*TMP*100 *Device Driver Review*

*TMP*100 temperature sensor driver here, is written by making minimal changes to the *TMP*112 temperature sensor driver. The differences between *TMP*112 and *TMP*100 temperature sensors are:

- *Extended mode of operation: TMP*112 offers an increased 13-bit resolution in the extended mode as against the normal 12-bit mode in *TMP*100.
- *Configuration register: TMP*112 offers a 16-bit (2-bytes) size configuration register while the *TMP*100 offers an 8-bit configuration register.
- *Conversion rate: TMP*112 allows the user to specify the conversion rate (rate at which the ADC converts the temperature readings) whereas *TMP*100 does not provide the option.
- *Accuracy: TMP*112 is the most accurate sensor in comparison with *TMP*100 and *TMP*102 (similar to *TMP*112 with lesser accuracy).

Driver Kconfig

```
# earlier: config TMP100 #
menuconfig TMP100
        bool "TMP100 Temperature Sensor"
        depends on I2C
        help
                Enable drivers for TMP100 temperature sensor
```

menuconfig is a terminal-based interactive configuration interfaces available for exploring the available Kconfig options and making temporary changes. **bool** helps choose whether to compile the feature directly to the kernel or disable the feature totally. The depends is the way in which mutual requirements are satisfied in the Kconfig language.

Driver CMakeLists.txt

```
# Earlier
# zephyr_library()
# zephyr_library_sources(tmp100.c)

zephyr_include_directories(.)
zephyr_sources_ifdef(CONFIG_TMP100 tmp100.c)
```

zephyr_include_directories() is used to let Zephyr find the header file during builds. Any data that is accessed only when the feature is

enabled should be conditionally included by using #ifdef. Here we check if the *CONFIG_TMP*100 Kconfig option has been set and then conditionally include the driver source code.

*Driver source file tmp*100.*c*

```
#define DT_DRV_COMPAT ti_tmp100

#include <stdint.h>
#include <errno.h>
#include <device.h>
#include <drivers/i2c.h>
#include <sys/byteorder.h>
#include <sys/util.h>
#include <kernel.h>
#include <drivers/sensor.h>
#include <sys/__assert.h>
#include <logging/log.h>

LOG_MODULE_REGISTER(TMP100, CONFIG_TMP100_LOG_LEVEL);

#define TMP100_I2C_ADDRESS DT_INST_REG_ADDR(0)

#define TMP100_REG_TEMPERATURE 0x00
#define TMP100_REG_CONFIG 0x01
#define TMP100_TEMP_SCALE 62500

struct tmp100_data {
    const struct device *i2c;
    int16_t sample;
};

static int tmp100_reg_read(struct tmp100_data *drv_data,
   uint8_t reg, uint16_t *val)
{
    if (i2c_burst_read(drv_data->i2c, TMP100_I2C_ADDRESS,
                reg, (uint8_t *) val, 2) < 0) {
        return -EIO;
    }
*val = sys_be16_to_cpu(*val);
return 0;
}

static int tmp100_reg_write(struct tmp100_data *drv_data,
                uint8_t reg, uint16_t val)
```

```
{
    uint16_t val_be = sys_cpu_to_be16(val);
    return i2c_burst_write(drv_data->i2c, TMP100_I2C_ADDRESS,
                   reg, (uint8_t *)&val_be, 2);
}

static int tmp100_reg_update(struct tmp100_data *drv_data, uint8_t reg,
                 uint16_t mask, uint16_t val)
{
    uint16_t old_val = 0U;
    uint16_t new_val;

    if (tmp100_reg_read(drv_data, reg, &old_val) < 0) {
        return -EIO;
    }

    new_val = old_val & ~mask;
    new_val |= val & mask;

    return tmp100_reg_write(drv_data, reg, new_val);
}
```

As discussed in the *TMP*112 driver section, above three functions are used to read, write, and update the registers of the sensor. The temperature register is a read-only register where as the configuration register is a read/write permitted register.

```
static int tmp100_attr_set(const struct device *dev,
   enum sensor_channel chan,
   enum sensor_attribute attr,
   const struct sensor_value *val)
{
    struct tmp100_data *drv_data = dev->data;
int64_t value;

if (chan != SENSOR_CHAN_AMBIENT_TEMP) {
return -ENOTSUP;
}

if(attr == SENSOR_ATTR_FULL_SCALE) {
if (val->val1 == 128) {
value = 0x00;
}
else {
return -ENOTSUP;
```

```
}
    }

    if (tmp100_reg_update(drv_data, TMP100_REG_CONFIG,
                1, value) < 0) {
            LOG_DBG("Failed to set attribute!");
            return -EIO;
        }
    return 0;
}
```

This function is written to set the full-scale reading attribute to the sensor.

```
static int tmp100_sample_fetch(const struct device *dev,
       enum sensor_channel chan)
{
struct tmp100_data *drv_data = dev->data;
uint16_t val;
    __ASSERT_NO_MSG(chan == SENSOR_CHAN_ALL || chan == SENSOR_CHAN_AMBIENT_TEMP);
    if (tmp100_reg_read(drv_data, TMP100_REG_TEMPERATURE, &val) < 0) {
        return -EIO;
    }

drv_data->sample = arithmetic_shift_right((int16_t)val, 4);
return 0;
}

static int tmp100_channel_get(const struct device *dev,
      enum sensor_channel chan,
      struct sensor_value *val)
{
struct tmp100_data *drv_data = dev->data;
    int32_t uval;

if (chan != SENSOR_CHAN_AMBIENT_TEMP) {
return -ENOTSUP;
}

    uval = (int32_t)drv_data->sample * TMP100_TEMP_SCALE;
    val->val1 = uval / 1000000;
    val->val2 = uval % 1000000;

return 0;
}
```

The above two functions are written to fetch the sample from the internal register of the sensor by arithmetically right-shifting by 4 bits of the 2-bytes worth sample data to get a 12-bit resolution.

The fetched sampled data has to be converted to readable form. This is performed in the *tmp*100*_channel_get*() function.

```
static const struct sensor_driver_api tmp100_driver_api = {
    .attr_set = tmp100_attr_set,
.sample_fetch = tmp100_sample_fetch,
.channel_get = tmp100_channel_get,
};

int tmp100_init(const struct device *dev)
{
struct tmp100_data *drv_data = dev->data;
drv_data->i2c = device_get_binding(DT_INST_BUS_LABEL(0));
    if (drv_data->i2c == NULL) {
        LOG_DBG("Failed to get pointer to %s device!",
                DT_INST_BUS_LABEL(0));
        return -EINVAL;
    }
return 0;
}

static struct tmp100_data tmp100_driver;
DEVICE_DT_INST_DEFINE(0, tmp100_init, NULL, &tmp100_driver, NULL, POST_KERNEL, CONFIG_SENSOR_INIT_PRIORITY
```

A structure is defined which holds all the API functions to be utilized in he driver code.

The **init** function is written to obtain the pointer to the device.

device_get_binding() returns the pointer to device structure by taking in the name of the device to be searched. **DT_INST_BUS_LABEL(inst)** returns the label property of the instance's bus controller which is passed to the **device_get_binding()** and a function called **LOG_DBG()** which is used to write DEBUG level messages to log.

Details about **DEVICE_DT_INST_DEFINE()** is explained in the *TMP*112 driver section.

*TMP*100 *Application CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.13.1)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(tmp100)

FILE(GLOB app_sources src/*.c)
target_sources(app PRIVATE ${app_sources})
# Below line is new addition #
add_subdirectory(../drivers tmp100)
```

Here, we add a subdirectory to include the driver.

*TMP*100 *Application prj.conf*

```
CONFIG_STDOUT_CONSOLE=y
CONFIG_I2C=y
CONFIG_SENSOR=y
CONFIG_TMP100=y
CONFIG_ASSERT=y
```

Above are the configuration options to enable the standard output to the console, *I2C* driver, sensor driver, custom *TMP*100 driver, and the __ASSERT() macro respectively.

*TMP*100 *Application sample.yaml*

```
sample:
  name: TMP100 Sensor Sample
tests:
  sample.sensor.tmp100:
    harness: console
    tags: sensors
    depends_on: i2c
    filter: dt_compat_enabled("ti,tmp100")
    harness_config:
        type: one_line
        regex:
            - "temp is (.*)"
        fixture: fixture_i2c_tmp100
```

Key-value pairs described in the YAML files gives the description of the contents of the nodes. From the file above, we can observe that some properties are what the sensor depends on, what is the regular expression, it's compatibility, etc.

*TMP*100 *Overlay file*

```
&i2c0 {
    compatible = "nordic,nrf-twi";
    status = "okay";
    sda-pin = <26>;
    scl-pin = <27>;
tmp100@48 {
compatible = "ti,tmp100";
reg = <0x48>;
        status = "okay";
label = "TMP100";
};
};
```

Overlay files are used to append the pre-existing devicetree structure with any additional settings. Here, we are defining a new node for the TMP100 sensor mentioning it's I2C address specific to the particular board. The I2C peripheral and it's node address is present in the vendor-provided devicetree source file. The driver uses the information in the overlay file to configure the hardware. The board by default has no sensor attached on it's I2C bus. Overlay file has to contain the node and its properties as an add-on

*TMP*100 *Application Kconfig*

```
menu "Zephyr"
source "Kconfig.zephyr"
endmenu

module = TMP100
module-str = TMP100
source "subsys/logging/Kconfig.template.log_config"
# The below line is the important addition #
rsource "../drivers/Kconfig"
```

rsource here means relative source to be chosen which is the driver folder's Kconfig file.

*TMP*100 *Application source code*

```
#include <zephyr.h>
#include <device.h>
#include <drivers/sensor.h>
#include <sys/printk.h>
```

```
#include <sys/__assert.h>

static void do_main(const struct device *dev)
{
    int ret;
    struct sensor_value temp_value;
    struct sensor_value attr;

    attr.val1 = 128;
attr.val2 = 0;

    ret = sensor_attr_set(dev, SENSOR_CHAN_AMBIENT_TEMP, SENSOR_ATTR_FULL_SCALE, &attr);

    if(ret)
    {
        printk("sensor_attr_set failed ret %d\n", ret);
        return;
    }

    while(1) {
        ret = sensor_sample_fetch(dev);
        if (ret) {
            printk("sensor_sample_fetch failed ret %d\n", ret);
            return;
        }

        ret = sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &temp_value);
        if (ret) {
            printk("sensor_channel_get failed ret %d\n", ret);
            return;
        }

        printk("temp is %d\n", temp_value.val1);

        k_sleep(K_MSEC(1000));
    }
}

void main(void)
{
const struct device *dev = DEVICE_DT_GET_ANY(ti_tmp100);

    __ASSERT(dev != NULL, "Failed to get device binding");
    __ASSERT(device_is_ready(dev), "Device %s is not ready", dev->name);
```

```
printk("device is %p, name is %s\n", dev, dev->name);

do_main(dev);
}
```

The working of the application code of *TMP*100 is exactly the same as that of the application code for *TMP*112.

The final output of the temperature sensor can be seen after we run:

```
west build -p auto -b nrf52840dk_nrf52811 tmp100
west flash --erase
minicom -D /dev/ttyACM* -b 115200
```

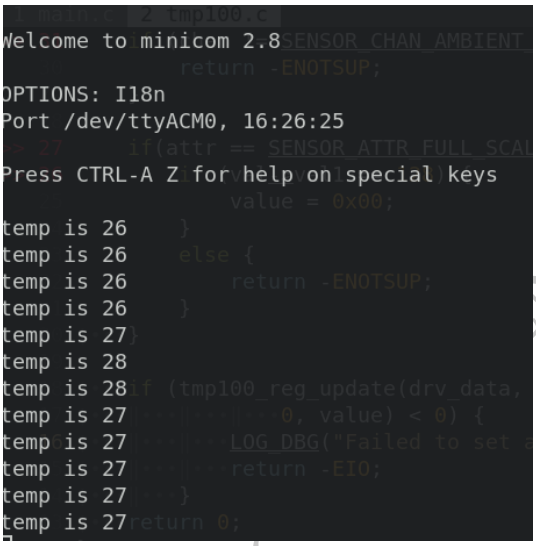

Figure 19: TMP100 Output