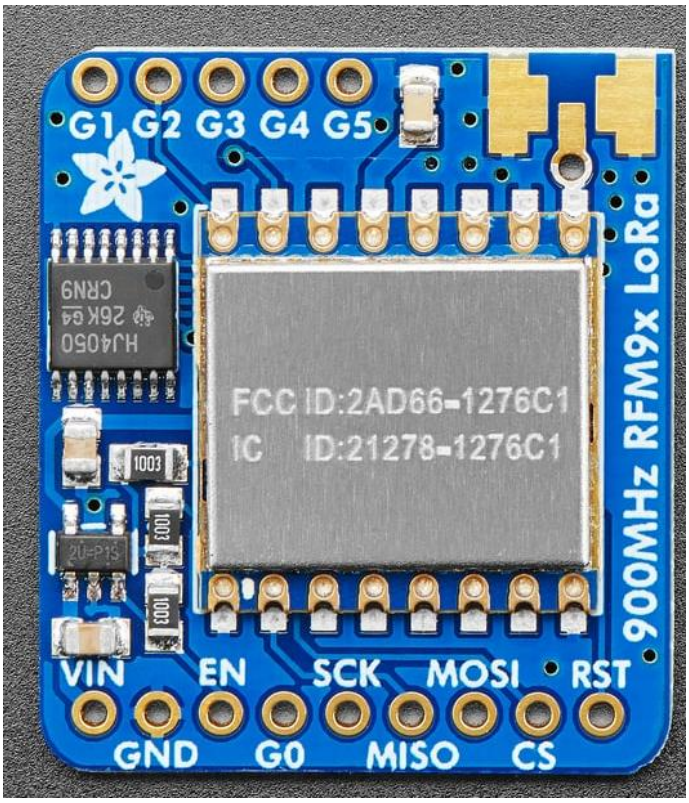
Erweiterung mit LoRa

Die Kommunikation mit LoRa erfolgt mit zwei Adafruit RFM95W LoRa Radio Transceiver Breakouts.

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



Diese Funkmodule gibt es in vier Varianten (zwei Modulationsarten und zwei Frequenzen). Die RFM69-Funkmodule sind einfach zu handhaben und sind gut bekannt. Die LoRa-Funkmodule sind spannender und leistungsfähiger, aber auch teurer.

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

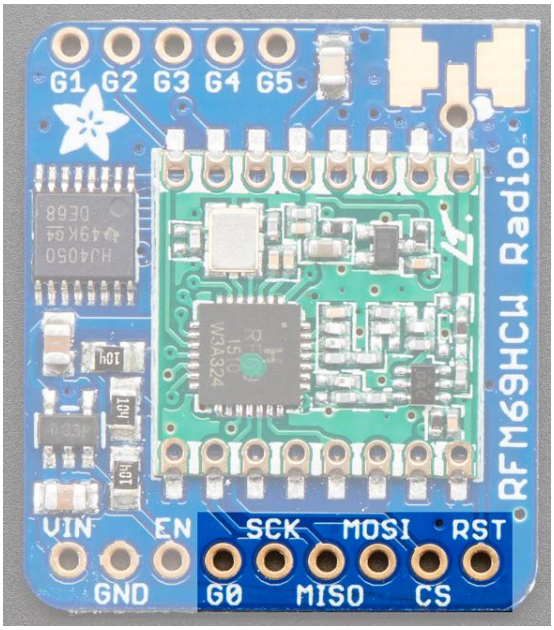
Jedes Funkmodul wird mit einer Stiftleiste, einem 3,3-V-Spannungsregler und einem Levelshifter geliefert, der 3-5 V Gleichspannung und Logik verarbeiten kann. Eine ausführliche Anleitung findet man unter:

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

Es wird folgende Library benötigt:

https://github.com/adafruit/RadioHead obsolet

The library requires hardware SPI and does not have software SPI support so you must use the hardware SPI port. All pins going into the breakout have level shifting circuitry to make them 3-5V logic level safe.



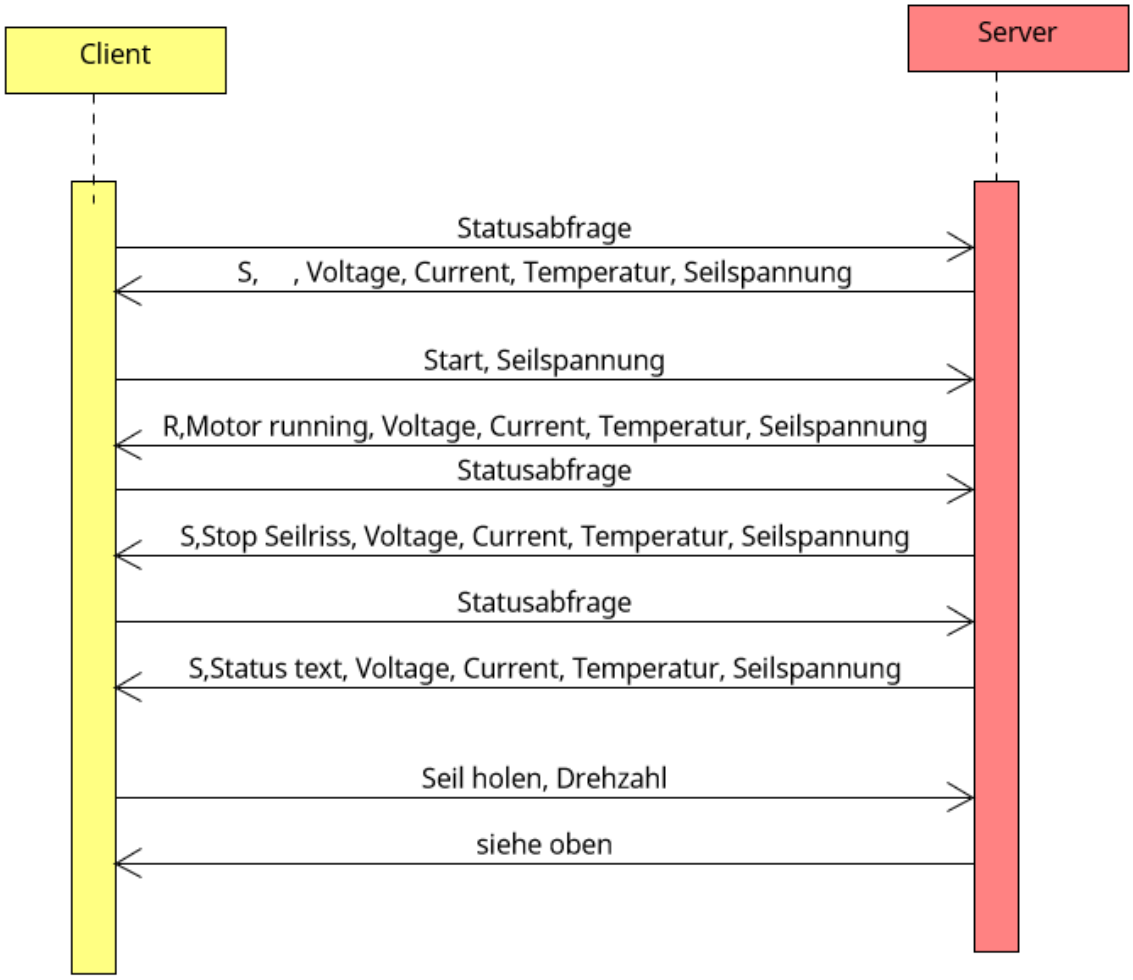
- Vin connects to the Arduino 5V pin. If you're using a 3.3V Arduino, connect to 3.3V
- GND connects to Arduino ground
- SCK - This is the SPI Clock pin, its an input to the chip
- MISO - this is the Microcontroller In Serial Out pin, for data sent from the radio to your processor, 3.3V logic level
- 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
```

LoRa Kommunikation

Die Kommunikation erfolgt nach dem Client Server Prinzip. Der Lora Sender arbeitet als Client und die Windensteuerung als Server.



UML Sequenzdiagramm 1

LoRa Sender

Der LoRa Client -Sender- wird mit einem Arduino Mega und einem RFM95W Modul realisiert. Im Sender befindet sich das Display, wie in der alten Windensteuerung und der Fußschalter.

LoRa Empfänger

Der Lora Server wird mit dem Arduino Mega und einem RFM95W Modul aber ohne Display realisiert. Die Software wird für die alte und die neue Variante einheitlich entwickelt. Die Variante wird lediglich mit einer #define Anweisung unterschieden.

```
/-----------------------------------------------------------------
// Winden Konfiguration
//-----------------------------------------------------------------
enum WindenTyp {
  LoRaServer,
  LoRaClient,
  OhneLoRa,
};
int WindenKonfig = LoRaServer;
//int WindenKonfig = OhneLoRa;
```

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:

- SN = Start normal
- SF = Start fast
- SH = Seil holen
- ST = Statusabfrage
- SE = Motor Bremsen und Stop

Die Value enthält die Seilspannung bzw. die Drehzahl.

Beispiele:

- „SN,0120“ Start normal mit 120 Newton
- „SH,2500“ Seil holen mit 2500 RPM
- „ST“ Statusabfrage
- „SE“ Motor Stop , Fußtaste gelöst

Nach dem Start der Windensteuerung fragt der Client jede Sekunde den Status der Windensteuerung ab. Der Server antwortet mit folgendem Datensatz:

- „Status“ RX = Motor läuft, SX = Motor steht
  X enthält folgende Werte:
  S = Seilriss bzw. ausgeklinkt
  N = Notaus
  H = Seilspannung zu hoch
  space = default
- Spannung
- Strom,
- Temperatur
- Seilspannung

alle Werte durch Komma getrennt.

Beispiel: „R “,25,45,35,120

Zusätzlich zur o.g. Payload werden noch folgende Informationen übertragen:

- destination address
- sender address
- message ID
- payload length

so dass ca. 20 Zeichen übertragen werden. Die sogenannte Airtime beträgt bei einem Spreading Faktor von SF7 ca. 70 Msec. Bei einem Startvorgang vom maximal 18 Sekunden soll jede Sekunde eine Statusmeldung erfolgen. Die Airtime beträgt dann 1,2 Sekunden pro Startvorgang. Der Duty Cycle beträgt damit bei einem Startvorgang maximal 7%. Na ja, ist etwas zu hoch aber nur für 18 Sekunden.

LoRa Ralisierung

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

Die Methoden der Klasse LoRa sind nochmals in der eigenen Klasse myLora gekapselt.

Damit reduziert sich das Coding für Server und Client wie folgt:

Server

```
//--------------------------------------------------------------------
void loop()
//--------------------------------------------------------------------
{
  LoRa.receive();  // put the radio into receive mode
}
//--------------------------------------------------------------------
void onReceive(int packetSize)
//--------------------------------------------------------------------
{
  if (myLora.receive(packetSize, ingoing)) {
    ;
    Serial.print("Got ");
    Serial.println(ingoing);

    // status; VoltageAkt; CurrentMax; TemperaturAkt; Seilspannung;
    myLora.ServerSendReply("S ", 24.3, 60.0, 23.7, 120);
  }
}
```

Client

```
//--------------------------------------------------------------------
void loop()
```

```
//---------------------------------------------------------------------
{
  if (millis() - lastSendTime > interval) {
    myLora.ClientSendMessage("S ", 120);
    lastSendTime = millis();          // timestamp the message
    interval = random(2000) + 1000;  // 2-3 seconds
  }

  // parse for a packet, and call onReceive with the result:
  onReceive(LoRa.parsePacket());
}
//---------------------------------------------------------------------
void onReceive(int packetSize)
//---------------------------------------------------------------------
{
  if (myLora.receive(packetSize, ingoing)) {
    myLora.ClientMapToData(ingoing);
    PrintRec();
  }
}
```

In der Klasse myLora sind folgende Methoden implementiert:

```
//********************************************************************** /
class LoRaWinde
///*********************************************************************/
{
public:
  LoRaWinde();
  void init();
  void sendMessage(String outgoing);
  bool receive(int packetSize, String& ingoing);
  void ClientMapToData(String Record);
  void ServerSendReply(
    String status,
    float VoltageAkt,
    float CurrentMax,
    float TemperaturAkt,
    float Seilspannung);
  void ClientSendMessage(
    String status,
    float Value);
```

Beispiel für die Methode sendMessage:

```
//---------------------------------------------------------------------
void LoRaWinde::sendMessage(String outgoing)
```

```
//---------------------------------------------------------------------
{
  LoRa.beginPacket();             // start packet
  LoRa.write(destination);        // add destination address
  LoRa.write(localAddress);       // add sender address
  LoRa.write(msgCount);           // add message ID
  LoRa.write(outgoing.length());  // add payload length
  LoRa.print(outgoing);           // add payload
  LoRa.endPacket(true);           // finish packet and send it
  msgCount++;                     // increment message ID
}
```