Vorwort

Diese Einführung in die Programmierung eines Arduino-basierenden Systems wie den MAKERbuino soll lediglich Geschmack auf mehr machen. Sie erhebt nicht den Anspruch auf Vollständigkeit. Beim Programmieren kommt man nicht wirklich um die englische Sprache herum. Viele Begriffe und vor allem die Befehle sind englisch. Daher wird hier versucht die Zusammenhänge zwischen den englischen Begriffen herzustellen. Es ist wichtig, dass man diese Begriffe versteht, damit man versteht was passiert und wo man eingreifen muss.

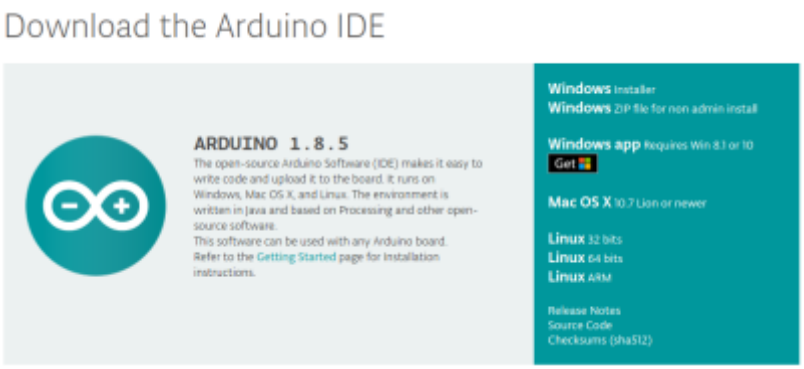
Viel Spaß damit

Die Entwicklungsumgebung

Um Programme für den MAKERbuino oder andere Arduino-basierten Systems schreiben zu können benötigt man ein paar Programme. Zunächst benötigt man einen sogenannten Editor in dem man seinen Quelltext oder in Englisch `Sourcecode` schreiben kann. Ein Editor ist also ein Programm in dem man Texte schreiben kann. Das ist der Ausgangspunkt für jedes Programm, dass man schreiben möchte. Der `Sourcecode` ist die Beschreibung der eigentlichen Abläufe im Programm. Er wird in einer für uns Menschen „verständlichen“ Sprache der sogenannten Programmiersprache geschrieben. Leider kann der Prozessor diese Sprachen nicht direkt verstehen. Er spricht nur „binären Op-Code“ [1)]. Damit der Prozessor trotzdem weiß, was er zu tun hat, muss der `Sourcecode` in binären Op-Code übersetzt werden. Diese Aufgabe übernimmt der ogenannte `Compiler`. Er versteht den `Sourcecode` und kann diesen dann in die nötigen Op-Code-Befehle, die der Prozessor benötigt übersetzen.

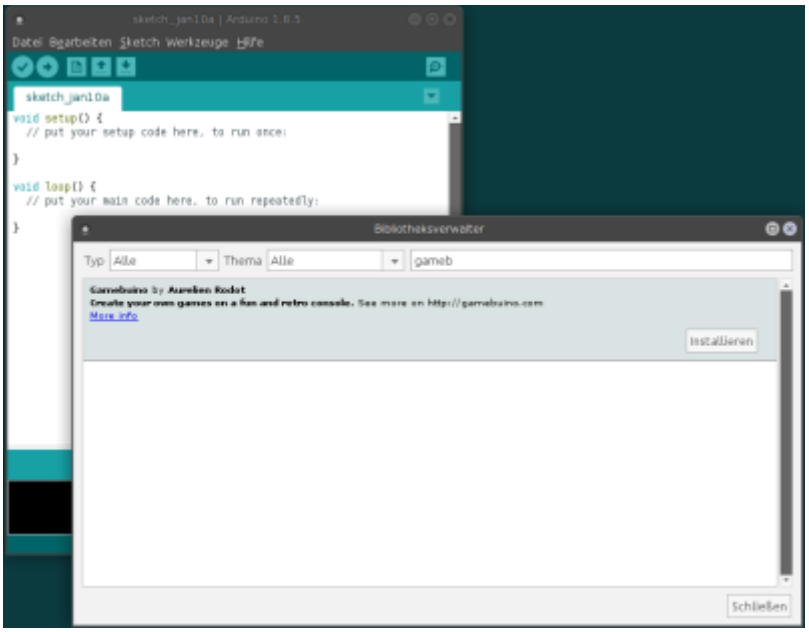
Es gibt noch weitere Helfer, aber dazu später mehr. Zunächst müssen wir also einen Editor und einen Compiler für den verwendeten Prozessor, den **Arduino**, installieren. Nun könnten wir alles einzeln installieren oder wir verwenden eine sogenannte IDE [2)]

Als IDE verwenden wir die `Ardunio-IDE` die **hier heruntergeladen** werden kann. Aktuell ist die Version 1.8.5 (Stand: Januar 2018). Im zweiten Abschnitt befinden sich die Download-Links für die verschiedenen Betriebssysteme. Weg zum Download: `arduino.cc → Software`



Die Arduino-IDE wird wie jedes Programm einfach installiert. Anschließend kann die Arduino-IDE gestartet werden. Es sollte der Dialog erscheinen, der `sketch_… Arduino 1.8.5` im Titel anzeigt. Damit wir für den MAKERbuino einer speziellen Form des Arduino Programme entwickeln können, benötigen wir noch eine sogeannte Bibliothek oder im Englischen `library` in der spezielle bereits vorbereitete Programmteile zusammengefasst sind, die uns das Leben mit dem MAKERbuino

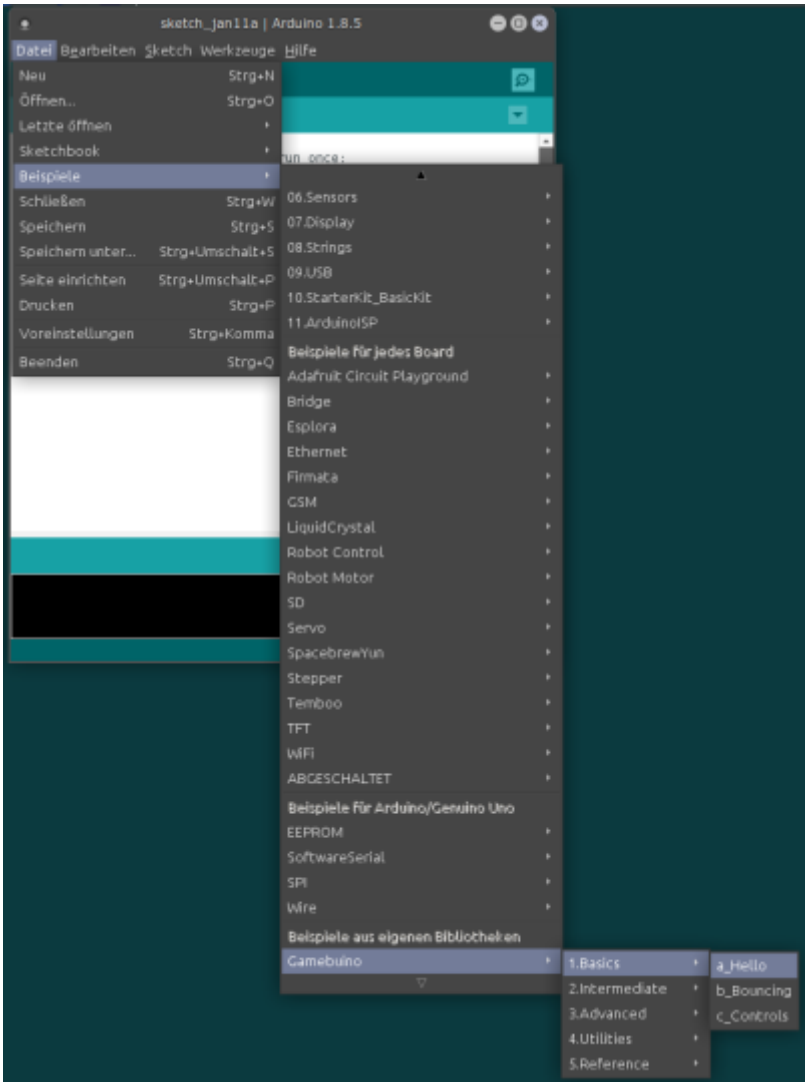
vereinfachen. Der MAKERbuino ist kompatibel, also verhält sich genauso, zum Gamebuino. Daher benötigen wir die Bibliothek des Gamebuino. Diese kann einfach nachinstalliert werden. Über `Sketch → Bibliothek einbinden → Bibliotheken verwalten…` Hier kann im oberen rechten Bereich `gamebuino` eingegeben werden. Es sollte dann wie im Bild unten eine kurze Information zur Bibliothek angezeigt werden. Klickt man in diesen Bereich, dann erscheint ein `Installieren`-Button, den man anklickt. Nach einer kurzen Installation steht im oberen Bereich nun `INSTALLED`.



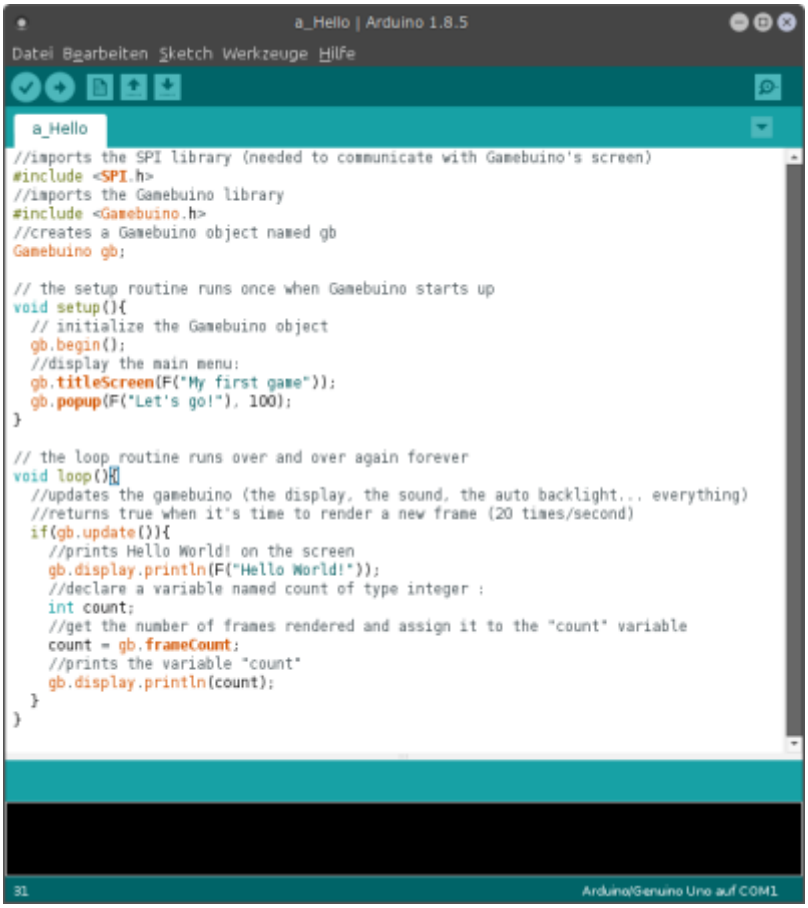
WICHTIG: Nach der Installation der Bibliothek muss die Arduino-IDE ggf. neugestartet werden, damit die neue Bibliothek eingebunden wird.

Das erste Programm "Hello World!"

Jetzt geht es an das erste Programm. Hierzu öffnen wir ein Beispielprogramm des Gamebuino.



Folgender Sourcecode sollte erscheinen:



Kurze Erklärung der Abschnitte (kann übersprungen werden)

Im Folgenden werden die Grundlagen einer Sourcecode-Datei erklärt.

Der erste Abschnitt: Einbinden von fremdem Sourcecode

Der erste Abschnitt des Sourcecodes bindet alle notwendigen Bibiliotheken ein. Dies geschieht mit dem Befehl `#include`[3]. Das Zeichen # gibt dem Compiler eine Anweisung, also ist `include` kein Befehl, der in binäre Code für den Prozessor übersetzt wird, sondern lediglich an den Compiler selbst gerichtet ist.

In spitzen Klammern wird nun die sogenannte `Header`-Datei der Bibliothek angegeben. In diesen `Header`-Dateien erklärt man normalerweise welche Variablen bzw. Funktionen innerhalb einer Bibliothek zur Verfügung stehen. Durch das `#include` wird der Inhalt genau diese `Header`-Dateien während der Übersetzung des Sourcecodes in den eigenen Sourcecode „kopiert“. Damit wird alles aus den entsprechenden Bibliotheken im eigenen Sourcecode nutzbar ohne den fremden Sourcecode tatsächlich immer kopieren zu müssen.

Hier werden also die beiden Bibliotheken `SPI`[4] zum nutzen der Programmierschnittstelle und `Gamebuino` für den MAKERbuino selbst eingebunden.

Weiterhin wird hier noch die Variable `gb` definiert. Vor dieser Variable steht der sogenannte `Type` der

Variable. Hier `Gamebuino`. Dieser Variablentype wird in der Heade-Datei `Gamebuino.h` definiert und enthält alles Wichtige um den MAKERbuino zu programmieren. Die Variable existiert zwar zu diesem Zeitpunkt, aber sie kann noch nicht genutzt werden. Um dies zu tun muss sie zunächst initialisiert werden, also quasi mit Leben gefüllt werden. Das heißt das Gerüst der Variablen existiert schon, aber der Inhalt ist noch leer.

Der zweite Abschnitt: Der Sourcecode

Im zweiten Abschnitt kommt der eigentliche Sourcecode. Hier sind zwei sogenannte Funktionen (`void setup(){…}` und `void loop(){…}`) definiert.

Um zu verstehen was hier passiert, muss man wissen, dass es bei vielen Systemen festdefinierte Funktionen gibt, die bestimmte Aufgaben haben. Die beiden `setup` und `loop` gehören dazu.

Alle Funktionen haben denselben Aufbau:

```
RÜCKGABEWERT FUNKTIONSNAME(EINGANGSWERTE){
BEFEHLE, die innerhalb der Funktion abgearbeitet werden
}

Beispiel:
int myfunct(int var1) {
    print(var1);
    var1=var1+1;
    return var1;
}
```

Die Funktion heißt hier `myfunct` sie erwartet als Eingangswert[5] einen `int`-Wert[6] und gibt ebenfalls einen `int`-Wert zurück hier `return var1`.

Bei `void setup()` bedeutet `void` in diesem Zusammenhang `kein Rückgabewert` und die leeren runden Klammern bedeuten `kein Eingangswert`.

setup(): Die Initialisierung des Systems

Die `setup()`-Funktion wird genau einmal beim Programmstart durchlaufen und dient dazu die Grundkonfiguration des Programms vorzunehmen. Hier werden meist die globalen Variablen wie `gb` initialisiert, also mit Werten befüllt `gb.begin()`.

Auch kann man hier z.B. den Titelbildschirm `gb.titleScreen` einblenden lassen oder sogenannte popup-Nachrichten `gb.popup` (eingeblendete Nachrichten).

Wichtig: Hier gehört nicht hinein, dass mehrfach aufgeführt werden sollen. Also irgendwelche Steuerungen oder sonstige Programmteile. Ansonsten passiert alles genau einmal und danach passiert nichts mehr.

loop(): Die eigentliche Hauptroutine

Die `loop()`-Funktion ist die Hauptroutine, die immer und immer wieder durchlaufen wird. Das bedeutet das alle Befehle, die innerhalb dieser Funktion stehen immer wieder durchlaufen werden. Also sollte darüber gut nachgedacht werden, was hier passieren soll. Es wird ansonsten sehr oft wiederholt.

Im Kommentar wird erklärt, dass der Gamebuino 20 mal in der Sekunde ein `update` als eine Aktualisierung durchführt. Innerhalb dieses `updates` wird der Bildschirm, die Soundausgabe, die Hintergrundbeleuchtung usw. aktualisiert. Darum müssen wir also glücklicherweise nicht kümmern (sofern alles wie gewünscht funktioniert). Während eines solchen Aktualisierungszyklus sollte man keine Änderungen durch das eigene Programm durchführen. Dies könnte zu unschönen Ergebnissen führen z.B. ist der obere Teil des Bildschirms schon aktualisiert worden und kurz danach wird ein Wert für diesen Teil geändert.

Damit also nicht permanent etwas geschieht, kann man darauf warten, dass ein solcher `update`-Prozess durchlaufen wurde und im Anschluss einige Befehle abarbeiten. Das Objekt `gb` hat hierzu eine Funktion[7)] definiert, die mit `gb.update()` aufrufen werden kann. Der `.`-Punkt zwischen `gb` und `update()` deutet daraufhin, dass die Funktion `update()` ein Teil des Objektes `gb` ist.

Mit Hilfe der Funktion `gb.update()` lässt darauf warten, dass der Gamebuino einen `update`-Zyklus beendet hat, in dem man innerhalb der `loop`-Funktion mit Hilfe einer `if`-Anweisung auf den Rückgabe werden `true` prüft.

```
if (gb.update()){
  ...
}
```

Ausprobieren des ersten Programms

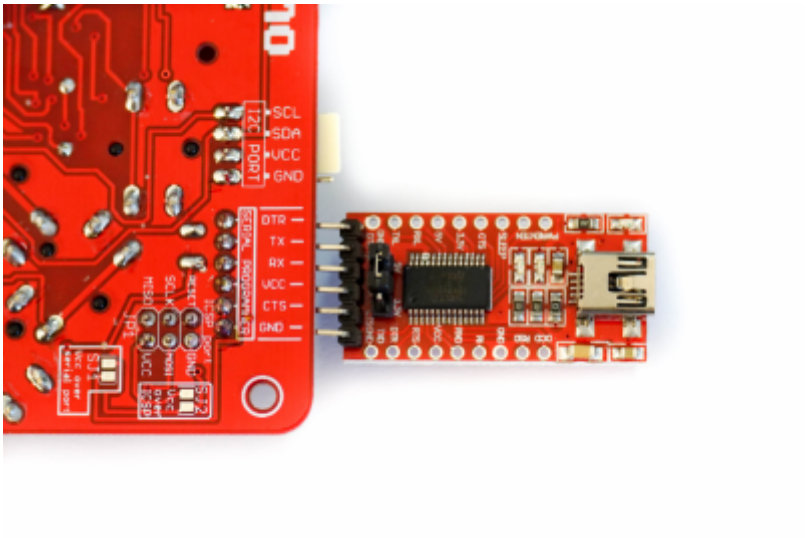
Der Sourcecode alleine ist zunächst wie oben erklärt nicht auf dem MAKERbuino lauffähig. Daher muss er in den binären Op-Code vom Compiler übersetzt werden anschließend ausgeführt werden. Hierzu gibt es mehrer Möglichkeiten:

1. per RS232-zu-USB-Konverter den erzeugten Op-Code direkt auf dem MAKERbuino übertragen
2. den erzeugten Op-Code auf die SDCard kopieren und anschließend den MAKERbuino starten
3. den erzeugten Op-Code in einem Software-Emulator[8)] ausführen

Mit RS232-zu-USB-Konverter Programme auf den MAKERbuino übertragen

Jede der drei Möglichkeiten hat Vor- und Nachteile. Bei kleinen Programmen ist die erste Option am schnellsten. Hier wird der Op-Code direkt auf die Konsole übertragen und ausgeführt. Ein Nachteil dabei ist allerdings die Übertragungszeit, die bei größeren Programmen sehr lange dauern könnte.

Um diese Möglichkeit zu nutzen wird die beiliegende RS232-zu-USB-Platine benötigt.



Im obigen Bild wurde der Konverter direkt an den MAKERbuino angeschlossen. Man kann so die Ausrichtung der beiden Platinen zu einander sehen. Dies ist wichtig, da jeweils der TX-Pin [9] mit dem RX-Pin

Dies ist zwar prinzipiell möglich, sollte allerdings nur in Ausnahmen getan werden. Es besteht hier die Gefahr, dass die Stiftleiste verbiegt oder die Konverterplatine abgebrochen wird, da sie mehr oder weniger fest mit dem MAKERbuino verbunden ist.

Die bessere

1)
Op-Code: Operationcode, also Befehle, die durch Zahlenwerte dargestellt werden
2)
IDE: **i**ntegrated **d**evelopment **e**nviroment; integrierte Entwicklungsumgebungen enthalten alle notwendigen Programme wie Editor, Compiler, Linker etc. und sind meist sehr komfortabel
3)
include: Binde ein; macht z.B. eine fremde Bibliothek im eigenen Sourcecode nutzbar
4)
SPI: **s**erial **p**rogramming **i**nterface, serielle Programmierschnittstelle
5)
an sie gerichtet
6)
int: **int**eger; ganzzahlige Zahl zB. -3, 2 oder 5
7)
Methode
8)
Emulator: bildet die Hardware als Programm nach und führt die Befehle so aus wie die Hardware
9)
TX: transmit; Sende-Pin