APPLICATION NOTE

CryptoAuthLib

Driver Support for Atmel CryptoAuthentication Devices

Introduction

The Atmel® CryptoAuthLib is a software library incorporated in firmware and drivers designed to work with Atmel CryptoAuthentication™ devices such as the Atmel ATECC508A and ATSHA204A CryptoAuth devices. A few of its key attributes include:

- **Ease of Use** – A Basic API serves the needs of most applications
- **Powerful** – For sophisticated applications and developers, the full power of the device is available through a core API.
- **Portable** – Runs on small processors and desktop systems alike.
- **Extensible** – Is architected to easily support new MCU platforms or protocols.
- **X.509 Certificate Support** – Has an API for storing, retrieving, and manipulating X.509 certificates.
- **TLS Integration APIs**

CryptoAuthLib is a key component of any application or device driver that requires crypto services from Atmel CryptoAuthentication devices. It is written in C and can be executed on hosts as varied as an Atmel SAMD21 ARM® M0+ Cortex, Windows PC, or embedded Linux platform.

This document discusses CryptoAuthLib, how to get started, how to incorporate it into an application, general design and use patterns, and the integration details required if CryptoAuthLib is to be ported into a hardware platform not currently supported by CryptoAuthLib.

Table of Contents

1 CryptoAuthLib Overview

CryptoAuthLib is divided into several API categories covering various features of crypto authentication application needs. The primary APIs are:

- **Basic API** – Best for ease of use.
- **Core API** – Best for use of any device feature, power developer.
- **PKI X.509 Certificates** – PKI applications which store and retrieve X.509 certificates.
- **PKI TLS** – Secure communication key agreement protocols.
- **Crypto Utilities** – General software hash implementations.
- **HAL** – Hardware abstraction layer integrates with physical interfaces.

Additional details regarding these APIs are discussed in Section 3.1, “CryptoAuthLib API Levels”.

1.1 X.509 Certificates

A critical element of most applications using an ATECC CryptoAuthentication device is the ability to store and retrieve PKI X.509 certificates from the device. CryptoAuthLib incorporates powerful certificate management features saving the developer many hours of development time writing certificate management code for a specific driver.

1.2 TLS Integration API

An increasingly common need and application of the ATECC508A is to help create secure communication channels. Using the hardware abstraction API of OpenSSL, CryptoAuthLib supports the APIs needed to integrate TLS with the ATECC508A and future ECDH-capable CryptoAuth devices.

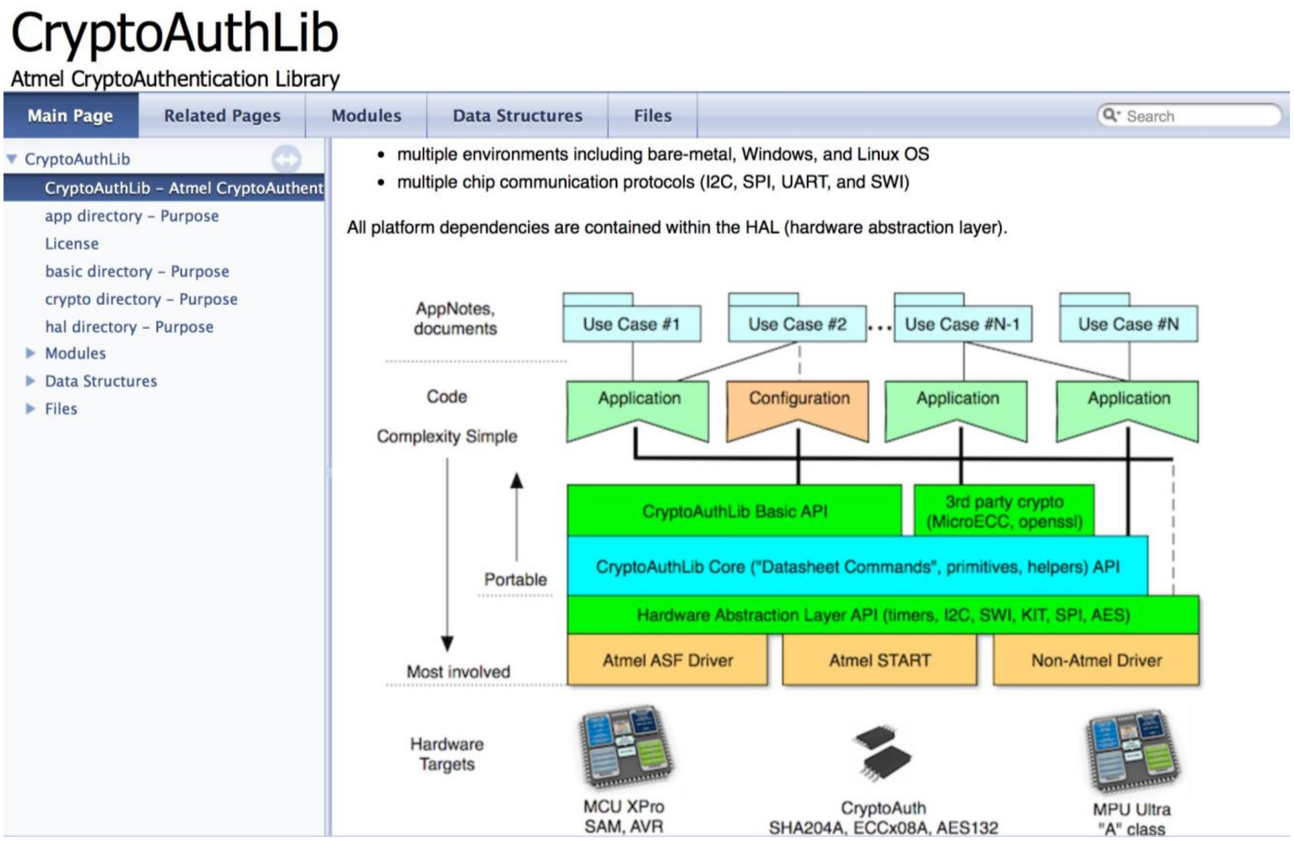
1.3 Flexible and Optimized

CryptoAuthLib is organized so the developer may easily incorporate or exclude different types of functionality from the driver or application. For example, the TLS implementation is not required if not creating a secure channel with OpenSSL and TLS. If the authentication scheme doesn’t involve certificates, the certificate support portion of the library does not need to be included.

1.4 API Documents

The CryptoAuthLib API documentation is available as a navigable linked HTML and is contained in the `docs/html` directory of the CryptoAuthLib distribution (launch `index.html` with a web browser).

Figure 1-1. CryptoAuthLib Launch Index.html

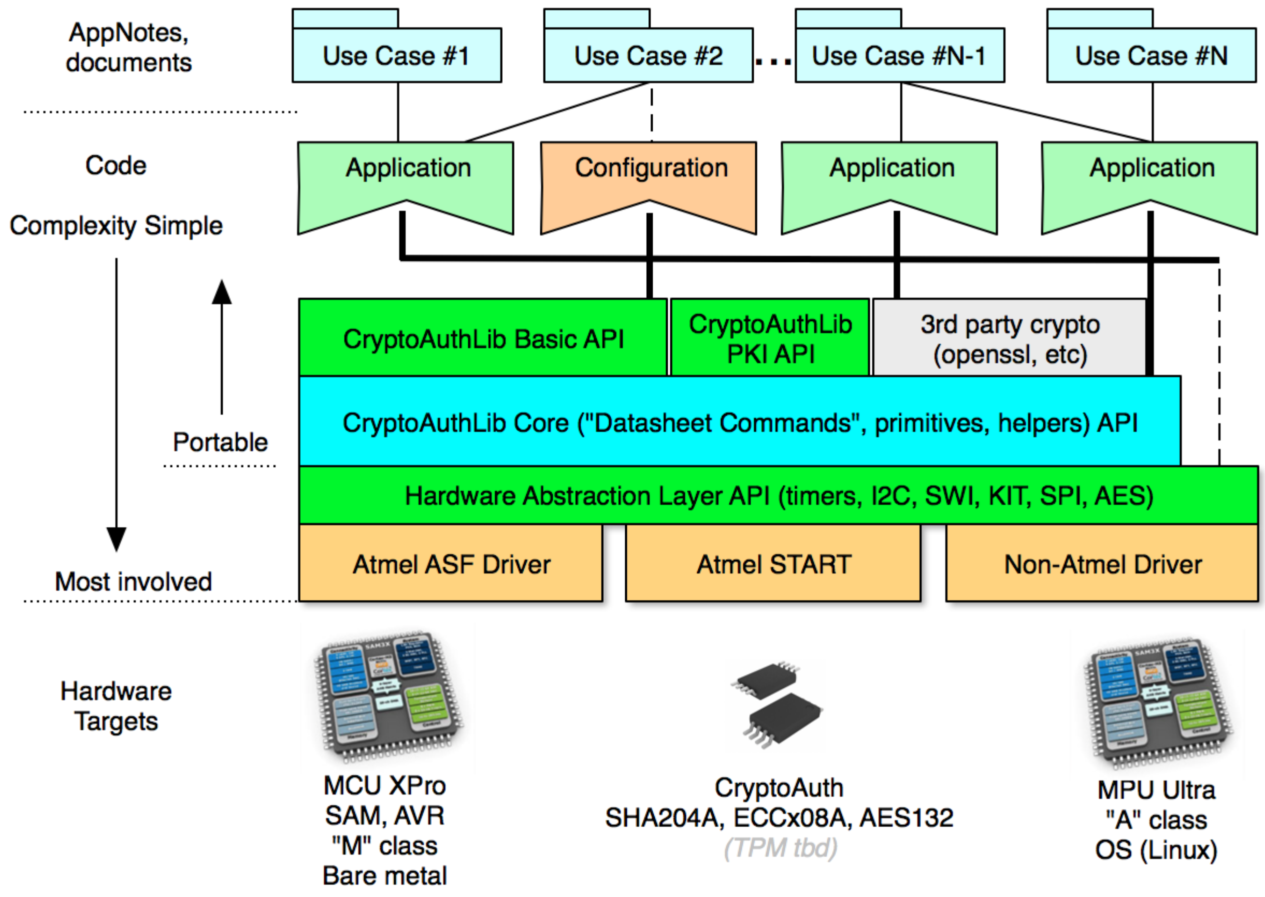


- multiple environments including bare-metal, Windows, and Linux OS
- multiple chip communication protocols (I2C, SPI, UART, and SWI)

All platform dependencies are contained within the HAL (hardware abstraction layer).

2 General Architecture

Figure 2-1. General Architecture


AppNotes, documents
Use Case #1
Use Case #2
Use Case #N-1
Use Case #N
Code
Complexity Simple
Application
Configuration
Application
Application
CryptoAuthLib Basic API
CryptoAuthLib PKI API
3rd party crypto (openssl, etc)
CryptoAuthLib Core ("Datasheet Commands", primitives, helpers) API
Portable
Hardware Abstraction Layer API (timers, I2C, SWI, KIT, SPI, AES)
Atmel ASF Driver
Atmel START
Non-Atmel Driver
Most involved
Hardware Targets
MCU XPro SAM, AVR "M" class Bare metal
CryptoAuth SHA204A, ECCx08A, AES132 (TPM tbd)
MPU Ultra "A" class OS (Linux)

3 CryptoAuthLib "Hello World"

This "hello world" example demonstrates how easy it is to perform an operation with a CryptoAuthentication device. In this case, generate a random number using the ATECC508A hardware random number generator using the CryptoAuthLib Basic API. This code is excerpted from an Atmel Studio application starting with an "LED Toggle" example for the Atmel SAM D21 and adding some CryptoAuthLib function calls to get a random number.

Hello World Example

```
#include <asf.h>
#include "cryptoauthlib.h"

int main(void)
{
        uint8_t random_number[32];

        system_init();

        /*Configure system tick to generate periodic interrupts */
        SysTick_Config(system_gclk_gen_get_hz(GCLK_GENERATOR_0));

        config_led();

        // initialize CryptoAuthLib for an ECC default I2C interface
        atcab_init(&cfg_ateccx08a_i2c_default);

        atcab_random(&random_number);  // get a random number from the chip

        // use random for challenge and authentication…

        while (true) {
                // your application code
        }
}
```

Step 1	To start, include the `cryptoauthlib.h` header file.
Step 2	Call `atcab_init()` with a pointer for a standard default configuration for an ATECC508A I^2C interface.
Step 3	Call `atcab_random()` passing a pointer to some space to receive the 32 byte random number.

This is a trivial example which demonstrates that with two basic function calls, it is easy to start exercising the CryptoAuthentication device and be on the way to building full-fledged authentication enabled applications.

`atcab_init()`needs only to be called once in the life of the application if the application will only be communicating to one CryptoAuthentication device. There are a few other things shown later which the compiler will need in order to build it, but this is the essence of jumping into CryptoAuthLib.

3.1 CryptoAuthLib API Levels

The CryptoAuthLib can be accessed at four levels:

- **Basic API** – Convenient, simple access to CryptoAuthentication devices.
- **Core API** – Powerful CryptoAuthentication Datasheet Command Primitives.
- **PKI API** – Certificates, Crypto utilities, and TLS communication support.
- **Hardware Abstraction Layer (HAL)** – Communicate to a device through a HAL hiding the physical details from the code.

3.1.1 Basic API

The Basic API is the easiest way to get started and is often the most convenient way to use the CryptoAuthentication devices for most applications whether a novice or expert.

Basic API methods manage device states such as wake, idle, and sleep to avoid built-in watchdogs, as well as account for execution times of the device. In short, some of the lower-level details working with the device are handled with the Basic API leaving the developer to focus on what the application is suppose to do.

Basic API methods tend to combine a sequence of lower level commands in order to accomplish a particular transaction type with the CryptoAuthentication device.

3.1.2 Core API

Using the Core API datasheet command primitives requires much more knowledge about how the CryptoAuthentication device works and requires that other device states are managed such as wake, idle, and sleep. If the Basic API is used, those details are managed automatically; therefore it’s recommended that the Basic API is used until device familiarity is gained or additional functionality is required from the device that isn’t in the basic API methods.

The datasheet command primitives expose the full power of the Atmel CryptoAuthentication devices for those occasions when needed. With CryptoAuthLib, the Basic API and/or the datasheet command primitives are both in the same application.

3.1.3 PKI – Certificate API

The CryptoAuthLib Certificate API provides the mechanisms needed to store X.509 certificates in an ATECC device, as well as read and reconstruct the certificate in memory.

In handling certificates, a certificate can be thought of as a two-part entity.

1. The first part is the “boiler-plate”, the data that never changes regardless of the certificate specifics. As the name suggests, the boiler-plate is effectively a template.
2. The second part is the dynamic data that changes from device to device. Dynamic data is used to drop into the certificate to make it a valid X.509 certificate.

The Certificate API helps store the dynamic data of a certificate into the ATECC device. When the full X.509 representation is needed, the Crypto API provides function calls that can be used to reconstruct the full X.509 certificate using the template and the data read from the ATECC device.

3.1.4 PKI - TLS API

The TLS API provides a set of methods to implement the OpenSSL callbacks required to integrate an ATECC508A or future ECDH capable devices with TLS. There is a separate Application Note discussing the details of the TLS API integration with OpenSSL. The TLS API provides the mechanism needed to create TLS session keys using the ECDH protocol and services from the Atmel CryptoAuthentication device.

3.1.5 Crypto Utilities API

The Crypto Utilities API provides various hash algorithm support on the host, in software such as SHA1 and SHA-256. The Basic and Core APIs provide support for SHA-256 via the CryptoAuthentication devices which support the command. The Crypto Utilities API is for application support of hosts which may not have access to a CryptoAuthentication device or requires support such as SHA1 not supported in the hardware.

3.1.6 Hardware Abstraction Layer (HAL) API

The HAL provides an extremely powerful way to separate the concerns of manipulating hardware from the CryptoAuthentication commands and higher level application use of those commands to perform authentication duties.

The HAL provides an interface to the CryptoAuthentication device using whatever communication mechanism it chooses including wire-level protocols, compiler library support for the protocol, and manages multiple busses or interface instances.

If porting CryptoAuthLib to a platform not currently supported by Atmel, it is primarily a matter of implementing the HAL API for the specific target platform. The remaining CryptoAuthLib is written to be very portable among various compilers and architectures.

3.2 CryptoAuthLib Naming Conventions

Methods include:

- **Core Datasheet API:** Start with the prefix “atca_” letters standing for “Atmel CryptoAuthentication”. This separates the C namespace in a way unlikely to conflict with functions from all the other subsystems typically found in an embedded applications
- **Basic API:** Start with “atcab_”. Anytime a method prefixed with atcab_, it came from the Basic API.
- **Helpers:** These methods fall into several categories. The main ones include:
  - **Crypto:** Software implementations of various crypto algorithms such as SHA1 and SHA256. Crypto methods are prefixed with “atcac_”.
  - **TLS (Transport Layer Security) Communication Protocols:** TLS methods are prefixed with “atcatls_”.
  - **Cert:** Functions related to certificate manipulations in memory and to and from the ATECC devices. Cert functions are prefixed with “atcacert_”
  - **Host:** Functions related to performing host-side parallel computations in order to compare with values received from the CryptoAuthentication devices. Host functions are prefixed with “atcah_”

Table 3-1. CryptoAuthLib Naming Conventions

Methods	Prefix
Core Datasheet API	atca_
Basic API	atcab_
Helpers	
Crypto	atcac_
TLS	atcatls_
Cert	atcacert_
Host	atcah_

3.3 CryptoAuthLib Directory Structure

The directory structure of CryptoAuthLib intuitively follows the basic design library as illustrated below.

Figure 3-1. CryptoAuthLib Directory Structure

```
└── cryptoauthlib
    ├── app
    ├── docs
    │   ├── cryptoauthlib-uml
    │   └── html
    ├── lib
    │   ├── atcacert
    │   ├── basic
    │   ├── crypto
    │   ├── hal
    │   ├── host
    │   └── tls
    └── test
        ├── atcacert
        └── tls
```

3.4 API Documents

The CryptoAuthLib API documentation can be accessed by going to `cryptoauthlib/docs/html/index.html`.

3.5 Placing CryptoAuthLib In Your Project

The CryptoAuthLib source tree can be placed just about anywhere in the project source tree as long as the compiler is notified on how to include files required by it. (See Section 3.6, "Compiling CryptoAuth Hello World with Atmel Studio"). However, for Atmel Studio projects, it is commonly included as a subdirectory under the ./src folder.

Figure 3-2. CryptoAuthLib Source Tree

```
└── src
    ├── ASF
    │   ├── common
    │   ├── common2
    │   ├── sam0
    │   └── thirdparty
    ├── config
    └── cryptoauthlib
        ├── app
        ├── docs
        ├── lib
        └── test
```

3.6 Compiling CryptoAuth Hello World with Atmel Studio

Of course there's more to writing any hello-world application than tossing in a few lines of code. The compiler must be notified on how to build the application, such as which files to include and exclude and which symbols to define that control the behavior of the library.

This section discusses how to setup Atmel Studio to compile CryptoAuthLib. The same principles apply to other compilers and platforms as well.

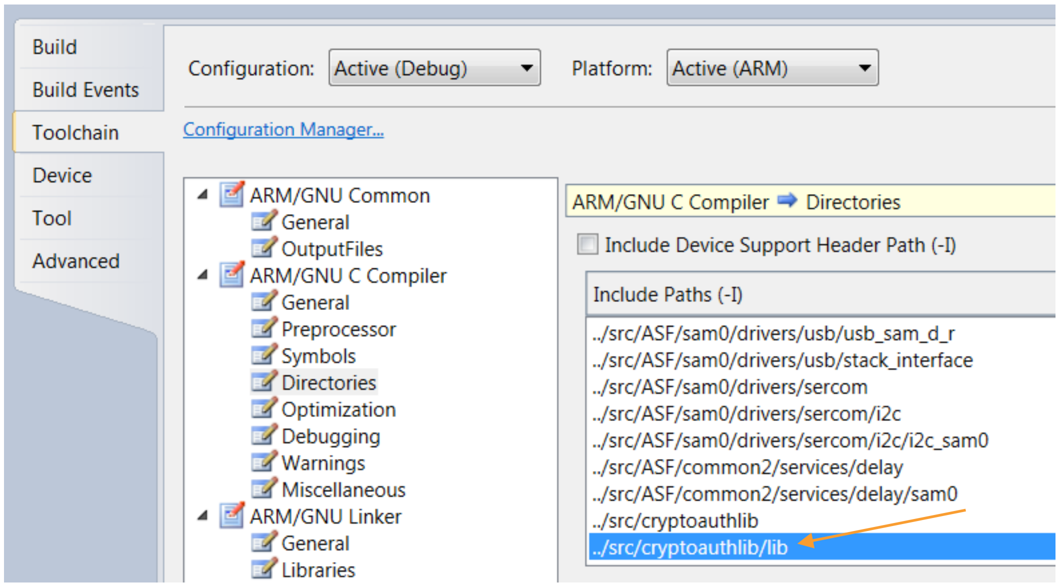
3.6.1 Set Include Path to CryptoAuthLib

Set the include path to the CryptoAuthLib directory: `src/cryptoauthlib/lib`

Do this by going to the ***Project Properties*** > select ***Toolchain*** > select ***src/cryptoauthlib/lib*** in the directories dialog list as highlighted in the screen shot below and add to include this path.

Most code and headers for the library can be found under the library directory.

Figure 3-3. CryptoAuthLib Path Selection



Configuration: Active (Debug)   Platform: Active (ARM)

☐ Include Device Support Header Path (-I)

Include Paths (-I)

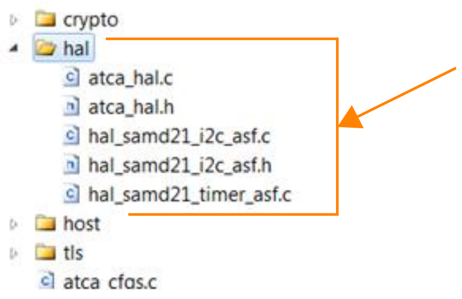
```
../src/ASF/sam0/drivers/usb/usb_sam_d_r
../src/ASF/sam0/drivers/usb/stack_interface
../src/ASF/sam0/drivers/sercom
../src/ASF/sam0/drivers/sercom/i2c
../src/ASF/sam0/drivers/sercom/i2c/i2c_sam0
../src/ASF/common2/services/delay
../src/ASF/common2/services/delay/sam0
../src/cryptoauthlib
../src/cryptoauthlib/lib
```

3.6.2 Choosing the Appropriate CryptoAuthLib HAL

Since all HAL implementations come with CryptoAuthLib, the IDE (Atmel Studio, for example) needs to be notified to include certain HAL files and exclude others.

For example, if using I^{2}C with an ASF application on a SAM D21, there's a CryptoAuthLib HAL implementation for that combination. Those files must be selected and all other HAL implementations that do not apply should be excluded.

Figure 3-4. HAL Implementation Examples



This example shows selecting the HAL for SAM D21 and ASF. The files selected are:

```
atca_hal.c
atca_hal.h
```

These are the two files that define the HAL interface for any implementation and are required regardless of which HAL chosen. Also shown selected are:

```
hal_samd21_i2c_asf.c
hal_samd21_i2c_asf.h
hal_samd21_timer_asf.c
```

These files implement the HAL API using the ASF I^{2}C API for SAM D21 (and SAM R21).

3.6.3 Defining Compiler Symbols

The CryptoAuthLib is designed to be a very flexible, but easy system to use. Part of the flexibility of CryptoAuthLib is the support for multiple types of hardware interfaces such as I^{2}C, Single-Wire, and UART interfaces. In order for the CryptoAuthLib code to know what interface type to build, one or more interface symbols must be defined.

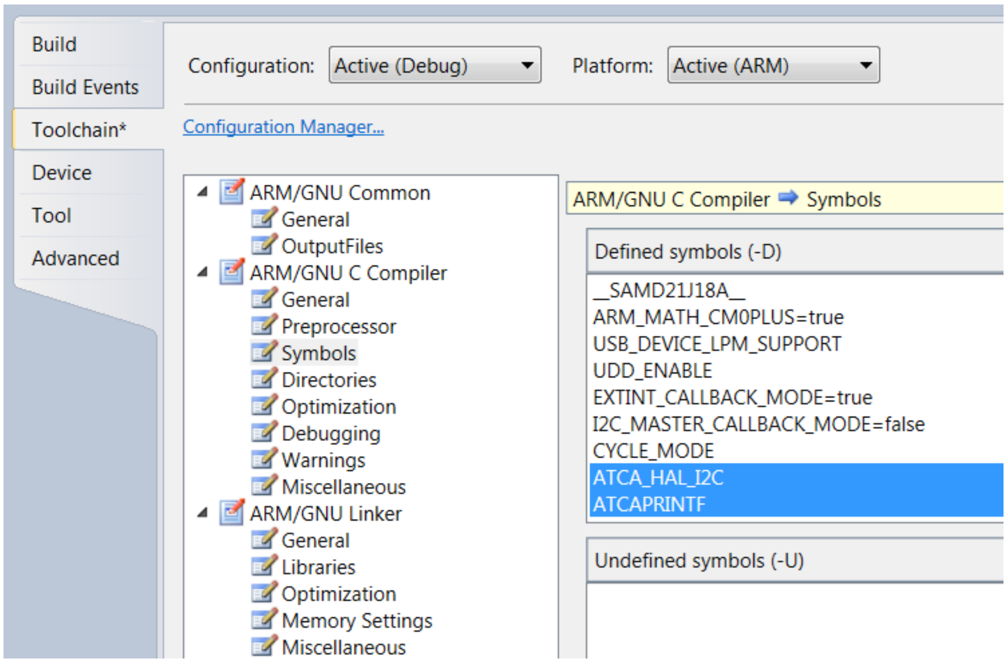
In the example below, two compiler symbols are added to an Atmel Studio project. To add two symbols, use ***Project Properties*** > select ***Toolchain*** > then select the two symbols from the ***Defined symbols (-D)*** list.

```
ATCA_HAL_I2C
ATCAPRINTF
```

`ATCA_HAL_I2C` allows the compiler to pull in a HAL implementation for I^{2}C. Other choices include:

• `ATCA_HAL_SWI`	Single-Wire Interface; typically bit-banged.
• `ATCA_HAL_UART`	Used with either Kit Protocol or using the UART to encode/decode Atmel Single-Wire Interfaces.

Figure 3-5.



Configuration: Active (Debug)
Platform: Active (ARM)

Defined symbols (-D)

```
__SAMD21J18A__
ARM_MATH_CM0PLUS=true
USB_DEVICE_LPM_SUPPORT
UDD_ENABLE
EXTINT_CALLBACK_MODE=true
I2C_MASTER_CALLBACK_MODE=false
CYCLE_MODE
ATCA_HAL_I2C
ATCAPRINTF
```

`ATCAPRINTF` allows the compiler to build the CryptoAuthLib functions which use the sprintf family of C library functions. For space optimization reasons, there may be an application for which is not wanted to be pulled into the sprintf family. In that case, do not define `ATCAPRINTF` in the project; however, certain helper functions that use printf will not be usable.

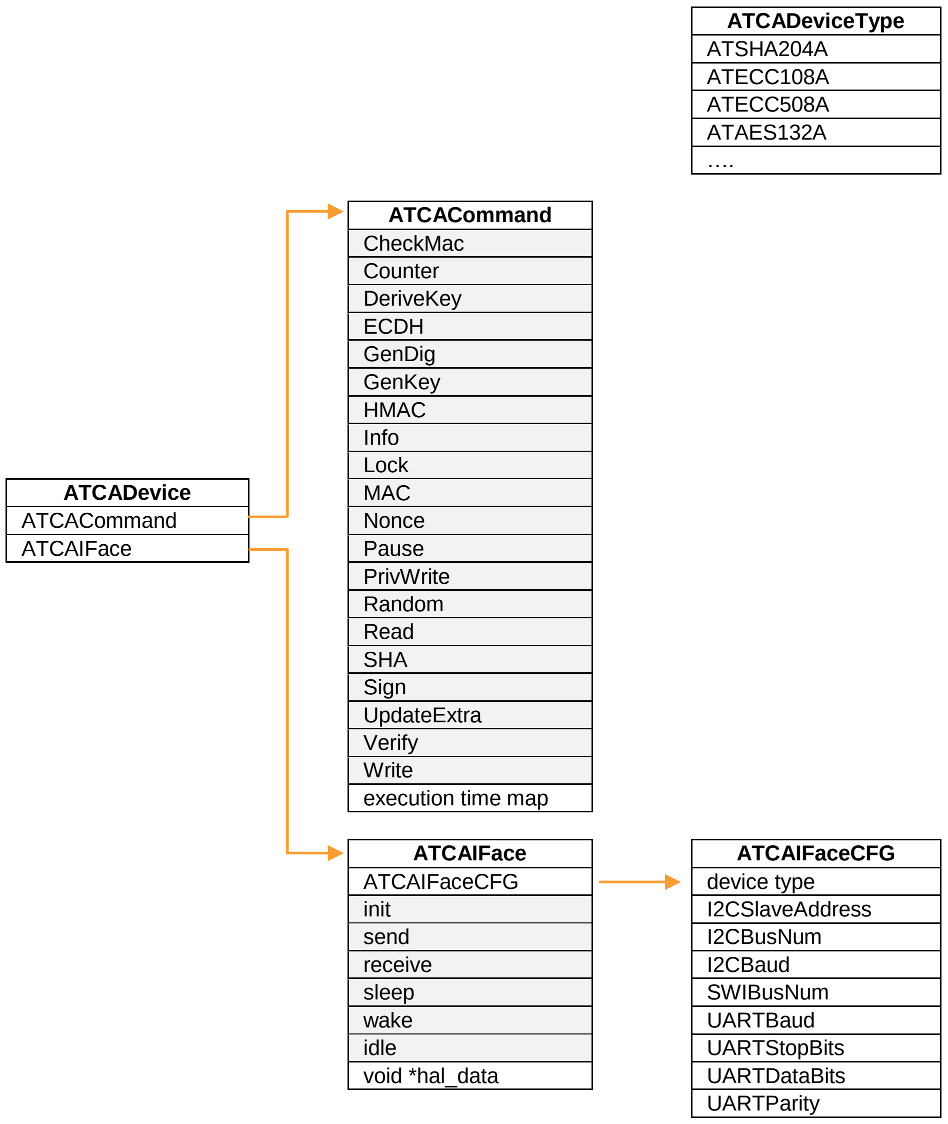
4 CryptoAuthLib Object Model

Although CryptoAuthLib is implemented in C, it follows an object-oriented design method in order to help make it easier to develop and use. If sticking with the Basic API, the details in this section are not required.

There are three primary object types in CryptoAuthLib:

- ATCAIface
- ATCACommand
- ATCADevice

Figure 4-1. CryptoAuthLib Object Design – ATCADevice



These objects form the Core API of CryptoAuthLib. The Basic API uses these objects to implement higher level functions in a simpler way so the Basic API in that sense is a client of the Core API and these objects.

4.2 ATCADevice

ATCADevice is a composite of ATCAIface and ATCACommand instances. There is one instance of an ATCADevice per physical device expected to manage the system.

In the case of the Basic API, initializing the Basic API creates an ATCADevice behind the scenes which is used for all Basic API operations. This simplifies access to the CryptoAuth device yet maintains the full power of the object model with no redundant implementations between Basic and Core APIs.

4.3 ATCAIface

ATCAIface defines an API for a logical interface representing a channel to communicate with the physical hardware, a CryptoAuthentication device. Its purpose is to hook associated HAL API methods so that all the upper layers of CryptoAuthLib do not need to be concerned with the physical level details of communicating to the device. Whether the device is a single-wire device, I^{2}C, UART, or some other physical level of a transport mechanism, the Core and Basic API levels do not need to know the details.

This allows applications to easily be ported to different CryptoAuthentication physical interfaces including I^{2}C, Single-Wire Interface, and Kit protocol without changing the application, only the configuration of the interface to communicate to the physical device.

An ATCAIface instance has a few basic methods which have analogs in the HAL implementation for the physical layer.

Besides the constructor and destructor, an ATCAIface object provides the following methods to its clients:

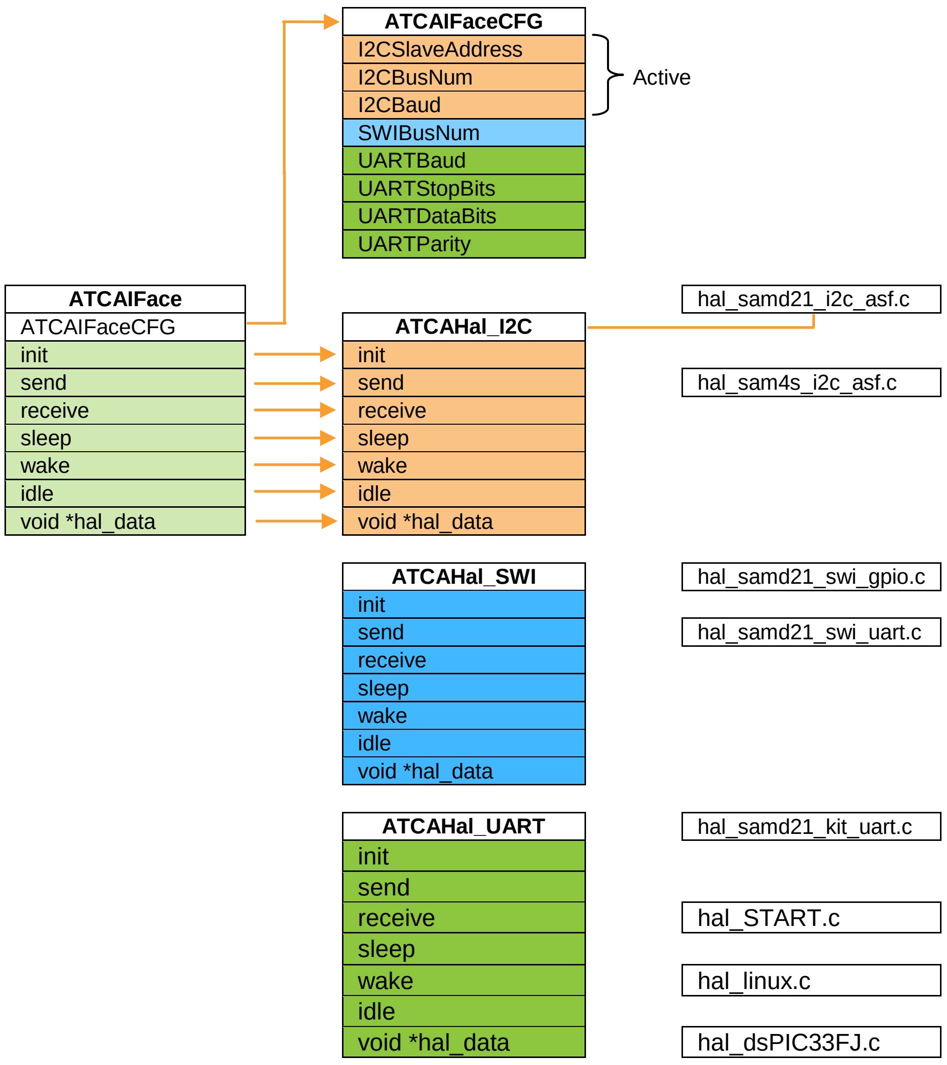
```
// IFace methods
ATCA_STATUS atinit(ATCAIface caiface);
ATCA_STATUS atsend(ATCAIface caiface, uint8_t *txdata, int txlength);
ATCA_STATUS atreceive(ATCAIface caiface, uint8_t *rxdata, uint16_t *rxlength);
ATCA_STATUS atwake(ATCAIface caiface);
ATCA_STATUS atidle(ATCAIface caiface);
ATCA_STATUS atsleep(ATCAIface caiface);

// accessors
ATCAIfaceCfg * atgetifacecfg(ATCAIface caiface);
void* atgetifacehaldat(ATCAIface caiface);
```

ATCAIface provides a way to initialize the hardware interface, send wake, idle and sleep commands, and send and receive byte streams to the actual CryptoAuthentication device.

Each of these methods is an entry point associated with an equivalent HAL method to implement the intent of the function at the physical layer.

The ATCAIface is a key concept in CryptoAuthLib and provides a major mechanism used to port the library to many different types of physical interfaces, as well as framework APIs.

Figure 4-2. CryptoAuthLib Object Design – ATCAIFace



From the above diagram, there could be one of many different physical HAL implementations that are compatible with an ATCAIface object. That is the purpose of the HAL; to implement the methods in ATCAIface in a way that the client of the ATAIface object is unaware of the physical implementation used to communicate to the device.

Since ATCAIface is a logical interface rather than a physical interface, it could be imagined that a HAL implementation using a wireless technology to communicate with a remote CryptoAuth device as though it existed locally through a hardware interface. More practically speaking, the logical interface doesn't need to change whether the HAL implements a bit-banged or UART-based single wire protocol.

4.3.1 ATCAIfaceCfg

A related object is the ATCAIfaceCfg object which describes the parameters of a logical interface. This object instance is passed to the ATCAIface constructor method. The ATCAIfaceCfg is a union of several different types of interfaces; however, only one is considered active per ATCAIfaceCfg instance. Each interface type is a structure within the union.

Each type of interface is defined in logical terms, not physical. For example, an I^{2}C bus number scheme is logically defined with an ATCAIfaceCfg object. It's the HAL implementation's job to map the logical bus number to the hardware it is targeting. Logical I^{2}C bus number 1 could mean a SERCOM1 on SAM D21, but could mean something different to a different HAL implementation.

Generally, the ATCAIfaceCfg parameters correspond to specific development boards in a way which implies some tacit knowledge of how the HAL works. For this reason, the ATCAIFaceCFG details often change based on the target runtime environment.

Some default configurations can be found in the core API directory, `atca_cfgs.c` and `atca_cfgs.h`.

Default I^{2}C Configuration for Communicating to a CryptoAuth Xplained Pro board Using a SAM D21 Xplained Pro Example

```
ATCAIfaceCfg cfg_ateccx08a_i2c_default = {
        .iface_type = ATCA_I2C_IFACE,
        .devtype = ATECC508A,
        .atcai2c.slave_address = 0xC0,
        .atcai2c.bus = 2,
        .atcai2c.baud = 400000,
        //.atcai2c.baud = 100000,
        .wake_delay = 800,
        .rx_retries = 20
};
```

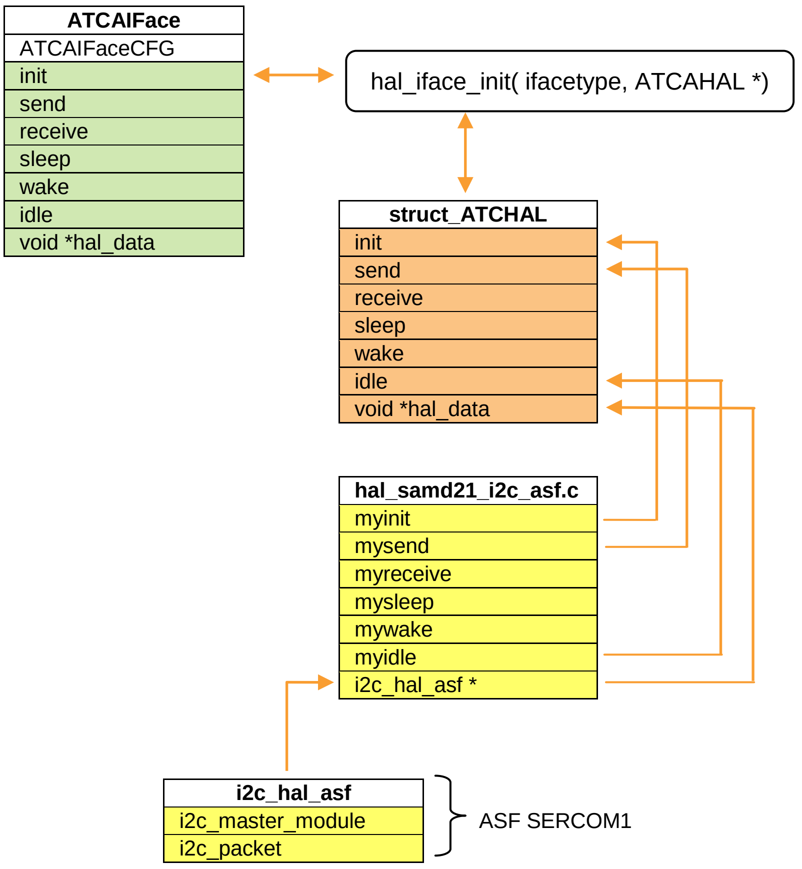
These attributes are all valid definitions regardless of whether the I^{2}C bus is implemented on a SAM D21 or even a non-Atmel MCU. It can be thought of as "I^{2}C is I^{2}C regardless of what micro is implementing it." Therefore, every I^{2}C device has an address, a baud rate, etc.

As a result, it defines various logical attributes of the I^{2}C interface including the actual interface type, the device type expected on the bus, the logical bus number (2), and the address of the device. The HAL implementation maps these logical parameters to their physical implementation, so the HAL could map logical bus 2 to any I^{2}C module it knows about. In the case of Atmel SERCOM based devices, logical bus number maps to the SERCOM number of the MCU.

4.4 How the HAL is Linked to ATCAIFace

This section covers an example showing how the HAL methods are initialized in the interface instance without having the HAL implementation bleed into the top layers. ATCAHAL is used temporality as an intermediary object to facilitate the connection, which at that point can be deleted.

Figure 4-3. CryptoAuthLib Object Design – HAL and ATCAIFace



4.5 ATCACommand

The ATCACommand is an object that builds commands as an array of bytes to send to the CryptoAuthentication device. ATCACommand does not send the command, it only builds the command. As part of that process, it accepts a packet, ATCAPacket, which contains the parameters for the command. From that information, it determines the expected length of the response packet and computes the CRC of the command to send.

5 Porting Guide for CryptoAuthLib

This section covers the design of CryptoAuthLib which allows it to be adapted to numerous architectures and platforms. The current CryptoAuthLib can be built using:

- AtmelStudio 6 GCC / ARM / ASF I2C SERCOM HAL
- AtmelStudio 7 GCC / ARM / START I2C HAL
- Microsoft Visual Studio / Windows HID HAL
- Linux GCC / CDC-Kit Protocol HAL

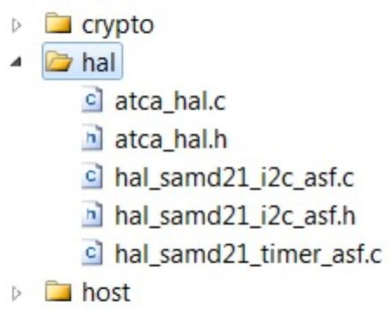
This list will continue growing over time, so check the Atmel website often for updates to CryptoAuthLib.

When it is required to write a driver for a platform for which there is not yet a HAL implementation, it is generally very straightforward to write a new HAL implementation; implement a handful of methods to tie the device communication hardware (ie: I^{2}C) to CryptoAuthLib. CryptoAuthLib is unaware of the lower level implementation details. Those details are up to the developer to decide as the HAL writer.

5.1 Writing Your Own Hardware Abstraction Layer (HAL)

CryptoAuthLib defines a hardware abstraction layer (HAL) API. The only thing required to adapt CryptoAuthLib to a new platform is to implement the HAL API targeting the hardware. Architecture specific code is isolated to the HAL implementation and a successful HAL implementation will not allow any architecture specific details to “bleed” into the upper layers.

HAL implementations belong in files in the ‘`hal`’ folder of CryptoAuthLib lib directory. There are many more `./hal/ files` published with CryptoAuthLib than will ever be used in a single application because several architectures, frameworks, and protocols are represented in different HAL implementations and not all of them will apply.



The file naming convention for a HAL implementation file is:

```
hal_[processor]_i[nterface]_[framework]
```

For example, the HAL for a SAMD21 I^{2}C using the ASF is:

```
hal_samd21_i2c_asf.c
hal_samd21_i2c_asf.h
```

For HAL timer implementation of the same example:

```
hal_samd21_timer_asf.c
```

These files are then included in the application’s Makefile or IDE project file.

5.2 HAL API

The API is quite simple. In general, for each major type of interface such as I^{2}C, Single-Wire, and UART, there is a set of HAL methods. Only the methods required to implement the particular physical interface are required. In other words, it is not required to implement the physical interfaces on the platform chosen if that interface type will not be used.

Each interface type has the following method types:

- **Init** – Called by CryptoAuthLib when a device interface is requested.
- **PostInit** – Called as the last step in the init to give HAL implementer an opportunity to perform any house-keeping required after the init is complete.
- **Send** – Called whenever a stream of bytes needs to be sent to a CryptoAuth device.
- **Receive** – Called whenever CryptoAuthLib is expected to receive bytes from the CryptoAuth device.
- **Wake** – Called when the device is about to be accessed and must be in the wake state.
- **Idle** – Called when the intermediate result in the SRAM of the CryptoAuth device (TempKey) is required and the watchdog timer of the device is likely to expire (which would erase TempKey).
- **Sleep** – Is for low-power applications which want to put the CryptoAuth device into sleep state.
- **Release** – Called when the interface usage is complete.

Each interface type has a variation on these names related to the interface name. This allows multiple types of interfaces to be used simultaneously with CryptoAuthLib. For example, an application that is a bridge between UART and Kit Protocol to an I^{2}C device can be implemented.

The only other aspect of HAL besides the device interface is the timer interface used for execution delays during CryptoAuthentication device operations.

A CryptoAuthLib ATCAIface object has exact corresponding methods that map to a specific HAL API but in the process abstracts the physical level interface from the rest of CryptoAuthLib, so it doesn't need to be concerned with the physical details of communicating with the device.

5.2.1 I^{2}C HAL API

I^{2}C HAL declarations:

```
#ifdef ATCA_HAL_I2C
ATCA_STATUS hal_i2c_init( void *hal, ATCAIfaceCfg *cfg);
ATCA_STATUS hal_i2c_post_init(ATCAIface iface);
ATCA_STATUS hal_i2c_send(ATCAIface iface, uint8_t *txdata, int txlength);
ATCA_STATUS hal_i2c_receive( ATCAIface iface, uint8_t *rxdata, uint16_t *rxlength);
ATCA_STATUS hal_i2c_wake(ATCAIface iface);
ATCA_STATUS hal_i2c_idle(ATCAIface iface);
ATCA_STATUS hal_i2c_sleep(ATCAIface iface);
ATCA_STATUS hal_i2c_release(void *hal_data );
ATCA_STATUS hal_i2c_discover_buses(int i2c_buses[], int max_buses);
ATCA_STATUS hal_i2c_discover_devices(int busNum, ATCAIfaceCfg *cfg, int *found );
#endif
```

5.2.2 SWI HAL API

Single-Wire Interface API HAL declarations:

```
#ifdef ATCA_HAL_SWI
ATCA_STATUS hal_swi_init(void *hal, ATCAIfaceCfg *cfg);
ATCA_STATUS hal_swi_post_init(ATCAIface iface);
ATCA_STATUS hal_swi_send(vATCAIface iface,, uint8_t *txdata, int txlength);
ATCA_STATUS hal_swi_receive( ATCAIface iface, uint8_t *rxdata, uint16_t *rxlength);
ATCA_STATUS hal_swi_wake(ATCAIface iface);
ATCA_STATUS hal_swi_idle(ATCAIface iface);
ATCA_STATUS hal_swi_sleep(ATCAIface iface);
ATCA_STATUS hal_swi_release(void *hal_data );
ATCA_STATUS hal_swi_discover_buses(int swi_buses[], int max_buses);
ATCA_STATUS hal_swi_discover_devices(int busNum, ATCAIfaceCfg *cfg, int *found);
#endif
```

5.2.3 UART HAL API

```
#ifdef ATCA_HAL_UART
ATCA_STATUS hal_uart_init(void *hal, ATCAIfaceCfg *cfg);
ATCA_STATUS hal_uart_post_init(ATCAIface iface);
ATCA_STATUS hal_uart_send(ATCAIface iface, uint8_t *txdata, int txlength);
ATCA_STATUS hal_uart_receive( ATCAIface iface, uint8_t *rxdata, uint16_t *rxlength);
ATCA_STATUS hal_uart_wake(ATCAIface iface);
ATCA_STATUS hal_uart_idle(ATCAIface iface);
ATCA_STATUS hal_uart_sleep(ATCAIface iface);
ATCA_STATUS hal_uart_release(ATCAIface iface);
ATCA_STATUS hal_uart_discover_buses(int uart_buses[], int max_buses);
ATCA_STATUS hal_uart_discover_devices(int busNum, ATCAIfaceCfg *cfg, int *found);
#endif
```

5.2.4 KIT CDC USB

```
#ifdef ATCA_HAL_KIT_CDC
ATCA_STATUS hal_kit_cdc_init(void *hal, ATCAIfaceCfg *cfg);
ATCA_STATUS hal_kit_cdc_post_init(ATCAIface iface);
ATCA_STATUS hal_kit_cdc_send(ATCAIface iface, uint8_t *txdata, int txlength);
ATCA_STATUS hal_kit_cdc_receive( ATCAIface iface, uint8_t *rxdata, uint16_t *rxlength);
ATCA_STATUS hal_kit_cdc_wake(ATCAIface iface);
ATCA_STATUS hal_kit_cdc_idle(ATCAIface iface);
ATCA_STATUS hal_kit_cdc_sleep(ATCAIface iface);
ATCA_STATUS hal_kit_cdc_release(void *hal_data);
ATCA_STATUS hal_kit_cdc_discover_buses(int i2c_buses[], int max_buses);
ATCA_STATUS hal_kit_cdc_discover_devices(int busNum, ATCAIfaceCfg *cfg, int *found);
#endif
```

5.2.5 KIT HID USB

```
#ifdef ATCA_HAL_KIT_HID
ATCA_STATUS hal_kit_hid_init(void *hal, ATCAIfaceCfg *cfg);
ATCA_STATUS hal_kit_hid_post_init(ATCAIface iface);
ATCA_STATUS hal_kit_hid_send(ATCAIface iface, uint8_t *txdata, int txlength);
ATCA_STATUS hal_kit_hid_receive(ATCAIface iface, uint8_t *rxdata, uint16_t *rxlength);
ATCA_STATUS hal_kit_hid_wake(ATCAIface iface);
ATCA_STATUS hal_kit_hid_idle(ATCAIface iface);
ATCA_STATUS hal_kit_hid_sleep(ATCAIface iface);
ATCA_STATUS hal_kit_hid_release(void *hal_data);
ATCA_STATUS hal_kit_hid_discover_buses(int i2c_buses[], int max_buses);
ATCA_STATUS hal_kit_hid_discover_devices(int busNum, ATCAIfaceCfg *cfg, int *found);
#endif
```

5.2.6 HAL for Timers

CryptoAuthLib needs very simple timing support, primarily for execution delays while the CryptoAuth device is performing an operation.   These methods generally map easily to your platform's delay timer library methods:

```
void atca_delay_us(uint32_t delay);
void atca_delay_10us(uint32_t delay);
void atca_delay_ms(uint32_t delay);
```

5.2.7 General HAL Development Approach

In creating a new HAL, the typical approach is to map each HAL API method to a library method supported by the platform's compiler and libraries. Most embedded systems have pre-built libraries for I^{2}C, GPIO, and UART. Moreover, the HAL must implement some level of interface management where it is envisioned there will be multiple instances of a CryptoAuthentication device within the system.

Bus and device discovery implementations can relatively be complicated or simple depending upon the developer's needs. To get up and running quickly, it's often easiest to leave the discovery APIs stubbed out and not used by the application until later in the application's life-cycle as it requires it.

The main notion behind the discovery APIs is to hide the complexity of interrogating the MCU for identifiable buses and the buses for devices that respond to the application. The result of a discovery is a list of ATCAIfaceCfg objects that can be used to instantiate ATCADevice objects so it is easily addressed with the full power of the CryptoAuthLib. The discovery API is primarily for dynamic hardware environments where CryptoAuthDevices may come and go on the bus. If this use-case is unnecessary, leave the discovery API stubbed out for another implementation as needed.

5.3 Considerations When Writing a HAL Implementation

Every type of communication mechanism has various considerations based on the physical attributes of the method.

5.3.1 Bus Interfaces

I^{2}C, as an example, implements a bus and therefore can have multiple devices on each bus. There may be multiple CryptoAuthentication devices on the same bus, each device having a different address. Alternativley, there could be multiple busses, each with one or more CryptoAuthentication devices on them. The HAL implementation must consider the architecture it wants to support.

CryptoAuthLib only knows about an abstract interface, ATCAIface object. That interface is tied to a particular HAL implementation. ATCAIface maintains logical configuration information for a specific device on the bus. If multiple devices are present on the same bus, the HAL must implement the notion of a local bus interface to which there are multiple interfaces reference the same bus. This way, the HAL maintains the bus master while the CryptoAuthLib maintains the notion of multiple instances of CryptoAuthentication devices.

Please have a look at existing HAL implementations found the `../hal/ directory`. For example, `hal_samd21_i2c_asf.c` implements a bus master architecture where each ATCAIface instance adds a reference count to the bus master each time the `hal_i2c_init()` method is called when an interface is instantiated. It also manages multiple busses.

5.3.2 UARTs

UART is a very flexible interface type. It can be used to drive a single-wire protocol or it can communicate typical serial data of the developer's own choosing to a remote device. An application level protocol can be layered over a physical HAL interface type and leave it at the HAL level.

For example, various Atmel CryptoAuthentication kits such as the Atmel AT88CK101-xxx, AT88CK490, and AT88CK590 kits use a serial communication protocol called Kit Protocol. There is an implementation of Kit Protocol that uses UART HAL to communicate with these kits.

This allows CryptoAuthLib implementation to run with a CDC USB port talking to AT88CK101-xxx, for example, on an Ubuntu Linux system.

Example the `./lib/hal` directory for files related to 'kit' and 'cdc'.

The Kit Protocol with UART is one example of a UART protocol. Another is the Atmel Single-Wire protocol implemented with a UART versus being bit-banged. Think of UART as a physical manifestation of a specific serial protocol that needs to be implemented.

5.3.3 HID Devices

Similar to UART, it is possible to build HAL implementations that communicate via HID USB. Check out the `./lib/hal` directory for files related 'hid' for an example HAL implementation for Windows HID.

5.3.4 Adding a Completely New Type of Physical Interface

The current CryptoAuthLib supports the following interface configurations (ATCAIfaceCfg):

- I^{2}C (most common)
- Single-Wire UART (Atmel Single-Wire Protocol)
- UART / CDC USB (Kit Protocol)
- HID

As mentioned, new serial protocols besides Kit can be added and still use the existing HAL implementations.

However, if there is a need for a completely new type of physical level interface, the process is straightforward. A new embedded structure to the union of ATCAIfaceCfg object (see atca_iface.h) that reflects the physical interface would be needed.

Keep in mind the ATCAIFaceCFG is explicitly defined to be a “Logical” reflection of the interface type. For example, in I^{2}C there is a logical bus number. The HAL implementation can map that logical bus number to any physical bus of his or her choosing. It so happens that logical bus number 0 maps to SERCOM0 in a SAMD21 ASF implementation, and bus number 1 to SERCOM1, and so on. There is no rule that says this must be so.

The configuration structure holds logical data about an interface, but the HAL is the one that maps that to a physical manifestation of that logical device.

6 Updating an Application With New CryptoAuthLib Releases

In general, Atmel strive's to keep the CryptoAuthLib API as stable as possible so that updating the application to use a new release of CryptoAuthLib is simple and pain-free. Atmel build's its own set of applications for professional personalization services and other cases, so Atmel is sensitive to the need to keep the API stable as possible.

In most cases, it's a simple matter of replacing the CryptoAuthLib code files in the project and recompiling (assuming the CryptoAuthLib has not been modified directly.)

The `atcab_version()` method can be used to determine the current version of the library. The library is intended to be used and released as a complete set of files; therefore, it is not advisable to modify the CryptoAuthLib files or mix and match files from one revision to another. The `atcab_version()` string is a date of the format yyyymmdd, so it is possible for the application to determine, based on the string, whether a library implementation is up to date according to its requirements. Most developers statically link CryptoAuthLib with their firmware, so compatibility is known at compile and test time.

From time to time, APIs may change, new features get added and bugs get fixed. The new library can be incorporated into a project by replacing the files found in the `cryptoauthlib` directory of the project.

Once replaced and recompiled, insure the new CryptoAuthLib update works appropriately in the run-time environment. The Unit test runners can be executed by adding the `cryptoauthlib/test` directory to the project and rebuilding.

```
atca_unit_tests(ATECC508A)
atca_basic_tests(ATECC508A)
certdata_unit_tests()
certio_unit_tests()
```

If using a CryptoAuthentication device other than ATECC508A, substitute the appropriate device name in place of the parameters above.

The cryptoauth-d21-host tester Atmel Studio solution can be reviewed for examples on how to kick off unit tests and basic tests and choose to only run specific tests relevant to the application by "cherry picking" the specific Unity tests found in the files: `atca_basic_tests.c` or `atca_unit_tests.c`.

7 Revision History

Doc Rev.	Date	Comments
8984B	01/2016	Updated to reflect latest features and API changes. Removed SPI.
8984A	10/2015	Initial document release.