

AINF Ausarbeitung Korb 6

Multithreading

Wenn man ein Programm mit nur einen Thread schreibt, würde bei allen Aktionen, welche mehr als ~ 100ms benötigen, das Programm für den Benutzer hängen (Application not responding == schlecht). Daher müssen solche Aktionen in einen Hintergrundthread verlagert werden, damit die Benutzeroberfläche weiterhin bedienbar bleibt. Wie können mehrere Threads auf einem Prozessor ausgeführt werden? Beispiel Philosophenproblem: Alle wollen gleichzeitig essen, es gibt aber nur eine Gabel am Tisch. Also wird die Gabel immer reihum gegeben. Wenn man das schnell genug macht, sieht es aus, als ob alle gleichzeitig essen, dabei isst immer nur einer. So macht es auch der Thread Scheduler im Betriebssystem. Multithreading ist eines der schwersten Gebiete in der Programmierung.

Thread

Die Klasse Thread ist die einfachste Möglichkeit, Multithreading in Java zu betreiben. Sie implementiert das Interface Runnable. Mit dieser Möglichkeit hat man weder die Möglichkeit, Parameter zu übergeben, noch Rückgabewerte zu erhalten. Die Klasse Thread verfügt über die Methoden start, stop, interrupt und isInterrupted. Mit start wird der Thread von außen gestartet. Mit stop könnte man einen Thread auf OS-Ebene stoppen. Dies ist jedoch nicht zu empfehlen, da er quasi gekillt wird. Daher ist diese Methode auch deprecated und wird demnächst entfernt werden. Man sollte innerhalb des Threads die Methode isInterrupted zyklisch auf true abfragen und dann die Ausführung der Hintergrundaufgabe beenden. Diese Methode liefert true, wenn die Methode interrupt von außen aufgerufen wird. Hier ein Codebeispiel:

```
1. package mat;
2.
3. public class MyThread extends Thread
4. {
5.     @Override
6.     public void run()
7.     {
8.         int cnt = 0;
9.         while (!isInterrupted())
10.        {
11.            System.out.format("Runde: %d%n", ++cnt);
12.            try
13.            {
14.                Thread.sleep(100);
15.            }
16.            catch (InterruptedException ex)
17.            {
18.                return;
19.            }
20.        }
21.    }
22.
23. public static void main(String[] args)
24. {
25.     MyThread thread = new MyThread();
26.     thread.start();
27.     try
28.     {
29.         Thread.sleep(1000);
30.     } catch (InterruptedException ignore) {}
31.     thread.interrupt();
32. }
33. }
```

Executors

Executors sind eine Gruppe von Klassen welche nicht nur Runnables, sondern auch Callables verarbeiten können. Daher kann komfortabel ein Ergebnis in Form eines Objekts zurückgeliefert werden. Grob kann man die Executors in ScheduledExecutorService und ExecutorService einteilen, wobei erstere ein getimtes Ausführen ermöglichen, dabei kann man jedoch nur Runnables und keine Callables verwenden. Erzeugt werden diese Executors mit Factory-Methoden, welche die Klasse Executors bereitstellt. Gestartet wird ein neuer Thread mit der Methode submit(), bzw. scheduleWithFixedDelay oder scheduleAtFixedRate beim ScheduledExecutorService. Dabei liefert die jeweilige Methode ein Future-Objekt zurück. Dieses ist das zukünftige Ergebnis, wenn ein Callable übergeben wurde. Über dieses Future-Objekt kann man die Ausführung auch beenden. Den ganzen ExecutorService kann man mit shutdown beenden.

```
1. package mat;
2.
3. import java.util.concurrent.*;
4.
5. public class Test
6. {
7.
8.     private ExecutorService exe = Executors.newSingleThreadExecutor();
9.     private Future ft;
10.
11.    public void start()
12.    {
13.        ft = exe.submit(new Callable<String>()
14.        {
15.            @Override
16.            public String call() throws Exception
17.            {
18.                for (int i = 0; i <= 100000; i++);
19.                return "Huhu";
20.            }
21.        });
22.    }
23.
24.    public String check() throws InterruptedException, ExecutionException
25.    {
26.        if (ft.isDone())
27.            return (String) ft.get();
28.        else
29.            return "Not ready yet";
30.    }
31.
32.    public void stop()
33.    {
34.        ft.cancel(true);
35.        exe.shutdown();
36.    }
37.
38.    public static void main(String[] args)
39.    {
40.        try
41.        {
42.            Test t = new Test();
43.            t.start();
44.            Thread.sleep(1000);
45.        }
46.    }
47. }
```

```
45.     System.out.println(t.check());
46.     t.stop();
47. } catch (Exception ex)
48. {
49.     ex.printStackTrace();
50. }
51. }
52. }
```

ThreadSave

ThreadSave heißt, dass auf eine Methode immer nur ein Thread zugreifen darf. Der einfachste Weg, dies zu erreichen, ist die Methode synchronized zu machen. Der Kopf wird dann einfach um das Schlüsselwort synchronized ergänzt, z.B: public static synchronized int add(int... num). Dann darf immer nur ein Thread diese Methode verwenden. Dies kann jedoch speziell bei langen Methoden nicht zielführend sein, da die Performance darunter leidet. Man kann auch den Zugriff auf ein Codesegment für eine gewisse Zahl an Prozessen sperren, dafür verwendet man die Klasse Semaphore. Wird als Maximum beim Semaphore 1 eingestellt, spricht man auch von Mutex. Nachfolgend ein Codebeispiel aus dem Internet.

```
1. package mat;
2.
3. import java.util.concurrent.Semaphore;
4.
5. public class SemaphoreTest
6. {
7.     static Semaphore semaphore = new Semaphore(1);
8.
9.     static class MyLockerThread extends Thread
10.    {
11.        String name = "";
12.
13.        MyLockerThread(String name)
14.        {
15.            this.name = name;
16.        }
17.
18.        public void run()
19.        {
20.            try
21.            {
22.                System.out.println(name + " : acquiring lock...");
23.                System.out.println(name + " : available Semaphore permits now: "
24.                    + semaphore.availablePermits());
25.                semaphore.acquire();
26.                System.out.println(name + " : got the permit!");
27.
28.            try
29.            {
30.                for (int i = 1; i <= 5; i++)
31.                {
32.                    System.out.println(name + " : is performing operation " + i
33.                        + ", available Semaphore permits : "
34.                        + semaphore.availablePermits());
35.                    // sleep 1 second
36.                    Thread.sleep(1000);
37.                }
38.            }
```

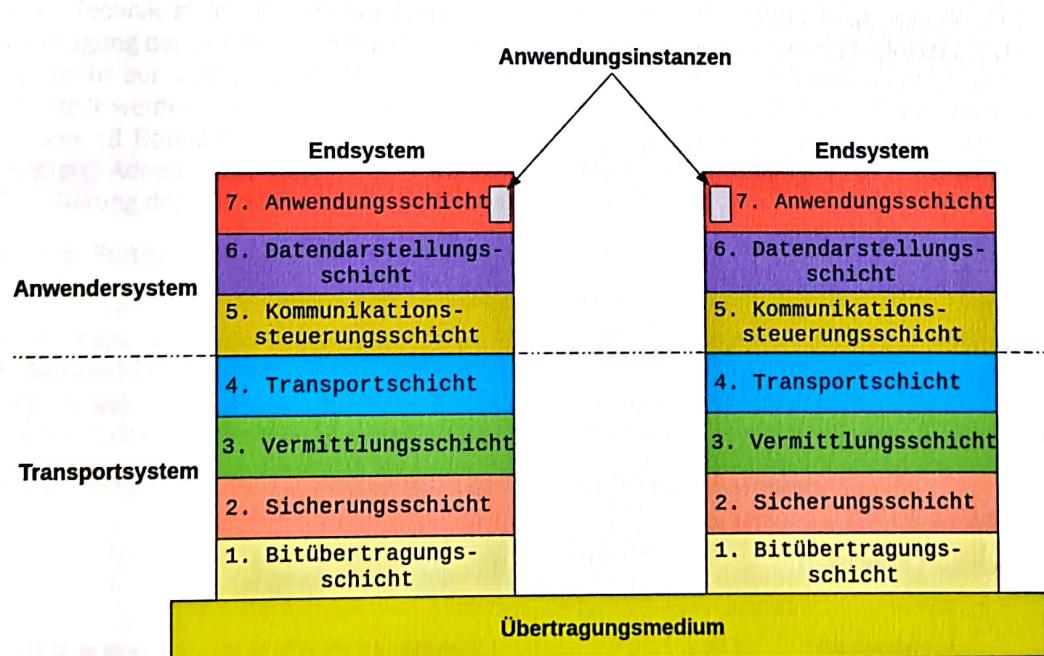
```

39.     finally
40.     {
41.         // calling release() after a successful acquire()
42.         System.out.println(name + " : releasing lock...");
43.         semaphore.release();
44.         System.out.println(name + " : available Semaphore permits now: "
45.             + semaphore.availablePermits());
46.     }
47. } catch (InterruptedException e)
48. {
49.     e.printStackTrace();
50. }
51. }
52. }
53.
54. public static void main(String[] args)
55. {
56.     System.out.println("Total available Semaphore permits : "
57.         + semaphore.availablePermits());
58.     MyLockerThread t1 = new MyLockerThread("A");
59.     t1.start();
60. }
61. }

```

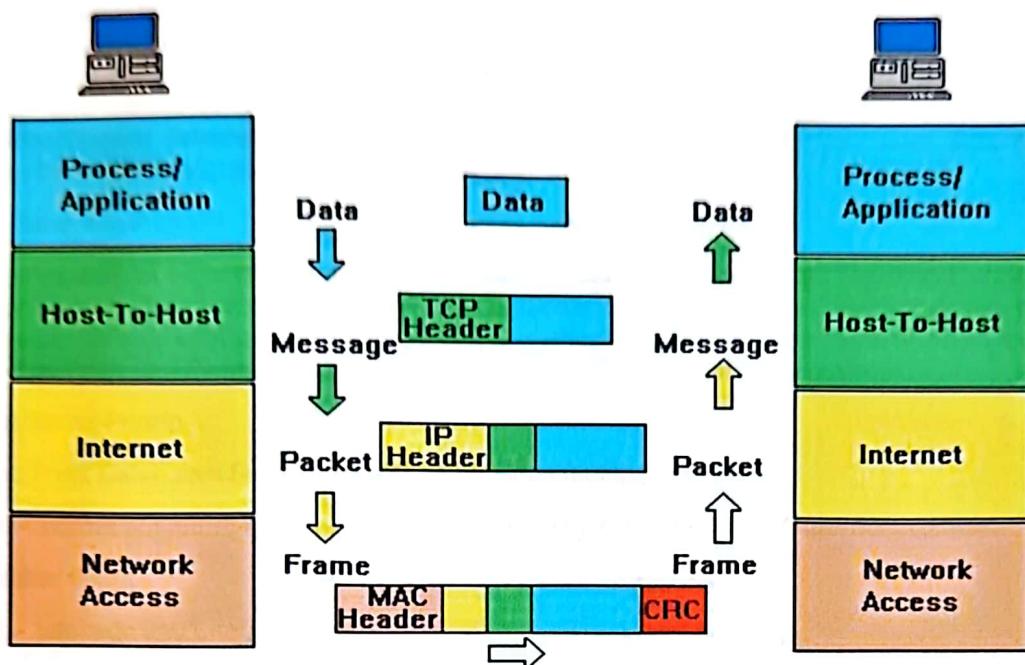
Networking

ISO-OSI-Modell:



Entsprechende Technologien: Layer 1 und 2: Ethernet, Layer 3: IP, Layer 4: TCP, Layer 5-7: OS und Programm;

So setzt sich ein fertiges Ethernet-Frame zusammen: Mit jeder Schicht kommen Daten zu den eigentlichen Nutzdaten dazu.



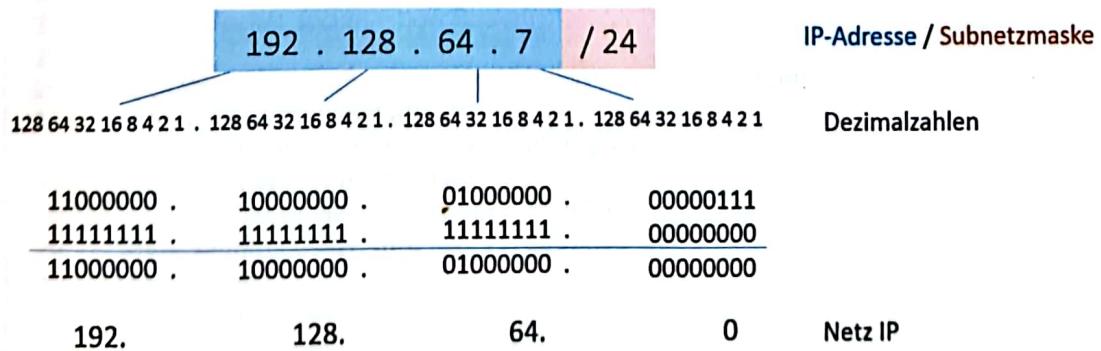
Ethernet

Diese Technik stellt die unterste Ebene im OSI-Modell dar. Sie regelt alles, was die Physische Übertragung der Daten betrifft (im LAN). Es stehen Geschwindigkeiten von 10 Mbit/s bis 10 Gbit/s und mehr zur Verfügung. Dafür gibt es Kabel, welche in verschiedene Kategorien (zB CAT 6) eingeteilt werden. Bei einer 1 Gbit Verbindung können bis zu 100 MB pro Sekunde übertragen werden. (8 Datenbits + 1 Startbit + 1 Stopbit). Zur Adressierung werden MAC (Media Access Control) -Adressen verwendet. Dieses Protokoll ist nicht routingfähig und daher wäre die Realisierung des Internets nur mit Ethernet nicht möglich.

Internet Protocol

IP stellt Layer 3 dar. Es arbeitet mit sogenannten IP-Adressen. Dieses Protokoll ist routingfähig, das heißt, dass sich die Pakete selbst ihren Weg durch das Netzwerk zum Zielrechner suchen. Außerdem können Teilnehmer in Subnetze unterteilt werden. Es gibt IPv4 und IPv6. Eine v4 Adresse besteht aus 32 Bit. Daher können mit IPv4 maximal etwas über 4 Milliarden Geräte verbunden werden. Das ist mittlerweile zu wenig, daher wurde IPv6 eingeführt, es verwendet 128 Bit Adressen.

Rechenbeispiel IP-Adresse, Netzadresse, Subnetzmaske, Geräteadresse:



Die Subnetzmaske kann entweder gleich wie die IP-Adresse als $255.255.255.0$ oder verkürzt als Anzahl der 1er mit /24 angegeben werden.

Netzanteil

Transmission Control Protocol

TCP ist ein verbindungsorientiertes und paketvermitteltes Protokoll. Das heißt dass zwischen den Kommunikationspartnern eine Verbindung aufgebaut wird. Es arbeitet mit Ports, wobei 65535 (16Bit) verschiedene Ports auf jeder Netzwerkkarte zur Verfügung stehen. Die Kombination aus Port und IP-Adresse heißt Socket. Bsp. für Socket: 192.168.0.101:443; Das Protokoll sorgt dafür, dass wirklich alle Daten zuverlässig übertragen werden. Ist zum Beispiel von 2000 Paketen eines defekt, wird dieses im Nachhinein noch einmal angefordert. Diese Verhalten ist bei einer Dateiübertragung beispielsweise wichtig, jedoch bei Anwendungen wie VoIP oder Streaming unbrauchbar. Dafür gibt es UDP.

User Datagram Protocol

Diese Protokoll ist verbindungslos und sendet die Daten einfach weg an den Empfänger, ohne zu überprüfen, ob diese vollständig ankommen. So können zwar Informationen verloren gehen, dies muss aber für eine Echtzeitübertragung zB bei Streamen in Kauf genommen werden. Es kommen ebenfalls Ports zum Einsatz, um die konkrete Empfängeranwendung zu adressieren.

Client-Server-Prinzip

Client: Fragt Daten vom Server ab, muss nicht immer laufen.

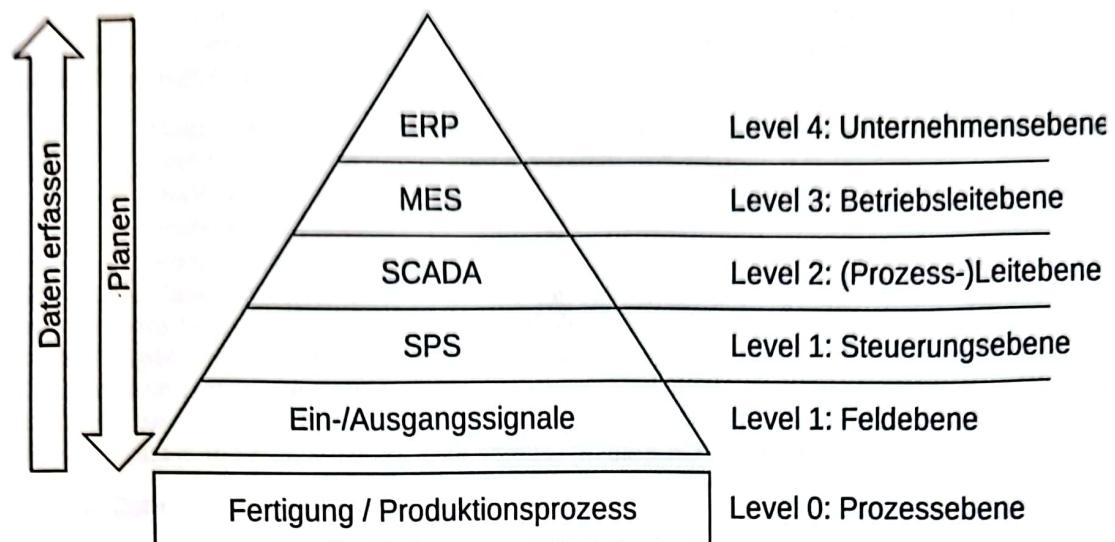
Server: Wartet auf Anfragen vom Client und behandelt diese; läuft in der Regel durchgehend;

Einfacher Client

```
1. package ue01.net;
2.
3. import java.io.*;
4. import java.net.*;
5.
6. public class SimpleClient
7. {
8.
9.     /**
10.      * Sends a request and receives the response
11.      *
12.      * @param host Hostname
13.      * @param port Port
14.      * @param request Message to send as request
15.      * @return The response that is sended to the server
16.      * @throws Exception Exception is thrown e.g. in case the Host is unreachable
17.     */
18.     public static String sendRequestAndRecieveResponse(String host, int port,
19.                                                       String request) throws Exception
20.     {
21.         final InetAddress adr = InetAddress.getByName(host);
22.         try (final Socket socket = new Socket())
23.         {
24.             SocketAddress endpoint = new InetSocketAddress(adr, port);
25.             socket.connect(endpoint);
26.             final BufferedWriter writer = new BufferedWriter(
27.                 new OutputStreamWriter(
28.                     socket.getOutputStream(), "utf8"));
29.             writer.write(request+"\n");
30.             writer.flush();
31.             socket.shutdownOutput();
32.             final BufferedReader reader = new BufferedReader(
33.                 new InputStreamReader(
34.                     socket.getInputStream(), "utf8"));
```

```
35.     String line;
36.     final StringBuilder builder = new StringBuilder();
37.     while ((line = reader.readLine()) != null)
38.         builder.append(line).append("\n");
39.     return builder.toString();
40. }
41.
42. }
43.
44. public static void main(String[] args)
45. {
46.     for (int i = 0; i < 4; i++)
47.         try
48.         {
49.             final String response
50.                 = sendRequestAndRecieveResponse(
51.                     "10.0.0.2", 8150, "SBL789 255 000 000");
52.             System.out.println("Anwort eingetroffen:");
53.             System.out.println(response);
54.             System.out.println("— A Traum —");
55.         }
56.         catch (Exception e)
57.         {
58.             System.out.println("— Ze fix —");
59.             e.printStackTrace();
60.         }
61.     }
62. }
```

Automatisierungspyramide (geregelt in IEC 62264)



Ebene	Eingesetzte Systeme	Typische Aufgaben
Unternehmensebene	ERP	Produktionsgroßplanung, Bestellabwicklung
Betriebsleitebene	MES, MIS, LIMS	Produktionsfeinplanung, Produktionsdatenerfassung, KPI-Ermittlung; Material-Management, Qualitätsmanagement
(Prozess)leitebene	Prozessleitsystem/HMI/SCADA	Bedienen und Beobachten, Rezeptverwaltung und Ausführung, Messwertarchivierung
Steuerungsebene	SPS	Steuerung, Regelung
Feldebene	Prozesssignale, /Ausgabemodule, Feldbus	Ein-Schnittstelle zum technischen Produktionsprozess über Ein- und Ausgangssignale
Sensor-/Aktorebene	Parallelverdrahtung oder intelligente Systeme wie z. B.: AS-Interface, IO-Link	Einfache und schnelle Datensammlung, meist binärer Signale

Modbus

Beim Modbus handelt es sich um einen offenen Feldbus, welcher 1979 von Gould-Modicon zur Kommunikation mit deren hauseigenen speicherprogrammierbaren Steuerungen vorgestellt wurde. Er verfügt über drei verschiedene Modi:

- Modbus ASCII: Hier kann immer ein ASCII-Zeichen nach dem anderen gesendet werden (rein textuell). Dies erfolgt über die serielle Schnittstelle (z.B. RS-485)
- Modbus RTU: Hier werden die Daten binär übertragen, ebenfalls über die serielle Schnittstelle.
- Modbus TCP: Hier werden TCP-Pakete über das TCP/IP-Protokoll versendet.

Ein Modbus-Paket ist verschieden zusammengestellt. Dies hängt von dem Modus ab. Die PDU, Protocol Data Unit, ist immer gleich und besteht aus dem Function Code, welcher die Art der Anfrage angibt und den eigentlichen Daten. Bei den Modi ASCII und RTU kommen noch Bereiche für Adresse und Prüfsumme hinzu, dann ist von einer ADU (Application Data Unit) die Rede. Bei TCP sind diese zusätzlichen Daten nicht erforderlich, das hier TCP/IP-Pakete versendet werden, welche bereits in sich eine Prüfsumme und eine Adresse (in diesem Fall ein Socket) haben.

Modbus-Daten-Modell

Das Modbus-Daten-Modell unterscheidet vier verschiedene Bereiche:

Name	Zweck	Beispiel
Discrete Inputs	ein einzelnes Bit, welches nur lesbar ist	Taster
Coils	les- und beschreibbares Bit	LED oder Spule (daher der Name)
Input Registers	lesbares Register (16-Bit-Wert)	ADC, Temperatursensor
Hold Registers	les- und beschreibbares Register (16-Bit)	DAC, Pulsweitenmodulatormodul

Funktionscodes

Function Codes (dt. Funktionscodes) werden verwendet, um die Art der Anfrage anzugeben. Hier ist ein Beispiel einiger genormter Codes:

Code	Zweck	Datenmenge
1	Coils lesen	1 Bit
2	Discrete Inputs lesen	1 Bit
3	Hold-Register lesen	16 Bit
4	Input-Register lesen	16 Bit
5	ein Coil beschreiben	1 Bit
6	ein Register beschreiben	16 Bit
15	mehrere Coils beschreiben	min. 1 Bit
16	mehrere Register beschreiben	min. 16 Bit

JNI / JNA

Java Native Interface wird verwendet, um Funktionalitäten zu realisieren, welche standardmäßig nicht in Java vorgesehen sind und auf Betriebssystemressourcen zugreifen müssen (z.B. serielle Schnittstelle). Dazu benötigt man Jar-Dateien auf der JavaVM-Seite und binäre Bibliotheken (kompilierte C-Funktionen, .dll bei Windows oder .so bei Linux) auf der Betriebssystemseite. Java Native Access heißt, dass man direkt auf die Betriebssystemfunktionen zugreift (keine binären Bibliotheken notwendig).

Serielle Schnittstelle (aus dem Steinerskript raubkopiert)

Grundprinzip

Die Abkürzung **UART** steht für **Universal Asynchronous Receiver and Transmitter**, also für eine universelle asynchrone Empfangs- und Sendeeinheit. Beim Atmega16/328P wird diese Einheit **USART** genannt, da man mit ihr auch synchron mit Hilfe einer Takteleitung kommunizieren kann.

Beim PC sprechen wir von der seriellen Schnittstelle nach EIA-232 (RS-232) mit ihrem typisch 9 poligen SUB-D Steckverbinder. Die Schnittstellen heißen unter Windows COM1, COM2, COM3, ... und unter Linux /dev/ttys0, /dev/ttys1, ... Heutige PCs und Laptops verfügen meist nur mehr über USB Schnittstellen. Mit Hilfe geeigneter Konverter und "Virtual Com Treiber" lässt sich die serielle **UART-Schnittstelle** auch über USB realisieren. Diese Schnittstellen heißen unter Windows COM1, COM2, COM3, ... und unter Linux /dev/ttys0, /dev/ttys1, ...

Grundschaltung

UART Grundbeschaltung

In der Minimalvariante wird für jede Datenrichtung eine Leitung benötigt. Am Pin TxD (Transmit x Data) werden digitale Spannungssignale generiert (zum Beispiel 5V=1, 0V=0), und beim Pin RxD (Receive x Data) werden die Spannungssignale empfangen und verarbeitet. Die gemeinsame elektrische Masse (GND) ist als Bezugspunkt für die elektrischen Spannungssignale erforderlich.

Aufbau eines Frames

UART Frame

Über die Datenleitung werden die Daten bitseriell übertragen. Das heißt, die Bits werden zeitlich hintereinander liegend einzeln übertragen. Wenn keine Daten übertragen werden hat die Datenleitung den Zustand **IDLE** (untätig). Der Sender startet die Übertragung mit dem Startbit (St). Anschließend folgen eine bestimmte Anzahl an Daten-Bits, beginnend mit dem LSB (Least Significant Bit), also dem Bit 0. Danach folgt eventuell (optional) ein Parity-Bit (P). Abgeschlossen wird die Übertragung mit einem oder zwei Stop-Bits (Sp). Start-Bits haben immer den Wert 0, Stop-Bits immer den Wert 1. Das stellt sicher, dass jeder Frame-Beginn vom Empfänger sicher erkannt werden kann (durch die Signalflanke 1->0 am Beginn). Anschließend kann der nächste Frame folgen, oder die Leitung auch wieder den Zustand **IDLE** annehmen. Beim Atmega16/328P können 5, 6, 7, 8 oder 9 Daten-Bits übertragen werden. Üblicherweise werden heute 8 Daten-Bits (Bytes) verwendet. Das optionale Parity-Bit ermöglicht eine einfache Datensicherung. Möglich ist ein Even- oder eine Odd-Parity. Bei der Even-Parity bekommt das Bit P jenen Wert 0 oder 1, wodurch die Anzahl der 1er (Daten-Bits + Parity) geradzahlig ist. Bei einer Odd-Parity ist die Anzahl der 1er ungeradzahlig. Die Bitübertragungsgeschwindigkeit (Baud-Rate), die Anzahl der Daten-Bits (5, 6, 7, 8 oder 9), die Parity Konfiguration (none, even oder odd) muss beim Sender und Empfänger gleich eingestellt werden. Die Anzahl der Stop-Bits (1, 1.5 oder 2) kann auch unterschiedlich konfiguriert sein. Zu beachten ist, dass nicht immer jede beliebige Bitrate (Baud-Rate) möglich ist. Beim PC oder Laptop sind aufgrund der Einschränkungen im 8250 UART oft nur die folgenden Geschwindigkeiten möglich (angegeben in bps = Bits per Second):

50, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200

Unser Sure AVR DEM2 Board sowie unser Asuro verwendet die Geschwindigkeit 57600 bps. Die Konfiguration wird oftmals in Form eines Kurzcode, bestehend Geschwindigkeit + "/" Daten-Bits + Parity-Typ (N, E oder O) + Stop-Bits angegeben.

Beispiel Asuro: 57600/8N1 ... 57600bps, 8 Data-Bits, No Parity Bit, 1 Stop-Bit

UART Schnittstellen verfügen im Gegensatz zum USB über keine Hot-Plug Capability. Das heißt, die Kommunikationsparameter Geschwindigkeit, Anzahl der Daten-Bits, Parity-Bit und Anzahl der Stop-Bits müssen beim Sender und Empfänger richtig eingestellt werden. Bei einer reinen Textübertragung erscheinen bei einer fehlerhaften Konfiguration meist nur wirre Zeichen am Bildschirm.

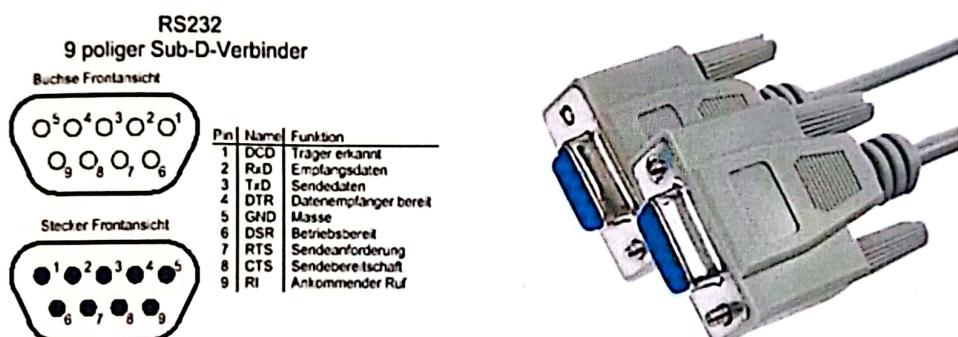
RS-232 (EIA-232)

Die RS-232 Schnittstelle (RS = Recommended Standard, EIA-232, EIA = Electronic Industries Alliance) existiert seit 1960, war lange Zeit die am häufigsten eingesetzte serielle Schnittstelle (zum Beispiel beim PC).

Sie wurde durch USB abgelöst und ist heute nur mehr in Ausnahmefällen zu finden.

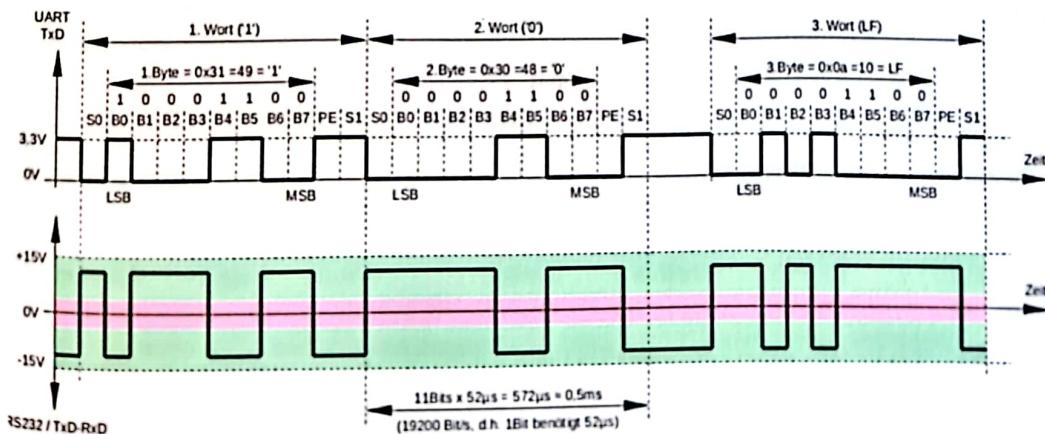
Eigenschaften der RS-232 Schnittstelle sind:

- Übertragung von Wörtern (7 Bit, 8 Bit = 1 Byte, 9 Bit)
- Spannungsschnittstelle (-15V bis +15V)
- 9- und 25-polige D-Sub Stecker (als Stecker oder Buchse)
- Software- und Hardware-Handshake zur Datenflusssteuerung möglich



Signalpegel

Asymmetrische Signale (zum Beispiel 0V und 3,3V) sind für die Übertragung über größere Distanzen ungeeignet. Bei der RS-232 werden um 0V symmetrische Spannungen mit hohen Signalpegeln (typisch $\pm 12V$) generiert. Eine pseudodifferentielle Signalübertragung (Signale werden zur Masse gespiegelt) soll die Fehlerrate reduzieren. Dieses aus den 1960er Jahren stammende Prinzip hat allerdings auch Nachteile, es verbraucht viel Leistung und ist für hohe Übertragungsgeschwindigkeiten im MBit/s Bereich nicht geeignet. Bei der Datenleitung TxD können 0er-Bits einen Spannungswert zwischen +3V und +15V aufweisen. 1er-Bitsweisen einen Spannungswert zwischen -3V und -15V auf. Bei Steuerleitungen ist die Zuordnung umgekehrt (0er Bits haben dort eine negative Spannung).



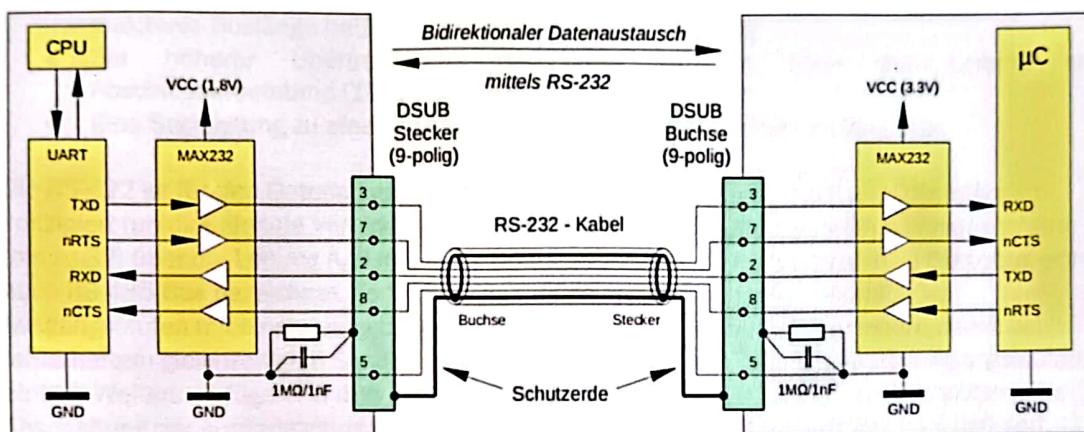
Im oberen Bild ist die Übertragung von 3 Bytes zu sehen. Die Bytes sind nach ASCII codiert. Die ersten beiden Bytes stellen den Text "10" dar. Das 3. Byte repräsentiert das ASCII-Steuerzeichen LF, den Zellenvorschub (LF = Line-Feed = 0x0a = 10). Es werden also drei Zahlen mit dem Wert 49, 48 und 10 übertragen. Die Zahlen werden dual codiert (49=0x31=00110001b, 48=0x30=00110000b, 10=0x0a=00001010b) bitseriell über die Leitung gesendet. Es werden immer die niederwertigen Bits zuerst gesendet, also LSB (B0) zuerst und MSB (B7) zuletzt. Im Ruhezustand weist das TxD/RxD Signal den Spannungspegel von "Logisch 1", also eine Spannung zwischen -3V bis -15V, auf. Das Even-Parity Bit (PE) sorgt dafür, dass die Anzahl der 1er (Daten-Bits B0 bis B7 und Parity-Bit PE) immer gerade - also durch 2 teilbar - ist. Der Empfänger kann das Parity-Bit überprüfen und auf diese Weise einzelne Bitfehler erkennen. Mehrfache Bitfehler sind damit natürlich nicht immer erkennbar.

Pegelwandler

Für die Pegel-Wandlung der digitalen Signale (zum Beispiel von CMOS 0V/3,3V auf RS-232 Pegel -15V/+15V) sind kostengünstige Bausteine mit eingebautem DC/DC Spannungswandler verfügbar, zum Beispiel der MAX232 von der Firma MAXIM.

Aufbau

Das folgende Bild zeigt das Beispiel einer Kommunikation zwischen einem PC und einem Microcontroller (μC).



Die seriellen Daten werden im PC vom UART am Pin TxD (Transmit x Data) erzeugt und am Pin RxD (Receive x Data) abgetastet. Die Datensignale werden am Pin TxD erzeugt und am Pin RxD abgetastet. Für einen funktionierenden Datentransfer muss das TxD Signal des PC beim RxD Signal des μC enden, und das TxD Signal des μC muss beim RxD Signal des PC enden. Es gibt daher für jede Kommunikationsrichtung eine eigene Leitung. Der Kabelschirm und die Steckergehäuse sind über eine R/C-Kombination an die Signalmasse angebunden. Hochfrequente Störsignale werden daher über die niedrige Impedanz der R/C Kombination zur Masse abgeleitet. Die Signalmasse (das Nullpotential) wird im Kabel als eigene Leitung mitgeführt. Die Pegelwandlung zwischen TTL/CMOS-Pegeln beim μC und den RS-232 Pegel wird mit Hilfe der MAX232 Bausteine vorgenommen. 4AHME FIVU - μC-Peripherie

Datenflusskontrolle (Flow Control, Handshaking)

Manchmal besteht die Situation, dass der Empfänger die Daten erst verarbeiten muss und während dieser Verarbeitung keine neuen Daten eintreffen dürfen. Durch Methoden der Datenflusskontrolle kann der Empfänger das Sendeverhalten des Senders steuern.

Hardware-Handshake

Dabei werden zwei zusätzliche Signalleitungen benötigt.

RTS: Request to Send (Output)

CTS: Clear to Send (Input)

Im oberen Bild sind diese Pins als $nRTS$ und $nCTS$ zu erkennen. Das "n" deutet auf die negative Logik hin. Wenn der Empfänger $nRTS=0$ erzeugt, so darf der Sender senden. Bei $nRTS=1$ darf der Sender nicht senden.

Software-Handshake

Hier übernehmen die ASCII Steuerzeichen XON (17) und XOFF (19) die Aufgabe der Datenflusskontrolle. Wenn der Empfänger dem Sender das Zeichen XOFF sendet, so warter der Sender mit der Entsendung weitere Zeichen bis der Empfänger ein XON sendet.

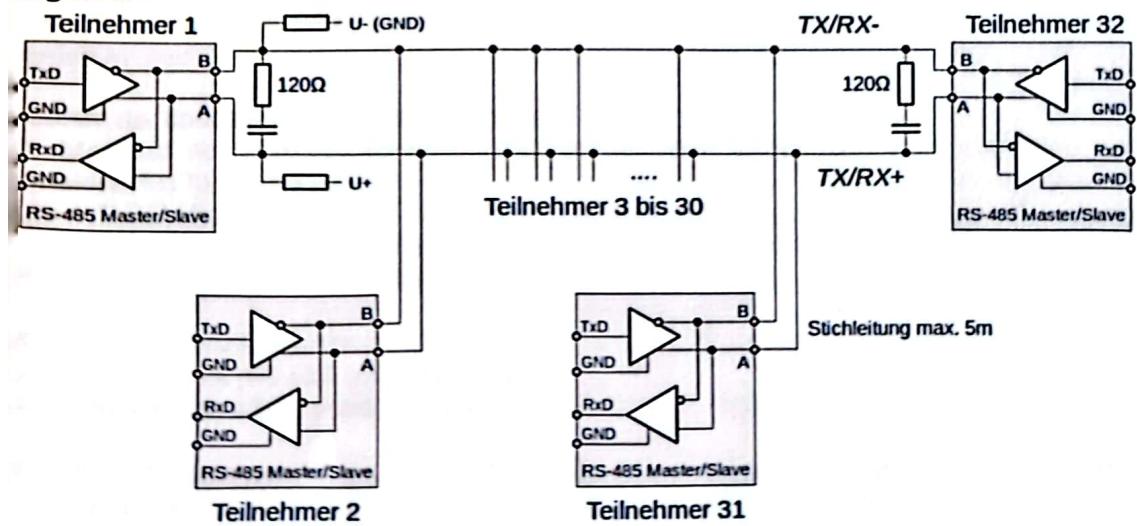
RS-422/RS-485

Bei der RS-422/RS-485 Schnittstelle wird durch eine echte differentielle Übertragungstechnik hinsichtlich Fehlerrate ein besseres Ergebnis als bei der RS-232 erzielt, trotz geringerer Spannungspegel. Die RS-422/RS-485 Schnittstelle ist durch die folgenden Eigenschaften gekennzeichnet:

- Voll differentielle Übertragungstechnik mit Signalpegeln $\pm 6V$
- Die Übertragung funktioniert auch mit den Signalpegeln 0V und 5V
- Es werden zwei Leitungen A (nicht invertiertes Signal) und B (invertiertes Signal) für den Datentransfer verwendet
- Die maximale Datenübertragungsrate beträgt 10MBit/s (bei max. 12m Länge) Die maximale Buslänge beträgt 1200m (bei maximal 90kBit/s)
- Bei höherer Übertragungsgeschwindigkeit ist am Ende der Leitung ein Abschlusswiderstand (120Ω) erforderlich.
- Eine Stichleitung zu einem einzelnen Teilnehmer darf maximal 5m lang sein
-

Die RS-422 ist für den Datenaustausch zwischen einem Sender und maximal 10 Empfängern konzipiert (unidirektionale Verbindung). Bei der RS-485 werden alle Kommunikationsteilnehmer (bis zu 32) über die Leitung A/B miteinander verbunden. Diese Konfiguration wird **Datenbus** oder auch **RS-485 Bus** bezeichnet. Es findet eine bidirektionale Kommunikation statt. Die Ausgangsstufen mit einem eingebauten Widerstand sind kurzschlussfest gemacht, damit bei fehlerhaftem gleichzeitigem Senden zweier Busteilnehmer keine Zerstörung der Ausgangsstufen eintritt. Weiters verfügen RS-485 Ausgangstreiber über ein "enable" Steuersignal, welches eine Abschaltung der Ausgangsstufe ermöglicht. Über das Kommunikationsprotokoll wird definiert, welcher Busteilnehmer gerade senden darf.

Die RS-485 kann als halbduplex Variante (siehe unteres Bild) implementiert sein (2-Draht Variante). Bei einem bidirektionalen Datenaustausch zwischen 2 Teilnehmern muss jede Richtung zeitversetzt gesendet werden, da ein gleichzeitiger Datenaustausch in beide Richtungen nicht möglich ist.



Beim Vollduplex-Betrieb ist ein zusätzliches A/B Leitungspaar erforderlich (4-Draht Variante). Die Receiver (Empfänger) in den einzelnen Teilnehmern werden in diesem Fall nicht an den Transmitter (Sender) angeschlossen, sondern mit dem zweiten A/B Leitungspaar verbunden. Ein Teilnehmer kann dann zu gleicher Zeit senden und empfangen. Die bekannteste, auf der 2-Draht Technik der RS-485 Schnittstelle basierende Anwendung, ist der Feldbus.

Die Zuordnung der Differenzspannung zu einem logischen Digitalwert ist in der EIA-485 Norm nicht definiert, aber in der Regel wie folgt implementiert:

- $(UA - UB) < -0,3V \rightarrow \text{MARK} \rightarrow \text{OFF} \rightarrow \text{"Logisch 1"}$
- $(UA - UB) > +0,3V \rightarrow \text{SPACE} \rightarrow \text{ON} \rightarrow \text{"Logisch 0"}$

Wie bei der RS-232 wird "Logisch 1" als Ruhepegel verwendet. Jedes Daten-Wort startet mit einem Starbit (=0), gefolgt von 5-8 Datenbits (LSB zuerst) und 1-2 Stopbits (=1). Zwischen dem letzten Datenbit (dem MSB) und dem Stopbit kann optional ein Even- oder Odd-Parity-Bit liegen.

Asynchrone Kommunikation - Signalabtastung

Beim Datenempfang wird das Spannungssignal am Pin RxD abgetastet. Sender und Empfänger sind zwar auf die gleiche Geschwindigkeit konfiguriert, haben aber in der Regel eine unterschiedliche Taktquelle.

Die Taktfrequenz von Sender und Empfänger ist daher entweder völlig unterschiedlich, oder bei gleicher Hardware zumindest leicht abweichend:

beim internen RC-Oszillatoren typisch $\pm 2\%$ (stark temperaturabhängig)

bei Verwendung eines externen Quarzes oder Resonator typisch $\pm 100\text{ppm}$ ($\pm 0,01\%$) (leicht temperaturabhängig)

Der Empfänger kann mit Hilfe der fallenden Flanke (Stop-Bit->Start-Bit oder Idle->Start-Bit) mit der Abtastung des Start-Bit beginnen. Dabei wird je nach U2X Bit das Start-Bit zeitlich in 8 oder 16 Teile zerlegt. Zu drei Zeitpunkten in der Bit-Mitte wird das Signal abgetastet. Nur wenn mindestens 2 der 3 Samples den Wert 0 ergeben, wird das Start-Bit akzeptiert. Das verhindert, dass kurze Störungen (Spikes) die Kommunikation ungewollt starten.

Abtastung Startbit:

Anschließend werden die Daten-Bits und das optionale Parity-Bit abgetastet. Auch hier kommt ein ähnlicher Mechanismus wie beim Start-Bit zur Anwendung um Störungen zu verhindern.

Abtastung Datenbit Beim Stop-Bit wird nur das erste Stop-Bit abgetastet. Hat es nicht den Wert 1, so wird das Frame Error Bit FE (Bit 4 im I/O-Register UCSRA) gesetzt.

Abtastung Stopbit: Da die Takte von Sender und Empfänger in der Regel leicht abweichen und nicht synchron sind, wird sich der Abtastpunkt im Laufe der Bits leicht nach vorne oder hinten verschieben. Ist die Verschiebung zu groß, so kommt es zu Übertragungsfehlern. Das ist auch der Grund warum bei einer UART-Kommunikation nur wenige Daten-Bits in einem Stück übertragen werden können.

Problem der unterschiedlichen Taktfrequenzen

Ein Merkmal der asynchronen Kommunikation ist, dass Sender und Empfänger über kein gemeinsames Taktsignal verfügen. Selbst wenn es sich um zwei gleiche Systeme handelt, also zwei Sure AVR DEM Boards mit einem 12MHz Quarz, so werden die steigenden Flanken der Taktsignale in den beiden Systemen praktisch nie übereinander liegen, denn aufgrund von Toleranzen wird die Frequenz (als auch der Anlauf) nie exakt gleich sein.

Typische Frequenz-Toleranzen:

RC-Oszillator: $\pm 5\%$ (bis $\pm 1\%$ bei Kalibrierung)

Quarz, Resonator: $\pm 30\text{ppm}$ bis $\pm 100\text{ppm}$ ($10000\text{ppm} = 1\%$, $\rightarrow 0,003\% \dots 0,01\%$)

Die Frequenztoleranzen führen unweigerlich dazu, dass die Bitbreite des Senders mit der Bitbreite des Empfängers nicht exakt übereinstimmt. Je mehr Bits in einem Frame übertragen werden, desto größer wird beim Empfänger bei der Abtastung die maximale Abweichung aus der Bitmitte sein. Ab