Documentation of Task 01

Further files of this task: See `Task_1/` .

Task 1.1 Raspberry Pi

Hardware preparations:

- Install Raspherry Pi Imager using `sudo apt install rpi-imager` (on Debian-based distros)
- Take sd card with reader and insert in pc
- Open "Imager" and select sd card and Raspherry Pi OS Lite 32 bit
- Press `Ctrl` + `Shift` + `X` :
- Activate Hostname
- Activate SSH - Password
- Username: pi-esho
- Password: RasPi
- Press Write
- insert the SD card in the Raspberry Pi and connect power and ethernet
- we can now run `ssh pi-esho@raspberrypi` to connect to the Raspberry Pi

Software preparations

- update the software with:

```
sudo apt update
sudo apt upgrade -y
```

Task 1.2 OpenOCD

- We follow the instructions on the OpenOCD GitHub Repository
- To build OpenOCD from source we first have to install some dependencies

```
sudo apt update
sudo apt upgrade
sudo apt install git build-essential libtool libtool-bin pkg-config autoconf automake
texinfo
```

- after that we can clone the repository, configure openocd to enable the `bcm2835gpio` interface, build and install openocd

```
# clone the openocd repository
git clone https://github.com/openocd-org/openocd.git
# navigate into the folder
cd openocd
# checkout the latest release v0.12.0
git checkout 9ea7f3d647c8ecf6b0f1424002dfc3f4504a162
# needed when building from the git repository
./bootstrap
# generates the Makefiles required to build. We enable the bcm2835gpio interface
./configure --enable-bcm2835gpio
# build openocd
make
# install it
sudo make install
```

- Compilation succeded without any errors noticed.
- The following pins are involved in the SWD communication:

	SWDIO	SWCLK	(Source)
Pins on the Cortex-M0	PA13	PA14	STM32F0 Reference Manual, page 922
Pin numbers of the STM32F0 on the extension board	34	37	ESA Board Documentation, PCB Schematic (U11)
Wires on the extension board	JTAG_SWD	JTAG_SWCLK	ESA Board Documentation, PCB Schematic (U11)
Pin numbers on the J1 header on the extension board / on the 40-pin header on the Raspberry Pi	31	29	ESA Board Documentation, PCB Schematic (J1)
GPIO pins on the Cortex-A53	GPIO 6	GPIO 5	Raspberry Pi hardware - Raspberry Pi Documentation

- Thus, GPIO 5 will be used for SWDLK and GPIO 6 for SWDIO. Our configuration file `stm32f0raspberry.cfg` looks like this:

```
adapter driver bcm2835gpio

adapter gpio swdio 6
adapter gpio swclk 5

transport select swd

source [find target/stm32f0x.cfg]

bindto 0.0.0.0

init
targets
```

- With openocd installed, we can run it with our config file. Output after running `openocd -f stm32f0raspberry.cfg` :

```
Open On-Chip Debugger 0.12.0-01004-g9ea7f3d64 (2024-05-16-10:11)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : BCM2835 GPIO JTAG/SWD bitbang driver
Info : clock speed 1006 kHz
Info : SWD DPIDR 0x0bb11477
Info : [stm32f0x.cpu] Cortex-M0 r0p0 processor detected
Info : [stm32f0x.cpu] target has 4 breakpoints, 2 watchpoints
Info : starting gdb server for stm32f0x.cpu on 3333
Info : Listening on port 3333 for gdb connections
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
 0* stm32f0x.cpu       cortex_m   little stm32f0x.cpu       running

Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Task 1.3 Compile a Linux Kernel

Cross-Compile on a x86 machine

- To cross-compile the Linux kernel on a x86 machine, we mainly follow the steps descriped in the Raspberry Pi Linux Kernel Documentation.
- First, we ensure all necessary dependencies and the cross-compiler are installed on our system:

```
sudo apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev
libncursesw5-dev
sudo apt install crossbuild-essential-armhf # this installs the 32-bit cross-compiler
```

- In the meantime, we clone the Raspberry Pi Linux kernel repository with

```
git clone --depth=1 https://github.com/raspberrypi/linux
cd linux
```

- To compile the kernel:

```
# set shell variable with the target kernel version
KERNEL=kernel7
# create config
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
# show config and close it afterwards
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
# compile the kernel
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j36 zImage modules dtbs
```

- The config file is attached as `raspberry-pi-linux-kernel-config_cross-compiling`
- the final kernel image is then located at `./arch/arm/boot/Image`
- To install the kernel, we insert the Pi's SD card back into our computer and mount both partitions to a subfolder of our local repository for convenient access:

```
mkdir mnt
mkdir mnt/{fat32,ext4}
sudo lsblk # => sdcard is sdf on my system
sudo umount /dev/sdf{1,2} # sdcard was already mounted automatically elsewhere by the OS
sudo mount /dev/sdf1 mnt/fat32/
sudo mount /dev/sdf2 mnt/ext4/
```

- We can then remove all the kernel images currently stored on the SD card and install our self-compiled kernel:

```
sudo rm mnt/fat32/kernel*.img

sudo env PATH=$PATH make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
INSTALL_MOD_PATH=mnt/ext4 modules_install # install kernel modules

# copy kernel and device tree blobs onto the sdcard
sudo cp arch/arm/boot/zImage mnt/fat32/$KERNEL.img
sudo cp arch/arm/boot/dts/broadcom/*.dtb mnt/fat32/
sudo cp arch/arm/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm/boot/dts/overlays/README mnt/fat32/overlays/

sudo umount /dev/sdf{1,2}
```

- After all that, we insert the SD card back into our Raspberry Pi and it powers on normally. `uname -a` returns

```
Linux esa-pi 6.6.30-v7+ #1 SMP Thu May 16 16:26:36 CEST 2024 armv7l GNU/Linux
```

  so we are now indeed using our self-compiled kernel.

Cross-Compilation on a fedora Linux

- Not everyone in our group uses a Debian-based Linux. On fedora, the build depencencies have to be installed via `dnf`.
- On fedora, we use the `gcc-arm-linux-gnu` cross-compiler. The cross-compiler package can be installed with

```
sudo dnf install gcc-arm-linux-gnu
```

and in every `make` command, we have to use `CROSS_COMPILE=arm-linux-gnu-` instead of `CROSS_COMPILE=arm-linux-gnueabihf-`.

Compile Time

- On a system with an Intel Core i7-13700k (where the CPU is power-limited to 155 watts), compilation took around 2.5 mins. `time make ARCH=arm ...` returned:

```
real    2m28,310s
user    52m23,139s
sys     4m46,420s
```

- In contrast, compiling with 12 threads on a Laptop took about 28 min and 12 sec.

Compile on the Raspberry pi

- To compile and install the kernel directly on the Raspberry Pi, we again mainly follow the steps described in the Raspberry Pi Linux Kernel Documentation. We execute the following steps:

```
# install git and build dependencies
sudo apt install git bc bison flex libssl-dev make libncurses5-dev libncursesw5-dev

# clone the repository
git clone --depth=1 https://github.com/raspberrypi/linux

# set shell variable with the target kernel version
KERNEL=kernel7
# create config
make bcm2709_defconfig
# show config and close it afterwards
make menuconfig

# compile the kernel
make -j4 zImage modules dtbs

# install the kernel
sudo make modules_install
sudo cp arch/arm/boot/dts/broadcom/*.dtb /boot/firmware/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/firmware/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/firmware/overlays/
sudo rm /boot/firmware/*.img
sudo cp arch/arm/boot/zImage /boot/firmware/$KERNEL.img
```

- The config file is attached as `raspberry-pi-linux-kernel-config_local`.

- After a reboot, `uname -a` returns

```
Linux esa-pi 6.6.30-v7+ #1 SMP Thu May 16 22:35:38 CEST 2024 armv7l GNU/Linux
```

so compilation succeeded.

Compile Time

- On our Raspberry Pi 3s, compilation took almost 3 hours. During compilation, the Pi operated at its thermal limit of 85°C so the Cortex-A53 most likely got throttled heavily. `time make -j4 ...` returned:

```
real    167m29.066s
user    616m37.755s
sys     33m2.246s
```

Documentation of Task 02

Further files of this task: See `Task_2/`.

Prerequisites:

- Install `gcc-arm-none-eabi` on the Raspberry Pi so that we are able to cross-compile to the Cortex-M0 from the Pi itself:

```
sudo apt install gcc-arm-none-eabi
```

- Note: Ensure to clone submodules too, s.t. the CMSIS and HAL drivers are available for compilation:

```
git submodule update --init --recursive
```

Compilation and Flashing:

In the Makefile, `better-blinky.c` (Task 2) is set up as the default source code to compile and execute on the extension board. To compile or flash `blinky.c` (Task 1) instead, run

```
make CSRC=blinky.c
```

and

```
make flash CSRC=blinky.c
```

respectively.

Makefile

The following additions were made:

- Line 6: the path to the CMSIS library. In our repository, STM32CubeF0 is located within /utils.
- Line 44: the full command to download the compiled ELF file to the Cortex-M0. Therefore, the `stm32f0raspberry.cfg` file is needed, which we created in Task 1. Information on how the program command works can be found in the OpenOCD Documentation at page 126.

CMSIS

The CMSIS library for our M0 processor is located at `STM32CubeF0/Drivers/CMSIS/Device/ST/STM32F0xx/Include/stm32f030xc.h`.

Note: There is also a device-independant header file in the same folder `stm32f0xx.h` which by itself includes the actual device-specific header file based on a predefined macro.

Some Remarks About `better-blinky`

- We think of "LED toggle *period*" as the time a whole blinking cycle takes, i.e. the time between the LED being turned on and the time it is being turned on next.
  E.g., if the period is 1000 ms, then the LED gets turned on for 500 ms, turned off for 500 ms and then back on again.
- We decided to limit the period setting to 50 to 10000 ms.
- The joystick pins are configured s.t. the movements seem natural when J2 is facing upwards. This is different to the STICK_UP, STICK_LEFT, ... signals on the extension board.

Important registers

- `MODER` mode register (p. 134)
- `PUPDR` pull-up/pull-down register (p. 135)
- `SYSCFG->EXTICR` external interrupt configuration register 1 (p. 144)
- `IMR` Interrupt mask register (p. 177)
- `RTSR` Rising trigger selection register (p. 177)
- `FTSR` Falling trigger selection register (p. 178)
- `ODR` output data register (p. 136)
- `CR1` : 'Control Register 1' – this register is used to enable and disable the peripheral.
- `CNT` : 'Counter Register' – this register hold's the timer's current counter value. It counts up from 0 once the timer is started.
- `PSC` : 'Prescaler Register' – this register will hold the timer's prescaler. A prescaler value of `N` will tick the timer's counter register up by one every `N+1` clock cycles.
- `ARR` : 'Autoreload Register' – the autoreload value is the 'period' of the timer. An autoreload value of `N` will cause the timer to trigger an update event every time the `CNT` register counts up to `N`.
- `EGR` : 'Event Generation Register' – Setting the `UG` bit in this registers resets all of the timer's counters and tells it to use the currently-set prescaler/autoreload values.
- `DIER` : 'DMA/Interrupt Enable Register' – Setting the `UIE` bit in this register sets the timer to trigger a hardware interrupt when an update event occurs. Usually, that happens when the 'Counter' register matches the 'Autoreload' value.

- `SR`: 'Status Register' – The `UIF` flag is set in this register when a timer's hardware interrupt triggers, and must be cleared before another one can occur.

source: Reference manual and "Bare Metal" STM32 Programming (Part 5): Timer Peripherals and the System Clock – Vivonomicon's Blog (23.5.24)

Documentation of Task 03

Further files of this task: See `Task_3/`.

Task 3.1

SWD uses two pins: The SWCLK pin represents the clock while the SWD pin represents the data pin. Both pins are mapped to GPIO pins.

SWD can be split into three phases:

Connection setup:
This phase consists of a lineReset where the SWD is written with 1s more than 50 times. After that the JTAG-to-SWD sequence is transmitted. After this another lineReset follows and the IDCODE register must be read.

Send phase:
Sending data via SWD is done in three steps. First the header bits are transmitted containing addresses and parities. Next an acknowledgement is read. Last a data word sized 32 bits is transmitted with a parity bit.

Read phase:
Read data via SWD is also done in three steps. First the header bits are transmitted containing addresses and parities. Next an acknowledgement is read. Last a data word sized 32 bits is read with a parity bit.

With SWD, the least-significant bit is always transmitted first.

In our example, we only need to read the IDCODE register, so we do not have any send phase.

To execute our SWD implementation and read the IDCODE register, enter these on the Raspberry Pi:

```
cd <repository>/utils/pigpio
mkdir build
cd build
cmake ..
make

cd <repository>/Task_3/1_Serial_Wire_Debugging
mkdir build
cd build
cmake ..
make

sudo ./SWD
```

This results in:

```
IDCODE = 0x0bb11477
```

which matches the `SWD DPIDR 0x0bb11477` line output by OpenOCD.

The whole communication log looks like this:

```
----------------------SWD setup----------------------
Line Reset
JTAG_to_SWD
Line Reset
Request: 10100101  AP/DP:0 R/W:1 Address:00 Parity:1
Acknowledge: 100
Receive: 11101110001010001000110111010000  Parity:1
 → Read: 0x0bb11477
--------------------SWD setup done--------------------
Request: 10100101  AP/DP:0 R/W:1 Address:00 Parity:1
Acknowledge: 100
Receive: 11101110001010001000110111010000  Parity:1
 → Read: 0x0bb11477
```

Additional Sources:

- Programming Internal Flash Over the Serial Wire Debug Interface

Task 3.2

Note: As a quick introduction to debugging with OpenOCD, this short video might be helpful.

Software preparations

On our host PC, we need to install `arm-none-eabi-gcc` and `arm-none-eabi-gdb`.

On MacOS, these can easily installed with `brew install gcc-arm-embedded`.

On Debian-based Linux distros, `arm-none-eabi-gcc` can be installed with `apt install gcc-arm-none-eabi`. Instead of `arm-none-eabi-gdb`, you can use `gdb-multiarch` or download an old version, as described here:

- Download gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2 from https://developer.arm.com/downloads/-/gnu-rm
- Extract via sudo tar xjf gcc-arm-none-eabi-10.3-2021.10-x86_64-linux.tar.bz2
- Execute ./arm-none-eabi-gdb (found in bin)

Start the GDB Server on the Raspberry Pi

Start OpenOCD on the Raspberry Pi by running:

```
openocd -f Task_1/stm32f0raspberry.cfg

Result:
Open On-Chip Debugger 0.12.0-01004-g9ea7f3d64 (2024-05-08-19:50)
Licensed under GNU GPL v2
For bug reports, read
      http://openocd.org/doc/doxygen/bugs.html
Info : BCM2835 GPIO JTAG/SWD bitbang driver
Info : clock speed 1006 kHz
Info : SWD DPIDR 0x0bb11477
Info : [stm32f0x.cpu] Cortex-M0 r0p0 processor detected
Info : [stm32f0x.cpu] target has 4 breakpoints, 2 watchpoints
Info : starting gdb server for stm32f0x.cpu on 3333
Info : Listening on port 3333 for gdb connections
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
0* stm32f0x.cpu       cortex_m   little stm32f0x.cpu       running

Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

So OpenOCD has now started a GDB Server on port 3333 of the Raspberry Pi.

Debug Session

On our host PC, we `cd` into `Task_2/Environment_baremetal`. In the following, we want to debug `blinky.c`:

- To do that, we first compile our program

```
➜ tobii@MacBook-Pro-von-Tobias:~/.../Task_2/Environment_baremetal$ make CSRC=blinky.c
arm-none-eabi-gcc -I../../utils/STM32CubeF0/Drivers/CMSIS/Include -
I../../utils/STM32CubeF0/Drivers/CMSIS/Device/ST/STM32F0xx/Include -mlittle-endian -
mcpu=cortex-m0 -mthumb -mfloat-abi=soft -g3 -c blinky.c -o build/blinky.o
arm-none-eabi-gcc -I../../utils/STM32CubeF0/Drivers/CMSIS/Include -
I../../utils/STM32CubeF0/Drivers/CMSIS/Device/ST/STM32F0xx/Include -mlittle-endian -
mcpu=cortex-m0 -mthumb -mfloat-abi=soft -g3 -c startup_stm32f030xc.s -o
build/startup_stm32f030xc.o
arm-none-eabi-gcc -mlittle-endian -mcpu=cortex-m0 -mthumb -mfloat-abi=soft -g3 -
TSTM32F030xC_FLASH.ld -nostartfiles -nostdlib -o main.elf  build/blinky.o
build/startup_stm32f030xc.o
```

- We then start `arm-none-eabi-gdb` and connect to the target (i.e. our M0 processor) using `target extended-remote tcp:hostIP:port`

```
➜ tobii@MacBook-Pro-von-Tobias:~/.../Task_2/Environment_baremetal$ arm-none-eabi-gdb
main.elf
GNU gdb (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.90.20231008-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=aarch64-apple-darwin20.6.0 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main.elf...
(gdb) target extended-remote tcp:192.168.2.2:3333
Remote debugging using tcp:192.168.2.2:3333
0x08000ab4 in ?? ()
```

- Load the compiled program onto our M0. This is just an alternative to using the flash command of OpenOCD.

```
(gdb) load
Loading section .isr_vector, size 0xbc lma 0x8000000
Loading section .text, size 0x29c lma 0x80000bc
Start address 0x080002dc, load size 856
Transfer rate: 2 KB/sec, 428 bytes/write.
```

- Reset the target and read the M0's registers

```
(gdb) monitor reset halt
[stm32f0x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x080002dc msp: 0x20008000
(gdb) info reg
r0             0xffffffff          -1
r1             0xffffffff          -1
r2             0xffffffff          -1
r3             0xffffffff          -1
r4             0xffffffff          -1
r5             0xffffffff          -1
r6             0xffffffff          -1
r7             0xffffffff          -1
r8             0xffffffff          -1
r9             0xffffffff          -1
r10            0xffffffff          -1
r11            0xffffffff          -1
r12            0xffffffff          -1
sp             0x20008000          0x20008000
lr             0xffffffff          -1
pc             0x80002dc           0x80002dc <Reset_Handler>
xPSR           0xc1000000          -1056964608
msp            0x20008000          0x20008000
psp            0xfffffffc          0xfffffffc
primask        0x0                 0
basepri        0x0                 0
faultmask      0x0                 0
control        0x0                 0
```

- Create a breakpoint at the beginning of main and let the M0 run up to this breakpoint. Read out the registers again.

```
(gdb) break main
Breakpoint 1 at 0x80000c0: file blinky.c, line 21.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb) c
Continuing.

Breakpoint 1, main () at blinky.c:21
21          setup_sys_clk();
(gdb) info reg
r0             0x20000000          536870912
r1             0x0                 0
r2             0x20000000          536870912
r3             0x20000000          536870912
r4             0xffffffff          -1
r5             0xffffffff          -1
r6             0xffffffff          -1
r7             0x20007ff8          536903672
r8             0xffffffff          -1
r9             0xffffffff          -1
r10            0xffffffff          -1
r11            0xffffffff          -1
r12            0xffffffff          -1
sp             0x20007ff8          0x20007ff8
lr             0x8000323           134218531
pc             0x80000c0           0x80000c0 <main+4>
xPSR           0x61000000          1627389952
msp            0x20007ff8          0x20007ff8
psp            0xfffffffc          0xfffffffc
primask        0x0                 0
basepri        0x0                 0
faultmask      0x0                 0
control        0x0                 0
```

- Set a breakpoint at the beginning of `toggle_led()`. We can then run our program until this breakpoint is being hit by typing `continue` (or the `c` shortcut). This way, every time we enter `c`, our LED gets toggled.

```
(gdb) break toggle_led
Breakpoint 2 at 0x80002b0: file blinky.c, line 106.
(gdb) c
Continuing.

Breakpoint 2, toggle_led () at blinky.c:106
106         GPIOA->ODR ^= (1 << LED_PIN);
(gdb) c
Continuing.

Breakpoint 2, toggle_led () at blinky.c:106
106         GPIOA->ODR ^= (1 << LED_PIN);
(gdb) c
Continuing.

Breakpoint 2, toggle_led () at blinky.c:106
106         GPIOA->ODR ^= (1 << LED_PIN);

[...]

(gdb)
```

- We want to examine the toggling of the LED more closely:
  - Currently, our LED is turned off.
  - Using the examine command (`x`), let's read `1` word (`w`) at address `0x48000014` and display the contents in binary format (`t`). `0x48000014` is the address of the GPIOA_ODR register, as we know from the M0 reference manual.
  - This yields in `00000000000000000000000000000000` , which makes sense because our LED is off.
  - After entering `next` , the next statement of `toggle_led()` is executed and the LED turns on.
  - If we repeat our examine command, we now get `00000000000000000000000010000000` so the 7-th bit is now `1` , which makes sense because our LED is connected to the 7-th pin on port A. More information on the examine command here.

```
(gdb) frame
#0  toggle_led () at blinky.c:106
106         GPIOA->ODR ^= (1 << LED_PIN);
(gdb) x/1tw 0x48000014
0x48000014:     00000000000000000000000000000000
(gdb) next
107     }
(gdb) x/1tw 0x48000014
0x48000014:     00000000000000000000000010000000
```

Messing with gdb has resulted in some additional log messages at OpenOCD on the Raspberry Pi:

```
Info : accepting 'gdb' connection on tcp/3333
Info : device id = 0x10006442
Info : flash size = 256 KiB
undefined debug reason 8 - target needs reset
[stm32f0x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x08000ab4 msp: 0x20008000
[stm32f0x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x080002dc msp: 0x20008000
[stm32f0x.cpu] halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x080002dc msp: 0x20008000
```

Task 3.3

Software preparations

Install Eclipse on the host PC:

- Download the Eclipse Installer
- Extract the installer

```
tar -xf eclipse-inst-jre-linux64.tar.gz
```

- Run the installer

```
./eclipse-inst
```

- Within the Eclipse Installer, select "Eclipse IDE for Embedded C/C++ Developers"

Open our project within Eclipse

We now open our ESHO1 project from git within Eclipse as a general project, which seems to be the best way to get it working there. To do that, follow these steps:

- Open Eclipse and close the Welcome dialogue
- Click "File" → "Import"
- Select "Projects from Git"
- Select "Clone URI" and enter `gitlab:ESHO1_24/Group07` as the URI
- At the Branch Selection, just click "Next"
- For the destination, create a new folder within the current Eclipse workspace and make sure to check "Clone submodules"
- Select "Import as general project" and hit "Finish"

We can now explore our whole ESHO1 project. Open `Task_2/Environment_baremetal/blinky.c`, which is the file we will be debugging.

Compilation

As our project structure of Task 2 does not really seam to fit to what Eclipse expects, we figure the easiest way to compile our program is just to do it manually:

- Open a Terminal by pressing `Ctrl` + `Shift` + `T` and enter

```
make CSRC=blinky.c
```

Note: Eclipse's Project Explorer does not refresh automatically. To Refresh, select the current project and hit `F5`.

We can then copy the compiled program onto our Raspberry Pi:

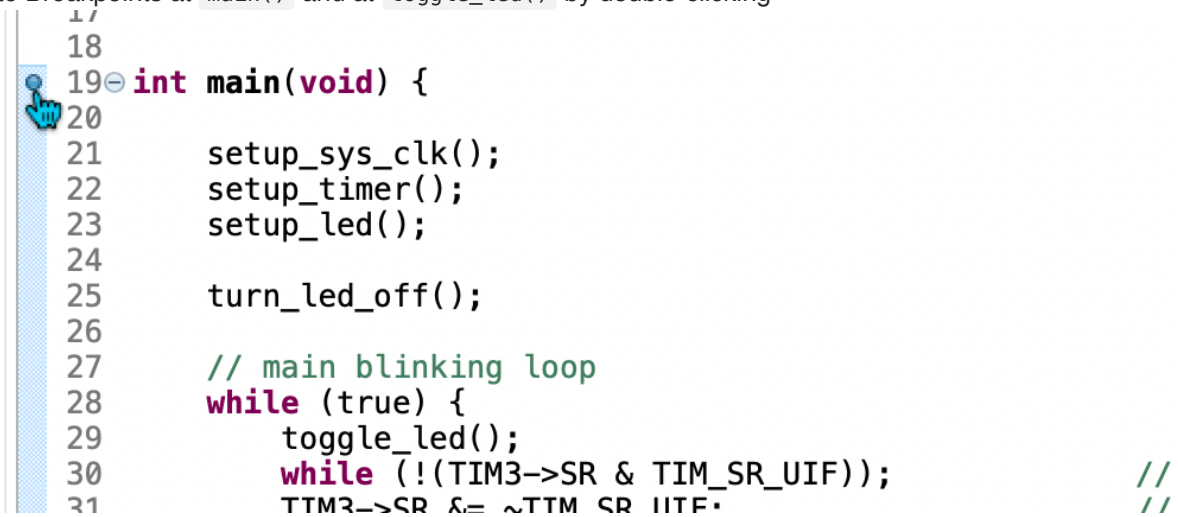
```
scp main.elf <Raspberry Pi's IP Address>:/home/<Raspberry Pi's Username>/
```

Debugging Session

- Start OpenOCD on the Raspberry Pi from the home directory:

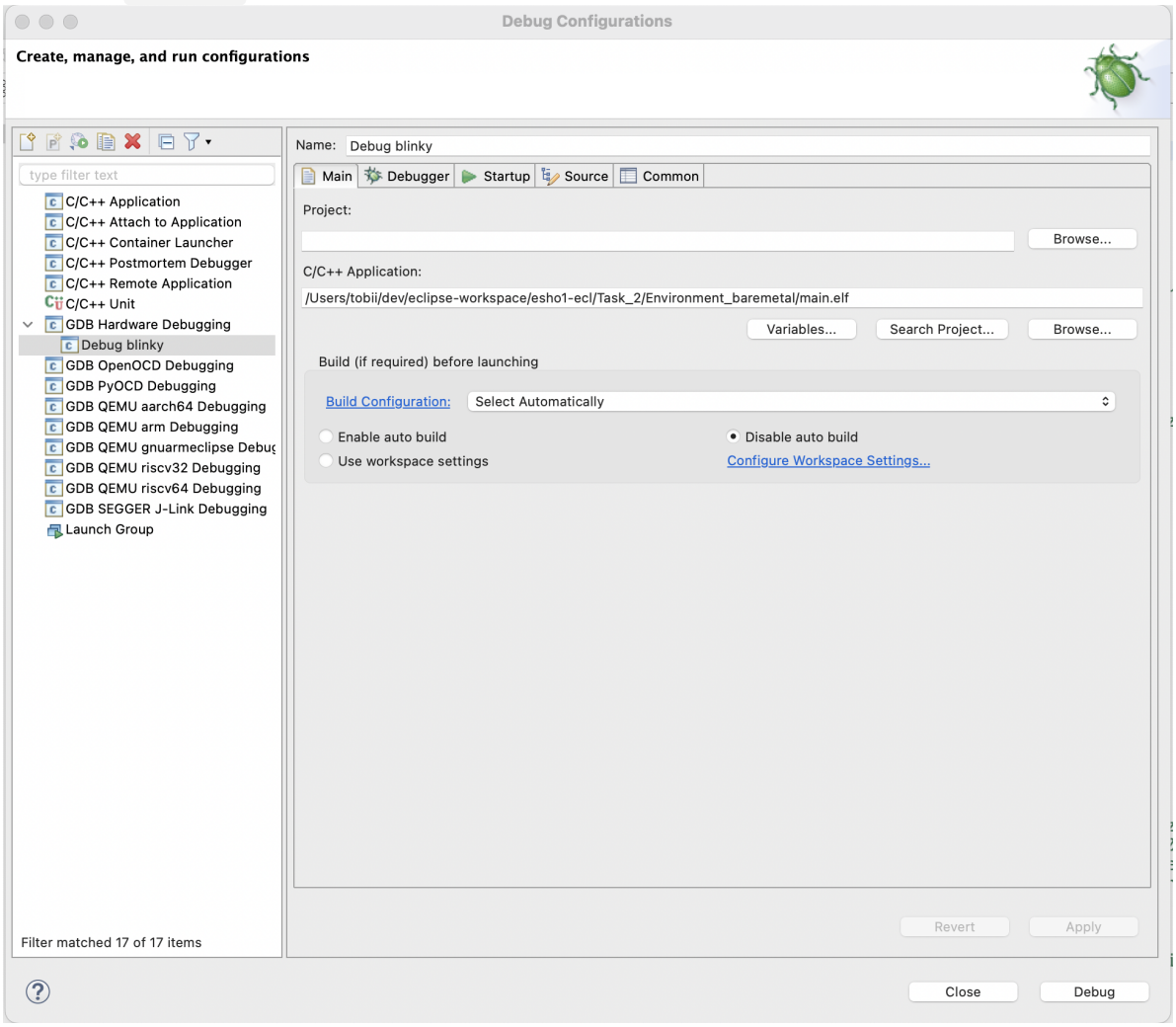
```
openocd -f esho1/Task_1/stm32f0raspberry.cfg
```

- Create Breakpoints at `main()` and at `toggle_led()` by double-clicking

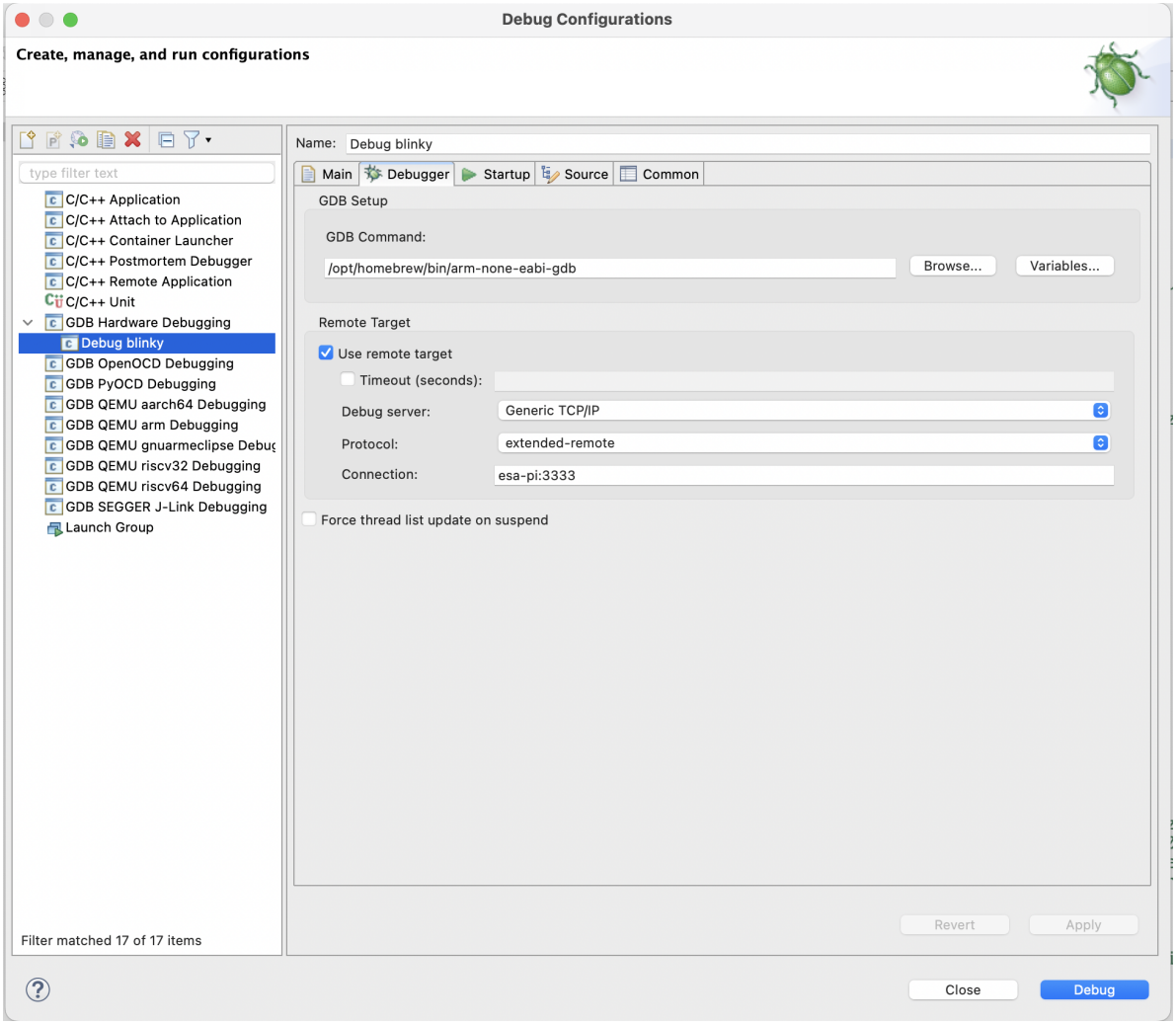


- Click "Run" → "Debug Configurations" and create a new "GDB Hardware Debugging" configuration:

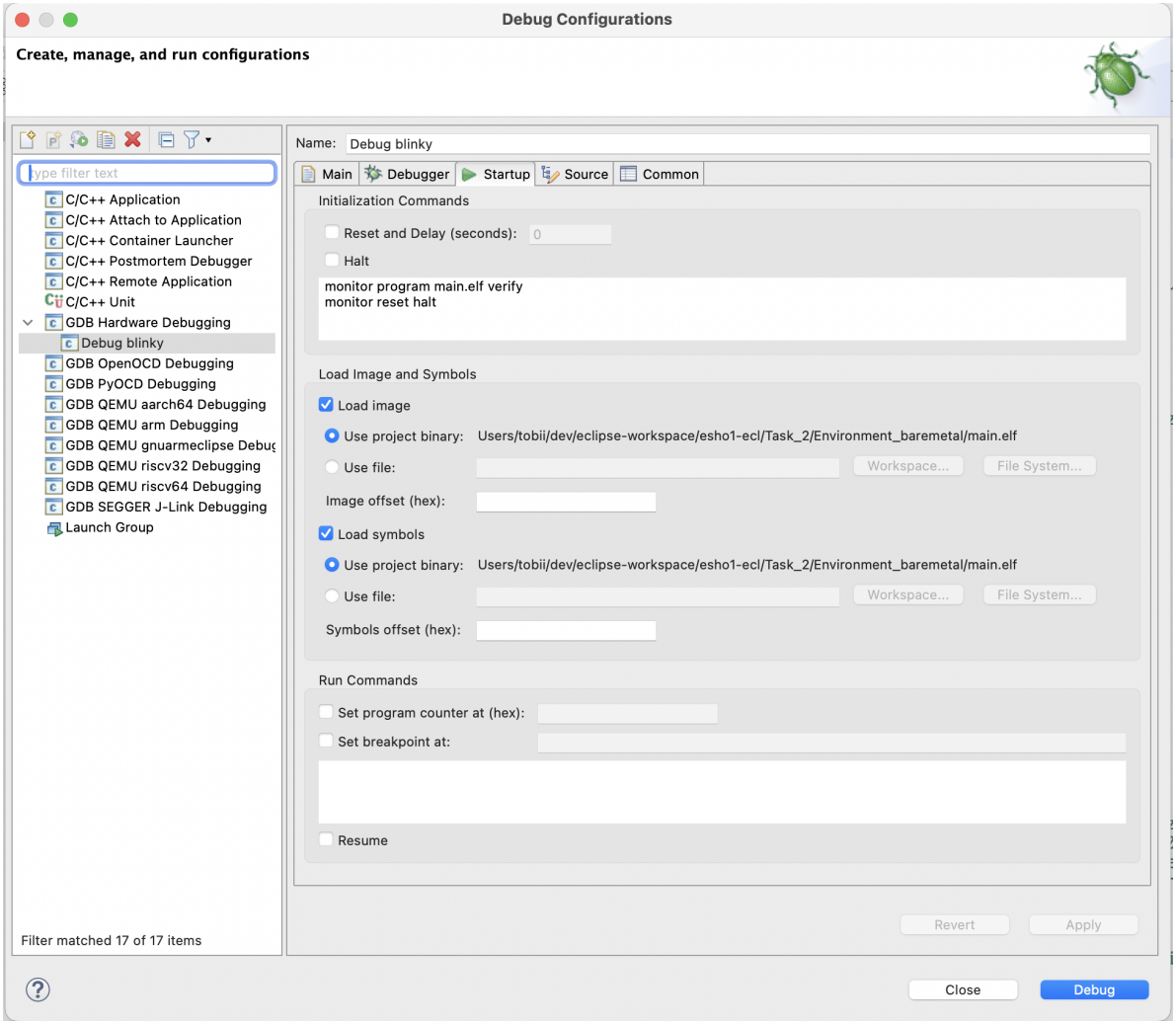
- Select the `main.elf` file and disable auto build



- Specify the full path to the debugger, select `Generic TCP/IP`, `extended-remote` and specify the hostname (or IP address) of the Raspberry Pi and the port `3333`



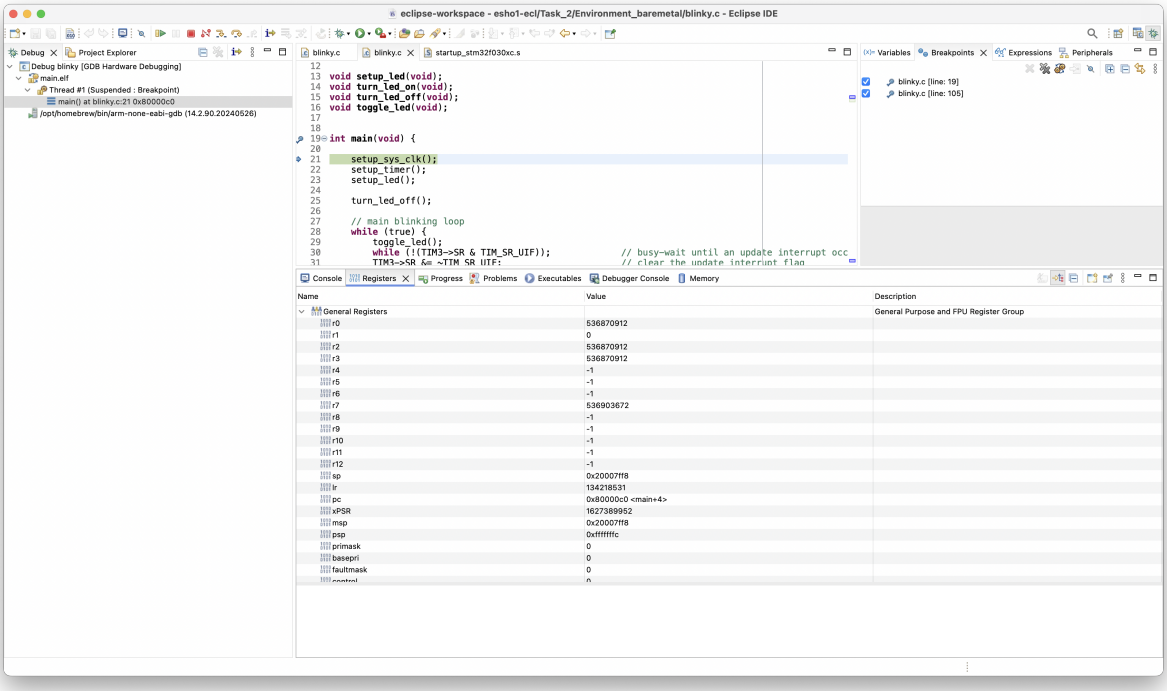
- Enter the Initialization Commands as shown and make sure that "Load image" and "Load symbols" are ticked



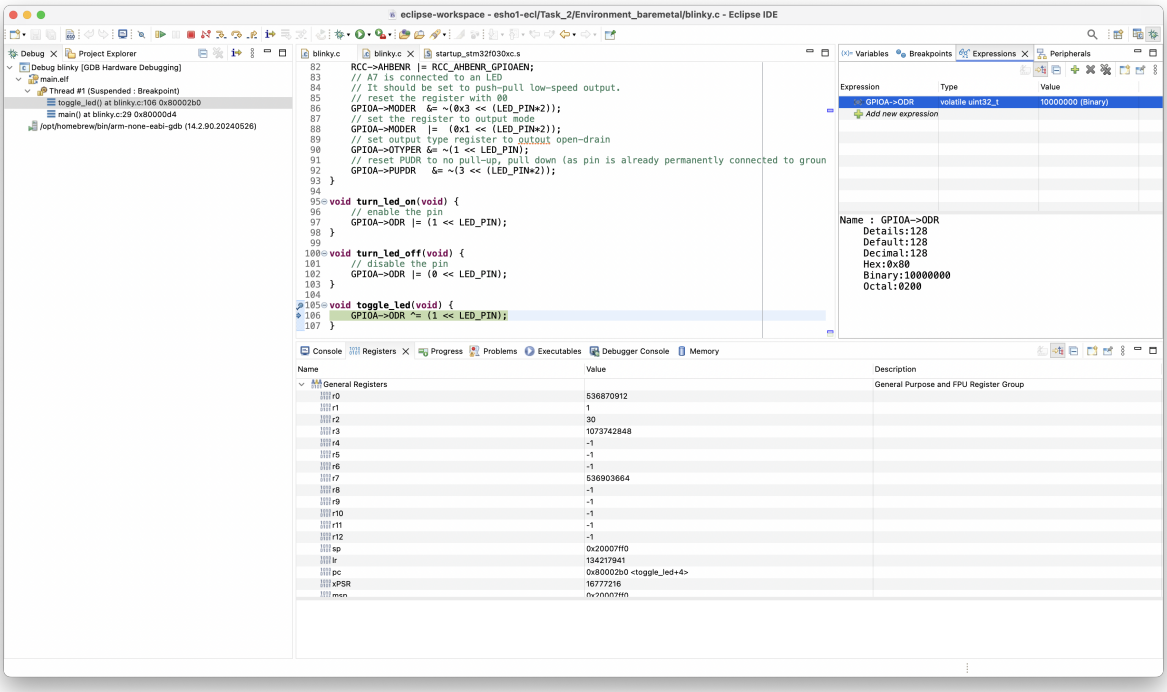
- No Changes to "Source" and "Common" are neccessary

- Click "Debug" and switch to "Debug View"

- After hitting "Resume" (F8), the execution stops at the beginning of `main()`. We can examine the registers with the "Registers" tab:



- After that, just like in 3.2, every time we hit "Resume", the LED gets toggled
- To examine the state of the GPIOA_ODR register, we can add a watch expression: Right-click anywhere on the code tab and select "Add Watch Expression...", then enter `GPIOA->ODR`. Now, every time we resume, we can see the value of the register change between `0b0` and `0b10000000`.

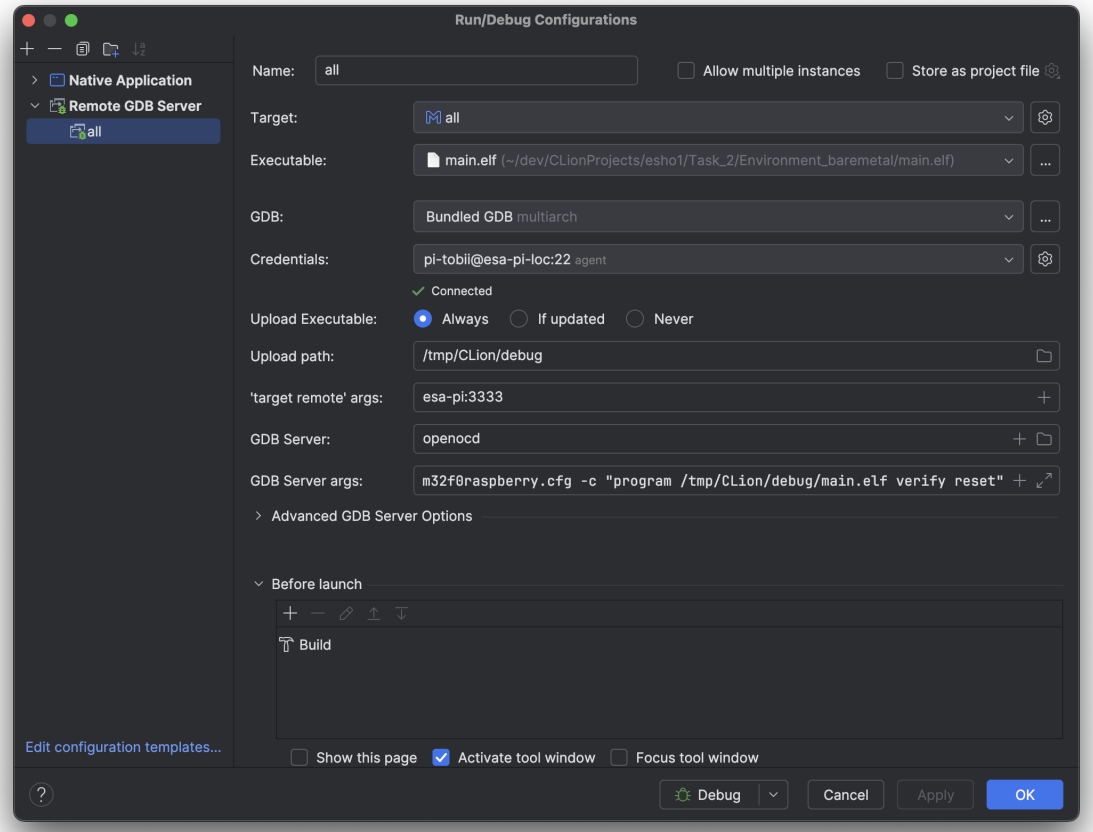


Use CLion instead of Eclipse and have an easy life

With CLion, the debugging process described above is much more simple. The compiled elf file can be copied to the Raspberry Pi, flashed onto the M0 and debugged all in one step by utilising the "Remote GDB Server" target:

- Install CLion from JetBrains Toolbox
- Get our ESHO1 project from Git via "New" → "Project from Version Control..." and entering `gitlab:ESHO1_24/Group07` as the URL
- Navigate to the Makefile of the task to test, right-click and select "Load Makefile Project"
  Note: The Makefile can be changed by doing this with another Makefile
- Build the project
- Add a new "Remote GDB Server" run configuration:
  - Select the Makefile target which results in the elf file to debug
  - Select the elf file to flash and debug
  - At "Credentials", setup an ssh connection to the Raspberry Pi
  - At "Target Remote args", enter the hostname or IP address of the Raspberry Pi and the port 3333
  - At "GDB Server", enter `openocd`
  - Enter the following "GDB Server args":

```
-f <absolute path to stm32f0raspberry.cfg>
-c "program /tmp/CLion/debug/main.elf verify"
-c "reset halt"
```



- Hit the Debug Button

Troubleshooting

Problems might arise when CLion tries to connect to the debugger before OpenOCD has finished flashing the elf file. A hacky way around that is to force gdb to sleep before trying to connect by redefining the `target remote` command. This way, you can also fix that CLion does not provide a way of using `target extended-remote` instead of `target remote`:

- Create a file `.gdbinit` at your home directory with the following content:

```
define target remote
shell sleep 5
target extended-remote $arg0
end
```

Task 4

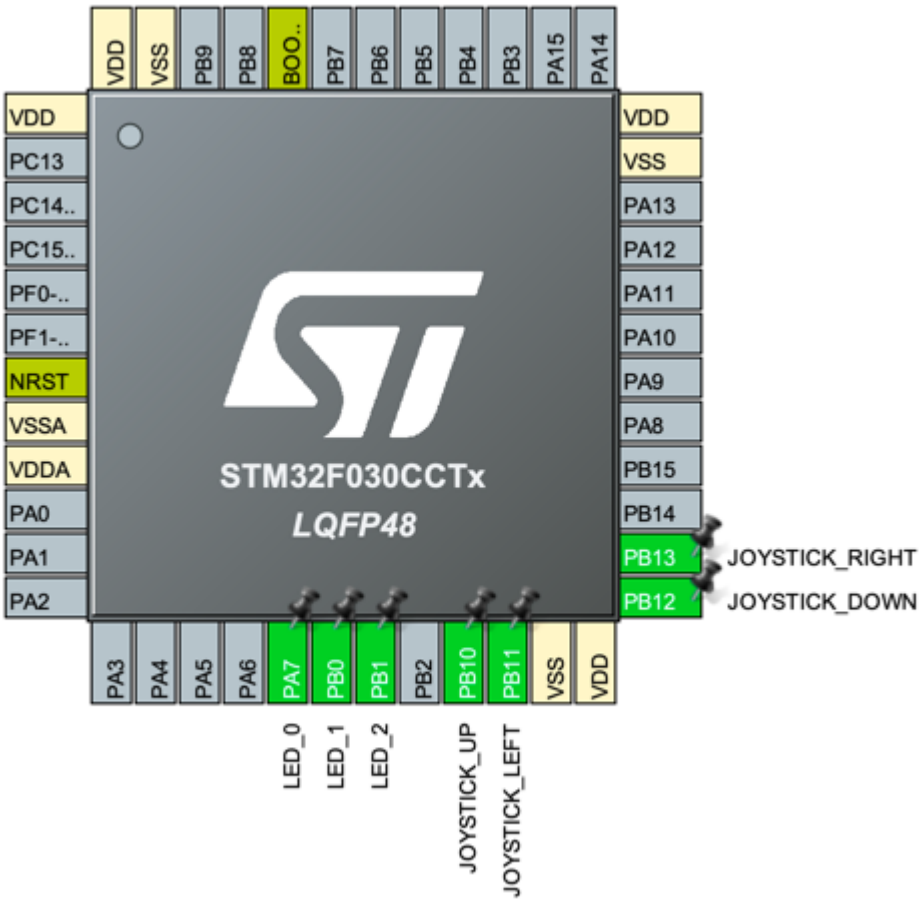
Further files of this task: See `Task_4/`.

4.1 Better blinky with HAL

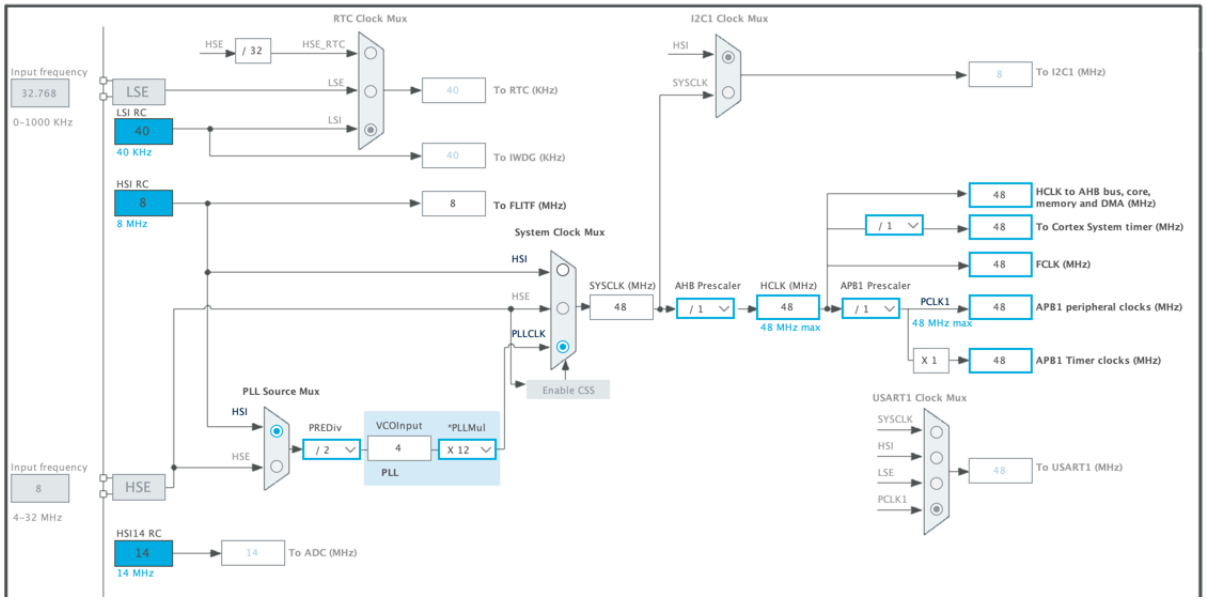
In this and subsequent HAL tasks, we used STM32CubeMX. It generates a project structure and code that initialises the M0 features needed for each task.

In this task, we used STM32CubeMX to

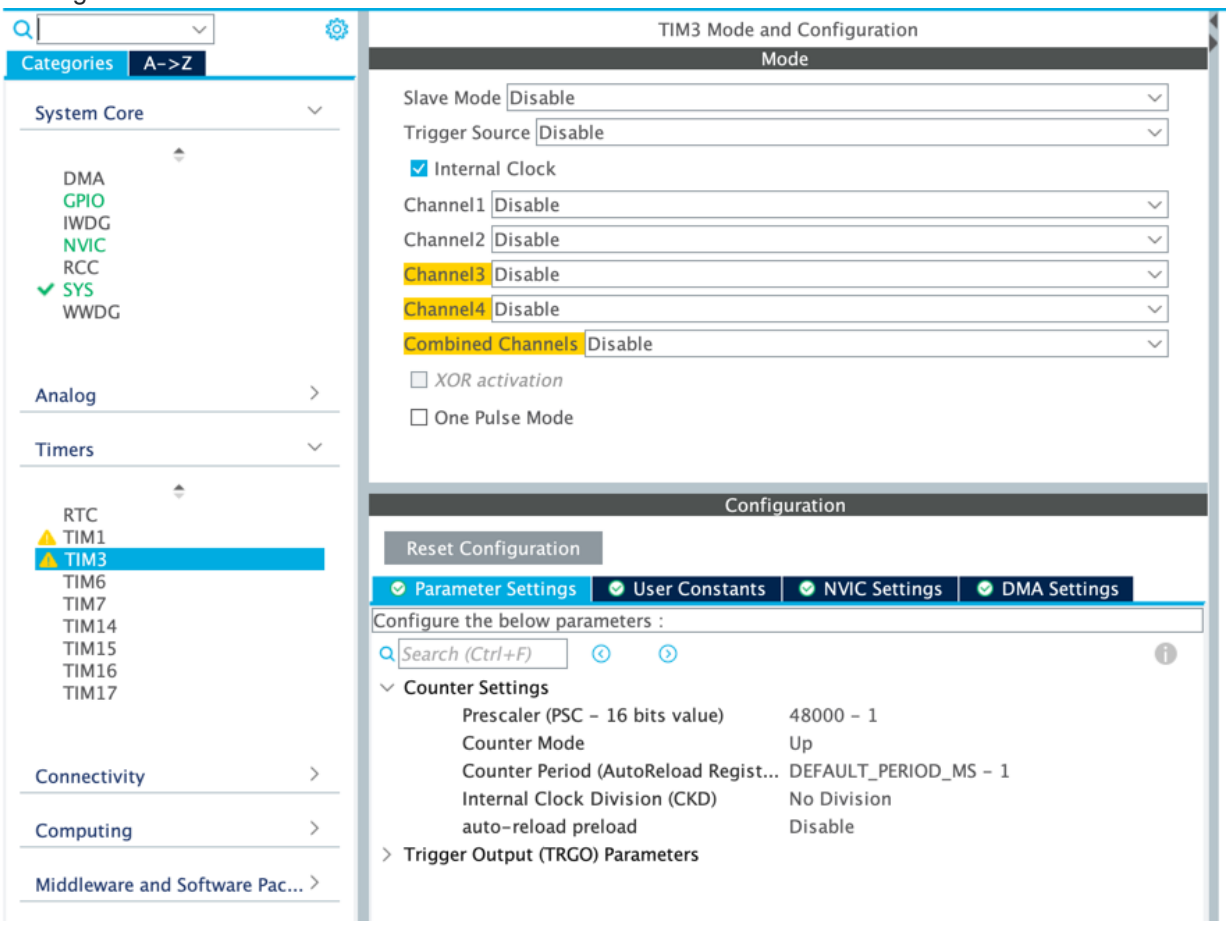
- set the GPIO Input/Output pins on the chip,



- set the system clock to 48 MHz and



- to configure TIM3



using the same settings we set manually in Task 2.

Note: We needed to adapt the resulting code and the `CMakeLists.txt` file to our needs and to our project structure, so regeneration of the code from STM32CubeMX will arise problems.

To compile the HAL tasks, use CMake:

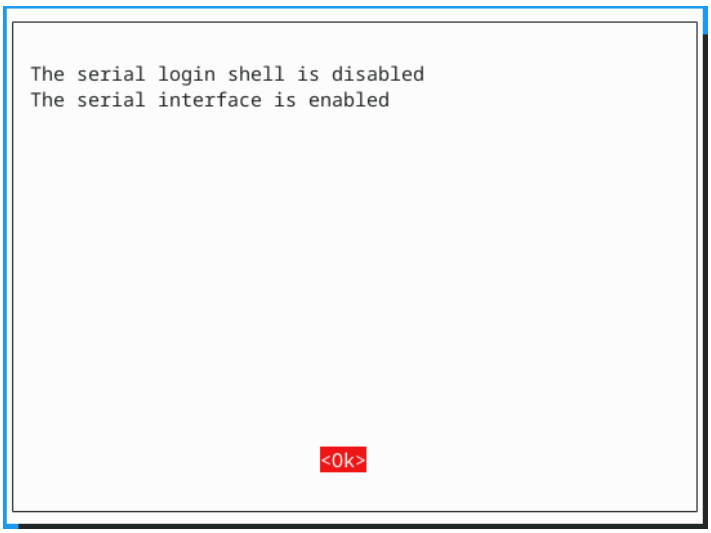
```
mkdir build
cd build

cmake ..
make
```

4.2 UART

To use UART, we need to enable it first

- open the raspberry pi config with: `sudo raspi-config`
- select `3. Interace options`
- select `Serial Port`
- It will ask you two questions.
  1. `Would you like a login shell to be accessible over serial?` -> select `No`
  2. `Would you like the serial port hardware to be enabled?` -> select `Yes`
- You should see the following message:



- select `OK`
- select `finish and reboot`

The involved GPIO pins on the raspberry pi for the UART communication are

- 15 (rx = receive)
- 14 (tx = transmit)

On our extension board, they are connected to USART 5 on the M0

On the raspberry pi we can now select `/dev/serial0` as UART port.

Note: `/dev/serial0` is a symbolic link to `/dev/ttyS0`

For more information, see the official documentation (Configuration - Raspberry Pi Documentation)

Python Software Preparations on the Raspberry Pi

We use the `pyserial` python library, which can be installed via `pip` in the following way:

```
# Change to your home folder
cd

# Install python3 and pip
sudo apt install python3-full pip

# Create and activate a virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install pyserial
pip3 install pyserial
```

Execute Benchmark

1. Compile/Run the M0 Code normally
2. Start the Python Code for the Raspberry Pi (`Code-raspi/main.py`). The script `raspi-run.sh` can be used to simplify this when you are on your host machine:

   ```
   ./raspi-run.sh <Raspberry Pi's Hostname/IP/SSH-Host-Name> <Raspberry Pi's username>
   ```

   Note: Within CLion, the Build Target "Shell Script" can be used to call the script. Be sure to set the working directory to 2_Universal_Asynchronous_Receiver_Transmitter.

Testing UART with 1000x10 Bytes:

```
UART_Benchmark started with 1000 times 10 bytes.
Test succeeded for baudrate   9600. Elapsed time: 20.95 s.
Test succeeded for baudrate  19200. Elapsed time: 10.53 s.
Test succeeded for baudrate  57600. Elapsed time: 3.56 s.
Test succeeded for baudrate 115200. Elapsed time: 1.82 s.
UART_Benchmark finished successfully.
```

We can observe that the time required for the communication approximately decreases linearly to the baudrate.

Side note about USART vs UART

UART stands for **U**niversal **A**synchronous **R**eceiver **T**ransmitter whereas the **USART** stands for **U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter. The term Synchronous enables the USART to send an additional clock signal to the receiving device. The data is then sampled at a predefined edge (Rising or Falling) of the clock. The USART mode uses **3 pins** (clock, Tx and Rx) compared to the **2 pins** (Tx and Rx) used in the UART. Source 26.6.24

Task 3

Addresses and datasheets

v3 Sensor Board:

Sensor	Datasheet	Slave Address (default) (7bit)	read address (8bit)	write address (8bit)
LPS331AP (barometer)	URL	1011101b	10111011b	10111010b
LSM404D (accelerometer and magnetometer)	URL	0011101b	00111011b	00111010b
L3GD20H (gyro)	URL	1101011b	11010111b	11010110b

Note: The L3GD20H sensor will not be used in this task, as reading gyroscope measurements is not required.

v5 Sensor Board:

Sensor	Datasheet	Slave Address (default) (7bit)	read address (8bit)	write address (8Bit)
LSM6DS33 (gyro and accelerometer)	URL	1101011b	11010111 (D7h)	11010110 (D6h)
LIS3MDL (magnetometer)	URL	0011110b	00111101 (3Dh)	00111100 (3Ch)
LPS25H (barometer)	URL	1011101b	10111011 (BBh)	10111010 (BAh)

The source code was partly inspired by the official Arduino libraries for the sensors.

- GitHub - pololu/lsm6-arduino: Arduino library for Pololu LSM6DS33 and LSM6DSO boards (under MIT License)
- GitHub - pololu/lps-arduino: Arduino library for Pololu LPS25H and LPS331AP boards (under MIT License)
- GitHub - pololu/lis3mdl-arduino: Pololu Arduino library for LIS3MDL magnetometer (under MIT License)
- GitHub - pololu/lsm303-arduino (Under MIT License)

Measurement Process

The M0 waits for two numbers to be sent via UART at first:

1. The number of measurements to take
2. Additional delay between each measurement in ms (Note: Without additional delay, expect to receive about one new measurement dataset per second with the v3 board. This is due to the M0 waiting for each sensor to have new data available.)

The measurements are then transmitted via UART in a CSV-like style.

To run this process:

1. Compile/Run the M0 Code normally
2. Start the Python Code for the Raspberry Pi (`python-uart/main.py`).
    - As in 4.2, `raspi-run.sh` may be used to simplify this when you are on your host machine.
    - When `main.py` is called without any arguments, it asks for the number of measurements to take and if you want an additional delay between each measurement
    - `main.py` may be called with arguments instead so that the output can be redirected into a csv file directly.

      ```
      python3 main.py <number of measurements> <delay> > file.csv
      ```

    - Note: Activate the virtual environment before calling `main.py` manually.
3. While the M0 is still running, `main.py` can be called again as often as needed.

Exemplary UART transfers

An example run of `main.py` :

```
pi-tobii@esa-pi:~ $ source .venv/bin/activate
(.venv) pi-tobii@esa-pi:~ $ python3 main.py 15 3000
>>>     Barometer ID: 0xbb
>>> Accelerometer ID: 0x49
>>>  Magnetometer ID: 0x49
 t [s], T [K],   p [Pa],a_x [m/s^2],a_y [m/s^2],a_z [m/s^2],  B_x [T],  B_y [T],  B_z [T]
  0.01,315.49, 89284.64,       0.10,       0.09,      -8.25, 0.000017, 0.000046, 0.000031
  3.02,315.49, 89281.03,      -0.80,      -9.00,      -0.14, 0.000029, 0.000060,-0.000022
  6.04,315.50, 89292.24,      -9.95,      -0.40,       2.09, 0.000042,-0.000006,-0.000034
  9.05,315.50, 89290.84,      -0.48,       9.50,       0.73, 0.000009,-0.000039,-0.000033
 12.06,315.49, 89295.65,       5.26,      -1.80,       4.51, 0.000034, 0.000055,-0.000005
 15.07,315.48, 89293.97,       6.10,       5.70,      -3.06,-0.000013,-0.000010, 0.000041
 18.09,315.44, 89290.36,       1.53,       9.19,      -1.08, 0.000004,-0.000023, 0.000034
 21.10,315.45, 89283.35,      10.25,      -0.67,       2.33,-0.000031, 0.000032, 0.000008
 24.11,315.43, 89301.20,       0.50,      -2.09,      11.09, 0.000013, 0.000047,-0.000038
 27.12,315.48, 89285.25,      14.62,       4.47,      10.50, 0.000039, 0.000040,-0.000025
 30.14,315.41, 89290.75,      19.60,      11.19,      18.38, 0.000038, 0.000037,-0.000027
 33.15,315.34, 89287.01,      -0.99,       9.69,       1.17, 0.000008,-0.000032,-0.000033
 36.16,315.34, 89290.94,     -10.15,       1.92,       1.50, 0.000032,-0.000015,-0.000029
 39.18,315.36, 89284.91,      -9.81,       0.58,       2.17, 0.000035,-0.000008,-0.000033
 42.19,315.34, 89283.89,      -9.70,      -0.13,       2.22, 0.000036,-0.000006,-0.000034
>>> End of measurements.
```

During measurement, the Raspberry Pi has been tilted and shaken, so a change in the acceleration values can be observed.
The pressure values are quite low as this task was done during holiday in the Alps.

Task 4

The measurement process is identical to Task 3.

Calibration

Before any measurement can be received, calibration has to happen.
Set the minimum illuminance by moving the joystick downwards, and the maximum illuminance by moving the joystick upwards.

Between measurements, the calibration values are kept. To delete the calibration, reset the M0. At any time, minimum or maximum illuminance may be recalibrated by moving the stick upwards or downwards again.

The Board LEDs show the status of the calibration:

- LED_1 (red): on while not fully calibrated
- LED_0 (green): steady while minimum illuminance calibrated, flashing after recalibration of minimum illuminance
- LED_2 (green): steady while maximum illuminance calibrated, flashing after recalibration of maximum illuminance

Exemplary UART transfers

```
pi-tobii@esa-pi:~ $ source .venv/bin/activate
(.venv) pi-tobii@esa-pi:~ $ python3 main.py
Number of measurements?               > 15
Additional delay between measurements? > 3000
>>> Ready to calibrate.
>>> Warning: Calibration failed. Min value has to be smaller than max value!
>>> Please try again.
>>> Calibration: min =  231.30, max = 3959.20
 t [s],E_V [%]
  0.00,  99.97
  3.00,  98.06
  6.00,  96.61
  9.01,  75.80
 12.01,  49.27
 15.01,  98.01
 18.01,  98.39
 21.01,  98.44
 24.01,  69.98
 27.01,  72.47
 30.01,  42.99
 33.01,   0.61
 36.01,  98.09
 39.02,  87.55
 42.02,  37.47
>>> End of measurements.
```

During measurement, a hand and different objects were moved across the sensor.

Documentation of Task 05

Further files of this task: See `Task_5/`.

5.1 Simulation of Analog Filters

Simulation schematic:
Open 1_Simulation_of_Analog_Filters/Filters/Filters.kicad_sch with KiCad

Resistor and Capacitor values:
Given:

- f_LP = 1500Hz
- f_HP = 4500Hz
- L = 0.1

Calculated:

- R_HP = 2*pi*L*f_HP = 900pi Ohm = 2827.43 Ohm
- C_HP = 1/(2*pi*R_HP*f_HP) = 1/(8100000pi²) Farad = 1.25088e-8 Farad ≈ 12.5 nF
- R_LP = 2*pi*L*f_LP = 300pi Ohm = 942.48 Ohm
- C_LP = 1/(2*pi*R_LP*f_LP) = 1/(900000pi²) Farad = 1.12579e-7 Farad ≈ 112.5 nF

We used the following parameters for our simulations: {500Hz, 1500Hz, 2500Hz, 3500Hz, 4500Hz, 6500Hz, 8500Hz}.

Transient simulation:
View 1_Simulation_of_Analog_Filters/Simulation/Transient Simulation/LinearTransient*

General: Input signal in red, Output signal in blue

The input signal is the combination of a 1kHz sinus wave and a 5kHz sinus wave.

As expected the high pass filter both with capacitor and with coil lead to an identical output signal using the computed values. The same holds true for the low pass filter. As the resulting plots are bit-wise identical, we do not provide different plots for RL_HP and RC_HP or RL_LP and RC_LP.

With a cutoff frequency of 1500 Hz, the low pass filter dampens the 5kHz signal and lets the 1kHz signal through:

With a cutoff frequency of 4500 Hz, the high pass filter dampens the 1kHz signal and lets the 5kHz signal through:

In both cases the dampening is not complete and remains of the dampened sinus wave are visible.

If we decrease the cutoff frequency of the low pass filter, the resulting signal contains less disturbance but the amplitude is lower, because the low frequencies get attenuated more. As expected, the low-pass filter has less

and less influence on the input signal as the cutoff frequency is increased.

Increasing the cut-off frequency of the high-pass filter results in a lower amplitude of the output signal. If the cutoff frequency is reduced, the disturbances in the output signal increase until it is almost identical to the input signal again.

Frequency response simulation:

We generated plots with all combinations of cut-off frequencies for high and low pass filters:

f_span = {500Hz, 1500Hz, 2500Hz, 3500Hz, 4500Hz, 6500Hz, 8500Hz}

LP x HP = f_span x f_span

View 1_Simulation_of_Analog_Filters/Simulation/Frequency Response Simulation/LinearAC_fLPf_span_fHP{f_span}.png.

The frequency response diagram shows the effect of the high and low pass filter (high pass: blue, low pass: orange). It varies according to the cut-off frequencies.

As expected, a higher cut-off frequency for the high-pass filter leads to a stronger attenuation of lower frequencies. With the low-pass filter, it is the other way around - as the cut-off frequency increases, lower frequencies are allowed through more and more.

Phase Shift:

View 1_Simulation_of_Analog_Filters/Simulation/Phase Shift Simulation/*.png

We can observe that the more the filters influence the output signal, the greater the phase shift. Specifically, the phase shift is highest for low-pass filters with low cut-off frequencies and for high-pass filters with high cut-off frequencies. The phase shift is always negative for the low-pass filter and positive for the high-pass filter.

Damping of all filters at 1kHz and 5kHz:

Calculated using 1_Simulation_of_Analog_Filters/DampingComputation.py:

```
High-Pass Filter
High-pass gain formula: 1/(sqrt(1+(1/(2*pi*C*R)²)))
Frequency: 1000   Cut-off Frequency:  500   Damping: -0.97 dB
Frequency: 1000   Cut-off Frequency: 1500   Damping: -5.12 dB
Frequency: 1000   Cut-off Frequency: 2500   Damping: -8.60 dB
Frequency: 1000   Cut-off Frequency: 3500   Damping: -11.22 dB
Frequency: 1000   Cut-off Frequency: 4500   Damping: -13.27 dB
Frequency: 1000   Cut-off Frequency: 6500   Damping: -16.36 dB
Frequency: 1000   Cut-off Frequency: 8500   Damping: -18.65 dB
Frequency: 5000   Cut-off Frequency:  500   Damping: -0.04 dB
Frequency: 5000   Cut-off Frequency: 1500   Damping: -0.37 dB
Frequency: 5000   Cut-off Frequency: 2500   Damping: -0.97 dB
Frequency: 5000   Cut-off Frequency: 3500   Damping: -1.73 dB
Frequency: 5000   Cut-off Frequency: 4500   Damping: -2.58 dB
Frequency: 5000   Cut-off Frequency: 6500   Damping: -4.30 dB
Frequency: 5000   Cut-off Frequency: 8500   Damping: -5.90 dB


Low-Pass Filter
Low-pass gain formula: 1/(sqrt(1+(2*pi*C*R)²))
Frequency: 1000   Cut-off Frequency:  500   Damping: -6.99 dB
Frequency: 1000   Cut-off Frequency: 1500   Damping: -1.60 dB
Frequency: 1000   Cut-off Frequency: 2500   Damping: -0.64 dB
Frequency: 1000   Cut-off Frequency: 3500   Damping: -0.34 dB
Frequency: 1000   Cut-off Frequency: 4500   Damping: -0.21 dB
Frequency: 1000   Cut-off Frequency: 6500   Damping: -0.10 dB
Frequency: 1000   Cut-off Frequency: 8500   Damping: -0.06 dB
Frequency: 5000   Cut-off Frequency:  500   Damping: -20.04 dB
Frequency: 5000   Cut-off Frequency: 1500   Damping: -10.83 dB
Frequency: 5000   Cut-off Frequency: 2500   Damping: -6.99 dB
Frequency: 5000   Cut-off Frequency: 3500   Damping: -4.83 dB
Frequency: 5000   Cut-off Frequency: 4500   Damping: -3.49 dB
Frequency: 5000   Cut-off Frequency: 6500   Damping: -2.02 dB
Frequency: 5000   Cut-off Frequency: 8500   Damping: -1.29 dB
```

5.2 Measurement of Analog Filters

Screenshots from the oscilloscope

- Waveform of unfiltered and filtered noise signal :
  - Waveform_1msSpan_LowPassCoil.png
  - Waveform_2msSpan_LowPassCoil.png
  - Waveform_1msSpan_LowPassCap.png
  - Waveform_2msSpan_LowPassCap.png
- FFT of unfiltered and filtered noise signal :
  - FFT_Filtered_LowPassCoil.png
  - FFT_Filtered_LowPassCap.png
  - FFT_Unfiltered_LowPassCoil.png
  - FFT_Unfiltered_LowPassCap.png

- Waveform of unfiltered and filtered sine signal :
    - Sinus_LowPassCoil.png
    - Sinus_LowPassCap.png

Diagrams from frequency response and filtered signals

- Waveform of unfiltered and filtered noise signal :
    - WhiteNoiseSignal_LowPassCoil_600K.png
    - WhiteNoiseSignal_LowPassCoil_6M.png
    - WhiteNoiseSignal_LowPassCap_600K.png
- FFT of unfiltered and filtered noise signal :
    - FFT_LowPassCap_600K.png
    - FFT_LowPassCap_600K_smoothed.png
    - FFT_LowPassCoil_6M.png
    - FFT_LowPassCoil_6M_smoothed.png
    - FFT_LowPassCoil_600K.png
    - FFT_LowPassCoil_600K_smoothed.png

(generated with postProcessing.py)

Comparison between measured and simulated behavior of signals
The FFT diagrams of the previous subtask are similar enough so that we just focus on FFT_LowPassCoil_6M. This data section has the most prevision. We already found out in 5.1 that the Low pass filter using a coil and a capacitor performs identical. Looking at FFT_LowPassCoil_6M.png shows a FFT that is not smooth at all. One can see a low-pass behavior but using the graph alone one would guess the cut-off to be between 500Hz and 2000Hz. To improve the visibility we smooth the FFT diagram with a Savgol filter using polynomials or order 5 and 30 basis points. This results in the diagram visible in FFT_LowPassCoil_6M_smoothed.png. The diagram is clearer and the cut-off frequency can be recognized if already known to be at 1500Hz.

Cut-off Frequency within diagrams:
For the smoothed diagrams, we read out the cut-off frequency (manually) and added it to the diagram:

- FFT_LowPassCap_600K_smoothed_Cut-off.png
- FFT_LowPassCoil_6M_smoothed_Cut-off.png
- FFT_LowPassCoil_600K_smoothed_Cut-off.png

The cut-off frequencies cannot be read off very precisely, as the data still contains significant fluctuations even after smoothing. However, a noticeable qualitative attenuation of the frequencies above 1500 Hz can be recognized in all three diagrams, so that the measurement approximately meets the expectation.

Summary of the measurement procedure and the results in a structured document

See MeasurementProtocol.pdf

5.3 Simulation of Digital Filters

Frequency response of FIR filters:

- View 3_Simulation_of_Digital_Filters/*.png for Frequency response and the Matlab tool settings

Filter coefficients of Nuttal-FIR filters:

- View 3_Simulation_of_Digital_Filters/*.fcf for filter coefficients of the different filters

Filter coefficients of KaiserWindow-FIR filters:

- View 3_Simulation_of_Digital_Filters/KaiserWindow_Optim.fcf for filter coefficients and the order of the filter
- View 3_Simulation_of_Digital_Filters/KaiserWindow_Optim.png for Frequency response
- View 3_Simulation_of_Digital_Filters/KaiserWindow_Optim_Settings.png for Matlab tool settings

Nuttal-FIR filter vs Task 5.1 Lowpass @ 1.5kHz:
For N=5 and N=10 the analog filter is the steeper one. Only N=20 leads to a digital Nuttal filter with a steeper amplitude response. This can be seen by comparing the values at f=2kHz. The analog filter has a -12dB damping at that point while the Nuttal filter for N=5 and N=10 has a damping of > -10dB.

Nuttal-FIR filter N=10 vs Rectangular N=10:
Rectangular is steeper between 1.5kHz and 2kHz but has a lower damping above 2kHz, staying above -30dB for large frequency bands.

Nuttal-FIR filter N=20 vs Kaiser Window:
The Nuttal window N=20 filter drops to a much higher damping at high frequencies. Between 1.5kHz and 3kHz the Nuttal window filter is steeper.

5.4 Realization of Digital Filters

FIR implementation: The FIR Filter is implemented in the Header "FIRFilter.h"

FIR filter design:
Reading frequency on the microcontroller and the filter design interact with each other.
A larger filter leads to a lower sampling frequency and a lower frequency allows a smaller filter.
We choose a sampling frequency of 50Hz as a compromise.
There are no large amplitudes in the accelerometer above 10Hz. As a result we choose 20Hz as our upper pass frequency.
The lower pass frequency is chosen to be 0.2 Hz. This is a compromise. Lower values lead to a larger filter that does not fit on the microcontroller. Any signals with a period of more than 5s is damped as a result.
FilterDesignAnalysis/BandpassFilterDesign.png shows the design parameters
FilterDesignAnalysis/BandpassAmplitudeDesigner.png shows the resulting amplitudes above the frequencies.

FIR filter evaluation:
FilterDesignAnalysis/TransientPlot*.png show the plotted data track with the filtered and unfiltered signal.
More detailed insights into the data track can be gained by using the FilterDesignAnalysis/postProcessingAcceleration.py script in combination with the data file FilterDesignAnalysis/AccelerationDataTrack.csv
FilterDesignAnalysis/FFTPlot*.png show the effect of the filter on the amplitudes measured my the accelerometer. Our filter is effectively able to dampen low frequencies below 0.2Hz. The static gravity is dampened away.

The intermediate frequencies are kept.
This proofs that our filter design was successful.

5.5 Choice of Filter Implementation

Advantage analog filter:

- Can be implemented cheaply with electric components without the need for computation power
- No complication by the necessity to pay attention to the digital frequency
- No additional components necessary to bridge the analogue to digital gap

Advantage digital filter:

- Can be implemented by software and therefore easily changed
- Easy implementation of high order filters only limited by memory consumption
- Components can not change their values due to age

Documentation of Task 06

Further files of this task: See `Task_6/`.

6.1 Dimming a LED by PWM

Short explanation of the schematic

Circuit with 5V voltage source V2, the LED, a resistor R2, which limits the current through the LED, and a transistor. We use the transistor as a switch. When the voltage source V3 activates the circuit the transistor "switches" on and the LED lights up.

A small base current to the BJT results in a larger collector current. The relationship between those two currents is called the DC current gain (hFE). The DC current gain depends in particular on the temperature and collector current and, according to the data sheet for the BC817-25, ranges from 160 to 400 at an ambient temperature of 25 °C and a collector current of 100 mA and a Collector-to-Emitter voltage of 1 V.

Simulation

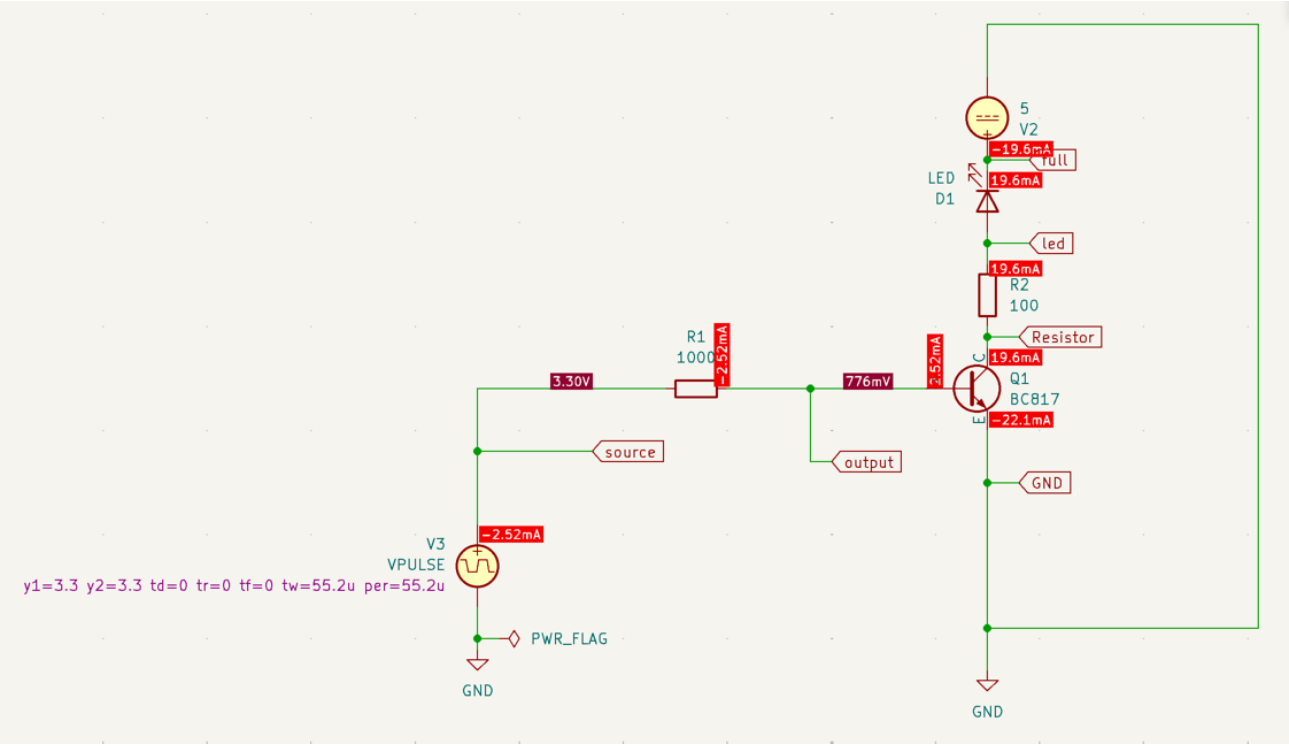
The simulated circuit is located at 1_Dimming_a_LED_by_PWM/Dimming_simulation.

Note: For the simulations to produce reasonable results, we had to change the polarity of the LED (or flip the voltage sources, alternatively) for some reason. We figure this might be due to a bug with the BC817 model.

Determining the Resistor Value for the LED

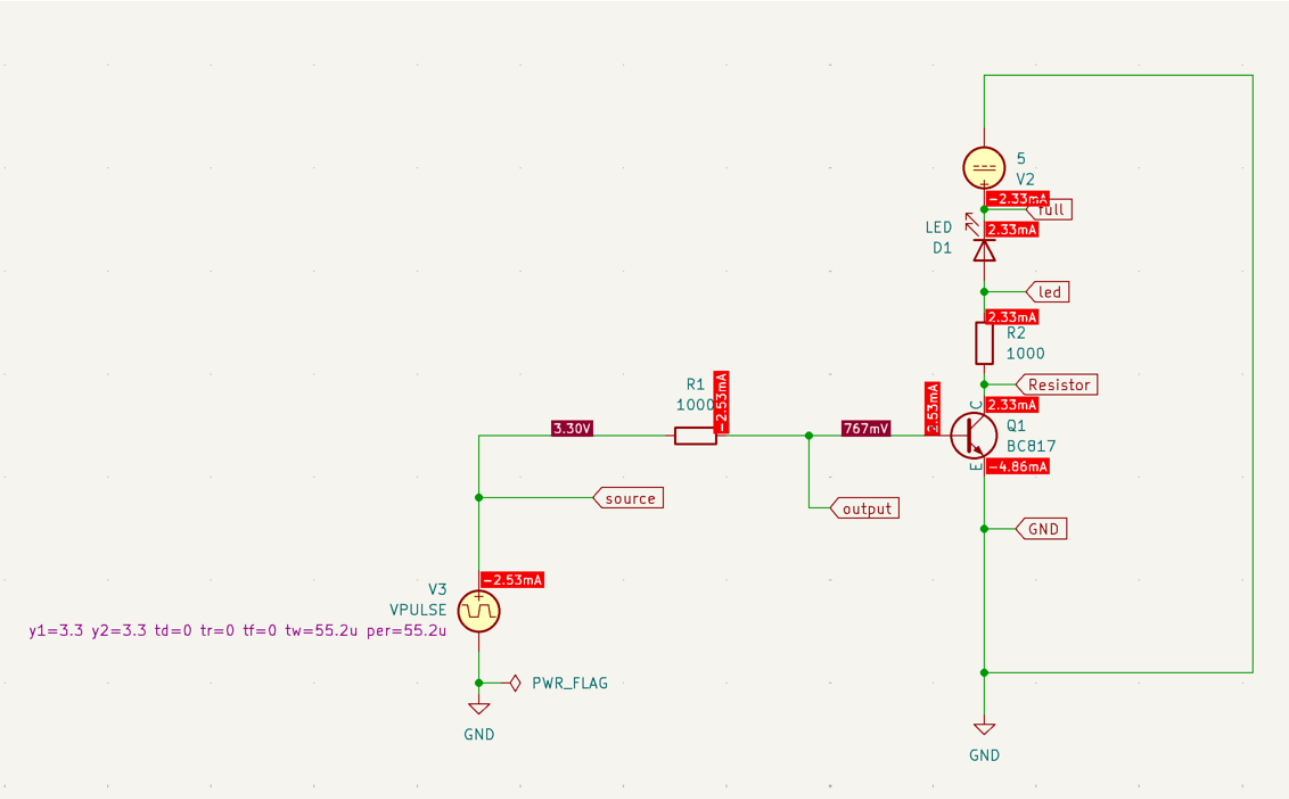
We performed an OP simulation to simulate the current flow through the LED.

The minimum resistance value seems to be about 100 Ohms, which leads to a simulated current flow of 19.6 mA:



In this case, the relationship between base current and collector current is 19.6 / 2.52 ≈ 7.78

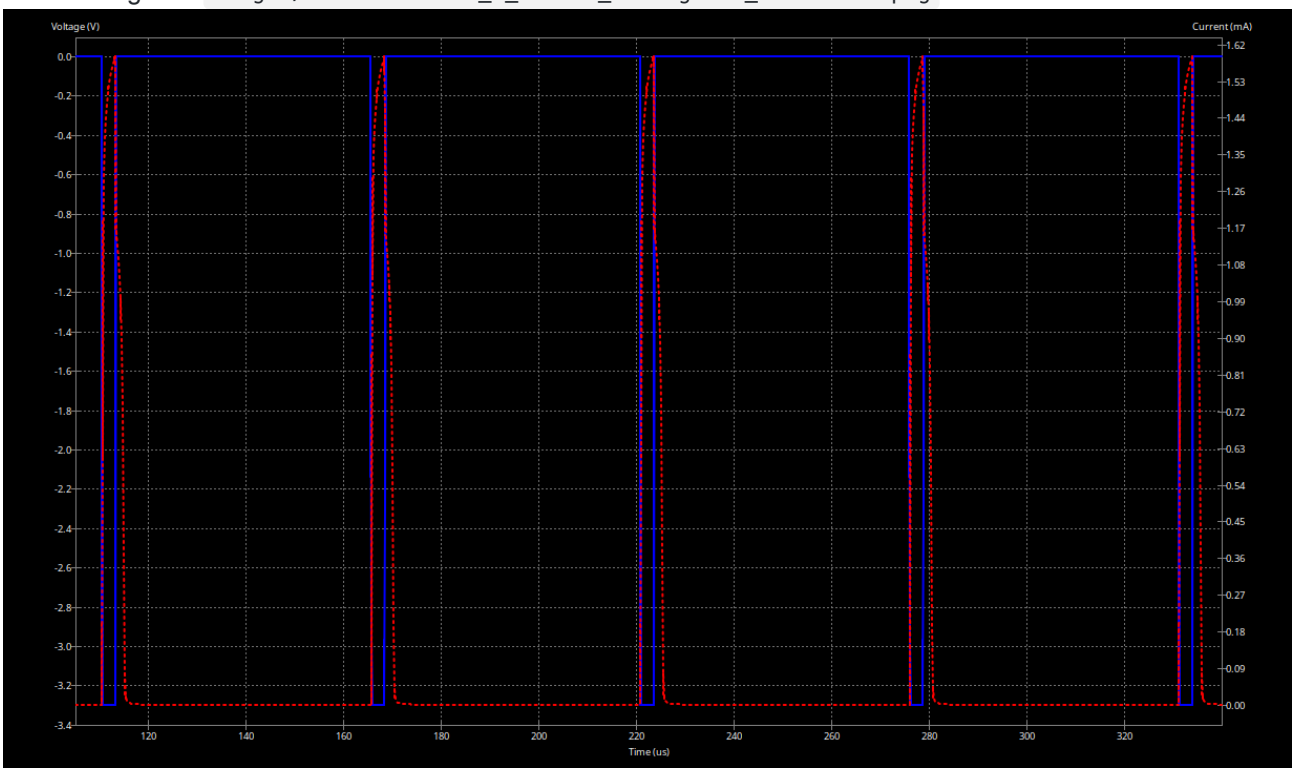
As this value is very close to the LED's limits, we figured a resistance value of 1000 Ohms would also be enough to light up the LED. In this case, only 2.33 mA flow through the LED:



The relationship between base current and collector current is 2.33 / 2.53 ≈ 0.92.

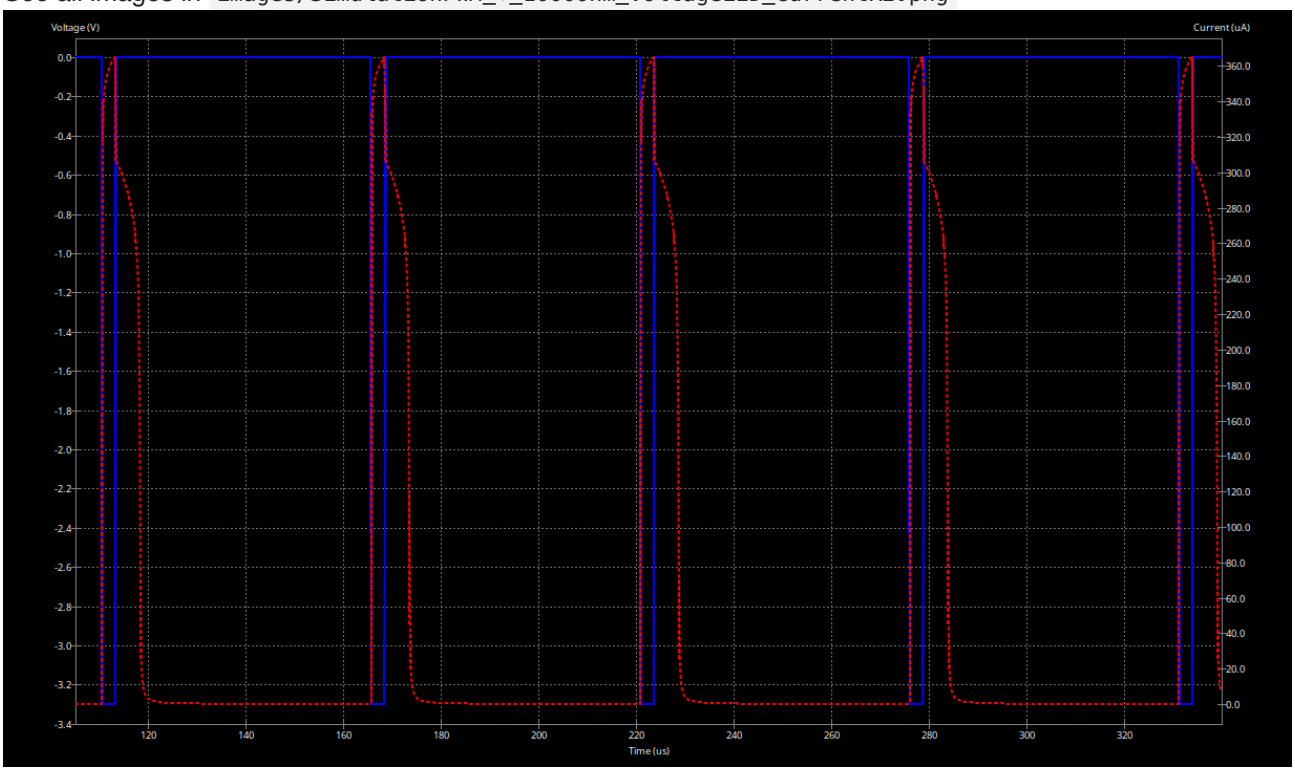
PWM Simulation

see all images in `images/SimulationPWM_*_100Ohm_VoltageLED_CurrentR2.png`



Example with 005perc: SimulationPWM_005perc_100Ohm_VoltageLED_CurrentR2

See all images in `images/SimulationPWM_*_1000Ohm_VoltageLED_CurrentR2.png`



Example with 005perc: SimulationPWM_005perc_1000Ohm_VoltageLED_CurrentR2

PWM Code

Within `1_Dimming_a_LED_by_PWM/Dimming_LED_pwm_code/` you'll find the code we used to perform the measurements in the Lab:

the code uses two timer `tim1` and `tim3` to control both the onboard LED_1 and an external LED (connected to `M4` / `PA9`)

Our software has three LED modes:

1. manually: use the joystick left and right to increase and decrease the brightness of the led
2. Auto-triangle mode: the LED increase and decreases the brightness with a triangle function
3. Heartbeat mode: the LED simulates a (human like) heartbeat

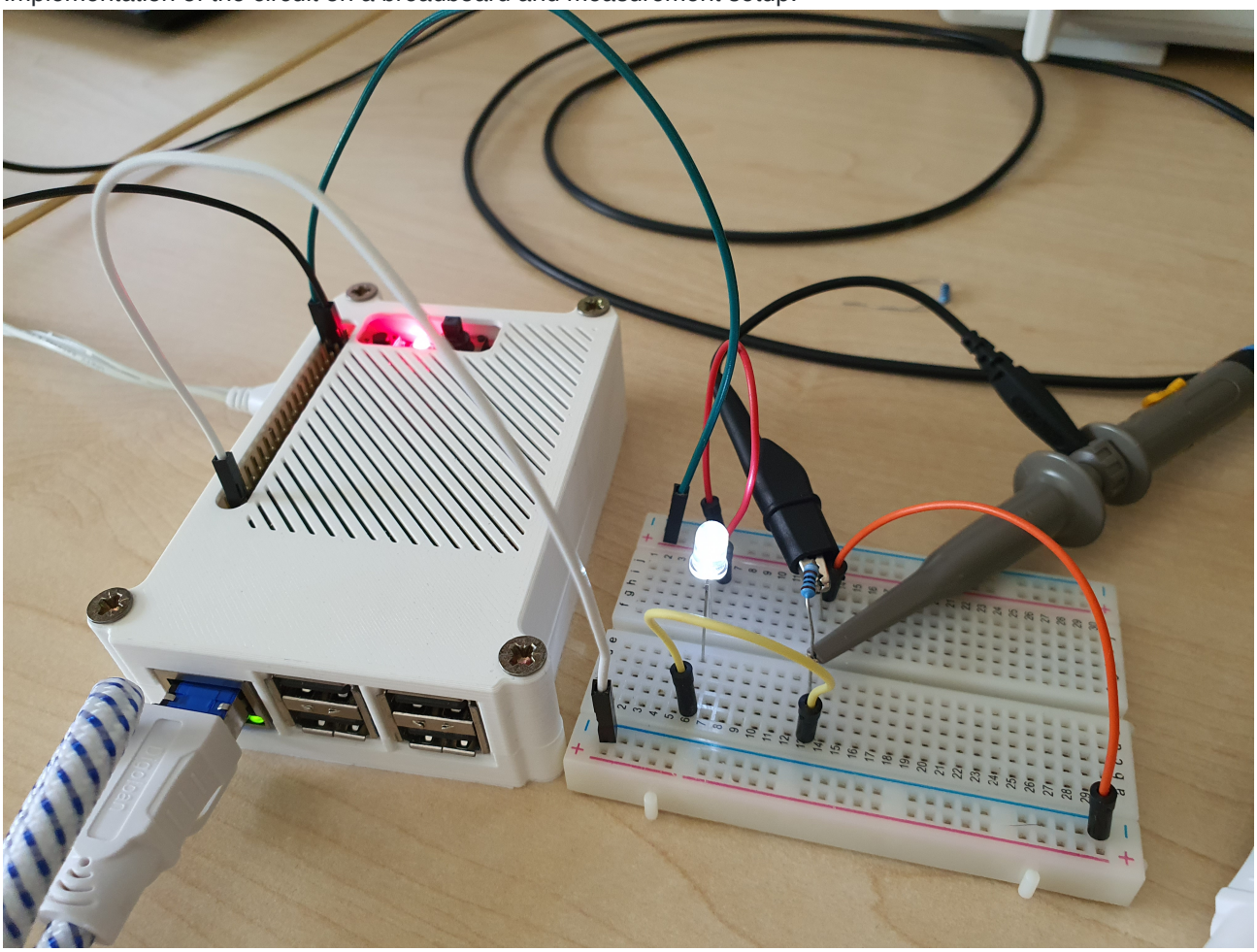
use the joystick up or down to iterate between the modes

To perform lab testings, set the `lab_mode` in main.cpp to true.

Both programs use a PWM frequency of 24 kHz. The duty cycle is initially set to 100 % and can be adjusted via the debugger, as described in main.cpp.

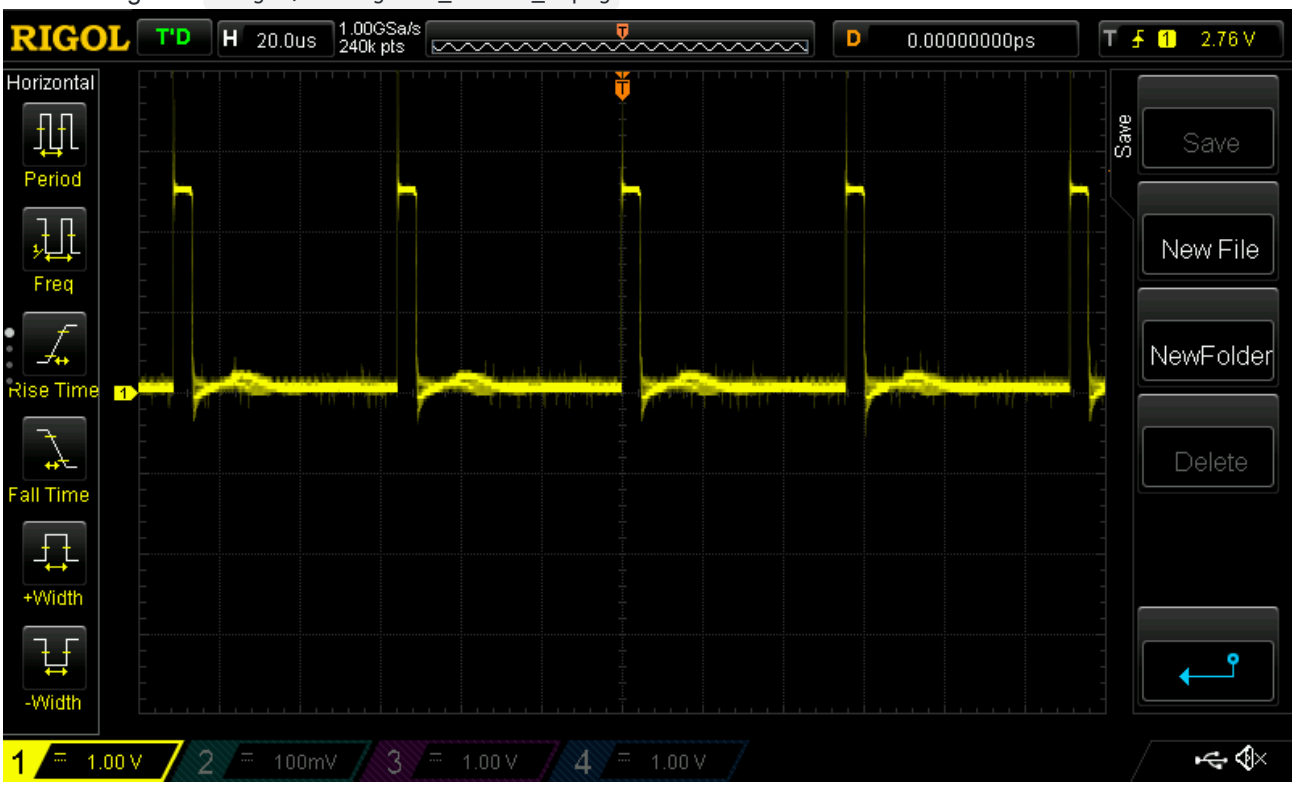
Measurements in Lab

Implementation of the circuit on a breadboard and measurement setup:



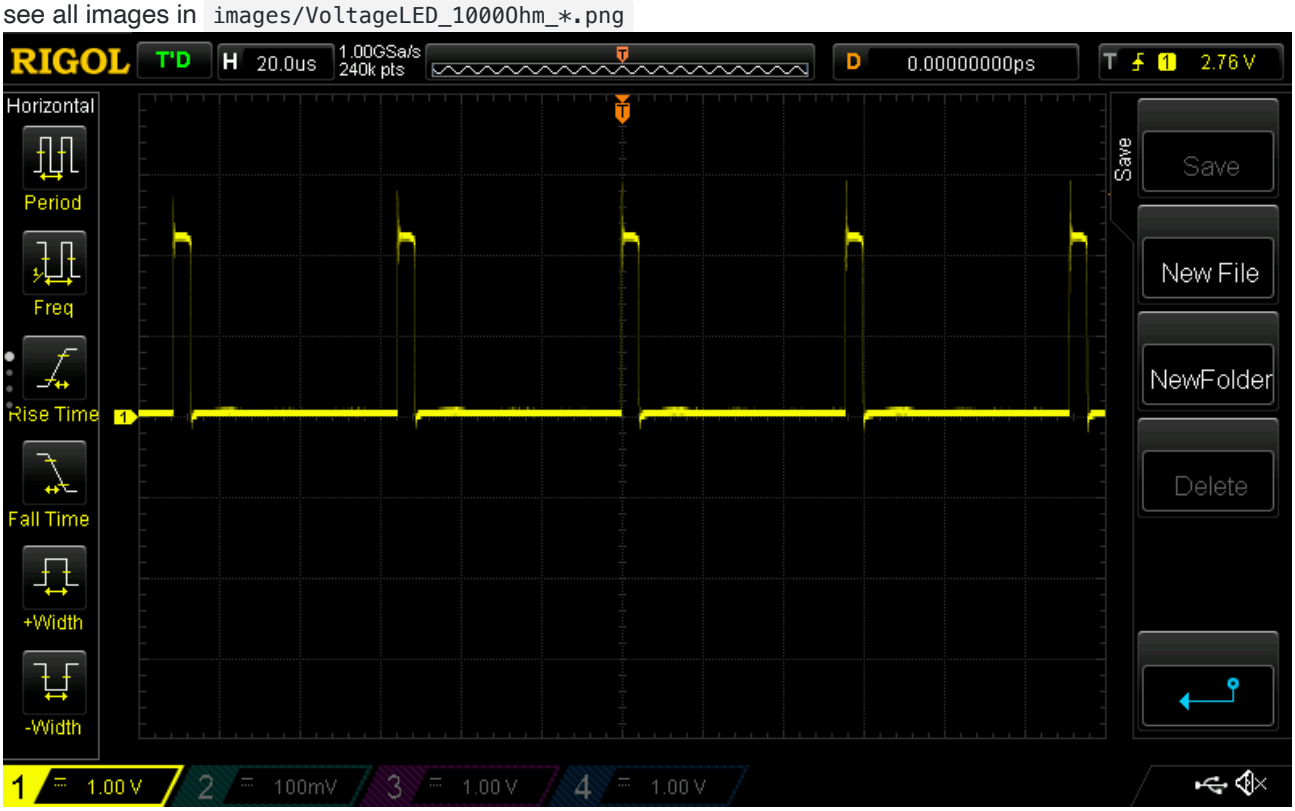
Voltage over LED with different duty cycles

see all images in `images/VoltageLED_1000hm_*.png`



Example for 005perc: VoltageLED_1000Ohm_005perc

see all images in `images/VoltageLED_1000hm_*.png`



Example with 005pers: VoltageLED_100Ohm_005perc

The simulated base voltage (from above) was made to fit the measured one. Obviously the simulated voltage is cleaner than the measured voltage.

Current through LED

As no ampere meter was available in the lab, the current has to be derived from the measured voltage (cf. U ≈ 2.4 V at `images/VoltageLED_100Ohm_100perc.png` and U ≈ 2.6 V at `images/VoltageLED_100Ohm_1000perc.png`):

U = R * I => I = U / R

- R = 100 Ohms: I = 2.4 V / 100 Ohms = 24 mA
- R = 1000 Ohms: I = 2.6 V / 1000 Ohms = 2.6 mA

Both measured values are a little higher than the simulated ones. Although the LED can withstand a maximum current flow of 25 mA according to the data sheet, we therefore carried out the majority of our measurements with a resistance of 1000 ohms for safety reasons so as not to damage the LED.

Voltage over the BJT (without LED) at 50% duty cycle

Cf. `images/VoltageRasPi_050perc.png`

Improved brightness curve

The human eye does not have a linear brightness sensitivity but a nearly logarithmic one. The non-1linearity allows us to see in low light situations like in moonlight (0.05-0.36lx) but also in full sunlight (130.000 lx) Wikipedia. We can also use a logarithmic scale to improve the brightness steps of our LED. We decided to use 100 steps, from 0 to 99, and calculate the current brightness with the formula: As `brightness_basis` we decided based on empirical research to use `3.26` as this gives us the best results.

```
pow(brightness_basis, log2(precision_pwm) * (new_brightness_step + 1) / number_of_steps) - 1;
```

This results in the following brightness steps, which we use as a new pulse for PWM.

```
[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3,
3, 4, 4, 4, 5, 5, 5, 6, 6, 7, 7, 8, 8, 9, 10, 11, 11, 12, 13, 14, 15, 17, 18, 19, 21, 23,
24, 26, 29, 31, 33, 36, 39, 42, 45, 49, 53, 57, 61, 66, 72, 77, 84, 90, 97, 105, 114,
123, 132, 143, 154, 167, 180, 194, 210, 227, 245, 264, 286, 308, 333, 360, 388, 419, 453,
489, 528, 570, 616, 665]
```

Previously we calculated the brightness steps linear with `new_brightness = (new_brightness_step * 665);` With brightness steps from 0 to 1 with 0.1 step size