EN 173 Lab Guides

Lab 6

Version 2025, 2025-01-20

Getting started with analog input

In this exercise you will read an analog input (created with a potentiometer). The value received by the microcontroller will be sent using the logger to the host computer via USB and viewed using a terminal program.

Unlike general digital input and output, only certain pins can be mapped to the ADC on the nRF52840, but which pins are mapped to particular channels is determined by the user. The available analog pins are given symbolic names, specified in Table 1.

Table 1. Analog input pin names on the nRF52840.

GPIO	Analog input name	Arduino label
P0.02	AIN0	
P0.03	AIN1	A0
P0.04	AIN2	A1
P0.05	AIN3	
P0.28	AIN4	A2
P0.29	AIN5	A3
P0.30	AIN6	A4
P0.31	AIN7	A5
VDD	VDD	

1. Create a new application.
2. Edit `prj.conf` to enable both logging and analog-to-digital conversion.

   Program 1. The ADC and logging modules must be enabled in `prj.conf`.

   ```
   CONFIG_LOG=y
   CONFIG_ADC=y
   ```

3. Generate a build configuration and create an overlay.
4. Edit the overlay file.

   Program 2. An ADC channel is specified and made available in the overlay.

   ```
   /{
     zephyr,user { ①
       io-channels = <&adc 0>; ②
     };
   };

   &adc {
     #address-cells = <1>;
     #size-cells = <0>;
   ```

```
    status = "okay";
    channel@0 {
      reg = <0>;
      zephyr,reference = "ADC_REF_INTERNAL"; ③
      zephyr,gain = "ADC_GAIN_1_6"; ④
      zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
      zephyr,input-positive = <NRF_SAADC_AIN0>; ⑤
      zephyr,resolution = <12>; ⑥
    };
  };
```

① The `zephyr,user` node of the devicetree is a special node that does not require a corresponding binding file.

② Channel 0 of the ADC is connected to the `io-channels` property.

③ Uses a 0.6 V reference that is internal to the ADC.

④ Sets the gain to 1/6 so the voltage on the input pin is scaled to not exceed the reference voltage. There are additional limits on an input pin: its voltage should never fall below ground or exceed the board's VDD (nominally 3.0 V).

⑤ If using the nRF52840, this connects P0.02 to this channel of the ADC. See Table 1.

⑥ Use the full 12-bit resolution of the ADC. The analog-to-digital converter (ADC) on the nRF52840 has a maximum resolution (in normal operating mode) of 12 bits.

5. Enter Program 3 into `main.c`.

Program 3. Read an analog input and print its value in both raw and millivolt format.

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/adc.h>
#include <zephyr/logging/log.h>

LOG_MODULE_REGISTER(Lab6_Program1, LOG_LEVEL_DBG);

const struct adc_dt_spec adc_channel = ADC_DT_SPEC_GET(DT_PATH(zephyr_user)); ①

int main(void) {
    int err;
    uint16_t buf; ②
    int val_mV; ③
    struct adc_sequence pot_reading = { ④
        .buffer = &buf, ⑤
        .buffer_size = sizeof(buf) ⑥
    };

    if (!adc_is_ready_dt(&adc_channel)) {
        LOG_ERR("ADC controller device is not ready");
        return -1;
    }
    err = adc_channel_setup_dt(&adc_channel); ⑦
    if (err < 0) {
        LOG_ERR("Could not setup channel.");
        return -1;
    }
    err = adc_sequence_init_dt(&adc_channel, &pot_reading); ⑧
    if (err < 0) {
        LOG_ERR("Could not initialize ADC sequence.");
        return -1;
    }

    while (true) {
        err = adc_read_dt(&adc_channel, &pot_reading); ⑨
        if (err < 0) {
            LOG_ERR("Could not read");
            continue;
        }
        val_mV = (int)buf; ⑩
        err = adc_raw_to_millivolts_dt(&adc_channel, &val_mV); ⑪
        if (err < 0) {
            LOG_INF("ADC reading: raw = %d", buf);
        } else {
            LOG_INF("ADC reading: raw = %d, volts = %d mV", buf, val_mV);
        }
        k_msleep(2000);
    }
}
```

① The `zephyr,user` node did not have a label or an alias. To refer to it, we needed to use its *path*. Devicetree names can include commas and dashes, put those must be changed to underscores when referring to them in C code. Thus the `zephyr,user` node is referenced with the name `zephyr_user`.

② Declare an unsigned (non-negative) 16-bit integer variable to hold the raw reading from the ADC.

③ An `int` in Zephyr is a signed 32-bit integer.

④ The ADC requires an `adc_sequence` structure. This one is named `pot_reading` (pot is common shorthand for potentiometer). There are additional elements of this structure that could be specified, but in this program only the essentials will be set: `buffer` and `buffer_size`.

⑤ When an `&` is placed before a variable name in C, it indicates that the memory address itself of this variable is retrieved, rather than the value stored at that location. By telling the ADC this memory address, it will be able to put values there (which we will access using the usual name, via the name `buf`).

⑥ It is possible for an ADC to read multiple values. When it does that, it starts writing the first value at the address specified by `buffer` and subsequent values are placed in the following memory addresses. The ADC needs to be told how many memory addresses have been set aside for this so it doesn't write to a memory address that is being used for something else. The `sizeof` returns the number of bytes associated with a particular variable. For a 16-bit unsigned integer, that is 2 bytes.

⑦ Preparing the ADC for use is a two-step process. First, the ADC channel is configured.

⑧ The second step is to link an `adc_sequence` structure to the ADC channel.

⑨ Calling `adc_read_dt` triggers the actual analog-to-digital conversion. That value is stored in `buf` (remember that `pot_reading` contained its memory address).

⑩ This is an example of *type casting*. It takes a 16-bit unsigned integer and reformats it as a 32-bit signed integer. At this point `val_mV` holds the same value as `buf`, but in a different sized container.

⑪ The memory location of `val_mV` is given to `adc_raw_to_millivolts_dt`. It uses this to read the raw value stored there. It then uses its knowledge of the ADC settings to convert that raw value to one in millivolts. The memory location is then used to update `val_mV` so it instead has the value in millivolts.

6. Make a connection between GND on the development board and the ground bus on a breadboard.
7. Make connection between VDD on the development board and a power bus on the breadboard. It will be more convenient if it is the power bus adjacent to the ground bus that you connected in the previous step.
8. Connect the black wire of the potentiometer to the ground bus and the red wire to the power bus.
9. Connect the green wire of the potentiometer to a terminal strip on the breadboard. Also make a connection from that terminal strip to P0.02.
10. Build your application and flash it to the development board.
11. Observing the log messages printed to the terminal, turn the potentiometer knob. The low end of the range should be close to 0 V and the high end of the range should be 3.0 V if you are powering your development board via USB.

Exercise 6.1

1. Connect the **1+** flywire of the Analog Discovery to the same terminal strip as the green wire of the potentiometer.
2. Connect both the **1-** and ↓ flywires to ground.
3. Start *Waveforms* and open the Voltmeter application. Click **[Run]** to continuously update the readings.
4. Using the voltmeter as a guide, adjust the potentiometer knob until you produce a DC voltage of 1.5 V (as close as possible). Record the two analog input values reported by the microcontroller (the raw value and the millivolt value).
5. Repeat these measurements for potentiometer settings producing voltmeter readings of 0.1 V and 2.9 V.

Measuring light with a light-dependent-resistor

You will be using a light-dependent resistor (photocell). Your goal is to determine the best resistor to use in the voltage divider so you get maximum sensitivity under the conditions you might encounter indoors.

Exercise 6.2

1. Construct a voltage divider consisting of a 10 kΩ resistor and the LDR, with the resistor connected to the power bus and the LDR connected to the ground bus. Connect a wire to the junction between the resistor and the LDR. The other end of the wire should go to P0.02.
2. Using Program 3, record the millivolt reading for the following conditions:
    a. the LDR is covered
    b. the LDR is exposed to bright light (such as the “flashlight” of a smartphone)
3. Repeat the light and dark measurements for two other resistors: 4.7 kΩ and 47 kΩ.
4. For the particular LDR that you have, which resistor should you choose to get the greatest difference between the light and dark readings?

Measuring temperature with an analog sensor

The TMP36 is an analog temperature sensor working over the range -40°C to +125°C. The TMP36 produces a voltage of 750 mV when it is at a temperature of 25°C. Each 1°C temperature increase causes the voltage to increase by 10 mV. It comes in a standard package called TO-92-3, as shown in Figure 1. Many other things come in the same package so carefully look for the TMP36 label in tiny print on its face.

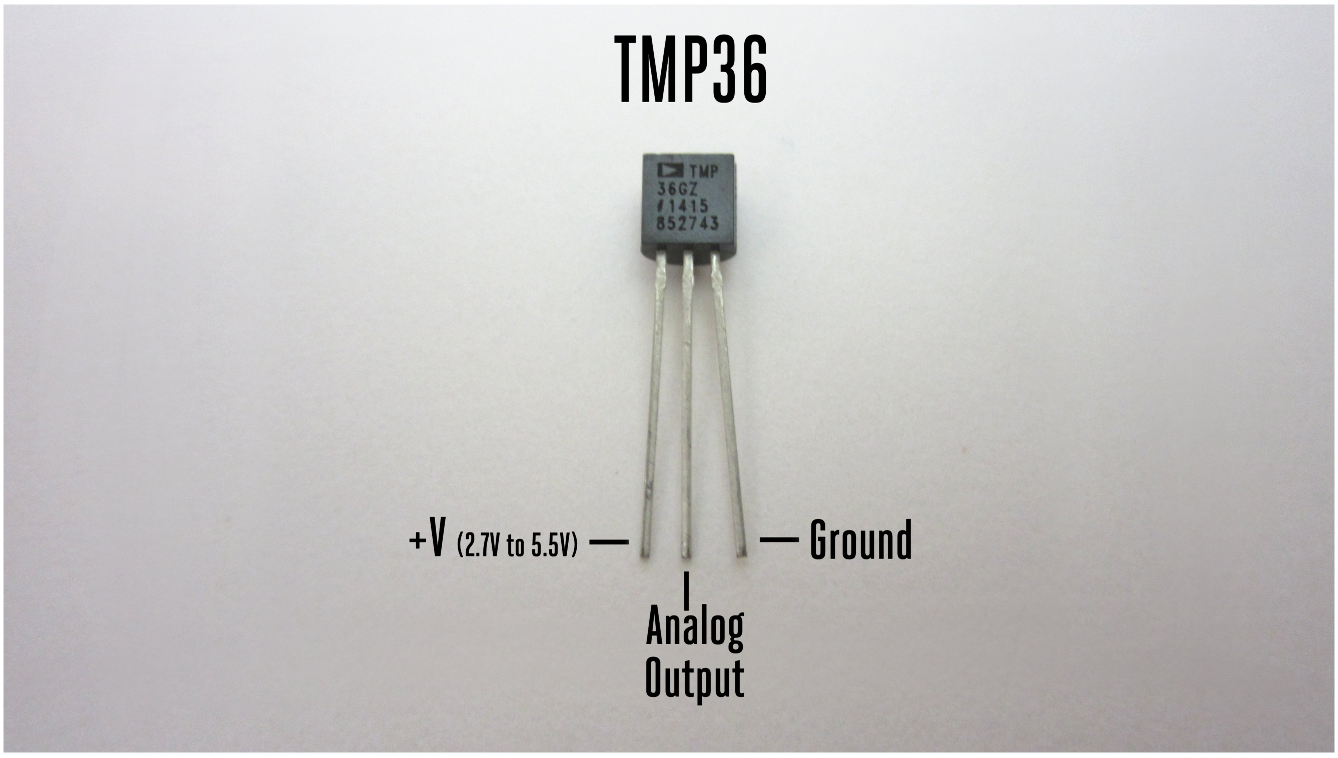

Figure 1. Pinout diagram for the TMP36 temperature sensor.

Previous examples have used a gain setting of 1/6, permitting input voltages up to 3.0 V (VDD). If we are using this to measure room air temperatures we might never expect a temperature above 40°C. This means the highest voltage that should be produced by the TMP36 is 900 mV. The goal is to select a gain setting that brings 900 mV to under 600 mV (the internal voltage reference of the ADC), but as close to it as possible. Given the possible options, a gain of 1/2 is the best fit to our criteria.

The long jumper wires commonly used while prototyping can pick up electromagnetic noise on this scale, so we will reduce that noise through averaging. The ADC can be configured to make multiple measurements as part of a single read command. We can also improve performance clustering the wires going to the ADC, GND, and VDD. To do this, we will switch the analog input to AIN1 (P0.03).

The `zephyr,user` node is not the only place we can use an ADC. In this exercise you will begin the work that would be needed to create a device driver for the Analog Device's TMP36. That requires creation of the appropriate binding file, similar to what was done previously with the servo.

1. Connect pin 1 of the TMP36 (the left leg when looking at its flat front) to the power bus strip. The power bus strip should be connected to the development board VDD.
2. Connect pin 2 of the TMP36 (the middle leg) to P0.03
3. Connect pin 3 of the TMP36 (the right leg) to the ground bus. As usual, the ground bus should also be connected to the development board GND.
4. Create a new application.
5. Create a new folder named `dts` at the top-level of your application (not inside any folder other than the one holding application itself). Inside of the `dts` folder create another folder `bindings`.
6. Create a file named `adi,tmp36.yaml` inside of the `bindings` folder. It is standard practice to name a binding file as follows:
7. Begin with an abbreviated form of the manufacturer's name. In this case the customary abbreviation for Analog Devices is `adi` (the same as its stock ticker).
8. This is followed by a comma and then the model of the device.
9. All of this is done using lowercase.
10. Add the following to that binding file:

```yaml
description: Analog Devices analog temperature sensor TMP36
compatible: "adi,tmp36"
include: sensor-device.yaml
properties:
  io-channels:
    required: true
    description: ADC channel for temperature sensor
  vtemp25:
    type: int
    default: 750
    description: |
      Temperature sensor voltage at 25 degrees Celsius
      in millivolts
  sensor-slope:
    type: int
    default: 10
    description: |
      Temperature sensor slope in millivolts per degree Celsius
```

11. Generate a build configuration and create an overlay.
12. Edit the overlay file. We are going to add the TMP36 using the binding just created.

```
&adc {
  #address-cells = <1>;
  #size-cells = <0>;
  status = "okay";
  channel@0 {
    reg = <0>;
    zephyr,gain = "ADC_GAIN_1_2";
    zephyr,reference = "ADC_REF_INTERNAL";
    zephyr,acquisition-time = <ADC_ACQ_TIME_DEFAULT>;
    zephyr,input-positive = <NRF_SAADC_AIN1>;
    zephyr,resolution = <12>;
  };
};

/{
  temp0: temp0 {
    compatible = "adi,tmp36";
    io-channels = <&adc 0>;
  };
};
```

13. Edit `prj.conf` to enable analog-to-digital conversion, logging, and output of floating point numbers.

```
CONFIG_ADC=y
CONFIG_LOG=y
CONFIG_CBPRINTF_FP_SUPPORT=y
```

14. You are now ready for the actual application code in `main.c`.

Program 4. Measure temperature using a TMP36 analog sensor.

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/adc.h>
#include <zephyr/logging/log.h>
#include <zephyr/devicetree.h>

LOG_MODULE_REGISTER(AnalogTemp, LOG_LEVEL_DBG);

#define NUM_ADC_READINGS    10
```

```
#define TMP36 DT_NODELABEL(temp0)
const struct adc_dt_spec tmp36 = ADC_DT_SPEC_GET(TMP36);

/* Use DT_PROP() to get volt-to-temp parameters */
#define MV_AT_25C DT_PROP(TMP36, vtemp25)
#define MV_PER_DEG_C DT_PROP(TMP36, sensor_slope)

int main(void) {
    int err;
    uint16_t buf[NUM_ADC_READINGS]; ①
    int val_mV;
    float avg_mV; ②
    float T_in_C;
    struct adc_sequence_options options = { ③
        .extra_samplings = NUM_ADC_READINGS-1, ④
        .interval_us = 100 ⑤
    };
    struct adc_sequence tmp_reading = {
        .options = &options, ⑥
        .buffer = buf,
        .buffer_size = sizeof(buf) ⑦
    };

    if (!adc_is_ready_dt(&tmp36)) {
        LOG_ERR("ADC controller device is not ready");
        return -1;
    }
    err = adc_channel_setup_dt(&tmp36);
    if (err < 0) {
        LOG_ERR("Could not setup channel.");
        return -1;
    }
    err = adc_sequence_init_dt(&tmp36, &tmp_reading);
    if (err < 0) {
        LOG_ERR("Could not initialize ADC sequence.");
        return -1;
    }

    while (true) {
        err = adc_read_dt(&tmp36, &tmp_reading);
        if (err < 0) {
            LOG_ERR("Could not read. Error code %d", err);
            k_msleep(2000);
            continue;
        }
        avg_mV = 0.0; ⑧
        for (int i = 0; i < NUM_ADC_READINGS; i++) { ⑨
            val_mV = (int)buf[i];
            adc_raw_to_millivolts_dt(&tmp36, &val_mV);
            avg_mV = avg_mV + (float)val_mV; ⑩
        }
        avg_mV = avg_mV/NUM_ADC_READINGS; ⑪
        T_in_C = 25.0 + (avg_mV - (float)MV_AT_25C)/(float)MV_PER_DEG_C; ⑫
        LOG_INF("T = %.1f deg C", T_in_C); ⑬
        k_msleep(5000);
    }
}
```

① The buffer for ADC values must now be an array because multiple values will be read.

② A `float` is required to store the average of multiple measurements because it will result in a non-integer value.

③ A structure to hold optional values that can be used when configuring an `adc_sequence`.

④ One measurement always take place in an ADC reading. This is the number of *additional* readings so it is one less than the total number of readings.

⑤ The time between the start of sequential ADC conversions, in microseconds. Note that it is *not* the time between the completion of one conversion and the beginning of the next. This means that if it is set to less than the time required for a single conversion to complete, an error will be generated.

⑥ The options previously stored are now added to the `adc_sequence` structure.

⑦ This is the size of the buffer (in bytes).

⑧ This variable will first be used to accumulate the sum of the measurement values. It must be set to 0 before those values are added to it.

⑨ A loop over all of the measurements.

⑩ Cast each measured millivolt value from an integer to a float before adding it to the accumulating sum.

⑪ The average is calculated by dividing the sum by the number of measurements.

⑫ Convert the average millivolt reading to a temperature using the sensor parameters.

⑬ The temperature is a floating point number so it must be displayed using the `%f` specifier. The `.1` between the `%` and the `f` indicates that it should be displayed to the tenths place.

Revisiting digital input

In this exercise you will revisit digital input, using the potentiometer to create a variable voltage but with the input now configured as digital, rather than analog. You will observe the digital logic levels.

Exercise 6.3

1. Connect VDD to the power bus on the breadboard, GND to the ground bus, and P0.02 to row 30.
2. Connect the potentiometer with the red lead to the power bus, the black lead to the ground bus, and the green lead to row 30.
3. Connect the **1+** flywire of the *Analog Discovery 2* to row 30. Connect **1-** and ⏚ to the

ground bus.

4. Create a new project containing Program 5. You will also need to generate an overlay defining P0.02 as a digital input called `ext_input` (no pull-up or pull-down resistors are needed).
5. Build and flash the application to your microcontroller.
6. Start *Waveforms* and open the Voltmeter application. Click **[Run]** to continuously update the readings.
7. Turn the potentiometer knob until the DC voltage reads 0.
8. Slowly turn the potentiometer knob until the LED turns on. Record the voltage.
9. Continue turning the potentiometer to produce an increasing voltage until the DC voltage reads 3.0 V.
10. Slowly turn the potentiometer knob the other way until the LED turns off. Record this voltage.
11. Starting from here, increase the voltage again until the LED turns on. Do you get the same voltage as you did the first time?
12. Decrease the voltage until the LED turns off. Is the voltage consistent with what you found earlier?

Program 5. Read a digital input and turn an LED on or off based on the input.

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

#define LED_NI  DT_ALIAS(led0)
#define EXT_INPUT_NI  DT_NODELABEL(ext_input)

const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(LED_NI, gpios);
const struct gpio_dt_spec ext_input = GPIO_DT_SPEC_GET(EXT_INPUT_NI, gpios);

int main(void) {
    gpio_pin_configure_dt(&led, GPIO_OUTPUT_INACTIVE);
    gpio_pin_configure_dt(&ext_input, GPIO_INPUT);

    while (true) {
        if (gpio_pin_get_dt(&ext_input)) {
            gpio_pin_set_dt(&led, 1);
        } else {
            gpio_pin_set_dt(&led, 0);
        }
    }
}
```

Your Turn

None today. Enjoy your weekend.