des Senders wird der RSSI angezeigt. Signalstärke oder RSSI (Received Signal Strength Indicator) zeigt die Signalqualität zwischen dem Sender und dem Empfänger an. Je näher der RSSI-Wert bei 0 liegt, desto besser ist das Signal. Ein RSSI-Wert von -50 dBm beispielsweise ist ein gutes Signal, während ein RSSI-Wert von -80 dBm ein schlechtes Signal anzeigt.

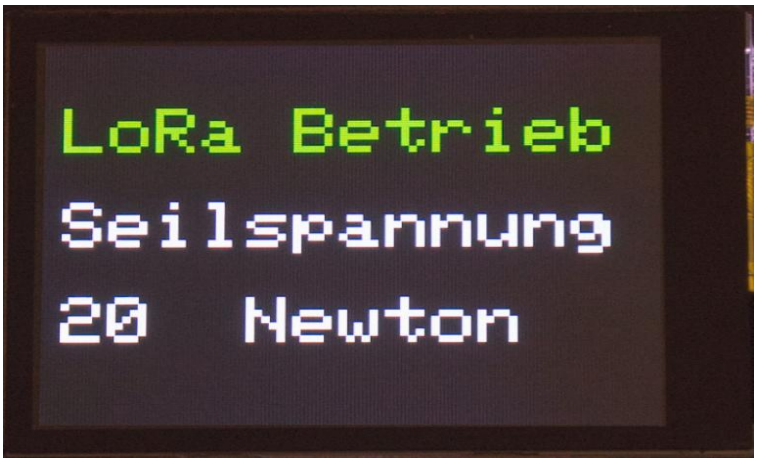


Abbildung 8 Anzeige LoRa Betrieb

Falls im laufenden Betrieb ein Verbindungsfehler auftritt, wird dies an der Winde und an der Fernbedienung angezeigt. Die Winde wird dann sofort gestoppt.

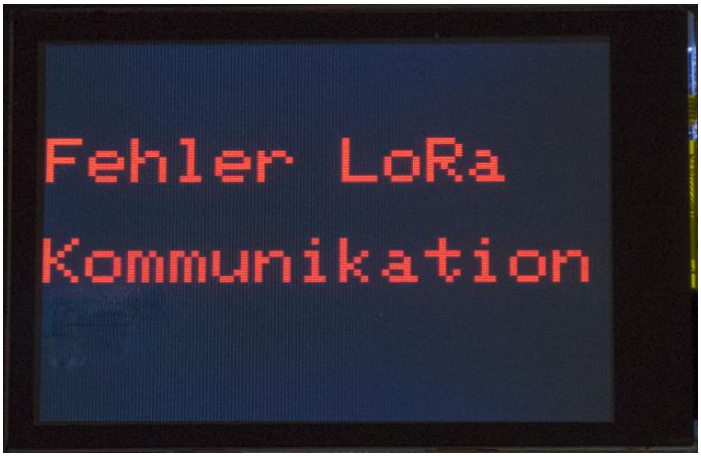


Abbildung 9 Fehler LoRa Kommunikation

Kritische Sytemzustände

Bei dem Betrieb der Winde können kritsche Systemzustände auftreten die zur Zerstörung der Elektronik oder Mechanik führen können.

- Motor Stillstand
- Wiederanlauf nach Lehrlauf
- Leerlauf in der Schlepppphase
- Strom zu hoch
- Seilspannung zu niedrig, Seilriss
- Blockade des des Schleppseils bei *Seil Holen*
- Verbindungsabbruch LoRa

Einige Zustände haben wir life mit entsprechenden Folgen erlebt.

Motor Stillstand

Im normalen Betrieb verteilt sich die Verlustleistung zeitlich auf alle sechs MOSFETs der Brücke, da die Kommutierung zyklisch zwischen den Phasen wechselt.

Bei sehr niedriger Drehzahl oder Motorstillstand wird der Strom hingegen überwiegend über ein einzelnes Phasenpaar (je ein High-Side- und ein Low-Side-MOSFET) geführt. Dadurch konzentriert sich die Verlustleistung auf wenige Bauteile.

Da bei Stillstand keine Gegen-EMK vorhanden ist, kann der Strom stark ansteigen. Die daraus resultierenden Leitverluste

$$(P = I^2 \cdot R_{DS(on)})$$

führen zu einer schnellen Erwärmung der betroffenen MOSFETs.

Mit steigender Temperatur erhöht sich der Einschaltwiderstand Rds(on) wodurch die Verlustleistung weiter zunimmt. Dieser positive Rückkopplungseffekt kann innerhalb sehr kurzer Zeit zu einer thermischen Überlastung und letztlich zur Zerstörung der MOSFETs führen.

Am robustesten ist die Fehlerbehandlung in diese Reihenfolge:

1. **Schnelle Strombegrenzung** als erste Schutzebene.
2. **Drehzahl-/Hall-basierte Stall-Erkennung** als zweite Ebene.
3. **Gestufte Reaktion**: PWM begrenzen, Integrator einfrieren, dann erst abschalten.

Denn nur die Strombegrenzung reagiert schnell genug auf den eigentlichen elektrischen Peak; die Hall-Auswertung erkennt eher den mechanischen Stillstand bzw. das Ausbleiben der Bewegung. (Funktion im Main Script, suche: MotorLastStillstTime)

Schnelle Strombegrenzung

Falls der Strom über einen einstellbaren Grenzwert steigt wurde der Motor abgeschaltet. Dies ist öfters passiert. Wir konnten zunächst dafür kein Grund erkennen. Deswegen haben wir den Algorithmus geändert.

Der Strom wird jetzt periodisch alle 2,5 Msec kontrolliert. Wenn der Strom zu hoch ist (z.Z. 85 Ampere), wird der aktuelle Duty Cycle alle 2.5Msec halbiert, bis der Strom wieder in einem zulässigen Bereich ist. Dabei muss der **I-Anteil des PID Reglers gelöscht werden**. Die normale PID Regelung läuft alle 10 Msec weiter. Falls der Strom nach mehrmaligen Korrekturen nicht in einen normalen Bereich sinkt sollte der Motor ausgeschaltet werden. Dies ist noch nicht realisiert.(Rolf ToDo).

Im folgenden Beispiel ist die Seilspannung von 71 auf 50 Newoton abgefallen. PWN-IN ist dann von 39 auf 75 gestiegen. Danach (Logg alle 20Msec) weiter gestiegen. Stomprüfung ermittel 96,95 Ampere, PWM wird von 75 auf 36 herunter geregelt.Der PID Regler übernimmt wieder bei 74 Newton mit PWM-In von 34.

	A	B	C	D	E	F	G	H
1	Motor start	Sist:	SSoll:	RI:	RS:	IN:	SD:	Strom:
333		71	74	5	0	39	99	9
334		70	74	5	0	40	99	10
335		71	74	5	0	39	99	11
336		50	74	5	0	75	99	8
337	Strom 96.95 PWM 75 PWM neu 36							
338		74	74	21	0	34	99	9
339		67	74	13	0	45	99	10
340		73	74	14	0	36	99	10
341		71	74	13	0	38	99	9

Hall-basierte Stall-Erkennung

Ein Motorstillstand kann erkannt werden, wenn die Interrupts der Hall Sensoren zu lange ausbleiben. Bei einer Drehzahl von 1000 RPM erfolgen beispielsweise 2.333 Interrupts pro 10 msec.

Drehzahl RPM	Anzahl Magnete	Anzahl Interrupts pro Umdrehung	Umdrehungen pro Sekunde	Umdrehungen pro 10 MSEC	Interrupt pro 10 MSEC
1,0000	14,0000	14,0000	0,0167	0,0002	0,0023
100,0000	14,0000	14,0000	1,6667	0,0167	0,2333
1000,0000	14,0000	14,0000	16,6667	0,1667	2,3333
10000,0000	14,0000	14,0000	166,6667	1,6667	23,3333

Dazu wird die Motor Stillstandszeit ermittelt. Die Stillstandszeit ist die Zeit zwischen zwei Interrupts. Es wird bei jedem Interrupt die aktuelle Zeit in msec gespeichert. Wenn keine Interrupts mehr erfolgen wird ca. alle 3,5 msec die *MotorLastStillstTime* addiert. Wenn wieder ein Interrupt erfolgt wird die *MotorLastStillstTime gelöscht.* Über einen Timer gesteuert, wird alle 3,5 msec geprüft, ob die aktuelle *MotorLastStillstTime* einen Schwellwert (z.Z. 100 Msec???) überschreitet. Wenn das der Fall ist, werden die PWM Signale wie folgt reduziert.

Wenn PWM IN > 0 ist PWM IN = PWM IN * 0,9
Wenn PWM SD < 99 ist PWM SD = PWM SD * 0,9

Im folgenden Beispiel war die *MotorLastStillstTime* überschritten. Das PWM-IN Signale wird mehrfach reduziert, nach dem Motto „immer in Bwegung bleiben“.

	A	B	C	D	E	F	G	H
1	Motor start	Sist:	SSoll:	RI:	RS:	IN:	SD:	Strom:
303		73	75	1	0	34	99	8
304		72	75	0	0	35	99	8
305		73	75	0	0	34	99	10
306		72	75	0	0	35	99	8
307	PWM_IN 33.25 PWM_SD 99.00							
308	PWM_IN 31.59 PWM_SD 99.00							
309	PWM_IN 30.01 PWM_SD 99.00							
310		84	75	0	0	15	99	8
311	PWM_IN 14.25 PWM_SD 99.00							
312		71	75	0	0	37	99	7
313		72	75	-1	0	35	99	8
314		72	75	-1	0	36	99	7

Abbildung 10 Motor Stillstand

Bei einer Drehzahl vom 1000 RPM bedeuten 100 Msec ohne Interrupt, das über 20 Interrupts fehlen müssen bevor die Regelung eingreift.

Der mögliche Regelungsbereich geht von Vollgas über Bremsen bis Leerlauf.
Vollgas PWM IN = 99 PWM SD = 99
Bremsen Max. PWM IN = 0 PWM SD = 99
Leerlauf PWM IN = 0 PWM SD = 0

Nach der temporären Änderung der PWM Signale erfolgt wieder die normale Regelung der Seilspannung.

~~Vorher wird der PID Regler zurückgesetzt, damit der aufgelaufen I-Anteil keine Rolle mehr spielt(Rolf ToDo). Dann erfolgt die Regelung nur wie folgt.~~

$$u = K_p e + K_d \dot{e}$$

~~Sonst drückt der PID im Hintergrund weiter hoch, und beim nächsten gültigen Hall-Zustand kommt wird die Stellgröße zu groß.~~

In der kurzen Zeit ist nichts aufgelaufen????

Ein kritischer Punkt ist der Übergang von **„manuell begrenzter PWM“** zurück zu **„PID bestimmt PWM“**. Das ist ein klassischer **Bumpless-Transfer-Fehler**.

Zustand REENGAGE: Der Motor läuft wieder, aber die normale PID-Regelung darf nicht sofort hart übernehmen. Dann erzeugt der PID beim ersten Zyklus einen Sprung.

Einfachste Minimalversion beim ersten Zyklus nach PWM Korrektur

- PWM *= 0.9 vorerst beibehalten
- PID **nicht auf 0 resetten**
- stattdessen beim Wiederfreigeben setzen:

$$I = PWM_{aktuell} - K_p e$$

Das ist die kleinste Änderung mit großem Nutzen. (Rolf ToDo ????)

Falls die Stillstandszeit einen Maximalwert überschreitet wird die Bremsphase des Motors eingeleitet.(suche: void Motor::CheckStrom)

Wiederanlauf nach Leerlauf

In der **Leerlaufphase** (Freilaufphase), wenn die MOSFETs hochohmig sind, wirkt der BLDC-Motor wie ein **Generator**. Er erzeugt eine Gegen-EMK (Back-EMF), die von der Drehzahl abhängt.

$$E = k_e \cdot \omega$$

Bei 760 kV Motor ist die Gegen-EMK bei 20.000 rpm ≈ **26 V**

Die Gegen-EMK liegt weiterhin an den Motorwicklungen an, Sie ist nur von der Drehzahl abhängt. Die Phasen „schweben“ elektrisch (keine feste Verbindung zu + oder −). Im **Idealfall kein Stromfluss** (I ≈ 0), weil der Stromkreis offen ist

In der Realität existieren kleine parasitäre Effekte bedingt durch:

- Kapazitäten im System
- evtl. Body-Dioden (aber nur bei bestimmten Spannungszuständen)

Damit gilt für die Leerlaufphase:

- Nahezu kein Phasenstrom
- Motor läuft frei weiter (nur mechanische Verluste bremsen)
- Keine elektrische Energie wird aufgenommen oder abgegeben
- Die mechanische Energie bleibt im System (Rotation)

Spannungsspitzen wenn die MOSFETs abschalten. Die Induktivität der Wicklungen will den Strom weiter treiben => kurzzeitig können Spannungsspitzen entstehen. Diese werden meist begrenzt durch Body-Dioden der MOSFETs oder parasitäre Pfade begrenzt

Wenn der Motor nach Leerlauf wieder eingeschaltet wird, kann die vorhandene Gegen-EMK **Stromsprünge verursachen** und je nach Phase sogar gegen die Ansteuerung arbeiten. Beim Einschalten „trifft“ man also auf ein bereits existierendes Spannungssystem. Der Strom hängt im Moment des Einschaltens von der aktuellen Phasenlage ab. :

$$I \sim \frac{U_{\text{ESC}} - E_{\text{Back-EMF}}}{Z_{\text{Phase}}}$$

Aus diesem Grund sollte der Motor mit geringem PWM Signal **bei der Funktion *Bremsen*** *über* eine Rampe wieder hochgefahren werden.

(Suche bool Motor::MotorBremsen4(.....)

Synchronous Rectification (aktiver Freilauf)
Statt den Motorstrom frei auslaufen zu lassen, werden die MOSFETs beim **aktiven Freilauf** gezielt ein geschaltet, damit der Strom weiterfließen kann. Damit kann man harte Stromsprünge vermeiden. Allerdings ist das Timing kompliziert zu realisieren. Die Variante haben wir **nicht realisiert**.

Leerlauf in der Schlepppphase

Die Situation bei Leerlauf in der Schlepppphase:

- Seilspannung sinkt → PID regelt runter
- PWM kann sehr klein werden oder ~0
- Motor wird praktisch **entkoppelt (Leerlauf / Coast)**
- Trommel + Seil + Modell halten den Motor weiter in Drehung
- Strom im Motor: **≈ 0**
- Motor verhält sich wie **Generator** → **Gegen-EMK bleibt hoch**

Das Problem ist die Wiedereinkoppeln in einen bereits drehenden Motor ohne vorhandenen Stromfluss. Wenn die Phase nicht perfekt passt ist das

Ergebnis eine harte Wiedereinkopplung, eine **Stromspitze** und ggf. Ruck im Seil.

Aus diesem Grund sollte die PWM nicht auf Null sinken, es muss ein kleines Mindestmoment gehalten werden.

```
if (running) {
    if (u_pid > 0 && u_pid < u_min)
        u_target = u_min;
    else
        u_target = u_pid;
} else {
    u_target = u_pid;
}
```

Abbildung 11Pseudocode für Korrektur PWM

Häufig ist der I-Anteil oft der Grund, der den Wiedereinstieg unnötig hart macht. Der I-Anteil summiert den Regelfehler auf. Der Regler hat intern noch „Zugbedarf“ wenn die normale Regelung wieder übernimmt. Dann kann der Ausgang sofort wieder nach oben springen.

Eine Möglichkeit ist, vor dem Starten der Regelung den I-Anteil so setzen, dass der Reglerausgang zum aktuellen PWM passt.

$$I = u_{aktuell} - K_p e$$

Alternativ, sobald der kritische unteren Bereich beginnt:

- Integrator einfrieren

Sobald normale Regelung wieder übernimmt:

- Integrator auf aktuellen PWM-Ausgang setzen

Das ist die kleinste saubere Änderung.

(ToDo Rolf????)

Blockade des des Schleppseils bei *Seil Holen*

Fehler Analyse Winden Start am 29.09.2025:
Bei der Funktion *Seil holen* ist die Verspannung der Umlenkrolle gerissen und das Schleppseil hat sich kurzfristig verhakt.

Bei einer Blockade des Schleppseils will der PID Regler die Seilspannung auf Sollwert regeln. Wenn jedoch das Seil blockiert ist, wird sich nach kurzer Zeit ein Gleichgewicht einstellen. Die Ist Seilspannung entspricht der Soll Seilspannung und der Motor bleibt stehen.

Auf der Zeitlupe https://youtu.be/c2SZCxQmwHc kann man erkennen, dass der Motor ca. 500 msec stehen bleibt.

Im Fall einer kompletten Blockade des Schleppseils sind die o.g. Änderungen des PWM Signals nicht ausreichend, der Regler raucht ab.

Eine volle Blockade des Schleppseils ist keine normaler Betriebszustand.

Die Funktion Seil holen ist mit einem PID Regler über die Drehzahl gesteuert. Jetzt wird zusätzlich die Seilspannung geprüft. Wenn die Seilspannung einen Maximalwert überschreitet wird das Bremsen eingeleitet. Dabe wird duch die Leerlaufphase das Seil sofort entspannt.

(suche: bool Motor::RunRewind(float SollRPM, float IstRPM, float SeilIST)

Verbindungsabbruch LoRa

Die LoRa Verbindung wird ständig mit Status Abfragen überwacht. Falls die Statusabfragen ausbleiben wird die Winde abgeschaltet .(suche: SiggisMotor.LoRaAbbruch)

Seilspannung zu niedrig

Die Überwachung der Seilspannung erfolgt erst 5 Sekunden nach dem Start um ein vorzeitiges Auslösen des Bremsvorganges zu vermeiden. Den Fall hatten wir auch schon einmal bei einem Schlepp von einem Kollegen mit einem leichten Segler. Außerdem muß die Seilspannung 10 Mal hintereinander den Mindestwert unterschreiten (ca. 200 msec) bevor der Bremsvorgang eingeleitet wird. (suche: Motor::SeilspLow(float seilspannung)

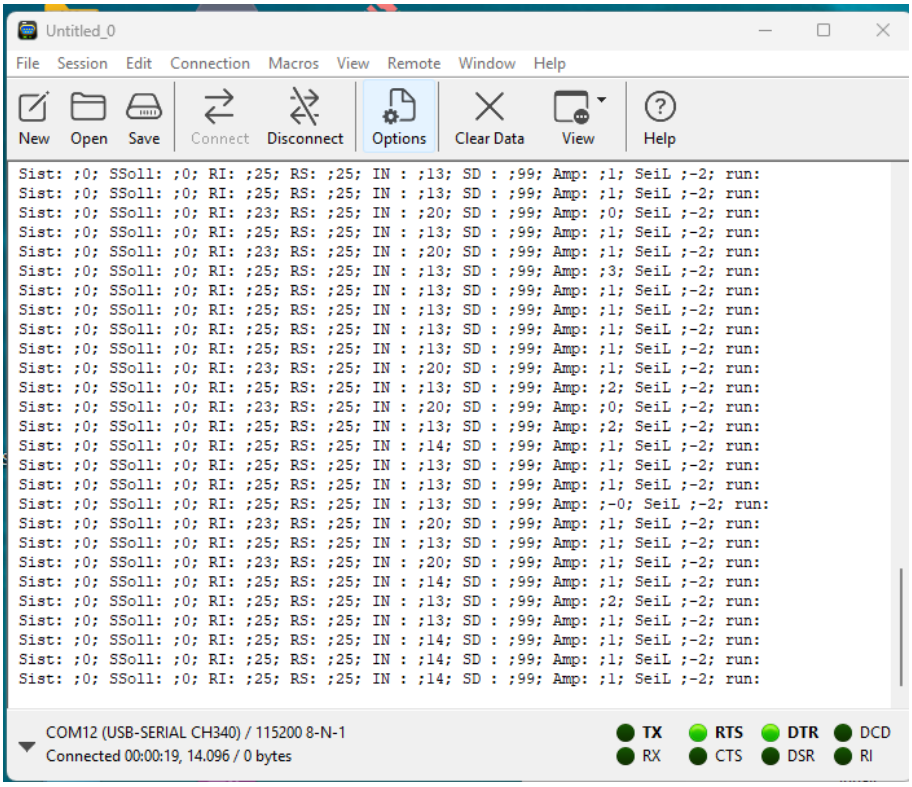
Logging

Logging mit *SimpleTerm - Advanced Serial Port Monitor*

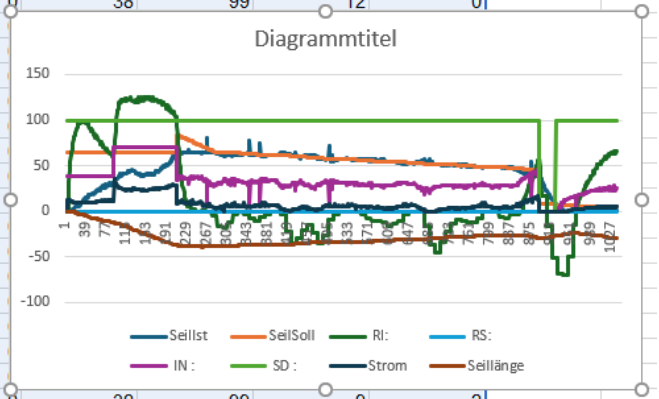
Die Logging Daten können mit dem seriellen Monitor mit Programm *SimpleTerm - Advanced Serial Port Monitor* ausgewertet werden.
(https://ptronix.com/)

Damit ist eine sofortige Auswertung der Daten auf dem Notebook möglich. Das Logging auf SD Karte nutzen wir nicht mehr.

Die Werte der gespeicherten csv Datei können mit Excel mir einem VBA Makro sofort graphisch dargestellt werden.



	Seillst	SeilSoll	RI:	RS:	IN :	SD :	Strom	Seillänge
LoRa Start								
X Read custom object from EEPROM:	WindenNr 2	Version: 153	Datum: 14.04.2025					
Motor start								
X Start mit SeilSoll = 65.00 Newt.								
	0	65	0	0	38	99	0	0
	0	65	0	0	38	99	11	0
	0	65	0	0	38	99	12	0
	0	65	3	0	38	99	12	0
	0	65	33	0	38	99	12	0
	0	65	45					
	0	65	48					
	0	65	54					
	0	65	62					
	1	65	64					
	2	65	67					
	2	65	72					
	3	65	75					
	3	65	79					
	3	65	82					
	5	65	84					
	5	65	86					
	5	65	87					
	5	65	89					
	5	65	91					
	6	65	92	0	38	99	9	-2
	6	65	92	0	38	99	9	-2
	6	65	96	0	38	99	9	-3
	5	65	96	0	38	99	9	-3
	5	65	96	0	38	99	9	-3
	7	65	96	0	38	99	9	-3

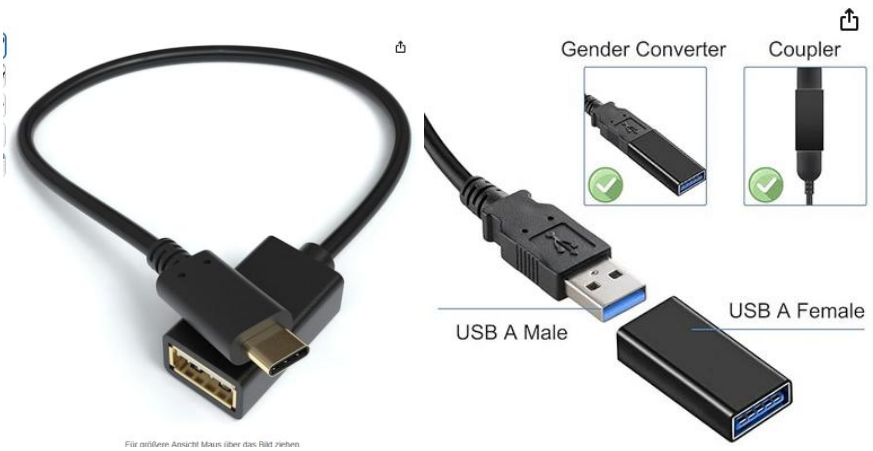


Logging mit Android Handy

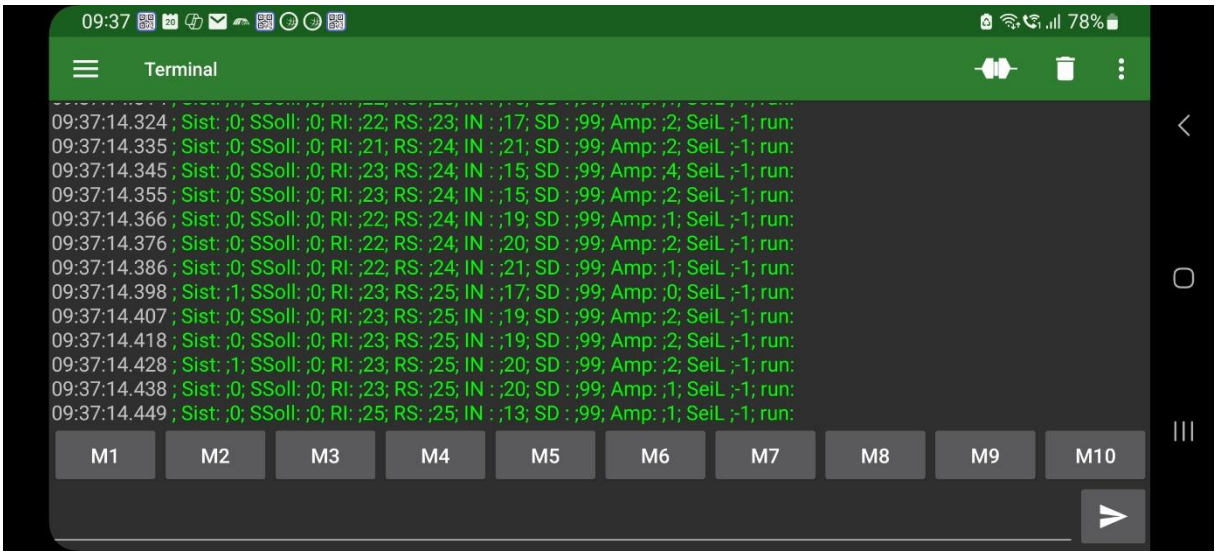
Da man nicht immer einem Notebook zur Verfügung hat, gibt es die Alternative die Daten mit dem Android Handy zu loggen. Die App kann im Play Store geladen werden.

https://play.google.com/store/apps/details?id=de.kai_morich.serial_usb_terminal&hl=de

Die App kann erfreulicherweise auch mit 115200 Baud übertragen. Zum Anschluß benötigt man einen USB Host Adapter und einen Gender Changer.

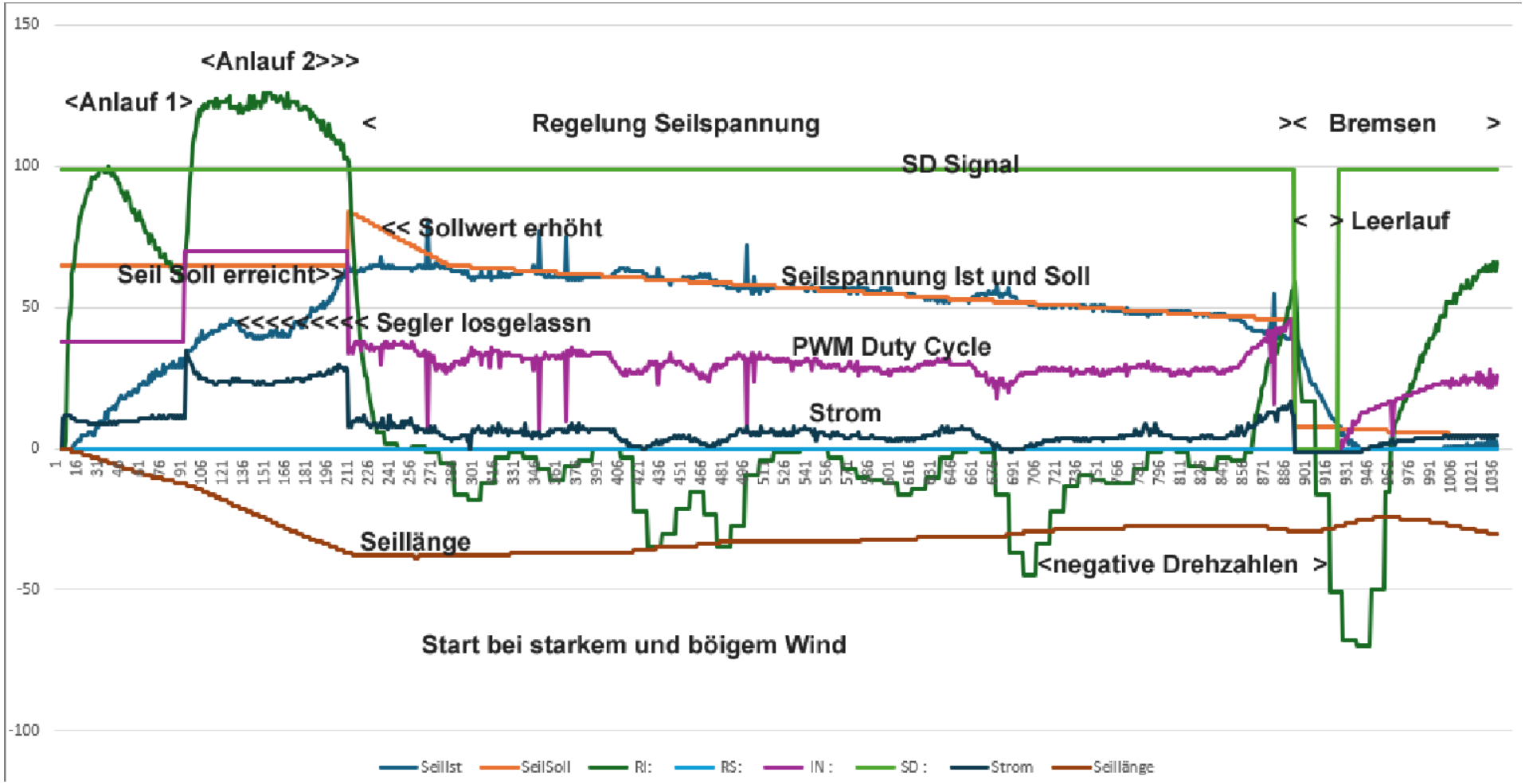


JAMEGA - USB auf USB-C OTG Adapter-Kabel und Xiatiaosann USB 3.0 Kupplung Adapter, USB A Buchse auf Stecker Geschlechtskonverter



Die Daten können auf dem Handy gespeichert werden und zur Auswertung später an ein Notebook gesendet werden.

Auswertung der Logs



Siggis Winde

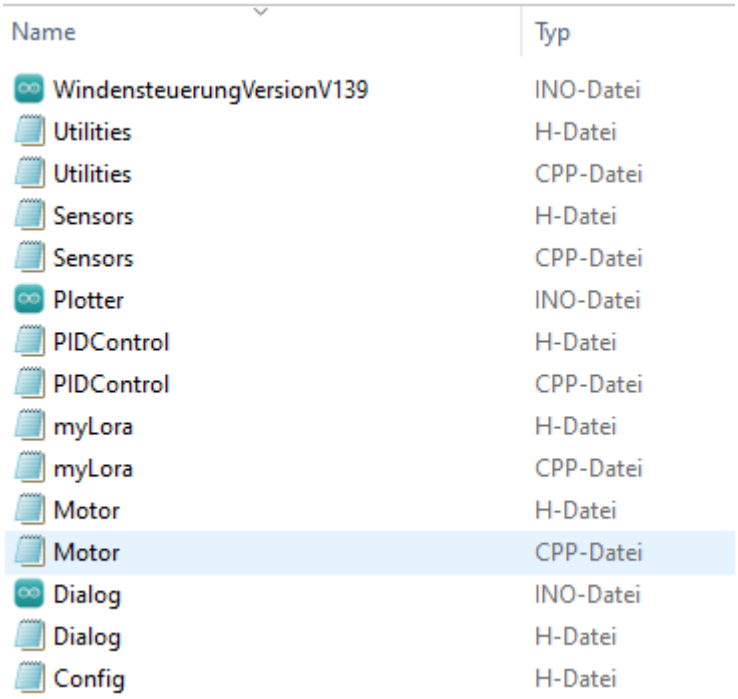
Die graphische Darstellung in Excel erfolgt automatisch mit einem Makro. Damit man bei den Auswertungen nicht den Überblick verliert, werden die folgenden Einstellungen und Ergebnisse der Winde am Ende des Logs protokolliert.

Seilsp. Kp = 1.70 Ki = 1.80 Kd = 0.00
Drehzahl Kp = 1.00 Ki = 1.75 Kd = 0.00
PWMIN Stufe 1 = 38 PWMIN. Stufe 2 = 70
Seilsp. Ende 1 = 0.50 Seilsp. Ende 2 = 1.00
Count 683 **Standardabweichung 4.52**
Max. Strom 39.28
Seilaenge aktuell-30.34
Einzugslaenge 0.00
SeilLaengeBeginnBremsen-28.84

Software

Aufbau des Arduino Sketch

Der Sketch Windensteuerung besteht aus den folgenden Includes.



Name	Typ
WindensteuerungVersionV139	INO-Datei
Utilities	H-Datei
Utilities	CPP-Datei
Sensors	H-Datei
Sensors	CPP-Datei
Plotter	INO-Datei
PIDControl	H-Datei
PIDControl	CPP-Datei
myLora	H-Datei
myLora	CPP-Datei
Motor	H-Datei
Motor	CPP-Datei
Dialog	INO-Datei
Dialog	H-Datei
Config	H-Datei

Liste der öffentlichen Includes :

- #include <Adafruit_GFX.h> // the core graphics library for all our displays
- #include <Adafruit_ILI9341.h>
- #include <Adafruit_FT6206.h>
- #include <SPI.h> // This library allows you to communicate with SPI devices
- #include <MegunoLink.h> // https://www.megunolink.com/
- #include "AVR_PWM.h" // Built by Khoi Hoang https: //github.com/khoih-prog/AVR_PWM
- #include <SD.h>
- #include "Arduino.h"
-

Die Software ist Objekt orientiert realisiert. Die Aufgaben wie Motor, PID Regler usw. sind in eigenen Libraries / Klassen implementiert.

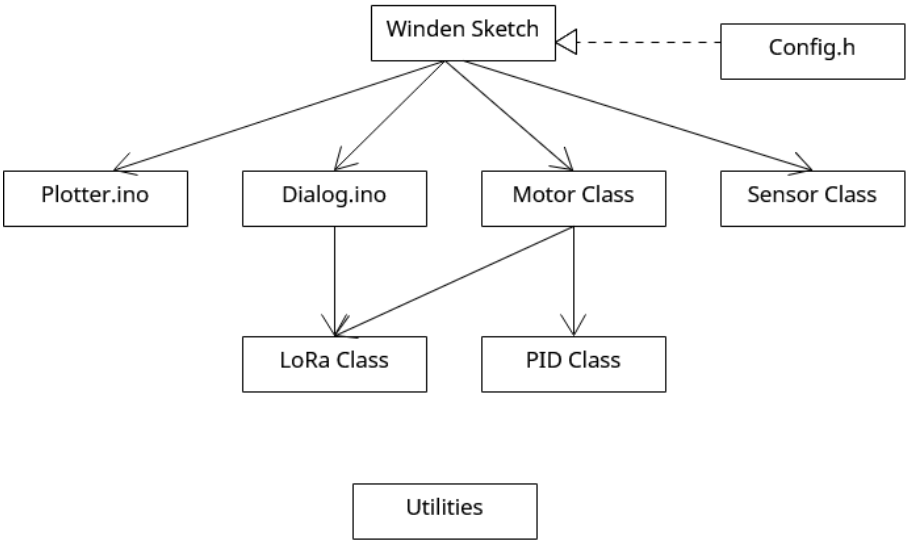


Abbildung 12 Klassendiagramm

Konfiguration

Die wichtigsten Parameter zur Steuerung der Winde sind in dem Include *config.h* gespeichert.

```
const bool zweiHallSensoren = true;

// Spannung checken
const bool CheckSpannung = false;  // Spannungsprüfung eingeschaltet = true Achtung
const int lipocell = 6;            // Anzahl Lipos

//---------------------------------------------------------------
//  Konstante
//---------------------------------------------------------------

// Drehzahlen
const int NotausRPM = 50.0;           // wenn nach 2 Sekunden nach Start die Drehzahl nicht
erreicht wird => Notaus
const int DefaultRPMDialog = 2500.0;  // Rewind Drehzahl Anfangswert für Dialog
const float MaxRPM = 5000.0;          // maximale Drehzahl Seil holen für Dialog
const float MinRPM = 2500.0;          // minimale Drehzahl Seil holen für Dialog

const float StartRPM = 3000.0;  // RPM für Bremsen mit SeilHolen, NEUE FUNKTION!!!!!

// Seilspannung
const float DefaultSeilSollDialog = 65.0;  // Seilspannung Soll in Newton für Dialog bei
Start der Winde
const float SeilHolenSeilspMax = 25.0;     // Seil verhedert beim Bremsen oder Rewind, max
Seilspannung
```

```
const float SeilspMin = 5.0;
// Die Überwachung der Seilspannung erfolgt allerdings erst eine
// Sekunde nach dem Start um ein vorzeitiges Auslösen des Bremsvorganges zu vermeiden.
// Außerdem muß die Seilspannung 10 Mal hintereinander den Mindestwert unterschreiten (ca.
200 msec)
// bevor der Bremsvorgang eingeleitet wird. Falls die Bedingungen erfüllt sind wird die
Seglerschlepp Phase
// beendet und die Bremsphase gestartet. Wenn der Fußschalter gelöst wird erfolgt ebenfalls
der Aufruf der Bremsphase.

const float Seilspmax = 150.0;        // Max einstellbare Seilspannung Dialog
const float SollSeilspBremsen = 8.0;  // Seilspannung Soll beim Bremsen
// Softanlauf Phase 1 und 2 mit festem Duty Cycle
const int DutyCycleSoftAnlauf1 = 38;   // 44
const int DutyCycleSoftAnlauf2 = 70;   //77
const float SeilspAnlauf1Ende = 0.50;  //60  // Die Phase 1 wird beendet, wenn die
Seilspannung 50% von der Sollseilspannung erreicht hat
const float SeilspAnlauf2Ende = 1.0;   // Die Phase wird beendet, wenn die Seilspannung ca.
100 % von der Sollseilspannung erreicht hat
// weitere Konstanten
const unsigned long Bremsdauer = 3000UL;  // Bremsdauer msec
const float MaxCurrent = 85.0;            // maximal zulässige Strom, bei
// Zeiten, Timer
const unsigned long SoftAnlauf1 = 3000000;         // Dauer Softanlauf Phase 1 in
Mikrosekunden
const unsigned long SoftAnlauf2 = 7000000;         // Dauer Softanlauf Phase 2 in
Mikrosekunden
const unsigned long MaxauerSoftAnlauf = 10000000;  // Max Dauer Softanlauf inkrosekunden
// ---------------------------------------------------------------
// PidParameter Motor
// ---------------------------------------------------------------
const int SpeedMappingRPM = 3000;  // Mapping Drehzahl Regler Ergebnis in PWM Signal
const int SpeedMappingSeilsp = 100;  // Mapping Seilspannung Regler Ergebnis in PWM Signal
struct PidParameter {  //Seilspannung
  float K = 1.0;
  float Kp = 1.7;  // 1.7; 20.03.25
  float Ki = 1.8;  // 2.0; 20.03.25
  float Kd = 0.0;
};
struct PidParameterRPM {  //Rewind und Soft Start
  float K = 1.0;
  float Kp = 1.0;
  float Ki = 1.75;
  float Kd = 0.00;
};
```

Winden Version

Da sich die einzelnen Winden teilweise in den Einstellparametern/Kalibrierung unterscheiden, wird die Windennummer, die SW Version und das Datum im EEPROM einmalig gespeichert und bei jedem Start wieder ausgelesen.

Abbildung 13 Winden Version

PID Regler

Der PID Regler ist in der Klasse PID Control realisiert. Es werden zwei Instanzen vom PID Controller erzeugt, eine für die Regelung der Seilspannung und eine für die Regelung der Drehzahl.

```
// ------------------------------------------------------------
// PID Controller
// ------------------------------------------------------------
#include "PidControl.h"

PidParameter PidParams;
PidParameterRPM PidParamsRPM;

PidControl PidControler(PidParams);        // PID Controller für Seilspannung
PidControl PidControlerRPM(PidParamsRPM);  // PID Controller für Drehzahl
```

Jede Instanz hat eigene PID Parameter.

```
// PidParameter Motor
// ------------------------------------------------------------
```

```
struct PidParameter {  //Seilspannung
  float K = 1.0;
  float Kp = 1.5;  //1.8
  float Ki = 3.0;  //3.0
  float Kd = 0.0;  //0.5;
};

struct PidParameterRPM {  //Rewind und Soft Start
  float K = 1.0;
  float Kp = 1.0 ;//1.5;
  float Ki = 1.75;  //3.0;no
  float Kd = 0.00;
};
```

Die Klasse PID Controller konnte ohne Änderungen von anderen Projekten übernommen werden. Die Klasse PID Controller besteht aus den folgenden Methoden.

Motor Steuerung

Der Motor wird mit Hilfe der Klasse *Motor.cpp* gesteuert. Es wird eine Instanz der Klasse im Hauptprogramm erzeugt. Als Referenz werden die Pointer auf beide PID Controller Instanzen übergeben.

Für ein völlig neues Projekt mit einem Brushless Motor könnten die Klassen PID Controller und Motor mit dem Include Config.h ohne weitere Anhängigkeiten in einem neuen Sketch verwendet werden.

Die Klasse Motor besteht aus den folgenden Methoden:

```
#include "Motor.h"
//************************************************************************ /
class Motor
///***********************************************************************/
{
public:
  Motor(PidControl* PIDcontr, PidControl* PIDcontr2, globalData* iParams);  // Constructor
  void MotorStart();
  void PWMStart();
  bool Run(float SeilSoll, float SeilIst, float& SeilSollAktuell, float& SollRPMaktuell, int Mode);
  void SetDutyCycle(int iSpeed, int MapLimit);
  bool RunRewind(float SollRPM, float IstRPM);
  void SetDutyCycleSD(int DutyCycle);
  void SetDutyCycleIN(int DutyCycle);
  bool MotorBremsen(float SeilIST, float& SeilSoll);
  bool MotorBremsen(float SeilIST, float& SeilSoll, float SollRPM, float IstRPM);
```

```
bool SeilspLow(float seilspannung);
bool CheckSeilspMax(float seilspannung);
void CheckStrom();
bool MindestRPM(float RPM);
void addStatistik();
void TimerCreate();
unsigned long DeltaTime();
unsigned long MotorStop();
float CalculateRopeLength(long RPM);
```

Die Methoden *RUN, RunRewind*, *Bremsen* und *MotorStart* werden vom Hauptprogramm aufgerufen. Die Methoden ermitteln den erforderlichen Duty Cycle durch Aufruf der Methoden des PID Controllers. Das erforderliche PWM Signal wird in der Methode *MotorStart* erzeugt.

PWM Signal

wir hatten zunächst naiv das PWM Signal klassisch mit ***analogWrite*** erzeugt. Bei unserem Arduino Mega bedeutet dies eine Frequenz von 490 Hz.

Bei Vollgas lief der Motor bei 2 Volt mit ca. 2 Ampere. Bei einem mittleren Duty Cycle zog der Motor über 7 Ampere und hörte sich ungesund an.

Nach entsprechender Recherche sind wir zu folgender Erkenntnis gelangt:
Bei der Pulsweitenmodulation kommt es in jedem PWM Zyklus sowohl zu einer Stromzunahme als auch zu einer Stromabnahme. Die Abweichung zwischen niedrigstem und höchstem Stromwert wird als Rippelstrom bezeichnet. Ein hoher Rippelstrom kann problematisch sein; er sollte daher so niedrig wie möglich gehalten werden. Ein hoher Rippelstrom verursacht hohe Motorverluste. Eine höhere Frequenz verkürzt die Zykluszeit der PWM, so daß der Rippelstrom reduziert wird.

Andererseits treten Schaltverluste der MOSFETs beim Umschalten vom leitenden in den nichtleitenden Zustand und umgekehrt auf. Die Verluste sind direkt proportional zur Schaltfrequenz.

Wir haben verschieden Frequenzen ausprobiert und waren mit dem Ergebnis bei 10Khz zufrieden.

Zur Generierung der PWM Signale wird folgende Library genutzt :https://github.com/khoih-prog/AVR_PWM

Um jedoch den Duty Cycle zu ändern ist jedesmal der Aufruf von
`PWM_Instance->setPWM(pinToUse, frequency, dutyCycle);`

erforderlich. Dabei wird jedoch in der Library jedesmal der Timer zurückgesetzt, was zu einem unsauberen Signal führt.
Ich habe das Ganze mit folgendem Workaround gelöst. Zum Ändern des DutyCycles wird nur noch das OCR3A Compare Register für PWM_IN und für PWM SD das OCR4A Register geändert. Man könnte auch selber direkt die Timer Register programmieren. Der Vorteil dieser Lösung ist, daß das Ganze auf verschiedenen Arduinos und für beliebige Frequenzen funktioniert. Lediglich das OCR3A und das OCR4A Compare Register muß an die jeweiligen Bedürfnisse angepaßt werden.

```
// ----------------------------------------------------------------
void Motor::PWMStart()
// ----------------------------------------------------------------
{                                                                  /*
   //https://github.com/khoih-prog/AVR_PWM
  frequency = 10000;                                               // 10Khz
  PWM_Instance = new AVR_PWM(PWM_IN_pin, frequency, 0.0);  //   Pin  5 => TIMER3A
  PWM_Instance->setPWM(PWM_IN_pin, frequency, 0);
  PWMPeriod = PWM_Instance->getPWMPeriod();

  PWM_InstanceSD = new AVR_PWM(PWM_SD_pin, frequency, 0.0);  //   Pin  3 => TIMER0B
  PWM_InstanceSD->setPWM(PWM_SD_pin, frequency, 0.0);
  PWMPeriodSD = PWM_InstanceSD->getPWMPeriod();
}
```

```
void Motor::SetDutyCycleIN(int DutyCycle)  // 0 - 100
// ----------------------------------------------------------------
{
  OCR3A = (float(map(DutyCycle, 0, 99, 0, int(PWMPeriod))));  // scale it to use
}
// ----------------------------------------------------------------
void Motor::SetDutyCycleSD(int DutyCycle)  // 0 - 100
// ----------------------------------------------------------------
{
  OCR4A = (float(map(DutyCycle, 0, 99, 0, int(PWMPeriodSD))));  // scale it to use
}
```

```
// ----------------------------------------------------------------
void Motor::SetDutyCycle(int iSpeedPID, int MapLimit)
// ----------------------------------------------------------------
```

```
{/*
  Für die Windensteuerung wird eine Regelbereich von Vollgas
  über bremsen bis zum Leerlauf benötigt.
  Dies wird erreicht indem das SD Signal ebenfalls mit einem PWM Signal versorgt wird.
  */
  if (iSpeedPID >= 0) {
    // Seilspannung zu tief
    speed = (map(iSpeedPID, 0, MapLimit, 0, 99));
    speed = constrain(speed, 0, 99);
    PWM_IN_DutyCycle = (abs(speed));  // PWM_IN Signal 1 - 99%
    SetDutyCycleIN(PWM_IN_DutyCycle);
    PWM_SD_DutyCycle = 99;  // PWM_SD Signal = 99%
    SetDutyCycleSD(PWM_SD_DutyCycle);
  } else {
    // Seilspannung zu hoch
    speed = (map(iSpeedPID, 0, -MapLimit, 99, 0));
    speed = constrain(speed, 0, 99);
    PWM_SD_DutyCycle = abs(speed);  // PWM_SD Signal = 1- - 99%
    SetDutyCycleSD(PWM_SD_DutyCycle);
    PWM_IN_DutyCycle = 0;  // // PWM_IN Signal = 0%
    SetDutyCycleIN(PWM_IN_DutyCycle);
  }
}
```

Berechnung Sollwertfunktion, Berechnung PID und Duty Cycle setzen

```
if (Mode == RunNormal || Mode == LoRaRunNormal) {
if (!targevalue_calcuate(SollFunktionsWert, SeilSoll, StartNormal, AnzValuesRunNormal))
return false;  // Sollfunktion berechnen
}
if (Mode == RunFast || Mode == LoRaRunFast) {
if (!targevalue_calcuate(SollFunktionsWert, SeilSoll, StartFast, AnzValuesRunNormal))
return false;  // Sollfunktion berechnen
}
// PID Controller aufrufen
 speed = (int(pPidControl->calculate(SeilIst, SollFunktionsWert)));
 SetDutyCycle(speed, 100); // Duty Cycle + Mapping Limit
```

Steuerung des Sollwerts der Seilspannung

Der Verlauf des Sollwertes der Seilspannung bzw. der Drehzahl kann über eine Methode der Klasse Motor gesteuert werden.

Die Funktionen sind können für

- Start Seilspannung
- Start fast Seilspannung
- StartRewind
- SollBremsen

getrennt eingestellt werden. Die Sollwert Verlaufs Kurven sind in einem Array definiert . Wobei die erste Spalte die Zeit in Millisekunden definiert und in der zweiten Spalte der Faktor mit dem der Sollwert multipliziert wird

```
#define AnzValuesRunNormal 5
  struct SollwertFunktion StartNormal[AnzValuesRunNormal] = {
    { 0, 0.0 },
    { 1, 1.3 },       // in 1 msec auf 130 % Seil Soll
    { 1500, 1.0 },    // nach 1501 msec reduziert auf 100%
    { 14000, 0.70 },  // nach 15001 msec 70% reduzieren
    { 99999, 0.0 },   //99999 bedeutet Ende Kennung
  };

#define AnzValuesRunFast 5  //
  struct SollwertFunktion StartFast[AnzValuesRunFast] = {
    { 0, 0.0 },
    { 1, 1.3 },       // in 1 msec auf 130 % Seil Soll
    { 1500, 1.0 },    // nach 1501 msec reduziert auf 100%
    { 14000, 1.0 },  // nach 15001 auf 100 % bleiben
    { 99999, 0.0 },   //99999 bedeutet Ende Kennung
  };

#define AnzValuesRewind 6
  struct SollwertFunktion StartRewind[AnzValuesRewind] = {
    { 0, 0 },
    { 1, 0.2 },
    { 100, 0.4 },
    { 1000, 1.0 },
    { 300000, 1.0 },  // 5 Minuten auf 100% bleiben
    { 99999, 0.0 },   // 99999 bedeutet Ende Kennung
  };

#define AnzValuesBremsen 4  //
  struct SollwertFunktion SollBremsen[AnzValuesBremsen] = {
    { 0, 0 },
    { 10, 1.0 },  // in 10 msec auf 100 % Seil Soll
    { 5100, 0.3 },
    { 99999, 0.0 },  //99999 bedeutet Ende Kennung
  };
```

Sensoren

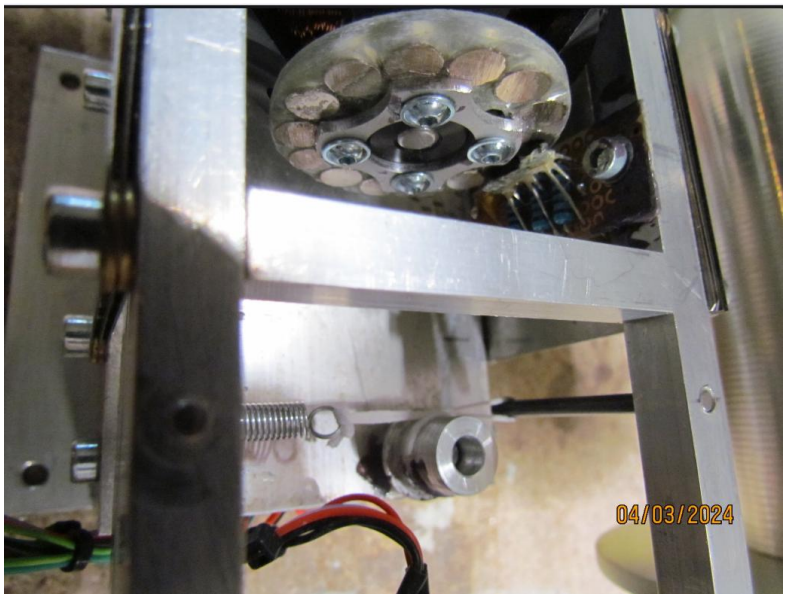
Folgende Sensoren müssen überwacht werden:

- Strom

- Spannung
- Seilspannung
- Drehzahl

Drehzahlmessung

Zur Drehzahlmessung sind 14 Magnete auf einer rotierenden Scheibe befestigt. Jedes Mal, wenn ein Magnet am Sensor vorbeikommt, erzeugt der Sensor ein Signal. Um negative Drehzahlen darzustellen werden zwei Hallsensoren ausgewertet.



Berechnung Drehzahl:
14 Magnete => pro Umdrehung bei Changing Interrupt 14 Interrupts damit pro Minute. RPM = (Interrupt Counter / Messintervall in micros) * 60 * 1000000 / 14.

Bei der Drehzahl Ermittlung wurde zunächst bei jedem Interrupt des Hall Sensors ein Zähler inkrementiert. Alle 10Msec wurde die Drehzahl mittels der Zählerdifferenz und der Delta Zeit ermittelt. Auch bei konstanter Drehzahl ist das Ergebnis bei niedrigen Drehzahlen nicht stabil. Das Intervall von 10 Msec wurde gewählt, weil der PID Regler auch alle 10 Msec aufgerufen wird. Bei einer Drehzahl von 100 RPM ergeben sich nur 0,233 Interrupts in 10 Msec. Bei einer Drehzahl von 1000 RPM ergeben sich nur 2,333 Interrupts in 10 Msec. Um stabile Werte zu erhalten wird die Drehzahl im 10 msec Intervall nur berechnet, wenn die Zahl der Interrupt > 4 oder < -4 ist. Wenn dies nicht der Fall ist , wird maximal 100 msec mit der Berechnung gewartet, dann wird die Drehzahl auf jeden Fall berechnet.

```
// ------------------------------------------------------------
// Drehzahl ermitteln
float SenssorData::CalculateRPM(int& isrCNT)
// ------------------------------------------------------------
{
  const float faktor = 60.0 * 1000000.0 / 14.0;  //Messung mit Changing
  float Delta = ((micros() - LastDeltaTime));
  if (Delta > 100000 || (isrCNT > 4 || isrCNT < -4)) {
    IstRPM = (isrCNT / Delta) * faktor;
    isrCNT = 0;
    LastDeltaTime = micros();
    Last = IstRPM;
  } else if (Delta > 100000 && isrCNT == 0) {
  }
  return IstRPM;
}
```

Ermittlung der Drehrichtung, bei jedem Interrupt wird die Drehrichtung ermittelt.

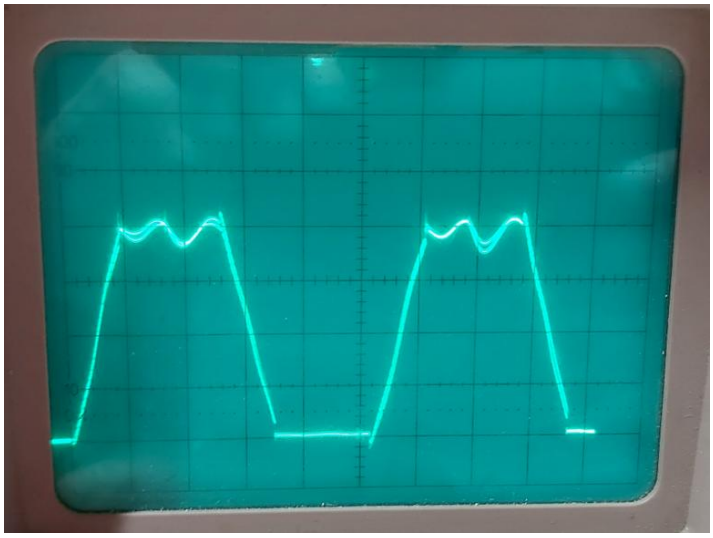
```
// ------------------------------------------------------------
bool SenssorData::RotationDirection()
// ------------------------------------------------------------
{
  status1 = (digitalRead(PinHallS1));
  status2 = (digitalRead(PinHallS2));
  if ((status1 == HIGH) & (status2 == HIGH) & (status2_old == LOW)) {
    forward = false;
  }
  if ((status2 == HIGH) & (status1 == HIGH) & (status1_old == LOW)) {
    forward = true;
  }
  status1_old = status1;
  status2_old = status2;
  return forward;
}
```

Interrupt Service Routinen

```
// ------------------------------------------------------------------------
//  Hall Sensor ISR Routinen
// ------------------------------------------------------------------------
void isrHall1() {
  noInterrupts();
  if (!zweiHallSensoren) {
    HallCount++; // für Ermittlung Drehzahl
  } else {
    if (Sensor.RotationDirection()) {
      SeilInterruptCounter--; // für Ermittlung Seilänge
```

```
      HallCount++; // für Ermittlung Drehzahl
    } else {
      SeilInterruptCounter++; // für Ermittlung Seilänge
      HallCount--; // für Ermittlung Drehzahl
    }
  }
  interrupts();
}
void isrHall2() {
  if (zweiHallSensoren) {  // Achtung
    noInterrupts();
    if (Sensor.RotationDirection()) {
      HallCount++;
    } else {
      HallCount--;
    };
    interrupts();
  }}
```

Vor der Inbetriebnahme der Winde müssen die Sensoren mittels Osziloskop kalibriert werden.



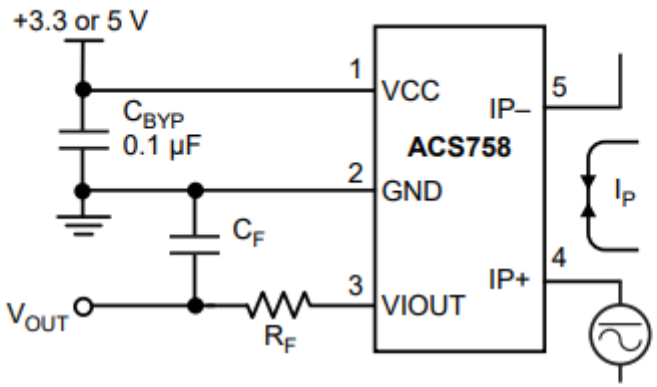
Strommessung

Bei ATmega-basierten Boards (UNO, Nano, Mini, Mega) dauert das Lesen eines analogen Eingangs etwa 100 Mikrosekunden (0,0001 s), so das die maximale Leserate etwa 10.000 Mal pro Sekunde beträgt.

Das ATmega-Datenblatt warnt auch davor, analoge Pins in unmittelbarer zeitlicher Nähe zu A/D-Lesungen (analogRead) an anderen analogen Pins zu lesen. Dies kann zu elektrischem Rauschen und Jitter im analogen System führen und hat uns tatsächlich Probleme bereitet. Es kann notwendig sein, nach der Manipulation analoger Pins eine kurze Verzögerung hinzuzufügen, bevor analogRead() zum Lesen anderer analoger Pins verwendet wird.

Andererseits muß der Strom sehr schnelle überwacht werden, da er bei einem Brushless Motor sehr schnell ansteigen kann. Aus diesem Grund werden die Sensoren in abwechselndem Rhythmus gelesen.

Der Strom wird mit einem ACS758xCB gemessen.



Application 1: The ACS758 outputs an analog signal, V_{OUT}, that varies linearly with the uni- or bi-directional AC or DC primary sampled current, I_P, within the range specified. C_F is for optimal noise management, with values that depend on the application.

Typical Application

Abbildung 14 ACS758xCB

Noise VNOISE TA= 25°C, 10 nF on VIOUT pin to GND

ACS758-Datasheet.
pdf

Falls der Strom über einen einstellbaren Grenzwert steigt wurde der Motor abgeschaltet. Dies ist öfters passiert. Deswegen haben wir den Algorithmus geändert. Wenn der Strom zu hoch ist, wird der aktuelle Duty Cycle bei jedem Abtastintervall reduziert, bis der Strom wieder in einem

zulässigen Bereich ist. Nach Austausch der Stromsensoren hat sich das **Problem erledigt**.

Spannungsmessung

Die Spannung wird mittels eines Spannungsteilers gemessen. Falls die Spannung unter einen einstellbaren Grenzwert sinkt, erfolgt eine Fehlermeldung auf dem Display.

Seilspannung

Die Seilspannung wird mit Dehnungsmessstreifen mit einem Poti gemessen. Der Motor wird mittels der Seilspannung geregelt. Die Kalibrierung der Seilspannung wird über die Winden Nummer ausgewählt.(siehe Seite 34)Konfiguration

Arduino

Es wird der Arduino Mega 2560 Rev3 eingesetzt.

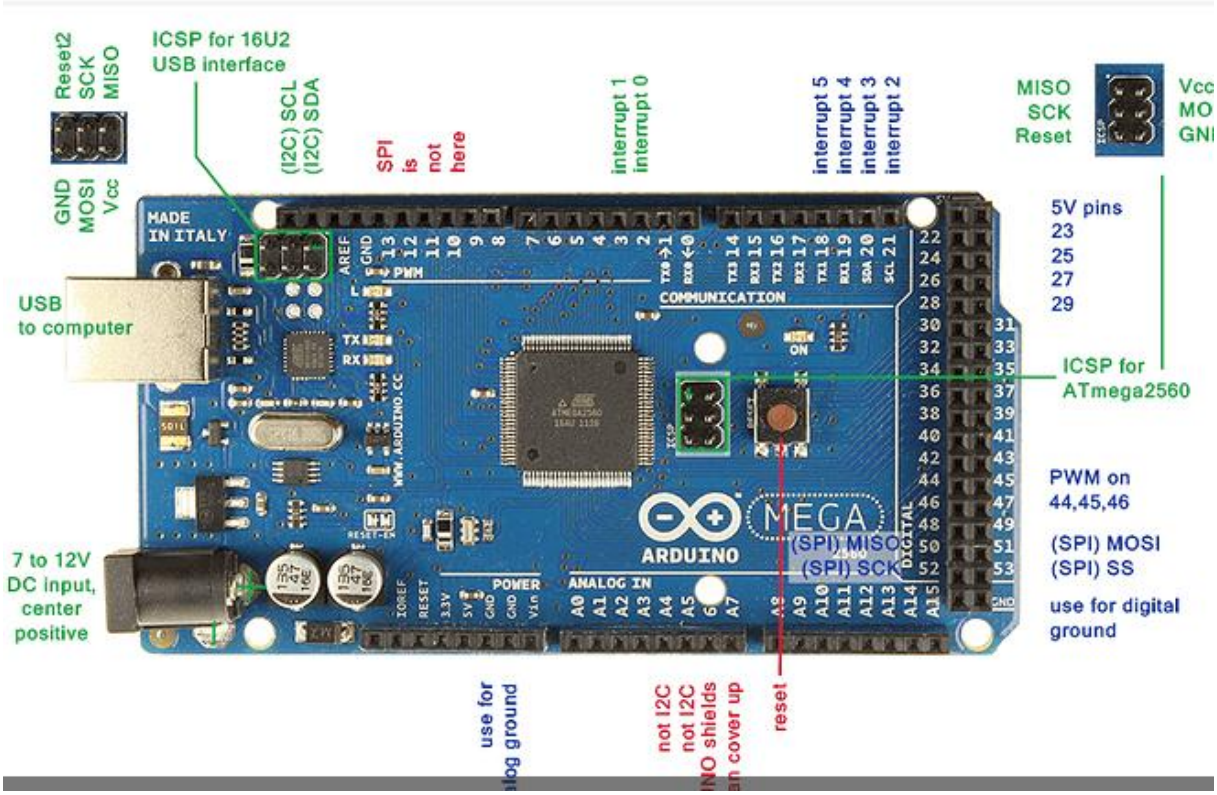


Abbildung 15 Arduino Mega

Pinout Arduino Mega

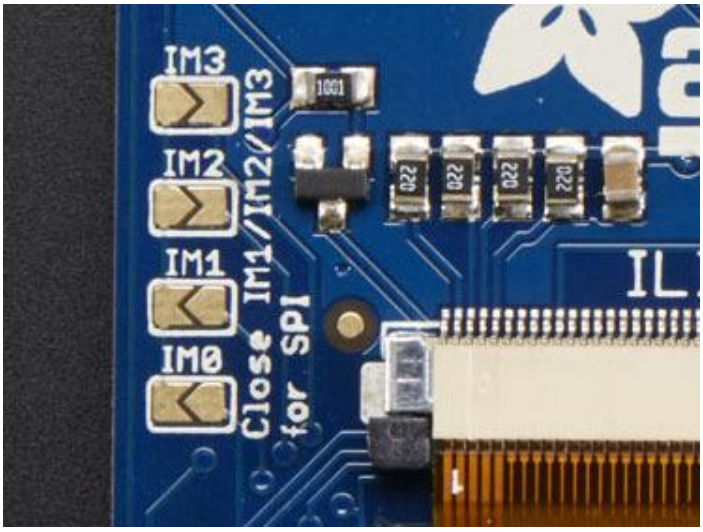
```
// ----------------------------------------------------------------
// PinOut
// ----------------------------------------------------------------
/* Digital Pins Usable For Interrupts
Mega, Mega2560, MegaADK 2, 3, 18, 19, 20, 21 (pins 20 & 21 are not available to use for interrupts while
*/
#define PinHallS1 2          // Drehzahl Messung
#define PinHallS2 3          // Drehzahl Messung NEU
#define StartSwitch 45       // Fußschalter
#define PWM_IN_pin 5         // Pin  5 => TIMER3A PWM_Instance  https://github.com/khoih-prog/AVR_PWM
#define PWM_SD_pin 6         // PWM shut down  // !!!! von 4 auf 6
#define SensorSeilspSimu A3  // Diagnose PWM Signal Poti
#define SensorSeilsp A14     // Echt
#define VoltPin A6           // Voltage VIN
#define currentPin A7        // Allegro ACS758 Current Sensor
#define TFT_CLK 52           // Display und LoRa Board = SCK
#define TFT_MISO 50          // Display und LoRa Board = MiSo
#define TFT_MOSI 51          // Display und LoRa Board = MoSi
#define TFT_CS 39            // Display
#define TFT_DC 41            // Display
#define SDA 20               // Realtime Clock und capacitive touch chip
#define SCL 21               // Realtime Clock und capacitive touch chip

// //Adafruit RFM95W LoRa Radio Transceiver Breakout - 868 or 915 MHz - RadioFruit
#define RFM95_CS 9
#define RFM95_RST 8
#define RFM95_INT 18
```

Display

Es ist das *Adafruit „TFT Adafruit 2.8" TFT Touch Shield v2Capacitive* eingestzt. Hier der Link zu dem Display https://learn.adafruit.com/adafruit-2-8-tft-touch-shield-v2?view=all.

SPI Mode Jumpers

Before you start, we'll need to tell the display to put us in SPI mode so it will know which pins to listen to. To do that, we have to connect tbe IM1, IM2 and IM3 pins to 3.3V. The easiest way to do that is to solder closed the IMx jumpers on the back of the PCB. Turn over the PCB and find the solder jumpers



With your soldering iron, melt solder to close the three jumpers indicated **IM1** **IM2** and **IM3** (do not solder closed **IM0**!)

Wiring

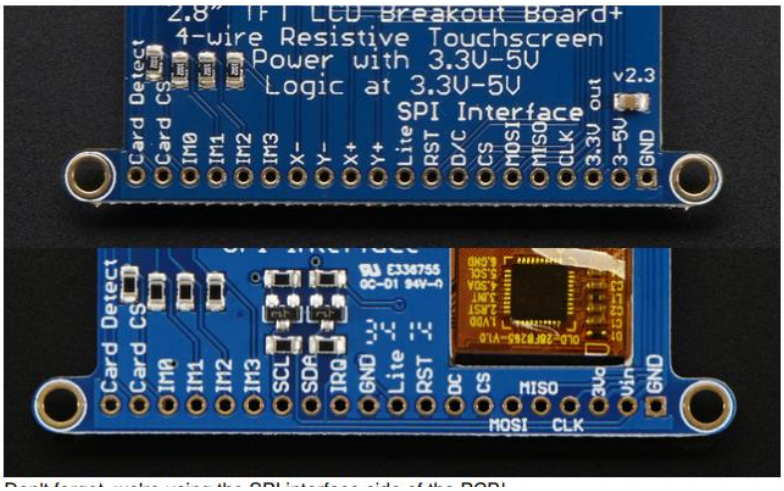
Wiring up the display in SPI mode is much easier than 8-bit mode since there's way fewer wires. Start by connecting the power pins

- **3-5V Vin** connects to the Arduino **5V** pin
- **GND** connects to Arduino ground
- **CLK** connects to SPI clock. On Arduino Uno/Duemilanove/328-based, thats **Digital 13**. On Mega's, its **Digital 52** and on Leonardo/Due its **ICSP-3** (See SPI Connections for more details)
- **MISO** connects to SPI MISO. On Arduino Uno/Duemilanove/328-based, thats **Digital 12**. On Mega's, its **Digital 50** and on Leonardo/Due its **ICSP-1** (See SPI Connections for more details)
- **MOSI** connects to SPI MOSI. On Arduino Uno/Duemilanove/328-based, thats **Digital 11**. On Mega's, its **Digital 51** and on Leonardo/Due its **ICSP-4** (See SPI Connections for more details)

- **CS** connects to our SPI Chip Select pin. We'll be using **Digital 10** but you can later change this to any pin. **hier 41**
- **D/C** connects to our SPI data/command select pin. We'll be using **Digital 9** but you can later change this pin too**. hier 39**
- Siehe Include *Config.h*

```
#define TFT_CLK 52
#define TFT_MISO 50
#define TFT_MOSI 51
#define TFT_CS 39  // 10
#define TFT_DC 41  // 9
```

SPI Wiring and Test



Achtung: Abbildung Resitive Rouch Screen, Y- und X- entspricht SCL und SDA.

The capacitive touch chip shares the same power and ground as the display, the only new pins you must connect are **SDA** and **SCL** - these must connect to the Arduino I2C pins.

- Connect the **SCL** pin to the I2C clock **SCL** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **A5**, on a Mega it is also known as **digital 21** and on a Leonardo/Micro, **digital 3**
- Connect the **SDA** pin to the I2C data **SDA** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **A4**, on a Mega it is also known as **digital 20** and on a Leonardo/Micro, **digital 2**

```
#define SDA 20      // Realtime Clock und capacitive touch chip
#define SCL 21      // Realtime Clock und capacitive touch chip
```

Erweiterung mit LoRa

Die Kommunikation mit LoRa erfolgt mit zwei unterschiedlichen LoRa Radio Transceiver Breakouts.

In der Winde wird ein Adafruit RFM95W LoRa Radio Transceiver Breakouts eingesetzt. Als LoRa Sender benutzen wir ein Heltec_ESP32_LoRa_v3 Board.

Das Adafruit RFM95W LoRa Board benötig wie unser Arduino Mega 5 Volt Versorgungsspannung.

Für den Sender haben wird das Heltec Board aufgrund der Größe und aufgrund des integrierten Displays entschieden. Damit benötigen wir keinen Fuss Schalter sondern können das Display am Fernsteuer Sender befestigen.

LoRa Kommunikation

Die Kommunikation erfolgt nach dem Client Server Prinzip. Der Lora Sender arbeitet als Client und die Windensteuerung als Server. Es gilt streng das *Frage Antwort* Prinzip.

LoRa Protokoll

Alle Werte sind durch ein Komma getrennt und ohne Nachkommastellen, die Satzlänge ist nicht konstant. Der LoRa Client sendet folgende Telegramme:

- Header
- und eine Value

der Header kann folgende Werte annehmen:

- `#define WindeStatusabfrage "A"`
- `#define WindeStarten "S"`
- `#define WindeStartenFast "F"`
- `#define WindeStoppen "E"`
-

Die Value enthält die Seilspannung.

Beispiele:

- „S,120“ Start normal mit 120 Newton
- „F,60“ Start fast 60 Newton
- „A,75“ Statusabfrage mit Seilspannung 75 Newton zur Anzeige auf der Winde

Nach dem Start der Windensteuerung fragt der Client periodisch den Status der Windensteuerung ab. Der Server antwortet mit folgenden Headern:

```
#define WindeStatusabfrageOk "R"
#define WindeError "X"
```

Es wird ein eigenes radio.setSyncWord(z.B. 0x21); definieren. Dadurch werden andere Sender ausgefiltert und werden gar nicht erst durch radio.readData (Nachricht); an den Sketch gemeldet.

Beim Einschalten des Senders schaltet die Winde automatisch in den LoRa Mode um.

LoRa Airtime, **Maximaler Duty-Zyklus**
In Europa werden der Duty Cycle durch Abschnitt 4.3.3 der Norm ETSI EN300.220-2 V3.2.1 (2018-06) geregelt.

Die ETSI EN300.220 sagt sinngemäß:

Duty cycle is the ratio of transmission time over a rolling one hour period.

Das Wort **rolling** ist entscheidend.

Die ETSI-Norm verlangt **ZWEI getrennte Bedingungen** gleichzeitig.

✅ Regel 1 — Duty Cycle über 1 Stunde

Im 1-%-Band gilt:

> Maximal **36 Sekunden Airtime in ANY rolling 60-minute window**

Das ist die eigentliche rechtliche Grenze.

✅ Regel 2 — Mindest-Off-Time nach Sendung

Nach jeder Sendung:

$$OffTime = Airtime \times 99$$

Diese Regel verhindert, dass du alles am Stück sendest.

Wir senden und empfangen max. 5 Byte Nettodaten, damit haben wir eine maximale Airtime von 215 msec und damit minimal zu viel (200msec). Bei einer Reduzierung des Spreading Faktors auf SF 9 würde sich die Airtime halbieren (108 msec).

- BW = 125 kHz
- SF = 10
- CR = 4/6 (setCodingRate4(6))
- Preamble: nicht gesetzt ⇒ meist Default 8 Symbole
- CRC: nur wenn du LoRa.enableCrc() nutzt (sonst oft aus)
- SyncWord 0x21: keine Airtime-Änderung
- TxPower: keine Airtime-Änderung
- explicit Header

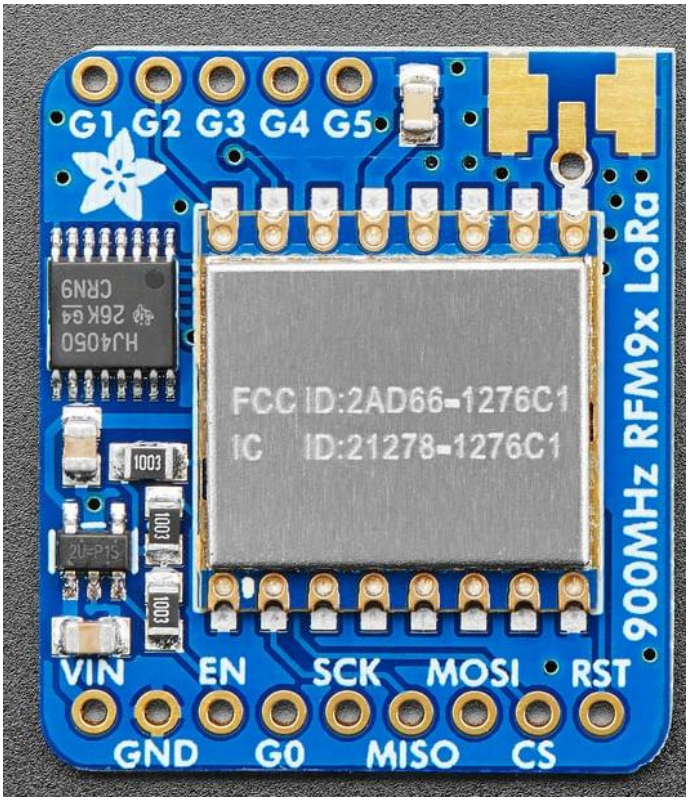
Adafruit RFM95W LoRa Radio Transceiver Breakouts

Wir verwenden die 900-MHz-Funkversion, die entweder für 868MHz, 915MHz oder 434 Mhz eingesetzt werden kann, die genaue Funkfrequenz wird beim Laden der Software festgelegt, da sie dynamisch eingestellt werden kann.

Jedes Funkmodul wird mit einer Stiftleiste, einem 3,3-V-Spannungsregler und einem Levelshifter geliefert, der 3-5 V Gleichspannung und Logik verarbeiten kann. Eine ausführliche Anleitung findet man unter:

https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/overview

https://www.adafruit.com/product/3072



Es wird folgende Library benötigt:

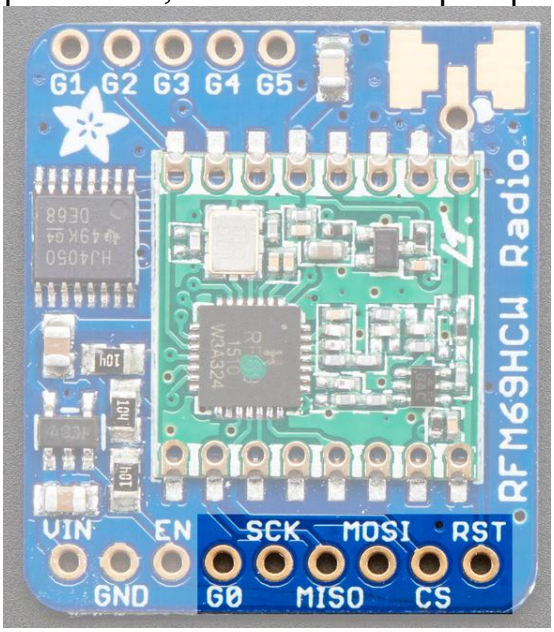
https://github.com/sandeepmistry/arduino-LoRa

Semtech SX1276/77/78/79 wiring

Semtech SX1276/77/78/79	Arduino
VCC	3.3V
GND	GND
SCK	SCK
MISO	MISO
MOSI	MOSI
NSS	10
NRESET	9
DIO0	2

NSS, NRESET, and DIO0 pins can be changed by using LoRa.setPins(ss, reset, dio0). DIO0 pin is optional, it is only needed for receive callback mode. If DIO0

pin is used, it must be interrupt capable via attachInterrupt(...).



- Vin connects to the Arduino 5V pin. If you're using a 3.3V Arduino, connect to 3.3V
- GND connects to Arduino ground
- SCK - This is the SPI Clock pin, its an input to the chip
- MISO - this is the Microcontroller In Serial Out pin, for data sent from the radio to your processor, 3.3V logic level
- MOSI - this is the Microcontroller Out Serial In pin, for data sent from your processor to the radio
- CS - this is the Chip Select pin, drop it low to start an SPI transaction. Its an input to the chip
- RST - this is the Reset pin for the radio. It's pulled high by default which is reset. Pull LOW to turn on the radio
- G0 - the radio's "GPIO 0" pin, also known as the IRQ pin, used for interrupt request notification from the radio to the microcontroller, 3.3V logic level.

Die Standard Pin Belegung wird modifiziert, da einige Pins in der alten Windensteuerung belegt sind

```
#define RFM95_CS 9
#define RFM95_RST 8
#define RFM95_INT 2
```

Mega Pin Belegung	Beteichnung RFM95W
#define RFM95_CS 9	CS
#define RFM95_RST 8	RST

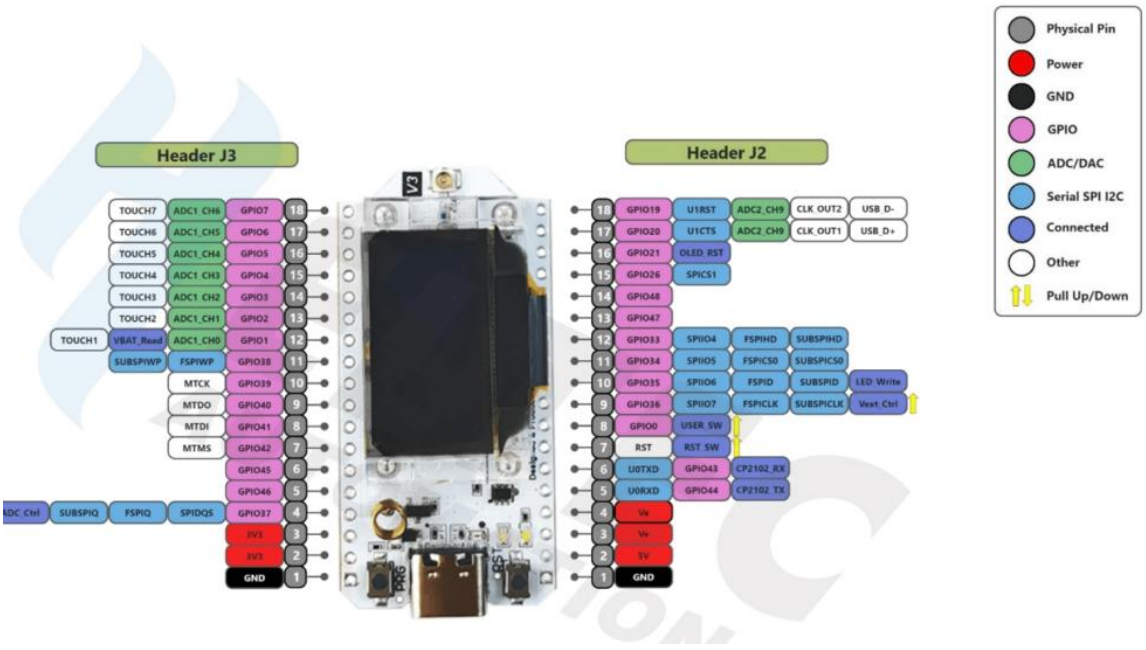
`#define RFM95_INT 18`	GO
`#define TFT_CLK 52`	SCK
`#define TFT_MISO 50`	MISO
`#define TFT_MOSI 51`	MOSI

Heltec_ESP32_LoRa_v3 Board

Es gibt eine chinesische Firma namens Heltec, die ein cooles kleines Entwicklungsboard herstellt, das einen Espressif ESP32S3 (mit WiFi und Bluetooth), ein 128x64 Pixel großes OLED-Display und ein SX1262 863-928 MHz Radio enthält.

WiFi LoRa 32 ist ein klassisches IoT Dev-Board, das von Heltec Automation entwickelt und hergestellt wird.

Support LoRa point-to-point and LoRaWan protocols.




```
/ ------------------------------------------------------------
// PinOut
// ------------------------------------------------------------
#define PinSeilspannung 19  // GiPo 18 Header J2
#define PinStartSwitch 46   // Gipo 46 Header J3
#define PinModeFast 45      // Gipo 45 Header J3
// ------------------------------------------------------------
```

3.2 Power supply

Except when USB or 5V Pin is connected separately, lithium battery can be connected to charge it. In other cases, only a single power supply can be connected.

Table 3.2: Power supply

Power supply mode	Minimum	Typical	Maximum	Company
Type-C USB(≥500mA)	4.7	5	6	V
Lithium battery(≥250mA)	3.3	3.7	4.2	V
5V pin(≥500mA)	4.7	5	6	V
3V3 pin(≥150mA)	2.7	3.3	3.5	V

LoRa Ralisierung mit Adafruit RFM95W

Die endgültige Realisierung der LoRa Kommunikation erfolgte auf Basis der Klasse „LoRa.h“ https://github.com/sandeepmistry/arduino-LoRa von Sandeep Mistry.

Die Klasse bietet folgende Features bzw. Vorteile gegenüber der Radio Head Klasse:

- Senden ohne blocking
- Beispiele für Adressierung
- Verwendung der C++ Stream Klasse anstelle von alten C Character Arrays, damit einfache Verwendung mit Print bzw. Write Anweisung möglich

Das Senden ohne blocking ist wichtig, die Winde sonst während des Sendevorgangs blockiert wäre.

Die Beispiele für einen LoRa Sender und Empfänger sind recht simpel (siehe https://github.com/sandeepmistry/arduino-LoRa/tree/master/examples)

Um die Programme für Sender und Empfänger möglichts einfach zu gestalten wurde die Library *lora.h* durch eine eigene Library *mylora.h* gekapselt. Die Library *mylora.h* übernimmt

- Senden der Daten
- Senden ohne blocking
- Empfang der Daten mit Plausibiltätskontrolle
- Das Mapping der Sende und Empfangsdaten

LoRa Ralisierung mit Heltec LoRa V3

Die Heltec LoRa V3 Version funktioniert nicht mehr mit der Library <lora.h> da als LoRa Chip ein SX1262 LoRa node chip verwendet wird . Zum Glück hat Rop Gonggrijp die geniale Library <heltec_unofficial.h> entwickelt, die die Version V3 unterstützt. https://github.com/ropg/heltec_esp32_lora_v3

Examles : https://github.com/ropg/Heltec_ESP32_LoRa_v3

Der *non-Blocking Mode*

> Zitat vom Entwickler der Radio Head Library Rop Gonggrijp :
> Using blocking receive is not recommended, as it will lead to significant amount of timeouts, inefficient use of processor time and can some miss packets! Instead, interrupt receive is recommended.

Die Verwendung von blockierendem Empfang wird nicht empfohlen, da dies zu einer beträchtlichen Anzahl von Timeouts und einer ineffizienten Nutzung der Prozessorzeit führt und einige Pakete fehlen können! Stattdessen wird der Interrupt-Empfang empfohlen.

Die Alternative ist das Senden und Empfangen im non blocking Mode. Die Übertragung der Nachrichten erfolgt quasi parallel zum Hauptprogramm. Damit ist der eigentliche Sketch während der Übertragung weiter lauffähig. Das Ende der Übertragung (Senden und Empfangen), wird mit einem Interrupt gemeldet.
Die folgenden Einstellungen müssen bei Sender und Empfänger gleich sein.

```
#define FREQUENCY 868.0
#define BANDWIDTH 125.0
#define SPREADING_FACTOR 10
#define TRANSMIT_POWER 0
#define SyncWord 0x21
```

Siggis Winde

Aufbau der Winde

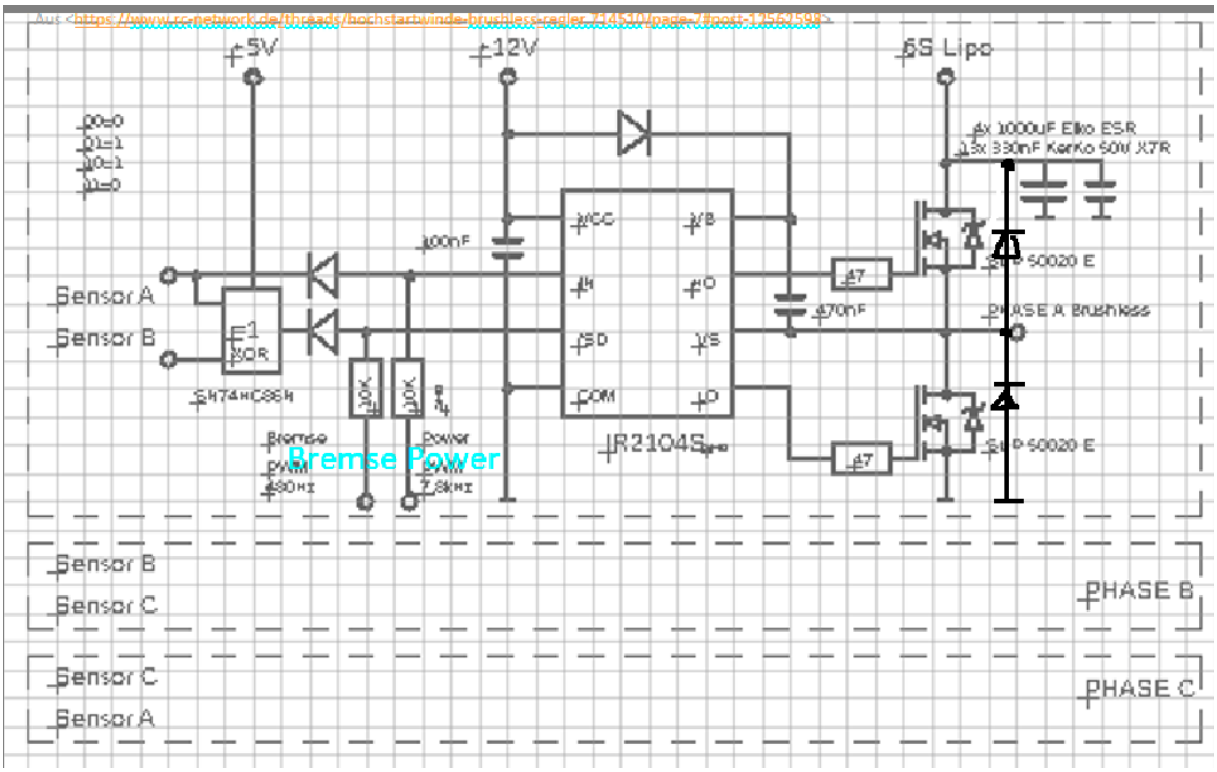
Siggi hat die komplette Mechanik erstellt und die Elektronik (Halbbrücke und Leistungsteil) realisiert.

- Brushless Motor **Surpass C5055 760KV**
- LiPo mit 6 Zellen
- Halbbrücke
- MOSFETLeistungsteil
- Umkehrspindel für Schleppseil
- Drehzahl Messung mit 2 Hallsensoren
- Messung Seilspannung mit Federstahl und Poti
- Funk Fernsteuerung über LoRa
- Touch Screen
- Arduino Mega
- Überwachung von Strom und Spannung

Elektronik

Hier die genialen Unterlagen vom Windenbauer Peter an denen wird uns orientiert haben.

https://www.rc-network.de/threads/hochstartwinde-brushless-regler.714510/



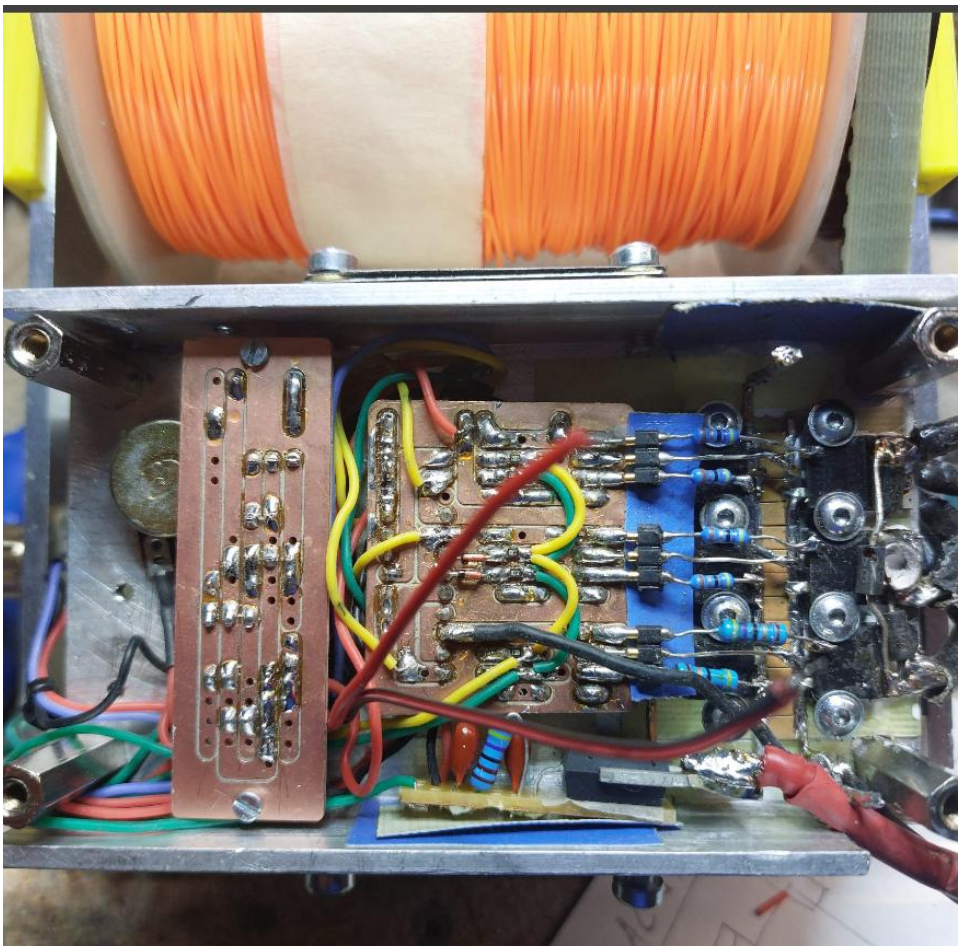
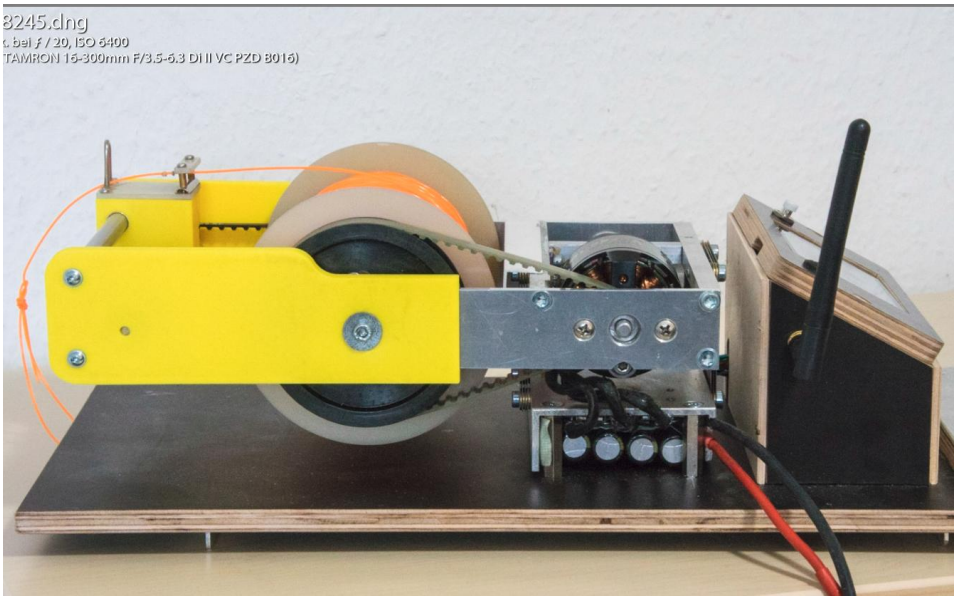


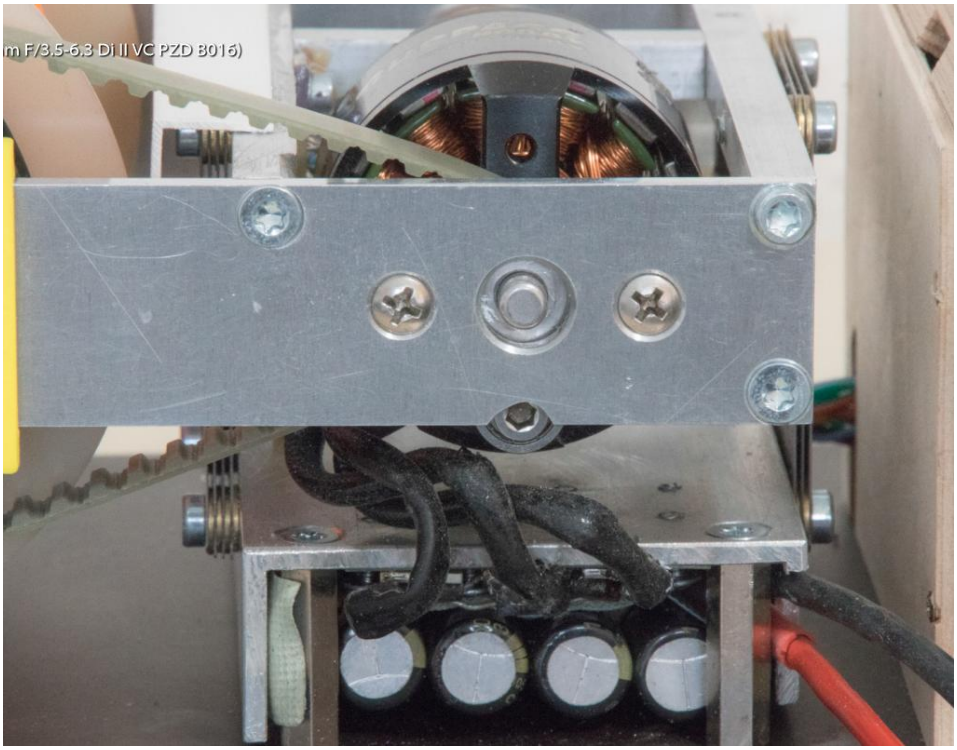
Abbildung 16 Elektronik mit MOSFETs

Die Leiterbahnen sind mit einer CNC Fräse hergestellt.

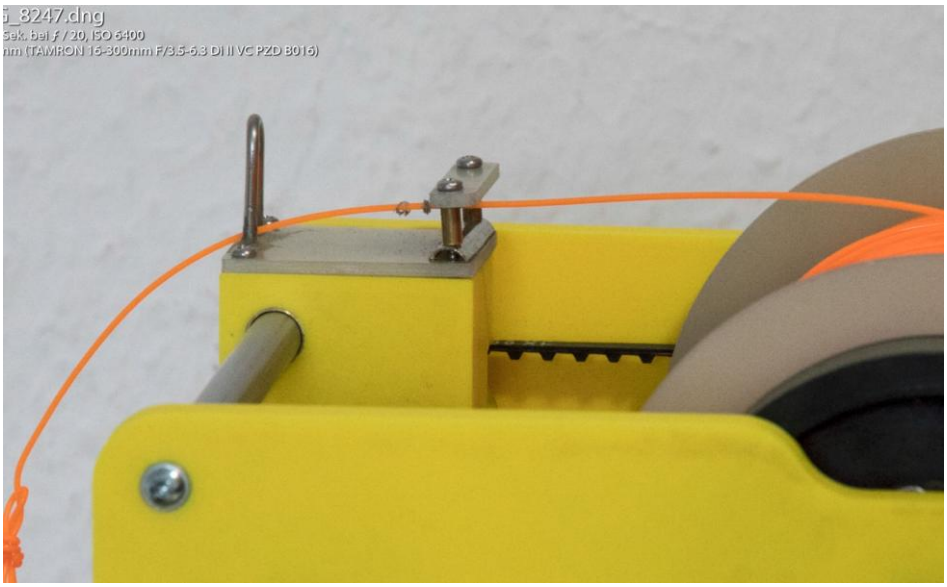
Siggis Winde

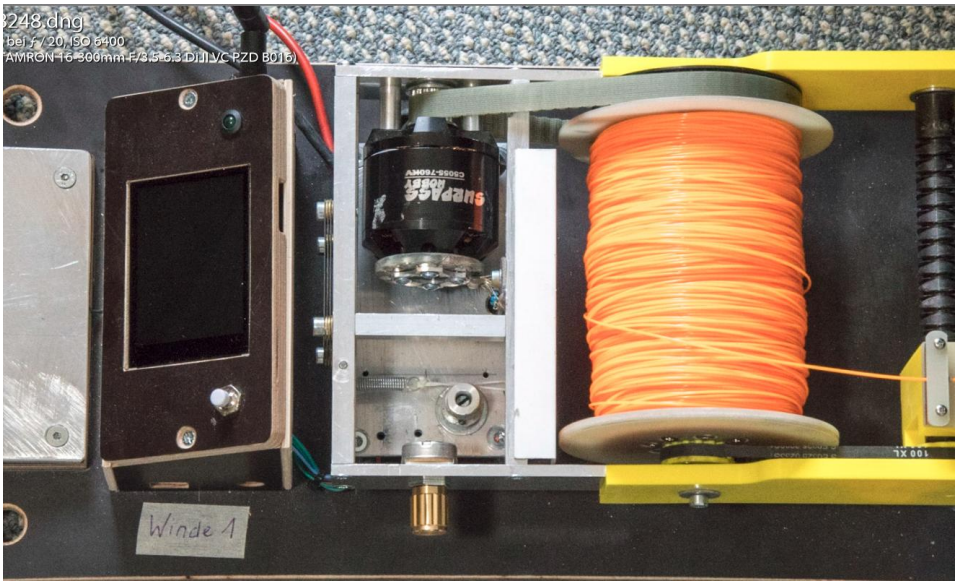
Mechanik



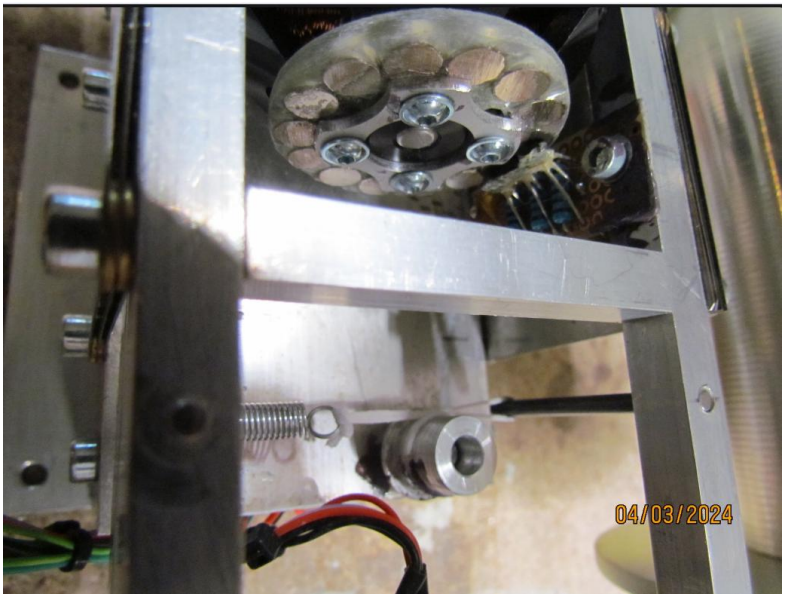


Siggis Winde





Drehzahl Messung mit zwei Hallsensoren



LoRa Sender

Für den Sender haben wird das *Heltec_ESP32_LoRa_v3 Board* aufgrund der Größe und aufgrund des integrierten Displays entschieden. Damit benötigen wir keinen Fussschalter sondern können das Display am Fernsteuer Sender befestigen.

Anhang

Drehzahlen

Drehzahl RPM	Anzahl Magnete	Anzahl Interrupts pro Umdrehung	Umdrehungen pro Sekunde	Umdrehungen pro 10 MSEC	Interrupt pro 10 MSEC
1,0000	14,0000	14,0000	0,0167	0,0002	0,0023
100,0000	14,0000	14,0000	1,6667	0,0167	0,2333
1000,0000	14,0000	14,0000	16,6667	0,1667	2,3333
10000,0000	14,0000	14,0000	166,6667	1,6667	23,3333

Interupts pro 10 MSEC	Interupts pro Sekunde	Interrupts pro Umdrehung	Umdrehungen pro Sekunde	Drehzahl RPM	Drehzahl RPM
1	100	14	7,1429	429	428,5714
10	1000	14	71,4286	4286	4285,7143
23	2333,3	14	166,6643	10000	9999,8571
100	10000	14	714,2857	42857	42857,1429
500	50000	14	3571,4286	214286	214285,7143

Siggis Winde

Winden Konfiguration

	Winde 1 NEU?	Winde 2
	Peter 50mm Trommel	Auslegung 40mm Trommel
Lipo	6	6
Spannung	22,2	22,2
MOTOR		
	Surpass 5055-760KV	Surpass 5055-760KV
Drehzahl [KV]	760	760
Drehmoment [Nm/A]	0,013	0,013
Max. Strom [A]	80	80
Max. Drehmonent [Nm]	1,01	1,01
Drehzahl [1/min]	16872	16872
WINDE		
Getriebe I	10	10
Getriebe II	48	36
Übersetzung	0,21	0,28
Übersetzung 1:x	4,80	3,60
Trommeldurchm. [mm]	50	40
Trommeldreh [Nm]	4,83	3,62
Trommeldreh [1/min]	3515	4687
Zugkraft Seil max. [N]	193	181
Zugkraft [kg]	19,675	18,445
Seilgeschw. [m/s]	9,2	9,8
Seilgeschw. [km/h]	33,1	35,3

Seileinzug bei ausgelegtem Seil (Trommel fast leer)! angenommene Seilaufwicklung/Umdrehung (gemittelt)	170mm	140mm

Logging erweitert. 06.03.2026

Start und Ende Logging Kennung LogA LogE.
Zur Unterscheidung von Textmeldungen und Log Daten am Anfang ein“;Sist“.
Das sind die Zeilen die in das CSV File müssen. Der Rest landet auch im Text File.

Start mit SeilSoll = 45.00 Newt.
LogA
;Sist: ;0; SSoll: ;45; RI: ;0; RS: ;0; IN : ;25; SD : ;99; Strom: ;-0; lISRTime:;0; run:; Anl:
;Sist: ;0; SSoll: ;45; RI: ;0; RS: ;0; IN : ;25; SD : ;99; Strom: ;2; lISRTime:;7; run:; Anl:

Seilsp. erreicht, weiter mit Stufe 2
;Sist: ;24; SSoll: ;45; RI: ;46; RS: ;0; IN : ;50; SD : ;99; Strom: ;3; lISRTime:;0; run:; Anl:
;Sist: ;25; SSoll: ;45; RI: ;48; RS: ;0; IN : ;50; SD : ;99; Strom: ;10; lISRTime:;0; run:; Anl:
;Sist: ;37; SSoll: ;45; RI: ;80; RS: ;0; IN : ;50; SD : ;99; Strom: ;6; lISRTime:;0; run:; Anl:

Seilsp. in Stufe 2 erreicht, Softanlauf Ende
Ende Softanlauf Seilsp. erreicht
;Sist: ;49; SSoll: ;58; RI: ;84; RS: ;0; IN : ;14; SD : ;99; Strom: ;6; lISRTime:;0; run:
;Sist: ;50; SSoll: ;58; RI: ;82; RS: ;0; IN : ;13; SD : ;99; Strom: ;0; lISRTime:;0; run:

Fusstaste OFF, bremsen starten !!!
;Sist: ;48; SSoll: ;48; RI: ;88; RS: ;0; IN : ;0; SD : ;0; Strom: ;-0; lISRTime:;0; bre:
;Sist: ;48; SSoll: ;48; RI: ;87; RS: ;0; IN : ;0; SD : ;0; Strom: ;-0; lISRTime:;0; bre:

Auslauf 8 Newt!
;Sist: ;3; SSoll: ;48; RI: ;75; RS: ;0; IN : ;0; SD : ;0; Strom: ;-0; lISRTime:;0; bre:
;Sist: ;0; SSoll: ;8; RI: ;71; RS: ;0; IN : ;12; SD : ;99; Strom: ;0; lISRTime:;0; bre:

Seilsp. Kp = 1.70 Ki = 1.80 Kd = 0.00
Drehzahl Kp = 1.00 Ki = 1.75 Kd = 0.00
Seilsp. Ende 1 = 0.50 Seilsp. Ende 2 = 1.00
StatisticCnt 84 Standardabweichung 22.36
Max. Strom 40.50
Seilaenge aktuell -16.27
PWM1 25
PWM2 50

max. Motor Stillstand Zeit 15 msec