ATSHA204A, ATECC108A, and ATECC508A

Firmware Development Library

USER GUIDE

Features

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

Introduction

This user guide describes how to use the Atmel® ATSHA204A and ATECC108A/508A Firmware Development Library with your customized security project and how to tune it towards your hardware. To fully understand this document, it is required to have the library code base available at:

www.atmel.com/tools/CryptoAuthentication_ATECCx08A_Development_Library.aspx.

The ATECC108A and ATECC508A are fully backwards compatible with the ATSHA204A. As a result, the ATECC108A/508A library is an extension of the ATSHA204 library. All ATSHA204A references can be found in the ATECC108A/508A library as well, with the difference only in the prefix, ***eccx08***... instead of ***sha204***.... There are additional definitions added into ATECC108A/508A library to support ATECC508A specific commands.

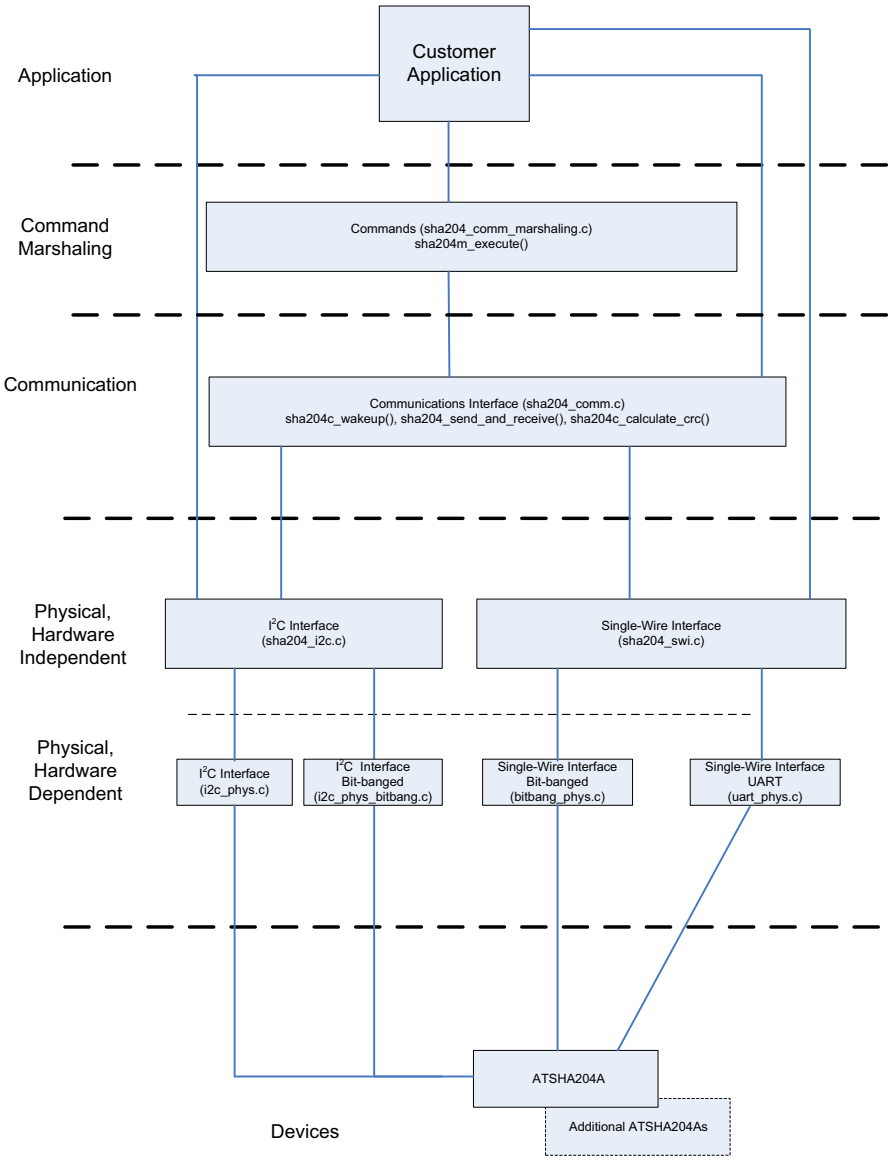
Table of Contents

Overview

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, you can 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. ATSHA204A and ATECCx08A Design



Note: The difference between the libraries are only in the prefix, ***eccx08...*** instead of ***sha204...***. There are additional definitions added into the ATECC108A library to support ATECC508A specific commands.

Physical Layer

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

- I^2C Interface
- Single-Wire Interface (SWI)

The Physical layer provides a common calling interface that abstracts the hardware (or SWI). 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.

Atmel provides an implementation of the I^2C and SWI interfaces for the Atmel AVR® AT90USB1287 microcontroller.

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).

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.

Application Layer (API)

Customers may build an API layer on top of the library to provide an easier interface for their security solution.

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. To make porting the library to a different target as easy as possible, specific coding rules were applied as noted below.

- 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 I^2C peripherals on 32-bit CPUs implement hardware-dependent module functionality. For such cases, porting involves discarding the hardware-dependent I^2C module altogether and adapting the functions in the hardware-independent I^2C module to the peripheral, or to an I^2C 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 your specific CPU. If hardware or software timers are available in your system, you might replace the pieces of the library that use loop counters with calls to those timer functions. All timing values that all layers need to access are defined in sha204_config.h.

Robustness

The library applies retry mechanisms in its communication layer (sha204_comm.c) in case of communication failures. Therefore, there is no need for an application to implement such retries.

Optimization

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

- Only 8-bit 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 your project.
- Some speed and size penalties are incurred in the communication layer (sha204_comm.c) due to increased robustness. For instance, implementing retries increases code size, while error checking (CRC, Count byte in response buffer) reduces speed and increases code size.
- Arrays for certain commands and memory addresses are declared as “const,” which allows compilers to skip copying such arrays to RAM at startup.

Example Projects

Atmel provides example projects for an AT90USB1287 microcontroller. To become familiar with the library, we advise customers to use an Atmel development kit, such as an Atmel AT88CK101 or CryptoAuthXplained PRO. With this and an integrated development environment (some can be downloaded for free, such as Atmel Studio), you will be able to rebuild the library, download the binary to the target, and start a debug session.

Project Integration

Integrating the library into your 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 compiles under C99, but should also compile under ANSI C, with the exception of double slashes used for comments.

Figure 2. CryptoAuthXplained PRO Daughterboard on an Atmel Xplained Development Board

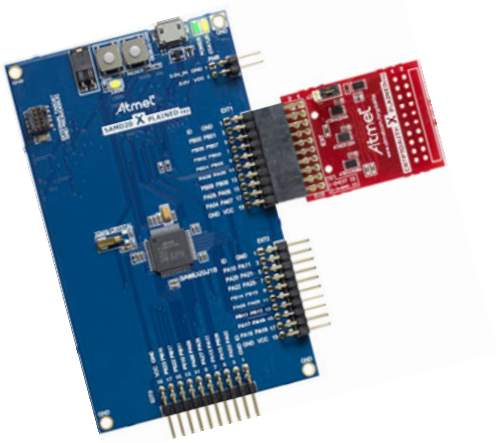
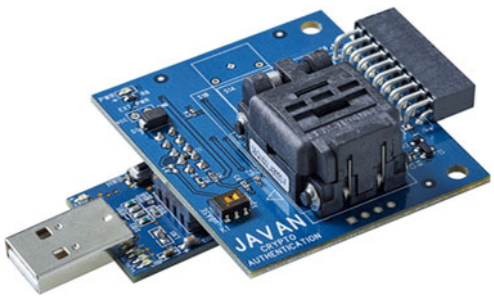


Figure 3. AT88CK101 Development Kit



Folder Structure

All modules reside in one folder. Because of this, you can either add the entire folder to your project and then exclude the modules you don't need from compilation, or you can add the modules that you do need one by one. Which modules to exclude from compilation depend on the interface you plan to use. The table below shows which modules to include in your project, depending on your interface. The modules in the other three columns must be excluded if they do not appear in the column you selected.

Table 1. Interface Modules

Interface	SWI GPIO	SWI UART	I2C	I2C_bitbang
Hardware Independent File	sha204_swi.c	sha204_swi.c	sha204_i2c.c	sha204_i2c.c
Hardware Dependent Files	bitbang_phys.c	uart_phys.c	i2c_phys.c	i2c_phys_bitbang.c
	bitbang_config.h	uart_config.h	i2c_phys.h	i2c_phys_bitbang.h
	swi_phys.h	swi_phys.h		
		avr_compatible.h		
Compilation Switch	SHA204_GPIO_BITBANG	SHA204_GPIO_UART	SHA204_I2C	SHA204_I2C_BITBANG

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 you plan to use (SWI or I^2C) and the timer_utilities.c timer function (see Section, "Timer Functions" on page 8).

Physical Layer Modules

To port the hardware-dependent modules for SWI or I^2C to your target, you have several options:

- Implement the modules from scratch.
- Modify the UART or I^2C module(s) provided by your target library.
- Create a wrapper around your target library that matches the software interface of the ATSHA204A and ATECCx08A library's Physical layer. For instance, your target library for I^2C 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 (sha204_swi.c / sha204_i2c.c) to match the functions in your target library. The hardware-dependent module for I^2C reflects a simple I^2C peripheral, where single I^2C operations can be performed (Start, Stop, Write Byte, Read Byte, etc.). Many targets contain more sophisticated I^2C peripherals, where registers have to be loaded first with an I^2C address, a Start or Stop condition, a data buffer pointer, etc. In such cases, sha204_i2c.c has to be rewritten.

The hardware-dependent modules provided by Atmel use loop counters for timeout detection. When porting, you can either adjust the loop counter start values, which get decremented while waiting for flags to be set or cleared, or you can use hardware timers or timer services provided by a real-time operating system you may be using. These values are defined in bitbang_phys.h (SWI GPIO), uart_phys.h (SWI UART), and i2c_phys.h (I2C), respectively.

Communication Layer Timeout Tuning

For SWI, it can take a maximum time of 312.5µs after sending a transmit flag until the device responds. For I²C, this time depends on the I²C clock frequency. For many AVR 8-bit CPUs, the maximum frequency is 400kHz.

For SWI, every polling cycle takes 312.5µs, while for I²C at 400kHz, every polling cycle takes 37µs. These values are defined as SHA204_RESPONSE_TIMEOUT in sha204_config.h. If you are running I²C at a frequency other than 400kHz, calculate the value using the formulas below or measure it and change the value in sha204_config.h.

Two descriptions follow about how to establish the SHA204_RESPONSE_TIMEOUT.

1. With an oscilloscope or logic analyzer, measure the time it takes for one loop iteration in the inner do-while loop inside the sha204c_send_and_receive function *or*
2. If you cannot measure the time for one device polling iteration, you can 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:

   $$t_{comm(GPIO)} = 312.5\mu s,$$

   $$t_{commoverhead(GPIO)} = 0, \text{ (negligible)}$$

   $$t_{comm(I^2C,\ when\ I^2C\ address\ gets\ "nacked")} = \frac{I^2C\_address\_byte \cdot 9\ clocks}{I^2C\ clock} + t_{start} + t_{stop},$$

   $$t_{commoverhead(I^2C)} = t_{executestart\ function} + t_{dataregister\ write} + t_{executestop\ function},$$

   $$t_{loop} = t_{loop(sha204\_send\_and\_receive)}, \text{ (do-while loop inside function)}$$

   I²C example, clocked at 200kHz:

   $$t_{comm(I^2C,\ when"nacked")} = \frac{I^2C\_address\_byte \cdot 9\ clocks}{0.2MHz} + 2.6\mu s + 3.6\mu s = 51.2\mu s,$$

   $$t_{commoverhead(I^2C)} = 18.6\mu s,$$

   $$t_{loop(I^2C)} = 13.0\mu s,$$

   $$t_{poll(I^2C)} = 51.2\mu s + 18.6\mu s + 13.0\mu s = 82.8\mu s$$

Timer Functions

The library provides two blocking timer functions, delay_10us and delay_ms. If you have hardware or software timers available in your system, you may want to replace the library timer functions with those. This way, you may be able to convert the provided blocking timer functions into non-blocking (interrupt driven or task switched) ones. Be aware that because delay_ms uses a parameter of uint8_t type, this function can only provide a delay of up to 255ms. The delay_ms function is used to read the response buffer after a memory write or command execution delay. Both are shorter than 255ms.

Tuning

By decreasing robustness, features, and/or modularity, you can decrease code size and increase execution speed. This chapter describes a few areas where you could start tuning the library towards smaller code size and/or faster execution. As most of such modifications affect size and speed, they are described in unison.

Removal of Command Marshaling Layer

This modification achieves the maximum reduction of code size, but removing the command marshaling layer makes the library more difficult to use. It does not need any modifications of the library code.

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 sha204_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. You could remove the CRC check on responses, or you could disable retries by setting SHA204_RETRY_COUNT in sha204_config.h to zero.

SWI Interface Using UART Instead of GPIO

The code space for the single-wire interface is smaller when using a UART than GPIO. Also, the code execution is allowed to be slower than when using GPIO since a UART buffers at least one byte. Since the maximum UART bit width is 4.34µs, the GPIO code has to be executed at a speed that allows reliable generation of this bit. There is no such low limit in execution speed for a UART implementation.

I^2C Interface Using GPIO bit-bang instead of an I^2C Hardware

Usually, the maximum I^2C frequency supported by an I^2C hardware is 400kHz. When implemented on a fast processor, the I^2C bit-bang can be chosen over I^2C hardware to allow the I^2C frequency to be more than 400kHz. The maximum I^2C frequency which can be reached by using the bit-bang method depends on the CPU clock, GPIO pin sink current, and I^2C pull-up resistor value.

Although the bit-bang implementation can reach a higher frequency, the maximum I^2C frequency supported by the ATECCx08A devices is 1Mhz; therefore, the I^2C bit-bang frequency should *not* be set to be higher than 1Mhz.

A theory and example code for I^2C bit-bang implementation on AVR microcontroller is described in the application note, “Atmel AVR156: TWI Master Bit Bang Driver” located at: http://www.atmel.com/Images/doc42010.pdf.

Revision History

Doc. Rev.	Date	Comments
8770D	07/2015	Updated to include the ATSHA204A and ATECCx08A, heading from “Using UART Instead of GPIO” to “SWI Interface using UART Instead of GPIO”, and section, “I^2C Interface Using GPIO bit-bang instead of an I^2C Hardware“.
8770C	01/2014	Added I^2C Interface Bit-banged.
8770B	06/2013	Added ATECC108 device and update template.
8770A	05/2011	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**