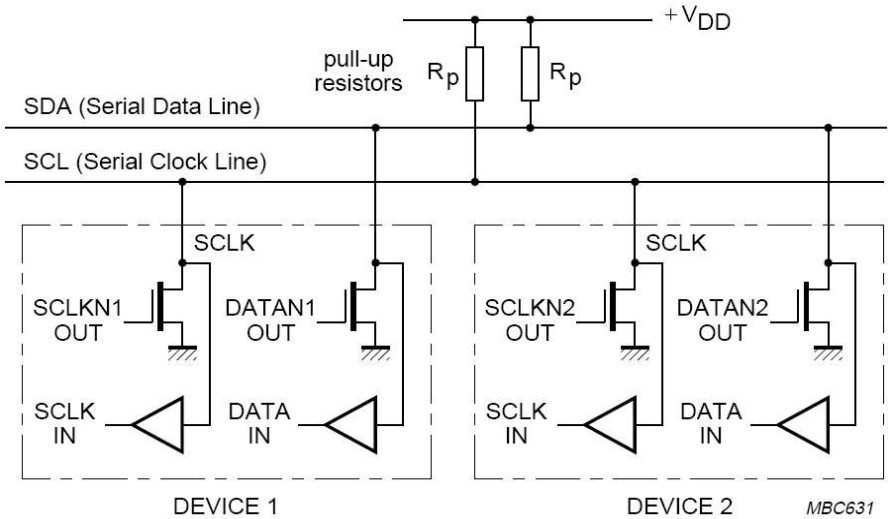
Der I²C-Bus(Inter Integrated Circuit-Bus) wurde 1995 von Phillips entwickelt. Wie der Name schon sagt wird er vorallem für das verbinden von Chips auf einer Platine verwendet.
Er wird in vielen Consumer Produkten wie Fernsehern, Kameras und Stereoanlagen verwendet.

Eigenschaften

- Seriell
  Dadurch ist er langsamer als ein Paralleler Bus, jedoch werden weniger Leiterbahnen(2) und Pins bei den diversen Devices benötigt, wodurch die Kosten reduziert werden.
- Datentransferraten
  Es gibt 3 verschiedene Modes
  - Normal Mode: 100kbit/sec => 12,5kByte/sec
  - Fast Mode: 400kbit/sec => 50kByte/sec
  - High Speed Mode: 3,2Mbit/sec => 400kByte/sec

Anschluss

Die Chips werden in and die Leitungen angeschlossen, die wiederum mit Pull-up Widerständen an VDD hängen.



Wired-AND-Schaltung(wenn Hardwaremäßig implementiert)

Jedes I²C Device muss 2 Anschlusspins für die 2 Leitungen verwenden die bidirektional arbeiten.

- SCL – Serial Clock Line
  Gibt regelmäßige Impulse aus -> Takt
- SDL – Serial Data Line
  überträgt Datenbits

Konzept

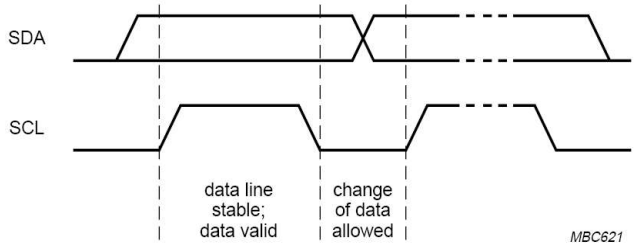
Devices die am Bus hängen sind entweder

- Master
  - beginnt Datentransfer
  - legt fest ob Lesen oder Schreiben
  - gibt Clock-Impulse auf SCL
- oder Slave
  - vom Master angesprochenes Gerät

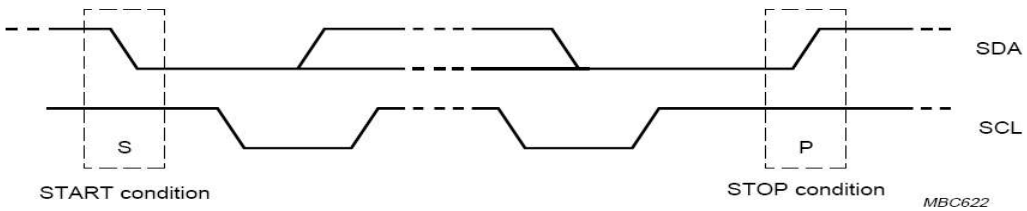
Beide haben die möglichkeit zu Senden und zu Empfangen, jedoch legt der Master dies für die Slaves fest.
Es gibt auch Devices die Slave oder Master sein können, wie zum Beispiel diverse Microcontroller.

Datenübertragung

Bitlevel



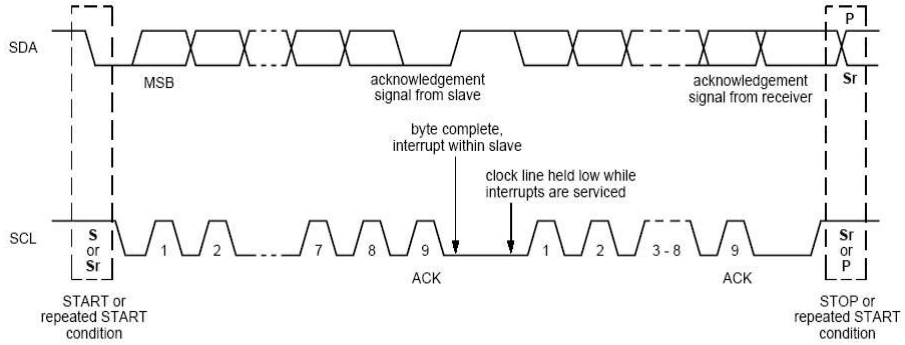
- Daten sind gültig wenn SCL high und SDA konstant.
- Während SCL low ist, darf SDA wechseln



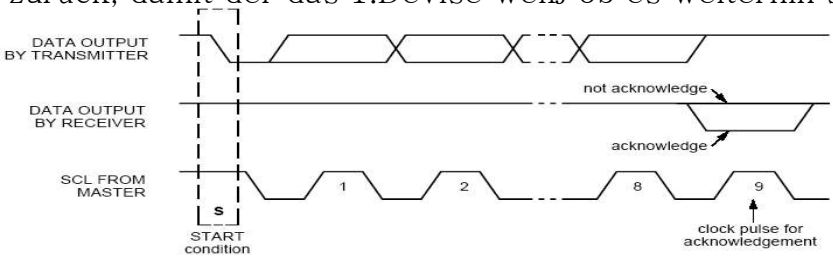
- Wenn SCL high ist und sich
  - SDA fallende Flanke --> Start Condition
  - SDA steigende Flanke --> Stop Condition

Ein Datetransfer beginnt normalerweise mit einer Start-Condition und endet mit einer Stop-Condition. Ein Sonderfall ist die sogenannte Repeated-Start-Condition , bei ihr wird statt der Stop-Condition eine erneute Start-Condition gesendet. Somit kann der Master weiterhin die Leitung besetzen und kann so Zeit einsparen.

Bytelevel



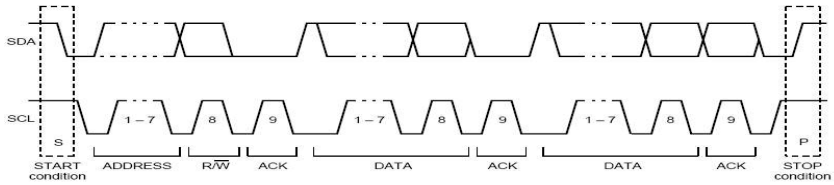
Nach einer Start oder Repeated-Start-Condition wird immer Bytewise (Most Significant Bit zuerst) übertragen. Nach jedem Byte gibt das 2. Device ein ACK oder NACH zurück, damit der das 1.Devise weiß ob es weiterhin senden soll.



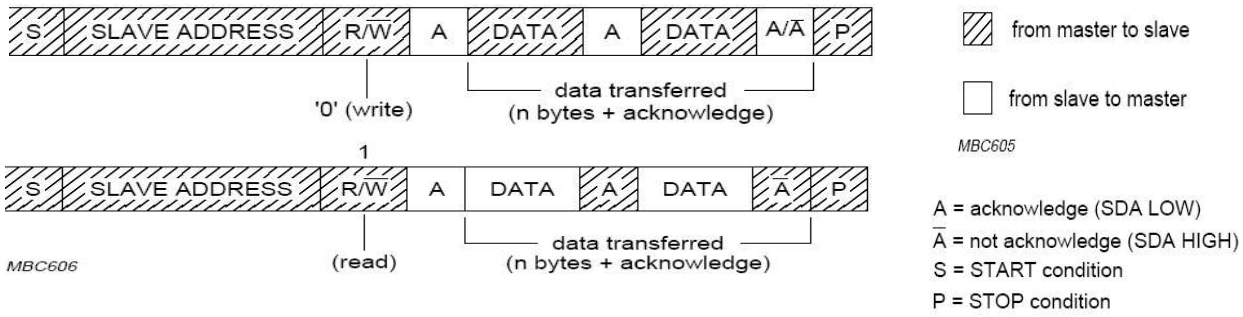
Eine Besonderheit ist das Clockstretching für langsamere Devices die durch das "runterziehn" der SCL die Abstände zwischen den einzelnen Clocktakten verlängern können.

Adressierung der Slaves

7-Bit



Die ersten 7 Bits des 1 Byte nach der Start-Condition sind die Adresse des Devices.



Je nach dem ob gelesen oder geschrieben wird ändert sich wer auf den Bus Daten ausgibt und wer das Acknowledge-Bit sendet. Es gibt schon diverse Adressen die für bestimmte Zwecke vorbelegt sind. (s. I²C Skriptum von Philips)

10-Bit

Eine dieser vorbelegten Adressen wird für die 10-Bit-Adressierung verwendet. Durch diese Vorbelegung wird auch eine gleichzeitige Verwendung mit 7-Bit-Adressierung verwendet.

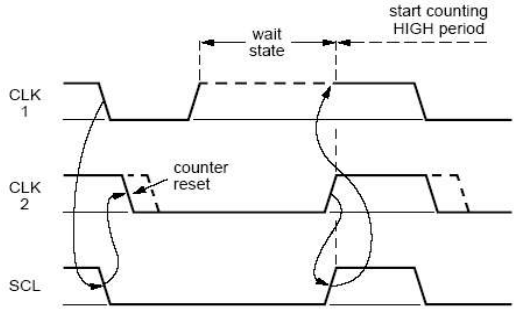
Die Adresse besteht somit aus 2 Bytes

- 1. Byte:
  - 1-5 Bit --> 11110
  - 6,7 Bit --> ersten 2 Bit der Adresse
  - 8 Bit --> lesen/schreiben
- 2. Byte:
  - 1-8 Bit --> 3-10 Bit der Adresse

Multimaster

Clock-Synchronisation

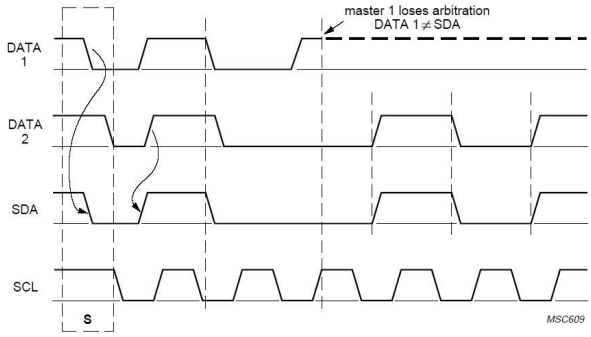
-> ermöglicht das abgleichen mehreren Master auf diesselben Clock-Impulse



- CLK1 wird low
- dadurch wird SCL low und CLK1 low
- wenn CLK1 wieder high will, geht das nicht weil CLK1 noch runterzieht
- wenn CLK2 auf high, wird die SCL wieder losgelassen
- geschieht nach jedem Takt

Arbitrierung

-> teilt den Bus den Mastern zu (es wird CSMA/CD verwendet)



1.Abschnitt:
D1 vergleicht ob seine Ausgabe gleich SDA
2.Abschnitt:
D2 vergleicht ob seine Ausgabe gleich Sda
3.Abschnitt:
D1 versucht lässt die Leitung los um auf High zu kommen, D2 zieht jedoch weiter runter, dadurch verliert D1 die Leitung und darf den Bus weiter verwenden.

Beispielfunktionen für PIC16F877 als Master

Es werden 4k7 als Pullupwiderstände verwendet.
Stromversorgung über PIC16F877 mit 5V.

```
/* ******************************************************  */
/* Apoloner, Titz                                         */
/* 4AHDVN 10.12.03                                        */
/* ****************************************************** */
/* Ansprechen über I2C von 2 24LC512 EEPROm               */
/* und 1 DS1337 Real Time Clock                           */
/* ****************************************************** */
/* Adressen(Adressbytes für lesen):                       */
/*      EEPROM 1 = 0xa0                                   */
/*      EEPROM 2 = 0xa2                                   */
/*      RTC  = 0xd0                                       */
/* ****************************************************** */

// Pic ist Master(hardwareunterstützt)
#use I2C(MASTER, SDA=PIN_C4, SCL=PIN_C3)

//Funktionen für EEPROM

// schreiben auf eeprom
// addrl = lower Adressbyte, addrh = higher Adressbyte,
// data = 1.Datenbyte, control = Adresse des Devices(EEPROM)
void writebyte (byte addrl, byte addrh, byte data, byte control)
{
      byte ack;

      i2c_start();
      // Start-Condition - clock leitung wird auf low gehalten bis
      // was geschrieben wird

      i2c_write(control);
      // 1010 = control für den  chip
      // z.B.: 0000 =  3 Adressbits + 1=read/0=write

      i2c_write(addrh);
      // Addresse sind 2 byte, man darf aber immer nur 1 schicken

      i2c_write(addrl);
      // 2.Byte der Adresse

      ack = i2c_write(data);
      // Schreiben der Daten

      // Prüft auf ACK bzw NACK
      if (ack != 0)
            printf ("\n\rFehler!");

      i2c_stop();
      //Stop-Condition

      delay_ms(10);
}
```

```
// lesen von eeprom
// Parameter s. writebyte
byte readbyte (byte addrl, byte addrh, byte control)
{
      byte read;
      // gelesenes Byte

      i2c_start();
      // Start-Condition

      i2c_write(control);
      // Chip merkt sich was als letztes angesprochen wurde

      i2c_write(addrh);
      // Addresse sind 2 byte, man darf aber immer nur 1 schicken
      i2c_write(addrl);
      // 2.Byte der Adresse

      i2c_start();
      // Repeated Start-Condition

      i2c_write(control+1);
      // letzes bit wird von read auf write geändert
      read = i2c_read(0);
      // 0 = NACK -> Ende des Datentransfers

      i2c_stop();
      // Stop-Condition

      delay_ms(10);
      // kurze Pause, kann noch kürzer sein

      return read;
}
```

```
// Funktionen für die RTC

void setRTC(byte minute, byte second,byte addr)
{
      // umrechnen der Werte, in das Format, welches die RTC           //
verwendet, geht nur so für Sekunden und Minuten
      writeRTC(addr, 1, ((minute/10)<<4)+(minute%10));
      writeRTC(addr, 0, ((second/10)<<4)+(second%10));
}

// lesen von Minuten aus RTC
byte getMinute(byte addr)
{
      int minute;
      minute = readRTC(addr, 1);
      minute = ((minute&0x0F)+(minute>>4)*10);
      return minute;
}

//lesen von Sekunden aus RTC
byte getSecond(byte addr)
{
      int second;
      second = readRTC(addr, 0);
      second = ((second&0x0F)+(second>>4)*10);
      return second;
}

// schreiben eines Bytes in RTC
// control = Adresse der RTC (vorgegeben), adr = interne Adresse,
// data = Datenbyte
void writeRTC(byte control, byte adr, byte data)
{
      i2c_start();
      i2c_write(control);
      i2c_write(adr);
      i2c_write(data);
      i2c_stop();
      delay_ms(11);
}

// lesen eines Bytes von RTC, Parameter s. writeRTC
byte readRTC(byte control, byte adr)
{
      byte data;
      i2c_start();
      i2c_write(control);
      i2c_write(adr);
      i2c_start();
      i2c_write(control+1);
      data = i2c_read(0);
      i2c_stop();
      return data;
}
```