„https://www.rc-modellbau-portal.de/index.php?threads/nrf24l01-2-4-ghz-sende-und-empfangsmodul-f%C3%BCr-arduino.6753/

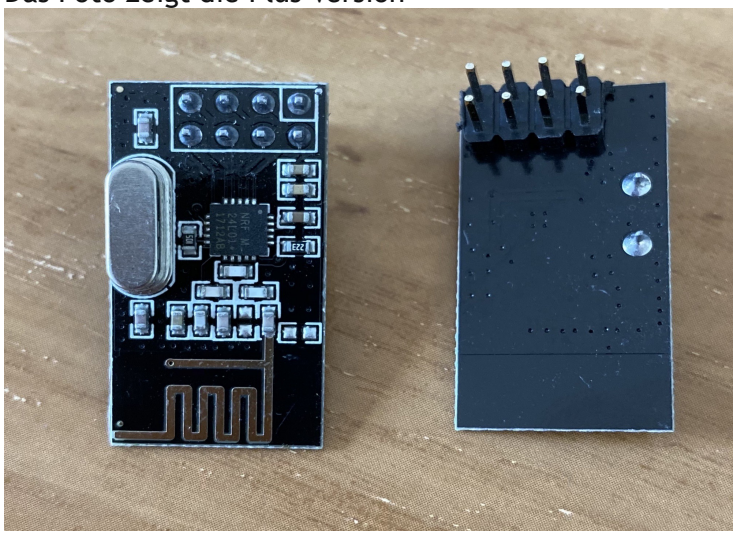
Vorwort / Einführender Text

Das NRF24L01 ist ein 2,4 GHz Sende- und Empfangsmodul, um mit Arduino & Co und Raspberrys usw. Daten drahtlos miteinander austauschen zu können. Es gibt zwar noch 433 Mhz Module zum Datenaustausch, die haben nur leider den Nachteil, dass man zum Senden und Empfangen jeweils ein eigenes Modul benötigt, d.h. für eine bidirektionale Kommunikation muss man jeweils zwei Module an einen Arduino anschließen.

Beim NRF24 benötigt man zu dem Zweck nur ein einziges Modul pro Arduino, weil der NRF24 sowohl senden, als auch empfangen kann. Es gibt zwar auch arduinoähnliche Controllerboards, die bereits ein 2,4 GHz Kommunikationsmodul besitzen (ESP8266), allerdings brauchen diese Module stets ein funktionierendes WLAN mit Adressen, Zugängen, Passwörtern usw.. Das NRF24 ist da etwas einfacher gestrickt. Da müssen nur die beteiligten Kommunikationspartner quasi eine gemeinsame Adresse haben, ohne jegliches WLAN. Man kann auch damit recht komplizierte Netzwerke errichten, bei denen Datenpakete auch über selbst erstellte Knotenpunkte durchgereicht werden können (z.B. zur Reichweitenvergrößerung), aber man muss es nicht und kann alles sehr einfach und übersichtlich halten.

Bei meinen Aktivitäten bin ich auch noch darüber gestolpert, dass es offensichtlich zwei unterschiedliche Module des NRF24L01 gibt, und zwar einmal den "einfachen" NRF24L01 und den NRF24L01+. Der mit dem Plus kann als Sender mit seinem Gegenpart automatisch feststellen, ob ein Datentelegramm auch vollständig und richtig beim Empfänger angekommen ist. Sollte das nicht der Fall sein, wird das Datentelegramm erneut gesendet, ohne dass man selbst irgendeine zusätzliche Zeile Code in sein Programm einbauen muss. Die normalen Module und die Plus Module unterscheiden sich optisch nicht, man muss nur bei der Bestellung darauf achten, dass man auch die + Version bekommt/bestellt.

Das Foto zeigt die Plus Version



Zuletzt bearbeitet: 23. Oktober 2019

BAXL, 21. Oktober 2019

1.Anschluß / Verkabelung

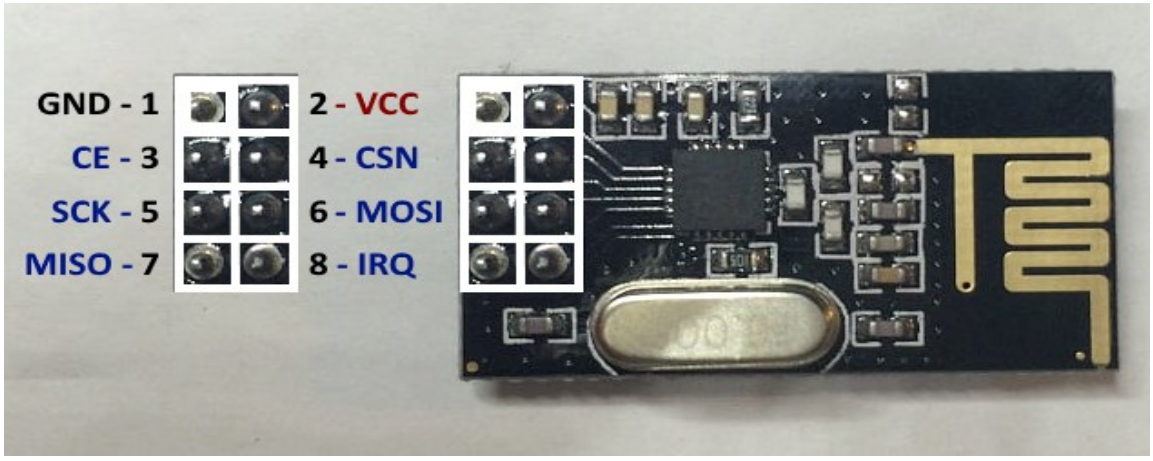
Kommen wir nun zum Hardwareteil. Die Module müssen angeschlossen werden.
Als **Betriebsspannung** benötigen die **NRF24 3,3V**, klemmt man versehentlich 5 V an, kann man das Modul wegschmeißen. Natürlich wird auch Masse benötigt.

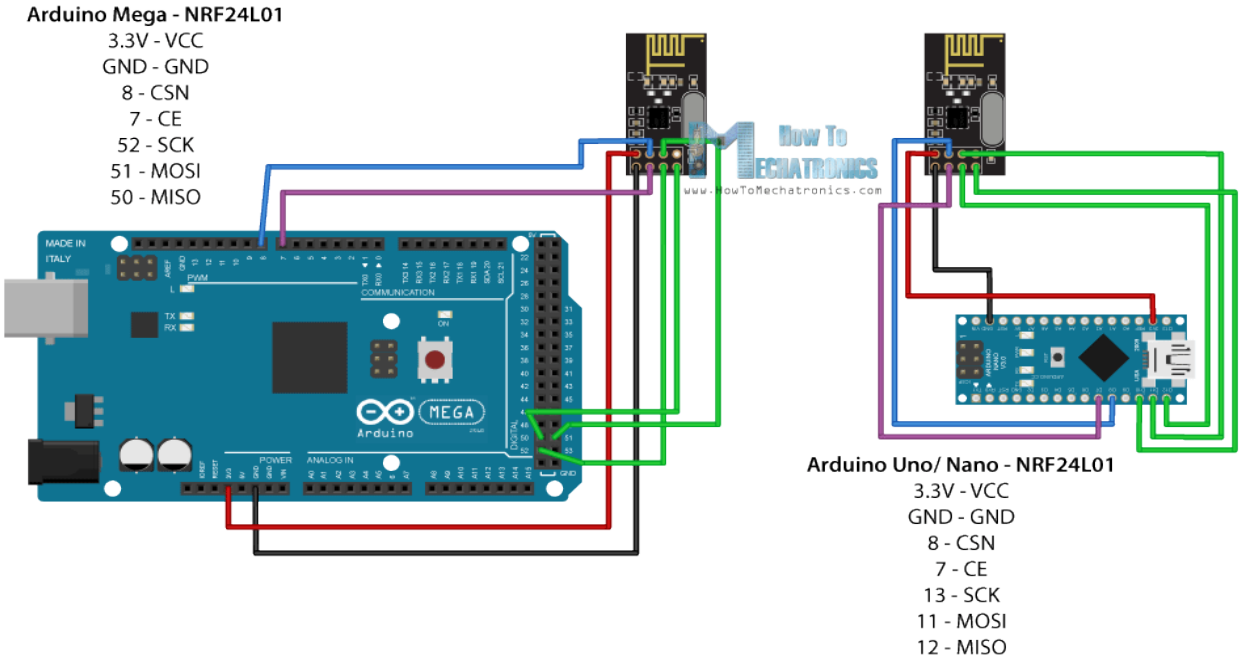
Die 6 Signalpins sind "5V tolerant". Warum schreibe ich das in Hochkommas? Weil dort eigentlich auch 3,3V Signale angelegt werden sollen, das Modul aber auch 5V Signal verträgt. Die Signalpins heißen MISO, MOSI, CSN, CE, SCK und IRQ. Je nach verwendetem Controller können die Pins unterschiedlich angeschlossen sein. Bei den Arduinos (UNO, Nano usw.) liegen die die Signalpins auf CE - D8, CSN - D7, SCK - D13, MOSI - D11, MISO - D12. Beim Arduino Mega sind SCK - D52, MOSI - D51, MISO - D50. Bei einem Raspberry sind die Anschlüsse wieder anders. Darum ist es wichtig die Pinbelegungen und Signalbezeichnungen am NRF24 Modul zu kennen.

(Die Belegungsangaben sind auf die Bilder bezogen, in meinem Projekt verwende ich aber für die Signalpins am Arduino Nano CE - D8, CSN - D9)

!!!! Ganz wichtig ist auch ein 10 myF Elko zwischen der 3,3V Betriebsspannung und Masse!!!

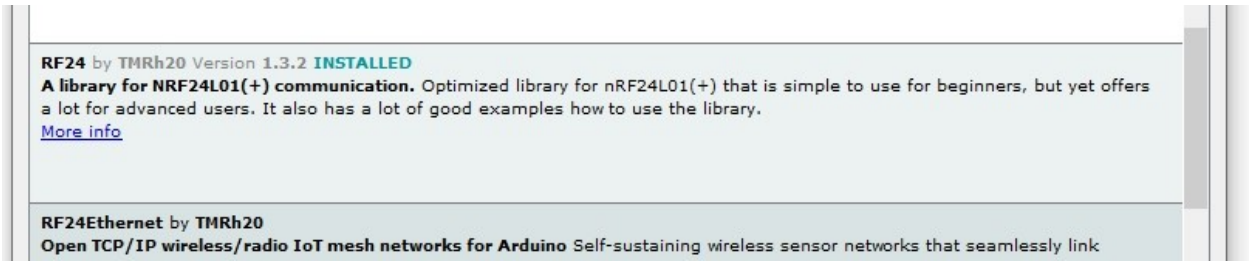
Manchmal findet man auf den Schaltplänen diesen Kondensator, aber ganz oft auch nicht. Warum ist der wichtig? Ganz einfach, die 3,3V Spannungsversorgung der Arduinos ist nicht sonderlich stark, die NRF24 Module benötigen beim Senden durchaus ordentlich Strom. Den Strom kann die 3,3V Versorgung vom Arduino nicht immer liefern, also braucht man den Kondensator als zusätzlichen Strompuffer, der in den Zeiten, wenn nicht gesendet wird, wieder aufgeladen wird.





Achtung, die Pinbelegung in den Bildern kann im Programmlisting abweichen, bitte die Angaben im Listing verwenden!

Edit: für die Bedienung des NRF24l01 Moduls verwende ich folgende Library:



Zuletzt bearbeitet: 29. November 2019

BAXL, 21. Oktober 2019

2.Programmierung / Ansteuerung

Man kann den NRF24 durchaus selbst zu Fuß ansteuern, das ist aber nicht so einfach. Freundliche Zeitgenossen haben für die Ansteuerung bereits passende Programmteile (Funktionen) geschrieben und diese

in eine so genannte Bibliothek gepackt. Bei den Arduinos heißen diese Bibliotheken auch Librarys. Die Bibliotheken sind offen und frei verfügbar.

Für die praktische Nutzung der NRF24 Module findet man Beispielprogramme, wenn man die passende Bibliothek (Library) in seine Arduino Programmieroberfläche einbindet, aber auch auf unzähligen Internetseiten, wo ebenfalls nette Leute mit edlen Vorsätzen Hilfestellungen geben wollen. Leider sind die Beispiele manchmal eigentlich sehr simpel, aber künstlich durch umständliches Zusatzzeugs verkompliziert worden. Eine Erklärung findet man auch nicht jedes Mal. Hinzu kommt, dass scheinbar jeder gewisse Probleme anders löst. Nun gut, dann füge ich die x-te Variante hinzu .

Was man unbedingt machen muss, ist die entsprechenden Librarys einzubinden. Derer gibt es viele. In meinen Beispielen sieht das so aus:

Beispiel für den NRF als Sendemodul

```
// Bibliotheken zur Bedienung des NRF24 Moduls

#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
```

Es müssen auch noch ein paar Variablen bereit gestellt werden

```
// Variablen und Instanzendefinition für das NRF Modul

RF24 radio(8, 9); // CE, CSN - die Zahlen geben die Digitalports am Arduino an, Instanz um das Modul zu starten
const byte address[6] = "00001"; //Adresse, auf dem die Empfangsdaten gesendet werden sollen. Der Empfänger benötigt dieselbe Adresse!
byte EingangsSignal = 0; // Variable, um ein Eingagssignal vom Arduino Port zwischen zu speichern, irgendetwas müssen wir schließlich senden
```

In der Initialisierungsroutine, die auch void setup() heißt, benötigt man folgende Minimalprogrammzeilen

```
radio.begin(); // Start der 2,4 GHz Wireless Kommunikation
radio.openWritingPipe(address); // Setzen der Sendeadresse zur Übermittlung der Daten
radio.setPALevel(RF24_PA_HIGH);
// Leistung des NRF Moduls je nach Entfernung kann man von MIN bis MAX einstellen (MAX,HIGH,LOW,MIN)
radio.stopListening(); // Das angeschlossene Modul wird als Sender konfiguriert

pinMode(7 , INPUT); // Port zum Einlesen des Schalterzustandes/Eingangssignals konfigurieren
```

In der Hauptschleife void loop() sieht es beim Sender so aus:

```
Eingangssignallllllll = digitalRead(button_pin); // Einlesen des Eingangssignals von Pin 7 des Arduinos
radio.write(&EingangsSignal, sizeof(EingangsSignal)); //Senden des Schalterstatus zum Empfänger
```

Das war es auch schon.

Was passiert in meinem Beispiel eigentlich?
Ich habe am digitalen Eingangspin 7 des Arduinos etwas angeschlossen, das ein Eingangssignal dort anlegt, also (0 oder 1) oder (HIGH oder LOW). Das kann ein einfacher Taster sein, aber auch ein PIR Bewegungsmelder, der einen digitalen Ausgang besitzt.

Dieses Signal wird eingelesen und in der Variablen EingangsSignal gespeichert. Diese Information sendet der Arduino permanent per NRF24 aus.

Spoiler: Beispielprogramm Sender

```
// Programm zur Übermittlung eines Schalterzustandes mit einem 2,4 GHz NRF24 Modul
// Bibliotheken zur Bedienung des NRF24 Moduls
#include <SPI.h>
„#include <nRF24L01.h>
```

```
#include <RF24.h>


// Variablen und Instanzendefinition für das NRF Modul
RF24 radio(8, 9);
// CE, CSN - die Zahlen geben die Digitalports am Arduino an, Instanz um das Modul zu starten
const byte address[6] = "00001"; //Adresse, auf dem die Empfangsdaten gesendet werden sollen.
//Der Empfänger benötigt dieselbe Adresse!
boolean button_state = 0;

int button_pin = 7; // Signalpin zum Einlesen des Schaltsignals (Taster, Bewegungsmelder etc.)

void setup() {
Serial.begin(9600); // Start des seriellen Ausgabe per USB an einen PC falls man sich zur
Kontrolle im Programmablauf etwas anzeigen lassen möchte
pinMode(button_pin, INPUT); // Port zum Einlesen des Schalterzustandes konfigurieren
radio.begin(); // Start der 2,4 GHz Wireless Kommunikation
radio.openWritingPipe(address); // Setzen der Sendeadresse zur Übermittlung der Daten
radio.setPALevel(RF24_PA_HIGH);
// Leistung des NRF Moduls je nach Entfernung kann man von MIN bis MAX einstellen
//(MAX,HIGH,LOW,MIN)
radio.stopListening(); // Das angeschlossene Modul wird als Sender konfigurieret
}

void loop()
{

button_state = digitalRead(button_pin); // Einlesen des Schalterzustandes
radio.write(&button_state, sizeof(button_state)); //Senden des Schalterstatus zum Empfänger

delay(50); // kurze Verschnaufpause, damit der Empfänger sich nicht verschluckt
}
```

Zuletzt bearbeitet: 22. Oktober 2019

BAXL, 21. Oktober 2019
#3

mobsy, Crazy virus 4.0 und BlackbirdXL1 gefällt das.

3.Ha, alles logisch, alles einfach! Oder doch nicht?

Was zur Hölle bedeutet die Zeile

radio.write(&EingangsSignal, sizeof(EingangsSignal));

In der Zeile soll also das Eingangssignal übermittelt werden. Klar, steht ja da. Aber was bedeutet in der ach so einfachen Zeile "&EingangsSignal", warum steht da noch ein & (Kaufmanns-Und) und was bedeutet "sizeof(EingangsSignal)", aber diesmal ohne Kaufmanns-Und? Das sind die kleinen Dinger, über die ich immer gestolpert bin.

Die Erklärung:

Wenn das NRF24-Modul Daten übermitteln soll, dann wird die Nutzlast, also die Information "EingangsSignal", in ein Daten-Päckchen eingepackt. Damit das Programm weiß, wie groß das Päckchen werden muß, muß natürlich vorher klar sein, wie groß der Inhalt (Nutzlast) ist. Das findet die Sendefunktion über sizeof(EingangsSignal). "Size" ist Englisch und heißt schlicht "Größe" und "of" heißt "von", also würde der Befehl ins Deutsche übersetzt GrößeVon(EingangsSignal) heißen. Damit sagt man der Sendefunktion also, wie groß die Nutzlast ist.

Kommen wir zu dem Kaufmanns-Und vor "EingangsSignal". Das ist so ein Spezialität von Programmiersprachen, in unserem Fall die Programmiersprache C.
Wenn ich irgendwelche Informationen während eines Programmablaufes speichere, dann landen diese Daten irgendwo im RAM (Arbeistsspeicher) des Controller-Moduls. Damit der Prozessor die Daten dort wiederfinden kann, haben die Speicherzellen so eine Art Hausnummer oder Postleitzahl.

So und jetzt kommt etwas, wo man vielleicht 2, 3, 4 oder auch 5 mal lesen muß um es endlich zu verstehen.
(Ging mir auch so)

Die Programmfunktion radio.write() bekommt zwei Informationen. Einmal einen Hinweis, **WO** die Information "EingangsSignal" zu finden ist (Das symbolisiert das & in "&EingangsSignal"), die übermittelt werden soll und über den Umweg des Funktionsaufrufes sizeof(), wie groß der Inhalt ist. Mit Größe ist gemeint, wieviel Platz die Information in dem Datenpäckchen belegt.
Soll heißen, ob die Information in ein Byte, zwei Bytes oder auch mehr Bytes gesteckt werden muss. In einem Byte kann man nämlich nur Zahlen von 0 bis 254 stecken.

Es wird dem Programm per & also nicht die Information selbst gesagt (z.B. 1 oder 3 oder 378 oder auch "Hallo"), sondern wo diese Information zu finden ist. dabei ist die Bezeichnung hinter dem & eigentlich so eine Art Hausnummer oder Postleitzahl. Die Programmierer sagen dazu Zeiger, es wird dem Programm ein Zeiger, also die Hausnummer, auf die Variable, bzw. dem Inhalt der Variablen übergeben. Das Programm weiß dann, in der Hausnummer (Speicherzellenadresse) finde ich den Zahlenwert, den ich bearbeiten, bzw. wie in unserem Beispiel, per NRF24 verschicken soll.

Zuletzt bearbeitet: 13. November 2019

BAXL, 21. Oktober 2019

4.Doch der Reihe nach.

Wir haben also die Librarys SPI.h, nRF24L01.h und,RF24.h eingebunden. Für die verwendete Library siehe Startpost #1.

Als Erstes erstelle ich ein ein neues Objekt, oder auch Instanz genannt, über das die Programmfunktionen der Librarys aufgerufen werden können. Ich könnte ja theoretisch auch zwei NRF24 Module anschließen und bedienen. Dann muß dem Programm ja klar sein auf welches der beiden Module sich der jeweilige Befehl bezieht.
Weil ich in meinem Beispiel nur ein NRF24-Modul anschließe, nehme ich als Name für dieses Modul (Objekt) einfach nur radio. Ich könnte das auch "Sender", "NRFModul", "Schnudelhupf" oder sonstwie nennen. Hätte ich zwei NRF24 angeschlossen, bräuchte ich zwei Objekte, die ich radio1 und radio2 nennen könnte. Je nachdem, welches der beiden NRF Module ich dann bedienen möchte, muß ich beim Funktionsaufruf den Objektnamen davor setzen. Z.B. radio1.Funktionsaufruf, oder radio2.Funktionsaufruf. Es wird dann in der Bibliothek derselbe Code benutzt, hat aber nur Auswirkung auf das NRF24 Modul, dessen Name vor dem Punkt steht.

Ich sage dem Programm also, ich habe ein Modul und das soll "radio" heißen und über die Arduinopins 8 und 9 angesteuert werden.
Das mache ich mit dem Befehl:

```
RF24 radio(8, 9);
```

In der Library sollen alle Funktionen, die irgendwie RF24 heißen, von dem Modul mit Namen radio aufgerufen werden können.

In der Setuproutine gehts dann weiter. Mit `radio.begin();` werden einige weitere versteckte Funktionen der Library aufgerufen, damit der gesamte Mechanismus gestartet wird.

Weil man mehrere NRF24 parallel in einem Netzwerk betreiben kann, brauchen die für den Datenweg quasi jeweils ein eigenes Gleis. Dieses Gleis "baut" man mit dem Befehl:

`radio.openWritingPipe(PipeAdresse);` Für PipeAdresse setzt man einen Wert ein, der 5 Bytes lang ist. Das kann direkt ein hexadezimaler Wert wie 0xFF00FF00FB sein, oder man nimmt schlicht einen String mit 5 Zeichen,

wie eben "00001". Man kann zum Schreiben immer nur gleichzeitig ein Gleis (Pipe) nutzen, darum braucht nur die PipeAdresse eingesetzt werden.

Möchte man mehrere Gleise gleichzeitig mit einem Controller (Arduino als Empfänger) **lesen**, brauchen die eine Nummerierung (PipeNummer), die von 0 bis 5 gehen kann. Dann sieht natürlich auch der o.g. Befehl etwas anders aus, nämlich

```
radio.openReadingPipe(PipeNummer, PipeAdresse);
```

Für jede PipeNummer muß dann auch die jeweilige, gewünschte PipeAdresse eingetragen werden. Es rollen also auf mehreren Gleisen (Pipeadressen) Daten ein, die ich in einem Sackbahnhof auf Zwischengleise (PipeNummer) umleite und nacheinander abholen kann.

Je nachdem, welche Strecken überwunden werden müssen (Entfernung, Hauswände usw.), kann man die Sendeleistung einstellen. Das geht in mehreren Stufen.
Dazu dient der Befehl:

```
radio.setPALevel(RF24_PA_HIGH); // Leistung des NRF Moduls je nach Entfernung kann man von MIN bis MAX einstellen (MAX,HIGH,LOW,MIN)
```

Und jetzt kommt ein Befehl, der darüber entscheidet, ob das Modul als Sender, oder als Empfänger arbeiten soll.

```
radio.stopListening(); // Das angeschlossene Modul wird als Sender konfiguriert
```

oder

```
radio.startListening(); // Das angeschlossene Modul wird als Empfänger konfiguriert
```

Die letztgenannten Befehle kann man beliebig in einem Programmablauf aufrufen, je nachdem, ob man gerade senden, oder empfangen möchte.

Tja und im eigentlichen Programm taucht dann der Sende, oder Empfangsbefehl auf, nämlich

```
radio.write(&Information, sizeof(Information));
```

oder

```
radio.read(&Information, sizeof(Information));
```

Der Empfänger benötigt noch zusätzlich die Funktion

```
radio.available()
```

um die Information zu erhalten, ob tatsächlich Daten eingegangen sind und im Empfangspuffer lagern.

Das passende Beispielprogramm für den Empfänger :
Spoiler: Empfängerprogramm

```
// Bibliotheken für 2,4 GHz Modul
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

RF24 radio(8, 9); // CE, CSN
const byte address[6] = "00001";
boolean button_state = 0;
int led_pin = 7;


void setup() {
```

```
pinMode(7, OUTPUT);
Serial.begin(9600);
radio.begin();
radio.openReadingPipe(0, address); //Setting the address at which we will receive the data
radio.setPALevel(RF24_PA_MIN);
//You can set this as minimum or maximum depending on the distance between the transmitter
//and receiver.
radio.startListening(); //This sets the module as receiver
Serial.println("gestartet");
}

void loop()
{
if (radio.available()) //Looking for the data.
{
Serial.println("empfange!");
radio.read(&button_state, sizeof(button_state)); //Reading the data
if(button_state == HIGH)
{
digitalWrite(7, HIGH);
}
else
{
digitalWrite(7, LOW);
}

}
delay(50);
}
```

Zuletzt bearbeitet: 29. November 2019

BAXL, 21. Oktober 2019

5.Mehrere Informationen senden

Wenn man sich für eine Wetterstation, oder Hausüberwachung gleich mehrere Werte übermitteln möchte (Raumtemperatur, Helligkeit, Luftfeuchte, Bewegung, usw.), kann man natürlich für jeden dieser Werte einen eigenen radio.write() Befehl aufrufen und die Messdaten nacheinander verschicken. Das funktioniert sogar, man muss die Werte dann allerdings auch genau in der gesendeten Reihenfolge am Empfänger einsammeln. Die Werte können sogar einen unterschiedlichen Datentyp haben. Mal nur Byte, wenn man als Bewegung ja und nein hat, also 0 und 1, oder Gleitkomma (float), wenn es eine Temperatur sein soll usw..

Das habe ich auch so gemacht und irgendwie hat es auch mehr schlecht als recht funktioniert, weil der Sender immer ins Straucheln kam (man erinnere sich an den Elko als Strompuffer). Am Empfänger kamen regelmäßig verstümmelte Daten an. Nach laaaangem Suchen bin ich dann über die Lösung gestolpert. Man baut sich so eine Art Datencontainer, in dem mehrere Daten verpackt werden, allerdings hübsch der Reihe nach, sowohl beim Sender, als auch beim Empfänger. Dadurch ist es möglich alle Daten mit einem Sendevorgang zu übermitteln, wodurch die Stromversorgung im Ergebnis weniger belastet wird und jeweils nur ein Übermittlungsvorgang erforderlich ist.

Das Zauberding nennt sich "struct". Soll wohl soviel wie Struktur heißen. Man baut also den Container und sagt, was da rein soll. Mein Containertyp heißt Wert, ich hätte ihn ebenso container oder Kiste nennen können. "Wert" ist aber nur eine Art Blaupause oder Muster. In meinem Beispiel stecken darin, ob eine Bewegung von einem Bewegungssensor erkannt wurde "Event", eine Temperatur "Temp" und der gemessene Helligkeitswert "Lux".

```
struct Wert {byte Event; int Temp; int Lux;} Messung;
```

Mit struct Wert{...} lege ich nur die innere Struktur des Datencontainers fest. Damit ich diese Struktur auch

verwenden kann, muss ich einen echten Datenbehälter benennen und der heißt in meinem Beispiel "Messung". Auch hier könnte ich gleich mehrere Datenbehälter bereitstellen, die genau die innere Struktur von Wert haben. Dann könnte die o.g. Programmzeile auch so aussehen:

struct Wert {byte Event; int Temp; int Lux;} MessungBad, MessungWohnzimmer, MessungSchlafzimmer;

Es können also durchaus auf einem Rutsch mehrere benutzbare Datenbehälter erstellt werden.

Das könnte für das Empfängerprogramm wichtig sein, falls ich von mehreren Sendestationen Daten empfangen und anzeigen möchte.

Damit ist dann auch klar, dass beim Sender und beim Empfänger jeweils der gleiche Containertyp mit exakt demselben Inhalt definiert werden muss. Ich kann mir ja auch nicht vom Postboten 10l Öl liefern lassen und das in einen 5l Kanister umfüllen wollen .

Wenn ich dann den Sende-, oder Empfangsbefehl schreibe, dann steht da:

```
radio.write(&Messung, sizeof(Messung)); // Sendebefehl
```

oder

```
radio.read(&Messung, sizeof(Messung)); // Empfangsbefehl
```

Das hat jetzt den Vorteil, dass alle Messdaten auf einen Rutsch gesendet werden und der NRF24 nicht mehrmals Energie benötigt (Wieder Stichwort Pufferelko).

Ich wiederhole das gebetsmühlenartig, weil ich damit immer wieder so meine Problemchen hatte. Endgültige Abhilfe schafft da wirklich nur eine eigene, leistungsfähigere 3,3V Stromversorgung (min. 150mA) oder eine Zusatzplatine für NRF24l01 Module, die direkt an die leistungsfähigere 5V Versorgung eines Arduinos angeschlossen wird. Diese Platine hat einen 3,3V Spannungsregler an Bord, um dem NRF24 Modul die 3,3V bereitzustellen.

Bei der Verwendung einer getrennten 3,3V Versorgung sollte man nicht vergessen, die Massen miteinander zu verbinden!

Zuletzt bearbeitet: 29. November 2019

BAXL, 21. Oktober 2019

6.Neues Problem - Datentypen und Speicher

Wie wir wissen, benötigen die unterschiedlichen Datentypen unterschiedlich viel Speicherplatz, oder auch Platz im Datentelegramm.
Wenn ich z.B. eine Temperatur mit einem DS18B20 Temperatursensor messe, dann liefert mir der DS18B20 eine Kommazahl (float) zurück. Ich muß im Programm den Messwert also in einer Variaben speichern, die den Datentyp float hat. Der Datentyp float belegt aber mal eben 32 Bits, macht 4 Byte, obwohl der DS18B20 eigentlich nur 2 Nachkommastellen liefert.

Vom reinen Zahlenwert würde die gemessene Temperatur auch locker in zwei Bytes passen. Darum greife ich zu Trick 17, auch im Hinblick auf die Datenübertragung. Würde ich drei Temperaturen als float verschicken, bräuchte ich schon 12 Bytes. Je länger ein Datenpaket ist, umso größer ist auch die Wahrscheinlichkeit von Übertragungsfehlern.Außerdem ist die Gesamtgröße eines Datentelegramms auf 32 Byte begrenzt.

Also mache ich Folgendes. Weil der DS18B20 nur zwei Nachkommastellen liefert, multipliziere ich den gemessenen Wert einfach mit 100 und habe keine Nachkommastellen mehr. Dann wandele ich den Wert in den Datentyp Integer (int) um, der nur 2 Bytes Speicher benötigt .

Wenn ich die Daten verschickt habe, teile ich den empfangenen Wert durch 100 und mache wieder ein float daraus.

Achtung Stolperfalle!!!

Wenn ich beim Empfänger die Rechenoperation: Temperatur = EmpfangenerWert / 100 mache, dann verschwinden plötzlich die Zahlenwerte der zwei Nachkommastellen und werden einfach zu Null.
Das ist doof und hat mich viel Kopfschmerzen gekostet!!! Irgendwo in den Tiefen des Internets habe ich herausgefunden, dass man durch einen kleinen Trick den Datentyp float erzwingen mussMan darf also die Rechenoperation nicht wie oben schreiben sondern so:

Temperatur = EmpfangenerWert / 100.0

Man achte auf die zusätzliche Null hinter dem Komma?, nein das ist ein Punkt, weil in der Arduino IDE Kommazahlen mit einem Punkt geschrieben werden müssen. danach klappte die Sache.

Kleiner Nebenhinweis zum Datetyp float.

Gleitkommazahlen können ja bis ins Unendliche hinter dem Komma gehen. Der Prozessor kann aber nicht bis dahin auflösen. Es werden bei den Berechnungen also immer Rundungsfehler auftauchen und die Zahl Null ist oft nicht wirklich exakt 0, sondern kann irgendwo ganz Hinten noch irgendeinen klitzekleinen Wert beinhalten. Vereinfacht gesagt kann 0 auch 0,0000000000000001 sein. Vergleicht man also einen ausgerechneten Wert mit einer Zahl, so schreibt man nicht
if (Rechenwert == Vergleichzahl) {..weitere Befehle... }, sondern
if (almostequal(Rechenwert,Vergleichzahl)){..weitere Befehle... } .

Sowas weiß normalerweise kaum einer.

7.Einen habe ich gerade noch, die Übertragungsgeschwindigkeit. Auch hier wissen wir, je schneller es geht, umso kleiner sind die Abstände zwischen den Datenflanken und umso eher kann es zu Übertragungsfehlern kommen. Einfacher Trick, ich reduziere einfach die Geschwindigkeit. Die Datenrate kann bis 2MBit / s gehen. Das kann man am NRF24 einstellen. Die langsamste Geschwindigkeit beträgt 250kB / s. Also setze ich die Geschwindigkeit in der Setuproutine einfach runter mit:

```
radio.setDataRate(RF24_250KBPS);// Übertragungsgeschwindigkeit setzen RF_1MBPS, RF_2MBPS
```

Die Übertragung läuft dann immer noch schnell genug ab.

Das ist erst Mal in Kürze das Wichtigste. Als Nächstes kommt das Problem mit mehreren Sendern und einem Empfänger, aber da kämpfe ich noch.

BAXL, 21. Oktober 2019

8.**Neue Erkenntnisse**

Mir ist es in den obigen Posts leider auch passiert, dass ich quasi blind abgeschrieben habe, weil ich gewisse Dinge noch nicht gepeilt hatte. Es kann äußerst hilfreich sein, sich die Dokumentation der verwendeten Librarys anzusehen.

Dokumentation RF24 Library

Plötzlich erkennt man, warum manche Befehle wie aufgerufen werden müssen und vor allen Dingen, welche Daten man überhaupt eintragen kann/darf. Speziell bei der Anzahl der gleichzeitig verwendbaren Pipes in einem Programm hatte ich irrtümlich 127 angenommen, weil ich das mit dem Befehll setChannel() verwechselt habe, bei dem man tatsächlich eine Zahl zwischen 0 und 127 einsetzen kann/darf. Bei der Anzahl der Pipes geht aber nur eine Zahl von 0 bis 5, was unterm Strich 6 Pipes bedeutet, wobei die Pipe 0 auch noch gewisse Besonderheiten hat, die ich noch verstehen muß.

Der Befehl setChannel() definiert tatsächlich, auf welcher 2,4GHz- Frequenz gesendet werden soll. Es steht normalerweise ein Frequenzband von (2,4 GHz) 2400MHz bis 2484MHz zur Verfügung. Wobei der Channel in setChannel() nichts mit den 2,4GHz Kanälen zu tun hat. In der Regel überspannt ein NRF-Kanal drei 2,4GHz Kanäle, so umfasst z.B. der "NRF24 Kanal 1" die Frequenzen von 2401MHz bis 2123 MHz, was den "2,4GHz Kanälen" 1 bis 3 entspricht.

0 => 2400 Mhz (RF24 channel 1)
1 => 2401 Mhz (RF24 channel 2)
76 => 2476 Mhz (RF24 channel 77)
83 => 2483 Mhz (RF24 channel 84)
124 => 2524 Mhz (RF24 channel 125)
125 => 2525 Mhz (RF24 channel 126)

Channel	Frequency (GHz)	Range	Channel Range
1	2.412	2.401 -2.423	1 - 3
2	2.417	2.406 - 2.428	1 - 4
3	2.422	2.411 - 2.433	1 - 5
4	2.427	2.416 - 2.438	2 - 6
5	2.432	2.421 - 2.443	3 – 7
6	2.437	2.426 - 2.448	4 – 8
7	2.442	2.431 - 2.453	5 – 9
8	2.447	2.436 - 2.458	6 - 10
9	2.452	2.441 - 2.463	7 – 11
10	2.457	2.446 - 2.468	8 – 11
11	2.462	2.451 - 2.473	9 - 11
12	2.467	2.456 - 2.478	Not US
13	2.472	2.461 - 2.483	Not US
14	2.484	2.473 - 2.495	Not US

Wozu braucht man das?

Klare Kiste, man hat im Haus mittlerweile WLAN und andere Geräte, die auf 2,4GHz Kanälen senden, unglücklicherweise kommen auch noch die WLANs der Nachbarn dazu, die ebenfalls bis in die eigenen 4 Wände blasen. Mit den WLAN Routern kann man, je nach Firmware, auch die Kanalbelegungen des Umfeldes anzeigen lassen. Es ist nun die Frage, ob man vorzugsweise die selbstgenutzten Kanäle zusätzlich mit den NRF24 belastet, oder ob man einen Kanal auswählt, den der Nachbarrouter gerne nutzt .

9. Zugelassene Kanäle/Frequenzen und Sendeleistung

Bei den Kanälen sollte man aber etwas vorsichtig sein, man darf in Deutschland leider nicht den komplett möglichen Frequenzbereich ausschöpfen, dafür gibt es natürlich wieder gesetzliche Vorschriften. Bei der Bundesnetzagentur bin ich fündig geworden. Eine Übersicht des Erlaubten Frequenzbereiches findet sich in diesem Dokument. Für den privaten Bereich freigegeben sind die Frequenzen 2400,0 –2483,5 MHz, sodass beim NRF Kanal 14 Schluss ist. Natürlich gelten auch die 0,1W = 100 mW Sendeleistungsbeschränkungen, was +20dBm entspricht.

dBm	mW	dBm	mW	dBm	mW
-30	0,0010	-10	0,1000	12	15,8489
-28	0,0016	-8	0,1585	14	25,1189
-26	0,0025	-6	0,3000	16	39,8107
-24	0,0040	-4	0,3981	18	63,0957
-22	0,0063	-2	0,6310	20	100,0000
-20	0,0100	0	1,0000	22	158,4893
-18	0,0158	2	1,5849	24	251,1886
-16	0,0251	4	2,5119	26	398,1072
-14	0,0398	6	3,9811	28	630,9573
-12	0,0631	8	6,3090	30	1000,0000
		10	10,000		

Die NRF24 haben
Sensor PA Level == RF24_PA_MIN entspricht -18dBm, RF24_PA_LOW entspricht -12dBm, RF24_PA_HIGH entspricht -6dBM, und RF24_PA_MAX entspricht 0dBm

0dBm = 1 mW, was auch der höchsten Sendeleistung der NRF24 Module entspricht. Wir bewegen uns damit

also auf der sicheren Seite.

Wenn es aber etwas mehr sein darf, dafür gibt es NRF24 Module mit einer richtigen Antenne und nicht nur die futzelig kleine Leiterbahn auf dem Chip. Diese Teile schaffen es bis +20dBm, was dann der maximal erlaubten 100mW entspricht. Die genehmigen sich in der Spitze beim Senden bis zu 115mA und selbst im Empfangsbetrieb fließen gerne 45mA.

Nur wird man die auf keinen Fall mehr problemlos am 3,3V Ausgang eine Arduinos betreiben können. Der 3,3V Arduinoausgang liefert maximal 50mA! Für solche Zwecke kann man aber auf **spezielle Adapterplatinen** im Zubehör zurückgreifen, die eine separate 5V Spannungsversorgung haben und diese auf die 3,3V herunterregulieren. Wichtig ist, dass die angeschlossenen 5V einen ordentlichen Strom liefern können, am besten bis zu 250 mA.

Zuletzt bearbeitet: 29. November 2019

BAXL, 23. Oktober 2019

10.Daten von mehreren Sendern empfangen

In den Vorposts wurden bereits einige Dinge beschrieben, die man benötigt, um auf einem Empfänger Daten von mehreren Sendern empfangen und verarbeiten zu können.
Leider findet man meist nur Beispiele, die gewisse Probleme immer nur separat, aber nicht in einem Gesamtpaket beschreiben bzw. lösen.

Bevor ich jetzt einfach nur einen fertigen Programmcode hier hineinkopiere, sind ein paar Erklärungen notwendig.
Wir wollen also von mehreren Sendern jeweils mehrere Daten, an einen einzigen Empfänger schicken.Was ist also das Problem?

Problem 1: Wenn ich mehrere unterschiedlich Werte von einem Sender schicken will, dann ist es unpraktisch, wenn ich jeden Wert einzeln auf die Reise schicke. Besser ist es, wenn ich gleich mehrere Werte in ein Datenpaket packe und dann übermittel.

Problem 2: Wenn mehrere Sender Daten zur Verfügung stellen wollen, dürfen die sich nicht gegenseitig stören, darum erhält jeder Sender einmal eine Adresse und zusätzlich wird im Empfänger noch eine weitere "Datenzwischenspeicherstation" (Pipe) reserviert.

Problem 3: Wenn im Empfänger mehrere Sender ihre Daten abladen, müssen diese für die Weiterverarbeitung getrennt aus den "Datenzwischenspeicherstation" abgeholt und dann in getrennten Verarbeitungsspeichern abgelegt werden.

Lösung Problem 1:
Mehrere einzelne Daten packt man in einen Container, den man mit

struct ContainerName { datentyp Wert1, datentyp Wert2, datentyp Wert3...}
NameEchterContainer;

definiert.

Nehmen wir als Beispiel eine Temperatur, eine Zahl und einen Schaltzustand, dann könnte es etwa so aussehen

struct Messwerte { float temp, int Zahl, byte SchaltZustand} MessungenWohnzimmer;

Die einzelnen Werte werden dann mit

MessungenWohnzimmer.temp = 25.85;
MessungenWohnzimmer.Zahl = 1376;

MessungenWohnzimmer.Schaltzustand = 1;

zugewiesen, bzw. aus Messsensoren gewonnen.

Lösung Problem 2:

Die Adresse für jeden Sender und seinem Sendekanal (nicht der 2,4GHz Kanal!) vergibt man einen fünfstelligen Wert, der so gewählt werden soll, dass bei der Übertragung möglichst viele auf- und absteigende Flanken bei den Datenbits entstehen. Dadurch soll die Übertragung selbst wohl zuverlässiger sein. Gerne werden solche Konstrukte gewählt (in hexadezimaler Form):

0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL, 0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL

Die jeweils vor den Zahlen stehenden 0x sagen dem System nur, dass jetzt eine Zahl in hexadezimaler Schreibweise folgt. Die beiden LL dahinter zeigen nur an, dass es sich um den Datentyp long long handelt. Der eigentliche Wert wäre dann so z.B.:
Aus 0xF0F0F0F0D2LL wird als Wert F0 F0 F0 F0 D2

Noch ein wichtiger Hinweis zur Auswahl der Adressen. Bei einem Empfänger mit mehreren Pipes (Datenkanälen) müssen die ersten 4 Adressbytes identisch sein, die Unterscheidung der einzelnen Datenkanälen erfolgt im 5ten und letzten Adressbyte. Das gilt ebenfalls für den fall, dass ein Sender auch als Empfänger fungieren soll, auch da müssen die ersten 4 Adressbytes identisch sein.

Achtung! Es können maximal 6 unterschiedliche Datenkanäle vom Empfänger gehändelt werden.

Der Sender benötigt natürlich nur eine einzige Kanalangabe, es sei denn, der Sender soll ebenfalls etwas empfangen können, doch das ist ein anderes Thema für sich. Ich beschränke mich aktuell auf nur eine Datenrichtung.

Die Kanalangabe beim Sender und beim Empfänger muss natürlich identisch sein. Und weil der Empfänger eben mehrere Datenkanäle bedienen kann, müssen ihm auch mehrere Empfangsadressen angegeben werden.

Die Empfangsadressen packt man vorzugsweise beim Empfänger in ein Datenfeld (Array), die man über einen Index aufrufen kann.

Beispiel:

static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL, 0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL};

pipe[] ist lediglich der Name des Adressarrays, er könnte auch datenKanal[], KanalAdresse[] oder so ähnlich heißen. Der Einfachheit halber und um Fehler auszuschließen, setze ich das Datenarray auch genau so beim Sender und rufe nur die gewünschte Adresse ab.

z.B:

pipe[0] enthält die Adresse 0xF0F0F0F0E1LL, pipe[4] die Adresse 0xF0F0F0F0A5LL.

Achtung, bei den Datenarrays fängt man immer mit dem Nullten Datenelement an zu zählen! D.h., die erste Datenkanaladresse wäre pipe[0].

Im Empfänger landen die anfangs beschriebenen Wertecontainer in den Zwischenspeichern von pipe[0] bis pipe[5].

Wenn jetzt von einem Sender Daten übermittelt werden, landen die in der passenden Pipe, aus der man sie abholen muss. Es findet eigentlich eine Zuordnung/Verbindung zwischen der längeren Datenkanaladresse zu einer Pipe statt, die nur ganz einfach von 0 bis 5 durchnummeriert ist. Diese Verbindung muss man mit dem Befehl

radio.openReadingPipe("Nummer der Pipe von 0 bis 5", "5 Byte lange Datenkanaladresse");

Das könnte so aussehen:

```
radio.openReadingPipe( 1, 0xF0F0F0F0E1LL);
```

oder so

```
radio.openReadingPipe( 4, "00003");
```

oder so (falls man die Adressen in einem Array abgelegt hat und die Nummer des Clienten im Vereinbarungsteil definiert):

```
static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL,
0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL};

radio.openReadingPipe( ClientNummer, pipe[pipeNr]);
```

Weil die Sender nun nicht permanent Daten senden, sollte man im Programm einen Hinweis bekommen, wann Daten eingegangen sind. Dazu dient der

Befehl: `radio.available(&ClientNummer);`

dessen Rückgabeergebnis nur Wahr oder Falsch sein kann. Bekommt man ein "Wahr", dann liegen Daten bereit und ClientNummer sagt, in welcher Pipe die Daten liegen, man kann also zuordnen, von welchem Sender die Daten kommen, damit ist man in der Lage, die Daten zu separieren.

Lösung Problem 3:

Weil wir nun in den verschiedenen Pipes beim Empfänger Daten erhalten, müssen wir die abholen. Zweckmäßig ist es dazu, die passende Pipe, also auch die zugehörige Adresse anzugeben.

In meiner praktischen Anwendung, die ich später noch genauer vorstellen werde, sollen vier Messwertgruppen auf einem Display angezeigt werden. Dafür dienen die vier Zeilen auf dem 2004 LCD Display. Das sind die Informationen Temperatur, Lichtstärke und ob eine Bewegung erkannt wurde. D.h. ich möchte in Zeile 1 die Daten aus dem Wohnzimmer, in Zeile 2 aus meinem Arbeitszimmer, Zeile 3 Kinderzimmer und Zeile 4 das Badezimmer anzeigen lassen.

Als Zeilenangabe kann ich damit die Angabe ClientNummer verwenden. Im Moment werte ich noch nicht aktiv aus, sondern zeige nur an. Eine zusätzliche Zwischenspeicherung der Messwerte entfällt also aktuell. Später, wenn ich herausfinden Möchte, ob ein offenes Fenster im Bad vergessen wurde, muss ich die abgeholten Werte noch in einem zusätzlichen Datenfeld (Array) zwischenspeichern.

11.Hier kommt jetzt erst der versprochene (funktionierende) Code aus meiner noch recht einfach gestrickten Anwendung. Zuerst der Sender.

Am Sender ist jeweils ein DS18B20 Temperatursensor, ein PIR Bewegungsmelder und eine Fotozelle angeschlossen.
Der DS18B20 Temperatursensor liefert mit Temperaturen mit zwei Nachkommastellen, also eine Gleitkommazahl oder auch float genannt.
Der Bewegungsmelder liefert nur den digitalen Wert 0 oder 1, den ich in ein Byte (Datentyp byte) packe.
Die Fotozelle kann Zahlen zwischen 0 und 1023 durch die Messung an einem analogen Eingang erzeugen, für die ich den Datentyp Integer int verwende.

Diese Werte packe ich in ein Datenfeld mit dem Namen Messung über die Vereinbarung

```
struct Wert {byte Event; int Temp; int Lux;} Messung;
```

Event ist dann die erkannte Bewegung des PIR, Temp, die Temperatur des DS18B20 und Lux, der gemessene Wert von der Fotozelle.

Als nächstes benötige ich das Datenfeld mit den Kommunikationsadressen, dass ich über den folgenden Befehl erstelle:

```
static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL, 0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL};
```

Als kleinen Trick, um mir später bei mehreren Sendemodulen mit unterschiedlichen Senderadressen das Programmieren zu erleichtern, definiere ich einfach eine Variable ClientNummer mit:

```
byte ClientNummer = 3; // Mögliche Werte: 1-6
```

Den Wert ändere ich manuell bevor ich einen Arduino als Sender brenne (programmiere). **Es dürfen nicht mehrere Sender dieselbe ClientNummer haben!**

Hier der komplette Code für den Vereinbarungsteil:

Code:

```
// Programm zur Übermittlung eines Schalterzustandes und weiterer Messwerte mit einem 2,4 GHz NRF24 Modul

// Die Betriebsspannung vom NRF24 Modul MUSS!! an 3,3V vom Arduino angeschlossen werden
// ACHTUNG!!! ca 10 nF Elko zwischen 3,3V und Masse schalten um die Übertragung zu stabilisieren

// Bibliotheken zur Bedienung des NRF24 Moduls
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

struct Wert {byte Event; int Temp; int Lux;} Messung;

// Einbinden eines Dallas DS18B20 Temperatursensors
#include <DallasTemperature.h>      // Library für Dallas Temperatursensoren
#define ONE_WIRE_BUS 5
// der Eindrahtbus für die DS18B20 liegen auf Digitalport 5
#define DS18B20_Aufloesung 10
// Die Sensoren sollen eine 10 Bit Auflösung liefern

OneWire oneWire(ONE_WIRE_BUS);
// Erstellen einer OneWire Instanz für den DS18B20 Temperatursensor
```

```
DallasTemperature myDS18B20(&oneWire);
DeviceAddress tempDeviceAddress;
// temporäre Adresse um die Auflösung des DS18B20 setzen zu können
// keine Ahnung warum das so ist, aber es funktioniert

// Variablen für die Temperaturmessung und -übertragung
float Raumtemperatur;
// Variablen für die Temperaturwerte der Messstellen definieren
// die Raumtemperatur float wird zuert mit 100 multipliziert und zum Integer gemacht
int intTemperatur = 25;
// Variablen für die Temperaturwerte als Integer

int LichtWert;
// Werte zwischen 0 und 1024, 0 = ganz hell, 1024 total dunkel
// Lampenlich 250  bis 500, Tageslicht kleiner 250, kein Licht größer 500
byte LichtSensor = 4;

// Variablen und Instanzendefinition für das NRF Modul
RF24 radio(8, 9);                 // CE, CSN - die Zahlen geben die Digitalports am
Arduino an,  Instanz um das Modul zu starten
static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL,
0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL};
byte ClientNummer = 3; // Mögliche Werte: 1-6

const byte address[6] = "00002";
//Adresse, auf dem die Empfangsdaten gesendet werden sollen. Der Empfänger benötigt
//dieselbe Adresse!
byte button_pin = 7;
// Signalpin zum Einlesen des Schaltsignals (Taster, Bewegungsmelder etc.)
byte Signal = 0;             // Ausgabewert Signalpin landet in Signal
byte merker = 0;

// Variablen zur Zeitmessung ohne delay
  unsigned long MessIntervallms = 5000;   // Messintervall in Millisekunden
  unsigned long startzeit;
// Merker für die Systemzeit beim Eintreten einer Bewegung
  unsigned long vergangene_zeit;
// ausgerechneter Wert zwischen Merker und aktueller Systemzeit

  unsigned long SendeIntervallms = 10000; // Sendeintervall in Millisekunden
  unsigned long startzeitSI;
// Merker für die Systemzeit beim Eintreten einer Bewegung
  unsigned long vergangene_zeitSI;
// ausgerechneter Wert zwischen Merker und aktueller Systemzeit
```

Kommen wir zur Setuproutine

Zuerst muß natürlich das Sendemodul aktiviert werden mit:

radio.begin();

Zur Sicherung der Kommunikation reduziere ich im Setup die Übertragungsgeschwindigkeit auf 250kB pro Sekunde und erhöhe die Sendeleistung auf Maximum mit den Befehlen:

radio.setDataRate(RF24_250KBPS);// Übertragungsgeschwindigkeit setzen RF_1MBPS, RF_2MBPS
radio.setPALevel(RF24_PA_HIGH); // Leistung des NRF Moduls je nach Entfernung kann man von MIN bis MAX einstellen (MAX,HIGH,LOW,MIN)

Ganz wichtig, jetzt muss der Datenkanal festgelegt werden. Das mache ich über das bereits definierte Adress-Array und die gewünschten ClietNummer.

radio.openWritingPipe(pipe[ClientNummer-1]); // Setzen der Sendeadresse zur Übermittlung der Daten

Warum plötzlich ClientNummer-1? Ganz einfach, der Lesbarkeit halber nummeriere ich meine Sender von 1 - 6 durch, die dazugehörigen Datenkanaladressen stecken aber in einem Array, dass die enthaltenen Werte ab Speicherstelle 0 (Null) ablegt. .

Nun muß dem NRF24 gesagt werden, dass er ein Sender sein soll, dazu dient der Befehl:

radio.stopListening(); // Das angeschlossene Modul wird als Sender konfiguriert

Der Befehl hat eine umgedrehte Logik . stopListening bedeutet soviel wie "höre auf zuzuhören", was im Umkehrschluss heißt, "Du sollst was erzählen", also senden. Sieht irgendwie blöd aus, ist aber so.

Setupteil:

Code:

```
void setup() {
  Serial.begin(9600);              // Start des seriellen Ausgabe per USB an einen PC
  pinMode(button_pin, INPUT);
// Port zum Einlesen des Schalterzustandes konfigurieren

  radio.begin();                   // Start der 2,4 GHz Wireless Kommunikation
  radio.setDataRate(RF24_250KBPS);
// Übertragungsgeschwindigkeit setzen RF_1MBPS, RF_2MBPS
  radio.openWritingPipe(pipe[ClientNummer-1]);
// Setzen der Sendeadresse zur Übermittlung der Daten
  radio.setPALevel(RF24_PA_HIGH);
// Leistung des NRF Moduls je nach Entfernung kann man von  MIN bis MAX einstellen
//(MAX,HIGH,LOW,MIN)
  radio.stopListening();
// Das angeschlossene Modul wird als Sender konfigurieret

  myDS18B20.begin();               // Start des DS18B20 Sensors
  myDS18B20.setResolution(tempDeviceAddress, DS18B20_Aufloesung);
// Setzen der Messauflösung

  LichtWert = analogRead(LichtSensor);     // Lichtwert aus Fotozelle auslesen
  // Datenarray mit sinnvollen Werten füllen
```

```
  Messung.Event=0;
  Messung.Temp=25;
  Messung.Lux=LichtWert;
}
```

Nun kommt der Teil, indem Werte gemessen und verschickt werden sollen. Ich fange mit der Temperatur an.

```
Raumtemperatur = LeseTemperaturDS(3); // Funktion zum Einlesen der Temperatur aufrufen 2 x Einlesen lassen
intTemperatur = Raumtemperatur*100; // Wert mal 100 und in Int umwandeln
Messung.Temp = intTemperatur;
```

Zum Einlesen der Temperatur habe ich eine kleine Funktion geschrieben, die den Messwert mehrmals vom DS18B20 einlesen soll und mir den Mittelwert zurückliefert. Im obigen beispiel übergebe ich eine 3 und sage damit, dass der Sensor drei mal ausgelesen werden soll. Ich könnte dort auch 1 oder 2 eintragen. Allerdings dauert mit jeder zusätzlichen Messung der Funktionsaufruf länger, weil der DS18B20 nicht der Schnellste ist.

Für die Übertragung wird der Temperaturwert noch in den den Typ Integer umgewandelt. Das muss man nicht machen, man spart aber Datenbytes bei der Übertragung. Der endgültige Messwert landet dann im Datenpaket mit "Messung.Temp = intTemperatur;"

So ähnlich geht es dann mit der Lichtmessung.

```
LichtWert = analogRead(LichtSensor); // Lichtwert aus Fotozelle auslesen
Messung.Lux=LichtWert;
```

Das Signal vom Bewegungsmelder hole ich vom Digitaleingang 7. Im Programm habe ich den Pin vorher in die Variable

```
byte button_pin = 7; // Signalpin zum Einlesen des Schaltsignals (Taster, Bewegungsmelder etc.)
```

gesteckt. Warum der Umweg? Es könnte sein, dass vielleicht der Datenpin 7 mal anderweitig benötigt wird, dann brauche ich nur ganz oben einmal die Nummer des neuen Datenpins angeben und muss nicht im Programmcode überall suchen wo die 7 stecken könnte .

Den Messwert hole ich dann per:

```
Signal = digitalRead(button_pin); // Einlesen des Schalterzustandes
Messung.Event=Signal;
```

ab und schiebe in gleich in das Datenpäckchen zum Transport.

Für das Senden rufe ich dann nur noch meine kleine Funktion SendeDaten2(); auf, die am Ende des Programms hinter der Hauptschleife loop() steht.

```
void DatenSenden2()
{
Serial.println("sende!");
radio.write(&Messung, sizeof(Messung)); //Senden des Schalterstatus zum Empfänger
}
```

Das Verarbeitungsprogramm void loop()

Code:

```
void loop()
{
        Signal = digitalRead(button_pin);      // Einlesen des Schalterzustandes
        Messung.Event=Signal;
         if (Signal == 0){ merker = 0;
         ;}

       // Temperatur alle 10s einlesen
       vergangene_zeitSI = millis() - startzeitSI;
// Abfrage ob Wartezeit um ist.
       if ((vergangene_zeitSI > SendeIntervallms) && (merker ==0))
// Wenn die Wartezeit vorrüber ist
       {
         Raumtemperatur = LeseTemperaturDS(3);
// Funktion zum Einlesen der Temperatur aufrufen 2 x Einlesen lassen
         intTemperatur = Raumtemperatur*100;
// Wert mal 100 und in Int umwandeln
         Messung.Temp = intTemperatur;
         Serial.println(intTemperatur);
         LichtWert = analogRead(LichtSensor);  // Lichtwert aus Fotozelle auslesen
         Messung.Lux=LichtWert;
         Serial.println(LichtWert);
         startzeitSI = millis();
       }

       // Daten Senden wenn SendeWartezeit um ist oder ein neuer Alarm erkannt wird
       vergangene_zeit = millis() - startzeit;
       if ((vergangene_zeit > MessIntervallms)||((merker == 0) && (Signal == 1)))
       // Wenn die Wartezeit vorüber ist oder eine Bewegung erkannt wird und
//vorher kein Signal da war
       {
         DatenSenden2();
         startzeit = millis();                 // Neue Startzeit setzen
         merker=1; // Merker um nur einmal bei einer Bewegung zu senden
      }
delay(100);

}
```

Zusatzfunktionen, die hinter void loop() definiert werden:

Code:

```
// Funktion zum Auslesen eines !einzigen! DS Sensors ohne Adresse, Der Wert wird drei
mal eingelesen und der Mittelwert gebildet
float LeseTemperaturDS(byte wiederholungen)
{
  if (wiederholungen > 3) {wiederholungen =3;}
```

```
// wenn zu oft wiederholt wird verzögert sich der Programmablauf zu stark
  if (wiederholungen <= 0){wiederholungen = 1;}
// wenn ein Wert kleiner oder gleich Null gesetzt wird, wird einmal gemessen
  int x = 0;
  float messwert = 0;
  for (x = 1; x < wiederholungen+1; x++)
  {
    myDS18B20.requestTemperatures();
// DS18B20 anweisen eine Temperatur zu messen
    messwert = messwert + myDS18B20.getTempCByIndex(0);
// Messwert des ersten verfügbaren Sensors (Index 0) abrufen
    delay(10);
  }
  return messwert / wiederholungen;
// Mittelwert berechnen und an den Programmaufruf zurückgeben
// Ende LeseTemperaturDS
}

void DatenSenden2()
{
  Serial.println("sende!");
  radio.write(&Messung, sizeof(Messung));
//Senden des Schalterstatus zum Empfänger
  //radio.write(&Messung.intTemperatur, sizeof(intTemperatur));
// Senden der Zeichenkette (CharArray) zum Empfänger
  //radio.write(&LichtWert, sizeof(LichtWert));
// Senden der Helligkeit zum Empfänger
}
```

Es geht noch weiter, muss aber mal weg ....

Zuletzt bearbeitet: 29. November 2019

12.Das ganze Programm für den Sender sieht dann so aus:

Achtung bei der ClientNummer, die muss für jeden einzelnen Sender anders angegeben werden. Die Werte liegen zwischen 1 bis 6. Es ist möglich, dass noch Variablen definiert wurden, die keine Verwendung mehr finden, das kann beim Umschreiben des Programms passiert sein und ich habe das noch nicht ausgemistet. Manchmal lasse ich etwas noch im Programm, falls meine Änderungen nicht funktionieren und ich den Rückschritt einfacher hinbekomme.

Code:

```
// Programm zur Übermittlung eines Schalterzustandes mit einem 2,4 GHz NRF24 Modul

// Die Betriebsspannung vom NRF24 Modul MUSS!! an 3,3V vom Arduino angeschlossen werden
// ACHTUNG!!! ca 10 nF Elko zwischen 3,3V und Masse schalten um die Übertragung zu
stabilisieren

// Bibliotheken zur Bedienung des NRF24 Moduls
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
```

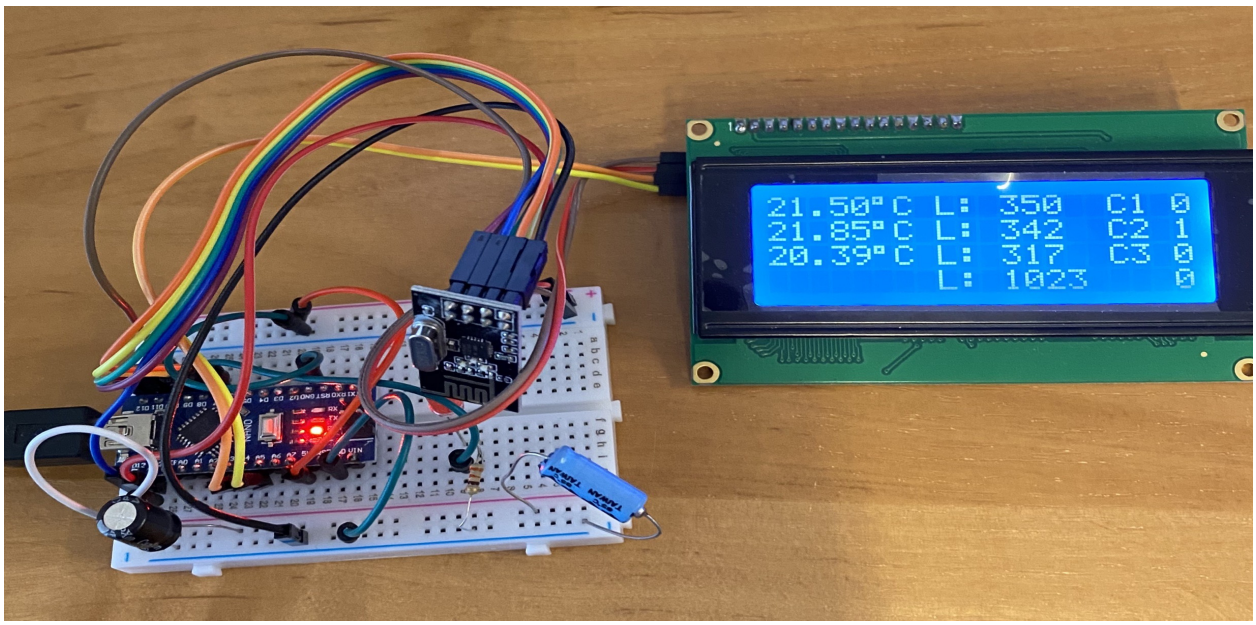
```
struct Wert {byte Event; int Temp; int Lux;} Messung;

// Einbinden eines Dallas DS18B20 Temperatursensors
#include <DallasTemperature.h>      // Library für Dallas Temperatursensoren
#define ONE_WIRE_BUS 5
// der Eindrahtbus für die DS18B20 liegen auf Digitalport 5
#define DS18B20_Aufloesung 10
// Die Sensoren sollen eine 11 Bit Auflösung liefern

OneWire oneWire(ONE_WIRE_BUS);
// Erstellen einer OneWire Instanz für den DS18B20 Temperatursensor
DallasTemperature myDS18B20(&oneWire);
DeviceAddress tempDeviceAddress;
// temporäre Adresse um die Auflösung des DS18B20 setzen zu können
// keine Ahnung warum das so ist, aber es funktioniert

// Variablen für die Temperaturmessung und -übertragung
float Raumtemperatur;
// Variablen für die Temperaturwerte der Messstellen definieren
// die Raumtemperatur float wird zuert mit 100 multipliziert und zum Integer gemacht
int intTemperatur = 25;
// Variablen für die Temperaturwerte als Integer

int LichtWert;
// Werte zwischen 0 und 1024, 0 = ganz hell, 1024 total dunkel
// Lampenlich 250  bis 500, Tageslicht kleiner 250, kein Licht größer 500
byte LichtSensor = 4; // Analogport für das Messignal vom Lichtsensor

// Variablen und Instanzendefinition für das NRF Modul
RF24 radio(8, 9);
// CE, CSN - die Zahlen geben die Digitalports am Arduino an,
//Instanz um das Modul zu starten
static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL,
0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL};
byte ClientNummer = 3; // Mögliche Werte: 1-6

const byte address[6] = "00002";
//Adresse, auf dem die Empfangsdaten gesendet werden sollen. Der Empfänger benötigt
// dieselbe Adresse!
byte button_pin = 7;
// Signalpin zum Einlesen des Schaltsignals (Taster, Bewegungsmelder etc.)
byte Signal = 0;             // Ausgabewert Signalpin landet in Signal
byte merker = 0;

// Variablen zur Zeitmessung ohne delay
  unsigned long MessIntervallms = 5000;   // Messintervall in Millisekunden
  unsigned long startzeit;
// Merker für die Systemzeit beim Eintreten einer Bewegung
```

```
  unsigned long vergangene_zeit;
// ausgerechneter Wert zwischen Merker und aktueller Systemzeit

  unsigned long SendeIntervallms = 10000; // Sendeintervall in Millisekunden
  unsigned long startzeitSI;
// Merker für die Systemzeit beim Eintreten einer Bewegung
  unsigned long vergangene_zeitSI;
// ausgerechneter Wert zwischen Merker und aktueller Systemzeit


void setup() {
  Serial.begin(9600);             // Start des seriellen Ausgabe per USB an einen PC
  pinMode(button_pin, INPUT);
// Port zum Einlesen des Schalterzustandes konfigurieren

  radio.begin();                  // Start der 2,4 GHz Wireless Kommunikation
  radio.setDataRate(RF24_250KBPS);
// Übertragungsgeschwindigkeit setzen RF_1MBPS, RF_2MBPS
  radio.openWritingPipe(pipe[ClientNummer-1]);
// Setzen der Sendeadresse zur Übermittlung der Daten
  radio.setPALevel(RF24_PA_HIGH);
// Leistung des NRF Moduls je nach Entfernung kann man von  MIN bis MAX einstellen
//(MAX,HIGH,LOW,MIN)
  radio.stopListening();
// Das angeschlossene Modul wird als Sender konfigurieret

  myDS18B20.begin();              // Start des DS18B20 Sensors
  myDS18B20.setResolution(tempDeviceAddress, DS18B20_Aufloesung);
// Setzen der Messauflösung

  LichtWert = analogRead(LichtSensor);     // Lichtwert aus Fotozelle auslesen
  // Datenarray mit sinnvollen Werten füllen
  Messung.Event=0;
  Messung.Temp=25;
  Messung.Lux=LichtWert;
}

void loop()
{
        Signal = digitalRead(button_pin);      // Einlesen des Schalterzustandes
        Messung.Event=Signal;
         if (Signal == 0){ merker = 0;
         ;}

      // Temperatur alle 10s einlesen
      vergangene_zeitSI = millis() - startzeitSI;
// Abfrage ob Wartezeit um ist.
```

```
      if ((vergangene_zeitSI > SendeIntervallms) && (merker ==0))
// Wenn die Wartezeit vorrüber ist
      {
        Raumtemperatur = LeseTemperaturDS(3);
// Funktion zum Einlesen der Temperatur aufrufen 2 x Einlesen lassen
        intTemperatur = Raumtemperatur*100;
// Wert mal 100 und in Int umwandeln
        Messung.Temp = intTemperatur;
        Serial.println(intTemperatur);
        LichtWert = analogRead(LichtSensor);  // Lichtwert aus Fotozelle auslesen
        Messung.Lux=LichtWert;
        Serial.println(LichtWert);
        startzeitSI = millis();
      }

      // Daten Senden wenn SendeWartezeit um ist oder ein neuer Alarm erkannt wird
      vergangene_zeit = millis() - startzeit;
      if ((vergangene_zeit > MessIntervallms)||((merker == 0) && (Signal == 1)))
      // Wenn die Wartezeit vorrüber ist oder eine Bewegung erkannt wird und vorher
kein Signal da war
      {
        DatenSenden2();
        startzeit = millis();                // Neue Startzeit setzen
        merker=1; // Merker um nur einmal bei einer Bewegung zu senden
     }
delay(100);

}

// Funktion zum Auslesen eines !einzigen! DS Sensors ohne Adresse, Der Wert wird drei
mal eingelesen und der Mittelwert gebildet
float LeseTemperaturDS(byte wiederholungen)
{
  if (wiederholungen > 3) {wiederholungen =3;}
// wenn zu oft wiederholt wird verzögert sich der Programmablauf zu stark
  if (wiederholungen <= 0){wiederholungen = 1;}
// wenn ein Wert kleiner oder gleich Null gesetzt wird, wird einmal gemessen
  int x = 0;
  float messwert = 0;
  for (x = 1; x < wiederholungen+1; x++)
  {
    myDS18B20.requestTemperatures();                    // DS18B20 anweisen eine
Temperatur zu messen
    messwert = messwert + myDS18B20.getTempCByIndex(0); // Messwert des ersten
verfügbaren Sensors (Index 0) abrufen
    delay(10);
  }
```

```
  return messwert / wiederholungen; // Mittelwert berechnen und an den Programmaufruf
zurückgeben
// Ende LeseTemperaturDS
}

void DatenSenden2()
{
  Serial.println("sende!");
  radio.write(&Messung, sizeof(Messung));   //Senden des Schalterstatus zum Empfänger
  //radio.write(&Messung.intTemperatur, sizeof(intTemperatur)); // Senden der
Zeichenkette (CharArray) zum Empfänger
  //radio.write(&LichtWert, sizeof(LichtWert));         // Senden der Helligkeit zum
Empfänger
}
```

13.Kommen wir nun zum etwas schwierigeren Teil der Sache, dem Empfangsprogramm für den Master, d.h. für die Anzeigeeinheit. Die Anzeigeeinheit verfügt nur über einen NRF24L01 2,4GHz Empfangsmodul, natürlich dem Arduino Nano (Clone) und einem 2004 Display. Das Display kann 20 Zeichen in vier Zeilen darstellen. Für die Stabilisierung der 3,3V Spannung, wenn der NRF24 aktiv wird, benötigt man noch einen Elko als Pufferspeicher. Ich verwende jetzt meistens einen 45 myF Elko. Die noch sichtbare Diode hat keine Funktion mehr und diente eine Zeit lang nur dazu, eine Bewegung beim Sender anzuzeigen. Bei mehreren Sendern wird das aber umständlich, weshalb ich das nun schlicht auf dem Display für jeden einzelnen Sender anzeige.



Ich beschränke mich bei den Erklärungen primär auf das Handling des NRF24L01, natürlich sind auch Programmzeilen zur Bedienung des Displays vorhanden und Programmzeilen zum Auswerten der eingelesenen Werte. Dazu mehr im Anschluss.

Die verwendeten Bibliotheken (Librarys) sind die gleichen wie beim Sender, ebenso die Definition der Kommunikationsadressen.

```
// Bibliotheken zur Bedienung des NRF24 Moduls
#include <SPI.h>
#include <nRF24L01.h>
```

```
#include <RF24.h>

// Variablen für die Verarbeitung der empfangenen Daten
struct Wert {byte Event; int Temp; int Lux;} Messung;
static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL, 0xF0F0F0F0B4LL,
0xF0F0F0F0A5LL, 0xF0F0F0F096LL};
uint8_t ClientNummer;
byte pipeNr = 0;

// Variablen zur Bedienung des NRF24 Moduls
RF24 radio(8, 9); // CE, CSN - die Zahlen geben die Digitalports am Arduino an, Instatz um das Modul zu
starten
byte button_state = 0; // Überprüfung des eingehenden Schaltsignals
int led_pin = 7; // Signalpin zur Anzeige des eingehenden Schaltsignals
float Temperatur = 0; // Fließkommavariable für die Temperatur
int tempTrans = 0; // Ankommender Temperaturwert der noch durch 100.0 geteilt werden muß
int LichtWert = 0; // Werte zwischen 0 und 1024, 0 = ganz hell, 1024 total dunkel
// Lampenlich 250 bis 500, Tageslicht kleiner 250, kein Licht größer 500
```

Kompletter Vereinbarungsteil als Code:

Code:

```
// Programm zum Empfangen eines Schalterzustandes mit einem 2,4 GHz NRF24 Moduls

// Die Betriebsspannung vom NRF24 Modul MUSS!! an 3,3V vom Arduino angeschlossen werden
// ACHTUNG!!! ca 10 nF Elko zwischen 3,3V und Masse schalten um die Übertragung zu
stabilisieren

#include <LiquidCrystal_I2C.h> // LiquidCrystal_I2C Bibliothek einbinden
LiquidCrystal_I2C lcd(0x27, 20, 4);
//Hier wird festgelegt um was für einen Display es sich handelt.
// In diesem Fall eines mit 16 Zeichen in 2 Zeilen und der HEX-Adresse 0x27.
// Für ein vierzeiliges I2C-LCD verwendet man den Code "LiquidCrystal_I2C lcd(0x27, 20, 4)"


// Bibliotheken zur Bedienung des NRF24 Moduls
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>


// Variablen für die Verarbeitung der empfangenen Daten
struct Wert {byte Event; int Temp; int Lux;} Messung;
static const uint64_t pipe[6] = {0xF0F0F0F0E1LL, 0xF0F0F0F0D2LL, 0xF0F0F0F0C3LL,
0xF0F0F0F0B4LL, 0xF0F0F0F0A5LL, 0xF0F0F0F096LL};
uint8_t ClientNummer;
byte pipeNr = 0;

// Variablen zur Bedienung des NRF24 Moduls
  RF24 radio(8, 9);
// CE, CSN - die Zahlen geben die Digitalports am Arduino an,
//Instanz um das Modul zu starten
  byte button_state = 0;                // Überprüfung des eingehenden Schaltsignals
  int led_pin = 7;
```

```
// Signalpin zur Anzeige des eingehenden Schaltsignals
  float Temperatur = 0;                 // Fliesskommavariable für die Temperatur
  int tempTrans = 0;
// Ankommender Temperaturwert der noch durch 100.0 geteilt werden muß
  int LichtWert = 0;
// Werte zwischen 0 und 1024, 0 = ganz hell, 1024 total dunkel
// Lampenlich 250  bis 500, Tageslicht kleiner 250, kein Licht größer 500

// Variablen zur Zeitmessung
  unsigned long MessIntervallms = 3200;
// Messintervall ca. 3,2 s, damit die blinde Zeit überbrückt wird
  unsigned long startzeit[] ={0,0,0,0,0,0};
// Merker für die Systemzeit beim Eintreten einer Bewegung
  unsigned long vergangene_zeit[] ={0,0,0,0,0,0};
// ausgerechneter Wert zwischen Merker und aktueller Systemzeit

// Merkervariablen zur Bewegungserkennung
  byte Bewegung[] = {0 ,0 ,0 ,0, 0, 0};
// Merker, ob eine Bewegung erkannt wurde
  int Anz_Bewegungen = 0;               // Merker, wie oft eine Bewegung erkannt wurde
  byte StatusClient[] = {0,0,0,0,0,0};  // Merker, ob eine Bewegung erkannt wurde

// Variablen für die Darstellung auf dem Display
byte AlarmSpalte = 19;
byte LichtSpalte = 8;
byte TemperaturSpalte =0;
```

14.

In der Setuproutine müssen ein paar Dinge erledigt werden, damit das NRF24 Modul aktiviert wird, das Display angesteuert werden kann und die Variablen bereits mit sinnvollen Werten gefüllt werden können. Um beim Start bereits etwas auf dem Display sehen zu können, werden ein paar Dummydaten angezeigt, bis sinnvolle Messwerte von den Messstellen gesendet werden.

Code:

```
void setup() {

  // Initialisierung des Schaltausganges und der seriellen Schnittstelle zum PC

  Serial.begin(9600);

  // Initialisierung der LCD-Anzeige und der Hintergrundbeleuchtung
  lcd.init();       //Im Setup wird der LCD gestartet
  lcd.backlight();
//Hintergrundbeleuchtung einschalten (lcd.noBacklight(); schaltet die Beleuchtung aus).

  radio.begin();
  radio.setDataRate(RF24_250KBPS);
// Übertragungsgeschwindigkeit setzen RF_1MBPS, RF_2MBPS

  for (pipeNr = 0; pipeNr < 6; pipeNr++){
```

```
    radio.openReadingPipe(pipeNr, pipe[pipeNr]);
// Adressen, auf dem die Empfangsdaten erwartet werden sollen
    delay(50);
   }

  radio.setPALevel(RF24_PA_MAX);
// Leistung des NRF Moduls je nach Entfernung kann man von  MIN bis MAX einstellen
//(MAX,HIGH,LOW,MIN)
  radio.startListening();
// Das angeschlossene Modul wird als Empfänger konfigurieret

  Serial.println("gestartet");
  startzeit[0] = millis(); // Setzen des Merkers für die Wartezeit
  startzeit[1] = startzeit[0]; startzeit[2] = startzeit[0]; startzeit[3] =
startzeit[0]; startzeit[4] = startzeit[0];
  lcd.setCursor(0, 0);
// Cursor auf das erste Zeichen in der ersten Zeile setzen
  lcd.print("NRF24 Testprg 4.11.");

// Kurze Anzeige des Programms auf dem Display
  delay(2000);
  clearZeile(1);

  // Ein paar Ausgaben auf das Display, damit es bis zum ersten empfangenen Datenpaket
//nicht leer aussieht
  zeigeLicht(1023, 0, LichtSpalte);
  zeigeLicht(1023, 1, LichtSpalte);
  zeigeLicht(1023, 2, LichtSpalte);
  zeigeLicht(1023, 3, LichtSpalte);

  zeigeAlarm(0, 0 , AlarmSpalte); // Alarmwert, Zeilennummer, Spalte
  zeigeAlarm(0, 1 , AlarmSpalte); // Alarmwert, Zeilennummer, Spalte
  zeigeAlarm(0, 2 , AlarmSpalte); // Alarmwert, Zeilennummer, Spalte
  zeigeAlarm(0, 3 , AlarmSpalte); // Alarmwert, Zeilennummer, Spalte

  Messung.Event=0;
  Messung.Temp=25;
  Messung.Lux=LichtWert;

}
```

In der Verarbeitungsschleife lese ich die Messwerte von den verschiedenen Clients ein und gebe sie sortiert auf dem Display aus.

Damit der Arduino nicht wie wild versucht, bereits gesendete Daten permanent auf dem Display auszugeben (dann fängt es an zu flackern) springe ich in die Anzeigeroutinen nur dann, wenn ein Ereignis stattfindet, also ein Sender etwas abliefert, oder nach einer festgelegten Zeit, wenn nichts passiert. Dafür nehme ich nicht die Funktion delay(), weil das den Arduino ausbremst, denn während er in einer Delayschleife steckt, kann er nichts anderes mehr machen und würde

ein einlaufendes Datenpaket nicht erkennen und abholen können. Die Geheimfunktion heißt millis() und liefert die aktuelle Systemzeit des Arduinos in Millisekunden. Natürlich muss man dann ein paar zusätzliche Zwischenspeicher für die Systemzeiten vorhalten.

Wichtig ist, dass ich für jeden möglichen Sender (Client) eigene Werte speichere, um jeden Client gesondert bedienen zu können. Die Werte liegen dann in Datenarrays, die ich über die jeweilige ClientNummer adressieren kann. Zum Beispiel:

```
startzeit[ClientNummer] = millis(); // Neue Startzeit setzen
Bewegung[ClientNummer]=0;
StatusClient[ClientNummer] = 1 ;
```

Diese Zeilen sind aber aus dem Zusammenhang des Programms gerissen und tauchen dort in der Reihenfolge nicht auf.

Hier der Code für die Hauptschleife void loop()

Code:

```
void loop() {
// Anfang der Hauptschleife

if (radio.available(&ClientNummer))
// Wenn ein Code Empfangen wird springt das Programm in die Auswerteschleife
  {
    Serial.print("Clientnummer:");Serial.println(ClientNummer);
    radio.read(&Messung, sizeof(Messung));
    EmpfangeDaten();
    button_state = Messung.Event;
    tempTrans = Messung.Temp;
    LichtWert = Messung.Lux;

    if (tempTrans != 0){
      Serial.print("tempTrans:");
      Serial.println(tempTrans);
      Temperatur = tempTrans/100.0;
// Temperatur wieder in einen Kommawert umrechnen mit 2 Nachkommastellen
      Serial.println(Temperatur);

      zeigeTemperatur(Temperatur, ClientNummer, TemperaturSpalte);
      zeigeLicht(LichtWert, ClientNummer, LichtSpalte);

      lcd.setCursor(16, ClientNummer); lcd.print("C");lcd.print(ClientNummer+1);
    }

    if (button_state == LOW) // Wenn die Empfangenen Daten "0" sind,
    {

            Bewegung[ClientNummer]=0;
    }

    else if (button_state == HIGH)
```

```
// Wenn der Empfangenen Daten "1" sind
    {
      Bewegung[ClientNummer]=1; // Bewegungsmerker setzen
      startzeit[ClientNummer] = millis(); // Neue Startzeit setzen
    }
  }

     vergangene_zeit[ClientNummer] = millis() - startzeit[ClientNummer];
// Abfrage ob Wartezeit um ist.

      if ((Bewegung[ClientNummer] == 1)&& (StatusClient[ClientNummer] == 0)){

          StatusClient[ClientNummer] = 1 ;
          zeigeLicht(LichtWert, ClientNummer, LichtSpalte);
// Lichtwert, Zeilennummer, Spalte
          lcd.setCursor(16, ClientNummer); lcd.print("C");lcd.print(ClientNummer+1);
          zeigeAlarm(Bewegung[ClientNummer], ClientNummer, AlarmSpalte);
// Alarmwert, Zeilennummer, Spalte
          zeigeTemperatur(Temperatur, ClientNummer, TemperaturSpalte);

      }

      if ((vergangene_zeit[ClientNummer] > MessIntervallms) && (Bewegung[ClientNummer]
== 0)&&(StatusClient[ClientNummer] == 1)){
// Wenn die Wartezeit ohne Ereignis vorbei ist und die Lampe noch an, wird sie ausgeschaltet
          Bewegung[ClientNummer]=0;
          StatusClient[ClientNummer] = 0;
          zeigeAlarm(Bewegung[ClientNummer], ClientNummer, AlarmSpalte);
// Alarmwert, Zeilennummer, Spalte
          Serial.print("keine Bewegung mehr erkannt
Client");Serial.println(ClientNummer);
          zeigeLicht(LichtWert, ClientNummer, LichtSpalte);
// Lichtwert, Zeilennummer, Spalte

      }

    delay(50);
// Ende der Hauptschleife
}
```

Die Hauptschleife enthält natürlich die Lösungen für verschiedene Probleme beim Abholen der Daten, zwischenspeichern und ausgeben der Infos, das lässt sich aber nicht kurz mit zwei Sätzen erklären. Das hebe ich mir für später auf.

Um den Code der Hauptschleife übersichtlicher zu gestalten, habe ich gleiche, immer wiederkehrende Programmzeilen in Funktionen ausgelagert und ans Ende des Programms gelegt. Das spart zusätzlich Speicherplatz. Es handelt sich hauptsächlich um Programmzeilen zur Darstellung der Messwerte auf dem Display und um die Programmzeilen zum Abholen der empfangenen Messwerte. Weil ich die Zeilen überwiegend gut mit Kommentaren versehen habe,

sollten die Erklärungen dort ausreichen.

Im aktuellen Stadium sind sicher noch weitere Optimierungen möglich.

Code:

```
void clearZeile(byte Z)
{
lcd.setCursor(0, Z-1);
// Cursor auf das erste Zeichen in der übermittelten Zeile setzen
  lcd.print("                    ");
// 20 Leezeichen um die Zeile quasi zu löschen falls noch Fragmente aus vorherigen
//Anzeigedaten dort übrig geblieben sind

}

void zeigeLicht(int LW, byte Zeile, byte Spalte){
          lcd.setCursor(Spalte, Zeile); lcd.print("L: ");
          lcd.setCursor(Spalte+3,Zeile); lcd.print("    ");
          lcd.setCursor(Spalte+3,Zeile); lcd.print(LW);
}

void zeigeAlarm(byte Aktion, byte Zeile, byte Spalte){
  lcd.setCursor(Spalte, Zeile);lcd.print(Aktion);

}

void zeigeTemperatur(float Waerme,byte Zeile, byte Spalte){
  lcd.setCursor(Spalte, Zeile); lcd.print(Waerme); lcd.print("\xDF" "C ");
}

void EmpfangeDaten(){
    radio.read(&button_state, sizeof(button_state));
//Einlesen des Schalterstatus vom Sender
    radio.read(&tempTrans, sizeof(tempTrans));
//Einlesen der Zeichenkette vom Sender
    radio.read(&LichtWert, sizeof(LichtWert));
//Einlesen der Zeichenkette vom Sender
}
```

Zuletzt bearbeitet: 5. November 2019

BAXL, 5. November 2019

15.Reservepost für evtl. noch folgende Ausführungen. Ab jetzt kann dann auch hier drin von anderen Mitgliedern gepostet werden, falls Fragen bestehen, oder hilfreiche Anmerkungen sind.

16.Die Geheimnisse der kryptisch aussehenden Datentypen für Variablen

Kleiner Exkurs zur Definition von Variablen. Variable im Programm sind quasi Platzhalter oder bildlich gesprochen, kleine Kästchen, in die man Zahlen, Buchstaben, Wörter usw. packen kann. Der Prozessor geht dafür direkt beim Start des Programms her und stellt die leeren Kästchen dahin. In Wirklichkeit reserviert er nur bestimmte Speicherstellen, damit die nicht für etwas

Anderes benutzt werden. Die Krux ist, dass der Prozessor vorher wissen muss, wie groß die Kästchen sein müssen, damit der erwartete Inhalt dort hineinpasst.

Der kleinste reservierbare Speicherbereich ist ein Byte, also 8 Bit. Damit können Zahlen von 0 bis 255 gespeichert werden. Darum auch die Bezeichnung des Datentyps als byte.
Dann gibt es noch den Datentyp word, der aus zwei Byte besteht und Zahlen von 0 bis 65535 speichern kann.
Buchstaben werden nicht als Buchstaben gespeichert, sondern als Zahl. Es wurden irgendwann mal die Buchstaben durchnummeriert und damit festgelegt, welcher Buchstabe mit welcher Zahl bezeichnet wird. So hat der Buchstabe E die Zahl 69 und das e die Zahl 101. Das bedeutet also, dass man ein Byte benötigt um einen Buchstaben zu speichern.

Je nach Programmiersprache findet man entsprechende Bezeichnungen der Datentypen, die aber im Prinzip immer darauf basieren, dass für irgendetwas, was gespeichert werden soll, eine gewisse Anzahl Bytes, also Speicherstellen, reserviert werden müssen. Bei einem PC mit Gigabyte großen Arbeitsspeichern und Terabytes großen Festplatten hat man Speicher satt und kann etwas großzügiger sein, aber bei kleinen Microcontrollern ist Speicher immer noch knapp, also sollte man vorher genau überlegen, für welche Information man Speicher braucht. So wäre es Verschwendung, Zahlen von 1 bis 6 im Datenyp Integer int zu speichern, obgleich der Datentyp byte vollkommen ausreicht.

Warum schreibe ich das alles jetzt. In meinem Programm finden sich jede Menge Variablendefinitionen - die man allgemein so kennt und man auch in der Referenz der Arduino IDE nachlesen kann. Dem aufmerksamen Beobachter wird aber aufgefallen sein, dass da auch so Sachen wie : uint64_t oder uint8_t stehen.

Jaaaa, die Dinger habe ich mir eingeschleppt, als ich aus Programmbeispielen Programmteile abgekupfert habe. Ich hatte keine Ahnung was die Dinger bedeuten. Irgendwann bin ich über die erlauchte Zunft der sortenreinen C-programmierer gestolpert. Die benutzen diese Datentypen, die die Programmierumgebung vom Arduino auch versteht, weil die "Arduino-Programmiersprache" eigentlich auch C ist und sortenreinen C-Code verarbeiten kann.

bei uint8_t handelt es sich um den Datentyp Byte ohne Vorzeichen. Wobei das kleine u unsigned bedeutet.
Man kann darin Zahlen von 0 bis 255 speichern.

Bei uint64_t handelt es sich um etwas, was viele Namen haben kann, hier eine kleine Auswahl: Int64, QWord/Quadword, long long, Long/long

eigentlich haben wir aber nur eine Zahl ohne Vorzeichen (also immer positiv), die von 0 bis 18.446.744.073.709.551.615 gehen kann.

Ich wollte das alles nur mal erwähnen, weil ich ebenfalls über solche Kleinigkeiten gestolpert bin. Für den C-Crack, der schon gefühlte 1000 Jahre programmiert, ist das sicher kalter Kaffee, für Meinereiner ist das aber ein böhmisches Dorf (zumindest solange bis ich es auch weiß). So versuche ich Schritt für Schritt die Mysterien der so hammermäßig cool aussehenden Programme der "Experten" zu entzaubern.

Auf einer Internetseite habe ich eine schöne schriftliche Umschreibung weiterer Datentypen gefunden, möchte das aber nicht einfach kopieren, darum verlinke ich dahin.

Das ist der **Blog vom Arduinopilot**

Es gibt noch viele andere Seiten, auf denen mit viel Liebe und Hingabe die Datentypen erklärt

werden, weshalb ich hier selbst nicht 398ste Variante veröffentlichen muss. Mir ist er nur wichtig die Kleinigkeiten aufzugreifen, die oftmals für kleine Irritationen sorgen und nirgendwo beschrieben werden, oder die man erst nach viel Suchen und Forschen findet.

Zuletzt bearbeitet: 5. November 2019

BAXL, 5. November 2019

17.Kommen wir nun zu der Logik in den Programmen

Dabei ist die Logik im Sender und im Empfänger zu unterscheiden. Ich beginne mit dem Empfänger.

Kern der Aktion ist es, in einem Raum die Temperatur und, die Helligkeit zu messen und ob sich etwas in dem Raum tut, also sich jemand in dem Raum befindet.

Als Datenlieferanten dienen, wie bereits erwähnt, ein Temperatursensor des Typs DS18B20, eine Fotozelle und ein PIR Bewegungsmelder.
Am Beginn meiner Planung sollte der Bewegungssensor schlicht eine Alarmfunktion erfüllen, um in einer Alarmanlage eine Bewegung zu detektieren, damit ggf. eine Lampe eingeschaltet wird, oder eine Meldung an eine Zentrale erfolgt. Im Verlauf des Projektes ist die reine Alarmfunktion etwas mehr in den Hintergrund getreten, weil sich das Ziel mehr in Richtung SmartHome bewegte. Aus den Anfängen stammen natürlich gewisse Anforderungen an eine direkte, verzögerungsfreie Meldung einer Bewegung, was sich auch im Code widerspiegelt. Das zur Grundidee.

Die Umsetzung:

Der Arduino fragt nun permanent den Signaleingang vom PIR ab. Der Eingang könnte auch von einem Mikroschalter (Tür- / Fensterkontakt) oder auch einem anderen zu überwachenden digitalen Ereignis stammen.

In der Hauptschleife wird deshalb zuerst und bei jedem Schleifendurchlauf dieser Eingang (bei mir Digitalpin 7) abgefragt.
Sobald ein Signal erkannt wird, soll unmittelbar eine Meldung zum Master (Empfänger) erfolgen. Ist keine Bewegung mehr detektiert, also Digitalpin 7 Null, soll nichts gemacht werden und ein Merker wird ebenfalls auf Null gesetzt. Das wird aber nicht sofort gesendet.

Der PIR hat nach seinem Ansprechen eine gewisse Nachlaufzeit, in der permanent das Bewegungssignal anliegt. In der Zeit brauch natürlich nicht dauernd gesendet werden. Außerdem hat der PIR nach dem Ausbleiben einer Bewegung eine Totzeit, in der keine Bewegung gemeldet werden kann, weil der PIR in der Zeit quasi blind ist. Diese Totzeit beträgt ca. 3,2 Sekunden. Diese Totzeit wird später im Empfänger kompensiert.

Wird ein Signal erkannt, springt das Programm in einen Auswerteprogrammteil und setzt ein Datentelegramm ab. (siehe weiter unten)

Nun ist bei meiner weiteren Aktivität noch die Temperaturmessung und die Helligkeitsmessung dazugekommen. Diese Werte sollen ebenfalls relativ aktuell am Master angezeigt werden, allerdings muß das nicht in Echtzeit passieren, damit würde der Sender und auch der Empfänger in der Verarbeitung überlastet. Also muß eine gewisse Verzögerung eingebaut werden. Diese Messverzögerung beträgt 5s oder auch 5000ms (von mir willkürlich gewählt). Ich rufe dazu die Systemzeit mit millis() ab und speichere diese in einer temporären Variablen, die ich Startzeit nenne und vom Datentyp unsigned long ist.

Nun wird die Systemzeit regelmäßig abgerufen und mit der Startzeit (für die Temperatur und Helligkeitsmessung) verglichen. Beträgt der Unterschied 5000ms, springt das Programm in eine Schleife, in der einmal die Temperatur vom DS18B20 abgefragt und die Helligkeit an der Fotozelle gemessen wird. Die Daten lege ich bereits in den Datencontainer ab, und setze die nun aktuelle Startzeit wirder auf die Systemzeit millis(). Diese Wartezeit ist auch deshalb erforderlich, weil das Abrufen der Temperatur vom DS18B20 unter Umständen etwas länger dauert (mehr als 1 s) und der Arduino sonst damit permanent beschäftigt ist, bzw. auch blockiert wird.

Für das Senden der Mess- und Bewegungsinformation definiere ich ebenfalls eine Wartezeit, die sogar 10s beträgt, d.h., alle 10s schickt der Arduino einen kompletten Datensatz mit Temperatur, Helligkeit und

Bewegungsmeldung zum Empfänger. Dafür gibt es dann die zweite Wartezeit, die SendeIntervalls heißt und 10000 ms beträgt. Die Startzeit kommt jeweils auch aus millis() und wird in startzeitSI gespeichert. Der Name ist aus Startzeit und SendeIntervall entstanden.

```
// Temperatur alle 5s einlesen
vergangene_zeitSI = millis() - startzeitSI; // Abfrage ob Wartezeit um ist.
if ((vergangene_zeitSI > SendeIntervallms) && (merker ==0))
// Wenn die Wartezeit vorrüber ist
{
Raumtemperatur = LeseTemperaturDS(3);
// Funktion zum Einlesen der Temperatur aufrufen 3 x Einlesen lassen
intTemperatur = Raumtemperatur*100; // Wert mal 100 und in Int umwandeln

Messung.Temp = intTemperatur; // Messwert in den Datencontainer packen
LichtWert = analogRead(LichtSensor); // Lichtwert aus Fotozelle auslesen
Messung.Lux=LichtWert; // Messwerte in den Datencontainer packen
startzeitSI = millis(); // neue Startzeit für das nächste Messintervall setzen
}
```

Wie bekomme ich es nun hin, dass immer bei einer Bewegung gesendet wird oder wenn die Wartezeit für das Sendeintervall um ist?

Dazu habe ich eine einfache Abfrage definiert, die all diese Bedingungen verknüpft. Allerdings gibt es noch das Problem, wenn der PIR seinen Ausgang eingeschaltet hat und dieses Signal länger ansteht. Es soll dann nicht permanent immer wieder gesendet werden, es reicht, wenn das erkannte Bewegungssignal nur einmal geschickt wird, bis es wieder verschwindet. Für den Zweck gibt es einen Merker, den ich am Programmstart auf Null setze und beim Erkennen einer Bewegung auf 1.

Der Auswerteteil sieht deshalb so aus:

```
// Daten Senden wenn SendeWartezeit um ist oder ein neuer Alarm erkannt wird
vergangene_zeit = millis() - startzeit;
if ((vergangene_zeit > MessIntervallms)||((merker == 0) && (Signal == 1)))
// Wenn die Wartezeit vorrüber ist oder eine Bewegung erkannt wird und vorher kein Signal da war
{
DatenSenden2();
startzeit = millis(); // Neue Startzeit setzen
merker=1; // Merker um nur einmal bei einer Bewegung zu senden
}
```

Ich prüfe darin 1., ob die Wartezeit von 10s um ist, dann ob der Merker 0 ist und das eingelesene Signal 1. D.h. wenn vorher keine Bewegung war und jetzt eine erkannt wird, soll gesendet werden, oder wenn die Wartezeit um ist, selbst wenn die Bewegung noch besteht und bereits gesendet wurde. (das muss man mehrmals lesen um es zu verstehen)

Das ist nun im Wesentlichen die Auswertelogik des Senders.

18.Als Nächstes haben wir das Empfängerprogramm, dessen größtes Problem/Umstand der Empfang mehrere Sender und Darstellung der empfangenen Daten auf ein Display war. Auch hier habe ich die Situation, dass zu einem unbekannten Zeitpunkt Daten einlaufen und die Anzeige stets aktuell gehalten werden soll. Das Einlaufen der Daten wird quasi automatisch erkannt, indem der NRF24 gefragt wird, ob neue Daten bereitstehen. Das passiert über:

```
if (radio.available(&ClientNummer)) { ... }
```

Der Rest sind zwei Programmteile zur Auswertung und dann etwas Fleißarbeit, um die erhaltenen Daten anzeigen zu lassen. Auch hier ist es wieder kontraproduktiv, wenn permanent alle aktuellen Daten bei jedem Schleifendurchlauf an das Display gesendet werden, weil das Display dann schlicht zu flackern anfängt, weil es nicht mehr richtig hinterher kommt.

Es gibt also wieder zwei Fälle, die auftreten können. Fall 1 betrifft die Bewegungsmeldung, die ohne Verzögerung angezeigt werden soll und Fall 2, dass in regelmäßigen Abständen die Temperatur- und Helligkeitswerte aktualisiert werden sollen.

Weil ich das für mehrere Clienten machen will, benötige ich diverse zwischengespeicherte Werte eben mehrfach. Die Umstellung von einem Client auf mehrere war relativ einfach, weil ich aus den bereits vorhandenen Variablen(namen) lediglich Datenfelder (Arrays) machen brauchte.

So wurde z.B. aus startzeit - startzeit[] = {0,0,0,0,0,0};

Das Array wird bei dieser Deklaration bereits in jeder Speicherzelle mit 0 (Null) gefüllt. Für die anderen Variablen, die mehrfach für jeden einzelnen Mess Clienten benötigt werden, muss das ebenso gemacht werden.

Glücklicherweise laufen die Datenpakete in diesen Pipes ein, die von 0 bis 5 durchnummeriert sind. Wenn mir der Befehl radio.available(&ClientNummer) meldet, dass Daten angekommen sind, dann kann ich über ClientNummer die Zuordnung der Daten vornehmen, indem ich ClientNummer einfach als Index für die verschiedenen Datenarrays der Merkervariablen verwende.

Weiterhin ist ClientNummer noch ganz praktisch, um die Messwerte den einzelnen Displayzeilen zuzuordnen. Die Daten von ClientNummer = 0 landen im Display in Anzeigezeile = 0, die von ClientNummer = 1 in Zeilennummer 1 usw.. Bei Zeilennummer 3, welches die 4. Zeile ist leider Schluß, weil mein 2004 LCD Display leider nur 4 Zeilen darstellen kann. Darum ist auch schon ein OLED-Display unterwegs, doch dazu später mehr.

Für die Aktualisierung der Anzeige benötige ich eigentlich nur eine einzige zusätzliche Wartezeit, nämlich die, die die blinde Zeit eines PIR-Sensors der Sender überbrückt. Weiterhin muß beim Ausbleiben von Bewegungen bei einem Clienten, die entsprechende Null in der Anzeige nicht permanent aktualisiert werden. Ich detektiere also lediglich auf die Änderung des eingehenden Signals für eine Bewegung. Darum auch eine Merkervariable für jeden Clienten.

Ansonsten wird bei einem einlaufenden Datenpaket jeweils nur die zugehörige Zeile des entsprechenden Clienten aktualisiert. Soll heißen, kommt von Messstelle 1 etwas, wird die Anzeigezeile Null aktualisiert. Wir erinnern uns, die Klienten (Messstellen) nummeriere ich von 1- 6 durch, die Datenarrays und auch das Display fangen bei 0 - Null an zu zählen.

Die Auswertelogik sieht nun derart aus, dass bei einem ankommenden Datenpaket zuerst einmal die Messwerte herausgezogen und in der richtigen Zeile aktualisiert werden.

Nun haben wir den Fall, dass bei einem der Klienten eine Bewegung erkannt wurde. Dann wird die jeweilige Anzeige aktualisiert, aber nur dann, wenn der Status sich vom vorhergehenden Datenpaket für diese Messstelle verändert hat. Also nur dann, wenn aus 1 die 0, oder aus 0 die 1 geworden ist, sonst passiert nix. (Wie gesagt, sonst flackert das Display)

Ich spare mir von den Messstellen noch die 3,2s blinde PIR-Zeit aus.

Das war es auch schon. Das Handling mit der Systemzeit, den Zwischenmerkern für die Systemzeit usw. funktioniert genau so wie bei den Sendern. Wichtig war nur, welche Bedingungen ich für die Anzeige und deren Aktualisierung gewählt habe.

Bei Unklarheiten kann hier gerne nachgefragt werden. Das ist auch nur für meinen speziellen Fall so gemacht, jemand anderes, der keine Bewegungsmelder benötigt, kann auch auf diese etwas umständlich erscheinende Auswerte- und Anzeigelogik verzichten und ganz einfach immer dann aktualisieren, wenn eine Messstelle eine Temperatur oder Lichtstärke sendet.

19.Zitat von BlackbirdXL1: ↑

Danke für das Lob, ich bin aber noch nicht am Ende angekommen, da droht noch die Sache mit mehr als 6 Messstellen und falls eine Messstelle mit ihrer Sendeleistung nicht bis zum Master kommt (Stichwort Mesh). Aber dafür muss ich noch etwas lesen, verstehen und experimentieren.

Im Moment habe ich da absolut keine Peilung. Vorher will ich noch das OLED-Display einsetzen und in meinem Hirn reift der erste Prototyp, der nicht mehr auf einem Steckbrett ist und mittelfristig in ein Gehäuse gepackt werden soll.

Sender

```
/ Programm zur Übermittlung eines Schalterzustandes mit einem 2,4 GHz NRF24 Modul

// Bibliotheken zur Bedienung des NRF24 Moduls
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

// Variablen und Instanzendefinition für das NRF Modul
RF24 radio(8, 9);

// CE, CSN - die Zahlen geben die Digitalports am Arduino an, Instanz um das Modul zu starten
const byte address[6] = "00001";

//Adresse, auf dem die Empfangsdaten gesendet werden sollen. Der Empfänger benötigt dieselbe Adresse!
boolean button_state = 0;

int button_pin = 7; // Signalpin zum Einlesen des Schaltsignals (Taster, Bewegungsmelder etc.)

void setup() {
Serial.begin(9600);

// Start des seriellen Ausgabe per USB an einen PC falls man sich zur Kontrolle im Programmablauf

//etwas anzeigen lassen möchte
pinMode(button_pin, INPUT); // Port zum Einlesen des Schalterzustandes konfigurieren
radio.begin(); // Start der 2,4 GHz Wireless Kommunikation
radio.openWritingPipe(address); // Setzen der Sendeadresse zur Übermittlung der Daten
radio.setPALevel(RF24_PA_HIGH);

// Leistung des NRF Moduls je nach Entfernung kann man von MIN bis MAX einstellen

//(MAX,HIGH,LOW,MIN)
radio.stopListening(); // Das angeschlossene Modul wird als Sender konfigurieret
}

void loop()
{

button_state = digitalRead(button_pin); // Einlesen des Schalterzustandes
radio.write(&button_state, sizeof(button_state)); //Senden des Schalterstatus zum Empfänger

delay(50); // kurze Verschnaufpause, damit der Empfänger sich nicht verschluckt
}
```

Empfänger

```
/ Bibliotheken für 2,4 GHz Modul
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

RF24 radio(8, 9); // CE, CSN
const byte address[6] = "00001";
boolean button_state = 0;
int led_pin = 7;


void setup() {
pinMode(7, OUTPUT);
Serial.begin(9600);
radio.begin();
radio.openReadingPipe(0, address); //Setting the address at which we will receive the data
radio.setPALevel(RF24_PA_MIN);

//You can set this as minimum or maximum depending on the distance between the transmitter and

//receiver.
radio.startListening(); //This sets the module as receiver
Serial.println("gestartet");
}

void loop()
{
if (radio.available()) //Looking for the data.
{
Serial.println("empfange!");
radio.read(&button_state, sizeof(button_state)); //Reading the data
if(button_state == HIGH)
{
digitalWrite(7, HIGH);
}
else
{
digitalWrite(7, LOW);
}

}
delay(50);
}
```