SMART ARM-based MCUs

AT03244: SAM D20/D21/D10/D11/DA1/C21 Digital-to-Analog (DAC) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the conversion of digital values to analog voltage. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripheral is used by this module:

- DAC (Digital-to-Analog Converter)

The following devices can use this module:

- Atmel | SMART SAM D20/D21
- Atmel | SMART SAM D10/D11
- Atmel | SMART SAM DA1
- Atmel | SMART SAM 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

The Digital-to-Analog converter converts a digital value to analog voltage. The SAM DAC module has one channel with 10-bit resolution, and is capable of converting up to 350k samples per second (ksps).

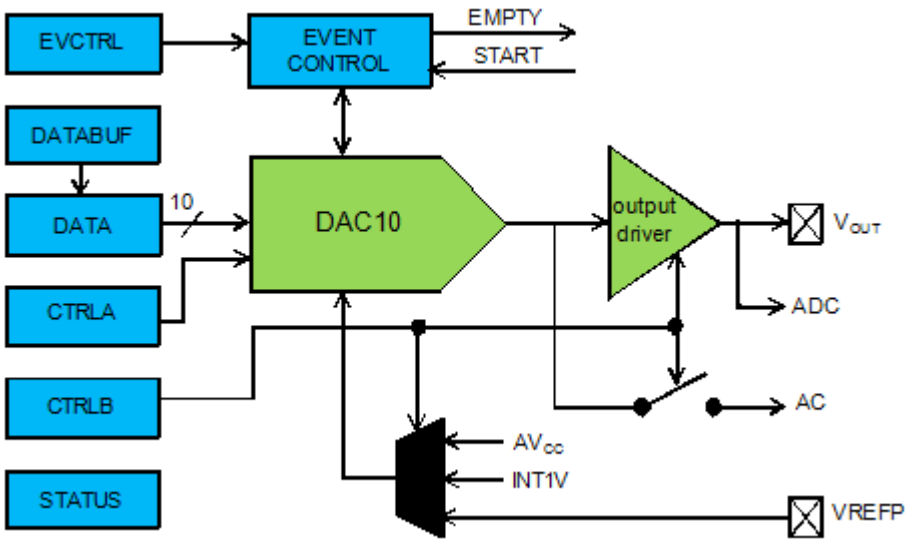
A common use of DAC is to generate audio signals by connecting the DAC output to a speaker, or to generate a reference voltage; either for an external circuit or an internal peripheral such as the Analog Comparator.

After being set up, the DAC will convert new digital values written to the conversion data register (DATA) to an analog value either on the VOUT pin of the device, or internally for use as an input to the AC, ADC, and other analog modules.

Writing the DATA register will start a new conversion. It is also possible to trigger the conversion from the event system.

A simplified block diagram of the DAC can be seen in Figure 3-1 DAC Block Diagram on page 6.

Figure 3-1 DAC Block Diagram



3.1. Conversion Range

The conversion range is between GND and the selected voltage reference. Available voltage references are:

- AVCC voltage reference
- Internal 1V reference (INT1V)
- External voltage reference (AREF)

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.

The output voltage from a DAC channel is given as:

$$V_{OUT} = \frac{DATA}{0x3FF} \times VREF$$

3.2. Conversion

The digital value written to the conversion data register (DATA) will be converted to an analog value. Writing the DATA register will start a new conversion. It is also possible to write the conversion data to the DATABUF register, the writing of the DATA register can then be triggered from the event system, which will load the value from DATABUF to DATA.

3.3. Analog Output

The analog output value can be output to either the VOUT pin or internally, but not both at the same time.

3.3.1. External Output

The output buffer must be enabled in order to drive the DAC output to the VOUT pin. Due to the output buffer, the DAC has high drive strength, and is capable of driving both resistive and capacitive loads, as well as loads which combine both.

3.3.2. Internal Output

The analog value can be internally available for use as input to the AC or ADC modules.

3.4. Events

Events generation and event actions are configurable in the DAC. The DAC has one event line input and one event output: *Start Conversion* and *Data Buffer Empty*.

If the Start Conversion input event is enabled in the module configuration, an incoming event will load data from the data buffer to the data register and start a new conversion. This method synchronizes conversions with external events (such as those from a timer module) and ensures regular and fixed conversion intervals.

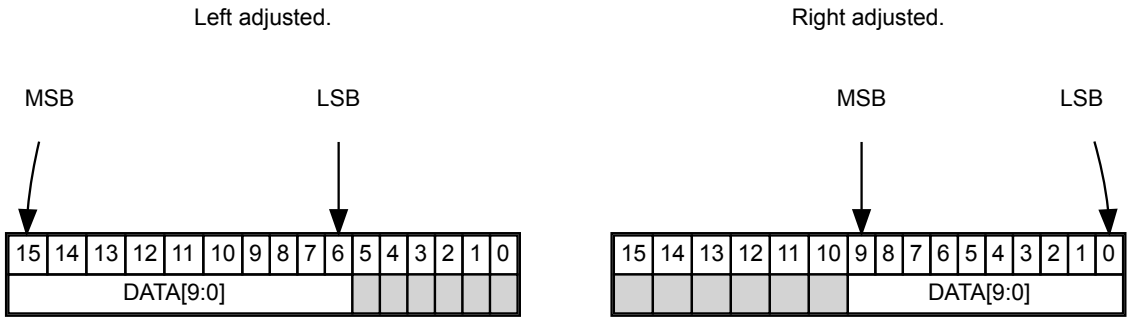
If the Data Buffer Empty output event is enabled in the module configuration, events will be generated when the DAC data buffer register becomes empty and new data can be loaded to the buffer.

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

3.5. Left and Right Adjusted Values

The 10-bit input value to the DAC is contained in a 16-bit register. This can be configured to be either left or right adjusted. In Figure 3-2 Left and Right Adjusted Values on page 8 both options are shown, and the position of the most (MSB) and the least (LSB) significant bits are indicated. The unused bits should always be written to zero.

Figure 3-2 Left and Right Adjusted Values



3.6. Clock Sources

The clock for the DAC interface (CLK_DAC) is generated by the Power Manager. This clock is turned on by default, and can be enabled and disabled in the Power Manager.

Additionally, an asynchronous clock source (GCLK_DAC) is required. These clocks are normally disabled by default. The selected clock source must be enabled in the Power Manager before it can be used by the DAC. The DAC core operates asynchronously from the user interface and peripheral bus. As a consequence, the DAC needs two clock cycles of both CLK_DAC and GCLK_DAC to synchronize the values written to some of the control and data registers. The oscillator source for the GCLK_DAC clock is selected in the System Control Interface (SCIF).

4. Special Considerations

4.1. Output Driver

The DAC can only do conversions in Active or Idle modes. However, if the output buffer is enabled it will draw current even if the system is in sleep mode. Therefore, always make sure that the output buffer is not enabled when it is not needed, to ensure minimum power consumption.

4.2. Conversion Time

DAC conversion time is approximately 2.85µs. The user must ensure that new data is not written to the DAC before the last conversion is complete. Conversions should be triggered by a periodic event from a Timer/Counter or another peripheral.

5. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

For a list of examples related to this driver, see Examples for DAC Driver.

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Type dac_callback_t

```
typedef void(* dac_callback_t )(uint8_t channel)
```

Type definition for a DAC module callback function.

7.2. Structure Definitions

7.2.1. Struct dac_chan_config

Configuration for a DAC channel. This structure should be initialized by the dac_chan_get_config_defaults() function before being modified by the user application.

7.2.2. Struct dac_config

Configuration structure for a DAC instance. This structure should be initialized by the dac_get_config_defaults() function before being modified by the user application.

Table 7-1 Members

Type	Name	Description
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral
bool	left_adjust	Left adjusted data
enum dac_output	output	Select DAC output
enum dac_reference	reference	Reference voltage
bool	run_in_standby	The DAC behaves as in normal mode when the chip enters STANDBY sleep mode
bool	voltage_pump_disable	Voltage pump disable

7.2.3. Struct dac_events

Event flags for the DAC module. This is used to enable and disable events via dac_enable_events() and dac_disable_events().

Table 7-2 Members

Type	Name	Description
bool	generate_event_on_buffer_empty	Enable event generation on data buffer empty
bool	on_event_start_conversion	Start a new DAC conversion

7.2.4. Struct dac_module

DAC 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.

7.3. Macro Definitions

7.3.1. DAC Status Flags

DAC status flags, returned by dac_get_status() and cleared by dac_clear_status().

7.3.1.1. Macro DAC_STATUS_CHANNEL_0_EMPTY

```
#define DAC_STATUS_CHANNEL_0_EMPTY
```

Data Buffer Empty Channel 0 - Set when data is transferred from DATABUF to DATA by a start conversion event and DATABUF is ready for new data.

7.3.1.2. Macro DAC_STATUS_CHANNEL_0_UNDERRUN

```
#define DAC_STATUS_CHANNEL_0_UNDERRUN
```

Under-run Channel 0 - Set when a start conversion event occurs when DATABUF is empty.

7.3.2. Macro DAC_TIMEOUT

```
#define DAC_TIMEOUT
```

Define DAC features set according to different device families.

7.4. Function Definitions

7.4.1. Configuration and Initialization

7.4.1.1. Function dac_is_syncing()

Determines if the hardware module(s) are currently synchronizing to the bus.

```
bool dac_is_syncing(
        struct dac_module *const dev_inst)
```

Checks to see if the underlying hardware peripheral module(s) are currently synchronizing across multiple clock domains to the hardware bus. This function can be used to delay further operations on a module until such time that it is ready, to prevent blocking delays for synchronization in the user application.

Table 7-3 Parameters

Data direction	Parameter name	Description
[in]	dev_inst	Pointer to the DAC software instance struct

Returns

Synchronization status of the underlying hardware module(s).

Table 7-4 Return Values

Return value	Description
true	If the module synchronization is ongoing
false	If the module has completed synchronization

7.4.1.2. Function dac_get_config_defaults()

Initializes a DAC configuration structure to defaults.

```
void dac_get_config_defaults(
        struct dac_config *const config)
```

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

The default configuration is as follows:

- 1V from internal bandgap reference
- Drive the DAC output to the VOUT pin
- Right adjust data
- GCLK generator 0 (GCLK main) clock source
- The output buffer is disabled when the chip enters STANDBY sleep mode

Table 7-5 Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

7.4.1.3. Function dac_init()

Initialize the DAC device struct.

```
enum status_code dac_init(
        struct dac_module *const dev_inst,
        Dac *const module,
        struct dac_config *const config)
```

Use this function to initialize the Digital to Analog Converter. Resets the underlying hardware module and configures it.

Note: The DAC channel must be configured separately.

Table 7-6 Parameters

Data direction	Parameter name	Description
[out]	module_inst	Pointer to the DAC software instance struct
[in]	module	Pointer to the DAC module instance
[in]	config	Pointer to the config struct, created by the user application

Returns
Status of initialization.

Table 7-7 Return Values

Return value	Description
STATUS_OK	Module initiated correctly
STATUS_ERR_DENIED	If module is enabled
STATUS_BUSY	If module is busy resetting

7.4.1.4. Function dac_reset()

Resets the DAC module.

```
void dac_reset(
        struct dac_module *const dev_inst)
```

This function will reset the DAC module to its power on default values and disable it.

Table 7-8 Parameters

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

7.4.1.5. Function dac_enable()

Enable the DAC module.

```
void dac_enable(
        struct dac_module *const dev_inst)
```

Enables the DAC interface and the selected output. If any internal reference is selected it will be enabled.

Table 7-9 Parameters

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

7.4.1.6. Function dac_disable()

Disable the DAC module.

```
void dac_disable(
        struct dac_module *const dev_inst)
```

Disables the DAC interface and the output buffer.

Table 7-10 Parameters

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

7.4.1.7. Function dac_enable_events()

Enables a DAC event input or output.

```
void dac_enable_events(
        struct dac_module *const module_inst,
        struct dac_events *const events)
```

Enables one or more input or output events to or from the DAC module. See dac_events for a list of events this module supports.

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

Table 7-11 Parameters

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

7.4.1.8. Function dac_disable_events()

Disables a DAC event input or output.

```
void dac_disable_events(
        struct dac_module *const module_inst,
        struct dac_events *const events)
```

Disables one or more input or output events to or from the DAC module. See dac_events for a list of events this module supports.

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

Table 7-12 Parameters

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

7.4.2. Configuration and Initialization (Channel)

7.4.2.1. Function dac_chan_get_config_defaults()

Initializes a DAC channel configuration structure to defaults.

```
void dac_chan_get_config_defaults(
        struct dac_chan_config *const config)
```

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

The default configuration is as follows:

- Start Conversion Event Input enabled
- Start Data Buffer Empty Event Output disabled

Table 7-13 Parameters

Data direction	Parameter name	Description
[out]	config	Configuration structure to initialize to default values

7.4.2.2. Function dac_chan_set_config()

Writes a DAC channel configuration to the hardware module.

```
void dac_chan_set_config(
        struct dac_module *const dev_inst,
        const enum dac_channel channel,
        struct dac_chan_config *const config)
```

Writes a given channel configuration to the hardware module.

Note: The DAC device instance structure must be initialized before calling this function.

Table 7-14 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Channel to configure
[in]	config	Pointer to the configuration struct

7.4.2.3. Function dac_chan_enable()

Enable a DAC channel.

```
void dac_chan_enable(
        struct dac_module *const dev_inst,
        enum dac_channel channel)
```

Enables the selected DAC channel.

Table 7-15 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Channel to enable

7.4.2.4. Function dac_chan_disable()

Disable a DAC channel.

```
void dac_chan_disable(
        struct dac_module *const dev_inst,
        enum dac_channel channel)
```

Disables the selected DAC channel.

Table 7-16 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Channel to disable

7.4.3. Channel Data Management

7.4.3.1. Function dac_chan_write()

Write to the DAC.

```
enum status_code dac_chan_write(
        struct dac_module *const dev_inst,
        enum dac_channel channel,
        const uint16_t data)
```

Note: To be event triggered, the enable_start_on_event must be enabled in the configuration.

Table 7-17 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	channel	DAC channel to write to
[in]	data	Conversion data

Returns
Status of the operation.

Table 7-18 Return Values

Return value	Description
STATUS_OK	If the data was written

7.4.3.2. Function dac_chan_write_buffer_wait()

Write to the DAC.

```
enum status_code dac_chan_write_buffer_wait(
        struct dac_module *const module_inst,
        enum dac_channel channel,
        uint16_t * buffer,
        uint32_t length)
```

Note: To be event triggered, the enable_start_on_event must be enabled in the configuration.

Table 7-19 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	channel	DAC channel to write to

Data direction	Parameter name	Description
[in]	buffer	Pointer to the digital data write buffer to be converted
[in]	length	Length of the write buffer

Returns
Status of the operation.

Table 7-20 Return Values

Return value	Description
STATUS_OK	If the data was written or no data conversion required
STATUS_ERR_UNSUPPORTED_DEV	The DAC is not configured as using event trigger
STATUS_BUSY	The DAC is busy to convert

7.4.4. Status Management

7.4.4.1. Function dac_get_status()

Retrieves the current module status.

```
uint32_t dac_get_status(
        struct dac_module *const module_inst)
```

Checks the status of the module and returns it as a bitmask of status flags.

Table 7-21 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct

Returns
Bitmask of status flags.

Table 7-22 Return Values

Return value	Description
DAC_STATUS_CHANNEL_0_EMPTY	Data has been transferred from DATABUF to DATA by a start conversion event and DATABUF is ready for new data
DAC_STATUS_CHANNEL_0_UNDERRUN	A start conversion event has occurred when DATABUF is empty

7.4.4.2. Function dac_clear_status()

Clears a module status flag.

```
void dac_clear_status(
        struct dac_module *const module_inst,
        uint32_t status_flags)
```

Clears the given status flag of the module.

Table 7-23 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	status_flags	Bit mask of status flags to clear

7.4.5. Callback Configuration and Initialization

7.4.5.1. Function dac_chan_write_buffer_job()

Convert a specific number digital data to analog through DAC.

```
enum status_code dac_chan_write_buffer_job(
        struct dac_module *const module_inst,
        const enum dac_channel channel,
        uint16_t * buffer,
        uint32_t buffer_size)
```

Note: To be event triggered, the enable_start_on_event must be enabled in the configuration.

Table 7-24 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	channel	DAC channel to write to
[in]	buffer	Pointer to the digital data write buffer to be converted
[in]	length	Size of the write buffer

Returns
Status of the operation.

Table 7-25 Return Values

Return value	Description
STATUS_OK	If the data was written
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode
STATUS_BUSY	The DAC is busy to accept new job

7.4.5.2. Function dac_chan_write_job()

Convert one digital data job.

```
enum status_code dac_chan_write_job(
        struct dac_module *const module_inst,
        const enum dac_channel channel,
        uint16_t data)
```

Note: To be event triggered, the enable_start_on_event must be enabled in the configuration.

Table 7-26 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software device struct
[in]	channel	DAC channel to write to
[in]	data	Digital data to be converted

Returns
Status of the operation.

Table 7-27 Return Values

Return value	Description
STATUS_OK	If the data was written
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode
STATUS_BUSY	The DAC is busy to accept new job

7.4.5.3. Function dac_register_callback()

Registers an asynchronous callback function with the driver.

```
enum status_code dac_register_callback(
        struct dac_module *const module,
        const enum dac_channel channel,
        const dac_callback_t callback,
        const enum dac_callback type)
```

Registers an asynchronous callback with the DAC driver, fired when a callback condition occurs.

Table 7-28 Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the DAC software instance struct
[in]	callback	Pointer to the callback function to register
[in]	channel	Logical channel to register callback function
[in]	type	Type of callback function to register

Returns
Status of the registration operation.

Table 7-29 Return Values

Return value	Description
STATUS_OK	The callback was registered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode

7.4.5.4. Function dac_unregister_callback()

Unregisters an asynchronous callback function with the driver.

```
enum status_code dac_unregister_callback(
        struct dac_module *const module,
        const enum dac_channel channel,
        const enum dac_callback type)
```

Unregisters an asynchronous callback with the DAC driver, removing it from the internal callback registration table.

Table 7-30 Parameters

Data direction	Parameter name	Description
[in, out]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Logical channel to unregister callback function
[in]	type	Type of callback function to unregister

Returns
Status of the de-registration operation.

Table 7-31 Return Values

Return value	Description
STATUS_OK	The callback was unregistered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode

7.4.6. Callback Enabling and Disabling (Channel)

7.4.6.1. Function dac_chan_enable_callback()

Enables asynchronous callback generation for a given channel and type.

```
enum status_code dac_chan_enable_callback(
        struct dac_module *const module,
        const enum dac_channel channel,
        const enum dac_callback type)
```

Enables asynchronous callbacks for a given logical DAC channel and type. This must be called before a DAC channel will generate callback events.

Table 7-32 Parameters

Data direction	Parameter name	Description
[in, out]	dac_module	Pointer to the DAC software instance struct
[in]	channel	Logical channel to enable callback function
[in]	type	Type of callback function callbacks to enable

Returns
Status of the callback enable operation.

Table 7-33 Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode

7.4.6.2. Function dac_chan_disable_callback()

Disables asynchronous callback generation for a given channel and type.

```
enum status_code dac_chan_disable_callback(
        struct dac_module *const module,
        const enum dac_channel channel,
        const enum dac_callback type)
```

Disables asynchronous callbacks for a given logical DAC channel and type.

Table 7-34 Parameters

Data direction	Parameter name	Description
[in, out]	dac_module	Pointer to the DAC software instance struct
[in]	channel	Logical channel to disable callback function
[in]	type	Type of callback function callbacks to disable

Returns
Status of the callback disable operation.

Table 7-35 Return Values

Return value	Description
STATUS_OK	The callback was disabled successfully
STATUS_ERR_UNSUPPORTED_DEV	If a callback that requires event driven mode was specified with a DAC instance configured in non-event mode

7.4.6.3. Function dac_chan_get_job_status()

Gets the status of a job.

```
enum status_code dac_chan_get_job_status(
        struct dac_module * module_inst,
        const enum dac_channel channel)
```

Gets the status of an ongoing or the last job.

Table 7-36 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Logical channel to enable callback function

Returns
Status of the job.

7.4.6.4. Function dac_chan_abort_job()

Aborts an ongoing job.

```
void dac_chan_abort_job(
        struct dac_module * module_inst,
        const enum dac_channel channel)
```

Aborts an ongoing job.

Table 7-37 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	Logical channel to enable callback function

7.4.7. Configuration and Initialization (Channel)

7.4.7.1. Function dac_chan_enable_output_buffer()

Enable the output buffer.

```
void dac_chan_enable_output_buffer(
        struct dac_module *const dev_inst,
        const enum dac_channel channel)
```

Enables the output buffer and drives the DAC output to the VOUT pin.

Table 7-38 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	DAC channel to alter

7.4.7.2. Function dac_chan_disable_output_buffer()

Disable the output buffer.

```
void dac_chan_disable_output_buffer(
        struct dac_module *const dev_inst,
        const enum dac_channel channel)
```

Disables the output buffer.

Note: The output buffer(s) should be disabled when a channel's output is not currently needed, as it will draw current even if the system is in sleep mode.

Table 7-39 Parameters

Data direction	Parameter name	Description
[in]	module_inst	Pointer to the DAC software instance struct
[in]	channel	DAC channel to alter

7.5. Enumeration Definitions

7.5.1. Enum dac_callback

Enum for the possible callback types for the DAC module.

Table 7-40 Members

Enum value	Description
DAC_CALLBACK_DATA_EMPTY	Callback type for when a DAC channel data empty condition occurs (requires event triggered mode)
DAC_CALLBACK_DATA_UNDERRUN	Callback type for when a DAC channel data underrun condition occurs (requires event triggered mode)
DAC_CALLBACK_TRANSFER_COMPLETE	Callback type for when a DAC channel write buffer job complete (requires event triggered mode)

7.5.2. Enum dac_channel

Enum for the DAC channel selection.

Table 7-41 Members

Enum value	Description
DAC_CHANNEL_0	DAC output channel 0

7.5.3. Enum dac_output

Enum for the DAC output selection.

Table 7-42 Members

Enum value	Description
DAC_OUTPUT_EXTERNAL	DAC output to VOUT pin
DAC_OUTPUT_INTERNAL	DAC output as internal reference
DAC_OUTPUT_NONE	No output

7.5.4. Enum dac_reference

Enum for the possible reference voltages for the DAC.

Table 7-43 Members

Enum value	Description
DAC_REFERENCE_INT1V	1V from the internal band-gap reference
DAC_REFERENCE_AVCC	Analog V_{CC} as reference
DAC_REFERENCE_AREF	External reference on AREF

8. Extra Information for DAC Driver

8.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
ADC	Analog-to-Digital Converter
AC	Analog Comparator
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
Add configuration for using 14-bit hardware dithering (SAMC21 support)
Added new configuration parameters `databuf_protection_bypass`, `voltage_pump_disable`. Added new callback functions `dac_chan_write_buffer_wait`, `dac_chan_write_buffer_job`, `dac_chan_write_job`, `dac_get_job_status`, `dac_abort_job` and new callback type `DAC_CALLBACK_TRANSFER_COMPLETE` for DAC conversion job
Initial Release

9. Examples for DAC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Digital-to-Analog (DAC) 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 DAC - Basic
- Quick Start Guide for DAC - Callback

9.1. Quick Start Guide for DAC - Basic

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

- Analog V_{CC} as reference
- Internal output disabled
- Drive the DAC output to the V_{OUT} pin
- Right adjust data
- The output buffer is disabled when the chip enters STANDBY sleep mode

9.1.1. Quick Start

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 dac_module dac_instance;
```

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

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

    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);
}
```

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

```
configure_dac();
configure_dac_channel();
```

9.1.1.3. Workflow

1. Create a module software instance structure for the DAC module to store the DAC driver state while 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. 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);
      ```

4. Enable the DAC module.

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

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
uint16_t i = 0;

while (1) {
    dac_chan_write(&dac_instance, DAC_CHANNEL_0, i);

    if (++i == 0x3FF) {
        i = 0;
    }
}
```

9.1.2.2. Workflow

1. Create a temporary variable to track the current DAC output value.

```
uint16_t i = 0;
```

2. Enter an infinite loop to continuously output new conversion values to the DAC.

```
while (1) {
```

3. Write the next conversion value to the DAC, so that it will be output on the device's DAC analog output pin.

```
dac_chan_write(&dac_instance, DAC_CHANNEL_0, i);
```

4. Increment and wrap the DAC output conversion value, so that a ramp pattern will be generated.

```
if (++i == 0x3FF) {
    i = 0;
}
```

9.2. Quick Start Guide for DAC - Callback

In this use case, the DAC will convert 16 samples using interrupt driven conversion. When all samples have been sampled, a callback will be called that signals the main application that conversion is compete.

The DAC will be set up as follows:

- Analog V_{CC} as reference
- Internal output disabled
- Drive the DAC output to the V_{OUT} pin
- Right adjust data
- The output buffer is disabled when the chip enters STANDBY sleep mode
- DAC conversion is started with RTC overflow event

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:

```
#define DATA_LENGTH (16)
```

```
struct dac_module dac_instance;
```

```
struct rtc_module rtc_instance;
```

```
struct events_resource event_dac;
```

```
static volatile bool transfer_is_done = false;
```

```
static uint16_t dac_data[DATA_LENGTH];
```

Callback function:

```
void dac_callback(uint8_t channel)
{
    UNUSED(channel);

    transfer_is_done = true;
}
```

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

```
void configure_rtc_count(void)
{
    struct rtc_count_events  rtc_event;

    struct rtc_count_config config_rtc_count;

    rtc_count_get_config_defaults(&config_rtc_count);

    config_rtc_count.prescaler           = RTC_COUNT_PRESCALER_DIV_1;
    config_rtc_count.mode                = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif

    rtc_count_init(&rtc_instance, RTC, &config_rtc_count);

    rtc_event.generate_event_on_overflow = true;

    rtc_count_enable_events(&rtc_instance, &rtc_event);

    rtc_count_enable(&rtc_instance);
}
```

```
void configure_dac(void)
{
    struct dac_config config_dac;

    dac_get_config_defaults(&config_dac);

#if (SAML21)
    dac_instance.start_on_event[DAC_CHANNEL_0] = true;
#else
    dac_instance.start_on_event = true;
#endif

    dac_init(&dac_instance, DAC, &config_dac);

    struct dac_events events =
#if (SAML21)
        { .on_event_chan0_start_conversion = true };
#else
        { .on_event_start_conversion = true };
#endif

    dac_enable_events(&dac_instance, &events);
}
```

```
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);
}
```

Define a data length variables and add to user application (typically the start of `main()`):

```
uint32_t i;
```

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

```
configure_rtc_count();

rtc_count_set_period(&rtc_instance, 1);

configure_dac();

configure_dac_channel();

dac_enable(&dac_instance);

configure_event_resource();

dac_register_callback(&dac_instance, DAC_CHANNEL_0,
        dac_callback,DAC_CALLBACK_TRANSFER_COMPLETE);

dac_chan_enable_callback(&dac_instance, DAC_CHANNEL_0,
        DAC_CALLBACK_TRANSFER_COMPLETE);

for (i = 0;i < DATA_LENGTH;i++) {
    dac_data[i] = 0xfff * i;
}
```

9.2.1.3. Workflow

1. Create a module software instance structure for the DAC module to store the DAC driver state while 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. RTC module is used as the event trigger for DAC in this case, create a module software instance structure for the RTC module to store the RTC driver state.

   ```
   struct rtc_module rtc_instance;
   ```

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

3. Create a buffer for the DAC samples to be converted by the driver.

   ```
   static uint16_t dac_data[DATA_LENGTH];
   ```

4. Create a callback function that will be called when DAC completes convert job.

   ```
   void dac_callback(uint8_t channel)
   {
   ```

```
    UNUSED(channel);

    transfer_is_done = true;
}
```

5. 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. Configure the DAC module with starting conversion on event.

        ```
        #if (SAML21)
            dac_instance.start_on_event[DAC_CHANNEL_0] = true;
        #else
            dac_instance.start_on_event = true;
        #endif
        ```

    4. Initialize the DAC module.

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

    5. Enable DAC start on conversion mode.

        ```
        struct dac_events events =
        #if (SAML21)
                { .on_event_chan0_start_conversion = true };
        #else
                { .on_event_start_conversion = true };
        #endif
        ```

    6. Enable DAC event.

        ```
        dac_enable_events(&dac_instance, &events);
        ```

6. 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);
        ```

7. Enable DAC module.

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

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

```
struct rtc_count_events  rtc_event;
```

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

```
struct rtc_count_config config_rtc_count;
```

    3. Initialize the RTC configuration struct with the module's default values.

```
rtc_count_get_config_defaults(&config_rtc_count);
```

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

    4. Change the RTC module configuration to suit the application.

```
config_rtc_count.prescaler           = RTC_COUNT_PRESCALER_DIV_1;
config_rtc_count.mode                = RTC_COUNT_MODE_16BIT;
#ifdef FEATURE_RTC_CONTINUOUSLY_UPDATED
    config_rtc_count.continuously_update = true;
#endif
```

    5. Initialize the RTC module.

```
rtc_count_init(&rtc_instance, RTC, &config_rtc_count);
```

    6. Configure the RTC module with overflow event.

```
rtc_event.generate_event_on_overflow = true;
```

    7. Enable RTC module overflow event.

```
rtc_count_enable_events(&rtc_instance, &rtc_event);
```

    8. Enable RTC module.

```
rtc_count_enable(&rtc_instance);
```

9. Configure the Event resource.
    1. Create an event resource config struct, which can be filled out to adjust the configuration of a physical event peripheral.

```
struct events_config event_config;
```

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

```
events_get_config_defaults(&event_config);
```

    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 event module configuration to suit the application.

```
event_config.generator      = EVSYS_ID_GEN_RTC_OVF;
event_config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
event_config.path           = EVENTS_PATH_ASYNCHRONOUS;
event_config.clock_source   = GCLK_GENERATOR_0;
```

4. Allocate the event resource.

```
events_allocate(&event_dac, &event_config);
```

5. Attach the event resource with user DAC start.

```
#if (SAML21)
    events_attach_user(&event_dac, EVSYS_ID_USER_DAC_START_0);
#else
    events_attach_user(&event_dac, EVSYS_ID_USER_DAC_START);
#endif
```

10. Register and enable the DAC Write Buffer Complete callback handler.
    1. Register the user-provided Write Buffer Complete callback function with the driver, so that it will be run when an asynchronous buffer write job completes.

```
dac_register_callback(&dac_instance, DAC_CHANNEL_0,
        dac_callback,DAC_CALLBACK_TRANSFER_COMPLETE);
```

    2. Enable the Read Buffer Complete callback so that it will generate callbacks.

```
dac_chan_enable_callback(&dac_instance, DAC_CHANNEL_0,
        DAC_CALLBACK_TRANSFER_COMPLETE);
```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

```
dac_chan_write_buffer_job(&dac_instance, DAC_CHANNEL_0,
        dac_data, DATA_LENGTH);

while (!transfer_is_done) {
    /* Wait for transfer done */
}

while (1) {
}
```

9.2.2.2. Workflow

1. Start a DAC conversion and generate a callback when complete.

```
dac_chan_write_buffer_job(&dac_instance, DAC_CHANNEL_0,
        dac_data, DATA_LENGTH);
```

2. Wait until the conversion is complete.

```
while (!transfer_is_done) {
    /* Wait for transfer done */
}
```

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

```
while (1) {
}
```

10. Document Revision History

Doc. Rev.	Date	Comments
42110E	09/2015	Add SAM C21 and SAM DA1 support
42110D	12/2014	Add SAM D10/D11 support
42110C	01/2014	Add SAM D21 support
42110B	06/2013	Added additional documentation on the event system. Corrected documentation typos.
42110A	06/2013	Initial document release

Atmel | 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**