



# Building a Linux system for the STM32MP1: connecting an I2C sensor

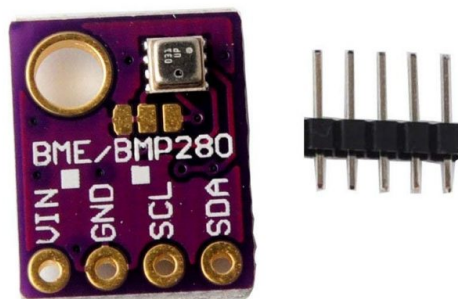
After showing how to build and run a minimal Linux system for the STM32MP157 Discovery board in a [previous blog post](#), we are now going to see how to connect an I2C sensor, adjust the Device Tree to enable the I2C bus and I2C device, and how to adjust the kernel configuration to enable the appropriate kernel driver.

List of articles in this series:

1. [Building a Linux system for the STM32MP1: basic system](#)
2. Building a Linux system for the STM32MP1: connecting an I2C sensor

## Choosing an I2C sensor

For this project, we wanted an I2C sensor that was at least capable of measuring the temperature, so we simply started by search *i2c temperature sensor* on Amazon. After a bit of research, we found that the [BME280](#)



[sensor](#) from Bosch was available on several inexpensive break-out boards, and it already had a [device driver](#) in the upstream Linux kernel. When choosing hardware, it is always important to check whether it is already supported or not in the upstream Linux kernel. Having a driver already integrated in the upstream Linux kernel has a number of advantages:

- The driver is readily available, you don't have to integrate a vendor-provided driver, with all the possible integration issues
- The driver has been reviewed by the Linux kernel maintainers, so you can be pretty confident of the code quality
- The driver is using standard Linux interfaces, and not some vendor-specific one
- The driver will be maintained in the long run by the kernel community, so you can continue to update your Linux kernel to benefit from security updates, bug fixes and new features

In addition, it also turns out that the BME280 sensor not only provides temperature sensing, but also pressure and humidity, which makes it even more interesting.

Among the numerous inexpensive BME280 break-out boards, we have chosen specifically [this one](#), but plenty of others are available. The following details will work with any other BME280-based break-out board.

## Connecting the I2C sensor

From a connectivity point of view, our I2C sensor is pretty simple: a VIN

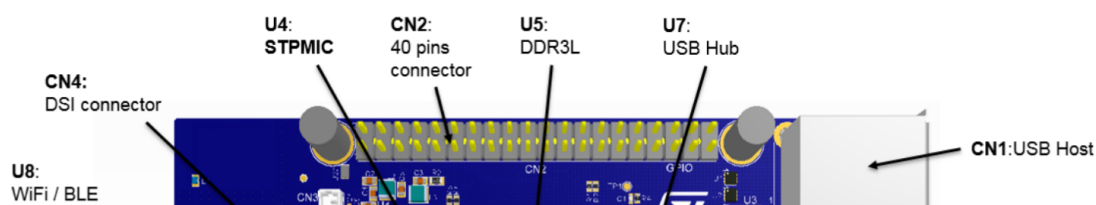
signal for power, a GND signal for ground, a SCL for the I2C clock and a SDA for the I2C data.

To understand how to connect this sensor to the Discovery board, we need to start with the [board user manual](#).

The Discovery board has two main expansion connectors: CN2 and the Arduino connectors.

## Connector CN2

Connector CN2 is a 40-pin male header on the front side of the board:

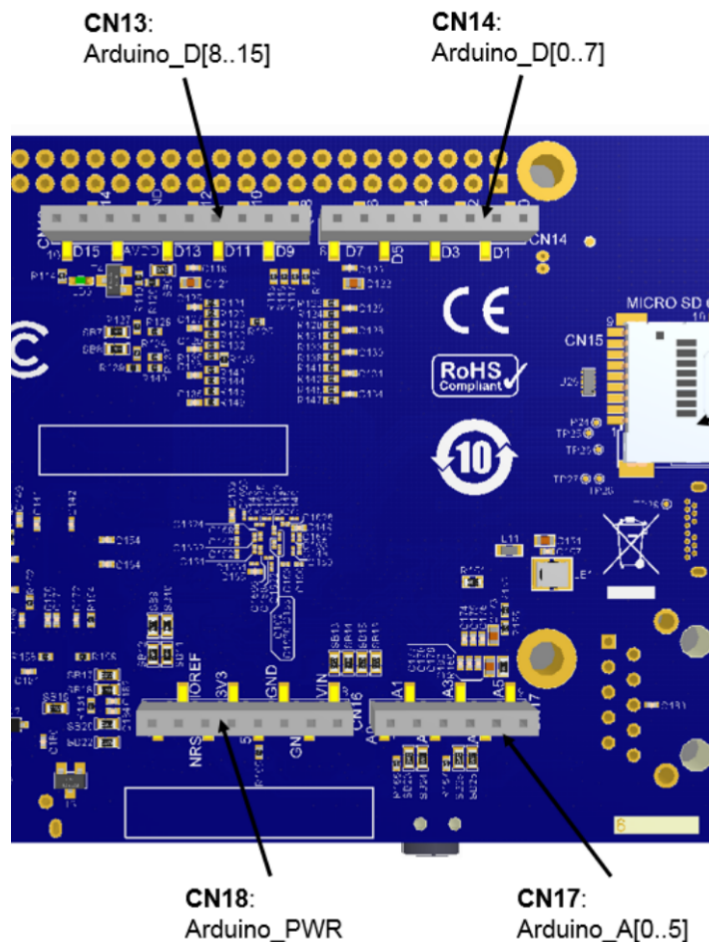


Section 7.17 of the board user manual documents the pin-out of this connector. There is one I2C bus available, through the I2C1\_SDA (pin 27) and I2C1\_SCL (pin 28) signals.

GND	-	25	26	PF3	GPIO7
I2C1_SDA	PF15	27	28	PD12	I2C1_SCL
GPIO5 / MCO2	PG2	29	30	-	GND

## Arduino connectors

Connectors CN13, CN14, CN17, CN18 are female connectors on the back side of the board. They are compatible in pin-out and form-factor with the Arduino connector:



Section 7.16 of the board user manual documents the pin-out for these connectors. There is one I2C bus available as well in CN13, through the I2C5\_SDA (pin 9) and I2C5\_SCL (pin 10) signals.

Connector	Pin name	Signal name	STM32 pin	Comment
CN13	1	ARD_D8	PG3	IO
	2	ARD_D9	PH6	TIM12_CH1
	3	ARD_D10	PE11	SPI4_NSS and TIM1_CH2
	4	ARD_D11	PE14	SPI4_MOSI and TIM1_CH4
	5	ARD_D12	PE13	SPI4_MISO
	6	ARD_D13	PE12	SPI4_SCK
	7	GND	-	GND
	8	VREFP	-	VREF+
	9	ARD_D14	PA12	I2C5_SDA
	10	ARD_D15	PA11	I2C5_SCL

## Choosing the connector

According to the block diagram in *Figure 3* of the board user manual, the I2C1 bus is already used to connect the touchscreen, the USB hub, the audio codec and the HDMI transceiver. However, I2C5 doesn't seem to be used at all. In addition, with the screen mounted on the Discovery board, the CN2 connector is beneath the screen, which makes it a bit more difficult to use than the Arduino connectors on the back side.

We will therefore use the I2C5 bus, through the Arduino connector CN13. Pin 9 will be used to connect the data signal of our sensor, and pin 10 will be used to connect the clock signal of our sensor.

## Finalizing the connectivity

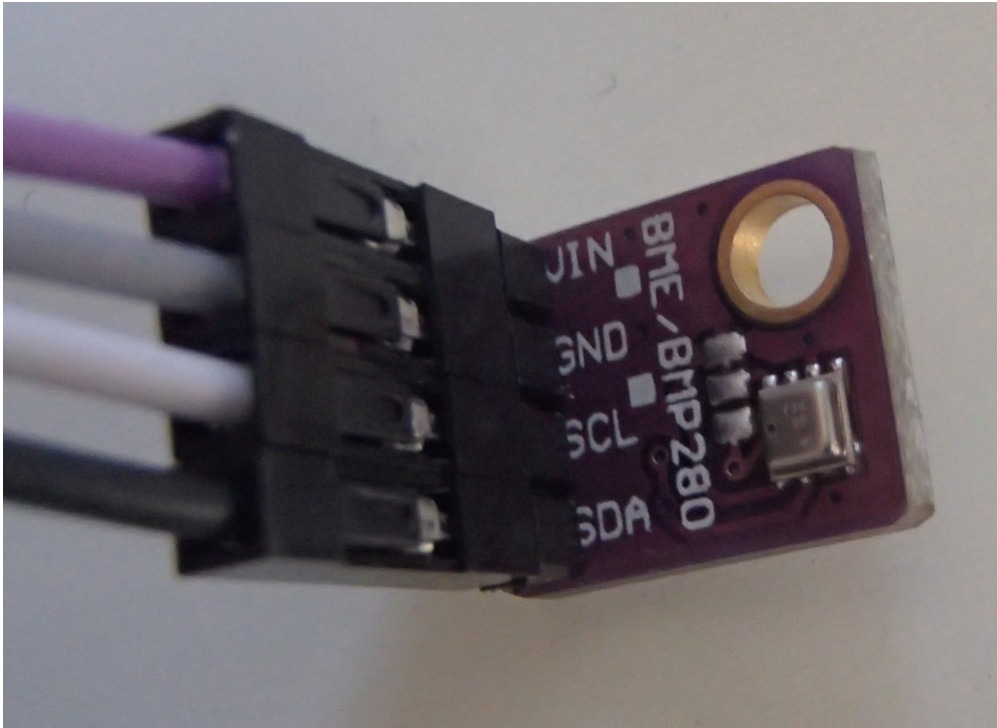
We still have to find out how to connect the VIN and GND pins.

According to the [BME280 datasheet](#), *VDDmain supply voltage range: 1.71V to 3.6V*. The Arduino connector CN16 provides either 3.3V or 5V, so we'll chose 3.3V (pin 4). And this connector also has multiple ground pins, among which we will chose pin 6.

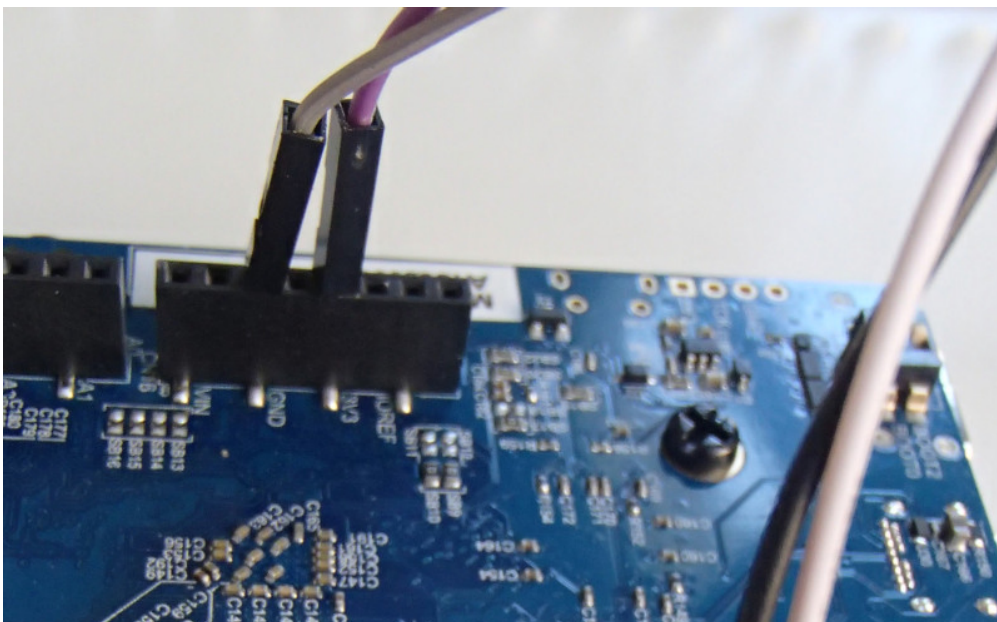
Overall, this gives us the following connections:

Sensor signal	Arduino connector	Pin
VIN	CN16	pin 4
GND	CN16	pin 6
SDA	CN13	pin 9
SCL	CN13	pin 10

Here are a few pictures of the setup. First, on the sensor side, we have a purple wire for VIN , a grey wire for GND , a white wire for SCL and a black wire for SDA :



On the board side, we can see the purple wire ( VIN ) going to pin 4 of CN16, the grey wire ( GND ) going to pin 6 of CN16, the white wire ( SCL ) going to pin 10 of CN13 and the black wire ( SDA ) going to pin 9 of CN13.





embedded architectures like memory-mapped buses, I2C, SPI and several others do not allow the operating system to ask the hardware: what peripherals are connected ? what are their characteristics ?. The operating system needs to know which devices are available and what their characteristics are. This is where the *Device Tree* comes into play: it is a data structure that describes in the form of a tree all the *devices* that we have in our hardware platform, so that the Linux kernel knows the topology of the hardware.

On ARM platforms, each particular board is described by its own Device Tree file. In our case, the STM32MP157 Discovery Kit 2 is described by the Device Tree file [arch/arm/boot/dts/stm32mp157c-dk2.dts](#) in the Linux kernel source code. This human-readable source file, with a `.dts` extension, is compiled during the Linux kernel build process into a machine-readable binary file, with a `.dtb` extension.

This `stm32mp157c-dk2.dts` describes the hardware of our Discovery Kit 2 platform. In fact, it only describes what is specific to the Discovery Kit 2: the display panel, the touchscreen, the WiFi and Bluetooth chip. Everything else is common with the Discovery Kit 1 platform, which is why the `stm32mp157c-dk2.dts` file includes the [arm/boot/dts/stm32mp157a-dk1.dts](#) file. Indeed, `stm32mp157a-dk1.dts` describes the hardware on the Discovery Kit 1, which is the same as the Discovery Kit 2, without the display, touchscreen and WiFi/Bluetooth chip.

In turn, the `stm32mp157a-dk1.dts` includes three other Device Tree files:

- [arm/boot/dts/stm32mp157c.dtsi](#), which describes all the devices inside the STM32MP157 system-on-chip. It will be used by all Device Tree files describing boards based on the STM32MP157 processor.
- [arm/boot/dts/stm32mp157c-m4-srm.dtsi](#), which describes the Cortex-M4 resources, which we are not going to discuss further for the moment



- [arm/boot/dts/stm32mp157cac-pinctrl.dtsi](#), which provides some pin-muxing related details, which are specific to the SoC package being used.

At this point, we won't give much more generic details about the Device Tree, as it's an entire topic on its own. For additional details, you could check the *Device Tree for Dummies* presentation for your author ([slides](#), [video](#)) or the [devicetree.org](#) web site.

## I2C controllers in the Device Tree

Zooming in to the topic of I2C, we can see that [arm/boot/dts/stm32mp157c.dtsi](#) describes 6 I2C controllers through six different nodes in the Device Tree:

- i2c1: i2c@40012000
- i2c2: i2c@40013000
- i2c3: i2c@40014000
- i2c4: i2c@5c002000
- i2c5: i2c@40015000
- i2c6: i2c@5c009000

This list of six I2C controllers nice matches the list of I2C controllers in the [STM32MP157 datasheet](#), and their base address in the memory map, section 2.5.2:

0x40012000 - 0x400123FF	1 KB	I2C1	<a href="#">I2C registers</a>
0x40013000 - 0x400133FF	1 KB	I2C2	<a href="#">I2C registers</a>
0x40014000 - 0x400143FF	1 KB	I2C3	<a href="#">I2C registers</a>
0x5c002000 - 0x5c0023FF	1 KB	I2C4	<a href="#">I2C registers</a>

0x40015000 - 0x400153FF	1 KB	I2C5	<a href="#">I2C registers</a>
0x5C009000 - 0x5C0093FF	1 KB	I2C6	<a href="#">I2C registers</a>

In the file `arm/boot/dts/stm32mp157a-dk1.dts`, we can see that the I2C1 bus is enabled, and that a cs42151 audio codec (I2C address 0x4a) and a sii9022 HDMI transceiver (I2C address 0x39) are connected to it:

```
&i2c1 {
    status = "okay";

    cs42151: cs42151@4a {
        compatible = "cirrus,cs42151";
        reg = <0x4a>;
    };

    hdmi-transmitter@39 {
        compatible = "sil,sii9022";
        reg = <0x39>;
    };
};
```

Also, on the I2C4 bus, we can see the USB-C controller (I2C address 0x28) and the PMIC (I2C address 0x33):

```
&i2c4 {
    status = "okay";

    typec: stusb1600@28 {
        compatible = "st,stusb1600";
        reg = <0x28>;
    };
};
```

```
        pmic: stpmic@33 {
            compatible = "st,stpmic1";
            reg = <0x33>;
        };
};
```

So, to enable our I2C5 bus, we will simply need to add:

```
&i2c5 {
    status = "okay";
    clock-frequency = <100000>;
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
};
```

to enable the bus. This piece of code adds the following Device Tree properties to the I2C5 Device Tree node:

- `status = "okay"` which simply tells the Linux kernel: *I really intend to use this device, so please enable whatever driver is needed to use this device*
- `clock-frequency = <100000>` tells Linux at which frequency we want to operate the I2C bus: in this case, 100 kHz
- The `pinctrl` properties configure the pin muxing, so that the pins are configured in the I2C function when the system is running (the `default` state) and into a different state to preserve power when the system is in suspend to RAM ( `sleep` state). Both `i2c5_pins_a` and `i2c5_pins_sleep_a` are already defined in [arch/arm/boot/dts/stm32mp157-pinctrl.dtsi](#).

For now, this doesn't describe any device on the bus, but should be

sufficient to have the bus enabled in Linux. The question now is how to make this modification in our Device Tree in the proper way ?

## Changing the Linux kernel source code

When Buildroot builds each package, it extracts its source code in `output/build/<package>-<version>`, so the source code of our Linux kernel has been extracted in `output/build/linux-custom/`. One could therefore be tempted to make his code changes directory in `output/build/linux-custom/`, but this has a number of major drawbacks:

1. `output/build/linux-custom/` is not under version control: it is not part of a Linux kernel Git repository, so you can't version control your changes, which is really not great
2. `output/build/linux-custom/` is a temporary folder: if you do a `make clean` in Buildroot, this folder will be entirely removed, and re-created during the next Buildroot build

So, while doing a change directly in `output/build/linux-custom/` is perfectly fine for quick/temporary changes, it's not a good option to make changes that will be permanent.

To do this in a proper way, we will use a feature of Buildroot called `pkg_OVERRIDE_SRCDIR`, which is documented in section *8.12.6 Using Buildroot during development* of the [Buildroot manual](#). This feature allows to tell Buildroot: for a given package, please don't download it from the usual location, but instead take the source code for a specific location on my system. This specific location will of course be under version control, and located outside of Buildroot, which allows to solve the two issues mentioned above.

So, let's get set this up for the Linux kernel source code:

1. Start in the parent folder of Buildroot, so that the Linux kernel source code ends up being side-by-side with Buildroot
2. Clone the official upstream Linux kernel repository. Even though we could directly clone the STMicro Linux kernel repository, your author always finds it nicer to have the `origin` Git remote set up to the official upstream Git repository.

```
git clone git://git.kernel.org/pub/scm/linux/kernel
/git/torvalds/linux.git
```

3. Move inside this Git repository

```
cd linux/
```

4. Add the STMicro Linux kernel repository as a remote:

```
git remote add stmicro https://github.com/STMicroelectronics
/linux.git
```

5. Fetch all the changes from the STMicro Linux kernel repository:

```
git fetch stmicro
```

6. Check out the version of the Linux kernel that Buildroot is using, i.e the tag `v4.19-stm32mp-r1.2`:

```
git checkout v4.19-stm32mp-r1.2
```

At this point, our `linux/` folder contains the *exact* same source code as what Buildroot has retrieved. It is time to make our Device Tree change by editing `arch/arm/boot/dts/stm32mp157c-dk2.dts` and at the end of it, add:

```
&i2c5 {
    status = "okay";
    clock-frequency = <100000>;
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;
};
```

Once done, we need to tell Buildroot to use our kernel source code, using the `pkg_OVERRIDE_SRCDIR` mechanism. To this, create a file called `local.mk`, in the top-level Buildroot source directory, which contains:

```
LINUX_OVERRIDE_SOURCE = $(TOPDIR)/../linux
```

This tells Buildroot to pick the Linux kernel source from `$(TOPDIR)/../linux`. We'll now ask Buildroot to wipe out its Linux kernel build, and do a build again:

```
$ make linux-dirclean
$ make
```

If you look closely at what Buildroot will do, it will do a *rsync* of the Linux kernel source code from your `linux/` Git repository to `output/build/linux-custom` in Buildroot, and then do the build. You can

check `output/build/linux-custom/arch/arm/boot/dts/stm32mp157c-dk2.dts` to make sure that your I2C5 change is there!

If that is the case, then reflash `output/images/sdcard.img` on your SD card, and run the new system on the board. It's now time to test the I2C bus!

## Testing the I2C bus

After booting the new system on your Discovery board and logging in as root, let's have a look at all I2C related devices:

```
# ls -l /sys/bus/i2c/devices/
total 0
lrwxrwxrwx 0-002a -> ../../../../devices/platform/soc/40012000.i2c/i2c-0/0-002a
lrwxrwxrwx 0-0038 -> ../../../../devices/platform/soc/40012000.i2c/i2c-0/0-0038
lrwxrwxrwx 0-0039 -> ../../../../devices/platform/soc/40012000.i2c/i2c-0/0-0039
lrwxrwxrwx 0-004a -> ../../../../devices/platform/soc/40012000.i2c/i2c-0/0-004a
lrwxrwxrwx 2-0028 -> ../../../../devices/platform/soc/5c002000.i2c/i2c-2/2-0028
lrwxrwxrwx 2-0033 -> ../../../../devices/platform/soc/5c002000.i2c/i2c-2/2-0033
lrwxrwxrwx i2c-0 -> ../../../../devices/platform/soc/40012000.i2c/i2c-0
lrwxrwxrwx i2c-1 -> ../../../../devices/platform/soc/40015000.i2c/i2c-1
lrwxrwxrwx i2c-2 -> ../../../../devices/platform/soc/5c002000.i2c/i2c-2
lrwxrwxrwx i2c-3 -> ../../../../devices/platform/soc/40012000.i2c/i2c-0/i2c-3
```

This folder is part of the `sysfs` filesystem, which is used by the Linux kernel to expose to user-space applications all sort of details about the hardware devices connected to the system. More specifically, in this folder, we have symbolic links for two types of devices:

- The I2C busses: `i2c-0`, `i2c-1`, `i2c-2` and `i2c-3`. It is worth mentioning that the bus numbers do not match the datasheet: they are simply numbered from 0 to N. However, the `i2c-0` symbolic link shows it's the I2C controller at base address `0x40012000`, so it's I2C1 in the datasheet, `i2c-1` is at base address

0x40015000 so it's I2C5 in the datasheet, and i2c-2 at base address 0x5c002000 is I2C4 in the datasheet. i2c-3 is special as it's not an I2C bus provided by the SoC itself, but the I2C bus provided by the HDMI transmitter to talk with the remote HDMI device (since this is unrelated to our discussion, we won't go into more details on this).

- The I2C devices: 0-002a, 0-0038, 0-0039, 0-004a, 2-0028, 2-0033. These entries have the form B-SSSS where B is the bus number and SSSS is the I2C address of the device. So you can see that for example 0-004a corresponds to the cs42151 audio codec we mentioned earlier.

In our case, we are interested by I2C5, which is known by Linux as i2c-1. We will use the i2cdetect utility, provided by Busybox, to probe the different devices on this bus:

```
# i2cdetect -y 1
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- 76 --
```

Interesting, we have a device at address 0x76 ! Try to disconnect either VIN, SDA or SCL of your I2C sensor, and repeat the command:

```
# i2cdetect -y 1
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                -- -- -- -- -- -- -- -- -- -- -- -- -- --
```



```
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- --
```

The device at 0x76 has disappeared, so it looks like our sensor is at I2C address 0x76. To confirm this, let's have a look at what the [BME280 datasheet](#) says about the I2C address of the device, in section *6.2 I2C Interface*:

The 7-bit device address is 111011x. The 6 MSB bits are fixed. The last bit is changeable by SDO value and can be changed during operation. Connecting SDO to GND results in slave address 1110110 (0x76); connection it to V<sub>DDIO</sub> results in slave address 1110111 (0x77), which is the same as

So, the I2C address is indeed 0x76 when the SDO pin of the sensor is connected to GND, which is probably what our BME280 break-out board is doing. It matches the address we have detected with `i2cdetect` !

Now, let's talk to our device. According to section *5.4 Register description* of the datasheet, there is a *Chip ID* register, at offset 0xD0 that is supposed to contain 0x60 :

#### 5.4.1 Register 0xD0 "id"

The "id" register contains the chip identification number `chip_id[7:0]`, which is 0x60. This number can be read as soon as the device finished the power-on-reset.

We can read this register using the `i2cget` command:

```
# i2cget -y 1 0x76 0xd0
```

```
0x60
```

Good, this matches the expected value according to the BME280 datasheet, so it seems like communication with our I2C device is working, let's move on to enabling the BME280 sensor driver.

## Enabling the sensor driver

As discussed earlier, this BME280 sensor already has a driver in the upstream Linux kernel, in the IIO subsystem. IIO stands for Industrial Input/Output, and this subsystem contains a lot of drivers for various ADCs, sensors and other types of measurement/acquisition devices. In order to use this driver for our BME280 device, we will essentially have to do two things:

1. Enable the driver in our Linux kernel configuration, so that the driver code gets built as part of our kernel image
2. Describe the BME280 device in our Device Tree so that the Linux kernel knows we have one such device, and how it is connected to the system

## Adjusting the kernel configuration

In the previous blog post, we explained that the Linux kernel configuration used to build the kernel for the STM32 Discovery board was located at `board/stmicroelectronics/stm32mp157-dk/linux.config`. Obviously, we are not going to edit this file manually: we need to run the standard Linux kernel configuration tools.

It turns out that Buildroot has convenient shortcuts to manipulate

the Linux kernel configuration. We can run the Linux kernel *menuconfig* configuration tool by running:

```
$ make linux-menuconfig
```

At this point, it is really important to not be confused by the fact that both Buildroot and the Linux kernel use the same configuration utility, but each have its own configuration. The Buildroot configuration describes your overall system (target architecture, which software components you want, which type of filesystem you want, etc.) while the Linux kernel configuration describes the kernel configuration itself (which drivers you want, which kernel features you need, etc.). So make sure to not confuse the *menuconfig* of Buildroot with the *menuconfig* of the Linux kernel!

Once you have run `make linux-menuconfig`, the *menuconfig* of the Linux kernel will show up. You will then enable the following option:

```
Device Drivers
+- Industrial I/O support
  +- Pressure sensors
    +- Bosch Sensortec BMP180/BMP280 pressure sensor I2C
    driver
```

Make sure to enable this option with a star `<*>` so that the driver is compiled inside the kernel image itself and not as a separate kernel module. You can then exit the *menuconfig* utility, and confirm that you want to save the configuration.

At this point, the Linux kernel configuration file in `output/build`

/linux-custom/.config has been changed. You can confirm it by running:

```
$ grep CONFIG_BMP280 output/build/linux-custom/.config
CONFIG_BMP280=y
CONFIG_BMP280_I2C=y
CONFIG_BMP280_SPI=y
```

However, as we explained earlier, the `output/build/linux-custom/` folder is temporary: it would be removed when doing a `Buildroot make clean`. We would like to permanently keep our Linux kernel configuration. Once again, Buildroot provides a nice shortcut to do this:

```
$ make linux-update-defconfig
```

After running this command, the kernel configuration file `board/stmicroelectronics/stm32mp157-dk/linux.config` has been updated, and this file is not temporary, and is under version control. If you run `git diff`, you can see the change on this file:

```
$ git diff
[...]
index 878a0c39f1..12f3e22647 100644
--- a/board/stmicroelectronics/stm32mp157-dk/linux.config
+++ b/board/stmicroelectronics/stm32mp157-dk/linux.config
@@ -169,6 +169,7 @@ CONFIG_STM32_LPTIMER_CNT=y
 CONFIG_STM32_DAC=y
 CONFIG_IIO_HRTIMER_TRIGGER=y
 CONFIG_IIO_STM32_LPTIMER_TRIGGER=y
```

```
+CONFIG_BMP280=y
CONFIG_PWM=y
CONFIG_PWM_STM32=y
CONFIG_PWM_STM32_LP=y
```

We're all set for the kernel configuration!

## Describing the BME280 in the Device Tree

We now need to tell the Linux kernel that we have a BME280 sensor and how it is connected to the system, which is done by adding more details into our Device Tree. We have already enabled the I2C5 bus, and we now need to describe one device connected to it: this gets done by creating a *child node* of the I2C controller node.

How do we know what to write in the Device Tree node describing the BME280 ? Using *Device Tree bindings*. Those *bindings* are specification documents that describe how a given device should be represented in the Device Tree: which properties are available, what are their possible values, etc. All Device Tree bindings supported by the Linux kernel are documented in [Documentation/devicetree/bindings](#) in the Linux kernel source code. For our BME280 device, the binding is at [Documentation/devicetree/bindings/iio/pressure/bmp085.txt](#).

This document tells us that we have one required property, the `compatible` property, with the range of possible values. Since we have a BME280 sensor, we'll use `bosch,bme280`. The other properties are *optional*, so we'll ignore them for now. However, what this binding does explicitly mention is the fact that a `reg` property is also mandatory, to tell the Linux kernel the I2C address of the device. This `reg` property is however visible in the example given in this binding

document.

So, we'll go back to our `linux/` directory outside of Buildroot, where we cloned the Linux kernel repository, and we'll adjust our Device Tree file `arch/arm/boot/dts/stm32mp157c-dk2.dts` so that it contains:

```
&i2c5 {
    status = "okay";
    clock-frequency = <100000>;
    pinctrl-names = "default", "sleep";
    pinctrl-0 = <&i2c5_pins_a>;
    pinctrl-1 = <&i2c5_pins_sleep_a>;

    pressure@76 {
        compatible = "bosch,bme280";
        reg = <0x76>;
    };
};
```

## Re-building the kernel

Let's now ask Buildroot to rebuild the Linux kernel, with our Device Tree change and kernel configuration change. Instead of rebuilding from scratch, we'll just ask Buildroot to restart the build of the Linux kernel, which will be much faster:

```
$ make linux-rebuild
```

As part of this, Buildroot will re-run `rsync` from our `linux/` kernel Git repository to `output/build/linux-custom/`, so that we really build the latest version of our code, which includes our Device Tree change.

However, this just rebuilds the Linux kernel, and not the complete SD card image, so also run:

```
$ make
```

To regenerate the SD card image, write it on your SD card, and boot your system.

## Testing the sensor

After booting the system, if we check `/sys/bus/i2c/devices`, a new entry has appeared:

```
lrwxrwxrwx 1-0076 -> ../../../../devices/platform/soc/40015000.i2c/i2c-1/1-0076
```

If we following this symbolic link, we can see a number of interesting information:

```
# ls -l /sys/bus/i2c/devices/1-0076/
total 0
lrwxrwxrwx driver -> ../../../../../../bus/i2c/drivers/bmp280
drwxr-xr-x iio:device2
-r--r--r-- modalias
-r--r--r-- name
lrwxrwxrwx of_node -> ../../../../../../firmware/devicetree/base/soc/i2c@40015000/pressure@76
drwxr-xr-x power
lrwxrwxrwx subsystem -> ../../../../../../bus/i2c
-rw-r--r-- uevent
```

Here we can see that this device is bound with the device driver named `bmp280`, and that its Device Tree node is `base/soc/i2c@40015000`

```
/pressure@76 .
```

Now, to actually use the sensor, we need to understand what is the user-space interface provided by IIO devices. The [kernel documentation](#) gives some hints:

*There are two ways for a user space application to interact with an IIO driver.*

- */sys/bus/iio/iio:deviceX/, this represents a hardware sensor and groups together the data channels of the same chip.*
- */dev/iio:deviceX, character device node interface used for buffered data transfer and for events information retrieval.*

So, we'll try to explore the `/sys/bus/iio/` option:

```
# ls -l /sys/bus/iio/devices/
total 0
lrwxrwxrwx   iio:device0 -> ../../../../devices/platform/soc/48003000.adc
/48003000.adc:adc@0/iio:device0
lrwxrwxrwx   iio:device1 -> ../../../../devices/platform/soc/48003000.adc
/48003000.adc:adc@100/iio:device1
lrwxrwxrwx   iio:device2 -> ../../../../devices/platform/soc/40015000.i2c/i2c-1
/1-0076/iio:device2
lrwxrwxrwx   iio:device3 -> ../../../../devices/platform/soc/48003000.adc
/48003000.adc:temp/iio:device3
lrwxrwxrwx   trigger0 -> ../../../../devices/platform/soc/40004000.timer/trigger0
```

Here we can see a number of IIO devices: our IIO device is `iio:device2`, as can be seen by looking at the target of the link. The other ones are IIO devices related to the ADC on the STM32 processor. Let's check what we have inside `/sys/bus/iio/devices/iio:device2/`:



```
# ls -l /sys/bus/iio/devices/iio\:device2/
total 0
-r--r--r--    dev
-rw-r--r--    in_humidityrelative_input
-rw-r--r--    in_humidityrelative_oversampling_ratio
-rw-r--r--    in_pressure_input
-rw-r--r--    in_pressure_oversampling_ratio
-r--r--r--    in_pressure_oversampling_ratio_available
-rw-r--r--    in_temp_input
-rw-r--r--    in_temp_oversampling_ratio
-r--r--r--    in_temp_oversampling_ratio_available
-r--r--r--    name
lrwxrwxrwx    of_node -> ../../../../../../firmware/devicetree/base/soc
/i2c@40015000/pressure@76
drwxr-xr-x    power
lrwxrwxrwx    subsystem -> ../../../../../../bus/iio
-rw-r--r--    uevent
```

This is becoming interesting! We have a number of files that we can read to get the humidity, pressure, and temperature:

```
# cat /sys/bus/iio/devices/iio\:device2
/in_humidityrelative_input
49147
# cat /sys/bus/iio/devices/iio\:device2/in_pressure_input
101.567167968
# cat /sys/bus/iio/devices/iio\:device2/in_temp_input
24380
```

Now, let's check the kernel documentation at [Documentation/ABI/testing/sysfs-bus-iio](#) to understand the units used in these files:

```
What:          /sys/bus/iio/devices/iio:deviceX/in_tempX_input
Description:   Scaled temperature measurement in milli degrees Celsius.

What:          /sys/bus/iio/devices/iio:deviceX/in_pressure_input
Description:   Scaled pressure measurement from channel Y, in kilopascal.
```

```
What:          /sys/bus/iio/devices/iio:deviceX/in_humidityrelative_input
Description:   Scaled humidity measurement in milli percent.
```

So here we are: we are able to read the data from our sensor, and the Linux kernel driver does all the conversion work to convert the raw values from the sensors into usable values in meaningful units.

## Turning our kernel change into a patch

Our Device Tree change is for now only located in our local Linux kernel Git repository: if another person builds our Buildroot configuration, he won't have access to this Linux kernel Git repository, which Buildroot knows about thanks to the `LINUX_OVERRIDE_SRCDIR` variable. So what we'll do now is to generate a Linux kernel patch that contains our Device Tree change, add it to Buildroot, and ask Buildroot to apply it when building the Linux kernel. Let's get started.

First, go in your Linux kernel Git repository in `linux/`, review your Device Tree change with `git diff`, and if everything is alright, make a commit out of it:

```
$ git commit -as -m "ARM: dts: add support for BME280 sensor
on STM32MP157 DK2"
```

Then, generate a patch out of this commit:

```
$ git format-patch HEAD^
```

This will create a file called `0001-ARM-dts-add-support-for-BME280-sensor-on-STM32MP157-.patch` that contains our Device Tree change.

Now, back in Buildroot in the `buildroot/` folder, create the `board/stmicroelectronics/stm32mp157-dk/patches/` folder and a sub-directory `board/stmicroelectronics/stm32mp157-dk/patches/linux`. Copy the patch into this folder, so that the file hierarchy looks like this:

```
$ tree board/stmicroelectronics/stm32mp157-dk/
board/stmicroelectronics/stm32mp157-dk/
├── genimage.cfg
├── linux.config
├── overlay
│   ├── boot
│   │   └── extlinux
│   │       └── extlinux.conf
├── patches
│   └── linux
│       └── 0001-ARM-dts-add-support-for-BME280-sensor-on-
STM32MP157-.patch
├── readme.txt
└── uboot-fragment.config
```

Now, run Buildroot's *menuconfig*:

```
$ make menuconfig
```

And in Build options, set global patch directories to the value

board/stmicroelectronics/stm32mp157-dk/patches/ . This tells Buildroot to apply patches located in this folder whenever building packages. This way, when the `linux` package will be built, our patch in `board/stmicroelectronics/stm32mp157-dk/patches/linux/` will be applied.

We can now remove the `local.mk` file to disable the `pkg_OVERRIDE_SRCDIR` mechanism, and ask Buildroot to rebuild the Linux kernel:

```
$ rm local.mk
$ make linux-dirclean
$ make
```

If you pay attention to the Linux kernel build process, you will see that during the *Patching* step, our Device Tree patch gets applied:

```
>>> linux custom Patching

Applying 0001-ARM-dts-add-support-for-BME280-sensor-on-
STM32MP157-.patch using patch:
patching file arch/arm/boot/dts/stm32mp157c-dk2.dts
```

You can of course reflash the SD card at the end of the build, and verify that everything still works as expected.

Let's save our Buildroot configuration change:

```
$ make savedefconfig
```

And commit our Buildroot changes:

```
$ git add board/stmicroelectronics/stm32mp157-dk/linux.config
$ git add board/stmicroelectronics/stm32mp157-dk/patches/
$ git add configs/stm32mp157_dk_defconfig
$ git commit -s -m "configs/stm32mp157_dk: enable support for
BME280 sensor"
```

We can now share our Buildroot change with others: they can build our improved system which has support for the BME280 sensor.

## Conclusion

In this article, we have learned a lot of things:

- How to connect an I2C sensor to the Discovery board
- What is the Device Tree, and how it is used to describe devices
- How to use Buildroot's `pkg_OVERRIDE_SRCDIR` mechanism
- How to enable the I2C bus in the Device Tree and test its operation using `i2cdetect` and `i2cget`
- How to change the Linux kernel configuration to enable a new driver
- How to interact using `sysfs` with a sensor supported by the IIO subsystem
- How to generate a Linux kernel patch, and add it into Buildroot

In our next article, we'll look at adding support for the Qt5 graphical library into our system, as a preparation to developing a Qt5 application that will display our sensor measurements on the Discovery board screen.



**Author: Thomas Petazzoni**

Thomas Petazzoni is CTO and embedded Linux and kernel engineer at Bootlin. He is a lead developer of [Buildroot](#) and also a contributor to the Linux kernel. [More details...](#) [View all posts by Thomas Petazzoni](#)

---



Thomas Petazzoni / May 16, 2019 / Technical /

[Bootlin](#) / [Privacy Policy](#) / Proudly powered by [WordPress](#)