Atmel CryptoAuthentication

ATAES132A Firmware Development Library

USER GUIDE

Introduction

This user guide describes how to use the Atmel® CryptoAuthentication™ ATAES132A Firmware Development Library with a customized security project and how to tune it towards the hardware. To fully understand this document, it is required to have the library code base.

The ATAES132A is fully backwards compatible with the ATAES132. As a result, the ATAES132A library is an extension of the ATAES132 library. There are additional definitions added into ATAES132 library to support ATAES132A specific commands.

Features

- Layered and Modular Design
- Compact and Optimized for 8- and 32-bit Microcontrollers
- Easy to Port
- Supports I^2C and SPI Communication
- Distributed as Source Code

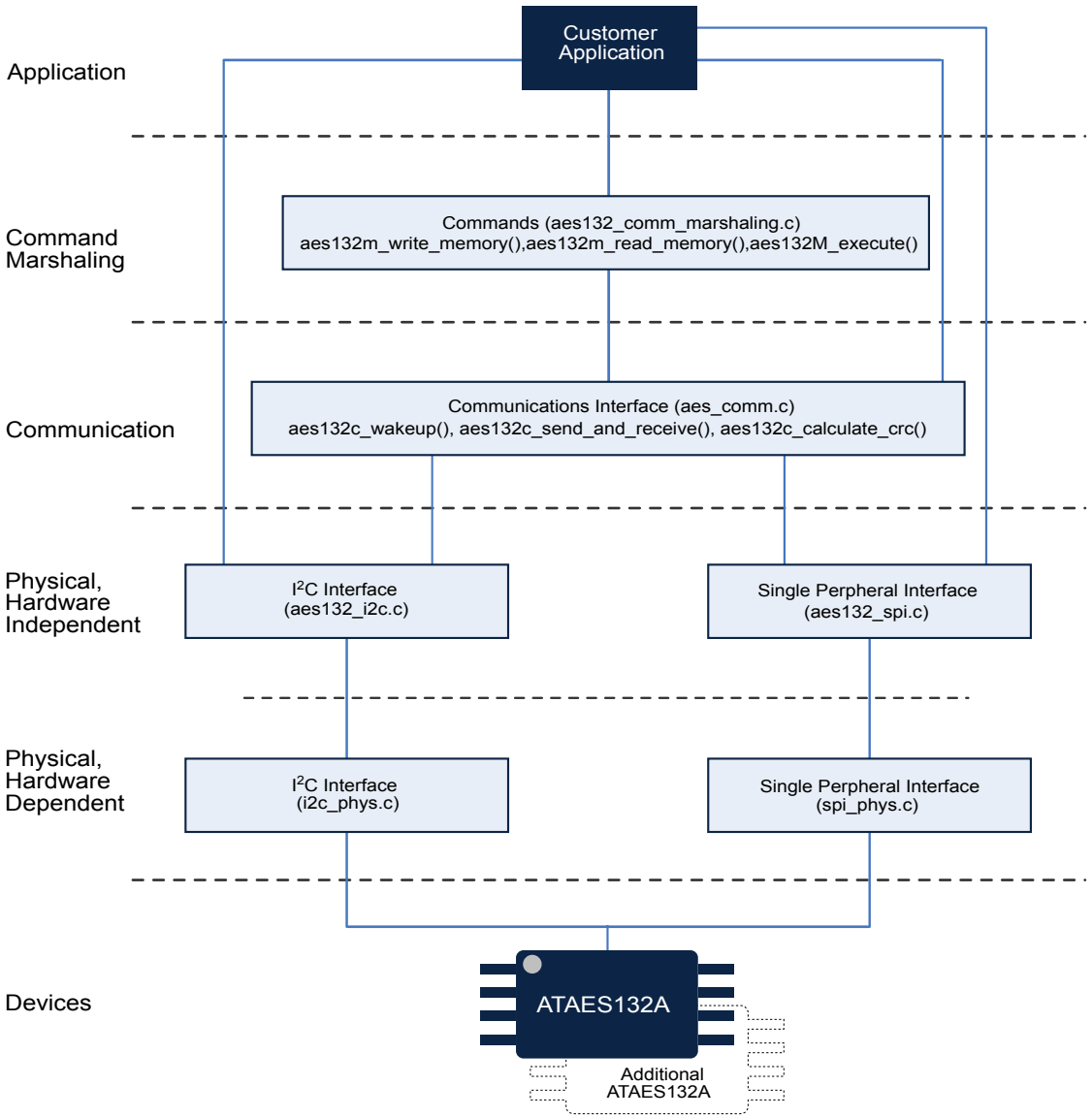
Table of Contents

1. Overview

1.1. Layered Design

The library consists of logically layered components in each successive layer. Since the library is distributed as C source code, a customer application project can use or include specific parts of the library code. For instance, compile and link the command marshaling layer functionality or exclude it. For an embedded application that only wants authentication from the device, it would make sense for that application to construct the byte stream per the device specification and communicate directly with the silicon via one of the communication methods available. This would generate the smallest code size and simplest code.

Figure 1-1. ATAES132A Library Architecture



1.1.1. Physical Layer

The Physical layer is divided into hardware-dependent and hardware-independent parts. Two physical interfaces are provided:

- I²C Interface
- Serial Peripheral Interface (SPI)

The Physical layer provides a common calling interface that abstracts the hardware. By keeping the hardware-independent function names the same for the interfaces, the driver modules can easily be exchanged in a project/makefile without touching the source code.

The hardware-dependent layer of the interface provides the mechanism to port to virtually any micro-controller. The developer only needs to implement the communication primitives for the supported interface protocol of choice, i.e. I²C or SPI. For most systems, this is the only change needed to fulfill porting requirements.

As examples, Atmel provides an implementation of the I²C and SPI interfaces for Atmel | SMART SAM D ARM® Cortex®-M0+ based microcontrollers.

1.1.2. Communication Layer

The Communication layer provides a straightforward conduit for data exchange between the device and the application software. Data exchange is based on sending a command and reading its response after command execution. This layer retries a communication sequence in case of certain communication errors reported by the Physical layer or the Device Status Register, or when there is an inconsistent response packet (value in Count byte, CRC).

1.1.3. Command Marshaling Layer

The Command Marshaling layer is built on top of the Communication layer to implement commands that the device supports. Such commands are assembled or marshaled into the correct byte streams expected by the device.

1.1.4. Application Layer

The Application Layer provides a high level abstraction of security commands to facilitate fast deployment of custom security requirements without having to deal granular cryptographic transaction minutiae.

1.2. Portability

The library has been tested for building applications and running them without errors for several target platforms, including the Atmel AVR 8 bit MCU family and SAM D ARMCortex-M0+ based micro-controllers. To make porting the library to a different target as easy as possible, specific coding rules were applied:

- No structures are used to avoid any “packed” and addressing issues on 32-bit targets.
- Functions in hardware-dependent modules (`spi_phys.c` and `i2c_phys.c`) do not “know” any specifics of the device. It will be easy to replace these functions with others from target libraries or with your own. Many I²C peripherals on 32-bit CPUs implement hardware-dependent module functionality. For such cases, porting involves discarding the hardware-dependent I²C module altogether and adapting the functions in the hardware-independent I²C module to the peripheral, or to an I²C library provided by the CPU manufacturer or firmware development tool.
- Where 16-bit variables are inserted into or extracted from a communication buffer (LSB first), no type casting is used `[(uint8_t *) &uint16_variable]`, but the MSB and LSB are calculated (`msb = uint16_variable>> 8; lsb = uint16_variable & 0xFF`). There is no need for a distinction between big-endian and little-endian targets.

- Delays and timeouts are implemented using loop counters instead of hardware timers. They need to be tuned to the specific CPU. If hardware or software timers are available in the system, possibly replace the pieces of the library that use loop counters with calls to those timer functions.

1.3. Robustness

The library applies retry mechanisms in its communication layer (`aes132_comm.c`) in case of communication failures. Therefore, there is no need for an application to implement such retries.

1.4. Optimization

In addition to the size and speed optimizations left to the compiler, certain requirements were established for the code:

- Only 8- and 16-bit variables are used, and so there is no need to import 32-bit compiler libraries. This also makes the library run faster on 8-bit targets.
- The layered architecture makes it easy to reduce code size by removing layers and/or functions that are not needed in the project.
- Robustness is enhanced through features like error checking and automatic transaction retries where necessary.
- Arrays for certain commands and memory addresses are declared as “const,” which allows compilers to skip copying such arrays to RAM at startup.

2. Project

2.1. Example Projects

Atmel provides example projects for SAM D ARMCortex-M0+ based micro-controllers. The fastest way to gain familiarity is to use an Atmel development kit, such as an Atmel SAM D21 Xplained Pro Evaluation Kit (www.atmel.com/tools/atsamd21-xpro.aspx). With this and an integrated development environment such as the freely downloadable Atmel Studio® IDE (www.atmel.com/tools/ATMELSTUDIO.aspx), makes it easy to quickly compile, download, and execute the example application that comes with the library.

2.2. Project Integration

Integrating the library into the project is straightforward. What to modify in the physical layer modules and in certain header files is explained in the following subchapters. The header file “includes” do not contain paths, but only file names. Only one compilation switch to select the interface is used. The source code fully complies with the C99 coding standards, and also complies with ANSI C with the sole exception of use of both “//” and “/*…*/” styles for commenting, often overlooked by default configurations of most C compilers.

2.2.1. Folder Structure

All modules reside in one folder. Because of this, add either the entire folder to the project and then exclude the modules not needed from compilation, or add the modules that are needed one by one. Which modules to exclude from compilation depend on the interface used. The table below shows which modules to include in the project, based on that interface used. Modules belonging to unused interface must be excluded.

Table 2-1. Interfaces Modules

Interface	SPI	I²C
Hardware Independent File	`aes132_spi.c`	`aes132_i2c.c`
Hardware Dependent Files	`spi_phys.c`	`i2c_phys.c`
	`spi_phys.h`	`i2c_phys.h`
Compilation Switch	`AES132_SPI`	`AES132_I2C`

2.2.2. Porting

When porting the library to other targets or when using CPU clock speeds other than the ones provided by the examples, certain modules have to be modified, including the physical layer modules that will be used (SPI or I²C).

2.2.2.1. Physical Layer Modules

To port the hardware-dependent modules for SPI or I²C to the target, there are several options:

- Implement the modules from scratch.
- Modify the SPI or I²C module(s) provided by the target library.
- Create a wrapper around the target library that matches the software interface of the ATAES132A library’s Physical layer. For instance, the target library for I²C might use parameters of different type, number, or sequence than those in the `i2c_phys.c` module [e.g., `i2c_send_bytes(uint8_t count, uint8_t *data)`].

- Modify the calls to hardware-dependent functions in the hardware-independent module for the Physical layer (`aes132_spi.c` / `aes132_i2c.c`) to match the functions in the target library. The hardware-dependent module for I²C reflects a simple I²C peripheral, where single I²C operations can be performed (Start, Stop, Write Byte, Read Byte, etc.). Many targets contain more sophisticated I²C peripherals, where registers have to be loaded first with an I²C address, a Start or Stop condition, a data buffer pointer, etc. In such cases, `aes132_i2c.c` has to be modified accordingly.

The hardware-dependent modules provided by Atmel use loop counters for timeout detection. When porting, either adjust the loop counter start values, which get decremented while waiting for flags to be set or cleared, or use hardware timers or timer services provided by a real-time operating system that is being used.

2.2.2.2. Communication Layer Timeout Tuning

The polling interval for SPI and I²C depends on each interface frequency and the CPU frequency, so they will need to be adjusted accordingly. These values are defined as `AES132_STATUS_REG_POLL_TIME_ACK and AES132_STATUS_REG_POLL_TIME_NACK` in `aes132_i2c.h` and `aes132_spi.h`.

Two descriptions follow about how to establish the `AES132_STATUS_REG_POLL_TIME_ACK` and `AES132_STATUS_REG_POLL_TIME_NACK`.

1. With an oscilloscope or logic analyzer, measure the time it takes for one loop iteration in the inner do-while loop inside the `aes132c_send_and_receive` function *or*
2. If the time for one device polling iteration cannot be measured, derive it by establishing three separate values:
   - The transmission time for one byte.
   - The transmission overhead time (for instance, setting peripheral registers or checking peripheral status).
   - The loop iteration time. Consider the following formulas:

     Time to poll the device:

     $$t_{poll} = t_{comm} + t_{commoverhead} + t_{loop}$$

     where:

     $$t_{comm(SPI)} = \frac{(4\text{-}bytes)(8\text{-}clocks)}{SPI\ Frequency}$$

     $$t_{commoverhead(SPI)} = t_{dataregister\ write} + t_{dataregister\ read}$$

     $$t_{comm(I2C,\ when\ I2C\ address\ gets\ "nacked")} = \left(\frac{(1\text{-}byte)\ (9\text{-}clocks)}{I^2C\ \ Frequency}\right) + t_{start} + t_{stop}$$

     $$t_{commoverhead(I2C,\ when\ I2C\ address\ gets\ "nacked")} = t_{exexcutestart\ function} + t_{dataregister\ write} + t_{executestop\ function}$$

     $$t_{comm(I2C,\ when\ I2C\ address\ gets\ "acked")} = \left(\frac{(5\text{-}bytes)\ (9\text{-}clocks)}{I^2C\ \ Frequency}\right) + 2\left(t_{start}\right) + t_{stop}$$

$t_{commoverhead(I2C,\ when\ I2C\ address\ gets\ "acked")} = 2\left(t_{exexcutestart\ function}\right) + 5\left(t_{dataregister\ write}\right) + t_{executestop\ function}$

$t_{loop} = t_{loop(aes132\_send\_and\_receive)}$(do while loop inside function)

SPI example, clocked at 8MHz:

$t_{comm(SPI)} = \frac{32}{8MHz} = 4.0\mu s$

$t_{commoverhead(SPI)} = 5.8\mu s$

$t_{loop(SPI)} = 13.2\mu s$

$t_{poll(I2C,\ whenI2C\ \ address\ gets\ "nacked")} = 4.0\mu s + 5.8\mu s + 13.2\mu s + 23.0\mu s$

I²C example, clocked at 200kHz:

$t_{comm(I2C,\ when\ I2C\ address\ \ gets\ \ "nacked")} = \frac{9}{200kHz} + \ 2.2\mu s + \ 0.8\mu s = \ 48.0\mu s$

$t_{commoverhead(I2C,\ when\ I2C\ address\ \ gets\ \ "nacked")} = \ 18.6\mu s$

$t_{comm(I2C,\ when\ I2C\ address\ \ gets\ \ "acked")} = \frac{(5)(9)}{200kHz} + \ 2(2.2\mu s) + \ 0.8\mu s = \ 230.2\mu s$

$t_{commoverhead(I2C,\ when\ I2C\ address\ \ gets\ \ "acked")} = \ 27.3\mu s$

$t_{loop(I2C)} = \ 13.0\mu s$

$t_{poll(I2C,\ \ when\ I2C\ address\ \ gets\ \ "nacked")} = 48.0\mu s + \ 18.6\mu s + 13.0\mu s = \ 79.6\mu s$

$t_{poll(I2C,\ \ when\ I2C\ \ address\ \ gets\ \ "acked")} = 230.2\mu s + \ 27.3\mu s + 13.0\mu s = \ 270.5\mu s$

3. Tuning

By optionally trading-off convenient features like automatic error retries and/or modularity, the code size can be optimized and the execution speed improved. This chapter describes a few areas where one could tune the library towards even smaller code size and/or faster execution. As most of such modifications affect size and speed, they are described in unison.

3.1. Removal of Command Marshaling Layer

This modification achieves the maximum reduction of code size, but removing the command marshaling layer may require more than cursory familiarity with the library. It does not need any modifications of the library code.

3.2. Removal of Communication Layer

This modification probably achieves the maximum of a combined reduction of code size and increase in speed, but at the expense of communication robustness and ease of use. It does not need any modifications of the library code. Without the presence of the communication layer, an application has to provide the CRC for commands it is sending and for evaluating the status byte in the response. The application can still use any definitions contained it might need in `aes132_comm.h`, such as the codes for the response status byte.

There are other ways to reduce code size and increase the speed of the communication layer. It is either possible to remove the CRC check on responses or disable retries by setting `AES132_RETRY_COUNT_ERROR` and `AES132_RETRY_COUNT_RESYNC` in `aes132_comm.h` to zero.

3.3. I²C Interface Using GPIO aka "bit-bang" Instead of I²C Hardware

Usually, the maximum I²C frequency supported by an I²C hardware is 400kHz. When implemented on a fast processor, the I²C bit-bang can be chosen over I²C hardware to allow the I²C frequency to be more than 400kHz. The maximum I²C frequency which can be reached by using the bit-bang method depends on the CPU clock, GPIO pin sink current, and I²C pull-up resistor value.

Note: Although the bit-bang implementation can reach a higher frequency, the maximum I²C frequency supported by the ATAES132A device is 1Mhz; therefore, the I²C bit-bang frequency should *not* be set to be higher than 1Mhz.

A theory and example code for I²C bit-bang implementation on AVR microcontroller is described in the application note, “Atmel AVR156: TWI Master Bit Bang Driver” located at www.atmel.com/Images/doc42010.pdf.

4. Revision History

Doc. Rev.	Date	Comments
A	01/2016	Initial document release.

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