This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Analog-to-Digital Converter functionality, for the conversion of analog voltages into a corresponding digital form. The following driver Application Programming Interface (API) modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- ADC (Analog-to-Digital Converter)

The following devices can use this module:

- Atmel | SMART SAM L21/L22
- Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information
- Examples
- API Overview
# Table of Contents

Introduction ...................................................................................................................... 1

1. Software License ....................................................................................................... 3

2. Prerequisites .............................................................................................................. 4

3. Module Overview ....................................................................................................... 5
   3.1. Sample Clock Prescaler ............................................................................................... 5
   3.2. ADC Resolution ........................................................................................................ 5
   3.3. Conversion Modes .................................................................................................... 6
   3.4. Differential and Single-ended Conversion ............................................................... 6
   3.5. Sample Time ............................................................................................................. 6
   3.6. Averaging .................................................................................................................. 6
   3.7. Offset and Gain Correction ....................................................................................... 7
   3.8. Pin Scan .................................................................................................................... 8
   3.9. Window Monitor ....................................................................................................... 8
   3.10. Events .................................................................................................................... 8

4. Special Considerations .............................................................................................. 9

5. Extra Information ..................................................................................................... 10

6. Examples ................................................................................................................... 11

7. API Overview .......................................................................................................... 12
   7.1. Variable and Type Definitions .................................................................................. 12
   7.2. Structure Definitions .............................................................................................. 12
   7.3. Macro Definitions ................................................................................................... 14
   7.4. Function Definitions ............................................................................................... 15
   7.5. Enumeration Definitions ...................................................................................... 26

8. Extra Information for ADC Driver .......................................................................... 33
   8.1. Acronyms ................................................................................................................. 33
   8.2. Dependencies .......................................................................................................... 33
   8.3. Errata ...................................................................................................................... 33
   8.4. Module History ....................................................................................................... 33

9. Examples for ADC Driver ....................................................................................... 34
   9.1. Quick Start Guide for ADC - Basic ......................................................................... 34
   9.2. Quick Start Guide for ADC - Callback .................................................................... 36
   9.3. Quick Start Guide for Using DMA with ADC/DAC .............................................. 39

10. Document Revision History ................................................................................... 47
1. Software License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of Atmel may not be used to endorse or promote products derived from this software without specific prior written permission.

4. This software may only be redistributed and used in connection with an Atmel microcontroller product.

THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
2. Prerequisites

There are no prerequisites for this module.
3. **Module Overview**

This driver provides an interface for the Analog-to-Digital conversion functions on the device, to convert analog voltages to a corresponding digital value. The ADC has up to 12-bit resolution, and is capable of converting up to 1,000,000 samples per second (MSPS).

The ADC has a compare function for accurate monitoring of user defined thresholds with minimum software intervention required. The ADC may be configured for 8-, 10-, or 12-bit result, reducing the conversion time. ADC conversion results are provided left or right adjusted which eases calculation when the result is represented as a signed integer.

The input selection is flexible, and both single-ended and differential measurements can be made. For differential measurements, an optional gain stage is available to increase the dynamic range. In addition, several internal signal inputs are available. The ADC can provide both signed and unsigned results.

The ADC measurements can either be started by application software or an incoming event from another peripheral in the device, and both internal and external reference voltages can be selected.

**Note:** Internal references will be enabled by the driver, but not disabled. Any reference not used by the application should be disabled by the application.

A simplified block diagram of the ADC can be seen in **Figure 3-1 Module Overview** on page 5.

![Figure 3-1. Module Overview](image)

3.1. **Sample Clock Prescaler**

The ADC features a prescaler, which enables conversion at lower clock rates than the input Generic Clock to the ADC module. This feature can be used to lower the synchronization time of the digital interface to the ADC module via a high speed Generic Clock frequency, while still allowing the ADC sampling rate to be reduced.

3.2. **ADC Resolution**

The ADC supports full 8-, 10-, or 12-bit resolution. Hardware oversampling and decimation can be used to increase the effective resolution at the expense of throughput. Using oversampling and decimation mode the ADC resolution is increased from 12-bit to an effective 13-, 14-, 15-, or 16-bit. In these modes the conversion rate is reduced, as a greater number of samples is used to achieve the increased
resolution. The available resolutions and effective conversion rate is listed in Table 3-1 Effective ADC Conversion Speed Using Oversampling on page 6.

Table 3-1. Effective ADC Conversion Speed Using Oversampling

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Effective conversion rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-bit</td>
<td>Conversion rate divided by 4</td>
</tr>
<tr>
<td>14-bit</td>
<td>Conversion rate divided by 16</td>
</tr>
<tr>
<td>15-bit</td>
<td>Conversion rate divided by 64</td>
</tr>
<tr>
<td>16-bit</td>
<td>Conversion rate divided by 256</td>
</tr>
</tbody>
</table>

3.3. Conversion Modes

ADC conversions can be software triggered on demand by the user application, if continuous sampling is not required. It is also possible to configure the ADC in free running mode, where new conversions are started as soon as the previous conversion is completed, or configure the ADC to scan across a number of input pins (see Pin Scan).

3.4. Differential and Single-ended Conversion

The ADC has two conversion modes; differential and single-ended. When measuring signals where the positive input pin is always at a higher voltage than the negative input pin, the single-ended conversion mode should be used in order to achieve a full 12-bit output resolution.

If however the positive input pin voltage may drop below the negative input pin the signed differential mode should be used.

3.5. Sample Time

The sample time for each ADC conversion is configurable as a number of half prescaled ADC clock cycles (depending on the prescaler value), allowing the user application to achieve faster or slower sampling depending on the source impedance of the ADC input channels. For applications with high impedance inputs the sample time can be increased to give the ADC an adequate time to sample and convert the input channel.

The resulting sampling time is given by the following equation:

\[ t_{SAMPLE} = (sample\_length + 1) \times \frac{ADC\_CLK}{2} \]

3.6. Averaging

The ADC can be configured to trade conversion speed for accuracy by averaging multiple samples in hardware. This feature is suitable when operating in noisy conditions.

You can specify any number of samples to accumulate (up to 1024) and the divide ratio to use (up to divide by 128). To modify these settings the ADC_RESOLUTION_CUSTOM needs to be set as the resolution. When this is set the number of samples to accumulate and the division ratio can be set by the
configuration struct members `adc_config::accumulate_samples` and `adc_config::divide_result`. When using this mode the ADC result register will be set to be 16-bit wide to accommodate the larger result sizes produced by the accumulator.

The effective ADC conversion rate will be reduced by a factor of the number of accumulated samples; however, the effective resolution will be increased according to Table 3-2 Effective ADC Resolution From Various Hardware Averaging Modes on page 7.

Table 3-2. Effective ADC Resolution From Various Hardware Averaging Modes

<table>
<thead>
<tr>
<th>Number of samples</th>
<th>Final result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12-bit</td>
</tr>
<tr>
<td>2</td>
<td>13-bit</td>
</tr>
<tr>
<td>4</td>
<td>14-bit</td>
</tr>
<tr>
<td>8</td>
<td>15-bit</td>
</tr>
<tr>
<td>16</td>
<td>16-bit</td>
</tr>
<tr>
<td>32</td>
<td>16-bit</td>
</tr>
<tr>
<td>64</td>
<td>16-bit</td>
</tr>
<tr>
<td>128</td>
<td>16-bit</td>
</tr>
<tr>
<td>256</td>
<td>16-bit</td>
</tr>
<tr>
<td>512</td>
<td>16-bit</td>
</tr>
<tr>
<td>1024</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

3.7. **Offset and Gain Correction**

Inherent gain and offset errors affect the absolute accuracy of the ADC.

The offset error is defined as the deviation of the ADC's actual transfer function from ideal straight line at zero input voltage.

The gain error is defined as the deviation of the last output step's midpoint from the ideal straight line, after compensating for offset error.

The offset correction value is subtracted from the converted data before the result is ready. The gain correction value is multiplied with the offset corrected value.

The equation for both offset and gain error compensation is shown below:

\[
ADC_{RESULT} = (VALUE_{CONV} + CORR_{OFFSET}) \times CORR_{GAIN}
\]

When enabled, a given set of offset and gain correction values can be applied to the sampled data in hardware, giving a corrected stream of sample data to the user application at the cost of an increased sample latency.

In single conversion, a latency of 13 ADC Generic Clock cycles is added for the final sample result availability. As the conversion time is always less than the propagation delay, in free running mode this
latency appears only during the first conversion. After the first conversion is complete, future conversion results are available at the defined sampling rate.

3.8. Pin Scan

In pin scan mode, the first ADC conversion will begin from the configured positive channel, plus the requested starting offset. When the first conversion is completed, the next conversion will start at the next positive input channel and so on, until all requested pins to scan have been sampled and converted. SAM L21/L22 has automatic sequences feature instead of pin scan mode. In automatic sequence mode, all of 32 positives inputs can be included in a sequence. The sequence starts from the lowest input, and go to the next enabled input automatically.

Pin scanning gives a simple mechanism to sample a large number of physical input channel samples, using a single physical ADC channel.

3.9. Window Monitor

The ADC module window monitor function can be used to automatically compare the conversion result against a preconfigured pair of upper and lower threshold values.

The threshold values are evaluated differently, depending on whether differential or single-ended mode is selected. In differential mode, the upper and lower thresholds are evaluated as signed values for the comparison, while in single-ended mode the comparisons are made as a set of unsigned values.

The significant bits of the lower window monitor threshold and upper window monitor threshold values are user-configurable, and follow the overall ADC sampling bit precision set when the ADC is configured by the user application. For example, only the eight lower bits of the window threshold values will be compared to the sampled data whilst the ADC is configured in 8-bit mode. In addition, if using differential mode, the 8th bit will be considered as the sign bit even if bit 9 is zero.

3.10. Events

Event generation and event actions are configurable in the ADC.

The ADC has two actions that can be triggered upon event reception:

- Start conversion
- Flush pipeline and start conversion

The ADC can generate two events:

- Window monitor
- Result ready

If the event actions are enabled in the configuration, any incoming event will trigger the action.

If the window monitor event is enabled, an event will be generated when the configured window condition is detected.

If the result ready event is enabled, an event will be generated when a conversion is completed.

Note: The connection of events between modules requires the use of the SAM Event System Driver (EVENTS) to route output event of one module to the input event of another. For more information on event routing, refer to the event driver documentation.
4. **Special Considerations**

An integrated analog temperature sensor is available for use with the ADC. The bandgap voltage, as well as the scaled I/O and core voltages can also be measured by the ADC. For internal ADC inputs, the internal source(s) may need to be manually enabled by the user application before they can be measured.
5. Extra Information

For extra information, see Extra Information for ADC Driver. This includes:

- Acronyms
- Dependencies
- Errata
- Module History
6. **Examples**

For a list of examples related to this driver, see *Examples for ADC Driver.*
7. **API Overview**

7.1. **Variable and Type Definitions**

7.1.1. **Type adc_callback_t**

```c
typedef void(* adc_callback_t)(struct adc_module *const module)
```

Type of the callback functions.

7.2. **Structure Definitions**

7.2.1. **Struct adc_config**

Configuration structure for an ADC instance. This structure should be initialized by the `adc_get_config_defaults()` function before being modified by the user application.

Table 7-1. Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum adc_accumulate_samples</td>
<td>accumulate_samples</td>
<td>Number of ADC samples to accumulate when using the ADC_RESOLUTION_CUSTOM mode</td>
</tr>
<tr>
<td>enum adc_clock_prescaler</td>
<td>clock_prescaler</td>
<td>Clock prescaler</td>
</tr>
<tr>
<td>enum gclk_generator</td>
<td>clock_source</td>
<td>GCLK generator used to clock the peripheral</td>
</tr>
<tr>
<td>struct adc_correction_config</td>
<td>correction</td>
<td>Gain and offset correction configuration structure</td>
</tr>
<tr>
<td>bool</td>
<td>differential_mode</td>
<td>Enables differential mode if true</td>
</tr>
<tr>
<td>enum adc_divide_result</td>
<td>divide_result</td>
<td>Division ration when using the ADC_RESOLUTION_CUSTOM mode</td>
</tr>
<tr>
<td>enum adc_event_action</td>
<td>event_action</td>
<td>Event action to take on incoming event</td>
</tr>
<tr>
<td>bool</td>
<td>freerunning</td>
<td>Enables free running mode if true</td>
</tr>
<tr>
<td>bool</td>
<td>left_adjust</td>
<td>Left adjusted result</td>
</tr>
<tr>
<td>enum adc_negative_input</td>
<td>negative_input</td>
<td>Negative MUX input</td>
</tr>
<tr>
<td>bool</td>
<td>on_demand</td>
<td>ADC On demand control</td>
</tr>
<tr>
<td>Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>enum adc_positive_input</td>
<td>positive_input</td>
<td>Positive MUX input</td>
</tr>
<tr>
<td>uint32_t</td>
<td>positive_input_sequence_mask_enable</td>
<td>Positive input enabled mask for conversion sequence. The sequence start from the lowest input, and go to the next enabled input automatically when the conversion is done. If no bits are set the sequence is disabled.</td>
</tr>
<tr>
<td>enum adc_reference</td>
<td>reference</td>
<td>Voltage reference</td>
</tr>
<tr>
<td>bool</td>
<td>reference_compensation_enable</td>
<td>Enables reference buffer offset compensation if true. This will increase the accuracy of the gain stage, but decreases the input impedance; therefore the startup time of the reference must be increased.</td>
</tr>
<tr>
<td>enum adc_resolution</td>
<td>resolution</td>
<td>Result resolution</td>
</tr>
<tr>
<td>bool</td>
<td>run_in_standby</td>
<td>ADC run in standby control</td>
</tr>
<tr>
<td>uint8_t</td>
<td>sample_length</td>
<td>This value (0-63) control the ADC sampling time in number of half ADC prescaled clock cycles (depends of \texttt{ADC_PRESCALER} value), thus controlling the ADC input impedance. Sampling time is set according to the formula: Sample time = ((sample_length+1) * (ADCclk / 2)).</td>
</tr>
<tr>
<td>bool</td>
<td>sampling_time_compensation_enable</td>
<td>Enables sampling period offset compensation if true</td>
</tr>
<tr>
<td>struct adc_window_config</td>
<td>window</td>
<td>Window monitor configuration structure</td>
</tr>
</tbody>
</table>

7.2.2. **Struct adc_correction_config**

Gain and offset correction configuration structure. Part of the \texttt{adc\_config} struct and will be initialized by \texttt{adc\_get\_config\_defaults}. 

Atmel AT11380: SAM L/C Analog-to-Digital Converter (ADC) Driver [APPLICATION NOTE]  
Atmel-42451B-SAM-Analog-to-Digital-Converter-ADC-Driver_AT11380_Application Note-12/2015
Table 7-2. Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>correction_enable</td>
<td>Enables correction for gain and offset based on values of gain_correction and offset_correction if set to true</td>
</tr>
<tr>
<td>uint16_t</td>
<td>gain_correction</td>
<td>This value defines how the ADC conversion result is compensated for gain error before written to the result register. This is a fractional value, 1-bit integer plus an 11-bit fraction, therefore $1/2 \leq \text{gain_correction} &lt; 2$. Valid gain_correction values range from $0b01000000000$ to $0b11111111111$.</td>
</tr>
<tr>
<td>int16_t</td>
<td>offset_correction</td>
<td>This value defines how the ADC conversion result is compensated for offset error before written to the result register. This is a 12-bit value in two's complement format.</td>
</tr>
</tbody>
</table>

7.2.3. Struct adc_events

Event flags for the ADC module. This is used to enable and disable events via \texttt{adc\_enable\_events()} and \texttt{adc\_disable\_events()}.  

Table 7-3. Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>generate_event_on_conversion_done</td>
<td>Enable event generation on conversion done</td>
</tr>
<tr>
<td>bool</td>
<td>generate_event_on_window_monitor</td>
<td>Enable event generation on window monitor</td>
</tr>
</tbody>
</table>

7.2.4. Struct adc_module

ADC software instance structure, used to retain software state information of an associated hardware module instance.  

\textbf{Note:} The fields of this structure should not be altered by the user application; they are reserved for module-internal use only.

7.2.5. Struct adc_window_config

Window monitor configuration structure.  

Table 7-4. Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32_t</td>
<td>window_lower_value</td>
<td>Lower window value</td>
</tr>
<tr>
<td>enum adc_window_mode</td>
<td>window_mode</td>
<td>Selected window mode</td>
</tr>
<tr>
<td>int32_t</td>
<td>window_upper_value</td>
<td>Upper window value</td>
</tr>
</tbody>
</table>

7.3. Macro Definitions

7.3.1. Module Status Flags

ADC status flags, returned by \texttt{adc\_get\_status()} and cleared by \texttt{adc\_clear\_status()}.  

Atmel AT11380: SAM L/C Analog-to-Digital Converter (ADC) Driver [APPLICATION NOTE]  
Atmel-42451B-SAM-Analog-to-Digital-Converter-ADC-Driver_AT11380_Application_Note-12/2015
7.3.1.1. Macro ADC_STATUS_RESULT_READY

```c
#define ADC_STATUS_RESULT_READY
```
ADC result ready.

7.3.1.2. Macro ADC_STATUS_WINDOW

```c
#define ADC_STATUS_WINDOW
```
Window monitor match.

7.3.1.3. Macro ADC_STATUS_OVERRUN

```c
#define ADC_STATUS_OVERRUN
```
ADC result overwritten before read.

7.3.2. Macro FEATURE_ADC_SUPPORT_MASTER_SLAVE

```c
#define FEATURE_ADC_SUPPORT_MASTER_SLAVE
```
Output Driver Strength Selection feature support.

7.4. Function Definitions

7.4.1. Driver Initialization and Configuration

7.4.1.1. Function adc_init()

Initializes the ADC.

```c
enum status_code adc_init(
    struct adc_module *const module_inst,
    Adc * hw,
    struct adc_config * config)
```
Initializes the ADC device struct and the hardware module based on the given configuration struct values.

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[out]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>hw</td>
<td>Pointer to the ADC module instance</td>
</tr>
<tr>
<td>[in]</td>
<td>config</td>
<td>Pointer to the configuration struct</td>
</tr>
</tbody>
</table>

Returns

Status of the initialization procedure.
Table 7-6. Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The initialization was successful</td>
</tr>
<tr>
<td>STATUS_ERR_INVALID_ARG</td>
<td>Invalid argument(s) were provided</td>
</tr>
<tr>
<td>STATUS_BUSY</td>
<td>The module is busy with a reset operation</td>
</tr>
<tr>
<td>STATUS_ERR_DENIED</td>
<td>The module is enabled</td>
</tr>
</tbody>
</table>

7.4.1.2. Function adc_get_config_defaults()

Initializes an ADC configuration structure to defaults.

```c
void adc_get_config_defaults(
    struct adc_config *const config)
```

Initializes a given ADC configuration struct to a set of known default values. This function should be called on any new instance of the configuration struct before being modified by the user application.

The default configuration is as follows:

- GCLK generator 0 (GCLK main) clock source
- Internal bandgap reference
- Div 2 clock prescaler
- 12-bit resolution
- Window monitor disabled
- Positive input on ADC PIN 1
- Negative input on Internal ground
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running disabled
- All events (input and generation) disabled
- ADC run in standby disabled
- ADC On demand disabled
- No sampling time compensation
- Disable the positive input sequense
- No reference compensation
- No gain/offset correction
- No added sampling time

Table 7-7. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[out]</td>
<td>config</td>
<td>Pointer to configuration struct to initialize to default values</td>
</tr>
</tbody>
</table>
7.4.2. Status Management

7.4.2.1. Function adc_get_status()

Retrieves the current module status.

```
uint32_t adc_get_status(
    struct adc_module *const module_inst)
```

Retrieves the status of the module, giving overall state information.

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
</tbody>
</table>

**Returns**

Bitmask of ADC_STATUS_* flags.

**Table 7-9. Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_STATUS_RESULT_READY</td>
<td>ADC result is ready to be read</td>
</tr>
<tr>
<td>ADC_STATUS_WINDOW</td>
<td>ADC has detected a value inside the set window range</td>
</tr>
<tr>
<td>ADC_STATUS_OVERRUN</td>
<td>ADC result has overrun</td>
</tr>
</tbody>
</table>

7.4.2.2. Function adc_clear_status()

Clears a module status flag.

```
void adc_clear_status(
    struct adc_module *const module_inst,
    const uint32_t status_flags)
```

Clears the given status flag of the module.

**Table 7-10. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>status_flags</td>
<td>Bitmask of ADC_STATUS_* flags to clear</td>
</tr>
</tbody>
</table>

7.4.3. Enable, Disable, and Reset ADC Module, Start Conversion and Read Result

7.4.3.1. Function adc_enable()

Enables the ADC module.

```
enum status_code adc_enable(
    struct adc_module *const module_inst)
```

Enables an ADC module that has previously been configured. If any internal reference is selected it will be enabled.
7.4.3.2. Function adc_disable()
Disables the ADC module.

```c
enum status_code adc_disable(
    struct adc_module *const module_inst)
```

Disables an ADC module that was previously enabled.

7.4.3.3. Function adc_reset()
Resets the ADC module.

```c
enum status_code adc_reset(
    struct adc_module *const module_inst)
```

Resets an ADC module, clearing all module state, and registers to their default values.

7.4.3.4. Function adc_enable_events()
Enables an ADC event input or output.

```c
void adc_enable_events(
    struct adc_module *const module_inst,
    struct adc_events *const events)
```

Enables one or more input or output events to or from the ADC module. See `Struct adc_events` for a list of events this module supports.

**Note:** Events cannot be altered while the module is enabled.

---

### Table 7-11. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
</tbody>
</table>

### Table 7-12. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
</tbody>
</table>

### Table 7-13. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
</tbody>
</table>

### Table 7-14. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Software instance for the ADC peripheral</td>
</tr>
<tr>
<td>[in]</td>
<td>events</td>
<td>Struct containing flags of events to enable</td>
</tr>
</tbody>
</table>
7.4.3.5. Function adc_disable_events()

Disables an ADC event input or output.

```c
void adc_disable_events(
    struct adc_module *const module_inst,
    struct adc_events *const events)
```

Disables one or more input or output events to or from the ADC module. See Struct adc_events for a list of events this module supports.

**Note:** Events cannot be altered while the module is enabled.

Table 7-15. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Software instance for the ADC peripheral</td>
</tr>
<tr>
<td>[in]</td>
<td>events</td>
<td>Struct containing flags of events to disable</td>
</tr>
</tbody>
</table>

7.4.3.6. Function adc_start_conversion()

Starts an ADC conversion.

```c
void adc_start_conversion(
    struct adc_module *const module_inst)
```

Starts a new ADC conversion.

Table 7-16. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
</tbody>
</table>

7.4.3.7. Function adc_read()

Reads the ADC result.

```c
enum status_code adc_read(
    struct adc_module *const module_inst,
    uint16_t * result)
```

Reads the result from an ADC conversion that was previously started.

Table 7-17. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[out]</td>
<td>result</td>
<td>Pointer to store the result value in</td>
</tr>
</tbody>
</table>

**Returns**

Status of the ADC read request.
Table 7-18. Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The result was retrieved successfully</td>
</tr>
<tr>
<td>STATUS_BUSY</td>
<td>A conversion result was not ready</td>
</tr>
<tr>
<td>STATUS_ERR_OVERFLOW</td>
<td>The result register has been overwritten by the ADC module before the result was read by the software</td>
</tr>
</tbody>
</table>

7.4.4. Runtime Changes of ADC Module

7.4.4.1. Function adc_flush()

Flushes the ADC pipeline.

```c
void adc_flush(
    struct adc_module *const module_inst)
```

Flushes the pipeline and restarts the ADC clock on the next peripheral clock edge. All conversions in progress will be lost. When flush is complete, the module will resume where it left off.

Table 7-19. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
</tbody>
</table>

7.4.4.2. Function adc_set_window_mode()

Sets the ADC window mode.

```c
void adc_set_window_mode(
    struct adc_module *const module_inst,
    const enum adc_window_mode window_mode,
    const int16_t window_lower_value,
    const int16_t window_upper_value)
```

Sets the ADC window mode to a given mode and value range.

Table 7-20. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>window_mode</td>
<td>Window monitor mode to set</td>
</tr>
<tr>
<td>[in]</td>
<td>window_lower_value</td>
<td>Lower window monitor threshold value</td>
</tr>
<tr>
<td>[in]</td>
<td>window_upper_value</td>
<td>Upper window monitor threshold value</td>
</tr>
</tbody>
</table>

7.4.4.3. Function adc_set_positive_input()

Sets positive ADC input pin.

```c
void adc_set_positive_input(
    struct adc_module *const module_inst,
    const enum adc_positive_input positive_input)
```
Sets the positive ADC input pin selection.

**Table 7-21. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>positive_input</td>
<td>Positive input pin</td>
</tr>
</tbody>
</table>

**7.4.4. Function adc_set_negative_input()**

Sets negative ADC input pin for differential mode.

```c
void adc_set_negative_input(
    struct adc_module *const module_inst,
    const enum adc_negative_input negative_input)
```

Sets the negative ADC input pin, when the ADC is configured in differential mode.

**Table 7-22. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>negative_input</td>
<td>Negative input pin</td>
</tr>
</tbody>
</table>

**7.4.5. Enable and Disable Interrupts**

**7.4.5.1. Function adc_enable_interrupt()**

Enable interrupt.

```c
void adc_enable_interrupt(
    struct adc_module *const module_inst,
    enum adc_interrupt_flag interrupt)
```

Enable the given interrupt request from the ADC module.

**Table 7-23. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC instance struct</td>
</tr>
<tr>
<td></td>
<td>interrupt</td>
<td>Interrupt to enable</td>
</tr>
</tbody>
</table>

**7.4.5.2. Function adc_disable_interrupt()**

Disable interrupt.

```c
void adc_disable_interrupt(
    struct adc_module *const module_inst,
    enum adc_interrupt_flag interrupt)
```

Disable the given interrupt request from the ADC module.
Table 7-24. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>interrupt</td>
<td>Interrupt to disable</td>
</tr>
</tbody>
</table>

7.4.6. Callback Management

7.4.6.1. Function adc_register_callback()

Registers a callback.

```c
void adc_register_callback(
    struct adc_module *const module,
    adc_callback_t callback_func,
    enum adc_callback callback_type)
```

Registers a callback function which is implemented by the user.

**Note:** The callback must be enabled for the interrupt handler to call it when the condition for the callback is met.

Table 7-25. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module</td>
<td>Pointer to ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>callback_func</td>
<td>Pointer to callback function</td>
</tr>
<tr>
<td>[in]</td>
<td>callback_type</td>
<td>Callback type given by an enum</td>
</tr>
</tbody>
</table>

7.4.6.2. Function adc_unregister_callback()

Unregisters a callback.

```c
void adc_unregister_callback(
    struct adc_module * module,
    enum adc_callback callback_type)
```

Unregisters a callback function which is implemented by the user.

Table 7-26. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module</td>
<td>Pointer to ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>callback_type</td>
<td>Callback type given by an enum</td>
</tr>
</tbody>
</table>

7.4.6.3. Function adc_enable_callback()

Enables callback.

```c
void adc_enable_callback(
    struct adc_module *const module,
    enum adc_callback callback_type)
```
Enables the callback function registered by `adc_register_callback`. The callback function will be called from the interrupt handler when the conditions for the callback type are met.

**Table 7-27. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module</td>
<td>Pointer to ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>callback_type</td>
<td>Callback type given by an enum</td>
</tr>
</tbody>
</table>

**Returns**

Status of the operation.

**Table 7-28. Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>If operation was completed</td>
</tr>
<tr>
<td>STATUS_ERR_INVALID</td>
<td>If operation was not completed, due to invalid callback_type</td>
</tr>
</tbody>
</table>

**7.4.6.4. Function adc_disable_callback()**

Disables callback.

```c
void adc_disable_callback(
    struct adc_module *const module,
    enum adc_callback callback_type)
```

Disables the callback function registered by the `adc_register_callback`.

**Table 7-29. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module</td>
<td>Pointer to ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>callback_type</td>
<td>Callback type given by an enum</td>
</tr>
</tbody>
</table>

**Returns**

Status of the operation.

**Table 7-30. Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>If operation was completed</td>
</tr>
<tr>
<td>STATUS_ERR_INVALID</td>
<td>If operation was not completed, due to invalid callback_type</td>
</tr>
</tbody>
</table>

**7.4.7. Job Management**

**7.4.7.1. Function adc_read_buffer_job()**

Read multiple samples from ADC.

```c
enum status_code adc_read_buffer_job(
    struct adc_module *const module_inst,
```
Read `samples` from the ADC into the `buffer`. If there is no hardware trigger defined (event action) the driver will retrigger the ADC conversion whenever a conversion is complete until `samples` has been acquired. To avoid jitter in the sampling frequency using an event trigger is advised.

**Table 7-31. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>samples</td>
<td>Number of samples to acquire</td>
</tr>
<tr>
<td>[out]</td>
<td>buffer</td>
<td>Buffer to store the ADC samples</td>
</tr>
</tbody>
</table>

**Returns**

Status of the job start.

**Table 7-32. Return Values**

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The conversion job was started successfully and is in progress</td>
</tr>
<tr>
<td>STATUS_BUSY</td>
<td>The ADC is already busy with another job</td>
</tr>
</tbody>
</table>

### 7.4.7.2. Function `adc_get_job_status()`

Gets the status of a job.

```c
enum status_code adc_get_job_status(
    struct adc_module * module_inst,
    enum adc_job_type type)
```

Gets the status of an ongoing or the last job.

**Table 7-33. Parameters**

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>type</td>
<td>Type of to get status</td>
</tr>
</tbody>
</table>

**Returns**

Status of the job.

### 7.4.7.3. Function `adc_abort_job()`

Aborts an ongoing job.

```c
void adc_abort_job(
    struct adc_module * module_inst,
    enum adc_job_type type)
```

Aborts an ongoing job.
### 7.4.8. Positive Input Sequence

#### 7.4.8.1. Function `adc_enable_positive_input_sequence()`

Enable positive input sequence mask for conversion.

```c
void adc_enable_positive_input_sequence(
    struct adc_module *const module_inst,
    uint32_t positive_input_sequence_mask_enable)
```

The sequence start from the lowest input, and go to the next enabled input automatically when the conversion is done. If no bits are set the sequence is disabled.

#### 7.4.8.2. Function `adc_disable_positive_input_sequence()`

Disable positive input in the sequence.

```c
void adc_disable_positive_input_sequence(
    struct adc_module *const module_inst)
```

#### 7.4.8.3. Function `adc_get_sequence_status()`

Get ADC sequence status.

```c
void adc_get_sequence_status(
    struct adc_module *const module_inst,
    bool * is_sequence_busy,
    uint8_t * sequence_state)
```

Check if a sequence is done and get last conversion done in the sequence.
Table 7-37. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>module_inst</td>
<td>Pointer to the ADC software instance struct</td>
</tr>
<tr>
<td>[out]</td>
<td>is_sequence_busy</td>
<td>Sequence busy status</td>
</tr>
<tr>
<td>[out]</td>
<td>sequence_state</td>
<td>This value identifies the last conversion done in the sequence</td>
</tr>
</tbody>
</table>

7.4.9. Function adc_set_master_slave_mode()

Set ADC master and slave mode.

```c
void adc_set_master_slave_mode(
    struct adc_module *const master_inst,
    struct adc_module *const slave_inst,
    enum adc_dual_mode_trigger_selection dualsel)
```

Enable ADC module Master-Slave Operation and select dual mode trigger.

Table 7-38. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>master_inst</td>
<td>Pointer to the master ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>slave_inst</td>
<td>Pointer to the slave ADC software instance struct</td>
</tr>
<tr>
<td>[in]</td>
<td>dualsel</td>
<td>Dual mode trigger selection</td>
</tr>
</tbody>
</table>

7.5. Enumeration Definitions

7.5.1. Enum adc_accumulate_samples

Enum for the possible numbers of ADC samples to accumulate. This setting is only used when the ADC_RESOLUTION_CUSTOM resolution setting is used.

Table 7-39. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_ACCUMULATE_DISABLE</td>
<td>No averaging</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_2</td>
<td>Average 2 samples</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_4</td>
<td>Average 4 samples</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_8</td>
<td>Average 8 samples</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_16</td>
<td>Average 16 samples</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_32</td>
<td>Average 32 samples</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_64</td>
<td>Average 64 samples</td>
</tr>
<tr>
<td>ADC_ACCUMULATE_SAMPLES_128</td>
<td>Average 128 samples</td>
</tr>
</tbody>
</table>
7.5.2. **Enum adc_callback**

Callback types for ADC callback driver.

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_CALLBACK_READ_BUFFER</td>
<td>Callback for buffer received</td>
</tr>
<tr>
<td>ADC_CALLBACK_WINDOW</td>
<td>Callback when window is hit</td>
</tr>
<tr>
<td>ADC_CALLBACK_ERROR</td>
<td>Callback for error</td>
</tr>
</tbody>
</table>

7.5.3. **Enum adc_clock_prescaler**

Enum for the possible clock prescaler values for the ADC.

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV2</td>
<td>ADC clock division factor 2</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV4</td>
<td>ADC clock division factor 4</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV8</td>
<td>ADC clock division factor 8</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV16</td>
<td>ADC clock division factor 16</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV32</td>
<td>ADC clock division factor 32</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV64</td>
<td>ADC clock division factor 64</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV128</td>
<td>ADC clock division factor 128</td>
</tr>
<tr>
<td>ADC_CLOCK_PRESCALER_DIV256</td>
<td>ADC clock division factor 256</td>
</tr>
</tbody>
</table>

7.5.4. **Enum adc_divide_result**

Enum for the possible division factors to use when accumulating multiple samples. To keep the same resolution for the averaged result and the actual input value, the division factor must be equal to the number of samples accumulated. This setting is only used when the ADC_RESOLUTION_CUSTOM resolution setting is used.
Table 7-42. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_DIVIDE_RESULT_DISABLE</td>
<td>Don't divide result register after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_2</td>
<td>Divide result register by 2 after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_4</td>
<td>Divide result register by 4 after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_8</td>
<td>Divide result register by 8 after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_16</td>
<td>Divide result register by 16 after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_32</td>
<td>Divide result register by 32 after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_64</td>
<td>Divide result register by 64 after accumulation</td>
</tr>
<tr>
<td>ADC_DIVIDE_RESULT_128</td>
<td>Divide result register by 128 after accumulation</td>
</tr>
</tbody>
</table>

7.5.5. Enum adc_dual_mode_trigger_selection

Enum for the trigger selection in dual mode.

Table 7-43. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_DUAL_MODE_BOTH</td>
<td>Start event or software trigger will start a conversion on both ADCs</td>
</tr>
<tr>
<td>ADC_DUAL_MODE_INTERLEAVE</td>
<td>START event or software trigger will alternatingly start a conversion on ADC0 and ADC1</td>
</tr>
</tbody>
</table>

7.5.6. Enum adc_event_action

Enum for the possible actions to take on an incoming event.

Table 7-44. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_EVENT_ACTION_DISABLED</td>
<td>Event action disabled</td>
</tr>
<tr>
<td>ADC_EVENT_ACTION_FLUSH_START_CONV</td>
<td>Flush ADC and start conversion</td>
</tr>
<tr>
<td>ADC_EVENT_ACTION_START_CONV</td>
<td>Start conversion</td>
</tr>
</tbody>
</table>

7.5.7. Enum adc_interrupt_flag

Enum for the possible ADC interrupt flags.
Table 7-45. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_INTERRUPT_RESULT_READY</td>
<td>ADC result ready</td>
</tr>
<tr>
<td>ADC_INTERRUPT_WINDOW</td>
<td>Window monitor match</td>
</tr>
<tr>
<td>ADC_INTERRUPT_OVERRUN</td>
<td>ADC result overwritten before read</td>
</tr>
</tbody>
</table>

7.5.8. Enum adc_job_type

Enum for the possible types of ADC asynchronous jobs that may be issued to the driver.

Table 7-46. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_JOB_READ_BUFFER</td>
<td>Asynchronous ADC read into a user provided buffer</td>
</tr>
</tbody>
</table>

7.5.9. Enum adc_negative_input

Enum for the possible negative MUX input selections for the ADC.

Table 7-47. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN0</td>
<td>ADC0 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN1</td>
<td>ADC1 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN2</td>
<td>ADC2 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN3</td>
<td>ADC3 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN4</td>
<td>ADC4 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN5</td>
<td>ADC5 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN6</td>
<td>ADC6 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_PIN7</td>
<td>ADC7 pin</td>
</tr>
<tr>
<td>ADC_NEGATIVE_INPUT_GND</td>
<td>Internal ground</td>
</tr>
</tbody>
</table>

7.5.10. Enum adc_oversampling_and_decimation

Enum for the possible numbers of bits resolution can be increased by when using oversampling and decimation.

Table 7-48. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_OVERSAMPLING_AND_DECIMATION_DISABLE</td>
<td>Don't use oversampling and decimation mode</td>
</tr>
<tr>
<td>ADC_OVERSAMPLING_AND_DECIMATION_1BIT</td>
<td>1-bit resolution increase</td>
</tr>
</tbody>
</table>
7.5.11. **Enum adc_positive_input**

Enum for the possible positive MUX input selections for the ADC.

**Table 7-49. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_POSITIVE_INPUT_PIN0</td>
<td>ADC0 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN1</td>
<td>ADC1 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN2</td>
<td>ADC2 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN3</td>
<td>ADC3 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN4</td>
<td>ADC4 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN5</td>
<td>ADC5 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN6</td>
<td>ADC6 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN7</td>
<td>ADC7 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN8</td>
<td>ADC8 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN9</td>
<td>ADC9 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN10</td>
<td>ADC10 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN11</td>
<td>ADC11 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN12</td>
<td>ADC12 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN13</td>
<td>ADC13 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN14</td>
<td>ADC14 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN15</td>
<td>ADC15 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN16</td>
<td>ADC16 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN17</td>
<td>ADC17 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN18</td>
<td>ADC18 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN19</td>
<td>ADC19 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN20</td>
<td>ADC20 pin.</td>
</tr>
<tr>
<td>Enum value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN21</td>
<td>ADC21 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN22</td>
<td>ADC22 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_PIN23</td>
<td>ADC23 pin</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_TEMP</td>
<td>Temperature reference.</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_BANDGAP</td>
<td>Bandgap voltage</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_SCALEDCOREVCC</td>
<td>1/4 scaled core supply</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_SCALEDIOVCC</td>
<td>1/4 scaled I/O supply</td>
</tr>
<tr>
<td>ADC_POSITIVE_INPUT_DAC</td>
<td>DAC input</td>
</tr>
</tbody>
</table>

7.5.12. **Enum adc_reference**

Enum for the possible reference voltages for the ADC.

Table 7-50. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_REFERENCE_INTREF</td>
<td>Internal Bandgap Reference</td>
</tr>
<tr>
<td>ADC_REFERENCE_INTVCC0</td>
<td>1/1.48V&lt;sub&gt;CC&lt;/sub&gt; reference</td>
</tr>
<tr>
<td>ADC_REFERENCE_INTVCC1</td>
<td>1/2V&lt;sub&gt;CC&lt;/sub&gt; (only for internal V&lt;sub&gt;CC&lt;/sub&gt; &gt; 2.1V)</td>
</tr>
<tr>
<td>ADC_REFERENCE_AREFA</td>
<td>External reference A</td>
</tr>
<tr>
<td>ADC_REFERENCE_INTVCC2</td>
<td>VDDANA</td>
</tr>
</tbody>
</table>

7.5.13. **Enum adc_resolution**

Enum for the possible resolution values for the ADC.

Table 7-51. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_RESOLUTION_12BIT</td>
<td>ADC 12-bit resolution</td>
</tr>
<tr>
<td>ADC_RESOLUTION_16BIT</td>
<td>ADC 16-bit resolution using oversampling and decimation</td>
</tr>
<tr>
<td>ADC_RESOLUTION_10BIT</td>
<td>ADC 10-bit resolution</td>
</tr>
<tr>
<td>ADC_RESOLUTION_8BIT</td>
<td>ADC 8-bit resolution</td>
</tr>
<tr>
<td>ADC_RESOLUTION_13BIT</td>
<td>ADC 13-bit resolution using oversampling and decimation</td>
</tr>
<tr>
<td>ADC_RESOLUTION_14BIT</td>
<td>ADC 14-bit resolution using oversampling and decimation</td>
</tr>
</tbody>
</table>
### Enum value | Description
---|---
ADC_RESOLUTION_15BIT | ADC 15-bit resolution using oversampling and decimation
ADC_RESOLUTION_CUSTOM | ADC 16-bit result register for use with averaging. When using this mode the ADC result register will be set to 16-bit wide, and the number of samples to accumulate and the division factor is configured by the `adc_config::accumulate_samples` and `adc_config::divide_result` members in the configuration struct.

#### 7.5.14. Enum adc_window_mode

Enum for the possible window monitor modes for the ADC.

**Table 7-52. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_WINDOW_MODE_DISABLE</td>
<td>No window mode</td>
</tr>
<tr>
<td>ADC_WINDOW_MODE_ABOVE_LOWER</td>
<td>RESULT &gt; WINLT</td>
</tr>
<tr>
<td>ADC_WINDOW_MODE_BELOW_UPPER</td>
<td>RESULT &lt; WINUT</td>
</tr>
<tr>
<td>ADC_WINDOW_MODE_BETWEEN</td>
<td>WINLT &lt; RESULT &lt; WINUT</td>
</tr>
<tr>
<td>ADC_WINDOW_MODE_BETWEEN_INVERTED</td>
<td>!(WINLT &lt; RESULT &lt; WINUT)</td>
</tr>
</tbody>
</table>
8. Extra Information for ADC Driver

8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog-to-Digital Converter</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital-to-Analog Converter</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
</tbody>
</table>

8.2. Dependencies

This driver has the following dependencies:

- System Pin Multiplexer Driver

8.3. Errata

There are no errata related to this driver.

8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

<table>
<thead>
<tr>
<th>Changelog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Release</td>
</tr>
</tbody>
</table>
9. Examples for ADC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Analog-to-Digital Converter (ADC) Driver. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for ADC - Basic
- Quick Start Guide for ADC - Callback
- Quick Start Guide for Using DMA with ADC/DAC

9.1. Quick Start Guide for ADC - Basic

In this use case, the ADC will be configured with the following settings:

- 1V from internal bandgap reference
- Div 4 clock prescaler
- 12-bit resolution
- Window monitor disabled
- No gain
- Positive input on ADC PIN x (depend on default configuration)
- Negative input to GND (single ended)
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running disabled
- All events (input and generation) disabled
- Sleep operation disabled
- No reference compensation
- No gain/offset correction
- No added sampling time
- Pin scan mode disabled

9.1.1. Setup

9.1.1.1. Prerequisites

There are no special setup requirements for this use-case.

9.1.1.2. Code

Add to the main application source file, outside of any functions:

```c
struct adc_module adc_instance;
```

Copy-paste the following setup code to your user application:

```c
void configure_adc(void)
{
    struct adc_config config_adc;
    adc_get_config_defaults(&config_adc);
```
#if (SAMC21)
    adc_init(&adc_instance, ADC1, &config_adc);
#else
    adc_init(&adc_instance, ADC, &config_adc);
#endif
    adc_enable(&adc_instance);
}

Add to user application initialization (typically the start of `main()`):
configure_adc();

9.1.3. Workflow

1. Create a module software instance structure for the ADC module to store the ADC driver state while in use.
   ```c
   struct adc_module adc_instance;
   ```
   **Note:** This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the ADC module.
   1. Create an ADC module configuration struct, which can be filled out to adjust the configuration of a physical ADC peripheral.
      ```c
      struct adc_config config_adc;
      ```
   2. Initialize the ADC configuration struct with the module's default values.
      ```c
      adc_get_config_defaults(&config_adc);
      ```
      **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set ADC configurations.
   ```c
   #if (SAMC21)
    adc_init(&adc_instance, ADC1, &config_adc);
   #else
    adc_init(&adc_instance, ADC, &config_adc);
   #endif
   ```

4. Enable the ADC module so that conversions can be made.
   ```c
   adc_enable(&adc_instance);
   ```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:
```c
adc_start_conversion(&adc_instance);
uint16_t result;

do {
    /* Wait for conversion to be done and read out result */
} while (adc_read(&adc_instance, &result) == STATUS_BUSY);
```
9.1.2.2. Workflow

1. Start conversion.
   ```c
   adc_start_conversion(&adc_instance);
   ```

2. Wait until conversion is done and read result.
   ```c
   uint16_t result;
   do {
       /* Wait for conversion to be done and read out result */
   } while (adc_read(&adc_instance, &result) == STATUS_BUSY);
   ```

3. Enter an infinite loop once the conversion is complete.
   ```c
   while (1) {
       /* Infinite loop */
   }
   ```

9.2. Quick Start Guide for ADC - Callback

In this use case, the ADC will convert 128 samples using interrupt driven conversion. When all samples have been sampled, a callback will be called that signals the main application that conversion is complete.

The ADC will be set up as follows:

- $V_{CC}/2$ as reference
- Div 8 clock prescaler
- 12-bit resolution
- Window monitor disabled
- 1/2 gain
- Positive input on ADC PIN 0
- Negative input to GND (single ended)
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running disabled
- All events (input and generation) disabled
- Sleep operation disabled
- No reference compensation
- No gain/offset correction
- No added sampling time
- Pin scan mode disabled

9.2.1. Setup

9.2.1.1. Prerequisites

There are no special setup requirements for this use-case.
9.2.1.2. Code

Add to the main application source file, outside of any functions:

```c
struct adc_module adc_instance;
#define ADC_SAMPLES 128
uint16_t adc_result_buffer[ADC_SAMPLES];
```

Callback function:

```c
volatile bool adc_read_done = false;
void adc_complete_callback(
    struct adc_module *const module)
{
    adc_read_done = true;
}
```

Copy-paste the following setup code to your user application:

```c
void configure_adc(void)
{
    struct adc_config config_adc;
    adc_get_config_defaults(&config_adc);
#if (!SAML21) && (!SAML22) && (!SAMC21)
    config_adc.gain_factor = ADC_GAIN_FACTOR_DIV2;
#else
    config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV8;
    config_adc.reference = ADC_REFERENCE_INTVCC1;
#endif
    config_adc.positive_input = ADC_POSITIVE_INPUT_PIN5;
#else
    config_adc.positive_input = ADC_POSITIVE_INPUT_PIN6;
#endif
    config_adc.resolution = ADC_RESOLUTION_12BIT;
#if (SAMC21)
    adc_init(&adc_instance, ADC1, &config_adc);
#else
    adc_init(&adc_instance, ADC, &config_adc);
#endif
    adc_enable(&adc_instance);
}
void configure_adc_callbacks(void)
{
    adc_register_callback(&adc_instance, adc_complete_callback, ADC_CALLBACK_READ_BUFFER);
    adc_enable_callback(&adc_instance, ADC_CALLBACK_READ_BUFFER);
}
```

Add to user application initialization (typically the start of `main()`):

```c
configure_adc();
configure_adc_callbacks();
```
9.2.1.3. Workflow

1. Create a module software instance structure for the ADC module to store the ADC driver state while in use.

```c
struct adc_module adc_instance;
```

**Note:** This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Create a buffer for the ADC samples to be stored in by the driver asynchronously.

```c
#define ADC_SAMPLES 128
uint16_t adc_result_buffer[ADC_SAMPLES];
```

3. Create a callback function that will be called each time the ADC completes an asynchronous read job.

```c
volatile bool adc_read_done = false;
void adc_complete_callback(  
    struct adc_module *const module)
{
    adc_read_done = true;
}
```

4. Configure the ADC module.

   1. Create an ADC module configuration struct, which can be filled out to adjust the configuration of a physical ADC peripheral.

   ```c
   struct adc_config config_adc;
   ```

   2. Initialize the ADC configuration struct with the module's default values.

   ```c
   adc_get_config_defaults(&config_adc);
   ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

   3. Change the ADC module configuration to suit the application.

   ```c
   #if (!SAML21) && (!SAML22) && (!SAMC21)
   config_adc.gain_factor = ADC_GAIN_FACTOR_DIV2;
   #endif
   config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV8;
   config_adc.reference = ADC_REFERENCE_INTVCC1;
   #if (SAMC21)
   config_adc.positive_input = ADC_POSITIVE_INPUT_PIN5;
   #else
   config_adc.positive_input = ADC_POSITIVE_INPUT_PIN6;
   #endif
   config_adc.resolution = ADC_RESOLUTION_12BIT;
   ```

4. Set ADC configurations.

   ```c
   #if (SAMC21)
   adc_init(&adc_instance, ADC1, &config_adc);
   #else
   adc_init(&adc_instance, ADC, &config_adc);
   #endif
   ```

5. Enable the ADC module so that conversions can be made.

```c
adc_enable(&adc_instance);
```
5. Register and enable the ADC Read Buffer Complete callback handler.
   1. Register the user-provided Read Buffer Complete callback function with the driver, so that it
      will be run when an asynchronous buffer read job completes.
      
      ```c
      adc_register_callback(&adc_instance, 
                         adc_complete_callback, ADC_CALLBACK_READ_BUFFER);
      ```
   2. Enable the Read Buffer Complete callback so that it will generate callbacks.
      
      ```c
      adc_enable_callback(&adc_instance, ADC_CALLBACK_READ_BUFFER);
      ```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

```c
system_interrupt_enable_global();
adc_read_buffer_job(&adc_instance, adc_result_buffer, ADC_SAMPLES);
while (adc_read_done == false) {
    /* Wait for asynchronous ADC read to complete */
}
while (1) {
    /* Infinite loop */
}
```

9.2.2.2. Workflow

1. Enable global interrupts, so that callbacks can be generated by the driver.
   
   ```c
   system_interrupt_enable_global();
   ```
2. Start an asynchronous ADC conversion, to store ADC samples into the global buffer and generate
   a callback when complete.
   
   ```c
   adc_read_buffer_job(&adc_instance, adc_result_buffer, ADC_SAMPLES);
   ```
3. Wait until the asynchronous conversion is complete.
   
   ```c
   while (adc_read_done == false) {
       /* Wait for asynchronous ADC read to complete */
   }
   ```
4. Enter an infinite loop once the conversion is complete.
   
   ```c
   while (1) {
       /* Infinite loop */
   }
   ```

9.3. Quick Start Guide for Using DMA with ADC/DAC

The supported board list:
- SAM D21 Xplained Pro
- SAM D11 Xplained Pro
- SAM L21 Xplained Pro
- SAM DA1 Xplained Pro
This quick start will convert an analog input signal from AIN4 and output the converted value to DAC on PA2. The data between ADC and DAC with be transferred through DMA instead of a CPU intervene.

The ADC will be configured with the following settings:
- 1/2 VDDANA
- Div 16 clock prescaler
- 10-bit resolution
- Window monitor disabled
- No gain
- Positive input on ADC AIN4
- Averaging disabled
- Oversampling disabled
- Right adjust data
- Single-ended mode
- Free running enable
- All events (input and generation) disabled
- Sleep operation disabled
- No reference compensation
- No gain/offset correction
- No added sampling time
- Pin scan mode disabled

The DAC will be configured with the following settings:
- Analog V\textsubscript{CC} as reference
- Internal output disabled
- Drive the DAC output to PA2
- Right adjust data
- The output buffer is disabled when the chip enters STANDBY sleep mode

The DMA will be configured with the following settings:
- Move data from peripheral to peripheral
- Using ADC result ready trigger
- Using DMA priority level 0
- Beat transfer will be triggered on each trigger
- Loopback descriptor for DAC conversion

### 9.3.1. Setup

#### 9.3.1.1. Prerequisites

There are no special setup requirements for this use-case.
9.3.1.2. Code

Add to the main application source file, outside of any functions:

```c
struct dac_module dac_instance;
struct adc_module adc_instance;
struct dma_resource example_resource;

COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor SECTION_DMAC_DESCRIPTOR;
```

Copy-paste the following setup code to your user application:

```c
void configure_adc(void)
{
    struct adc_config config_adc;
    adc_get_config_defaults(&config_adc);

    #if !(SAML21)
    #if !(SAMC21)
        config_adc.gain_factor     = ADC_GAIN_FACTOR_DIV2;
    #endif
    config_adc.resolution      = ADC_RESOLUTION_10BIT;
    #endif
    config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV16;
    config_adc.reference       = ADC_REFERENCE_INTVCC1;
    config_adc.positive_input  = ADC_POSITIVE_INPUT_PIN4;
    config_adc.freerunning     = true;
    config_adc.left_adjust     = false;

    #if (SAMC21)
        adc_init(&adc_instance, ADC1, &config_adc);
    #else
        adc_init(&adc_instance, ADC, &config_adc);
    #endif

    adc_enable(&adc_instance);
}

void configure_dac(void)
{
    struct dac_config config_dac;
    dac_get_config_defaults(&config_dac);

    #if (SAML21)
        config_dac.reference = DAC_REFERENCE_INTREF;
    #else
        config_dac.reference = DAC_REFERENCE_AVCC;
    #endif

    dac_init(&dac_instance, DAC, &config_dac);
}

void configure_dac_channel(void)
{
    struct dac_chan_config config_dac_chan;
}```
void configure_dma_resource(struct dma_resource *resource) {
    struct dma_resource_config config;
    dma_get_config_defaults(&config);
    #if (SAML21)
        config.peripheral_trigger = ADC1_DMAC_ID_RESRDY;
    #else
        config.peripheral_trigger = ADC_DMAC_ID_RESRDY;
    #endif
    config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
    dma_allocate(resource, &config);
}

void setup_transfer_descriptor(DmacDescriptor *descriptor) {
    struct dma_descriptor_config descriptor_config;
    dma_descriptor_get_config_defaults(&descriptor_config);
    descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
    descriptor_config.dst_increment_enable = false;
    descriptor_config.src_increment_enable = false;
    descriptor_config.block_transfer_count = 1000;
    descriptor_config.source_address = (uint32_t)(&adc_instance.hw->RESULT.reg);
    #if (SAML21)
        descriptor_config.destination_address = (uint32_t)(&dac_instance.hw->DATA[DAC_CHANNEL_0].reg);
    #else
        descriptor_config.destination_address = (uint32_t)(&dac_instance.hw->DATA.reg);
    #endif
    descriptor_config.next_descriptor_address = (uint32_t)descriptor;
    dma_descriptor_create(descriptor, &descriptor_config);
}

Add to user application initialization (typically the start of main()):

    configure_adc();
    configure_dac();
    configure_dac_channel();
    dac_enable(&dac_instance);
    configure_dma_resource(&example_resource);
    setup_transfer_descriptor(&example_descriptor);
    dma_add_descriptor(&example_resource, &example_descriptor);
9.3.1.3. Workflow

Configure the ADC

1. Create a module software instance structure for the ADC module to store the ADC driver state while it is in use.

   ```
   struct adc_module adc_instance;
   ```

   **Note:** This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the ADC module.
   1. Create an ADC module configuration struct, which can be filled out to adjust the configuration of a physical ADC peripheral.

      ```
      struct adc_config config_adc;
      ```

   2. Initialize the ADC configuration struct with the module's default values.

      ```
      adc_get_config_defaults(&config_adc);
      ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra configurations.

   ```
   #if !(SAML21)
   #if !(SAMC21)
   config_adc.gain_factor     = ADC_GAIN_FACTOR_DIV2;
   #endif
   config_adc.resolution      = ADC_RESOLUTION_10BIT;
   #endif
   config_adc.clock_prescaler = ADC_CLOCK_PRESCALER_DIV16;
   config_adc.reference       = ADC_REFERENCE_INTVCC1;
   config_adc.positive_input  = ADC_POSITIVE_INPUT_PIN4;
   config_adc.freerunning     = true;
   config_adc.left_adjust     = false;
   ```

4. Set ADC configurations.

   ```
   #if (SAMC21)
   adc_init(&adc_instance, ADC1, &config_adc);
   #else
   adc_init(&adc_instance, ADC, &config_adc);
   #endif
   ```

5. Enable the ADC module so that conversions can be made.

   ```
   adc_enable(&adc_instance);
   ```

Configure the DAC

1. Create a module software instance structure for the DAC module to store the DAC driver state while it is in use.

   ```
   struct dac_module dac_instance;
   ```

   **Note:** This should never go out of scope as long as the module is in use. In most cases, this should be global.

2. Configure the DAC module.
1. Create a DAC module configuration struct, which can be filled out to adjust the configuration of a physical DAC peripheral.

   ```
   struct dac_config config_dac;
   ```

2. Initialize the DAC configuration struct with the module's default values.

   ```
   dac_get_config_defaults(&config_dac);
   ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Set extra DAC configurations.

   ```
   #if (SAML21)
       config_dac.reference = DAC_REFERENCE_INTREF;
   #else
       config_dac.reference = DAC_REFERENCE_AVCC;
   #endif
   ```

4. Set DAC configurations to DAC instance.

   ```
   dac_init(&dac_instance, DAC, &config_dac);
   ```

5. Enable the DAC module so that channels can be configured.

   ```
   dac_enable(&dac_instance);
   ```

3. Configure the DAC channel.

1. Create a DAC channel configuration struct, which can be filled out to adjust the configuration of a physical DAC output channel.

   ```
   struct dac_chan_config config_dac_chan;
   ```

2. Initialize the DAC channel configuration struct with the module's default values.

   ```
   dac_chan_get_config_defaults(&config_dac_chan);
   ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

3. Configure the DAC channel with the desired channel settings.

   ```
   dac_chan_set_config(&dac_instance, DAC_CHANNEL_0, &config_dac_chan);
   ```

4. Enable the DAC channel so that it can output a voltage.

   ```
   dac_chan_enable(&dac_instance, DAC_CHANNEL_0);
   ```

**Configure the DMA**

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

   ```
   struct dma_resource_config config;
   ```

2. Initialize the DMA resource configuration struct with the module's default values.

   ```
   dma_get_config_defaults(&config);
   ```

   **Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.
3. Set extra configurations for the DMA resource. ADC_DMAC_ID_RESRDY trigger causes a beat transfer in this example.

```c
#if (SAMC21)
    config.peripheral_trigger = ADC1_DMAC_ID_RESRDY;
#else
    config.peripheral_trigger = ADC_DMAC_ID_RESRDY;
#endif
config.trigger_action = DMA_TRIGGER_ACTION_BEAT;
```

4. Allocate a DMA resource with the configurations.

```c
dma_allocate(resource, &config);
```

5. Create a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```c
struct dma_descriptor_config descriptor_config;
```

6. Initialize the DMA transfer descriptor configuration struct with the module's default values.

```c
daia_descriptors_get_config_defaults(&descriptor_config);
```

**Note:** This should always be performed before using the configuration struct to ensure that all values are initialized to known default settings.

7. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address.

```c
descriptor_config.beat_size = DMA_BEAT_SIZE_HWORD;
descriptor_config.dst_increment_enable = false;
descriptor_config.src_increment_enable = false;
descriptor_config.block_transfer_count = 1000;
descriptor_config.source_address = (uint32_t)(&adc_instance.hw->RESULT.reg);
#if (SAMC21)
    descriptor_config.destination_address = (uint32_t)(&dac_instance.hw->DATA[DAC_CHANNEL_0].reg);
#else
    descriptor_config.destination_address = (uint32_t)(&dac_instance.hw->DATA.reg);
#endif
    descriptor_config.next_descriptor_address = (uint32_t)descriptor;
```

8. Create the DMA transfer descriptor.

```c
daia_descriptor_create(descriptor, &descriptor_config);
```

9. Add DMA descriptor to DMA resource.

```c
daia_add_descriptor(&example_resource, &example_descriptor);
```

9.3.2. Use Case

9.3.2.1. Code

Copy-paste the following code to your user application:

```c
adc_start_conversion(&adc_instance);
daia_start_transfer_job(&example_resource);
while (true) {
}
```
9.3.2.2. Workflow

1. Start ADC conversion.
   ```c
   adc_start_conversion(&adc_instance);
   ```

2. Start the transfer job.
   ```c
   dma_start_transfer_job(&example_resource);
   ```

3. Enter endless loop.
   ```c
   while (true) {
   }
   ```
## 10. Document Revision History

<table>
<thead>
<tr>
<th>Doc. Rev.</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>42451B</td>
<td>12/2015</td>
<td>Added support for SAM L22</td>
</tr>
<tr>
<td>42451A</td>
<td>07/2015</td>
<td>Initial document release</td>
</tr>
</tbody>
</table>