Rechnerorganisation
Hochschule für Technik und Wirtschaft - HTW Berlin
Sebastian Bauer
Wintersemester 2019/2020
4. Laboraufgabe vom 25. November 2019
Abnahme bis spätestens am 13. Januar 2020

Die Laborübungen beschäftigen sich mit der hardwarenahen Software-Entwicklung am Beispiel des STM32 Nucleo-32 Boards. Im vierten Labor ist unser Ziel, eine Art **digitale Wasserwaage** zu bauen, die über die LED-Leiste des Grundboards abgelesen werden kann und die Daten ebenfalls über die serielle Schnittstelle versendet. Studierende, die das Grundboard mit LED-Leiste nicht besitzen, nutzen dafür die einzelne LED des Nucleo-Boards. Im zweiten Teil gilt es, daraus ein **Eingabegerät** (engl. *human interface device*, HID) für Linux-Rechner zu bauen. Die finale Implementierung dieser Aufgabe ist optional. Die Vorbereitungen, also Aufgabe 6, gehören jedoch zum Pflichtteil.

Als zusätzliches Bauteil nutzen wir dafür den MPU-6050, das die Funktionalität eines 3-Achsen Beschleunigungssensor (Accelerometer), einen 3-Achsen Drehratensensor (Gyroskop) und die eines Thermometers bereitstellt.

Die Aufgabenbeschreibung setzt wieder einen funktionierenden Bare-Metall-Compiler voraus und benutzt ein Debian-basiertes Linux als Grundlage. Zu beachten ist, dass aufgrund der Vielzahl der Systemkonfigurationen die Schritte nicht auf jedem System in gleicher Weise funktionieren müssen. Sie stellen dann lediglich eine Idee dar, die selbständig auf das eigene System noch angepasst werden kann.

Anders als das Labor 3 muss dieses Labor nicht komplett in Assembler geschrieben werden. Den meisten Code schreiben wir in C und der Fokus dieser Beschreibung liegt auf das direkte Ansprechen der Hardware unter Umgehung des HALs. Als Buildsystem wird wieder *GNU Make* benutzt, das wir in der letzten Aufgabe kennengelernt haben.

Folgende weiterführende Literatur spielt für diesen Versuch eine Rolle:

- MPU-6050 Ressourcen
- Wikipediaeintrag zu I2C und `https://i2c.info/i2c-bus-specification`
- Programming Manual der Mikrocontrollerfamilie
- Reference Manual des Mikrocontrollerfamilie
- User Manual des Nucleo-Boards
- Datenblatt des Mikrocontrollers (insbesondere die dortige Tabelle 14)

Ebenso ist die Pinout-Belegung (siehe Tabelle 1 auf Seite 11) und für die Verbindungen zur LED-Leiste der Grundboard-Schaltplan von Interesse (siehe Abbildung 1 auf Seite 5).

Die Lösung der Aufgabe muss nicht unbedingt linear erfolgen. Es kann gerne zwischen Studium der Manuals, Notieren von Adressen und praktischer Programmierug sowie Testen hin- und hergesprungen werden. Auch muss die Lösung nicht strikt so aussehen, wie es die einzelnen Aufgaben vorgeben. Wichtig ist die Funktionalität des Programms.

---

1. **MPU-6050, I^2C und Pin-Multiplexing verstehen.** Der MPU-6050 soll bei uns über I^2C an den Mikrocontroller angebunden werden. Wie immer benötigen wir zu Testzwecken auch Zugriff auf eine serielle Schnittstelle.

(a) Verschaffe Dir einen Überblick über den I²C-Bus, ganz im Allgemeinen als auch wie unserer Mikrocontroller diesen Bus unterstützt (Kapitel 26 im Reference Manual)

(b) Verschaffe Dir einen Überblick über die serielle Kommunikation des Mikrocontrollers (Kapitel 27 im Reference Manual).

(c) Das Packaging des Mikrocontrollers auf dem Board ist sehr klein, so dass nicht alle Funktionen zur selben Zeit über die Pins abrufbar sind. Welche Funktionen ein Pin erfüllt, ist eine Laufzeiteinstellung des Mikrocontrollers, die wir setzen können. Beispielsweise kann Pin B6 neben GPIO, auch die Funktionen `USART1_TX`, `I2C1_SCL`, `TIM16_CH1N` oder `TSC_G5_IO3` erfüllen. Welche Einstellung welche Funktion aktiviert, lässt sich über den *Alternative-Funktion-Modus* eines Pins einstellen, über das das Datenblatt Auskunft gibt.

Es ist sinnvoll die Möglichkeiten und Einschränkungen schon vorab zu klären, so dass wir eine gute Konfiguration für unsere Aufgabe finden.

i. Informiere Dich über das Pin-Multiplexing des Mikrocontrollers. Das GPIO-Kapitel (Kapitel 8) aus dem Reference Manual ist hier nützlich. Dort sind alle Register beschrieben, die wir benötigen. Welche Register sind das?

ii. Welche Pins nutzt das Nucleo-Board, um Nutzern den Zugriff auf USART über USB bereitzustellen? Nutze für die Beantwortung das User Manual des Nucleo-Boards.

iii. Notiere den Wert des *Alternative-Funktion-Modus*, zu dem wir die Pins setzen müssen, damit diese USART-Signale führen. Nutze hierfür Tabelle 14 des Datenblatt. Das Kürzel RX steht für *receiver* und das Kürzel TX für *transmitter*.

iv. Um welchen USART-Block handelt es sich? Davon ist dann die Basisadresse abhängig.

v. Identifiziere via Abbildung 1 die Pins des Nucleo-Boards, die mit den LEDs des Grundboards verbunden sind. Wir müssen später diese Pins als Ausgabe konfigurieren.

vi. Vom MPU-6050 benötigen wir nur vier Pins: VCC, GND, SCL, SDA. VCC schließen wir an den 3,3 V-Pin des Boards und GND an den GND-Pin des Boards an. An welchen Pins des Grundboards bzw. des Nucleo-Boards werden die andern beiden Pins angeschlossen?

vii. Auch I²C-Unterstützung gibt es in mehreren Blöcken. Welchen benötigen wir? Davon hängt dann die Basisadresse ab.

viii. Notiere den Wert des *Alternative-Funktion-Modus*, zu dem wir die Pins setzen müssen, damit diese I²C-Signale führen. Tabelle 14 des Datenblatt ist auch hier nützlich.

Zur Beantwortung der Fragen kann auch **STMCubeMX** herangezogen werden.

2. **Adressen notieren.**

Notiere alle wichtigen Adressen und Werte der von uns wahrscheinlich interessanten Register. Bedenke, dass die Hardware-Komponenten des Mikrocontrollers in den meisten Fällen erst explizit über den RCC-Block angeschaltet werden müssen. Wir benötigen auch Zugriff auf die I/O Pins der Ports A und B sowohl als GPIO und müssen Pins davon sowohl als Ausgabepins als auch mit alternativen Funktionen (für USART2 und I²C) versehen. Basisadressen finden sich in Tabelle 1 im Reference Manual.

3. **Minimales Beispiel in C**.

Bevor wir mit dem eigentlichen Programm beginnen können, müssen wir sicherstellen, dass wir überhaupt in C entwickeln können und diese Software dann auch funktioniert. Hierzu schreiben wir zunächst ein minimales Programm, das wieder einmal die Textzeichenkette „Hello, world“ auf den seriellen Strom ausgibt, dieses Mal von uns selbst in allen Ebenen geschrieben.

(a) Erstelle ein neues Verzeichnis `level` in das alle Quelltextdateien abgelegt werden sollen.

(b) Lege darin ein `makefile` an mit den Targets `all`, `level.bin`, `level.elf` sowie `clean` und `upload`.

(c) Welche Targets sind `.PHONY`?

(d) Erstelle im `level`-Verzeichnis eine Datei mit Namen `head.S`. Hier soll sich der Vektor befinden, den Wert des Stacks sowie die Einsprungsadresse beinhaltet. Der Inhalt von `head.S` könnte so aussehen:

```
    .global _start

Vectors:
    .word 0x20001800    /* initial value of the sp */
    .word _start + 1
```

Das Label `_start` möchten wir in dieser Datei also nur referenzieren, aber nicht definieren.

(e) Schreibe eine Makeregel, um aus der Datei `head.S` eine Objektdatei `head.o` zu erzeugen. Der Aufruf ist identisch zum Aufruf im letzten Labor, nur der Dateiname unterscheidet sich.

Anmerkung: GNU-Make erlaubt es, innerhalb einer Regeldefinition Platzhalter zu verwenden, den sogenannten *automatischen Variablen*. Beispielsweise referenziert `$@` immer den Namen des aktuellen Ziels während `$<` für die erste aufgezählte Abhängigkeit steht. Damit könnte der komplette Eintrag auc so aussehen (vor den Anweisungen müssen sich Tabs befinden!):

```
head.o: head.S
    arm-none-eabi-gcc $< -mcpu=cortex-m0 -mthumb -c -o $@
```

Weitere Platzhalter werden in der Make-Dokumentation auf `https://www.gnu.org/software/make/manual/make.html#Automatic-Variables` aufgelistet. Viele sind sehr nützlich, um Redundanz zu vermeiden.

(f) Überprüfe die Richtigkeit der Makeregel via:

```
$ make head.o
```

Es sollte kein Fehler angezeigt werden und die Datei `head.o` sollte nun vorliegen (nutze `ls`, um Dich zu vergewissern).

(g) Erstelle eine neue Datei `usart.c`. Hier sollen alle USART-spezifischen Funktionen abgelegt werden. Die zugehörige Header-Datei `usart.h` soll auch angelegt werden: Sie enthält die Prototypen aller öffentlichen Funktionen aus `usart.c` sowie ihrer Dokumentationen im Doxygen-Format. Die Header-Datei `usart.h` könnte folgenden Inhalt haben (hier jedoch ohne Doxygen-Dokumentation):

```
#ifndef LEVEL_USART_H
#define LEVEL_USART_H

#include <stdint.h>

void usart_putc(uint8_t c);
void usart_puts(const char *str);
void usart_putx(uint32_t val);

#endif
```

(h) Füge eine neue Makeregel zum Makefile hinzu, die aus `usart.c` die Datei `usart.o` generiert. Die Datei `usart.c` kann zunächst leer sein.

(i) Überprüfe die Richtigkeit der Makeregel via:

```
$ make usart.o
```

(j) Definiere die Register als konstanter Zeiger auf die Registeradresse mit `volatile`-Qualifier innerhalb von `usart.c`. Zum Beispiel für das `CR1`-Register des USART2-Blockes:

```
static volatile uint32_t * const usart2_cr1 = (uint32_t*)(USART2_BASE + 0x00);
```

Die USART2-Basisadresse `USART2_BASE` muss zuvor natürlich definiert sein.

Die alternative Variante, eine Struktur für die Definition des gesamten Blocks zu nehmen, ist auch möglich.

(k) Was bedeutet noch einmal `volatile`?

(l) Implementiere die Funktion `void usart_putc(uint8_t c)` in der Datei `usart.c`. Sie soll das Zeichen, das `c` repräsentiert, auf den USART ausgeben. In der Vorlesung haben wir eine mögliche Implementierung bereits besprochen.

(m) Basierend auf `usart_putc()` schreibe die Funktion `usart_puts(const char *str)`, die eine komplette nullterminierte Zeichenkette ausgeben soll.

(n) Schreibe nun die letzte Funktion `usart_putx(uint32_t val)`, die einen Wert als ASCII-Zeichenkette über USART ausgibt. Das heißt, ein Aufruf von `usart_putx(0xDEADBEEF)` soll auf der Schnittstelle die ASCII-Zeichenkette `DEADBEEF` ausgeben.

(o) Erstelle nun den Quelltext mit der Einstiegsfunktion `_start()` (so wie wir das Label in `head.S` genannt haben). Nenne die Datei `level.c`. Im ersten Schritt kann `_start()` noch leer sein.

(p) Füge eine neue Makeregel für `level.o` hinzu. Da wir gleich `usart.h` darin via `#include` einbinden werden, ist `level.o` zusätzlich von `usart.h` abhängig.

(q) Komplettiere nun die Makeregel für `level.elf`. Wir wollen alle Objektdateien zu einer Datei mit Hilfe des Linkers zusammenbinden. Achte auf die korrekten Abhängigkeiten (welche brauchen wir?). Das Kommando sieht in der Shell wie folgt aus:

```
$ arm-none-eabi-gcc head.o usart.o level.o -mcpu=cortex-m0 -mthumb \
-nostartfiles -Wl,-Ttext=0x8000000 -o level.elf
```

Im Makefile können wieder automatische Variablen genutzt werden. Wichtig ist allerdings, das `head.o` tatsächlich am Anfang steht, da hier der Vektor definiert wird. Die Reihenfolge der anderen Dateien spielt keine Rolle.

(r) Vergiss das Committen Deiner Änderungen nicht.

(s) Teste den Build-Zyklus

- Die Eingabe von `make` erstellt alle Dateien
- Die Eingabe von `make clean` löscht alle durch `make` erzeugten Dateien wieder

Passe die Regeln solange an, bis alles nach Erwartung läuft.

(t) Nutze im Hauptprogramm innerhalb von `_start()` nun `usart_puts(const char *str)`, um „Hello, world“ auszugeben. Die Funktionsprototypen sollen aus der Datei `usart.h` mit `#include` eingebunden werden.

Damit USART funktioniert, müssen wir USART zunächst via RCC aktivieren. Zusätzlich müssen die Pins auf die USART-Funktionalität gestellt werden (Register `GPIOx_MODER` und `GPIOx_AFRL` bzw. `GPIOx_AFRH`). Auf USART-Seite muss die Baudrate eingestellt werden (`USART2_BRR`) und die Funktion eingeschaltet werden (`USART2_CR1`). Im Anhang des Reference Manuals sind hierfür Beispiele zu finden (siehe A.19).

Es ist sinnvoll, dies im Kontext der `_start()`-Funktion zu tun, wird aber idealerweise in eine eigene Funktion ausgelagert, die durch `_start()` aufgerufen wird.

(u) Teste das Programm (am einfachsten mit Hilfe des `upload`-Targets, das ggf. noch anzupassen ist)

(v) Teste im Anschluss ein paar Beispiele mit `usart_putx()`.

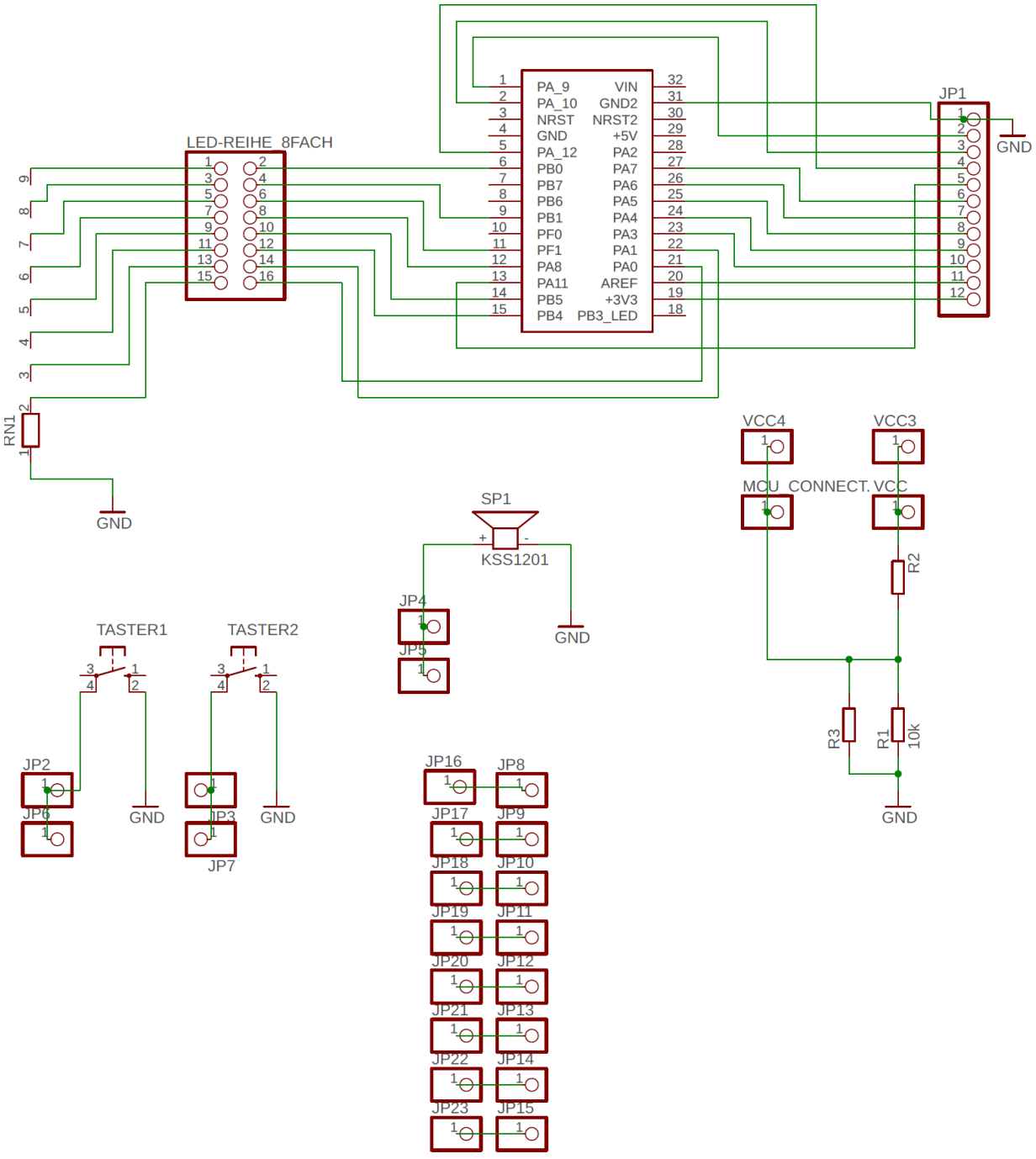

Abbildung 1: **Grundboard-Schaltplan**

4. **Wer bin ich**. Wir möchten den I²C-Bus in Gang setzen. Unser erstes Ziel wird sein, das sogenannte `WHOAMI`-Registerwert des MPUs mit Hilfe von I²C-Bus abzufragen. Das ist ein geeigneter Test, um zu schauen, ob die angeschlossene Hardware prinzipiell ansprechbar ist. Dazu müssen wir zunächst die Grundlagen schaffen. Hierfür werden wir drei Funktionen schreiben:

   - `int32_t i2c_write8(uint8_t slave_addr, uint8_t data);`
   - `int32_t i2c_read8(uint8_t slave_addr);`
   - `int32_t i2c_read_reg8(uint8_t slave_addr, uint8_t reg_addr);`

   (a) Lege zwei neue Dateien `i2c.c` und `i2c.h` an.

   (b) Füge die Übersetzung von `i2c.c` nach `i2c.o` in den Build-Prozess ein.

   (c) Modifiziere die Makeregel von `level.elf` so, dass `i2c.o` hinzugebunden wird.

   (d) Teste, ob der Build-Prozess noch immer funktioniert, sowohl `make` als auch `make clean`.

   (e) Committe den neuen Stand.

   (f) Schreibe die Prototypen in die Header-Datei. Die Header-Datei sollte auch Include-Guards besitzen.

   (g) Als Taktfrequenz für unsere I²C-Verbindung nehmen wir 400 kHz, den sogenannten *fast mode*. Setzen kann man diesen durch das Register `I2C1_TIMINGR`. Um den Wert dafür zu bestimmen, kann `STMCubeMX` genutzt werden, das ihn in der Eingabemaske zur I2C-Konfiguration anzeigt. Abbildung 2 auf Seite 7 zeigt einen Screenshot mit der korrekten Einstellung.

   (h) Im Abschnitt A.14 des Reference Manuals sind Beispiele zur Programmierung der I2C-Hardware zur Konfiguration und zum Empfang und Senden von Daten angegeben. Studiere diese Beispiele. Nutze aber auch die allgemeinen Ressourcen zu I²C und die Informationen aus Kapitel 26 im Reference Manual, um die Programmierweise zu verstehen.

   (i) Definiere nun `i2c_write8()` und `i2c_read8()` in `i2c.c`. Überlege anhand des Namens, was die Funktionen tun sollen.

   (j) Das Lesen eines Registers von einem I²C-Gerät geschieht oft so, dass man zunächst die Adresse des auszulesenden Registers zum Gerät übermittelt, um anschließend den Wert des Registers einzulesen. Definiere `i2c_read_reg8()` entsprechend dieses Vorgehens.

   (k) Nutze `i2c_read_reg8()`, um den Wert des `WHOAMI`-Registers im Hauptprogramm einzulesen und anschließend mit `usart_putx()` auszugeben. Die 7-bittige Slave-Adresse des MPUs lautet `0x68` und die Adresse des Registers ist `0x75`.

   Um die Lesbarkeit des Codes zu erleichtern, sollten die benutzten Werte (Register und Geräteadresse) über ein Symbol angesprochen werden. Wie in der vorangegangenen Teilaufgabe schon verwendet, definiert man symbolische Namen oft auf Präprozessorebene mit `#define`, es kann aber auch das C-Sprachkonstrukt `enum` benutzt werden. Zum Beispiel:

   ```
   #define IMU_I2C_SLAVE_ADDR 0x68
   #define MPUREG_WHOAMI 0x75
   ```

   oder via `enum`

   ```
   enum {IMU_I2C_SLAVE_ADDR = 0x68};
   enum {MPUREG_WHOAMI = 0x75};
   ```

   Du kannst somit `i2c_read_reg8(IMU_I2C_SLAVE_ADDR, MPUREG_WHOAMI)` anstelle von dem weniger aussagenden `i2c_read_reg8(0x68, 0x75)` schreiben.

   (l) Überprüfe den Wert auf Plausibilität (vgl. hierzu das Datenblatt des MPUs, das über die MPU-6050 Ressourcen beziehbar ist, die Webseite `https://www.i2cdevlib.com/devices/mpu6050#registers` ist auch sehr hilfreich). Es kann sein, dass die Implementierung beim ersten, zweiten, dritten oder x-ten Mal noch nicht richtig funktioniert, weil noch Bugs in den benutzen, von uns selbst geschriebenen Funktionen sind. Hilfreich ist in solchen Fällen oft ein

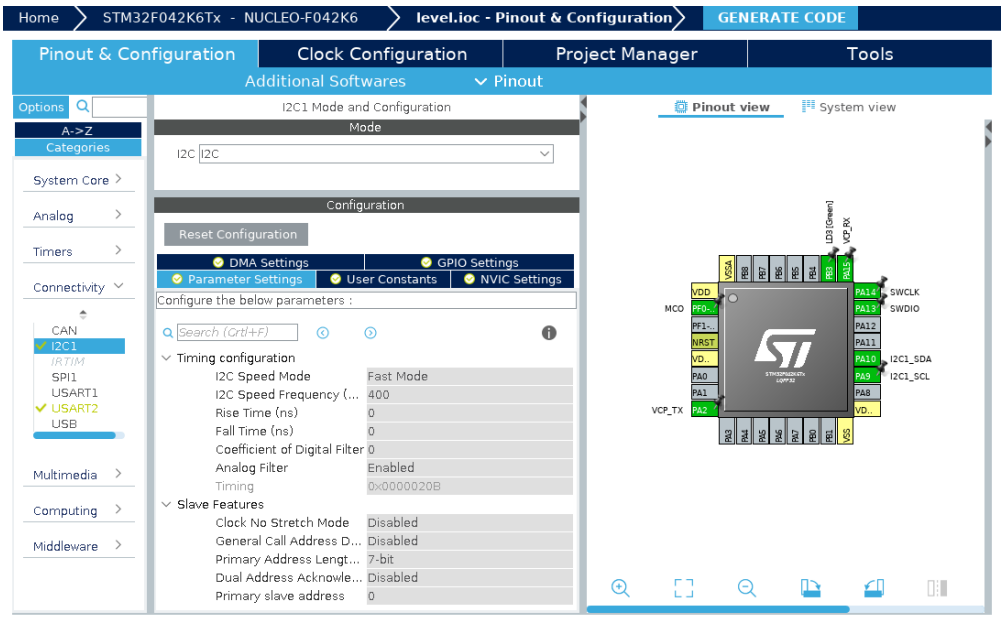

Abbildung 2: **STM32CubeMX-Einstellungen**. Uns interessiert der Wert, den für für das `I2C1_TIMINGR`-Register nehmen.

Oszilloskop, um die Probleme einzugrenzen. Allerdings führt auch Logging mit Hilfe der von uns verfassten USART-Funktionen, die Probleme zu verstehen, zu isolieren und schließlich zu beheben. Mögliche Fehlerquellen wären:

- Der I^2C Bus ist nicht im RCC eingeschaltet.
- Die Pins sind nicht auf I^2C-Signale eingestellt.
- Die angesprochene Geräteadresse stimmt nicht der Adresse des MPUs überein.
- Das Timing-Setting ist nicht korrekt.
- Andere Einstellungen sind nicht korrekt.

(m) Modifiziere den Code, die Plausibilitätsprüfung zur Laufzeit durchzuführen. Im Fehlerfall soll das Programm abbrechen und eine Fehlermeldung auf dem seriellen Strom ausgeben.

5. **Wasserwaage**. Entwickele nun die Firmware für die Wasserwaage. Die Libelle (Luftblase) soll durch eine eingeschaltete LED nachgebildet werden. Es ist wichtig, dass die I^2C-Kommunikation aus der vorherigen Ausgabe klappt, bevor mit dieser Aufgabe begonnen werden kann.

   (a) Mache Dich erneut mit dem Datenblatt des MPUs vertraut.

   (b) Auf Seite 8 des Datenblatts stehen ganz unten die Resetwerte der Register. Eines davon ist das `PWR_MGMT_1`-Register. Welchen Resetwert hat es? Welche Bedeutung hat dieser Wert? Und wie müsste er sein, damit das Gerät arbeitet?

   (c) Dem I2C-Modul fehlt noch eine Funktion zum Schreiben eines Registers. Definiere diese Funktion mit sinnvollen Argumenten und nenne sie `i2c_write_reg8()`. Ingesamt müssen zwei Bytes geschrieben werden: zuerst die Adresse des zu setzenden Registers, dann den zu setzenden Wert.

   (d) Setze das Register auf den Wert, der das Gerät arbeiten lässt mit Hilfe von `i2c_write_reg8()`. Dieses und alle weiteren Register sollten wegen der leichteren Lesbarkeit wieder über Symbole angesprochen werden.

   (e) Setzte dann die Empfindlichkeit des Beschleunigungssensors auf $\pm 16g$ mit Hilfe des Registers `ACCEL_CONFIG`.[1]

[1] $g = 9{,}81\,\mathrm{m/s^2}$

(f) Suche im Datenblatt nach möglichen Kandidaten für Register, über die man die aktuellen Messwerte des Beschleunigungssensor auslesen kann. Sie sind als Low- und High-Register ausgezeichnet.

(g) Wie lässt sich aus den 8-Bit-Low- und Highwerten 16-Bit-Werte ermitteln?

(h) Das Board soll zunächst ruhig auf dem Tisch liegen. Wie sind die zu erwartenden Messwerte?

(i) Welche Beschleunigung erfährt der Sensor in Richtung der X-Achse, wenn nach der Messung Register `0x3b` den Wert `0x01` und Register `0x3c` den Wert `0x04` hat?

(j) Lese die Beschleunigungswerte aller drei Achsen fortwährend ein und gebe sie zunächst per USART2-Block aus.

(k) Überprüfe die Werte auf Plausibilität.

(l) Schreibe nun Code, der die Lage des Boards direkt auf dem Board visualisiert. Studierende, die das *Grundboard* haben, gehen wie folgt vor:

- Denke Dir eine mathematische Funktion aus, die die eingelesenen Werte auf einen Wert abbildet, der den Zustand der LEDs sinnvoll repräsentiert.
- Implementiere diese Funktion in einer neuen Quelltextdatei `levelmap.c`. Die Funktion selbst soll nichts über die Hardware-Spezifika benötigen.
- Erstelle eine zugehörige Header-Datei, z. B., `levelmap.h`, die neben Include-Guards den Prototypen der Funktion enthält.
- Implementiere einen Unit-Test in einer separaten Datei `levelmap_test.c`. Diese Datei enthält eine `main`-Funktion und muss nicht auf dem Board ausgeführt werden, sondern auf dem Entwicklungsrechner. Es kann z. B. `assert()` aus `assert.h` benutzt werden, um auf die zu erwartenden Ergebnisse zu testen.[2] Es sollen mindestens sechs verschiedenen Eingaben getestet werden. Der neue Teil des Makefiles kann so aussehen:

```
levelmap_test: levelmap.c levelmap.h levelmap_test.c
    gcc levelmap.c levelmap_test.c -ggdb -O3 -Wall -o $@

.PHONY: test
test: levelmap_test
    ./levelmap_test
```

- Wenn der Test durchläuft, integriere die neue Funktion im Hauptprogramm der Wasserwaage, um die gewünschte Funktionalität zu erhalten.

Studierende *ohne Grundboard* nehmen die einzelne LED des Nucleo-Boards, und versehen diese z. B. mit folgender Funktionalität:

- Die LED leuchtet permanent, wenn das Board bzw. der Sensor waagerecht auf dem Tisch ruht.
- Andernfalls blinkt die LED und zwar um so schneller, je größer die Abweichung zu dem waagerechten Zustand ist.
- Ein Unit-Test wie bei der Variante mit dem Grundboard ist ebenfalls zu implementieren.

(m) Glückwunsch. Die erste nicht triviale Applikation ist somit fertig :)

6. **HID – Vorbereitungen**. Unser Hardware-Setup soll jetzt als weitere Eingabemöglichkeit für einen beliebigen Linux-basierten Rechner dienen, um z. B. den Mauszeiger zu bewegen. Hierzu müssen wir ein Treiberprogramm für den Linux-Rechner schreiben, das die Daten vom Board empfängt und auswertet. Wir können hierbei im Userspace bleiben, da Linux mit dem *uinput*-Framework es erlaubt, virtuelle Eingabegeräte zu erstellen. Eine Beschreibung dazu gibt es hier: `https://www.kernel.org/doc/html/latest/input/uinput.html`. Die Idee ist es also, ein Programm zu schreiben, dass die Daten unserer Hardware empfängt und in entsprechende Format des *uinput*-Frameworks umwandelt.

[2] Für größere Projekte bietet es sich an, Unit-Test-Frameworks zu nutzen.

(a) Mache Dich mit dem Linux *uinput*-Framework und dessen Programmierung vertraut.

(b) Das *uinput*-Framework wird durch das gleichnamige Linux-Kernel-Modul bereitgestellt. Nicht alle Distributionen müssen dieses mitliefern. Stelle sicher, dass das Modul bei Dir prinzipiell verfügbar ist, indem Du es in den laufenden Kernel einbindest:

```
$ modprobe uinput
```

Hierzu bedarf es Rootrechte!

(c) Der geladenen Treiber fügt dem System ein neues Gerät hinzu `/dev/uinput`. Verifiziere dessen Existenz und analysiere die Ausgabe:

```
$ ls /dev/uinput -l
```

Erkennbar ist, dass nur Root dieses Gerät lesen und beschreiben kann, zumindest wenn dies noch nicht verändert wurde. Wir möchten aus Sicherheitsgründen unser Programm aber nicht mit diesen Privilegien starten, deshalb stellen wir dieses Gerät anderen Nutzern zur Verfügung mit folgender Strategie: Es gibt eine neue Gruppe `uinput`, in der wie alle Nutzer eintragen, die eben Zugriff auf das Gerät haben sollen.[3] Erstelle zuerst die Gruppe:

```
$ groupadd -f uinput
```

Füge den gewünschten Benutzer zur Gruppe hinzu, z. B. so:

```
$ gpasswd -a <user> uinput
```

(d) Damit das Gerät `/dev/uinput` Nutzern der Gruppe `uinput` zur Verfügung steht, muss die Gruppe von `/dev/uinput` auf `uinput` gestellt sowie die Gruppenbits entsprechend angepasst werden:

```
$ chgrp uinput /dev/uinput
$ chmod g+rw /dev/uinput
$ ls -l /dev/uinput
crw-rw---- 1 root uinput 10, 223 Oct  4 08:42 /dev/uinput
```

(e) Übersetze und teste das Beispiel von der oben erwähnten Dokumentationsseite, das eine Mausbewegung nachbildet. Verwende hierzu den normalen Nutzer, also nicht Root.

(f) Die Änderungen aus Teilaufgabe d überleben einen Neustart nicht. Soll dies permanent geschehen, ist eine Möglichkeit, eine sogenannte udev-Regel zu schreiben. Erstelle hierzu die Datei `/etc/udev/rules.d/99-uinput.rules` mit:

```
$ cat <<EOF >/etc/udev/rules.d/99-uinput.rules
KERNEL=="uinput", GROUP="uinput", MODE:="0660", OPTIONS+="static_node=uinput"
EOF
```

Teste im Anschluss, ob nach einem Neustart die gewünschten Rechte tatsächlich für `/dev/uinput` eingestellt wurden:

```
$ ls -l /dev/uinput
```

(g) Die Übertragung der Daten soll über die serielle Schnittstelle erfolgen. Prinzipiell lässt sich der Mikrocontroller auch direkt als USB-Gerät benutzen, aber hierzu müssten wir einen weiteren USB-Steckplatz an den Controller anschließen, was wir aus Zeitgründen nicht tun. Wir nehmen deshalb den vorhandenen USB-Anschluss und nutzen hierbei wie gewohnt den USART2-Block.

Definiere ein Format, in der die Daten zum Rechner übertragen werden. Das Board soll die Werte von allen Achsen des Beschleunigungssensor und auch die des Gyroskopes, die wir bisher noch ignoriert haben, übermitteln. Der Einfachheit halber soll es ein ASCII-Format sein.

(h) Offensichtlich müssen wir in unserer Treiber-Applikation den seriellen Datenstrom einlesen. Wie stellt man programmatisch die Baudrate ein? Ein fast minimales Beispiel hierfür ist auf `https://gist.github.com/lategoodbye/f2d76134aa6c404cd92c` zu finden. Relevant sind

[3] Für den Produktiveinsatz ist das immer noch kein optimales Setup. Wieso?

demnach die zwei `ioctl()`s: `TCGETS2` und `TCSETS2`. Geh durch den Code durch und versuche ihn zu verstehen.

7. **HID – Implementierung**. Implementiere die Funktion des neuen Eingabegeräts sowohl auf Mikrocontroller-Seite (Firmware) als auch auf der Seite des Arbeitsrechners (Treiber). Die Funktion der Wasserwaage soll beibehalten werden. Diese Teilausgabe ist aus Zeitgründen optional.
   (a) Kopiere das existierende `level`-Projekt in ein neues Verzeichnis `hid/firmware`. Das bedeutet, dass sich die Quellen für den Controller von jetzt an in einem Unterverzeichnis `firmware` im neuen `hid`-Verzeichnis befinden.
   (b) Passe darin das Makefile an und nenne die Dateien sinnvoll um, weil das Unterprojekt jetzt `hidfirmware` heißen soll.
   (c) Implementiere das Auslesen des Gyroskopes (zuvor haben wir nur die des Beschleunigungssensors ausgelesen) und schreibe alle Daten in den seriellen Ausgabestrom wie in der letzten Aufgabe definiert.
   (d) Teste und committe alle Änderungen.
   (e) Erzeuge ein neues Verzeichnis `hid/driver`. Darin soll der Quelltext des Treibers abgelegt werden.
   (f) Erzeuge darin eine Quelltextdatei `hiddriver.c`, das lediglich ein `Hello, world` ausgeben soll.
   (g) Erstelle ein Makefile, das aus dem Quelltext die unter Linux ausführbare Datei erzeugt. Beachte: der Compiler ist der systemeigene und nicht der Cross-Compiler für unser Board. Wir nehmen also `gcc` oder `clang`. Zu einem Makefile gehören auch die `all`- und `clean`-Targets.
   (h) Überprüfe den Build-Prozess und teste, ob der „Treiber“ aktuell tatsächlich `Hello, world` ausgibt.
   (i) Implementiere die geforderte Funktionalität. Benötigt wird eine Initialisierungsphase (für die serielle Kommunikation und `uinput`-Erstellung) und eine Hauptschleife, die fortwährend Daten vom seriellen Strom einliest und diese nach einer sinnvollen Umformung an das `uinput`-Framework weiterleitet. Besonders die Umformung ist interessant und erfordert einen Unit-Test. Die Lösung der Aufgabe soll kleinschrittig erfolgen. Teste und committe kleine Änderungen so früh wie möglich.

       Es ist in Ordnung, wenn der Programmabbruch benutzerseitig zunächst via `Strg+C` zu erfolgen hat, weil hierfür nichts besonders zu tun ist.
   (j) Wie könnte man den Treiber automatisch starten, sobald unser Grundboard eingesteckt wird?
   (k) Optionaler Teil des optionalen Teils: Implementiere folgende Funktion: Die beiden auf dem Grundboard vorhandenen Taster sollen als linke und rechte Maustaste auf dem Arbeitsrechner fungieren.

JP1 Pin	Funktion	MCU Pin	IO-Typ
1	GND		
2	I2C1_SCL, UART1_TX	PA9	FTf
3	I2C1_SDA, UART1_RX	PA10	FTf
4	UART1_RTS, CAN1_TD	PA12	FTf
5	UART1_CTS, CAN1_RD	PA11	FTf
6	SPI1_MOSI, ADC1/7	PA7	TTa
7	SPI1_MISO, ADC1/6	PA6	TTa
8	SPI1_SCLK, ADC1/5	PA5	TTa
9	SPI1_SSEL, ADC1/4	PA4	TTa
10	ADC1/3	PA3	TTa
11	AREF		
12	3,3 V		

Tabelle 1: **Pinout von JP1 auf dem Grundboard.** Pin 1 befindet sich auf der Seite, die den geringeren Abstand zum Boardende hat.