

Analog to Digital Conversion Example

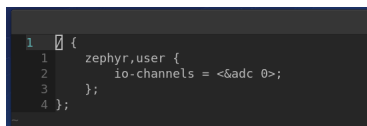
This example explains the procedure to read ADC samples of one or two ADC channels and it prints the readings on to the serial monitor. Also, if the driver supports, the ADC readings will be converted to *milli volts*.

Below given is the brief explanation of the ADC source code.

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

#if !DT_NODE_EXISTS(DT_PATH(zephyr_user)) || \
    !DT_NODE_HAS_PROP(DT_PATH(zephyr_user), io_channels)
#error "No suitable devicetree overlay specified"
#endif
```

Here, we check if the devicetree node exists in the tree using the board's overlay file contents. Here is an image of the overlay.



```
1 {
2   zephyr,user {
3     io-channels = <adc 0>;
4   };
}
```

Figure 8: nRF52840 DK Overlay for ADC

```
#define ADC_NUM_CHANNELS DT_PROP_LEN(DT_PATH(zephyr_user), io_channels)

#if ADC_NUM_CHANNELS > 2
#error "Currently only 1 or 2 channels supported in this sample"
#endif

#if ADC_NUM_CHANNELS == 2 && !DT_SAME_NODE( \
    DT_PHANDLE_BY_IDX(DT_PATH(zephyr_user), io_channels, 0), \
    DT_PHANDLE_BY_IDX(DT_PATH(zephyr_user), io_channels, 1))
#error "Channels have to use the same ADC."
#endif

#define ADC_NODE DT_PHANDLE(DT_PATH(zephyr_user), io_channels)

/* Common settings supported by most ADCs */
#define ADC_RESOLUTION 12
#define ADC_GAIN ADC_GAIN_1
#define ADC_REFERENCE ADC_REF_INTERNAL
#define ADC_ACQUISITION_TIME ADC_ACQ_TIME_DEFAULT
```

Next, we declare the number of ADC channels taken from the function **DT_PROP_LEN(node_id, property)** which is used to find the logical array length of any property.

This example is written for using either one or two ADC channels. We need to throw an error if the board has more than two ADC channels. The function **DT_SAME_NODE(node_id1, node_id2)** checks if two node identifiers are referring to the same node or not.

DT_INST_PHANDLE_BY_IDX() is used to get a **DT_DRV_COMPAT** instance's node identifier for a phandle in a property.

Here, **phandle** refers to a pointer for the node pointing to the definition of the node in that file. **DT_PATH()** returns a node identifier for a devicetree path. This is done to check if the sensor channels are using the same ADC unit or otherwise.

DT_PHANDLE(node_id, prop) is used to get a node identifier for a phandle property's value. It gets a pointer to the ADC node of the board. Next, we define some common settings of the ADC such as gain, resolution, acquisition time, etc. *Acquisition time* refers to the time required to capture input voltage during sampling, equivalent, in a way, to the *sampling time*.

```
static uint8_t channel_ids[ADC_NUM_CHANNELS] = {
    DT_IO_CHANNELS_INPUT_BY_IDX(DT_PATH(zephyr_user), 0),
#ifdef ADC_NUM_CHANNELS == 2
    DT_IO_CHANNELS_INPUT_BY_IDX(DT_PATH(zephyr_user), 1)
#endif
};
static int16_t sample_buffer[ADC_NUM_CHANNELS];
```

We then get the number of channels along with their *channel_id*'s using **DT_IO_CHANNELS_INPUT_BY_IDX(node_id, idx)**, where *idx* is the logical index at which io-channel property is retrieved and *node_id* is the identifier.

A *sample_buffer* is initialized with its size equal to the number of channels.

```
struct adc_channel_cfg channel_cfg =
{
    .gain = ADC_GAIN,
    .reference = ADC_REFERENCE,
    .acquisition_time = ADC_ACQUISITION_TIME,
    .channel_id = 0,
    .differential = 0
};
```

```

struct adc_sequence sequence =
{
    .channels    = 0,
    .buffer      = sample_buffer,
    .buffer_size = sizeof(sample_buffer),
    .resolution  = ADC_RESOLUTION,
};

```

We also need to initialize a structure to hold the configurations of the ADC channel such as gain, reference, acquisition time, channel ID, and channel type (differential, in this case).

Another structure named **adc_sequence** is a pointer to a structure which is used to define additional options for the sequence. If its a pointer to NULL, then the sequence consists of a single sampling. The additional options are channels, buffer, buffer size, resolution, etc.

```

void main(void)
{
    int err;
    const struct device *dev_adc = DEVICE_DT_GET(ADC_NODE);
    if (!device_is_ready(dev_adc))
    {
        printk("ADC device not found\n");
        return;
    }
    for (uint8_t i = 0; i < ADC_NUM_CHANNELS; i++)
    {
        channel_cfg.channel_id = channel_ids[i];
        #ifdef CONFIG_ADC_NRF5_SAADC
            channel_cfg.input_positive = SAADC_CH_PSELP_PSELP_AnalogInput0 + channel_ids[i];
        #endif
        adc_channel_setup(dev_adc, &channel_cfg);
        sequence.channels |= BIT(channel_ids[i]);
    }

    int32_t adc_vref = adc_ref_internal(dev_adc);
}

```

In the main function, we first define an integer *err* to store the value after the **adc_read()** sets a read request.

We also need to set device pointer to the ADC node taken from the devicetree and then check if the device is ready to use by verifying if the provided device pointer is for a device in a state where it can be utilized with a standard API.

The channels are to be configured individually before sampling. Using a *for loop*, the item **channel_id** of **channel_cfg** is set to the **channel_id**'s declared as *uint8_t* earlier.

We also perform a check to see if the board supports SAR ADC or SAADC (Successive Approximation Analog-to-digital Converter) which employs a Successive Approximation Register to utilize it.

adc_channel_setup() is used to configure the channel before selecting it for a read request.

The *channels* property of the **struct adc_sequence** *sequence* has to be added individually. Therefore we use a utility called **BIT(n)**, where *n* is the position (in our case, we are adding individually using the indexes of *channel_ids*) by performing *bit wise OR* operation.

adc_ref_internal() is the most important function. This is precisely what gets us the internal reference voltage and it is stored in *adc_vref* variable.

```
while (1)
{
err = adc_read(dev_adc, &sequence);
    if (err != 0)
    {
        printk("ADC reading failed with error %d.\n", err);
        return;
    }

    printk("ADC reading:");
    for (uint8_t i = 0; i < ADC_NUM_CHANNELS; i++)
    {
        int32_t raw_value = sample_buffer[i];
        printk(" %d", raw_value);
        if (adc_vref > 0)
        {
            int32_t mv_value = raw_value;
            adc_raw_to_millivolts(adc_vref, ADC_GAIN, ADC_RESOLUTION, &mv_value);
            printk(" = %d mV ", mv_value);
        }
    }

    printk("\n");
    k_sleep(K_MSEC(1000));
}
```

An infinite loop is initiated in which we check if *err* is equal to zero. If it is not zero, then the ADC data has not been read.

If it is equal to zero, then the raw ADC values can be read from the **sample_buffer**.

As mentioned earlier, the raw readings can be converted to the unit *milli volts* if the driver supports it by using the function **adc_raw_to_millivolts()** which is then printed on to the console.

We need to run

```
west build -p auto -b nrf52840dk_nrf52811 adc
west flash --erase
```

If the build process is successful and the program has been flashed on to the board's memory, we should be able to view the output on the console by typing

```
minicom -D /dev/ttyACM* -b 115200
```

`/dev/ttyACM*` can be replaced with appropriate COM port name (For example, `/dev/ttyUSB*`) as per the local machine. More information on `minicom` is provided in section I on interfacing MPU6050.

Image shown below is the serial monitor output which displays the ADC readings taken from the ADC channel in *milli volts* unit.

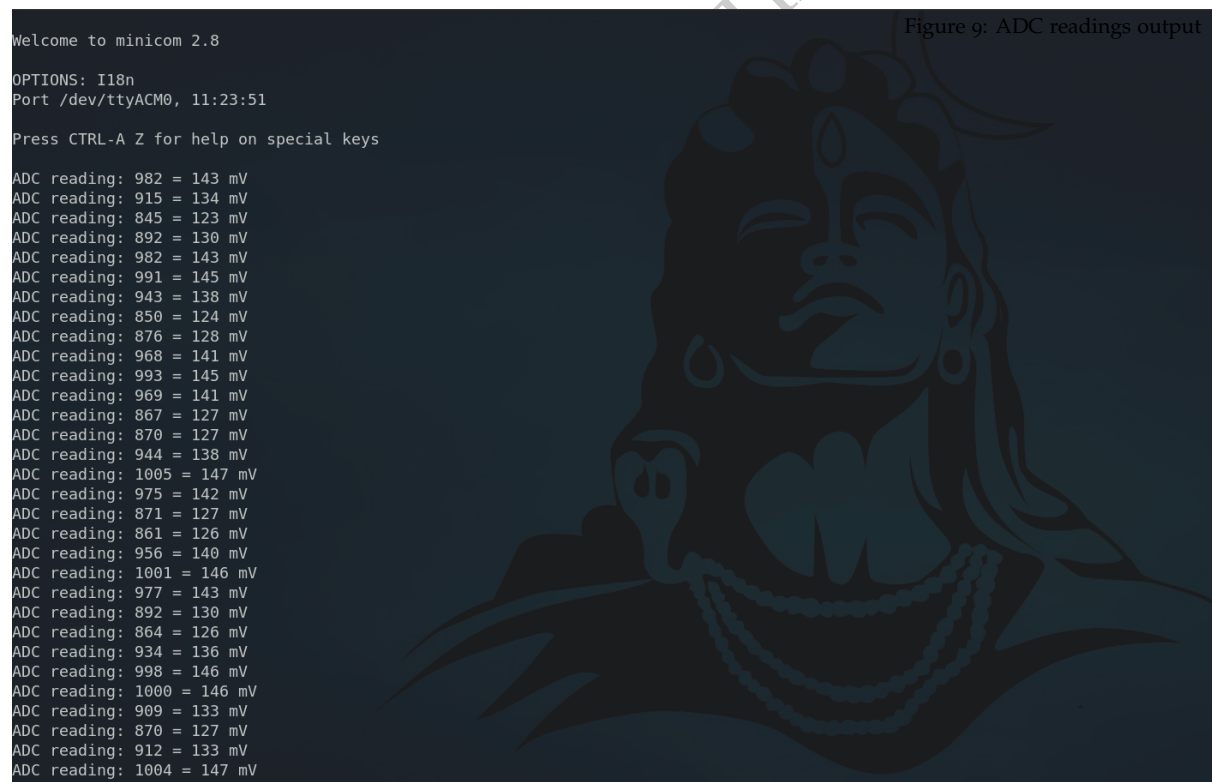


Figure 9: ADC readings output