Atmel AVR4030: AVR Software Framework - Reference Manual

Features

- **Architecture description**
- **Code style**
- **Design style**
- **Directory structure**

1 Introduction

The Atmel® AVR® Software Framework (abbreviated ASF, www.atmel.com/asf) provides software drivers and libraries to build applications for Atmel megaAVR®, AVR XMEGA® and AVR UC3 devices. It has been designed to help develop and glue together the different components of a software design. It can easily integrate into an operating system (OS) or run as a standalone product.

In this application note developers can read about how the ASF is designed, which rules apply, how to use and develop code with the ASF.

This document is not a getting started guide but rather describes the underlying architecture of the ASF.

Atmel Microcontrollers

Application Note

Rev. 8432A-AVR-08/11

2 Software installation and setup

2.1 Downloading

The Atmel ASF is included in Atmel AVR Studio® 5 (http://www.atmel.com/avrstudio5). A separate package is available for IAR™ and AVR32 Studio users on http://www.atmel.com/asf. AVR Studio 5 users do not need this package as the ASF is integrated in AVR Studio 5.

2.2 Release notes

The ASF release notes document is available on http://www.atmel.com/asf and described:

- Supported tools
- Supported devices
- New features
- API changes
- Bug fixes
- Known issues

2.3 Bug tracker

The official AVR Software Framework bug tracker is located at http://asf.atmel.com/bugzilla/. This should be used for all bug reports regarding ASF.

2.4 Getting started

Refer to the Atmel application note AVR4029: AVR Software Framework - Getting Started, to be found on http://www.atmel.com/asf.

3 ASF directory structure

The Atmel AVR Software Framework is split in five main parts, the *avr32/* directory, the *xmega/* directory, the *mega/* directory, the *common/* directory, and the *thirdparty/* directory. These five directories represent the Atmel AVR UC3 architecture, the Atmel megaAVR, and the Atmel AVR XMEGA architecture, what is common between all architectures and finally third party libraries.

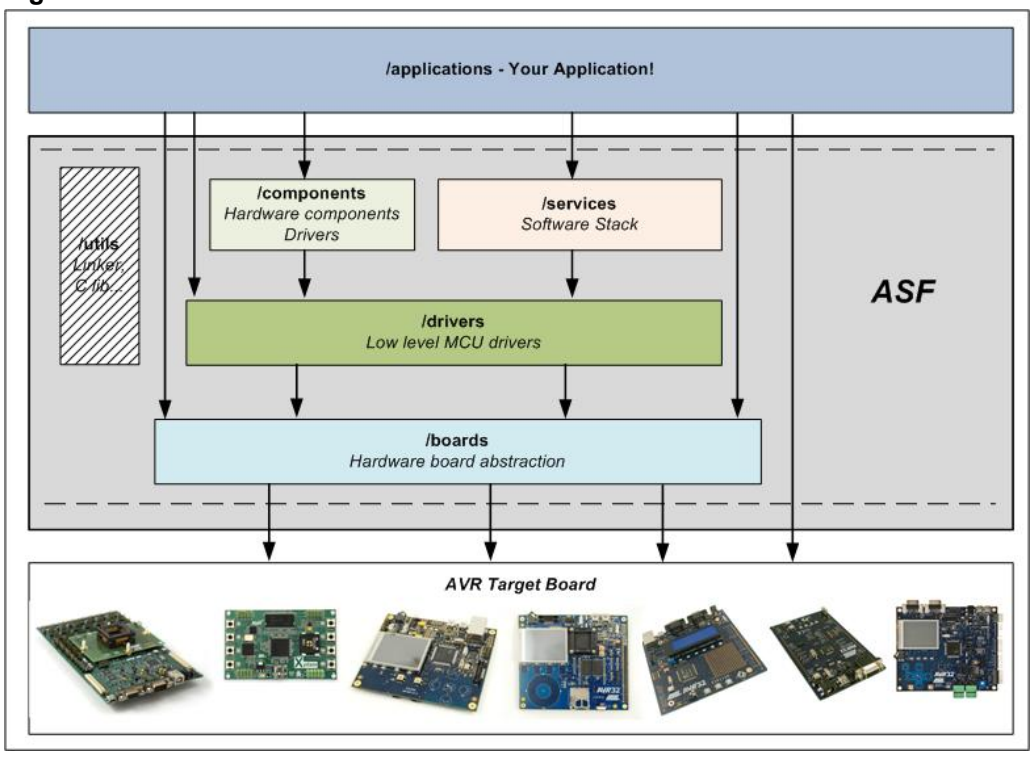
An overview of what is in the ASF root folder:

```
avr32/
common/
mega/
readme.html
thirdparty/
xmega/
```

Each architecture (and the common directory) are split into several subdirectories, these directories contains the various modules; boards, drivers, components, services and utilities. See the list below and Figure 3-1 for an overview of how the various modules are wired together.

```
applications/
boards/
components/
docsrc/
drivers/
readme.html
services/
utils/
```

Figure 3-1. Atmel AVR Software Framework modules structure.



3.1 Architecture and common directory structure

3.1.1 applications/

This directory provides application examples that are based on services, components and drivers modules. These applications are more high level and might have multiple dependencies into several modules, thus demoing advanced applications like web server, various USB demos, bootloader, audio player, etc.

3.1.2 boards/

This directory contains the various board definitions for the given architecture. The board code abstracts the modules above the board from the physical wiring, I/O initialization, initialization of external devices, etc. The board code will also identify what board features are available to the modules above.

The board entry point *board.h* header file used by all applications is located at *common/boards/board.h* since it is shared between multiple architectures.

3.1.3 components/

This directory provides software drivers to access external hardware components such as memory (for example, Atmel DataFlash®, SDRAM, SRAM, and NAND flash), displays, sensors, wireless, etc.

The components are placed in the *common/* directory if it is shared between the architectures; otherwise it is placed in the appropriate architecture directory.

3.1.4 docsrc/

This directory contains documentation shared between the various HTML documentation pages in the different modules.

3.1.5 drivers/

Each driver is composed of a driver.c and driver.h file that provides low level register interface functions to access a peripheral or device specific feature. The services and components will interface the drivers.

3.1.6 services/

This directory will provide more application oriented software such as a USB classes, FAT file system, architecture optimized DSP library, graphical library, etc.

The services are placed in the *common/* directory if it is shared between the architectures; otherwise it is placed in the appropriate architecture directory.

3.1.7 utils/

This directory provides several linker script files, common files for the build system and C/C++ files with general usage defines, macros and functions. The *utils/* directory also provide ways to make a common interface for differences between toolchains for a specific architecture.

The code utilities are placed in the *common/* directory if they are shared between the architectures; otherwise they are placed in the appropriate architecture directory.

3.2 Third party directory structure

The */thirdparty* directory is made of all software with a different license than the Atmel Corporation application note license text.

An overview of what is in the *thirdparty/* directory:

```
cyberom/
freertos/
qtouch/
…
```

Each of this module in the *thirdparty/* directory should specified a license file in the *thirdparty/*<module>/license.txt.

4 Compiler support

Atmel AVR Software Framework aims for being independent of the compiler in use; hence the various differences between compilers are stowed away in an architecture specific header file. This file is located below each architecture directory at *utils/compiler.h*.

Currently ASF supports GCC and IAR for both 8-bit and 32-bit AVR. The latest available toolchain version should be the one used for developing.

To get started with ASF tools, refer to the Atmel application note AVR4029: AVR Software Framework - Getting Started, to be found on http://www.atmel.com/asf.

4.1 AVR Studio 5

The ASF is integrated into Atmel AVR Studio 5, based on GNU GCC compiler; refer to the http://www.atmel.com/avrstudio5 for more information.

4.2 GNU compiler collection

GNU makefile are provided for all ASF projects:

- For example, for 32-bit AVR devices, the GCC project files for the GPIO peripheral bus driver example for the Atmel AT32UC3A0512 device on the Atmel EVK1100 board are located in:

  `avr32/drivers/gpio/peripheral_bus_example/at32uc3a0512_evk1100/gcc`
- For example, for Atmel AVR XMEGA devices the GCC project for the DMA driver example for the Atmel ATxmega128A1 on the Atmel AVR Xplained board are located in:

  `xmega/drivers/dma/example/atxmega128a1_xplain/gcc`

4.3 IAR Embedded Workbench

IAR Embedded Workbench® workspace are provided for ASF projects.

- For example, for 32-bit Atmel AVR devices, the IAR project files is located in:

  `avr32/drivers/gpio/peripheral_bus_example/ at32uc3a0512_evk1100/iar`
- For example, for AVR XMEGA devices the IAR project is located in:

  `xmega/drivers/dma/example/atxmega128a1_xplain/iar`

4.4 Toolchain header files

Since the toolchain header files are not bug free some routines must be followed to ensure that they become bug free.

The current way is to ship updated toolchain header files along with AVR Software Framework. Users must then update the toolchain with these to be assured the compiler is generating firmware as expected.

4.4.1 Bug reporting

When a developer encounters a bug in the current toolchain, where the toolchain already have the latest header files update, it is vital that a bug report is made.

4.4.1.1 Temporary workaround for defined symbols

While waiting for a new release of the toolchain header file, a temporary workaround is allowed in the source code. The workaround typically undefined the wrong definition and define it correctly within the source code for the specific module in development. The workaround must also have a line of documentation in front of it stating it is a workaround and should be removed when fixed in the header files.

```
//! \todo Remove workaround for bug in header files.
#undef DMA_CTRL
#define DMA_CTRL    _SFR_MEM8(0xCAFE)
```

The *#undef* line is to make sure the code does not automatically fail after the toolchain header files update. The Doxygen formatted documentation will make sure the workaround pops up when generating the documentation, which gives the developers a remainder about unfinished code.

4.4.1.2 Temporary workaround for type definitions

While waiting for a new release of the toolchain header file, a temporary workaround is allowed in the source code. The workaround is to define a new type definition as it should be, but alter the name by adding *_tmpfix* as a suffix before the *_t* part of the type definition name.

The workaround must also have a line of documentation in front of it stating it is a workaround and should be removed when fixed in the header files.

```
//! \todo Remove workaround for bug in header files.
typedef struct avr32_dmaca_tmpfix {
    unsigned long        sar0;
    (...)
} avr32_dmaca_tmpfix_t;
```

The Doxygen formatted documentation will make sure the workaround pops up when generating the documentation, which gives the developers a remainder about unfinished code.

4.4.2 Update of header files

The various architectures in Atmel ASF have a header files package located in the *utils/header_files/* directory. A *readme.txt* in the same directory instructs the user how to update his toolchain header files. No direct editing must be done in the header files package in the ASF repository.

5 Code style

This chapter contains the naming rules and general code style that is required used on all code components in the Atmel AVR Software Framework.

5.1 General naming rules

If the name of a function, variable, constant or type originates from a data sheet or other specification document, it should match the style used there as closely as possible.

For example, it's perfectly ok, and even preferable, to name the request identifier field of a USB setup request bRequest even though it violates the coding style specified on this page, as any person familiar with the USB 2.0 specification will immediately understand what the field is for.

5.2 Function and variable names

- Functions and variables are named using all lower case letters: [a-z] and [0-9]
- Underscore '_' is used to split function and variable names into more logical groups
- Variable name must be different of type name used (wrong example "static name name[2]")

5.2.1 Example

```
void this_is_a_function_prototype(void);
```

5.2.2 Rationale

All-lowercase names are easy on the eyes, and it's a very common style to find in C code.

5.3 Constants

- Constants are named using all upper case letters: [A-Z] and [0-9]
- Underscore '_' is used to split constant names into more logical groups
- Enumeration constants shall follow this rule
- Constants made from an expression must have braces around the entire expression; single value constants may skip this

5.3.1 Examples

```
#define BUFFER_SIZE             512
#define WTK_FRAME_RESIZE_WIDTH  (WTK_FRAME_RESIZE_RADIUS + 1)

enum buffer_size = {
    BUFFER_SIZE_A = 128,
    BUFFER_SIZE_B = 512,
};
```

5.3.2 Rationale

Constants should stand out from the rest of the code, and all-uppercase names ensure that. Also, all-uppercase constants are very common in specification documents and data sheets, from which many such constants originate.

The braces around an expression are vital to avoid an unexpected evaluation. For example a constant consisting of two variables added together, which are later multiplied with a variable in the source code.

Enumerations are constants too, so it makes sense to have them follow the same rule as preprocessor constants.

5.4 Type definitions

- stdint.h and stdbool.h types must be used when available
- Type definitions are named using all lower case letters: [a-z] and [0-9]
- Underscore '_' is used to split names into more logical groups
- Every type definition must have a trailing '_t'

5.4.1 Example

```
typedef uint8_t buffer_item_t;
```

5.5 Structures and unions

- Structures and unions follow the naming rule as functions and variables
- Do not use typedefs unless it is really necessary. Typedefs are only ok in the following cases:
    - The type definition is architecture-dependent and may be defined as a struct, union or scalar

5.5.1 Examples

```
struct cmd {
    uint8_t length;
    uint8_t *payload;
};

union cmd_parser {
    struct cmd cmd_a;
    struct cmd cmd_b;
};
```

5.6 Function like macro

- Function like macros follow the same naming rules as the functions and variables. This way it is easier to exchange them for inline functions at a later stage
- Where possible function like macros should be blocked into a do {} while (0)
- Function-like macros must never access their arguments more than once
- All macro arguments as well as the macro definition itself must be parenthesized

5.6.1 Example

```
#define set_io(id) do {       \
    PORTA |= (1 << (id));     \
} while (0)
```

5.6.2 Rationale

We want function-like macros to behave as much as regular functions as possible. This means that they must be evaluated as a single statement; the do { } while (0) wrapper for "void" macros and surrounding parentheses for macros returning a value ensure this.

The macro arguments must be parenthesized to avoid any surprises related to operator precedence; we want the argument to be fully evaluated before it's being used in an expression inside the macro. Also, evaluation of some macro arguments may have side effects, so the macro must ensure it is only evaluated once (sizeof and typeof expressions don't count).

5.7 Indentation

- Indentation is done by using the TAB character. This way one ensures that different editor configurations do not clutter the source code and alignment
- The TAB characters must be used before expressions/text, for the indentation
- TAB must not be used after an expression/text, use spaces instead. This will ensures good readability of the source code independent of the TAB size
- The size of the TAB character is not fixed. Anyway, it is recommended that developers use a TAB character that is large enough, so that readability is achieved. Moreover, a large indentation is an easy way to avoid deep nesting of control blocks

5.7.1 Example

```
enum scsi_asc_ascq {
[TAB]                                   [spaces]
    SCSI_ASC_NO_ADDITIONAL_SENSE_INFO         = 0x0000,
    SCSI_ASC_LU_NOT_READY_REBUILD_IN_PROGRESS = 0x0405,
    SCSI_ASC_WRITE_ERROR                      = 0x0c00,
    SCSI_ASC_UNRECOVERED_READ_ERROR           = 0x1100,
    SCSI_ASC_INVALID_COMMAND_OPERATION_CODE   = 0x2000,
    SCSI_ASC_INVALID_FIELD_IN_CDB             = 0x2400,
    SCSI_ASC_MEDIUM_NOT_PRESENT               = 0x3a00,
    SCSI_ASC_INTERNAL_TARGET_FAILURE          = 0x4400,
};
```

5.7.2 Rationale

The size of the TAB character can be different for each developer. We can not impose a fixed size. In order to have the best readability, the TAB character can only be used, on a line, before expressions and text. After expressions and text, the TAB character must not be used, use spaces instead.

The entire point about indentation is to show clearly where a control block starts and ends. With large indentations, it is much easier to distinguish the various indentation

levels from each others than with small indentations (with two-character indentations it is almost impossible to comprehend a non-trivial function).

Another advantage of large indentations is that it becomes increasingly difficult to write code with increased levels of nesting, thus providing a good motivation for splitting the function into multiple, more simple units and thus improve the readability further. This obviously requires the 80-character rule to be observed as well.

If you're concerned that using TABs will cause the code to not line up properly, please see the section about continuation.

5.8 Text formatting

- One line of code, documentation, etc. should not exceed 80 characters, given an eight spaces TAB indentation
- Text lines longer than 80 characters should be wrapped and double indented

5.8.1 Example

```
/* This is a comment which is exactly 80 characters wide for example
showing. */

    dma_pool_init_coherent(&usbb_desc_pool, addr, size,
                   sizeof(struct usbb_sw_dma_desc), USBB_DMA_DESC_ALIGN);

#define unhandled_case(value)                                          \
    do {                                                               \
            if (ASSERT_ENABLED) {                                      \
                    dbg_printf_level(DEBUG_ASSERT,                     \
                            "%s:%d: Unhandled case value %d\n",        \
                            __FILE__, __LINE__, (value));              \
                    abort();                                           \
            }                                                          \
    } while (0)
```

5.8.2 Rationale

Keeping line width below 80 characters will make sure the contents of files is viewable, even on small screens, without breaking the lines. It also helps identify obsessive levels of nesting. In general improves readability.

5.9 Space

- Put spaces around binary and ternary operators
- After an expression/text, use spaces, instead of the TAB character
- Do not put space after unary operators
- Do not put spaces between parentheses and the expression inside them
- Do not put space between function name and parameters in function calls and function definitions

5.9.1 Example

```
fat_dir_current_sect
    = ((uint32_t)(dclusters[fat_dchain_index].cluster + fat_dchain_nb_clust - 1)
            * fat_cluster_size) + fat_ptr_data + (nb_sect % fat_cluster_size);
```

5.10 Continuation

- Continuation is used to break a long expression that does not fit on a single line
- Continuation should be done by adding an extra TAB to the indentation level

5.10.1 Example

```
static void xmega_usb_udc_submit_out_queue(struct xmega_usb_udc *xudc,
            usb_ep_id_t ep_id, struct xmega_usb_udc_ep *ep)
{
    (...)
}

#define xmega_usb_read(reg) \
    mmio_read8((void *)(XMEGA_USB_BASE + XMEGA_USB_##reg))
```

5.10.2 Rationale

By indenting continuations using an extra TAB, we ensure that the continuation will always be easy to distinguish from code at both the same and the next indentation level. The latter is particularly important in if, while and for statements.

Also, by not requiring anything to be lined up (which will often cause things to end up at the same indentation level as the block which is being started), things will line up equally well regardless of the TAB size.

5.11 Comments

- Short comments may use the

```
// Comment
(...)
```

- Long (multi line) comments shall use the

```
/*
 * Long comment that might wrap multiple lines ...
 */
(...)
```

5.12 Braces

- The opening brace shall be put at the end of the line in all cases, except for function definition. The closing brace is put at the same indent level than the expression
- The code inside the braces is indented
- Single line code blocks should also be wrapped in braces

5.12.1 Examples

```
if (byte_cnt == MAX_CNT) {
    do_something();
} else {
    do_something_else();
}
```

5.13 Pointer declaration

When declaring a pointer, link the star (*) to the variable

5.13.1 Example

```
uint8_t *p1;
```

5.14 Compound statements

- The opening brace is placed at the end of the line, directly following the parenthesized expression
- The closing brace is placed at the beginning of the line following the body
- Any continuation of the same statement (for example, an 'else' or 'else if' statement, or the 'while' in a do/while statement) is placed on the same line as the closing brace
- The body is indented one level more than the surrounding code
- The 'if', 'else', 'do', 'while' and 'switch' keywords are followed by a space

5.14.1 Examples

```
if (byte_cnt == MAX1_CNT) {
    do_something();
} else if (byte_cnt > MAX1_CNT) {
    do_something_else();
} else {
    now_for_something_completely_different();
}

while (i <= 0xFF) {
    ++i;
}

do {
    ++i;
} while (i <= 0xFF);

for (i = 0; i < 0xFF; ++i) {
    do_something();
}

/* Following example shows how to break a long expression. */
for (uint8_t i = 0, uint8_t ii = 0, uint8_t iii = 0;
          (i < LIMIT_I) && (ii < LIMIT_II) && (iii == LIMIT_III);
          ++i, ++ii, ++iii) {
    do_something();
}
```

5.14.2 Rationale

This is the standard K&R style. It is both readable and space-efficient. Placing braces on separate lines doesn't contribute anything to readability, but may consume a lot of extra vertical space. The indentation ensures that the body is visually separated from the rest.

5.15 "Switch Case" statement

- The switch block follows the same rules as other compound statements
- The case labels are on the same indentation level as the switch key word
- *break* is on the same indentation level as the code inside each label
- The code inside each label is indented

5.15.1 Example

```
switch (byte_cnt) {
case 0:
        ...
        break;

case 1:
case 2:
        ...
        break;

default:
        ...
}
```

5.16 Preprocessor directives

- The # operator must always be put on the beginning of the line
- The directives are indented (if needed) after the #

5.16.1 Example

```
#if (UART_CONF == UART_SYNC)
# define INIT_CON       (UART_EN | UART_SYNC | UART_PAUSE)
#elif (UART_CONF == UART_ASYNC)
# define INIT_CON       (UART_EN | UART_ASYNC)
#elif (UART_CONF==UART_PCM)
# define INIT_CON       (UART_EN | UART_PCM | UART_NO_HOLE)
#else
# error Unknown UART configuration
#endif
```

6 Design style

6.1 Module file name and placement

- The module file name should be the same as the module name itself
- The files should be grouped in a directory named after the module
- The module directory should be placed appropriately in Atmel ASF as given by the directory structure definition
- No new top-level directories should be created
- Common modules should go in the common/ directory, architecture specific modules go into its appropriate architecture directory
    - Architecture parts of a common module should be grouped together with the common module

6.1.1 Exception

The rules above does not always make sense, or hinder a strait forward solution. Hence it is possible to loosen on the rules above, as long as the ASF maintainers approves of the deviation.

In addition special files like *conf_*.h*, main application files, etc. may deviate from the rule.

6.1.2 Examples

6.1.2.1 Location of a driver

```
{avr32, common, ...}/drivers/<module>/<module>.{c h}
avr32/drivers/gpio/gpio.c
```

6.1.2.2 Location of an architecture specific service

```
{avr32, xmega, ...}/services/<module>/<module>.{c h}
common/services/delay/delay.c
```

6.1.2.3 Location of an common service with architecture specific parts

```
common/services/<module>/<module>.{c h}
common/services/<module>/<arch>/<arch_module>.{c h}

common/services/clock/sysclk.h
common/services/clock/xmega/xmega_sysclk.h
```

6.1.2.4 Location of a component

```
{avr32, common, ...}/components/<module>/<module>.{c h}
avr32/components/touch/resistive_touch.c
```

6.1.2.5 Exception when having a sub-structure in a driver

```
avr32/drivers/usbb/usbb_device.c
avr32/drivers/usbb/usbb_host.c
avr32/drivers/usbb/usbb_otg.c
```

6.2 Common application programming interface

Atmel ASF offers some shared services and components between architectures, but it will not offer a shared interface for drivers. In addition the top level board.h is shared between all architectures along with common parts of the code utilities in the *utils/* directory.

6.2.1 Shared services

All the shared services are located in the *common/services/* directory. These common services will have an identical interface for all architectures.

6.2.2 Shared components

The shared components are located in the *common/components/* directory, and typically uses shared services to add common interfaces to external devices like Atmel DataFlash.

6.2.3 Shared code utilities

Shared utilities are located in the *common/utils/* directory, these utilities will have identical interface for all architectures. Typical shared utilities are interrupt control and standard input/output (stdio) module.

6.3 Similar application programming interface

Ideally all modules should have an identical API across the architectures, but since this is not always applicable each module should strive for a similar API. In addition the layers above can wire the sub-modules together when appropriate.

For modules shared between the various architectures the developer should reach for, as far as possible, a compatible interface. Compatible is here a way to describe an interface which can be shared with a reasonable amount of glue code (if needed at all).

The reason for doing a similar API instead of a shared API might be one of several reasons:

- Reduce flash and RAM footprint
- Reduce power consumption
- Improve performance

For some situations a similar API might not make sense at all, and an architecture specific module can be implemented. This is rationalized by different architectures are targeted against various application segments. You would not expect to see an Atmel tinyAVR® interfacing and decoding compressed music from a multimedia player.

6.3.1 Examples

The examples below shows the typical guidelines used for new modules, here hardware drivers or a communication service in specific.

6.3.1.1 Initialization

The recommended way to initialize a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_init(...)
adc_init(adc_t *adc, adc_options_t *options)
```

6.3.1.2 Enable

The recommended way to enable a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_enable(...)
adc_enable(adc_t *adc)
```

6.3.1.3 Disable

The recommended way to disable a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_disable(...)
adc_disable(adc_t *adc)
```

6.3.1.4 Start

The recommended way to start a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_start(...)
adc_start(adc_t *adc)
```

6.3.1.5 Stop

The recommended way to stop a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_stop(...)
adc_stop(adc_t *adc)
```

6.3.1.6 Write

The recommended way to write data to a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_write(...)
adc_write(adc_t *adc, uint16_t value)
```

6.3.1.7 Read

The recommended way to read data from a module is the example below; parameters may alter between architectures if similar API methodic is applied.

```
<module>_read(...)
adc_read(adc_t *adc, uint16_t *value)
```

6.4 Documentation

Atmel AVR Software Framework is based on inline documentation formatted to be read by the Doxygen documentation tool. For more information about the open source project Doxygen, visit http://www.doxygen.org/.

Doxygen bases its input on comment blocks in the source code, and triggers on a special key /** and *//!* in the input files.

All source code files should have the Atmel Corporation application note license text. In addition it should have some information about what the file is all about.

NOTE

For existing released software, the year field starts at the year the file was created, and is updated to list the year the file was last changed.

- For example, created in 2008:

  ```
  Copyright (C) 2008 Atmel Corporation. All rights reserved.
  ```

- For example, created and changed in 2008:

  ```
  Copyright (C) 2008 Atmel Corporation. All rights reserved.
  ```

- For example, created in 2008, changed in 2009:

  ```
  Copyright (C) 2008 - 2009 Atmel Corporation. All rights reserved.
  ```

- For example, created in 2008, changed in 2010:

  ```
  Copyright (C) 2008 - 2010 Atmel Corporation. All rights reserved.
  ```

- For example, created in 2006, changed in 2009, 2010, and 2011:

  ```
  Copyright (C) 2006 - 2011 Atmel Corporation. All rights reserved.
  ```

It is recommended to group the documentation into modules to provide a nicer output in the generated documentation. This is done by the Doxygen tags *\defgroup* and *\ingroup*.

Doxygen also provides a lot of functionality to improve the contents and layout of the generated documentation visit the Doxygen website and browse the documentation there to get further details.

6.4.1 Examples

6.4.1.1 Doxygen documentation opening tag for multi line comments

```
/**
 * <documentation>
 */
```

6.4.1.2 Doxygen documentation opening tag for single line comments

```
//! <documentation>
```

6.4.1.3 File documentation

```
/**
 * \file
 *
 * \brief AVR XMEGA Direct Memory Access Controller driver
 *
 * Copyright (C) 2011 Atmel Corporation. All rights reserved.
 *
 * \page License
 *
 * <Atmel Corporation application note license>
 */
```

6.4.1.4 Making a Doxygen group

```
/**
 * \defgroup sensible_group_name My module (ABBREVIATED)
 *
 * This is some contents that will show up within this group.
 */
```

6.4.1.5 Adding contents to a documentation group

```
/**
 * \ingroup sensible_group_name
 *
 * The contents written here will be merged with the contents
 * written in the previously mentioned Doxygen tag. This makes it
 * possible to split out the documentation between several files.
 */
```

6.5 API symbol definitions

API symbols are fine to use, but give them values with the corresponding values provided by the toolchain.

6.5.1 Example

Preferred way to declare API defines:

```
#define OSC_MODE_EXTERNAL      AVR32_PM_OSCCTRL0_MODE_EXT_CLOCK
```

Non-valid way to declare API defines, avoid when possible:

```
#define OSC_MODE_EXTERNAL      0
```

6.6 Hardware driver clock management

The clock module will disable the clock for most non-vital modules during initialization. For example, this means that all the PR bits are set on Atmel AVR XMEGA devices. Drivers must therefore ensure that their hardware module is clocked before interfacing them.

It is preferred that drivers use the c*lock service* to enable and disable the peripheral clock for the driver module. On modules where it is not possible to go through the c*lock service* it is of course acceptable to work directly with the hardware registers.

The *clock service* is available in the *common/services/clock* directory, both implementation and documentation. The module functions of interest are prefixed with *sysclk_*.

6.7 Hardware driver sleep management

All hardware drivers should update the *sleep manager* about which sleep level is appropriate for the current activity. It is vital that each locked sleep mode has its corresponding *unlock* call. For more information about the sleep manager see the *common/services/sleepmgr/* directory, and the module functions are prefixed with *sleepmgr_*.

6.8 Hardware driver interrupt level

- The interrupt handlers used in a driver should, to the extent possible, have a configurable interrupt level
- Configuration of the interrupt level is done through the *conf_intlvl.h* header file
- The driver must define sane default values if none are provided through *conf_intlvl.h*
- The configuration symbols should be on the form CONFIG_<MODULE>_INTLVL
    - If more specific interrupt configuration is needed, then they should appear as CONFIG_<MODULE>_<SOURCE>_INTLVL

6.8.1 Examples

```
CONFIG_DMA_INTLVL                PMIC_LVL_LOW
CONFIG_USART_DRE_INTLVL          PMIC_LVL_MEDIUM
CONFIG_USART_RXC_INTLVL          PMIC_LVL_HIGH
```

7 Table of contents

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chou-ku, Tokyo 104-0033
JAPAN
Tel: (+81) 3523-3551
Fax: (+81) 3523-7581

8432A-AVR-08/11