SMART ARM-based Microcontrollers

AT03245: SAM D/R/L/C Event System (EVENTS) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's peripheral event resources and users within the device, including enabling and disabling of peripheral source selection and synchronization of clock domains between various modules. The following API modes is covered by this manual:

- Polled API
- Interrupt hook API

The following peripheral is used by this module:

- EVSYS (Event System Management)

The following devices can use this module:

- Atmel | SMART SAM D20/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

1. Software License

2. Prerequisites

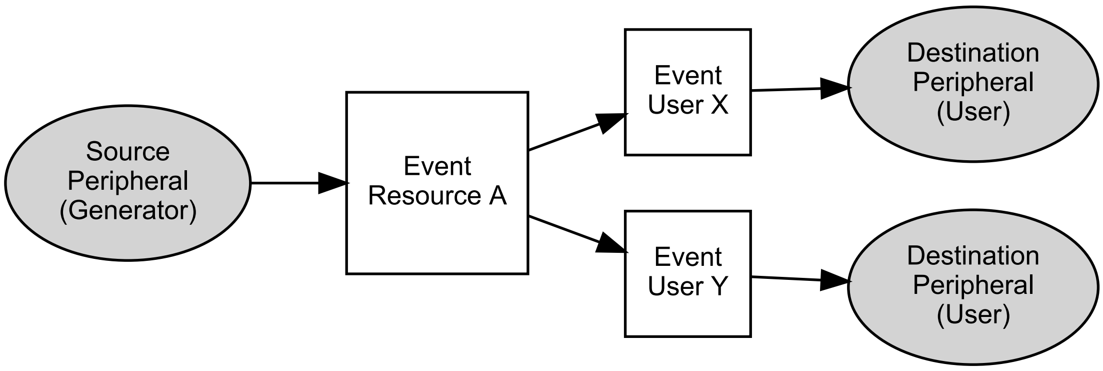
There are no prerequisites for this module.

3. Module Overview

Peripherals within the SAM devices are capable of generating two types of actions in response to given stimulus; set a register flag for later intervention by the CPU (using interrupt or polling methods), or generate event signals, which can be internally routed directly to other peripherals within the device. The use of events allows for direct actions to be performed in one peripheral in response to a stimulus in another without CPU intervention. This can lower the overall power consumption of the system if the CPU is able to remain in sleep modes for longer periods (SleepWalking), and lowers the latency of the system response.

The event system is comprised of a number of freely configurable Event resources, plus a number of fixed Event Users. Each Event resource can be configured to select the input peripheral that will generate the events signal, as well as the synchronization path and edge detection mode. The fixed-function Event Users, connected to peripherals within the device, can then subscribe to an Event resource in a one-to-many relationship in order to receive events as they are generated. An overview of the event system chain is shown in Figure 3-1 Module Overview on page 6.

Figure 3-1. Module Overview


Source Peripheral (Generator)
Event Resource A
Event User X
Event User Y
Destination Peripheral (User)
Destination Peripheral (User)

There are many different events that can be routed in the device, which can then trigger many different actions. For example, an Analog Comparator module could be configured to generate an event when the input signal rises above the compare threshold, which then triggers a Timer Counter module to capture the current count value for later use.

3.1. Event Channels

The Event module in each device consists of several channels, which can be freely linked to an event generator (i.e. a peripheral within the device that is capable of generating events). Each channel can be individually configured to select the generator peripheral, signal path, and edge detection applied to the input event signal, before being passed to any event user(s).

Event channels can support multiple users within the device in a standardized manner. When an Event User is linked to an Event Channel, the channel will automatically handshake with all attached users to ensure that all modules correctly receive and acknowledge the event.

3.2. Event Users

Event Users are able to subscribe to an Event Channel, once it has been configured. Each Event User consists of a fixed connection to one of the peripherals within the device (for example, an ADC module, or Timer module) and is capable of being connected to a single Event Channel.

3.3. Edge Detection

For asynchronous events, edge detection on the event input is not possible, and the event signal must be passed directly between the event generator and event user. For synchronous and re-synchronous events, the input signal from the event generator must pass through an edge detection unit, so that only the rising, falling, or both edges of the event signal triggers an action in the event user.

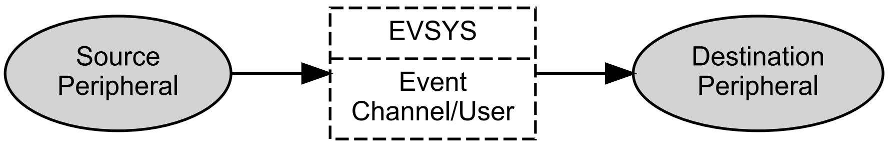
3.4. Path Selection

The event system in the SAM devices supports three signal path types from the event generator to Event Users: asynchronous, synchronous, and re-synchronous events.

3.4.1. Asynchronous Paths

Asynchronous event paths allow for an asynchronous connection between the event generator and Event Users, when the source and destination peripherals share the same Generic Clock channel. In this mode the event is propagated between the source and destination directly to reduce the event latency, thus no edge detection is possible. The asynchronous event chain is shown in Figure 3-2 Asynchronous Paths on page 7.

Figure 3-2. Asynchronous Paths



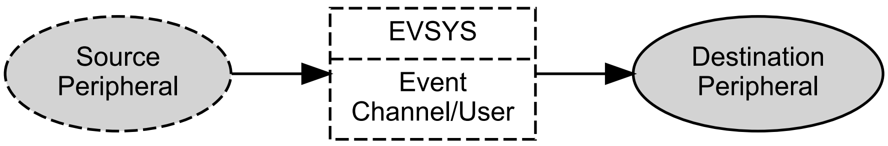
Note: Identically shaped borders in the diagram indicate a shared generic clock channel.

3.4.2. Synchronous Paths

The Synchronous event path should be used when edge detection or interrupts from the event channel are required, and the source event generator and the event channel shares the same Generic Clock channel. The synchronous event chain is shown in Figure 3-3 Synchronous Paths on page 7.

Not all peripherals support Synchronous event paths; refer to the device datasheet.

Figure 3-3. Synchronous Paths



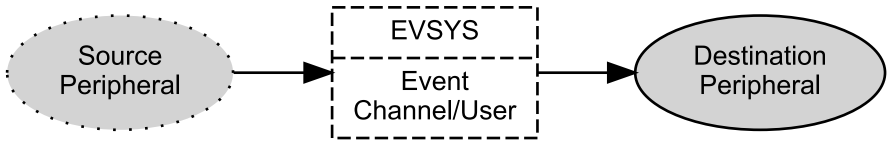
Note: Identically shaped borders in the diagram indicate a shared generic clock channel.

3.4.3. Re-synchronous Paths

Re-synchronous event paths are a special form of synchronous events, where when edge detection or interrupts from the event channel are required, but the event generator and the event channel use different Generic Clock channels. The re-synchronous path allows the Event System to synchronize the incoming event signal from the Event Generator to the clock of the Event System module to avoid missed events, at the cost of a higher latency due to the re-synchronization process. The re-synchronous event chain is shown in Figure 3-4 Re-synchronous Paths on page 8.

Not all peripherals support re-synchronous event paths; refer to the device datasheet.

Figure 3-4. Re-synchronous Paths

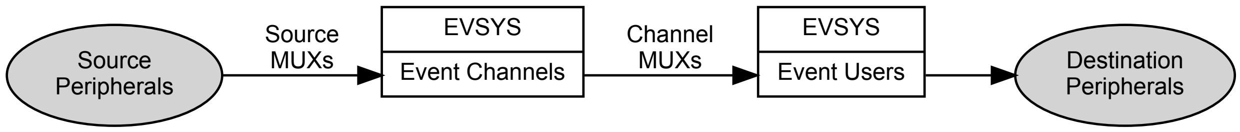


Note: Identically shaped borders in the diagram indicate a shared generic clock channel.

3.5. Physical Connection

Figure 3-5 Physical Connection on page 8 shows how this module is interconnected within the device.

Figure 3-5. Physical Connection



3.6. Configuring Events

For SAM devices, several steps are required to properly configure an event chain, so that hardware peripherals can respond to events generated by each other, as listed below.

3.6.1. Source Peripheral

1. The source peripheral (that will generate events) must be configured and enabled.
2. The source peripheral (that will generate events) must have an output event enabled.

3.6.2. Event System

1. An event system channel must be allocated and configured with the correct source peripheral selected as the channel's event generator.
2. The event system user must be configured and enabled, and attached to # event channel previously allocated.

3.6.3. Destination Peripheral

1. The destination peripheral (that will receive events) must be configured and enabled.
2. The destination peripheral (that will receive events) must have an input event enabled.

4. Special Considerations

There are no special considerations for this module.

5. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

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

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Type events_interrupt_hook

```
typedef void(* events_interrupt_hook )(struct events_resource *resource)
```

7.2. Structure Definitions

7.2.1. Struct events_config

This event configuration struct is used to configure each of the channels.

Table 7-1. Members

Type	Name	Description
uint8_t	clock_source	Clock source for the event channel
enum events_edge_detect	edge_detect	Select edge detection mode
uint8_t	generator	Set event generator for the channel
enum events_path_selection	path	Select events channel path

7.2.2. Struct events_hook

Event hook structure.

Table 7-2. Members

Type	Name	Description
events_interrupt_hook	hook_func	Event hook function
struct events_hook *	next	Next event hook
struct events_resource *	resource	Event resource

7.2.3. Struct events_resource

Event resource structure.

Note: The fields in this structure should not be altered by the user application; they are reserved for driver internals only.

7.3. Macro Definitions

7.3.1. Macro EVSYS_ID_GEN_NONE

```
#define EVSYS_ID_GEN_NONE
```

Use this to disable any peripheral event input to a channel. This can be useful if you only want to use a channel for software generated events. Definition for no generator selection.

7.4. Function Definitions

7.4.1. Function events_ack_interrupt()

Acknowledge an interrupt source.

```
enum status_code events_ack_interrupt(
        struct events_resource * resource,
        enum events_interrupt_source source)
```

Acknowledge an interrupt source so the interrupt state is cleared in hardware.

Table 7-3. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns
Status of the interrupt source.

Table 7-4. Return Values

Return value	Description
STATUS_OK	Interrupt source was acknowledged successfully

7.4.2. Function events_add_hook()

Insert hook into the event drivers interrupt hook queue.

```
enum status_code events_add_hook(
        struct events_resource * resource,
        struct events_hook * hook)
```

Inserts a hook into the event drivers interrupt hook queue.

Table 7-5. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	hook	Pointer to an events_hook struct instance

Returns
Status of the insertion procedure.

Table 7-6. Return Values

Return value	Description
STATUS_OK	Insertion of hook went successful

7.4.3. Function events_allocate()

Allocate an event channel and set configuration.

```
enum status_code events_allocate(
        struct events_resource * resource,
        struct events_config * config)
```

Allocates an event channel from the event channel pool and sets the channel configuration.

Table 7-7. Parameters

Data direction	Parameter name	Description
[out]	resource	Pointer to a events_resource struct instance
[in]	config	Pointer to a events_config struct

Returns
Status of the configuration procedure.

Table 7-8. Return Values

Return value	Description
STATUS_OK	Allocation and configuration went successful
STATUS_ERR_NOT_FOUND	No free event channel found

7.4.4. Function events_attach_user()

Attach user to the event channel.

```
enum status_code events_attach_user(
        struct events_resource * resource,
        uint8_t user_id)
```

Attach a user peripheral to the event channel to receive events.

Table 7-9. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	user_id	A number identifying the user peripheral found in the device header file

Returns
Status of the user attach procedure.

Table 7-10. Return Values

Return value	Description
STATUS_OK	No errors detected when attaching the event user

7.4.5. Function events_create_hook()

Initializes an interrupt hook for insertion in the event interrupt hook queue.

```
enum status_code events_create_hook(
        struct events_hook * hook,
        events_interrupt_hook hook_func)
```

Initializes a hook structure so it is ready for insertion in the interrupt hook queue.

Table 7-11. Parameters

Data direction	Parameter name	Description
[out]	hook	Pointer to an events_hook struct instance
[in]	hook_func	Pointer to a function containing the interrupt hook code

Returns
Status of the hook creation procedure.

Table 7-12. Return Values

Return value	Description
STATUS_OK	Creation and initialization of interrupt hook went successful

7.4.6. Function events_del_hook()

Remove hook from the event drivers interrupt hook queue.

```
enum status_code events_del_hook(
        struct events_resource * resource,
        struct events_hook * hook)
```

Removes a hook from the event drivers interrupt hook queue.

Table 7-13. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	hook	Pointer to an events_hook struct instance

Returns
Status of the removal procedure.

Table 7-14. Return Values

Return value	Description
STATUS_OK	Removal of hook went successful
STATUS_ERR_NO_MEMORY	There are no hooks instances in the event driver interrupt hook list
STATUS_ERR_NOT_FOUND	Interrupt hook not found in the event drivers interrupt hook list

7.4.7. Function events_detach_user()

Detach a user peripheral from the event channel.

```
enum status_code events_detach_user(
        struct events_resource * resource,
        uint8_t user_id)
```

Deattach a user peripheral from the event channels so it does not receive any more events.

Table 7-15. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an event_resource struct instance
[in]	user_id	A number identifying the user peripheral found in the device header file

Returns
Status of the user detach procedure.

Table 7-16. Return Values

Return value	Description
STATUS_OK	No errors detected when detaching the event user

7.4.8. Function events_disable_interrupt_source()

Disable interrupt source.

```
enum status_code events_disable_interrupt_source(
        struct events_resource * resource,
        enum events_interrupt_source source)
```

Disable an interrupt source so can trigger execution of an interrupt hook.

Table 7-17. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns
Status of the interrupt source enable procedure.

Table 7-18. Return Values

Return value	Description
STATUS_OK	Enabling of the interrupt source went successful
STATUS_ERR_INVALID_ARG	Interrupt source does not exist

7.4.9. Function events_enable_interrupt_source()

Enable interrupt source.

```
enum status_code events_enable_interrupt_source(
        struct events_resource * resource,
        enum events_interrupt_source source)
```

Enable an interrupt source so can trigger execution of an interrupt hook.

Table 7-19. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns
Status of the interrupt source enable procedure.

Table 7-20. Return Values

Return value	Description
STATUS_OK	Enabling of the interrupt source was successful
STATUS_ERR_INVALID_ARG	Interrupt source does not exist

7.4.10. Function events_get_config_defaults()

Initializes an event configurations struct to defaults.

```
void events_get_config_defaults(
        struct events_config * config)
```

Initailizes an event configuration struct to predefined safe default settings.

Table 7-21. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to an instance of struct events_config

7.4.11. Function events_get_free_channels()

Get the number of free channels.

```
uint8_t events_get_free_channels( void )
```

Get the number of allocatable channels in the events system resource pool.

Returns
The number of free channels in the event system.

7.4.12. Function events_is_busy()

Check if a channel is busy.

```
bool events_is_busy(
        struct events_resource * resource)
```

Check if a channel is busy, a channel stays busy until all users connected to the channel has handled an event.

Table 7-22. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to a events_resource struct instance

Returns
Status of the channels busy state.

Table 7-23. Return Values

Return value	Description
true	One or more users connected to the channel has not handled the last event
false	All users are ready to handle new events

7.4.13. Function events_is_detected()

Check if an event is detected on the event channel.

```
bool events_is_detected(
        struct events_resource * resource)
```

Check if an event has been detected on the channel.

Note: This function will clear the event detected interrupt flag.

Table 7-24. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns
Status of the event detection interrupt flag.

Table 7-25. Return Values

Return value	Description
true	Event has been detected
false	Event has not been detected

7.4.14. Function events_is_interrupt_set()

Check if interrupt source is set.

```
bool events_is_interrupt_set(
        struct events_resource * resource,
        enum events_interrupt_source source)
```

Check if an interrupt source is set and should be processed.

Table 7-26. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct instance
[in]	source	One of the members in the events_interrupt_source enumerator

Returns
Status of the interrupt source.

Table 7-27. Return Values

Return value	Description
true	Interrupt source is set
false	Interrupt source is not set

7.4.15. Function events_is_overrun()

Check if there has been an overrun situation on this channel.

```
bool events_is_overrun(
        struct events_resource * resource)
```

Note: This function will clear the event overrun detected interrupt flag.

Table 7-28. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns
Status of the event overrun interrupt flag.

Table 7-29. Return Values

Return value	Description
true	Event overrun has been detected
false	Event overrun has not been detected

7.4.16. Function events_is_users_ready()

Check if all users connected to the channel are ready.

```
bool events_is_users_ready(
        struct events_resource * resource)
```

Check if all users connected to the channel are ready to handle incoming events.

Table 7-30. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns

The ready status of users connected to an event channel.

Table 7-31. Return Values

Return value	Description
true	All the users connected to the event channel are ready to handle incoming events
false	One or more users connected to the event channel are not ready to handle incoming events

7.4.17. Function events_release()

Release allocated channel back the the resource pool.

```
enum status_code events_release(
        struct events_resource * resource)
```

Release an allocated channel back to the resource pool to make it available for other purposes.

Table 7-32. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns

Status of the channel release procedure.

Table 7-33. Return Values

Return value	Description
STATUS_OK	No error was detected when the channel was released
STATUS_BUSY	One or more event users have not processed the last event
STATUS_ERR_NOT_INITIALIZED	Channel not allocated, and can therefore not be released

7.4.18. Function events_trigger()

Trigger software event.

```
enum status_code events_trigger(
        struct events_resource * resource)
```

Trigger an event by software.

Note: Software event works on either a synchronous path or resynchronized path, and edge detection must be configured to rising-edge detection.

Table 7-34. Parameters

Data direction	Parameter name	Description
[in]	resource	Pointer to an events_resource struct

Returns
Status of the event software procedure.

Table 7-35. Return Values

Return value	Description
STATUS_OK	No error was detected when the software tigger signal was issued
STATUS_ERR_UNSUPPORTED_DEV	If the channel path is asynchronous and/or the edge detection is not set to RISING

7.5. Enumeration Definitions

7.5.1. Enum events_edge_detect

Event channel edge detect setting.

Table 7-36. Members

Enum value	Description
EVENTS_EDGE_DETECT_NONE	No event output
EVENTS_EDGE_DETECT_RISING	Event on rising edge
EVENTS_EDGE_DETECT_FALLING	Event on falling edge
EVENTS_EDGE_DETECT_BOTH	Event on both edges

7.5.2. Enum events_interrupt_source

Interrupt source selector definitions.

Table 7-37. Members

Enum value	Description
EVENTS_INTERRUPT_OVERRUN	Overrun in event channel detected interrupt
EVENTS_INTERRUPT_DETECT	Event signal propagation in event channel detected interrupt

7.5.3. Enum events_path_selection

Event channel path selection.

Table 7-38. Members

Enum value	Description
EVENTS_PATH_SYNCHRONOUS	Select the synchronous path for this event channel
EVENTS_PATH_RESYNCHRONIZED	Select the resynchronizer path for this event channel
EVENTS_PATH_ASYNCHRONOUS	Select the asynchronous path for this event channel

8. Extra Information for EVENTS Driver

8.1. Acronyms

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

Acronym	Description
CPU	Central Processing Unit
MUX	Multiplexer

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.

Changelog
Fix a bug in internal function `_events_find_bit_position()`
Rewrite of events driver
Initial Release

9. Examples for EVENTS Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Event System (EVENTS) 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 EVENTS - Basic
- Quick Start Guide for EVENTS - Interrupt Hooks

9.1. Quick Start Guide for EVENTS - Basic

In this use case, the EVENT module is configured for:

- Synchronous event path with rising edge detection on the input
- One user attached to the configured event channel
- No hardware event generator attached to the channel

This use case allocates an event channel. This channel is not connected to any hardware event generator, events are software triggered. One user is connected to the allocated and configured event channel.

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, before any functions, according to the kit used:

- SAM D20 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER         EVSYS_ID_USER_TC3_EVU
```

- SAM D21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER         EVSYS_ID_USER_TC3_EVU
```

- SAM R21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER         EVSYS_ID_USER_TC3_EVU
```

- SAM D11 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC2_MCX_0
#define CONF_EVENT_USER         EVSYS_ID_USER_TC1_EVU
```

- SAM L21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_NONE
#define CONF_EVENT_USER         EVSYS_ID_USER_PORT_EV_0
```

- SAM L22 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_NONE
#define CONF_EVENT_USER         EVSYS_ID_USER_PORT_EV_0
```

- SAM DA1 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_MCX_0
#define CONF_EVENT_USER         EVSYS_ID_USER_TC3_EVU
```

- SAM C21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_NONE
#define CONF_EVENT_USER         EVSYS_ID_USER_PORT_EV_0
```

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

```
static void configure_event_channel(struct events_resource *resource)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator      = CONF_EVENT_GENERATOR;
    config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
    config.path           = EVENTS_PATH_SYNCHRONOUS;
    config.clock_source   = GCLK_GENERATOR_0;

    events_allocate(resource, &config);
}

static void configure_event_user(struct events_resource *resource)
{
    events_attach_user(resource, CONF_EVENT_USER);
}
```

Create an event resource struct and add to user application (typically the start of `main()`):

```
struct events_resource example_event;
```

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

```
configure_event_channel(&example_event);
configure_event_user(&example_event);
```

9.1.1.3. Workflow

1. Create an event channel configuration struct, which can be filled out to adjust the configuration of a single event channel.

```
struct events_config config;
```

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

```
events_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. Adjust the configuration struct to request that the channel is to be attached to the specified event generator, that rising edges of the event signal is to be detected on the channel, and that the synchronous event path is to be used.

```
config.generator      = CONF_EVENT_GENERATOR;
config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
config.path              = EVENTS_PATH_SYNCHRONOUS;
config.clock_source      = GCLK_GENERATOR_0;
```

4. Allocate and configure the channel using the configuration structure.

```
events_allocate(resource, &config);
```

Note: The existing configuration struct may be re-used, as long as any values that have been altered from the default settings are taken into account by the user application.

5. Attach a user to the channel.

```
events_attach_user(resource, CONF_EVENT_USER);
```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
while (events_is_busy(&example_event)) {
    /* Wait for channel */
};

events_trigger(&example_event);

while (true) {
    /* Nothing to do */
}
```

9.1.2.2. Workflow

1. Wait for the event channel to become ready to accept a new event trigger.

```
while (events_is_busy(&example_event)) {
    /* Wait for channel */
};
```

2. Perform a software event trigger on the configured event channel.

```
events_trigger(&example_event);
```

9.2. Quick Start Guide for EVENTS - Interrupt Hooks

In this use case, the EVENT module is configured for:

- Synchronous event path with rising edge detection
- TC4 as event generator on the allocated event channel (TC0 is used for SAM L22)
- One event channel user attached
- An event interrupt hook is used to execute some code when an event is detected

In this use case TC is used as event generator, generating events on overflow. One user attached, counting events on the channel. To be able to execute some code when an event is detected, an interrupt hook is used. The interrupt hook will also count the number of events detected and toggle a LED on the board each time an event is detected.

Note: Because this example is showing how to set up an interrupt hook there is no user attached to the user.

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, before any functions, according to the kit used:

- SAM D20 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_TC0_EVU
```

```
#define CONF_TC_MODULE TC4
```

- SAM D21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- SAM R21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- SAM D11 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC2_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC2
```

- SAM L21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- SAM L22 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC0_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC0
```

- SAM DA1 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

- SAM C21 Xplained Pro:

```
#define CONF_EVENT_GENERATOR    EVSYS_ID_GEN_TC4_OVF
#define CONF_EVENT_USER         EVSYS_ID_USER_DMAC_CH_0
```

```
#define CONF_TC_MODULE TC4
```

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

```

static volatile uint32_t event_count = 0;

void event_counter(struct events_resource *resource);

static void configure_event_channel(struct events_resource *resource)
{
    struct events_config config;

    events_get_config_defaults(&config);

    config.generator      = CONF_EVENT_GENERATOR;
    config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
    config.path           = EVENTS_PATH_SYNCHRONOUS;
    config.clock_source   = GCLK_GENERATOR_0;

    events_allocate(resource, &config);
}

static void configure_event_user(struct events_resource *resource)
{
    events_attach_user(resource, CONF_EVENT_USER);
}

static void configure_tc(struct tc_module *tc_instance)
{
    struct tc_config config_tc;
    struct tc_events config_events;

    tc_get_config_defaults(&config_tc);

    config_tc.counter_size    = TC_COUNTER_SIZE_8BIT;
    config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_FREQ;
    config_tc.clock_source    = GCLK_GENERATOR_1;
    config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV64;

    tc_init(tc_instance, CONF_TC_MODULE, &config_tc);

    config_events.generate_event_on_overflow = true;
    tc_enable_events(tc_instance, &config_events);

    tc_enable(tc_instance);

}

static void configure_event_interrupt(struct events_resource *resource,
        struct events_hook *hook)
{
    events_create_hook(hook, event_counter);

    events_add_hook(resource, hook);
    events_enable_interrupt_source(resource, EVENTS_INTERRUPT_DETECT);
}
```

```
void event_counter(struct events_resource *resource)
{
    if(events_is_interrupt_set(resource, EVENTS_INTERRUPT_DETECT)) {
        port_pin_toggle_output_level(LED_0_PIN);

        event_count++;
        events_ack_interrupt(resource, EVENTS_INTERRUPT_DETECT);

    }
}
```

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

```
struct tc_module        tc_instance;
struct events_resource example_event;
struct events_hook      hook;

system_init();
system_interrupt_enable_global();

configure_event_channel(&example_event);
configure_event_user(&example_event);
configure_event_interrupt(&example_event, &hook);
configure_tc(&tc_instance);
```

9.2.1.3. Workflow

1. Create an event channel configuration structure instance which will contain the configuration for the event.

```
struct events_config config;
```

2. Initialize the event channel configuration struct with safe default values.
Note: This shall always be performed before using the configuration struct to ensure that all members are initialized to known default values.

```
events_get_config_defaults(&config);
```

3. Adjust the configuration structure:
   - Use EXAMPLE_EVENT_GENRATOR as event generator
   - Detect events on rising edge
   - Use the synchronous event path
   - Use GCLK Generator 0 as event channel clock source

```
config.generator      = CONF_EVENT_GENERATOR;
config.edge_detect    = EVENTS_EDGE_DETECT_RISING;
config.path           = EVENTS_PATH_SYNCHRONOUS;
config.clock_source   = GCLK_GENERATOR_0;
```

4. Allocate and configure the channel using the configuration structure.

```
events_allocate(resource, &config);
```

5. Make sure there is no user attached. To attach a user, change the value of EXAMPLE_EVENT_USER to the correct peripheral ID.

```
events_attach_user(resource, CONF_EVENT_USER);
```

6. Create config_tc and config_events configuration structure instances.

   ```
   struct tc_config config_tc;
   struct tc_events config_events;
   ```

7. Initialize the TC module configuration structure with safe default values.
   Note: This function shall always be called on new configuration structure instances to make sure that all structure members are initialized.

   ```
   tc_get_config_defaults(&config_tc);
   ```

8. Adjust the config_tc structure:
   - Set counter size to 8-bit
   - Set wave generation mode to normal frequency generation
   - Use GCLK generator 1 to as TC module clock source
   - Prescale the input clock with 64

   ```
   config_tc.counter_size    = TC_COUNTER_SIZE_8BIT;
   config_tc.wave_generation = TC_WAVE_GENERATION_NORMAL_FREQ;
   config_tc.clock_source    = GCLK_GENERATOR_1;
   config_tc.clock_prescaler = TC_CLOCK_PRESCALER_DIV64;
   ```

9. Initialize, configure, and assosiate the tc_instance handle with the TC hardware pointed to by TC_MODULE.

   ```
   tc_init(tc_instance, CONF_TC_MODULE, &config_tc);
   ```

10. Adjust the config_events structure to enable event generation on overflow in the timer and then enable the event configuration.

    ```
    config_events.generate_event_on_overflow = true;
    tc_enable_events(tc_instance, &config_events);
    ```

11. Enable the timer/counter module.

    ```
    tc_enable(tc_instance);
    ```

12. Create a new interrupt hook and use the function event_counter as hook code.

    ```
    events_create_hook(hook, event_counter);
    ```

13. Add the newly created hook to the interrupt hook queue and enable the event detected interrupt.

    ```
    events_add_hook(resource, hook);
    events_enable_interrupt_source(resource, EVENTS_INTERRUPT_DETECT);
    ```

14. Example interrupt hook code. If the hook was triggered by an event detected interrupt on the event channel this code will toggle the LED on the Xplained PRO board and increase the value of the event_count variable. The interrupt is then acknowledged.

```
void event_counter(struct events_resource *resource)
{
    if(events_is_interrupt_set(resource, EVENTS_INTERRUPT_DETECT)) {
        port_pin_toggle_output_level(LED_0_PIN);

        event_count++;
        events_ack_interrupt(resource, EVENTS_INTERRUPT_DETECT);

    }
}
```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

```
while (events_is_busy(&example_event)) {
    /* Wait for channel */
};

tc_start_counter(&tc_instance);

while (true) {
    /* Nothing to do */
}
```

9.2.2.2. Workflow

1. Wait for the event channel to become ready.

```
while (events_is_busy(&example_event)) {
    /* Wait for channel */
};
```

2. Start the timer/counter.

```
tc_start_counter(&tc_instance);
```

10. Document Revision History

Doc. Rev.	Date	Comments
42108G	12/2015	Added support for SAM D09 and SAM L22
42108F	08/2015	Added support for SAM L21, SAM DA1, and SAM C20/C21
42108E	12/2014	Added support for interrupt hook mode. Added support for SAM R21 and SAM D10/D11.
42108D	01/2014	Update to support SAM D21 and corrected documentation typos
42108C	11/2013	Fixed incorrect documentation for the event signal paths. Added configuration steps overview to the documentation.
42108B	06/2013	Corrected documentation typos
42108A	06/2013	Initial release