SMART ARM-based Microcontrollers

AT03264: SAM D/R/L/C Watchdog (WDT) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Watchdog Timer module, including the enabling, disabling, and kicking within the device. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The following peripherals are used by this module:

- WDT (Watchdog Timer)

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

The Watchdog module (WDT) is designed to give an added level of safety in critical systems, to ensure a system reset is triggered in the case of a deadlock or other software malfunction that prevents normal device operation.

At a basic level, the Watchdog is a system timer with a fixed period; once enabled, it will continue to count ticks of its asynchronous clock until it is periodically reset, or the timeout period is reached. In the event of a Watchdog timeout, the module will trigger a system reset identical to a pulse of the device's reset pin, resetting all peripherals to their power-on default states and restarting the application software from the reset vector.

In many systems, there is an obvious upper bound to the amount of time each iteration of the main application loop can be expected to run, before a malfunction can be assumed (either due to a deadlock waiting on hardware or software, or due to other means). When the Watchdog is configured with a timeout period equal to this upper bound, a malfunction in the system will force a full system reset to allow for a graceful recovery.

3.1. Locked Mode

The Watchdog configuration can be set in the device fuses and locked in hardware, so that no software changes can be made to the Watchdog configuration. Additionally, the Watchdog can be locked on in software if it is not already locked, so that the module configuration cannot be modified until a power on reset of the device.

The locked configuration can be used to ensure that faulty software does not cause the Watchdog configuration to be changed, preserving the level of safety given by the module.

3.2. Window Mode

Just as there is a reasonable upper bound to the time the main program loop should take for each iteration, there is also in many applications a lower bound, i.e. a *minimum* time for which each loop iteration should run for under normal circumstances. To guard against a system failure resetting the Watchdog in a tight loop (or a failure in the system application causing the main loop to run faster than expected) a "Window" mode can be enabled to disallow resetting of the Watchdog counter before a certain period of time. If the Watchdog is not reset *after* the window opens but not *before* the Watchdog expires, the system will reset.

3.3. Early Warning

In some cases it is desirable to receive an early warning that the Watchdog is about to expire, so that some system action (such as saving any system configuration data for failure analysis purposes) can be performed before the system reset occurs. The Early Warning feature of the Watchdog module allows such a notification to be requested; after the configured early warning time (but before the expiry of the Watchdog counter) the Early Warning flag will become set, so that the user application can take an appropriate action.

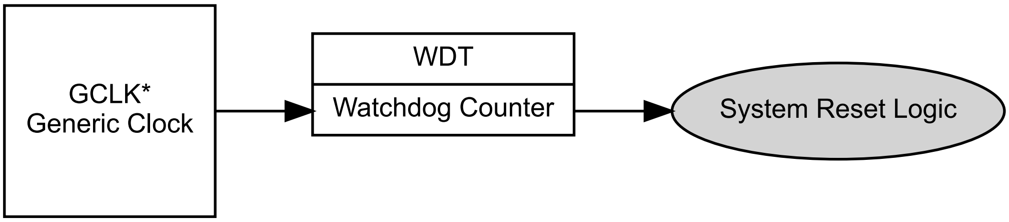
Note: It is important to note that the purpose of the Early Warning feature is *not* to allow the user application to reset the Watchdog; doing so will defeat the safety the module gives to the user application.

Instead, this feature should be used purely to perform any tasks that need to be undertaken before the system reset occurs.

3.4. Physical Connection

Figure 3-1 Physical Connection on page 7 shows how this module is interconnected within the device.

Figure 3-1. Physical Connection



Note: Watchdog Counter of SAM L21/L22 is *not* provided by GCLK, but it uses an internal 1KHz OSCULP32K output clock.

4. Special Considerations

On some devices the Watchdog configuration can be fused to be always on in a particular configuration; if this mode is enabled the Watchdog is not software configurable and can have its count reset and early warning state checked/cleared only.

5. Extra Information

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

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

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

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Callback Configuration and Initialization

7.1.1.1. Type wdt_callback_t

```
typedef void(* wdt_callback_t )(void)
```

Type definition for a WDT module callback function.

7.2. Structure Definitions

7.2.1. Struct wdt_conf

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

Table 7-1. Members

Type	Name	Description
bool	always_on	If true, the Watchdog will be locked to the current configuration settings when the Watchdog is enabled
enum gclk_generator	clock_source	GCLK generator used to clock the peripheral except SAM L21/L22/C21/C20
enum wdt_period	early_warning_period	Number of Watchdog timer clock ticks until the early warning flag is set
bool	enable	Enable/Disable the Watchdog Timer
enum wdt_period	timeout_period	Number of Watchdog timer clock ticks until the Watchdog expires
enum wdt_period	window_period	Number of Watchdog timer clock ticks until the reset window opens

7.3. Function Definitions

7.3.1. Configuration and Initialization

7.3.1.1. Function wdt_is_syncing()

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

```
bool wdt_is_syncing( void )
```

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

Returns
Synchronization status of the underlying hardware module(s).

Table 7-2. Return Values

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

7.3.1.2. Function wdt_get_config_defaults()

Initializes a Watchdog Timer configuration structure to defaults.

```
void wdt_get_config_defaults(
        struct wdt_conf *const config)
```

Initializes a given Watchdog Timer configuration structure to a set of known default values. This function should be called on all new instances of these configuration structures before being modified by the user application.

The default configuration is as follows:

- Not locked, to allow for further (re-)configuration
- Enable WDT
- Watchdog timer sourced from Generic Clock Channel 4
- A timeout period of 16384 clocks of the Watchdog module clock
- No window period, so that the Watchdog count can be reset at any time
- No early warning period to indicate the Watchdog will soon expire

Table 7-3. Parameters

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

7.3.1.3. Function wdt_set_config()

Sets up the WDT hardware module based on the configuration.

```
enum status_code wdt_set_config(
        const struct wdt_conf *const config)
```

Writes a given configuration of a WDT configuration to the hardware module, and initializes the internal device struct.

Table 7-4. Parameters

Data direction	Parameter name	Description
[in]	config	Pointer to the configuration struct

Returns
Status of the configuration procedure.

Table 7-5. Return Values

Return value	Description
STATUS_OK	If the module was configured correctly
STATUS_ERR_INVALID_ARG	If invalid argument(s) were supplied
STATUS_ERR_IO	If the Watchdog module is locked to be always on

7.3.1.4. Function wdt_is_locked()

Determines if the Watchdog timer is currently locked in an enabled state.

```
bool wdt_is_locked( void )
```

Determines if the Watchdog timer is currently enabled and locked, so that it cannot be disabled or otherwise reconfigured.

Returns
Current Watchdog lock state.

7.3.2. Timeout and Early Warning Management

7.3.2.1. Function wdt_clear_early_warning()

Clears the Watchdog timer early warning period elapsed flag.

```
void wdt_clear_early_warning( void )
```

Clears the Watchdog timer early warning period elapsed flag, so that a new early warning period can be detected.

7.3.2.2. Function wdt_is_early_warning()

Determines if the Watchdog timer early warning period has elapsed.

```
bool wdt_is_early_warning( void )
```

Determines if the Watchdog timer early warning period has elapsed.

Note: If no early warning period was configured, the value returned by this function is invalid.

Returns
Current Watchdog Early Warning state.

7.3.2.3. Function wdt_reset_count()

Resets the count of the running Watchdog Timer that was previously enabled.

```
void wdt_reset_count( void )
```

Resets the current count of the Watchdog Timer, restarting the timeout period count elapsed. This function should be called after the window period (if one was set in the module configuration) but before the timeout period to prevent a reset of the system.

7.3.3. Callback Configuration and Initialization

7.3.3.1. Function wdt_register_callback()

Registers an asynchronous callback function with the driver.

```
enum status_code wdt_register_callback(
        const wdt_callback_t callback,
        const enum wdt_callback type)
```

Registers an asynchronous callback with the WDT driver, fired when a given criteria (such as an Early Warning) is met. Callbacks are fired once for each event.

Table 7-6. Parameters

Data direction	Parameter name	Description
[in]	callback	Pointer to the callback function to register
[in]	type	Type of callback function to register

Returns
Status of the registration operation.

Table 7-7. Return Values

Return value	Description
STATUS_OK	The callback was registered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

7.3.3.2. Function wdt_unregister_callback()

Unregisters an asynchronous callback function with the driver.

```
enum status_code wdt_unregister_callback(
        const enum wdt_callback type)
```

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

Table 7-8. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to unregister

Returns
Status of the de-registration operation.

Table 7-9. Return Values

Return value	Description
STATUS_OK	The callback was Unregistered successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

7.3.4. Callback Enabling and Disabling

7.3.4.1. Function wdt_enable_callback()

Enables asynchronous callback generation for a given type.

```
enum status_code wdt_enable_callback(
        const enum wdt_callback type)
```

Enables asynchronous callbacks for a given callback type. This must be called before an external interrupt channel will generate callback events.

Table 7-10. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to enable

Returns

Status of the callback enable operation.

Table 7-11. Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

7.3.4.2. Function wdt_disable_callback()

Disables asynchronous callback generation for a given type.

```
enum status_code wdt_disable_callback(
        const enum wdt_callback type)
```

Disables asynchronous callbacks for a given callback type.

Table 7-12. Parameters

Data direction	Parameter name	Description
[in]	type	Type of callback function to disable

Returns

Status of the callback disable operation.

Table 7-13. Return Values

Return value	Description
STATUS_OK	The callback was disabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

7.4. Enumeration Definitions

7.4.1. Callback Configuration and Initialization

7.4.1.1. Enum wdt_callback

Enum for the possible callback types for the WDT module.

Table 7-14. Members

Enum value	Description
WDT_CALLBACK_EARLY_WARNING	Callback type for when an early warning callback from the WDT module is issued

7.4.2. Enum wdt_period

Enum for the possible period settings of the Watchdog timer module, for values requiring a period as a number of Watchdog timer clock ticks.

Table 7-15. Members

Enum value	Description
WDT_PERIOD_NONE	No Watchdog period. This value can only be used when setting the Window and Early Warning periods; its use as the Watchdog Reset Period is invalid.
WDT_PERIOD_8CLK	Watchdog period of 8 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_16CLK	Watchdog period of 16 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_32CLK	Watchdog period of 32 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_64CLK	Watchdog period of 64 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_128CLK	Watchdog period of 128 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_256CLK	Watchdog period of 256 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_512CLK	Watchdog period of 512 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_1024CLK	Watchdog period of 1024 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_2048CLK	Watchdog period of 2048 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_4096CLK	Watchdog period of 4096 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_8192CLK	Watchdog period of 8192 clocks of the Watchdog Timer Generic Clock
WDT_PERIOD_16384CLK	Watchdog period of 16384 clocks of the Watchdog Timer Generic Clock

8. Extra Information for WDT Driver

8.1. Acronyms

The table below presents the acronyms used in this module:

Acronym	Description
WDT	Watchdog Timer

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
Driver updated to follow driver type convention: • wdt_init, wdt_enable, wdt_disable functions removed • wdt_set_config function added • WDT module enable state moved inside the configuration struct
Initial Release

9. Examples for WDT Driver

This is a list of the available Quick Start guides (QSGs) and example applications for SAM Watchdog (WDT) 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 WDT - Basic
- Quick Start Guide for WDT - Callback

9.1. Quick Start Guide for WDT - Basic

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

- System reset after 2048 clocks of the Watchdog generic clock
- Always on mode disabled
- Basic mode, with no window or early warning periods

This use case sets up the Watchdog to force a system reset after every 2048 clocks of the Watchdog's Generic Clock channel, unless the user periodically resets the Watchdog counter via a button before the timer expires. If the Watchdog resets the device, a LED on the board is turned off.

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 configure_wdt(void)
{
    /* Create a new configuration structure for the Watchdog settings and fill
     * with the default module settings. */
    struct wdt_conf config_wdt;
    wdt_get_config_defaults(&config_wdt);

    /* Set the Watchdog configuration settings */
    config_wdt.always_on      = false;
#if !((SAML21) || (SAMC21) || (SAML22))
    config_wdt.clock_source   = GCLK_GENERATOR_4;
#endif
    config_wdt.timeout_period = WDT_PERIOD_2048CLK;

    /* Initialize and enable the Watchdog with the user settings */
    wdt_set_config(&config_wdt);
}
```

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

```
configure_wdt();
```

9.1.1.3. Workflow

1. Create a Watchdog module configuration struct, which can be filled out to adjust the configuration of the Watchdog.

```
struct wdt_conf config_wdt;
```

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

```
wdt_get_config_defaults(&config_wdt);
```

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 set the timeout period and lock mode of the Watchdog.

```
config_wdt.always_on      = false;
#if !((SAML21) || (SAMC21) || (SAML22))
    config_wdt.clock_source   = GCLK_GENERATOR_4;
#endif
    config_wdt.timeout_period = WDT_PERIOD_2048CLK;
```

4. Setups the WDT hardware module with the requested settings.

```
wdt_set_config(&config_wdt);
```

9.1.2. Quick Start Guide for WDT - Basic

9.1.2.1. Code

Copy-paste the following code to your user application:

```
enum system_reset_cause reset_cause = system_get_reset_cause();

if (reset_cause == SYSTEM_RESET_CAUSE_WDT) {
    port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
}
else {
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
}

while (true) {
    if (port_pin_get_input_level(BUTTON_0_PIN) == false) {
        port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

        wdt_reset_count();
    }
}
```

9.1.2.2. Workflow

1. Retrieve the cause of the system reset to determine if the Watchdog module was the cause of the last reset.

```
enum system_reset_cause reset_cause = system_get_reset_cause();
```

2. Turn on or off the board LED based on whether the Watchdog reset the device.

```
if (reset_cause == SYSTEM_RESET_CAUSE_WDT) {
    port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
}
else {
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
}
```

3. Enter an infinite loop to hold the main program logic.

   ```
   while (true) {
   ```

4. Test to see if the board button is currently being pressed.

   ```
   if (port_pin_get_input_level(BUTTON_0_PIN) == false) {
   ```

5. If the button is pressed, turn on the board LED and reset the Watchdog timer.

   ```
   port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);

   wdt_reset_count();
   ```

9.2. Quick Start Guide for WDT - Callback

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

- System reset after 4096 clocks of the Watchdog generic clock
- Always on mode disabled
- Early warning period of 2048 clocks of the Watchdog generic clock

This use case sets up the Watchdog to force a system reset after every 4096 clocks of the Watchdog's Generic Clock channel, with an Early Warning callback being generated every 2048 clocks. Each time the Early Warning interrupt fires the board LED is turned on, and each time the device resets the board LED is turned off, giving a periodic flashing pattern.

9.2.1. Setup

9.2.1.1. Prerequisites

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

9.2.1.2. Code

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

```
void watchdog_early_warning_callback(void)
{
    port_pin_set_output_level(LED_0_PIN, LED_0_ACTIVE);
}

void configure_wdt(void)
{
    /* Create a new configuration structure for the Watchdog settings and fill
     * with the default module settings. */
    struct wdt_conf config_wdt;
    wdt_get_config_defaults(&config_wdt);

    /* Set the Watchdog configuration settings */
    config_wdt.always_on            = false;
#if !((SAML21) || (SAMC21) || (SAML22))
    config_wdt.clock_source         = GCLK_GENERATOR_4;
#endif
    config_wdt.timeout_period       = WDT_PERIOD_4096CLK;
    config_wdt.early_warning_period = WDT_PERIOD_2048CLK;

    /* Initialize and enable the Watchdog with the user settings */
    wdt_set_config(&config_wdt);
}
```

```
void configure_wdt_callbacks(void)
{
    wdt_register_callback(watchdog_early_warning_callback,
        WDT_CALLBACK_EARLY_WARNING);

    wdt_enable_callback(WDT_CALLBACK_EARLY_WARNING);
}
```

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

```
configure_wdt();
configure_wdt_callbacks();
```

9.2.1.3. Workflow

1. Configure and enable the Watchdog driver.
   1. Create a Watchdog module configuration struct, which can be filled out to adjust the configuration of the Watchdog.

      ```
      struct wdt_conf config_wdt;
      ```

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

      ```
      wdt_get_config_defaults(&config_wdt);
      ```

      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 set the timeout and early warning periods of the Watchdog.

      ```
      config_wdt.always_on            = false;
      #if !((SAML21) || (SAMC21) || (SAML22))
          config_wdt.clock_source         = GCLK_GENERATOR_4;
      #endif
          config_wdt.timeout_period       = WDT_PERIOD_4096CLK;
      config_wdt.early_warning_period = WDT_PERIOD_2048CLK;
      ```

   4. Sets up the WDT hardware module with the requested settings.

      ```
      wdt_set_config(&config_wdt);
      ```

2. Register and enable the Early Warning callback handler.
   1. Register the user-provided Early Warning callback function with the driver, so that it will be run when an Early Warning condition occurs.

      ```
      wdt_register_callback(watchdog_early_warning_callback,
          WDT_CALLBACK_EARLY_WARNING);
      ```

   2. Enable the Early Warning callback so that it will generate callbacks.

      ```
      wdt_enable_callback(WDT_CALLBACK_EARLY_WARNING);
      ```

9.2.2. Quick Start Guide for WDT - Callback

9.2.2.1. Code

Copy-paste the following code to your user application:

```
port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);

system_interrupt_enable_global();

while (true) {
```

```
    /* Wait for callback */
}
```

9.2.2.2. Workflow

1. Turn off the board LED when the application starts.

   ```
   port_pin_set_output_level(LED_0_PIN, LED_0_INACTIVE);
   ```

2. Enable global interrupts so that callbacks can be generated.

   ```
   system_interrupt_enable_global();
   ```

3. Enter an infinite loop to hold the main program logic.

   ```
   while (true) {
       /* Wait for callback */
   }
   ```

10. Document Revision History

Doc. Rev.	Date	Comments
42124E	12/2015	Added support for SAM L21/L22, SAM DA1, SAM D09, and SAM C20/C21
42124D	12/2014	Added SAM R21 and SAM D10/D11 support
42124C	01/2014	Add SAM D21 support
42124B	06/2013	Corrected documentation typos
42124A	06/2013	Initial release

Enabling Unlimited Possibilities®

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

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death (“Safety-Critical Applications”) without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.