# C#

## Wiederholung

Du solltest aus dem 2. Jahrgang folgende Begriffe kennen und erklären können, sowie über deren praktischen Einsatz Bescheid wissen:

* Interpreter – Compiler
* Datentypen
* Arrays
* Verzweigung (if)
* Schleifen
* Klassen
* Objekte
* Datenkomponenten
* Methoden
* Properties
* Sichtbarkeit
* UML
* Vererbung
* Versionsverwaltung (GIT)
* Einfache GUI mit Windows Forms
* Lesen und Schreiben aus einer Textdatei

## GIT-Konzept

Ein Repository mit dem Namen “**2022\_SEW3\_<nachname>”** erstellen.   
Wir arbeiten heuer auf einer lokalen Festplatte: **C:\DEV\<name>**   
**Am Ende jeder Einheit pushen. Dateien auf einer lokalen Festplatte sind niemals sicher!**

## Fehlerbehandlung

Exception Handling (Außnahmbehandlung) ist ein Prinzip der objektorientierten Programmierung um auf Programmfehler zu reagieren.   
Während der Laufzeit können im Programm auftreten. Z.b.:

* Nullreference 🡪 Zugriff auf eine uninitalisierte Vaiable (null)
* Division durch 0
* Cast 🡪 z.B.: int.Parse mit einem Text als Parameter
* Index-out-of-range 🡪 bei Verwendung von Arrays
* Eine Datei wird nicht gefunden
* Ein bestimmter Server ist nicht erreichbar.

Aus diesem Grund sollen Programmteile, wo man mit einem Fehler rechnet (Murphys Law) mit einem try umspannt werden. In einem catch-Block kann der Fehler behandelt werden (nochmals probieren, aussagekräftige Fehlermeldung, …). In einem optionalen finally-Block können noch zusätzliche Operationen ausgeführt werden. Dieser finally-Block wird immer ausgeführt, auch wenn keine Exception auftritt (z.B.: Datei schließen oder von einem Server abmelden, …).

try {

// Hier ist Code, der einen Fehler verursachen kann.

In x = 7 / 0;

}

Catch (Excepion) {

// Hier ist die Fehlerbehandlung

Console.WrieLineEine("Division durch 0 ist einfach keine gute Idee");

}

Finally {

// Optionaler Teil, wird jedoch immer aufgerufen, egal ob Fehler oder nicht.

Console.WriteLine("Ich wünsche dir noch einen schönen Tag.");

}

Es gibt verschiedene Arten von Exceptions, jede dieser "Unterarten" ist von einer Basisklasse Exception abgeleitet.

Mithilfe des Schlüsselwortes throw kann selbst eine Exception geworfen werden. Dazu muss lediglich ein neues Exception Objekt erzeugt werden. Es empfiehlt sich (um der C# Konvention zu folgen) bei Bedarf eigene Exceptions von den Basisklasse Exception abzuleiten (oder eine bereits vordefinierte zu verwenden, wenn passend).

public class JosephinumException : Excpetion { }

throw new JosephinumException();

Jede abgeleitete Exception soll mit dem Wort Exception enden.

Exceptions nicht für normalen Programmablauf verwenden, denn Exceptions sind langsam. Z.B.: lässt sich eine DivisionByZeroExcepion einfach durch eine if-Anweisung, welche den Divisor überprüft vermeiden.

## Static

Statische Elemente (Methoden, Variablen, …) hängen am Datentyp. Nichtstatische Elemente hängen am konkreten Objekt.

Static: Die Variable oder Methode gehör zur Klasse. Es gibt diese Variable oder Methode nur einmal pro Klasse. Z.B.: Main Methode in jedem Programm

* static void Main(strin[] args)
* int.Parse(…)
* Console.WriteLine(…)
* Math.Cos(…)
* …

statische Variable:

* Math.PI

In einer statischen Methode können von der eigenen Klasse nur statische Methoden aufgerufen werden.

Statische Elemente können immer dann verwendet werden, wenn eine Klasse keinen inneren Zustand hat.

Kein static: Die Variable (Datenkomponente) oder Methode gehört zu einem konkreten Objekt. Die entspricht unserer bisherigen Verwendung. In diesem Fall haben die Objekte einen inneren Zustand (z.B.: Objekt mit Vornamen, Zwischenergebnis, …).

## Generische Listen

Bei generischen Listen handelt es sich um eine typsichere Liste. Eine Liste ein verwaltetes Array, das automatisch um das Doppelte vergrößert, sobald es vollläuft. Es ist somit bei der Initialisierung einer Liste keine Angabe über die Länge der Liste notwendig (das erleichtert uns das Leben erheblich 😊).

Die Typsicherheit entsteht durch Angabe des Datentyps in spitzen Klammern.

List<int> meineListe = new List<int>();

Obriges Beispiel erzeugt eine typsichere Liste vom Datentyp int.

Die generische Liste ist im Namensraum System.Collections.Generic definiert.

Die Verwendung einer generischen Liste entspricht im Wesentlichen dem eines Arrays. Die Anzahl der enthaltenen Elemente kann über das Property Count abgefragt werden.

Dictionaries

Eine Hashfunktion bildet Daten (Text, Nachricht, Passwort, …) beliebiger Länge auf einen Hashwert (Hashcode) mit einer festen Länge (z.B.: 256 Bit) ab.

Dabei muss die Hashfunktion folgende Eigenschaften erfüllen:

* Konsistent
  + Bei mehrmaliger Verwendung der Hashfunktion für die gleiche Eingabe, muss immer der gleiche Hashcode generiert werden.
* Gleichverteilung der Hashcodes auf Eingabewerte
  + Verschiedene Eingaben, sollen Unterschiedliche Hashcodes zugeordnet werden.
* Ähnlichen Informationen werden unterschiedlichen Hashcodes zugeordnet werden.
* Effizient

Hashfunktionen finden in der Informatik in verschiedenen Teilbereichen Anwendung.

Beispiel einer Hashfunktion welche einen Text auf eine einstellige Zahl abbildet.

"Edsger W. Dijkstra" 12

"Dennis Ritchie" 21

"Tim Berners Lee" 30 🡨 Hash-Funktion

"Kurt Gödel" 44

"Tobias Nagl" 53

Praktisch finden Hashfunktionen beispielsweise Einsatz in sogenannten Dictionaries. Ein Dictionary besteht aus Key-Value-Pairs (Schlüssel-Wert-Paaren). Dabei kann unter Angabe des Keys auf den Wert zugegriffen werden. Dies funktioniert ähnlich einem Wörterbuch.

Wenn der Wert eines Schlüssels abgefragt wird, muss zuvor überprüft werden, ob dieser Schlüssel überhaupt im Dictionary vorhanden ist.

z.B:

if (myDict.ContainsKey(…)) {  
 …  
}

Intern wird für die Speicherung der Daten ein Array verwende. Dabei wird für jeden Schlüssel ein numerischer Hashwert berechnet, welcher anschließend als Index in einem Array zur Datenspeicherung verwendet wird. Durch die Hashfunktion kann hiermit auch in sehr großen Datenmengen (z.B. mehrere Millionen) ein sehr schneller Zugriff auf ein Element sichergestellt werden.

## Arbeiten mit strings

Ein Text besteht aus einzelnen Buchstaben sogenannten Characters (kurz char). Man kann einen string al Array von char sehen und auch genauso über die eckigen Indexklammern [] darauf zugreifen.

Ein char hat zwei verschiedene Arten der Repräsentation einmal als Zeichen, zum Beispiel 'a' (einfaches Hochkomma) und einmal als Zahl, zum Beispiel 97. In C# sind Tex in UTF-16 codiert.

Properties:

* Length – Länge des strings

Methoden:

* IndexOf("Suchbegriff") 🡪 ermittelt den Index des Suchbegriffs (erstes Vorkommen)
* Split('Trennzeichen') 🡪 zerlegt den Text in einzelne Texte. Das Trenneichen gibt an wo abgeschnitten werden soll. Ergebnis: Array
* ToLower(),ToUpper() 🡪 erstellt einen neuen Text n Klein- oder Großbuchstaben
* Trim() 🡪 entfernt etwaige führende oder nachstehende Leerzeichen.
* string.isNullOrEmpty() 🡪 prüft ob ein Text null oder "" ist.

Wenn man mehrere strings zu einem string zusammenfügen möchte, wird der Code teilweise unübersichtlich. Besser lesbar wird der Code mit der Methode sting.Format()

string a = string.Format("{0} liebe {1} {2}!", "Hallo", 3, "IT");

obiges Beispiel erzeugt: "Hallo liebe 3 IT!"

Die geschwungenen Klammern stellen Platzhalter dar.

Zusätzlich zu dem Platzhalter kann auch noch eine Formatanweisung angegeben werden:

string.Format("{0:c}", 3.456) // erzeugt €3.46

string.Format("{0:f6}", 3.456) // erzeugt 3.456000

string.Format("{0:f2}", 3.456) // erzeugt 3.46

Für weitere Formatstring siehe beispielsweise:

<http://www.independent-software.com/net-string-formatting-in-csharp-cheat-sheet.html>

## 1.8 Assembly (DLL)

Eine Klassenbibliothek (auch DLL genannt) erlaubt die Auslagerung von Klassen und Namensräumen in eine separate Datei. Eine DLL kann nicht direkt ausgeführt werden da diese keinen Einstiegspunkt (z.B. die Main Methode) besitzt. Jedoch können wir über Verweise oder einen Import DLLs in unser Programm einbinden.

Klassenbibliotheken dienen einerseits zur Strukturierung von großen Programmen, viel wichtiger jedoch erlauben sie die Wiederverwendung von Code, da eine DLL natürlich in verschiedenen Programmen eingebunden werden kann.

# 2 Mikrocontroller

Als Mikrocontroller werden Halbleiterchips bezeichnet, die einen Prozessor und zugleich auch Peripheriefunktionen enthalten. Man spricht auch von Ein-Chip-Computersystemen.

Ein sehr bekannter Vertreter davon ist der Arduino. Dies ist ein Mikrocontroller mit digitalen und analogen Ein- und Ausgängen. Die Programmierung erfolgt in C bzw. C++. Arduino basiert auf einem Microchip AVR (Plattform) und ist ein 8-Bit-Mikrocontroller.

Es gibt mehrere Unterarten des Arduinos:

* Arduino UNO R3
* Arduino Nano
* Arduino Mini
* Arduino Micro
* Arduino Mega

Unterscheidung zu Raspberry Pi:

Ein Raspberry Pi ist ein Einplatinencomputer. Es handelt sich dabei um einen vollwertigen PC mit einem Betriebssystem.

Eigenschaften von Arduino und Raspberry Pi:

|  |  |  |
| --- | --- | --- |
|  | Arduino Uno | Raspberry Pi 4B |
| Prozessor | ATmega328P @16 MHz | ArmV8 @1,4 GHz |
| RAM | 2kb | Bis zu 8 GB |
| Multitasking | Nein | Ja |
| Leistungsaufnahme | 0,25 Watt | 10 Watt |
| Flash Speicher | 32 kB | Bis 256 GB (SD karte) |
| USB-Anschlüsse | 1 zur Programmierung | 4 für die Peripherie |
| Betriebssystem | - | Linux, Windows IoT |
| Preis | 10 € | 40€ |

Eignung:

Ist das Projekt eher hardwareorientiert (Sensoren, Aktoren, …) dann ist vermutlich Arduino die bessere Wahl. Erfordert das Projekt jedoch umfangreicher Rechenleitung (z.B. Verarbeitung von Bilddaten, …) oder benötigen Sie einen Monitor zur Anzeige, ist vermutlich der Raspberry Pi die bessere Wahl.

Mit sogenannten Shields kann ein Arduino erweitert werden.

# 3 C/C++

C ist eine imperative (Abfolge von Anweisungen) und prozedurale Programmiersprache.

C++ ist eine Erweiterung der Programmiersprache C und ermöglicht eine objektorientierte Programmierung.

Die Programmiersprachen enthalten sowohl alle notwendigen Sprach Konstrukte einer höheren Programmiersprache, als auch Elemente, die sehr maschinennahe Operationen ermöglichen.

Maschinennahe Operationen:

* Direkter Zugriff auf Speicheradressen mittels Zeiger
* Registervariablen
* Direkte Manipulation einzelner Bits

Installation von Visual C:

* Visual Studio Installer aufrufen
* C++ Komponenten installieren (Desktopentwicklung mit C++)
* Überprüfen, ob Windows 10 SDK ausgewählt ist

oder <https://www.onlinegdb.com/>

Neues Projekt in Visual Studio anlegen:

* Leeres C++ Projekt
* Rechtsklick auf Quelldateien 🡪 Hinzufügen 🡪 neues Element 🡪 Auf Dateiendung "c" achten
* Compiler umstellen 🡪 Rechtsklick auf Projekt 🡪 Eigenschaften 🡪 C/C++ 🡪 Erweitert 🡪 Kompilierungsart auf "Als C-Code kompilieren" umstellen.

# 3.1 Zeiger (Pointer)

|  |
| --- |
| byte a; byte b; byte feld[6]; // zwei Byte am Arduino  int f;  byte c; byte d; |

byte a (Adresse 180) byte b (Adresse 181)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

byte feld[0] (Adresse 182) byte feld[1] (Adresse 183)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

byte feld[2] (Adresse 184) byte feld[3] (Adresse 185)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

byte feld[4] (Adresse 186) byte feld[5] (Adresse 187)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

int f (Adresse 188) Adresse 189

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

byte c (Adresse 190) byte d (Adresse 191)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

In Wirklichkeit steht an einer Programmstelle, an der wir eine Variable verwenden, im Maschinencode ein Verweis auf die Position im Arbeitsspeicher 🡪 also ein Zeiger.

Der Operator & vor einem Variablenname gibt uns die Speicheradresse zurück.

Je nach Betriebssystem (8bit beim Arduino oder 64 bit auf einem PC) können wir eine maximale Größe des Arbeitsspeichers verwalten. Somit benötigen wir in den Variablen (Beispiel 19Pointer) auch unterschiedliche Größen für die Adressen. Korrekt verwenden wir dazu somit die Schreibweise int\* (Zeiger auf eine int-Variable). Die Größe dieses Datentyps wird je nach Betriebssystem angepasst.

Mit dem \* Operator kann dann wieder auf den Wert der Variable zugegriffen werden (siehe Beispiel 19Zeiger).

Zugriff auf Variablen direkt über Speicheradressen. Zugriff auf Elemente des Arrays erfolgt über Startadresse plus "Offset" 🡪 ergibt die Speicheradresse des gesuchten Elements.

# 3.2 Call-by-value und Call-by-reference

Bisher haben wir nur die Werte von Variablen an Funktionen übergeben. Diese Art der Parameterübergabe wird "call by value" genannt. Sie hat den Nachteil, dass nur ein Wert mit return zurückgegeben werden kann. Soll aber eine oder mehrere variablen in der Funktion verändert werden, so muss als Parameter die Adresse der Variable übergeben werden (& Operator). Oder anders gesagt, ein Zeiger auf diese Variable ("call by reference"). Die Adresse übergibt man bei einfachen Variablen (int, byte, float,…) mit dem Adressoperator &.

Siehe Beispiel: Funktion swap2 in Projekt 19Pointer

In C besteht ein sehr enger Zusammenhang zwischen Arrays und Zeigern. Der Name des Arrays (ohne Indexklammern) ist nicht anders als ein Zeiger auf das erste Element. Aus diesem Grund ist der Adressoperator bei der Übergabe an die Funktion nicht nötig. Arrays können nur als call by reference übergeben werden.

Siehe Beispiel: Funktion reset im Projekt 20ZeigerArrays

Bei call by value wird der Wert des Arguments in den Parameter der Funktion kopiert. Dabei werden unterschiedliche Speicherplätze genutzt. Der Speicherplatz des Parameters wird beim Aufruf der Funktion belegt und bei der Rückkehr der Funktion freigegeben. Die Verweise auf den Parameter beziehen sich auf einen eindeutigen Speicher und sind nur innerhalb der Funktion bekannt. Alle Änderungen, die innerhalb der Funktion vorgenommen werden, werden nicht im tatsächlichen Argument wiedergespiegelt.

Bei call by reference wird die Adresse der Argumente in die Parameter der Funktion kopiert. Die Referenz auf den Parameter referenziert denselben Speicher wie im Argument. Alle innerhalb der Funktion vorgenommen Änderungen werden im Argument wiedergegeben, d.h. alle Operationen werden an der Adresse des Arguments gespeicherten Werts ausgeführt und der geänderte Wert wie an derselben Adresse gespeichert.   
Wird "neuer" Speicher in einer Funktion benötigt (z.B. ein neues Array), dann muss diese Variable mit dem Schlüsselwort static versehen werden.

Siehe Beispiel: Funktion getRandom im Projekt 20ZeigerArrays

# 4. Arduino

## 4.1 Referenz Arduino

<https://www.arduino.cc/reference/de/>

Grundsätzlich kann für den Arduino C++ verwendet werden. Meist wird jedoch nur prozedural am Arduino programmiert, jedoch durchaus Klassen aus Bibliotheken verwendet (z.B.: string).

Struktur eines Programmes:

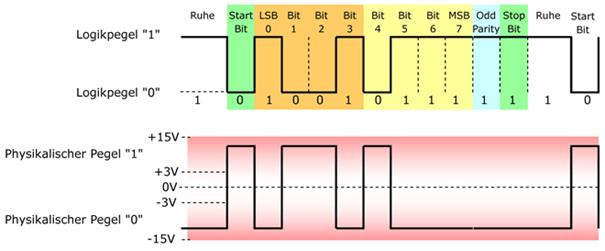
* setup  
  Wird zur Initialisierung (nach dem Aufwachen) einmal ausgeführt.
* Loop  
  Diese wird wiederholt aufgerufen. Nach Beendigung der Funktion, wird sie sofort wieder aufgerufen.

## 4.2 Serie Datenkommunikation (RS-232):

Serielle Datenkommunikation ist eine sehr einfache und bewährte Möglichkeit, über kurze Distanzen Daten zu übertragen. Dafür sind nur 3 Leitungen nötig:

* Tx (Transceive – senden)
* Rx (Receive – empfangen)
* Masse

Ein vielfach genutztes Protokoll, welches auf einer seriellen Kommunikation basiert ist RS-232. Hier erfolgt die Kommunikation "wortweise", d.h. je nach Konfiguration zw. 5 bis 9 Bits pro Wort (meist 7 oder 8).



Ruhezustand: logische 1  
Beginn der Übertragung: logische 0  
MSB oder LSB (most significant oder least sigificant): logische 1  
Datenbits: 0 oder 1  
Paritätsbit: 1 wenn ungerade, sonst 0 (Prüfziffer)  
Stopbit: 1  
Rugezustand: logische 1

Der RS-232 Standard spezifiziert nicht wie die eigentlichen Daten übertragen werden.  
Kommunikationsparameter wie Datenrate (baud rate), Zeichenkodierung, Datenkompression, Fehlererkennung und Übertragungsprotokoll müssen vor der Datenübertragung eingestellt bzw. von Sende- und Empfangsgerät ausgehandelt werden.

RS-232 ist ein Beispiel für ISO/OSI Schicht 1.

Die Baudrate gibt an, wie oft ein Signal in einem Kommunikationskanal den Zustand ändert. Sind wie bei RS.232 nur zwei Zustände (0 oder 1) möglich, sind die Bitrate und Baudrate identisch.

|  |  |
| --- | --- |
| Bitrate (Bit/s) | Bitdauer |
| 50 | 20 ms |
| … |  |
| 9600 | 104 µs |
| … |  |
| 38400 | 26 µs |
| … |  |
| 115 200 | 8,68 µs |
| … |  |
| 500 000 | 2 µs |

Wichtige Funktionen:

* pinMode(pin, mdoe)
* digitalRead(pin)  
  Liefert 0 (LOW) oder 1 (HIGH) als Rückgabewert
* analogRead(pin)  
  Liefert den Analogwert des Pins in der jeweiligen Auflösung des Analog-Digital-Wandlers (Arduino UNO 10bit) 🡪 0 bis 1023
* digitalWrite(pin, value)  
  HIGH or LOW
* analogWrite  
  Nur sinnvoll auf einem Digital Pin mit PWM (Pulsweitenmodulation, Zeichen \*). Pin: 3,5,6,9,10,1  
  pin: pin-Nummer  
  value: duty cycle: (0-255)

Pulsweitenmodulation: Grundsätzlich kann ein digitaler Ausgang nur zwischen 5V (High) und 0V (Low) hin- und herschalten, beherrscht jedoch keine Zwischenwerte. Allerdings können diese Ausgänge sehr schnell hin- und herschalten. Geschieht dies schnell genug, kann das menschliche Auge dies nicht mehr wahrnehmen. Arduino arbeitet mit 1000 Hertz. Somit können durch Anpassen des Tastverhältnisses "Zwischenwerte" zwischen 5V und 0V verwirklicht werden.

## 4.3 Zeichenketten (strings)

Ursprünglich mussten Zeichenketten immer als Array vom Typ char angelegt werden. Mittlerweile gibt es die Klasse String, welche einen komfortableren Umgang damit erlaubt – die Zeichenketten werden dabei zu Objekten.

## 4.4 Interrupts

Die Aufgabe eines Interrupts ist sicherzustellen, dass der Prozessor schnell auf wichtige Ereignisse reagiert. Wenn ein bestimmtes Signal erkannt wird, dann unterbricht (wie der Name andeutet) ein Interrupt was auch immer der Prozessor tut und führt Code aus. Sobald der Code ausgeführt wurde, geht der Prozessor auf den ursprünglichen Punkt zurück und fährt dort fort, als wäre nichts geschehen.

In unserem Beispiel "Button\_ohne\_Interrupt" wird in loop () fortlaufend der Wert des Buttons ausgelesen. In dieser Zeit kann jedoch nichts anderes auf dem Arduino gemacht werden. Anstatt den Pin die ganze Zeit zu überwachen, können wir die Arbeit den Pin zu überwachen einem Interrupt übergeben und die loop() freimachen (siehe Beispiel "Button\_mit\_Interrupt").

Beachte: Die Variable buttonState wurde als volatile gekennzeichnet. Volatile ist ein C-Schlüsselwort, das auf Variablen angewendet wird. Das bedeutet, dass der Wert dieser Variable nicht komplett unter Programmkontrolle ist (z.B.: Benutzereingabe). Im Detail zeigt volatile dem Computer, dass er die Variable aus dem RAM und nicht aus dem Speicherregister laden soll. Eine Variable sollte als volatile deklariert werden, wenn sie von anders geändert werden kann außer in dem Codeteil in dem sie auftaucht (z.B.: Interrupt).

attachInterrupt meldet einen Interrupt an.  
Die Funktion digitalPinToInterrupt ermittelt die korrekte Interrupt-Id. Diese ist je nach Arduino Modell unterschiedlich.  
Es kann auf verschiedene Pin-Werte, bzw. Pin-Änderungen reagiert werden. Mögliche Argumente:

* CHANGE (wenn sich Wert ändert)
* LOW
* HIGH
* RISING (wenn der Pin von LOW auf HIGH wechselt)
* FALLING (wenn der Pin von HIGH auf LOW wechselt)

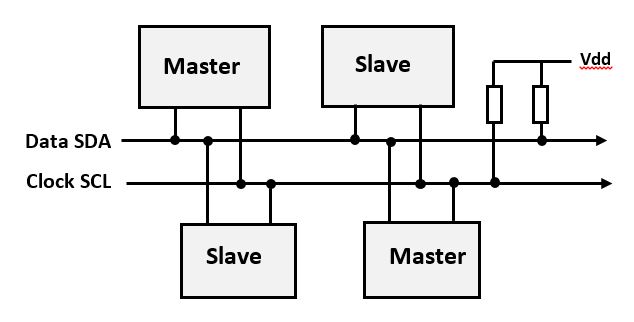
## 4.5 Handshake

In der Informatik ist ein Kommunikationsprotokoll eine Vereinbarung, nach der die Datenübertragung zwischen zwei oder mehreren Parteien abläuft.

## 4.6 I²C

Aufgrund des fehlenden Taktsignals ist RS-232 besonders bei höheren Übertragungsgeschwindigkeiten oder langen Leistungswegen anfällig für Störungen und daher nicht besonders robust. Daher entwickelte Philips Anfang der 1980er Jahren die I²C (I-Quadrat-C).

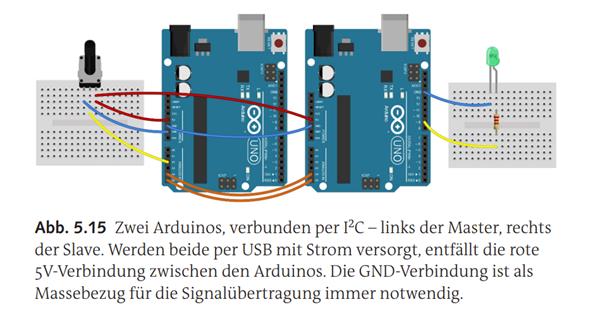
Im Gegensatz zur seriellen Verbindung, welche stets nur zwei Kommunikationspartner verbindet, handelt es sich hierbei um einen Datenbus, welcher optional mehrere Komponenten miteinander verbinden kann.



SCL – Serial Clock

SDA – Serial Data

Elektrisch gesehen besteht der Bus aus lediglich zwei Leitungen, an die alle Busteilnehmer angeschlossen sind, sowie einem gemeinsamen Masse Bezug. Eine der Leitungen dient als Taktsignal (Serial Clock – SCL), die andere wird zum Datenaustauch (Serial Date – SDA) verwendet. Dabei fungiert stets ein Teilnehmer als Master, er generiert das Taktsignal und Koordiniert die Kommunikation. Alle anderen Komponenten agieren als Slaves. Jeder Slave benötigt eine eindeutige Adresse (Zahl), über die er vom Master angesprochen werden kann.

Der Master kann den Slaves jederzeit Daten senden. Die einzelnen Slaves dürfen jedoch nur senden, wenn sie vom Master abgefragt werden, also eine Sendeerlaubnis erhalten. Dadurch wird vermieden, dass mehrere Komponenten auf die einzige vorhandene Datenleitung senden, denn dann wäre das Signal gestört.

Arduino hat keine separaten Pins für I²C, daher werden standardmäßig am Arduino UNO A4 als SDA undA5 als SCL verwendet.

Empfang der Daten wird über Interrupts realisiert (onReceive). Der Parameter "Anzahl" liefert die Anzahl an empfangenen Bytes.

Wenn der Master Daten von einem Slave "anfordert", geschieht dies mit der Methode requestForm. Der Slave soll sich auf onRequest anmelden, damit bei einer Datenanforderung ein Interrupt ausgelöst wird.