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

Projekt im WiSe 23/24
WSN
Kontakt:wsnprojekt@fb2.fra-uas.de

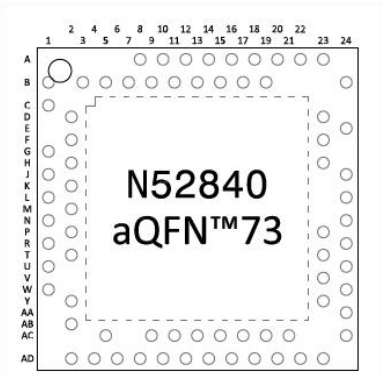
Blatt 1

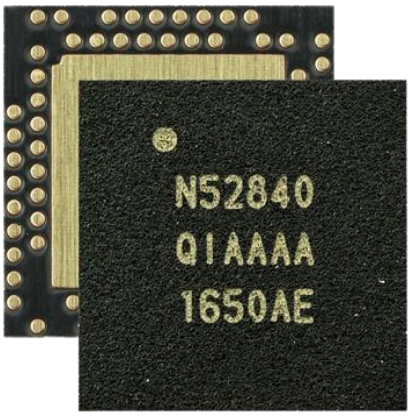
Einführung in Programmierung eingebetteter Systeme mittels Mikrocontroller (SoC) nRF52840

Sie haben von uns drei Developer Kits und ein USB-Dongles erhalten, die auf dem Mikrocontroller *nRF52840* der Firma *Nordic Semiconductor* basieren. Ihre Aufgabe ist es, sich in das Software Developer Kit (SDK) für den Mikrocontroller nRF52840 einzuarbeiten, externe Hardware wie z.B. Leds, Buttons und Sensoren anzusteuern und Daten drahtlos über das *Thread*-Netzwerkprotokoll zu übertragen. Das erste Übungsblatt soll Ihnen die ersten Grundlagen für die Programmierung des Mikrocontroller nRF52840 vermitteln.

1 Grundlagen und Vorbereitung

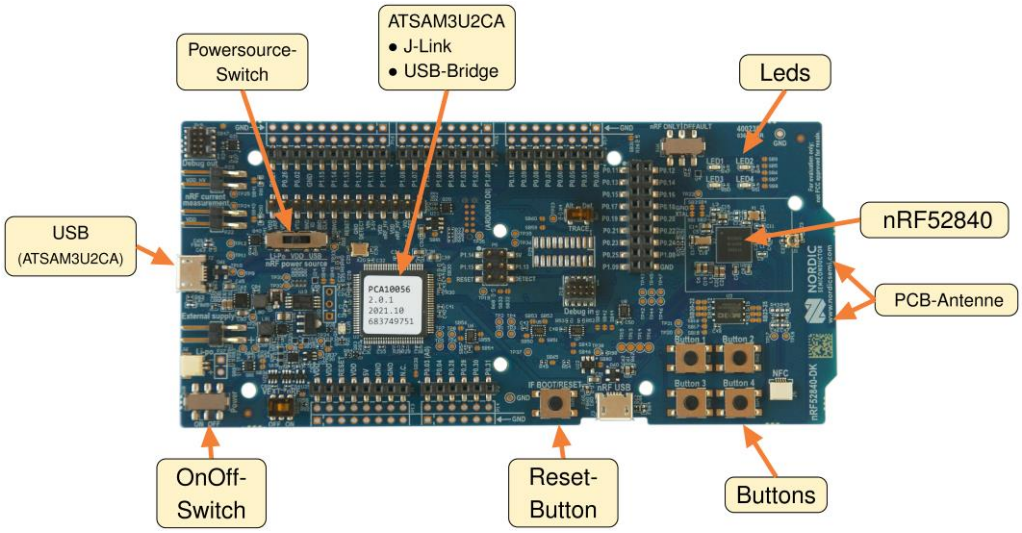
Jedes Funkmodul in Ihrem Set basiert auf dem Mikrocontroller nRF52840 von Nordic Semiconductor.





Der nRF52840 besteht aus einem *ARM Cortex M4* 32-Bit-Prozessor mit 1 MB Flashspeicher für den Programmcode und 256 kB Arbeitsspeicher. Zudem besitzt er eine 2,4 GHz Transceivereinheit, mit der Bluetooth Low Energy, Bluetooth Mesh, IEEE 802.15.4, NFC, ANT, ZigBee und Thread-Netzwerke realisiert werden können. Solch leistungsfähige Mikrocontroller werden schon allein, um sich von herkömmlichen Mikrocontrollern abzugrenzen, auch SoCs (*System on a Chip*) genannt. Ein SoC zeichnet sich insbesondere dadurch aus, dass viele Funktionalitäten auf einem einzigen Chip (*Die*) untergebracht werden, wie z.B. Transceiver, Prozessor, Speicher aber auch sowas, wie ein DMA-Modul (direkter Speicherzugriff für Peripheriegeräte). Es handelt sich bei dem Begriff SoC aber eher um eine marketingtechnische Bezeichnung als eine klare Definition und Abgrenzung zum Mikrocontroller.

In diesem Projekt benutzen wir das Developer Kit *nRF52840-DK*. Dieses Kit bietet uns neben einem verbauten J-Link zum Programmieren des nRF52840 zusätzlich 4 LEDs, 4 Buttons, etliche Pinanschlüsse, einen USB-Port und einiges mehr.



2 Installation der Software

Nordic Semiconductor ermöglicht es Software für ihre SoCs auf den Betriebssysteme Windows, Linux und MacOS zu entwickeln. Der Installationsprozess für die Entwicklungsumgebungen ist zwar ähnlich unterscheidet sich zum Teil aber auch etwas. Wir beschreiben hier den Installationsprozess für ein Windows 10 64-Bit Betriebssystem. Für andere Betriebssysteme laden Sie sich die passende Software herunter und überprüfen ggf. die Installationshinweise auf den entsprechenden Webseiten.

2.1 Installieren der nRF Command Line Tools

Das Command Line Tool enthält einige Werkzeuge für das Programmieren und Debuggen von Mikrocontrollern der Firma Nordic. Die wichtigste Komponente ist das dort enthaltene Programm *nrfprog.exe*, welches das Programmieren von Mikrocontroller mit einem J-Link Programmiergerät der Firma *Segger* über die Kommandozeile ermöglicht. Laden Sie das nRF Command Line Tool[1] für Ihr Betriebssystem herunter und installieren dieses mit den Standardparametern. Wir nutzen hier die *Version 10.23.0 x64*. Das Tool installiert im Übrigen auch die Software *Segger J-Link*, welche das Programmieren bzw. Flashen von Mikrocontrollern mit einem J-Link Programmiergerät über eine grafische Oberfläche ermöglicht. In dem Developerboard ist ein solcher J-Link-Programmer integriert.

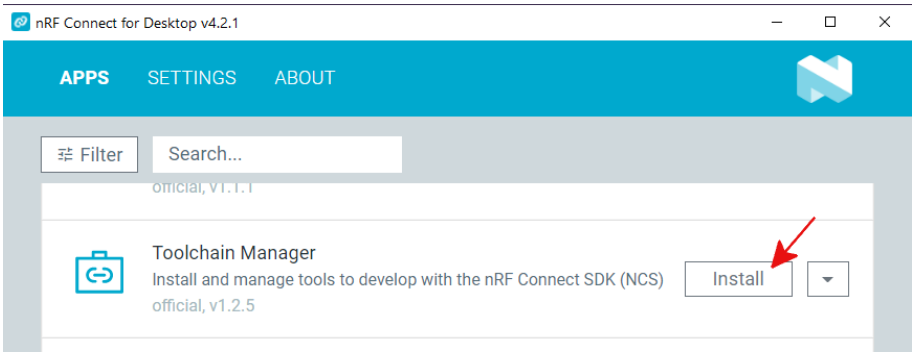
2.2 Installieren von nRF Connect for Desktop

nRF Connect for Desktop ist ein Framework von Nordic Semiconductor für Mikrocontroller der nRF-Serie. Es stellt eine Reihe nützlicher Tools zur Auswahl und übernimmt deren Installation. Laden Sie sich nRF Connect for Desktop[2] für ihr Betriebssystem herunter und installieren Sie es mit den Standardparametern auf ihrem Computer. Wir verwenden hier *nRF Connect* in der *Version v4.2.0*.

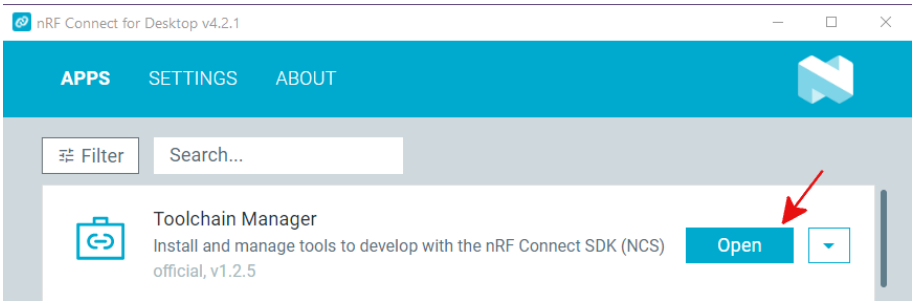
Nach dem Start der Anwendung sehen Sie eine Übersicht mit verschiedenen Programmen. Uns interessiert hier zunächst nur der *Toolchain Manager*. Eine Toolchain ist dafür verantwortlich, aus unserem Sourcecode

[1] https://www.nordicsemi.com/Products/Development-tools/nrf-command-line-tools/download
[2] https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-desktop

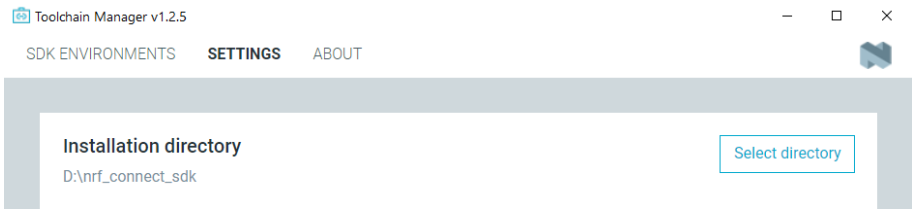
eine ausführbare Anwendung zu kompilieren. Der Toolchain Manager übernimmt die Installation und Verwaltung der Toolchains und des zugehörigen Software Development Kits (SDK). Durch Klicken auf die Schaltfläche *Install* wird der Manager installiert.



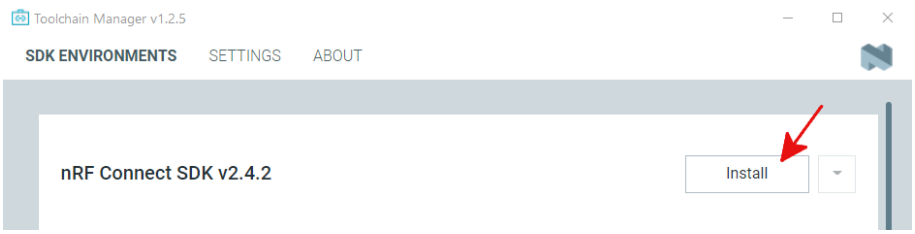
Ist die Installation erfolgreich abgeschlossen, erscheint der Button *Open* mit dem Sie den Toolchain Manager öffnen.



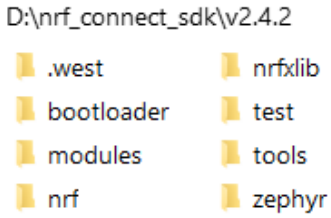
Im Toolchain-Manager setzen Sie zunächst unter Settings das Verzeichnis in dem die Toolchain und das SDK installiert werden soll. Wählen Sie ein Verzeichnis möglichst nah an der Wurzel und ohne Leerzeichen, um später keine Probleme wegen zu langer und ungültiger Dateinamen mit dem Compiler zu bekommen.



Anschließend installieren Sie das *nrf Connect SDK*. Wir benutzten die Version 2.4.2. Da das SDK aus vielen verschiedenen Repositorys zusammengestellt und recht groß ist, dauert der Installationsprozess etwas länger.



Nach der Installation des SDK finden wir im Installationsverzeichnis folgende Ordner:

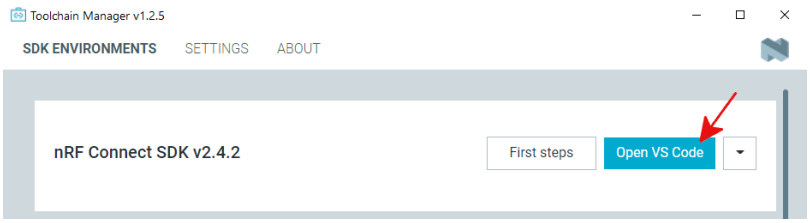


Das Verzeichnis *zephyr* beinhaltet das Echtzeitbetriebssystem *Zephyr*, welches als Basis für alle Anwendungen dient. Es gibt komfortabel Zugriff auf Hardwareressourcen und ermöglicht die Benutzung von Multithreading. Im Verzeichnis *nrf* und *nrfxlib* befinden sich für Nordic Semiconductor SoCs spezifische zusätzliche Bibliotheken.

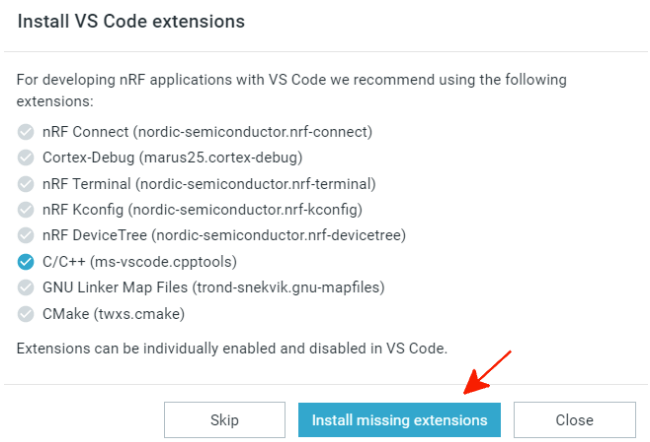
2.3 Installieren von Visual Studio Code

Von *Visual Studio Code* stehen für Windows verschiedene Versionen zum Download[3] zur Verfügung. Es gibt eine Version, die ausschließlich im User-Verzeichnis installiert wird und keine Administratorberechtigung erfordert. Auch wenn es fast keine Einschränkung bei der User-Version gibt, werden wir hier die System-Version installieren, die allen Benutzern zur Verfügung steht und im Programme-Verzeichnis installiert wird. Laden Sie sich die für ihre Rechnerarchitektur passende Version herunter und installieren diese mit den Standardparametern.

Um Visual Studio Code mit dem *nrf SDK* und der *Toolchain* nutzen zu können, müssen einige Erweiterungen installiert werden. Glücklicherweise erledigt das für uns recht komfortabel der Toolchain-Manager durch einen Klick auf den Button *Open VS Code*.

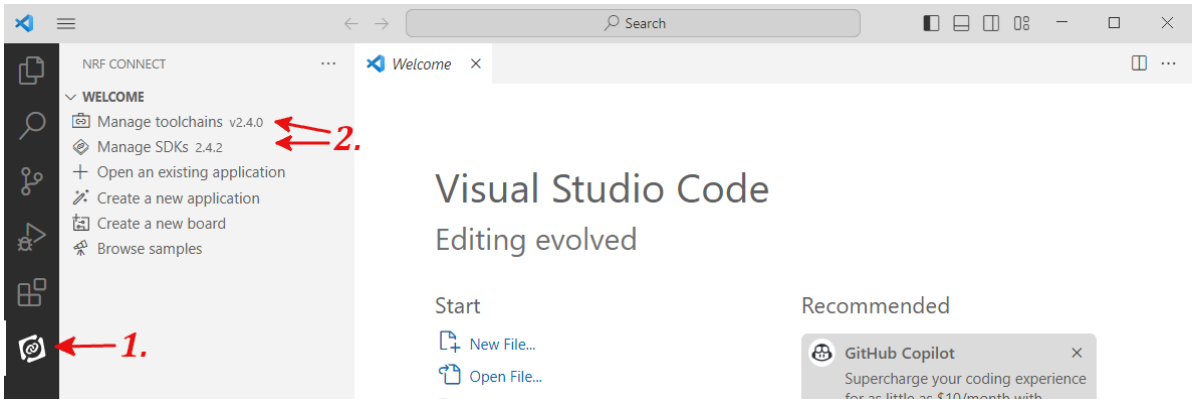


Im darauf erscheinenden Popup-Fenster klicken Sie auf *Install missing extensions*:



[3] https://code.visualstudio.com/download

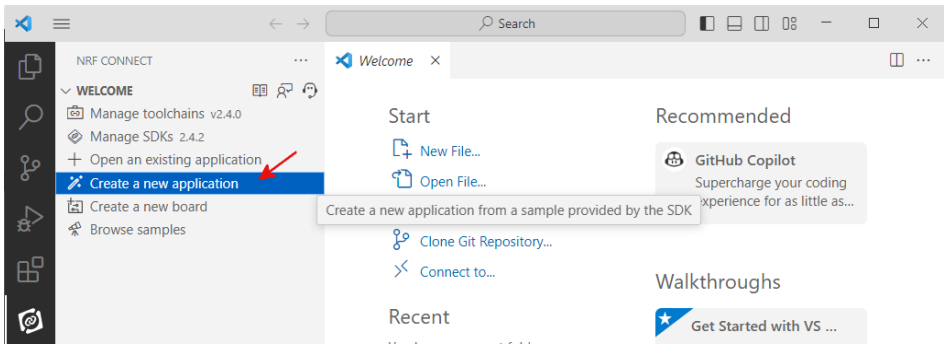
Im Anschluss starten Sie VS Code, klicken in der linken Leiste auf *nrf Connect* (1.) und überprüfen unter dem Menüpunkt *Welcome*, ob die aktuelle Toolchain und das aktuelle SDK ausgewählt sind (2.). Sind mehrere verschiedene Toolchain-Versionen installiert, kann hier auf eine andere Version gewechselt werden.



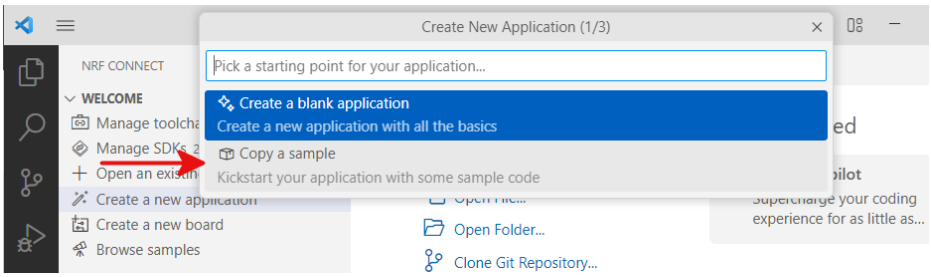
Die Installation von Visual Studio Code ist damit vollständig. Wir können jetzt VS Code starten und unsere erste Anwendung für unser Developer Kit nRF52840-DK programmieren.

3 Erste Beispielapplikation blinky

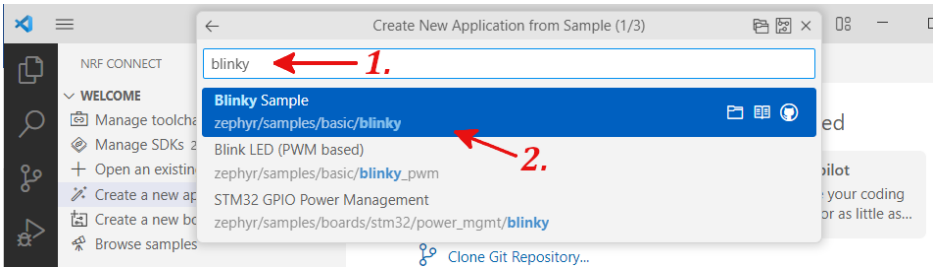
Als erste Anwendung werden wir die Beispielapplikation *blinky* in unseren Workspace installieren, das Beispiel kompilieren und die Firmware auf unser Developerboard nRF52840-DK übertragen. Zunächst klicken Sie wieder auf *nrf Connect* in der linken Leiste und klicken im WELCOME-Menü auf *Create a new application*:



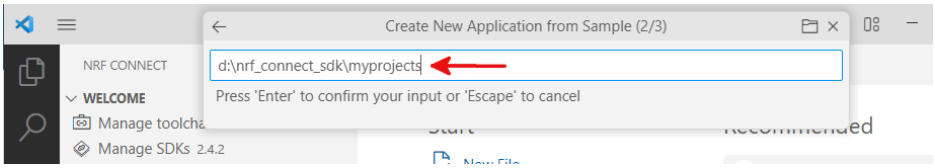
Im folgenden Popup-Fenster wählen Sie *Copy a Sample*:



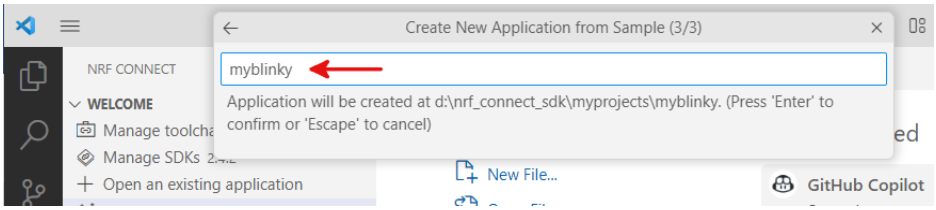
Anschließend geben Sie im Suchfeld (1.) *blinky* ein und wählen das Beispiel *Blinky Sample* (2.):



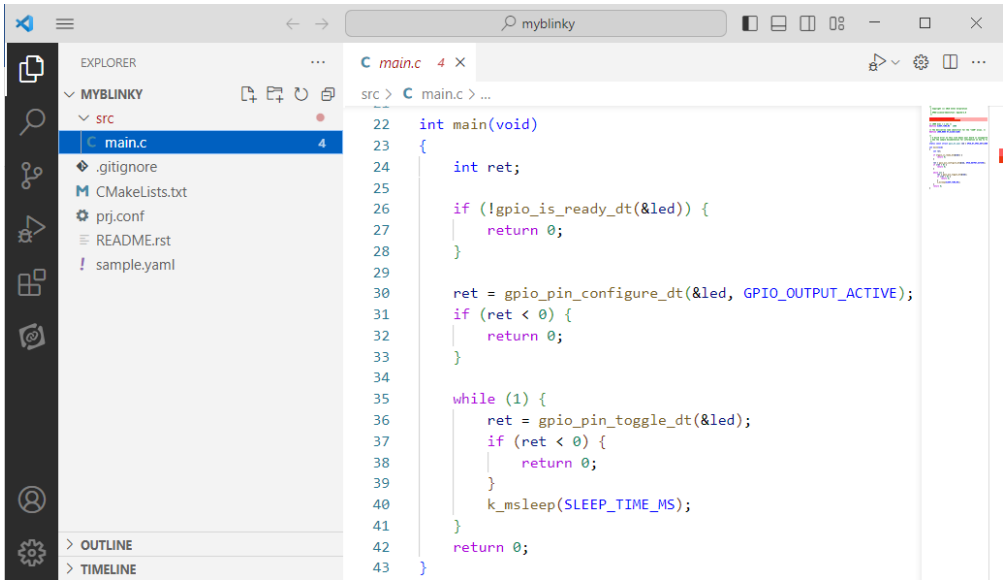
Im Schritt 2 ist der Speicherort des Projektes auszuwählen. Auch hier empfiehlt es sich lange Dateinamen zu vermeiden. Hier benutzen wir das Verzeichnis, in dem auch die Toolchain installiert ist und erstellen den Unterordner *myprojects*:



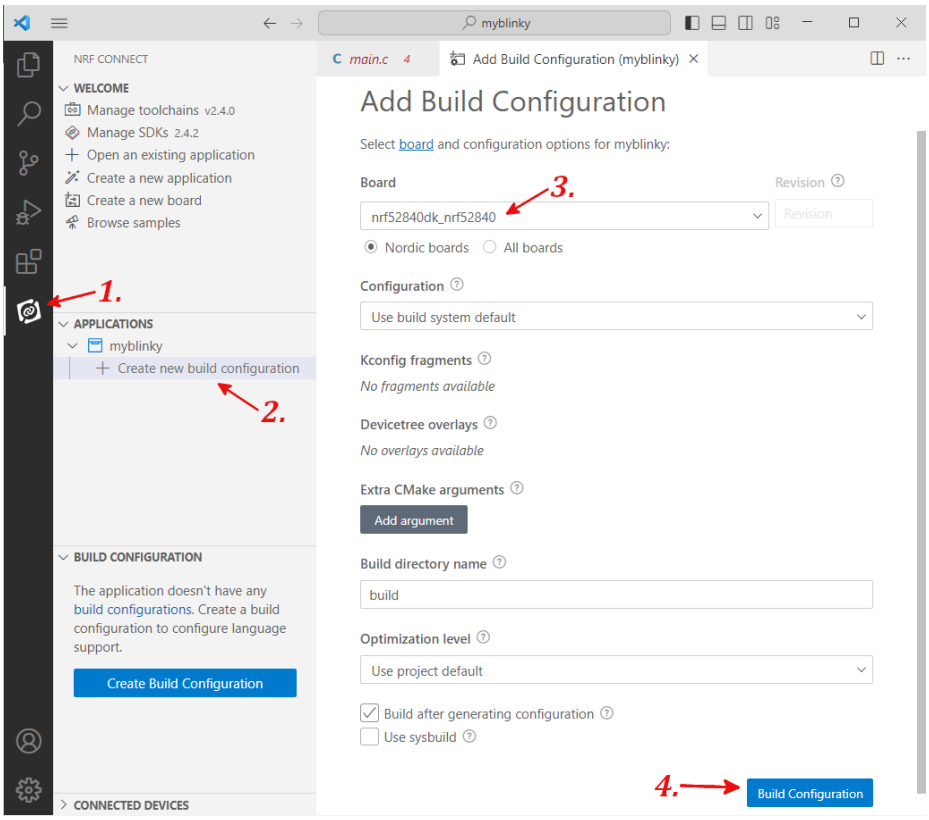
Abschließend müssen Sie noch einen Namen für das Projekt vergeben:



Nach der Erstellung des Projektes landen Sie automatisch im *Projekt-Explorer* von VS Code und sehen alle zugehörigen Projektdateien, die von der Erweiterung in unserem Projektordner angelegt wurden. Die Programmlogik des Beispiels finden Sie unter dem Ordner *src* in der Datei *main.c*.

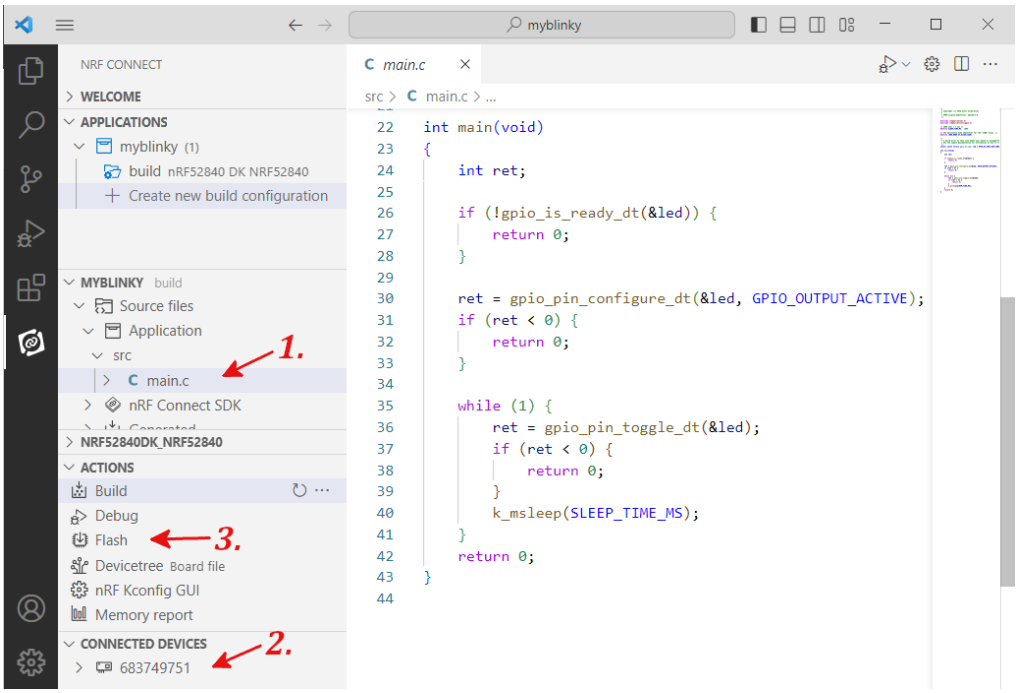


Auch im nrf Connect-Menü finden sich das Projekt. Hier müssen Sie zunächst für den Kompilierungsprozess *build*-Konfigurationsdateien erstellen, die zu dem Mikrocontroller des Boards passen. Hierzu klicken Sie unter *Applications/myblinky* auf *Create new build configuration* (2.).



Anschließend wählen Sie für das Developerboard nRF52840-DK den Boardtyp *nrf52840dk_nrf52840* (3.) aus und klicken auf *Build Configuration* (4.).

Nach dem eine Build-Konfiguration für das Board angelegt wurde, finden Sie einige neue Menüpunkte in der nRFConnect-Erweiterung. Unter *Source files→Application* (1.) finden Sie jetzt ebenfalls die Datei *main.c*:

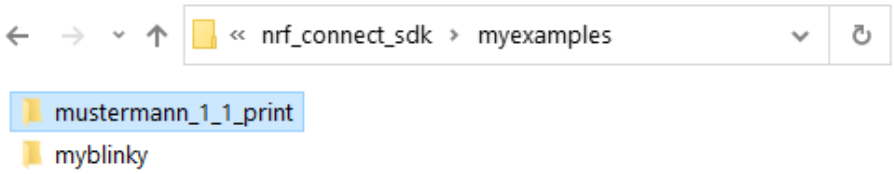


Wenn Sie das Developerboard an den USB-Port anschließen und dieses angeschaltet ist, erscheint die auf dem ATSAM3U22CA aufgeklebte Seriennummer unter CONNECTED DEVICES (2.). Mit einem Klick auf ACTIONS→Flash übertragen Sie die kompilierte Firmware auf das Developerboard und die mit LED1 beschriftete LED fängt an im Sekundentakt zu blinken.

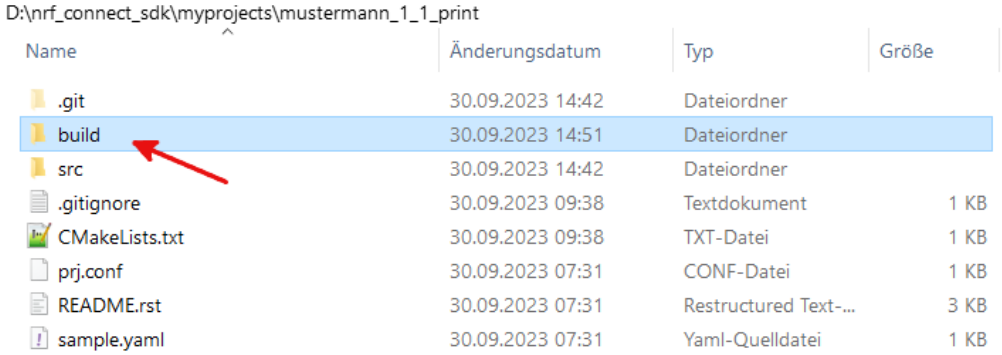
Da das Beispielprogramm *blinky* für den Anfang etwas schwer verständlich ist, werden wir dieses Projekt nur als Basisprojekt nehmen und uns langsam an die Programmierung der LEDs herantasten.

4 Textausgabe

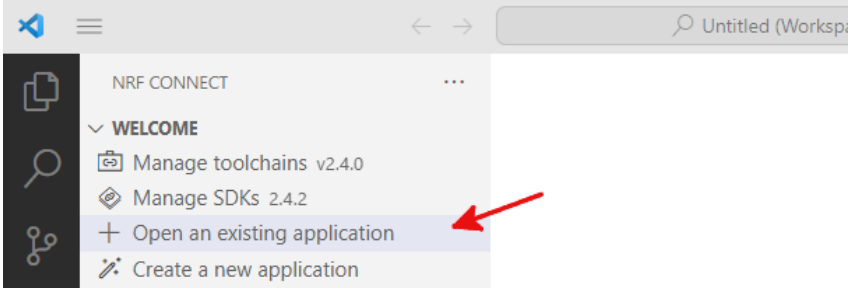
Unser erstes Programm wird eine einfache Textausgabe sein. Gehen Sie zunächst über den Dateibrowser zu dem Ordner *myblinky* im SDK-Verzeichnis, kopieren diesen und benennen Sie diesen um. Verwenden Sie bitte Ihren Nachnamen, die Übungsblattnummer und die Aufgabennummer, jeweils mit einem Unterstrich getrennt, als Namen. Zum Beispiel: *mustermann_1_1*. Fügen Sie auch gerne noch einen kurzen Text, wie z.B. *print* hinzu, der die Aufgabe beschreibt:



Bevor Sie das Projekt in VS Code einbinden, löschen Sie das zuvor erstellte *build*-Verzeichnis in diesem Projekt, da dieses Build auf den alten Dateinamen verweist:

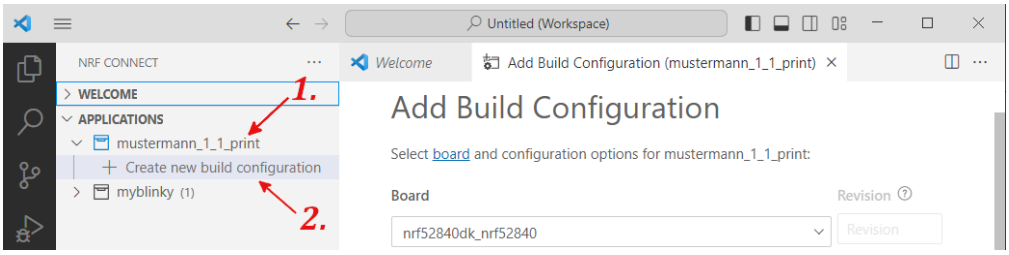


In VS Code fügen Sie den neuen Ordner als existierende Anwendung hinzu:



Da wir jetzt mit mehreren Projekten in VS Code arbeiten, müssen wir darauf achten, dass unter *APPLICATIONS* das richtige Projekt ausgewählt ist (1.). Zudem müssen wir eine neue build-Konfiguration erstellen

(2.). Wurde zuvor die alte build-Konfiguration nicht gelöscht, ist diese zuvor über VS Code zu löschen. Die neue build-Konfiguration erstellen Sie äquivalent zu dem blinky-Beispiel:



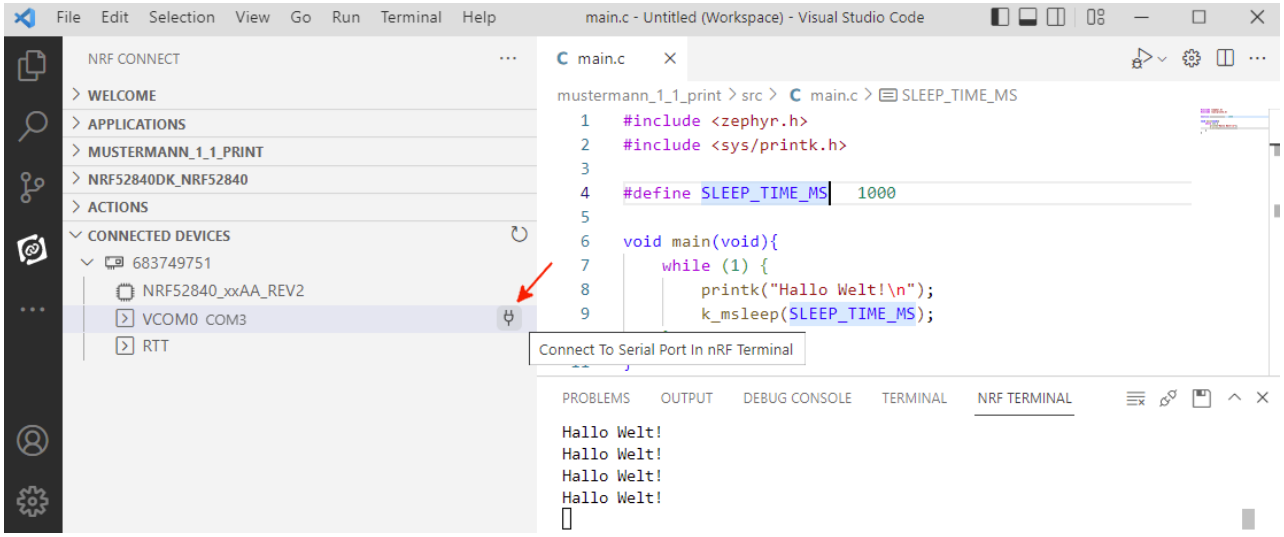
Anschließend können Sie die Quelldatei *main.c* öffnen. Das *nRF connect SDK* nutzt als Basis für eine Anwendung das Echtzeitbetriebssystem *Zephyr*. Dies ist sehr komfortabel, da es z.B. Multithreading und einen einheitlichen Zugriff auf Hardwareressourcen ermöglicht. Insbesondere die Textausgabe gestaltet sich so sehr einfach:

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

#define SLEEP_TIME_MS   1000

int main(void){
    while (1) {
        printk("Hallo Welt!\n");
        k_msleep(SLEEP_TIME_MS);
    }
}
```

Um zu verhindern, dass ein Mikrocontroller in einen undefinierten Zustand gerät, sollte ein Programm für ihn niemals terminieren. In der Regel wird dafür in der main-Funktion eine Endlosschleife definiert. Die Funktion `k_msleep` pausiert einen Thread für die angegebene Zeit und fungiert ähnlich wie eine Verzögerungsfunktion (delay). Die Funktion `printk` ermöglicht uns die Ausgabe von Text und unterstützt die gleiche Syntax wie die C-Funktion *printf*. Dies ist sehr praktisch und nicht selbstverständlich für Mikrocontroller. Normalerweise müssten wir die Schnittstelle, wie zum Beispiel UART, konfigurieren, Zeichenarrays entsprechend formatieren und an die Schnittstelle senden. Diese Aufgabe übernimmt das Betriebssystem Zephyr für uns. Die Ausgabe erfolgt über den voreingestellte UART0-Port des nRF52840-Chips mit einer Baudrate von 115200. Das Developerboard besitzt zudem eine integrierte UART-USB-Bridge, die UART in USB umwandelt. Wenn das Board an den USB-Port eines PCs angeschlossen ist, kann die Textausgabe mithilfe eines Terminalprogramms wie PuTTY sichtbar gemacht werden. Praktischerweise beinhaltet auch VS Code ein Terminalprogramm. Hierfür klicken Sie unter *CONNECTED DEVICES* auf Ihr Developerboard und wählen unter dem virtuellen COM-Port *Connect To Serial Port in nRF Terminal* aus:



Aufgabe 1: Ausgabe einer Zählvariable.

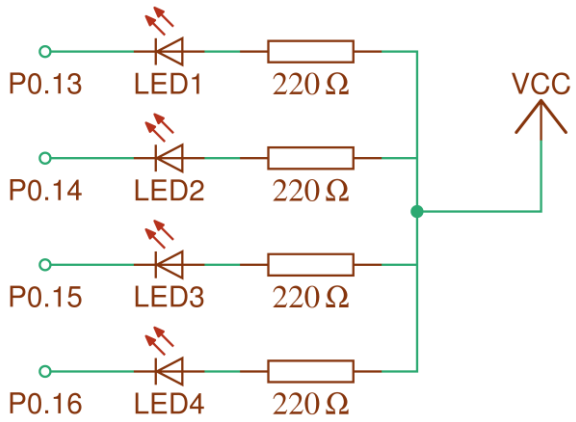
Erweitern Sie das obenstehende Programm so, dass eine Zählvariable im Sekundentakt um eins erhöht wird und diese dann über die Funktion `printk` ausgegeben wird. Erstellen Sie einen Screenshot ihrer Ausgabe und legen diesen Ihrer Abgabe im Projektordner bei.

Hinweis zur Abgabe

Halten Sie sich für die Projektordner an die Namenkonvention <Nachname_Blattnr_Aufgabennr> (z.B. musterman_1_1. Vor der Abgabe löschen Sie die build-Konfiguration, so dass fast ausschließlich der Sourcecode übrigbleibt und der gepackte Projektordner anstatt etlicher Megabyte nur einige KiloByte groß ist. Dies können Sie entweder manuell machen, indem Sie im Projekt den Ordner build löschen oder über VS Code in dem Sie im APPLICATIONS-Menü bei dem entsprechenden Build auf *More Actions→Remove Build Configuration* klicken. Die kompletten Aufgaben eines Aufgabenblattes packen Sie bitte in ein zip-Archive zusammen und laden dies in Moodle in dem zum Aufgabeblatt gehörenden Bereich hoch.

5 LEDs Programmieren

Das Developerboard nRF52840-DK hat vier eingebaute LEDs. Der nRF52840 besitzt insgesamt 48 frei konfigurierbare GPIO-Pins. Diese Pins können entweder als einfache Ein-und Ausgänge benutzt oder wahlweise mit speziellen Funktionen versehen werden. Die Adressierung erfolgt über die zwei Ports P0 und P1 und die Nummerierung geht von P0.00 bis P0.31 bzw. von P1.00 bis P1.15. Die LEDs am Developerboard nRF52840-DK sind wie folgt an den nRF52840 angeschlossen:



Die Leds werden angeschaltet, wenn ein LOW-Signal am entsprechenden Pin des Chips anliegt und ausgeschaltet werden, wenn der Pin auf HIGH geschaltet wird. Diese Beschaltung ist für Einsteiger etwas ungewöhnlich, aber durchaus häufiger anzutreffen.

Als erstes werden wir die Ansteuerung der LEDs über die nRF-GPIO-Bibliothek implementieren und nicht über das Betriebssystem. Vor der ersten Nutzung müssen wir einen entsprechenden GPIO-Pin als Ausgang definieren:

```
nrf_gpio_cfg_output(LED1_PIN);          //set pin as output
```

Anschließend können wir diesen Pin entsprechend auf High oder Low setzen:

```
nrf_gpio_pin_set(LED1_PIN);             //set pin to HIGH (LED off)
nrf_gpio_pin_clear(LED1_PIN);           //set pin to LOW (LED on)
```

Für eine saubere Programmierung definieren wir die Pin-Nummer über ein passendes Makro:

```
#define LED1_PIN           NRF_GPIO_PIN_MAP(0,13)
```

Das Programm um LED1 im Sekundentakt an- und auszuschalten sieht damit wie folgt aus:

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <hal/nrf_gpio.h>
#define SLEEP_TIME_MS   1000
#define LED1_PIN        NRF_GPIO_PIN_MAP(0,13)
int main(void){
    nrf_gpio_cfg_output(LED1_PIN);  //set pin as output
    while (1) {
        nrf_gpio_pin_set(LED1_PIN);     //set pin to HIGH (LED off)
        k_msleep(SLEEP_TIME_MS);
        nrf_gpio_pin_clear(LED1_PIN);   //set pin to LOW (LED on)
        k_msleep(SLEEP_TIME_MS);
    }
}
```

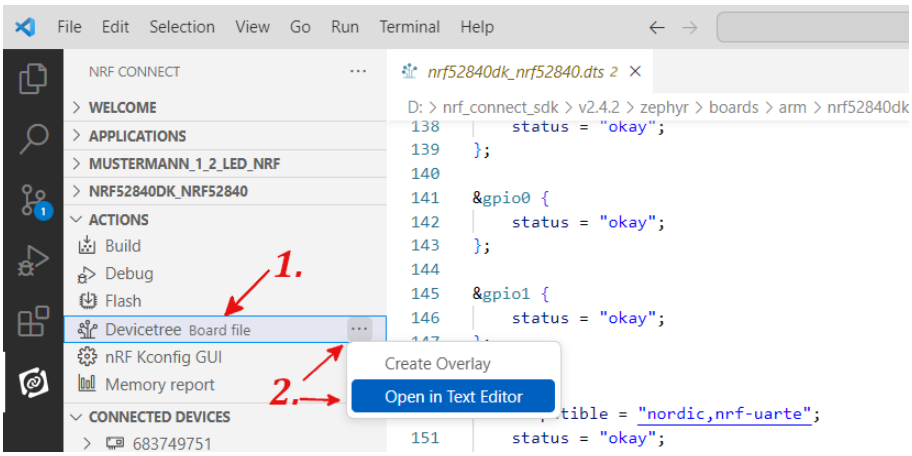
Aufgabe 2: Steuern Sie die LEDs über die GPIO-Pins und nutzen Sie dazu die Funktionen der nrf_gpio Bibliothek.

Passen Sie das vorherige blinky-Beispiel so an, dass Sie über die Bibliothek *nrf_gpio.h* die Ausgänge des nrf52840 steuern, an dem die Leds angeschlossen sind. Schalten Sie alle 4 Leds der Reihe nach an und wieder aus.

Denken Sie daran sich wieder eine Kopie des blinky-Projektes zu machen, nach unseren Konventionen umzubenennen, die build-Dateien zu löschen und neu erzeugen zu lassen. Durch klicken auf Flash können Sie das Programm direkt kompilieren und auf das Developerboard übertragen.

6 GPIOs und LEDs über das Betriebssystem ansteuern

In unserem ersten Beispiel haben wir GPIO-Pins direkt über Funktionen aus einer nRF-Bibliothek angesteuert. Wenn ein Board die LEDs an anderen Pins angeschlossen hat, müssten wir die entsprechenden Pins ändern. Für ein Board mit einem anderen Chip müssten wir sogar das Programm umschreiben. Da das *nRF Connect SDK* als darunterliegendes Betriebssystem *Zephyr* verwendet, funktioniert unser Programm überall dort, wo Zephyr als Betriebssystem eingesetzt wird, unabhängig des benutzten Mikrocontrollers oder wie etwas angeschlossen ist. Zumindest solange die Ressource zur Verfügung steht und keine speziellen mikrocontrollerabhängigen Bibliotheken genutzt werden. Die Hardwareverwaltung und -ansteuerung funktioniert, angelehnt an das Konzept in Linux, mit Hilfe eines sogenannten *Devicetrees*. In einem Devicetree werden alle Hardwarekomponenten, die zur Verfügung stehen, wie z.B. GPIO-Pins, LEDs, Schnittstellen und Buttons, definiert. Den Devicetree für unser Board finden Sie unter dem Menü ACTION→Devicetree:



Klicken Sie direkt auf das Menü (1.) startet eine grafische Oberfläche. Die zugehörige Konfigurationsdatei können Sie durch Klicken auf *More Actions*→*Open in Text Editor* öffnen. Dort finden wir auch eine Beschreibung der zwei GPIO-Ports, deren Status auf *okay* gesetzt ist:

```
&gpio0 {
    status = "okay";
};

&gpio1 {
    status = "okay";
};
```

Dies ist im Übrigen nur eine Überschreibung des Parameters *status*, der standardmäßig auf *disabled* gesetzt ist, um Ressourcen zu sparen. Die kompletten Parameter eines Knotens werden aus verschiedenen Dateien zusammengesetzt. Wir können uns alle Parameter eines Knotens anzeigen lassen, wenn wir in VS Code mit der Maus über den selbigen gehen:

```
140  &gpio0 {
141      /soc/gpio@50000000/
142
143      NRF5 GPIO node
144      gpio0: gpio@50000000 {
145          compatible = "nordic,nrf-gpio";
146          gpio-controller;
147          reg = <0x50000000 0x200 0x50000500 0x300>;
148          #gpio-cells = <2>;
149          status = "okay";
150          port = <0>;
151      };
152
153      pinctrl-1 = <&uart0_sleep>;
```

Die Knoten sind in einer hierarchischen Baumstruktur organisiert. Der GPIO0-Port hat hier den Namen `gpio@50000000` und besitzt den Elternknoten `soc` (`/soc/gpio@50000000`). Jeder Knoten des Devicetrees wird bei der Kompilierung intern in C-Code umgewandelt und erhält einen Namen, der mit *DT_N* für *Device Tree Node* beginnt. Der vollständige Knotenname des GPIO0-Port wird intern zu `DT_N_S_soc_S_gpio_50000000`.

Um die LED1 am GPIO Port 0 Pin 13 über Zephyr anzusteuern, müssen wir unser Programm ein bisschen anpassen. Neben der Änderung der gpio-Bibliothek und die damit einhergehenden Änderungen der Funktionsnamen, benötigen wir zuvor einen Zeiger auf eine Variable des struct-Datentyps *device*. Dieser Datentyp repräsentiert ein Gerät aus dem Devicetree:

```
#include <zephyr/kernel.h>
#include <zephyr/sys/printk.h>
#include <zephyr/drivers/gpio.h>
#define SLEEP_TIME_MS       1000
#define GPIO_0_LED_1   13
int main(void){
    const struct device *gpio0_dev = device_get_binding("gpio@50000000");
    gpio_pin_configure(gpio0_dev, GPIO_0_LED_1, GPIO_OUTPUT);
    while (1) {
        gpio_pin_set(gpio0_dev, GPIO_0_LED_1, 1);
        k_msleep(SLEEP_TIME_MS);
        gpio_pin_set(gpio0_dev, GPIO_0_LED_1, 0);
        k_msleep(SLEEP_TIME_MS);
    }
}
```

Jetzt haben wir die LEDs zwar über die GPIO-Funktionen von Zephyr gesteuert, allerdings immer noch direkt über die GPIO-Pins. Im Devicetree gibt es für die vier Leds des Developerboard einen Definitionsteil.

Zu beachten ist, dass die Nummerierung der LEDs in Zephyr bei 0 beginnt und nicht bei 1 wie die Beschriftung auf dem nrf52840 Developerboard:

```
    leds {
        compatible = "gpio-leds";
        led0: led_0 {
            gpios = <&gpio0 13 GPIO_ACTIVE_LOW>;
            label = "Green LED 0";
        };
        led1: led_1 {
            gpios = <&gpio0 14 GPIO_ACTIVE_LOW>;
            label = "Green LED 1";
        };
        led2: led_2 {
            gpios = <&gpio0 15 GPIO_ACTIVE_LOW>;
            label = "Green LED 2";
        };
        led3: led_3 {
            gpios = <&gpio0 16 GPIO_ACTIVE_LOW>;
            label = "Green LED 3";
        };
    };
```

Wir sehen, dass hier zu jeder LED eine Referenz auf den GPIO-Port, die Pin-Nummer und ein Flag hinterlegt ist, in welchem Zustand die Led anschaltet. Über Makros können wir jeden dieser Parameter einzeln abfragen:

```
#define LED0_NODE       DT_NODELABEL(led0)                  //DT_N_S_leds_S_led_0
#define LED0_GPIO_NODE  DT_PHANDLE_BY_IDX(LED0_NODE,gpios,0) //DT_N_S_soc_S_gpio_50000000
#define LED0_GPIO_NAME  DEVICE_DT_NAME(LED0_GPIO_NODE)      //gpio_50000000
#define LED0_PIN0       DT_GPIO_PIN(LED0_NODE, gpios)       //13
#define LED0_FLAG       DT_GPIO_FLAGS(LED0_NODE, gpios)     //1
```

Um den Knotennamen der Led zu ermitteln, haben wir hier auf das Label `led0:` zugriffen. Das Label steht mit einem Doppelpunkt separiert vor dem Knotennamen. Alternative hätten wir aber auch über den Pfad und das Makro `DT_PATH(leds, led_0)` den Knotennamen ermitteln können.

Somit können wir äquivalent wie oben für jede Led Makros setzen und diese über die Funktion `gpio_pin_set` schalten:

```
int main(void){
    const struct device *dev = device_get_binding(LED0_GPIO_NAME);
    gpio_pin_configure(dev, LED0_PIN0, GPIO_OUTPUT | LED0_FLAG);
    while (1) {
        gpio_pin_set(dev, LED0_PIN0, 1);
        k_msleep(SLEEP_TIME_MS);
        gpio_pin_set(dev, LED0_PIN0, 0);
        k_msleep(SLEEP_TIME_MS);
    }
}
```

Aufgabe 3: Benutzen Sie die Betriebssystemfunktionen zum Setzen der GPIO-Pins und um die Leds anzusteuern.

Passen Sie das vorherige Led-Beispiel so an, dass Sie anstatt der Nutzung der nRF-Bibliothek die GPIO-Pins über die Funktionen von Zephyr ansteuern und das Konzept des Devicetrees nutzen. Lassen Sie alle vier Leds der Reihe nach an- und wieder ausschalten.

Tipp zum Verständnis

Mit dem Betriebssystem Zephyr wird sehr viel über Makros gearbeitet, was ein bisschen gewöhnungsbedürftig ist. Mit den folgenden zwei Hilfsmakros können Sie sich zum besseren Verständnis den Inhalt von Makros über die Textausgabe anzeigen lassen:

```
#define STR(x)   #x
#define SHOW_DEFINE(x) printk("%s=%s\n", #x, STR(x))
```

Durch den Aufruf von `SHOW_DEFINE(LED0_NODE)`, `SHOW_DEFINE(LED0_GPIO_NAME)`, `SHOW_DEFINE(LED0_PIN0)`, `SHOW_DEFINE(LED0_FLAG)` wird dann der Inhalt der entsprechenden Makros über die Konsole ausgegeben.

Um vier Leds zu schalten benötigten wir im vorherigen Beispiel recht viele Makros. Das Ganze geht noch etwas eleganter. Als Makro benötigen wir in diesem Fall nur den Knotenpfad:

```
#define LED0_NODE   DT_NODELABEL(led0)              //DT_N_S_leds_S_led_0
```

In der main-Funktion definieren wir uns für jede LED eine Variable vom Typ `gpio_dt_spec`, die alle Informationen der Led aus dem Devicetree beinhaltet:

```
static const struct gpio_dt_spec led0_spec = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
```

Beim Datentyp `gpio_dt_spec` handelt es sich einfach nur um einen Container, der den Port, die Pin und das Flag speichert. Die GPIO-Pins konfigurieren wir anschließend über die Funktion `gpio_pin_configure_dt`:

```
gpio_pin_configure_dt(&led0_spec, GPIO_OUTPUT);
```

An- und ausschalten können wir die Led mit der Funktion `gpio_pin_set_dt`:

```
gpio_pin_set_dt(&led0_spec,1);          //Led on
gpio_pin_set_dt(&led0_spec,0);          //Led off
```

Aufgabe 4: Benutzen Sie die Datenstruktur gpio_dt_spec zum Schalten der Leds.

Passen Sie das vorherige Led-Beispiel so an, dass Sie für die vier Leds nur noch das Makro für den LED-Knoten benötigen und die Datenstruktur `gpio_dt_spec` benutzen um die Leds zu steuern.

7 Buttons

Die Nutzung von Buttons kann sehr praktisch sein. Durch Drücken eines Buttons kann beispielsweise ein Ereignis ausgelöst werden, wie das Schalten einer Lampe, das Starten einer Sensormessung oder das Versenden von Daten. Das Developerboard stellt neben den eingebauten Leds auch vier Buttons zur Verfügung. Das Betriebssystem Zephyr macht die Nutzung von Buttons recht einfach. Wir konfigurieren ähnlich zu den LEDs den GPIO-Pin, setzen ihn allerdings diesmal als Eingang. Zudem spezifizieren wir bei welchem Ereignis ein Interrupt ausgelöst wird:

```
gpio_pin_configure_dt(&button0_spec, GPIO_INPUT);
gpio_pin_interrupt_configure_dt(&button0_spec, GPIO_INT_EDGE_TO_ACTIVE);
```

Anschließend definieren wir eine Callback-Funktion, die aufgerufen wird, sobald wir den Button drücken. Wir schalten hier als Test eine LED.

```
void button_pressed_callback(const struct device *gpiob, struct gpio_callback *cb,
                             gpio_port_pins_t pins){
    gpio_pin_toggle_dt(&led0_spec);
}
```

Ein Zeiger auf die Callback-Funktion wird neben weiterer Parametern in einer Datenstruktur vom Typ `gpio_callback` gespeichert und an das Betriebssystem übergeben:

```
static struct gpio_callback button0_cb;
gpio_init_callback(&button0_cb, button_pressed_callback, BIT(button0_spec.pin) );
gpio_add_callback(button0_spec.port, &button0_cb);
```

Das vollständige Programm zum Schalten einer Led über einen Button sieht damit wie folgt aus:

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

#define SLEEP_TIME_MS   1000

#define LED0_NODE       DT_NODELABEL(led0)                  //DT_N_S_leds_S_led_0
#define BUTTON0_NODE    DT_NODELABEL(button0)               //DT_N_S_buttons_S_button_0

static const struct gpio_dt_spec led0_spec = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
static const struct gpio_dt_spec button0_spec = GPIO_DT_SPEC_GET(BUTTON0_NODE, gpios);
static struct gpio_callback button0_cb;

void button_pressed_callback(const struct device *gpiob, struct gpio_callback *cb,
                             gpio_port_pins_t pins){
    gpio_pin_toggle_dt(&led0_spec);
}

int main(void){

    gpio_pin_configure_dt(&led0_spec, GPIO_OUTPUT);
    gpio_pin_configure_dt(&button0_spec, GPIO_INPUT);

    gpio_pin_interrupt_configure_dt(&button0_spec, GPIO_INT_EDGE_TO_ACTIVE);
    gpio_init_callback(&button0_cb, button_pressed_callback, BIT(button0_spec.pin) );
    gpio_add_callback(button0_spec.port, &button0_cb);
```

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

Aufgabe 5: Schalten zweier Leds mit zwei Buttons der Leds.

Erweitern Sie das obige Beispiel so, dass Sie mit Button0 Led0 an- und ausschalten und mit Button1 Led1.