Frankfurt University of Applied Sciences
Fachbereich 2, Informatik
Dipl. Inf. Markus Krauße

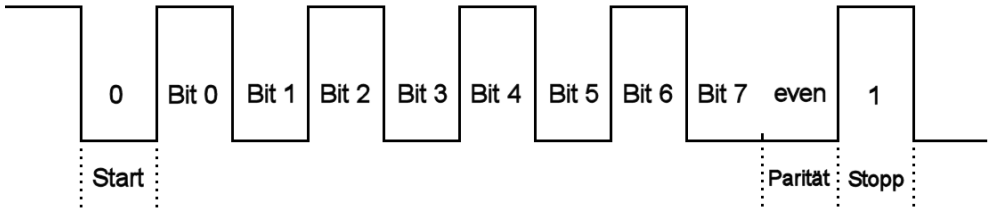
Kontakt:wsnprojekt@fb2.fra-uas.de

Blatt 2

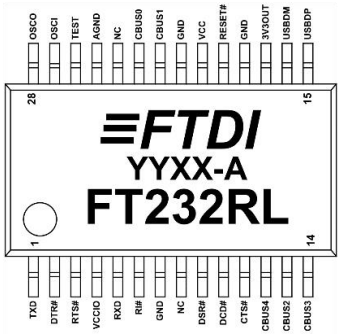
Benutzung von Schnittstellen, Ansteuern von Sensoren

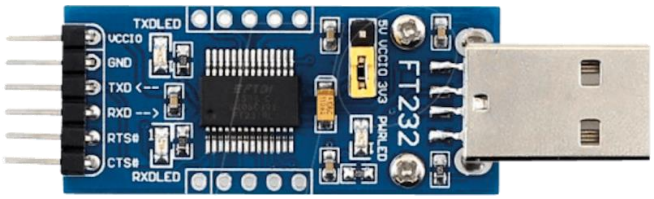
1 UART-Schnittstelle

In Aufgabenblatt 1 haben wir die UART-Schnittstelle kennengelernt, mit deren Hilfe Text über die Konsole ausgeben werden kann. Fast alle Mikrocontroller besitzen mindestens eine UART-Schnittstelle. Über UART-Schnittstellen können Daten seriell gesendet und empfangen werden. Sie dienen zum Austausch von Daten mit Geräten, die ebenfalls eine UART-Schnittstelle besitzen (z. B. andere Mikrocontroller, PCs, Sensoren). UART-Schnittstellen verfügen über eine Sendeleitung (*TXD*) und eine Empfangsleitung (*RXD*). Die Verdrahtung erfolgt an der Gegenstelle entsprechend über Kreuz. Wird eine softwareseitige Datenflusssteuerung eingesetzt, ist diese Verbindung für eine Datenübertragung ausreichend. Zusätzlich besteht die Möglichkeit den Datenfluss hardwareseitig zu steuern. Dazu besitzen UART-Schnittstellen die Steuerleitungen *RTS* (Request To Send) und *CTS* (Clear To Send). Diese sind an der Gegenstelle ebenfalls über Kreuz zu verdrahten. Sobald der Empfänger die CTS-Leitung auf logisch 1 setzt (z.B. wegen eines vollen Speichers), wird die Signalübertragung angehalten. Eine Übertragung von Daten beginnt mit einem Startbit (logisch 0). Dem Startbit folgen 5-9 Datenbits, ein optionales Paritybit und 1-2 Stoppbits (logisch 1). Die Datenbits werden immer mit dem niederwertigsten Bit zuerst gesendet. Eine Abfolge von 8-Bits 10101010 inklusive Startbit, Paritätsbit und einem Stoppbit hat beispielsweise folgendes Format:



Zwei Mikrocontroller können direkt über die UART-Schnittstelle miteinander kommunizieren, sofern sie mit dem gleichen TTL-Pegel (Signalspannung) arbeiten. Um eine Kommunikation über UART mit einer USB-Schnittstelle eines PCs zu ermöglichen, wird eine USB-UART-Bridge benötigt. Ein recht gängiger Typ einer USB-UART-Bridge basiert z.B. auf dem Chip *FT232* der Firma *FTDI*.





Die USB-UART-Bridge wandelt den Signalpegel von 5V der USB-Schnittstelle auf den Signalpegel von 3,3V eines Mikrocontrollers um. Das Betriebssystem des PCs benötigt für eine USB-UART-Bridge die Installation eines Treibers. In Windows wird durch den Treiber[1] eine COM-Schnittstelle im Betriebssystem eingerichtet, über die die Kommunikation mit der USB-UART-Bridge erfolgt. Diese COM-Schnittstelle kann mit gängigen Programmiersprachen angesteuert werden und so Daten mit einem Mikrocontroller austauschen. Die Daten der COM-Schnittstelle können auch einfach mit Hilfe eines Terminalprogramms (z.B. *Putty* oder *HTerm*) ausgelesen werden, um die Ausgaben des Mikrocontrollers in einem Terminalfenster zu visualisieren. Unser Developerboard hat einen zusätzlichen Mikrocontroller integriert, der neben der Funktionalität eines J-Link Programmiergerätes auch als USB-UART-Bridge fungiert. Diese Funktionalität haben wir bereits in den vorherigen Aufgaben für die Ausgabe von Daten mit der Funktion *printk* verwendet. Da es jedoch vorkommen kann, dass wir neben der Textausgabe auch einen Sensor über die UART-Schnittstelle auslesen wollen, werden wir hier die UART-Schnittstelle direkt ansteuern.

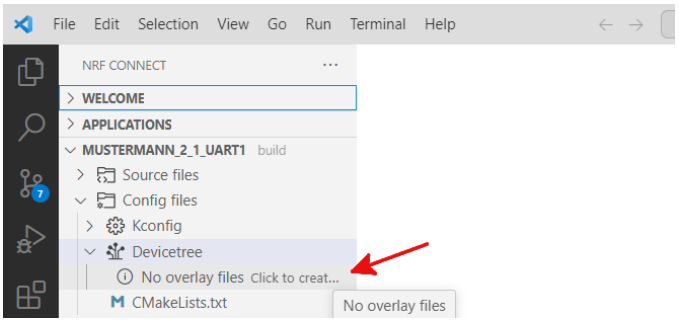
Aufgabe 1: Textausgabe auf der Konsole

Der Mikrocontroller nRF52840 stellt zwei UART-Schnittstellen zur Verfügung. Programmieren Sie eine Anwendung, die einen Counter im Sekundentakt hochzählt und über die UART1-Schnittstelle des nRF52840 und die beiliegende FT232 USB-UART-Bridge als Text an die USB-Schnittstelle eines PCs sendet. Erstellen Sie einen Screenshot der Ausgabe des Terminalprogramms und speichern diesen ebenfalls im Projektordner.

Kopieren Sie zunächst wieder die Basisanwendung und benennen diese nach unseren Konventionen um. Im Devicetree finden wir die folgende Konfiguration für die UART1-Schnittstelle:

```
arduino_serial: &uart1 {
    current-speed = <115200>;
    pinctrl-0 = <&uart1_default>;
    pinctrl-1 = <&uart1_sleep>;
    pinctrl-names = "default", "sleep";
};
```

Wenn wir allerdings mit der Maus über den uart1-Knoten gehen, sehen wir, dass der Status auf *disable* gesetzt ist. Wir müssen die Uart1-Schnittstelle damit zunächst aktivieren und generieren dafür eine Overlay-Datei indem wir bei unserem Projekt unter dem Ordner *Devicetree* auf *No overlay files-Click to create one* klicken.



Dies erzeugt die Datei *nrf52840dk_nrf52840.overlay* in unserem Projektordner und gibt uns die Möglichkeit Parameter zu ändern, ohne die Original Devicetree-Datei *nrf52840dk_nrf52840.dts* zu überschreiben. Setzen Sie den Status des uart1-Knoten in der erzeugten Overlay-Datei auf okay:

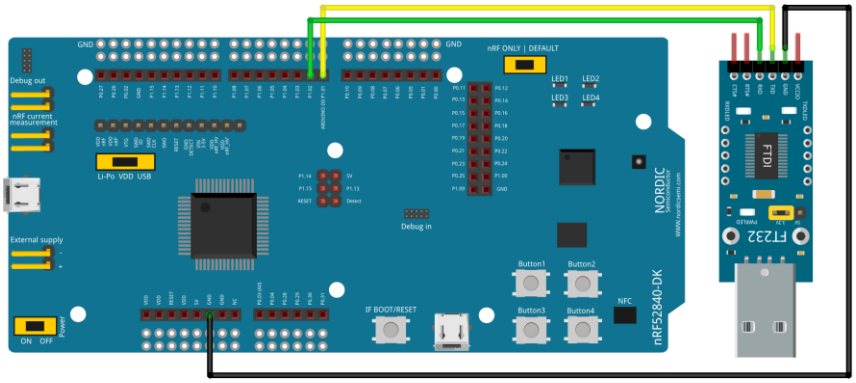
[1] VCP-Treiber für den FTDI FT232 finden Sie unter: https://ftdichip.com/drivers/vcp-drivers/

```
&uart1 {
    status = "okay";
};
```

Die Pin-Zuweisung erfolgt in Zephyr über einen sogenannten Pin-Controller. Dieser kann z.B. mehrere Pin-Zustände verwalten, im Allgemeine sind dies der *default*-Zustand und *sleep*-Zustand. Wir werden die Zustände der Uart1-Schnittstelle aber nicht ändern, sondern interessieren uns nur für die Pinbelegung. Diese finden wir unter dem Knoten `/pin-controller/uart1_default` (Wir erreichen diese Knoten in VSC durch Drücken der STRG-Taste und Anklicken von `&uart1_default`):

```
uart1_default: uart1_default {
    group1 {
        psels = <NRF_PSEL(UART_RX, 1, 1)>;
        bias-pull-up;
    };
    group2 {
        psels = <NRF_PSEL(UART_TX, 1, 2)>;
    };
};
```

Der RX-Pin ist auf Port 1 Pin 1 und der TX-Pin auf Port 1 Pin 2 festgelegt. Die USB-UART-Bridge müssen Sie wie folgt an das nRF52840 Developerboard anschließen (RX bei der USB-UART-Bridge ist natürlich TX am Developerboard und umgekehrt):



Im ersten Schritt definieren wir wieder eine Devicestruktur mit der wir auf die UART1-Schnittstelle zugreifen. Außerdem deklarieren wir eine Buffer-Array in der wir unseren Ausgabetext speichern, inklusive einer Hilfsvariable, die die Anzahl der auszugebenden Zeichen festhält und eine Counter-Variable:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/sys/printk.h>
#include <zephyr/drivers/uart.h>
#include <stdio.h>

#define SLEEP_TIME_MS   1000

#define UART1_NODE      DT_NODELABEL(uart1)      //DT_N_S_soc_S_uart_40028000
static const struct device *uart_dev = DEVICE_DT_GET(UART1_NODE);

static uint8_t tx_buf[15];
static int tx_buf_length;
static int counter=0;
```

Wir nutzen, die UART-Schnittstelle im sogenannten asynchronen Modus, d.h. Rückmeldungen über Ereignisse erhalten wir mit Hilfe einer Callback-Funktion. Da wir zunächst nur Text senden, werden wir diese erst einmal ohne Logik implementieren:

```
void uart1_cb(const struct device *dev,  struct uart_event *evt, void *user_data){
    switch (evt->type) {
        default:
            break;
    }
}
```

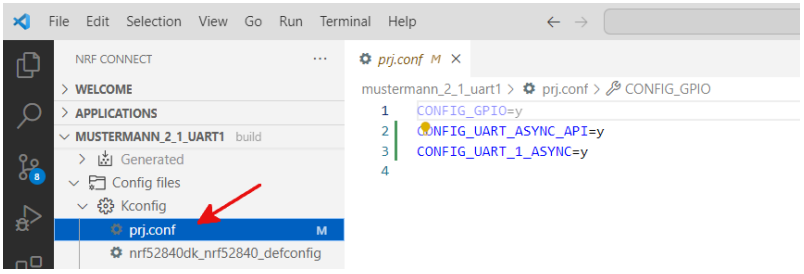
In der main-Funktion setzen wir die Callback-Funktion, zählen dann in der while-Schleife die Zählervariable hoch und geben diese über die UART-Schnittstelle aus:

```
int main(void){
    if (!device_is_ready(uart_dev)) {
        printk("uart_dev not ready\n");
        return;
    }

    uart_callback_set(uart_dev, uart1_cb, NULL);

    while (1) {
        k_msleep(SLEEP_TIME_MS);
        tx_buf_length= sprintf(tx_buf, "Counter: %d\n\r", counter);
        uart_tx(uart_dev, tx_buf , tx_buf_length, 100 * USEC_PER_MSEC);
        counter++;
    }
}
```

Bevor wir die UART-Schnittstelle nutzen können, müssen die entsprechenden Bibliotheken in das Projekt eingebunden werden. Dies geschieht recht einfach über die Projektkonfigurationsdatei prj.conf.:

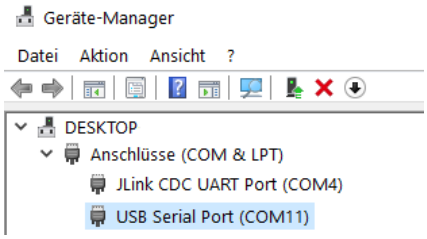


Dort setzen wir die Parameter:

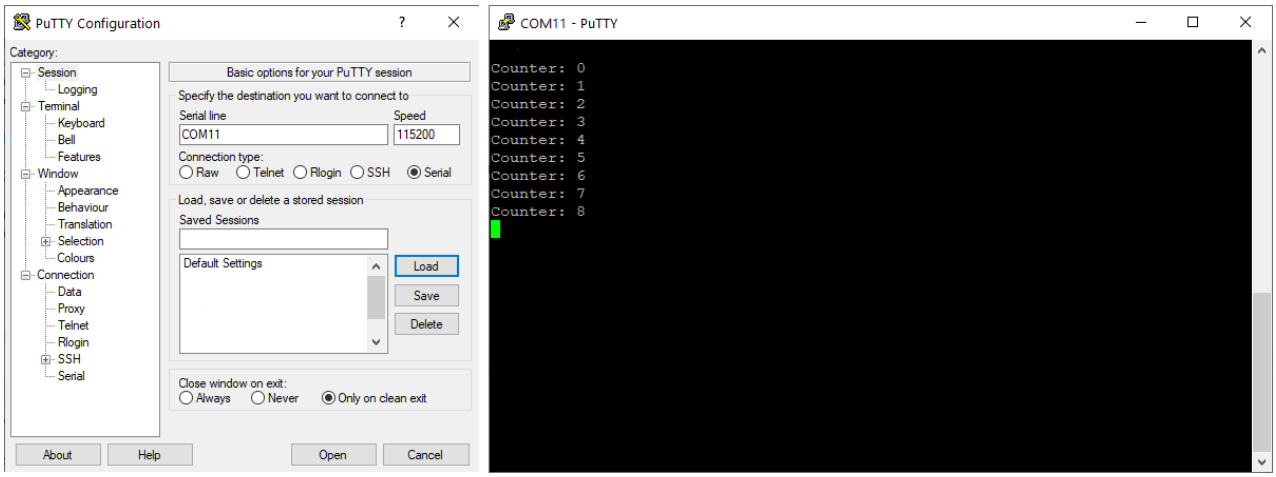
```
CONFIG_UART_ASYNC_API=y
CONFIG_UART_1_ASYNC=y
```

Jetzt können Sie das Programm kompilieren und auf das Developerboard flashen. Dann Verbinden Sie die Bridge mit einem USB-Port ihres PC's. Falls auf Ihrem PC kein Treiber für den FT232 Chip vorhanden ist,

laden Sie diesen von der FTDI-Webseite[2] runter und installieren diesen auf ihrem PC. Wird jetzt ein Terminalprogramm (z.B. Putty oder HTem) gestartet, muss der Port an dem die Bridge angeschlossen ist, ausgewählt werden. Der Port kann in Windows durch Öffnen des Gerätemanagers ermittelt werden:



Der Verbindungstyp muss auf Serial gestellt und für die Geschwindigkeit die Übertragungsrate 115200 Baud ausgewählt werden. Das Paritätsbit und die Flusskontrolle sind auszuschalten und die Anzahl der Stoppbits auf 1 zu setzen. Nach aufgebauter Verbindung erscheint im Terminalfenster die Ausgabe:



Aufgabe 2: Daten einlesen über die UART-Schnittstelle

Lesen Sie Daten über die UART1-Schnittstelle ein und geben diese über mit *printk* über die UART0-Schnittstelle aus. Zum Testen geben Sie bitte ihre Matrikelnummer ein, machen einen Screenshot von der Ausgabe und legen diesen der Abgabe bei.

Wenn wir einen Sensor über die UART-Schnittstelle ansteuern, müssen auch Daten von diesem empfangen werden können. Dies funktioniert in Zephyr bei der asynchronen Übertragung durch das Abfangen von Events in der registrierten Callback-Funktion. Zudem benötigen wir ein weiteres Buffer-Array, welches die Daten zwischen speichert:

```
static uint8_t rx_buf[20];

void uart1_cb(const struct device *dev,  struct uart_event *evt, void *user_data){
    switch (evt->type) {
        case UART_RX_RDY: ;//empty statement
            int offset = evt->data.rx.offset;
            int len = evt->data.rx.len;
            for (size_t i = offset; i < (offset+len); i++){
```

[2] https://www.ftdichip.com/Drivers/VCP.htm (CDM v2.12.36.4 WHQL Certified.zip)

```
                if (evt->data.rx.buf[i]=='\r'){
                    printk("\n");
                }else{
                    printk("%c", evt->data.rx.buf[i]);
                }
            }
            break;

        case UART_RX_DISABLED:
            uart_rx_enable(uart_dev, rx_buf, sizeof(rx_buf), 50 * USEC_PER_MSEC);
            break;

        default:
            break;
    }
}
```

Da das Empfangen von Daten über die UART-Schnittstelle recht stromintensiv im Verbrauch ist, müssen wir den Empfang über den Aufruf der Funktion `uart_rx_enable` explizit aktivieren. Der letzte Parameter dieser Funktion ist ein Timeout-Parameter, der angibt nach welchem Zeitintervall der Eingabe des letzten Zeichens spätestens die Callback-Funktion aufgerufen wird:

```
int main(void){
    if (!device_is_ready(uart_dev)) {
        printk("uart_dev not ready\n");
        return 1;
    }

    uart_callback_set(uart_dev, uart1_cb, NULL);
    uart_rx_enable(uart_dev, rx_buf, sizeof(rx_buf), 50 * USEC_PER_MSEC);

    while (1) {
        k_msleep(SLEEP_TIME_MS);
    }
}
```

Nachdem der Empfang von Daten aktiviert wurde, wird auf die Eingabe von Zeichen gewartet. Bei Erhalt werden diese der Reihe nach im Buffer-Array gespeichert. Wurde nach 50ms kein weiteres Zeichen mehr empfangen, wird die Callback-Funktion mit dem Event `UART_RX_RDY` aufgerufen, durch welche die bereits erhaltenen Zeichen ausgegeben werden. Ist das Buffer-Array voll, d.h. es wurden mehr als 20 Zeichen empfangen, wird das Event `UART_RX_DISABLED` ausgelöst, der Buffer wird wieder freigegeben und der Empfang deaktiviert. Deswegen aktivieren wir bei diesem Event direkt wieder den Empfang und übergeben erneut den freigegeben Buffer.

Je nachdem wie schnell ein Sensor Daten sendet und wie viele, können und müssten beim Timeout und bei der Buffergröße je nach Anwendungsfall entsprechende Anpassungen vorgenommen werden. Für die händische Eingabe funktionieren die hier gewählten Werte aber recht gut. Allerdings werden Sie z.B. eine Verzögerung sehen, wenn Sie eine Taste permanent gedrückt halten. Reduzieren Sie den Timeout auf 10ms verschwindet aber auch diese Verzögerung.

2 Analog-Digital-Converter (ADC)

An einen Mikrocontroller können über verschiedene Eingänge wie z. B. UART, I²C, GPIOs oder ADCs Sensoren angeschlossen werden. Messergebnisse, die von einem Sensor in analoger Form geliefert werden,

müssen über einen *Analog-Digital-Converter* (ADC) in digitale Zahlenwerte konvertiert werden. Bevor ein Signal an den ADC übergeben werden kann, ist meist eine Signalanpassung z.B. über einen Verstärker oder eines Spannungsteilers nötig, so dass der Bereich der Eingangsspannung möglichst gut zum Spannungsbereich des ADCs passt.

Nehmen wir an, ein Temperatursensor kann Temperaturen von 0°C bis 100°C messen und liefert je nach gemessener Temperatur über einen Spannungsteiler eine Spannung zwischen 0V und 2V. Zur Vereinfachung gehen wir hier von einer linearen Temperaturkorrelation aus. Ein 8-Bit ADC hat die Möglichkeit, die angeschlossene Spannung als eine Zahl von 0 bis 255 darzustellen. Der ADC vergleicht die angeschlossene Spannung mit einer Referenzspannung. Die Referenzspannung kann ein interner Referenzwert, die Versorgungsspannung oder eine externe Spannungsquelle sein. Bei einer Referenzspannung von 2,56V liegt die Genauigkeit des ADCs in unserem Beispiel bei etwa 0,01V (2,56V/256). Der Temperatursensor liefert allerdings nur eine Spannung von 0V bis maximal 2V, d. h. in diesem Bereich stehen uns lediglich 200 ADC-Werte zur Verfügung und die maximale Genauigkeit der Temperaturmessung reduziert sich alleine hierdurch auf 100°C/200=0.5°C. Die Ungenauigkeit des Temperatursensors selbst beeinflusst die Messgenauigkeit zusätzlich. Die Spannung V_{ADC} aus einem ausgelesenen ADC-Wert i mit b-Bit Genauigkeit und einer Referenzspannung V_{ref} ergibt sich wie folgt:

$$V_{ADC} = \frac{V_{ref}}{2^b} \cdot i$$

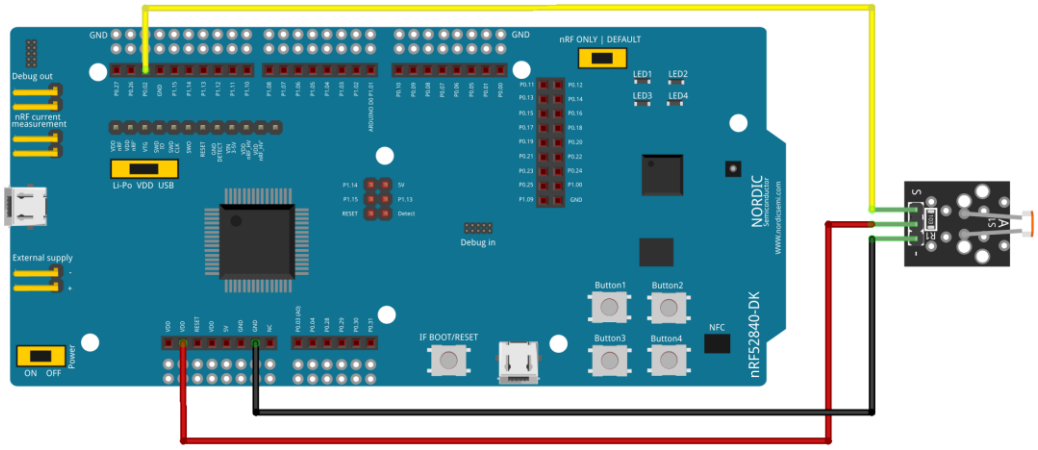
Aufgabe 3: Bestimmen Sie die Lichtintensität durch eines am ADC-Port des nRF52840 angeschlossenen Lichtsensors (LDR)

In Ihrem Set befindet sich ein Lichtsensor (*LDR*). Schließen Sie den Sensor an einen ADC-Port des nRF52840 Developerkits an, lesen Sie die ADC-Werte aus, rechnen diese in einen Spannungswert um und geben den ADC-Wert und die ADC-Spannung über die UART-Schnittstelle an einen PC aus. Machen Sie von der Ausgabe einen Screenshot und fügen Sie diesen Ihren Projektdateien hinzu.

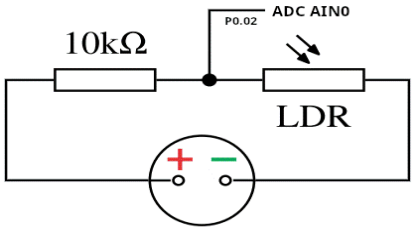
Der nRF52840 besitzt integriert einen ADC mit 8 Kanälen (Chanel). Auf jedem dieser Kanälen kann der ADC Messwerte einlesen und umwandeln. Dies geschieht intern allerdings nicht parallel, sondern in Reihe indem auf den entsprechenden Kanal mittels eines Multiplexer umgeschaltet wird. Zudem hat der nRF52840 8-Ports bzw. Pins, die als Eingabe für den ADC genutzt werden können. Die Pins können nach Belieben einem Kanal zugewiesen werden. Wir werden in diesem Beispiel Kanal 0 und den Port AIN0 nutzen. Die nachfolgende Tabelle zeigt die 8 ADC-Ports und ihre zugehörigen Pins an:

ADC-Port:	AIN0	AIN1	AIN2	AIN3	AIN4	AIN5	AIN6	AIN7
Pin:	P0.02	P0.03	P0.04	P0.05	P0.28	P0.29	P0.30	P0.31

Das Breakoutboard mit dem Fotowiderstand hat drei Anschlüsse. Den mit S gekennzeichneten Pin schließen Sie an den ADC-Pin AIN0 bzw. P0.02 des Mikrocontrollers an, den mittleren Pin an die positive 3V Versorgungspannung VDD und den mit Minus gekennzeichneten Pin an die Masse (GND):



Der Lichtsensor ist ein Fotowiderstand und ist als Widerstand eines Spannungsteilers konzipiert. Dadurch fließen, auch wenn sich der Widerstand des LDRs 0Ω annähert, keine hohen Ströme:



Um in Zephyr den ADC nutzen zu können, müssen wir die Funktionalität zunächst in der Projektkonfigurationsdatei *prj.conf* aktivieren:

```
CONFIG_ADC=y
```

Weiterhin müssen wir sicherstellen, dass der ADC im Devicetree aktiviert ist:

```
&adc {
    status = "okay";
};
```

Zunächst initialisieren wir das ADC-Device und definieren einige Makros mit den Parametern für den ADC-Zugriff:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/adc.h>
#include <zephyr/sys/printk.h>

#define SLEEP_TIME_MS   1000

#define ADC_NODE        DT_NODELABEL(adc)  //DT_N_S_soc_S_adc_40007000
static const struct device *adc_dev = DEVICE_DT_GET(ADC_NODE);

#define ADC_RESOLUTION      10
#define ADC_CHANNEL         0
#define ADC_PORT            SAADC_CH_PSELP_PSELP_AnalogInput0   //AIN0
#define ADC_REFERENCE       ADC_REF_INTERNAL                    //0.6V
#define ADC_GAIN            ADC_GAIN_1_5                        //ADC_REFERENCE*5
```

Der ADC des nRF52840 erlaubt eine Auflösung von bis zu 14-Bit. Uns genügen hier allerdings 10-Bit, d.h. unsere ADC liefert Werte von 0 bis $2^{10} - 1$ (1023). Wir benutzen den Kanal 0 und den Port AIN0. Als Referenzspannung, mit welcher der ADC das Eingangssignal vergleicht, nutzen wir das interne Spanungssignal des nRF52840, welches eine Spannung von 0.6V hat. Zusätzlich spezifizieren wir einen Multiplikatorfaktor von 5 (ADC_GAIN_1_5). Somit kann der ADC Eingangswerte von $0V$ bis $5 \cdot 0.6V = 3V$ umwandeln. Das passt für uns recht gut, da die VDD-Spannung 3V ist. Die Konfiguration des Kanals speichern wir in einer struct-Variablen:

```
struct adc_channel_cfg channel_cfg = {
    .gain = ADC_GAIN,
    .reference = ADC_REFERENCE,
    .acquisition_time = ADC_ACQ_TIME_DEFAULT,
    .channel_id = ADC_CHANNEL,
#ifdef CONFIG_ADC_NRFX_SAADC
    .input_positive = ADC_PORT
#endif
};
```

Der Parameter `input_positive` ist spezifisch für den ADC der nRF-Mikrokontroller. Diese können zwei Eingänge (`input_positive` und `input_negative`) bedienen und aus diesen die Differenz bilden. Wir benutzen hier allerdings nur einen Eingang. Als nächstes benötigen wir eine Datenstruktur zum Speichern der Ergebnisse der ADC-Umwandlung:

```
int16_t sample_buffer[1];
struct adc_sequence sequence = {
    .channels    = BIT(ADC_CHANNEL),
    .buffer      = sample_buffer,
    .buffer_size = sizeof(sample_buffer),
    .resolution  = ADC_RESOLUTION
};
```

Haben wir mehrere Kanäle definiert, können wir die Messwerte durch eine einzige Anweisung bestimmen lassen. Deswegen besteht der Buffer aus einem Array. Mit dem Parameter `channels werden` über eine or-Verknüpfung dann die zu messenden Kanäle festgelegt. Wir nutzen in diesem Beispiel allerdings nur einen Kanal.

Über die Funktion `adc_channel_setup` setzen wir in der main-Funktion die Parameter des Kanals und mit der Funktion `adc_read` initialisieren wir eine Messung:

```
int main(void){
    if (!device_is_ready(adc_dev)) {
        printk("adc_dev not ready\n"); return 1;
    }

    int err;
    err= adc_channel_setup(adc_dev, &channel_cfg);
    if (err != 0) {
        printk("ADC adc_channel_setup failed with error %d.\n", err); return 1;
    }

    while (1) {
        err = adc_read(adc_dev, &sequence);
```

```
        if (err != 0) {
            printk("ADC reading failed with error %d.\n", err); return 1;
        }
        printk("ADC-Wert: %d\n", sample_buffer[0]);
        k_msleep(SLEEP_TIME_MS);
    }
}
```

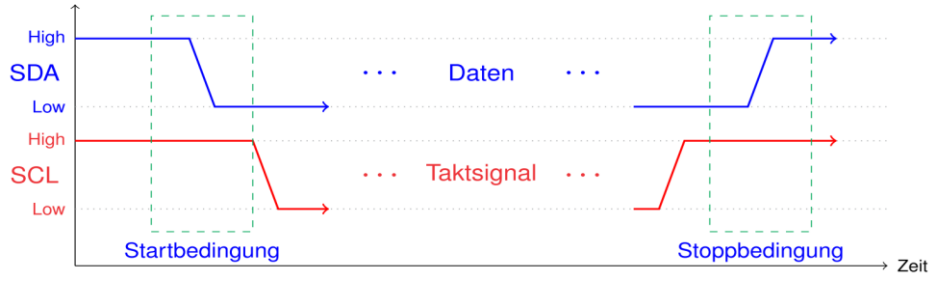
Den ADC-Wert könnten wir in einen Spannungswert umwandeln, indem wir diesen durch 2^{10} teilen und mit der Spannung 3V multiplizieren. Zephyr stellt uns allerdings auch eine praktische Umrechnungsfunktion zur Verfügung:

```
int32_t mv_value = sample_buffer[0];
int32_t adc_vref = adc_ref_internal(adc_dev);
adc_raw_to_millivolts(adc_vref, ADC_GAIN,   ADC_RESOLUTION, &mv_value);
printk("ADC-Spannung: %d mV  \n", mv_value);
```

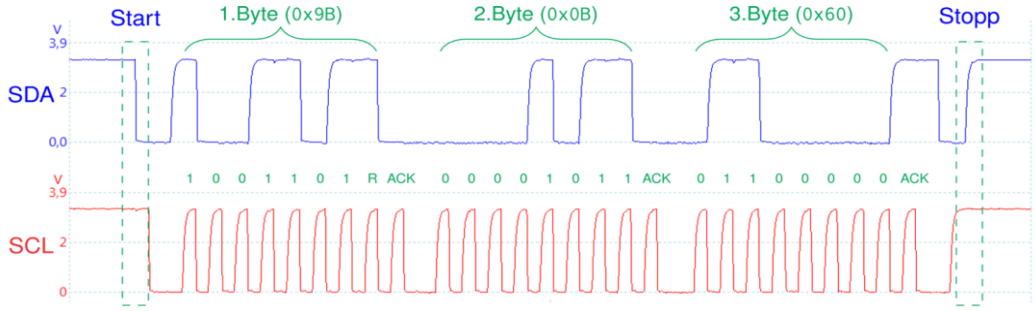
3 I²C-Bus

Der I²C-Bus ist eine weitere Möglichkeit Sensoren an einen Mikrocontroller anzuschließen. Hierbei handelt es sich um ein Bussystem mit zwei Leitungen, eine Datenleitung und eine CLK-Leitung zur Steuerung des Zeittaktes. Im Allgemeinen übernimmt der Mikrocontroller die Rolle des Controllers, er generiert das Taktsignal und liefert auch das Grundsignal für die Datenleitung. Wenn angeschlossene Geräte (Target) Daten senden, müssen sie die Datenleitung lediglich auf Masse (GND) ziehen und müssen keine Spannungssignale selbst erzeugen.

An den I²C-Bus können bis zu 112 Sensoren gleichzeitig angeschlossen werden. Übertragen werden Daten auf dem Bus immer in Oktetten, d.h. 8 Bits. Adressiert werden die Sensoren über eine 7-Bit Adresse. Der I²C-Temperatursensor LM73 hat z. B. die Adresse 0x4D. Eingeleitet wird eine Übertragung durch die sogenannte *Startbedingung* und beendet durch die *Stoppbedingung*. Bei der Startbedingung ist die CLK-Leitung auf High und die Datenleitung wird von High auf Low gezogen. Bei der Stoppbedingung muss die CLK-Leitung wieder auf High liegen und die Datenleitung wird von Low auf High gezogen.



Will der Mikrocontroller Daten an einen Sensor senden, wird zuerst die Startbedingung erfüllt. Um einen bestimmten Sensor anzusprechen, wird anschließend dessen Adresse (LM73: 0x4D = 0b1001101) plus einem R/W-Bit mit dem Wert 1 (Readbit) auf den Datenbus gesendet (LM73: 0x9B = 0b10011011). Das höchstwertige Bit (MSB) wird immer zuerst gesendet. Ein Empfänger eines Bytes bestätigt den Empfang eines 8-Bit Datenpaketes dadurch, dass er die Datenleitung auf Low zieht (ACK), sofern er weitere Datenpakete empfangen kann und mit einem High, wenn er keine weiteren Daten mehr empfangen kann oder möchte. Nach der Leseanweisung für einen Sensor sendet dieser an den Mikrocontroller die angeforderten Daten. Beim LM73 sieht eine beispielhafte Datenkommunikation aufgenommen mit einem Oszilloskop wie folgt aus:

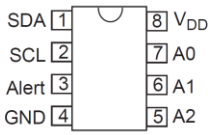


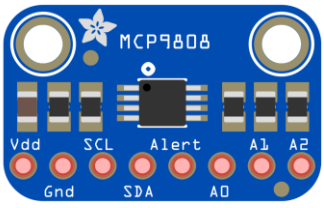
Für das Senden von Daten vom Mikrocontroller an einen Sensor wird das R/W-Bit auf den Wert 0 gesetzt. Nach dem ACK des Sensors, kann der Mikrocontroller entsprechende 8-Bit Blöcke an den Sensor senden.

Aufgabe 4: Lesen Sie mit dem nRF52840 den I²C-Temperatursensor MCP9808 aus

Schließen Sie den MCP9808 an den I²C-Bus des nRF52840 an, ermitteln mit diesem durch die Ansteuerung über die I2C-Schnittstelle alle 2 Sekunden die aktuelle Temperatur und geben diese über die UART-Schnittstelle an Ihrem PC aus. Erstellen Sie einen Screenshot ihrer ausgelesenen Sensorwerte und legen Sie diesen ihrem Projektordner bei.

Der MCP9808 ist ein I²C-Temperatursensor der Firma *Microchip* mit einer typischen Genauigkeit von 0,25°C. Wir nutzen hier ein Breakoutboard der Firma *Adafruit*, welches mit diesem Temperatursensor bestückt ist:



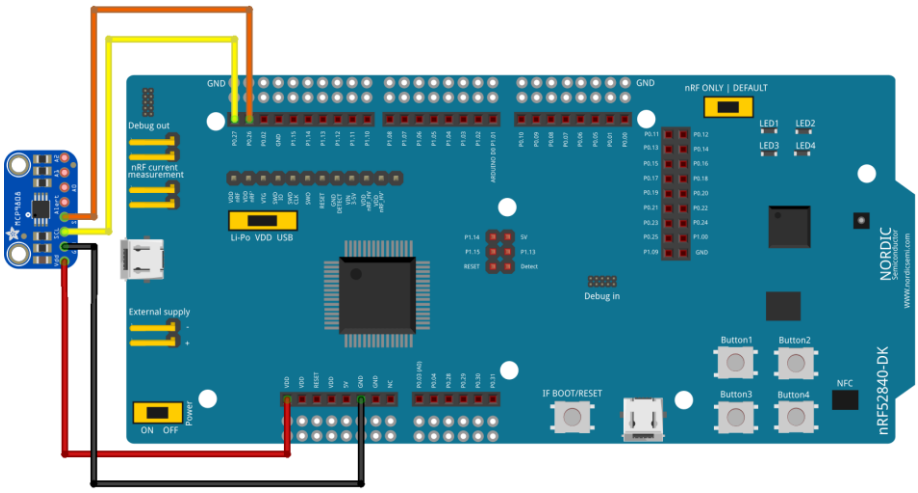


Um die Eigenschaften eines Sensors zu bestimmen oder um zu erfahren wie ein Sensor angesteuert wird, sollte wir immer erst ein Blick in das Datenblatt[3] werfen. Normerweise finden wir dies auf der Herstellerseite des Sensors.

[3] https://www.microchip.com/wwwproducts/en/en556182

Die Pins A_0, A_1, und A_2 definieren die I²C-Adresse (0b0011A_2 A_1 A_0) des MCP9808. Auf dem Board von Adafruit sind diese mit drei *Pulldown*-Widerständen auf Masse (GND) gezogen, was zu eine I²C-Adresse von $0b0011000 = 0x18$ führt.

Bevor wir den MCP9808 mit dem nRF52840 ansteuern können, müssen wir diesen entsprechend verbinden. VDD und GND wird für die Spannungsversorgung und SDA und SCL für den Anschluss an den I²C-Bus benötigt. Der MCP9808 verträgt eine Eingangsspannung von 2,7V bis 5,5V. Am nRF5820-Developerboard benutzen wir in diesem Beispiel Pin P0.26 für die SDA-Leitung und P0.27 für die SCL-Leitung:



Der MCP9808 ist ein relativ einfach ansteuerbarer I²C-Temperatursensor. Er hat zwar einige Einstellmöglichkeiten und Register, die sich manipulieren lassen, im Grundzustand ist er aber schon so konfiguriert, dass wir direkt das Temperaturregister $0x05$ auslesen können. Der Wert des Temperaturregisters wird bei der zu Beginn eingestellten Auflösung von 0,0625°C etwa alle 250ms automatisch mit einem neuen aktuellen Wert gefüllt.

Zunächst setzen wir zwei Parameter in der Projektkonfigurationsdatei:

```
CONFIG_I2C=y
CONFIG_CBPRINTF_FP_SUPPORT=y
```

Der erste Parameter aktiviert die I²C-Funktionalität in Zephyr und der zweite Parameter ermöglicht eine Ausgabe von *float*-Variablen über die Funktion `printk`, was bei der Temperaturausgabe sehr hilfreich ist.

Als Makros definieren wir die Adresse des MCP9808 und das Temperaturregister. Zudem generieren wir ein I²C-Device und deklarieren ein zwei Byte großes Buffer-Array zum Austausch der Daten mit dem Temperatursensor:

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/drivers/i2c.h>

#define SLEEP_TIME_MS                   1000

#define MCP9808_I2C_ADDRESS             0x18
#define MCP9808_TEMPERATURE_REGISTER    0x05

#define I2C_NODE        DT_NODELABEL(i2c0)          //DT_N_S_soc_S_i2c_40003000
static const struct device *i2c_dev = DEVICE_DT_GET(I2C_NODE);
static uint8_t i2c_buffer[2];
```

Aus dem Temperaturregister werden wir 2 Bytes im Integer-Format auslesen. Um den Temperaturwert als reelle Zahl zu erhalten, müssen diese Bytes richtig zusammengesetzt werden. Nach Datenblatt ist zur Berechnung der Temperatur T ≥0 die folgende Formel anzuwenden:

$$T = upperByte \times 2^4 + lowerByte \times 2^{-4}$$

Und für eine Temperatur T <0:

$$T = 256 - (upperByte \times 2^4 + lowerByte \times 2^{-4})$$

Ob es sich um einen negativen oder positiven Temperaturwert handelt, kann aus dem 4.ten Bit des *upperBytes* herausgelesen werden. Als Funktion implementiert sieht das Ganze wie folgt aus:

```
float mcp9808_calculateTemperature(uint8_t upperByte, uint8_t lowerByte){
    float temperature = (upperByte&0x0F)*16+(float)(lowerByte)/16;
    if ((upperByte & 0x10) == 0x10){ //Temperatur < 0°C
        temperature=256-temperature; //von 2^8 subtrahieren, da Zweierkompliment
    }
    return temperature;
}
```

Um den Sensor auszulesen sind zwei Schritte notwendig. Zuerst senden wir über die Funktion `i2c_write` ein Byte mit der Adresse des Temperaturregisters an den MCP9808 um dessen Pointer auf das entsprechende Register zu setzen. Anschließend weisen wir den MCP9808 über die Funktion `i2c_read` an, uns den Inhalt der zwei Bytes des Temperaturregisters zu senden:

```
i2c_write(i2c_dev, i2c_buffer, 1, MCP9808_I2C_ADDRESS);
i2c_read(i2c_dev,i2c_buffer,2,MCP9808_I2C_ADDRESS);
```

Mit Umwandlung und Fehlerbehandlung sieht unser main-Funktion dann wie folgt aus:

```
int main(void){

    int err;
    if (!device_is_ready(i2c_dev)) {
        printk("i2c_dev not ready\n"); return 1;
    }

    while (true) {
        i2c_buffer[0]=MCP9808_TEMPERATURE_REGISTER;

        do{
            err = i2c_write(i2c_dev, i2c_buffer, 1, MCP9808_I2C_ADDRESS);
            if (err < 0){ printk("MCP9808 write failed: %d\n", err); break; }

            err = i2c_read(i2c_dev,i2c_buffer, 2, MCP9808_I2C_ADDRESS);
            if (err < 0){ printk("MCP9808 read failed: %d\n", err); break; }

            float temperature = mcp9808_calculateTemperature(i2c_buffer[0],i2c_buffer[1]);
            printk("MCP9808: %.2f Cel \n", temperature);

        }while (false);
        k_msleep(SLEEP_TIME_MS);
    }
}
```

Zu beachten ist der Trick mit der *do-while*-Schleife zur Fehlerbehandlung. Diese wird nur einmal ausgeführt, tritt aber ein Fehler auf wird diese abgebrochen.

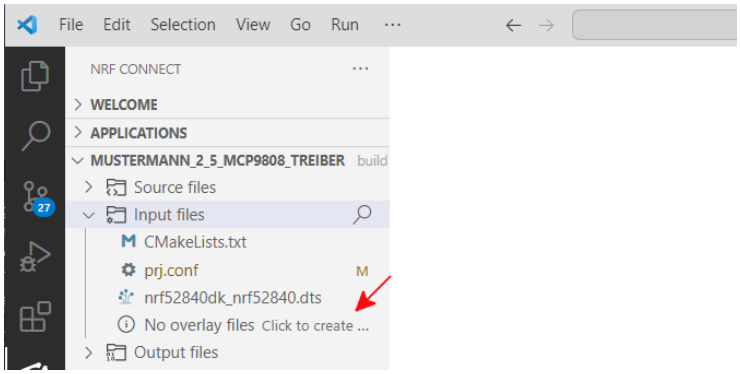
4 Temperatursensor MCP9808 als Gerät über einen Treiber auslesen

Wir haben im vorherigen Beispiel den Temperatursensor MCP9808 direkt über die I²C-Schnittstelle und die zugehörigen Funktionen von Zephyr angesteuert. Sofern wir das Konzept von I²C verstehen, halte ich diese Vorgehensweise häufig für die Sinnvollste. Wir lernen hierbei am besten den Sensor kennen und senden nur die notwendigsten Befehle. In Zephyr gibt es aber auch die Möglichkeit den Sensor über einen Treiber und den Devicetree auszulesen. Für erstaunlich viele Sensoren sind in Zephyr bereits Treiber integriert. Mit diesen Treibern können wir recht schnell einen Sensor in Betrieb nehmen und diesen testen. Wir wollen uns dies am Beispiel des Temperatursensors MCP9808 anschauen.

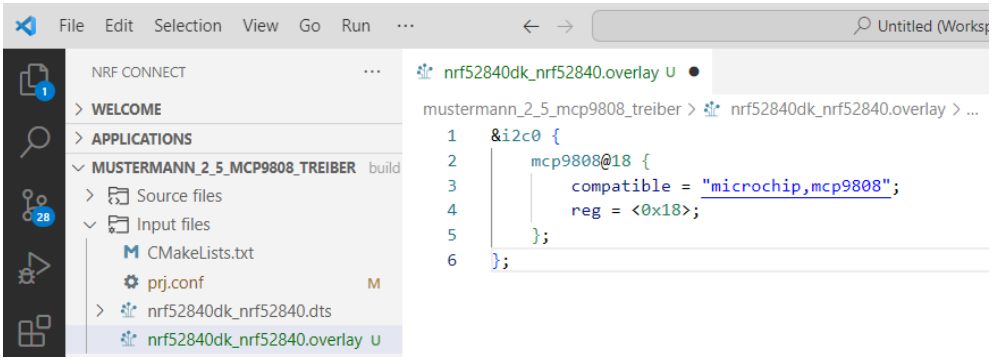
Aufgabe 5: Lesen Sie mit dem nRF52840 den Temperatursensor MCP9808 über die Treiber von Zephyr aus

Ermitteln Sie mit dem MCP9808 wie in der letzten Aufgabe die Temperatur, benutzen Sie allerdings zur Ansteuerung die Treiber des MCP9808 von Zephyr. Erstellen Sie einen Screenshot ihrer ausgelesenen Sensorwerte und legen Sie diesem ihrem Projektordner bei.

Damit wir den MCP9808-Treiber nutzen können und es beim Kompilieren zu keiner Fehlermeldung kommt, müssen wir die Hardware im Devicetree definieren. Hierzu bearbeiten wir nicht direkt den Devicetree, sondern erstellen eine sogenannte Overlay-Datei, die entsprechende Parameter hinzufügt oder überschreibt. Dazu klicken wir bei unserem Projekt unter den *Input Files* auf *No overlay files Click to create one*:



Dadurch wird in unserm Projektordner die Datei nrf52840dk_nrf52840.overlay erstellt. In diese Datei erweitern wir den i2c0-Knoten mit der Ressource des MCP9808:



Anschließend setzen wir in der Projektkonfigurationsdatei *prj.conf* folgende Parameter, um die benötigten Bibliotheken ins Projekt einzubinden:

```
CONFIG_I2C=y
CONFIG_SENSOR=y
CONFIG_MCP9808=y
CONFIG_CBPRINTF_FP_SUPPORT=y
```

Die Ansteuerung des MCP9808 erfolgt, wie wir in Zephyr bereits gewohnt sind, über ein Device und den Aufruf entsprechender Funktionen. Über die Funktion `sensor_sample_fetch` erhalten wir Sensorwerte vom MCP9808 und speichern diese im Buffer des Treibers. Anschließend können wir die einzelnen Sensordaten über die Funktion `sensor_channel_get` abfragen:

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/drivers/sensor.h>

#define SLEEP_TIME_MS   1000

const struct device *mcp9808_dev = DEVICE_DT_GET_ANY(microchip_mcp9808);
static struct sensor_value temp;

int main(void){
    if (!device_is_ready(mcp9808_dev)) {
        printk("mcp9808_dev not ready\n"); return 1;
    }

    int err;

    while (true) {

        do{
            err = sensor_sample_fetch(mcp9808_dev);
            if (err < 0){ printk("MCP9808 write failed: %d\n", err); break; }

            err = sensor_channel_get(mcp9808_dev, SENSOR_CHAN_AMBIENT_TEMP, &temp);
            if (err < 0){ printk("MCP9808 read failed: %d\n", err); break; }

            printk("MCP9808: %.2f Cel\n", sensor_value_to_double(&temp));
        }while (false);

        k_msleep(SLEEP_TIME_MS);
    }
}
```

5 Scheduling Tasks (Timer, Work Queues und Threads)

Zuweilen wollen wir Tasks in periodischen Abständen aufrufen, wie im vorherigen Beispiel das Auslesen von Sensorwerten. Bei Mikrocontrollern ist das gar nicht so trivial, da der aktuelle Programmfluss unterbrochen werden muss und dabei andere Aufgaben blockiert werden könnten. In der herkömmlichen Mikrocontroller-programmierung gibt es im allgemeinen keine Threads und keinen Taskmanager. Der Programmfluss wird durch Interrupt-Subroutinen unterbrochen und muss dann so gesteuert werden, dass wichtige Aufgaben nicht zu lange blockiert werden. Zephyr unterstützt Multithreading und nimmt uns hier einige Arbeit ab. Unsere *main*-Funktion ist auch nur ein weiterer Thread, der neben anderen Threads läuft. Durch die Funktion `k_msleep` bringen wir unseren Thread in den Ruhemodus und andere Threads können derzeit bearbeitet werden. So nutzen wir quasi die einfachste Möglichkeit eine Aufgabe periodisch ausführen zu lassen. Haben wir aber mehrere Aufgaben in unterschiedlichen Intervallen auszuführen wird diese Art der Programmierung schwierig. Um unterschiedliche Aufgaben in bestimmten Abständen durchzuführen stehen uns Timer zur Verfügung.

5.1 Timer

Ein Timer ruft nach Ablauf einer bestimmten Zeit eine zuvor registrierte Funktion aus. Der Aufruf dieser Funktion kann entweder einmalig oder periodisch stattfinden. Wir werden uns hier eine Kopie der vorherigen Aufgabe machen und einen Timer implementieren, der eine Led blinken lässt. Dazu fügen wir wieder die Funktionalität der Led in unseren Programmcode hinzu:

```
#include <zephyr/drivers/gpio.h>
#define LED0_NODE   DT_NODELABEL(led0)                  //DT_N_S_leds_S_led_0
static const struct gpio_dt_spec led0_spec = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
```

Anschließend definieren wir die Callback-Funktion, welche aufgerufen werden soll, wenn der Timer abläuft:

```
void mytimer_cb(struct k_timer *dummy){
    gpio_pin_toggle_dt(&led0_spec);
}
```

Die Timervariable `mytimer` vom Typ `k_timer` könnten wir selbst implementieren und initialisieren oder wir können wie häufig in Zephyr einfacher ein Makro dafür nutzen:

```
K_TIMER_DEFINE(mytimer, mytimer_cb, NULL);
```

In der main-Funktion, müssen wir den LED-Port als Ausgang definieren und den Timer starten:

```
gpio_pin_configure_dt(&led0_spec, GPIO_OUTPUT);
k_timer_start(&mytimer, K_SECONDS(1), K_SECONDS(1));
```

Übertragen wir dieses Programm auf unser Board blinkt die LED1 im Sekundentakt und gibt weiterhin die Temperatur des MCP9808 aus. Jetzt könnten wir auf die Idee kommen den Code für den MCP9808 in die Callback-Funktion des Timers zu packen. Dies wird allerdings nicht funktionieren. Timer lösen bei Ablauf einen Interrupt aus und unterbrechen andere anstehende Aufgaben. Deswegen sollten die Aufgaben, die in dieser Callback ausgeführt werden, so kurz wie möglich sein. Ausgabe über die UART- oder die I²C-Schnitt-stelle sind aber bereits zu viel für eine sogenannte ISR[4] in Zephyr. Die Lösung hierfür ist die Nutzung einer Work Queue.

[4] Interrupt Service Routine

5.2 Work Queue

Komplexere Aufgaben, die mehr Zeit in Anspruch nehmen, können wir in eine *Work Queue* auslagern. Das Ist eine Queue, die nach dem FIFO[5]-Prinzip funktioniert. Wir reihen eine anstehende Aufgabe (*work*) in diese Queue ein und diese wird ausgeführt, sobald diese an der Reihe ist. Für dieses Konzept müssen wir zunächst eine Callback-Funktion definieren, welche ausgeführt werden soll, sobald die Aufgabe (*work*) an der Reihe ist:

```
static void mcp9808_work_cb(struct k_work *work){
    int err;
    if (!device_is_ready(mcp9808_dev)) {
        printk("mcp9808_dev not ready\n"); return;
    }
    do{
        err = sensor_sample_fetch(mcp9808_dev);
        if (err < 0){ printk("MCP9808 write failed: %d\n", err); break; }

        err = sensor_channel_get(mcp9808_dev, SENSOR_CHAN_AMBIENT_TEMP, &temp);
        if (err < 0){ printk("MCP9808 read failed: %d\n", err); break; }

        printk("MCP9808: %.2f Cel\n", sensor_value_to_double(&temp));
    }while (false);
}
```

Anschließend definieren und initialisieren wir die *work*-Variable über ein Makro:

```
K_WORK_DEFINE(mcp9808_work, mcp9808_work_cb);
```

Jetzt passen wir die Callback-Funktion des Timers so an, dass er die entsprechende Aufgabe `mcp9808_work` in die Warteschlange schiebt:

```
void mytimer_cb(struct k_timer *dummy){
    gpio_pin_toggle_dt(&led0_spec);
    k_work_submit(&mcp9808_work);
}
K_TIMER_DEFINE(mytimer, mytimer_cb, NULL);
```

Aus der main-Funktion verschwinden dementsprechend die Aufgaben für den Temperatursensor:

```
int main(void){
    gpio_pin_configure_dt(&led0_spec, GPIO_OUTPUT);
    k_timer_start(&mytimer, K_SECONDS(2), K_SECONDS(2));
    while (true) {
        k_msleep(SLEEP_TIME_MS);
    }
}
```

[5] First In/First Out

Aufgabe 6: Nutzen Sie zum Auslesen des Temperatursensor MCP9808 einen Timer und eine Work Queue

Ermitteln Sie mit dem MCP9808 wie in der letzten Aufgabe die Temperatur, benutzen Sie hierfür allerdings einen Timer und eine Work Queue. Erstellen Sie einen Screenshot ihrer ausgelesenen Sensorwerte und legen Sie diesem ihrem Projektordner bei.

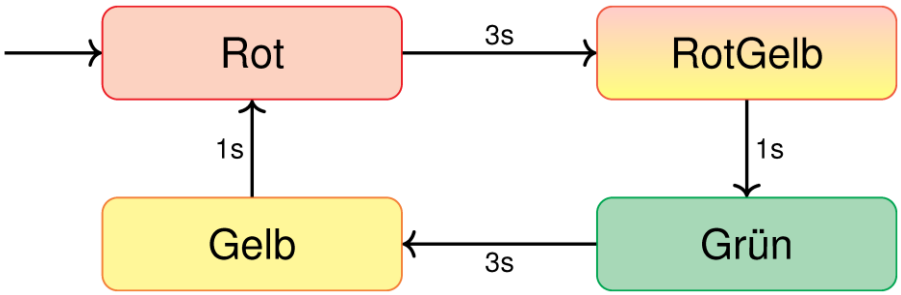
Wir nutzen hier die *System Work Queue*, welche auch für anstehende Systemaufgaben genutzt wird. Auch hier sollten wir die *Work Queue* nicht zu lange blockieren. Für diesen Fall können wir eine eigene *Applikations Work Queue* implementieren oder gar einen eigenen Thread erstellen.

5.3 Threads

In Zephyr können wir eigene Threads definieren. Wie bei einem Work wird hier eine Callback-Funktion definiert, in der die zu erledigenden Aufgaben sind. Allerdings terminiert diese Funktion nicht, sondern, beinhaltet wie unser main-Thread eine unendliche *while*-Schleife die nicht terminiert. Für jeden Thread müssen wir zudem eigenen Speicherplatz reservieren. Die Kommunikation zwischen unterschiedlichen Threads ist deswegen auch nicht ganz einfach. Für unsere Sensoranwendungen ist im Allgemeinen die Nutzung einer Work Queue die bessere Lösung.

5.4 Endliche Automaten

Eine elegante Möglichkeit ein Programm und den Prozessablauf zu strukturieren ist die Nutzung von endlichen Automaten (engl. *finite state machine* oder kurz *FSM*). Eine Verkehrsampelschaltung kann als FSM wie folgt modelliert werden:



Wir können so einen Automaten recht einfach über eine *switch*-Anweisung und einer sich verändernden Timer-Variablen realisieren.

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>

#define RED_SLEEP_TIME_MS           3000
#define RED_YELLOW_SLEEP_TIME_MS    1000
#define GREEN_SLEEP_TIME_MS         3000
#define YELLOW_SLEEP_TIME_MS        1000

typedef enum{
    RED,
    RED_YELLOW,
    GREEN,
    YELLOW
} AppState_t;
```

```
static AppState_t state = RED;

int main(void){
    while (1) {
        switch(state){
            case RED:
                printk("Rot\n");
                state=RED_YELLOW;
                k_msleep(RED_SLEEP_TIME_MS);
                break;

            case RED_YELLOW:
                printk("RotGelb\n");
                state=GREEN;
                k_msleep(RED_YELLOW_SLEEP_TIME_MS);
                break;

            case GREEN:
                printk("Grün\n");
                state=YELLOW;
                k_msleep(GREEN_SLEEP_TIME_MS);
                break;

            case YELLOW:
                printk("Gelb\n");
                state=RED;
                k_msleep(YELLOW_SLEEP_TIME_MS);
                break;
        }
    }
}
```

Um einen FSM zu implementieren stellt Zephyr auch das Framework SMF zur Verfügung. Die entsprechende Funktionalität aktivieren wir in der Projektdatei:

```
CONFIG_SMF=y
```

In der main-Datei fügen wir den Header der SMF-Bibliothek hinzu und definieren ein Array für die Zustände. Für eine übersichtliche Programmierung greifen wir nicht per Indexnummer auf die Zustände zu, sondern definieren einen *enum*-Datentyp mit den Zustandsnamen:

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/smf.h>

#define RED_SLEEP_TIME_MS           3000
#define RED_YELLOW_SLEEP_TIME_MS    1000
#define GREEN_SLEEP_TIME_MS         3000
#define YELLOW_SLEEP_TIME_MS        1000

static const struct smf_state states[];

enum state { RED, RED_YELLOW , GREEN, YELLOW};
```

Jeder Zustand wird im SMF durch eine eigene Funktion implementiert, die als Übergabeparameter einen Datetyp enthält in dem kontextabhängige Parameter übergeben werden können. Wir benötigen diese Parameter nicht, müssen dieses Objekt aber trotzdem für die interne Struktur des SMF initialisieren:

```
struct s_object {
```

```
   struct smf_ctx ctx; /* This must be first */
   /* Other state specific data add here */
} s_obj;
```

Um die Zustände in unterschiedlichen Zeitintervall ausführen zu können, benötigen wir eine entsprechende Variable:

```
int32_t sleep_msec;
```

Anschließend implementieren wir die einzelnen Zustände durch Funktionen:

```
static void red_run(void *o){
    printk("RED\n");
    sleep_msec=RED_SLEEP_TIME_MS;
    smf_set_state(SMF_CTX(&s_obj), &states[RED_YELLOW]);
}

static void red_yellow_run(void *o){
    printk("RED YELLOW\n");
    sleep_msec=RED_YELLOW_SLEEP_TIME_MS;
    smf_set_state(SMF_CTX(&s_obj), &states[GREEN]);
}

static void green_run(void *o){
    printk("GREEN\n");
    sleep_msec=GREEN_SLEEP_TIME_MS;
    smf_set_state(SMF_CTX(&s_obj), &states[YELLOW]);
}

static void yellow_run(void *o){
    printk("YELLOW\n");
    sleep_msec=YELLOW_SLEEP_TIME_MS;
    smf_set_state(SMF_CTX(&s_obj), &states[RED]);
}
```

Wie wir sehen, wird in jedem Zustand der Folgezustand und die entsprechende Wartezeit gesetzt. Die einzelnen Funktionen werden dann dem entsprechenden Zustand im Zustandsarray zugewiesen:

```
static const struct smf_state states[] = {
        [RED] = SMF_CREATE_STATE(NULL, red_run, NULL),
        [RED_YELLOW] = SMF_CREATE_STATE(NULL, red_yellow_run, NULL),
        [GREEN] = SMF_CREATE_STATE(NULL, green_run, NULL),
        [YELLOW] = SMF_CREATE_STATE(NULL, yellow_run, NULL),
};
```

Wir nutzen hier nur die Funktion für den eigentlichen Zustand. Das Makro `SMF_CREATE_STATE` hat allerdings drei Parameter, da auch Funktionen definiert werden können, die beim Eintreten und/oder beim Verlassen des jeweiligen Zustandes ausgeführt werden.

In der main-Funktion müssen wir jetzt nur den Startzustand festlegen, jeweils den aktuellen Zustand des Automaten ausführen lassen und die Wartezeit zum nächsten Zustand einhalten:

```
int main(void){
     int32_t ret;
    /* Set initial state */
    smf_set_initial(SMF_CTX(&s_obj), &states[RED]);
    while (1) {
        ret = smf_run_state(SMF_CTX(&s_obj));
```

```
        if (ret) { break; }
        k_msleep(sleep_msec);
    }
}
```

Aufgabe 7: Programmieren Sie eine simulierte Ampelschaltung mit dem State-Machine-Framework (SMF)

Implementieren Sie die obige Ampelschaltung mit dem SMF. Erstellen Sie einen Screenshot ihrer Ausgabe und legen Sie diesem ihrem Projektordner bei.