Von delay bis zur Methode

Dies soll ein kleines Tutorial sein, das dem Anfänger anhand eines einfachen Beispiels zeigen soll wie man *delay()* durch *millis()* ersetzt und was der Nachtwächter damit zu tun hat. Wir werden uns eine eigene Funktionen schreiben, eine eigene Klasse erstellen und daraus ein Objekt verwenden. Ich werde hier nicht darauf eingehen was Funktionen und Klassen sind, dazu gibt es Bücher oder z.B. https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Inhaltsverzeichnis oder http://www.cplusplus.com/doc/tutorial/

Es geht mir darum zu zeigen wie man eine Aufgabe mit verscheiden Möglichkeiten mehr oder weniger gut realisieren kann. Es wird im Grunde immer der gleiche Sketch mit den gleichen Variablennamen verwendet, was es dem Anfänger leicht macht die Unterscheide zu erkennen. Wir werden zunächst einen Grundsketch erstellen und dann Schritt für Schritt vorgehen.

Als Beispiel habe ich mir eine einfache Aufgabe, die vor ein paar Tagen hier im Forum als Post auftrat, ausgedacht. Dabei soll mit einem Tasters eine LED ein geschaltet werden. Die LED soll dabei langsam heller werden. Wird der Taster losgelassen soll die LED langsam wieder dunkler werden. Und weil es so schön ist, wollen das zweimal haben je für eine rote LED und für eine grüne LED.

Wenn wir nun unter den mitgelieferten Beispielen in der IDE suchen werden wir das Beispiel *Fading* finden. Wenn wir das nun auf unseren Arduino laden stellen wir fest das die LED schön hell und dunkel wird, genau das was wir wollen. Leider finden wir aber keinen Taster mit dem wir das ein oder ausschalten könnten, und das ist bei der verwendeten for Schleife auch ziemlich unsinnig. Wenn Anfänger über das Problem nachdenken kommt es häufig zu der Frage „wie kann ich eine for Schleife vorzeitig abbrechen? “. Da ist also schon der Ansatz falsch. Was liegt also näher als das neu zu machen. For Schleifen brauchen wir nicht wir haben ja schon loop. Natürlich gibt es sinnvolle Verwendungen von for Schleifen, schließlich gibt es die in jeder Programmiersprache. For schleifen machen aber, wie alle anderen Schleifen auch, bei einem Controller nur dann Sinn wenn sie schnell abgearbeitet werden, das ist aber sicher nicht der Fall wenn ein *delay()* mit in der Schleife steht und die Schleife womöglich 1000 mal durchlaufen werden muß.

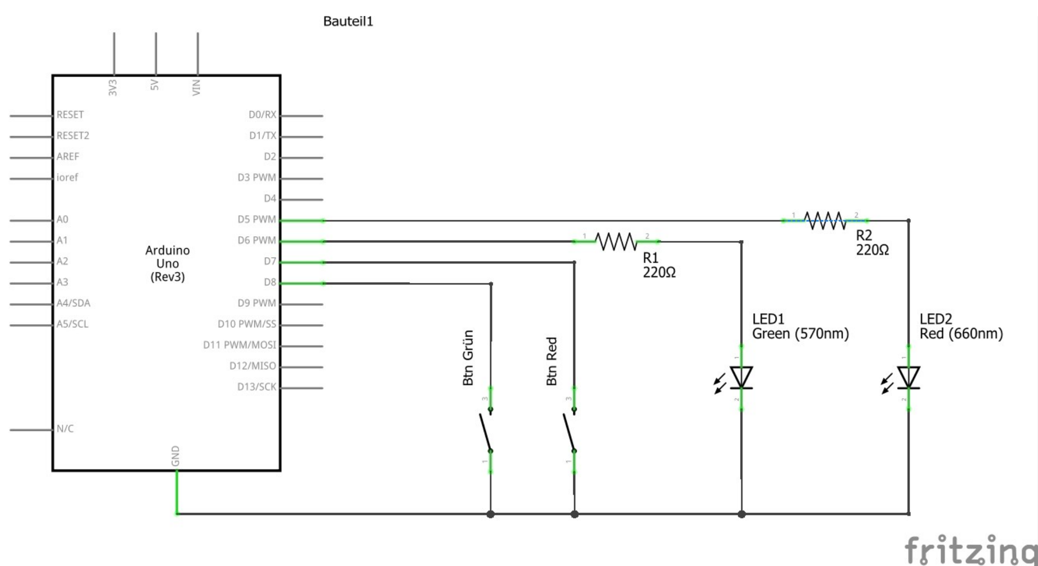

Schaubild 1: Schaltplan zum Tutorial

Der Grundsketch

Gerade für Anfänger ist es gut zuerst einen Programmablauf zu erstellen, hilft er doch die Gedanken logisch zu sortieren. Oft stellt man dabei auch schon mal fest das irgend etwas nicht logisch ist in dem geplanten Vorgehen. Für unseren Grundsketch könnte das so aussehen.

wenn der Taster gedrückt ist dann
wenn der max Wert noch nicht erreicht ist dann
wert um Schritt erhöhen
Wert an LED ausgeben
etwas warten
Wenn Taster nicht gedrückt (else) dann
wenn min Wert noch nicht erreicht dann
wert um Schritt verringern
Wert an LED ausgeben
etwas warten
zum Anfang

Mit etwas Übung kann man nun aus fast jeder einzelnen Textzeile eine Codezeile machen und schon ist unser Grundsketch fertig. In den Allgemeinteil und das Setup kommen noch ein paar Definitionen für Variable und die Festlegung der E/A Pin´s. Um die Geschwindigkeit des „Fading” leichter ändern zu können werden die Schrittgröße und die Wartezeit als Variable im Allgemeinteil vorgegeben. Die Pins und Variable haben aussagekräftige Namen erhalten aus denen sich ablesen lässt für was sie gut sind.

Im Setup finden wir noch die Zeile *Serial.begin(9600)* . Eigentlich brauchen wir das nicht, ich schreibe es aber immer schon mal mit dazu dann kann man sich zum Finden eines Fehlers schnell mal ein *Serial.print()* an geeigneter Stelle mit einbauen.

Im Loop finden wir ein *else*, das zu einem *If* etwas weiter oben gehört. wer nicht weiß was das ist und wie das funktioniert sollte hier jetzt erst mal eine Pause einlegen und im folgenden link weiter lesen, üben. reference/de/language/structure/control-structure/else/

Übrigens sprechen Anfänger immer schon mal von If Schleifen. If ist keine Schleife sondern eine Verzweigung. So jetzt geht’s aber los.

Zu diesem Abschnitt gehört der Sketch fade1.ino .

Den Sketch für zweite LED erweitern

Nun, das ist eigentlich ganz einfach. Wir erweitern unseren Grund-Sketch um die Festlegung der beiden neuen Pins, für den zweiten neuen Taster und die grüne LED. Dann benötigen wir noch eine neue Variable für die Helligkeit der grünen LED.

Das was bisher im loop steht können wir mit copy&past nochmal hinzufügen. Anschließend müssen wir noch die Namenszusätze der Variablen in dem neuen, zweiten Teil von Rot auf Gruen ändern. Wichtig dabei : verwendet keine Umlaute der Compiler mag das nicht.

Der ein oder andere Leser wir sich nach dem Test des Sketches fade2 am Ziel seiner Wünsche sehen. Schließlich haben wir jetzt einen Sketch der die Aufgabe augenscheinlich gut erfüllt. Mit je einem Taster wird eine LED langsam heller und dunkler und das unabhängig voneinander. Wenn wir allerdings genau hinschauen werden wir feststellen das die Zeit des Fade Vorganges nicht konstant ist. Laufen gerade beide Vorgänge gleichzeitig kann es bis zu doppelt so lange dauern bis der Vorgang abgeschlossen ist. Nun wie kommt das denn ? Wenn wir uns den Programmablauf

genau ansehen stellen wir fest das *delay()* in einem Umlauf des Loop zweimal bearbeitet wird wenn beide fade Vorgänge gleichzeitig laufen. Damit ist die Umlaufzeit des Loop nicht konstant und damit die Geschwindigkeit des Fade Vorganges auch nicht.

Nun kann man auf die Idee kommen die vier einzelnen *delay()* raus zu nehmen und durch ein einziges am ende zu ersetze. Probiert es aus, das funktioniert und macht genau was es soll, die fade Zeit ist jetzt konstant, da die Loop Zeit immer konstant ist. Allerdings hängt der Controller bei jedem Umlauf in dem *delay()* in einer Pause fest und macht sonst nichts als warten. Er wird eigentlich blockiert durch das *delay()*. Für unser Beispiel spielt das keine Rolle. Wenn man sich allerdings vorstellt das der Controller gleichzeitig auch noch was anderes machen soll was zeitkritisch ist , dann wird das so nichts werden.

Zu diesem Abschnitt gehört der Sketch fade2.ino

Delay() durch millis() ersetzen

Mit dem bisher gelernten schaut Euch mal das Beispiel „Blink“ aus der IDE an und denkt mal darüber nach wie Ihr zu der dauernd Blinkende LED mit einem zusätzlichen Taster eine zweite LED einschaltet solange der Taster gedrückt wird. Wer möchte kann das gerne mal versuchen. Bestenfalls bekommt Ihr es so hin das die zweite LED an oder aus geht wenn auch die Este LED ein oder aus geschaltet wird. Wenn man nur Kurz auf den Taster drückt passiert meist gar nichts. Warum ist das so ? Nun es liegt an dem verwendeten *delay()*. Damit wird der Programmablauf angehalten und es geht erst dann weiter wenn die Zeit abgelaufen ist. Der Controller kann nicht feststellen ob inzwischen der Taster gedrückt wurde weil der Programmablauf durch das *delay()* blockiert wurde.

Wenn man das blockierende *delay()* ersetzen will, und mal die Suchmaschine anwirft , wird man relativ schnell bei dem Beispiel „BlinkWithoutDelay“ und der Nachtwächter Erklärung hier im Forum landen. Jeder der sich ein bisschen ernsthaft mit der Arduino Welt beschäftigen will, sollte das verstanden haben und morgens im Halbschlaf anwenden können. Was *millis()* nun eigentlich ist findet man in der Referenz z.B hier Arduino Reference oder auch über die Hilfe in der IDE.

Es gibt sicher auch tausende Erklärungen im Netz dazu wie man *millis()* einsetzt. Zudem gibt es etliche Bibliotheken die Timer Objekte bereitstellen und damit die Verwendung einfacher machen.

Was hat nun unser Aufgabe mit dem Nachtwächter zu tun? Lasst es mich mal so erklären. Wenn es dunkel wird (Taster gedrückt) dreht der Nachtwächter seine Runden. Eine Runde dauert 5 Minuten. Alle 15 Minuten soll er die Lampe kontrollieren. Immer wenn er an einer Lampe vorbeikommt schaut er auf seinem Zettel nach wann er das letzte mal hier kontrolliert hat. Wenn er feststellt das es Zeit ist wieder was zu tun, schreibt er die aktuelle Uhrzeit auf seinen Zettel und schaut nach ob die Lampe schon ganz hell ist. Wenn nein, berechnet er wie hell er die Lampe jetzt einstellen muss. Wenn er feststellt sie ist schon ganz hell, macht er nichts , ansonsten stell er den neuen Wert an der Lampe ein und geht weiter. Das macht er so die ganze Nacht. Wenn es nun hell wird(Taster nicht gedrückt) geht er ähnlich vor, nur das er die Lampe jedes mal etwas dunkler stellt wenn es an der Zeit ist. Jetzt stellen wir also fest, der arme Kerl ist eine arme Socke, er rennt da Tag und Nacht herum und hat fast nichts zu tun. Unserem Controller geht es aber ähnlich, der langweilt sich eigentlich auch. Ich hoffe ich hab Euch jetzt nicht komplett verwirrt , das war nicht meine Absicht.

Also, das oben geschriebenen kann man in Grunde in wenige Codezeile packen. Damit es zu keinen Überlauf oder Fehler kommt wenn *millis()* irgendwann, nach etwa 40 Tagen, mal wieder von vorne anfängt, hat sich die Abfrage ob was zu tun ist in folgenden Form durchgesetzt.

```
if(millis() - altzeit >= delaytime){      // ist die Zeit schon um
  altzeit = millis();                     // aktuelle Zeit merken
  // mach hier was immer zu tun ist
}
```

zudem sollten alle Werte, die irgendwie mit *millis()* in Verbindung kommen, vom Datentyp unsigned long oder uint32_t sein. Wie Ihr die Variable dabei benennt bleibt Euch überlassen. Für unsere beiden LED´s müssen wir das natürlich getrennt machen. Für jede Lampe müssen wir uns merken wann wir das letzte mal da waren. Klar das muss der Nachtwächter ja auch so machen wenn er seine Runde dreht.

```
uint32_t delaytime = 50;
uint32_t altzeitRot, altzeitGruen;
```

Wenn Ihr nicht genau wisst was es mit den Datentypen auf sich hat dann solltet Ihr das nochmals nachlesen.

Zusammnegefasst: Wenn unser Code nicht blockieren soll oder wenn wir mehrere Dinge gleichzeitig machen wollen können wir *delay()* nicht verwenden.

Zu diesem Abschnitt gehört der Sketch fade3.ino

Eine eigene Funktion erstellen

Unser Sketch verwendet zwei Teile die im Wesentlichen gleich sind. Sie unterscheiden sich lediglich dadurch das bei Variablennamen einmal Rot und einmal Gruen als Zusatz verwendet wird. Um Code mehrfach zu verwenden sind Funktionen bestens geeignet. Oft werden Funktionen auch verwendet um lediglich das Programm zu strukturieren. Häufig werden dann darin Globale Variable verwendet oder auch verändert. Das sieht dann oft so aus:

```
void funktionsname()
```

Es gibt weder Rückgabewert noch einen Übergabewert. Das widerspricht dem eigentlichen Grundgedanken, schließlich sollen Funktionen universell und mehrfach einsetzbar sein. Dennoch macht es viel Sinn zur Strukturierung des Sketches so vorzugehen. Programme lassen sich so besser lesen, insbesondere dann wenn der Funktionsname schon beschreibt was darin passiert.

Wenn man eine Funktion schreiben will, sollte man sich gut überlegen was mit in die Funkion rein soll und wie die Schnittstelle aussieht. In unserem Fall habe ich mich dazu entschieden drei Variable zu übergeben. Die beiden letzten davon kennen wir bereits.

```
void fadeLed(bool ein, uint32_t &altzeit, byte &hell)
```

das & Zeichen vor den beiden Variablen legt hier fest das die entsprechende Variable als Referenz übergeben wird, und damit von der Funktion geändert werden kann. Die Variable ist dann sowohl Übergabe als auch Rückgabewert. Zum Nachlesen *„Übergabe als Wert oder Referenz“* sind hier das Stichwort. Der Name der internen Variablen und der Name von Variablen im Funktionsaufruf haben nicht miteinander zu tun, sie können völlig unterschiedlich sein, können aber auch gleich sein. Ich nenne die interne Variable hier z.B altzeit, schließlich wird sie für beide LED benutzt. Beim Funktionsaufruf wir dann einmal altzeitRot und einmal altzeitGruen übergeben. Der aufrufende Code muss nach der Rückgabe sicherstellen das die Variablen bis zum nächsten mal gespeichert bleiben.

Die bool Variable „ein“ stellt den Status des Tasters dar, er muss nicht zurück gegeben werden, er wird also ohne & verwendet. Dazu gibt es noch etwas Neues. Wenn Wir bisher den Taster Status

benötigt haben, dann haben wir Ihn mit *digitalRead(pin)* eingelesen. Ab jetzt lesen wir den Taster nur einmal je Umlauf ein und speicher das Ergebnis in einen bool Variablen die wir dann an die Funktion übergeben. Das macht bei Statuswerten die nur 0/1 b.z.w true / false sein können eigentlich immer Sinn da sie sich einfacher abfragen und logisch verknüpfen lassen. So macht z.B. die Zeile

```
if( stat1 & stat2 & !stat3)
```

durchaus Sinn und lässt sich auch einfach lesen. Aber das ist ein ganz anderes Thema. Hier jetzt der fertige Sketch mit unserer Funktion.

Im Sketch fade4.ino habe wir die Funktionalität des Faden´s in einer Funktion mit dem Namen fadeLed zusammen gefasst. Diese Funktion können wir mehrfach verwenden. Wir übergeben beim Aufruf der Funktion die aktuellen Werte an die Funktion und bekommen von Ihr die neuen Werte zurück geliefert.

Zu diese Abschnitt gehört der Sketch fade4.ino

Jetzt erarbeiten wir noch eine etwas andere Funktion. Im obigen Beispiel ist es ja so, das der aufrufende Code die Werte der Variablen speichern muss. Schön wäre es wenn die Funktion das selber machen könnte und zudem einen Rückgabewert auf „normalem" Weg über *return* liefert würde. Dazu bietet sich die Möglichkeit an die Variablen innerhalb der Funktion als *static* zu deklarieren damit würden die Werte zwischen zwei Aufrufen nicht verloren gehen. Der aufrufende Code müsste dann mitteilen ob es sich um die Rote oder Grüne LED handelt. Damit das möglichst einfach geht werden wir die beiden Variablen in der Funktion in je einem Array speichern. Über den Index sprechen wir das richtige Element des Array an. Der aufrufende Code benötigt die Variablen nun nicht mehr. Da unsere Funktion jetzt einen richtigen Rückgabewert liefert können wir den Funktionsaufruf auch gleich in unserer Ausgabe verwenden.

```
 analogWrite(LedGruen, fadeLed(statgruen,1));
```

Zu diesem Abschnitt gehört der Sketch fade 4a.ino

Zusammengefasst: Funktionen geben uns die Möglichkeit unseren Code zu strukturieren und Code mehrfach zu verwenden. Die Möglichkeiten zur Speicherung von Daten für einen mehrfachen Aufruf sind jedoch begrenzt.

Wer hier noch etwas üben will kann sich ja mal überlegen wie man die Funktion so ändern kann das das Einlesen des Tasters und die Ausgabe an die LED mit in der Funktion ist. Als Parameter müsste dann der Pin für Taster und LED an die Funktion übergeben werden.

Objekte und Methode

Tatsächlich wurden Konstrukte die Daten innerhalb der Funktion speichern früher häufig verwendet, bis schlaue Leute auf die Idee kamen Klassen zu erfinden. Mit Klassen ist es möglich den obigen Nachteil von Funktionen zu umgehen. Datenspeicher und Funktionen werden hier zusammengeführt. In das Thema Klassen, Objekte, Methoden müsst Ihr Euch wieder selber einlesen.

Ich zeige Euch wie das in unserem Beispiel verwendet werden kann.

Wir benötigen also eine Klasse die unsere Daten aufnehmen und speichern kann und eine Methode die die gewünschte Funktionalität zur Verfügung stellt. Etwas vereinfacht könnte man sagen Funktionen heißen jetzt Methoden.

Zur Erzeugung einer Klasse wird am Anfang das Schlüsselwort *class* verwendet und am Ende eine geschweifte Klammer mit einem Semikolon. Innerhalb wird festgelegt was privat und was public ist. In unserem Beispiel soll der Datenspeicher für die Variablen privat sein , die Methode muss public sein damit sie von außen zugänglich ist.

Zusätzlich benötigen wir noch einen Konstruktor der bei der Erstellung einer Instanz automatisch bearbeitet wird. Er kümmert sich zur Laufzeit um das Anlegen der Daten und in unserem Fall um die Übernahme der globalen Werte fadstep und delaytime.

Wir erzeugen uns also zwei Objekte der Klasse Fade mit unterschiedlichem Namen und übergeben unsere Werte. Jedes Objekt stellt uns dann seine Eigenschaften und Methoden zur Verfügung .

```
Fade LEDrot(fadstep,delaytime), LEDgruen(fadstep,delaytime);// zwei Objekte
erzeugen
```

Auf die Methode eines Objektes wird über deren Namen und einen Punkt zugegriffen. Ähnlich wie bei einer Struktur (struct).

```
byte wert = LEDrot.fade(statrot);
```

eine Methode kann wie eine Funktion einen Wert zurück liefern. Für die Übergabeparameter gelten die gleichen Bedingungen wie bei Funktionen.

Zusammengefasst: Wenn wir einen Code mehrfach nutzen wollen und gleichzeitig für jeden Aufruf einen Datenspeicher benötigen ist der Einsatz von Objekten und Methoden genau richtig.

Zu diesem Abschnitt gehört der Sketch fade5.ino

An dieser Stelle bedanke ich mich bei Tommy56 der mich als Co-Moderator unterstützt, und mir besonders beim Thema Klassen auf die Sprünge geholfen hat.

Jetzt hoffe ich, ich habe mich verständlich ausgedrückt, es hat Euch Spas gemacht, und Ihr habt was gelernt.