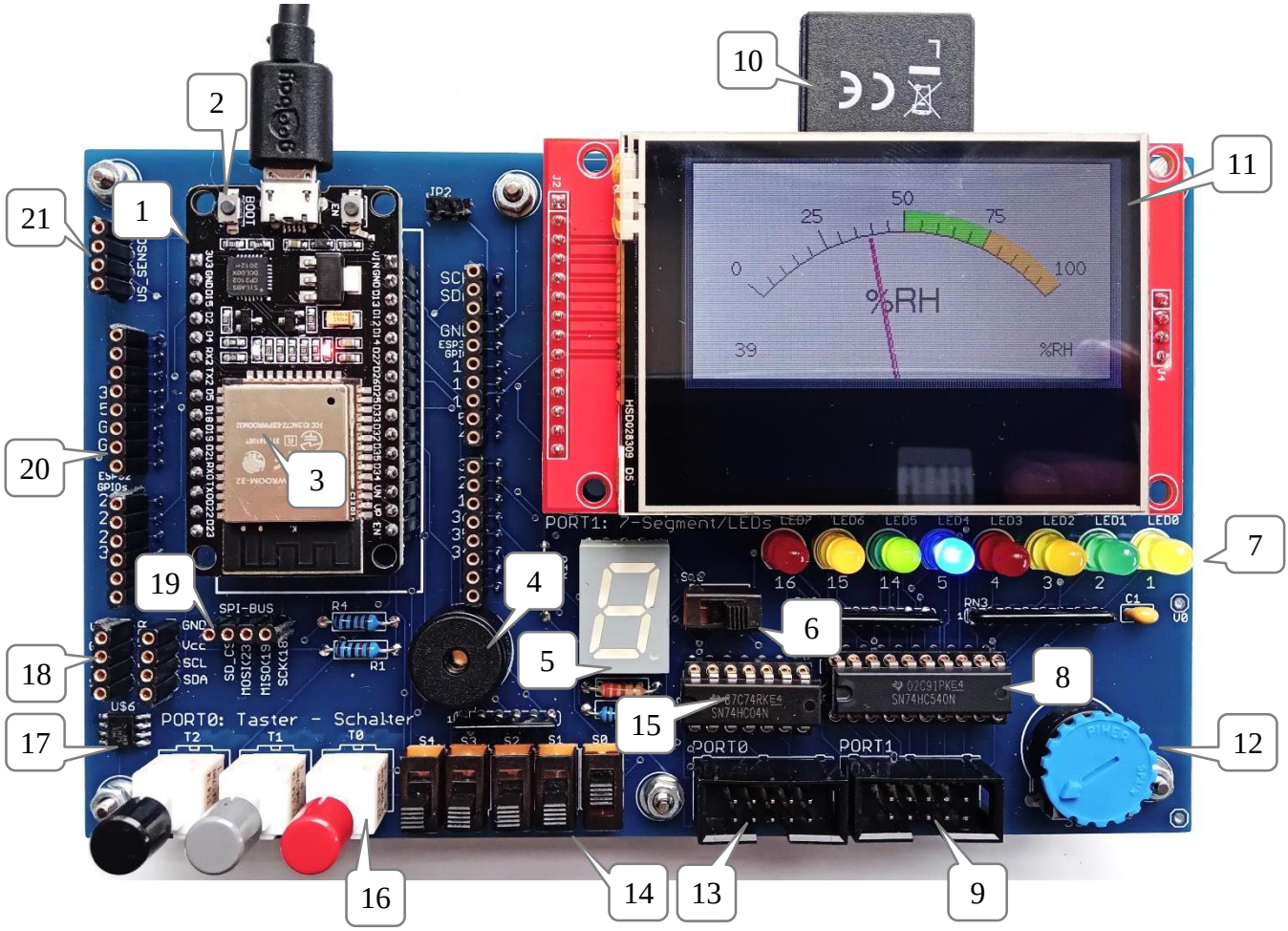
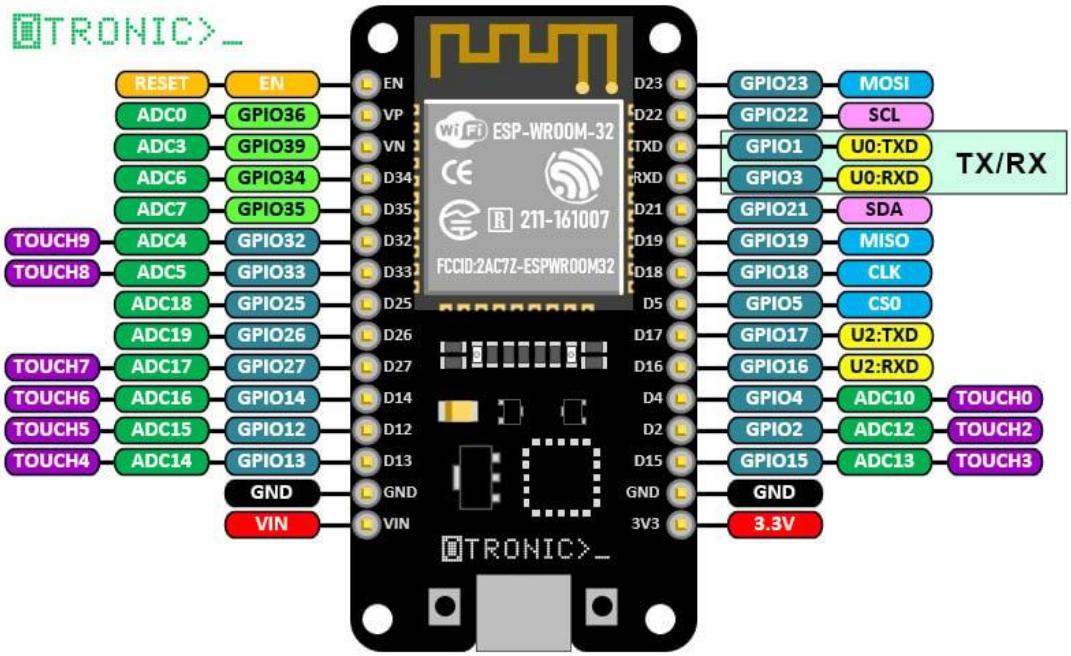
# Mikrocontroller ESP32




## Platinenerklärung:

**1** ESP32-Dev-Kit V1
**2** Boot Taster (Programmiermodus)
**3** ESP32-Wroom-32 Modul (µC)
**4** Summer
**5** 7-Segmentanzeige
**6** Umschalter (LED/ 7-Segmentanzeige)
**7** LED's (GPIOs: 1, 2, 3, 4, 5, 14, 15 16)
**8** LED-Treiber 74HC540
**9** Anschluss für LED's (GPIOs: 1, 2, 3, 4, 5, 14, 15, 16)
**10** SD-Karte
**11** TFT-Touch-Display mit SD-Karte (2,8 Zoll, 320x240 Pixel, Steuerung via SPI-Bus)
**12** Poti Analogeingabe 0…3,3V (GPIO 39)
**13** Anschluss für Schalter (GPIOs: 25, 26, 27, 32, 33)
**14** Schiebeschalter (GPIOs: 25, 26, 27, 32, 33)
**15** IC für Tastenentprellung
**16** Taster (GPIOs: rot: 34, grau: 35, schwarz: 36)
**17** Temperatursensor LM75
**18** 2 x Steckplätze für I²C-Sensoren
**19** Steckplatz für SPI-Bus und CS für SD-Karte
**20** Steckplatz für Arduino-Shields
**21** Steckplatz für Ultraschallsensor HC-SR04

## ESP32-Dev-Kit V1

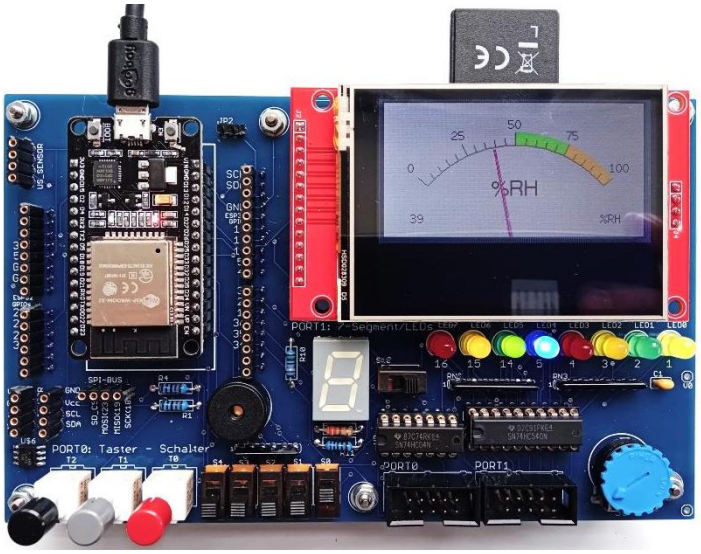



[https://www.otronic.nl/nl/esp32-wroom-4mb-devkit-v1-board-met-wifi-bluetooth.html#gallery-3]

Alle GPIOs sind PWM-fähig

**Achtung:** Pin 34-39 können ausschließlich als Eingänge verwendet werden. Sie haben keine Software-Pull-Ups/Downs

## Pin-Belegung der Platine:




### LED/Siebensegment (alle PWM-fähig)

| | | |
|---|---|---|
| LED 0 (P1.0) | **GPIO 1** | Segment a |
| LED 1 (P1.1) | **GPIO 2** | Segment b |
| LED 2 (P1.2) | **GPIO 3** | Segment c |
| LED 3 (P1.3) | **GPIO 4** | Segment d |
| LED 4 (P1.4) | **GPIO 5** | Segment e |
| LED 5 (P1.5) | **GPIO 14** | Segment f |
| LED 6 (P1.6) | **GPIO 15** | Segment g |
| LED 7 (P1.7) | **GPIO 16** | Segment h<br>Summer |

### Taster (low-active)

| | |
|---|---|
| rot | **GPIO 34** |
| grau | **GPIO 35** |
| schwarz | **GPIO 36** |

### Schalter

| | | |
|---|---|---|
| S0 | **GPIO 25** | *Ultraschallsensor Trigger* |
| S1 | **GPIO 26** | *Ultraschallsensor Echo* |
| S2 | **GPIO 27** | |
| S3 | **GPIO 32** | |
| S4 | **GPIO 33** | |

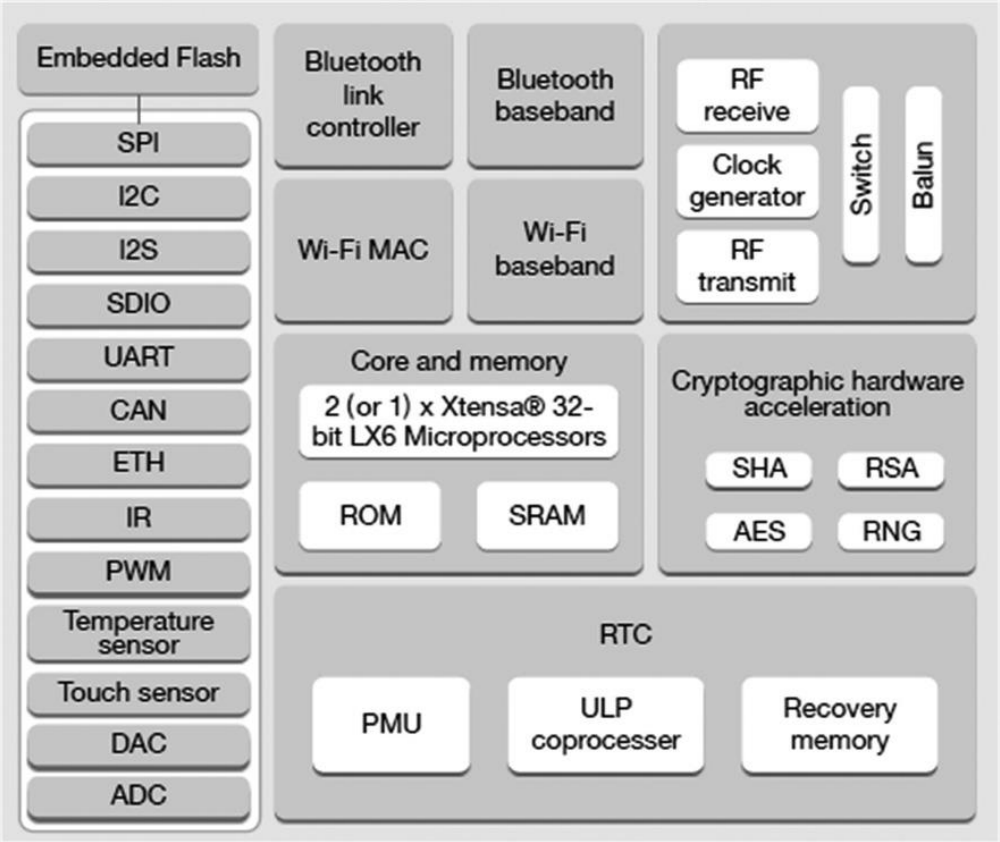
### Poti

| |
|---|
| **GPIO 39** |

### TFT-Touch-Display (SPI)

| | |
|---|---|
| **GPIO 12** | CS-Display |
| **GPIO 17** | CS-Touch |
| | CS-SD Karte muss manuell gesteckt werden |

## Funktionsblöcke des ESP32-Contollers




[Quelle: Das offizielle ESP32 Handbuch, S. 25, Elektor-Verlag, 2018]

## Technische Daten:

| | |
|---|---|
| **CPU** | 32-Bit Xtensa LX6 Dual-Core Mikroprozessor |
| **Frequenz** | 240 MHz |
| **SRAM** | 512 KB |
| **ROM** | 448 KB für das Booten und Kernfunktionen |
| **Embedded Flash** | 4 MB |
| **GPIOs** | 34 (**G**eneral **P**urpose **I**nput **O**utput – frei programmierbare Ein- und Ausgänge) |
| **ADC** | 12-Bit Analog-Digital Wandler mit 18 Kanälen |
| **PWM** | 16 unabhängige PWM Kanäle |

## Gebräuchlichste Datentypen:

| | |
|---|---|
| **bool** | Speichert die boolschen Wahrheitswerte **true** und **false** (belegt 1Byte speicher) |
| **char** | 1 Byte für Zeichen -128 ... 127 |
| **unsigned char** | 1 Byte: 0...255 (entspricht dem Datentyp **byte** und dem typedef **uint8_t**) |
| **short** | 2 Byte: -32768..32767 |
| **unsigned short** | 2 Byte: 0...65535 (entspricht dem Datentyp **word** und dem typedef **uint16_t**) |
| **int** | 4 Byte: -2147483648 ... 2147483647 |
| **unsigned int** | 4 Byte: 0 ... 4294967295 (entspricht dem typedef **uint32_t**) |
| **float/ double** | Für Kommazahlen |

## Die Klasse String:

Strings in C: char my_uC[6] = “ESP32“; //In das letzte Element wird das
//String-Endezeichen \0 geschrieben

Die Klasse String kann in den meisten fällen C-Strings ersetzten.

```
String my_uC = “ESP32“;
```

Beispiele für String-Operationen:

```
String str1 = “ ESP“;
String str2;
String str3;
str2 = String(32);          //Integer-Zahl 32 in String umwandeln und str2 zuweisen

str3 = str1 + str2 +“ !!“;  //String zusammensetzten. In str3 steht also: ESP32 !!
```

**Einige Methoden der String-Klasse:**

| | |
|---|---|
| **length()** | Länge des Strings. Beispiel: i = str1.length; |
| **toInt()** **toFloat()** **toDouble()** | String in entsprechenden Datentyp umwandeln Beispiel: i = str1.toInt(); |
| **c_str()** | String in \0-terminierten C-String umwandeln |
| **remove()** | Löscht Zeichen ab einer angegebenen Position. |
| **replace()** | Zeichen ersetzten. |
| **trim()** | Leerzeichen löschen. |
| **indexOf()** | String suchen |
| **substr()** | String extrahieren |

# Befehls-/Funktionsliste auf einen Blick

## Zeitfunktionen

| | |
|---|---|
| **delay** (uint32_t **ms**) | Unterbricht das Programm für die als Parameter angegebene Zeit in Millisekunden (ms) |
| **delayMicroseconds** (uint32_t **us**) | Unterbricht das Programm für die als Parameter angegebene Zeit in Mikrosekunden (µs) |
| **millis(), micros()** | Gibt die Anzahl an Milli- bzw. Mikrosekunden seit dem Programmstart zurück. Die Rückgabe erfolgt mit dem Datentyp unsigned long. Bei Zeitüberlauf (millis(): ca. 50 Tage; micros(): ca. 70 min) beginnt der Wert wieder bei 0. |

## Digitale Ein-/Ausgabe

| | |
|---|---|
| **pinMode** (uint8_t **pin,** uint8_t **mode**) | Pin als Ein- oder Ausgang konfigurieren |
| **digitalWrite** (uint8_t **pin,** uint8_t **value**) | Schaltet den Pin ein (1/HIGH) oder aus (0/LOW) |
| uint8_t **digitalRead** (uint8_t **pin**) | Liest den Zustand des Pins ein |

Parameterliste :
**pin**: 0…39 (GPIO-Nummer)
**mode**: INPUT (1), OUTPUT (2)
**value**: LOW (0), HIGH (1)

## Port- und einfache TFT-Display Steuerung (Header FVS.h)

Für die Port- und einfache Display-Steuerung muss der Header FVS.h eingebunden werden. In diesem Header befinden sich spezielle Funktionen für das Schulboard.

**#include "FVS.h"**

### Port-Steuerung

| | |
|---|---|
| **portMode** (uint8_t **port,** uint8_t **mode**) | Port als Ein- oder Ausgang konfigurieren |
| **portWrite** (uint8_t **port,** uint8_t **value**) | Überträgt den Wert von value an den entsprechenden Port |
| uint8_t **portRead** (uint8_t **port**) | Liest den Zustand des entsprechenden Ports ein und gibt einen Wert zw. 0 … 255 zurück. |

Parameterliste :
**port**: 0, 1
**mode**: INPUT (1), OUTPUT (2)
**value**: 0 … 255

### TFT-Display Steuerung

In dem Header FVS.h ist die Klasse Fvs_tft implementiert. Die Klasse erbt von TFT_eSPI (TFT_eSPI.h). Ein globales Objekt **Tft** der Klasse steht zur Display-Steuerung zur Verfügung.

Methoden:

| | |
|---|---|
| **Tft.begin()** | Display initialisieren (Schriftgröße, -farbe, Hintergrundfarbe und Orientierung werden festgelegt) |
| **Tft.setCursorCharacter(**uint8_t **row**, uint8_t **column)** | Cursor zeichenweise positionieren. Parameter: **row**: 1…10 **column:** 1…17 |
| **Tft.setCursor(**int16_t **x**, int16_t **y)** | Cursor pixelweise positionieren. Parameter: **x**: 0…319 **y:** 0…239 |
| **Tft.print(data)** | Daten auf dem Display an der aktuellen Cursor-Position ausgeben. Bei der print-Methode handelt es sich um eine überladene Methode. Es können alle gebräuchlichen Datentypen übergeben werden. |
| **Tft.println(data)** | Wie print, springt zusätzlich an den Anfang der nächsten Zeile. |
| **Tft.clearDisplay()** | Löscht das Display. |

## Analog-Digital Wandler

| | |
|---|---|
| uint16_t **analogRead** (uint8_t **pin**) | Liest den Wert an dem analogen Pin ein und wandelt ihn um. **Rückgabewert:** 0 … 4095 |
| long **map** (long value, long fromLow, long fromHigh, long toLow, long toHigh) | Bildet eine Zahl von einem Bereich in einen anderen ab. Das heißt, ein Wert von fromLow würde auf toLow, einen Wert von fromHigh bis toHigh, Werte zwischen dazwischen auf Werte dazwischen usw. zugeordnet. |

## Ext. Interrupts

Soll ein **Pin** als externer Interrupt verwendet werden, muss dieser zunächst mit der Funktion pinMode() als **Eingang (INPUT)** konfiguriert werden. Zusätzlich muss eine ISR-Funktion programmiert werden.

**attachInterrupt(**digitalPinToInterrupt**(pin), ISR,** int **mode)**

Parameterliste :
**pin:** entspricht der GPIO-Nummer (0..39), von der aus der ext. Interrupt ausgelöst werden soll.
**ISR:** Name der Interrupt-Service-Routine
**mode:** FALLING (2), Interrupt löst aus, wenn der Pin von HIGH nach LOW wechselt. Weitere Optionen: RISING (1), CHANGE (3)

## Timer

Es stehen 4 unabhängige 64-Bit Hardwaretimer mit 16-Bit Teilern zur Verfügung. **Takt: 80MHz**

**hw_timer_t*** *timerName* = NULL; globale „Timervariable“ *timerName* erzeugen

*timerName* = **timerBegin**(**timerNr**, **teiler**, true) Timer *timerName* initialisieren; true = count up

Timer-Interrupt konfigurieren:

| | |
|---|---|
| **timerAttachInterrupt**(*timerName*, **&ISR**, true) | Interrupt hinzufügen; true = flankengesteuert |
| **timerAlarmWrite**(*timerName*, **isr_at**, true) | Interrupt Verhalten festlegen; true = autoreload |
| **timerAlarmEnable**(*timerName*) | Interrupt aktivieren und Timer starten |
| **timerAlarmDisable**(*timerName*) | Interrupt deaktivieren und Timer stoppen |

Funktionen für die Timersteuerung:

| | |
|---|---|
| **timerStart**(*timerName*) | Timer/Zählvorgang starten |
| **timerStop**(*timerName*) | Timer/Zählvorgang stoppen |
| **timerRestart**(*timerName*) | Zählerstand auf 0 setzten |

Parameterliste :

*timerName*: frei wählbarer Name
**timerNr:** 0…3
**teiler:** Timertakt = 80MHz/**teiler**
**&ISR:** Adresse der Timer-ISR
**isr_at:** Bei diesem Zählerstand, wird der Interrupt ausgelöst (Endwert).

## PWM

Es stehen 16-PWM Kanäle (0-15) zur Verfügung, denen jeweils ein GPIO-Pin zugeordnet werden kann. Für jeden Kanal kann die Auflösung und die Frequenz separat eingestellt werden.

| | |
|---|---|
| **ledcSetup**(uint8_t **channel**, double **freq**, uint8_t **res**) | PWM-Kanal konfigurieren |
| **ledcAttachPin**(uint8_t **pin**, uint8_t **channel**) | GPIO-Pin mit PWM-Kanal verknüpfen |
| **ledcWrite**(uint8_t **channel**, uint32_t **value**) | PWM-Signal erzeugen (Einschaltdauer festlegen) |

Parameterliste :

**channel:** 0…15
**freq:** Frequenz in Hz, z. B. 10000
**res:** 1…16 (Auflösung), z. B. 8
**pin:** GPIO-Nummer (Pin muss Output-fähig sein)
**value:** 0…255 (für res = 8)

**Hinweis:** Für die Erzeugung der PWM-Signale werden Timer verwendet. Ein für die PWM verwendeter Timer kann nicht zusätzlich als unabhängiger Hardware-Timer verwendet werden. Die Zuordnung von PWM-Kanal und Timer ist nachfolgend dargestellt.

| PWM-Kanal | Timer Nr. |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |
| 4 | 2 |
| 5 | 2 |
| 6 | 3 |
| 7 | 3 |

| PWM-Kanal | Timer Nr. |
|---|---|
| 8 | 0 |
| 9 | 0 |
| 10 | 1 |
| 11 | 1 |
| 12 | 2 |
| 13 | 2 |
| 14 | 3 |
| 15 | 3 |

# Klassen für die Kommunikation via externer Schnittstellen

## RS232 (Serial)

Kommunikation über die serielle Schnittstelle bzw. auch mit dem seriellen Monitor. Hierfür wird die **Klasse bzw. das Objekt Serial** zur Verfügung gestellt.

Einige Methoden:

| | |
|---|---|
| **begin(baud)** | öffnet eine serielle Verbindung. Der Parameter **baud** stellt die Baudrate ein (z. B. **9600, 115200, …**).<br>Beispiel: **Serial.begin(115200);**<br>Es können ggf. noch weitere Einstellungen wie z. B. Anzahl der Daten-, Stopp- und Paritätsbits vorgenommen werden. |
| **print(data)**<br>**println(data)** | Gibt Daten (als Zeichen) an die serielle Verbindung aus. Die Methode ist überladen, es können fast alle Datentypen übergeben werden. println() fügt zusätzlich einen Absatz ein. |
| int **available()** | Gibt die Anzahl der empfangenen Zeichen zurück. Diese befinden sich im Empfangspuffer und stehen zur Weiterverarbeitung zur Verfügung |
| **read()** | Liest eingegangene Daten byteweise aus dem Empfangspuffer aus. Beispiel: |
| **readString()** | Liest einen String aus dem Empfangspuffer aus. |

```
String str
while (Serial.available())
{   str = str + Serial.read();}
```

Weitere Methoden zum Lesen und gezielten verarbeiten der eingegangenen Daten:
**readStringUntil(), readBytes(), readBytesUntil(), find(), findUntil(), parseInt(), parseFloat()**

## I²C (Wire)

Kommunikation über den I²C Bus. Hierfür stellt die Bibliothek **Wire** das **Objekt Wire** zur Verfügung.

**#include <Wire.h>**

Einige Methoden:

| | |
|---|---|
| **begin(*adresse*)** | Meldet den ESP32 mit der angegebenen Adresse (***adresse***) am I²C-Bus an. Ist der ESP32 der Master, so entfällt das Angeben der Adresse. Durch weitere Parameter können die Pins und die Frequenz festgelegt werden. |
| **beginTransmission(*adresse*)**<br>**write(data)**<br>**endTransmission()** | Wire.beginTransmission(adresse) baut eine Datenverbindung mit dem Busteilnehmer der angegebenen 7-Bit Adresse (***adresse***) auf. Mit einem oder mehreren Wire.write(data) können nun Daten in den Sendepuffer geschrieben werden. Durch Wire.endTransmission() werden die im Sendepuffer stehenden Daten übertragen. |
| **requestFrom(adresse, bytes)**<br>int **available()**<br>int **read()** | Mit Wire.requestFrom() fordert der Master den unter ***adresse*** (7-Bit Adresse) angesprochenen Slave auf Daten zu senden. ***bytes*** definiert die Anzahl an Bytes die der Slave senden soll.<br>Mit Wire.available() kann geprüft werden, wie viele Bytes sich im Empfangspuffer befinden. Diese können dann mir Wire.read() ausgelesen werden. (Diese Methoden sind analog zu den Methoden der Serial-Klasse) |

## Bluetooth (BluetoohhSerial)

Serielle Kommunikation via Bluethooth. Hierfür wird die Bibliothek **BluethoothSerial** benötigt. Die Bibliothek stellt die gleichnamige Klasse zur Verfügung, von der im Programm ein globales Objekt erzeugt werden muss.

**#include "BluetoothSerial.h"**

| | |
|---|---|
| **BluetoothSerial** *name*; | globales Objekt der Klasse BluethoothSerial erzeugen. *name* beliebig wählbar. |

Einige Methoden:

| | |
|---|---|
| **begin(**String **btName)** | Bluetooth initialisieren und den ESP32 als Bluetooth-Gerät mit dem Namen ***btName*** sichtbar machen. |
| bool **connected()** | Überprüft ob eine Bluetooth-Verbindung zu einem anderen Gerät besteht. |

Zum Senden und Empfangen von Daten stellt die Klasse BluetoothSerial die gleichen Methoden zur Verfügung wie die Klasse Serial (Funktionsweise siehe oben).

**print(), println(),** int **available(), int read(), readStringUntil(), find(), findUntil(), parseInt(), parseFloat(), …**

# TFT-Touch-Display

## Bibliothek: TFT_eSPI

Das Display kann mit der Bibliothek TFT_eSPI betrieben werden.
In der Datei User_Setup.h müssen die GPIO- und Treiber Konfigurationen vorgenommen werden (siehe rechts).

```
#define ILI9341_DRIVER

#define TFT_MISO 19
#define TFT_MOSI 23
#define TFT_SCLK 18
#define TFT_CS   12
#define TFT_DC   13
#define TFT_RST  -1
#define TOUCH_CS 17
```

**Technische Daten:**
Größe: 2,8 Zoll
Ansteuerung: SPI-Bus
Treiber IC: ILI9341
Farbtiefe: 262K/65K
Auflösung (Pixel):
240RGB*320Pixel

## Farbdefinitionen:

```
// Default color definitions
#define TFT_BLACK       0x0000      /*   0,   0,   0 */
#define TFT_NAVY        0x000F      /*   0,   0, 128 */
#define TFT_DARKGREEN   0x03E0      /*   0, 128,   0 */
#define TFT_DARKCYAN    0x03EF      /*   0, 128, 128 */
#define TFT_MAROON      0x7800      /* 128,   0,   0 */
#define TFT_PURPLE      0x780F      /* 128,   0, 128 */
#define TFT_OLIVE       0x7BE0      /* 128, 128,   0 */
#define TFT_LIGHTGREY   0xD69A      /* 211, 211, 211 */
#define TFT_DARKGREY    0x7BEF      /* 128, 128, 128 */
#define TFT_BLUE        0x001F      /*   0,   0, 255 */
#define TFT_GREEN       0x07E0      /*   0, 255,   0 */
#define TFT_CYAN        0x07FF      /*   0, 255, 255 */
#define TFT_RED         0xF800      /* 255,   0,   0 */
#define TFT_MAGENTA     0xF81F      /* 255,   0, 255 */
#define TFT_YELLOW      0xFFE0      /* 255, 255,   0 */
#define TFT_WHITE       0xFFFF      /* 255, 255, 255 */
#define TFT_ORANGE      0xFDA0      /* 255, 180,   0 */
#define TFT_PINK        0xFE19      /* 255, 192, 203 */
#define TFT_BROWN       0x9A60      /* 150,  75,   0 */
#define TFT_GOLD        0xFEA0      /* 255, 215,   0 */
#define TFT_SILVER      0xC618      /* 192, 192, 192 */
#define TFT_SKYBLUE     0x867D      /* 135, 206, 235 */
#define TFT_VIOLET      0x915C      /* 180,  46, 226 */
```

## Bibliotheksfunktionen:

**Klasse:** TFT_eSPI
**Konstruktor:** TFT_eSPI(int16_t _W = TFT_WIDTH, int16_t _H = TFT_HEIGHT);

### Einige Methoden:

```
void    setRotation(uint8_t r); //Set the display image orientation 0, 1, 2 or 3

void    init(uint8_t tc = TAB_COLOUR), begin(uint8_t tc = TAB_COLOUR);
```

### Text rendering and font handling support funtions

```
void    setCursor(int16_t x, int16_t y);                // Set cursor

        setCursor(int16_t x, int16_t y, uint8_t font);  // Set cursor and font

        setTextColor(uint16_t color);

        setTextColor(uint16_t fgcolor, uint16_t bgcolor);

        setTextSize(uint8_t size);

        setTextDatum(uint8_t datum);  // Set text datum position
                                      #define TL_DATUM 0 // Top left (default)
                                      #define TC_DATUM 1 // Top centre
                                      #define TR_DATUM 2 // Top right
                                      #define ML_DATUM 3 // Middle left
                                      #define MC_DATUM 4 // Middle centre
```

### Graphics drawing

```
void    drawPixel(int32_t x, int32_t y, uint32_t color);

        drawLine(int32_t xs, int32_t ys, int32_t xe, int32_t ye, uint32_t color);

        fillScreen(uint32_t color);

        drawRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color);

        fillRect(int32_t x, int32_t y, int32_t w, int32_t h, uint32_t color);

        drawRoundRect(int32_t x, int32_t y, int32_t w, int32_t h, int32_t radius, uint32_t color),

        fillRoundRect(int32_t x, int32_t y, int32_t w, int32_t h, int32_t radius, uint32_t color);

        drawSpot(float ax, float ay, float r, uint32_t fg_color, uint32_t bg_color = 0x00FFFFFF);

        drawCircle(int32_t x, int32_t y, int32_t r, uint32_t color);

        fillCircle(int32_t x, int32_t y, int32_t r, uint32_t color);

        drawTriangle(int32_t x1,int32_t y1, int32_t x2,int32_t y2, int32_t x3, int32_t y3, uint32_t color);

        fillTriangle(int32_t x1,int32_t y1, int32_t x2,int32_t y2, int32_t x3, int32_t y3, uint32_t color);
```

### Touch-Methoden

```
uint8_t getTouch(uint16_t* x, uint16_t* y);   /* Get the screen touch coordinates, returns true if screen has been
                                                 touched.*/
```

Beim Aufruf der Methode müssen die Adressen von zwei Variablen des Typs uint16_t übergeben werden:
**Tft.getTouch(&x, &y);** bzw. **if ( Tft.getTouch(&x, &y) )**

Je nach Kalibrierung muss der x- oder y-Wert noch angepasst werden. Z. B. **319-x oder 239-y**

Das Display kann mit Tft.setTouch() kalibriert werden. Mit dem **Sketch** Datei→Beispiele→TFT_eSPI→ Generic→ **Touch_calibrate.ino** kann man sich den hierfür benötigten Code direkt im seriellen Monitor anzeigen lassen und ins setup des Programms kopieren. **Die Rotation muss auf 3** gesetzt werden (tft.setRotation(3))!