Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the Direct Memory Access Controller (DMAC) module within the device. The DMAC can transfer data between memories and peripherals, and thus off-load these tasks from the CPU. The module supports peripheral to peripheral, peripheral to memory, memory to peripheral, and memory to memory transfers.

The following peripheral is used by the DMAC Driver:
  • DMAC (Direct Memory Access Controller)

The following devices can use this module:
  • Atmel | SMART SAM D21
  • Atmel | SMART SAM R21
  • Atmel | SMART SAM D09/D10/D11
  • Atmel | SMART SAM L21/L22
  • Atmel | SMART SAM DA1
  • Atmel | SMART SAM C20/C21

The outline of this documentation is as follows:
  • Prerequisites
  • Module Overview
  • Special Considerations
  • Extra Information
  • Examples
  • API Overview
# Table of Contents

- Introduction .................................................................................................................. 1
- 1. Software License ......................................................................................................... 3
- 2. Prerequisites .............................................................................................................. 4
- 3. Module Overview ....................................................................................................... 5  
  3.1. Driver Feature Macro Definition .............................................................................. 6  
  3.2. Terminology Used in DMAC Transfers ................................................................... 6  
  3.3. DMA Channels ......................................................................................................... 6  
  3.4. DMA Triggers ........................................................................................................... 7  
  3.5. DMA Transfer Descriptor ......................................................................................... 7  
  3.6. DMA Interrupts/Events ............................................................................................ 7  
- 4. Special Considerations .............................................................................................. 8  
- 5. Extra Information ...................................................................................................... 9  
- 6. Examples .................................................................................................................. 10  
- 7. API Overview .......................................................................................................... 11  
  7.1. Variable and Type Definitions ................................................................................. 11  
  7.2. Structure Definitions ............................................................................................... 11  
  7.3. Macro Definitions .................................................................................................... 13  
  7.4. Function Definitions ............................................................................................... 13  
  7.5. Enumeration Definitions ......................................................................................... 21  
- 8. Extra Information for DMAC Driver ........................................................................ 24  
  8.1. Acronyms ................................................................................................................ 24  
  8.2. Dependencies ......................................................................................................... 24  
  8.3. Errata ..................................................................................................................... 24  
  8.4. Module History ....................................................................................................... 24  
- 9. Examples for DMAC Driver ..................................................................................... 25  
  9.1. Quick Start Guide for Memory to Memory Data Transfer Using DMAC ........... 25  
- 10. Document Revision History ................................................................................... 29
1. **Software License**

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

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

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

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

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

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

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

SAM devices with DMAC enables high data transfer rates with minimum CPU intervention and frees up CPU time. With access to all peripherals, the DMAC can handle automatic transfer of data to/from modules. It supports static and incremental addressing for both source and destination.

The DMAC when used with Event System or peripheral triggers, provides a considerable advantage by reducing the power consumption and performing data transfer in the background. For example, if the ADC is configured to generate an event, it can trigger the DMAC to transfer the data into another peripheral or SRAM. The CPU can remain in sleep during this time to reduce the power consumption.

<table>
<thead>
<tr>
<th>Device</th>
<th>Dma channel number</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAM D21/R21/C20/C21</td>
<td>12</td>
</tr>
<tr>
<td>SAM D09/D10/D11</td>
<td>6</td>
</tr>
<tr>
<td>SAM L21</td>
<td>16</td>
</tr>
</tbody>
</table>

The DMA channel operation can be suspended at any time by software, by events from event system, or after selectable descriptor execution. The operation can be resumed by software or by events from the event system. The DMAC driver for SAM supports four types of transfers such as peripheral to peripheral, peripheral to memory, memory to peripheral, and memory to memory.

The basic transfer unit is a beat, which is defined as a single bus access. There can be multiple beats in a single block transfer and multiple block transfers in a DMA transaction. DMA transfer is based on descriptors, which holds transfer properties such as the source and destination addresses, transfer counter, and other additional transfer control information. The descriptors can be static or linked. When static, a single block transfer is performed. When linked, a number of transfer descriptors can be used to enable multiple block transfers within a single DMA transaction.

The implementation of the DMA driver is based on the idea that the DMA channel is a finite resource of entities with the same abilities. A DMA channel resource is able to move a defined set of data from a source address to destination address triggered by a transfer trigger. On the SAM devices there are 12 DMA resources available for allocation. Each of these DMA resources can trigger interrupt callback routines and peripheral events. The other main features are:

- Selectable transfer trigger source
  - Software
  - Event System
  - Peripheral
- Event input and output is supported for the four lower channels
- Four level channel priority
- Optional interrupt generation on transfer complete, channel error, or channel suspend
- Supports multi-buffer or circular buffer mode by linking multiple descriptors
- Beat size configurable as 8-bit, 16-bit, or 32-bit

A simplified block diagram of the DMA Resource can be seen in Figure 3-1 Module Overview on page 6.
3.1. Driver Feature Macro Definition

<table>
<thead>
<tr>
<th>Driver Feature Macro</th>
<th>Supported devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEATURE_DMA_CHANNEL_STANDBY</td>
<td>SAM L21/L22/C20/C21</td>
</tr>
</tbody>
</table>

Note: The specific features are only available in the driver when the selected device supports those features.

3.2. Terminology Used in DMAC Transfers

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beat</td>
<td>It is a single bus access by the DMAC. Configurable as 8-bit, 16-bit, or 32-bit.</td>
</tr>
<tr>
<td>Burst</td>
<td>It is a transfer of n-beats (n=1,4,8,16). For the DMAC module in SAM, the burst size is one beat. Arbitration takes place each time a burst transfer is completed.</td>
</tr>
<tr>
<td>Block transfer</td>
<td>A single block transfer is a configurable number of (1 to 64k) beat transfers</td>
</tr>
</tbody>
</table>

3.3. DMA Channels

The DMAC in each device consists of several DMA channels, which along with the transfer descriptors defines the data transfer properties.

- The transfer control descriptor defines the source and destination addresses, source and destination address increment settings, the block transfer count, and event output condition selection
- Dedicated channel registers control the peripheral trigger source, trigger mode settings, event input actions, and channel priority level settings

With a successful DMA resource allocation, a dedicated DMA channel will be assigned. The channel will be occupied until the DMA resource is freed. A DMA resource handle is used to identify the specific DMA resource. When there are multiple channels with active requests, the arbiter prioritizes the channels requesting access to the bus.
3.4. **DMA Triggers**

DMA transfer can be started only when a DMA transfer request is acknowledged/granted by the arbiter. A transfer request can be triggered from software, peripheral, or an event. There are dedicated source trigger selections for each DMA channel usage.

3.5. **DMA Transfer Descriptor**

The transfer descriptor resides in the SRAM and defines these channel properties.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Field width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptor Next Address</td>
<td>32 bits</td>
</tr>
<tr>
<td>Destination Address</td>
<td>32 bits</td>
</tr>
<tr>
<td>Source Address</td>
<td>32 bits</td>
</tr>
<tr>
<td>Block Transfer Counter</td>
<td>16 bits</td>
</tr>
<tr>
<td>Block Transfer Control</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Before starting a transfer, at least one descriptor should be configured. After a successful allocation of a DMA channel, the transfer descriptor can be added with a call to `dma_add_descriptor()`. If there is a transfer descriptor already allocated to the DMA resource, the descriptor will be linked to the next descriptor address.

3.6. **DMA Interrupts/Events**

Both an interrupt callback and a peripheral event can be triggered by the DMA transfer. Three types of callbacks are supported by the DMA driver: transfer complete, channel suspend, and transfer error. Each of these callback types can be registered and enabled for each channel independently through the DMA driver API.

The DMAC module can also generate events on transfer complete. Event generation is enabled through the DMA channel, event channel configuration, and event user multiplexing is done through the events driver.

The DMAC can generate events in the below cases:

- When a block transfer is complete
- When each beat transfer within a block transfer is complete
4. **Special Considerations**

There are no special considerations for this module.
5. **Extra Information**

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

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

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

7.1. **Variable and Type Definitions**

7.1.1. **Type dma_callback_t**

```c
typedef void(* dma_callback_t )(struct dma_resource * const resource)
```

Type definition for a DMA resource callback function.

7.1.2. **Variable descriptor_section**

```c
DmacDescriptor descriptor_section
```

ExInitial description section.

7.1.3. **Variable g_chan_interrupt_flag**

```c
uint8_t g_chan_interrupt_flag
```

7.2. **Structure Definitions**

7.2.1. **Struct dma_descriptor_config**

DMA transfer descriptor configuration. When the source or destination address increment is enabled, the addresses stored into the configuration structure must correspond to the end of the transfer.

**Table 7-1. Members**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum dma_beat_size</td>
<td>beat_size</td>
<td>Beat size is configurable as 8-bit, 16-bit, or 32-bit</td>
</tr>
<tr>
<td>enum dma_block_action</td>
<td>block_action</td>
<td>Action taken when a block transfer is completed</td>
</tr>
<tr>
<td>uint16_t</td>
<td>block_transfer_count</td>
<td>It is the number of beats in a block. This count value is decremented by one after each beat data transfer.</td>
</tr>
<tr>
<td>bool</td>
<td>descriptor_valid</td>
<td>Descriptor valid flag used to identify whether a descriptor is valid or not</td>
</tr>
<tr>
<td>uint32_t</td>
<td>destination_address</td>
<td>Transfer destination address</td>
</tr>
</tbody>
</table>
### Struct dma_events_config

Configurations for DMA events.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>event_output_enable</td>
<td>Enable DMA event output</td>
</tr>
<tr>
<td>enum dma_event_input_action</td>
<td>input_action</td>
<td>Event input actions</td>
</tr>
</tbody>
</table>

#### 7.2.3. Struct dma_resource

Structure for DMA transfer resource.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dma_callback_t</td>
<td>callback[]</td>
<td>Array of callback functions for DMA transfer job</td>
</tr>
<tr>
<td>uint8_t</td>
<td>callback_enable</td>
<td>Bit mask for enabled callbacks</td>
</tr>
<tr>
<td>uint8_t</td>
<td>channel_id</td>
<td>Allocated DMA channel ID</td>
</tr>
<tr>
<td>DmacDescriptor *</td>
<td>descriptor</td>
<td>DMA transfer descriptor</td>
</tr>
</tbody>
</table>
### 7.2.4. Struct dma_resource_config

DMA configurations for transfer.

**Table 7-4. Members**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum status_code</td>
<td>job_status</td>
<td>Status of the last job</td>
</tr>
<tr>
<td>uint32_t</td>
<td>transferred_size</td>
<td>Transferred data size</td>
</tr>
</tbody>
</table>

### 7.3. Macro Definitions

#### 7.3.1. Macro DMA_INVALID_CHANNEL

```c
#define DMA_INVALID_CHANNEL
```

DMA invalid channel number.

#### 7.3.2. Macro FEATURE_DMA_CHANNEL_STANDBY

```c
#define FEATURE_DMA_CHANNEL_STANDBY
```

### 7.4. Function Definitions

#### 7.4.1. Function dma_abort_job()

Abort a DMA transfer.

```c
void dma_abort_job(
    struct dma_resource * resource)
```

This function will abort a DMA transfer. The DMA channel used for the DMA resource will be disabled. The block transfer count will also be calculated and written to the DMA resource structure.

**Note:** The DMA resource will not be freed after calling this function. The function `dma_free()` can be used to free an allocated resource.
Table 7-5. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in, out]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

7.4.2. Function dma_add_descriptor()

Add a DMA transfer descriptor to a DMA resource.

```c
enum status_code dma_add_descriptor(
    struct dma_resource * resource,
    DmacDescriptor * descriptor)
```

This function will add a DMA transfer descriptor to a DMA resource. If there was a transfer descriptor already allocated to the DMA resource, the descriptor will be linked to the next descriptor address.

Table 7-6. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
<tr>
<td>[in]</td>
<td>descriptor</td>
<td>Pointer to the transfer descriptor</td>
</tr>
</tbody>
</table>

Table 7-7. Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The descriptor is added to the DMA resource</td>
</tr>
<tr>
<td>STATUS_BUSY</td>
<td>The DMA resource was busy and the descriptor is not added</td>
</tr>
</tbody>
</table>

7.4.3. Function dma_allocate()

Allocate a DMA with configurations.

```c
enum status_code dma_allocate(
    struct dma_resource * resource,
    struct dma_resource_config * config)
```

This function will allocate a proper channel for a DMA transfer request.

Table 7-8. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in, out]</td>
<td>dma_resource</td>
<td>Pointer to a DMA resource instance</td>
</tr>
<tr>
<td>[in]</td>
<td>transfer_config</td>
<td>Configurations of the DMA transfer</td>
</tr>
</tbody>
</table>

Returns

Status of the allocation procedure.
Table 7-9. Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The DMA resource was allocated successfully</td>
</tr>
<tr>
<td>STATUS_ERR_NOT_FOUND</td>
<td>DMA resource allocation failed</td>
</tr>
</tbody>
</table>

7.4.4. Function dma_descriptor_create()

Create a DMA transfer descriptor with configurations.

```c
void dma_descriptor_create(
    DmacDescriptor * descriptor,
    struct dma_descriptor_config * config)
```

This function will set the transfer configurations to the DMA transfer descriptor.

Table 7-10. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>descriptor</td>
<td>Pointer to the DMA transfer descriptor</td>
</tr>
<tr>
<td>[in]</td>
<td>config</td>
<td>Pointer to the descriptor configuration structure</td>
</tr>
</tbody>
</table>

7.4.5. Function dma_descriptor_get_config_defaults()

Initializes DMA transfer configuration with predefined default values.

```c
void dma_descriptor_get_config_defaults(
    struct dma_descriptor_config * config)
```

This function will initialize a given DMA descriptor configuration structure to a set of known default values. This function should be called on any new instance of the configuration structure before being modified by the user application.

The default configuration is as follows:

- Set the descriptor as valid
- Disable event output
- No block action
- Set beat size as byte
- Enable source increment
- Enable destination increment
- Step size is applied to the destination address
- Address increment is beat size multiplied by 1
- Default transfer size is set to 0
- Default source address is set to NULL
- Default destination address is set to NULL
- Default next descriptor not available
### Table 7-11. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[out]</td>
<td>config</td>
<td>Pointer to the configuration</td>
</tr>
</tbody>
</table>

#### 7.4.6. Function `dma_disable_callback()`

Disable a callback function for a dedicated DMA resource.

```c
define dma_disable_callback(  
    struct dma_resource * resource,  
    enum dma_callback_type type)  
```

#### Table 7-12. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
<tr>
<td>[in]</td>
<td>type</td>
<td>Callback function type</td>
</tr>
</tbody>
</table>

#### 7.4.7. Function `dma_enable_callback()`

Enable a callback function for a dedicated DMA resource.

```c
define dma_enable_callback(  
    struct dma_resource * resource,  
    enum dma_callback_type type)  
```

#### Table 7-13. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
<tr>
<td>[in]</td>
<td>type</td>
<td>Callback function type</td>
</tr>
</tbody>
</table>

#### 7.4.8. Function `dma_free()`

Free an allocated DMA resource.

```c
define status_code dma_free(  
    struct dma_resource * resource)  
```

This function will free an allocated DMA resource.

#### Table 7-14. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in, out]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

**Returns**

Status of the free procedure.
Table 7-15. Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The DMA resource was freed successfully</td>
</tr>
<tr>
<td>STATUS_BUSY</td>
<td>The DMA resource was busy and can't be freed</td>
</tr>
<tr>
<td>STATUS_ERR_NOT_INITIALIZED</td>
<td>DMA resource was not initialized</td>
</tr>
</tbody>
</table>

7.4.9. Function dma_get_config_defaults()

Initializes config with predefined default values.

```c
void dma_get_config_defaults(
    struct dma_resource_config * config)
```

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

The default configuration is as follows:
- Software trigger is used as the transfer trigger
- Priority level 0
- Only software/event trigger
- Requires a trigger for each transaction
- No event input/output
- DMA channel is disabled during sleep mode (if has the feature)

Table 7-16. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[out]</td>
<td>config</td>
<td>Pointer to the configuration</td>
</tr>
</tbody>
</table>

7.4.10. Function dma_get_job_status()

Get DMA resource status.

```c
enum status_code dma_get_job_status(
    struct dma_resource * resource)
```

Table 7-17. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

Returns
Status of the DMA resource.
7.4.11. Function dma_is_busy()

Check if the given DMA resource is busy.

```c
bool dma_is_busy(
    struct dma_resource * resource)
```

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

Returns

Status which indicates whether the DMA resource is busy.

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>The DMA resource has an on-going transfer</td>
</tr>
<tr>
<td>false</td>
<td>The DMA resource is not busy</td>
</tr>
</tbody>
</table>

7.4.12. Function dma_register_callback()

Register a callback function for a dedicated DMA resource.

```c
void dma_register_callback(
    struct dma_resource * resource,
    dma_callback_t callback,
    enum dma_callback_type type)
```

There are three types of callback functions, which can be registered:

- Callback for transfer complete
- Callback for transfer error
- Callback for channel suspend

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
<tr>
<td>[in]</td>
<td>callback</td>
<td>Pointer to the callback function</td>
</tr>
<tr>
<td>[in]</td>
<td>type</td>
<td>Callback function type</td>
</tr>
</tbody>
</table>

7.4.13. Function dma_reset_descriptor()

Reset DMA descriptor.

```c
void dma_reset_descriptor(
    struct dma_resource * resource)
```

This function will clear the DESCADDR register of an allocated DMA resource.
7.4.14. Function `dma_resume_job()`
Resume a suspended DMA transfer.

```c
void dma_resume_job(
    struct dma_resource * resource)
```

This function try to resume a suspended transfer of a DMA resource.

Table 7-21. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

7.4.15. Function `dma_start_transfer_job()`
Start a DMA transfer.

```c
enum status_code dma_start_transfer_job(
    struct dma_resource * resource)
```

This function will start a DMA transfer through an allocated DMA resource.

Table 7-22. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in, out]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

Returns
Status of the transfer start procedure.

Table 7-23. Return Values

<table>
<thead>
<tr>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS_OK</td>
<td>The transfer was started successfully</td>
</tr>
<tr>
<td>STATUS_BUSY</td>
<td>The DMA resource was busy and the transfer was not started</td>
</tr>
<tr>
<td>STATUS_ERR_INVALID_ARG</td>
<td>Transfer size is 0 and transfer was not started</td>
</tr>
</tbody>
</table>

7.4.16. Function `dma_suspend_job()`
Suspend a DMA transfer.

```c
void dma_suspend_job(
    struct dma_resource * resource)
```

This function will request to suspend the transfer of the DMA resource. The channel is kept enabled, can receive transfer triggers (the transfer pending bit will be set), but will be removed from the arbitration scheme. The channel operation can be resumed by calling `dma_resume_job()`.

Note: This function sets the command to suspend the DMA channel associated with a DMA resource. The channel suspend interrupt flag indicates whether the transfer is truly suspended.
Table 7-24. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

7.4.17. Function dma_trigger_transfer()

Will set a software trigger for resource.

```c
void dma_trigger_transfer(
    struct dma_resource * resource)
```

This function is used to set a software trigger on the DMA channel associated with resource. If a trigger is already pending no new trigger will be generated for the channel.

Table 7-25. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
</tbody>
</table>

7.4.18. Function dma_unregister_callback()

Unregister a callback function for a dedicated DMA resource.

```c
void dma_unregister_callback(
    struct dma_resource * resource,
    enum dma_callback_type type)
```

There are three types of callback functions:

- Callback for transfer complete
- Callback for transfer error
- Callback for channel suspend

The application can unregister any of the callback functions which are already registered and are no longer needed.

Table 7-26. Parameters

<table>
<thead>
<tr>
<th>Data direction</th>
<th>Parameter name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>resource</td>
<td>Pointer to the DMA resource</td>
</tr>
<tr>
<td>[in]</td>
<td>type</td>
<td>Callback function type</td>
</tr>
</tbody>
</table>

7.4.19. Function dma_update_descriptor()

Update DMA descriptor.

```c
void dma_update_descriptor(
    struct dma_resource * resource,
    DmacDescriptor * descriptor)
```

This function can update the descriptor of an allocated DMA resource.
7.5. **Enumeration Definitions**

7.5.1. **Enum dma_address_increment_stepsize**

Address increment step size. These bits select the address increment step size. The setting apply to source or destination address, depending on STEPSEL setting.

**Table 7-27. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_1</td>
<td>The address is incremented by (beat size * 1).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_2</td>
<td>The address is incremented by (beat size * 2).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_4</td>
<td>The address is incremented by (beat size * 4).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_8</td>
<td>The address is incremented by (beat size * 8).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_16</td>
<td>The address is incremented by (beat size * 16).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_32</td>
<td>The address is incremented by (beat size * 32).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_64</td>
<td>The address is incremented by (beat size * 64).</td>
</tr>
<tr>
<td>DMA_ADDRESS_INCREMENT_STEP_SIZE_128</td>
<td>The address is incremented by (beat size * 128).</td>
</tr>
</tbody>
</table>

7.5.2. **Enum dma_beat_size**

The basic transfer unit in DMAC is a beat, which is defined as a single bus access. Its size is configurable and applies to both read and write.

**Table 7-28. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_BEAT_SIZE_BYTE</td>
<td>8-bit access.</td>
</tr>
<tr>
<td>DMA_BEAT_SIZE_HWORD</td>
<td>16-bit access.</td>
</tr>
<tr>
<td>DMA_BEAT_SIZE_WORD</td>
<td>32-bit access.</td>
</tr>
</tbody>
</table>

7.5.3. **Enum dma_block_action**

Block action definitions.

**Table 7-29. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_BLOCK_ACTION_NOACT</td>
<td>No action.</td>
</tr>
<tr>
<td>DMA_BLOCK_ACTION_INT</td>
<td>Channel in normal operation and sets transfer complete interrupt flag after block transfer.</td>
</tr>
</tbody>
</table>
### 7.5.4. Enum dma_callback_type

Callback types for DMA callback driver.

**Table 7-30. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_CALLBACK_TRANSFER_ERROR</td>
<td>Callback for any of transfer errors. A transfer error is flagged if a bus error is detected during an AHB access or when the DMAC fetches an invalid descriptor.</td>
</tr>
<tr>
<td>DMA_CALLBACK_TRANSFER_DONE</td>
<td>Callback for transfer complete.</td>
</tr>
<tr>
<td>DMA_CALLBACK_CHANNEL_SUSPEND</td>
<td>Callback for channel suspend.</td>
</tr>
<tr>
<td>DMA_CALLBACK_N</td>
<td>Number of available callbacks.</td>
</tr>
</tbody>
</table>

### 7.5.5. Enum dma_event_input_action

DMA input actions.

**Table 7-31. Members**

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_EVENT_INPUT_NOACT</td>
<td>No action.</td>
</tr>
<tr>
<td>DMA_EVENT_INPUT_TRIG</td>
<td>Normal transfer and periodic transfer trigger.</td>
</tr>
<tr>
<td>DMA_EVENT_INPUT_CTRIG</td>
<td>Conditional transfer trigger.</td>
</tr>
<tr>
<td>DMA_EVENT_INPUT_CBLOCK</td>
<td>Conditional block transfer.</td>
</tr>
<tr>
<td>DMA_EVENT_INPUT_SUSPEND</td>
<td>Channel suspend operation.</td>
</tr>
<tr>
<td>DMA_EVENT_INPUT_RESUME</td>
<td>Channel resume operation.</td>
</tr>
<tr>
<td>DMA_EVENT_INPUT_SSKIP</td>
<td>Skip next block suspend action.</td>
</tr>
</tbody>
</table>

### 7.5.6. Enum dma_event_output_selection

Event output selection.
### Table 7-32. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_EVENT_OUTPUT_DISABLE</td>
<td>Event generation disable.</td>
</tr>
<tr>
<td>DMA_EVENT_OUTPUT_BLOCK</td>
<td>Event strobe when block transfer complete.</td>
</tr>
<tr>
<td>DMA_EVENT_OUTPUT_RESERVED</td>
<td>Event output reserved.</td>
</tr>
<tr>
<td>DMA_EVENT_OUTPUT_BEAT</td>
<td>Event strobe when beat transfer complete.</td>
</tr>
</tbody>
</table>

#### 7.5.7. Enum dma_priority_level

DMA priority level.

### Table 7-33. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_PRIORITY_LEVEL_0</td>
<td>Priority level 0.</td>
</tr>
<tr>
<td>DMA_PRIORITY_LEVEL_1</td>
<td>Priority level 1.</td>
</tr>
<tr>
<td>DMA_PRIORITY_LEVEL_2</td>
<td>Priority level 2.</td>
</tr>
<tr>
<td>DMA_PRIORITY_LEVEL_3</td>
<td>Priority level 3.</td>
</tr>
</tbody>
</table>

#### 7.5.8. Enum dma_step_selection

DMA step selection. This bit determines whether the step size setting is applied to source or destination address.

### Table 7-34. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_STEPSEL_DST</td>
<td>Step size settings apply to the destination address.</td>
</tr>
<tr>
<td>DMA_STEPSEL_SRC</td>
<td>Step size settings apply to the source address.</td>
</tr>
</tbody>
</table>

#### 7.5.9. Enum dma_transfer_trigger_action

DMA trigger action type.

### Table 7-35. Members

<table>
<thead>
<tr>
<th>Enum value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA_TRIGGER_ACTON_BLOCK</td>
<td>Perform a block transfer when triggered.</td>
</tr>
<tr>
<td>DMA_TRIGGER_ACTON_BEAT</td>
<td>Perform a beat transfer when triggered.</td>
</tr>
<tr>
<td>DMA_TRIGGER_ACTON_TRANSACTION</td>
<td>Perform a transaction when triggered.</td>
</tr>
</tbody>
</table>
8. Extra Information for DMAC Driver

8.1. Acronyms

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

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DMAC</td>
<td>Direct Memory Access Controller</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
</tbody>
</table>

8.2. Dependencies

This driver has the following dependencies:

- System Clock Driver

8.3. Errata

There are no errata related to this driver.

8.4. Module History

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

<table>
<thead>
<tr>
<th>Changelog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add SAM C21 support</td>
</tr>
<tr>
<td>Add SAM L21 support</td>
</tr>
<tr>
<td>Initial Release</td>
</tr>
</tbody>
</table>
9. **Examples for DMAC Driver**

This is a list of the available Quick Start Guides (QSGs) and example applications for **SAM Direct Memory Access Controller (DMAC) 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 Memory to Memory Data Transfer Using DMAC

**Note:** More DMA usage examples are available in peripheral QSGs. A quick start guide for TC/TCC shows the usage of DMA event trigger; SERCOM SPI/USART/I²C has example for DMA transfer from peripheral to memory or from memory to peripheral; ADC/DAC shows peripheral to peripheral transfer.

9.1. **Quick Start Guide for Memory to Memory Data Transfer Using DMAC**

The supported board list:

- SAM D21 Xplained Pro
- SAM R21 Xplained Pro
- SAM D11 Xplained Pro
- SAM L21 Xplained Pro
- SAM L22 Xplained Pro
- SAM DA1 Xplained Pro

In this use case, the DMAC is configured for:

- Moving data from memory to memory
- Using software trigger
- Using DMA priority level 0
- Transaction as DMA trigger action
- No action on input events
- Output event not enabled

9.1.1. **Setup**

9.1.1.1. **Prerequisites**

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

9.1.1.2. **Code**

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

```c
#define DATA_LENGTH (512)
static uint8_t source_memory[DATA_LENGTH];
static uint8_t destination_memory[DATA_LENGTH];
static volatile bool transfer_is_done = false;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor SECTION_DMAC_DESCRIPTOR;

static void transfer_done(struct dma_resource* const resource ) {
```
transfer_is_done = true;
}

static void configure_dma_resource(struct dma_resource *resource) {
    struct dma_resource_config config;
    dma_get_config_defaults(&config);
    dma_allocate(resource, &config);
}

static void setup_transfer_descriptor(DmacDescriptor *descriptor) {
    struct dma_descriptor_config descriptor_config;
    dma_descriptor_get_config_defaults(&descriptor_config);
    descriptor_config.block_transfer_count = sizeof(source_memory);
    descriptor_config.source_address = (uint32_t)source_memory + sizeof(source_memory);
    descriptor_config.destination_address = (uint32_t)destination_memory + sizeof(source_memory);
    dma_descriptor_create(descriptor, &descriptor_config);
}

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

configure_dma_resource(&example_resource);
setup_transfer_descriptor(&example_descriptor);
dma_add_descriptor(&example_resource, &example_descriptor);
dma_register_callback(&example_resource, transfer_done, DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource, DMA_CALLBACK_TRANSFER_DONE);
for (uint32_t i = 0; i < DATA_LENGTH; i++) {
    source_memory[i] = i;
}

9.1.1.3. Workflow

1. Create a DMA resource configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.
   
   ```c
   struct dma_resource_config config;
   ```

2. Initialize the DMA resource configuration struct with the module’s default values.
   
   ```c
   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. Allocate a DMA resource with the configurations.
   
   ```c
   dma_allocate(resource, &config);
   ```
4. Declare a DMA transfer descriptor configuration structure, which can be filled out to adjust the configuration of a single DMA transfer.

```c
struct dma_descriptor_config descriptor_config;
```

5. Initialize the DMA transfer descriptor configuration struct with the module’s default values.

```c
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.

6. Set the specific parameters for a DMA transfer with transfer size, source address, and destination address. In this example, we have enabled the source and destination address increment. The source and destination addresses to be stored into descriptor_config must correspond to the end of the transfer.

```c
descriptor_config.block_transfer_count = sizeof(source_memory);
descriptor_config.source_address = (uint32_t)source_memory +
           sizeof(source_memory);
descriptor_config.destination_address = (uint32_t)destination_memory +
           sizeof(source_memory);
```

7. Create the DMA transfer descriptor.

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

8. Add the DMA transfer descriptor to the allocated DMA resource.

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

9. Register a callback to indicate transfer status.

```c
dma_register_callback(&example_resource, transfer_done,
                   DMA_CALLBACK_TRANSFER_DONE);
```

10. Set the transfer done flag in the registered callback function.

```c
static void transfer_done(struct dma_resource* const resource )
{
    transfer_is_done = true;
}
```

11. Enable the registered callbacks.

```c
dma_enable_callback(&example_resource, DMA_CALLBACK_TRANSFER_DONE);
```

**9.1.2. Use Case**

**9.1.2.1. Code**

Add the following code at the start of `main()`:

```c
struct dma_resource example_resource;
```

Copy the following code to your user application:

```c
dma_start_transfer_job(&example_resource);
dma_trigger_transfer(&example_resource);
while (!transfer_is_done) {
    /* Wait for transfer done */
}
while (true) {
    /* Nothing to do */
}

9.1.2.2. Workflow

1. Start the DMA transfer job with the allocated DMA resource and transfer descriptor.
   
   ```c
   dma_start_transfer_job(&example_resource);
   ```

2. Set the software trigger for the DMA channel. This can be done before or after the DMA job is started. Note that all transfers needs a trigger to start.
   
   ```c
   dma_trigger_transfer(&example_resource);
   ```

3. Waiting for the setting of the transfer done flag.
   
   ```c
   while (!transfer_is_done) {
       /* Wait for transfer done */
   }
   ```
### 10. Document Revision History

<table>
<thead>
<tr>
<th>Doc. Rev.</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>42257C</td>
<td>12/2015</td>
<td>Added support for SAM L21/L22, SAM C21, SAM D09, and SAM DA1</td>
</tr>
<tr>
<td>42257B</td>
<td>12/2014</td>
<td>Added support for SAM R21 and SAM D10/D11</td>
</tr>
<tr>
<td>42257A</td>
<td>02/2014</td>
<td>Initial release</td>
</tr>
</tbody>
</table>