Hardware Drivers

Contents

nRF5 SDK v11.0.0-2.alpha

Hardware Drivers

ADC *nRF51 only*

Clock

GPIOTE

I2S *nRF52 only*

LPCOMP

PDM *nRF52 only*

PPI

PWM *nRF52 only*

QDEC

RNG

SAADC *nRF52 only*

SPI master

SPI slave

SWI

Timer

TWI master

TWI slave *nRF52 only*

UART

WDT

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.

1. nRF5 SDK v11.0.0-2.alpha

ADC

*This information applies to the **nRF51 Series** only.*

The ADC hardware access layer (HAL) provides basic APIs for accessing the registers of the analog-to-digital converter (ADC). See the API documentation for the ADC HAL for details.

The ADC HAL Example provides sample code that you can use to quickly get started.

Clock

The clock driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the clock. See the API documentation for the Clock HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the Clock driver for details.

The clock driver is responsible for starting the high-accuracy, high-frequency clock source and the low-frequency clock (without a SoftDevice). It also performs calibration without a SoftDevice or internal RC as an LFCLK source.

Driver configuration

Enable the driver in the configuration file `nrf_drv_config.h` before using it in an application. The clock driver supports only the static, compile-time configuration due to the possibility of other modules attempting to initialize it. The configuration includes the high-frequency clock frequency, the low-frequency clock source, and the interrupt priority.

```
#define CLOCK_ENABLED 1
```

Enabling the clock will initialize the Clock driver to its default state:

- High-accuracy, high-frequency clock HFCLK is off
- Low-frequency clock LFCLOCK is off (without a SoftDevice)
- Calibration has not started (without a SoftDevice only)

Using the clock driver

The Clock Example provides sample code that you can use to quickly get started.

Controlling the HFCLK

If a module needs a high-precision clock, it can request the clock driver to start it. If a callback is provided, it will be called after the request is handled. If the requested clock was already started when nrf_drv_clock_hfclk_request is called, the callback will be called from the context of this function. Otherwise, it will be called from the context of the clock interrupt handler. To check if the HFCLK is on, call nrf_drv_clock_hfclk_is_running.

When the module is finished with the HFCLK, you should release it by calling nrf_drv_clock_hfclk_release. The clock is turned off if no other module is requesting it.

The SoftDevice API is used in the driver if a SoftDevice is present.

Controlling the LFCLK (without a SoftDevice only)

If a module needs a low-frequency clock, it can request the clock driver to start it. If a callback is provided, it will be called after the request is handled. If the requested clock was already started when nrf_drv_clock_lfclk_request is called, the callback will be called from the context of this function. Otherwise, it will be called from the context of the clock interrupt handler. To check if the LFCLK is on, call nrf_drv_clock_lfclk_is_running.

When the module is finished with the LFCLK, you should release it by calling nrf_drv_clock_lfclk_release. The clock is turned off if no other module is requesting it.

The SoftDevice API is used in the driver if a SoftDevice is present.

Controlling the calibration of the LFCLK (without a SoftDevice and using RC as source clock)

The internal RC can be used as source for the LFCLK, but it is not as precise as a crystal. It must be periodically calibrated against a high-frequency crystal. The clock module supports autonomous calibration. It also has a built-in low-power timer dedicated to calibration.

The calibration process consists of three phases:

- Delay (using a clock calibration timer); delay can be set to 0 to start calibration immediately
- Request the high-accuracy HFCLK
- Perform calibration

Call nrf_drv_clock_calibration_start to start the calibration process. Parameters specify the delay (given in 0.25 ms units) and an optional callback to be called once calibration is completed.

To check if there is an ongoing calibration, call nrf_drv_clock_is_calibrating.

At any time, nrf_drv_clock_calibration_abort can be called to stop calibration. If a callback was provided for nrf_drv_clock_calibration_start, the callback is executed when the process stops, with the event type set to NRF_DRV_CLOCK_EVT_CAL_ABORTED.

Periodic calibration can be achieved by calling nrf_drv_clock_calibration_start from the calibration completion handler:

```
 void clock_handler(nrf_drv_clock_evt_type_t event)
 {
 if (event == NRF_DRV_CLOCK_EVT_CAL_DONE)
{ ret = nrf_drv_clock_calibration_start(10, clock_handler); }
}
```

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
3. nRF5 SDK v11.0.0-2.alpha

GPIOTE

The GPIOTE driver controls the GPIO and GPIOTE peripherals and configures and controls output and input pins. Output pins can be configured to be controlled manually or by a GPIOTE task. Input pins can be configured to be controlled in different modes:

- High accuracy: An independent GPIOTE event is used to detect changes on the pin. If a pin is configured to be controlled in this mode, a high frequency clock is required.
- Low accuracy/low power: A PORT event senses level changes on the pin. One PORT event can be used for multiple pins. Therefore, it is not that accurate and cannot be used to track high-speed pin changes. On the other hand, if a pin is configured in this mode, the system can turn off the high frequency clock when sleeping.

TASK/EVENT channels allocation

The number of TASK/EVENT channels that can be used to drive an output pin using a GPIOTE task or to generate an event on input pin change is limited. The driver manages those channels. The user cannot control which channels are used.

When nrf_drv_gpiote_in_init or nrf_drv_gpiote_out_init is called, the driver allocates a channel from the pool. If all channels are already allocated, the functions return an error code. Note that nrf_drv_gpiote_out_init allocates a channel only if nrf_drv_gpiote_out_config_t indicates that the pin will be controlled by a task.

When nrf_drv_gpiote_in_uninit or nrf_drv_gpiote_out_uninit is called, the driver frees the allocated channel.

Driver configuration

To ensure that the driver configuration is known to all modules, only static configuration is supported. Therefore, nrf_drv_gpiote_init has no input parameters. The static configuration parameters are located in the `nrf_drv_config.h` file.

The configuration parameters include:

- Interrupt priority level
- Maximum amount of low-power events (value needed to statically allocate array for user handlers)

Driver initialization

The GPIOTE driver is a shared resource that can be used by multiple modules in an application. Therefore, it can be initialized only once. If a module is using the driver, it must check if it has already been initialized by calling the function nrf_drv_gpiote_is_init. If this function returns false, the module must initialize the driver by calling the function nrf_drv_gpiote_init.

The following code example shows how to initialize the driver:

```
uint32_t err_code;
if(!nrf_drv_gpiote_is_init())
{
    err_code = nrf_drv_gpiote_init();
}
```

Controlling output pins

After initialization, output pins can be configured to be controlled manually or by a GPIOTE task.

Initialization

Output pins are initialized by calling the function nrf_drv_gpiote_out_init. The argument of type nrf_drv_gpiote_out_config_t specifies the configuration of the output pin. You can use the following macros to set the configuration:

- GPIOTE_CONFIG_OUT_SIMPLE: Indicates that the pin will be driven by driver function calls and not by a task. As parameter, you can specify the initial pin state.
- GPIOTE_CONFIG_OUT_TASK_LOW: Indicates that a task will be used to clear the pin. The pin is initially set.
- GPIOTE_CONFIG_OUT_TASK_HIGH: Indicates that a task will be used to set the pin. The pin is initially cleared.
- GPIOTE_CONFIG_OUT_TASK_TOGGLE: Indicates that a task will be used to toggle the pin. The pin is initially set. As parameter, you can specify the initial pin state.

The configuration can be modified before calling nrf_drv_gpiote_out_init.

The following code example configures the 10th pin to be used by the task that sets the pin. The pin is initially set.

```
nrf_drv_gpiote_out_config_t config = GPIOTE_CONFIG_OUT_TASK_HIGH;
config.init_set = true;
err_code = nrf_drv_gpiote_out_config(10, &config);
```

Manually controlled output pins

When a pin is configured as non-task pin, it can be controlled by calling the following functions:

- nrf_drv_gpiote_out_set
- nrf_drv_gpiote_out_clear
- nrf_drv_gpiote_out_toggle

The driver asserts if the functions are called on a pin that is configured as task pin.

Task-controlled output pins

When a pin is configured as task pin, it can be controlled using the CPU by writing 1 to the relevant TASK register, or using PPI. Before a pin can be controlled, it must be enabled by calling the function nrf_drv_gpiote_out_task_enable.

When a pin is initialized as task pin, it allocates one of the available TASK/EVENT channels. Therefore, you must call nrf_drv_gpiote_out_uninit when the pin is no longer used.

The following code example shows how to use a task pin to be toggled using a task:

```
nrf_drv_gpiote_out_config_t config = GPIOTE_CONFIG_OUT_TASK_TOGGLE
      (false);
err_code = nrf_drv_gpiote_out_init(10, &config);
if (err_code != NRF_SUCCESS)
{
    // handle error condition
}
ppi_task_addr = nrf_drv_gpiote_out_task_addr_get(10);
// Configure PPI using ppi_task_address or use the address to manually
      control the task.

nrf_drv_gpiote_out_task_enable(10);

// Task is enabled and can be used by PPI.

nrf_drv_gpiote_out_task_disable(10);

// Task is disabled and cannot be controlled.

nrf_drv_gpiote_out_task_enable(10);

// Task is enabled again and can be controlled using ppi_task_addr.

nrf_drv_gpiote_out_uninit(10);

// Pin is uninitialized. Task is disabled and TASK/EVENT channel is
      freed.
```

Controlling input pins

The GPIOTE peripheral can generate events on pin level change using the following methods:

- Using a dedicated IN_EVENT: When using a dedicated IN_EVENT, one pin is assigned to an event. Only a limited number of pins can be sensed this way because there is a limited number of TASK/EVENT channels. When an IN_EVENT is enabled, it requests the high frequency clock. Therefore, this method is not suitable for scenarios where the system sleeps for a long period of time, expecting to be waken up by a pin level change. This method supports detection of toggling, low-to-high, and high-to-low transition.
- Using a PORT event: A PORT event is a single event that is triggered whenever the value of any of the input pins that have sensing enabled changes. Pins can be configured to sense low-to-high or high-to-low transition. Toggle sensing is emulated in the driver by changing the sense configuration whenever a PORT event for a given pin is detected. Using a PORT event requires only a low frequency clock. Therefore, this method is

suitable for scenarios where the application expects to be inactive for a long period of time, waiting for a pin level change. The event can be shared by multiple pins; therefore it should be used for sensing slowly changing signals only.

Initialization

Input pins are initialized by calling the function nrf_drv_gpiote_in_init. The argument of type nrf_drv_gpiote_in_config_t specifies the configuration of the input pin. You can use the following macros to set the configuration:

- GPIOTE_CONFIG_IN_SENSE_LOTOHI: Indicates that the input pin will generate an event when the input value is changed to high. As parameter, you can specify if the input should have high accuracy or not.
- GPIOTE_CONFIG_IN_SENSE_HITOLO: Indicates that the input pin will generate an event when the input value is changed to low. As parameter, you can specify if the input should have high accuracy or not.
- GPIOTE_CONFIG_IN_SENSE_TOGGLE: Indicates that the input pin will generate an event when the input value is changed. As parameter, you can specify if the input should have high accuracy or not.

The configuration can be modified before calling nrf_drv_gpiote_in_init. Note that macros are not setting any pull.

During the initialization of an input pin, the user must provide an event handler. The event handler will be called by the driver when the configured transition occurs and the interrupt is enabled.

The following code example configures the 10th pin to have pull-up and generate an event on pin toggling:

```
nrf_drv_gpiote_in_config_t config = GPIOTE_CONFIG_IN_SENSE_TOGGLE;
config.pull = NRF_GPIO_PIN_PULLUP;
err_code = nrf_drv_gpiote_in_config(10, &config, pin_event_handler);
```

Usage

From a software point of view, the input sensing functionality is independent of the pin mode. If a high-accuracy pin is used, it can detect signals that are changed faster. Turning off high accuracy, on the other hand, enables the low-power feature.

Events that are generated by an input pin level change can be used to trigger events to be utilized by PPI, and optionally they can trigger a user event handler.

To start sensing an input pin, it must be enabled.

Pin input values can always be checked by calling the function nrf_drv_gpiote_in_is_set.

The following code example shows how to use a pin to detect toggling using high accuracy:

```
nrf_drv_gpiote_in_config_t config = GPIOTE_CONFIG_IN_SENSE_TOGGLE
      (true);
err_code = nrf_drv_gpiote_in_init(10, &config, pin_event_handler);
if (err_code != NRF_SUCCESS)
{
    // handle error condition
}
ppi_event_addr = nrf_drv_gpiote_in_event_addr_get(10);
// Configure PPI using ppi_event_address.

// Enable event, interrupt is not enabled.
nrf_drv_gpiote_in_event_enable(10, false);

// Event is enabled and can be used by PPI.

nrf_drv_gpiote_in_event_disable(10);

// Task is disabled and cannot be controlled.

// Event and corresponding interrupt are enabled.
nrf_drv_gpiote_in_event_enable(10, true);

// Event is enabled again and can be used by PPI.
// Additionally, any toggling sensed will trigger interrupt

nrf_drv_gpiote_in_uninit(10);

// Pin is uninitialized. Event is disabled and TASK/EVENT channel is
      freed.
```

Working with a SoftDevice

When a SoftDevice is enabled, it can interrupt the application at any time for a certain amount of time. This can lead to the situation where some pin level changes are missed. If the application must track an input pin that can change its level frequently, PPI should be used with a high-accuracy event together with a TIMER in counter mode to count the detected transitions.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.

I2S

*This information applies to the **nRF52 Series** only.*

The Inter-IC Sound (I2S) interface driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the I2S peripheral. See the API documentation for the I2S HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the I2S driver for details.

Key features include:

- Three modes of operation:
  - Reception (RX) only
  - Transmission (TX) only
  - Both directions simultaneously
- Data exchange through a single data handler that is called by the driver when new data is received or new data is needed for transmission

The I2S Loopback Example provides sample code that you can use to quickly get started.

Driver configuration

The configurable parameters include:

- Pins to be used for I2S peripheral signals
- Operating mode (master or slave)
- Format of frames, used channels (stereo, left or right)
- Width and alignment of samples
- Clock settings (defining the sampling frequency in master mode)

The default configuration for the I2S driver is located in `nrf_drv_config.h`. This configuration is used when nrf_drv_i2s_init is called with a NULL pointer to the configuration structure.

The driver must be explicitly enabled in `nrf_drv_config.h` before it can be used. A proper data handling function must be provided during the initialization.

Using the I2S driver

First, define the data handling function for I2S transfers. For example:

```
static void data_handler(uint32_t const * p_data_received,
                         uint32_t       * p_data_to_send,
                         uint16_t         number_of_words)
{
    if (p_data_received != NULL)
    {
        // Process the received data.
    }

    if (p_data_to_send != NULL)
    {
        // Provide the data to be sent.
```

```
    }
}
```

Next, call nrf_drv_i2s_init to initialize and configure the instance. See the nrf_drv_i2s_config_t structure documentation for possible configuration options.

For example:

```
uint32_t err_code;

nrf_drv_i2s_config_t config = NRF_DRV_I2S_DEFAULT_CONFIG;
config.mck_setup = NRF_I2S_MCK_32MDIV21;
config.ratio     = NRF_I2S_RATIO_96X;
err_code = nrf_drv_i2s_init(&config, data_handler);
if (err_code != NRF_SUCCESS)
{
    // Initialization failed. Take recovery action.
}
```

Start the transfer by calling nrf_drv_i2s_start and providing a proper buffer for a given direction if it should be enabled. See the following examples:

- RX only:

```
nrf_drv_i2s_start(m_buffer_rx, NULL,        I2S_BUFFER_SIZE, 0);
```

- TX only:

```
nrf_drv_i2s_start(NULL,        m_buffer_tx, I2S_BUFFER_SIZE, 0);
```

- Both directions:

```
nrf_drv_i2s_start(m_buffer_rx, m_buffer_tx, I2S_BUFFER_SIZE, 0);
```

If both RX and TX are enabled, you can use the NRF_DRV_I2S_FLAG_SYNCHRONIZED_MODE flag to force a common call to the data handler for RX and TX data. Normally, the driver calls the data handler as soon as the event for a given direction is generated by the I2S peripheral. In synchronized mode, the call is deferred until events for both directions occur. This mode is useful, for example, when received data should be processed and then transmitted.

At any time, the transfer can be stopped immediately by calling nrf_drv_i2s_stop.

When the I2S driver is no longer needed or its configuration must be changed, call nrf_drv_i2s_uninit.

LPCOMP

The Low Power Comparator (LPCOMP) driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the LPCOMP. All functions in this layer are implemented as inline functions. See the API documentation for the LPCOMP HAL for details.

The driver layer provides support for single instances and APIs on a higher level than the HAL. See the API documentation for the LPCOMP driver for details.

The LPCOMP Example provides sample code that you can use to quickly get started.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
6. nRF5 SDK v11.0.0-2.alpha

PDM

*This information applies to the **nRF52 Series** only.*

The PDM driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the PDM peripheral. See the API documentation for the PDM HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the PDM driver for details.

Key features include:

- Asynchronous conversion using DMA.
- Mono or stereo sampling.
- Adjustable gain levels.
- PCM output.

Driver configuration

The PDM default configuration is located in `nrf_drv_config.h`. The PDM peripheral must be explicitly enabled in `nrf_drv_config.h` before the driver can be used. You must provide a valid configuration structure and an event handler function during initialization. The driver requires two sample buffers that are filled alternately.

The interface is initialized by calling the nrf_drv_pdm_init function. The driver configuration includes:

- Sampling mode.
- PDM clock frequency.
- Gain levels.
- Two sample buffers.
- Interrupt priority level.

Example:

```
uint16_t buff[2][512]

void pdm_event_handler(uint32_t * p_buffer, uint16_t length)
{
    // Process the data.
}

/* ... */

nrf_drv_pdm_config_t pdm_cfg = NRF_DRV_PDM_DEFAULT_CONFIG(CLK_PIN,
      DOUT_PIN,
                                                          buff[0], buff
      [1], 512);
ret_code_t err_code = nrf_drv_pdm_init(&pdm_cfg, &pdm_event_handler);
```

Using the PDM driver

After initialization, sampling is started by calling the nrf_drv_pdm_start function. When the current buffer is filled with data, the event handler function is called with the buffer pointer and size as arguments, and the driver starts filling the second buffer. The output PCM samples are stored as 16-bit signed integers. If stereo mode is selected, the buffer will contain alternating left and right channel samples. Sampling runs continuously until nrf_drv_pdm_stop is called.

PPI

The Programmable Peripheral Interconnect (PPI) driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the PPI. All functions in this layer are implemented as inline functions. See the API documentation for the PPI HAL for details.

The driver layer provides support for the PPI module and APIs on a higher level than the HAL. See the API documentation for the PPI driver for details.

The PPI Example provides sample code that you can use to quickly get started.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
8. nRF5 SDK v11.0.0-2.alpha

PWM

*This information applies to the **nRF52 Series** only.*

The Pulse Width Modulation (PWM) module driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the PWM peripheral. See the API documentation for the PWM HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the PWM driver for details.

Key features include:

- Multi-instance support
- Two kinds of playback:
  - Simple: a single sequence of duty cycle values
  - Complex: two concatenated sequences
- Sequences are played back a specified number of times
- Optionally, the whole playback can be repeated in a loop
- A user-defined event handler can be used to perform additional actions when a sequence or the whole playback is finished
- If no event handler is used, playbacks can be carried out without involving the CPU

The PWM Driver Example provides sample code that you can use to quickly get started.

Driver configuration

The configurable parameters include:

- Pins to be used as outputs for PWM channels (up to 4 per instance)
- Base clock frequency and top counter value (these parameters jointly determine the PWM period and resolution)
- Mode of loading the sequence values from RAM and distributing them to compare registers:
  - Common: One value is used in all channels.

    - Grouped: One value is used in channels 0 and 1, and another one in channels 2 and 3.
    - Individual: Separate values are used in each channel.
    - Waveform: Similar to Individual mode, but only three channels are used. The fourth value is loaded into the pulse generator counter as its top value.
- Mode of advancing the active sequence (automatic or triggered via a task)

The default configuration for the PWM driver is located in `nrf_drv_config.h`. This configuration is used when nrf_drv_pwm_init is called with a NULL pointer to the configuration structure.

Before it can be used for a particular instance of the peripheral, the driver must be explicitly enabled for this instance in `nrf_drv_config.h`.

Using the PWM driver

First, call nrf_drv_pwm_init to initialize and configure the driver for a given instance of the peripheral. See the nrf_drv_pwm_config_t structure documentation for possible configuration options.

For example:

```
uint32_t err_code;
nrf_drv_pwm_config_t const config0 =
{
    .output_pins =
    {
        BSP_LED_0 | NRF_DRV_PWM_PIN_INVERTED, // channel 0
        NRF_DRV_PWM_PIN_NOT_USED,             // channel 1
        NRF_DRV_PWM_PIN_NOT_USED,             // channel 2
        NRF_DRV_PWM_PIN_NOT_USED,             // channel 3
    },
    .base_clock = NRF_PWM_CLK_1MHz,
    .count_mode = NRF_PWM_MODE_UP,
    .top_value  = 1000,
    .load_mode  = NRF_PWM_LOAD_COMMON,
    .step_mode  = NRF_PWM_STEP_AUTO
};
err_code = nrf_drv_pwm_init(&m_pwm0, &config0, NULL);
if (err_code != NRF_SUCCESS)
{
    // Initialization failed. Take recovery action.
}
```

Next, define a sequence of duty cycle values using the nrf_pwm_sequence_t structure. For example:

```
static nrf_pwm_values_common_t seq_values[] =
{
```

```
    0, 200, 400, 600, 800, 1000
};
nrf_pwm_sequence_t const seq =
{
    .values.p_common = seq_values,
    .length          = NRF_PWM_VALUES_LENGTH(seq_values),
    .repeats         = 0,
    .end_delay       = 0
};
```

Start the playback of the sequence by calling nrf_drv_pwm_simple_playback.

You can play the sequence several times (for example, 3 times):

```
nrf_drv_pwm_simple_playback(&m_pwm0, &seq, 3, 0);
```

Additionally, the playback can be repeated in a loop:

```
nrf_drv_pwm_simple_playback(&m_pwm0, &seq, 3, NRF_DRV_PWM_FLAG_LOOP);
```

If a more complicated pattern of duty cycles is required that involves two sequence definitions, use nrf_drv_pwm_complex_playback.

If a sequence must be updated during the playback, define a proper event handler and pass it as parameter in the call to nrf_drv_pwm_init. By default, the handler is called at the end of the playback (or right before the next iteration in a loop), but it can be configured to be called when a sequence is completely loaded.

For example:

```
static void event_handler(nrf_drv_pwm_evt_type_t event_type)
{
    if (event_type == NRF_DRV_PWM_EVT_END_SEQ0)
    {
        // Update sequence 0.
        /* ... */
    }
    else if (event_type == NRF_DRV_PWM_EVT_END_SEQ1)
    {
        // Update sequence 1.
        /* ... */
    }
}

/* ... */

    err_code = nrf_drv_pwm_init(&m_pwm0, &config0, event_handler);

/* ... */
```

```
    nrf_drv_pwm_complex_playback(&m_pwm0, &seq0, &seq1, 1,
        NRF_DRV_PWM_FLAG_LOOP |
        NRF_DRV_PWM_FLAG_SIGNAL_END_SEQ0 |
        NRF_DRV_PWM_FLAG_SIGNAL_END_SEQ1 |
        NRF_DRV_PWM_FLAG_NO_EVT_FINISHED);
```

Entries in the array of duty cycle values can be modified directly, since the peripheral loads the values as the sequence is played back. If the whole sequence definition must be changed, use nrf_drv_pwm_sequence_update. If only some part of the definition must be modified, you can use one of the other updating functions.

At any time, the playback can be stopped by calling nrf_drv_pwm_stop. The pulse generation will stop at the end of the current PWM period.

When the PWM driver is no longer needed or its configuration must be changed, call nrf_drv_pwm_uninit.

QDEC

The quadrature decoder (QDEC) driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the QEDC. See the API documentation for the QDEC HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the QDEC driver for details.

The QDEC Example provides sample code that you can use to quickly get started.

RNG

The random number generator (RNG) driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the random number generator. See the API documentation for the RNG HAL and driver for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the RNG driver for details.

The Random Number Generator Example provides sample code that you can use to quickly get started.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
11. nRF5 SDK v11.0.0-2.alpha

SAADC

*This information applies to the **nRF52 Series** only.*

The SAADC driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the SAADC peripheral. See the API documentation for the SAADC HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the SAADC driver for details.

Key features include:

- Blocking function to convert one sample on a single channel.
- Non-blocking function for triggering conversion on all enabled channels to a provided buffer.
- Double buffering: you can set up two buffers, and conversion to the new buffer starts immediately after the first one is filled.
- PAN-28 (scan mode not functional) workaround inside the driver: scan mode is emulated inside the driver, resulting in an interrupt being generated at every conversion on every channel.
- Optimal performance with interrupt only at the end of buffer conversion is achieved only on a single channel due to PAN-28.

Driver configuration

The SAADC default configuration is located in `nrf_drv_config.h`. The SAADC must be explicitly enabled in `nrf_drv_config.h` before the driver can be used. If nrf_drv_saadc_init is called with a NULL pointer to the configuration structure, the default configuration from `nrf_drv_config.h` is used. You must provide an event handler function during initialization.

Driver configuration includes:

- Resolution
- Enabling or disabling oversampling
- Interrupt priority level

Note

If oversampling is enabled, only one channel can be enabled. nrf_drv_saadc_channel_init asserts if this condition is not met.

Each channel is configured independently. Channels are configured and enabled by the function nrf_drv_saadc_channel_init. NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE or NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL can be used to build a default configuration for a channel.

Example:

```
nrf_saadc_channel_config_t channel_config =
      NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN6);
channel_config.gain = NRF_SAADC_GAIN1_2;
//Initialization and enabling of channel 0 to use analog input 6.
err_code = nrf_drv_saadc_channel_init(0, &channel_config);
```

nrf_drv_saadc_gpio_to_ain can be used to convert a GPIO pin number to an analog input number.

Using the SAADC driver

The SAADC driver can be used in blocking mode or non-blocking mode.

Blocking mode

The function nrf_drv_saadc_sample_convert is blocking and returns when the requested channel is sampled. The channel must be initialized before it can be used. If the SAADC is busy, the function returns with an error. In blocking mode, the driver does not use the peripheral interrupt and there is no context switching inside the driver.

Non-blocking mode

The function nrf_drv_saadc_buffer_convert can be used to start conversion in non-blocking mode. The function returns immediately after the buffer is configured. If the driver is busy, it returns with an error. nrf_drv_saadc_buffer_convert sets the SAADC up for conversion, but does not trigger sampling. To trigger sampling, call the function nrf_drv_saadc_sample or, through PPI, use the task SAMPLE from SAADC. nrf_drv_saadc_task_address_get can be used to get the task address. Single sampling triggers conversion on all initialized channels. When the requested buffer is filled with samples, an event of type NRF_DRV_SAADC_EVT_DONE is generated.

Example for sampling triggered by the CPU:

```
err_code = nrf_drv_saadc_buffer_convert(buffer, 1);
//Check error.
err_code = nrf_drv_saadc_sample();
//Check error.
//Event handler is called immediately after conversion is finished.
```

Example for setting up sampling by PPI:

```
err_code = nrf_drv_saadc_buffer_convert(buffer, length);
//Check error.
uint32_t sample_task = nrf_drv_saadc_task_address_get
       (NRF_SAADC_TASK_SAMPLE);
//Set task end point in PPI using sample_task variable.
```

The driver supports double buffering, which means that nrf_drv_saadc_buffer_convert can be called twice and the buffer that is provided in the second call will be used immediately after the first one is filled.

Continuous conversion can be achieved by setting up two buffers and calling nrf_drv_saadc_buffer_convert again in the event handler to switch between them later.

Example for setting up two buffers:

```
err_code = nrf_drv_saadc_buffer_convert(buffer1, length1);
//Check error.
err_code = nrf_drv_saadc_buffer_convert(buffer2, length2);
//Check error.
```

Example for setting up a completed buffer again in the event handler:

```
void event_handler(nrf_drv_saadc_evt_t * p_event)
{
    if (p_event->type == NRF_DRV_SAADC_EVT_DONE)
    {
        err_code = nrf_drv_saadc_buffer_convert
      (p_event->data.done.p_buffer, SAMPLES_IN_BUFFER);
        //Check error.
    }
}
```

To configure the SAADC to continue conversion to the second buffer, SAADC interrupts must be handled. However, if only one channel is used, EasyDMA could automatically start processing using the second buffer if PPI is used to bind SAADC END events with the SAADC START task. This is not done by the driver because it is only controlling the SAADC peripheral, but it can be done in the application code.

Limits

The driver can generate events of type NRF_DRV_SAADC_EVT_LIMIT if the converted sample on a given channel is exceeding a limit. You can set two limits per channel: high and low. The function nrf_drv_saadc_limit_set can be used to configure these limits. The function sets both limits; if only one limit should be enabled, the second can be disabled by using NRF_DRV_SAADC_LIMITH_DISABLED for the high limit or NRF_DRV_SAADC_LIMITL_DISABLED for the low limit.

The limit event will be generated by every sample that exceeds the limit. If this is not desired, it can be disabled in the event handler.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.

SPI master

The SPI master driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the SPI and SPIM peripherals. See the API documentation for the SPI HAL and SPIM HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the SPI master driver for details.

Key features include:

- Multi-instance support.
- Common API for two types of peripherals: legacy peripherals (SPI) and peripherals using EasyDMA (SPIM). The type of peripheral that is used can be configured independently for each driver instance in `nrf_drv_config.h`.
- Two modes of data transfer: blocking and non-blocking (with end notification done via an event handler that is provided during initialization).
- EasyDMA and legacy support (SPIM and SPI).
- Supports triggering transfer from PPI (SPIM only).
- Supports post-incrementation of buffer addresses and repeated transfers (SPIM only).

Note that peripherals using EasyDMA can work only with buffers that are placed in the Data RAM region. Under certain circumstances, compilers might choose to use a different region for data placement and, for example, place a constant buffer in the code FLASH. In such a case, the SPIM peripheral cannot be used to transfer data from the buffer.

To quickly get started, use the sample code from the SPI Master Example.

Driver configuration

The SPI master driver can use multiple instances of the SPI/SPIM peripherals, and it provides a common API for both peripheral types. The instances of the peripherals that are to be assigned to the driver must be selected statically in `nrf_drv_config.h`. In the same way, you configure whether a given instance uses EasyDMA (thus whether SPIM or SPI is used).

For example:

```
#define SPI0_ENABLED 1
#define SPI0_USE_EASY_DMA 0
```

For enabled instances, you must provide the default configuration settings that are used by the NRF_DRV_SPI_DEFAULT_CONFIG macro. These settings are also defined in `nrf_drv_config.h`, for example:

```
#define SPI0_CONFIG_SCK_PIN          2
#define SPI0_CONFIG_MOSI_PIN         3
#define SPI0_CONFIG_MISO_PIN         4
#define SPI0_CONFIG_IRQ_PRIORITY     APP_IRQ_PRIORITY_LOW
```

Basic usage

Call the NRF_DRV_SPI_INSTANCE macro with the ID of the peripheral instance that you want to use to create a driver instance. For example, this code will create a driver instance that uses the SPI0 or SPIM0 peripheral (depending on the value assigned to SPI0_CONFIG_USE_EASY_DMA in `nrf_drv_config.h`):

```
static const nrf_drv_spi_t m_spi_master_0 = NRF_DRV_SPI_INSTANCE(0);
```

Next, call nrf_drv_spi_init to initialize and configure the instance. See the nrf_drv_spi_config_t structure documentation for possible configuration options.

For example:

```
uint32_t err_code;

nrf_drv_spi_config_t config = NRF_DRV_SPI_DEFAULT_CONFIG(0);
config.frequency = NRF_DRV_SPI_FREQ_1M;
config.mode      = NRF_DRV_SPI_MODE_3;
config.bit_order = NRF_DRV_SPI_BIT_ORDER_LSB_FIRST;

err_code = nrf_drv_spi_init(&m_spi_master_0, &config, NULL);
if (err_code != NRF_SUCCESS)
{
    // Initialization failed. Take recovery action.
}
```

After successful initialization, you can call nrf_drv_spi_transfer to request SPI transfers.

The SPI master driver can automatically control the Slave Select signal (see nrf_drv_spi_config_t::ss_pin). This feature can only be used if there is exactly one slave. If there are multiple slaves, the Slave Select signal must be controlled externally.

When the SPI master driver instance is no longer needed or its configuration must be changed, call nrf_drv_spi_uninit.

Advanced usage

In non-blocking mode, you can use the nrf_drv_spi_xfer function that supports complex transfers. As parameters, nrf_drv_spi_xfer takes a pointer to a transfer descriptor (nrf_drv_spi_xfer_desc_t) and flags that enable more features.

The following sections show complex scenarios that use different transfer types. These scenarios are not supported with SPI, but require SPIM.

Starting a transfer from PPI

You can use nrf_drv_spi_xfer to set up a transfer and start it synchronously using PPI. To do so, specify the NRF_DRV_SPI_FLAG_HOLD_XFER flag and use nrf_drv_spi_start_task_get to get the address of the task that starts the transfer.

The following example shows how to configure the driver to setup a transfer without starting it:

```
nrf_drv_spi_xfer_desc_t xfer = NRF_DRV_SPI_XFER_TRX(p_tx_data,
      tx_length, p_rx_data, rx_length);
ret_code_t ret = nrf_drv_spi_xfer(&spi, &xfer,
      NRF_DRV_SPI_FLAG_HOLD_XFER);
// SPIM is now configured and ready to be started.
if (ret == NRF_SUCCESS)
{
  uint32_t start_tsk_addr = nrf_drv_spi_start_task_get(&spi);
  // Set up PPI to trigger the transfer.
}
```

Repeated transfers

You can configure the SPI driver to perform a certain amount of PPI-triggered transfers. To do so, specify the NRF_DRV_SPI_FLAG_REPEATED_XFER flag. The driver will not put the instance into busy state in this case, because in this mode, you control when the sequence of transfers is complete. Use the END event to count the number of completed transfers. To get the address of the END event, use nrf_drv_spi_end_event_get.

Specify the NRF_DRV_SPI_FLAG_TX_POSTINC and NRF_DRV_SPI_FLAG_RX_POSTINC flags to enable post-incrementation of buffer addresses.

Additionally, you can specify the NRF_DRV_SPI_FLAG_NO_XFER_EVT_HANDLER flag to disable calling the user event handler after each transfer completion. If you specify this flag, transfer completion interrupts are disabled if EasyDMA and SPIM is used.

The following example shows how to set up a sequence of SPI transfers that will be triggered using PPI (for example, using RTC or TIMER compare events). A single transfer consists of a transmission of 1 byte and a reception of 3 bytes. The TX data is repeated for all transfers, but the RX data from all transfers is collected in the buffer. Therefore, NRF_DRV_SPI_FLAG_RX_POSTINC is set so that the buffer address for the received data is incremented. The end of sequence is controlled externally, and the processing of data is postponed until the end of the sequence (the user event handler is disabled).

```
nrf_drv_spi_xfer_desc_t xfer = NRF_DRV_SPI_XFER_TRX(p_tx_data, 1,
       p_rx_data, 3);
uint32_t flags = NRF_DRV_SPI_FLAG_HOLD_XFER           |
                 NRF_DRV_SPI_FLAG_RX_POSTINC          |
                 NRF_DRV_SPI_FLAG_NO_XFER_EVT_HANDLER |
                 NRF_DRV_SPI_FLAG_REPEATED_XFER;

ret_code_t ret = nrf_drv_spi_xfer(&spi, &xfer, flags);
// SPIM is now configured and ready to be started.
if (ret == NRF_SUCCESS)
{
  uint32_t start_tsk_addr = nrf_drv_spi_start_task_get(&spi);
  // Set up PPI to trigger the transfer.
  uint32_t end_evt_addr = nrf_drv_spi_end_event_get(&spi);
  // Set up PPI to count the number of transfers.
}
```

In this example, there is no interrupt from SPI unless an error condition is detected.

Events

When a transfer is complete, which means that the requested amount of data was transferred or an error condition was detected, the driver generates an event (unless this was suppressed with the NRF_DRV_SPI_FLAG_NO_XFER_EVT_HANDLER flag). This event contains the transfer descriptor.

It is possible to start another transfer from the event handler context.

There is no context blocking in the interrupt handler. Therefore, it is assumed that the driver API will not be called from a context with higher priority than the instance interrupt, because this might cause driver failure.

SPI slave

The SPI slave driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the SPIS peripheral. See the API documentation for the SPIS HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the SPI slave driver for details.

Key features include:

- Multi-instance support
- SPI slave EasyDMA functionality for reading from and writing to the RAM

Note that peripherals using EasyDMA can work only with buffers that are placed in the data RAM region. Under certain circumstances, compilers might choose to use a different region for data placement and, for example, place a constant buffer in the code FLASH. In such a case, the SPIS peripheral cannot be used to transfer data from the buffer.

To quickly get started, use the sample code from the SPI Slave Example.

Driver configuration

The SPI slave driver can use multiple instances of SPIS. The instances of the peripherals that are to be assigned to the driver must be selected statically in `nrf_drv_config.h`, for example:

```
#define SPIS1_ENABLED 1
```

For enabled instances, you must provide the default configuration settings that are used by the NRF_DRV_SPIS_DEFAULT_CONFIG macro. These settings are also defined in `nrf_drv_config.h`, for example:

```
#define SPIS1_CONFIG_SCK_PIN          2
#define SPIS1_CONFIG_MOSI_PIN         3
#define SPIS1_CONFIG_MISO_PIN         4
#define SPIS1_CONFIG_IRQ_PRIORITY     APP_IRQ_PRIORITY_LOW
```

Using the SPI slave driver

Call the NRF_DRV_SPIS_INSTANCE macro with the ID of the peripheral instance that you want to use to create a driver instance. For example, this code will create a driver instance that uses the SPIS1 peripheral:

```
static const nrf_drv_spis_t m_spi_slave_1 = NRF_DRV_SPI_INSTANCE(1);
```

Next, call nrf_drv_spis_init to initialize and configure the instance. See the nrf_drv_spis_config_t structure documentation for possible configuration options. To receive data, the SPI buffers must be set by calling nrf_drv_spis_buffers_set after initialization.

For example:

```
static volatile uint8_t m_slave_tx_buffer[8];
static volatile uint8_t m_slave_rx_buffer[8];

static void spi_slave_handler(nrf_drv_spis_event_t event)
{
    if (event.evt_type == NRF_DRV_SPIS_XFER_DONE)
    {
        // Received bytes: event.rx_amount.
    }
}

    /* ... */

    uint32_t err_code;

    nrf_drv_spis_config_t config = NRF_DRV_SPIS_DEFAULT_CONFIG(1);
    config.mode      = NRF_DRV_SPIS_MODE_3;
    config.bit_order = NRF_DRV_SPIS_BIT_ORDER_LSB_FIRST;

    err_code = nrf_drv_spis_init(&m_spi_slave_1, &config,
      spi_slave_handler);
    if (err_code != NRF_SUCCESS)
    {
        // Initialization failed. Take recovery action.
    }

    err_code = nrf_drv_spis_buffers_set(mp_spis,
                                        (uint8_t*) m_slave_tx_buffer,
      sizeof(m_slave_tx_buffer),
                                        (uint8_t*) m_slave_rx_buffer,
      sizeof(m_slave_rx_buffer));
    if (err_code != NRF_SUCCESS)
    {
        // Buffer setup failed. Take recovery action.
    }
```

The callback handler function will be called after every successful buffer setup and every completed transfer. New buffers must be set by calling nrf_drv_spis_buffers_set after every finished transaction. Otherwise, the transaction is ignored, and the default character is clocked out.

When the SPI slave driver instance is no longer needed or its configuration must be changed, call nrf_drv_spis_uninit.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
14. nRF5 SDK v11.0.0-2.alpha

SWI

The SWI driver provides functions to manage software interrupts (SWI). It allows users to allocate SWIs and pass extra flags to interrupt handler functions.

Usage

Software interrupts are allocated from the available SWI pool. By default, there are six interrupts available: SWI0 to SWI5. Note that this number is decreased if you use a SoftDevice or the Gazell or ESB protocol. For more information about resource usage for a specific protocol library, see the appropriate specification.

Initialization

You must initialize the driver by calling the nrf_drv_swi_init function. If initialization is successful, the function returns NRF_SUCCESS.

Allocation

To allocate an interrupt, call the nrf_drv_swi_alloc function. As input, you must provide a pointer to the nrf_swi_t variable, an event handler function, and a value that indicates the interrupt priority. If allocation is successful, the function returns NRF_SUCCESS, and the provided pointer variable is set.

Event handler and flags

When allocating the driver, you must provide a callback function (see nrf_swi_handler_t). The arguments for this function are the SWI and some user flags. The user flags are represented as bit mask.

To trigger the interrupt, call nrf_drv_swi_trigger with a specific flag number. When an SWI is triggered, the corresponding callback function is called with all specified flags (represented as a bit mask). The flags can be used, for example, for interrupt source tracking, or they can be ignored if not needed. Note that the callback function might be invoked with more than one flag set if the SWI is triggered more than once with different flag numbers from a same or higher priority interrupt handler.

Special options

To disable specific SWIs from the pool, add a global define during compilation (SWI_DISABLE0 to SWI_DISABLE5). Any disabled SWI will not be allocated and will therefore not be available to the user.

Example

See the following code for a usage example. Note that this example is not power optimized.

```
static nrf_swi_t my_swi;

void swi_event_handler(nrf_swi_t swi, nrf_swi_flags_t flags)
{
    /* If "my_swi" was triggered and flag #3 is present... */
    if ((swi == my_swi) && (flags & (1 << 3)))
    {
        /* Do something. */
    }
    return;
}

/*
    ....
 */

ret_code_t err_code;
nrf_swi_t my_swi;

/* Initialize the library. */
err_code = nrf_drv_swi_init();
APP_ERROR_CHECK(err_code);

/* Allocate and setup one SWI. */
err_code = nrf_drv_swi_alloc(&my_swi, swi_event_handler,
      APP_IRQ_PRIORITY_LOW);
APP_ERROR_CHECK(err_code);

while (true)
{
    /* Trigger SWI with flag #3. */
    nrf_drv_swi_trigger(my_swi, 3);

    /* Wait for a second. */
    nrf_delay_ms(1000);
}
```

Usage with a SoftDevice

The library can be used with a SoftDevice. However, the amount of interrupts that are available to the user will be limited in this case. See the SoftDevice specification for more information about the resource usage.

Be careful when specifying APP_IRQ_PRIORITY_HIGH as SWI priority. Long interrupt routines with high priority might affect the proper operation of the SoftDevice.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
15. nRF5 SDK v11.0.0-2.alpha

Timer

The timer driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the timer. All functions in this layer are implemented as inline functions. See the API documentation for the Timer HAL for details.

The driver layer provides support for a multi-instance timer and APIs on a higher level than the HAL. See the API documentation for the Timer driver for details.

The Timer Example provides sample code that you can use to quickly get started.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
16. nRF5 SDK v11.0.0-2.alpha

TWI master

The TWI driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the TWI registers. See the API documentation for the TWI HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the TWI master driver for details.

Key features include:

- Repeated start
- No multi-master
- Only 7-bit addressing
- Supports clock stretching
- Includes blocking and non-blocking mode
- EasyDMA and legacy support (TWIM and TWI)
- TXRX and TXTX transfers (non-blocking mode only)
- Multi-instance
- Supports triggering transfer from PPI (TWIM only)
- Supports post-incrementation of buffer addresses and repeated transfers (TWIM only)

Initialization

Before using the instance of the driver, it must be initialized with the nrf_drv_twi_init function. If no event handler is provided, the instance is configured to use blocking mode. After initialization, the instance must be enabled with nrf_drv_twi_enable.

The instance can be disabled using nrf_drv_twi_disable and uninitialized using nrf_drv_twi_uninit.

Basic usage

The TWI master driver supports two modes of operation:

- In **blocking mode**, the functions nrf_drv_twi_rx and nrf_drv_twi_tx return if the requested transfer is complete, or if an error was reported by the peripheral. In blocking mode, the driver does not use the interrupt; there is no context switch in the function.
- In **non-blocking mode**, the functions nrf_drv_twi_tx, nrf_drv_twi_rx, and nrf_drv_twi_xfer return with NRF_SUCCESS immediately after the transfer is set up or with NRF_ERROR_BUSY if the driver is busy (because a previous transfer is not completed). The event handler is called from the TWI interrupt context when the transfer is complete (if enabled) or if an error occurs.

The following code example for basic usage in blocking mode shows a simple write sequence:

```
uint32_t err_code;
uint8_t tx_data[] = {'a', 'b', 'c', 'd', 'e'};
const nrf_drv_twi_t twi = NRF_DRV_TWI_INSTANCE(0);

err_code = nrf_drv_twi_init(&twi, NULL, NULL);
APP_ERROR_CHECK(err_code);

nrf_drv_twi_enable(&twi);

err_code = nrf_drv_twi_tx(&twi, SLAVE_ADDRESS, tx_data, sizeof
       (tx_data), false);
APP_ERROR_CHECK(err_code);
```

Advanced usage

In non-blocking mode, you can use the nrf_drv_twi_xfer function that supports complex transfers. As parameters, nrf_drv_twi_xfer takes the driver instance, a pointer to a transfer descriptor (nrf_drv_twi_xfer_desc_t), and flags that enable more features.

Note that TWI supports only the NRF_DRV_TWI_FLAG_TX_NO_STOP flag. All other flags require TWIM.

nrf_drv_twi_xfer_desc_t::type specifies the type of the transfer. The following transfer types are supported:

Transfer type	Description

NRF_DRV_TWI_XFER_TX	TX transfer. By default, this transfer ends with a stop condition. Specify NRF_DRV_TWI_FLAG_TX_NO_STOP to suppress the stop condition. Use the macro NRF_DRV_TWI_XFER_DESC_TX to create a transfer descriptor with this type.
NRF_DRV_TWI_XFER_RX	RX transfer with a stop condition at the end of the transfer. Use the macro NRF_DRV_TWI_XFER_DESC_RX to create a transfer descriptor with this type.
NRF_DRV_TWI_XFER_TXRX	TX transfer followed by an RX transfer with a repeated start condition. The transfer ends with a stop condition. Note that there is no interrupt after the TX transfer is completed (TWIM only). Use the macro NRF_DRV_TWI_XFER_DESC_TXRX to create a transfer descriptor with this type.
NRF_DRV_TWI_XFER_TXTX	TX transfer followed by a TX transfer with a repeated start condition. By default, the transfer ends with a stop condition. Specify NRF_DRV_TWI_FLAG_TX_NO_STOP to suppress the stop condition. Use the macro NRF_DRV_TWI_XFER_DESC_TXTX to create a transfer descriptor with this type.

The following sections show complex scenarios that use different transfer types. These scenarios are not supported with TWI, but require TWIM.

Starting a transfer from PPI

You can use nrf_drv_twi_xfer to set up a transfer and start it synchronously using PPI. To do so, specify the NRF_DRV_TWI_FLAG_HOLD_XFER flag and use nrf_drv_twi_start_task_get to get the address of the task that starts the transfer.

The following example shows how to configure the driver to setup a TX transfer without starting it:

```
nrf_drv_twi_xfer_desc_t xfer = NRF_DRV_TWI_XFER_DESC_TX(addr, p_data,
       length);
ret_code_t ret = nrf_drv_twi_xfer(&twi, &xfer,
       NRF_DRV_TWI_FLAG_HOLD_XFER);
// TWIM is now configured and ready to be started.
if (ret == NRF_SUCCESS)
{
  uint32_t start_tsk_addr = nrf_drv_twi_start_task_get(&twi,
       xfer.type);
  // Set up PPI to trigger the transfer.
}
```

Repeated transfers

You can configure the TWI driver to perform a certain amount of PPI-triggered transfers. To do so, specify the NRF_DRV_TWI_FLAG_REPEATED_XFER flag. The driver will not put the instance into busy state in this case, because in this mode, you control when the sequence of transfers is

complete. Use the STOPPED event to count the number of completed transfers. To get the address of the STOPPED event, use nrf_drv_twi_stopped_event_get.

Specify the NRF_DRV_TWI_FLAG_TX_POSTINC and NRF_DRV_TWI_FLAG_RX_POSTINC flags to enable post-incrementation of buffer addresses.

Additionally, you can specify the NRF_DRV_TWI_FLAG_NO_XFER_EVT_HANDLER flag to disable calling the user event handler after each transfer completion. If you specify this flag, transfer completion interrupts are disabled if EasyDMA and TWIM is used, and the user event handler will be called only if an ERROR interrupt is triggered.

The following example shows how to set up a sequence of NRF_DRV_TWI_XFER_TXRX transfers that will be triggered using PPI (for example, using RTC or TIMER compare events). A single transfer consists of a transmission of 1 byte and a reception of 3 bytes. The TX data is repeated for all transfers, but the RX data from all transfers is collected in the buffer. Therefore, NRF_DRV_TWI_FLAG_RX_POSTINC is set so that the buffer address for the received data is incremented. The end of sequence is controlled externally, and the processing of data is postponed until the end of the sequence (the user event handler is disabled).

```
nrf_drv_twi_xfer_desc_t xfer = NRF_DRV_TWI_XFER_DESC_TXRX(addr,
       p_tx_data, 1, p_rx_data, 3);
uint32_t flags = NRF_DRV_TWI_FLAG_HOLD_XFER           |
                 NRF_DRV_TWI_FLAG_RX_POSTINC          |
                 NRF_DRV_TWI_FLAG_NO_XFER_EVT_HANDLER |
                 NRF_DRV_TWI_FLAG_REPEATED_XFER;

ret_code_t ret = nrf_drv_twi_xfer(&twi, &xfer, flags);
// TWIM is now configured and ready to be started.
if (ret == NRF_SUCCESS)
{
  uint32_t start_tsk_addr = nrf_drv_twi_start_task_get(&twi,
      xfer.type);
  // Set up PPI to trigger the transfer.
  uint32_t stopped_evt_addr = nrf_drv_twi_stopped_event_get(&twi);
  // Set up PPI to count the number of transfers.
}
```

In this example, there is no interrupt from TWI unless an error condition is detected.

Events

When a transfer is complete, which means that the requested amount of data was transferred or an error condition was detected, the driver generates an event (unless this was suppressed with the NRF_DRV_TWI_FLAG_NO_XFER_EVT_HANDLER flag). This event contains the transfer type and the transfer descriptor. If an error occurred, the length field is set to the amount of bytes that were transferred before the error occurred.

It is possible to start another transfer from the event handler context.

The functions nrf_drv_twi_rx, nrf_drv_twi_tx, and nrf_drv_twi_xfer internally disable all TWI/TWIM interrupts when setting up transfers, and then re-enable them before leaving the function. There is no context blocking in the interrupt handler. Therefore, it is assumed that the driver API will not be called from a context with higher priority than the instance interrupt, because this might cause driver failure.

The following code shows an event handler that uses TX RX with a repeated start:

```
void twi_event_handler(nrf_drv_twi_evt_t * p_event, void * p_context)
{
    if ((p_event->type == NRF_DRV_TWI_EVT_DONE) &&
    (p_event->xfer_desc.type == NRF_DRV_TWI_XFER_TXRX))
    { xfer_completed = true; }
}
```

The following code shows the transfer starting:

```
nrf_drv_twi_xfer_desc_t xfer = NRF_DRV_TWI_XFER_DESC_TXRX(addr,
      p_tx_buffer, tx_length, p_rx_buffer, rx_length);
ret_code_t ret = nrf_drv_twi_xfer(&twi, &xfer, 0);
if (ret == NRF_SUCCESS)
{
  while(xfer_completed == false){}
}
```

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.

17. nRF5 SDK v11.0.0-2.alpha

TWI slave

*This information applies to the **nRF52 Series** only.*

The two-wire interface slave with EasyDMA (TWIS) driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the TWI. See the API documentation for the TWIS HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the TWI slave with EasyDMA driver for details.

Key features include:

- Simple events communication:

    - Data request
    - Transfer pending
    - Transfer finished
    - Error
- Two 7-bit addresses for listening
- Supports clock stretching

Driver configuration

The following configuration options are available in `nrf_drv_config.h`:

- TWIS_ASSUME_INIT_AFTER_RESET_ONLY: Set this option for better performance if you activate TWI only once.
- TWIS_NO_SYNC_MODE: Set this option to always use Asynchronous mode.

Modes of operation

The TWIS driver supports two different modes: synchronous mode and asynchronous mode.

Synchronous mode

In synchronous mode, the driver works completely without interrupts. To configure the driver to use synchronous mode, pass a NULL pointer as the pointer to the event handler during driver initialization.

In synchronous mode, events are pooled every time a state function is called. Pooling and state machine processing can be entered only once at the same time. Therefore, if a function is called from the main thread and, while it is processed, an interrupt occurs that tries to process a TWIS event again, the checking function in the interrupt is skipped.

The TWIS driver is a slave driver. Therefore, the reasons for configuring this driver to use synchronous mode are different than for a master driver. For a slave driver, it does not make sense to support sending data from the main thread and from interrupts at the same time. State testing functions like nrf_drv_twis_is_busy are not guaranteed to process the current state in the interrupts if the same functions are processed in the main thread. Interrupts should not wait for a state change. Otherwise, they might be locked.

The following example code shows how to use synchronous mode:

```
nrf_drv_twis_t instance = NRF_DRV_TWIS_INSTANCE(0);

int main(void)
{
    // Initialization
    bool waitingRx = false;
    nrf_drv_twis_init(&instance, NULL, NULL);
    nrf_drv_twis_enable(&instance);

    // Main loop
    while(1)
```

```
    {
        // It is important to process it first
        if(waitingRx && !nrf_drv_twis_is_pending_rx(&instance))
        {
            waitingRx = false;
            // Check of errors
            uint32_t err = nrf_drv_twis_error_get(&instance);
            if(err)
            {
                // ... process errors ....
            }
            else
            {
                // ... process received data ...
            }
        }
        if(nrf_drv_twis_is_waiting_tx_buff(&instance))
        {
            // Prepare TX buffer (response for READ command)
            nrf_drv_twis_tx_prepare(&instance, txbuffer, sizeof
      (txbuffer));
        }
        if(nrf_drv_twis_is_waiting_rx_buff(&instance))
        {
            // Prepare RX buffer (response for WRITE command)
            nrf_drv_twis_rx_prepare(&instance, rxbuffer, sizeof
      (rxbuffer));
            waitingRx = true;
        }

        // ... The rest of the main lop tasks ...
    }
}
```

Asynchronous mode

In asynchronous mode, all events are processed in internal interrupt service runtime. When anything requires processing, the event handler is called. See nrf_drv_twis_evt_type_t for a list of defined events.

The event handler is always called in the following pattern: REQ -> DONE or ERROR. This means that if we get a TWIS_EVT_READ_REQ message, the answer will always be TWIS_EVT_READ_DONE or TWIS_EVT_READ_ERROR. The answer will be sent even if there is no STOP condition on the bus, but reading finishes for example by a repeated START condition.

There are different ways to prepare buffers for receiving or sending data:

- Prepare the buffer when required inside the event handler.

- Prepare the buffer outside of the event handler by using a flag set in the event handler or by checking if the buffer is required by nrf_drv_twis_is_waiting_rx_buff or nrf_drv_twis_is_waiting_tx_buff.
- Prepare the buffer before it is requested. If the buffer is not required, the nrf_drv_twis_evt_t::buf_req field is cleared, and the buffer can be set for the next transaction in that event.

The following pseudocode shows how to implement an event handler:

```
void twis_event_handler(nrf_drv_twis_evt_t const * const p_event)
{
    switch(p_event->type)
    {
    case TWIS_EVT_READ_REQ:
        if(p_event->data.buf_req)
        {
            // Function size_t get_buffer_to_send(char const * * const
      rptr)
            // needs to be implemented
            const char * rptr;
            size_t size = get_buffer_to_send(&rptr);
            nrf_drv_twis_tx_prepare(&m_slave_inst, rptr, size);
        }
        break;
    case TWIS_EVT_READ_DONE:
        break;
    case TWIS_EVT_WRITE_REQ:
        if(p_event->data.buf_req)
        {
            // Function size_t get_buffer_to_write(char * * const wptr)
            // needs to be implemented
            char *wptr;
            size_t size = get_buffer_to_write(&wptr);
            nrf_drv_twis_rx_prepare(&m_slave_inst, wptr, size);
        }
        break;
    case TWIS_EVT_WRITE_DONE:
        // Function void mark_buffer_ready(size_t amount)
        // needs to be implemented
        mark_buffer_ready(p_event->data.rx_amount);
        break;
    default:
        break;
    }
}
```

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.
18. nRF5 SDK v11.0.0-2.alpha

UART

The UART driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the UART and UARTE peripherals. See the API documentation for the UART HAL and UARTE HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the UART driver for details.

Key features include:

- Common API for two types of peripherals: legacy peripherals (UART) and peripherals using EasyDMA (UARTE). The type of peripheral that is used can be configured either during build time or at runtime during initialization.
- Two modes of data transfer: blocking and non-blocking (with end notification done via an event handler that is provided during initialization).
- Driver supports aborting ongoing RX or TX transfers.
- Supported modes: with and without hardware flow control and with and without parity bit.
- Double RX buffering allowing to receive data continuously.

Note that peripherals using EasyDMA can work only with buffers that are placed in the Data RAM region. Under certain circumstances, compilers might choose to use a different region for data placement and, for example, place a constant buffer in the code FLASH. In such a case, the UARTE peripheral cannot be used to transfer data from the buffer.

Driver configuration

The UART default configuration is located in `nrf_drv_config.h`. If UARTE is present on the chip, the driver can be configured at runtime to support UART mode, UARTE mode, or both. The following example shows how to configure the driver to support both modes in runtime and have legacy mode as the default:

```
#ifdef NRF52
#define UART0_CONFIG_USE_EASY_DMA false
#define UART_EASY_DMA_SUPPORT     1
#define UART_LEGACY_SUPPORT       1
#endif //NRF52
```

If only one mode is used in the application, disable the second mode in `nrf_drv_config.h` to achieve a lower memory footprint and better performance.

Call nrf_drv_uart_init with the p_config argument set to NULL to use the default configuration. To use a custom configuration, provide a user configuration structure, for example:

```
nrf_drv_uart_config_t config = NRF_DRV_UART_DEFAULT_CONFIG;
config.use_easy_dma = true;
ret_code = nrf_drv_uart_init(&config, NULL);
```

Using the UART driver

Blocking mode

If no event handler is provided during the initialization of the driver, the driver will operate in blocking mode. In this case, nrf_drv_uart_tx and nrf_drv_uart_rx will not return until the requested transfer is completed, nrf_drv_uart_rx_abort or nrf_drv_uart_tx_abort is called from a different context, or an error is detected. When an abort function is called, nrf_drv_uart_tx or nrf_drv_uart_rx returns with an error code. When an error is reported by the peripheral,nrf_drv_uart_rx returns with an error code.

In blocking mode, the driver does not use a peripheral interrupt, and there is no context switching inside the driver.

Non-blocking mode

If an event handler (nrf_uart_event_handler_t) is provided during the initialization of the driver, the driver will operate in non-blocking mode. In this case, nrf_drv_uart_tx and nrf_drv_uart_rx will return immediately after transfer is started. Completion will be notified to the user by the event handler that was provided during initialization. The event handler is called in the context of the UART/UARTE interrupt. If an error is reported by the peripheral, it is also reported by the event handler. The error event contains information about the amount of data that was transferred before the error occurred.

If nrf_drv_uart_rx_abort or nrf_drv_uart_tx_abort is called, the event handler notifies that the transfer is completed, but the length field is set to the amount of data that was transferred before the abort function was called.

Enabling RX without providing data buffer

You can enable RX without providing a buffer for incoming data. In this case, incoming data (up to 6 bytes) stays in the hardware FIFO. If more than 6 bytes are received, an overrun error condition occurs. This condition will be reported by the event handler or at the next nrf_drv_uart_rx call.

Note
This functionality is provided only for UART mode.

```
ret_code = nrf_drv_uart_init(NULL, NULL);
nrf_drv_uart_rx_enable();
//UART RX is enabled
nrf_drv_uart_rx(buffer,2);
```

```
//UART RX is still enabled.
nrf_drv_uart_rx_disable();
```

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.

WDT

The watchdog timer (WDT) driver includes two layers: the hardware access layer (HAL) and the driver layer (DRV).

The hardware access layer provides basic APIs for accessing the registers of the WDT. All functions in this layer are implemented as inline functions. See the API documentation for the WDT HAL for details.

The driver layer provides APIs on a higher level than the HAL. See the API documentation for the WDT driver for details.

The WDT Example provides sample code that you can use to quickly get started.

This document was last updated on Fri Dec 18 2015.
Please send us your feedback about the documentation! For technical questions, visit the Nordic Developer Zone.