SMART ARM-based Microcontrollers

AT03229: SAM D/R/L/C Peripheral Access Controller (PAC) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontroller provides an interface for the locking and unlocking of peripheral registers within the device. When a peripheral is locked, accidental writes to the peripheral will be blocked and a CPU exception will be raised.

The following peripherals are used by this module:

- PAC (Peripheral Access Controller)

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:

Table of Contents

1. Software License

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

The SAM devices are fitted with a Peripheral Access Controller (PAC) that can be used to lock and unlock write access to a peripheral's registers (see Non-Writable Registers). Locking a peripheral minimizes the risk of unintended configuration changes to a peripheral as a consequence of Run-away Code or use of a Faulty Module Pointer.

Physically, the PAC restricts write access through the AHB bus to registers used by the peripheral, making the register non-writable. PAC locking of modules should be implemented in configuration critical applications where avoiding unintended peripheral configuration changes are to be regarded in the highest of priorities.

All interrupt must be disabled while a peripheral is unlocked to make sure correct lock/unlock scheme is upheld.

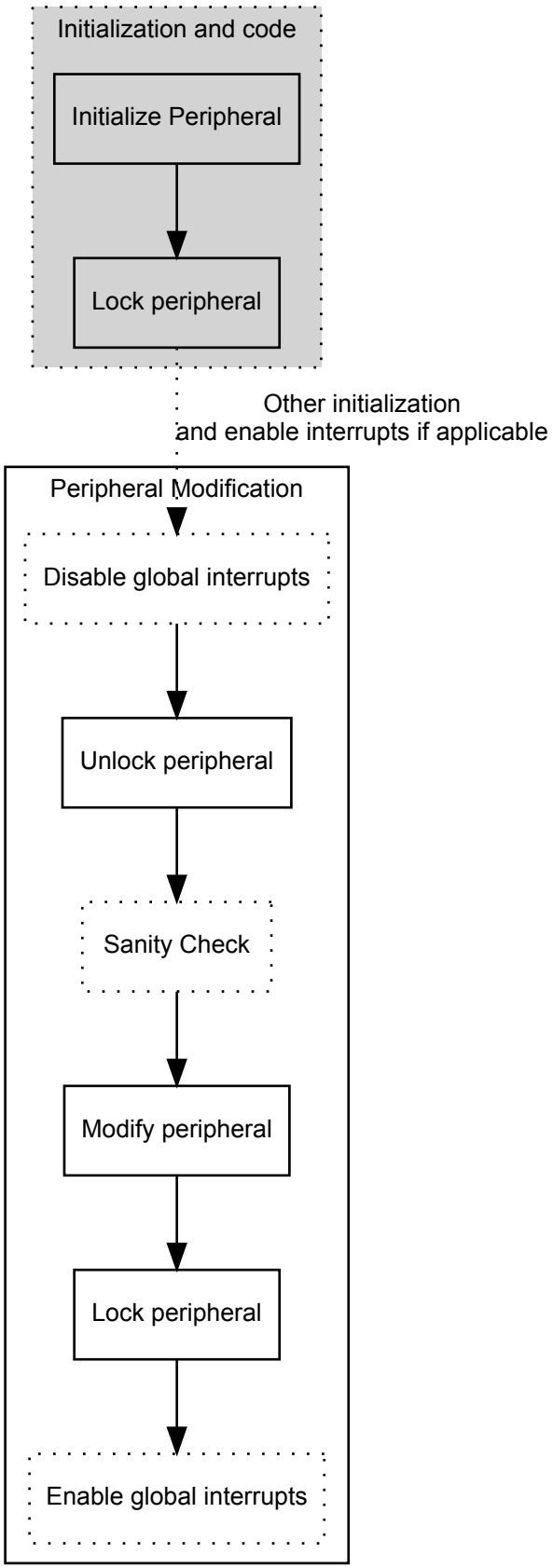
3.1. Locking Scheme

The module has a built in safety feature requiring that an already locked peripheral is not relocked, and that already unlocked peripherals are not unlocked again. Attempting to unlock and already unlocked peripheral, or attempting to lock a peripheral that is currently locked will generate a CPU exception. This implies that the implementer must keep strict control over the peripheral's lock-state before modifying them. With this added safety, the probability of stopping runaway code increases as the program pointer can be caught inside the exception handler, and necessary countermeasures can be initiated. The implementer should also consider using sanity checks after an unlock has been performed to further increase the security.

3.2. Recommended Implementation

A recommended implementation of the PAC can be seen in Figure 3-1 Recommended Implementation on page 6.

Figure 3-1. Recommended Implementation



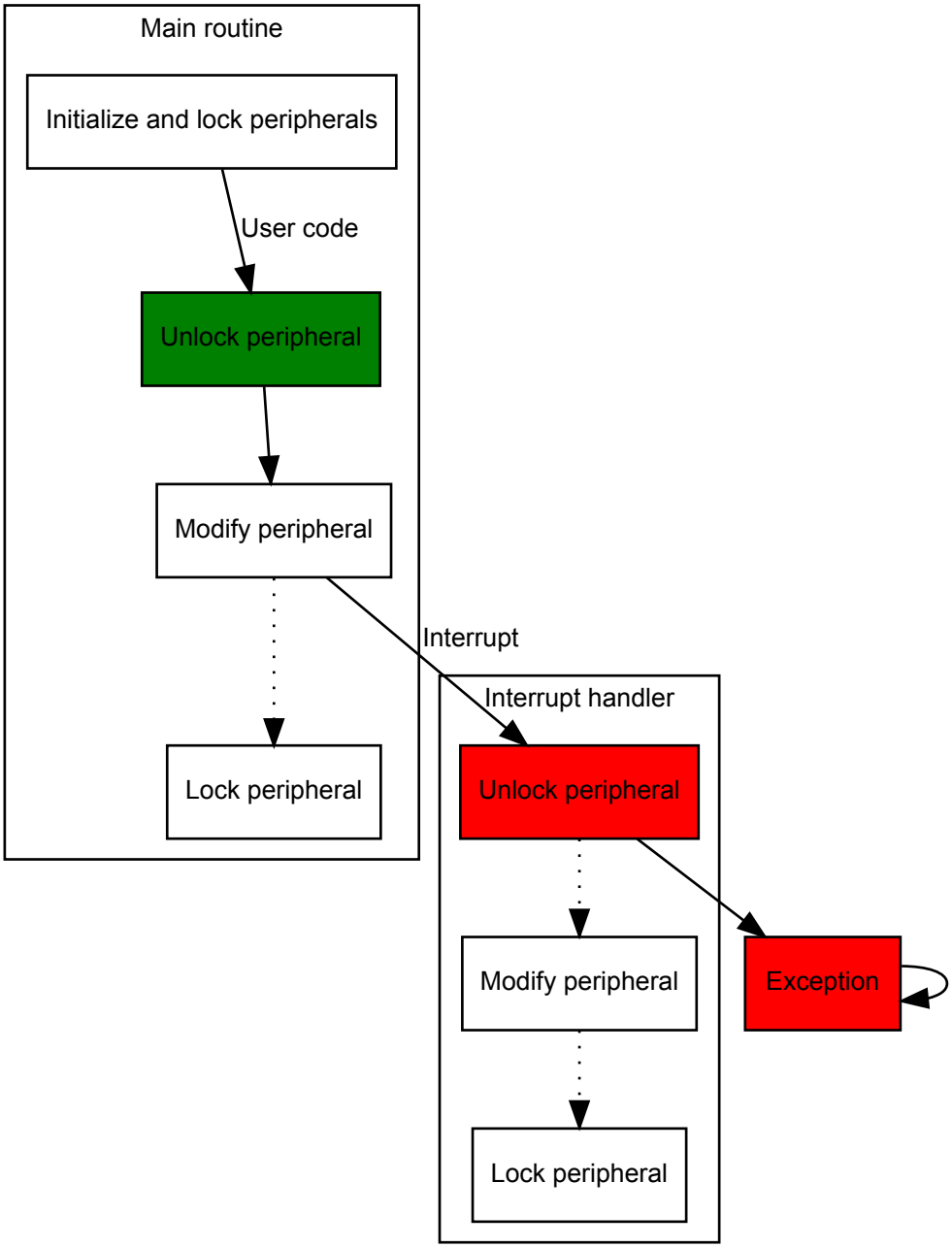
3.3. Why Disable Interrupts

Global interrupts must be disabled while a peripheral is unlocked as an interrupt handler would not know the current state of the peripheral lock. If the interrupt tries to alter the lock state, it can cause an exception as it potentially tries to unlock an already unlocked peripheral. Reading current lock state is to be avoided as it removes the security provided by the PAC (Reading Lock State).

Note: Global interrupts should also be disabled when a peripheral is unlocked inside an interrupt handler.

An example to illustrate the potential hazard of not disabling interrupts is shown in Figure 3-2 Why Disable Interrupts on page 8.

Figure 3-2. Why Disable Interrupts



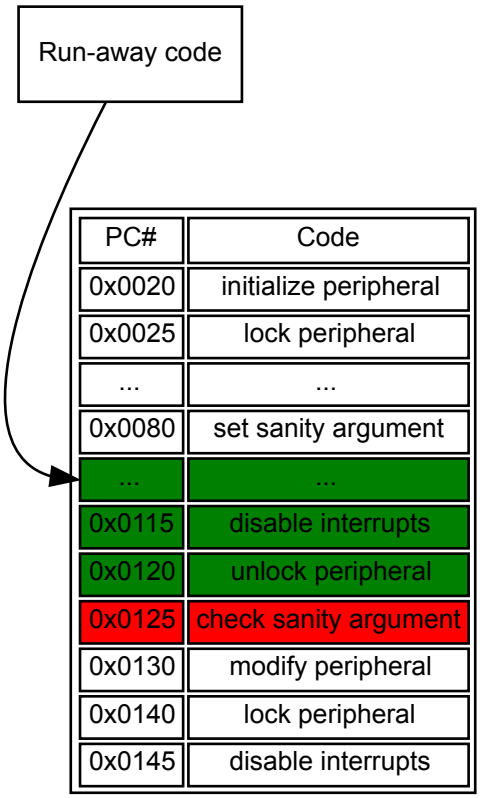
3.4. Run-away Code

Run-away code can be caused by the MCU being operated outside its specification, faulty code, or EMI issues. If a runaway code occurs, it is favorable to catch the issue as soon as possible. With a correct implementation of the PAC, the runaway code can potentially be stopped.

A graphical example showing how a PAC implementation will behave for different circumstances of runaway code in shown in Figure 3-3 Run-away Code on page 9 and Figure 3-4 Run-away Code on page 10.

Figure 3-3. Run-away Code

1. Run-away code is caught in sanity check. A CPU exception is executed.



2. Run-away code is caught when modifying locked peripheral. A CPU exception is executed.

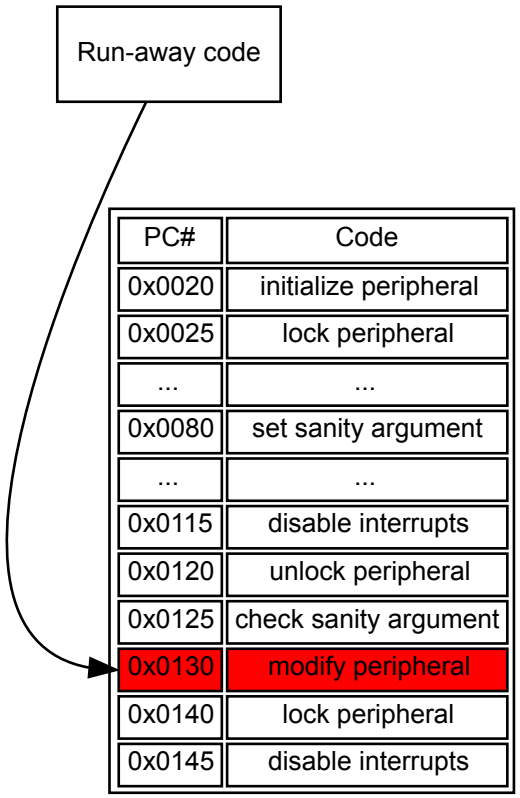
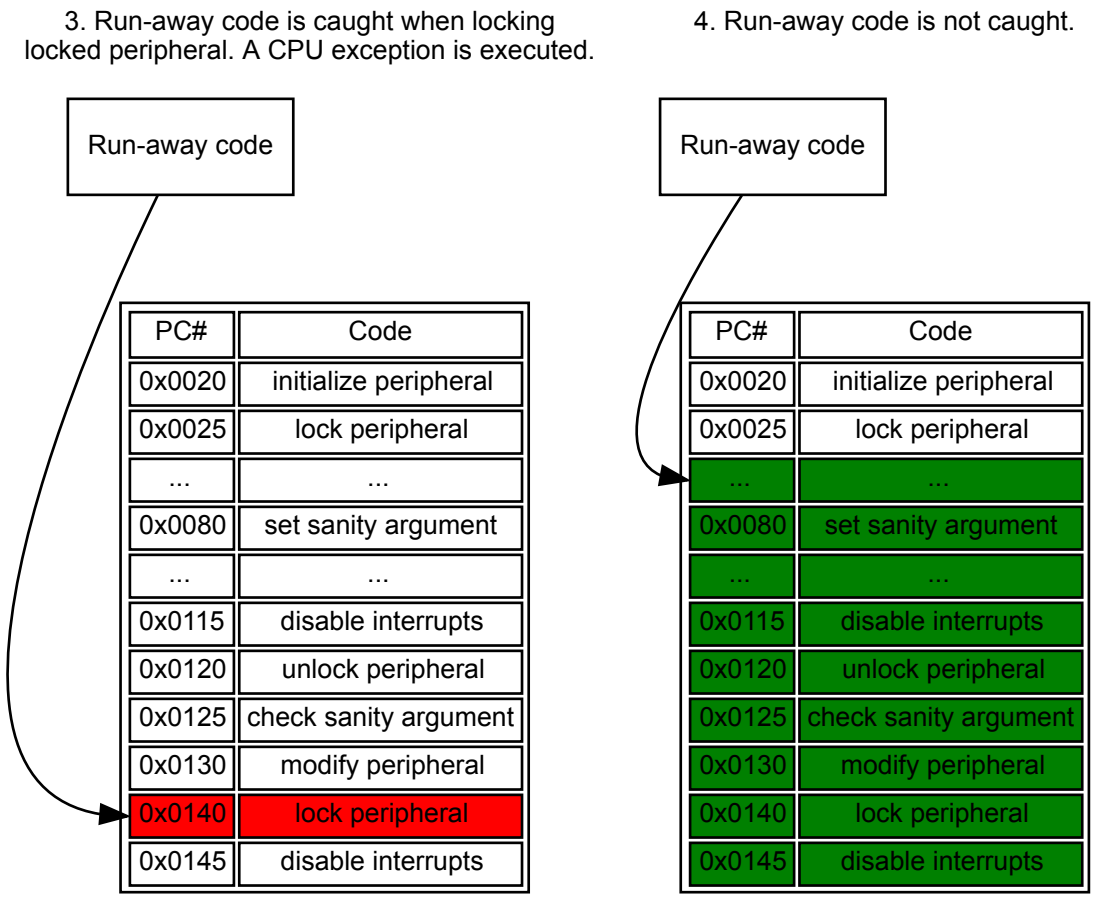


Figure 3-4. Run-away Code



In the example, green indicates that the command is allowed, red indicates where the runaway code will be caught, and the arrow where the runaway code enters the application. In special circumstances, like example 4 above, the runaway code will not be caught. However, the protection scheme will greatly enhance peripheral configuration security from being affected by runaway code.

3.4.1. Key-Argument

To protect the module functions against runaway code themselves, a key is required as one of the input arguments. The key-argument will make sure that runaway code entering the function without a function call will be rejected before inflicting any damage. The argument is simply set to be the bitwise inverse of the module flag, i.e.

```
system_peripheral_<lock_state>(SYSTEM_PERIPHERAL_<module>,
        ~SYSTEM_PERIPHERAL_<module>);
```

Where the lock state can be either lock or unlock, and module refer to the peripheral that is to be locked/unlocked.

3.5. Faulty Module Pointer

The PAC also protects the application from user errors such as the use of incorrect module pointers in function arguments, given that the module is locked. It is therefore recommended that any unused peripheral is locked during application initialization.

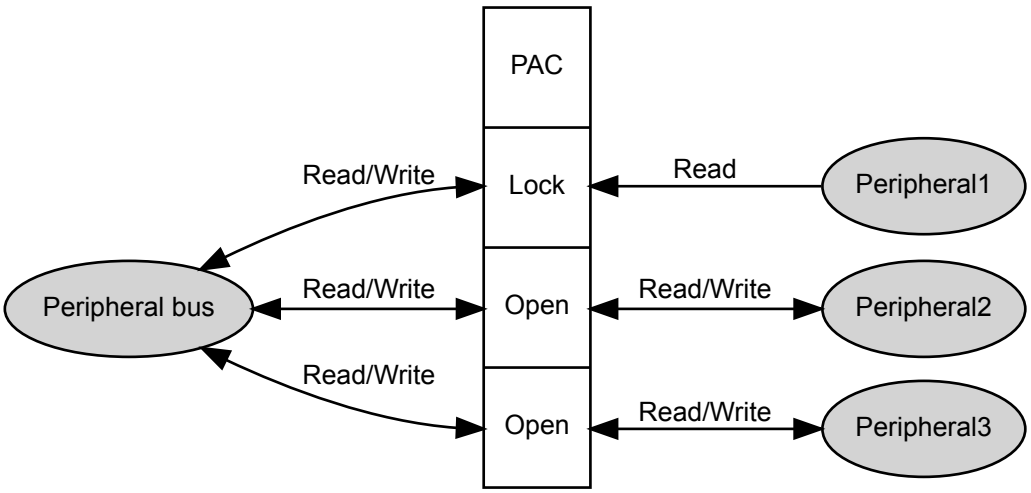
3.6. Use of __no_inline

Using the function attribute `__no_inline` will ensure that there will only be one copy of each functions in the PAC driver API in the application. This will lower the likelihood that runaway code will hit any of these functions.

3.7. Physical Connection

Figure 3-5 Physical Connection on page 11 shows how this module is interconnected within the device.

Figure 3-5. Physical Connection



4. Special Considerations

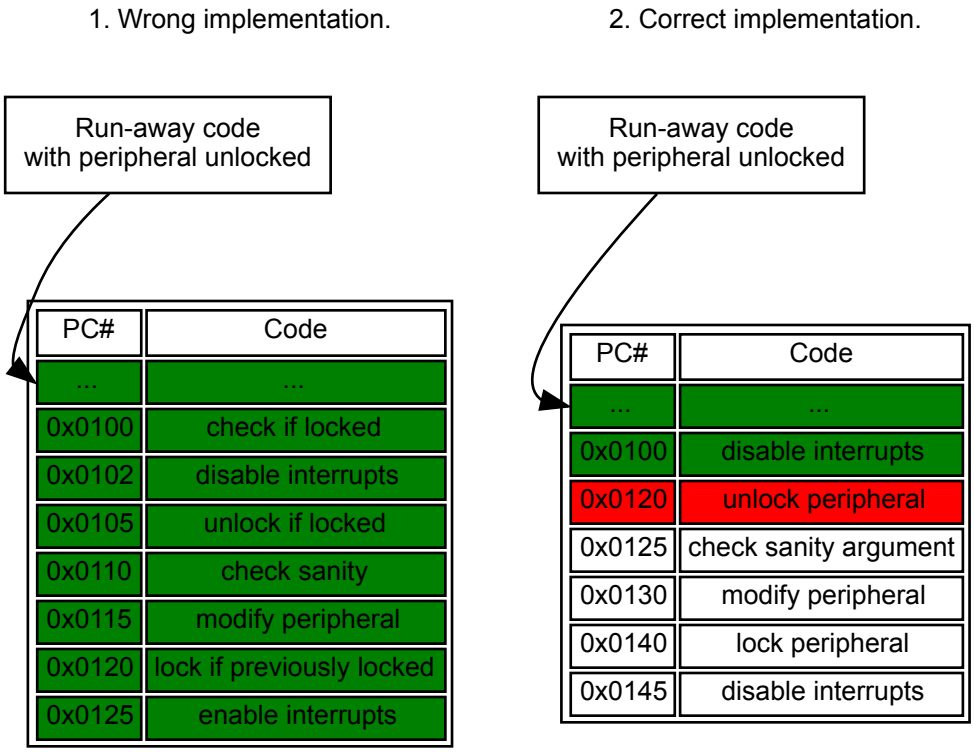
4.1. Non-Writable Registers

Not all registers in a given peripheral can be set non-writable. Which registers this applies to is showed in List of Non-Write Protected Registers and the peripheral's subsection "Register Access Protection" in the device datasheet.

4.2. Reading Lock State

Reading the state of the peripheral lock is to be avoided as it greatly compromises the protection initially provided by the PAC. If a lock/unlock is implemented conditionally, there is a risk that eventual errors are not caught in the protection scheme. Examples indicating the issue are shown in Figure 4-1 Reading Lock State on page 12.

Figure 4-1. Reading Lock State



In the left figure above, one can see the runaway code continues as all illegal operations are conditional. On the right side figure, the runaway code is caught as it tries to unlock the peripheral.

5. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

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

7. API Overview

7.1. Macro Definitions

7.1.1. Macro SYSTEM_PERIPHERAL_ID

```
#define SYSTEM_PERIPHERAL_ID(peripheral)
```

Retrieves the ID of a specified peripheral name, giving its peripheral bus location.

Table 7-1. Parameters

Data direction	Parameter name	Description
[in]	peripheral	Name of the peripheral instance

7.2. Function Definitions

7.2.1. Peripheral Lock and Unlock

7.2.1.1. Function system_peripheral_lock()

Lock a given peripheral's control registers.

```
__no_inline enum status_code system_peripheral_lock(
        const uint32_t peripheral_id,
        const uint32_t key)
```

Locks a given peripheral's control registers, to deny write access to the peripheral to prevent accidental changes to the module's configuration.

Warning Locking an already locked peripheral will cause a CPU exception, and terminate program execution.

Table 7-2. Parameters

Data direction	Parameter name	Description
[in]	peripheral_id	ID for the peripheral to be locked, sourced via the SYSTEM_PERIPHERAL_ID macro
[in]	key	Bitwise inverse of peripheral ID, used as key to reduce the chance of accidental locking. See Key-Argument

Returns
Status of the peripheral lock procedure.

Table 7-3. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied

7.2.1.2. Function system_peripheral_unlock()

Unlock a given peripheral's control registers.

```
__no_inline enum status_code system_peripheral_unlock(
        const uint32_t peripheral_id,
        const uint32_t key)
```

Unlocks a given peripheral's control registers, allowing write access to the peripheral so that changes can be made to the module's configuration.

Warning Unlocking an already locked peripheral will cause a CUP exception, and terminate program execution.

Table 7-4. Parameters

Data direction	Parameter name	Description
[in]	peripheral_id	ID for the peripheral to be unlocked, sourced via the SYSTEM_PERIPHERAL_ID macro
[in]	key	Bitwise inverse of peripheral ID, used as key to reduce the chance of accidental unlocking. See Key-Argument

Returns
Status of the peripheral unlock procedure.

Table 7-5. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied

7.2.2. APIs available for SAM L21/L22/C20/C21.

7.2.2.1. Function system_peripheral_lock_always()

Lock a given peripheral's control registers until hardware reset.

```
__no_inline enum status_code system_peripheral_lock_always(
        const uint32_t peripheral_id,
        const uint32_t key)
```

Locks a given peripheral's control registers, to deny write access to the peripheral to prevent accidental changes to the module's configuration. After lock, the only way to unlock is hardware reset.

Warning Locking an already locked peripheral will cause a CPU exception, and terminate program execution.

Table 7-6. Parameters

Data direction	Parameter name	Description
[in]	peripheral_id	ID for the peripheral to be locked, sourced via the SYSTEM_PERIPHERAL_ID macro
[in]	key	Bitwise inverse of peripheral ID, used as key to reduce the chance of accidental locking. See Key-Argument

Returns
Status of the peripheral lock procedure.

Table 7-7. Return Values

Return value	Description
STATUS_OK	If the peripheral was successfully locked
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied

7.2.2.2. Function system_pac_enable_interrupt()

Enable PAC interrupt.

```
void system_pac_enable_interrupt( void )
```

Enable PAC interrupt so can trigger execution on peripheral access error, see SYSTEM_Handler().

7.2.2.3. Function system_pac_disable_interrupt()

Disable PAC interrupt.

```
void system_pac_disable_interrupt( void )
```

Disable PAC interrupt on peripheral access error.

7.2.2.4. Function system_pac_enable_event()

Enable PAC event output.

```
void system_pac_enable_event( void )
```

Enable PAC event output on peripheral access error.

7.2.2.5. Function system_pac_disable_event()

Disable PAC event output.

```
void system_pac_disable_event( void )
```

Disable PAC event output on peripheral access error.

8. Extra Information for PAC Driver

8.1. Acronyms

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

Acronym	Description
AC	Analog Comparator
ADC	Analog-to-Digital Converter
EVSYS	Event System
NMI	Non-Maskable Interrupt
NVMCTRL	Non-Volatile Memory Controller
PAC	Peripheral Access Controller
PM	Power Manager
RTC	Real-Time Counter
SERCOM	Serial Communication Interface
SYSCTRL	System Controller
TC	Timer/Counter
WDT	Watch Dog Timer

8.2. Dependencies

This driver has the following dependencies:

- None

8.3. Errata

There are no errata related to this driver.

8.4. Module History

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

Changelog
Initial Release

9. Examples for PAC Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Peripheral Access Controller (PAC) 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 PAC - Basic

9.1. Quick Start Guide for PAC - Basic

In this use case, the peripheral-lock will be used to lock and unlock the PORT peripheral access, and show how to implement the PAC module when the PORT registers needs to be altered. The PORT will be set up as follows:

- One pin in input mode, with pull-up and falling edge-detect
- One pin in output mode

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:

```
void config_port_pins(void)
{
    struct port_config pin_conf;

    port_get_config_defaults(&pin_conf);

    pin_conf.direction  = PORT_PIN_DIR_INPUT;
    pin_conf.input_pull = PORT_PIN_PULL_UP;
    port_pin_set_config(BUTTON_0_PIN, &pin_conf);

    pin_conf.direction = PORT_PIN_DIR_OUTPUT;
    port_pin_set_config(LED_0_PIN, &pin_conf);
}
```

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

```
config_port_pins();
```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
system_init();

config_port_pins();

system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
        ~SYSTEM_PERIPHERAL_ID(PORT));
```

```
#if(SAML21)|| (SAML22) || (SAMC21) || defined(__DOXYGEN__)
    system_pac_enable_interrupt();
#endif
    system_interrupt_enable_global();

while (port_pin_get_input_level(BUTTON_0_PIN)) {
    /* Wait for button press */
}

system_interrupt_enter_critical_section();

system_peripheral_unlock(SYSTEM_PERIPHERAL_ID(PORT),
        ~SYSTEM_PERIPHERAL_ID(PORT));

port_pin_toggle_output_level(LED_0_PIN);

system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
        ~SYSTEM_PERIPHERAL_ID(PORT));

system_interrupt_leave_critical_section();

while (1) {
    /* Do nothing */
}
```

9.1.2.2. Workflow

1. Configure some GPIO port pins for input and output.

   ```
   config_port_pins();
   ```

2. Lock peripheral access for the PORT module; attempting to update the module while it is in a protected state will cause a CPU exception. For SAM D20/D21/D10/D11/R21/DA0/DA1, it is Hard Fault exception; For SAM L21/C21, it is system exception, see SYSTEM_Handler().

   ```
   system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
           ~SYSTEM_PERIPHERAL_ID(PORT));
   ```

3. Enable global interrupts.

   ```
   #if (SAML21) || (SAML22) || (SAMC21) || defined(__DOXYGEN__)
       system_pac_enable_interrupt();
   #endif
       system_interrupt_enable_global();
   ```

4. Loop to wait for a button press before continuing.

   ```
   while (port_pin_get_input_level(BUTTON_0_PIN)) {
       /* Wait for button press */
   }
   ```

5. Enter a critical section, so that the PAC module can be unlocked safely and the peripheral manipulated without the possibility of an interrupt modifying the protected module's state.

   ```
   system_interrupt_enter_critical_section();
   ```

6. Unlock the PORT peripheral registers.

   ```
   system_peripheral_unlock(SYSTEM_PERIPHERAL_ID(PORT),
           ~SYSTEM_PERIPHERAL_ID(PORT));
   ```

7. Toggle pin 11, and clear edge detect flag.

   ```
   port_pin_toggle_output_level(LED_0_PIN);
   ```

8. Lock the PORT peripheral registers.

```
system_peripheral_lock(SYSTEM_PERIPHERAL_ID(PORT),
        ~SYSTEM_PERIPHERAL_ID(PORT));
```

9. Exit the critical section to allow interrupts to function normally again.

```
system_interrupt_leave_critical_section();
```

10. Enter an infinite while loop once the module state has been modified successfully.

```
while (1) {
    /* Do nothing */
}
```

10. List of Non-Write Protected Registers

Look in device datasheet peripheral's subsection "Register Access Protection" to see which is actually available for your device.

Module	Non-write protected register
AC	INTFLAG
	STATUSA
	STATUSB
	STATUSC
ADC	INTFLAG
	STATUS
	RESULT
EVSYS	INTFLAG
	CHSTATUS
NVMCTRL	INTFLAG
	STATUS
PM	INTFLAG
PORT	N/A
RTC	INTFLAG
	READREQ
	STATUS
SYSCTRL	INTFLAG
SERCOM	INTFALG
	STATUS
	DATA
TC	INTFLAG

Module	Non-write protected register
	STATUS
WDT	INTFLAG
	STATUS
	(CLEAR)

11. Document Revision History

Doc. Rev.	Date	Comments
42107F	12/2015	Added support for SAM L21/L22, SAM C20/C21, SAM D09, and SAM DA1
42107E	12/2014	Added support for SAM R21 and SAM D10/D11
42107D	01/2014	Added support for SAM D21
42107C	10/2013	Extended acronyms list
42107B	06/2013	Corrected documentation typos
42107A	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**