

Zephyr and application setup

This lab will use the [Zephyr](#) real-time operating system. Zephyr is an open-source project of the [Linux Foundation](#).

You will have two options for working on this lab: you can use your own computer, or you can use Telecom Paris lab computers.

Using your own computer

If you wish to do this lab on your own computer, it must be running a Unix-like system such as Linux or macOS. You may be able to do the lab in Microsoft Windows but do not expect help from the teachers (you're welcome to provide a step-by-step guide that would be included here).

The Zephyr web site contains [instructions to install Zephyr](#).

You will have to perform the following steps:

- Install the required packages on your computer (such as `cmake`).
- Decide where you will install Zephyr (for example `$HOME/zephyrproject`).
- Create a virtual Python 3 environment (for example in `$HOME/zephyrproject/venv`):

```
$ mkdir $HOME/zephyr
$ python3 -mvenv $HOME/zephyr/venv
```

- Activate this environment by sourcing the `activate` file:

```
$ source $HOME/zephyr/venv/bin/activate
```

- Install `west`, Zephyr toolbox utility:

```
$ pip install west
```

- Get Zephyr source code from git using `west` and update its dependent git repositories:

```
$ cd $HOME/zephyrproject
$ west init
$ west update
```

- Register Zephyr location for `cmake`:

```
$ west zephyr-export
```

- Install Zephyr Python requirements:

```
$ pip install -r zephyr/requirements.txt
```

- Install Zephyr SDK (see the [instructions to install the SDK](#)).
- Register Zephyr SDK location for `cmake` by running `setup.sh` found inside the SDK directory.

- Install udev rules as per the instructions page.

Keep in mind the following points:

- Zephyr and its SDK might take up to 25GB on your computer. Please ensure that you have enough
- Do not forget to activate the virtual Python environment in any terminal where you wish to use Zephyr.

Using Telecom Paris computer lab

It is possible to use Telecom Paris computer lab instead of your own computer. However, this will be much slower because many files will transit over the network.

Install Zephyr

You must set the environment variable `ZEPHYR_BASE` to `/comelec/softs/opt/zephyr-rtos/sources/zephyr`. Modify your shell init file so that this variable is always in your environment.

You must execute the `/comelec/softs/opt/zephyr-rtos/conf/cmake-register-zephyr.sh` script once. It will update your `cmake` configuration (in `$HOME/.cmake/packages`) with Zephyr location information.

You must activate the Python virtual environment in every shell where you want to use `west`, by executing `source /comelec/softs/opt/zephyr-rtos/sources/west-venv/bin/activate`.

You're all set. This is a relatively recent development of Zephyr, and we might update it from time to time, but that should be a transparent operation for you.

Zephyr application

Now that Zephyr has been installed, we will be able to write our first application. Rather than using a real board, we will start by using a PC emulator.

Anatomy of an application

Our Zephyr applications will be [freestanding](#). It means that we will keep our application code outside Zephyr source tree in order to only keep the application in our git repository.

A freestanding Zephyr application will make use of several directories:

- The virtual Python environment in which we have installed `west` and the required Python libraries.
- The Zephyr source code and its associated libraries. In order to use `west`, we will set the `ZEPHYR_BASE` environment variable to point at it.
- Your application source code, which you will create in your git repository.
- A build directory: Zephyr applications are built *out of tree*. It means that the build process will never mix the source code with the build artifacts. If your build directory is named `build`, you can remove it at any time in order to free up some disk space.

Creating an application

Your first Zephyr application

Create a directory in your git working directory. Create the following two files.

In `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr)
project(my_zephyr_app)

target_sources(app PRIVATE src/main.c)
```

In `src/main.c`:

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

We will now build the application for our target. In this case, we will generate code for a PC using an Intel CPU in bare-board mode and emulate the PC using the [QEMU](#) emulator.

Provided that `west` is in your `$PATH` and that `$ZEPHYR_BASE` is properly set if needed, you should be able to build your application for the board `qemu_x86`:

```
$ west build -b qemu_x86
```

(if you do not use `west`, you can use the equivalent `cmake -B build -DBOARD=qemu_x86 -GNinja && ninja -C build zephyr/zephyr.elf`)

This builds `zephyr/zephyr.elf` into the `build` directory. You should ignore this directory in your `.gitignore` as its content should never be added to the git repository.

You can now run the application. You do not need to give the board again as it is remembered.

```
$ west build -t run
-- west build: running target run
[0/1] To exit from QEMU enter: 'CTRL+a, x'[QEMU] CPU: qemu32,+nx,+pae
SeaBIOS (version zephyr-v1.0.0-0-g31d4e0e-dirty-20200714_234759-fv-az50-
zephyr)
Booting from ROM..
*** Booting Zephyr OS build v3.2.0-rc2-37-gd9f327fb8434 ***
Hello, world
```

(without `west: ninja -C build run`)

Congratulations: you just ran your first Zephyr application. You can quit qemu with `ctrl-a x` (control key + `a`, then `x`), or simply kill the qemu process with `ctrl-c`.

Adding a shell

Zephyr comes with an interactive shell. We want to add it to our project.

Run `west build -t menuconfig` (or `ninja -C build menuconfig`) and look for the `SHELL` option. You can find it under "Sub Systems and OS Services / Shell". Activate the shell by pressing the space key (the box will be ticked). You must also disable the `QEMU_ICOUNT` option found in "Board Options" as it won't work nice with the shell. You do not need to change any other option. Leave `menuconfig` by pressing `q` and saving the file.

You can now run the application again and use the interactive shell:

```
$ west build -t run
[0/1] To exit from QEMU enter: 'CTRL+a, x'[QEMU] CPU: qemu32,+nx,+pae
SeaBIOS (version zephyr-v1.0.0-0-g31d4e0e-dirty-20200714_234759-fv-az50-
zephyr)
Booting from ROM..
*** Booting Zephyr OS build v3.2.0-rc2-37-gd9f327fb8434 ***
Hello, world

uart:~$
```

You can now enter commands such as "help" at the prompt ("uart:~\$ " by default).

Making the change persistent

The changes we made with `menuconfig` are stored in `build/zephyr/.config` using the [Kconfig format](#). As soon as we remove the `build` directory, we will lose them.

Make the change persistent by creating a `prj.conf` file at the top-level of your application:

```
CONFIG_SHELL=y
CONFIG_SHELL_PROMPT_UART="(zephyr) "
```

We have also changed the prompt. Let's rebuild the application from scratch:

```
$ rm -rf build
$ west build -t run -b qemu_x86
[0/1] To exit from QEMU enter: 'CTRL+a, x'[QEMU] CPU: qemu32,+nx,+pae
SeaBIOS (version zephyr-v1.0.0-0-g31d4e0e-dirty-20200714_234759-fv-az50-
zephyr)
Booting from ROM..
*** Booting Zephyr OS build v3.2.0-rc2-37-gd9f327fb8434 ***
Hello, world

(Zephyr)
```

Do not forget to add `prj.conf` to git. Commit, then push.

Exploring the threads

Use the `kernel threads` shell command to examine the running threads. You should be able to identify the following ones:

- `idle` : the idle thread, which takes care of putting the system to sleep. Zephyr will run the threads with lower priority values first. The idle thread has the highest priority value, which means that it will run only when nothing else needs to run.
- `shell_uart` : the shell we are executing commands into. The shell runs into its own thread. It uses a very low priority task (with a high numerical value) as to not disturb the system it is interacting with.
- `sysworkq` : Zephyr default [workqueue](#). A workqueue is a thread on which jobs are sent to be executed serially, for example from an interrupt service routine (ISR). Note the negative priority: the workqueue is a very important thread.

Finding main

However, we don't see `main` in the list. Indeed, our main function exits as soon as it has printed.

Add an infinite loop at the end of `main()` . Why can't you use the shell? Use a proper way of making `main()` wait forever without consuming any resource. [Zephyr documentation](#) will be a very useful resource. Do not forget to include `zephyr/kernel.h` to use kernel services.

Find the priority of the `main` thread. Change it in the project configuration file and check that it has been properly set.

Changing board

Using ARM architecture

It is easy to switch to a new board in Zephyr. For example, instead of `qemu_x86` which emulates a PC, we will emulate a Cortex-M3 based system using `qemu_cortex_m3`.


Recompile and run your application for the `qemu_cortex_m3` board and check that everything still works as expected. If you run the command `file build/zephyr/zephyr.elf`, you should see that it is indeed an ARM application.

Using a real physical board

We will now use a [IoT node](#) board as we did in the [SE203 class](#). This board has a number of sensors and effectors that can be useful in the future.

You can see the list of supported boards using `west boards`. The board we are interested in is the `disco_l475_iot1`. Compile your application for this board.

Our boards are configured to use [Segger JLink](#) tools. We can flash the program and reset the board by using:

 If you use Telecom Paris computer lab, you must place the following directory at the beginning of your `PATH`: `/comelec/softs/opt/Segger/JLink_current/`. You should update your shell configuration file to do so each time you log in.

```
$ west flash --runner jlink --reset
```

(if you don't use `west`, you can use `ninja -C build flash` provided that you set the `BOARD_FLASH_RUNNER` variable to `jlink` at cmake time, see [this page](#) for more information)

You can now connect to the `/dev/ttyACM0` (for Linux, other systems do have equivalent names) and interact with the shell:

```
$ tio /dev/ttyACM0
(Zephyr)
```

Note how Zephyr was able to use the serial port on your board without you telling it explicitly. In `build/zephyr/zephyr.dts` will have access to the application [device tree](#).

The device tree describes the hardware and interacts with the right device drivers.

Note the following fragments:

```
/ {
    chosen {
        zephyr,console = &usart1;
        zephyr,shell-uart = &usart1;
    };

    soc {
        usart1: serial@40013800 {
            compatible = "st,stm32-usart", "st,stm32-uart";
            reg = < 0x40013800 0x400 >;
            clocks = < &rcc 0x60 0x4000 >;
            interrupts = < 0x25 0x0 >;
            status = "okay";
            pinctrl-0 = < &usart1_tx_pb6 &usart1_rx_pb7 >;
            pinctrl-names = "default";
            current-speed = < 0x1c200 >;
        };
    };
};
```

You can notice that the console (`printf()`) and the shell both opted to choose the device whose alias is `usart1` (the serial port connected to the JLink compatible chip on the board, accessible under `/dev/ttyACM0` or a similar name). Also, you can see how `usart1` is defined and require the use of the `st,stm32-usart` and `st,stm32-uart` device drivers.

Using RTT for the shell instead of the UART

Since we use the Segger JLink tools, we can request the use of Segger RTT instead of a serial port for the shell. By adding the following line to `prj.conf` you can indicate that the shell must use a RTT interface instead of a shell:

```
CONFIG_USE_SEGGER_RTT=y
CONFIG_SHELL_PROMPT_RTT="(Zephyr) "
CONFIG_SHELL_BACKEND_SERIAL=n
CONFIG_SHELL_BACKEND_RTT=y
```

You will get a warning about the now unused `CONFIG_SHELL_PROMPT_UART` : this is due to the fact that the serial backend for the shell is no longer used.

Flash and run the application on the board. You should notice that only the "Hello, world!" arrives through the serial port when you reset the board. Launch JLink RTT logger after flashing the board:

```
$ JLinkRTTLogger -device STM32L475VG -RTTChannel 1 -if SWD -Speed 4000
```

and connect to port 19021 using nc :

```
$ nc localhost 19021
```

You can now type command in the nc window and interact with the shell.

Now revert your changes as we will keep the shell on the serial port for now.

Some Zephyr concepts

In this part, you will learn how to use some Zephyr concept, such as the device drivers, the device tree and kernel services.

Using the device tree

You can find the consolidated device tree for your application in `build/zephyr/zephyr.dts` after the build phase. This device tree brings together your board-specific device tree, as well as the overlays defined specifically for your application or for additional components that you may declare to be using (such as an Arduino compatible shield).

Zephyr uses C (or C++) macros to access the device tree. The use of macros allows those to be resolved at compile time: a pre-processing phase compiles the device tree into a set of C headers whose content is manipulated through the macros.

An example: GPIOA

Here is a very simplified excerpt of the device tree for your application centered around `gpioa`:

```
/ {
  soc {
    pinctrl: pin-controller@48000000 {
      compatible = "st,stm32-pinctrl";

      gpioa: gpio@48000000 {
        compatible = "st,stm32-gpio";
        gpio-controller;
        #gpio-cells = < 0x2 >;
        reg = < 0x48000000 0x400 >;
        clocks = < &rcc 0x4c 0x1 >;
        phandle = < 0xb >;
      };
    };
  };
};
```

The device tree is made of nodes arranged in a tree fashion: a node may have children. Also, every node can have attributes.

In this example, `/` represents the root node which is the base of the tree. `soc` is a child of `/`, its path is `/soc`.

`pin-controller@48000000` is a child of `/soc`, its path is `/soc/pin-controller@48000000`. We also give this node a *label* `gpioa`. From now on, we will be able to reference this node by its label instead of using its full path. In the device tree itself, we will use `&gpioa` to reference the node.

Let's assume that we want to use `gpioa` in our application. We can reference the `gpioa` device by its label and ask Zephyr to find it at compile time:

```
#include <zephyr/device.h>

// Define GPIOA as the node whose label is gpioa in the device tree
#define GPIOA_NODE DT_NODELABEL(gpioa)

// Get the struct device pointer to gpioa at compile time
static const struct device * const gpioa_dev = DEVICE_DT_GET(GPIOA_NODE);
```

Now that you have a reference to `gpioa`, you can setup the led at run-time and make it blink:

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

int main() {
    if (!device_is_ready(gpioa_dev)) {
        return -ENODEV; // Device is not ready
    }
    // Configure the led as an output, initially inactive
    gpio_pin_configure(gpioa_dev, 5, GPIO_OUTPUT_INACTIVE);
    // Toggle the led every second
    for (;;) {
        gpio_pin_toggle(gpioa_dev, 5);
        k_sleep(K_SECONDS(1));
    }
    return 0;
}
```

Note that Zephyr peripherals are initialized during different boot phases. If somehow you have misconfigured a peripheral, it might not be ready when you start, at which case you should abort. This is what is done at the beginning of `main`.

Ensure that you can make the led blink. Easy, no? Yes, but you can do much better.

Commit, push.

Ensure that you understand everything you've used before going on.

Let's do less manual work

As you might have noted, we had to retrieve the `const struct device *` corresponding to `gpioa` and configure and toggle pin 5. Of course, we do not want to hardcode the port and the pin number in our code.

Finding the led

If you look at `build/zephyr/zephyr.dts`, you'll notice some interesting fragments:

```
/ {
    aliases {
        led0 = &green_led_1;
    };
    leds {
        compatible="gpio-leds";
        green_led_1: {
            gpios = < &gpioa 0x5 0x0 >;
        };
    };
};
```

The node `/aliases` is a special one: it defines global aliases on nodes that can then be found with macro `DT_ALIAS`, such as in `DT_ALIAS(led0)`.

This would in turn reference the `green_led_1` node label, which designated a node which contains an interesting `gpios` property `< &gpioa 0x5 0x0 >` with:

- a reference to a port (`&gpioa`)
- a pin number (`0x5`)
- no flags (`0x0`)

Note that the flags could have contained `GPIO_ACTIVE_LOW`, which would have meant that the `ACTIVE` state is the `LOW` state, and the `INACTIVE` state the `HIGH` state. By using these flags when initializing the port in your code, you can then use `ACTIVE` and `INACTIVE` (rather than `HIGH` and `LOW`) and get the proper behaviour.

Our goal will be to retrieve and use this gpio structure.

Retrieving the GPIO structure

Zephyr has a handy `struct gpio_dt_spec` structure which contains three fields: a port

device (such as `gpioa`), a pin number and the flags. Functions from the GPIO driver have an equivalent that takes such a struct as a parameter. Also, some macros allow the easy parsing of this structure:

```
// Designate the "led0" alias from the device tree
#define LED_NODE DT_ALIAS(led0)

// Get the GPIO (port, pin and flags) corresponding to the
// led0 node alias at compile time.
static const struct gpio_dt_spec led_gpio = GPIO_DT_SPEC_GET(LED_NODE,
gpio);

int main() {
    if (!device_is_ready(led_gpio.port)) {
        return -ENODEV;
    }
    // Note that we use INACTIVE, not LOW. On some boards,
    // the behaviour of the output may be reversed, but this
    // is not our concern. This info belongs to the device tree.
    gpio_pin_configure_dt(&led_gpio, GPIO_OUTPUT_INACTIVE);
    for (;;) {
        gpio_pin_toggle_dt(&led_gpio);
        k_sleep(K_SECONDS(1));
    }
    return 0;
}
```

Do it, check that it works. Commit push.

Note how the only reference to the hardware is now the `led0` name. But you can do better.

Change the led

What if you wanted to use another led as `led0`?

Create a file named `boards/disco_l475_iot1.overlay` at the top of your project. Now, you can redefine the alias:

```
/ {
    aliases {
        led0 = &green_led_2;
    };
};
```

Since this is a new file, you must reconfigure the project. Remove the `build` directory before rebuilding the project.

Check that the other green led blinks. Did you notice that you haven't changed one single line of C code?

Oh, enough played, you can remove this board specific files that you created and return the led to its original location.

The led device driver

In fact, Zephyr also has a led device driver. Each device can drive several leds. You can retrieve a `const struct device * const` directly from the `/leds` node and forget about even initializing the ports and so on.

Retrieve a pointer to the leds device and make the two leds at index 0 and 1 alternate every second (one is on, the other is off). Do not try to use `led_blink()` : this function is optional and is not implemented for the GPIO led driver.

If you get an error, maybe you should see if you have enabled everything you need in `prj.conf`. Is the led driver enabled?

Is it working now? Great!

However, note that we work with `const struct device *` all the time: the device drivers are not strongly typed. This is your responsibility to only call the right functions on the right device.

Adding a new led

If you look at the [board user manual](#) you'll notice two other leds: a yellow one and a blue one. Both are driven at the same time: writing a high level to the corresponding port turns the yellow led on and the blue led off, writing a low level turns the yellow led off and the blue led on.

We want to add this new led as a child to the `/leds` node. Create a file named `boards/disco_l475_iot1.overlay` at the top of your project. In it, you will be able to add things to enrich the device tree:

```
/ {
    leds {
        /* What you will add here will be added to the /leds node */
    };
};
```

Add a new led `led_3` in `leds` which corresponds to the yellow led. Use the board user manual to find on which port the led is connected.

You will have to remove the `build` directory in order to have the build system consider the new file.

Once this is done, modify your code so that:

- turn on all leds (at index 0, 1, and 2)
- wait 1s
- turn all leds off
- wait 1s
- start over

Does the yellow led light up at the same time as the green leds? Great!

Changing the meaning of the output

What if you consider that the led that should get turned on is the blue led instead of the yellow led? Change the flag in the overlay file that you have created to `GPIO_ACTIVE_LOW`. Run your program again.

Great, right?

If you have not done so, you **need** to read [the introduction to device trees](#) in Zephyr documentation.

Being more autonomous

You'll do more things on your own from now on.

Copy the example code from `$ZEPHYR_BASE/samples/basic/button` and run it on your board. You'll see that:

- The main program does some setup (gpio and interrupt) then copies the state of the switch `sw0` to the led `led0`.
- When the button is pressed, an interrupt is received and a message is printed.

Make sure you understand [how the interrupts work](#), and what the ISR does.

Removing the loop

In order to let the system sleep, you will receive an interrupt both when the button is pressed and when it is released. In the interrupt service routine, you'll copy its state to the led.

Commit and push your code when it's done.

Adding a pause

We now want a different behaviour: when the button is pressed, the led will light up. It will go down one second later, unless the button is pressed again in the meantime.

Of course you do not want to use `k_sleep()` in an interrupt callback routine. You will use a workqueue in which you'll submit delayable work. You can use the system workqueue, or you can start your own in a new thread.

Using sensors

We will now use various sensors that populate our board.

Using a sensor

In this part, we will use the [VL53L0X time-of-flight sensor](#) which can detect a distance using laser-ranging.

Using the board documentation (or the image of the sensor), locate the sensor on the board.

Referencing it as a sensor

Zephyr has a `sensor` class of devices. In its simplest mode of operation, the user:

- requests that a measure be initiated;
- then retrieves the result of the measure in channels (for example `SENSOR_CHAN_HUMIDITY` for a humidity sensor).

All values are returned as a pair of integers, avoiding the need for floating point calculations in many cases. For some sensors, only the first value will be filled. The [Zephyr documentation on sensors](#) describes those interfaces.

In order to know what channels are available for the vl53l0x driver, locate its source files in Zephyr source tree (starting from `$ZEPHYR_BASE`) and look for symbols starting with `SENSOR_CHAN_` .

Locate and use the device

In your source code, you might want to use the `DEVICE_DT_GET_ANY()` macro in order to locate any vl53l0x sensor on your board. In this case you know there is only one, so you don't have to look for its node label or its alias.

In a loop, print repeatedly (for example every tenth of seconds) the distance between the sensor and the obstacle above it (for example your hand) using the sensor interface.

Make the led blink at a rate depending on the distance

Using several threads and communicating through a [Message queue](#), make one of the green leds blink fast when your hand is near the board and much more slowly when it is far from the board.

Accelerometer

We will use the [LSM6DSL accelerometer](#) sensor located on our board using the I²C device driver.

The LSM6DSL documentation is accessible [here](#).

Check that you can address the LSM6DSL sensor

Check that you are able to read the `WHO_AM_I` register from the LSM6DSL sensor and that it returns the expected value.

Did you hardcode the address of the device and the I²C bus to use? If you did, you can do better: the `I2C_DT_SPEC_GET()` macro let you retrieve a `struct i2c_dt_spec` structure which contains the bus and the device address corresponding to a node located on a I²C bus.

Locate the node corresponding to the LSM6DSL accelerometer in your device tree (you should now remember that it is available in `build/zephyr/zephyr.dts`) and statically initialize a `struct i2c_dt_spec` for it.

Since you used a static initialization, do not forget to check with `device_is_ready()` that the I²C device is ready before using it for the first time.

You can use I²C functions ending in `_dt` to avoid explicitly giving the bus and the device address.

Configure the LSM6DSL accelerometer

Configure the LSM6DSL sensor so that you it performs measures at 1.6Hz. Check that reading the accelerometer registers give you a credible value when you turn the board on one side or another.

Configure the interrupt line connected to the microcontroller so that you know when a new measure has been done. When this happens, trigger a read of the accelerometer axes and display the value.

Note on displaying floating-point values

By default, Zephyr standard C library does not implement printing of floating-point values. If you want to use `%f` or `%g` formatters in functions of the `printf` family, you can switch to newlib, another libc implementation:

```
CONFIG_NEWLIB_LIBC=y
CONFIG_NEWLIB_LIBC_FLOAT_PRINTF=y
```

However, it is best not to use floating point if you don't need to.

Pitfalls and advices

Are some things not working? Or you think you can do better? Only if you are stuck or when you are done, check the following items:

- Do you use `i2c_dt_spec` for the I²C bus and address?
- Do you use `gpio_dt_spec` for the interrupt line?
- Did you configure the interrupt line as an input?
- Did you register the interrupt callback before the first measure?
- Did you use a work queue to perform the I²C data reading transaction in order not to do it in an interrupt callback?
- Did you reset the LSM6DSL to ensure that the interrupt line is low initially?
- In order to read the accelerometer data with one I²C transaction, did you keep the auto-increment of addresses enabled?
- Did you use the logging module in order to selectively display data?

Combining sensors

Now that you are able to read the accelerometer data, you should be able to read the gyroscope data as well. We will use both sensors to get an estimate of the various angles around X, Y, and Z axis.

Determine the partial board attitude

Using only the accelerometer, determine the board tilt angles (the angles compared from the "flat-on-the-table" position). You will have to use some maths there. Check that it looks consistent with the position of the board.

Read the gyroscope as well

Configure the gyroscope for continuous reading. The same interrupt line will be used to signal data from the accelerometer and data from the gyroscope. Use the `STATUS_REG` register to read only the relevant data and display it.

Use the gyroscope to determine the partial board attitude

Assuming that the board is flat on the table when the program starts, use the gyroscope data to compute the board tilt angles. You'll see that the angles derive slowly as each of the axes has a bias. This is inherent to gyroscopes.

Design a complementary filter between the accelerometer and the gyroscope

Store the accelerometer and gyroscope data in mutex-protected global variables. From a new thread, read those data regularly and compute the new angles using a complementary filter. Choose update frequencies high enough so that the variables are relatively up-to-date.

For example, you can compute the angles by taking 95% of the accelerometer-based computation and 5% of the gyroscope based computation (based on the previous angles).

Check if this gives you results that are both stable and reactive enough.

LED blinking

Make the green leds blink according to the absolute tilt angles. The leds will be steady when the board is flat, and will blink very fast when the board is tilted by 90°.

Writing a device driver

We will now write a Zephyr device-driver for the [DM163](#) led device driver from SiTI. Most of you are already familiar with it, as it is the one present on the [RGB matrix board used in SE203](#).

Our driver will be an *out of tree* device driver. It means that its source will be kept in the same directory as our application. This is often how drivers are initially built. When they become ready for public consumption, they can be integrated into Zephyr sources.

We will build our device driver so that it is compatible with the `led` driver that you have used already. It means that you will be able to use `led_on()` , `led_off()` and other led-related commands to drive the output of the DM163.

Remember that the DM163 drives only the columns of the led matrix. It is not aware of what a row is, it only knows columns of 8 RGB leds. In order to try our driver, we will have to turn one line transistor on.

As creating a device driver may seem a complicated task at first (it isn't), you will be guided step-by-step. Ensure you don't miss any step and ask questions early.

Structure

Create a new directory `dm163_example` to host your new application and driver code. Once we are done, the file structure will look like this:

```
dm163_example
├── boards
│   └── disco_l475_iot1.overlay
├── CMakeLists.txt
├── dm163_module
│   └── zephyr
│       ├── CMakeLists.txt
│       ├── dm163.c
│       ├── Kconfig
│       └── module.yml
├── dts
│   └── bindings
│       └── siti,dm163.yaml
├── prj.conf
├── src
│   └── main.c
```

(you do not need to create this hierarchy right now, you'll be guided when it is time to do that)

Here are some explanations of the structure:

- You are already familiar with `CMakeLists.txt`, `prj.conf` and `src/main.c`.
- The `dm163_module` directory contains the DM163 device driver.
- The `dts/bindings` directory contains the `siti,dm163.yaml` file which describes what is permitted and what is required in a node marked compatible with our device driver, whose name is, as you guessed, `siti,dm163` (from its maker SiTI and its model DM163).
- `boards/disco_l475_iot1.overlay` contains the device-tree part describing how the DM163 device is connected to our board.

From now on, all file names we indicate in the instructions will be relative to the top of the `dm163_example` folder.

What is in Zephyr's `struct device`?

A `struct device` in Zephyr is made of several fields, including:

- A `config` pointer, which points to read-only data describing the configuration gathered from the device tree and the `Kconfig` options. In our case, the

configuration will be a structure containing information about the pins to use in order to drive the DM163 chip.

- A `data` pointer, which points to data describing the current state of the device driver. In our case, it will contain the brightness and color information for the leds.
- A `api` pointer, which points to a read-only data structure compatible with the driver we want to be compatible with. Here we want to be compatible with the `led_driver` API, so our `api` field will be a `const struct led_driver_api * const`.

Most functions will receive a `struct device *` parameter which you can use to extract the `config` and `data` field containing the information you need. For example, to turn on one of the leds controlled by the DM163, you will modify the state of the led in the `data` structure and send the new led configuration using pins whose configuration is located in the `config` structure.

There exist other fields to store power management operations if the device can be put to sleep, as well as the initialization function to call to initialize the device.

Configuration

Our configuration structure for the DM163 driver will look like this:

```
struct dm163_config {  
    const struct gpio_dt_spec en;  
    const struct gpio_dt_spec gck;  
    const struct gpio_dt_spec lat;  
    const struct gpio_dt_spec rst;  
    const struct gpio_dt_spec selbk;  
    const struct gpio_dt_spec sin;  
};
```

Those names correspond to the name of the pins in the [DM163 documentation](#). We want to fill this structure from data coming from the device tree.

Create a `dtb/bindings/siti,dm163.yaml` file containing the following description:

```
description: SITI DM163 LED controller node  
  
compatible: "siti,dm163"  
  
properties:  
    sin-gpios:  
        type: phandle-array  
        required: true  
    selbk-gpios:  
        type: phandle-array  
        required: true  
    lat-gpios:  
        type: phandle-array  
        required: true  
    gck-gpios:  
        type: phandle-array  
        required: true  
    rst-gpios:  
        type: phandle-array  
        required: true  
    en-gpios:  
        type: phandle-array  
        required: false
```

This file contains information about what may be and must be present in a device-tree node marked as being compatible with the "siti,dm163" device driver. The `phandle-array` type corresponds to *a list of phandles and 32-bit cells (usually specifiers)*, for example `<&dma0 2>, <&dma0 3>` (according to [Zephyr documentation](#)). In our case, we will have only one phandle and its corresponding information, such as `<&gpio0 5 0>`.

Some pins names end with "_b" in the documentation. It means that they are active low (when pulled to ground) instead of active high (when pulled to VCC). In our case, we wil

note those pins as `GPIO_ACTIVE_LOW` in the device tree: setting them to `ACTIVE` will pull them to ground and setting them to `INACTIVE` will pull them to VCC.

Also note that we named the properties `-gpios`; even though we only have one GPIO, it is customary to end the name in `s` if the type allows you to enter several values.

Populate the `boards/disco_l475_iot1.overlay` with the following hardware information which corresponds to how the led matrix DM163 is connected to our microcontroller:

```
/ {
    dm163: dm163 {
        compatible = "siti,dm163";
        selbk-gpios = <&gpioC 5 0>;
        lat-gpios = <&gpioC 4 GPIO_ACTIVE_LOW>;
        rst-gpios = <&gpioC 3 GPIO_ACTIVE_LOW>;
        gck-gpios = <&gpioB 1 0>;
        sin-gpios = <&gpioA 4 0>;
    };
};
```

Here, we define a node `/dm163` with node label `dm163`. As we declare it to be compatible with the `siti,dm163` device driver (which we still have to write), the node content is checked against the bindings file we created just before (`dtb/bindings/siti,dm163.yaml`).

Note that `en-gpios` is absent. It was marked as `required: false` in the bindings description. This pin is sometimes connected directly to the ground, which means that the DM163 output is always enabled. If it is present in the device tree, we will have to configure it as an output and make it low, if it is absent we have nothing to do to enable the DM163 outputs.

Reading the device-tree into the struct `dm163_config`

Let's create `dm163_module/zephyr/dm163.c` with the following content:

```

// Enter the driver name so that when we initialize the (maybe numerous)
// DM163 peripherals we can designate them by index.
#define DT_DRV_COMPAT siti_dm163

#include <zephyr/device.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/drivers/led.h>
#include <zephyr/kernel.h>
#include <zephyr/logging/log.h>

LOG_MODULE_REGISTER(dm163, LOG_LEVEL_DBG);

struct dm163_config {
    const struct gpio_dt_spec en;
    const struct gpio_dt_spec gck;
    const struct gpio_dt_spec lat;
    const struct gpio_dt_spec rst;
    const struct gpio_dt_spec selbk;
    const struct gpio_dt_spec sin;
};

#define CONFIGURE_PIN(dt, flags)
\
do {
\
    if (!device_is_ready((dt)->port)) {
\
        LOG_ERR("device %s is not ready", (dt)->port->name);
\
        return -ENODEV;
\
    }
\
    gpio_pin_configure_dt(dt, flags);
\
} while (0)

static int dm163_init(const struct device *dev) {
    const struct dm163_config *config = dev->config;

    LOG_DBG("starting initialization of device %s", dev->name);

    // Disable DM163 outputs while configuring if this pin
    // is connected.
    if (config->en.port) {
        CONFIGURE_PIN(&config->en, GPIO_OUTPUT_INACTIVE);
    }
    // Configure all pins. Make reset active so that the DM163
    // initiates a reset. We want the clock (gck) and latch (lat)
    // to be inactive at start. selbk will select bank 1 by default.
    CONFIGURE_PIN(&config->rst, GPIO_OUTPUT_ACTIVE);
    CONFIGURE_PIN(&config->gck, GPIO_OUTPUT_INACTIVE);
    CONFIGURE_PIN(&config->lat, GPIO_OUTPUT_INACTIVE);
    CONFIGURE_PIN(&config->selbk, GPIO_OUTPUT_ACTIVE);
    CONFIGURE_PIN(&config->sin, GPIO_OUTPUT);
    k_usleep(1); // 100ns min

```



```

// Cancel reset by making it inactive.
gpio_pin_set_dt(&config->rst, 0);
// Enable the outputs if this pin is connected.
if (config->en.port) {
    gpio_pin_set_dt(&config->en, 1);
}
LOG_INF("device %s initialized", dev->name);
return 0;
}

// Macro to initialize the DM163 peripheral with index i
#define DM163_DEVICE(i)
\
\
\
/* Build a dm163_config for DM163 peripheral with index i, named
*/ \
/* dm163_config_/i/ (for example dm163_config_0 for the first peripheral)
*/ \
static const struct dm163_config dm163_config_##i = {
\
    .en = GPIO_DT_SPEC_GET_OR(DT_DRV_INST(i), en_gpios, {0}),
\
    .gck = GPIO_DT_SPEC_GET(DT_DRV_INST(i), gck_gpios),
\
    .lat = GPIO_DT_SPEC_GET(DT_DRV_INST(i), lat_gpios),
\
    .rst = GPIO_DT_SPEC_GET(DT_DRV_INST(i), rst_gpios),
\
    .selbk = GPIO_DT_SPEC_GET(DT_DRV_INST(i), selbk_gpios),
\
    .sin = GPIO_DT_SPEC_GET(DT_DRV_INST(i), sin_gpios),
\
};
\
\
DEVICE_DT_INST_DEFINE(i, &dm163_init, NULL, NULL, &dm163_config_##i,
\
    POST_KERNEL, CONFIG_LED_INIT_PRIORITY, NULL);

// Apply the DM163_DEVICE to all DM163 peripherals not marked "disabled"
// in the device tree and pass it the corresponding index.
DT_INST_FOREACH_STATUS_OKAY(DM163_DEVICE)

```

Since you can have several identical peripheral on a board, they will be identified by an index. The `DT*_INST_*` macros work with peripherals of type `DT_DRV_COMPAT` (which we defined at the top of the file). Here, the `DM163_DEVICE` macro will be applied to each peripheral marked compatible with the `siti_dm163` driver. Note how the `siti`, from the device tree (in "siti,dm163") as been replaced by a `_` for compatibility with the C language.

You have probably understood already what `DT_DRV_INST(i)` does: it designates the

node in the device tree containing instance `i` of driver `DT_DRV_COMPAT`. Also, `DEVICE_DT_INST_DEFINE()` creates an instance of the `struct device`. After those macros have executed, you end up with having statically created a `struct device` for every instance of the DM163 peripherals.

The other interesting parameters are:

- `POST_KERNEL`: the driver must be initialized after the kernel has started because we use the `k_sleep()` kernel service in `dm163_init()`.
- `CONFIG_LED_INIT_PRIORITY`: amongst all the drivers initialized in the `POST_KERNEL` phase, initialize this one at the same time as the other led drivers. We could have defined our own initialization priority but there is no need to do so.

Enabling the device driver

We need to create a `Kconfig` file in `dm163_module/zephyr` which adds a new `CONFIG_DM163_DRIVER` option. Let's make this option be selected by default as soon as there is a peripheral wanting to use the `siti,dm163` driver in the device tree. In this case, `CONFIG_DT_HAS_SITI_DM163_ENABLED` will be defined automatically by the device tree compiler. Also, we need to ensure that `CONFIG_GPIO` and `CONFIG_LED` are selected when we use this driver, as we use `gpios` and will make our driver compatible with the led driver. Here is the content of `dm163_module/zephyr/Kconfig`:

```
config DM163_DRIVER
    bool "Support for the DM163 led driver"
    default y
    depends on DT_HAS_SITI_DM163_ENABLED
    select GPIO
    select LED
```

There are two steps needed to complete our module. Create a file for `cmake` in `dm163_module/zephyr/CMakeLists.txt` indicating where the module sources are:

```
if(CONFIG_DM163_DRIVER)
    # Unused for now as we do not have any .h, but we might need to add
    # .h files later for this driver
    zephyr_include_directories(.)

    zephyr_library()
    zephyr_library_sources(dm163.c)
endif()
```

and a `dm163_module/zephyr/module.yml` which indicates where the directory with the `cmake` file and the `Kconfig` file are located relative to the module directory:

```
build:
  cmake: zephyr
  kconfig: zephyr/Kconfig
```

We must now indicate to the top-level cmake where our module can be found if we need it by appending the `dm163_module` to the list of Zephyr extra modules. Use the following content in `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.20.0)

list(APPEND ZEPHYR_EXTRA_MODULES
  ${CMAKE_CURRENT_SOURCE_DIR}/dm163_module
)

find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(dm163_example)

target_sources(app PRIVATE src/main.c)
```

Let's now configure our project in `prj.conf` and add some logging capabilities:

```
CONFIG_LOG=y
```

Our main program in `src/main.c` does not need to be long:

```
int main() {
    return 0;
}
```

It is now time to build our project with `west build -b disco_l475_iot1` and to flash it with `west flash --runner jlink --reset`. If everything went well, you should see the following on the USB serial port (for example `/dev/ttyACM0`):

```
*** Booting Zephyr OS build zephyr-v3.2.0-714-g7288df4c41d3 ***
[00:00:00.000,000] <dbg> dm163: dm163_init: starting initialization of device
dm163
[00:00:00.000,000] <inf> dm163: device dm163 initialized
```

Yeah, we have the beginning of a device driver. Let's continue.

Dynamic data

As we defined the `struct dm163_config` to hold read-only configuration information about a peripheral, we now need to define a `struct dm163_data` to hold dynamic information.

The DM163 has two data banks:

- Bank 0 contains 6 bits per channel (a channel is the red, green, or blue color for one RGB led).
- Bank 1 contains 8 bits per channel.

The PWM value presented on a given channel will be proportional to $(\text{bank0} \div 64) \times (\text{bank1} \div 256)$. We will use bank 0 to represent the brightness of the led and bank 1 to represent the color intensity.

Typically, all 24 channels (for 8 leds) will have the same brightness. It is useful to reduce the overall power.

This gives us the structure of our `struct dm163_data`:

```
#define NUM_LEDS 8
#define NUM_CHANNELS (NUM_LEDS * 3)

struct dm163_data {
    uint8_t brightness[NUM_CHANNELS];
    uint8_t channels[NUM_CHANNELS];
};
```

We can now extend our `DM163_DEVICE` macro so that every `struct device` also contains a `struct dm163_data` to represent the dynamic data of a given DM163 peripheral:

```
// Macro to initialize the DM163 peripheral with index i
#define DM163_DEVICE(i)
\
\
/* Build a dm163_config for DM163 peripheral with index i, named
*/ \
/* dm163_config_/i/ (for example dm163_config_0 for the first peripheral)
*/ \
static const struct dm163_config dm163_config_##i = {
\
    .en = GPIO_DT_SPEC_GET_OR(DT_DRV_INST(i), en_gpios, {0}),
\
    .gck = GPIO_DT_SPEC_GET(DT_DRV_INST(i), gck_gpios),
\
    .lat = GPIO_DT_SPEC_GET(DT_DRV_INST(i), lat_gpios),
\
    .rst = GPIO_DT_SPEC_GET(DT_DRV_INST(i), rst_gpios),
\
    .selbk = GPIO_DT_SPEC_GET(DT_DRV_INST(i), selbk_gpios),
\
    .sin = GPIO_DT_SPEC_GET(DT_DRV_INST(i), sin_gpios),
\
};
\
\
/* Build a new dm163_data_/i/ structure for dynamic data
*/ \
static struct dm163_data dm163_data_##i = {};
\
\
DEVICE_DT_INST_DEFINE(i, &dm163_init, NULL, &dm163_data_##i,
\
                    &dm163_config_##i, POST_KERNEL,
\
                    CONFIG_LED_INIT_PRIORITY, NULL);
```

Let's initialize our data by adding the following fragments in `dm163_init()` . Right after the definition of `config` , we'll define `data` and retrieve it from the `const struct device *dev` that we received:

```
struct dm163_data *data = dev->data;
```

Then right after we mark the `config->rst` pin as inactive and before we enable `config->en`, we'll initialize the data and send it to the DM163 peripheral:

```
memset(&data->brightness, 0x3f, sizeof(data->brightness));
memset(&data->channels, 0x00, sizeof(data->channels));
flush_brightness(dev);
flush_channels(dev);
```

Here we set the brightness of every channel to its maximum 0x3f, or 2^6-1 , and the color value to 0. Then we send all the brightness and channel information to the DM163: all leds are off.

Of course we'll have to write those `flush_brightness()` and `flush_channels()` functions.

Utility functions

Write the following functions.

Sending data to the DM163

```
static void pulse_data(const struct dm163_config *config, uint8_t data, int bits);
```

This function sends `bits` bits of `data`, MSB first. Sending a bit means settings its value on `config->sin`, and making `config->gck` active then inactive (clock pulse).

For example `pulse_data(config, data, 6)` will send bit 5 of data, then bit 4, ..., and end with bit 0. `pulse_data(config, data, 8)` will send bit 7 of data, then bit 6, ..., and end with bit 0.

Sending channel information

```
static void flush_channels(const struct device *dev) {
    const struct dm163_config *config = dev->config;
    struct dm163_data *data = dev->data;

    for (int i = NUM_CHANNELS - 1; i >= 0; i--)
        pulse_data(config, data->channels[i], 8);
    gpio_pin_set_dt(&config->lat, 1);
    gpio_pin_set_dt(&config->lat, 0);
}
```

This function sends all channel information (in reverse order) to bank 1 and then does a latch pulse by setting `config->lat` as active (ground, because this pin is active low) then inactive (VCC). We also assume that the current selected bank (on `config->sb`) is 1 when we enter this function.

The code of the function is given to show the standard way of working with the `config` and `data` members of the `struct device` structure. By storing them into correctly typed

variable (while they are of type `void *` in `struct device`), we can easily reference their fields, such as `config->lat` or `data->channels`.

Sending brightness information

```
static void flush_brightness(const struct device *dev);
```

This function sends the brightness information for all the channels (in reverse order) to bank 0. Bank 0 must be selected first, and bank 1 must be selected back at the end of the function so that when we return bank 1 is selected again. The rationale here is that we are more likely to frequently change colors rather than change the brightness of the leds.

Setting brightness for a led

```
static int dm163_set_brightness(const struct device *dev, uint32_t led,
uint8_t value);
```

This function receives a brightness between 0 and 100 (inclusive) in `value`. It must set it in `data->brightness` for the three channels of the corresponding `led` after converting the `value` to the [0-63] interval. Once the brightness has been set, `flush_brightness()` must be called for this to be reflected on the actual leds.

The return value is 0 if all goes well, `-EINVAL` if the led number is incorrect.

Turning a led on and off

```
static int dm163_on(const struct device *dev, uint32_t led);
static int dm163_off(const struct device *dev, uint32_t led);
```

Those functions respectively turn a led on (full white) and off. Once the channels have been updated, `flush_channels()` must be called for this to be reflected on the actual leds.

Being compatible with the led API

We now have enough functions to be able to pretend to be compatible with Zephyr led driver API. Before the `DM163_DEVICE()` macro, build a read-only `struct led_driver_api` named `dm163_api`:

```
static const struct led_driver_api dm163_api = {
    .on = dm163_on,
    .off = dm163_off,
    .set_brightness = dm163_set_brightness,
};
```

Here we define three functions that can be called by the LED api functions (respectively `led_on()`, `led_off()` and `led_set_brightness()`). We let other functions be undefined (NULL) for the time being.

Update the `DEFINE_DT_INST_DEFINE()` call to include this newly defined structure as the last argument:

```
DEVICE_DT_INST_DEFINE(i, &dm163_init, NULL, &dm163_data_##i,
\
                        &dm163_config_##i, POST_KERNEL,
\
                        CONFIG_LED_INIT_PRIORITY, &dm163_api);
```

Using the LED API and testing our code

We can now build a real main program:

```
#include <zephyr/device.h>
#include <zephyr/kernel.h>
#include <zephyr/drivers/led.h>

#define DM163_NODE DT_NODELABEL(dm163)
static const struct device *dm163_dev = DEVICE_DT_GET(DM163_NODE);

int main() {
    if (!device_is_ready(dm163_dev)) {
        return -ENODEV;
    }
    // Set brightness to 5% for all leds so that we don't become blind
    for (int i = 0; i < 8; i++)
        led_set_brightness(dm163_dev, i, 5);
    // Animate the leds
    for (;;) {
        for (int col = 0; col < 8; col++) {
            led_on(dm163_dev, col);
            k_sleep(K_MSEC(100));
            led_off(dm163_dev, col);
        }
    }
}
```

Execute the program. Does it work?

If it does, think about how strange this is: you haven't enabled any line transistor and yet, one line is enabled. How comes? Since `CONFIG_DM163_DRIVER`, which is implicit because you have declared one such peripheral in the device tree, implies `CONFIG_LED`, then the GPIO leds are configured. And one of those leds correspond to the same pin as one of the lines. You got lucky.

Using the shell

If you activate the `CONFIG_SHELL` and `CONFIG_LED_SHELL` options, you'll see a `led` menu in the shell that can be used to turn the leds on or off:

```
uart:~$ led on dm163 4
dm163: turning on LED 4
uart:~$ led off dm163 4
dm163: turning off LED 4
```

Driving the lines

We want to add the information about the 8 lines to the device tree. Let's invent a new node type to do this, we'll call it "rgb_matrix".

Add a `dtb/bindings/rgb_matrix.yaml` file:

```
description: RGB matrix

compatible: "rgb_matrix"

properties:
  rows-gpios:
    type: phandle-array
    required: true
```

Any node identified as compatible with "rgb_matrix" in the device tree will need to have a `row-gpios` property in which we will store the lines information.

Let's add such a node in our board overlay `boards/disco_l475_iot1.overlay` which now becomes:

```
/ {
  dm163: dm163 {
    compatible = "siti,dm163";
    selbk-gpios = <&gpiorc 5 0>;
    lat-gpios = <&gpiorc 4 GPIO_ACTIVE_LOW>;
    rst-gpios = <&gpiorc 3 GPIO_ACTIVE_LOW>;
    gck-gpios = <&gpiob 1 0>;
    sin-gpios = <&gpioa 4 0>;
  };

  rgb_matrix: rgb_matrix {
    compatible = "rgb_matrix";
    rows-gpios = <&gpiob 2 0>, <&gpioa 15 0>, <&gpioa 2 0>, <&gpioa 7 0>,
      <&gpioa 6 0>, <&gpioa 5 0>, <&gpiob 0 0>, <&gpioa 3 0>;
  };
};
```

Using `GPIO_DT_SPEC_GET_BY_IDX`, we can retrieve one item from the phandle-array by its index. Here is the new content of `main.c`:

```

#include <zephyr/device.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/drivers/led.h>
#include <zephyr/kernel.h>

#define DM163_NODE DT_NODELABEL(dm163)
static const struct device *dm163_dev = DEVICE_DT_GET(DM163_NODE);

#define RGB_MATRIX_NODE DT_NODELABEL(rgb_matrix)

BUILD_ASSERT(DT_PROP_LEN(RGB_MATRIX_NODE, rows_gpios) == 8);

static const struct gpio_dt_spec rows[] = {
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 0),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 1),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 2),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 3),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 4),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 5),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 6),
    GPIO_DT_SPEC_GET_BY_IDX(RGB_MATRIX_NODE, rows_gpios, 7),
};

int main() {
    if (!device_is_ready(dm163_dev)) {
        return -ENODEV;
    }
    for (int row = 0; row < 8; row++)
        gpio_pin_configure_dt(&rows[row], GPIO_OUTPUT_INACTIVE);
    // Set brightness to 5% for all leds so that we don't become blind
    for (int i = 0; i < 8; i++)
        led_set_brightness(dm163_dev, i, 5);
    // Animate the leds on every row and every column
    for (;;) {
        for (int row = 0; row < 8; row++) {
            gpio_pin_set_dt(&rows[row], 1);
            for (int col = 0; col < 8; col++) {
                led_on(dm163_dev, col);
                k_sleep(K_MSEC(30));
                led_off(dm163_dev, col);
            }
            gpio_pin_set_dt(&rows[row], 0);
        }
    }
}

```

Nice, isn't it? Notice the `BUILD_ASSERT()` macro which checks at compile time that the `rows-gpios` property contains exactly eight items.

Why have we created a new node type instead of adding a property to the "siti,dm163" node? Because this has nothing to do with the DM163 which can perfectly be used to drive a single line of led.

More functions

Some other functions need to be implemented so that the driver can be fully (color mode) and efficiently driven.

Set led color

```
int dm163_set_color(const struct device *dev, uint32_t led, uint8_t
num_colors,
                    const uint8_t *color);
```

This function sets a led color. `color` contains the red, green, and blue component, in order. If some are missing, you can assume them to be 0. Don't forget to send the new channels value to the chip after doing so. As always, an incorrect led number or too great the number of colors will return `-EINVAL` instead of 0.

This corresponds to the `set_color` entry in the `struct led_driver_api` definition.

Set channels in bulk

```
int dm163_write_channels(const struct device *dev, uint32_t start_channel,
                        uint32_t num_channels, const uint8_t *buf);
```

Write channels in bulk. Make sure you check the bounds. This corresponds to the `write_channels` entry in the `struct led_driver_api` definition.

Using the functions

You can now use the shell `led set_color dm163 4 0 0 255` to display a beautiful blue led. Of course, it might be easier to stop the animation first to see it, and activate only one line.

Driving the whole matrix

Create a new image structure made of $8 \times 3 \times 8$ channels representing the whole led matrix. Make a thread which waits on a semaphore and display the next line using `led_write_channels()` when the semaphore is signaled. Make a timer signal the

semaphore approximately every $1 \div (8 \times 60)$ second. Enjoy your beautiful image.

Prevent race conditions

There is a risk that some thread changes the brightness (and sends the brightness information to the DM163) while another thread is changing a channel color value (and sends this information to the DM163): both threads will try to talk to the DM163 at the same time.

Add a mutex to the `dm163_data` structure, and take the mutex before flushing the colors or the brightness to the DM163.

To go further

Add a function with the following prototype in `dm163_module/zephyr/dm163.h`:

```
void dm163_turn_off_row(const struct device *dev, const struct gpio_dt_spec row);
```

(with the missing includes, and protection against multiple inclusion)

This function will remember the given `row` in the `dm163_data` structure. The next time `flush_channels()` is sending data to the DM163, as soon as data for the first six leds have been sent, the given `row` GPIO will be turned off. Then data for the remaining two leds will be sent and `row` will be forgotten.

By using this function, you might be able to turn a row off soon enough so that it has time to turn off completely before new data is latched, and late enough to keep a maximum of brightness. For example, this code

```
gpio_pin_set_dt(&rows[row], 0);    // Turn row "row" off
led_write_channels(&dm163_dev, ...); // Send data for row "row+1"
gpio_pin_set_dt(&rows[row+1], 1);  // Turn row "row+1" on
```

would be replaced by

```
dm163_turn_off_row(&dm163_dev, rows[row]); // Remember the row to turn off
led_write_channels(&dm163_dev, ...);        // Send data for row "row+1".
As a side                                     // effect, row "row" will be
turned off                                     // when 6/8th of the data has
been sent.
gpio_pin_set_dt(&rows[row+1], 1);
```

and give a better visual effect.

Spirit level

Now that you can drive the led matrix and that you know how to compute the tilt levels on the X and Y axes, you should be able to build a virtual [spirit level](#). The display should reflect the way the board is tilted, any way you like (a ball, a liquid like behavior, etc.).