SMART ARM-based Microcontrollers

AT11380: SAM L/C Analog-to-Digital Converter (ADC) Driver

APPLICATION NOTE

Introduction

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:

Table of Contents

1. Software License

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.

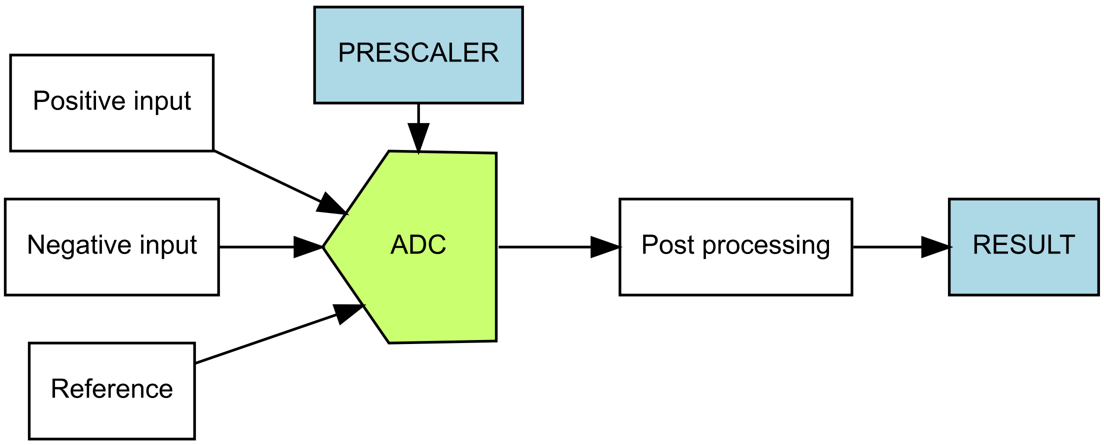
The input selection is flexible, and both single-ended and differential measurements can be made. For differential measurements, an optional gain stage is available to increase the dynamic range. In addition, several internal signal inputs are available. The ADC can provide both signed and unsigned results.

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



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

Resolution	Effective conversion rate
13-bit	Conversion rate divided by 4
14-bit	Conversion rate divided by 16
15-bit	Conversion rate divided by 64
16-bit	Conversion rate divided by 256

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

Number of samples	Final result
1	12-bit
2	13-bit
4	14-bit
8	15-bit
16	16-bit
32	16-bit
64	16-bit
128	16-bit
256	16-bit
512	16-bit
1024	16-bit

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:

$$ADC_{RESULT} = \left(VALUE_{CONV} + CORR_{OFFSET}\right) \times CORR_{GAIN}$$

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 correction 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

```
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

Type	Name	Description
enum adc_accumulate_samples	accumulate_samples	Number of ADC samples to accumulate when using the `ADC_RESOLUTION_CUSTOM` mode
enum adc_clock_prescaler	clock_prescaler	Clock prescaler
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
struct adc_correction_config	correction	Gain and offset correction configuration structure
bool	differential_mode	Enables differential mode if true
enum adc_divide_result	divide_result	Division ration when using the ADC_RESOLUTION_CUSTOM mode
enum adc_event_action	event_action	Event action to take on incoming event
bool	freerunning	Enables free running mode if true
bool	left_adjust	Left adjusted result
enum adc_negative_input	negative_input	Negative MUX input
bool	on_demand	ADC On demand control

Type	Name	Description
enum adc_positive_input	positive_input	Positive MUX input
uint32_t	positive_input_sequence_mask_enable	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.
enum adc_reference	reference	Voltage reference
bool	reference_compensation_enable	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.
enum adc_resolution	resolution	Result resolution
bool	run_in_standby	ADC run in standby control
uint8_t	sample_length	This value (0-63) control the ADC sampling time in number of half ADC prescaled clock cycles (depends of `ADC_PRESCALER` value), thus controlling the ADC input impedance. Sampling time is set according to the formula: Sample time = (sample_length+1) * (ADCclk / 2).
bool	sampling_time_compensation_enable	Enables sampling period offset compensation if true
struct adc_window_config	window	Window monitor configuration structure

7.2.2. Struct adc_correction_config

Gain and offset correction configuration structure. Part of the adc_config struct and will be initialized by adc_get_config_defaults.

Table 7-2. Members

Type	Name	Description
bool	correction_enable	Enables correction for gain and offset based on values of gain_correction and offset_correction if set to true
uint16_t	gain_correction	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 <= gain_correction < 2. Valid `gain_correction` values ranges from `0b010000000000` to `0b111111111111`.
int16_t	offset_correction	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.

7.2.3. Struct adc_events

Event flags for the ADC module. This is used to enable and disable events via adc_enable_events() and adc_disable_events().

Table 7-3. Members

Type	Name	Description
bool	generate_event_on_conversion_done	Enable event generation on conversion done
bool	generate_event_on_window_monitor	Enable event generation on window monitor

7.2.4. Struct adc_module

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

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

Type	Name	Description
int32_t	window_lower_value	Lower window value
enum adc_window_mode	window_mode	Selected window mode
int32_t	window_upper_value	Upper window value

7.3. Macro Definitions

7.3.1. Module Status Flags

ADC status flags, returned by adc_get_status() and cleared by adc_clear_status().

7.3.1.1. Macro ADC_STATUS_RESULT_READY

```
#define ADC_STATUS_RESULT_READY
```

ADC result ready.

7.3.1.2. Macro ADC_STATUS_WINDOW

```
#define ADC_STATUS_WINDOW
```

Window monitor match.

7.3.1.3. Macro ADC_STATUS_OVERRUN

```
#define ADC_STATUS_OVERRUN
```

ADC result overwritten before read.

7.3.2. Macro FEATURE_ADC_SUPPORT_MASTER_SLAVE

```
#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.

```
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 7-5. Parameters

Data direction	Parameter name	Description
[out]	module_inst	Pointer to the ADC software instance struct
[in]	hw	Pointer to the ADC module instance
[in]	config	Pointer to the configuration struct

Returns
Status of the initialization procedure.

Table 7-6. Return Values

Return value	Description
STATUS_OK	The initialization was successful
STATUS_ERR_INVALID_ARG	Invalid argument(s) were provided
STATUS_BUSY	The module is busy with a reset operation
STATUS_ERR_DENIED	The module is enabled

7.4.1.2. Function adc_get_config_defaults()

Initializes an ADC configuration structure to defaults.

```
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

Data direction	Parameter name	Description
[out]	config	Pointer to configuration struct to initialize to default values

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 7-8. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

Returns
Bitmask of `ADC_STATUS_*` flags.

Table 7-9. Return Values

Return value	Description
ADC_STATUS_RESULT_READY	ADC result is ready to be read
ADC_STATUS_WINDOW	ADC has detected a value inside the set window range
ADC_STATUS_OVERRUN	ADC result has overrun

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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	status_flags	Bitmask of `ADC_STATUS_*` flags to clear

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.

Table 7-11. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

7.4.3.2. Function adc_disable()

Disables the ADC module.

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

Disables an ADC module that was previously enabled.

Table 7-12. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

7.4.3.3. Function adc_reset()

Resets the ADC module.

```
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.

Table 7-13. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

7.4.3.4. Function adc_enable_events()

Enables an ADC event input or output.

```
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 7-14. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the ADC peripheral
[in]	events	Struct containing flags of events to enable

7.4.3.5. Function adc_disable_events()

Disables an ADC event input or output.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Software instance for the ADC peripheral
[in]	events	Struct containing flags of events to disable

7.4.3.6. Function adc_start_conversion()

Starts an ADC conversion.

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

Starts a new ADC conversion.

Table 7-16. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

7.4.3.7. Function adc_read()

Reads the ADC result.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[out]	result	Pointer to store the result value in

Returns
Status of the ADC read request.

Table 7-18. Return Values

Return value	Description
STATUS_OK	The result was retrieved successfully
STATUS_BUSY	A conversion result was not ready
STATUS_ERR_OVERFLOW	The result register has been overwritten by the ADC module before the result was read by the software

7.4.4. Runtime Changes of ADC Module

7.4.4.1. Function adc_flush()

Flushes the ADC pipeline.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

7.4.4.2. Function adc_set_window_mode()

Sets the ADC window mode.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	window_mode	Window monitor mode to set
[in]	window_lower_value	Lower window monitor threshold value
[in]	window_upper_value	Upper window monitor threshold value

7.4.4.3. Function adc_set_positive_input()

Sets positive ADC input pin.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	positive_input	Positive input pin

7.4.4.4. Function adc_set_negative_input()

Sets negative ADC input pin for differential mode.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	negative_input	Negative input pin

7.4.5. Enable and Disable Interrupts

7.4.5.1. Function adc_enable_interrupt()

Enable interrupt.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	interrupt	Interrupt to enable

7.4.5.2. Function adc_disable_interrupt()

Disable interrupt.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	interrupt	Interrupt to disable

7.4.6. Callback Management

7.4.6.1. Function adc_register_callback()

Registers a callback.

```
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

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_func	Pointer to callback function
[in]	callback_type	Callback type given by an enum

7.4.6.2. Function adc_unregister_callback()

Unregisters a callback.

```
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

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_type	Callback type given by an enum

7.4.6.3. Function adc_enable_callback()

Enables callback.

```
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

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_type	Callback type given by an enum

Returns
Status of the operation.

Table 7-28. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID	If operation was not completed, due to invalid callback_type

7.4.6.4. Function adc_disable_callback()

Disables callback.

```
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

Data direction	Parameter name	Description
[in]	module	Pointer to ADC software instance struct
[in]	callback_type	Callback type given by an enum

Returns
Status of the operation.

Table 7-30. Return Values

Return value	Description
STATUS_OK	If operation was completed
STATUS_ERR_INVALID	If operation was not completed, due to invalid callback_type

7.4.7. Job Management

7.4.7.1. Function adc_read_buffer_job()

Read multiple samples from ADC.

```
enum status_code adc_read_buffer_job(
        struct adc_module *const module_inst,
        uint16_t * buffer,
        uint16_t samples)
```

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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	samples	Number of samples to acquire
[out]	buffer	Buffer to store the ADC samples

Returns
Status of the job start.

Table 7-32. Return Values

Return value	Description
STATUS_OK	The conversion job was started successfully and is in progress
STATUS_BUSY	The ADC is already busy with another job

7.4.7.2. Function adc_get_job_status()

Gets the status of a job.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	type	Type of to get status

Returns
Status of the job.

7.4.7.3. Function adc_abort_job()

Aborts an ongoing job.

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

Aborts an ongoing job.

Table 7-34. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	type	Type of job to abort

7.4.8. Positive Input Sequence

7.4.8.1. Function adc_enable_positive_input_sequence()

Enable positive input sequence mask for conversion.

```
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.

Table 7-35. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[in]	eanble_seq_mask	Sequence mask

7.4.8.2. Function adc_disable_positive_input_sequence()

Disable positive input in the sequence.

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

Disable positive input in the sequence.

Table 7-36. Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct

7.4.8.3. Function adc_get_sequence_status()

Get ADC sequence status.

```
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

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the ADC software instance struct
[out]	is_sequence_busy	Sequence busy status
[out]	sequence_state	This value identifies the last conversion done in the sequence

7.4.9. Function adc_set_master_slave_mode()

Set ADC master and slave mode.

```
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

Data direction	Parameter name	Description
[in]	master_inst	Pointer to the master ADC software instance struct
[in]	slave_inst	Pointer to the slave ADC software instance struct
[in]	dualsel	Dual mode trigger selection

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

Enum value	Description
ADC_ACCUMULATE_DISABLE	No averaging
ADC_ACCUMULATE_SAMPLES_2	Average 2 samples
ADC_ACCUMULATE_SAMPLES_4	Average 4 samples
ADC_ACCUMULATE_SAMPLES_8	Average 8 samples
ADC_ACCUMULATE_SAMPLES_16	Average 16 samples
ADC_ACCUMULATE_SAMPLES_32	Average 32 samples
ADC_ACCUMULATE_SAMPLES_64	Average 64 samples
ADC_ACCUMULATE_SAMPLES_128	Average 128 samples

Enum value	Description
ADC_ACCUMULATE_SAMPLES_256	Average 256 samples
ADC_ACCUMULATE_SAMPLES_512	Average 512 samples
ADC_ACCUMULATE_SAMPLES_1024	Average 1024 samples

7.5.2. Enum adc_callback

Callback types for ADC callback driver.

Table 7-40. Members

Enum value	Description
ADC_CALLBACK_READ_BUFFER	Callback for buffer received
ADC_CALLBACK_WINDOW	Callback when window is hit
ADC_CALLBACK_ERROR	Callback for error

7.5.3. Enum adc_clock_prescaler

Enum for the possible clock prescaler values for the ADC.

Table 7-41. Members

Enum value	Description
ADC_CLOCK_PRESCALER_DIV2	ADC clock division factor 2
ADC_CLOCK_PRESCALER_DIV4	ADC clock division factor 4
ADC_CLOCK_PRESCALER_DIV8	ADC clock division factor 8
ADC_CLOCK_PRESCALER_DIV16	ADC clock division factor 16
ADC_CLOCK_PRESCALER_DIV32	ADC clock division factor 32
ADC_CLOCK_PRESCALER_DIV64	ADC clock division factor 64
ADC_CLOCK_PRESCALER_DIV128	ADC clock division factor 128
ADC_CLOCK_PRESCALER_DIV256	ADC clock division factor 256

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

Enum value	Description
ADC_DIVIDE_RESULT_DISABLE	Don't divide result register after accumulation
ADC_DIVIDE_RESULT_2	Divide result register by 2 after accumulation
ADC_DIVIDE_RESULT_4	Divide result register by 4 after accumulation
ADC_DIVIDE_RESULT_8	Divide result register by 8 after accumulation
ADC_DIVIDE_RESULT_16	Divide result register by 16 after accumulation
ADC_DIVIDE_RESULT_32	Divide result register by 32 after accumulation
ADC_DIVIDE_RESULT_64	Divide result register by 64 after accumulation
ADC_DIVIDE_RESULT_128	Divide result register by 128 after accumulation

7.5.5. Enum adc_dual_mode_trigger_selection

Enum for the trigger selection in dual mode.

Table 7-43. Members

Enum value	Description
ADC_DUAL_MODE_BOTH	Start event or software trigger will start a conversion on both ADCs
ADC_DUAL_MODE_INTERLEAVE	START event or software trigger will alternatingly start a conversion on ADC0 and ADC1

7.5.6. Enum adc_event_action

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

Table 7-44. Members

Enum value	Description
ADC_EVENT_ACTION_DISABLED	Event action disabled
ADC_EVENT_ACTION_FLUSH_START_CONV	Flush ADC and start conversion
ADC_EVENT_ACTION_START_CONV	Start conversion

7.5.7. Enum adc_interrupt_flag

Enum for the possible ADC interrupt flags.

Table 7-45. Members

Enum value	Description
ADC_INTERRUPT_RESULT_READY	ADC result ready
ADC_INTERRUPT_WINDOW	Window monitor match
ADC_INTERRUPT_OVERRUN	ADC result overwritten before read

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

Enum value	Description
ADC_JOB_READ_BUFFER	Asynchronous ADC read into a user provided buffer

7.5.9. Enum adc_negative_input

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

Table 7-47. Members

Enum value	Description
ADC_NEGATIVE_INPUT_PIN0	ADC0 pin
ADC_NEGATIVE_INPUT_PIN1	ADC1 pin
ADC_NEGATIVE_INPUT_PIN2	ADC2 pin
ADC_NEGATIVE_INPUT_PIN3	ADC3 pin
ADC_NEGATIVE_INPUT_PIN4	ADC4 pin
ADC_NEGATIVE_INPUT_PIN5	ADC5 pin
ADC_NEGATIVE_INPUT_PIN6	ADC6 pin
ADC_NEGATIVE_INPUT_PIN7	ADC7 pin
ADC_NEGATIVE_INPUT_GND	Internal ground

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

Enum value	Description
ADC_OVERSAMPLING_AND_DECIMATION_DISABLE	Don't use oversampling and decimation mode
ADC_OVERSAMPLING_AND_DECIMATION_1BIT	1-bit resolution increase

Enum value	Description
ADC_OVERSAMPLING_AND_DECIMATION_2BIT	2-bit resolution increase
ADC_OVERSAMPLING_AND_DECIMATION_3BIT	3-bit resolution increase
ADC_OVERSAMPLING_AND_DECIMATION_4BIT	4-bit resolution increase

7.5.11. Enum adc_positive_input

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

Table 7-49. Members

Enum value	Description
ADC_POSITIVE_INPUT_PIN0	ADC0 pin
ADC_POSITIVE_INPUT_PIN1	ADC1 pin
ADC_POSITIVE_INPUT_PIN2	ADC2 pin
ADC_POSITIVE_INPUT_PIN3	ADC3 pin
ADC_POSITIVE_INPUT_PIN4	ADC4 pin
ADC_POSITIVE_INPUT_PIN5	ADC5 pin
ADC_POSITIVE_INPUT_PIN6	ADC6 pin
ADC_POSITIVE_INPUT_PIN7	ADC7 pin
ADC_POSITIVE_INPUT_PIN8	ADC8 pin
ADC_POSITIVE_INPUT_PIN9	ADC9 pin
ADC_POSITIVE_INPUT_PIN10	ADC10 pin
ADC_POSITIVE_INPUT_PIN11	ADC11 pin
ADC_POSITIVE_INPUT_PIN12	ADC12 pin
ADC_POSITIVE_INPUT_PIN13	ADC13 pin
ADC_POSITIVE_INPUT_PIN14	ADC14 pin
ADC_POSITIVE_INPUT_PIN15	ADC15 pin
ADC_POSITIVE_INPUT_PIN16	ADC16 pin
ADC_POSITIVE_INPUT_PIN17	ADC17 pin
ADC_POSITIVE_INPUT_PIN18	ADC18 pin
ADC_POSITIVE_INPUT_PIN19	ADC19 pin
ADC_POSITIVE_INPUT_PIN20	ADC20 pin.

Enum value	Description
ADC_POSITIVE_INPUT_PIN21	ADC21 pin
ADC_POSITIVE_INPUT_PIN22	ADC22 pin
ADC_POSITIVE_INPUT_PIN23	ADC23 pin
ADC_POSITIVE_INPUT_TEMP	Temperature reference.
ADC_POSITIVE_INPUT_BANDGAP	Bandgap voltage
ADC_POSITIVE_INPUT_SCALEDCOREVCC	1/4 scaled core supply
ADC_POSITIVE_INPUT_SCALEDIOVCC	1/4 scaled I/O supply
ADC_POSITIVE_INPUT_DAC	DAC input

7.5.12. Enum adc_reference

Enum for the possible reference voltages for the ADC.

Table 7-50. Members

Enum value	Description
ADC_REFERENCE_INTREF	Internal Bandgap Reference
ADC_REFERENCE_INTVCC0	1/1.48V_{CC} reference
ADC_REFERENCE_INTVCC1	1/2V_{CC} (only for internal V_{CC} > 2.1V)
ADC_REFERENCE_AREFA	External reference A
ADC_REFERENCE_INTVCC2	VDDANA

7.5.13. Enum adc_resolution

Enum for the possible resolution values for the ADC.

Table 7-51. Members

Enum value	Description
ADC_RESOLUTION_12BIT	ADC 12-bit resolution
ADC_RESOLUTION_16BIT	ADC 16-bit resolution using oversampling and decimation
ADC_RESOLUTION_10BIT	ADC 10-bit resolution
ADC_RESOLUTION_8BIT	ADC 8-bit resolution
ADC_RESOLUTION_13BIT	ADC 13-bit resolution using oversampling and decimation
ADC_RESOLUTION_14BIT	ADC 14-bit resolution using oversampling and decimation

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

Enum value	Description
ADC_WINDOW_MODE_DISABLE	No window mode
ADC_WINDOW_MODE_ABOVE_LOWER	RESULT > WINLT
ADC_WINDOW_MODE_BELOW_UPPER	RESULT < WINUT
ADC_WINDOW_MODE_BETWEEN	WINLT < RESULT < WINUT
ADC_WINDOW_MODE_BETWEEN_INVERTED	!(WINLT < RESULT < WINUT)

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.

Acronym	Description
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
LSB	Least Significant Bit
MSB	Most Significant Bit
DMA	Direct Memory Access

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.

Changelog
Initial Release

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:

```
struct adc_module adc_instance;
```

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

```
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();
```

9.1.1.3. Workflow

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

      ```
      #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.

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

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
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);

while (1) {
```

```
    /* Infinite loop */
}
```

9.1.2.2. Workflow

1. Start conversion.

   ```
   adc_start_conversion(&adc_instance);
   ```

2. Wait until conversion is done and read result.

   ```
   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.

   ```
   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:

```
struct adc_module adc_instance;
```

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

Callback function:

```
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:

```
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;
#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;

#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()`):

```
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.

   ```
   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.

   ```
   #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.

   ```
   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.

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

      ```
      #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.

      ```
      #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);
      ```

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.

    ```
    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.

    ```
    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:

```
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.

```
system_interrupt_enable_global();
```

2. Start an asynchronous ADC conversion, to store ADC samples into the global buffer and generate a callback when complete.

```
adc_read_buffer_job(&adc_instance, adc_result_buffer, ADC_SAMPLES);
```

3. Wait until the asynchronous conversion is complete.

```
while (adc_read_done == false) {
    /* Wait for asynchronous ADC read to complete */
}
```

4. Enter an infinite loop once the conversion is complete.

```
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

- SAM C21 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_{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:

```
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:

```
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;
    dac_chan_get_config_defaults(&config_dac_chan);

    dac_chan_set_config(&dac_instance, DAC_CHANNEL_0, &config_dac_chan);

    dac_chan_enable(&dac_instance, DAC_CHANNEL_0);
}

void configure_dma_resource(struct dma_resource *resource)
{
    struct dma_resource_config config;

    dma_get_config_defaults(&config);

#if (SAMC21)
    config.peripheral_trigger = ADC1_DMAC_ID_RESRDY;
#else
    config.peripheral_trigger = ADC_DMAC_ID_RESRDY;
#endif
    config.trigger_action = DMA_TRIGGER_ACTON_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.

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

4. Allocate a DMA resource with the configurations.

```
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.

```
struct dma_descriptor_config descriptor_config;
```

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

```
dma_descriptor_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.

```
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;
```

8. Create the DMA transfer descriptor.

```
dma_descriptor_create(descriptor, &descriptor_config);
```

9. Add DMA descriptor to DMA resource.

```
dma_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:

```
adc_start_conversion(&adc_instance);

dma_start_transfer_job(&example_resource);

while (true) {
}
```

9.3.2.2. Workflow

1. Start ADC conversion.

```
adc_start_conversion(&adc_instance);
```

2. Start the transfer job.

```
dma_start_transfer_job(&example_resource);
```

3. Enter endless loop.

```
while (true) {
}
```

10. Document Revision History

Doc. Rev.	Date	Comments
42451B	12/2015	Added support for SAM L22
42451A	07/2015	Initial document release

Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**