Bedienungsanleitung

Dr.-Ing. Till Harbaum

18. Juni 2018

Für Tanja, Maya, Fabian und Ida

© 2017 Dr.-Ing. Till Harbaum <till@harbaum.org>

Projekt-Homepage: http://ftduino.de
Kontakt: mailto://info@ftduino.de
Forum: https://forum.ftcommunity.de/

Inhaltsverzeichnis

1 Einleitung **8**
1.1 Das ftDuino-Konzept . . . 8
1.1.1 Das fischertechnik-Baukastensystem . . . 8
1.1.2 Das Arduino-System . . . 9
1.2 Der ftDuino-Controller . . . 10
1.2.1 Mikrocontroller . . . 10
1.2.2 USB-Anschluss . . . 12
1.2.3 Reset-Taster . . . 12
1.2.4 Interne LEDs . . . 12
1.2.5 Spannungsversorgung . . . 12
1.2.6 Anschlüsse . . . 13
1.2.7 Hinweise für Arduino-erfahrene Nutzer . . . 16
1.3 Problemlösungen . . . 16
1.3.1 Die grüne Leuchtdiode im ftDuino leuchtet nicht . . . 16
1.3.2 Der ftDuino taucht am PC nicht als `COM:`-Port auf . . . 16
1.3.3 Der ftDuino funktioniert, aber die Ausgänge nicht . . . 16

2 Installation **18**
2.1 Treiber . . . 18
2.1.1 Windows 10 . . . 18
2.1.2 Windows 7 und Windows Vista . . . 18
2.1.3 Linux . . . 20
2.2 Arduino-IDE . . . 22
2.2.1 Installation mit dem Boardverwalter . . . 22
2.2.2 Updates . . . 23

3 Erste Schritte **24**
3.1 Der erste Sketch . . . 24
3.1.1 Download des Blink-Sketches auf den ftDuino . . . 25
3.1.2 Die Funktionsweise des Sketches . . . 26
3.1.3 Die Funktionen `setup()` und `loop()` . . . 26
3.1.4 Anpassungen am Sketch . . . 26
3.2 Ansteuerung von fischertechnik-Komponenten . . . 27
3.2.1 Der Sketch . . . 27
3.2.2 Eingänge . . . 28
3.3 Kommunikation mit dem PC . . . 29
3.3.1 Der serielle Monitor . . . 30
3.3.2 Sketchbeschreibung . . . 31
3.3.3 USB-Verbindungsaufbau . . . 31

4 Programmierung **32**
4.1 Textbasierte Programmierung . . . 32
4.2 Die Programmiersprache C++ . . . 34
4.3 Grundlagen . . . 34
4.3.1 Kommentare . . . 35
4.3.2 Fehlermeldungen . . . 36

4.3.3 Funktionen . . . 36
4.3.4 Die Funktionen `setup()` und `loop()` . . . 37
4.3.5 Beispiel . . . 39
4.4 Hilfreiche Bibliotheksfunktionen . . . 39
4.4.1 `pinMode(pin, mode)` . . . 39
4.4.2 `digitalWrite(pin, value)` . . . 40
4.4.3 `delay(ms)` . . . 40
4.4.4 `Serial.begin(speed)` . . . 40
4.4.5 `Serial.print(val)` und `Serial.println(val)` . . . 41
4.4.6 `ftduino.input_get()`, `ftduino.output_set()` und `ftduino.motor_set()` . . . 41
4.5 Variablen . . . 42
4.5.1 Datentyp `int` . . . 43
4.6 Bedingungen . . . 44
4.6.1 `if`-Anweisung . . . 44
4.7 Schleifen . . . 45
4.7.1 `while`-Schleife . . . 45
4.7.2 `for`-Schleife . . . 46
4.8 Beispiele . . . 46
4.8.1 Einfache Ampel . . . 47
4.8.2 Schranke . . . 47
4.9 Weiterführende Informationen . . . 49

5 Experimente **50**
5.1 Lampen-Zeitschaltung . . . 50
5.1.1 Sketch `LampTimer` . . . 50
5.2 Not-Aus . . . 52
5.2.1 Sketch `EmergencyStop` . . . 52
5.3 Pulsweitenmodulation . . . 55
5.3.1 Sketch `Pwm` . . . 55
5.4 Schrittmotoransteuerung . . . 59
5.4.1 Vollschrittsteuerung . . . 61
5.4.2 Halbschrittsteuerung . . . 62
5.5 Servomotoransteuerung . . . 64
5.5.1 Externe 6-Volt-Versorgung . . . 65
5.6 Die Eingänge des ftDuino . . . 67
5.6.1 Spannungsmessung . . . 67
5.6.2 Widerstandsmessung . . . 67
5.6.3 Ein Eingang als Ausgang . . . 68
5.7 Temperaturmessung . . . 68
5.7.1 Sketch `Temperature` . . . 69
5.8 Ausgänge an, aus oder nichts davon? . . . 71
5.8.1 Sketch `OnOffTristate` . . . 71
5.8.2 Leckströme . . . 72
5.9 Aktive Motorbremse . . . 72
5.10 USB-Tastatur . . . 74
5.10.1 Sketch `USB/KeyboardMessage` . . . 74
5.11 USB-GamePad . . . 75
5.11.1 Sketch `USB/GamePad` . . . 76
5.12 Entprellen . . . 77
5.12.1 Sketch `Debounce` . . . 77
5.13 Nutzung des I^2C-Bus . . . 80
5.13.1 Sketch `I2C/I2cScanner` . . . 81
5.13.2 MPU-6050-Sensor . . . 81
5.13.3 OLED-Display . . . 82
5.13.4 VL53L0X LIDAR-Distanzsensor . . . 83
5.13.5 ftDuino als I^2C-Client und Kopplung zweier ftDuinos . . . 84
5.14 WS2812B-Vollfarb-Leuchtdioden . . . 90
5.14.1 Sketch `WS2812FX` . . . 91

5.15 Musik aus dem ftDuino . . . 91
5.15.1 Sketch Music . . . 92
5.15.2 Sketch MusicPwm . . . 92
5.16 Der ftDuino als MIDI-Instrument . . . 93
5.16.1 Sketch MidiInstrument . . . 93

6 Modelle **94**
6.1 Automation Robots: Hochregallager . . . 94
6.2 ElectroPneumatic: Flipper . . . 95
6.3 ROBOTICS TXT Explorer: Linienfolger . . . 96
6.4 Idas Ampel . . . 97
6.4.1 Zustandsautomaten . . . 98

7 Community-Projekte **100**
7.1 ftduino_direct: ftDuino-Anbindung per USB an TXT und TX-Pi . . . 100
7.2 ftDuinIO: ftDuino-Kontroll-App für TXT und TX-Pi . . . 101
7.3 Brickly-Plugin: Grafische ftDuino-Programmierung in Brickly . . . 102
7.4 startIDE: Programmierung direkt auf dem TX-Pi oder TXT . . . 102

8 Bibliotheken **104**
8.1 FtduinoSimple . . . 105
8.1.1 Verwendung im Sketch . . . 105
8.1.2 bool input_get(uint8_t ch) . . . 106
8.1.3 bool counter_get_state(uint8_t ch) . . . 106
8.1.4 void output_set(uint8_t port, uint8_t mode) . . . 106
8.1.5 void motor_set(uint8_t port, uint8_t mode) . . . 107
8.1.6 Beispiel-Sketches . . . 107
8.2 Ftduino . . . 107
8.2.1 Die Eingänge I1 bis I8 . . . 107
8.2.2 void input_set_mode(uint8_t ch, uint8_t mode) . . . 108
8.2.3 uint16_t input_get(uint8_t ch) . . . 108
8.2.4 Die Ausgänge O1 bis O8 und M1 bis M4 . . . 108
8.2.5 void output_set(uint8_t port, uint8_t mode, uint8_t pwm) . . . 109
8.2.6 void motor_set(uint8_t port, uint8_t mode, uint8_t pwm) . . . 109
8.2.7 void motor_counter(uint8_t port, uint8_t mode, uint8_t pwm, uint16_t counter) . . . 110
8.2.8 bool motor_counter_active(uint8_t port) . . . 110
8.2.9 void motor_counter_set_brake(uint8_t port, bool on) . . . 110
8.2.10 Die Zählereingänge C1 bis C4 . . . 111
8.2.11 void counter_set_mode(uint8_t ch, uint8_t mode) . . . 111
8.2.12 uint16_t counter_get(uint8_t ch) . . . 111
8.2.13 void counter_clear(uint8_t ch) . . . 111
8.2.14 bool counter_get_state(uint8_t ch) . . . 112
8.2.15 void ultrasonic_enable(bool ena) . . . 112
8.2.16 int16_t ultrasonic_get() . . . 112

9 Selbstbau **113**
9.1 Erste Baustufe „Spannungsversorgung“ . . . 113
9.1.1 Bauteile-Polarität . . . 114
9.1.2 Kontroll-Messungen . . . 114
9.2 Zweite Baustufe „Mikrocontroller“ . . . 115
9.2.1 Funktionstest des Mikrocontrollers . . . 115
9.3 Dritte Baustufe „Eingänge“ . . . 116
9.4 Vierte Baustufe „Ausgänge“ . . . 117

A Schaltplan **118**

B Platinenlayout **119**

C Bestückungsplan **120**

D Maße **121**

E Gehäuse **122**

Kapitel 1

Einleitung

Elektronik- und Computermodule für Konstruktionsbaukästen gibt es seit den Anfängen der privat genutzten Heimcomputer der 80er Jahre. Diese Module verfügten über wenig eigene Intelligenz und waren vor allem für die Signalanpassung zwischen dem Heimcomputer und den Motoren und Schaltern der Baukastensysteme zuständig, weshalb diese Module in der Regel als "Interfaces" bezeichnet wurden, also als Schnittstelle zwischen Computer und Modell.

Über die Jahre stieg die Leistungsfähigkeit der Heimcomputer und auch die Elektronik-Module lernten dazu. Vor allem wurden aus "Interfaces" über die Zeit "Controller". Aus den weitgehend passiven Schnittstellen wurden Bausteine mit eigener Intelligenz, die den Heimcomputer bzw. später den PC nur noch zur Programmierung benötigten. Einmal programmiert konnten diese Controller das Modell auch eigenständig bedienen. Dazu wurden die auf dem PC entwickelten Programmdaten auf den Conroller geladen und dort gespeichert.

Die heutigen Controller von Lego oder fischertechnik sind selbst leistungsfähige Computer. Um deren Komplexität für den Endanwender benutzbar zu machen verbergen die Hersteller die Details der elektronischen Komponenten sowie der auf den Geräten laufenden Software hinter gefälligen Benutzeroberflächen. Leider verpassen solche Systeme auf diese Weise die Chance, Wissen über Aufbau und Funktion derartiger Controller zu vermitteln. Während sich die Hersteller gegenseitig darin übertreffen, komplexe mechanische Getriebe im Wortsinne begreifbar zu machen stellen sich die dazugehörigen Controller für den Anwender als undurchsichtige Bausteine dar.

Parallel hat sich seit der Jahrtausendwende die sogenannte Maker-Bewegung entwickelt, die den "Selbstmach"-Gedanken in den Bereich der Elektronikentwicklung trägt. Systeme wie der Raspberry-Pi und der Arduino laden dazu ein, alle technischen Details dieser komplett zugänglichen und dokumentierten Controller zu erforschen und eigene Entwicklungen zu betreiben. Große Communities bieten umfangreiches Know-How und stellen Plattformen zum Wissensaustausch zur Verfügung. Im Gegensatz zu den Controllern von fischertechnik und Lego steht hier das Innere des Controllers im Vordergrund. Allerdings erfordert der Einsatz dieser Controller oft einges an handwerklichem Geschick beim Aufbau der Elektronik selbst sowie speziell bei Robotik-Projekten bei der Umsetzung von mechanischen Komponenten.

1.1 Das ftDuino-Konzept

Die Idee hinter dem ftDuino ist es, die Brücke zwischen zwei Welten zu schlagen. Auf der einen Seite integriert er sich mechanisch und elektrisch nahtlos in die Robotics-Serie der fischertechnik-Konstruktionsbaukästen. Auf der anderen Seite fügt er sich perfekt in das Arduino-Ökosystem zur Software-Entwicklung von eingebetteten Systemen ein.

1.1.1 Das fischertechnik-Baukastensystem

Fischertechnik ist ein technikorientertes Konstruktionsspielzeug. Der Schwerpunkt liegt auf Mechanik, Elektromechanik, Elektronik und zunehmend auch Robotik und der dafür nötigen Integration von informationsverarbeitenden Komponenten.

Fischertechnik selbst entwickelt und vertreibt seit den frühen 80er Jahren Elektronik-Module, die eine Verbindung zwischen Computer und mechanischem Modell ermöglichen bzw. über eigene Intelligenz verfügen. Die dabei zum Einsatz kommenden Steckverbinder sowie die Sensoren (Taster, Schalter, Lichtsensoren, ...) und Aktoren (Lampen, Motoren, Ventile, ...) sind über die Jahre zueinander kompatibel geblieben und lassen sich nach wie vor beliebig miteinander kombinieren.

Abbildung 1.1: Der ftDuino im 3D-gedruckten Gehäuseprototyp

Die letzten zwei Controller-Generationen (fischertechnik TX- und TXT-Controller) haben eine vergleichbare mechanische Größe und verfügen über eine vergleichbare Anzahl und Art von Anschlüssen zur Verbindung mit dem Modell. Die Modelle aller aktuellen Robotics-Baukästen sind auf diese Anschlüsse ausgelegt und untereinander kombinierbar.

(a) TX-Controller

(b) TXT-Controller

Abbildung 1.2: Original-Controller von fischertechnik

Beide Original-Controller verfügen über acht analoge Eingänge, acht analoge Ausgänge, vier schnelle Zählereingänge und einen I^2C-Erweiterungsanschluss.

Fischertechnik selbst vertreibt die PC-Software RoboPro zur visuellen Softwareentwicklung für die hauseigenen Controller. Der Einstieg in RoboPro ist relativ einfach und spricht bereits Kinder an. Die Grenzen von RoboPro sind aber schnell erreicht, wenn es um praxisnahe und inhaltlich anspruchsvolle Projekte in weiterführenden Schulen, Universitäten und der Berufsausbildung geht. In diesen Bereichen haben sich Systeme wie die Arduino-Plattform etabliert.

1.1.2 Das Arduino-System

Das Arduino-Ökosystem hat sich in den letzten Jahren zum De-Facto-Standard für den Einstieg und die semiprofessionelle Entwicklung und Programmierung von eingebetteten Systemen etabliert. Eingebettete Systeme sind in der Regel mecha-

nisch kleine Computer und informationsverarbeitende Module, die innerhalb einer Maschine Steuer- und Regelaufgaben übernehmen und immer häufiger auch mit der Außenwelt kommunizieren.

(a) Arduino-Entwicklungsumgebung (IDE)

(b) Arduino-Leonardo-Controller

Abbildung 1.3: Die Arduino-Entwicklungsumgebung und -Controller

Die Arduino-IDE ist eine übersichtliche und leicht zu bedienende Programmeroberfläche, die sich auf Windows-, Linux- und Apple-PCs nutzen lässt. Die zu programmierenden Zielgeräte wie zum Beispiel der Arduino-Leonardo sind kleine und kostengünstige Platinen, die per USB mit dem PC verbunden werden. Sie kommen üblicherweise ohne Gehäuse und stellen über Steckverbinder eine Vielzahl von Signalleitungen zum Anschluss von Sensoren und Aktoren zur Verfügung. Typische mit der Arduino-Plattform zu erledigende Aufgaben sind einfache Messwerterfassungen (Temperaturlogging, . . .) und Steueraufgaben (Jalousiesteuerungen, . . .).

Programme, die mit der Arduino-IDE geschrieben wurden, werden in der Arduino-Welt als sogenannte "Sketches" bezeichnet. Mit Hilfe der Arduino-IDE können passende Sketches für den ftDuino geschrieben und über das USB-Kabel direkt auf das Gerät hinutergeladen werden.

Auch für einfache Robotik-Experimente ist die Arduino-Plattform bestens geeignet. Schwieriger ist oft eine mechanisch befriedigende Umsetzung selbst einfachster Robotik-Projekte. Diese Lücke kann das fischertechnik-System schließen.

1.2 Der ftDuino-Controller

Der ftDuino-Controller wurde bewusst mechanisch und elektrisch an den TX- und den TXT-Controller angelehnt, um ihn ebenfalls direkt mit den aktuellen Robotics-Kästen kombinieren zu können. Gleichzeitig wurde er softwareseitig mit dem Arduino-System kompatibel gehalten.

1.2.1 Mikrocontroller

Das Herz des ftDuino ist ein Mikrocontroller des Typs ATmega32u4. Dieser Mikrocontroller wird von Microchip (ehemals Atmel) hergestellt und findet auch im Arduino-Leonardo Verwendung. Sketches, die für den Leonardo übersetzt wurden, sind oft direkt auf dem ftDuino lauffähig.

Der ATmega32u4-Controller ist ein Mitglied der sogenannten AVR-Famile, auf der die meisten Arduino-Boards basieren. Die AVR-Controller sind klein, günstig und benötigen zum Betrieb nur wenig weitere Bauteile. Ihr Speicher und ihre Rechenleistung reicht für den Betrieb sämtlicher fischertechnik-Modelle der Roboticsreihe deutlich aus.

Der ATmega32u4 verfügt über 32 Kilobytes nicht-flüchtigem Flash-Speicher, der als Sketch-Programmspeicher verwendet wird sowie 2,5 Kilobytes internem RAM-Speicher zur Datenspeicherung. Der Prozessortakt beträgt 16 Megahertz. Jeweils ein Sketch kann im Flash-Speicher permanent gespeichert werden und bleibt auch erhalten wenn der ftDuino von der Spannungsversorgung getrennt wird.

Der ATmega32u4 ist eines der wenigen Mitglieder der AVR-Familie, das direkte USB-Unterstützung bereits auf dem Chip mitbringt. Auf diese Weise ist der ftDuino sehr flexibel als USB-Gerät am PC einsetzbar.

Abbildung 1.4: Die Anschlüsse des ftDuino

Abbildung 1.5: Blockdiagramm des ftDuino

Bootloader

Der ftDuino wird mit einem im ATmega32u4 vorinstallierten sogenannten Caterina-Bootloader ausgeliefert. Dieses Programm belegt permanent vier der 32 Kilobytes Flash-Speicher des ATmega32u4 und kann nicht ohne weiteres gelöscht oder verändert werden.

Der Bootloader ermöglicht die Kommunikation mit dem PC und erlaubt es, dass der PC Programmdaten in den verliebenden 28 Kilobytes Flash-Speicher ablegen bzw. austauschen kann. Der Bootloader ermöglicht auf diese Weise das Hinunterladen von Sketches in den ftDuino.

Dass der Bootloader aktiv ist und nicht etwa gerade ein Sketch ausgeführt wird, ist am Zustand der internen LEDs erkennbar (siehe 1.2.4).

1.2.2 USB-Anschluss

Die Verbindung zum PC zur Programmierung und Datenübertragung wird über USB hergestellt. Der ftDuino verfügt über eine sogenannte Mini-USB-Buchse und wird über ein handelsübliches Mini-USB-Kabel mit dem PC verbunden.

Abbildung 1.6: Strom- und USB-Anschluss des ftDuino

1.2.3 Reset-Taster

Normalerweise kann die Arduino-IDE durch entsprechende Kommandos über den USB-Anschluss den Bootloader des ftDuino aktivieren, um einen neuen Sketch hinunterzuladen. Enthält ein hinuntergeladener Sketch aber Fehler, die eine normale Programmausführung verhindern, dann kann es passieren, dass die USB-Kommunikation während der normalen Programmausführung nicht funktioniert und die Arduino-IDE den ftDuino von sich aus nicht mehr ansprechen kann.

Für diesen Fall verfügt der ftDuino über einen Reset-Taster. Wird dieser gedrückt, dann wird der Bootloader zwangsweise aktiviert und die LEDs zeigen entsprechend den Start des Bootloaders an.

Ein mit einem fehlerhaften Sketch versehener ftDuino kann daher problemlos mit einem korrigierten Sketch versehen werden, indem kurz vor dem Hinterladen der Reset-Taster kurz gedrückt wird. Mehr Details dazu finden sich im Abschnitt 1.3.

1.2.4 Interne LEDs

Der ftDuino verfügt über je eine grüne und rote interne Leuchtdiode (LED). Die grüne Spannungsversorungs-LED zeigt an, dass der interne 5-Volt-Zweig mit Spannung versorgt ist und der Mikrocontroller des ftDuino versorgt wird.

Die rote LED steht für eigene Verwendung zur Verfügung und kann vom Anwender aus eigenen Sketches heraus unter der Bezeichnug `LED_BUILTIN` angespochen werden (siehe Abschnitt 3.1).

Die rote LED wird auch vom Caterina-Bootloader des ftDuino verwendet. Ist der Bootloader aktiv, so leuchtet die LED im Sekundentakt sanft heller und dunkler ("fading").

1.2.5 Spannungsversorgung

Der ftDuino kann auf vier Arten mit Spannung versorgt werden:

USB Über USB wird der ftDuino immer dann versorgt, wenn keine weitere Stromversorgung angeschlossen ist. Die USB-Versorgung reicht allerdings nicht zum Betrieb der Analogausgänge. Lediglich die Eingänge können bei USB-Versorung verwendet werden. Zusätzlich ist die Genauigkeit einer Widerstandsmessung an den Analogeingängen deutlich herab gesetzt (siehe 1.2.6).

Hohlstecker Wird der ftDuino per Hohlstecker z.B. durch das fischertechnik Power Netzgerät 505287[1] oder dem Netzteil aus dem fischertechnik Power-Set 505283[2] mit 9 Volt versorgt, so wird der gesamte ftDuino daraus versorgt und der USB-Anschluss wird nicht belastet. Die Analogausgänge sind in diesem Fall benutzbar und die Widerstandsmessung an den Analogeingängen erfolgt mit voller Genauigkeit. Für den Einsatz von Fremdnetzgeräten bietet fischertechnik unter der Artikelnummer 134863[3] einen Adapter von üblichen 5mm-Hohlsteckern auf den von fischertechnik verwendeten 3,45mm-Stecker an.

9V=-Eingang Eine Versorgung des ftDuino z.B. per Batterieset oder mit dem Akku aus dem Akku Set 34969[4] entspricht der Versorgung per Hohlstecker. Wird der ftDuino sowohl über den 9V=-Eingang als auch per Hohlstecker versorgt, dann erfolgt die Versorgung aus der Quelle, die die höhere Spannung liefert. Eine Rückspeisung in den Akku oder eine Ladung des Akkus findet nicht statt.

I²C Über den I²C-Anschluss versorgt der ftDuino in erster Linie andere angeschlossene Geräte wie kleine Displays oder Sensoren. Es ist aber auch möglich, ihn selbst über diesen Anschluss zu versorgen. Es bestehen dabei die gleichen Beschränkungen wie bei der Versorgung über USB. Auf diese Weise ist zum Beispiel die Versorgung zweier gekoppelter ftDuinos aus einer einzigen Quelle möglich (siehe Abschnitt 5.13.5).

Abbildung 1.7: 3,45mm fischertechnik-Hohlstecker

1.2.6 Anschlüsse

Die Anschlüsse des ftDuino teilen sich in die fischertechnik-kompatiblen Ein- und Ausgänge auf, die für die üblichen 2,6mm-Einzelstecker geeignet sind, sowie die üblichen Steckverbinder aus dem Computerbereich. Die fischertechnik-kompatiblen Ein- und Ausgänge sind identisch zum fischertechnik-TXT-Controller angeordnet. Verdrahtungsschemata für den TXT können daher in der Regel direkt übernommen werden.

Analoge Eingänge

Der ftDuino verfügt über acht analoge Eingänge `I1` bis `I8`, über die Spannungen von 0 bis 10 Volt sowie Widerstände von 0 bis über 10 Kiloohm erfasst werden können.

Die Eingänge sind über hochohmige Serienwiderstände gegen Kurzschlüsse sowie Über- und Unterspannung abgesichert.

Jeder Eingang ist mit einem eigenen Analogeingang des ATmega32u4-Mikrocontrollers verbunden. Die Analogwerterfassung kann mit bis zu 10 Bit Auflösung erfolgen (entsprechend einem Wertebereich 0 bis 1023) und wird direkt in der Hardware des Mikrocontrollers durchgeführt.

Ein Spannungsteiler erweitert den Eingangsspannungsbereich des Mikrocontrollers von 0 bis 5 Volt auf den bei fischertechnik genutzten Bereich von 0 bis 10 Volt. Alle an fischertechnik-Modellen auftretenden Spannungen können damit erfasst werden.

Zur Widerstandsmessung kann kann jeder Eingang ftDuino-intern mit einem Widerstand gegen 5 Volt verschaltet werden. Dieser Widerstand wirkt mit einem externen Widerstand als Spannungsteiler und aus der am Mikrocontroller gemessenen Spannung kann der Wert des extern angechlossenen Widerstands gemessen werden. Alle von fischertechnik-Modellen üblicherweise verwendeten Widerstände können so erfasst werden.

[1] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=505287
[2] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=505283
[3] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=134863
[4] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=34969

Die analogen Eingänge sind nicht auf eine externe 9-Volt-Versorgung angewiesen, sondern funktionieren auch bei der Stromversorgung über den USB-Anschluss des PC. Allerdings sinkt in diesem Fall die Genauigkeit der Widerstandsmessung signifikant.

Analoge Ausgänge

Der ftDuino verfügt über acht analoge Ausgänge `O1` bis `O8`. Diese Ausgänge werden über zwei spezielle Treiberbausteine im ftDuino angesteuert. Die Treiberbausteine können jeden der acht Ausgänge unabhängig steuern. Sie sind identisch zu denen, die fischertechnik in den TX- und TXT-Controllern einsetzt. Die Ausgänge sind daher kompatibel zu allen fischertechnik-Motoren und -Aktoren, die auch am TX- und TXT-Controller betrieben werden können. Der maximal pro Ausgang verfügbare Strom beträgt 600mA bis 1,2A.

Die Ausgänge sind bei einer reinen USB-Stromversorgung des ftDuino nicht verfügbar.

Der verwendete Treiberbaustein MC33879 ist kurzschlussfest und robust gegen Über- und Unterspannung an den Ausgängen.

Alle acht Ausgänge können unabhängig voneinander gegen Masse oder Eingangsspannung sowie hochohmig geschaltet werden. Je zwei Einzelausgänge können zu einem Motorausgang kombiniert werden. Die Einzelausgänge `O1` und `O2` bilden dabei den Motorausgang `M1`, `O3` und `O4` bilden `M2` und so weiter.

Die Analogwerte an den Ausgängen werden durch eine sogenannte Pulsweitemodulation (PWM) erzeugt. Dabei werden die Ausgänge kontinuierlich schnell ein- und ausgeschaltet, so dass Motoren, Lampen und andere träge Verbraucher dem Signal nicht folgen können, sondern sich entsprechend des Mittelwerts verhalten. Dieses Verfahren wird in gleicher Weise auch im TX- und TXT-Controller angewendet.

Beide MC33879 werden vom Mikrocontroller des ftDuino intern über dessen sogenannte SPI-Schnittstelle angeschlossen. Da dadurch die speziellen PWM-Ausgänge des Mikrocontrollers nicht zur Erzeugung der der Pulsweitenmodulation herangezogen werden können muss das PWM-Signal durch den Sketch bzw. die verwendeten Software-Bibliotheken (siehe Kapitel 8) selbst erzeugt werden. Die sogenannte PWM-Frequenz wird dabei durch den verwendeten Sketch bestimmt und kann beliebig variiert werden.

Mehr Informationen zum Thema PWM finden sich in Abschnitt 5.3.

Zählereingänge

Der ftDuino verfügt über vier spezielle Zählereingänge `C1` bis `C4`. Diese Eingänge können rein digitale Signale erfassen und mit hoher Geschwindigkeit Ereignisse auswerten. Die maximal erfassbare Signalrate liegt je nach Sketch bei mehreren 10.000 Ereignissen pro Sekunde.

Die Zählereingänge sind kompatibel zu den Encodern der fischertechnik-Encoder-Motoren und können unter anderem zur Drehwinkelauswertung sowie zur Drehzahlbestimmung herangezogen werden.

Zählereingang `C1` verfügt zusätzlich über die Möglichkeit, einen fischertechnik ROBO TX Ultraschall-Distanzsensor 133009[5] auszuwerten.

I²C-Anschluss

Der I²C-Anschluss ist elektrisch und mechanisch zu dem des fischertechnik-TX-Controllers kompatibel. Der dort aus dem Gerät herausgeführte sogenannte I²C-Bus findet auch im Arduino-Umfeld häufige Verwendung und erlaubt den Anschluss passender Elektronikkomponenten wie Sensoren, Analog-Digital-Wandler, Displays und ähnlich. Außerdem ist über den I²C-Bus eine Kopplung mehrerer ftDuinos möglich sowie die Kopplung des ftDuino mit dem TX-Controller und dem TXT-Controller wie in Abschnitt 5.13 beschrieben.

Die Signale auf dem I²C-Anschluss nutzen wie am TX-Controller einen 5-Volt-Pegel. Zusätzlich werden aus der Spannungsversorgung des ftDuino 5 Volt zur Versorgung angeschlossener Komponenten bereitgestellt. Aus dem 5 Volt-Ausgang dürfen max. 100mA entnommen werden, um die ftDuino-interne Spannungsversorung nicht zu überlasten

Achtung! Der fischertechnik-TXT-Controller sowie für den Betrieb am TXT vorgesehene Komponenten sind aufgrund dessen 3,3 Volt-Signal-Pegel nicht direkt mit dem ftDuino kompatibel. Eine dirkete Verbindung zwischen TXT und ftDuino kann

[5] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=133009

Abbildung 1.8: Anschluss des Ultraschallsensors 133009

Abbildung 1.9: Buchsen- und Kabelbelegung des I²C-Bus am ftDuino

den TXT beschädigen. Sollen der TXT oder für den Betrieb am TXT vorgesehene Komponenten am ftDuino verwendet werden, so sind unbedingt passende I²C-Pegelanpassung zwischenzuschalten wie in Abschnitt 5.13.5 beschrieben.

Achtung! Die auf dem I²C-Anschluss liegenden Signale sind direkt und ungeschützt mit dem Mikrocontroller des ftDuino bzw. mit dessen Spannungsversorgung verbunden. Werden an diesem Anschluss Kurzschlüsse verursacht oder Spannungen über 5V angelegt, dann kann der ftDuino zerstört werden. Der I²C-Anschluss sollte daher nur von erfahrenen Anwendern verwendet werden. Aus diesem Grund wird der ftDuino mit einer Schutzkappe auf dem I²C-Anschluss vertrieben. Diese Kappe ist bei Bedarf vorsichtig mit einem flachen Schraubendreher zu entfernen.

Abbildung 1.10: Entfernen der Schutzkappe vom I²C-Anschluss des ftDuino

1.2.7 Hinweise für Arduino-erfahrene Nutzer

Es gibt ein paar fundamentale Unterschiede zwischen dem klassischen Arduino und dem ftDuino. In erster Linie sind dies die Schutz- und Treiberschaltungen, die beim ftDuino für die fischertechnik-Kompatibilität der Anschlüsse sorgt. Diese Schaltungen sind der Grund, warum man die Ein- und Ausgänge des ftDuino nicht mit den Arduino-üblichen `pinMode()` und `digitalWrite()`-Funktionen ansprechend kann. Diese Funktionen sind darauf ausgelegt, direkt Anschlüsse des ATmega32u4-Mikrocontrollers zu steuern und berücksichtigen nicht, dass der ftDuino zusätzliche Schaltungen beinhaltet.

Aus diesem Grund werden die fischertechnik-kompatiben Ein- und Ausgänge des ftDuino über eigene Bibliotheken angesteuert, wie in Kapitel 8 beschrieben.

Erfahrene Nutzer können unter Umgehung dieser Bibliotheken nach wie vor auch direkt mit der Hardware des ftDuino kommunizieren. Die Schaltpläne im Anhang A liefern alle dafür nötige Information.

1.3 Problemlösungen

1.3.1 Die grüne Leuchtdiode im ftDuino leuchtet nicht

Zunächst sollte der ftDuino von allen Verbindungen getrennt und ausschließlich über den USB-Anschluss mit dem PC verbunden werden. Die grüne Leuchtdiode im ftDuino muss sofort aufleuchten. Tut sie das nicht, dann sollte ein anderer PC bzw. ein anderer USB-Anschluss probiert werden.

Hilft das nicht, dann ist zunächst das USB-Kabel zu prüfen. Funktionieren andere Geräte an diesem Kabel? Gegebenenfalls muss das Kabel ausgetauscht werden.

1.3.2 Der ftDuino taucht am PC nicht als `COM:`-Port auf

Der ftDuino wird nicht mehr vom PC erkannt und es wird kein `COM:`-Port angelegt.

Leuchtet die grüne Leuchtdiode am ftDuino? Falls nicht sollte wie unter 1.3.1 verfahren werden.

Leuchtet die grüne Leuchtdiode, dann sollte ein kurzer Druck auf den Reset-Taster (siehe 1.2.3) den Bootloader des ftDuino für einige Sekunden aktivieren. Erkennbar ist dies am langsamen Ein- und Ausblenden der roten Leuchtdiode wie in Abschnitt 1.2.4 beschrieben. In dieser Zeit sollte der ftDuino vom PC erkannt werden. Dies wird u.a. unter Windows im Gerätemanager wie im Abschnitt 2.1.2 beschrieben angezeigt.

Wird der ftDuino nach einem Reset erkannt, aber verschwindet nach ein paar Sekunden aus der Ansicht des Gerätemanagers oder wird als unbekanntes Gerät angezeigt, dann wurde wahrscheinlich ein fehlerhafter Sketch auf den ftDuino geladen und die Arduino-IDE ist nicht in der Lage, sich eigenständig mit dem ftDuino zu verbinden. In diesem Fall sollte man das Blink-Beispiel (siehe Abschnitt 3.1) in der Arduino-IDE öffnen, den ftDuino per kurzem Druck auf den Reset-Taster in den Booloader-Modus versetzen und direkt danach die Download-Schaltfläche in der Arduino-IDE drücken. Sobald der funktionierende Sketch geladen wurde wird der ftDuino auch ohne manuelle Druck auf den Reset-Taster wieder von PC erkannt und der entsprechende `COM:`-Port taucht wieder auf.

Der ftDuino bleibt nur wenige Sekunden im Bootloader und kehrt danach in den normalen Sketch-Betrieb zurück. Zwischen dem Druck auf den Reset-Knopf und dem Start des Downloads aus der Arduino-IDE sollte daher möglichst wenig Zeit vergehen.

1.3.3 Der ftDuino funktioniert, aber die Ausgänge nicht

Um die Ausgänge zu benutzen muss der ftDuino mit einer 9-Volt-Spannunsgquelle entweder über den Hohlstecker-Anschluss oder über die üblichen fischertechnik-Stecker verbunden sein. Verfügt der ftDuino über keine ausreichende 9-Volt-Versorgung, so können die Ausgänge nicht betrieben werden. Da der ftDuino selbst schon mit geringerer Soannung läuft ist dessen Funktion kein sicheres Indiz dafür, dass eine ausreichende 9-Volt-Versorung vorhanden ist.

Ist der ftDuino über USB mit dem PC verbunden, dann versorgt er sich bei mangelhafter oder fehlender 9-Volt-Versorung von dort. Geht der ftDuino ganz aus, sobald man die USB-Verbindung trennt, dann ist keine 9-Volt-Versorgung gegeben

und es muss sichergestellt werden, dass Polarität und Spannung korrekt bzw. ausreichend sind. Ggf. muss die Batterie ausgetauscht oder der verwendete Akku geladen werden.

Kapitel 2

Installation

Die Installation der Software zur Benutzung des ftDuino erfolgt in mehreren Schritten. Zu allererst muss der Computer mit dem ftDuino bekannt gemacht werden, in dem ein passender Treiber dafür sorgt, dass der Computer erfährt wie er mit dem ftDuino zu kommunizieren hat.

Im zweiten Schritt wird dann die sogenannte Arduino-IDE installiert, also die eigentliche Programmierumgebung sowie die Arduino-IDE mit dem ftDuino verbunden.

Für die Installation und auch für die im Kapitel 3 folgenden ersten Schritte reicht es, den ftDuino per USB mit dem PC zu verbinden. Eine zusätzliche Stromversorgung per Netzteil oder Batterie ist erst nötig, wenn die Ausgänge des ftDuino verwendet werden sollen.

2.1 Treiber

Unter den meisten Betriebssystemen wird der ftDuino vom Computer direkt erkannt, sobald er angesteckt wird. Das trifft unter anderem auf Linux, MacOS X und Windows 10 zu, aber nicht für Windows 7.

2.1.1 Windows 10

Die Verwendung des ftDuinos unter Windows 10 erfordert keine Treiberinstallation durch den Anwender.

Sobald der ftDuino an einen PC unter Windows 10 angesteckt wird werden die passenden Treiber automatisch installiert. Windows 10 zeigt dies beim ersten Anschließen des ftDuino durch eine entsprechende Meldung am unteren rechten Bildschirmrand an. Nach einigen Sekunden ist die Installation abgeschlossen und der ftDuino benutzbar.

Weiteres an- und abstecken erzeugt keine weiteren Meldungen, allerdings ist die erfolgreiche Erkennung des ftDuino unter Windows 10 jederzeit an der typischen Melodie zu erkennen, die ein Windows-PC beim erkennen von Hardware ausgibt.

2.1.2 Windows 7 und Windows Vista

Windows 7 und Windows Vista bingen den passsenden Treiber ebenfalls bereits mit. Allerdings muss eine passende `.inf`-Datei geladen werden, um dafür zu sorgen, dass Windows diesen Treiber für den ftDuino nutzt.

Dass kein Treiber geladen ist erkennt man u.a. daran, dass der ftDuino im Gerätemanager uner "Andere Geräte" aufgeführt wird.

Die `.inf`-Datei ist unter https://raw.githubusercontent.com/harbaum/ftduino/master/ftduino/driver/ftduino.inf zu finden.

Abbildung 2.1: ftDuino ohne passenden Treiber unter Windows 7

Nach dem Download reicht ein Rechtsklick auf die Datei und die Auswahl von "Installieren" im folgenden Menü.

Abbildung 2.2: Rechtsklick auf `ftduino.inf`

Windows bietet daraufhin an, den Treiber zu installieren.

(a) Bestätigungsabfrage

(b) Ggf. folgende Sicherheitsabfrage

Abbildung 2.3: Installation des Treibers

Ggf. erfolgt noch eine Sicherheitsabfrage. Dieser Frage kann man getrost zustimmen, da der eigentliche Treiber bereits Teil von Windows 7 bzw. Windows Vista ist. Die `ftduino.inf`-Datei fordert Windows lediglich auf, ihn zu verwenden.

Sobald die Installation erfolgreich war wird der ftDuino als sogenannter `COM:`-Port eingebuden.

Je nach Betriebsmodus des ftDuino und je nach installierter Anwendung auf dem ftDuino befindet er sich im Anwendungsmodus oder im Bootloader. Windows unterscheidet zwischen beiden Zuständen und weist zwei unterschiedliche `COM:`-Ports zu. Das ist so gewollt und soll nicht weiter irritieren. In den meisten Fällen wir der Benutzer nur den Anwendungsmodus zu sehen bekommen.

(a) Anwendungsmodus (b) Bootloader

Abbildung 2.4: ftDuino mit passendem Treiber unter Windows 7

2.1.3 Linux

Der ftDuino wird von einem handelsüblichen Linux-PC ohne weitere manuelle Eingriffe erkannt. Da er das sogenannte "Abstract Control Model" (ACM) implementiert taucht er im Linux-System als `/dev/ttyACMX` auf, wobei X eine fortlaufende Nummer ist. Sind keine weiteren ACM-Geräte verbunden, so wird der ftDuino als `/dev/ttyACM0` eingebunden.

Mehr Details erfährt man z.B. direkt nach dem Anstecken des ftDuino mit dem `dmesg`-Kommando:

```
$ dmesg
...
[15822.397956] usb 3-1: new full-speed USB device number 9 using xhci_hcd
[15822.540331] usb 3-1: New USB device found, idVendor=1c40, idProduct=0538
[15822.540334] usb 3-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[15822.540336] usb 3-1: Product: ftDuino
[15822.540337] usb 3-1: Manufacturer: Till Harbaum
[15822.541084] cdc_acm 3-1:1.0: ttyACM0: USB ACM device
```

Die genauen Meldungen variieren von System zu System, aber der generelle Inhalt wird vergleichbar sein.

Weitere Details zum erkannten USB-Gerät liefert das `lsusb`-Kommando:

```
$ lsusb -vd 1c40:0538
Bus 003 Device 009: ID 1c40:0538 EZPrototypes
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2 ?
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  idVendor           0x1c40 EZPrototypes
  idProduct          0x0538
...
```

Diese Ausgaben sind besonders interessant, wenn man wie in Abschnitt 5.10 oder 5.16 beschrieben die erweiterten USB-Möglichkeiten des ftDuino nutzt.

"Device or resource busy"

Auch wenn Linux bereits den eigentlichen Gerätetreiber mitbringt kann es trotzdem nötig sein, die Systemkonfiguration anzupassen. Das Symptom ist, dass es beim Versuch, auf den ftDuino zuzugreifen, in der Arduino-IDE zu der folgenden Fehlermeldung kommt.

Abbildung 2.5: Fehlermeldung bei installiertem ModemManager

In diesem Fall ist die wahrscheinlichste Ursache, dass ModemManager, ein Programm zur Bedienung von Modems, installiert ist und sich mit dem ftDuino verbunden hat. Um das zu verhindern, dass der ModemManager versucht, sich mit dem ftDuino zu verbinden, ist die Eingabe des folgenden Kommandos nötig:

```
sudo wget -P /etc/udev/rules.d https://raw.githubusercontent.com/harbaum/ftduino/master/ftduino/driver/99-ftduino.rules
```

Die Datei `/etc/udev/rules.d/99-ftduino.rules` muss danach exakt folgenden Inhalt haben:

```
ATTRS{idVendor}=="1c40" ATTRS{idProduct}=="0537", ENV{ID_MM_DEVICE_IGNORE}="1"
ATTRS{idVendor}=="1c40" ATTRS{idProduct}=="0538", ENV{ID_MM_DEVICE_IGNORE}="1", MODE="0666"
```

Danach muss der ftDuino einmal kurz vom PC getrennt und wieder angesteckt werden und sollte danach problem zu verwenden sein.

Das Kommandos legt eine Datei namens `/etc/udev/rules.d/99-ftduino.rules` an. Diese Datei enthält Regeln, wie der Linux-Kernel mit bestimmten Ereignissen umgehen soll. In diesem Fall soll beim Einstecken eines USB-Gerätes mit der Hersteller-Identifikation `1c40` und den Geräteidentifikationen `0537` und `0538` dieses vom ModemManager ignoriert werden.

2.2 Arduino-IDE

Die integrierte Entwicklungsumgebung (IDE) für den Arduino bekommt man kostenlos für die gängigsten Betriebssysteme unter https://www.arduino.cc/en/Main/Software. Die Windows-Version mit eigenem Installer ist dort z.B. direkt unter dem Link https://www.arduino.cc/download_handler.php erreichbar. Diese Arduino-IDE wird zunächst installiert.

Um den ftDuino unter der Arduino-IDE nutzen zu können muss eine entsprechende Konfoiguration vorgenommen werden. Die Arduino-IDE erlaubt es, diesen Vorgang weitgehend zu automatisieren.

2.2.1 Installation mit dem Boardverwalter

Für die einfache Installation zusätzlicher Boards bringt die Arduino-IDE den sogenannten Boardverwalter mit. Zunächst muss dem Boardverwalter in den Arduino-Voreinstellungen mitgeteilt werden, wo die ftDuino-Konfiguration zu finden ist.

Dazu trägt man https://raw.githubusercontent.com/harbaum/ftduino/master/package_ftduino_index.json in den Voreinstellungen wie folgt ein. Beim Eintragen der entsprechende Zeile ist darauf zu achten, dass die URL Unterstriche (_) enthält, die ggf. beim Kopieren (Copy'n Paste) der URL aus diesem PDF-Dokument verloren gehen. In diesem Fall sollte die URL manuell eingegeben werden.

Abbildung 2.6: URL der ftDuino-Konfiguration in den Arduino-Voreinstellungen

Den eigentlichen Boardverwalter erreicht man danach direkt über das Menü der IDE unter `Werkzeuge` ▷ `Board: ...` ▷ `Boardverwalter...`.

Abbildung 2.7: Den Boardverwalter startet man aus dem Menü

Nachdem die `JSON`-Datei in den Voreinstellungen eingetragen wurde bietet der Boardverwalter automatisch die ftDuino-Konfiguration an.

Abbildung 2.8: Im Boardverwalter kann das ftDuino-Board installiert werden

Durch Klick auf `Installieren...` werden alle für den ftDuino nötigen Dateien automatisch heruntergeladen und installiert.

Nach erfolgreicher Installtion kann der ftDuino unter den Boards ausgewählt werden.

Abbildung 2.9: Auswahl des ftDuino-Boards

Ist bereits ein ftDuino angeschlossen und wurde der nötige Treiber installiert, so lässt sich der ftDuino nun unter `Port` auswählen.

Die Installation ist damit abgeschlossen. Während der Installation wurden bereits einige Beispielprogramme installiert. Diese finden sich im Menü unter `Datei` ▷ `Beispiele` ▷ `Examples for ftDuino`.

Abbildung 2.10: Beispiele zum ftDuino-Board

Diese Beispiele können direkt geladen und auf den ftDuino hinuntergeladen werden.

2.2.2 Updates

Die Arduino-IDE benachrichtigt automatisch über Softwareupdates der ftDuino-Konfiguration. Mit wenig Aufwand bleibt man so immer auf dem aktuellen Stand.

Kapitel 3

Erste Schritte

In diesem Kapitel geht es darum, erste Erfahrungen mit dem ftDuino und der Arduino-IDE zu sammeln. Voraussetzung ist, dass der ftDuino von einem passenden Treiber auf dem PC unterstützt wird und dass die Arduino-IDE wie in Kapitel 2 beschrieben installiert und für die Verwendung des ftDuinos vorbereitet wurde.

Zusätzlich zum ftDuino wird ein handelsübliches Mini-USB-Kabel benötigt, wie es z.B. auch mit den fischertechnik TX und TXT verwendet wird.

3.1 Der erste Sketch

Für die ersten Versuche benötigt der ftDuino keine seperate Stromversorgung. Es genügt, wenn er per USB vom PC versorgt wird. Die fischertechnik-Ein- und Ausgänge bleiben zunächst unbenutzt.

Als erstes kann man den folgenden Sketch direkt in der Arduino-IDE eingeben. Das Beispiel muss aber nicht zwingend manuell eingetippt werden, denn es findet sich als fertig mitgeliefertes Beispiel im `Datei`-Menü der Arduino-IDE unter `Datei` ▷ `Beispiele` ▷ `FtduinoSimple` ▷ `Blink`.

Abbildung 3.1: Die ftDuino-Beispiele in der Arduino-IDE

Alle vorinstallierten Beispiele können mit einem Klick geladen werden und es öffnet sich ein neues Fenster mit dem ausgewählten Beispiel.

```
/*
  Blink

  Schaltet die interne rote LED des ftDuino für eine Sekunde ein
  und für eine Sekunde aus und wiederholt dies endlos.

  Original:
  http://www.arduino.cc/en/Tutorial/Blink
*/
```

```
10
11  // die setup-Funktion wird einmal beim Start aufgerufen
12  void setup() {
13    // Konfiguriere den Pin, an den die interne LED angeschlossen ist, als Ausgang
14    pinMode(LED_BUILTIN, OUTPUT);
15  }
16
17  // die loop-Funktion wird immer wieder aufgerufen
18  void loop() {
19    digitalWrite(LED_BUILTIN, HIGH);   // schalte die LED ein (HIGH ist der hohe Spannungspegel)
20    delay(1000);                       // warte 1000 Millisekunden (eine Sekunde)
21    digitalWrite(LED_BUILTIN, LOW);    // schalte die LED aus, indem die Spannung auf
22                                       // niedrigen Pegel (LOW) geschaltet wird
23    delay(1000);                       // warte eine Sekunde
24  }
```

3.1.1 Download des Blink-Sketches auf den ftDuino

Der Blink-Sketch sollte nun geöffnet sein. Der ftDuino sollte an den PC angeschlossen sein und im Menü unter `Werkzeuge ▷ Board` der ftDuino ausgewählt sowie der richtige `COM:`-Port unter `Werkzeuge ▷ Port` ausgewählt sein.

Abbildung 3.2: Auswahl des Ports in der Arduino-IDE unter Windows

Dies zeigt die Arduino-IDE auch ganz unten rechts in der Statusleiste an.

(a) Linux (b) Windows

Abbildung 3.3: Der ausgewählte ftDuino wird in der Statusleiste angezeigt

Der Download des Sketches auf den ftDuino erfordert nur noch einen Klick auf die Download-Pfeil-Schaltfläche in der Arduino-IDE oben links.

Abbildung 3.4: Download-Schaltfläche der Arduino-IDE

Der Sketch wird von der IDE zunächst in Maschinencode übersetzt. Wenn die Übersetzung erfolgreich war wird der Maschinencode über die USB-Verbdinung auf den ftDuino übertragen und dort im Flash-Speicher abgelegt.

Während des Downloads zeigt die interne rote Leuchtdiode des ftDuino wie in Abschnitt 1.2.4 beschrieben an, dass der Bootloader aktiviert wird und dass der Download stattfindet.

Nach erfolgreichem Download startet der Sketch sofort und die interne rote Leuchtdiode blinkt langsam.

Abbildung 3.5: Blinkende interne rote Leuchtdiode im ftDuino

3.1.2 Die Funktionsweise des Sketches

Der Sketch-Code besteht zum überwiegenden Teil aus erklärenden Kommentaren, die für die Funktion des Sketches völlig unbedeutend sind und lediglich dem Verständnis durch einen menschlichen Leser dienen. Kommentarzeilen beginnen mit einem doppelten Schrägstrich (`//`). Mehrzeilige Kommentare werden durch `/*` und `*/` eingechlossen. In diesem Dokument sowie in der Arduino-IDE sind Kommentare an ihrer hellgrauen Färbung leicht zu erkennen. Tatsächlicher Code befindet sich lediglich in den Zeilen 12 bis 15 sowie den Zeilen 18 bis 24.

3.1.3 Die Funktionen `setup()` und `loop()`

Jeder Arduino-Sketch enthält mindestens die beiden Funktionen `setup()` (englisch für Einrichtung) und `loop()` (englisch für Schleife). Zwischen den beiden geschweiften Klammern ({ und }) befinden sich jeweils durch Semikolon abgetrennt die eigentlichen durch den ftDuino auszuführenden Befehle. Die Befehle in der Funktion `setup()` werden einmal bei Sketch-Start ausgeführt. Sie werden üblicherweise verwendet, um initiale Einstellungen vorzunehmen oder Ein- und Ausgänge zu parametrieren. Die Befehle der `loop()`-Funktion werden hingegen immer wieder ausgeführt solange der ftDuino eingechaltet bleibt oder bis er neu programmiert wird. Hier findet die eigentliche Sketch-Funktion statt und hier wird auf Sensoren reagiert und Aktoren werden angesteuert.

Auch das `Blink`-Beispiel arbeitet so. In der `setup()`-Funktion wird in Zeile 14 der mit der roten Leuchtdiode verbundene interne Anschluss im ftDuino zum Ausgang erklärt.

In der `loop()`-Funktion wird dann in Zeile 19 der interne Anschluss der roten Leuchtdiode eingeschaltet (Spannungspegel hoch, `HIGH`) und in Zeile 21 wird er ausgeschaltet (Spannungspegel niedrig, `LOW`). Zwischendurch wird jeweils in den Zeilen 20 und 23 1000 Millisekunden bzw. eine Sekunde gewartet. Die Leuchtdiode wird also eingeschaltet, es wird eine Sekunde gewartet, sie wird ausgeschaltet und es wird eine weitere Sekunde gewartet. Dies passiert immer und immer wieder, so dass die Leuchtdiode mit einer Frequenz von 0,5 Hertz blinkt.

3.1.4 Anpassungen am Sketch

Für den Einstieg ist es oft sinnvoll, mit einem vorgefertigten Sketch zu starten und dann eigene Änderungen vorzunehmen. Die Beispiele der Arduino-IDE stehen aber allen Benutzern eines PCs zur Verfügung und können daher zunächst nicht verändert werden. Nimmt man an einem Beispiel-Sketch Änderungen vor und versucht sie zu speichern, dann weist einen die Arduino-IDE darauf hin, dass man eine eigene Kopie anlegen soll. Dazu öffnet die Arduino-IDE einen Dateidialog und man hat die Möglichkeit, den Sketch vor dem Speichern umzubenennen z.B. in `SchnellBlink`.

Abbildung 3.6: Die Arduino-IDE fordert zum Speichern einer eigenen Kopie auf

Sobald man auf diese Weise eine eigene Kopie angelegt hat kann man sie beliebig verändern. Die eigene Kopie wird im Menü der Arduino-IDE unter `Datei ▷ Sketchbook` eingefügt und kann von dort später jederzeit wieder geladen werden.

Abbildung 3.7: Kopie `SchnellBlink` im Sketchbook der Arduino-IDE

Im Sketch kann man nun beispielsweise aus den 1000 Millisekunden in den Zeilen 20 und 23 jeweils 500 Millisekunden machen.

```
18  void loop() {
19    digitalWrite(LED_BUILTIN, HIGH);   // schalte die LED ein (HIGH ist der hohe Spannungspegel)
20    delay(500);                        // warte 500 Millisekunden (eine halbe Sekunde)
21    digitalWrite(LED_BUILTIN, LOW);    // schalte die LED aus, indem die Spannung auf
22                                       // niedrigen Pegel (LOW) geschaltet wird
23    delay(500);                        // warte eine halbe Sekunde
24  }
```

Nach erfolgreichem Download wird die Leuchtdiode dann jeweils für 0,5 Sekunden ein- und ausgeschaltet und die Blinkfrequenz verdoppelt sich auf ein Hertz.

3.2 Ansteuerung von fischertechnik-Komponenten

Um die interne Leuchtdiode des ftDuino blinken zu lassen hätten wir keinen ftDuino benötigt. Alle Arduinos verfügen über eine solche interne Leuchtdiode und hätten für unser erstes Beispiel verwendet werden können.

Seine speziellen Fähigkeiten spielt der ftDuino aus, wenn es darum geht mit den üblichen fischertechnik-Sensoren und -Aktoren umzugehen. Der Blink-Sketch soll daher so erweitert werden, dass zusätzlich zu Leuchtdiode eine am Ausgang O1 angeschlossene Lampe blinkt.

Angeschlossen wird dazu eine normale fischertechnik-Lampe mit einem Stecker an den Ausgang O1 des ftDuino und mit dem zweiten Stecker an einen der Masseanschlüsse des ftDuino. Masseanschlüsse sind die 12 Anschlüsse, die in der Abbildung 3.8 mit einem Massesymbol ⊥ verbunden sind.

Da nun die mit 9 Volt betriebenen fischertechnik-Ausgänge verwendet werden muss der ftDuino zusätzlich mit 9 Volt versorgt werden. Das kann z.B. über ein übliches fischertechnik-Netzteil erfolgen oder über einen Batteriehalter. Beide Anschlüsse sind verpolungsgeschützt, speziell beim Anschluss der Batterie kann man also keinen Schaden anrichten.

3.2.1 Der Sketch

Der folgende Beispiel-Sketch `BlinkO1` findet sich auch im `Datei`-Menü der Arduino-IDE unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ BlinkO1`.

```
1  // BlinkO1.ino
2  //
3  // Blinken einer Lamoe an Ausgang O1
4  //
5  // (c) 2018 by Till Harbaum <till@harbaum.org>
6
```

Abbildung 3.8: Blinkende fischertechnik-Lampe ftDuino

```
 7  #include <FtduinoSimple.h>
 8
 9  void setup() {
10    // LED initialisieren
11    pinMode(LED_BUILTIN, OUTPUT);
12  }
13
14  void loop() {
15    // schalte die interne LED und den Ausgang O1 ein (HIGH bzw. HI)
16    digitalWrite(LED_BUILTIN, HIGH);
17    ftduino.output_set(Ftduino::O1, Ftduino::HI);
18
19    delay(1000);                        // warte 1000 Millisekunden (eine Sekunde)
20
21    // schalte die interne LED und den Ausgang O1 aus (LOW bzw. LO)
22    digitalWrite(LED_BUILTIN, LOW);
23    ftduino.output_set(Ftduino::O1, Ftduino::LO);
24
25    delay(1000);                        // warte eine Sekunde
26  }
```

Der Sketch unterscheidet sich nur in wenigen Details vom ursprünglichen Blink-Sketch. Neu hinzugekommen sind die Zeilen 7, 17 und 23. In Zeile 7 wird eine Bibliothek eingebunden, die speziell für den ftDuino mitgeliefert wird und den Zugriff auf die Ein- und Ausgänge des ftDuino vereinfacht. In den Zeilen 17 und 23 wird der Ausgang `O1` eingeschaltet (`HI`) bzw. ausgeschaltet (`LO`). Weitere Details zu dieser Bibliothek finden sich in Kapitel 8.

Die Kommandos zum Ein- und Ausschalten der internen Leuchtdiode sind nach wie vor vorhanden, so dass die interne Leuchtdiode nun parallel zur extern angeschlossenen Lampe blinkt.

Weitere einfache Beispiele und Erklärungen zur Benutzung der Ein- und Ausgänge in eigenen Sketches finden sich in Abschnitt 8.1.1.

3.2.2 Eingänge

Zum Anschluss an übliche fischertechnik-Taster, -Fototransistoren und ähnlich verfügt der ftDuino über die acht Eingänge `I1` bis `I8` und die Zählereingänge `C1` bis `C4`.

Der Zugriff auf diese Eingänge erfolgt über passende Bibliotheken wie in Abschnitt 8 dargestellt. Über die `FtduinoSimple`-Bibliothek kann der Schaltzustand eines Tasters abgefragt werden:

```
1  #include <FtduinoSimple.h>
```

Abbildung 3.9: Taster am Eingang I1 des ftDuino

```
void setup() {
  // keine Initialisierung noetig
}

void loop() {
  // lies den Zustand einer Taste an Eingang I1
  if(ftduino.input_get(Ftduino::I1)) {
    /* ... tue etwas ... */
  }
}
```

Um Analogwerte wie Spannungen und Widerstände einzulesen ist der erweiterte Funktionsumfang der `Ftduino`-Bibliothek nötig. Bei ihrer Verwendung muss zunächst der Messmodus des Eingangs eingestellt werden, bevor Widerstands-Werte gelesen werden können:

```
#include <Ftduino.h>

void setup() {
  // Initialisierung der Ftduino-Bibliothek
  ftduino.init();

  // Eingang I1 zur Widerstandsmessung vorbereiten
  ftduino.input_set_mode(Ftduino::I1, Ftduino::RESISTANCE);
}

void loop() {
  // Auswertung des Widerstands an Eingang I1
  uint16_t widerstand = ftduino.input_get(Ftduino::I1);
  /* ... tue etwas ... */
}
```

In den Experimenten in Kapitel 5 finden sich diverse Beispiel, in denen die Eingänge des ftDuino ausgertet werden inklusive spezieller Sensoren wie dem Temperatursensor in Abschnitt 5.7.

3.3 Kommunikation mit dem PC

Der ftDuino ist primär dafür gedacht, ein Modell autonom zu steuern und während des Betriebs nicht auf die Hilfe eines PC angewiesen zu sein. Trotzdem gibt es Gründe, warum auch im laufenden Betrieb ein Datenaustausch mit einem PC erwünscht sein kann.

Vor allem während der Sketch-Entwicklung und bei der Fehlersuche hilft es oft sehr, wenn man sich z.B. bestimmte Werte am PC anzeigen lassen kann oder wenn man Fehlermeldungen im Klartext an den PC senden kann. Aber auch die Ausgabe z.B. von Messwerten an den PC zur weiteren Auswertung oder Speicherung ist oft hilfreich.

Ein Sketck kann dazu den `COM:`-Port zwischen ftDuino und PC für den Datenaustausch nutzen. Der ftDuino-Beispiel-Sketch `ComPort` zum Beispiel verwendet den `COM:`-Port, um ein paar einfache Textausgaben am PC zu erzeugen. Das `ComPort`-

Beispiel findet sich im `Datei`-Menü der Arduino-IDE unter `Datei` ▷ `Beispiele` ▷ `FtduinoSimple` ▷ `USB` ▷ `ComPort`. Auch der `ComPort`-Sketch verwendet keinen der Ausgänge und benötigt daher neben der USB-Verbindung keinerlei weitere Spannungsversorung.

```
1  /*
2     ComPort - Kommunikation mit dem PC über den COM:-Port
3
4   */
5
6  int zaehler = 0;
7
8  void setup() {
9    // Port initialisieren und auf USB-Verbindung warten
10   Serial.begin(9600);
11   while(!Serial);      // warte auf USB-Verbindung
12
13   Serial.println("ftDuino COM:-Port test");
14 }
15
16 void loop() {
17   Serial.print("Zähler: ");        // gib "Zähler:" aus
18   Serial.println(zaehler, DEC);    // gib zaehler als Dezimalzahl aus
19
20   zaehler = zaehler+1;             // zaehler um eins hochzählen
21
22   delay(1000);                     // warte 1 Sekunde (1000 Millisekunden)
23 }
```

3.3.1 Der serielle Monitor

Man kann ein beliebiges sogenanntes Terminalprogramm auf dem PC nutzen, um Textausgaben des ftDuino via `COM:`-Port zu empfangen. Die Arduino-IDE bringt prakischerweise selbst ein solches Terminal mit. Es findet sich im Menü unter `Werkzeuge` ▷ `Serieller Monitor`.

Abbildung 3.10: Der serielle Monitor findet sich im `Werkzeuge`-Menü

Man muss den `COM:`-Port nicht seperat einstellen, sondern es wird der `COM:`-Port übernommen, der bereits für den Sketch-Download eingestellt wurde.

Nach der Auswahl im Menü öffnet der serielle Monitor ein eigenes zusätzliches Fenster auf dem PC. Wurde der `ComPort`-Sketch bereits auf den ftDuino geladen, dann erscheinen sofort entsprechende Ausgaben, sobald der serielle Monitor geöffnet wurde.

Die Nutzung des `COM:`-Ports wird automatisch zwischen Sketch-Download und seriellem Monitor geteilt. Man kann also jederzeit bei geöffnetem seriellen Monitor den Sketch-Editor in den Vordergrund holen und einen Download starten. Der seriellem Monitor wird dann automatisch für die Zeit des Downloads deaktiviert.

Eine Besonderheit des USB-`COM:`-Ports die der ftDuino vom Arduino Leonardo erbt ist, dass die sogenannte Bitrate (oft auch Baudrate genannt) keine Bedeutung hat. Im `ComPort`-Beispiel wird in Zeile 10 eine Bitrate von 9600 Bit pro Sekunde eingestellt, in Abbildung 3.11 ist aber unten rechts eine Bitrate von 115200 Bit pro Sekunde eingestellt. Die USB-Verbindung

Abbildung 3.11: Der serielle Monitor

ignoriert diese Einstellungen und kommuniziert trotzdem fehlerfrei. Einige andere Mitglieder der Arduino-Familie wie der Arduino-Uno benötigen hier aber übereinstimmende Einstellungen.

3.3.2 Sketchbeschreibung

Der Sketch besteht aus wenig mehr als den eigentlich Aufrufen zur Bedienung des `COM:`-Ports. Da der `COM:`-Port auch oft "serieller Port", englisch "Serial" genannt wird fangen die entsprechenden Funktionsaufrufe alle mit `Serial.` an.

In Zeile 10 der sofort bei Sketchstart aufgerufenen `setup()`-Funktion wird zunächst der `COM:`-Port für die Kommunikation geöffnet. In Zeile 11 dann wird gewartet, bis der `COM:`-Port auf PC-Seite verfügbar ist und die erste Kommunikation stattfinden kann. In Zeile 13 wird danach eine erste Startmeldung an den PC geschickt.

Innerhalb der widerholt durchlaufenen `loop()`-Funktion wird dann in den Zeilen 17 und 18 zunächst der Text "Zähler: " ausgegeben, gefolgt von dem Inhalt der Variablen `zaehler` in Dezimaldarstellung. Die `println()`-Funktion führt nach der Ausgabe einen Zeilenvorschub aus. Die folgende Ausgabe erfolgt daher am Beginn der nächsten Bildschirmzeile.

Schließlich wird in Zeile 20 die `zaehler`-Variable um eins erhöht und eine Sekunde (1000 Millisekunden) gewartet.

3.3.3 USB-Verbindungsaufbau

Bei Geräten wie dem Arduino Uno, die einen seperaten USB-Kommunikationsbaustein für die USB-Kommunikation nutzen, besteht die USB-Verbindung durchgängig, sobald das Gerät mit dem PC verbunden ist. Beim ftDuino sowie beim Arduino Leonardo übernimmt der Mikrocontroller wichtige Aspekte der USB-Kommunikaiton selbst. Das bedeutet, dass die logische USB-Verbindung jedes Mal getrennt und neu aufgebaut wird, wenn ein neuer Sketch auf den Mikrocontroller übertragen wird.

Beginnt der ftDuino direkt nach dem Download mit der Textausgabe auf dem `COM:`-Port, so gehen die ersten Nachrichten verloren, da die USB-Verbindung noch nicht wieder aufgebaut ist. Daher wartet der Sketch in Zeile 11 darauf, dass die Kommunikationsverbindung zwischen ftDuino und PC wieder besteht, bevor die ersten Ausgaben erfolgen.

Man kann testweise diese Zeile einmal löschen oder auskommentieren (Programmcode, der per `//` in einen Kommentar verwandelt wurde, wird nicht ausgeführt).

```
 8  void setup() {
 9    // Port initialisieren und auf USB-Verbindung warten
10    Serial.begin(9600);
11  //  while(!Serial);       // warte auf USB-Verbindung
12
13    Serial.println("ftDuino COM:-Port test");
14  }
```

Lädt man diesen Sketch nun auf den ftDuino, so beginnt die Textausgabe im seriellen Monitor erst ab der Zeile `Zähler: 2`. Die beiden vorhergehenden Zeilen werden vom ftDuino an den PC gesendet, bevor die USB-Verbindung wieder steht und gehen daher verloren. Dieses Verhalten kann trotzdem gewünscht sein, wenn die Ausgabe über USB nur zusätzlich erfolgen soll und der ftDuino auch ohne angeschlossenen PC arbeiten soll.

Kapitel 4

Programmierung

Diese Kapitel erklärt die Programmierung eigener Programme bzw. Sketches für den ftDuino. Wer zunächst keine eigenen Programme schreiben möchte und erstmal die vorgefertigten Beispiele und Experimente ausführen möchte ohne deren Programmcode zu verstehen kann direkt mit Kapitel 5 weitermachen.

Der ftDuino wird mit einer ständig wachsenden Anzahl vorgefertigter Beispiel-Sketches ausgeliefert. Für viele Modelle und Versuche reicht es daher, wie in den vorhergehenden Kapiteln beschrieben diese Beispiele auf den ftDuino zu laden. Aber wie der Spass beim Bauen mit fischertechnik gerade dort beginnt, wo man die vorgegebenen Pfade der Bauanleitungen hinter sich lässt, so besteht auch beim ftDuino der eigentliche Nutzen darin, dass man ihn mit eigenen Sketches programmieren kann. Zusammen mit einem selbstentworfenen Modell ergeben sich so beeidruckende Möglichkeiten. Und ganz nebenbei lernt man nicht nur die mechanische Grundlagen kennen, sondern erhält zusätzlich einen realistischen Einblick in die Welt der Mikrocontroller-Programmierung.

Dieses Kapitel soll einen ersten Einblick in die Programmierung des ftDuino geben. Es werden die wesentlichen Sprachkonstrukte der auf dem ftDuino verwendeten Programmiersprache soweit erklärt, wie sie zum Verständnis der Programme der nachfolgenden Kapitel benötigt werden. Die Beschreibung beschränkt sich bewusst auf das allernötigste. Trotzdem sind Ende dieses recht kurzen Kapitels alle nötigen Grundlagen für erste eigene Sketches vorhanden.

4.1 Textbasierte Programmierung

Im Gegensatz zu den meisten Programmierumgebungen aus der Welt der Kostruktionsspielzeuge wird der Arduino nicht grafisch sondern textbasiert programmiert. Während die grafischen Umgebungen wie fischertechniks RoboPro oder Legos EV3-Programmier-App auf leichte Erlernbarkeit ausgelegt sind steht in der Arduino-IDE der Praxisbezug im Vordergrund. Arduino zu programmieren bedeutet auf die gleiche Art Programme zu schreiben, wie es auch professionelle Entwickler kommerzieller Produkte machen.

(a) grafisch

```
#include <FtduinoSimple.h>

void setup() {
}

void loop() {
  ftduino.output_set(Ftduino::O1, Ftduino::HI);
  delay(1000);
  ftduino.output_set(Ftduino::O1, Ftduino::LO);
  delay(1000);
}
```

(b) textbasiert

Abbildung 4.1: Blinken einer Lampe an Ausgang O1

Tatsächlich hat die textuelle Darstellung einige signifikante Vorteile. Vor allem bei großen Projekten ist eine ansprechend formatierte textuelle Darstellung daher wesentlich verständlicher als eine grafische.

(a) in der Arduino-IDE

(b) auf Github

Abbildung 4.2: Darstellung des gleichen Arduino-Sketches

Ein textbasiertes Programm besteht aus unformatiertem Text. Einge Programmierumgebungen wie die Arduino-IDE lassen dennoch farbige Hervorhebungen oder unterschiedliche Schriftgrößen zu oder zeigen im Programmtext Zeilennummern an. Im Gegensatz zu Texten aus einer Textverarbeitung sind diese Formatierungen aber nicht Teil des erstellten Programmes. Stattdessen sind sie Teil der Programmierumgebung selbst und die Formatierung geht z.B. bei der Weitergabe des Programmcodes verloren. Der gleiche Programmcode kann daher in einer anderen Umgebung vollkommen anders aussehen. Für die Funktion des Programmes ist das bedeutungslos. Es kommt nicht darauf an wie der Programmtext dargestellt wird. Für die eigentliche Programmfunktion ist ausschließlich der Inhalt des dargestellten Texts verantwortlich, nicht sein Aussehen.

Wie in Abbildung 4.2 zu sehen unterscheidet sich die Darstellung des identischen Programmcodes zwischen der Arduino-IDE und z.B. dem Web-Dienst github deutlich.

Auch dieses Handbuch verwendet eine eigene Darstellung und stellt zum Beispiel Zeilennummern dar. Diese Zeilennummern dienen nur der leichteren Bezugnahme auf einzelne Zeilen und dürfen nicht als Teil des Programms explizit eingegeben werden:

```
 1  // Blink01.ino
 2  //
 3  // Blinken einer Lamoe an Ausgang O1
 4  //
 5  // (c) 2018 by Till Harbaum <till@harbaum.org>
 6
 7  #include <FtduinoSimple.h>
 8
 9  void setup() {
10    // LED initialisieren
11    pinMode(LED_BUILTIN, OUTPUT);
12  }
13
14  void loop() {
15    // schalte die interne LED und den Ausgang O1 ein (HIGH bzw. HI)
16    digitalWrite(LED_BUILTIN, HIGH);
17    ftduino.output_set(Ftduino::O1, Ftduino::HI);
18
19    delay(1000);                        // warte 1000 Millisekunden (eine Sekunde)
20
21    // schalte die interne LED und den Ausgang O1 aus (LOW bzw. LO)
22    digitalWrite(LED_BUILTIN, LOW);
23    ftduino.output_set(Ftduino::O1, Ftduino::LO);
24
25    delay(1000);                        // warte eine Sekunde
26  }
```

Wer auch in der Arduino-IDE Zeilennummern dargestellt bekommen möchte kann sie in den Voreinstellungen der Arduino-IDE wie in Abbildung 4.3 dargestellt aktivieren.

Abbildung 4.3: Aktivierung von Zeilennummern in der Arduino-IDE

4.2 Die Programmiersprache C++

Arduinos und der ftDuino werden in der Programmiersprache C++, genauer dem Standard C++11 programmiert. Ein Arduino-Sketch unterscheidet sich in einigen wenigen fundamentalen Dingen von klassischen C++-Programmen, davon abgesehen entspricht die Programmoierung dem Standard. Die Programmiersprache C++ ist in der professionellen Softwareentwicklung weit verbreitet. Große Teile aller gängigen Betriebssysteme wie Windows, Linux oder MacOS sind in C++ bzw. einer eng mit C++ verwandten Programmiersprache geschrieben. Die Arduino-Programmierung liefert dadurch einen realistischen Einblick in den professionellen Bereich.

Die Sketches genannten C++-Programme der Arduino-Welt werden durch ein Compiler genanntes Programm in sogenannten Maschinen- oder Binärcode übersetzt, der vom Mikrocontroller des Arduino bzw. ftDuino verstanden wird. Dieser Binärcode kann dann auf den ftDuino per Download übertragen werden.

Der Compiler, der von der Arduino-IDE verwendet wird ist der sogenannte GCC[1]. Dieser Compiler findet auch in der Industrie Verwendung und wird unter anderem zur Übersetzung des Linux-Kernels verwendet, wie er auf jedem Android-Smartphone zum Einsatz kommt. Es handelt sich bei der Arduino-IDE und deren intern verwendeten Werkzeugen also keinesfalls um reine Hobby-Technik. Unter der anfängertauglichen Oberfläche verbergen sich im Gegenteil leistungsfähige und professionelle Komponenten.

4.3 Grundlagen

Ein Arduino-Sketch besteht aus einer textuellen Beschreibung. Der minimale Text, der einen gültigen Sketch bildet sieht folgendermaßen aus:

```
void setup() {
}

void loop() {
}
```

[1] GNU-Compiler-Collection GCC: https://gcc.gnu.org/

Um diesen Sketch einzugeben öffnet man zunächst die Arduino-IDE und wählt dann im Menü `Datei ▷ Neu`. Es öffnet sich ein neues Fenster, das genau die vorher abgebideten Zeilen bereits enthält. Zusätzlich sind noch zwei mit // beginnende Zeile vorhanden. Diese können entfernt werden, sodass der Text am Ende exakt dem obigen Beispiel entspricht.

(a) Übersetzung (b) Download

Abbildung 4.4: Schaltflächen der Arduino-IDE

Dieser minimale Sketch lässt sich in der Arduino-IDE erfolgreich übersetzen und auf den ftDuino laden. Dazu gibt es in der Arduino-IDE die beiden in Abbildung 4.4 dargestellten Schaltflächen. Eine Fläche startet die Übersetzung in Maschinencode. Diese Funktion benötigt keinen angeschlossenen ftDuino und mit ihr lässt sich schnell feststellen, ob ein Sketch keine fundamentalen Fehler enthält, die bereits die Übersetzung verhindern. Die zweite Schaltfläche startet den Download auf den ftDuino. Sollte der Sketch noch nicht übersetzt sein, so wird bei Klick auf die Download-Schaltfläche vorher noch automatisch die Übersetzung gestartet. Nur wenn diese Übersetzung erfolgreich ist wird der Download begonnen.

Hat man den Sketch per `Datei ▷ Neu` frisch angelegt, so wird einen die Arduino-IDE nun ggf. fragen, ob man den Sketch speichern möchte. Wenn man dem Speichern zustimmt kann man an dem Sketch später jederzeit weiter arbeiten.

Nach dem erfolgreichen Download wird der ftDuino keine sichtbare Reaktion zeigen. Der Sketch besteht aus zwei sogenannten Funktionen, eine mit dem Namen `setup()` und eine mit dem Namen `loop()`. Diese Funktionen bilden das Skelett eines jeden Arduino-Sketches. Sie enthalten jeweils in ein Paar geschweifter Klammern { und } eingebettete Anweisungen, die die eigentliche Funktionalität des Sketches beschreiben. Aufeinanderfolgende Anweisungen werden dabei per Semikolon (;) getrennt.

Befinden sich aber wie in diesem einfachen Beispiel überhaupt keine Anweisungen zwischen den Klammern, so löst der Sketch folgerichtig auch keine erkennbare Reaktion auf dem ftDuino aus.

4.3.1 Kommentare

Die beiden zunächst entferten mit // beginnenden Zeilen fügen dem Sketch ebenfalls keine Funktionalität zu. Um das zu überprüfen kann man noch einmal `Datei ▷ Neu` auswählen und den Sketch diesmal unverändert lassen:

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Ein erneuter Klick auf den Download-Button wird auch diesen Sketch übersetzen und auf den ftDuino laden. Wieder erfolgt am ftDuino keine erkennbare Funktion. Das liegt daran, dass die beiden zusätzlichen Zeilen reine Kommentarzeilen sind. Sie sind dazu gedacht, einem menschlichen Leser zusätzliche Erklärungen zu geben. Für die Übersetzung in Maschinencode sind diese Zeilen bedeutungslos. Alles, was in einer Zeile hinter den doppelten Schrägstrichen // steht wird bei der Erzeugung des Maschinencodes ignoriert.

Zusätzlich kann man Kommentare in /* und */ einschließen. Die Erzeugung von Maschinencode beginnt erst wieder hinter dem schließenen Element */. Außerdem dürfen Kommentare dieser Art über mehrere Zeilen gehen.

```
void setup() {
  /* mehrzeilige Kommentare
     sind auch möglich */

}
```

Einen Hinweis auf Kommentare gibt in der Arduino-IDE die Farbgebung. Kommentare sind immer hellgrau dargestellt und damit leicht vom eigentlichen Programmcode zu unterscheiden.

4.3.2 Fehlermeldungen

Als nächstes sollen zwei Programmanweisungen zwischen den geschweiften Klammern der `setup()`-Funktion eingefügt werden:

```
1  void setup() {
2    pinnMode(LED_BUILTIN, OUTPUT);
3    digitalWrite(LED_BUILTIN, HIGH);
4  }
5
6  void loop() {
7  }
```

Was diese beiden kryptischen Zeilen genau bedeuten wird später erklärt werden. Zunächst soll der Sketch lediglich wieder übersetzt und auf den ftDuino geladen werden. Ist das Programm fehlerfrei übersetzbar, so wird es erneut auf den ftDuino übertragen.

Leider ist es in diesem Fall aber nicht fehlerfrei. Wurde das Programm exakt wie abgebildet eingegeben, so sollte die Übersetzung des Programms mit der in Abbildung 4.5 zu sehenden Fehlermeldung abbrechen.

Abbildung 4.5: Anzeige eines Übersetzungsfehlers

im oberen Teil des Bildschirms wird die fehlerhafte Codezeile hervorghoben. Im unteren Teil des Fensters sind die Ausgaben des GCC-Compilers zu sehen. Die eigentliche Fehlermeldung lautet `'pinnMode' was not declared in this scope`.

Damit teilt der Compiler mit, dass er mit dem Begriff `pinnMode` nichts anfangen kann. Schon bei der Eingabe des Sketches gab es einen Hinweis auf dieses Problem, denn die Anweisung `pinnMode` wurde bei der Eingabe in der IDE lediglich schwarz dargestellt während z.B. in der zweiten Zeile das Wort `digitalWrite` orange hervorgehoben wurde. Die Arduino-IDE färbt die meisten ihr bekannten Anweisungen ein und ein nicht-gefärbter Teil kann einen Hinweis auf einen Fehler sein.

Tatsächlich hat sich hier ein Fehler eingeschlichen und statt `pinnMode` hätte es `pinMode` heißen müssen. Ändert man das Wort entsprechend, so färbt die IDE es erwartungsgemäß orange ein, die Übersetzung gelingt und ein Download ist möglich. Nach dem Download sollte die rote Leuchtdiode im ftDuino leuchten, denn genau das bewirken die zwei kryptischen Zeilen, die im folgenden Abschnitt genauer erklärt werden.

4.3.3 Funktionen

Das im Rahmen der Arduino-Programmierung wichtigste C++-Sprachelement sind die sogenannten Funktionen. Mit ihnen lassen sich bereits sinnvolle einfache Programm-Sketches schreiben.

Funktionsdefinitionen fassen Anweisungen zusammen. Das folgende Programmfragment zeigt die Definition einer Funktion namens `name`, die in ihrem Funktionskörper die beiden Anweisungen `anweisung1` und `anweisung2` enthält. Anweisungen werden durch Semikolon (`;`) voneinander getrennt.

```
void name() {
  anweisung1;
  anweisung2;
}
```

Eine Funktion wird in der Regel genutzt, um alle für eine komplexe Aufgabe benötigten Anweisungen zusammenzufassen und einen passenden Namen zu geben. Die Anweisungen im Körper einer Funktion werden nacheinander von oben nach unten ausgeführt.

Da die Ausführung der einzelnen Anweisungen je nach Komplexität nur wenige Mikrosekunden beträgt entsteht oft der Eindruck, als würden Dinge gleichzeitig passieren. Tatsächlich werden alle Anweisungen aber der Reihe nach ausgeführt. Dementsprechend können auch widersprüchliche Anweisungen konfliktfrei untereinander stehen und beispielsweise eine Lampe ein- und unverzüglich danach wieder ausgeschaltet werden. Diese Ereignisse passieren so kurz nacheinander, dass der Anwender nicht erkennen kann, dass z.B. eine Leuchtdiode für wenige Mikrosekunden eingeschaltet war.

Die Ausführung der Anweisungen kann direkt etwas bewirken und z.B. eine Leuchtdiode am ftDuino aufleuchten lassen:

```
void SchalteLeuchtdiodeEin() {
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH);
}
```

Eine Funktion in der Sprache C++ ist sehr nah an mathematische Funktionen angelehnt und kann wie diese ein Ergebnis liefern. Ob und was für ein Ergebnis die Funktion zurückliefert steht vor dem Funktionsnamen. Im vorliegenden Fall soll kein Ergebnis geliefert werden, daher steht vor dem Funktionsnamen `void`, englisch für "nichts". Ausserdem kann eine Funktion einen oder mehrere zu verarbeitende Eingabewerte erhalten, die zwischen den runden Klammern (`()`) angegeben werden. Die vorliegende Funktion benötigt keine Eingabewerte, die Klammern bleiben also leer.

Abbildung 4.6: Funktionsdefinition

Die Definition einer Funktion bewirkt, dass der Compiler die entsprecheden Anweisungen übersetzt und die nötigen Maschinenbefehle hintereinander im Maschinencode ablegt. Damit ist aber noch nichts darüber gesagt, wann diese Anweisungen tatsächlich ausgeführt werden.

Ausgeführt werden die Anweisungen der Funktion, sobald die Funktion aufgerufen wird. Dazu wird muss lediglich der Funktionsname mit den Parametern als Anweisung eingegeben werden. Da die Funktion `SchalteLeuchtdiodeEin` keine Parameter erwartet bleibt der Bereich zwischen den runden Klammern beim Aufruf leer. Der Funktionsaufruf muss dabei selbst zwingend im Funktionskörper einer Funktionsdefinition stehen.

```
void setup() {
  SchalteLeuchtdiodeEin();
}
```

Damit wird klar, dass die Anweisungen `pinMode(LED_BUILTIN, OUTPUT);` und `digitalWrite(LED_BUILTIN, HIGH);` ebenfalls Funktionsaufrufe waren. Beide Funktionen bekamen jeweils zwei durch Kommata getrennte Parameter mitgegeben.

4.3.4 Die Funktionen `setup()` und `loop()`

https://www.arduino.cc/reference/en/language/structure/sketch/setup/
https://www.arduino.cc/reference/en/language/structure/sketch/loop/

Wenn Funktionen selbst immer nur aus anderen Funktionen aufgerufen werden dürfen ergibt sich ein Henne-Ei-Problem bei der Frage, von wo aus denn der erste Funktionsaufruf erfolgt.

Abbildung 4.7: Ablauf eines Sketches

An dieser Stelle kommen die beiden Funktionsdefinitionen `setup()` und `loop()` ins Spiel, über die jeder Sketch mindestens verfügen muss. Fehlt eine oder beide Defintionen, so bricht die Arduino-IDE die Übersetzung mit einer Fehlermeldung ab.

Beiden Funktionen müssen nicht explizit aufgerufen werden. Stattdessen wird ihr Aufruf von der Arduino-IDE während der Übersetzung des Sketches automatisch in den erzeugte Maschinencode eingefügt. Die Funktion `setup()` wird dabei einmal bei Sketch-Start aufgerufen und die Funktion `loop()` wird in Folge immer wieder aufgerufen wie in Abbildung 4.7 dargestellt.

Mit Sketch-Start ist im Falle des ftDuino eine von drei Situationen gemeint:

1. Direkt nach dem Download eines übersetzten Sketches wird dessen Code gestartet.
2. Nach dem Anlegen der Stromversorgung an den ftDuino wird ein vorher per Download installierter Sketch-Code gestartet.
3. Wurde der Bootloader des ftDuino durch Druck auf den Reset-Taster gestartet, so bleibt dieser für acht Sekunden aktiviert. Wird der Bootloader in dieser Zeit nicht vom PC aus angesprochen, so beendet sich der Bootloader und der zuletzt heruntergeladene Sketch-Code wird stattdessen gestartet.

Hinweis für erfahrene Nutzer

Wer schon einmal mit C++ auf einem PC oder ähnlich zu tun hatte wird an dieser Stelle ggf. überrascht sein. In jenem Fall gab es die speziellen Funktionen `setup()` und `loop()` nicht. Stattdessen gab es eine Funktion namens `main()`, die bei Programmstart automatisch aufgerufen wurde.

Die Entwickler der Arduino-IDE haben sich entschieden, an dieser Stelle vom üblichen Standard abzuweichen, da das Konzept der `main()`-Funktion eher für nur zeitweilig auf einem komplexen Computer laufende Programme entwickelt wurde und sich nur bedingt in die Welt der Hardware-Programmierung einfügt.

Bibliotheksfunktionen

Wie im Falle der `name()`-Funktion geschehen kann man eigene Funktionen schreiben. Die Arduino-IDE bringt aber bereits eigene Funktionssammlungen mit. Die Sammlungen bestehen in erster Linie aus Funktionen, die man nicht selbst schreiben muss. Sie sind dem System von Vornherein mit Namen bekannt und man kann sie in eignene Programmen einfach aufrufen, ohne sie selbst schreiben zu müssen.

Die Funktionen `pinMode()` und `digitalWrite()` sind solche Bibliotheksfunktionen. Während die Sprache C++ universell ist und sich bei der Verwendungen auf einem PC oder einem Arduino nicht unterscheidet sind die Bibliotheksfunktionen in der Regel plattformspezifisch. Die Funktion `pinMode()` steht dem Programmierer nur auf dem Arduino zur Verfügung und bei der Entwicklung eines Programmes für einen Windows-PC würde der Aufruf dieser Funktion zu einem Übersetzungsfehler führen.

Dies ist auch der Grund, warum sich viele C++-Beispiele und Tutorials aus dem Internet nicht direkt auf den Arduino übertragen lassen. Handelt es sich bei diesen Prorgammen nicht zufällig bereits um Arduino- sondern um PC-Programme,

dann werden dort Funktions-Bibliotheken verwendet, die wiederum auf dem Arduino nicht vorhanden sind. Das liegt in erster Linie an der deutlich unterschiedlichen Hardware der verschiedenen Plattformen. Während ein PC-Programm in erster Linie Fenster auf dem Bildschirm öffnet und mit Benutzerumgabe umgeht wird ein Arduino-Programm eher Hardware-Eingänge auswerten und Ausgänge schalten.

Hinweis für erfahrene Nutzer

Auch bei der Verwendung von Bibliotheken gibt es einen Unterschied zu üblicher C++-Programmierung auf PCs. Die Arduino-IDE macht die grundlegenden Arduino-spezifischen Bibliotheken automatisch im Sketch bekannt, so dass sich Funktionen wie `pinMode()` direkt nutzen lassen.

Gängige C++-Compiler binden von sich aus keine Bibliotheken ein. Sie müssen durch sogenannte `#include`-Anweisungen explizit bekannt gemacht werden. Dies ist auf dem Arduino zumindest für die meisten mitgelieferten Bibliotheken nicht nötig.

4.3.5 Beispiel

Damit sind die wesentlichen Grundlagen erklärt und das folgende Beispiel verständlich.

```
/* Funktion zum Einschalten der LED */
void SchalteLedEin() {
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH);
}

void setup() {
  SchalteLedEin();
}

void loop() {
}
```

Bei Systemstart wird zunächst die `setup()`-Funktion aufgerufen. Die Funktion `setup()` enthält eine einzelne Anweisung in Form eines Aufrufs der Funktion `SchalteLedEin`. Diese Funktion ist wiederum in den Zeilen 2 bis 5 des Sketches definiert und besteht ihrerseits aus dem Aufruf zweier Bibliotheksfunktionen, die die eingebaute Leuchtdiode des ftDuino aufleuchten lassen.

Die `loop()`-Funktion wird nach Systemstart permanent immer wieder aufgerufen. Sie ist aber leer, über das Einschalten der Leuchtdiode direkt bei Systemstart zeigt der Sketch also keine weitergehende Funktion.

Der Kommentar in Zeile eins dient lediglich der Erklärung und hat keinen Einfluss auf die Übersetung des Sketches oder seine Funktion.

4.4 Hilfreiche Bibliotheksfunktionen

Wie in den vorigen Abschnitten angedeutet bringt die Arduino-IDE einige Bibliotheken mit, die Funktionen zur Verwendung spezieller Eigenschaften des Arduino bieten. Zusätzlich bringt die ftDuino Installtion eigene Bibliotheken mit, um auf die fischertechnik-Ein- und -Ausgänge des ftDuino zuzugreifen.

Im folgenden werden sieben der am häufigsten benötigten Funktionen beschrieben. Mit ihnen lassen sich bereits vielfältige Sketches schreiben.

Eine Beschreibung weiterer Funktionen der Arduino-IDE findet man online in derArduino-Language-Reference unter https://www.arduino.cc/reference/en/#functions während weitere ftDuino-spezifische Funktionen im Kapitel 8 beschrieben sind.

4.4.1 `pinMode(pin, mode)`

https://www.arduino.cc/reference/en/language/functions/digital-io/pinMode/

Diese Funktion konfiguriert einen Pin des ATmega32u4-Mikrocontrollers als Ein- (`mode = INPUT`) oder Ausgang (`mode = OUTPUT`). Diese Funktion wird auf Arduinos sehr häufif verwendet, da dort direkt mit den Anschlüssen des Mikrocontrollers gearbeitet wird. Auf dem ftDuino befinden sich an den meisten Anschlüssen Zusatzschaltungen zur Verbindung mit fischertechnik-Komponenten, die die direkte Verwendung der `pinMode()`-Funktion unnötig machen. Stattdessen bringen die ftDuino-Bibliotheken alle Funktionen mit, um die Ein- und Ausgänge fischertechnik-konform zu bedienen.

Die wesentliche Ausnahme bildet der Anschluss der internen roten Leuchtdiode. Für `pin` muss in dem Fall `LED_BUILTIN` verwendet werden.

```
// internen Anschlusspin der Leuchtdiode zum Ausgang erklären
pinMode(LED_BUILTIN, OUTPUT);
```

4.4.2 `digitalWrite(pin, value)`

https://www.arduino.cc/reference/en/language/functions/digital-io/digitalwrite/

Die Funktion `digitalWrite()` steuert einen durch die Funktion `pinMode()` zum Ausgang erklärten Pin. Sie wird daher im ftDuino in der Regel für die Leuchtdiode verwendet und in seltenen Fällen zur Steuerung der beiden am I^2C-Anschluss verfügbaren Signale.

Für `pin` gelten die gleichen Werte wie bei `pinMode()`-Funktion. Der Wert für `value` kann `HIGH` oder `LOW` sein, was den entsprechenden Pin ein- oder auschaltet.

```
// internen Anschlusspin der Leuchtdiode zum Ausgang erklären
pinMode(LED_BUILTIN, OUTPUT);
// Leuchtdiode einschalten
digitalWrite(LED_BUILTIN, HIGH);
```

4.4.3 `delay(ms)`

https://www.arduino.cc/reference/en/language/functions/time/delay/

Die meisten Aufgaben erledigt auch ein so einfacher Mikrocontroller wie der ATmega32u4 schneller als für Menschen wahrnehmbar. Um Abläufe auf ein passendes Maß zu senken ist die `delay()`-Funktion hilfreich. Sie erwartet als Parameter eine Wartezeit in Millisekunden.

Wie in Abschnitt 4.3.4 beschrieben wird die Funktion `setup()` nur einmal aufgerufen, die Function `loop()` aber daraufhin immer wieder. Der folgende Sketch lässt die Leuchtdiode daher kontinuierlich blinken.

```
void setup() {
  // internen Anschlusspin der Leuchtdiode zum Ausgang erklären
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  // Leuchtdiode einschalten
  digitalWrite(LED_BUILTIN, HIGH);
  // 1 Sekunde warten
  delay(1000);
  // Leuchtdiode ausschalten
  digitalWrite(LED_BUILTIN, LOW);
  // 1 Sekunde warten
  delay(1000);
}
```

4.4.4 `Serial.begin(speed)`

https://www.arduino.cc/reference/en/language/functions/communication/serial/begin/

Die Kommunikationsmöglichkeiten des ftDuino beschränken sich direkt am Gerät im Wesentlichen auf die fischertechnik Ein- und Ausgänge. Bei komplexeren Projekten wird es schwierig, nur am Verhalten des ftDuino den Programmfluss zu

verfolgen. Ein einfacher Weg, aus dem Sketch heraus Informationen an den Benutzer auszugeben ist die Verwendung der `Serial`-Bibliothek.

Auf PC-Seite werden die Ausgaben dieser Bibliothek über den sogenannten seriellen Monitor angezeigt. Dessen Benutzung wurde in Abschnitt 3.3.1 genauer beschrieben. Die Funktion `Serial.begin()` bereitet den Sketch auf die Verwendung des seriellen Monitors vor. Sie sollte daher am Beginn des Sketches bzw. in der `setup()`-Funktion aufgerufen werden.

Die Funktion `Serial.begin()` erwartet einen Parameter namens `speed`. Dieser Wert ist für die USB-Verbindung des ftDuino belanglos und sollte z.B. auf einen Wert von 115200 gesetzt werden.

```
void setup() {
  // Vorbereiten der seriellen Verbindung
  Serial.begin(115200);
}

void loop() {
}
```

4.4.5 `Serial.print(val)` und `Serial.println(val)`

https://www.arduino.cc/reference/en/language/functions/communication/serial/print/
https://www.arduino.cc/reference/en/language/functions/communication/serial/println/

Ist eine seiellen Verbindung per `Serial.begin()` eingerichtet, so können die Funktionen `Serial.print()` und `Serial.println()` zur Ausgabe von Nachrichten an den seriellen Monitor verwendet werden. Der Unterschied zwischen `Serial.print()` und `Serial.println()` liegt darin, dass nach der Ausgabe per `Serial.println()` eine neue Ausgabezeile begonnen wird, während weitere Ausgaben nach `Serial.print()` direkt in der gleichen Zeile erfolgen. Die Funktion `Serial.println()` wird daher oft genutzt, um komplexere Ausgaben abzuschließen.

Für `val` können unter anderem Zeichenketten und Zahlen verwendet werden. Zeichenketten müssen in doppelte Anführungszeichen (“) eingeschlossen werden.

```
void setup() {
  // Vorbereiten der seriellen Verbindung
  Serial.begin(115200);
  // 2 Sekunden Verzögerung, um dem PC Zeit zu geben,
  // die Verbindung entgegen zu nehmen
  delay(2000);
  // ein paar Ausgaben
  Serial.print("Die Antwort lautet: ");
  Serial.println(42);
}

void loop() {
}
```

4.4.6 `ftduino.input_get()`, `ftduino.output_set()` und `ftduino.motor_set()`

Ein Controller für ein fischertechnik-Modell muss natürlich Eingaben aus dem Modell empfangen und Reaktionen im Modell auslösen können. Die ftDuino-Bibkiotheken sind in allem Detailgrad in Kapitel 8 beschrieben. In diesem Absatz wird daher nur das absolute Minimum beschrieben.

Es wurde in Abschnitt 4.3.4 erklärt, dass es eine Besonderheit ist, dass Biblitheksfunktionen ohne weitere Vorbedingungen zur Verfügung stehen und im Sketch verwendet werden können. Dies trifft aber nur auf die Arduino-eigenen Bibliotheken zu wie sie in den bisherigen Beispielen verwendet wurden. Die ftDuino-Bibliotheken stehen nicht automatisch zur Verfügung sondern müssen am Anfang des Sketches mit einer `#include`-Anweisung bekannt gemacht werden.

```
#include <FtduinoSimple.h>

void setup() {
}

void loop() {
}
```

Erst dann sind die Funktionen aus dieser Bibliothek im Sketch nutzbar.

Abbildung 4.8: Lampe an Ausgang O1

Die wichtigsten Funktionen dafür sind `ftduino.input_get(port)` zum Abfragen eines Eingangs und `ftduino.output_set(port, mode)` zum Schalten eines Ausgangs. Bei `ftduino.input_get(port)` sind für `port` Werte von `Ftduino::I1` bis `Ftduino::I8` erlaubt. Zum Schalten eines Ausgangs gibt es die Funktion `ftduino.output_set(port, mode)`, wobei `port` Werte von `Ftduino::O1` bis `Ftduino::O8` annehmen darf und `mode` auf `Ftduino::HI` gesetzt wird, wenn der Ausgang eingeschaltet werden soll und auf `Ftduino::OFF` gesetzt wird, um den Ausgang auszuschalten.

Die Funktion `ftduino.motor_set(port, mode)` ist `ftduino.output_set(port, mode)` sehr ähnlich, nur dass hier ein Motorausgang `M1` bis `M4` geschaltet werden kann. Der Wert for `mode` kann dann `Ftduino::LEFT`, `Ftduino::RIGHT` oder `Ftduino::OFF` sein, je nachdem, ob sich der Motor linksherum, rechtsherum oder gar nicht drehen soll.

```
#include <FtduinoSimple.h>

void setup() {
  // Eingang I1 lesen
  ftduino.input_get(Ftduino::I1);
  // Ausgang O1 einschalten
  ftduino.output_set(Ftduino::O1, Ftduino::HI);
}

void loop() {
}
```

Dieser Sketch würde eine wie in Abbildung 4.8 angeschlossene Lampe zum Leuchten bringen. Achtung: Dazu muss der ftDuino mit 9 Volt versorgt sein.

4.5 Variablen

https://www.arduino.cc/reference/en/#variables

Oft besteht der Bedarf in einem Sketch etwas zu "merken" bzw. zu speichern. Wenn etwas beispielsweise mit einer bestimmten Häufigkeit passieren soll, dann muss währenddessen irgendwo vermerkt werden, wie oft es schon passiert ist bzw. wie oft es noch zu passieren hat. Ein anderes Beispiel sind einmalige Ereignisse wie "der Benutzer hat einen Taster gedrückt". Auch wenn dieses Eregnis vorübergeht und der Benutzer den Taster losgelassen hat soll die Aktion ggf. weitergeführt werden. Dazu muss irgendwo vermerkt werden, dass der Taster vor kurzem noch gedrückt war.

Für diesen Zweck gibt es sogenannte Variablen. Mit ihnen weist man den Compiler an, Platz im Speicher des ftDuino für etwas "gemerktes" zu reservieren. Während der Sketch im sogenannten Flash-Speicher des ftDuino abgelegt wird und daher auch über das Ausschalten des ftDuino hinaus erhalten bleibt. Wird der Speicherplatz für Variablen im flüchtigen RAM-Speicher des ftDuino abgelegt. Der gespeierte Wert einer Veriablen geht also beim Ausschalten des ftDuino verloren.

Eine Variable wird wie eine Funktion im Sketch definiert. Dazu erhält sie wie eine Funktion einen Namen. Außerdem muss angeben werden, welcher Art die in der Variablen abzulegenden Daten sind. Im folgenden Beispiel wird eine Variable namens `variablenName` für Daten vom Typ `int` angelegt.

```
int variablenName;

void setup() {
}

void loop() {
}
```

4.5.1 Datentyp `int`

https://www.arduino.cc/reference/en/language/variables/data-types/int/

Der Datentyp `int` ist der Standard-Datentyp. Beim ftDuino erlaubt dieser Datentyp das Ablegen ganzzahliger Werte im Bereich von -32768 bis +32767. Für die meisten Verwendungen ist das ausreichend.

Eine Variable wird genutzt, um Daten in ihr abzulegen und später wieder abzurufen. Das Ablegen von Werte geschieht, indem der Variablen ein Wert mit dem Gleichheitszeichen (=) zugewiesen wird. Die Ausdrücke auf der rechten Seite der Zuweisung können komplexe mathematische Funktionen beinhalten und sogar Funktionsaufrufe, um beispielsweise den Zustand eines Eingangs des ftDuino zu speichern.

```
// eine einfache Zuweisung
variablenName = 42;
// ein komplexer Ausdruck
variablenName = (4*8*8+38)/7;
// Zustand des Eingangs I1
variablenName = ftduino.input_get(Ftduino::I1);
```

Außerdem können in den Ausdrücken wiederrum Variablen enthalten sein. Dabei kann auch die auf der linken Seite stehende Variable auch auf der rechten Seite auftauchen, was dem mathematischen Verständnis einer Gleichung etwas widerspricht.

```
// Wert aus einer anderen Variable übernehmen
variablenName = andererVariablenName;
// Wert der Variablen verändern und wieder der Variablen zuweisen
variablenName = variablenName + 1;
```

Man darf diese Art der Zuweisung nicht als mathematischen Vergleich lesen, auch wenn die Darstellung das nahelegt. Stattdessen wird zunächst der Ausdruck auf der rechten Seite bereichnet und das Ergebnis wird danach der Variablen auf der linken Seite zugewiesen. Dieser zeitliche Ablauf führt dazu, dass eine Zuweisung wie

```
variablenName = variablenName + 1;
```

einen Sinn ergibt.

Variablen können auch als Parameter bei Funktionsaufrufen verwendet werden.

```
int variablenName;

void setup() {
  Serial.begin(115200);
  variablenName = 192/4;

  Serial.print("Variableninhalt: ");
  Serial.println(variablenName);
}

void loop() {
}
```

Wichtig ist hier die korrekte Verwendung der Anführungszeichen (“). Ein Wort oder Text in Anführungszeichen steht für den Text selbst und wird nicht weiter interpretiert. Ein Wort ohne Anführungszeichen kann u.a. für eine Anweisung, einen Funktionsnamen oder einen Variablennamen stehen und der Compiler versucht diesem Wort eine Bedeutung zuzuweisen.

```
  // gib das Wort variablenName aus
  Serial.println("variablenName");
  // gib den Inhalt der Variablen namens variablenName aus
  Serial.println(variablenName);
```

4.6 Bedingungen

Bisher haben alle Beispiele aus Anweisungen in einer festen Reihenfolge bestanden. Alle Anweisung wurden immer auf die gleiche Weise ausgeführt und z.B. Leuchtdioden haben geblinkt oder Meldungen wurden ausgegeben. Es ist aber zu keiner Zeit etwas abhängig von irgendeinem anderen Ereignis passiert.

Es ist aber bei der Robotersteuerung und auch generell in der Programmierung oft nötig, dass ein Programm auf Ereignisse reagiert. Der C++-Mechanismus dafür sind Anweisungen, die auf Bedingungen reagieren können.

4.6.1 `if`-Anweisung

https://www.arduino.cc/reference/en/language/structure/control-structure/if/

Abbildung 4.9: `if`-Anweisung

Die `if`-Anweisungen ist solch eine Anweisung. Sie erwartet eine Bedingung in runden Klammern und die auf die `if`-Anweisung folgende Anweisung im Bedingungskörper wird nur dann ausgeführt, wenn die Bedingung sich als "wahr" herausstellt.

```
if(12 > 5+3)
  Serial.println("zwoelf ist groesser als die Summe aus fuenf und drei");
```

In der Bedingung können unterschiedliche Vergleichsoperatoren verwendet werden. Die wichtigsten sind:

C++-Schreibweise	Vergleichsoperation
>	größer als
<	kleiner als
==	gleich
!=	ungleich

Abbildung 4.10: Taster an Eingang I1

Die Bedingung kann auch Funktionsaufrufe enthalten. Im Falle der `ftduino.input_get()`-Funktion ist das Ergebnis des Funktionsaufrufs bereits ein Wahrheitswert (wahr oder unwahr) und der Funktionsaufruf kann direkt verwendet werden. Soll mehr als eine Anweisung von der `if`-Anweisung betroffen sein, so kann man sie durch geschweifte Klammern ({ und }) zusammenfassen.

```
#include <FtduinoSimple.h>

void setup() {
  // Vorbereiten der seriellen Verbindung
  Serial.begin(115200);
}
```

```
void loop() {
  // testen, on die Taste an I1 gedrückt ist
  if(ftduino.input_get(Ftduino::I1)) {
    Serial.print("Der Taster ist gedrueckt");
    // eine viertel Sekunde warten
    delay(250);
  }
}
```

4.7 Schleifen

Programmschleifen sind ebenfalls ein sehr fundamentales Konzept. Erst durch sie wird es möglich, Teile eines Programmes mehrfach auszuführen. Ohne sie würden alle Anweisungen eines Programmes einmal der Reihe nach abgearbeitet.

Allerdings gab es in den bisherigen Beispilen bereits das ein oder andere Programm, das etwas mehrfach getan hat. Das liegt an der Arduio-spezifischen `loop()`-Funktion. Sie wird automatisch während des Programmablaufs immer wieder aufgerufen wie in Abbildung 4.7 abgebidet. Dadurch lassen sich in Arduino-Sketches auch ohne den Einsatz von entsprechenden C++-Anweisungen Programmteile wiederholen.

Dennoch ist es hilfreich, auch auf die C++-eigenen Mechanismen für Schleifen zurückgreifen zu können. Im Folgenden werden zwei davon beschrieben.

4.7.1 `while`-Schleife

https://www.arduino.cc/reference/en/language/structure/control-structure/while/

Abbildung 4.11: `while`-Schleife

Die `while`-Schleife erlaubt es, den folgenden Befehl im sogenannten Schleifenkörper solange zu widerholen, wie die Bedingung zwischen ihren runden Klammern erfüllt ist. Wie schon bei der `if`-Anweisung können mehrere folgenden Anweisungen durch geschweifte Klammern zusammengefasst werden. Die `while`-Schleife gilt dann für den gesamten Anweisungsblock. Ist die Bedingung von vornherein nicht erfüllt, so wird der Schleifenkörper nicht ausgeführt.

```
while(variablenName < 12) {
  Serial.println("Variable ist kleiner 12");
  variablenName = variablenName + 1;
}
```

Die Bedingung gleicht ebefalls der der `if`-Anweisung und kann die gleichen Operatoren enthalten. Das folgende Beispiel erweitert das Beispiel aus dem Abschnitt der `if`-Anweisung so, dass auf das Loslassen der Taste gewartet wird.

```
#include <FtduinoSimple.h>

void setup() {
  // Vorbereiten der seriellen Verbindung
  Serial.begin(115200);
}

void loop() {
  // testen, on die Taste an I1 gedrückt ist
  if(ftduino.input_get(Ftduino::I1)) {
    Serial.print("Der Taster ist gedrueckt");
    // eine viertel Sekunde warten
    delay(250);
```

```
      // warten, bis die Taste wieder losgelassen wird
      while(ftduino.input_get(Ftduino::I1)) {
        // der Raum zwischen den geschweiften Klammern kann
        // auch ganz leer bleiben, wenn während der
        // Widerholung keine weiteren Anweisungen aus
        // geführt werden sollen
      }
    }
  }
```

4.7.2 `for`-Schleife

https://www.arduino.cc/reference/en/language/structure/control-structure/for/

Etwas komplexer als die `while`-Schleife ist die `for`-Schleife. Sie enthält zwischen ihren Runden Klammern gleich drei durch Semikolon getrennte Anweisungen. Die erste wird vor Schleifenbeginn einmal ausgeführt, die zweite wird während der Ausführung der Schleife ausgewertet und bestimmt, wie die Bedingung der `while`-Schleife wie fäufig die Schleife ausgeführt wird. Die dritte Anweisung wird schliesslich *nach* jeder Ausführung des Schleifkörpers ausgeführt. Ist die Bedingung von vornherein nicht erfüllt, so wird der Schleifenkörper nicht ausgeführt. Die Vor-Anweisung wird in jedem Fall ausgeführt.

Abbildung 4.12: `for`-Schleife

Was recht aufwendig klingt wird versändlich, wenn man sich die übliche Anwednung der `for`-Schleife ansieht: Das Widerholung eines Befehls mit einer bestimmten Häufigkeit.

```
for(variablenName = 0 ; variablenName < 12; variableName = variableName + 1)
  Serial.println("Dieser Text wird 12 mal ausgegeben");
```

Die drei Anweisungen bzw. Bedingungen innerhalb der runden Klammern lauten:

```
variablenName = 0;
variablenName < 12;
variableName = variableName + 1;
```

Die erste Anweisung wird nur einmal zu Beginn der Schleife ausgeführt. In diesem Fall schreibt sie den Wert 0 in die Variable `variablenName` die zweite Anweisung ist die Bedinging, die bestimmt wie oft die Schleife ausgeführt wird. In diesem Fall solange der Inhalt der Variable `variablenName` kleiner als 12 ist. Und die dritte Anweisung wird schließlich am Ende jedes Durchlaufs der `for`-Schleife ausgeführt. In diesem Fall wird dort der Inhalt der Variablen `variablenName` um eins erhöht. In diesem Beispiel passiert also:

1. Der Inhalt der Variablen `variablenName` wird auf 0 gesetzt
2. Solange der Inhalt der Variablen `variablenName` kleiner als 12 ist ...
   (a) ... werden die Anweisungen im Schleifenkörper ausgeführt ...
   (b) ... und danach der Inhalt der Variablen `variablenName` um eins erhöht

Der Schleifenkörper wird also genau 12 mal ausgeführt.

4.8 Beispiele

Die in diesem Kapitel vorgestellten C++-Sprachkonstrukte sowie die ausgewählten Funktionen der Arduino- und der ftDuino-Bibliotheken bilden zwar nur einen geringen Teil ab. Aber schon diese wenigen Informationen reichen, um einige interessante Sketches selbst zu schreiben.

4.8.1 Einfache Ampel

Dieses Beispiel stellt eine eine einfache Bedarfs-Ampel dar. Nach dem Einschalten zeigt sie zunächst rot. Sobald der Taster gedrückt wird springt sie für 10 Sekunde auf grün und wechselt dann zurück in den Anfangszustand.

Abbildung 4.13: Einfache Ampel

Der dazugehörige Sketch ist sehr einfach. In der `setup()`-Funktion wird in Zeile 5 die rote Lampe eingeschaltet.

In der `loop()`-Funktion wird in Zeile 10 permanent getestet, ob der Taster gedrückt ist. Ist er gedrückt, so wird der gesamte Bedingungskörper in den Zeilen 11 bi 20 ausgeführt.

Dort wird in Zeile 12 die rote Lampe aus- und in Zeile 14 die grüne Lampe eingeschaltet. In Zeile 16 wird 10000 Millisekunden, also 10 Sekunden gewartet, bevor in den Zeilen 18 und 20 die grüne Lampe zunächst aus- und die rote dann eingeschaltet wird.

Dies ist ein Beispiel für Dinge, die aufgrund der Geschwindigkeit des Mikroprozessors als gleichzeitig wahrgenommen werden. Obwohl die Lampen in den Zeilen 12 und 14 bzw. 18 und 20 jeweils nacheinander ein- bzw. ausgeschaltet werden ist keine zeitliche Differenz erkennbar. Der Abstand von wenigen Mikrosekunden ist nicht erkennbar.

```
1  #include <FtduinoSimple.h>
2
3  void setup() {
4    // beim Start der Ampel leuchtet die rote Lampe
5    ftduino.output_set(Ftduino::O1, Ftduino::HI);
6  }
7
8  void loop() {
9    // testen, on die Taste an I1 gedrückt ist
10   if(ftduino.input_get(Ftduino::I1)) {
11     // rote Lampe ausschalten
12     ftduino.output_set(Ftduino::O1, Ftduino::OFF);
13     // grüne Lampe einschalten
14     ftduino.output_set(Ftduino::O2, Ftduino::HI);
15     // zehn Sekunden warten
16     delay(10000);
17     // grüne Lampe ausschalten
18     ftduino.output_set(Ftduino::O2, Ftduino::OFF);
19     // rote Lampe einschalten
20     ftduino.output_set(Ftduino::O1, Ftduino::HI);
21   }
22 }
```

4.8.2 Schranke

Das Schrankenbeispiel ist etwas komplexer. Es besteht aus einer motorisierten Schranke mit jeweils zwei Endtastern. Der Taster an `I2` wird betätigt, wenn sich die Schranke komplett geöffnet hat, der an `I3` wird betätigt, wenn die Schranke vollständig geschlossen ist.

Achtung: Taster `I2` und `I3` sind so verdrahtet, dass der Kontakt im unbetätigten Zustand geschlossen ist. Er öffnet sich, sobald die Taster betätigt werden, die Schranke also ganz geöffnet oder geschlossen ist.

Abbildung 4.14: Schranke

Bei Sketchstart wird zunächst der Motor in Zeile 10 so lange linksherum gedreht, solange der Kontakt des Tasters an `I2` in Zeile 15 als gechlossen erkannt wird, also solange die Schranke nicht vollständig geöffnet ist. Sobald der Taster betätigt wird wird der Motor in Zeile 18 gestoppt.

Sobald die Taste an `I1` in Zeile 23 gedrückt wird wird der Motor in Zeile 26 diesmal rechtsherum gestartet bis der Taster an `I3` betätigt wird. In Zeile 28 wird der Motor dann gestoppt.

In den Zeile n 31 bis 40 wird die Lampe an `O3` insgesamt fünfmal mit einer Pause von jeweils 500 Millisekunden ein- und wieder ausgeschaltet.

Danach wird die Schranke schließlich in den Zeilen 43 bis 45 wieder geschlossen.

```
1  #include <FtduinoSimple.h>
2
3  // Zählervariable zum Blinken
4  int zaheler;
5
6  void setup() {
7    // Beim Start fährt die Schranke auf
8
9    // motor linksherum drehen
10   ftduino.motor_set(Ftduino::M1, Ftduino::LEFT);
11
12   // Warten bis Schranke geschlossen offen ist. Da der Taster als
13   // Öffner verdrahtet ist läuft der Motor, solange der Taster
14   // geschlossen ist
15   while(ftduino.input_get(Ftduino::I2)) { }
16
17   // Solbald der Taster nicht mehr geschlossen ist Motor stoppen
18   ftduino.motor_set(Ftduino::M1, Ftduino::OFF);
19 }
20
21 void loop() {
22   // testen, on die Taste an I1 gedrückt ist
23   if(ftduino.input_get(Ftduino::I1)) {
24
25     // Motor rechtsherum drehen, bis Taster I3 geöffnet ist
26     ftduino.motor_set(Ftduino::M1, Ftduino::RIGHT);
27     while(ftduino.input_get(Ftduino::I3)) { }
28     ftduino.motor_set(Ftduino::M1, Ftduino::OFF);
29
30     // fünfmal mit der Lampe blinken
31     for(zaehler=0;zaehler<5;zaehler=zaehler+1) {
32       // Lampe an
33       ftduino.output_set(Ftduino::O3, Ftduino::HI);
34       // 500ms warten
35       delay(500);
36       // Lampe aus
37       ftduino.output_set(Ftduino::O3, Ftduino::OFF);
38       // 500ms warten
39       delay(500);
40     }
```

```
41
42      // Motor linksherum drehen, bis Taster I2 geöffnet ist
43      ftduino.motor_set(Ftduino::M1, Ftduino::LEFT);
44      while(ftduino.input_get(Ftduino::I2)) { }
45      ftduino.motor_set(Ftduino::M1, Ftduino::OFF);
46    }
47  }
```

4.9 Weiterführende Informationen

Es gibt im Internet viele Tutorials sowohl zur C++-Programmierung[2] als auch zur Arduino-Programmierung[3].

Diese und ähnliche Tutorials erklären viele weitere Sprachkonstrukte und Bibliotheksfunktionen. Solche Tutorials sind nicht auf den ftDuino zugeschnitten. Aber dieses Kapitel hat die nötigen Grundlagen geliefert, um auch mit anderen Tutorials weiterarbeiten zu können. Nicht alles aus der Welt der PC-Programmierung oder der Arduino-Programmierung lässt sich auf den ftDuino übertragen. Zusammen mit den Experimenten und Modellen der folgenden Kapitel sowie den fertig mitgelieferten Beispielprogrammen ist der tiefere Einstieg in die professionelle C++-Programmierung aber möglich.

[2] C++-Tutorial http://www.online-tutorials.net/c-c++-c/c++-tutorial-teil-1/tutorials-t-1-58.html
[3] Arduino-Tutorial http://www.arduino-tutorial.de

Kapitel 5

Experimente

Die Experimente dieses Kapitels konzentrieren sich auf einzelne Aspekte des ftDuino. Sie veranschaulichen einen Effekt oder ein Konzept und verwenden dafür nur minimale externe Komponenten. Die Experimente stellen an sich keine vollständigen Modelle dar, können aber oft als Basis dafür dienen.

Beispiele für komplexe Modelle finden sich im Kapitel 6.

5.1 Lampen-Zeitschaltung

Schwierigkeitsgrad: ★☆☆☆☆

Dieses sehr einfache Modell besteht nur aus einem Taster und einer Lampe und bildet die Funktion einer typischen Treppenhausbeleuchtung nach. Um Energie zu sparen wird hier nicht einfach ein Kippschalter genommen, um das Licht zu schalten. Stattdessen wird ein Taster verwendet und jeder Druck auf den Taster schaltet das Licht nur für z.B. zehn Sekunden ein. Wird während dieser Zeit der Taster erneut gedrückt, so verlängert sich die verbleibende Zeit wieder auf volle zehn Sekunden. Nach Ablauf der zehn Sekunden verlischt das Licht und das Spiel beginnt von vorn.

Abbildung 5.1: Lampen-Zeitschaltung

5.1.1 Sketch `LampTimer`

Der folgende Sketch findet sich bei installierter ftDuino-Unterstützung im Menü der Arduino-IDE unter `Datei` ▷ `Beispiele` ▷ `FtduinoSimple` ▷ `LampTimer`.

```
1  /*
2    LampTimer - Lampen-Zeitschaltuhr
3
4    (c) 2017 by Till Harbaum <till@harbaum.org>
5
```

```
6     Schaltet eine Lampe an Ausgang O1 für 10 Sekunden ein,
7     sobald ein Taster an Eingang I1 gedrückt wird.
8   */
9
10  #include <FtduinoSimple.h>
11
12  uint32_t startzeit = 0;
13
14  // Die Setup-Funktion wird einmal ausgeführt, wenn Reset gedrückt oder
15  // das Board gestartet wird.
16  void setup() { }
17
18  // Die Loop-Funktion wird endlos immer und immer wieder ausgeführt
19  void loop() {
20    // Teste, ob der Taster an I1 gedrückt ist
21    if(ftduino.input_get(Ftduino::I1)) {
22      // merke Startzeit
23      startzeit = millis();
24
25      // schalte Lampe ein (Ausgang HI)
26      ftduino.output_set(Ftduino::O1, Ftduino::HI);
27    }
28
29    // gültige Startzeit und seitdem mehr als 10 Sekunden
30    // (10.000 Millisekunden) verstrichen?
31    if((startzeit != 0) &&
32       (millis() > startzeit + 10000)) {
33      // vergiss Startzeit
34      startzeit = 0;
35      // schalte Lampe aus (Ausgang OFF)
36      ftduino.output_set(Ftduino::O1, Ftduino::OFF);
37    }
38  }
```

Sketchbeschreibung

Die für die Lampen-Zeitschaltung nötigen Funktionen des ftDuino sind sehr einfach und die Anwendung lässt sich mit der einfachen `FtduinoSimple`-Bibliothek (siehe Abschnitt 8.1) abdecken.

Der Arduino-Sketch enthält eine leere `setup()`-Funktion, da keine Initialisierung nötig ist. Sämtliche Funktionalität steckt in der `loop()`-Funktion.

Die Taste an `I1` wird über `input_get()` permanent abgefragt. Ist sie gedrückt, so wird die aktuelle Zeit seit Gerätestart in Millisekunden mit der Funktion `millis()` abgefragt und in der Variablen `startzeit` gespeichert und die Lamoe wird eingeschaltet. War die Lampe bereits an, dann bewirkt dieses zusätzliche Einschalten nichts, aber der bereits gesetzte Zeitwert in `startzeit` wird durch den aktuellen ersetzt.

Unabhängig davon wird permanent getestet, ob `startzeit` einen gültigen Wert enthält und ob die aktuelle Systemzeit bereits mehr als zehn Sekunden (10.000 Millisekunden) nach dem dort gespeicherten Wert liegt. Ist das der Fall, dann sind seit dem letzten Tastendruck mehr als zehn Sekunden vergangen und die Lampe wird ausgeschaltet sowie der Wert in `startzeit` auf Null gesetzt, um ihn als ungültig zu markieren.

Aufgabe 1: 20 Sekunden

Sorge dafür, dass die Lampe nach jedem Tastendruck 20 Sekunden lang an bleibt.

Lösung 1:

In Zeile 32 muss der Wert 10000 durch den Wert 20000 ersetzt werden, damit die Lampe 20000 Millisekunden, also 20 Sekunden eingeschaltet bleibt.

```
31    if((startzeit != 0) &&
32       (millis() > startzeit + 20000)) {
```

Aufgabe 2: Keine Verlängerung

Sorge dafür, dass ein weiterer Druck auf den Taster, während die Lampe bereits leutet, die verbleibende Zeit nicht wieder auf 10 Sekunden verlängert.

Lösung 2:

Vor der Zuweisung in Zeile 23 muss eine zusätzliche Abfrage eingefügt werden, die nur dann einen neue Wert setzt, wenn bisher keiner gesetzt war. Beide Zeilen zusammen sehen dann so aus:

```
23    if(startzeit == 0)
24      startzeit = millis();
```

Expertenaufgabe:

Schließt Du statt der Lampe eine Leuchtdiode an (der rote Anschluss der Leuchtdiode muss an Ausgang `O1`), dann wirst Du etwas merkwürdiges bemerken, wenn das Licht eigentlich aus sein sollte: Die Leuchtdiode leuchtet trotzdem ganz schwach, obwohl der Ausgang doch `OFF` ist. Wie kommt das?

Erklärung

Durch eine Lampe oder Leuchtdiode fließt ein Strom, wenn zwischen den beiden Anschlüssen ein Spannungsunterschied besteht. Den einen Anschluss haben wir fest mit Masse bzw. 0 Volt verbunden, der andere ist offen und wird von den Bauteilen im ftDuino nicht mit Spannung versorgt. Anders als bei einem mechanischem Schalter ist diese Trennung bei dem im ftDuino als sogenannter Ausgangstreiber verwendeten Halbleiterbaustein aber nicht perfekt. Ein ganz kleiner sogenannter Leckstrom fließt trotzdem zur 9V-Versorgungsspannung. Dieser kleine Strom reicht nicht, die Lampe zum Leuchten zu bringen. Aber er reicht, die wesentlich effizientere Leuchtdiode ganz leicht aufleuchten zu lassen.

Lässt sich daran etwas ändern? Ja! Statt den Ausgang komplett unbeschaltet zu lassen können wir dem Ausgangstreiber im ftDuino sagen, dass er den Ausgang fest auf Masse (0 Volt) schalten soll. Beide Anschlüsse der Leuchtdiode liegen dann fest auf Masse und die Einflüsse irgendwelcher Leckströme treten nicht mehr in Erscheinung. Dazu muss in der Zeile 36 die Konstante `OFF` durch `LO` ersetzt werden. `LO` steht für low, english niedrig und meint in diesem Fall 0 Volt. Die Leuchtdiode erlischt nun komplett nach Ablauf der Zeit.

```
36      ftduino.output_set(Ftduino::O1, Ftduino::LO);
```

Direkt nach dem Einschalten des ftDuino leuchtet die Leuchtdiode aber nach wie vor. Vielleicht findest Du selbst heraus, wie sich das ändern lässt. Tipp: Die bisher unbenutzte `setup()`-Funktion könnte helfen.

Mehr dazu gibt es im Abchnitt 5.8.

5.2 Not-Aus

Schwierigkeitsgrad: ★☆☆☆☆

Ein Not-Aus-Schalter kann Leben retten und scheint eine einfache Sache zu sein: Man drückt einen Taster und die betreffende Maschine schaltet sich sofort ab. Im Modell stellt ein XS-Motor mit Ventilator an Ausgang `M1` die Maschine dar. Ein Taster an Eingang `I1` bildet den Not-Aus-Taster.

5.2.1 Sketch `EmergencyStop`

```
1  /*
2    EmergencyStop - Not-Aus
3
4    (c) 2017 by Till Harbaum <till@harbaum.org>
5
6    Schaltet einen Ventilator aus, sobald der Not-Aus-Taster
```

Abbildung 5.2: Not-Aus

```
7     betätigt wird.
8   */
9
10  #include <FtduinoSimple.h>
11
12  // Die Setup-Funktion wird einmal bei Start des Systems ausgeführt
13  void setup() {
14    // Ventilator bei Start des Systems einschalten
15    ftduino.motor_set(Ftduino::M1, Ftduino::LEFT);
16
17    // Ausgang der internen roten LED aktivieren
18    pinMode(LED_BUILTIN, OUTPUT);
19    // und LED ausschalten
20    digitalWrite(LED_BUILTIN, LOW);
21  }
22
23  // Die Loop-Funktion wird endlos immer und immer wieder ausgeführt
24  void loop() {
25    // Teste, ob der Taster an I1 gedrückt ist
26    if(ftduino.input_get(Ftduino::I1)) {
27      // Motor bremsen
28      ftduino.motor_set(Ftduino::M1, Ftduino::BRAKE);
29      // interne rote LED einschalten
30      digitalWrite(LED_BUILTIN, HIGH);
31    }
32  }
```

Sketchbeschreibung

Der Sketch ist sehr kurz und einfach. In der `setup()`-Funktion wird in Zeile 15 bei Sketchstart der Motor gestartet. Zusätzlich wird in den Zeilen 18 bis 20 die rote interne Leuchtdiode des ftDuino für die spätere Verwendung aktiviert, aber zunächst ausgeschaltet gelassen.

In der `loop()`-Funktion wird in Zeile 26 permanent abgefragt, ob der Not-Aus-Taster geschlossen wurde. Ist das der Fall, dann wird der Motor in Zeile 28 sofort gestoppt und in Zeile 30 die rote Leuchtdiode eingeschaltet. Der Motor wird bewusst per `BRAKE` gestoppt statt `OFF`. Auf diese Weise wird der Motor kurzgeschlossen und aktiv gebremst, während er andernfalls langsam auslaufen würde, was im Notfall eine Gefahr darstellen würde.

Aufgabe 1: Kabelbruch

Not-Taster sind zwar an vielen Maschinen vorhanden. Glücklicherweise werden sie aber nur sehr selten wirklich benötigt. Das hat den Nachteil, dass kaum jemand bemerken wird, wenn mit dem Not-Aus-Taster etwas nicht stimmt. Anfälliger als der Taster selbst sind oft die Kabel und es kann im Arbeitsalltag leicht passieren, dass ein Kabel beschädigt wird. Oft sieht man das dem Kabel nicht an, wenn z.B. die Ummantelung unbeschädigt aussieht, durch zu starke Belastung aber dennoch die Kupferleiter im Inneren unterbrochen sind. Der Resultat ist ein sogenannter Kabelbruch.

Du musst kein Kabel durchreissen. Es reicht, wenn Du einen der Stecker am Kabel, das den Taster mit dem ftDuino verbindet, heraus ziehst. Der Not-Aus-Taster funktioniert dann nicht mehr und die Maschine lässt sich nicht stoppen. Eine gefährliche Situation.

Lösung 1:

Die Lösung für das Problem ist überraschend einfach. Wir haben unseren Not-Aus-Taster als Schließer angeschlossen. Das bedeutet, dass der Kontakt geschlossen wird, wenn der Taster betätigt wird. Man kann den fischertechni-Taster aber auch als Öffner verwenden. Der Kontakt ist dann im Ruhezustand geschlossen und wird bei Druck auf den Taster geöffnet.

Abbildung 5.3: Kabelbruchsicherer Not-Aus-Öffner

Mit dem aktuellen Sketch geht die Maschine bei einem Kabelbruch sofort in den Notzustand, da der Taster ja sofort als geschlossen erkannt wird. Also muss auch im Sketch die Logik herum gedreht werden. Das passiert durch folgende Änderung in Zeile 26:

```
26    if(!ftduino.input_get(Ftduino::I1)) {
```

Man muss genau hinschauen, um den Unterschied zu sehen. Hinter der öffnenden Klammer steht nun aus Ausrufezeichen, das in der Programmiersprache C für die logische Negation eines Ausdrucks steht. Die Bedingung wird nun also ausgeführt, wenn der Taster *nicht* geschlossen ist. Nach dieser Änderung sollte sich die Maschine wieder genau wie ursprünglich verhalten. Mit einer kleinen Änderung: Zieht man nun einen der Stecker am Not-Aus-Taster heraus, so stoppt die Maschine sofort. Bei einem Kabelbruch würde das ebenfalls passieren.

So eine Schaltung nennt man auf englisch "Fail-Safe": Wenn etwas kaputt geht, dann wechselt die Schaltung in einen sicheren Zustand. Der fischertechnik-3D-Drucker verwendet diese Schaltung zum Beispiel für die Endlagentaster. Ist hier ein Kabel abgerissen, dann fährt der Drucker seine Motoren nicht gewaltsam gegen die Endanschläge der Achsen. Stattdessen verweigert der Drucker die Arbeit komplett, sobald die Verbdindung zu einem Endlagentaster unterbrochen ist.

Expertenaufgabe:

Ein Kabel kann nicht nur unterbrochen werden. Es kann auch passieren, dass ein Kabel z.B. so stark gequetscht wird, dass die inneren Leiter Kontakt miteinander bekommen. Das passiert wesentlich seltener, stellt aber ebenfalls eine realistische Gefahr dar.

Vor diesem Fall würde unsere verbesserte Notschaltung nicht schützen und der Not-Aus-Taster würde in dem Fall wieder nicht funkionieren. Wir brauchen also eine Variante, bei der weder der geschlossene noch der offene Zustand der Verbindung als "gut" erkannt wird.

Lösung:

Die Lösung ist in diesem Fall etwas aufwändiger. Es müssen nun mindestens drei Zustände unterschieden werden: "normal", "unterbrochen" und "kurzgeschlossen". Reine Schalteingänge können aber nur die beiden Zustände "geschlossen" und "offen" unterscheiden.

Die Lösung ist, die analogen Fähigkeiten der Eingänge zu nutzen. Dazu kann man z.B. direkt am Taster einen 100 Ω-Widerstand in die Leitung integrieren.

Im Normalfall ist der Taster geschlossen und der am Eingang `I1` zu messende Widerstand beträgt 100 Ω. Ist die Leitung unterbrochen, dann ist der Widerstand unendlich hoch. Und ist die Leitung kurzgeschlossen, dann ist der Widerstand nahe 0 Ω. Die Maschine darf also nur dann laufen, wenn der Widerstand nahe an 100 Ω ist. Etwas Toleranz ist nötig, da der genau Wert des verwendeten Widerstands Fertigungstoleranzen unterworfen ist und auch der geschlossene Taster sowie sein Anschlusskabel über einen eigenen sehr geringen Widerstand verfügen, der den gemessenen Gesamtwiderstand beeinflusst.

Abbildung 5.4: Gegen Kabelbruch und Kurzschluss sicherer Not-Aus

Warum muss der Widerstand nahe am Taster angebracht werden? Was passiert, wenn er nahe am ftDuino eingesetzt wird und dann ein Kurzschluss im Kabel zwischen Widerstand und Taster auftritt?

5.3 Pulsweitenmodulation

Schwierigkeitsgrad: ★★☆☆☆

Wenn man eine Lampe mit variierender Helligkeit leuchten lassen möchte oder einen Motor mit regelbarer Geschwindigkeit laufen lassen will, dann benötigt man eine Möglichkeit, die Energieaufnahme der Lampe oder des Motors zu beeinflussen. Am einfachsten klappt das mit einer einstellbaren Spannungsquelle. Bei höherer Spannung steigt auch die Energieaufnahme der Lampe und sie leuchtet heller und der Motor dreht sich schneller, bei niedrigerer Spannung wird die Lampe dunkler und der Motor dreht sich langsamer. Für die Analogausgänge des ftDuino bedeutet das, dass sie eine zwischen 0 und 9 Volt kontinuierlich (analog) einstellbare Spannung ausgeben können sollen, um Lampen und Motoren von völliger Dunkelheit bzw. Stillstand bis zu maximaler Helligkeit bzw. Drehzahl betreiben zu können.

Der Erzeugung variabler Spannungen ist technisch relativ aufwändig. Es gibt allerdings einen einfachen Weg, ein verglaichbares Ergebnis zu erzielen. Statt die Spannung zu senken schaltet man die Spannung periodisch nur für sehr kurze Momente ein. Schaltet man die Spannung z.B. nur 50% der Zeit ein und 50% der Zeit aus, so wird über die Gesamtzeit gesehen nur die Hälfte der Energie übertragen. Ob man das Ergebnis als Blinken der Lampe oder als Stottern des Motors wahrnimmt oder ob die Lampe einfach mit halber Helligkeit leuchtet und der Motor mit halber Drehzahl dreht ist von der Geschwindigkeit abhängig, mit der man die Spannung ein- und auschaltet.

Abbildung 5.5: Pulsweitenmodulation

5.3.1 Sketch Pwm

```
/*
  Pwm - Pulsweitenmodulation

  (c) 2017 by Till Harbaum <till@harbaum.org>
*/

#include <FtduinoSimple.h>

uint16_t schaltzeit = 8192;   // 8192 entspricht je 1/2 Sekunde an und aus

// Die Setup-Funktion wird einmal ausgeführt, wenn Reset gedrückt oder
// das Board gestartet wird.
```

```
13 void setup() { }
14
15 // warte die angegebene Zeit. Der "zeit"-Wert 8192 soll dabei einer halben Sekunde
16 // entsprechen. Es muss also "zeit" mal 500000/8192 Mikrosekunden gewartet werden
17 void warte(uint16_t zeit) {
18   while(zeit--)
19     _delay_us(500000/8192);
20 }
21
22 // Die Loop-Funktion wird endlos immer und immer wieder ausgeführt
23 void loop() {
24   static uint8_t an_aus = false;      // der aktuelle Ausgang-an/aus-Zustand
25   static uint8_t i1=false, i2=false;  // letzter Zustand der Tasten an I1 und I2
26
27   // ist die Taste an I1 gedrückt?
28   if(ftduino.input_get(Ftduino::I1)) {
29     // und war die Taste vorher nicht gedrückt und ist die
30     // aktuelle Schaltzeit kleiner 8192?
31     if(!i1 && (schaltzeit < 8192)) {
32       // dann verdopple die Schaltzeit
33       schaltzeit *= 2;
34       // warte eine Millisekunde, falls die Taste nachprellt
35       _delay_ms(1);
36     }
37     // merke, dass die Taste an I1 zur Zeit gedrückt ist
38     i1 = true;
39   } else
40     // merke, dass die Taste an I1 zur Zeit nicht gedrückt ist
41     i1 = false;
42
43   // ist die Taste an I2 gedrückt?
44   if(ftduino.input_get(Ftduino::I2)) {
45     // und war die Taste vorher nicht gedrückt und ist die
46     // aktuelle Schaltzeit größer 1?
47     if(!i2 && (schaltzeit > 1)) {
48       // dann halbiere die Schaltzeit
49       schaltzeit /= 2;
50       // warte eine Millisekunde, falls die Taste nachprellt
51       _delay_ms(1);
52     }
53     // merke, dass die Taste an I2 zur Zeit gedrückt ist
54     i2 = true;
55   } else
56     // merke, dass die Taste an I2 zur Zeit nicht gedrückt ist
57     i2 = false;
58
59   // schalte den Ausgand O2 je nach Zustand der an_aus-Variable an oder aus
60   if(an_aus)
61     // wenn der aktuelle an_aus-Zustand wahr ist, dann schalte den Ausgang ein
62     ftduino.output_set(Ftduino::O1, Ftduino::HI);
63   else
64     // wenn der aktuelle an_aus-Zustand unwahr ist, dann schalte den Ausgang aus
65     ftduino.output_set(Ftduino::O1, Ftduino::OFF);
66
67   // warte die aktuelle Schaltzeit
68   warte(schaltzeit);
69
70   // wechsel den an_aus-Zustand
71   an_aus = !an_aus;
72 }
```

Sketchbeschreibung

Der Sketch schaltet den Ausgang `O1` in der `loop()`-Funktion in den Zeilen 60 bis 71 kontinuierlich ein und aus. Je nach Wert der Variable `an_aus` wird der Ausgang in Zeile 62 auf 9 Volt (`HI`) geschaltet oder in Zeile 65 ausgeschaltet (von der Spannungsversorgung getrennt). In Zeile 71 wird in jedem Durchlauf der `loop()`-Funktion der Zustand der Variable `an_aus` gewechselt, so dass der Ausgang im in jedem Durchlauf im Wechsel ein- und ausgeschaltet wird.

Nach jeden An/Aus-Wechsel wird in Zeile 68 etwas gewartet. Wie lange gewartet wird steht in der Variablen `schaltzeit` Sie gibt die Wartezeit in $^{1}/_{8192}$ halben Sekunden an. Dazu wird in der Funktion `wait()` in Zeile 19 so oft $^{500000}/_{8192}$

Mikrosekunden gewartet wie in der Variablen `schaltzeit` angegeben. Warum halbe Sekunden? Weil zweimal pro Zyklus gewartet wird, einmal wenn der Ausgang eingeschaltet ist und einmal wenn er ausgeschaltet ist. Wird jeweils eine halbe Sekunde gewartet, so dauert der gesamte Zylus eine Sekunde und der Ausgang wird einmal pro Sekunde für eine halbe Sekunde eingeschaltet. Der Ausgang wechselt also mit einer Frequenz von $^1/_{\text{Sek.}}$ oder einem Hertz.

Durch einen Druck auf den Taster an `I1` (Zeile 28) kann der Werte der Variablen `schaltzeit` verdoppelt (Zeile 33) und mit einem Druck auf den Taster an `I2` (Zeile 44) halbiert (Zeile 49) werden. Dabei wird der Wert von Schaltzeit auf den Bereich von 1 (Zeile 47) und 8192 (Zeile 31) begrenzt. Nun wird auch klar, warum dieser merkwürdig "krumme" Wert 8192 gewählt wurde: Da 8192 eine Zweierpotenz (2^{13}) ist lässt der Wert sich ohne Rundungsfehler bis auf 1 hinuterteilen und wieder hochmultiplizieren.

Da die Tasten nur beim Wechsel zwischen an und aus abgefragt werden muss man den Taster bei niedrigen Frequenzen einen Moment gedrückt haltem, bis sich die Blinkfrequenz verändert.

Wenn der Sketch startet leuchtet die Lampe einmal pro Sekunde für eine halbe Sekunde auf. Ein (langer) Druck auf den Taster an `I2` halbiert die Wartezeit und die Lampe blinkt zweimal pro Sekunde. Nach einem zweiten Druck auf den Taster blinkt sie viermal usw. Nach dem sechsten Druck blinkt sie 32 mal pro Sekunde, was nur noch als leichtes Flackern wahrnehmbar ist und nach dem siebten Druck gar 64 mal. Frequenzen oberhalb ca. 50 Hertz kann das menschliche Auge nicht mehr auflösen und die Lampe scheint mit halber Helligkeit zu leuchten. Die Frequenz weiter zu erhöhen hat dann keinen erkennbaren Effekt mehr.

Aufgabe 1: Die träge Lampe

Es ist in diesem Aufbau nicht nur das menschliche Auge, das träge ist. Die Lampe ist ebenfalls träge. Es dauert eine Zeit, bis sich ihr Glühfaden aufheizt und die Lampe leuchtet und es dauert auch eine Zeit, bis sich der Glühfaden wieder so weit abkühlt, dass die Lampe nicht mehr leuchtet.

Wesentlich schneller als Glühlampen sind Leuchtdioden. In ihnen muss sich nichts aufheizen oder abkühlen, sondern das Licht entsteht direkt durch optoelektrische Effekte im Halbleitermaterial der Leuchtdiode. Schließt man statt der Lampe eine Leuchtdiode an (rot markierter Anschluss an Ausgang `O1`), dann sieht das Verhalten zunächst ähnlich aus und wieder scheint ab einer Frequenz von 64 Hertz die Leuchtdiode gleichmäßig mit halber Helligkeit zu leuchten. Viele Menschen nehmen 64 Hz allerdings noch als leichtes Flimmern wahr und erst ab 100Hz redet man von einer wirklich flimmerfreien Darstellung.

Man kann das Flimmern der Leuchtdiode aber auch bei diesen Frequenzen noch beobachten, wenn sich die Leuchtdiode bewegt. Nutzt man ein etwas längeres Kabel, so dass die Leuchtdiode sich frei bewegen lässt und bewegt sie dann in einer etwas abgedunkelten Umgebung schnell hin- und her, so wird der Eindruck einer Reihe von unterbrochenen Leuchtstreifen entstehen.

Abbildung 5.6: Muster bei schneller Bewegung der flackernden Leuchtdiode

Je höher die PWM-Frequenz ist, desto kürzer sind die sichtbaren Leuchtstreifen.

Dieses Experiment kann man auch mit der Lampe wiederholen. Durch die Trägheit der Lampe sieht man nur einen durchgehenden Leuchtstreifen. Allerdings sollte man nicht allzu wild vorgehen, da der empfindliche Glühfaden einer leuchtenden Lampe bei Erschütterung leicht kaputt geht. Leuchtdioden sind auch in dieser Beziehung robust und lassen sich selbst von starken Erschütterungen nicht beeindrucken.

Aufgabe 2: Töne aus dem Motor

Ein Motor ist ebenfalls träge und nicht in der Lage, beliebig schnellen An-/Aus-Signalen zu folgen. Schon bei recht niedrigen PWM-Frequenzen dreht sich der Motor kontinuierlich mit halber Drehzahl. Das dabei vornehmlich zu vernehmende Geräusch ist das Laufgeräusch des Motors.

Wenn man den Motor aber mechanisch blockiert, indem man ihn z.B. mit der Hand festhält, dann wirddas Laufgeräsch unterdrückt und ein anderer Effekt wird hörbar: Die Spulen im Motor wirken wie ein Lautsprecher und man kann die PWM-Frequenz bei blockiertem Motor als Ton hören. Eine Veränderung der PWM-Frequenz hat dabei einen deutlich hörbaren Unterschied der Tonhöhe zur Folge.

Je höher die PWM-Frequenz, desto höher der am blockierten Motor hörbare Ton.

Aufgabe 3: Nachteil hoher PWM-Frequenzen

Im Fall der Lampe scheint eine höhere PWM-Frequenz ein reiner Vorteil zu sein, da das Flimmern mit höherer Frequenz abnimmt. Am Motor kann aber ein negativer Effekt beobachtet werden.

Läuft der Motor frei, so hängt die gehörte Tonhöhe des Motor-Laufgeräuschs mit seiner Drehgeschwindigkeit zusammen, während das PWM-Geräuch der vorigen Aufgabe in den Hintergrund tritt. Je schneller der Motor dreht, desto höher die Frequenz des Laufgeräuschs und umgekehrt.

Erhöht man nun die PWM-Frequenz, dann sinkt die Frequenz der Töne, die der Motor abgibt leicht. Er wird offensichtlich mit steigener PWM-Frequenz langsamer. Dieser Effekt ist damit zu erklären, dass der Motor eine sogenannte induktive Last darstellt. Er besteht im Wesentlichen aus Spulen, sogenannten Induktoren. Der Widerstand einer induktiven Last ist abhängig von der Frequenz einer angelegten Wechselspannung. Und nichts anderes ist das durch die PWM erzeugte An-/Aus-Signal. Je höher die Frequenz, desto höher der Widerstand der Spule und es fließt weniger Strom durch die Spule.

Es ist technisch möglich, die Ausgangsspannung zu glätten und diesen Effekt zu mildern. Diese Auslegung so einer Glättung ist allerdings von der verwendeten PWM-Frequenz und der Stromaufnahme des Motors abhängig. Außerdem beeinflusst sie das generelle Schaltverhalten des Ausgangs. Der Einsatz einer entsprechenden Glättung im ftDuino kommt daher nicht in Frage, da die universelle Verwendbarkeit der Ausgänge dadürch eingeschränkt würde.

Ziel bei der Auswahl der PWM-Frequenz ist also eine Frequenz, die hoch genug ist, um Lampenflackern oder Motorstottern zu verhindern, die aber dennoch möglichst gering ist, um induktive Widerstände in den Wicklungen der Motoren zu minimieren. Eine PWM-Frequenz von 100-200Hz erfüllt diese Bedingungen.

Motordrehzahl in Abhängigkeit des PWM-Verhältnisses

Die Motordrehzahl lässt sich durch das Verhältnis der An- und Ausphasen während der Pulsweitenmodulation beeinflussen. In den bisherigen Versuchen waren die An- und Ausphase jeweils gleich lang. Verändert man das Verhältnis der beiden Phasen, dann lässt sich die Helligkeit einer Lampe oder die Drehzahl eines Motors steuern. Die PWM-Frequenz kann dabei konstant bleiben.

Abbildung 5.7: Ausgewählte PWM-Verhältnisse von 0 bis 100%

Je länger die eingeschaltete Phase gegenüber der ausgeschalteten, desto heller leuchtet die Lampe und desto schneller dreht der Motor. Der genaue Zusammenhang zwischen Lampenhelligkeit und PWM-Verhältnis ist mangels entsprechender Messmöglichkeit nicht einfach festzustellen. Die sogenannten Encoder-Motoren haben aber eine eingebaute Möglichkeit zur Geschwindigkeitsmessung. Im Fall der TXT-Encodermotoren erzeugen diese Encoder 63 Signalimpulse pro Umdrehung der

Achse. Man kann also durch Auswertung der Encodersignale an den Zählereingängen des ftDuino die Drehzahl des Motors feststellen.

Abbildung 5.8: Anschluss des TXT-Encoder-Motors zur PWM-abhängigen Drehzahlmessung

Das Beispiel `Datei ▷ Beispiele ▷ Ftduino ▷ PwmSpeed` regelt das An-/Ausverhältnis der PWM langsam von 0 auf 100% hoch und misst dabei kontinuerlich für jeweils eine Sekunde die am Eingang `C1` anliegenden Impulse. Diese werden in Umdrehungen pro Minute umgerechnet und ausgegeben. Dabei kommt die vollständige Bibliothek `Ftduino` zum Einsatz, die die eigentliche Erzeugung der PWM-Signale bereits mitbringt. Die Erzeugung des PWM-Signals passiert vollständig im Hintergrund, so dass der Sketch selbst lediglich den Motor startet und dann eine Sekunde wartet.

Speist man die so gewonnenen Daten in den sogenannten "seriellen Plotter", der sich im Menü der Arduino-IDE unter `Werkzeuge ▷ Serieller Plotter` befindet, so kann man die Messergebnisse anschaulich visualisieren.

Abbildung 5.9: Leerlaufdrehzahl des TXT-Encoder-Motors in Abhängigkeit des PWM-Verhältnisses

Auf der horizontalen Achse ist das PWM-Verhältnis aufgetragen, beginnend mit "dauerhaft aus" ganz links bis "dauerhaft an" ganz rechts. Auf der vertikalen Achse ist die gemessene Drehzahl in Umdrehungen pro Minute dargestellt. Man sieht, dass der Zusammenhang im Leerlauf nicht linear ist. Bereits bei nur ca 25% der Zeit eingeschaltetem Signal wird 90% der maximalen Motordrehzahl erreicht.

Man kann z.B. indem man den Motor eine konstante Last anheben lässt nachprüfen, wie sich diese Kurve und Last verändert.

5.4 Schrittmotoransteuerung

Schwierigkeitsgrad: ★★★★☆

Gängige Elektromotoren, wie sie üblicherweise in Spielzeug eingesetzt werden sind sogenannte Asynchronmotoren. Diese in der Regel mit Gleichspannung versorgten Motoren zeichnen sich dadurch aus, dass sie beim Anlegen einer Spannung sofort

anfangen, sich zu drehen. Die Drehzahl richtet sich dabei nur indirekt nach außeren Einflüssen und der Motor dreht letztlich so schnell es ihm möglich ist. Für viele Anwendungen ist diese Art Motor sehr gut geeignet. Modellautos lassen sich ohne weitere Steuerung motorisieren und fahren auf mit diesen Motoren so schnell wie gerade möglich. Auch fischertechnik setzt diese Motoren in den meisten Fällen ein und der ftDuino erlaubt es, sie direkt an jeweils einen Motorausgang `M1` bis `M4` anzuschließen.

Aber es gibt Anwendungen, in denen diese Motorfamilie sehr große Schwächen zeigt. So ist es sehr schwierig, mit einfachen Asynchronmotoren eine exakte Drehzahl zu erreichen bzw. eine exakte Position anzufahren. Die Encoder-Motoren von fischertechnik versuchen dies durch zusätzliche Hardware zu ermöglichen. Aber auch diesem Vorgehen sind Grenzen gesetzt, speziell bei den fischertechnik-Encoder-Motoren, die nicht in der Lage sind, die Drehrichtung zu erfassen.

Abbildung 5.10: 17HS13-Schrittmotor am ftDuino

Für Aufgaben, bei denen es auf exaktes reproduzierbares Verhalten des Motors ankommt, gibt synchron arbeitende Motoren wie die sogenannten Schrittmotoren. Gängige Anwendungen dafür sind heutzutage Scanner und 3D-Drucker, aber auch die früher verwendeten Diskettenlaufwerke und frühe Festplatten verwendeten Schrittmotoren. Erkennbar ist die Nutzung von Schrittmotoren auch am charakteristischen Betriebsgeräusch, das findige Tüftler sogar nutzen, um mit solchen Motoren Musik zu erzeugen[1].

Abbildung 5.11: Anschluss des Schrittmotors an den ftDuino

Gängige Schrittmotoren bestehen aus einem drehbar gelagerten Anker aus Permanentmagneten, der von Elektromagneten umgeben ist. Der Permanentmagnet richtet sich entsprechend den ihm umgebenden Magnetfeldern aus. Im Spannungslosen

[1] Im Internet leicht zu finden unter dem Stichwort "Floppymusik"

Zustand lässt sich die Achse des Motors verlgeichsweise leicht drehen. Der dabei spürbare Widerstand resultiert daraus, dass der Permanentmagnet von den Eisenkernen der Elektromagnete auch im spannungslosen Zustand angezogen wird.

Abbildung 5.12: Vereinfachtes Schema eines Schrittmotors

Der in der Abbildung dargestellte vereinfachte Schrittmotor verfügt über einen Permanentmagneten und zwei Elekomagnete. Reale Schrittmotoren haben in der Regel mehr als zwei Spulen und der Anker bidet auch mehr als nur zwei magnetische Pole ab. Auf das Funktionsprinzip hat diese Vereinfachung keine Auswirkungen.

Übliche sogenannte bipolare Schrittmotoren verfügen über vier Anschlüsse, jeweils zwei für jeden der beiden Elektromagneten. Durch Anlegen einer Spannung werden die Elekromagneten magnetisiert. In der Folge richtet sich der Anker entsprechend aus. Die Polarität der angelegten Spannung bestimmt die Richtung des Magnetfelds der Elektromagneten.

5.4.1 Vollschrittsteuerung

Sind immer beide Spulen unter Spannung, so gibt es vier verschiedene Ausrichtungen der beiden Elektromagnetfelder und der Anker nimmt jeweils vier unterschiedliche Positionen ein. Wird eine entsprechende sich wiederholende Signalefolge an die Elektromagneten angelegt, so folgt der Anker den Signalen und dreht sich. Er folgt dabei exakt den sich welchselnden Magnetfeldern und dreht sich sychron zum angelegt Signalmuster. Geschwidigkeit und Position des Motors sind auf diese Weise exakt vorhersagbar. Sind immer alle Spulen unter Spannung und durchläuft der Zyklus daher genau vier Zustände, so spricht man von einer Vollschrittsteuerung.

Abbildung 5.13: Vollschrittansteuerung eines Schrittmotors

Ein Sketch, der kontinuiertlich das entsprechende Signalmuster erzeugt sieht folgendermaßen aus:

```
while(1) {
  ftduino.motor_set(Ftduino::M1, Ftduino::LEFT);
  delay(5);
  ftduino.motor_set(Ftduino::M2, Ftduino::LEFT);
  delay(5);
  ftduino.motor_set(Ftduino::M1, Ftduino::RIGHT);
  delay(5);
  ftduino.motor_set(Ftduino::M2, Ftduino::RIGHT);
```

```
    delay(5);
}
```

Ein vollständiges Beispiel findet sich in der Arduino-IDE unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ StepperMotor`.

Während sich der dargestellte vereinfachte Schrittmotor pro Signaländerung um 90° dreht und daher nach vier Schritten eine volle Umdrehung absolviert hat haben reale Schrittmotoren eine höhere Auflösung. Gängig ist ein Schrittwinkel von 1,8°. Erst nach 200 Schritten hat sich so ein Motor einmal komplett gedreht. Da im abgebildeten Listing nach jedem Schritt 5ms gewartet wird werden genau 200 Schritte pro Sekunde erzeugt. Ein gängiger 1,8°-Motor würde sich genau einmal pro Sekunden drehen.

Für die Experimente am ftDuino muss ein Motor gewählt werden, der mit den 9V-Ausgängen des ftDuino kompatibel ist. Der hier verwendete 17HS13 ist für eine Betriebsspannung von 12V ausgelegt, arbeitet aber auch mit den fischertechnik-üblichen 9 Volt zuverlässig. Die Motoren des fischertechnik-Plotters 30571[2] von 1985 waren für 6 Volt ausgelegt. Sollen diese Motoren am ftDuino betrieben werden, so ist dieser mit 6 statt den üblichen 9 Volt zu versorgen.

Die Abbildungen der Abläufe beinhalten in der unteren Hälfte jeweils die Signalverläufe an den vier Anschlüssen des Motors. Die Signalverläufe sind farblich ensprechend der sich ergebenden Megnetfeldrichtung hinterlegt. Man sieht, wie die beiden Anschlüsse eines Magneten immer genau gegenteilig angesteuert werden und sich das Magnetfeld beim Wechsel der Signale ändert. Die abgebildete Farbe entspricht jeweils der Polarität der dem Anker zugewandten Seite des Elektromagneten.

5.4.2 Halbschrittsteuerung

Eine höhere Winkel-Auflösung erreicht man, wenn man den Schrittmotor im sogenannten Halbschrittbetrieb ansteuert. Der Signalzyklus besteht in dem Fall nicht mehr aus vier sondern aus acht Schritten. In jedem zweiten Schritt wird einer der beiden Elektromagneten abgeschaltet, so dass sich der Anker nur nach dem verbliebenen Magneten ausrichtet. Die sich dadurch ergebenden vier Zwischenzustände sind von den resultierenden Winkel genau zwischen den vier Zuständen der Vollschrittansteuerung angeordnet. Der Motor kann also die doppelt so viele Winkel ansteuern und entsprechend genauer positioniert werden.

Abbildung 5.14: Halbschrittansteuerung eines Schrittmotors

In der Abbildung ist sichtbar, dass sich die beiden Signale zur Ansteuerung eines Elektromagneten nicht mehr gleichzeitig ändern, sondern dass es einen Versatz gibt, währenddessen beide Signale auf gleichem Pegel liegen. Der Magnet steht zu dieser Zeit nicht unter Spannung und hat kein Magnetfeld. Die Signalverläufe sind zu dieser Zeit daher nicht farbig hinterlegt.

Der Nachteil der Halbschrittansteuerung liegt darin, dass in den Zeiten, in denen nur ein Elektromagnet aktiv ist die Kraft des Motors reduziert ist.

Timer-Interrupt betriebener Schrittmotor

Der im vorhergehenden Abschmitt verwendete Sketch zur Schrittmotoranszteuerung hat vor allem einen großen Vorteil: Er ist anschaulich. Das Problem ist aber, dass die Ansteuerung Motors permanente Signalwechsel benötigt und daher die Motorfunktionen im Sketch permanent aktiv sein müssen, damit der Motor sich dreht. Der Sketch verbringt praktisch die ganze Zeit damit in den diversen `delay()`-Funktionsaufrufen, auf den nächsten Signalwechsel zu warten. Der große

[2] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=30571

Nachteil: Während der Sketch einen Motor bedient kann er kaum etwas anderes tun. Einen zweiten Motor gleichzeit mit ggf. sogar unterschiedlicher Drehzahl laufen zu lassen ist mit diesem einfachen Sketch kaum zu realisieren.

Das gleiche Problem stellte sich während der Entwicklung des ftDuino bereits mit den übrigen Komponenten des ftDuino auch die Auswertung der Analogeingänge, die PQM-Drehzahlregelung der Motorausgänge und die Auswertung der Zähler beanstruchen permant eine aktive Mitarbeit des Mikrocontrollers. Trotzdem muss der Anwender dafür in seinem Sketch keine Funktion vorsehen. All diese Dinge passieren weitgehend unbemerkt im Hintergrund. Solch eine Hintergrundfunktion wäre auch für den Betrieb des Schrittmotors wünschenswert.

Mikrocontroller wie der ATmega32u4 des ftDuino bestehen aus einem Mikroprozessor (der eigentlichen Recheneinheit) und diversen zusätzlichen Hardwarekomponenten wie z.B. USB-Schnittstellenfunktionen. Unter anderem verfügt der ATmega32u4 über einige sogenannte Timer. Einen Timer kann man sich wie eine unabhängig vom eigentlichen Prozessor arbeitende Uhr vorstellen. Man kann per Software festlegen, wie schnell die Uhr laufen soll und ob zu bestimmten Zeitpunkten ggf. bestimmte Dinge passieren sollen, aber das eigentliche Fortschreiten der Uhrzeit geschieht automatisch und ohne weiteres Zutun eines auf dem Prozessor ausgeführten Sketches. Eines der Dinge, die von so einem Timer regelmäßig ausgelöst werden können ist eine sogenannte Unterbrechungsanforderung (englisch Interrupt). Sie veranlasst den Prozessor, zu unterbrechen, was auch immer er gerade tut und sich fpr kurze Zeit einer anderen Aufgabe zu widmen.

Diese Art von Unterbrechung ist ideal, um z.B. einen Schrittmotor zu steuern. Soll der Schrittmotor z.B. 200 Schritte pro Sekunde bewegt werden, so kann ein Timer so programmiert werden, dass der Prozessor alle 5 Millesekunden unterbrochen wird. In dieser Unterbrechung muss der Prozessor dann das Magnetfeld des Motors einen Schritt weiter drehen und sich dann wieder seiner normalen Aufgabe widmen.

Das folgende Code-Segment programmiert den Timer 1 des ATmega32u4 so, dass die sogenannte Interrupt-Service-Routine exakt alle 5 Millisekunden ausgeführt wird. Der Motor ließe sich so durch passenden Programmcode innerhalb dieser Routine völlig unabhängig vom Hauptprogramm des Sketches betreiben.

```
1  #include <FtduinoSimple.h>
2
3  // die sogegannte Interrupt-Service-Routine (ISR) wird
4  // nicht vom Sketch selbst zur Ausführung gebracht, sondern
5  // die Hardware des ATmega32u$ läst die Ausführung auf Basis
6  // eines Timer-Ereignisses aus
7  ISR(TIMER1_COMPA_vect) {
8    // Diese Funktion wird alle 5ms ausgeführt.
9    // Das Weiterdrehen des Schrittmotor-Magnetfeldes
10   // könnte z.B. hier geschehen.
11   // ...
12 }
13
14 void setup() {
15   // Konfiguration des Timer 1 des ATmega32u4, die genaue
16   // Beschreibung der Register findet sich in Kapitel 14
17   // des Datenblatt:
18   // http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32
       U4_Datasheet.pdf
19
20   // Timer 1 soll im sogenannten CTC-Modus schalten mit OCR1A
21   // als obere Grenze. Der Timer läuft mit 1/256 CPU-Takt. Dieser
22   // wiederum beträgt 16 Mhz, der Timer läuft also mit 62,5 kHz.
23   // Um den Motor 200 Schritte pro Sekunde zu drehen muss der
24   // Motor immer dann einen Schritt machen, wenn der Timer 312
25   // (62500/200) seiner Zählschritte durchlaufen hat.
26   TCCR1A = 0;
27   TCCR1B = (1<<WGM12) | (1<<CS12); // Starte Timer 1 mit 1/256 F_CPU = 62.5kHz
28   TCCR1C = 0;
29
30   // Ereignis auslösen wenn 62400/200 Zählerschritte erreicht sind
31   TCNT1 = 0;
32   OCR1A = 62500/200;
33
34   // Eregniserzeugung bei Erreichen der Zielschritte auslösen
35   TIMSK1 = (1<<OCIE1A);
36 }
37
38 void loop() {
39   // die Hauptroutine kann benliebig genutzt werden und
40   // der Timer 1-Interrupt wird unabhängig regelmäßig
41   // ausgeführt
```

```
42  }
```

Das vollständige Beispiel findet sich unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ StepperMotorSrv`. Es entspricht weitgehend der vorgeschlagenen Nutzung von Timer 1. Ein zweiter Motor ließe sich z.B. ebenfalls im Hintergrund und unabhängig vom bereits vorhandenen Motor und auch unabhängig vom Hauptorpgramm des Sketches durch Timer 3 steuern. Mit zwei Schrittmotoren ist so recht elegant z.B. ein sogegannter Plotter zu realisieren.

5.5 Servomotoransteuerung

Schwierigkeitsgrad: ★★★☆☆

Neben den normalen Gleichspannungsmotoren und den Schrittmotoren aus Abschnitt 5.4 gibt es eine dritte vor allem im Modellbau verbreitete Art von Motoren, die sogenannten Servos. fischertechnik vertreibt einen Servo unter der Artikelnummer 132292[3].

Technisch bestehen Servos aus einfachen Gleichstrommotoren und einer einfachen Elektronik. Eine Messmechanik meldet dieser Elektronik ständig den aktuellen Stellwert (Winkel) des Servos. Die Elektronik vergleicht diesen mit einem externen Sollwert und regelt den Motor bei Bedarf nach. Das Servo folgt also mit seinem Drehwinkel einem externen Sollwert.

Servos haben daher ein drei-adriges Anschlusskabel. Zwei Adern werden zur Spannungsversorgung (rot = 6 Volt, braun = Masse) verwendet, die dritte Ader (orange) überträgt den Sollwert. Es lassen sich handelsübliche Servos verwenden, aber auch der fischertechnik-Servo 132292[4] aus dem 540585[5]-PLUS Bluetooth Control Set. Auch der fischertechnik RC-Servo 30275[6] von 1983 müsste auf diese Weise verwendbar sein. Das wurde jedoch bisher nicht überprüft.

(a) Versorgung durch den ftDuino

(b) Versorgung mit 7806-Regler

Abbildung 5.15: Anschluss des Servos an den ftDuino

Da Servos für eine Betriebsspannung von 6 Volt ausgelegt sind und oft auch bei 5 Volt noch funktionieren ist eine Versorgung aus den internen 5 Volt des ftDuino über den I²C-Anschluss möglich wie in Abbildung 5.15(a) dargestellt. Bei dieser Versorgung ist Vorsicht geboten, da die Stromaufnahme des Servos 100mA nicht überschreiten darf, um die interne Stromversorgung des ftDuino nicht zu überlasten. Die meisten Servos überschreiten diesen Wert deutlich und sollten daher nicht direkt aus dem ftDuino versorgt werden.

[3] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=132292
[4] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=132292
[5] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=540585
[6] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=30275

5.5.1 Externe 6-Volt-Versorgung

Wesentlich robuster und kaum aufwändiger ist die Versorgung über einen externen Spannungsregler z.B. vom Typ 7806, der unter dieser Bezeichnung leicht im Online-Handel zu finden ist. Dieser sogenannte Längsregler kann direkt an einen der 9-Volt-Ausgänge des ftDuino angeschlossen werden und stellt an seinem Ausgang eine auf 6 Volt reduzierte Spannung bereit.

Benötigte Teile:

1x Spannungsregler 7806
1x Servo-Verlängerungskabel JR-Stecker
2x roter ft-Stecker
1x grüner ft-Stecker
1x rote ft-Buchse

Der 7806 kann mit fischertechnik-üblichen Steckern versehen und dann direkt in den ftDuino eingesteckt werden. Der Steuersignal-Anschluss muss mit dem SDA-Anschluss des I^{2}C-Steckers verbunden werden. Die zusätzliche Nutzung des SCL-Anschlusses ermöglicht den Anschluss eines zweiten Servos. Trennt man das rote Kabel direkt am Stecker des Servos auf und versieht es mit einem fischertechnik-Stecker wie in Abbildung 5.15(b) dargestellt, so lässt sich der Stecker des Servos mit den zwei verbliebenden Adern direkt auf den I^{2}C-Anschluss stecken. Will man das Anschlusskabel des Servos nicht zerschneiden, dann kann man auch ein handelsübliche JR-Servo-Verlängerungskabel zerschneiden wie in Abbildung 5.16(b) dargestellt. Der aufgetrennte mittlere rote Anschluss wird dann mit dem fischertechnik-Stecker an den ebenfalls mit fischertechnik-Hülsen versehenen 7806 gesteckt.

(a) Servo an Regler und Adapterkabel

(b) Spannungsregler und Adapterkabel

Abbildung 5.16: Servo-Motor am ftDuino

Das Steuersignal eines Servos entspricht nicht dem I^{2}C-Standard. Stattdessen verwenden Servos ein einfaches Pulsweitensignal, das alle 20 Millisekunden widerholt wird. Der Puls selbst ist zwischen einer und zwei Millisekunden lang und bestimmt den Winkel, den das Servo einnehmen soll. Eine Millisekunde steht dabei für den minimalen Wert und zwei Millisekunden für den maximalen. Soll der Servomotor in Mittelstellung fahren, so ist dementsprechend ein Puls von 1,5 Millisekunden Länge nötig.

Das Servo verfügt über keine weitere Intelligenz und es werden die vorgegebenen Steuersignale nicht überprüft. Pulslängen kleiner einer Millisekunde oder größer zwei Millisekunden versucht das Servo ebenfalls in entsprechende Winkel umzusetzen. Dabei ist zu beachten, dass der Bewegung des Servos mechanische Grenzen gesetzt sind und das Servo Schaden nehmen kann, wenn es Positionen außerhalb seines normalen Arbeitsbereichs anzufahren versucht. Es ist daher nicht ratsam, den Bereich von ein bis zwei Millisekunden zu verlassen.

Um das nötige Puls-Signal auf den eigentlich für I^{2}C-Signale vorgesehenen Pins zu erzeugen muss auf Software zurückgegriffen werden. Ein Programmfragment, das den Servo in die Mittelposition bewegt könnte z.B. folgendermaßen aussehen:

```
1  void setup() {
```

Abbildung 5.17: Servo-Winkel in Anhängigkeit vom Stellsignal

```
2     // Port D.1 (SDA-Anschluss) auf Ausgang schalten
3     bitSet(DDRD, 1);
4   }
5
6   void loop() {
7     // Port D.1 auf High-Pegel (5V) legen
8     bitSet(PORTD, 1);
9
10    // 1500us (1.5ms) warten
11    _delay_us(1500);
12
13    // Port D.1 auf Low-Pegel (GND) legen
14    bitClear(PORTD, 1);
15
16    // 18500us (18.5ms) warten
17    _delay_us(20000-1500);
18  }
```

Hier wird der SDA-Anschluss am I²C zunächst in der `setup()`-Funktion zu einem unabhängig verwendbaren Ausgang konfiguriert. In der Folge kann in der `loop()`-Funktion der Ausgang auf Hi (5 Volt) oder Masse (GND) geschaltet werden, indem das entsprechende Bit im Register `PORTD` gesetzt oder gelöscht wird. Nach dem Einschalten des Ausgangs wird 1500 Mikrosekunden (1,5 Millisekunden) gewartet, nach dem Ausschalten 18,5 Millisekunden, so dass die Zykluszeit von insgesamt 20 Millisekunden erreicht wird.

Wie schon beim Schrittmotor ergibt sich das Problem, dass der Prozessor des ftDuino bei dieser einfachen Art der Programmierung permanent mit der Signalerzeugung ausgelastet wird und nebenbei keine anderen Aufgaben erledigen kann.

Die Lösung besteht wie beim Schrittmotor darin, die Signalerzeugung im Hintergrund durch einen Hardwaretimer zu veranlassen. Das Beispiel `Datei ▷ Beispiele ▷ FtduinoSimple ▷ ServoDemo` bringt eine einfache Klasse zur Servoansteuerung mit. Das eigentliche Hauptprogramm sieht dann folgendermaßen aus:

```
1   //
2   // Servo.ino
3   //
4
5   #include "Servo.h"
6
7   void setup() {
8     servo.begin();
9   }
10
11  void loop() {
12    static uint8_t value = Servo::VALUE_MAX/2;
13
14    if(value < Servo::VALUE_MAX) value++;
15    else                         value = 0;
16    servo.set(value);
17
18    delay(10);
19  }
```

Die Ansteuerung des Servos beschränkt sich auf den Aufruf der `servo.begin()`-Funktion, die die nötigen Timer im Hintergrund einrichtet. Der Winkel des Servos kann dann mit der `servo.set()`-Funktion von 0 (minimaler Winkel) bis `Servo::VALUE_MAX` (maximaler Winkel) eingestellt werden. Die Mittelposition wird z.B. durch `servo.set(Servo::VALUE_MAX/2)` angefahren.

5.6 Die Eingänge des ftDuino

Schwierigkeitsgrad: ★★☆☆☆

Wer sich schon einmal mit einem Arduino beschäftigt hat weiss, dass man dort relativ frei bestimmen kann, welche Anschlüsse man als Ein- oder Ausgänge verwenden möchte, da sämtliche Pins am Microcontroller in der Richtung umgeschaltet werden können. Beim ftDuino konnte diese Fähigkeit nicht erhaltem werden, da an den Eingägen zusätzliche Schutzschaltungen gegen Überspannung und Kurzschlüsse eingesetzt werden und die Signale Ausgänge verstärkt werden, um fischertechnik-Lampen und -Motoren betreiben zu können.

Abbildung 5.18: Interne Beschaltung der Eingänge I1 bis I8 des ftDuino

Jeder der acht Eingänge I1 bis I8 des ftDuino führt auf einen eigenen Analogeingang des ATmega32u4-Mikrocontrollers und kann von diesem unabhängig ausgewertet werden. Dazu kann der Mikrocontroller die Spannung des entsprechenden Eingangs messen.

5.6.1 Spannungsmessung

Bevor das Eingangssignal den Mikrocontroller erreicht wird es über einen Spannungsteiler aus zwei 47-Kiloohm-Widerständen geführt. Diese Widerstände erfüllen zwei Aufgaben. Erstens halbieren sie die Spannung eines angelegten Signals bevor es den Mikrocontroller erreicht. Da der Mikrocontroller ftDuino-intern mit 5 Volt betrieben wird und auch nur Signale im Bereich von 0 bis 5 Volt verarbeiten kann. Die Spannungshalbierung erweitert den messbaren Eingangsspannungsbereich auf 0 bis 10 Volt, wodurch sich die fischertechnik-üblichen Spannungen bis maximal 9 Volt verarbeiten lassen. Zum zweiten schützen diese Widerstände im Zusammenspiel mit den Mikrocontroller-internen Schutzdioden den Mikrocontroller vor Spannungen, die außerhalb des für ihn verträglichen 0 bis 5 Volt-Spannungsbereichs liegen. Spannungen bis zu 47 Volt an einem Eingang beschädigen den Mikrocontroller daher nicht.

5.6.2 Widerstandsmessung

Die 1 kΩ- und 2.2 kΩ-Widerstände sowie der Schalter haben keine Bedeutung, solange der Schalter offen ist. Acht dieser Schalter, je einer für jeden Eingang, befinden sich im ftDuino im Baustein mit der Bezeichnung CD4051 (IC1) wie im Schaltplan in Anhang A ersichtlich. Der Mikrocontroller kann genau einen der acht Schalter zu jeder Zeit schließen und auf diese Weise einen Widerstand von insgesamt 3.2 kΩ (1 kΩ plus 2.2 kΩ) vom jeweiligen Eingang gegen 5 Volt aktivieren.

Der Schalter wird geschlossen und die Widerstände werden aktiviert, wenn eine Widerstandsmessung erfolgen soll. Der 3.2 kΩ-Widerstand bildet dann mit einem zwischen Eingang und Masse angeschlossenen externen Widerstand einen Spannungsteiler. Aus der gemessenen Spannung kann dann der unbekannte externe Widerstand bestimmt werden.

Die Widerstandsmessung wird von der `Ftduino`-Bibliothek im Hintergrund ausgeführt. Dabei findet auch eine automatrische Umschaltung der Widerstandsmessung auf all jene Eingänge statt, die im Sketch zur Zeit zur Widerstandsmessung genutzt werden. Der Programmierer muss sich also um keine Details kümmern und kann jederzeit Widerstandswerte mit Hilfe der Funktion `ftduino.input_get()` (siehe 8.2.2) abfragen.

5.6.3 Ein Eingang als Ausgang

Die Tatsache, dass im Falle einer Widerstandsmessung ein Widerstand gegen 5 Volt geschaltet wird bedeutet, dass über den zu messenden extern angeschlossenen Widerstand ein Stromkreis geschlossen wird. Der Strom durch diesen Stromkreis is relativ gering. Wenn der Eingang direkt mit Masse verbunden ist beträgt der Gesamtwiderstand 3.2 kΩ und es fließt ein Strom von $I = 5V/3.2\,\mathrm{k}\Omega = 1,5625mA$.

Abbildung 5.19: Anschluss einer LED an Eingang `I1` des ftDuino

Dieser Strom reicht zwar nicht, um eine Lampe oder gar einen Motor zu betreiben. Aber eine Leuchtdiode kann man damit schwach zum Leuchten bringen. Schließt man eine Leuchtdiode direkt zwischen einem Eingang und Masse an und schaltet den Eingang auf Widerstandsmessung, so wird die LED ganz leicht leuchten.

Der tatsächliche Strom wird noch deutlich unter den vorausgesagten 1,5mA liegen, da zum einen direkt an der Leuchtdiode die sogenannte Vorwärtsspannung von ca. 0,7V abfällt und über den Widerständen daher nur eine Spannung von etwas über vier Volt anliegt.

Zum anderen fragt die `Ftduino`-Bibliothek im Hintergrund alle acht Eingänge ab und aktiviert jeden Eingang dabei nur $^1/_8$ der Zeit. Es fließt im Mittel daher auch nur $^1/_8$ des Stroms.

Die `FtduinoSimple`-Bibliothek schaltet ebenfalls die Widerstände ein und zwar für den jeweils zuletzt aktivierten Eingang. Dieser Widerstand ist dann dauerhaft aktiviert, bis ein anderer Eingang angefragt wurd. Das folgenden Code-Fragment lässt eine LED an Eingang `I1` im Sekundentakt blinken.

```
1  #include <FtduinoSimple.h>
2
3  void loop() {
4    // lies Wert von Eingang I1, aktiviert Widerstand auf I1
5    ftduino.input_get(Ftduino::I1);
6    delay(1000);
7    // lies Wert von Eingang I2, deaktiviert Widerstand auf I1
8    // (und aktiviert ihn auf I2)
9    ftduino.input_get(Ftduino::I2);
10   delay(1000);
11 }
```

5.7 Temperaturmessung

Schwierigkeitsgrad: ★☆☆☆☆

Fischertechnik vertreibt unter der Artikelnummer 36437[7] einen sogenannten NTC. Dieses unscheinbare Bauteil liegt einigen Robotics-Baukästen bei.

Ein NTC ist ein elektrischer Widerstand, der seinen Wert abhängig von der Umgebungstemperatur ändert. Er eignet sich daher zur Temperaturmessung. NTC steht dabei für "Negativer-Temperatur-Coeffizient", was bedeutet, dass der ohmsche Widerstand mit steigender Temperatur sinkt. NTCs werden im Deutschen auch als Heißleiter bezeichnet, da ihre Leitfähigkeit mit der Temperatur steigt.

Der Nennwiderstand R_N eines NTCs wird in der Regel bei einer Temperatur von 25 °C (298,15 K) angegeben. Der für den fischertechnik-Sensor angegebene Wert ist 1.5 kΩ. Der ohmsche Widerstand beträgt bei 25 °C also 1.5 kΩ.

Abbildung 5.20: Anschluss des Temperaturensors an Eingang I1

5.7.1 Sketch Temperature

Der folgende Sketch findet sich bei installierter ftDuino-Unterstützung im Menü der Arduino-IDE unter `Datei` ▷ `Beispiele` ▷ `Ftduino` ▷ `Temperature`.

```
1  //
2  // Temperaure.ino
3  //
4  // Abfrage eines fischertechnik Temperatur-Widerstands an Eingang I1
5  //
6  // (c) 2018 by Till Harbaum <till@harbaum.org>
7  //
8
9  #include <Ftduino.h>
10 #include <math.h>          // Für Fliesspunkt-Arithmetik
11
12 #define K2C 273.15         // Offset Kelvin nach Grad Celsius
13 #define B 3900.0           // sog. B-Wert des Sensors
14 #define R_N 1500.0         // Widerstand bei 25 Grad Celsius Referenztemperatur
15 #define T_N (K2C + 25.0)   // Referenztemperatur in Kelvin
16
17 float r2deg(uint16_t r) {
18   if(r == 0) return NAN;   // ein Widerstand von 0 Ohm ergibt keine sinnvolle Temperatur
19
20   // Widerstand in Kelvin umrechnen
21   float t = T_N * B / (B + T_N * log(r / R_N));
22
23   // Kelvin in Grad Celsius umrechnen
24   return t - K2C;
25
26   // alternativ: Kelvin in Grad Fahrenheit umrechnen
27 //  return t * 9 / 5 - 459.67;
28 }
29
30 void setup() {
31   // LED initialisieren
32   pinMode(LED_BUILTIN, OUTPUT);
```

[7] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=36437

```
33    digitalWrite(LED_BUILTIN, LOW);
34
35    Serial.begin(115200);
36    while(!Serial);
37
38    ftduino.init();
39
40    // Die Temperaturmessung erfolgt mit einem
41    ftduino.input_set_mode(Ftduino::I1, Ftduino::RESISTANCE);
42  }
43
44  void loop() {
45    uint16_t r = ftduino.input_get(Ftduino::I1);
46
47    Serial.print("I1: ");
48    Serial.print(r2deg(r));
49    Serial.println(" Grad Celsius");
50
51    delay(1000);
52  }
```

Sketchbeschreibung

Der Temperatur-Sketch verwendet an einigen Stellen sogenannte Fließkommazahlen, um Temperaturen zu speichern. So wird die in der Computertechnik verwendete interne Darstellung von nicht-ganzzahligen Werten (Dezimalbrüchen) genannt. Der Sketch bindet dafür in Zeile 10 die Datei `math.h` ein, um dem Sketch Zugriff auf Fließkommafunktionen zu geben. Zur Speicherung der Fließkommazahlen wird der Datentyp `float` z.B. in Zeile 21 verwendet.

Da der zur Temperaturmessung verwendete Sensor ein Widerstand ist wird in der `setup()`-Funktion in Zeile 41 der Eingang `I1` des ftDuino auf Widerstandsmessung eingestellt.

Der eigentlich Widerstandswert wird in Zeile 45 von Eingang `I1` ausgelesen und in der ganzzahligen Variablen `r` abgelegt. Während der Ausgabe des Wertes in Zeile 48 wird die Funktion `r2deg()` aufgerufen. Diese Funktion befindet sich in den Zeilen 17 bis 28. Sie nimmt einen ganzzahligen Widerstandswert in Ohm entgegen und liefert eine Temperatur in Grad Celsius als Fließkommawert zurück.

Zunächst erfolgt in Zeile 21 die Umrechnung des Widerstands in Kelvin. Dazu wird neben dem Widerstand R_N bei 25 °C auch der sogenannte B-Wert des Sensors benötigt. Dieser Wert beschreibt das Verhalten des Sensors außerhalb des 25 °C-Punkts und wie stark dabei der Widerstand auf Temperaturänderungen reagiert. Dieser Wert liegt bei dem von fischertechik vertriebenen Sensor bei 3900.

Für NTCs gilt näherungsweise[8]:

$$\frac{1}{T} = \frac{1}{T_N} + \frac{1}{B}\ln\left(\frac{R_T}{R_N}\right) \Leftrightarrow T = \frac{T_N * B}{B + T_N * \ln\left(\frac{R_T}{R_N}\right)}$$

mit

- T ... aktuelle Temperatur
- T_N ... Nenntemperatur (üblicherweise 25 °C)
- B ... B-Wert
- R_T ... Widerstand bei aktueller Temperatur
- R_N ... Widerstand bei Nenntemperatur

Nach der Umrechung liegt die Temperatur in Kelvin vor. Zur Umrechung in Grad Celsius muss Zeile 24 lediglich eine Konstante abgezogen werden. Eine Umrechnung in Grad Fahrenheit wäre etwas komplexer und ist beispielhaft in Zeile 27 dargestellt.

Die Genauigkeit der Temperaturmessung ist direkt von der Genauigkeit der Widerstandsmessung abhängig und diese ist wie in Abschnitt 1.2.5 erklärt von der Spannungsversorgung abhängig. Zur Temperaturmessung sollte der ftDuino daher aus einer 9v-Quelle mit Spannung versorgt werden. Eine Versorgung nur über die USB-Schnittstelle ist nicht ausreichend.

[8] https://de.wikipedia.org/wiki/Hei%C3%9Fleiter

5.8 Ausgänge an, aus oder nichts davon?

Schwierigkeitsgrad: ★★☆☆☆

Einen Ausgang kann man ein- oder ausschalten, das ist die gängige Sichtweise. Dass es aber noch einen weiteren Zustand gibt ist auf den ersten Blick vielleicht nicht offensichtlich.

Abbildung 5.21: Zwei Lampen an Ausgang O1

Die Ausgänge des ftDuino lassen sich in drei Zustände schalten: `Ftduino::HI`, `Ftduino::LO` und `Ftduino::OFF`.

Am offensichtlichsten ist der Zustand `Ftduino::HI`. In diesem Zustand wird der entsprechende Ausgang ftDuino-intern mit der 9-Volt-Versorgungspannung verbunden. Ist an diesem Ausgang eine Lampe oder ein Motor so angeschlossen, dass der zweite Anschluss an Masse liegt, so fließt ein Strom von der 9V-Quelle über den Ausgang durch Lampe oder Motor zur Masse. Der Motor dreht sich und die Lampe leuchtet. Im abgebildeten Beispiel leuchtet die rote Lampe.

Im Zustand `Ftduino::LO` ist der entsprechende Ausgang mit Masse verbunden. Eine wieder mit dem zweiten Anschluss an Masse angeschlossene Lampe wird nun nicht mehr leuchten, da beide Anschlüsse der Lampe auf Masse liegen und die Spannung zwischen beiden Anschlüssen daher 0 Volt beträgt. Schließt man den zweiten Anschluss der Lampe aber an 9 Volt an, so leuchtet sie nun. Der Strom fließt von der Spannungsversorgung des ftDuino über den 9-V-Anschluss, durch die Lampe und schließlich über den auf Masse liegenden Ausgang. Im abgebildeten Beispiel leuchtet nun die blaue Lampe.

Der dritte Zustand ist schließlich der Zustand `Ftduino::OFF`. In diesem Fall ist der Ausgang komplett offen. Er ist weder mit Masse noch mit 9 Volt verbunden und es fließt kein Strom über ihn. Als Resultat leuchten beiden Lampen mit halber Helligkeit, da der Strom nun vom Ausgang völlig unbeeinflusst durch beide Lampen fließt. Dieser Zustand wird oft auch mit dem englischen Begriff "tristate" bezeichnet und entsprechende Ausgänge an Halbleitern als "tristate-fähig". Im Deutschen beschreibt der Begriff "hochohmig" diesen dritten Zustand recht gut.

Der folgende Sketch wechselt im Sekundentakt zwischen den drei Zuständen. Man kann diesen Effekt zum Beispiel ausnutzen, um zwei Motoren oder Lampen an einem Ausgang unabhängig zu steuern, um Ausgänge zu sparen. Allerdings lassen sich bei dieser Verschaltung niemals beide Lampen gleichtzeitig ausschalten

5.8.1 Sketch `OnOffTristate`

```
1  /*
2    OnOffTristate - der dritte Zustand
3  */
4
5  #include <FtduinoSimple.h>
6
7  void setup() { }
8
9  // Die Loop-Funktion wird endlos immer und immer wieder ausgeführt
10 void loop() {
11   // Ausgang O1 auf 9V schalten
12   ftduino.output_set(Ftduino::O1, Ftduino::HI);
13   delay(1000);
14   // Ausgang O1 auf Masse schalten
15   ftduino.output_set(Ftduino::O1, Ftduino::LO);
```

```
16    delay(1000);
17    // Ausgang O1 unbeschaltet lassen
18    ftduino.output_set(Ftduino::O1, Ftduino::OFF);
19    delay(1000);
20  }
```

5.8.2 Leckströme

Ganz korrekt ist die Aussage, dass im hochohmigen bzw. Tristate-Zustand kein Strom fließt nicht. Über die Leistungsendstufen und deren interne Schutzschaltungen fließt oft trotzdem ein geringer Strom. In einigen Fällen wird dies sogar bewusst getan, um z.B. mit Hilfe dieses geringen Stromflusses das Vorhandensein eines angeschlossenen Verbauchers feststellen zu können. Diese sogenannten Leckströme wurden in Abschnitt 5.1.1 bereits beobachtet.

Ersetzt man die zwei Lampen im aktuellen Modell durch zwei Leuchtdioden, so wird man feststellen, dass die vom Ausgang nach Masse angeschlossene LED immer dann leicht glimmt, wenn der entsprechende Ausgang hochohmig geschaltet ist. Nur wenn der Ausgang auf Masse geschaltet ist leuchtet die LED nicht. Man kann also direkt an der LED die drei Zustände unterscheiden.

5.9 Aktive Motorbremse

Schwierigkeitsgrad: ★★☆☆☆

Das Abschalten eines Motors scheint rein elektrisch trivial zu sein. Sobald der Motor von der Spannungsversorgung getrennt wird bleibt er stehen. Im Wesentlichen stimmt das auch so.

Physikalisch bedeutet die Trennung von der Spannungsversorgung lediglich, dass dem Motor keine weitere Engergie zugeführt wird. Dass das letztlich dazu führt, dass der Motor anhält liegt daran, dass die in der Rotation des Motors gespeicherte Energie langsam durch Reibung z.B. in den Lagern der Motorwelle verloren geht. Wie lange es dauert, bis der Motor auf diese Weise zum Stillstand kommt hängt wesentlich vom Aufbau des Motors und der Qualität seiner Lager ab.

Zusätzlich wirken viele Gleichstrom-Elektromotoren, wie sie auch fischertechnik einsetzt, als Generator. Werden sie gedreht, so wird in ihren internen Elektromagneten eine Spannung induziert.

Abbildung 5.22: Der TXT-Encoder-Motor als Generator

Dieser Effekt lässt sich mit einer Leuchdiode leicht nachvollziehen. Schließt man die Leuchtdiode direkt an den Motor an und dreht dann manuell die Motorachse, so leuchtet die Leuchtdiode auf, wenn man den Motor in die richtige Richtung dreht und damit eine Spannung mit der für die Leuchtdiode passenen Polarität erzeugt. Man kann für diesen Versuch auch eine Glühlampe oder gar einen zweiten Motor nehmen. Deren gegenüber einer Leuchtdiode höhere Energieaufnahe erfordert aber ggf. ein etwas kräftigeres Drehen.

Je mehr Last ein Generator versorgen soll und je mehr Energie im entnommen werden soll, desto größer ist die mechanische Kraft, die nötig ist, um den Generator zu drehen. Höhere Last bedeutet in diesem Fall ein geringerer elektrischer Widerstand. Die Leuchtdiode mit ihrer verleichsweise geringen Last besitzt einen hohen elektrischen Widerstand, die Glühlampe und noch mehr der Motor besitzen einen geringen elektrischen Widerstand und belasten bzw. bremsen den Generator damit stärker. Die größte denkbare Last ist in diesem Fall der Kurzschluss. Er hat einen minimalen elektrischen Widerstand und sorgt für maximalen Stromfluss und damit maximale elektrische Last am Generator. Auch die Bremswirkung ist dabei maximal.

Dieser Effekt lässt sich nutzen, um einen Elektromotor zu bremsen. Wende beide Anschlüsse eines Motors miteinander verbunden, so fließt ein Strom, der eine Bremswirkung entwickelt. Das wurde bereits beim Not-Aus-Modell aus Abschnitt 5.2 genutzt, um den Motor im Notfall schnell zu stoppen. Ist dagegen z.B. einer der Anschlüsse des Motors offen, so ist kein geschlossener Stromkreis vorhanden und es fließt kein Strom und es tritt keine Bremswirkung auf. Wie groß ist dieser Effekt aber?

Abbildung 5.23: Elektrisch gebremster und ungebremster Elektromotor

Der fischertechnik-Encoder-Motor enthält eine Möglichkeit zur Drehzahlmessung wie schon im PWM-Experiment in Abschnitt 5.3 genutzt. Das Bremsverhalten dieses Motors lässt sich daher experimentell gut verfolgen.

Abbildung 5.24: Anschluss des TXT-Encoder-Motors an die Anschlüsse `M1` und `C1`

Das Beispiel `Datei ▷ Beispiele ▷ Ftduino ▷ MotorBrake` lässt den Motor an Ausgang `M1` alle fünf Sekunden für drei Umdrehungen laufen und misst dann für eine weitere Sekunde, wieviele weitere Impule der Encoder an Eingang `C1` liefert, nachdem er die drei Umdrehungen vollendet hat und abgeschaltet wurde.

Die Funktion `motor_counter_set_brake()` (siehe Abschnitt 8.2.9) wird dabei im Wechsel so eingerufen, dass der Motor frei ausläuft bzw. dass er aktiv gebremst wird.

Abbildung 5.25: Ausgabe bei Verwendung des TXT-Encoder-Motors

Wie in Abbildung 5.25 zu sehen macht die aktive Bremse einen deutlichen Unterschied. Während der Encoder-Motor ungebremst noch für weitere 90 Impulse weiter dreht, also fast 1,5 volle Umdrehungen, kommt er bei aktiber Bremse bereits nach fünf weiteren Impulse zum Stillstand. Das entspricht lediglich knapp $^1/_{13}$ Umdrehung.

5.10 USB-Tastatur

Schwierigkeitsgrad: ★★☆☆☆

Der ftDuino ist nicht vom klassischen Arduino Uno abgeleitet, sondern vom Arduino Leonardo. Der wesentliche technische Unterschied zwischen beiden Arduinos liegt in der Tatsache, dass der Arduino Uno einen seperaten Chip für die USB-Kommunikation mit dem PC verwendet, während diese Aufgabe beim Arduino Leonardo allein dem ATmega32u4-Mikrocontroller zufällt.

In den meisten Fällen macht das keinen Unterschied und die meisten Sketches laufen auf beiden Arduinos gleichermaßen. Es gibt allerdings sehr große Unterschiede in den Möglichkeiten, die sich mit beiden Arduinos bei der USB-Anbindung bieten. Während der USB-Chip im Uno auf das Anlegen eines `COM:`-Ports beschränkt ist zeigt sich der Leonardo und damit auch der ftDuino sehr viel flexibler und der ftDuino kann sich unter anderem gegenüber einem PC als USB-Tastatur ausgeben.

Da die Ausgänge des ftDuino bei diesem Modell nicht verwendet werden reicht die Stromversorgung über USB und es ist keine weitere Versorgung über Batterie oder Netzteil nötig.

Abbildung 5.26: Tastatur-Nachricht

5.10.1 Sketch `USB/KeyboardMessage`

```
/*
  KeyboardMessage - USB-Tastatur

  Der ftDuino gibt sich als USB-Tastatur aus und "tippt" eine Nachricht, sobald
  ein Taster an Eingang I1 für mindestens 10 Millisekunden gedrückt wird.

  Basierend auf dem Sketch:
  http://www.arduino.cc/en/Tutorial/KeyboardMessage

  Dieser Beispielcode ist Public-Domain.
*/

#include <FtduinoSimple.h>
#include <Keyboard.h>

unsigned long lastButtonEvent = 0;
uint16_t previousButtonState = Ftduino::OFF;    // for checking the state of a pushButton

void setup() {
  // initialize control over the keyboard:
  Keyboard.begin();
}

void loop() {
  // Taste an Eingang I1 auslesen
  uint16_t buttonState = ftduino.input_get(Ftduino::I1);

  // Hat sich der Zustand der Taste geändert?
  if(buttonState != previousButtonState) {
    // ja, Zeit des Wechsels merken
```

```
31        lastButtonEvent = millis();
32        // und den neuen Zustand merken, damit wir weitere
33        // Änderungen erkennen können
34        previousButtonState = buttonState;
35      }
36
37      // Gibt es ein unbearbeitetes Eregnis und hat sich der Zustand der Taste seitdem
38      // für mehr als 10 Millisekunden nicht geändert?
39      if(lastButtonEvent && ((millis() - lastButtonEvent) > 10)) {
40        // Zeit dieses Ereignisses vergessen
41        lastButtonEvent = 0;
42
43        // Taste wurde gedrückt
44        if(buttonState) {
45          // Nachricht "tippen"
46          Keyboard.println("Hallo vom ftDuino!");
47        }
48      }
49    }
```

Sketchbeschreibung

Die Arduino-IDE bringt bereits Bibliotheken mit, um USB-Geräte wie Mäuse und Tastaturen umzusetzen. Der eigentliche Sketch bleibt so sehr einfach und die komplizieten USB-Detail bleiben in den Bibliotheken verborgen. Entsprechend kurz ist auch dieser Sketch.

In der `setup()`-Funktion muss lediglich die Methode `Keyboard.begin()` aufgerunfen werden, um beim Start des ftDuino alle USB-settigen Vorkehrungen zu treffen, so dass der ftDuino vom PC als USB-Tastatur erkannt wird. Allerdings verfügt diese Tastatur zunächst über keine Tasten, so dass man kaum merkt, dass der PC nun über eine zusätzliche Tastatur zu verfügen meint.

Um die Tastatur mit Leben zu füllen muss in der `loop()`-Funktion bei Bedarf ein entsprechendes Tastensignal erzeugt und an den PC gesendet werden. In den Zeilen 25 bis 35 des Sketches wird ein Taster an Eingang `I1` abgefragt und sichergestellt, dass nur Tastendrücke über 10ms Länge als solche erkannt werden (mehr Details zu diesem sogenannten Entprellen findet sich in Abschnitt 5.12).

Immer wenn die Taste an `I1` gedrückt wurde, werden die Sketchzeilen 45 und folgende ausgeführt. Hier wird die Funktion `Keyboard.println()` aus der Arduino-Keyboard-Bibliothek aufgerufen und ein Text an den PC gesendet. Für den PC sieht es so aus, als würde der Text vom Anwender auf der Tastatur getippt [9].

Die Möglichkeit, Nachrichten direkt als Tastatureingaben zu senden kann sehr praktisch sein, erlaubt sie doch ohne weitere Programmierung auf dem PC, die automatische Eingabe z.B. vom Messwerten in eine Tabelle oder ähnlich. Natürlich lässt sich diese Fähigkeit aber auch für allerlei Schabernack nutzen, indem der ftDuino zeitgesteuert oder auf andere Ereignisse reagierend den überraschten Anwender mit unerwarteten Texteingaben irritiert. Bei solchen Späßen sollte man immer eine ordentliche Portion Vorsicht walten lassen, da der falsche Tastendruck zur falschen Zeit leicht einen Datenverlust zur Folge haben kann.

5.11 USB-GamePad

Schwierigkeitsgrad: ★★★☆☆

Aus der PC-Sicht ist der Unterschied zwischen einer USB-Tastatur und einen USB-Joystick oder -Gamepad minimal. Beide nutzen das sogenannte USB-HID-Protokoll (HID = Human Interface Device, also ein Schnittstellengerät für Menschen). Arduino-seitig gibt es aber den fundamentalen Unterschied, dass die Arduino-Umgebung zwar vorgefertigte Bibliotheksfunktionen für Tastaturen mitbringt, für Gamepads und Joysticks aber nicht. Um trotzdem ein USB-Gamepad zu implementieren ist daher im Sketch sehr viel mehr Aufwand zu treiben.

[9] Weitere Informationen und tiefergehenden Erklärungen zu den Arduino-Bibliotheken zur Maus- und Tastaturnachbildung finden sich unter https://www.arduino.cc/en/Reference/MouseKeyboard

Abbildung 5.27: GamePad mit vier Richtungstasten und zwei Feuerknöpfen

5.11.1 Sketch `USB/GamePad`

Das entsprechende Beispiel findet sich unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ USB ▷ GamePad`. Dieser Sketch besteht aus drei Dateien. Während `GamePad.ino` den eigentlichen Sketch enthält implementieren `HidGamePad.cpp` und `HidGamePad.h` denjenigen Teil der Gamepad-Unterstützung, die die Arduino-IDE nicht bietet. Interessant ist vor allem die `_hidReportDescriptor`-Struktur in der Datei `HidGamePad.cpp`.

```
 9  static const uint8_t _hidReportDescriptor[] PROGMEM = {
10    0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
11    0x09, 0x05,                    // USAGE (Gamepad)
12    0x85, REPORT_ID,               // REPORT_ID(3)
13    0xa1, 0x01,                    // COLLECTION (Application)
14    0x09, 0x01,                    //   USAGE (Pointer)
15    0xa1, 0x00,                    //   COLLECTION (Physical)
16    0x09, 0x30,                    //     USAGE (X)
17    0x09, 0x31,                    //     USAGE (Y)
18    0x15, 0x00,                    //     LOGICAL_MINIMUM(0)
19    0x26, 0xff, 0x00,              //     LOGICAL_MAXIMUM(255)
20    0x35, 0x00,                    //     PHYSICAL_MINIMUM(0)
21    0x46, 0xff, 0x00,              //     PHYSICAL_MAXIMUM(255)
22    0x75, 0x08,                    //     REPORT_SIZE(8)
23    0x95, 0x02,                    //     REPORT_COUNT(2)
24    0x81, 0x02,                    //     INPUT (Data,Var,Abs)
25    0xc0,                          //   END_COLLECTION
26    0x05, 0x09,                    //   USAGE_PAGE (Button)
27    0x19, 0x01,                    //   USAGE_MINIMUM (Button 1)
28    0x29, 0x02,                    //   USAGE_MAXIMUM (Button 2)
29    0x15, 0x00,                    //   LOGICAL_MINIMUM(0)
30    0x25, 0x01,                    //   LOGICAL_MAXIMUM(1)
31    0x95, 0x02,                    //   REPORT_COUNT(2)
32    0x75, 0x01,                    //   REPORT_SIZE(1)
33    0x81, 0x02,                    //   INPUT (Data,Var,Abs)
34    0x95, 0x06,                    //   REPORT_COUNT(6)
35    0x81, 0x03,                    //   INPUT (Const,Var,Abs)
36    0xc0                           // END_COLLECTION
37  };
```

Diese vergleichweise kryptische Struktur beschreibt die Fähigkeiten eines USB-HID-Geräts[10]. Sie beschreibt, um welche Art Gerät es sich handelt und im Falle eines Joysticks über was für Achsen und Tasten er verfügt.

In diesem Fall meldet das Gerät, dass es über zwei Achsen X und Y verfügt, die jede einen Wertebereich von 0 bis 255 abdecken. Weiterhin gibt es zwei Buttons, die jeweils nur den Zustand an und aus kennen. Für einen einfachen Joystick reicht diese Beschreibung. Es ist aber leicht möglich, die Beschreibung zu erweitern und zusätzliche Achsen und Tasten vorzusehen. Mit den insgesamt acht analogen und den vier digitalen Eingängen verfügt der ftDuino über ausreichend Verbindungsmöglichkeiten für komplexe Eingabegeräte.

Übliche HID-Geräte sind Tastaturen, Mäuse und Joysticks bzw. Gamepads. Aber die Spezifikation der sogenannten HID-

[10] Mehr Info unter http://www.usb.org/developers/hidpage/

Usage-Tabellen[11] sieht wesentlich originellere Eingabegeräte für diverse Sport-, VR-, Simulations- und Medinzingeräte und vieles mehr vor. Und natürlich ist mit den Ausgängen des ftDuino auch die Implementierung von Rückmeldungen über Lampen oder Motoren in Form von z.B. Force-Feedback möglich.

5.12 Entprellen

Schwierigkeitsgrad: ★★★☆☆

In einigen der vorherigen Sketches wurde unerwartet viel Aufwand betrieben, um Taster abzufragen. Im `Pwm`-Sketch aus Abschnitt 5.3.1 wurde in den Zeilen 35 und 51 eine Verzögerung von einer Millisekunde eingebaut und im `KeyboardMessage`-Sketch in Abschnitt 5.10 wurde in den Zeilen 31 und 39-41 ebenfalls die Zeit erfasst und in die Auswertung des Tastendrucks eingefügt. Die Frage, warum das nötig ist soll etwas näher betrachtet werden.

Der Grund für diese Verwendung von Zeiten bei der Auswertung von einzelnen Tastendrücken ist das sogenannte "Prellen". Mechanische Taster bestehen aus zwei Metallkontakten, die entweder getrennt sind oder sich berühren. In Ruhe sind die Kontakte getrennt und wenn der Taster betätigt wird, dann sorgt eine Mechanik dafür, dass die beiden Metallkontakte in Berührung kommen und der Kontakt geschlossen wird.

Folgender Sketch fragt kontinuierlich einen Taster an Eingang I1 ab und gibt auf dem `COM:`-Port eine Meldung aus, wenn sich der Zustand ändert. Zusätzlich zählt er mit, wie oft sich der Zustand insgesamt bereits geändert hat.

Abbildung 5.28: Entprellen

5.12.1 Sketch `Debounce`

```
1  /*
2    Debounce
3
4    Demonstriert Tastenprellen
5  */
6
7  #include <FtduinoSimple.h>
8
9  // die setup-Funktion wird einmal beim Start aufgerufen
10 void setup() {
11   Serial.begin(9600);
12
13   while(!Serial);       // warte auf USB-Verbindung
14
15   Serial.println("ftDuino Tastenprell-Beispiel");
16 }
17
18 uint8_t letzter_zustand = false;
19 uint8_t wechselzaehler = 0;
20
21 // die loop-Funktion wird immer wieder aufgerufen
```

[11] http://www.usb.org/developers/hidpage/Hut1_12v2.pdf

```
22  void loop() {
23    uint8_t zustand = ftduino.input_get(Ftduino::I1);  // Taster auslesen
24
25    if(zustand != letzter_zustand) {          // Hat der Zustand sich geändert?
26      wechselzaehler = wechselzaehler + 1;    // Ja, Zähler rum eins erhöhen
27
28      Serial.print("I1 ");                    // und eine Meldung ausgeben
29      Serial.print(wechselzaehler);
30      Serial.println(" mal geändert");
31      letzter_zustand = zustand;              // neuen Zustand als letzten merken
32    }
33  }
```

Sketchbeschreibung

In den Zeilen 10 is 16 wird wie schon beim `ComPort`-Beispiel aus Abschnitt 3.3 die Ausgabe an den PC vorbereitet und für den seriellen Monitor eine Nachricht ausgegeben.

In Zeile 23 wird kontinuierlich der Eingang `I1` abgefragt. Hat sich sein Zustand gegenüber dem in der Variablen `letzter_zustand` gespeicherten geändert, so wird dies in Zeile 25 festgestellt. In der Folge wird die Variable `wechselzaehler` erhöht und der neue Wert in den Zeilen 28 bis 30 ausgegeben.

Aufgabe 1: Es zählt zuviel

Etwas merkwürdiges passiert, wenn man den Sketch auf den ftDuino lädt und ausprobiert: Sobald die Taste gedrückt wird erscheinen gleich mehrere Meldungen über Zustandsänderungen des Eingangs und auch der Zähler zählt wesentlich weiter als erwartet. Was passiert hier?

Das Problem ist, dass im Moment des Schaltens der Kontakt nicht sofort perfekt schließt. Stattdessen berühren sich die Metallflächen kurz, federn dann für ein paar Mikrosekunden zurück und öffnen sich wieder für einen sehr kurzen Moment. Erst nach mehreren Federvorgängen kommen die Kontakte zur Ruhe und sind dauerhaft geschlossen.

Lösung 1:

Die einfachste Lösung des Problems liegt darin, ein klein wenig zu warten, bevor man nach einem Schaltereignis ein weiteres akzeptiert. Das erreicht man zum Beispiel, indem man nach Zeile 31 zusätzlich etwas wartet, wie es auch im `Pwm`-Sketch aus Abschnitt 5.3.1 getan wurde.

```
31      letzter_zustand = zustand;              // neuen Zustand als letzten merken
32      delay(10);                              // warte zehn Millisekunden
33    }
```

Nach dieser Änderung zählt der Sketch tatsächlich nur noch einzelne Tastendrücke. Diese einfache Lösung hat aber einen Nachteil: Die Ausführung des gesamten Sketches wird bei jeden Tastendruck für zehn Millisekunden pausiert. Hat der Sketch noch andere Aufgaben zu erledigen, dann wird die Verarbeitung dieser Aufgaben ebenfalls für diese zehn Millisekunden unterbrochen. Je nach verwendetem Taster lässt sich die Zeit auf unter eine Millisekunde verkürzen. Aber bei zu kurzer Wartezeit werden wieder falsche Ereignisse erkannt.

Eleganter ist es daher, bei jedem Ereignis mit der Funktion `millis()` einen Zeitstempel aus dem Systemzeitzähler zu nehmen und erst dann ein Eregnis als gültig zu erkennen, wenn das letzte Ereignis länger als 10 Millisekunden zurück liegt. Der `KeyboardMessage`-Sketch aus Abschnitt 5.10 löst das Problem auf genau diese Weise.

Aufgabe 2: Was passiert denn nun genau?

Wie lange der Taster prellt und wie er sich genau verhält konnte wir bisher nur vermuten. Lässt sich der ftDuino nutzen, um etwas genauer auf das Schaltverhalten des Tasters zu schauen?

Lösung 2:

Um Signalverläufe zu veranschaulichen verfügt die Arduino-IDE über ein sehr einfaches aber interessantes Werkzeug: Den sogenannten "seriellen Plotter" er findet sich im Menü unter `Werkzeuge ▷ Serieller Plotter` und öffnet wie der serielle Monitor ein eigenes Fenstern. Aber statt einen per `COM:`-Port empfangenen Text direkt anzuzeigen interpretiert der serielle Plotter die eingehenden Daten Zeile für Zeile als Werte, die grafisch in einer Kurve dargestellt (geplottet) werden.

Das folgende Beispiel ist unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ BounceVisu` zu finden.

```
1  /*
2    BounceVisu
3
4    visualisiert Tastenprellen
5  */
6
7  #include <FtduinoSimple.h>
8
9  #define EVENT_TIME  480    // 480us
10 uint8_t event[EVENT_TIME];
11
12 // die setup-Funktion wird einmal beim Start aufgerufen
13 void setup() {
14   Serial.begin(9600);
15   while(!Serial);      // warte auf USB-Verbindung
16 }
17
18 // die loop-Funktion wird immer wieder aufgerufen
19 void loop() {
20
21   // Warte bis Taster gedrückt
22   if(ftduino.input_get(Ftduino::I1)) {
23
24     // hole 480 Mikrosekunden lang im MiKrosekundentakt je einen Eingangswert
25     for(uint16_t i=0;i<EVENT_TIME;i++) {
26       event[i] = ftduino.input_get(Ftduino::I1);
27       _delay_us(1);
28     }
29
30     // gib zunächst 20 Nullen aus
31     for(uint16_t i=0;i<20;i++)
32       Serial.println(0);
33
34     // gib die eingelesenen 480 Werte aus
35     for(uint16_t i=0;i<EVENT_TIME;i++)
36       Serial.println(event[i]);
37
38     // Warte eine Sekunde
39     delay(1000);
40   }
41 }
```

Der Sketch wartet in Zeile 22 darauf, dass die Taste an Eingang `I1` gedrückt wird. Daraufhin zeichnet er für eine kurze Weile den Zustand des Eingang `I1` auf. In Zeile 9 ist festgelegt, dass 480 Werte aufgezeichent werden. Zwischen zwei Aufzeichnungen wird in Zeile 27 jeweils eine Mikrosekunde gewartet, so dass insgesamt über 480 Mikrosekunden aufgezeichnet wird. Ist die Aufzeichnung vollständig, dann werden zunächst 20 Zeilen Nullen ausgegeben und danach die vorher aufgezeichneten 480 Werte, so dass insgesamt 500 Werte ausgegeben werden. Die ersten 20 Werte representieren den Zustand vor der Aufzeichnung, als der Taster noch nicht gedrückt wurde.

Die insgesamt 500 Werte stellt der serielle Plotter als Kurve dar. Der Wert ist null, wenn der Kontakt als offen erkannt wird und eins, sobald der Kontakt geschlossen ist. Man sieht in der Grafik, wie der Taster zunächst ca. 40 Mikrosekunden lang mehrfach öffnet und schließt, dann liegt das Signal über 100 Mikrosekunden stabil an, bevor der Kontakt noch ein paar mal öffnet, um schließlich nach insgesamt 200 Mikrosekunden stabil geschlossen zu bleiben. Die in Lösung 1 eingesetzte Pause kann also auf gute 200 Mikrosekunden reduziert werden, ohne dass das Prellen Auswirkungen hätte.

Es ist nötig, die Werte vor der Ausgabe komplett zu erfassen und zu speichern, da die Übermittlung der Zeichen an den PC vergleichsweise viel Zeit in Anspruch nimmt. Würden die Werte sofort an den PC übermittelt, dann wäre die Auflösung von einer Mikrosekunde nicht zu erreichen, da die Datenübermittlung selbst schon länger dauert. Tatsächlich dauert auch

Abbildung 5.29: Verlauf des Prellens im seriellen Plotter

das Auslesen des Eingangs `I1` etwas Zeit und das Zeiterhalten unser Messung ist nicht sehr genau. Es genügt aber, um die prinzipiellen Abläufe darzustellen.

5.13 Nutzung des I²C-Bus

Schwierigkeitsgrad: ☆☆☆☆☆

Wie in Abschnitt 1.2.6 beschrieben verfügt der ftDuino über einen I²C-Anschluss. In der Arduino-Welt ist der I²C-Bus äußerst beliebt, denn er erlaubt den einfach Anschluss einer Vielzahl von preisgünstigen Erweiterungsbausteinen.

Der ftDuino wird mit einer Schutzkappe auf dem I²C-Anschluss ausgeliefert wie in Abschnitt 1.2.6 abgebildet. Diese Kappe muss vor Benutzung des I²C-Anschlusses entfernt werden.

Mit wenig Aufwand lassen sich die meisten Sensoren mit einem passenden Anschlusskabel für den ftDuino versehen. In Abbildung 5.30 ist beispielhaft die Verkabelung einer typischen im Online-Handel günstig erhältlichen MPU6050-Sensor-Platine dargestellt. Der Sensor ist damit direkt an den ftDuino anschließbar.

(a) Verdrahtungsschema (b) fertiger Sensor

Abbildung 5.30: MPU6050-Sensor mit Anschlusskabel für den ftDuino

Um den jeweiligen Sensor in eigenen Projekten zu verwenden sind in der Regel zusätzliche Code-Routinen oder Bibliotheken nötig. Die große Verbreitung der Arduino-Plattform führt dazu, dass man zu praktisch jedem gängigen Sensor mit wenig Suche passende Beispiele und Code-Bibliotheken findet[12].

[12] Eine große Sammlung von Sensorbibliotheken findet sich unter https://github.com/ControlEverythingCommunity.

Abbildung 5.31: Diverse I²C-Sensoren mit passendem Anschlusskabel an den ftDuino

5.13.1 Sketch `I2C/I2cScanner`

Für einen schnellen Test, ob die elektrische Verbindung zum Sensor korrekt ist reicht aber in der Regel ein einfacher Test der I²C-Kommunikation aus. Unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ I2C ▷ I2cScanner` findet sich ein einfaches I²C-Testprogramm, das am I²C-Bus nach angeschlossenen Sensoren sucht und deren Adresse ausgibt. Die jeweilige Adresse eines Sensors wird in der Regel vom Sensorhersteller fest vergeben. Im Falle des MPU-6050 ist dies die Adresse `0x68`. Diese Adresse wird bei korrektem Anschluss des Sensors angezeigt.

Abbildung 5.32: Ausgabe von `I2cScanner` bei angeschlossenem MPU-6050

5.13.2 MPU-6050-Sensor

Für den MPU6050 liefert die ftDuino-Umgebung ein eigenes Beispiel mit. Der Beispiel-Sketch unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ I2C ▷ MPU6050Test` liest die Beschelunigungswerte aus dem MPU-6050 aus und gibt sie auf dem seriellen Monitor aus.

Abbildung 5.33: Ausgabe von `MPU6050Test`

5.13.3 OLED-Display

Eine weitere naheliegenden Anwendung des I²C-Anschlusses ist der Anschluss eines kleinen Displays, mit dem z.B. direkt am ftDuino Messwerte ausgegeben werden können.

Für wenig Geld gibt es im Online-Handel OLED-Displays mit 0,96 Zoll Bilddiagonale. Mit einer Größe von etwas unter $3*3cm^2$ eignen sich diese Displays auch sehr gut für den Einbau in ein entsprechendes fischertechnik-kompatibles Gehäuse [13].

(a) Vorderseite

(b) Rückseite

(c) im Gehäuse

Abbildung 5.34: OLED-Display mit Anschlusskabel für den ftDuino

Beim Anlöten des Kabels muss man sich unbedingt am Aufdruck auf der Display-Platine und nicht an den Skizzen hier orientieren, da es in der Anschlussbelegung Unterschiede zwischen den ansonsten baugleichen Displays gibt.

Dieses Display verwendet als Display-Controller-Baustein den SSD1306[14] von Solomon Systech. Diese Display-Sorte ist im Arduino-Umfeld sehr beliebt und passende Bibliotheken gibt es im Internet[15][16].

Wichtig: Wie viele andere I²C-Sensoren auch ist das OLED-Display nicht ftDuino-spezifisch, sondern wird auch in anderen Arduino-Projekten eingesetzt. Daher ist dessen Unterstützung kein Teil der ftDuino-Installation, sondern es müssen die o.g. Adafruit-Bibliotheken unbedingt seperat installiert werden. Anderfalls wird die Übresetzung des Sketches mit einer Meldung der Art "`fatal error: Adafruit_GFX.h: No such file or directory`" oder ähnlich abgebrochen.

Zum Displaytest bringt die Adafruit_SSD1306-Bibliothek unter `Datei ▷ Beispiele ▷ Adafruit SSD1306 ▷ ssd1306_128x64_i2c` ein Beispiel. Erscheint vor dem Download die Meldung "Height incorrect, please fix Adafruit_SSD1306.h!", dann muss in der Datei `Adafruit_SSD1306.h` folgende Änderung vorgenommen werden, um die Unterstützung des 128x64-Displays zu aktivieren:

```
/*-------------------------------------------------------------------------*/
   #define SSD1306_128_64
//   #define SSD1306_128_32
//   #define SSD1306_96_16
/*=========================================================================*/
```

[13] https://www.thingiverse.com/thing:2542260
[14] Datenblatt des SSD1306: https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf
[15] Adafruit-SSD1306-Bibliothek: https://github.com/adafruit/Adafruit_SSD1306
[16] Adafruit-GFX-Bibliothek: https://github.com/adafruit/Adafruit-GFX-Library

Zusätzlich muss im Sketch selbst die I²C-Adresse von 0x3D nach 0x3C angepasst werden:

```
60    // by default, we'll generate the high voltage from the 3.3v line internally! (neat!)
61    display.begin(SSD1306_SWITCHCAPVCC, 0x3C);  // initialize with the I2C addr 0x3D (for the
          128x64)
62    // init done
```

Die ftDuino-Installation selbst bringt ebenfalls ein Beispiel mit, das dieses Display verwendet. Das Shootduino-Spiel findet sich unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ Shootduino`. Das Spiel erwartet drei Taster an den Eingängen I1, I2 und I3 zur Steuerung des Raumschiffs und ggf. eine Lampe an O1.

5.13.4 VL53L0X LIDAR-Distanzsensor

Fischertechnik liefert einen Ultraschallsensor zur Distanzmessung, der wie in Abschnitt 1.2.6 gezeigt auch am ftDuino betrieben werden kann. Dieser Ultraschallsensor sendet einen Ultraschallimpuls aus und misst die Laufzeit, bis die Schallwelle ein Hinternis erreicht und zum Sensor zurück reflektiert. Aus der Laufzeit und der bekannten Schallgeschwindigkeit lässt sich so die Distanz bestimmen.

In der Arduino-Welt gibt es eine interessante Alternative in Form des VL53L0X-Laser-Distanzsensors. Das Funktionsprinzip gleicht dem des Ultraschallsensors, allerdings kommt kein Schall, sondern Laserlicht zu Einsatz. Der VL53L0X lässt sich recht leicht über I²C mit dem ftDuino verbinden.

Abbildung 5.35: Anschlussschema des VL53L0X an den ftDuino

Ein passendes Gehäuse zum selbst-drucken findet sich im ftDuino-Repository[17].

Abbildung 5.36: VL53L0X in 3D-gedrucktem Gehäuse

Wie für fast alle üblichen I²C-Sensoren findet man auch beim VL53L0X bereits fertige Arduino-Bibliotheken und -Skeches im Internet[18].

[17] VL53L0X-Gehäuse: https://github.com/harbaum/ftduino/tree/master/addons/vl53l0x
[18] Adafruit-Bibliothek für den VL53L0X: https://github.com/adafruit/Adafruit_VL53L0X

5.13.5 ftDuino als I²C-Client und Kopplung zweier ftDuinos

Der ftDuino kann nicht nur andere Geräte über den I²C-Bus ansprechen, er kann sich selbst auch als passives Gerät am Bus ausgeben, um von einem anderen Gerät angesprochen zu werden.

Am einfachsten lässt sich diese Möglichkeit nutzen, wenn zwei ftDuinos direkt über I²C gekoppelt werden.

Abbildung 5.37: Kopplung zweier ftDuinos über I²C

Es wird dazu eine 1:1-Verbindung zwischen den beiden I²C-Anschlüssen der beteiligten Controller hergestellt. Ein Controller muss in diesem Aufbau als Master konfiguriert werden, einer als Slave. Enstprechende Beispiel-Sketches finden sich unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ I2C ▷ I2cMaster` und `Datei ▷ Beispiele ▷ FtduinoSimple ▷ I2C ▷ I2cSlave`.

Der Master fragt kontinuierlich einen an Eingang `I1` angeschlossenen Taster ab und sendet den Zustand des Tasters über I²C an den zweiten, als Slave konfigurierten ftDuino. Dieser schaltet dann eine Lampe an Ausgang `O1` entsprechend ein oder aus.

Die Spannungsversorgung des Masters kann dabei über die I²C-Verbindung erfolgen. Lediglich der Slave muss direkt mit 9 Volt versorgt sein, um die Lampe am Ausgang steuern zu können. Die Versorgung über I²C entspricht der Versorgung über USB mit den bekannten Einschränkungen wie in Abschnitt 1.2.5 beschrieben.

Erweiterter `I2cSlave`

Neben dem einfachen Beispiel `Datei ▷ Beispiele ▷ FtduinoSimple ▷ I2C ▷ I2cSlave`, das auf der funktionsreduzierten `FtduinoSimple`-Bibliothek aufbaut befindet sich unter `Datei ▷ Beispiele ▷ Ftduino ▷ I2C ▷ I2cSlave` ein auf der vollwertigen Bibliothek aufbauendes Beispiel, das die meisten Ein- und Ausgabefähigkeiten des ftDuino über I²C verfügbar macht.

Dieser erweiterte Sketch eignet sich als Basis für komplexe Modelle. Das hier abgebildete Beispiel verwendet drei ftDuinos zur Erweiterung des Master-ftDuino um weitere 24 Aus- und 36 Eingänge.

Der erste ftDuino im Bild ganz links bildet den Master, gefolgt von den drei Slaves. Auf dem ersten Slave läuft das unveränderte Beispiel aus `Datei ▷ Beispiele ▷ Ftduino ▷ I2C ▷ I2cSlave`. Für die weiteren Slaves muss die I²C-Adresse im Sketch jeweils angepasst werden. Das geschieht, indem in Zeile 16 im `I2cSlave`-Sketch die Adresse 43 ersetzt wird durch 44 bzw 45 für die beiden weiteren Slaves.

```
13  void setup() {
14    pinMode(LED_BUILTIN, OUTPUT); // LED initialisieren
15
16    Wire.begin(43);               // tritt I2C-Bus an Adresse #43 als "Slave" bei
17    Wire.onReceive(receiveEvent); // Auf Schreib-Ereignisse registrieren
18    Wire.onRequest(requestEvent); // Auf Lese-Ereignisse registrieren
19
20    ftduino.init();
21
```

Abbildung 5.38: Kopplung von vier ftDuinos über I²C

```
22    // alle Ausgänge sind hochohmig
23    memset(output_mode, 0, sizeof(output_mode));
24  }
```

Der Master enthält in diesem Fall die gesamte eigentliche Programmlogik und die Slaves nehmen fundamentale Steuerbefehle über I²C entgegen. Der Master-Sketch findet sich unter `Datei ▷ Beispiele ▷ Ftduino ▷ I2C ▷ I2cMaster`. In Zeile 12 sind dort die Adressen 43, 44 und 45 der drei Slaves bereits eingestellt. Sollen mehr oder weniger Slaves verwendet werden, so ist die Zeile 12 entsprechend anzupassen.

```
 9  // Liste der anzusteuernden I2c-Clients, beginnend mit 0 (der Master selbst)
10  // und -1 als Endemarkierung. In diesem Fall sind drei Clients unter den Adressen
11  // 43, 44 und 45 angeschlossen
12  static const int8_t clients[] = { 0, 43, 44, 45, -1 };
```

Die Stromversorgung aller vier ftDuinos kann aus einem fischertechik-Netzteil erfolgen, das an den Master angeschlossen wird. Die Slaves werden dann vom 9-Volt-Ausgang des Masters über zweipolige fischertechnik-Kabel versorgt.

Abbildung 5.39: Vier ftDuinos über I²Cgekoppelt

Das Beispiel `Datei ▷ Beispiele ▷ Ftduino ▷ I2C ▷ I2cMaster` eignet sich als Vorlage für komplexe Modelle. Je nach Modell kann es aber praktischer sein, auch die Slaves zu verändern und zu erweitern. Geräte bei Modellen mit weitgehend eigenständigen Untereinheiten (z.B. Taktstraße) kann es einfacher sein, die einzelnen Stationen jeweils von einem ftDuino weitgehend autonom steuern zu lassen und die I²C-Kommunikation auf das nötigste zu reduzieren, um z.B. die Ankunft eines neuen Bauteils in der Station anzukündigen.

Die folgenden Tabelle listete die I²C-Register, wie sie vom `I2cSlave.ino` für den ftDuino implementiert werden. Werte mit vorangestelltem $0x$ sind in Hexadezimalschreibweise dargestellt.

Registerbelegungen für Ausgänge O1 bis O8 bzw. M1 bis M4

$0x00$	Ausgangs-Modus O1/M1
	$0x0x$ - Betrieb als Einzelausgang $0x00$ - Einzelausgang offen/hochohmig (tristate) $0x01$ - Einzelausgang gegen +9V geschaltet (High) $0x02$ - Einzelausgang gegen Masse geschaltet (Low) $0x1x$ - Ausgang mit O2 zum Motorausgang M1 gekoppelt $0x10$ - Motorausgang ungebremst aus (off) $0x11$ - Motorausgang gebremst aus (brake) $0x12$ - Motorausgang linksdrehend an $0x13$ - Motorausgang rechsdrehend an
$0x01$	Ausgangswert (PWM) O1/M1 von 0 (aus) bis 255 (100% an) Der Zustand des Hardwareausgangs wird beim Schreiben dieses Registers aktualisiert
$0x02$	Ausgangs-Modus O2 Der Inhalt dieses Registers wird ignoriert, wenn Ausgangs-Modus O1/M1 (Register $0x00$) den Wert $0x1x$ enthält $0x00$ - Einzelausgang offen/hochohmig (tristate) $0x01$ - Einzelausgang gegen +9V geschaltet (High) $0x02$ - Einzelausgang gegen Masse geschaltet (Low)
$0x03$	Ausgangswert (PWM) O2 von 0 (aus) bis 255 (100% an) Der Inhalt dieses Registers wird ignoriert, wenn Ausgangs-Modus O1/M1 (Register $0x00$) den Wert $0x1x$ enthält Der Zustand des Hardwareausgangs wird beim Schreiben dieses Registers aktualisiert
$0x04$	Ausgangs-Modus O3/M2, siehe Ausgang-Modus O1 (Register $0x00$)
$0x05$	Ausgangswert O3/M2, siehe Ausgangwert O1 (Register $0x01$)
$0x06$	Ausgangs-Modus O4, siehe Ausgang-Modus O2 (Register $0x02$)
$0x07$	Ausgangswert O4, siehe Ausgangwert O2 (Register $0x03$)
$0x08$	Ausgangs-Modus O5/M3, siehe Ausgang-Modus O1 (Register $0x00$)
$0x09$	Ausgangswert O5/M3, siehe Ausgangwert O1 (Register $0x01$)
$0x0a$	Ausgangs-Modus O6, siehe Ausgang-Modus O2 (Register $0x02$)
$0x0b$	Ausgangswert O6, siehe Ausgangwert O2 (Register $0x03$)
$0x0c$	Ausgangs-Modus O7/M4, siehe Ausgang-Modus O1 (Register $0x00$)
$0x0d$	Ausgangswert O7/M4, siehe Ausgangwert O1 (Register $0x01$)
$0x0e$	Ausgangs-Modus O8, siehe Ausgang-Modus O2 (Register $0x02$)
$0x0f$	Ausgangswert O8, siehe Ausgangwert O2 (Register $0x03$)

Registerbelegungen für Eingänge I1 bis I8

Es können maximal die beiden Bytes eines Eingangs in einem gemeinsamen I2C-Transfer gelesen werden. Jeder Eingang muss einzeln gelesen werden.

$0x10$	schreiben: Eingangs-Modus I1 $0x00$ - Spannung $0x01$ - Widerstand $0x02$ - Schalter lesen: Eingangswert I1, Low-Byte (LSB)
$0x11$	lesen: Eingangswert I1, High-Byte (MSB)
$0x12$	Eingangs-Modus/Eingangswert I2, siehe Eingangs-Modus I1 (Register $0x10$)
$0x13$	Eingangswert I2, siehe Eingangswert I1 (Register $0x11$)
$0x14$	Eingangs-Modus/Eingangswert I3, siehe Eingangs-Modus I1 (Register $0x10$)
$0x15$	Eingangswert I3, siehe Eingangswert I1 (Register $0x11$)
$0x16$	Eingangs-Modus/Eingangswert I4, siehe Eingangs-Modus I1 (Register $0x10$)
$0x17$	Eingangswert I4, siehe Eingangswert I1 (Register $0x11$)
$0x18$	Eingangs-Modus/Eingangswert I5, siehe Eingangs-Modus I1 (Register $0x10$)
$0x19$	Eingangswert I5, siehe Eingangswert I1 (Register $0x11$)
$0x1a$	Eingangs-Modus/Eingangswert I6, siehe Eingangs-Modus I1 (Register $0x10$)
$0x1b$	Eingangswert I6, siehe Eingangswert I1 (Register $0x11$)
$0x1c$	Eingangs-Modus/Eingangswert I7, siehe Eingangs-Modus I1 (Register $0x10$)
$0x1d$	Eingangswert I7, siehe Eingangswert I1 (Register $0x11$)
$0x1e$	Eingangs-Modus/Eingangswert I8, siehe Eingangs-Modus I1 (Register $0x10$)
$0x1f$	Eingangswert I8, siehe Eingangswert I1 (Register $0x11$)

Registerbelegungen für Zählereingänge C1 bis C4

$0x20$	schreiben: Zähler-Modus C1 $0x00$ - aus $0x01$ - steigende Flanke $0x02$ - fallende Flanke $0x03$ - beide Flanken $0x04$ - Ultraschallsensor aktivieren lesen: Eingangszustand C1
$0x21$	schreiben: Zähler C1 $0x00$ - Zähler unverändert lassen sonst - Zähler löschen lesen: Zählerstand C1/Ultraschall-Distanz, Low-Byte (LSB)
$0x22$	Zählerstand C1/Ultraschall-Distanz, High-Byte (MSB)
$0x24$	schreiben: Zähler-Modus C2 $0x00$ - aus $0x01$ - steigende Flanke $0x02$ - fallende Flanke $0x03$ - beide Flanken lesen: Eingangszustand C2
$0x25$	schreiben: Zähler C2 $0x00$ - Zähler unverändert lassen sonst - Zähler löschen lesen: Zählerstand C2, Low-Byte (LSB)
$0x26$	Zählerstand C2, High-Byte (MSB)
$0x28$	Zähler-Modus C3, siehe Zähler-Modus C2 (Register $0x24$)
$0x29$	Zählerstand C3, Low-Byte (LSB), siehe Zählerstand C2 (Register $0x25$)
$0x2a$	Zählerstand C3, High-Byte (MSB), siehe Zählerstand C2 (Register $0x26$)
$0x2c$	Zähler-Modus C4, siehe Zähler-Modus C2 (Register $0x24$)
$0x2d$	Zählerstand C4, Low-Byte (LSB), siehe Zählerstand C2 (Register $0x25$)
$0x2e$	Zählerstand C4, High-Byte (MSB), siehe Zählerstand C2 (Register $0x26$)

ftDuino als I²C-Slave am PC

Natürlich lässt sich der ftDuino als I²C-Slave nicht nur an anderen ftDuinos betreiben, sondern auch an PCs und anderen Geräten, wenn sie mit einer entsprechenden I²C-Schnittstelle ausgerüstet sind. Im Fall eines PCs lässt sich die nötige I²C-Schnittstelle auch mit Hilfe eines einfachen Adapters über USB nachrüsten. Ein solcher Adapter ist der `i2c_tiny_usb`[19].

(a) Verdrahtungsschema (b) fertiger Adapter

Abbildung 5.40: Digispark/`i2c_tiny_usb` zum Anschluss an den I²C des ftDuino

Auf einem Linux PC[20] kann das Programm `i2c_detect` verwendet werden. Mit dem Paramter `-l` kann man sich zunächst eine Liste aller im PC installierten I²C-Busse ausgeben lassen.

```
$ i2cdetect -l
i2c-3   unknown         i915 gmbus dpc                          N/A
i2c-1   unknown         i915 gmbus vga                          N/A
i2c-8   i2c             em2860 #0                               I2C adapter
i2c-6   unknown         DPDDC-B                                 N/A
i2c-4   unknown         i915 gmbus dpb                          N/A
i2c-2   unknown         i915 gmbus panel                        N/A
i2c-0   unknown         i915 gmbus ssc                          N/A
i2c-9   i2c             i2c-tiny-usb at bus 001 device 023      I2C adapter
i2c-7   unknown         DPDDC-C                                 N/A
i2c-5   unknown         i915 gmbus dpd                          N/A
```

Der Digispark/`i2c_tiny_usb` erscheint in diesem Fall als `i2c-9`. Unter dieser Bus-Nummer lässt sich der I²C-Bus des `i2c_tiny_usb` nach Geräten absuchen.

```
$ i2cdetect -y 9
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- 2a -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

In diesem Fall wurde unter Adresse $2a (dezimal 42) der auf der `FtduinoSimple`-Bibliothek basierende einfache `I2cSlave` des ftDuino erkannt.

Im Repository[21] findet sich ein Python-Beispiel, mit dem vom PC aus auf den ftDuino zugegriffen werden kann. Für die unfangreicherer auf der `Ftduino`-Bibliothek basierende Variante gibt es ebenfalls ein Python-Beispiel im Repository[22].

ftDuino als I²C-Slave am TXT

Wie im Abschnitt 5.13 erwähnt ist mit 3,3 Volt betriebene I²C-Anschluss des fischertechnik-TXT-Controllers nicht elektrisch kompatibel zum mit 5 Volt betriebenen I²C-Anschluss des ftDuino.

[19] Weitere Infos zum `i2c_tiny_usb` finden sich unter https://github.com/harbaum/I2C-Tiny-USB
[20] Auch der Raspberry-Pi oder der fischertechnik TXT sind Linux-PCs
[21] https://raw.githubusercontent.com/harbaum/ftduino/master/ftduino/libraries/FtduinoSimple/examples/I2C/I2cSlave/master.py
[22] https://raw.githubusercontent.com/harbaum/ftduino/master/ftduino/libraries/Ftduino/examples/I2C/I2cSlave/master.py

Einfacher Levelshifter

Mit Hilfe eines passenden Pegel-Wandlers kann man aber leicht die nötige Signalanpassung vornehmen. Die Elektronik dazu ist preisgünstig im Online-Handel erhältlich. Man sollte darauf achten, dass die Elektronik die Spannungsversorgung der 3,3-Volt-Seite selbst aus der Versorgung der 5-Volt-Seite erzeugt, da der TXT selbst keine 3,3 Volt zur Verfügung stellt.

(a) Kabelschema

(b) Anschluss an den TXT

Abbildung 5.41: Levelshifter zur Verbindung von TXT und ftDuino

ftDuino-I²C-Expander

Die Funktion eines Levelshifters erfüllt auch der sogenannte I²C-Expander[23]. Dieses Gerät wurde zum Einsatz am ftDuino entworfen, kann aber auch am TXT oder TX betrieben werden.

(a) fertiges Gerät

(b) Einzelteile

Abbildung 5.42: I²C-Expander für den ftDuino

Der I²C-Expander stellt einen TXT-kompatiblen 10-poligen I²C-Anschluss bereit und vier 6-polige TX- bzw. ftDuino-kompatible. Die vier ftDuino-kompatiblen Anschlüsse sind 1-zu-1 verbunden und können zum Anschluss mehrerer I²C-Geräte an den ftDuino verwendet werden. Zusätzlich ist eine Levelshifter enthalten, der einen Anschluss an den TXT bzw. an dessen Sensoren erlaubt. Die Spannungsversorgung des Levelshifters erfolgt vom ftDuino.

Eine passende App für die Community-Firmware des fischertechnik-TXT-Controllers findet sich im cfw-apps-Repository[24].

Da I²C-Geräte am TXT auch unter RoboPro und der Originalfirmware angesteuert werden können lässt sich der ftDuino auf diese Weise auch unter RoboPro als Erweiterung des TXT nutzen.

TXT-I²C-Sensoren am ftDuino

Das Kabel zum Anschluss des ftDuino an den TXT ist nicht auf eines feste Richtung festgelegt. Es kann daher auch dazu verwendet werden, für den TXT entworfene Sensoren an den ftDuino anzuschließen.

[23] Der ftDuino-I²C-Expander: https://github.com/harbaum/ftduino/tree/master/addons/i2c-expander
[24] https://github.com/harbaum/cfw-apps/tree/master/packages/ftDuinoI2C

Getestet wurde dies mit dem "Kombisensor 158402[25] 3-in-1 Orientierungssensor"[26] basierend auf dem Bosch BMX055, für den es auch fertige Arduino-Sketches gibt[27]. Der Anschluss der 9-Volt-Versorgungsspannung erfolgt dabei exakt wie beim TXT an einem der 9-Volt-Ausfänge des ftDuino.

(a) am Levelshifter

(b) am I²C-Expander

Abbildung 5.43: Orientierungssensor von fischertechnik am ftDuino

5.14 WS2812B-Vollfarb-Leuchtdioden

Schwierigkeitsgrad: ★★★★★

Die Nutzung von WS2812B-Leuchtdioden am ftDuino erfordert etwas Lötarbeit sowie die Installation von Hilfs-Bibliotheken. Dieses Projekt wird daher nur einem fortgeschrittenen Nutzer empfohlen.

Der I²C-Anschluss des ftDuino ist zwar primät zum Anschluss von I²C-Geräten gedacht. Da die dort angeschlossenen Signale `SDA` und `SCL` aber auf frei benutzbaren Anschlüsse des ATmega32u4-Mikrocontrollers liegen können sie auch für andere Signalarten zweckentfremdet werden. Ein solches Signal ist der serielle synchrone Datenstrom, wie ihn die WS2812B-Leuchtdioden verwenden. Diese Lechtdioden gibt es für kleines Geld als Meterware bei diversen Online-Anbietern.

Um die interne Stromversorgung des ftDuino nicht zu überlasten sollten maximal zwei WS2812B-Leuchtdioden an der 5-Volt-Versorgung des I²C-Anschlusses des ftDuino betrieben werden. Sollen mehr Leuchtdioden verwendet werden, so ist eine seperate externe 5-Volt-Versorgung vorzusehen.

Abbildung 5.44: Anschluss von zwei WS2812B-Vollfarb-Leuchtdioden

[25] fischertechnik-Datenbank: https://ft-datenbank.de/search.php?keyword=158402
[26] https://content.ugfischer.com/cbfiles/fischer/Zulassungen/ft/158402-Kombisensor-Kurzanleitung-BMX055-2017-06-09.pdf
[27] https://github.com/ControlEverythingCommunity/BMX055

Jeder WS2812B-Leuchtdiodenstreifen vefügt über drei Eingangssignale: `Masse`, `+5V` und `DI`. Versorgungssignale `Masse` und `+5V` werden direkt mit ihren Gegenstücken am I²C-Anschluss verbunden. Das Signal `DI` steht für "Data In" und ist der Datensignaleingang der Leuchtdioden. Der ebenfalls am anderen Ende des Leuchtdiodenstreifens vorhandene Datenausgang (`DO`) darf nicht verwendet werden. Er leitet das über `DI` empfangene Signal gegebenenfalls an zusätzliche Leuchtdioden weiter. Das `DI`-Signal kann wahlweise mit dem `SCL` oder `SDA`-Pin des I²C-Anschluss verbunden werden. Der entsprechende Signalname muss später im Sketch eingetragen werden.

Die Leuchtdioden sollten mit Vorsicht angeschlossen werden. Kurzschlüsse oder falsche Verbindungen können die Leuchtdioden und den ftDuino beschädigen.

5.14.1 Sketch `WS2812FX`

Das Beispiel zu Ansteuern der WS2812B-Leuchtdioden sowie die nötige Code-Bibliothek können beispielsweise der WS2812BFX-Bibliothek[28] entnommen werden. Andere Bibliotheken zur Absteuerung der WS2812B-Leuchtdioden dürften gleichermaßen zu nutzen sein.

Die Installation der Bibliothek erfordert etwas Erfahrung mit der Arduino-IDE. Wurde die Bibliothek korrekt installiert, dann finden sich diverse Beispiele unter `Datei ▷ Beispiele ▷ WS2812FX`. Das Beispiel `auto_mode_cycle` ist gut geeignet, die Funktion der Leuchtdioden zu überprüfen.

Am Beginn des Sketches sind lediglich zwei kleine Änderungen vorzunehmen, um die Zahl und den verwendeten Anschluss der Leuchtdioden anzupassen.

```
1  #include <WS2812FX.h>
2
3  #define LED_COUNT 2
4  #define LED_PIN SCL
5
6  #define TIMER_MS 5000
```

5.15 Musik aus dem ftDuino

Schwierigkeitsgrad: ★★☆☆☆

Der ftDuino verfügt über keinen eingebauten Lautsprecher und kann daher ohne Hilfe keine Töne ausgeben. Es ist aber problemlos möglich, einen Lautsprecher an einen der Ausgänge des ftDuino anzuschließen. Dazu eignet sich natürlich besonders gut die fischertechnik Lautsprecherkassette 36936.

Abbildung 5.45: Anschluss der Lautsprecherkassette an den ftDuino

Wichtig ist, dass ein Vorwiderstand von mindestens 100 Ω zwischen den Lautsprecher und den ftDuino geschaltet wird. Werden die 9 Volt des ftDuino direkt auf den Lautsprecher gelegt, dann kann der Lautsprecher sehr leicht Schaden nehmen und erzeugte Töne wären extrem laut. Der Vorwiderstand begrenzt den maximal fließenden Strom und schützt Lautsprecher und Gehör.

[28] https://github.com/kitesurfer1404/WS2812FX

Schaltet man nun den Ausgang O2 mit 50% PWM-Verhältnis ein, so kann man das PWM-Signal direkt hören.

```
ftduino.output_set(Ftduino::O2, Ftduino::HI, Ftduino::MAX/2);
```

Durch das 50% PWM-Signal wird der Ausgang permanent zwischen HI und OFF umgeschaltet (siehe auch Abschnitt 5.3). Nur wenn der Ausgang HI ist fließt ein Strom vom Ausgang über den Lautsprecher zum Masse-Anschluss. Da die PWM-Frequenz der Ftduino-Bibliothek ca. 200 Hertz beträgt hört man dann einen Ton in dieser Frequenz.

Die PWM-Erzeugung wird wie im Abschnitt 5.3 beschrieben nicht über dafür vorgesehene PWM-Ausgänge des ATmega32u4-Mikrocontrollers erzeugt sondern durch Signale, die der Mikrocontroller kontinuierlich über seinen SPI-Bus an die Ausgangstreiber sendet. Dieses Vorgehen ist sehr flexibel und erlaubt es, alle acht Ausgänge mit unabhängigen Signalen zu steuern, es erfordert aber ein konstantes Mitarbeiten des Mikrocontrollers und nimmt einen gewissen Prozentsatz seiner Rechenzeit permanent in Anspruch. Je höher die PWM-Frequenz, desto häufiger muss der Mikrocontroller die Signale ändern und desto höher ist der Bedarf an Rechenzeit, die nicht im eigentlichen Sketch zur Verfügung steht. Bei 200 Hertz ist dieser Effekt zu vernachlässigen. Für eine Tonerzeugung sind aber Frequenzen im Kilohertzbereich nötig. Sollen nicht nur digitale an/aus-Rechtecksignale sondern z.B. auch analoge Sinussignale erzeugt werden, dann sind sogar PWM-Signale im Megaherz-Bereich nötig. Das kann der ATmega32u4 über den SPI-Bus nicht leisten.

Die MC33879-Ausgangstreiber haben jeweils zwei Eingänge[29], die am SPI-Bus vorbei direkt angesteuert werden können. Im ftDuino liegen die meisten dieser Eingänge auf Masse, aber der für den Ausgang O2 zuständige Eingang EN6 des Ausgangstreibers U3 ist mit dem Pin PB7 des ATmega32u4 verbunden. Damit lässt sich einer der Ausgangstransistoren des Ausgangstreibers über diesen Pin am ATmega32u4 direkt schalten. In der Arduino-Welt hat dieser Pin die Nummer 11 und lässt sich mit den üblichen Arduino-Funktionen schalten.

```
// Highside-Treiber von Ausgang O2 für eine Sekunde aktivieren
pinMode(11, OUTPUT);
digitalWrite(11, HIGH);
delay(1000);
digitalWrite(11, LOW);
```

Um den Effekt am Ausgang zu sehen muss die FtduinoSimple-Bibliothek in den Sketch eingebunden sein, da die nach wie vor nötige Initialisierung der Ausgangstreiber durch die Bibliothek erfolgt.

5.15.1 Sketch Music

Schwierigkeitsgrad: ★☆☆☆☆

Alternativ können auf diesen Pin auch die Arduino-Befehle zur Ton-Erzeugung angewendet werden. Der Beispielsketch unter Datei ▷ Beispiele ▷ FtduinoSimple ▷ Music nutzt dies.

```
// Kammerton A für eine Sekunde spielen
tone(11, 440, 1000);
```

5.15.2 Sketch MusicPwm

Schwierigkeitsgrad: ★★★★★

Der verwendete Pin PB7 ist Teil des ATmega32u4-internen Timer 1. Das bedeutet, dass spezielle Hardware des ATmega32u4 zur Signalerzeugung herangezogen werden kann. Damit können Signale hoher Prequenz ganz ohne Einsatz von Rechenleistung erzeugt werden.

Der Beispiel-Sketch Datei ▷ Beispiele ▷ FtduinoSimple MusicPwm nutzt das, um die exakt gleiche Melodie wie der vorige Sketch zu erzeugen. Der Code ist deutlich kryptischer und sieht komplizierter aus. Während die Arduino-tone()-Funktion aber im Hinterund in einem sogenannten Timer-Interrupt Rechenleistung benötigt, um den Ton zu erzeugen werden die Töne in diesem Beispiel allein aus der sogenannten Timer-Hardware des ATmega32u4-Prozessors erzeugt. Lediglich zum Ändern der Tonhöhe oder zum Stoppen der Tonausgabe ist der eigentliche Rechenteil des Prozessors verantwortlich.

In einem einfachem Musik-Beispiel wie diesem ist der Hintergrundbedarf an Rechenleistung zu vernachlässigen. Sollen aber sehr hohe Frequenzen, z.B. im Ultraschallbereich oder höher erzeugt werden, so steigt der Bedarf an Rechenleistung im Hintergrund, da mit höheren Frequenzen häufiger das Signal geändert werden muss. Die Verwendung der Timer-Hardware

[29] EN5 und EN6, siehe http://cache.freescale.com/files/analog/doc/data_sheet/MC33879.pdf

hat dieses Problem nicht. Die zur Tonerzeugung nötigen Signalwechsel, also das permanente Ein- und Ausschalten des Ausgangs in der gewünschten Ton-Frequenz wird automatisch erledigt und benötigt daher auch bei hohen Frequenzen keinerlei Rechenzeit des Mikrocontrollers.

5.16 Der ftDuino als MIDI-Instrument

Einen Lautsprecher anzuschließen ist nur eine Art, Töne zu erzeugen. Baukastensysteme wie fischertechnik laden natürlich ein, auf elektromechanische Weise Töne auszugeben z.B. mit Hilfe der Klangrohre aus der Dynamic-Baukastenserie.

Mit seiner flexiblen USB-Schnittstelle bietet der ftDuino eine elegante Methode, solche Fähigkeiten nutzbar zu machen. Die sogenannte MIDI-Schnittstelle[30] wurde entwickelt, um Computer und elektronische Musikinstrumente zu verbinden. Neben der elektrisch sehr speziellen MIDI-Verbindung gibt es eine Variante von MIDI über USB. Die Arduino-Umgebung bietet dafür die `MIDIUSB`-Bibliothek an, die über die Bibliotheksverwaltung der IDE direkt installiert werden kann.

Abbildung 5.46: Installation der MIDIUSB-Bibliothek in der Arduino-IDE

5.16.1 Sketch `MidiInstrument`

Der ftDuino-Beispielsketch unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ MidiInstrument` nutzt diese Bibliothek, um den ftDuino als MIDI-Gerät für den PC nutzbar zu machen. Treiber für entsprechende Geräte bringen die gängigen Betriebsysteme bereits mit und Windows, Linux und MacOS erkennen einen zum MIDI-Gerät konfigurierten ftDuino ohne weitere Treiberinstallation als USB-Audio-Gerät.

Der ftDuino ist ein sogenanntes USB-Verbundgerät. Das bedeutet, dass er mehrere USB-Funktionen gleichzeitig umsetzen kann. Im MIDI-Fall bedeutet das, dass er gegenüber dem PC als MIDI-Gerät erscheint und *gleichzeitig* weiterhin über die USB-`COM:`-Schnittstelle verfügt, was speziell während der Sketchentwicklung sehr praktisch sein kann.

Ein mit dem `MidiInstrument`-Sketch versehener ftDuino wird z.B. von einem Linux-PC mit den gängigen MIDI-Werkzeugen erkannt:

```
$ aplaymidi -l
 Port    Client name                      Port name
 14:0    Midi Through                     Midi Through Port-0
 24:0    ftDuino                          ftDuino MIDI 1

$ aplaymidi -p 24:0 demosong.mid
...
```

Der ftDuino wird eine Stimme des Songs auf einem an O2 angeschlossenen Lautsprecher abspielen und gleichzeitig auf dem `COM:`-Port Informationen über die empfangenen Befehle ausgeben und damit als Basis für eine elektromechanisches Instrument dienen.

Nur wenige MIDI-Dateien lassen sich mit diesem einfachen Setup befriedigend abspielen, weil dieses einfache Beispiel nur monophon ist und nur einen Ton zur Zeit abspielen kann. Mehrstimmige Lieder können nicht abgespielt werden. Diese Beschränkung lässt sich in einem mehrstimmigen mechanischen Musikmodell natürlich aufheben und ergibt sich allein aus der sehr simplen hier verwendeten Art der Tonerzeugung.

Eine monophone Beispieldatei `song.mid` findet sich im Verzeichnis des Sketches.

[30] MIDI, Musical Instrument Digital Interface, https://en.wikipedia.org/wiki/MIDI

Kapitel 6

Modelle

Während in den Experimenten aus Kapitel 5 der ftDuino-Controller im Mittelpunkt stand und nur wenige externe Komponenten Verwendung fanden geht es in diesem Kapitel um komplexere Modelle. Der ftDuino spielt dabei eine untergeordnete Rolle.

Sämtliche Modelle stammen aus aktuellen Baukästen bzw. sind nah an deren Modelle angelehnt, so dass ein Nachbau mit dem entsprechenden Kasten mögich ist.

6.1 Automation Robots: Hochregallager

Das Modell Hochregallager stammt aus dem Baukasten "Automation Robots". In der Originalanleitung wird der Einsatz des TX-Controllers beschrieben. Ein Zusatzblatt beschreibt den TXT-Controller.

Abbildung 6.1: Hochregal mit ftDuino

Der Beispielsketch `Datei ▷ Beispiele ▷ Ftduino ▷ HighLevelRack` steuert das Modell "Hochregallager" aus dem Baukasten 511933 "ROBO TX Automation Robots". Der Anschluss des ftDuino an das Modell entspricht dabei exakt dem Schaltplan für den TXT.

Die Bedienung erfolgt dabei aus dem seriellen Monitor vom PC aus[1].

[1] Hochregal-Video https://www.youtube.com/watch?v=Sjgv9RnBAbg

Abbildung 6.2: Serielle Kommunikation mit dem Hochregal

6.2 ElectroPneumatic: Flipper

Die Eingänge des ftDuino sind auch im Schalter-Modus mit den fischertechnik-Fototransistoren kompatibel. Ein beleuchteter Transistor liefert dann den Wahrheitswert "wahr", ein unbeleuchter den Wert "unwahr".

Abbildung 6.3: Flipper auf ftDuino-Basis

Der Beispiel-Sketch des Flippers aus dem ElectroPneumatic-Set findet sich unter `Datei ▷ Beispiele ▷ Ftduino ▷ Pinball`. Er nutzt die Fototransmistoren als Schaltereingänge für die Lichtschranken. Eine durch eine Kugel unterbrochene Lichtschranke liefert dann den Wert "unwahr":

```
if(!ftduino.input_get(Ftduino::I4)) {
  if(millis() - loose_timer > 1000) {
    // ...
  }
  loose_timer = millis();
}
```

Dabei wird ein Timer migeführt, der z.B. in diesem Fall dafür sorgt, dass frühestens eine Sekunde (1000 Millisekunden) nach einem Ereignis ein weiteres Ereignis erkannt wird.

Dieser Sketch nutzt ein OLED-Display, um verbliebene Spielbälle und den Punktestand anzuzeigen[2]. Da am ftDuino noch Ausgänge frei sind können stattdessen auch Lampen oder Leuchtdioden verwendet werden.

[2] Flipper-Video https://www.youtube.com/watch?v=-zmuOhcHRbY

6.3 ROBOTICS TXT Explorer: Linienfolger

Der mobile Linienfolger ist an die Modelle des "ROBOTICS TXT Explorer"-Sets angelehnt und nutzt den "IR Spursensor" dieses Sets.

Abbildung 6.4: Ein Linienfolger auf ftDuino-Basis

Ein passender Beispiel Sketch ist unter `Datei ▷ Beispiele ▷ Ftduino ▷ LineSensor` zu finden. Dieser Sketch wertet kontinuiertlich den Liniensensor aus, um eine schwarzen Linue zu folgen[3].

Der Liniensensor wird mit seinen gelben und blauen Kabeln an zwei beliebige der Eingänge `I1` bis `I8` angeschlossen. Zusätzlich erfolgt über die roten und grünen Kabel die Spannungsversorgung durch den ftDuino.

Abbildung 6.5: Verdrahtunsgschema des Linienfolgers

In diesem Fall ist der Spursensor an die Eingänge `I1` und `I2` angeschlossen. Der Sensor liefert nahezu maximale Spannung (ca. 9 Volt) wenn eine weiße Fläche erkannt wurd und nur wenig Millivolt, wenn die schwarze Linie erkannt wurde.

```
// beiden Eingänge auf Spannungsmessung einstellen
ftduino.input_set_mode(Ftduino::I1, Ftduino::VOLTAGE);
ftduino.input_set_mode(Ftduino::I2, Ftduino::VOLTAGE);

// beide Spannungen auslesen
uint16_t linker_wert = ftduino.input_get(Ftduino::I1);
uint16_t rechter_wert = ftduino.input_get(Ftduino::I2);

// eine Spannung kleiner 1 Volt (1000mV) bedeutet 'Linie erkannt'
```

[3] Linienfolger-Video https://www.youtube.com/watch?v=JQ8TLt5MC9k

```
if((linker_wert < 1000) && (rechter_wert < 1000)) {
  // beide Sensoren haben die Linie erkannt
  // ...
}
```

6.4 Idas Ampel

Ein klassisches Modell ist die Ampel bzw. die Fußgängerampel. Das Modell bildet eine Bedarfsampel mit je drei Lampen für die Autos und zwei für Fußgänger ab.

(a) Verdrahtungsschema

(b) Modell

Abbildung 6.6: Idas Ampel

Ein passender Beispiel-Sketch ist unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ PedestrianLight` zu finden. Er implementiert die Ampel in Form eines Zustandsautomaten. Die Lichtfolge entspricht dabei dem üblichen Ablauf in acht Schritten von "Autos haben grün, Fußgänger rot" über die Grünphase für die Fußgänger bis schließlich die Autos wieder fahren dürfen.

Abbildung 6.7: Die Zustände der Ampel

6.4.1 Zustandsautomaten

Die einfache und naheliegenden Umsetzung der Ampelsteuerung in Software bestünde aus einem Programm, das dem Verlauf der Ampelzustände direkt folgt. Der Programmablauf stoppt, wenn auf den Tastendruck oder auf den Ablauf einer Zeit gewartet wird und setzt fort, wenn das entsprechende Ereignis eingetreten ist. Im Folgenden ist dies exemplarisch für die ersten zwei Zustände der Ampel dargestellt.

```
void loop() {
  // warte auf Tastendruck
  while(!ftduino.input_get(BUTTON)) {
  };

  // Ampel schaltet ein, Autos haben grün, Fußgänger rot
  cars_green();
  pedestrians_red();
  delay(CARS_GREEN_PHASE);

  // Autos bekommen gelb
  cars_yellow();
  delay(YELLOW_PHASE);

  // ...
}
```

Dieses Programm ist kurz und leicht zu verstehen. Das ist eigentlich gut, hat aber einen entscheidenden Nachteil: Während auf ein Ereignis gewartet wird stoppt der gesamte Programmablauf und es ist nicht möglich, weitere Dinge parallel zu erledigen.

Der `PedestrianLight`-Sketch soll aber beispielsweise nebenbei mit der eingebauen LED des ftDuino blinken. Dies soll unterbrechnungsfrei passieren und unabhängig davon sein, in welchem Zustand sich die eigentliche Ampel gerade befindet.

Die Lösung ist ein Zustandsautomat.

```
// die loop-Funktion wird immer wieder aufgerufen
void loop() {
  // Zeitpunkt des nächsten Lichtwechsel-Ereignisses
  static unsigned long next_event = 0;
  // Aktueller Zustand der Amepl
  static char state = 0;

  // Die interne Leuchtdiode soll einmal pro Sekunde blinken
  static unsigned long flash_timer = 0;
  if(millis() > flash_timer + 10)
    digitalWrite(LED_BUILTIN, LOW);
  if(millis() > flash_timer + 1000) {
    digitalWrite(LED_BUILTIN, HIGH);
    flash_timer = millis();
  }

  // Teste ob ein Fussgänger im Zustand 0 (Ampel aus) den
  // Knopf gedrückt hat
  if((state == 0) && (ftduino.input_get(BUTTON)))
```

```
      state = 1;   // ja -> wechsel in Zustand 1

    if(state > 0) {

      // Teste, ob die eingestellte Zeit vergangen ist
      if(millis() > next_event) {
        switch(state) {

          // Ampel wechselt in Zustand 1: Autos haben grün, Fussgänger haben rot
          case 1: {
            // schalte Lampen
            cars_green();
            pedestrians_red();
            // setze Zeitpunkt für nächstes Ereignis
            next_event = millis() + CARS_GREEN_PHASE;
            // setze Zustand für nächstes Ereignis
            state++;   // Kurzschreibweise für "state = state + 1"
            break;
          }

          // Ampel wechselt in Zustand 2: Autos haben gelb, Fussgänger haben rot
          case 2: {
            cars_yellow();
            next_event = millis() + YELLOW_PHASE;
            state++;
            break;
          }

          // Ampel wechselt in Zustand 3: Autos haben rot, Fussgänger haben rot
          case 3: {
            // ...
            break;
          }

          // ...
        }
      }
    }
  }
```

Dieses Listing ist deutlich komplizierter. Aber es hat den großen Vorteil, dass an keiner Stelle aktiv gewartet wird. Stattdessen wird die Programmausführung ständig fortgesetzt. Um trotzdem die einzelnen Ampelphasen ablaufen lassen zu können werden zwei Variablen als Speicher angelegt (`next_event` und `state`). Hier wir permanent vermerkt, in welchem Zustand sich die Ampel befindet und wie lange dieser Zustand noch erhalten bleiben soll.

Auf diese Weise ist es möglich, die LED völlig unabhängig blinken zu lassen und ggf. auch weitere Steueraufgaben zu erledigen.

Geschuldet ist der große Aufwand der Tatsache, dass der ftDuino über kein eigenes Betriebssystem verfügt, das mehrere Programmteile (sogenannte Prozesse oder Threads) gleichzeitig bedienen könnte, wie es auf PCs und Smartphones z.B. üblich ist.

Der große Vorteil des einfachen ftDuino-Ansatzes liegt in der seiner exakten Vorhersagbarkeit. Jeder kennt es vom PC oder Smartphone, wenn das Betriebssystem im Hintergrund unerwartet "beschäftigt" ist und die Programmausführung stockt. Was bei einer Bedienoberfläche nur lästig ist kann bei Steuer- und Regelaufgaben leicht zu einem Problem werden, wenn z.B. ein Motor bei Erreichen einer bestimmten Position nicht schnell genug gestoppt wird. Aus diesem Grund kann der wesentlich einfachere ftDuino auf viele Dinge schneller und vorhersagbarer reagieren als z.B. ein vom Linux-Betriebssystem angetriebener TXT-Controller oder Raspberry-Pi. Ein weiterer positiver Effekt des nicht vorhandenen Betriebssystems ist der schnelle Systemstart. Ein ftDuino ist sofort nach dem Einschalten voll funktionsfähig und man muss keinen Betriebssystemstart abwarten, bevor das Gerät seine Aufgaben erfüllen kann.

Der schnelle Systemstart und das leicht vorhersagbare Verhalten sind die Hauptgründe, warum es auch im kommerziellen Umfeld immer einen Bedarf an solch einfachen Systemen wie dem ftDuino gibt, auch wenn der Einsatz komplexer, betriebssystembasierter Lösungen mit sinkenden Hardware-Preisen auch in immer einfacheren Geräten möglich wird.

Kapitel 7

Community-Projekte

Der ftDuino ist ein echtes Community-Projekt. Er basiert auf Ideen aus der fischertechnik-Community und integriert sich entsprechend gut in bestehende Community-Projekte. Während kommerzielle Produkte oft in Konkurrenz mit ihren eigenen Vorgängern stehen und dem Kunden vor allem Neues geboten werden soll können es sich Community-Projekte leichter erlauben, auch ältere und technisch in Konkurrenz stehende Systeme einzubinden.

Der ftDuino lässt sich wie in Abschnitt 5.13.5 beschrieben mit dem fischertechnik-TXT-Controller per I^{2}C koppeln. Auf dem TXT kommt dabei die sogenannte Community-Firmware[1] zum Einsatz. Entsprechende Programme zur Anbindung des ftDuino per I^{2}Cfinden sich ebenfalls dort[2].

7.1 `ftduino_direct`: ftDuino-Anbindung per USB an TXT und TX-Pi

Einfacher und robuster ist die Anbindung per USB an PCs, den TXT oder auch den Raspberry-Pi. Die Community stellt dazu einen Sketch sowie eine passende Pythin-Bibliothek zur Verfügung[3].

Mit Hilfe des Sketches stellt der ftDuino seine Anschlüsse einem per USB angeschlossenen übergeordneten Gerät zur Verfügung. Die Python-Bibliothek kann auf dem übergeordneten Gerät genutzt werden, um aus einem Python-Programm auf die Ein- und Ausgänge des ftDuino zuzugreifen.

Abbildung 7.1: ftDuino per USB am Raspberry-Pi

[1] Cfischertechnik TXT Community-Firmware http://cfw.ftcommunity.de/ftcommunity-TXT
[2] ftDuino I^{2}C für die CFW https://github.com/harbaum/cfw-apps/tree/master/packages/ftDuinoI2C
[3] `ftduino_direct`-Sketch https://github.com/PeterDHabermehl/ftduino_direct

Mit Hilfe dieser Bibliothek lassen sich bestehende Python-Programme für die Community-Firmware leicht auf die Nutzung eines ftDuino erweitern. Besoders interessant ist dies auf Gerätem wie dem Raspberry-Pi, da diese von Haus aus keine Schnittstelle zu fischertechnik-Sensoren und -Aktoren mitbringen.

Abbildung 7.2: startIDE auf TX-Pi mit Zugriff auf ftDuino

Programme wie startIDE[4] und Brickly[5] können mit Hilfe von ftduino_direct auf den ftDuino zugreifen und so auf dem Raspberry-Pi die fischertechnik-kompatiblen Anschlüsse des ftDuino nutzen.

7.2 ftDuinIO: ftDuino-Kontroll-App für TXT und TX-Pi

Die Installation des ftduino_direct-Sketches auf dem ftDuino kann wie gewohnt per Arduino-IDE erfolgen, benötigt aber einen klassischen PC. Um vom PC komplett unabhängig zu sein wurde die ftDuinIO-App für die Community-Firmware entwickelt.

Abbildung 7.3: Die ftDuinIO-App

Die App kann auf dem fischertechnik-TXT ebenso betrieben werden wie auf dem zum TX-Pi konfigurierten Raspberry-Pi und erlaubt es, Sketches auf den ftDuino zu laden sowie die ftduino_direct-Funktionen zu testen.

[4] startIDE: https://forum.ftcommunity.de/viewtopic.php?f=33&t=4297
[5] Brickly: https://cfw.ftcommunity.de/ftcommunity-TXT/de/programming/brickly/

7.3 `Brickly-Plugin`: Grafische ftDuino-Programmierung in Brickly

Brickly[6] ist eine auf Googles Blockly[7] basierende grafische Programmierungebung.

Abbildung 7.4: Die Brickly-Programmieroberfläche

Brickly wurde für den fischertechnik-TXT-Controller geschrieben und bringt alles mit, um dessen Ein- und Ausgänge nutzen zu können. Brickly selbst benötigt einen leistungsfähigen Controller, um darauf zu laufen. Dies kann ein fischertechnik-TXT-Controller oder auch ein Raspberry-Pi sein. Die Bedienung und Programmierung erfolgt dann im Web-Browser mit Hilfe eines per WLAN verbundenen Smartphones oder PCs.

Der ftDuino selbst ist nicht leistungsfähig genug, um Brickly auszuführen. Auch die nötige WLAN-Verbindung bringt der ftDuino nicht mit.

Stattdessen kann Brickly einen an den fischertechnik-TXT-Controller oder einen Raspberry-Pi (TX-Pi) angeschlossenen ftDuino ansteuern. Brickly nutzt dazu die in Abschnitt 7.1 beschriebene `ftduino_direct`-Anbindung.

Das ftDuino-plugin für Brickly[8] kann in eine bestehende Brickly-Installation direkt in der Browser-Oberfläche installiert werden.

Abbildung 7.5: Installation eines Brickly-Plugins

Danach können die ftDuino-Ein- und -Ausgänge direkt in Brickly-Programmen verwendet werden. Einem Raspberry-Pi erschließt sich so die fischertechnik-Welt, ein TXT-Controller kann so um 20 zusätzliche Ein- und 8 Ausgänge erweitert werden.

7.4 `startIDE`: Programmierung direkt auf dem TX-Pi oder TXT

Die üblichen Programmierumgebung RoboPro für den fischertechnik-TXT- und -TX-Controller benötigt einen Windows-PC zur Programmierung. Selbst das deutlich modernere Brickly (siehe Abschnitt 7.3) ist auf ein exteres Gerät zur Programm-

[6] Brickly-Anleitung https://github.com/EstherMi/ft-brickly-userguide/blob/master/de/brickly/index.md
[7] Google-Blockly https://developers.google.com/blockly/
[8] Brickly-ftDuino-Plugin https://github.com/harbaum/brickly-plugins#ftduino-io—ftduinoxml

Abbildung 7.6: Ein Brickly-Programm zur Ansteuerung des ftDuino

entwicklung angewiesen.

StartIDE[9] wurde dagegen so konzipiert, dass schon der kleine Touchbildschirm des fischertechnik-TXT-Controllers oder eines Rasapberry-Pi ausreicht, um direkt am Gerät Progamme erstellen zu können.

(a) Hauptbildschirm (b) Funktionsauswahl (c) IO-Konfiguration

Abbildung 7.7: Die Oberfläche der StartIDE

Die startIDE unterstützt neben den eigenen Anschlüssen des TXT auch die Anschlüsse einer ganzen Reihe per USB extern anzuschließender Interfaces und damit u.a. auch die des ftDuino. Dazu bedient sich startIDE der bereits in Abschnitt 7.1 vorgestellten ftduino_direct-Anbindung. Da die startIDE auch auf dem TX-Pi bzw. dem Raspberry-Pi läuft verleiht sie diesem die für den Einsatz im fischertechnik-Modellen nötigen Anschlüsse.

Das Gespann TX-Pi (bzw. Raspberry-Pi), ftDuino und startIDE erlaubt damit, fischertechnik-Modelle ganz ohne PC oder Smartphone zu programmieren.

Die ausführliche Bedienungsanleitung[10] der startIDE enthält weitere Details zur Nutzung des ftDuino.

[9] startIDE-Homepage: https://github.com/PeterDHabermehl/startIDE/
[10] startIDE-Anleitung: https://github.com/PeterDHabermehl/startIDE/blob/master/ddoc/Manual_140_de.pdf

Kapitel 8

Bibliotheken

Mit dem Schaltplan aus Anhang A und entsprechendem Know-How über den verbauten ATmega32U4-Controller lassen sich sämtliche Anschlüsse des ftDuino aus einem Arduino-Sketch ansteuern. Allerdings erfordert dieses Vorgehen einiges an Erfahrung und führt zu vergleichsweise komplexen Arduino-Sketches, da sämtlicher Code zur Ansteuerung der diversen Ein- und Ausgänge im Sketch selbst implementiert werden müsste.

Abbildung 8.1: Die ftDuino-Bibliotheken in der Arduino-IDE

Der ftDuino bringt daher sogenannte Bibliotheken mit. Das sind Code-Sammlungen, die bereits fertige Routinen zum Ansteuern der Ein- und Ausgänge des ftDuinos enthalten. Der eigentliche Arduino-Sketch wird dadurch sehr viel einfacher und kürzer und vor allem muss der Programmierer die Askpekte der Hardware selbst gar nicht komplett verstanden haben, da er lediglich einige einfach zu benutzende Routinen zum Zugriff auf die ftDuino-Hardware nutzt. Die Bibliotheken werden als Teil der ftDuino-Installation in der Arduino-IDE automatisch mit installiert. Aktualisierungen dieser Bibliothekten werden von der Arduino-IDE automatisch festgestellt und zum Update angeboten.

Trotzdem ist der direkte Zugriff natürlich weiterhin möglich. Der fortgeschtittene Programmierer kann also nach wie vor an allen Bibliotheken vorbei direkt auf die Hardware zugreifen. Der Code der Bibliotheken ist ebenfalls frei verfügbar[1], sodass der Anwender selbst ggf. Erweiterungen und Verbesserungen vornehmen kann.

Es gibt zwei Bibliotheken, um den ftDuino anzusteuern. Die `FtduinoSimple`-Bibliothek, die sehr einfach gehalten ist und nur ein ganz rudimentäres Funktions-Set bereit stellt und die `Ftduino`-Bibliothek, die deutlich komplexer ist und vielfältige Funktionen zur Signalein- und -ausgabe bietet.

[1] https://raw.githubusercontent.com/harbaum/ftduino/master/ftduino/libraries

8.1 FtduinoSimple

Die FtduinoSimple-Bibliothek ist eine sehr einfache Bibliothek. Sie erlaubt nur die Abfrage von einfachen Digitalwerten (an/aus) und das Ein- und Ausschalten von Ausgängen. Es ist mit ihr weder möglich, analoge Spannungen und Wiederstände einzulesen, noch die Ausgänge mit variablen Werten zu schalten.

Die Vorteile der FtduinoSimple-Bibliothek sind:

Einfachheit Die Bibliothek bietet nur wenige sehr einfach zu nutzende Funktionen. Man kann kaum etwas falsch machen.

Geringer Speicherbedarf Die Bibliothek belegt kaum Flash- oder RAM-Speicher des ftDuino. Fast der gesamte Speicher steht dem eigentlichen Sketch zur Verfügung.

Keine Seiteneffekte Die Bibliothek verwendet keine weitere interne Hardware des ATmega32u4-Controllers und implementiert z.B. keine eigenen Interrupt-Handler. Die gesamte interne Hardware (Timer, Zähler, Interrupts, ...) steht für eigenen Sketche zur Verfügung und man muss mit keinen unerwarteten Effekten rechnen, wenn man direkt auf die ATmega32u4-Hardware zugreift.

Um die FtduinoSimple-Bibliothek zu nutzen muss zu Beginn eines Sketches die entsprechende Include-Zeile eingefügt werden.

```
#include <FtduinoSimple.h>
```

8.1.1 Verwendung im Sketch

Die FtduinoSimple-Bibliothek verbirgt viele Details der Sensor- und Aktoransteuerung vor dem Nutzer und erlaubt es, mit wenigen und einfachen Code-Zeilen die Ein- und Ausgänge in einem Sketch zu verwendet.

Folgendes Beispiel zeigt das periodische Schalten eines Ausganges mit Hilfe der FtduinoSimple-Bibliothek:

```
1  #include <FtduinoSimple.h>
2
3  void setup() {
4    // keine Initialisierung noetig
5  }
6
7  void loop() {
8    // Ausgang O1 einschalten
9    ftduino.output_set(Ftduino::O1, Ftduino::HI);
10   delay(1000);
11   // Ausgang O1 ausschalten
12   ftduino.output_set(Ftduino::O1, Ftduino::LO);
13   delay(1000);
14 }
```

In den Zeilen 9 und 12 wird der Ausgang O1 des ftDuino ein- bzw. ausgeschaltet. Der erste Parameter der output_set()-Funktion gibt dabei den zu schaltenden Ausgang an, der zwei bestimmt, ob der Ausgang ein- oder ausgeschaltet werden soll.

Achtung: Um die Ausgänge benutzen zu können muss der ftDuino natürlich mit 9 Volt versorgt werden.

Die Eingänge des ftDuino lassen sich ebenfalls sehr einfach mit Hilfe der FtduinoSimple-Bibliothek abfragen:

```
1  #include <FtduinoSimple.h>
2
3  void setup() {
4    // keine Initialisierung noetig
5  }
6
7  void loop() {
8    // Eingang I1 einlesen
9    if(ftduino.input_get(Ftduino::I1)) {
10     // Ausgang O1 einschalten
11     ftduino.output_set(Ftduino::O1, Ftduino::HI);
12   } else {
13     // Ausgang O1 ausschalten
14     ftduino.output_set(Ftduino::O1, Ftduino::LO);
15   }
```

```
16  }
```

In Zeile 9 wird der Zustand eines an den Eingang `I1` angeschlossenen Schalters ermittelt. Je nachdem ob er grdrückt (geschlossen) ist oder offen wird in den Zeilen 11 bzw. 14 der Ausgang `O1` ein- oder ausgeschaltet.

8.1.2 `bool input_get(uint8_t ch)`

Diese Funktion liest den Zustand des Eingangs `ch` ein. Erlaubte Werte für `ch` sind `Ftduino::I1` bis `Ftduino::I8`. Der Rückgabewert ist `true`, wenn der Eingang mit Masse verbunden ist und `false`, wenn nicht. Auf diese Weise lassen sich z.B. leicht Taster abfragen, die zwischen dem jeweiligen Eigang und den korrespondierenden Masseanschluss daneben geschaltet sind.

Die Auswertung des Eingangs geschiet nicht im Hintergrund, sondern erfolgt genau in dem Moment, in dem die `input_get()`-Funktion aufgerufen wird. Vor allem, wenn dabei ein anderer Port abgefragt wird als im direkt vorhergehenden Aufruf von `input_get()` kommt es dadurch zu einer Verzögerung von einigen Mikrosekunden, da ftDuino-intern eine Umschaltung auf den geänderten Eingang durchgeführt werden muss.

Beispiel

```
// lies den Zustand einer Taste an Eingang I1
if(ftduino.input_get(Ftduino::I1)) {
  /* ... tue etwas ... */
}
```

8.1.3 `bool counter_get_state(uint8_t ch)`

Diese Funktion entspricht von ihrer Wirkungsweise `input_get()`. Allerdings wird `counter_get_state()` auf die Zählereingänge angewandt. Der Wertebereich für `ch` reicht demnach von `Ftduino::C1` bis `Ftduino::C4`.

Der Rückgabewerte ist `true`, wenn der Eingang mit Masse verbunden ist und `false`, wenn nicht.

Beispiel

```
// lies den Zustand einer Taste an Zaehler-Eingang C1
if(ftduino.counter_get_state(Ftduino::C1)) {
  /* ... tue etwas ... */
}
```

8.1.4 `void output_set(uint8_t port, uint8_t mode)`

Mit der Funktion `output_set()` können die Ausgänge `O1` bis `O8` gesteuert werden. Der Wertebereich für `port` reicht daher von `Ftduino::O1` bis `Ftduino::O8`.

Der Parameter `mode` beschreibt, in welchen Zustand der Ausgang gebracht werden soll. Mögliche Werte für `mode` sind `Ftduino::OFF`, wenn der Ausgang komplett unbeschaltet sein soll, `Ftduino::LO`, wenn der Ausgang gegen Masse geschaltet werden soll und `Ftduino::HI`, wenn der Ausgang auf 9 Volt geschaltet werden soll.

Beispiel

```
// Lampe zwischen Ausgang O1 und Masse leuchten lassn
ftduino.output_set(Ftduino::O1, Ftduino::HI);
```

Hinweis: Ausgänge können nur verwendet werden, wenn der ftDuino mit einer 9-Volt-Versorgung verbunden ist (siehe Abschnitt 1.2.5).

8.1.5 void motor_set(uint8_t port, uint8_t mode)

Die Funktion motor_set() bedient einen Motorausgang M1 bis M4. Motorausgänge werden durch die Kombination von zwei Ausgängen gebildet (M1 = O1 und O2, M2 = O3 und O4, ...). Der Wert für port liegt daher im Bereich von Ftduino::M1 bis Ftduino::M4.

Der Parameter mode gibt an, welchen Zustand der Motorausgang annehmen soll. Mögliche Werte für mode sind Ftduino::OFF, wenn der Motor ausgeschaltet sein soll, Ftduino::LEFT, wenn der Motor sich nach links drehen soll, Ftduino::RIGHT, wenn der Motor sich nach rechts drehen soll und Ftduino::BRAKE, wenn der Motor gebremst werden soll.

Der Unterschied zwischen Ftduino::OFF und Ftduino::BRAKE besteht darin, dass ein noch drehender Motor bei Ftduino::BRAKE durch Zusammenschalten der beiden Anschlüsse aktiv gebremst wird während der Motor bei Ftduino::OFF lediglich spannungslos geschaltet wird und langsam ausläuft.

Beispiel

```
// Motor an Ausgang M1 links herum laufen lassn
ftduino.motor_set(Ftduino::M1, Ftduino::LEFT);
```

Hinweis: Ausgänge können nur verwendet werden, wenn der ftDuino mit einer 9-Volt-Versorgung verbunden ist (siehe Abschnitt 1.2.5).

8.1.6 Beispiel-Sketches

Code-Beispiele zur Nutzung der FtduinoSimple-Bibliothek finden sich im Menü der Arduino-IDE unter Datei ▷ Beispiele ▷ FtduinoSimple.

8.2 Ftduino

Die Ftduino-Bibliothek kapselt alle Funktionen der ftDuino-Hardware, sodass der Anwender bequemen Zugriff auf alle Ein- und Ausgänge hat, ohne sich über die konkrete technische Umsetzung Gedanken machen zu müssen.

Die Ftduino-Bibliothek benötigt selbst etwas Flash-Speicher, RAM-Speicher und Hintergrund-Rechenleistung, so dass nicht alle Resourcen komplett dem Anwendungssketch zur Verfügung stehen. Zusätzlich macht sie Gebrauch von internen Resourcen des ATmega32u4 wie Timern und Interrupts wie jeweils bei den folgenden Funktionsbeschreibungen erwähnt.

Um die Ftduino-Bibliothek zu nutzen muss zu Beginn eines Sketches die entsprechende Include-Zeile eingefügt werden.

```
#include <Ftduino.h>
```

Zusätzlich muss vor Verwendung aller anderen Funktionen die init()-Funktion aufgerufen werden. Dies geschieht sinnvoller Weise früh in der setup()-Funktion.

```
// die setup-Funktion wird einmal beim Start aufgerufen
void setup() {
  // Benutzung der Ftduino-Bibliothek vorbereiten
  ftduino.init();
}
```

8.2.1 Die Eingänge I1 bis I8

Die Eingänge I1 bis I8 sind mit Analogeingängen des ATmega32u4-Mikrocontrollers im ftDuino verbunden. Diese Analogeingänge werden von der ftDuino-Bibliothek permanent im Hintergrund ausgewertet, da die Analog-Wandlung eine gewisse Zeit in Anspruch nimmt und auf diese Weise eine unerwünchte Verzögerung bei der Abfrage von Eingängen vermieden werden kann.

Die ftDuino-Bibliothek nutzt dazu den sogenannten ADC_vect-Interrupt. Die Analog-Digital-Wandler (ADCs) werden auf eine Messrate von ca 8900 Messungen pro Sekunde eingestellt. Jeder Eingang wird zweimal abgefragt, um eine stabile

zweite Messung zu erhalten, so dass für die 8 Eingänge insgesamt 16 Messungen nötig sind. Daraus ergibt sich eine Konvertierungsrate von ca. 560 Messungen pro Sekunde pro Eingang, die automatisch im Hintergrund erfolgen. Beim Auslesen ist der Messwert demnach maximal ca 2 Millisekunden alt und der Wert wird ungefähr alle 2 Millisekunden aktualisiert.

Der ftDuino kann einen sogenannte Pullup-Wiederstand an jedem der Eingänge aktivieren, so dass einer Spannungsmessung eine Wiederstandsmessung erfolgen kann. Auch das wird von der ftDuino-Bibliothek im Hintergrund verwaltet und die Umschaltung erfolgt automatisch vor der Messung. Sie ist auch der Grund, warum pro Kanal zwei Messungen erfolgen. Dies erlaubt den Signalen, sich nach dem Umschaltvorgang und vor der zweiten Messung zu stabilisieren.

8.2.2 `void input_set_mode(uint8_t ch, uint8_t mode)`

Die Funktion `input_set_mode()` setzt den Messmodus des Eingangs `ch`. Gültige Werte für `ch` reichen von `Ftduino::I1` bis `Ftduino::I8`.

Der Wert `mode` kann auf `Ftduino::RESISTANCE`, `Ftduino::VOLTAGE` oder `Ftduino::SWITCH` gesetzt werden. Die Funktion `input_get()` liefert in der Folge Widerstandswerte in Ohm, Spannungswerte in Millivolt oder den Schaltzustand eines Schalters als Wahrheitswert.

8.2.3 `uint16_t input_get(uint8_t ch)`

Diese Funktion liest den aktuellen Messwert des Eingangs `ch` aus. Gültige Werte für `ch` reichen von `Ftduino::I1` bis `Ftduino::I8`.

Der zurückgelieferte Messwert ist ein 16-Bit-Wert. Im Falle einer Spannungsmessung wird ein Wert zwischen 0 und 10.000 zurück geliefert, was einer Spannung von 0 bis 10 Volt entspricht. Im Falle einer Widerstandsmessung wird ein Widerstandswerk von 0 bis 65535 Ohm zurück geliefert, wobei der Wert 65535 auch bei allen Wiederständen größer 65 Kiloohm geliefert wird. Bedingt durch das Messprinzip werden die Werte oberhalb ca. 10 Kiloohm immer ungenauer. Bei einer Schaltermessung wird nur `true` oder `false` zurück geliefert, je nachdem ob der Eingang mit weniger als 100 Ohm gegen Masse verbunden ist (Schalter geschlossen) oder nicht.

Normalerweise liefert diese Funktion den letzten im Hintergrund ermittelten Messwert sofort zurück. Nur wenn direkt zuvor der Messmodus des Eingangs verändert wurde, dann kann es bis zu 2 Millisekunden dauern, bis die Funktion einen gültigen Messwert zurück liefert. Die Funktion blockiert in dem Fall die Programmausführung so lange.

Beispiel

```
// Widerstand an I1 auswerten
ftduino.input_set_mode(Ftduino::I1, Ftduino::RESISTANCE);
uint16_t widerstand = ftduino.input_get(Ftduino::I1);
```

8.2.4 Die Ausgänge `O1` bis `O8` und `M1` bis `M4`

Die Ausgänge `O1` bis `O8` sind acht unabhängige verstärkte Ausgänge zum direkten Anschluss üblicher 9-Volt-Aktoren von fischertechnik wie z.B. Lampen, Ventile und Motoren.

Je vier Ausgänge werden von einem Treiberbaustein vom Typ `MC33879A`[2] angesteuert. Dieser Baustein enhält acht unabhängig steuerbare Leistungstransistoren. Je zwei Transistoren können einen Ausgang auf Masse schalten, auf 9 Volt schalten oder ganz unbeschaltet lassen. Daraus ergeben sich die drei möglichen Zustände jedes Ausgangs `LO` (auf Masse geschaltet), `HI` (auf 9 Volt geschaltet) oder `OFF` (unbeschaltet).

Je zwei Ausgänge `Ox` können zu einem Motorausgang `Mx` kombiniert werden. Ausgänge `O1` und `O2` ergeben den Motorausgang `M1`, `O3` und `O4` den Motorausgang `M2` und so weiter. Der kombinierte Motorausgang kann die vier mögliche Zustände `OFF`, `LEFT`, `RIGHT` und `BRAKE` annehmen. In den Zuständen `LEFT` und `RIGHT` dreht ein angeschlossener Motor je nach Politär des Anschlusses links oder rechts-herum. Im Zustand `OFF` sind beide Ausgänge unbeschaltet und der Motor verhält sich, als

[2] Datenblatt unter http://cache.freescale.com/files/analog/doc/data_sheet/MC33879.pdf.

wäre er nicht angeschlossen und lässt sich z.B. relativ leicht drehen. Im Zustand BRAKE sind beiden Ausgänge auf Masse geschaltet und ein angeschlossener Motor wird gebremst und lässt sich z.B. schwer drehen.

Die Motortreiber sind über die sogenannte SPI-Schnittstelle des ATmega32u4 angeschlossen. Beide Motortreiber sind in Reihe geschaltet und werden bei jedem SPI-Datenstransfer beide mit Daten versorgt. Signaländerungen an den Ausgängen und speziell die PWM-Signalerzeugung (siehe Abschnitt 5.3) zur Erzeugung von Analogsignalen an den Ausgängen erfordern eine kontinuierliche Kommunikation auf dem SPI-Bus im Hintergrund. Dazu implemetiert die Ftduino-Bibliothek einen sogenannte SPI-Interrupt-Handler, der permament im Hintergrund läuft und permanent den Status der Motortreiber aktualisiert.

Hinweis: Die Ausgänge lassen sich nur nutzen, wenn der ftDuino mit einer 9-Volt-Spannungsquelle verbunden ist.

8.2.5 void output_set(uint8_t port, uint8_t mode, uint8_t pwm)

Diese Funktion schaltet einen Einzelausgang. Gültige Werte für port liegen im Bereich von Ftduino::O1 bis Ftduino::O8.

Der Parameter mode gibt an, in welchen Ausgangsmodus der Ausgang geschaltet werden soll. Erlaubte Werte für mode sind Ftduino::OFF (Ausgang ist ausgeschaltet), Ftduino::LO (Ausgang ist auf Masse geschaltet) und Ftduino::HI (Ausgang ist auf 9 Volt geschaltet).

Der pwm-Parameter gibt einen Wert für die Pulsweitenmodulation zur Erzeugung von Analogsignalen vor. Der Wert kann von 0 (Ftduino::OFF) bis 64 (Ftduino::MAX oder Ftduino::ON) reichen, wobei 0 für aus und 64 für an steht. Eine am Ausgang angeschlossene Lampe leuchtet bei 0 nicht und bei 64 maximal hell. Zwischenwerte erzeugen entsprechende Zwischenwerte und eine Lampe leuchtet bei einem pwm-Wert von 32 nur mit geringer Helligkeit. Es sollten wenn Möglich die Konstanten Ftduino::OFF, Ftduino::ON und Ftduino::MAX herangezogen werden, da diese bei einer Veränderung des PWM-Wertebereichs z.B. in späteren Versionen der Ftduino-Bibliothek leicht angepasst werden können. Zwischenwerte können dazu von den Konstanten abgeleitet werden (z.B. Ftduino::MAX/2).

Beispiel

```
// Ausgang O2 auf 50% einschalten
ftduino.output_set(Ftduino::O2, Ftduino::HI, Ftduino::MAX/2);
```

8.2.6 void motor_set(uint8_t port, uint8_t mode, uint8_t pwm)

Die Funktion motor_set() schaltet einen kombinierten Motorausgang. Gültige Werte für port liegen im Bereich von Ftduino::M1 bis Ftduino::M4.

Der Parameter mode gibt an, in welchen Ausgangsmodus der Motorausgang geschaltet werden soll. Erlaubte Werte für mode sind Ftduino::OFF (Ausgang ist ausgeschaltet), Ftduino::LEFT (Motor dreht links), Ftduino::RIGHT (Motor dreht rechts) und Ftduino::BRAKE (Motor wird aktiv gebremst, indem beide Einzelausgänge auf Masse geschaltet werden).

Der pwm-Parameter gibt einen Wert für die Pulsweitenmodulation zur Erzeugung von Analogsignalen vor. Der Wert kann von 0 (Ftduino::OFF) bis 64 (Ftduino::MAX oder Ftduino::ON) reichen, wobei 0 für aus und 64 für an steht. Ein am Ausgang angeschlossener Motor dreht in den Modi Ftduino::LEFT und Ftduino::RIGHT bei 0 nicht und bei 64 mit maximaler Drehzahl. Zwischenwerte erzeugen entsprechende Zwischenwerte und ein Motor dreht bei einem pwm-Wert von 32 nur mit geringerer Drehzahl (für Details zum Zusammenhang zwischen Motordrehzahl und PWM-Werte siehe Abschnitt 5.3). Es sollten wenn Möglich die Konstanten Ftduino::OFF, Ftduino::ON und Ftduino::MAX herangezogen werden, da diese bei einer Veränderung des PWM-Wertebereichs z.B. in späteren Versionen der Ftduino-Bibliothek leicht angepasst werden können. Zwischenwerte können dazu von den Konstanten abgeleitet werden (z.B. Ftduino::MAX/2). Im Modus Ftduino::BRAKE bestimmt der pwm-Wert, wie stark der Motor gebremst wird. Im Modus Ftduino::OFF hat der pwm-Wert keine Bedeutung.

Beispiel

```
// Motor an M3 mit 1/3 Geschwindigkeit links drehen
ftduino.motor_set(Ftduino::M3, Ftduino::LEFT, Ftduino::MAX/3);
```

8.2.7 void motor_counter(uint8_t port, uint8_t mode, uint8_t pwm, uint16_t counter)

Diese Funktion dient zur Ansteuerung von Encoder-Motoren und gleicht in ihren ersten drei Parametern der motor_set()-Funktion. Die Bedeutung dieser Parameter ist identisch.

Der zusätzliche vierte Parameter gibt an, wieviele Impulse der Encoder-Motor laufen soll. Die Schritte werden auf dem korrespondierenden Zählereingang gemessen, also auf Zählereingang C1 für Motorausgang M1, C2 für M2 und so weiter. Nach Ablauf der angegebenen Impulse wird der Motor gestoppt (siehe void motor_counter_set_brake()).

Das Zählen der Impulse und das Stoppen des Motors passieren im Hintergrund und unabhängig von der weiteren Sketchausführung. Die Zahl der pro Motorumdrehung erkannten Impulse hängt vom Motortyp ab. Die Motoren aus dem TXT Discovery Set liefern $63^1/_3$ Impulse pro Drehung der Motorachse, die Motoren aus den ursprünglich für den TX-Controller verkauften Sets liefern 75 Impulse pro Umdrehung.

8.2.8 bool motor_counter_active(uint8_t port)

Die Funktion motor_counter_active() liefert zurück, ob die Impulszählung für den durch port spezifizierten Motorausgang aktiv ist. Gültige Werte von port liegen im Bereich von Ftduino::M1 bis Ftduino::M4.

Aktiv bedeutet, dass der entsprechende Motor durch den Aufruf von motor_counter() gestartet wurde und der Impulszähler bisher nicht abgelaufen ist. Mit dieser Funktion kann u.a. auf das Ablaufen der Impulszählung und das Stoppen des Motors gewartet werden:

Beispiel

```
// TXT-Encodermotor an M4 für drei volle Umdrehungen starten
ftduino.motor_counter(Ftduino::M4, Ftduino::LEFT, Ftduino::MAX, 190);
// warten bis der Motor stoppt
while(ftduino.motor_counter_active(Ftduino::M4));
// Motor hat gestoppt
```

8.2.9 void motor_counter_set_brake(uint8_t port, bool on)

Diese Funktion bestimmt das Bremsverhaltren des Motors an Ausgang port, wenn er durch die Funktion motor_counter() gestartet wird.

Wird der Parameter on auf wahr (true) gesetzt, so wird der Motor nach Ablauf der Zeit aktiv gebremst. Ist er unwahr (false), so wird der Motor lediglich abgeschaltet und er läuft ungebremst aus. Die Standardeinstellung nach der Initialisierung der Bibliothek ist wahr, die aktive Bremsung ist also aktiviert.

In beiden Fällen läuft der Motor nach. Im gebremsten Fall läuft ein Encoder-Motor aus dem TXT-Discovery-Set unbelastet ca. 5 Impulse nach (ca. $^1/_{10}$ Umdrehung bzw. 28,5°). Im ungebremsten Fall läuft der gleiche Motor ca. 90 Impulse (ca. $1^1/_2$ Umdrehungen) nach.

Da die Zähler nach dem Stoppen des Encoders weiterlaufen ist der Nachlauf auch per Programm messbar:

Beispiel

```
// Bremse für Ausgang M4 abschalten
ftduino.motor_counter_set_brake(Ftduino::M4, false);
// TXT-Encodermotor an M4 für drei volle Umdrehungen starten
ftduino.motor_counter(Ftduino::M4, Ftduino::LEFT, Ftduino::MAX, 190);
// warten bis der Motor stoppt
while(ftduino.motor_counter_active(Ftduino::M4));
// etwas länger warten, um dem Motor Zeit zum Nachlaufen zu geben
delay(500);
// Zählerstand ausgeben
Serial.println(ftduino.counter_get(Ftduino::C4));
```

8.2.10 Die Zählereingänge C1 bis C4

Die Zählereingänge arbeiten im Gegensatz zu den Analogeingängen rein digital. Sie untescheiden nur, ob der jeweilige Eingang auf Masse geschaltet ist oder nicht. Dies geschieht üblicherweise durch einen Tester, der zwischen dem Zählereingang und seinem korrespondierenden Masseanschluss angeschlossen ist oder einem Encodermotor, dessen Endoderausgang mit dem Zählereingang verbunden ist. Die Zählereingänge haben interne Pull-Up-Widerstände. Das bedeutet, dass sie vom ftDuino als "high" bzw mit hohem Signalpegel erkannt werden, wenn kein Signal anliegt weil z.B. ein angeschlossender Taster nicht gedrückt ist. Ist der Taster geschlossen, dann schaltet er den Eingang auf Masse, was von ftDuino als "low" erkannt wird.

Die vier Zählereingänge sind direkt mit einem Interrupt-fähigen Eingang am ATmega32u4 verbunden. Technisch ist damit eine Reaktion im Bereich von mehreren hundertausend Zählimpulsen pro Sekunde möglich. Werden aber z.B. Tastendrücke gezählt, so wird das unvermeidliche Prellen (siehe Abschnitt 5.12) zu verfälschten Ergebnissen führen. Aus diesem Grund führt die `Ftduino`-Bibliothek im Hintergrund eine Filterungg durch und begrenzt die minimale Ereignislänge auf eine Millisekunde. Kürzere Ereignisse werden nicht gezählt.

Nach Systemstart sind alle vier Zähler genullt und deaktiviert. Eregnisse an den Eingängen verändern die Zähler also nicht.

Zusätzlich erlaubt der Zählereingang `C1` den Anschluss des fischertechnik ROBO TX Ultraschall-Distanzsensors 133009 wie in Abschnitt 1.2.6 dargestellt.

8.2.11 void counter_set_mode(uint8_t ch, uint8_t mode)

Diese Funktion setzt den Betriebsmodus eines Zählereingangs. Gültige Werte für `ch` reichen von `Ftduino::C1` bis `Ftduino::C4`.

Wird der `mode`-Wert auf auf `Ftduino::C_EDGE_NONE` gesetzt, dann werden keine Signalwechsel gezählt und der Zähler ist deaktiviert. Dies ist der Startzustand.

Wird `mode` auf `Ftduino::C_EDGE_RISING` gesetzt, so werden steigende Signalflanken, also Wechsel des Eingangssignals von Masse auf eine höhere Spannung, gezählt. Dies passiert z.B. wenn ein angeschlossener Taster losgelassen (geöffnet) wird.

Ein `mode`-Wert von `Ftduino::C_EDGE_FALLING` führt dazu, dass fallende Signalflanken, also Wechsel des Eingangssignals von einer höheren Spannung auf Masse, gezählt werden, was z.B. dann geschieht, wenn ein angeschlossener Taster gedrückt (geschlossen) wird.

Wird der `mode`-Wert schließlich auf `Ftduino::C_EDGE_ANY` gesetzt, so führen beide Signaländerungsrichtungen dazu, dass der Zähler erhöht wird. Sowohl das Drücken, als auch das Loslassen einen Testers wird dann z.B. gezählt.

8.2.12 uint16_t counter_get(uint8_t ch)

Diese Funktion liefert den aktuellen Zählerstand zurück. Gültige Werte für `ch` liegen im Bereich von `Ftduino::C1` und `Ftduino::C4`.

Der maximale Wert, der zurück geliefert wird ist 65535. Wird dieser Wert überschritten, dann springt der Zähler wieder auf 0 zurück.

8.2.13 void counter_clear(uint8_t ch)

Mit Hilfe der Funktion `counter_clear()` kann der Zählerstand auf Null gesetzt werden. Gültige Werte für `ch` liegen im Bereich von `Ftduino::C1` und `Ftduino::C4`.

Beispiel

```
// Eine Sekunde lang steigende (low-nach-high) Impulse an Eingang C1 zählen
ftduino.counter_set_mode(Ftduino::C1, Ftduino::C_EDGE_RISING);
ftduino.counter_clear(Ftduino::C1);
delay(1000);
uint16_t impulse = ftduino.counter_get(Ftduino::C1);
```

8.2.14 bool counter_get_state(uint8_t ch)

Der Zustand der Zählereingänge kann auch direkt mit der Funktion `counter_get_state()` abgefragt werden. Die Werte für `ch` müssen im Bereich von `Ftduino::C1` bis `Ftduino::C4` liegen.

Diese Funktion liefert wahr (`true`) zurück, wenn der Eingang mit Masse verbunden ist und unwahr (`false`) wenn er offen ist.

Es findet bei dieser Funktion keine Filterung statt, so dass z.B. Tastenprellen nicht unterdrückt wird. Es können auf diese Weise digitale Signale mit einer sehr hohen Frequenz erfasst werden.

8.2.15 void ultrasonic_enable(bool ena)

An Zählereingang `C1` kann alternativ der fischertechnik ROBO TX Ultraschall-Distanzsensors 133009 wie in Abschnitt 1.2.6 dargestellt betrieben werden. Die Funktion `ultrasonic_enable()` aktiviert die Unterstützung für den Sensor, wenn der Parameter `ena` auf wahr (`true`) gesetzt wird und deaktiviert sie, wenn er auf unwahr (`false`) gesetzt wird.

Wird die Unterstützung für den Ultraschallsensor aktiviert, so wird die Zählfunktion des Eingangs `C1` automatisch deaktiviert.

Ist der Ultraschallsensor aktiviert, so wird er kontinuierlich ca. zweimal je Sekunde im Hintergrund ausgewertet. Der jeweils aktuelle Messwert ist daher maximal 500 Millisekunden alt.

8.2.16 int16_t ultrasonic_get()

Die Funktion `ultrasonic_get()` liefert den Messwert eines an Zählereingang `C1` angeschlossenen Disanzsensors in Zentimetern zurück. Wurde seit aktivierung kein gültiger Messwert vom Sensor empfangen, so wird als Distanz -1 zurück geliefert. Dies geschieht auch, wenn kein Sensor angeschlossen ist.

Der Sensor selbst arbeitet im Bereich von 0 bis 1023 Zentimeter.

Beispiel

```
// Distanzensor an Eingang C1 abfragen
ftduino.ultrasonic_enable(true);
delay(1000);   // eine Sekunde Zeit für erste Messung geben
int16_t distanz = ftduino.ultrasonic_get();
```

Kapitel 9

Selbstbau

Es ist möglich, einen ftDuino manuell zu bauen. Die ersten Prototypen sind so entstanden. Bei manuellem Aufbau bietet es sich an, funktionsgruppenweise vorzugehen und jeweils die Funktionsfähigkeit schrittweise sicher zu stellen.

Basis für den Selbstbau ist die industriell gefertigte Platine aus den Anhängen B und C basierend auf dem Schaltplan entsprechend Anhang A.

9.1 Erste Baustufe „Spannungsversorgung“

Im ersten Schritt wird die Spannungsversorgung aufgebaut. Sie besteht aus den Kondensatoren `C6` bis `C11` sowie `C14`, den Dioden `D1` und `D3` bis `D5` sowie dem Spannungsregler `U2` und der Sicherung `F1`. Die Spannungsversorungs-Leuchtdiode (siehe Abschnitt 1.2.4) `LED2` mit zugehörigem Vorwiderstand `R35` werden ebenfalls montiert. Die Spule `L1` sowie der Kondensator `C1` können auch bereits installiert werden.

Abbildung 9.1: Komponenten der Spannungsversorung

Die 9V-Steckbuchse `121` sowie die beiden Bundhülsen `9VIN+` und `9VIN-` und die USB-Buchse `J1` werden ebenfalls jetzt schon bestückt.

9.1.1 Bauteile-Polarität

Bei folgenden Bauteilen ist auf die Polarität zu achten: `D1` und `D3` bis `D5`, `C6` bis `C8` und `C14` sowie `LED2`.

Der positive Anschluss der Kondensatoren ist durch einen aufgedruckten Balken oder Streifen gekennzeichnet. Im Bestückungsplan ist dieser Anschluss ebenfalls durch einen Balken und zusätzlich durch abgeschrägte Ecken markiert.

Die verschiedenen Dioden des ftDuino nutzen unterschiedliche Symbole beim Bestückungsdruck. Aber auch hier ist sowohl auf der Diode selbst als auch auf dem Bestückungssymbol ein Balken auf der Seite der Kathode vorhanden.

(a) Tantal-Kondensator (b) Diode (c) Leuchtdiode

Abbildung 9.2: Schaltplansymbol, Bauteilepolarität und Bestückungssymbol

Bei den Leuchtdioden (LEDs) ist es etwas schwieriger, die korrekte Polarität zu bestimmen. Auf der Unterseite der LED ist üblicherweise ein Dreieck oder ein „T“ aufgedruckt, das jeweils in Richtung Kathode weist.

Die Polarität von Leuchtdioden kann man mit einem einfachen Multimeter im Dioden-Test-Modus überprüfen. Bringt man die Anode der LED mit dem roten Multimeterkabel und die Kathode mit dem schwarzen Kabel in Kontakt, so leuchtet die LED schwach.

Sind alle Komponenten der Spannungsversorgung bestückt, dann sollte die grüne LED leuchten, sobald der USB-Stecker eingesteckt wird oder sobald 9 Volt über die Bundhülsen oder den 9-V-Rundstecker eingespeist werden.

Leuchtet die LED in mindestens einem der Fälle nicht, so kann man mit Hilfe eines Multimeters leicht verfolgen, wo die Spannung noch vorhanden ist und wo nicht mehr. Wahrscheinlichste Fehler liegen in der Polarität der Dioden oder der LED.

9.1.2 Kontroll-Messungen

Bei einer 9-Volt-Versorgung muss an den zwischen Pin 1 und 2 des I^2C-Anschlusses am noch unbestückten Wannenstecker `SV1` eine Spannung von 5 Volt (±0,4 Volt) messbar sein. Auf keinen Fall darf mit der Bestückung des Mikrocontrollers fortgefahren werden, wenn hier eine deutlich zu hohe Spannung gemessen wird.

Ebenfalls bei einer Versorgung aus einer 9-Volt-Quelle sollten an den bisher unbestückten beiden unteren 9-Volt-Ausgängen nahezu 9 Volt zu messen sein. Etwas Spannungsverlust gegenüber der Quelle tritt durch die Dioden `D3` bzw. `D4` und der Diode `D5` auf.

Zwischen den Pads 14 und 7 am bisher unbestückten `IC1` muss bei reiner USB-Versorgung eine Spannung von etwas unter 5 Volt zu messen sein. Ist das nicht der Fall, dann besitzt der Spannungsregler `U2` keine sogenannte Body-Diode (der empfohlene MCP 1755S hat diese) und die Funktion dieser Diode muss durch `D6` extern nachgerüstet werden. Bei Verwendung des empfohlenden MCP 1755S kann `D6` ersatzlos entfallen.

Die Diode `D5` verhindert, dass der Strom über die Body-Diode zu den Ausgangstreiber des ftDuino gelangt. Andernfalls würden die Ausgänge des ftDuino auch aus einer USB-5-Volt-Versorgung gespeist, was ggf. eine Überlastung des USB und/oder der Body-Diode zur Folge hätte.

9.2 Zweite Baustufe „Mikrocontroller“

Ist die Spannungsversorgung sichergestellt und vor allem liegen auch im 9-Volt-Betrieb am I²C-Anschluss stabile 5 Volt an, dann kann mit dem Mikrocontroller U1 fortgefahren werden.

Abbildung 9.3: Komponenten des Mikrocontrollers

Auch der Mikrocontroller darf nicht verpolt werden, was in diesem Fall heisst, dass er korrekt orientiert aufgelötet werden muss. Sowohl auf der Platine als auch auf dem Chip-Gehäuse findet sich in einer Ecke eine runde Markierung bzw. Vertiefung. Diese Markierung verweist auf den Pin 1 des Mikrocontrollers und bestimmt die korrekte Orientierung. Ist U1 verlötet werden direkt daneben C5 und R29 montiert.

Die übrigen in dieser Baustufe zu montierenden Komponenten beinhalten die Reset-Logik bestehend aus Taster PB1, sowie Diode D2 und Wiederstand R9. Kondensator C4 und die Widerstände R5 und R6 vervollständigen die USB-Schaltung. Den 16-MHz-Systemtakt erzeugen der Quartz Q1 mit den Kondensatoren C2 und C3.

Der I²C-Anschluss SV1 mit seinen Pullup-Widerständen R30 und R31 kann evenfalls jetzt bestückt werden.

Die Leuchtdiode LED1 mit ihrem Vorwiderstand R28 wird direkt vom Mikrocontroller angesteuert und wird ebenfalls jetzt montiert. Bei der Leuchtdiode muss wieder auf korrekte Polarität geachtet werden.

Der sogenannte ISP-Steckverbinder SV2 wird lediglich zum einmaligen Aufspielen des Bootloaders (siehe Abschnitt 1.2.1 benötigt und muss nicht unbedingt fest montiert werden, wenn das Standardgehäuse genutzt werden soll, da es im Gehäuse keinen Auschnitt für diesen Steckverbinder gibt. Die frei verfügbare Druckvorlage[1] hat einen entsprechenden Ausschnitt und kann auch bei fest montiertem SV2 genutzt werden.

9.2.1 Funktionstest des Mikrocontrollers

Der Mikrocontroller wird von Atmel (bzw. Microchip) mit einem USB-Bootloader[2] ausgeliefert. Beim Anschluss an einen PC sollte der Mikrocontroller daher vom PC als Gerät namens ATm32U4DFU erkannt werden. Ist das der Fall, dann sind die wesentlichen Komponenten funktionsfähig. Dass Windows keinen Treiber für dieses Gerät hat kann ignoriert werden.

Der DFU-Bootloader ist nicht kompatibel zur Arduino-IDE und die Arduino-IDE bringt ihren eigenen Bootloader mit. Dieser wird einmalig mit einem Programmiergerät über den Steckverbinder SV2 eingespielt („gebrannt“).

Zum Brennen des Arduino-Bootloaders unerstützt die Arduino-IDE eine ganze Reihe von Programmiergeräten. Es reicht eine einfache Variante, wie der USBasp[3]. Der USBasp wird per USB mit dem PC und über das 10-polige Flachbandkabel mit

[1] Gehäusedruckvorlage https://github.com/harbaum/ftduino/tree/master/case
[2] DFU-Bootloader, http://www.atmel.com/Images/doc7618.pdf
[3] USBasp - USB programmer for Atmel AVR controllers, http://www.fischl.de/usbasp/

Abbildung 9.4: Brennen des Bootloaders über die Arduino-IDE

`SV2` des ftDuino verbunden. Ist `SV2` nicht bestückt, so kann man einen entsprechenden Stecker lose in die Platine stecken und leicht so verkannten, dass alle Kontakte hergestellt werden. Der eigentliche Flashvorgang dauert nur wenige Sekunden und man kann den Stecker problemlos so lange leicht verkanntet festhalten.

Nach dem Brennen sollte der ftDuino unter diesem Namen vom Betriebssystems des PC erkannt werden. Er sollte sich von der Arduino-IDE ansprechen und mit einem Sketch programmieren lassen. Es bietet sich für einen ersten Test der Blink-Sketch unter `Datei ▷ Beispiele ▷ FtduinoSimple ▷ Blink` an, da die dafür nötige `LED1` ebenfalls gerade montiert wurde.

9.3 Dritte Baustufe „Eingänge“

Die dritte Baustufe ist sehr einfach zu löten und besteht primär aus Widerständen, die zum Schutz der Eingänge verwendet werden.

Abbildung 9.5: Komponenten der Eingänge

Die Komponenten der Analogeingänge `I1` bis `I8` sowie der Zählereingänge `C1` bis `C4` werden gleichtzeitig montiert.

Die Bestückung beginnt mit den Widerständen ganz links und `IC1` nach deren Montage die Eingänge `I1` bis `I8` vollständig sind.

Im zweiten Schritt werden dann die Widerstände R36 bis R46 sowie der Transistor T1 bestückt, was die Zählereingänge vervollständigt. Die Triggerschaltung für den Ultraschall-Entferungsmesser (siehe Abschnitt 1.2.6) ist damit auch vollständig.

Mit einem passenden Testprogramm sollte nun jeder Eingang einzeln getestet werden, um auch Kurzschlüsse zwischden den Eingängen zu erkennen. Sollte ein Eingang nicht ie gewünscht funktionieren, so kommt als Fehlerquelle auch der Mikrocontroller aus Baustufe 2 in Betracht.

9.4 Vierte Baustufe „Ausgänge“

In der vierten und letzten Baustufe werden die Komponenten montiert, die zum Betrieb der Ausgänge benötigt werden.

Abbildung 9.6: Komponenten der Ausgänge

Die Leistungstreiber U3 und U4 sollten zuerst bestückt werden, da sie nicht ganz einfach zu löten sind. Vor allem sollten die Bundhülsen nicht bestückt sein, da sie den Zugang zu den Anschlusspads der Treiber sehr erschweren.

Ergänzt werden die Treiber durch nur wenige Widerstände und Kondesatoren.

Auch die Ausgänge sollte einzel getestet werden, um sicherzustellen, dass keine Kürzschlüsse zwischen den Ausgängen existieren.

Anhang A: Schaltplan

Abbildung A.1: Schaltplan ftDuino Version 1.1

Anhang B: Platinenlayout

Abbildung B.1: Platinenlayout ftDuino Version 1.1

Anhang C: Bestückungsplan

Abbildung C.1: Bestückungsplan ftDuino Version 1.1

Anhang D: Maße

Abbildung D.1: Maße A ftDuino Version 1.1

Abbildung D.2: Maße B ftDuino Version 1.1

Anhang E: Gehäuse

Abbildung E.1: Untere Gehäuseschale

Abbildung E.2: Obere Gehäuseschale