21074 SEC2 LABORATORY MANUAL

Developing Secure Applications with CryptoAuthentication™ Devices

Introduction

The very foundation of security is **authentication**. Systems need to know they are communicating with authorized entities before any command and control or information sharing can take place.

Some examples of authentication are:

> Assure a battery is genuine and of the expected chemistry before it is plugged into a charger to ensure the safety of devices as well as their operators.
>
> Assure a consumable device such as an ink cartridge for a printer to prevent counterfeiting, cloning, and damaging a brand.

The Benefits of Using Hardware CryptoAuthentication® Devices:

- Unique 72 bit serial number
- SHA-256 Hash Engine
- MAC and CheckMac operations
- Hardware Key Storage
- Crypto Quality Random Number Generator
- Secure Environment for Authentication between Host and Client
- Monotonic Counters

This training covers the use of Hardware CryptoAuthentication™ devices for authentication.

The following topics will be covered:

- LAB 1 – Symmetric Authentication of a Remote Device
- LAB 2 – Symmetric Authentication between two Devices
- LAB 3 – Asymmetric Authentication of a Remote Device
- LAB 4 – Asymmetric Key Exchange using Elliptic Curve Diffie-Hellman protocol

Hardware and Software Requirements

- **Hardware Requirements**
  - (2) SAM D21 Xplained Pro (ATSAMD21-XPRO)
  - (3) CryptoAuth Xplained Pro (ATCRYPTOAUTH-XPRO)
  - (2) USB Cables (Type A to Micro B)
  - (2) Jumper Wires
- **Software Requirements**
  - Windows 7 or above
  - Atmel® Studio 7 Integrated Development Platform (Version v7.0.1417 or above)
  - Crypto Authorization Basics Project files

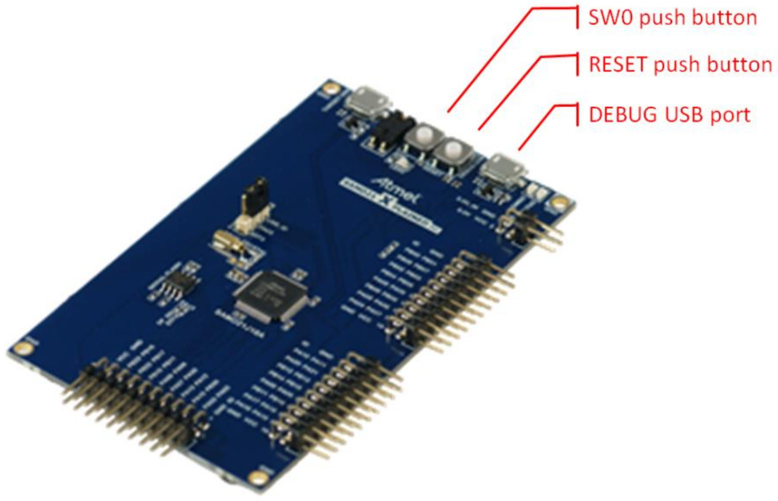
Table of Contents

Hardware Requirements

1.1 SAM D21 Xplained Pro Evaluation Kit (ATSAMD21-XPRO)

Quantity: 2

The **SAM D21 Xplained Pro** evaluation kit contains a SAM D21 ARM® Cortex®-M0+ microcontroller that you will program for the following labs. Programming and debugging functions are onboard. Just connect the DEBUG USB port to your PC running Atmel Studio 7.



1.1.1 Firmware Update

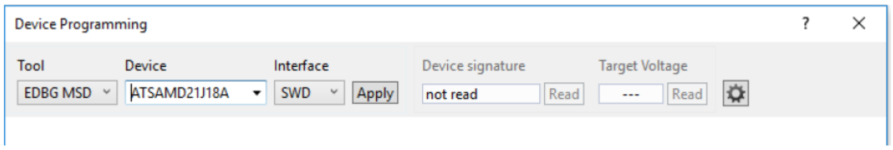
Check to see if your **SAM D21 Xplained Pro** boards have the latest embedded debug (EDBG) firmware version:

Step 1: Run Atmel Studio 7

Step 2: Connect the SAMD21 Xplained Pro board using the DEBUG USB port

Step 3: Go to the “Tools” menu of Studio 7 and select “Device Programming”

Step 4: Use the “Tool” dropdown menu and select “XPRO-EDBG ATMLxxxxx”
“ATSAMD21xxxx” appears under “Device” and “SWD” is under “Interface”



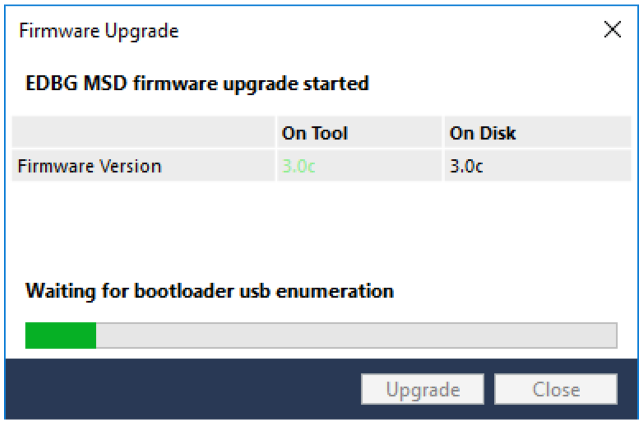
Device Programming

Tool	Device	Interface	Device signature	Target Voltage
EDBG MSD	ATSAMD21J18A	SWD	not read	---

Step 5: Click Apply

(Continued on the next page)

If the EDBG firmware needs to be updated the window shown below will appear:


Firmware Upgrade

EDBG MSD firmware upgrade started

	On Tool	On Disk
Firmware Version	3.0c	3.0c

Waiting for bootloader usb enumeration

If an upgrade is required, press “Upgrade”

If not, the device signature will fill in with the device’s unique serial# and you can close the window.

1.2 CryptoAuth Xplained Pro Evaluation Board (ATCRYPTOAUTH-XPRO)

Quantity 3: Two boards labeled “HOST” and one labeled “REMOTE”

The **CryptoAuth Xplained Pro** is an add-on evaluation board for the Xplained and Xplained Pro evaluation boards to support the ATSHA204A, ATAES132A, and ATECC508A.

1.3 USB Cable (Type A to Micro B)

Quantity: 2

1.4 Wire Jumper

Quantity: 2

2 Software Requirements

2.1 Atmel® Studio 7 Integrate Development Platform

Version v7.0.1417 or higher

Web Installer (recommended):

https://www.microchip.com/development-tools/atmel-studio-7

2.2 Crypto Authentication Library (CryptoAuthLib) Overview

Microchip provides a "C" software library to enable MCU communication with the CryptoAuthentication™ family of devices. This library is called the "CryptoAuthLib" which is an abbreviated name for Crypto Authentication Library. This library is ANSI C99 compatible. The library has a hardware abstraction layer (hal) which makes porting to any Microchip MCU extremely easy. The library provides abstraction of the low level registers and I^{2}C or SWI communication to the chip. The library can be found at:

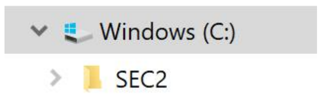
http://www.microchip.com/swlibraryweb/product.aspx?product=cryptoauthlib

The CryptoAuthLib has three public namespaces that are the primary interface into the library. These namespaces are "atcab_" (Atmel Crypto Authentication Basic), "atcah_" (Atmel Crypto Authentication Host), "atcac_" (Atmel Crypto Authentication Crypto). The atcab namespace is the primary interface to the ATCA device. Under this namespace you find all the initialization, chip provided cryptography, and general chip configuration and setup. The atcah namespace provides software simulations of the commands that are executed on an ATCA device. These commands enable advanced debugging and also simulation in the MCU when the MCU does not have a companion ATCA device to use for authentication actions. Finally the atcac namespace provides access to cryptographic commands implemented in software. These are typically used in host MCU systems that do not have hardware implementations of some cryptographic functions.

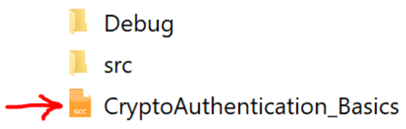
2.3 Setting up the Crypto Authentication Project

2.3.1 Decompress the Project File

The software project file comes in a compressed *.zip file. Unzip the contents to the C:/ directory of your PC. You should have a directory named "SEC2."

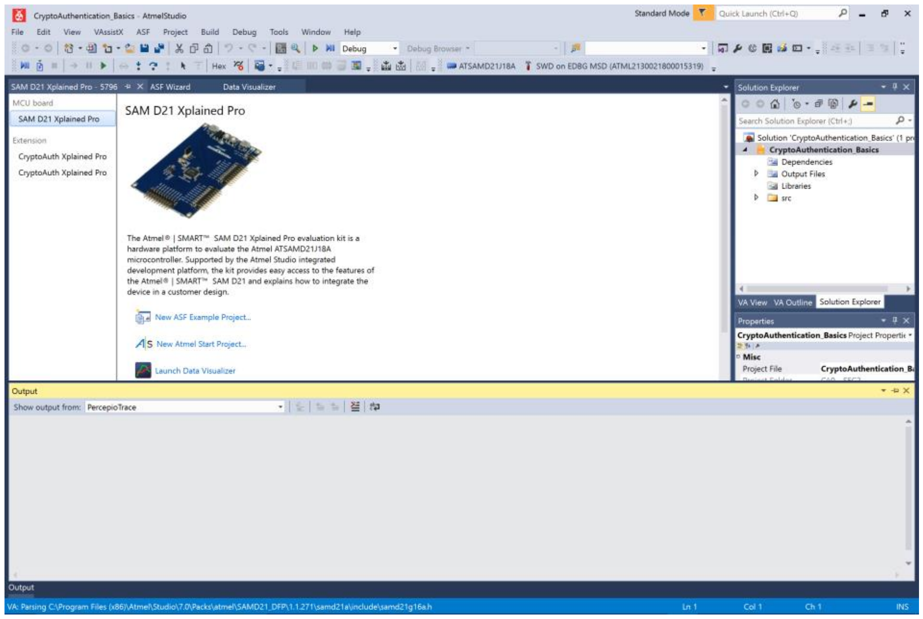


Please double click on the project file to open it. On some systems project opening is slow and no splash screen is displayed so please give it about 30 seconds to open.



2.3.2 Studio 7

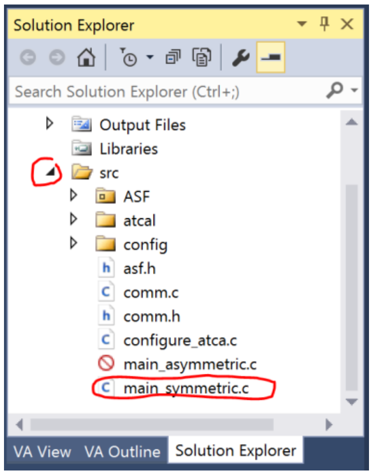
Once Studio 7 starts you should see the window shown below:


CryptoAuthentication_Basics - AtmelStudio
File Edit View VAssistX ASF Project Build Debug Tools Window Help
SAM D21 Xplained Pro - 5796
ASF Wizard
Data Visualizer
MCU board
SAM D21 Xplained Pro
Extension
CryptoAuth Xplained Pro
CryptoAuth Xplained Pro
SAM D21 Xplained Pro
New ASF Example Project...
New Atmel Start Project...
Launch Data Visualizer
Solution Explorer
Solution 'CryptoAuthentication_Basics'
CryptoAuthentication_Basics
Dependencies
Output Files
Libraries
src
Properties
Output
Show output from: PercepioTrace

The Atmel® | SMART™ SAM D21 Xplained Pro evaluation kit is a hardware platform to evaluate the Atmel ATSAMD21J18A microcontroller. Supported by the Atmel Studio integrated development platform, the kit provides easy access to the features of the Atmel® | SMART™ SAM D21 and explains how to integrate the device in a customer design.

Locate the pane on the upper right and click on the “Solution Explorer” tab if it is not already selected. This is where you will access the files in the project.

Expand the “src” folder and double click on the main project file `main_symmetric.c`. This will open `main_symmetric.c` as a tab in the main window.

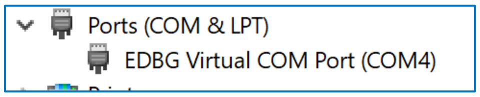


2.3.3 Open the Data Visualizer

The Data Visualizer allows you to communicate with the SAMD21 microcontroller through a serial connection using the onboard Embedded Debugger (EDBG) programming and debugging circuitry. We start by plugging the USB cable into the **SAM D21 Xplained Pro** board DEBUG USB port and PC. Then we open and configure the Data Visualizer.

Step 1: Connect the USB cable to the **SAM D21 Xplained Pro** board DEBUG USB port and the PC. The SAM D21 Xplained Pro board will then enumerate a USB serial port.

Step 2: Verify the COM port connection by opening the PC's "Device Manager" and navigating to "Ports (COM & LPT)." The port will be labeled "EDBG Virtual COM Port (COMx)." Make a note of the COM port number.

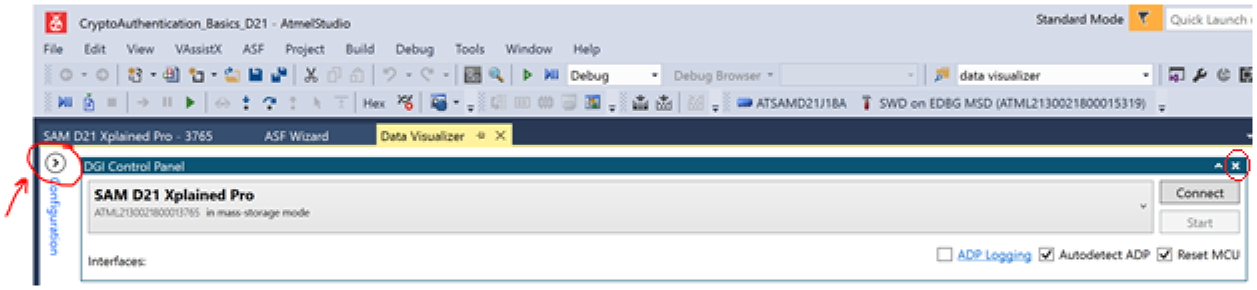


Ports (COM & LPT)
EDBG Virtual COM Port (COM4)

Step 3: In the Studio 7 menu bar, select "Tools" and then select "Data Visualizer" from the drop down box. A Data Visualizer window will display with the **SAM D21 Xplained Pro** detected.

Step 4: Click the "x" in the right corner of the DGI Control Panel to close it.

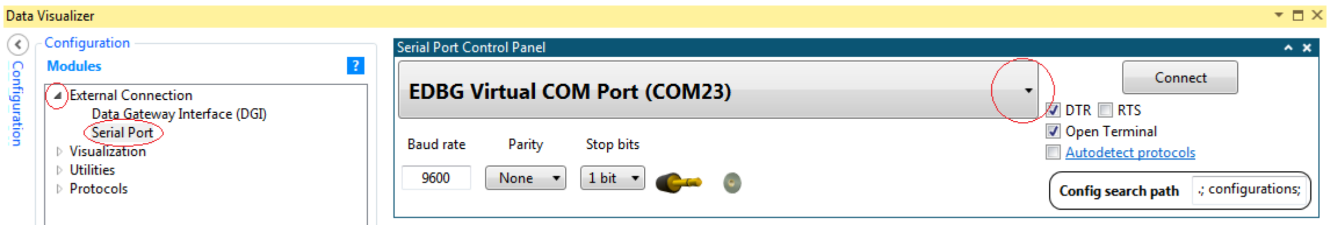
Step 5: On the left side, click on the expansion arrow labeled "Configuration":



SAM D21 Xplained Pro
ATML2130021800013765 in mass-storage mode

Step 6: Click on "External Connection" then double click on "Serial Port". On the right a "Serial Port Control Panel" will open.

Step 7: Ensure the COM port matches the one shown in the Device Manager. If it is not, click on the "down arrow" on the right hand side of the "Serial Port Control Panel" and select the COM port.

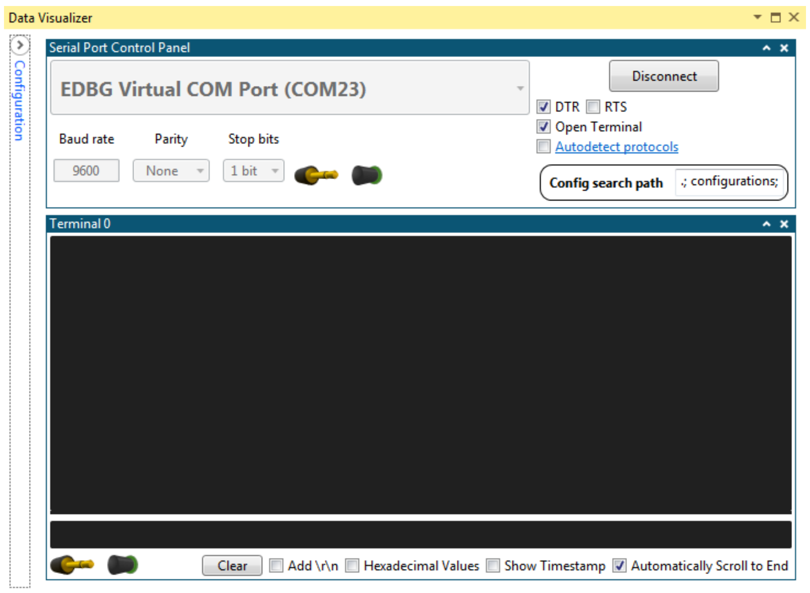


EDBG Virtual COM Port (COM23)

Baud rate: 9600
Parity: None
Stop bits: 1 bit
[✓] DTR [ ] RTS
[✓] Open Terminal
[ ] Autodetect protocols

Step 8: Confirm the baud, parity bit, and stop bits as shown above. Also confirm the DTR and Open Terminal check boxes are selected.

Step 9: Click on “Connect”. A new window will open named “Terminal”.



2.3.4 Testing the Data Visualizer serial connection

In these next steps you will add debug `printf`’s to the project application to test the serial communications between the Data Visualizer terminal window and the **SAM D21 Xplained Pro** is operating properly.

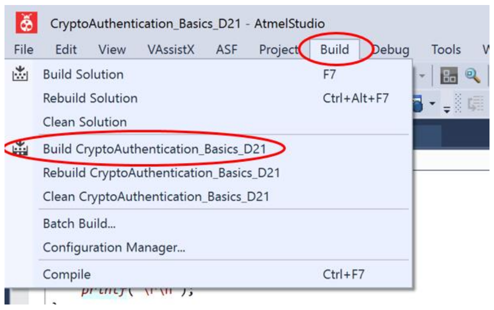
Step 1: Click on the `main_symmetric.c` tab.

Step 2: Use Ctrl+F to find comment: `//Step 1.1`

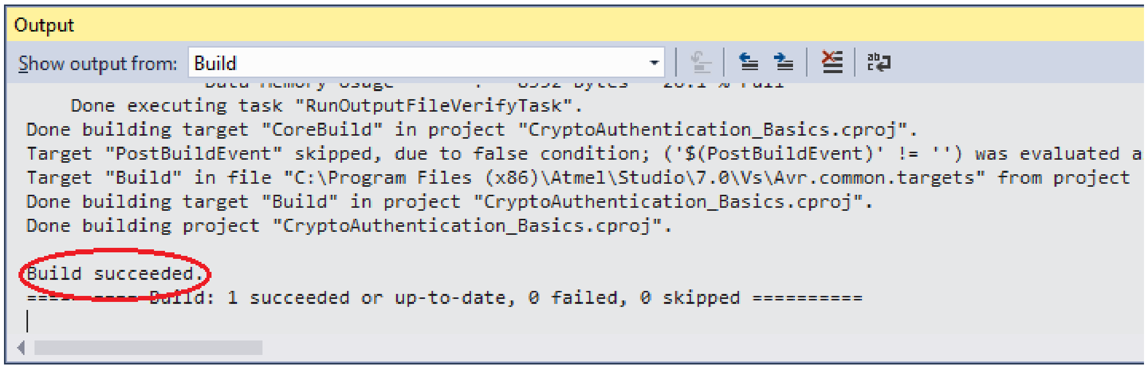
Step 3 Add the below code below the comment line:

```
//Step 1.1
printf("CryptoAuthLib Basics Disposable Symmetric Auth\n\r");
```

Step 4: In the Studio 7 menu bar, click on “Build”
then select “Build CryptoAuthentication_Basics_D21”

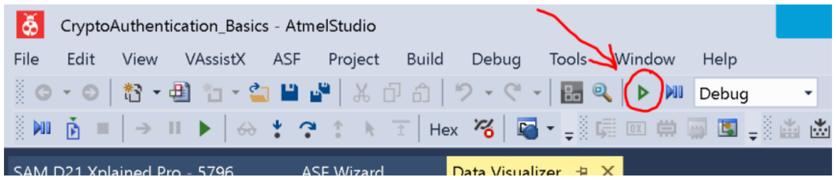

CryptoAuthentication_Basics_D21 - AtmelStudio
File
Edit
View
VAssistX
ASF
Project
Build
Debug
Tools
Build Solution F7
Rebuild Solution Ctrl+Alt+F7
Clean Solution
Build CryptoAuthentication_Basics_D21
Rebuild CryptoAuthentication_Basics_D21
Clean CryptoAuthentication_Basics_D21
Batch Build...
Configuration Manager...
Compile Ctrl+F7

Step 5: View the “Output” window and confirm that the build succeeded



```
Output
Show output from: Build
     Done executing task "RunOutputFileVerifyTask".
 Done building target "CoreBuild" in project "CryptoAuthentication_Basics.cproj".
 Target "PostBuildEvent" skipped, due to false condition; ('$(PostBuildEvent)' != '') was evaluated a
 Target "Build" in file "C:\Program Files (x86)\Atmel\Studio\7.0\Vs\Avr.common.targets" from project
 Done building target "Build" in project "CryptoAuthentication_Basics.cproj".
 Done building project "CryptoAuthentication_Basics.cproj".

Build succeeded.
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

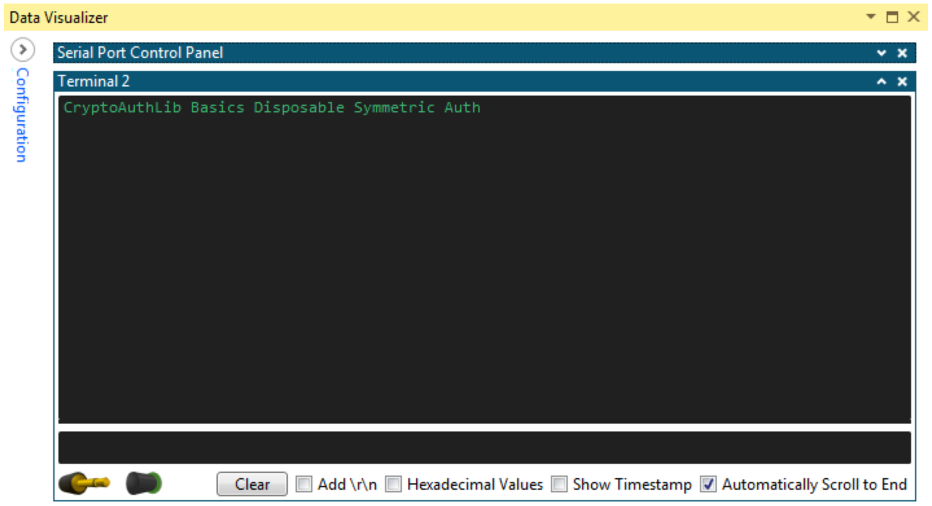
Step 6: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.



Step 7: Click on the Data Visualizer terminal window

Step 8: Press the board RESET button on the **SAM D21 Xplained** Pro board.

Step 9: Verify the text inside the printf is displayed in the terminal



```
CryptoAuthLib Basics Disposable Symmetric Auth
```

3 LAB 1 –Symmetric Authentication of a Remote Device

3.1 Overview

Symmetric authentication is a challenge/response process where a host device challenges a remote device, such as an accessory (for example a battery) or consumable device (printer cartridge), to assure that it is authentic and can be trusted. The challenged device is to respond with expected results.

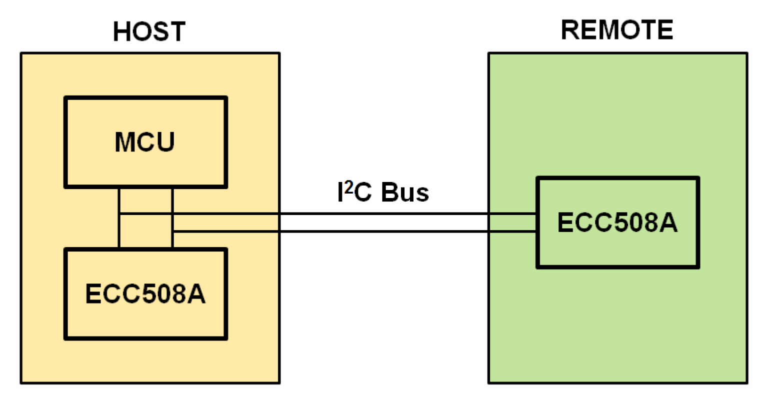
This method requires that both the host and the remote devices ***share the same key.*** Additionally the remote device can send a ***unique serial number*** to make its responses unique from other remote devices. *The problem that faces the embedded developer is how to securely embed these secrets into the host and the remote devices.*

The solution is to use a secure hardware key storage device to contain the shared key and unique serial number in the host and remote device. For this lab we will be using the ECC508A.

In this lab you will emulate a system that contains:

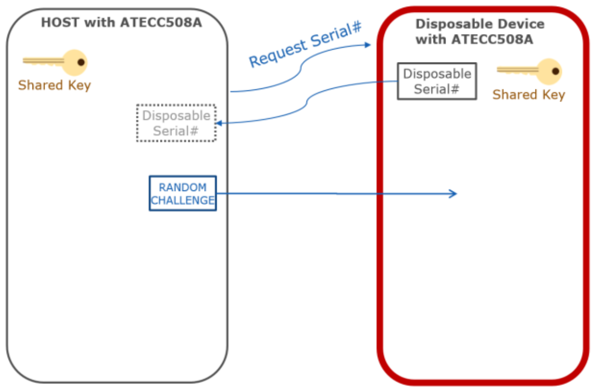
- **Host device** that contains a microcontroller and a secure hardware crypto authorization device, the ECC508A.
- **Remote device** that contains only a secure hardware crypto authorization device, the ECC508A (no microcontroller).

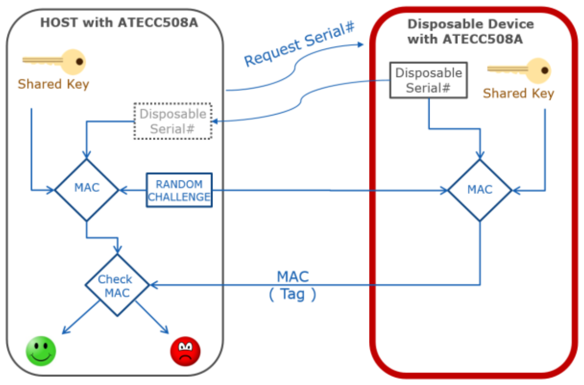
Both the host and the remote device's ECC508A have been preconfigured for you with a **shared secret key** and all ECC508 contain a unique serial number.



3.2 Authentication Process

Authentication begins with the host asking for the remote device's unique serial number. Next the host sends a random number which the host expects the remote device to hash with the shared secret. This is called a challenge because it challenges the remote device to provide a correct answer:





The remote device hashes the random number with its ***shared key*** and its ***unique serial number*** and sends back to the host the resulting output of the hash which in this context is called a Message Authentication Code (MAC).

The host checks the return MAC by repeating the same operation. It hashes its ***shared key*** with the random number and the remote device's unique serial number. The host compares the two results.

If the results match, then the challenge has been successfully responded to by the remote device and the host can trust the external device.

3.3 Lab Setup

3.3.1 Cryptograph Authentication Project

Follow the steps outlined in Section 2.3 to set up the Crypto Authentication Project

3.3.2 Hardware Requirements

For this lab you will need:

1. (1) SAM D21 Xplained Pro evaluation board
2. (2) CryptoAuth Xplained Pro evaluation boards
3. (1) USB Cable (Type A to Micro B)

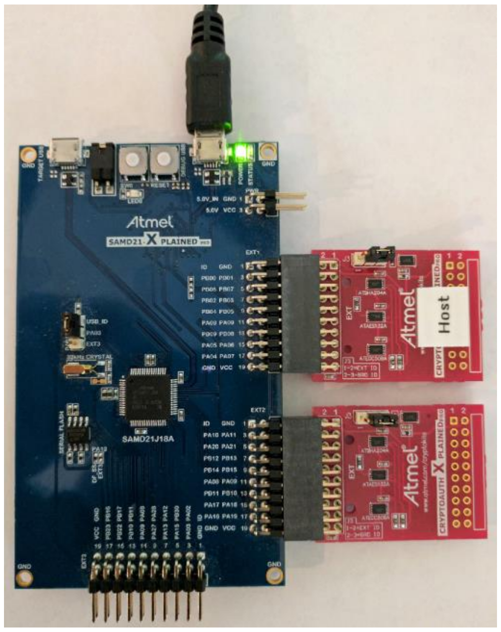
3.3.3 Plug in CryptoAuth Xplained Pro boards

Step 1: Unplug all boards and USB cables from any previous setup

Step 2: Insert the CryptoAuth Xplained Pro board labeled "HOST" to the SAM D21 Xplained Pro EXT1 port.

Step 3: Insert the CryptoAuth Xplained Pro board labeled "REMOTE" to the SAM D21 Xplained Pro EXT2 port.

Step 4: Connect a USB cable to the DEBUG USB port and PC.



3.4 Creating Application Software

Now we begin the task of creating the application software.

An **applications template** is provided. From this template you will create code segments for the lab. The completed application can be referenced for developing your own projects.

You will be introduced to a number of Application Programming Interface (API) commands that are part of the **CryptoAuthLib** library. The library makes developing code for the hardware crypto authorization devices much easier by providing a software framework around the register programming details of the devices.

In this lab section you will be adding and modifying a series of code blocks that will:

- Configure I^2C serial communications to the HOST ECC508A
- Initialize I^2C communications to the HOST ECC508A
- Read the unique serial number from the HOST ECC508A
- Configure I^2C serial communications to the REMOTE ECC508A
- Initialize I^2C communications to the REMOTE ECC508A
- Read the unique serial number from the REMOTE ECC508A

You are encouraged to add your own comments to the code to aid in your learning

3.4.1 Configure HOST ECC508A Communication Interface

In this step you will write code to configure I^2C communications to the HOST ECC508A CryptoAuth Xplained Pro board. It has been pre-configured specifically for this lab with the I^2C address of `0xC2`.

The CryptoAuthLib software running on the HOST microcontroller needs to be initialized with some information about the HOST ECC508 in order to be able to communicate with it. The CryptoAuthLib `atcab_init()` performs this function and can be called repeatedly to change between different devices' I2C addresses that will be used by the library API. The `atcab_init()` function takes a configuration structure as its arguments.

The code below initializes the structure in `main_symmetric.c` that will be passed to the `atcab_init()` function.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code (Ctrl+F) to find the comment: `//Step 1.2`

Step 3: Add the following code below the comment as shown:

```
//Step 1.2
ATCAIfaceCfg cfg_ateccx08a_i2c_host = {
     .iface_type                  = ATCA_I2C_IFACE,
     .devtype                     = ATECC508A,
     .atcai2c.slave_address       = 0xC2,
     .atcai2c.bus                 = 2,
     .atcai2c.baud                = 100000,
     .wake_delay                  = 1500,
     .rx_retries                  = 20
     };
```

3.4.2 Wait for SW0 Button Press

Next you will create a code segment that will wait for push button SW0 to be pressed (located next to the RESET button near the USB connectors). This action will be used to initialize the ECC508 and start the authentication process. Notice the `atcab_init()` references the configuration structure added in the previous step. All library calls after this will now use the I2C address set by `atcab_init()` to talk to the ECC508A.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.3`

Step 3: Add the following code:

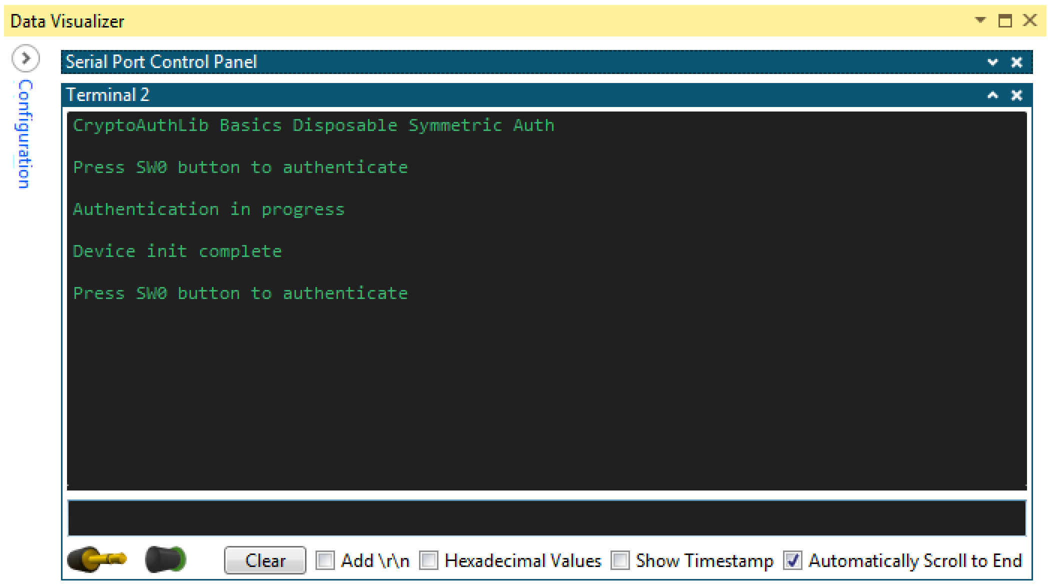
```
//Step 1.3
printf("Press SW0 button to authenticate\n\r");
while(port_pin_get_input_level(BUTTON_0_PIN) == SW0_INACTIVE);
printf("Authentication in progress\n\r");

volatile ATCA_STATUS status;

status = atcab_init( &cfg_ateccx08a_i2c_host );
CHECK_STATUS(status);


printf("Device init complete\n\r");
```

3.4.3 Test Compile Number 1

Step 1: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 2: Click on the Data Visualizer terminal window.

Step 3: Press the SW0 push button and verify that the output likes like below:



```
CryptoAuthLib Basics Disposable Symmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Press SW0 button to authenticate
```

3.4.4 Read the HOST Device Unique Serial Number

We would like to familiarize you with the **CryptoAuthLib** library of APIs. For this code segment, we’d like for you to look up the API to *read device serial number*. The place to look is in the `atca_basic.h` file. You can find `atca_basic.h` within the project by selecting the **Solution Explorer** window and navigating to “src/atcal/basic” directory. Find the most appropriate function to read the serial number.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.4`

Step 3: Add the following code:

```
//Step 1.4
uint8_t serial_number[ATCA_SERIAL_NUM_SIZE];
status = atcab_read_?????????????((uint8_t*)&serial_number);

CHECK_STATUS(status);

printf("Serial Number of host\r\n");
print_bytes((uint8_t*)&serial_number, 9);
printf("\r\n");
```

Step 4: Replace “atcab_read_?????????????” with the complete function name you find to read the serial number.

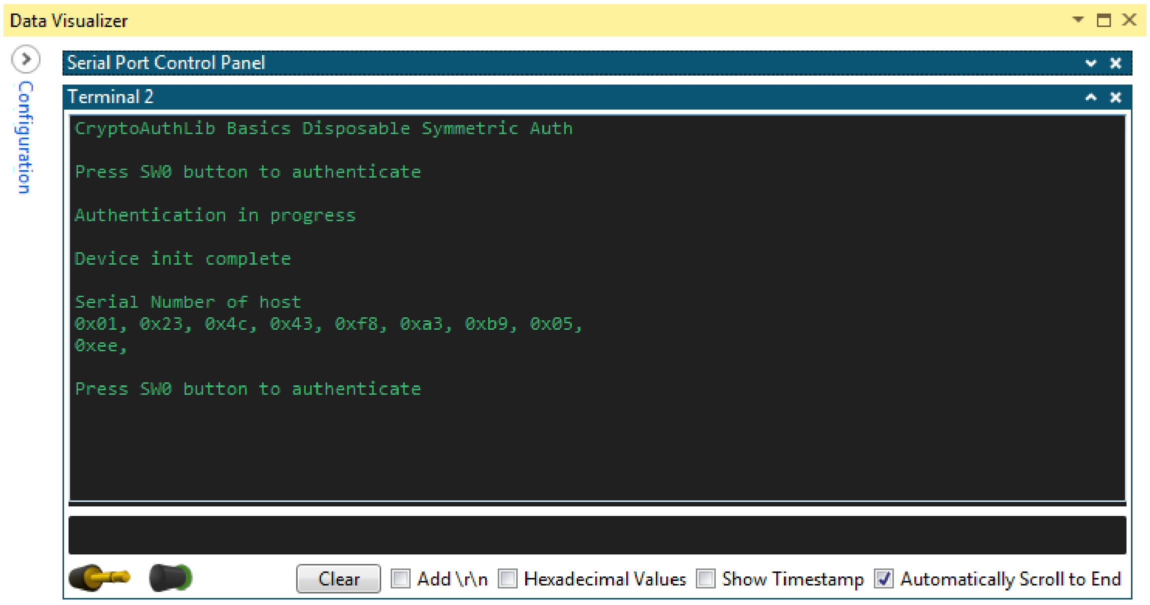
Hint: If you need help see Appendix A.1

3.4.5 Test Compile Number 2

Step 1: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 2: Click on the Data Visualizer terminal window.

Step 3: Press the SW0 push button and verify that the output looks like below:



```
CryptoAuthLib Basics Disposable Symmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Serial Number of host
0x01, 0x23, 0x4c, 0x43, 0xf8, 0xa3, 0xb9, 0x05,
0xee,

Press SW0 button to authenticate
```

The serial number will be in the format of:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
0x01	0x23	xx	xx	xx	xx	xx	xx	0xEE

Bytes 0, 1 and 8 are set by Microchip at the factory. These bytes will be the same for all standard devices. These bytes can change when Microchip does volume custom programming of devices for a customer. Bytes 2 – 7 will be unique for all Crypto Authentication devices.

3.4.6 Configure REMOTE ECC508A Communication Interface

Write a configuration structure to configure the REMOTE ECC508A. This code segment is very similar to the one you added for the HOST ECC508A. The ECC508A on the REMOTE CryptoAuth Xplained Pro board has been pre-configured specifically for this lab with the I^2C physical address of `0xC0`.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.5`

Step 3: Add the following code:

```
//Step 1.5
ATCAIfaceCfg cfg_ateccx08a_i2c_remote = {
      .iface_type = ATCA_I2C_IFACE,
      .devtype = ATECC508A,
      .atcai2c.slave_address = 0xC0,
      .atcai2c.bus = 2,
      .atcai2c.baud = 100000,
      .wake_delay = 1500,
      .rx_retries = 20
};
```

3.4.7 Read the REMOTE Device Unique Serial Number

Add a code segment that initializes reads the REMOTE ECC508A's unique serial number.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.6`

Step 3: Add the following code:

```
//Step 1.6
status = atcab_init( &cfg_ateccx08a_i2c_remote );
CHECK_STATUS(status);

status = atcab_read_serial_number((uint8_t*)&serial_number);
CHECK_STATUS(status);

printf("Serial Number of remote\r\n");
print_bytes((uint8_t*)&serial_number, 9);
printf("\r\n");
```

Note: The variable name where the <u>REMOTE device serial number</u> will be stored.

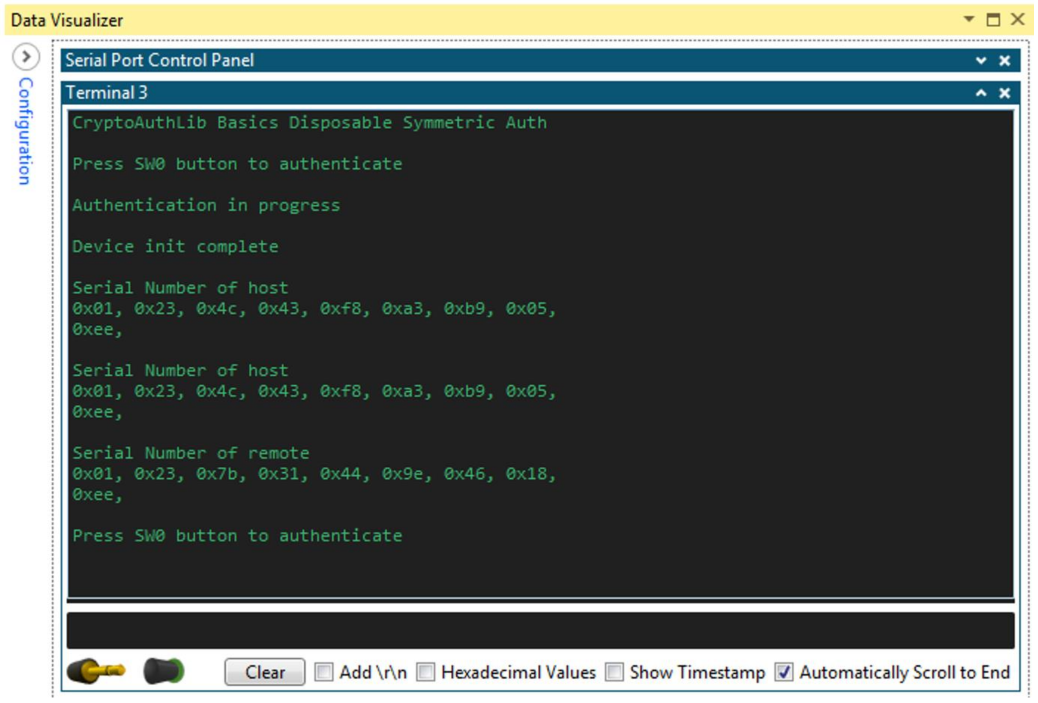
3.4.8 Test Compile Number 3

Step 1: On the Studio 7 menu bar, click on the "Start without debugging" button.

Step 2: Click on the Data Visualizer terminal window.

Step 3: Press the SW0 push button and verify that the output looks like below:

Verify that you can read the REMOTE crypto authentication device's unique serial number and that it is different from the HOST.



```
CryptoAuthLib Basics Disposable Symmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Serial Number of host
0x01, 0x23, 0x4c, 0x43, 0xf8, 0xa3, 0xb9, 0x05,
0xee,

Serial Number of host
0x01, 0x23, 0x4c, 0x43, 0xf8, 0xa3, 0xb9, 0x05,
0xee,

Serial Number of remote
0x01, 0x23, 0x7b, 0x31, 0x44, 0x9e, 0x46, 0x18,
0xee,

Press SW0 button to authenticate
```

3.4.9 Summary

What you have done so far is configure the microcontroller to communicate with the HOST and the REMOTE crypto authorization devices and created code to read the unique serial number from both.

In the next section you will begin adding code to perform the authentication of the REMOTE device

3.5 Authentication the REMOTE Device

Once I^{2}C serial communications to the HOST and REMOTE ECC508A's have been completed the process of authentication can start.

In this lab section you will:

Generating a NONCE on the HOST

Generating a Message Authentication Code (MAC) on the REMOTE using the NONCE.

Have the HOST check the MAC value returned by the REMOTE

3.5.1 Generating a NONCE

The next task is to construct a message that will allow our HOST to authenticate the REMOTE board. The first thing our HOST system must do is to generate a NONCE. The term NONCE means “number used once” and we will use a large random number for this purpose. A large random number has the property that it will, for all practical purposes, never occur again. The NONCE will be a one-time challenge used to authenticate the REMOTE board.

Note, if the challenge to the REMOTE board was repeated (not a NONCE) then the valid response (MAC) could be copied by an attacker watching the bus. The attacker could send this value to the host in the future if the attacker saw the HOST send the same challenge.

Within the **CryptoAuthLib** library, look in the `atca_basic.h` file to find the function to *generate a random number*. You can find `atca_basic.h` within the project by selecting the **Solution Explorer** window and navigating to “src/atcal/basic” directory. **Hint**, it is **NOT** one of the atcab_nonce..() functions. The nonce commands are input commands which load values into the ECC508A. In our case we want to read a random number from the ECC508A to be used by our code.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.7`

Step 3: Add the following code:

```
                //Step 1.7
                uint8_t nonce[32];

                status = atcab_??????((uint8_t*)&nonce);
                CHECK_STATUS(status);

                printf("Random from host\r\n");
                print_bytes((uint8_t*)&nonce, 32);
```

Step 4: Replace “atcab_??????” with the function name you find to create a random number.

Hint: See Appendix A.2 if you need help determining the correct function name.

Observation: Note the random number will be stored in the “nonce” variable.

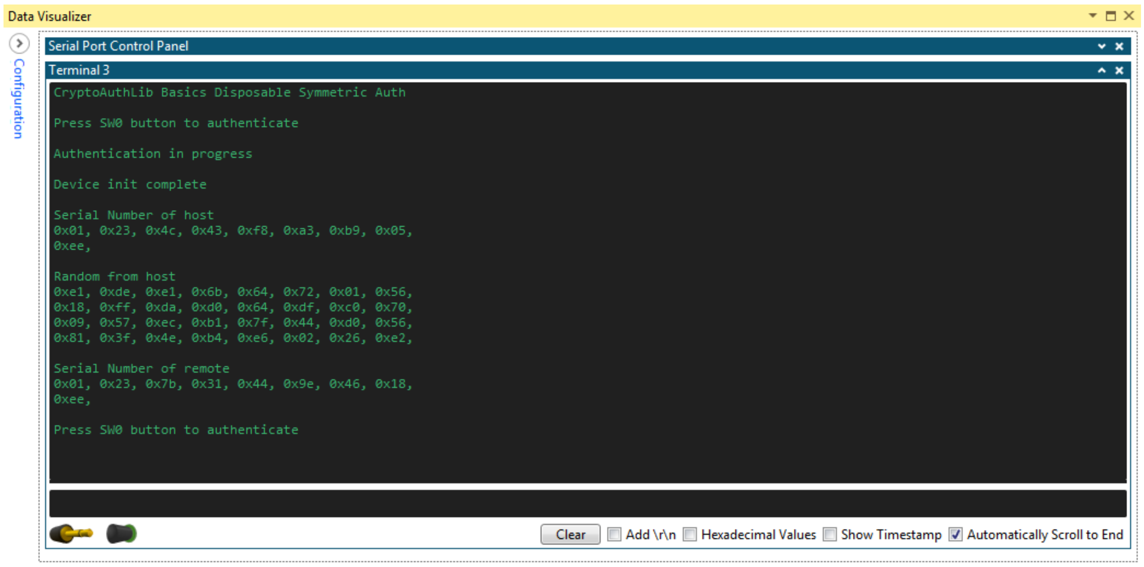
3.5.2 Test Compile Number 4

Step 1: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 2: Click on the Data Visualizer terminal window.

Step 3: Press the SW0 push button and verify that the output looks like below:

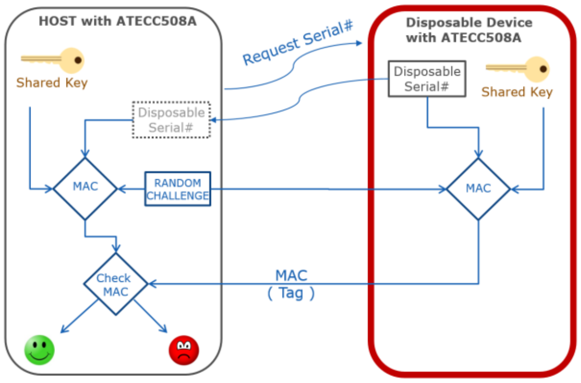
Verify that a random number is displayed each time SW0 is pressed.



```
CryptoAuthLib Basics Disposable Symmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Serial Number of host
0x01, 0x23, 0x4c, 0x43, 0xf8, 0xa3, 0xb9, 0x05,
0xee,

Random from host
0xe1, 0xde, 0xe1, 0x6b, 0x64, 0x72, 0x01, 0x56,
0x18, 0xff, 0xda, 0xd0, 0x64, 0xdf, 0xc0, 0x70,
0x09, 0x57, 0xec, 0xb1, 0x7f, 0x44, 0xd0, 0x56,
0x81, 0x3f, 0x4e, 0xb4, 0xe6, 0x02, 0x26, 0xe2,

Serial Number of remote
0x01, 0x23, 0x7b, 0x31, 0x44, 0x9e, 0x46, 0x18,
0xee,

Press SW0 button to authenticate
```

3.5.3 Generating a REMOTE Message Authentication Code (MAC)

Now that a large random number has been generated it can be sent to the REMOTE ECC508A as a challenge. The serial number, random number (challenge) and the secret key (inside the ECC508A) will be part of a “message” which will be hashed with SHA-256 by the MAC command on the ECC508A. Each value added to the message adds specific meaning to the MAC as interpreted by the HOST. The unique serial number means the MAC received could only have been generated a one specific device. The random number makes every MAC different so the HOST knows the REMOTE’s response is fresh. The secret key allows the REMOTE device’s MAC to prove the REMOTE device knows the secret key.



Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.8`

Step 3: Add the following code:

```
        //Step 1.8
        uint8_t mac[32];
        uint8_t slot = 0;
        uint8_t mode = (1<<6); // include serial number

        status = atcab_???(mode, slot, (const uint8_t*)&nonce, (uint8_t*)&mac);
        CHECK_STATUS(status);

        printf("MAC from remote\r\n");print_bytes((uint8_t*)&mac, 32);
```

Hint: See Appendix A.3 if you need help determining the correct function name.

Note: The variable “mac” stores the output of the command.

Here is some further information on the command you found in `atca_basic.h`.

```
ATCA_STATUS atcab_???( uint8_t mode,
                       uint16_t key_id,
                       const uint8_t* challenge,
                       uint8_t* digest );
```

The first argument, `mode,` is an input parameter that controls which fields within the device are used in the message that is internally hashed. We refer to the ECC508A data sheet **Table 9-34 Mode Encoding**:

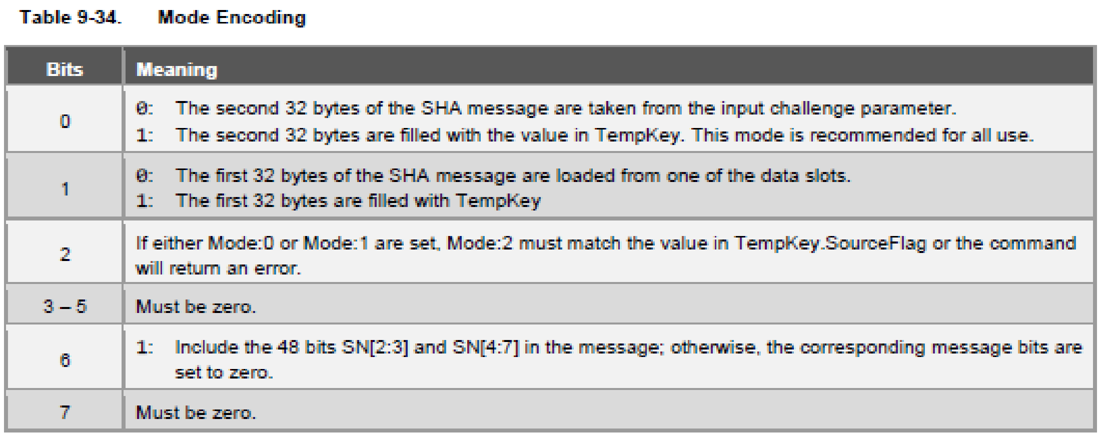


Table 9-34. Mode Encoding

Bits	Meaning
0	0: The second 32 bytes of the SHA message are taken from the input challenge parameter. 1: The second 32 bytes are filled with the value in TempKey. This mode is recommended for all use.
1	0: The first 32 bytes of the SHA message are loaded from one of the data slots. 1: The first 32 bytes are filled with TempKey
2	If either Mode:0 or Mode:1 are set, Mode:2 must match the value in TempKey.SourceFlag or the command will return an error.
3 – 5	Must be zero.
6	1: Include the 48 bits SN[2:3] and SN[4:7] in the message; otherwise, the corresponding message bits are set to zero.
7	Must be zero.

From **Table 9-34** we select the “mode” value of 0x40. This value includes the REMOTE device’s serial number in the message. The serial number gives each device a unique identity. For example, if the REMOTE device was a digital lottery ticket, its serial number along with the secret key would prove the digital ticket as a valid ticket and also either a winning or losing ticket based on its serial number.

The second argument `key_id` is an input parameter. Here we are telling the REMOTE ECC508A to use its internal protected shared key stored in slot 0. A key was preprogrammed to that slot specifically for this lab.

The third argument is `challenge`. It is an input parameter. A challenge can be static, select from a set, or unique. Having a unique challenge provides the best security. We will use the `nonce[ ]` array. This array contains the random number read by the HOST ECC508A. The term NONCE is use for the naming convention to indicate to the programmer what the variable's property. NONCE means number used once.

The forth argument is `digest`. This is the output parameter which will contain the Message Authentication Code (MAC). We provide a pointer to a 32 byte array called `mac[ ]` to store this value.

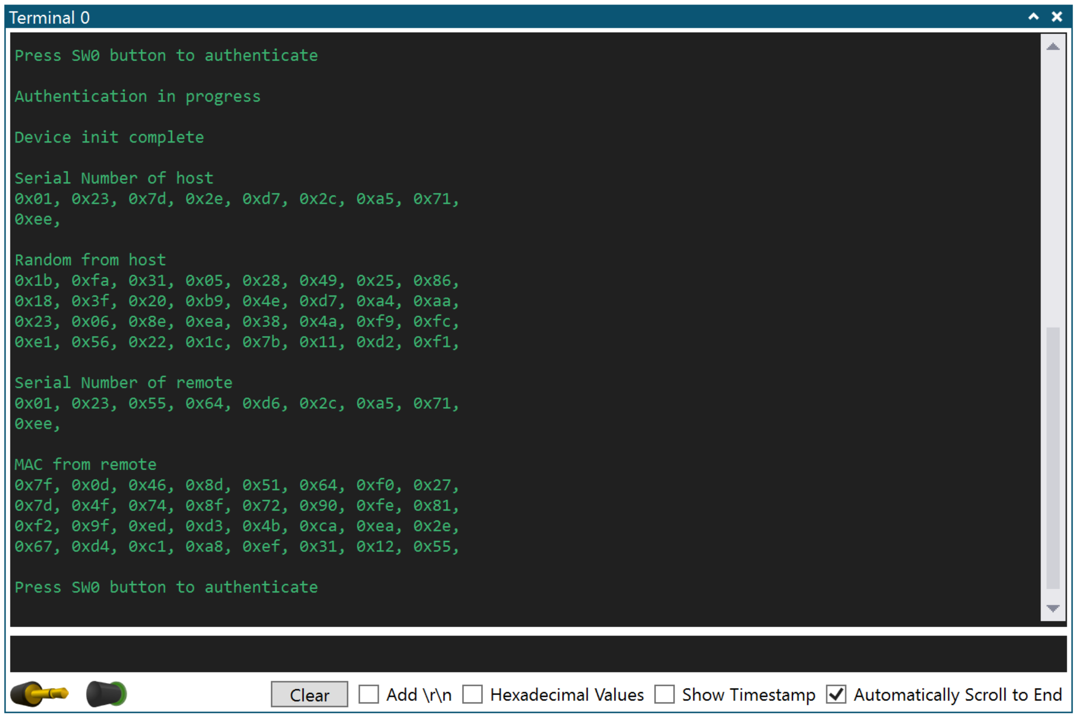
3.5.4 Test Compile Number 5

Step 1: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Step 2: Click on the Data Visualizer terminal window.

Step 3: Press the SW0 push button and verify that the output looks like below:

Verify that the REMOTE device's MAC value is displayed when you press SW0. Press SW0 repeatedly and verify that the MAC value changes. Why does the MAC value change? Think back about the input parameters to the MAC.



```
Press SW0 button to authenticate

Authentication in progress

Device init complete

Serial Number of host
0x01, 0x23, 0x7d, 0x2e, 0xd7, 0x2c, 0xa5, 0x71,
0xee,

Random from host
0x1b, 0xfa, 0x31, 0x05, 0x28, 0x49, 0x25, 0x86,
0x18, 0x3f, 0x20, 0xb9, 0x4e, 0xd7, 0xa4, 0xaa,
0x23, 0x06, 0x8e, 0xea, 0x38, 0x4a, 0xf9, 0xfc,
0xe1, 0x56, 0x22, 0x1c, 0x7b, 0x11, 0xd2, 0xf1,

Serial Number of remote
0x01, 0x23, 0x55, 0x64, 0xd6, 0x2c, 0xa5, 0x71,
0xee,

MAC from remote
0x7f, 0x0d, 0x46, 0x8d, 0x51, 0x64, 0xf0, 0x27,
0x7d, 0x4f, 0x74, 0x8f, 0x72, 0x90, 0xfe, 0x81,
0xf2, 0x9f, 0xed, 0xd3, 0x4b, 0xca, 0xea, 0x2e,
0x67, 0xd4, 0xc1, 0xa8, 0xef, 0x31, 0x12, 0x55,

Press SW0 button to authenticate
```

3.5.5 HOST checks the MAC value returned by the REMOTE

Now that the challenge has been sent to the REMOTE device and the MAC has been returned it is time for the HOST to authenticate the MAC value.

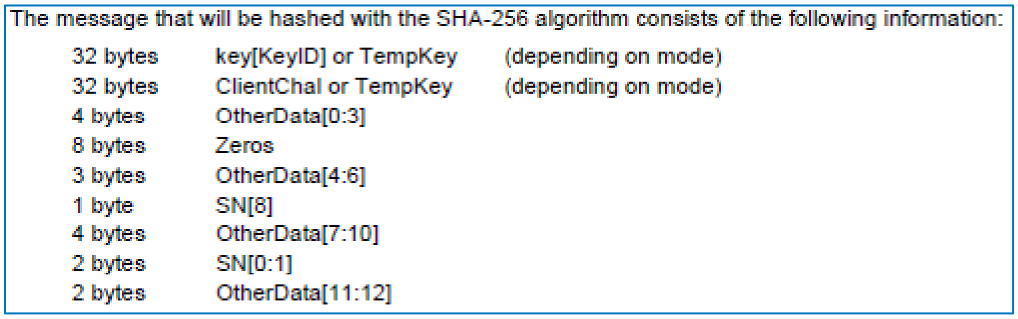
The HOST system now needs to test that the returned MAC value was correctly calculated. To do this we need the HOST authentication chip to perform the same calculation as done on the REMOTE ECC508A and compare the results. You might be tempted to use the `atcab_mac()` function that was used by the REMOTE to respond to the challenge and then compare the results. However, the MAC command using mode 6 (as we did) includes the serial number of the device it is executed on as part of the MAC calculation. The MAC command run on the HOST will include the HOST's serial number. So it will not match the MAC command run on the REMOTE because the HOST and REMOTE have different serial numbers.

So to make the comparison we will use a special command called CheckMac which allows a chip to check the output MAC value from another chip but, does not reveal the output to the host. If the MAC value was revealed then any ECC508A could pretend to be any other because the serial number is an input field in the MAC command (think back about the digital lottery ticket example on what the effect would be). The CheckMac command causes the ECC508A to check if the supplied MAC response from another chip matches the host's CheckMac calculated MAC. CheckMac just returns success or fail.

The function we will use is `atcab_checkmac()`. From `atca_basic.h` we see the function has the following arguments:

```
ATCA_STATUS atcab_checkmac( uint8_t mode,
                           uint16_t key_id,
                           const uint8_t *challenge,
                           const uint8_t *response,
                           const uint8_t *other_data);
```

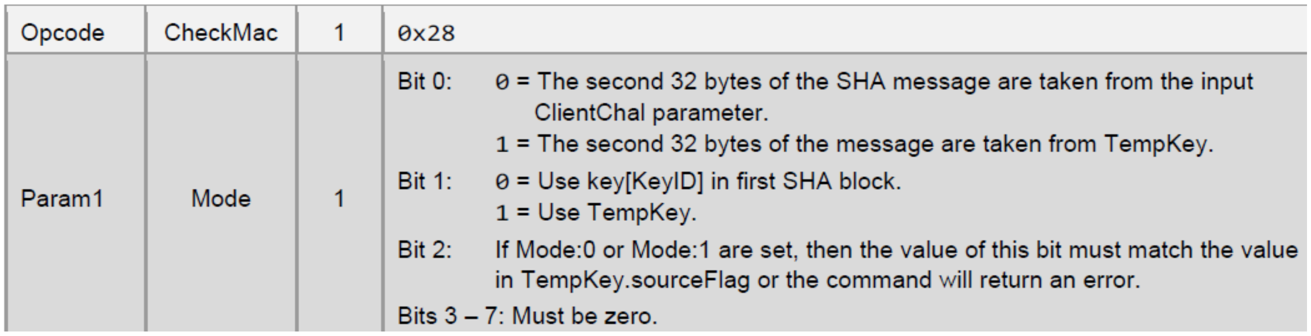
The first thing to understand is that the CheckMac command is internally filling a message buffer inside the ECC508A. The message buffer is the input to the internal SHA-256 hash engine. If you know all the values in this message you could run SHA-256 on it using anything, even a web SHA-256 calculator, and get the same value as the ECC508A.



The message that will be hashed with the SHA-256 algorithm consists of the following information:

32 bytes	key[KeyID] or TempKey	(depending on mode)
32 bytes	ClientChal or TempKey	(depending on mode)
4 bytes	OtherData[0:3]	
8 bytes	Zeros	
3 bytes	OtherData[4:6]	
1 byte	SN[8]	
4 bytes	OtherData[7:10]	
2 bytes	SN[0:1]	
2 bytes	OtherData[11:12]	

The first argument is `mode`. It is an input parameter that controls which fields within the device are used in the message. We refer to the ECC508A data sheet **Table 9-10 Input parameters**:



Opcode	CheckMac	1	`0x28`
Param1	Mode	1	Bit 0: `0` = The second 32 bytes of the SHA message are taken from the input ClientChal parameter. `1` = The second 32 bytes of the message are taken from TempKey. Bit 1: `0` = Use key[KeyID] in first SHA block. `1` = Use TempKey. Bit 2: If Mode:0 or Mode:1 are set, then the value of this bit must match the value in TempKey.sourceFlag or the command will return an error. Bits 3 – 7: Must be zero.

For the `atcab_checkmac()` function we will set the mode to `0x00`. Refer back to the message structure above and notice how the input parameter ClientChal is included in the message buffer.

The second argument is `key_id`. It is an input parameter. This tells the HOST ECC508A to use the shared key stored in device memory location slot 0 which was preprogrammed specifically for this lab.

The third argument is `challenge`. It is an input parameter. This corresponds to “ClientChal” value in the mode description. This is the random number we generated and sent to the REMOTE as a challenge in the array `nonce[ ]`.

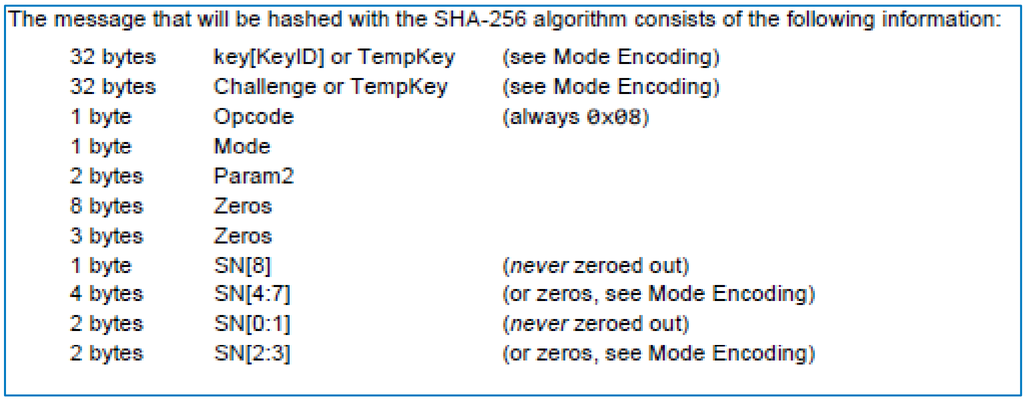
The forth argument is `response`. It is an input parameter. This is the MAC value we received from the REMOTE which is the result from the SHA-256 hash function run on the REMOTE.

Important

The fifth argument is `other_data`. It is an input parameter. Let us take a closer look at its format:

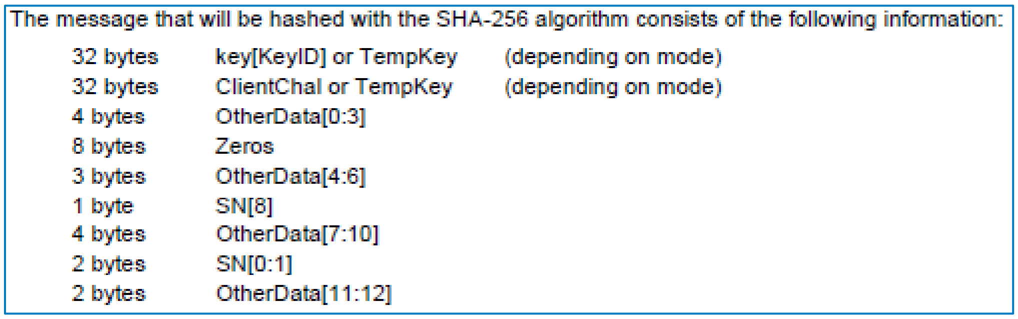
Referring to the ECC508A data sheet we see that `otherdata` size is 13 bytes.

Recall that we want the HOST ECC508A to reproduce the results of the SHA-256 hash function performed by the REMOTE ECC508A. In order to do this we need to make the HOST device’s CheckMac message buffer values match the REMOTE’s MAC message buffer values. Just picture the message buffers on each device as a byte array. <u>We need REMOTE message byte[x] to match HOST message byte[x] for all x bytes in the message</u>. To do this we refer to the ECC508A data sheet and examine the variable positions that were hashed using the `atcab_mac()` function on the REMOTE:


The message that will be hashed with the SHA-256 algorithm consists of the following information:

32 bytes	key[KeyID] or TempKey	(see Mode Encoding)
32 bytes	Challenge or TempKey	(see Mode Encoding)
1 byte	Opcode	(always 0x08)
1 byte	Mode	
2 bytes	Param2	
8 bytes	Zeros	
3 bytes	Zeros	
1 byte	SN[8]	(*never* zeroed out)
4 bytes	SN[4:7]	(or zeros, see Mode Encoding)
2 bytes	SN[0:1]	(*never* zeroed out)
2 bytes	SN[2:3]	(or zeros, see Mode Encoding)

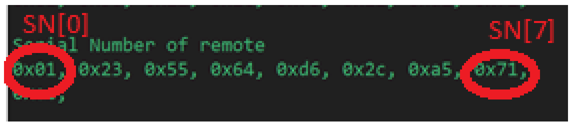
And we compare it to the variable positions that will be hashed using the `atcab_checkmac()` function on the HOST:


The message that will be hashed with the SHA-256 algorithm consists of the following information:

32 bytes	key[KeyID] or TempKey	(depending on mode)
32 bytes	ClientChal or TempKey	(depending on mode)
4 bytes	OtherData[0:3]	
8 bytes	Zeros	
3 bytes	OtherData[4:6]	
1 byte	SN[8]	
4 bytes	OtherData[7:10]	
2 bytes	SN[0:1]	
2 bytes	OtherData[11:12]	

Below is a table to help you line up the field names and bytes between the MAC and CheckMac command.

One of the values we need to add to the CheckMac message is the remote device serial number. Look back at your Data Visualizer windows where the remote serial number was printed. Notice the byte order of the device serial number printed in the terminal then look at the table below. In the table, SN represents the `serial_number` variable in our code that was printed to the terminal. Look at the last row as an example of how to map the data to the `otherdata[]` array. You can see that SN[2] needs to be loaded in to Otherdata[11] and SN[3] to Otherdata[12] to make the message buffers match. In the following exercise you will need to set all the bytes in `otherdata[]`.



```
Serial Number of remote
0x01, 0x23, 0x55, 0x64, 0xd6, 0x2c, 0xa5, 0x71,
```

bytes	MAC	CheckMac	Value
32	key[KeyID]	key[KeyID]	
32	Nonce	Nonce	
1	Opcode	Otherdata[0]	0x08
1	Mode	Otherdata[1]	0x40
2	Param2	Otherdata[2:3]	0x00,0x00
8	Zeros	Zeros	
3	Zeros	Otherdata[4:6]	0x00,0x00,0x00
1	SN[8]	SN[8]	
4	SN[4:7]	Otherdata[7:10]	0xd6,0x2c,0xa5,0x71
2	SN[0:1]	SN[0:1]	
2	SN[2:3]	Otherdata[11:12]	0x55,0x64

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 1.9`

Step 3: Add the following code:

```
        //Step 1.9
        status = atcab_init( &cfg_ateccx08a_i2c_host );
        uint8_t otherdata[CHECKMAC_OTHER_DATA_SIZE];
        memset(otherdata, 0x00, CHECKMAC_OTHER_DATA_SIZE);
        otherdata[0] = 0x08;
        otherdata[1] = ?????
        otherdata[7] = ?????
        otherdata[8] = ?????
        otherdata[9] = ?????
        otherdata[10] = ?????
        otherdata[11] = serial_number[2];
        otherdata[12] = serial_number[3];
        mode = 0;
        status = atcab_checkmac(mode,
                                slot,
                                (const uint8_t*)&nonce,
                                (const uint8_t*)&mac,
                                (const uint8_t*)&otherdata);
        if(status == ATCA_SUCCESS) {
             printf("Authenticated by host\r\n\r\n");
        } else {
             printf("Failed to authenticate\r\n\r\n");
        }
```

Step 4: Using the mapping table from above try and determine where the `serial_number[]` bytes and `mode` variable should be placed into the `otherdata` byte array locations so that the message will match the "MAC" command message buffer. Look at `otherdata[11]` (and 12) in the code and see if that helps you determine how to map the other locations.

Hint: See Appendix A.4 if you need help determining the correct mapping.

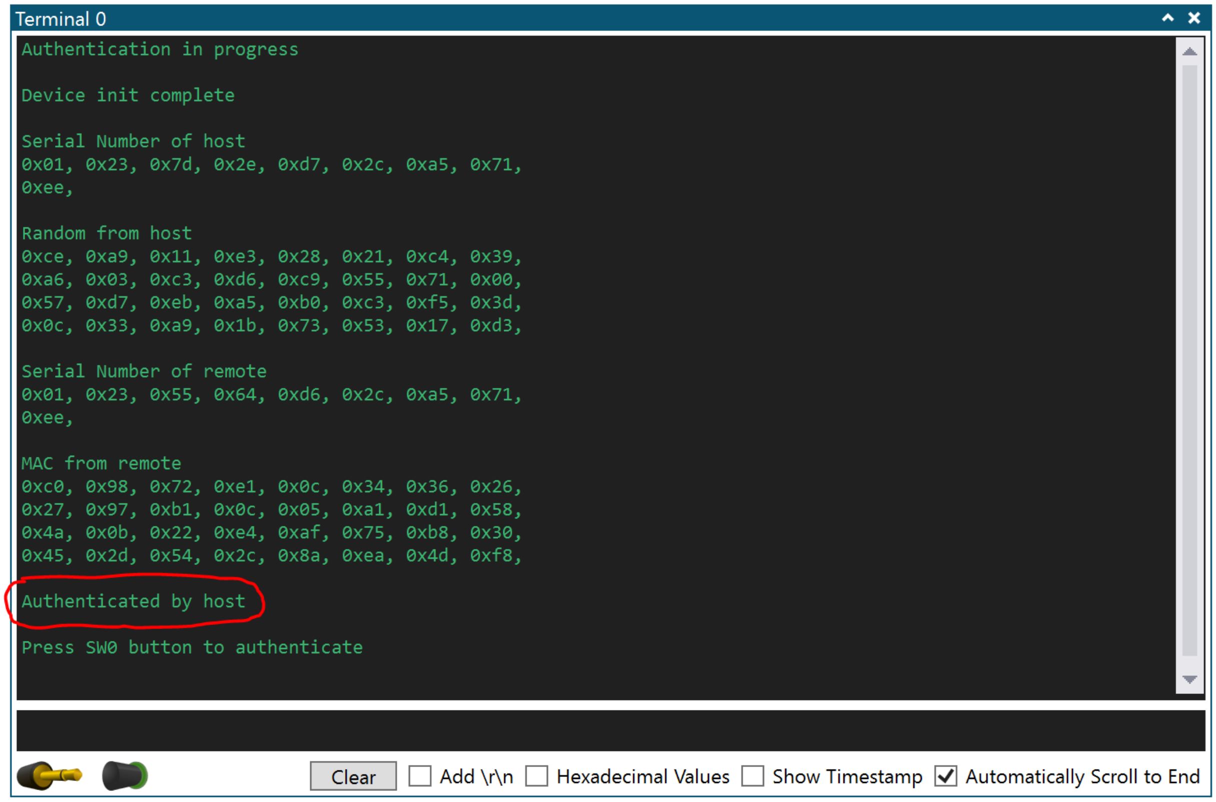
3.5.6 Test Compile Number 6

Step 1: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Step 2: Click on the Data Visualizer terminal window.

Step 3: Press the SW0 push button and verify that the output looks like below:

Verify the REMOTE MAC has been authenticated by the HOST microcontroller.



```
Authentication in progress

Device init complete

Serial Number of host
0x01, 0x23, 0x7d, 0x2e, 0xd7, 0x2c, 0xa5, 0x71,
0xee,

Random from host
0xce, 0xa9, 0x11, 0xe3, 0x28, 0x21, 0xc4, 0x39,
0xa6, 0x03, 0xc3, 0xd6, 0xc9, 0x55, 0x71, 0x00,
0x57, 0xd7, 0xeb, 0xa5, 0xb0, 0xc3, 0xf5, 0x3d,
0x0c, 0x33, 0xa9, 0x1b, 0x73, 0x53, 0x17, 0xd3,

Serial Number of remote
0x01, 0x23, 0x55, 0x64, 0xd6, 0x2c, 0xa5, 0x71,
0xee,

MAC from remote
0xc0, 0x98, 0x72, 0xe1, 0x0c, 0x34, 0x36, 0x26,
0x27, 0x97, 0xb1, 0x0c, 0x05, 0xa1, 0xd1, 0x58,
0x4a, 0x0b, 0x22, 0xe4, 0xaf, 0x75, 0xb8, 0x30,
0x45, 0x2d, 0x54, 0x2c, 0x8a, 0xea, 0x4d, 0xf8,

Authenticated by host

Press SW0 button to authenticate
```

3.5.7 Summary

Congratulations, you have completed an authorization of a remote device. Let's review what you performed:

- Initialized I^{2}C communications to the HOST ECC508A
- Initialized I^{2}C communications to the REMOTE ECC508A
- The HOST microcontroller requested the REMOTE's unique serial number
- The HOST ECC508A generated a large random number that you used as a challenge
- The challenge was used by the REMOTE ECC508A `atcab_mac()` to generate a MAC response
- The MAC response was verified by the HOST ECC508A using the `atcab_checkmac()` function

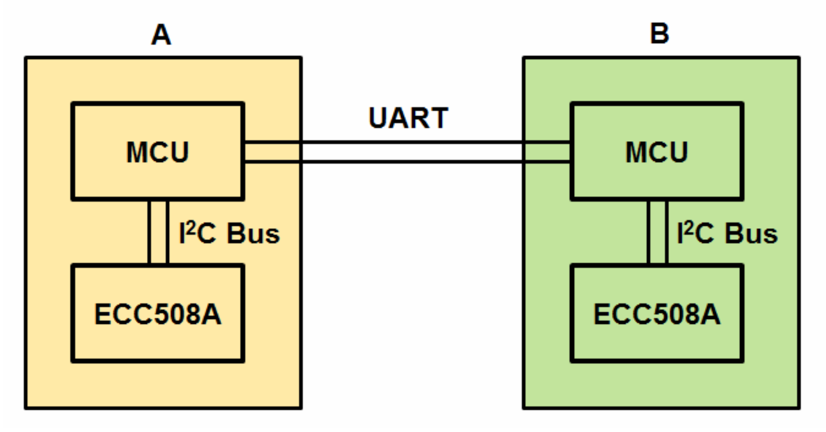
4 LAB 2 – Symmetric Authentication Between Two Devices

4.1 Overview

For this lab you will emulate a system that contains:

- **Device “A”** that contains a microcontroller and a secure hardware crypto authentication device, the ECC508A.
- **Device “B”** that contains a microcontroller and a secure hardware crypto authentication device, the ECC508A.

Both ECC508A devices have been preconfigured for you with a shared secret key specifically for this lab.



The individual microcontrollers will communicate over a UART in this example. However, the communication path can be any wired or wireless connection.

The code in this lab is structured so that the microcontrollers can communicate and authenticate in either direction: A authenticates B or B authenticates A.

The same code will be programmed into each device’s microcontroller.

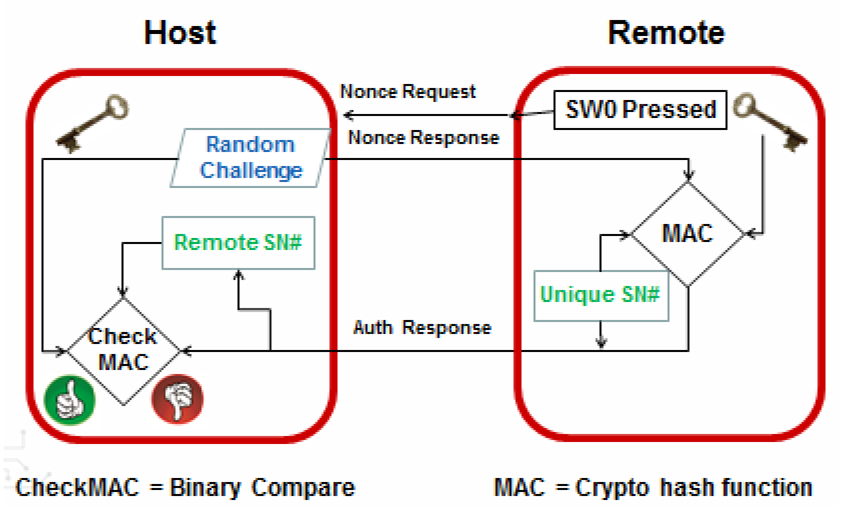
4.2 Authentication Process

Authentication between two MCUs over the communication link will precede nearly the same as it did using a single MCU with two ECC508A devices attached. However, now the serial number, challenge and MAC will need to be sent over the physical interface between the boards.

The code in this lab is structured so that the microcontrollers can communicate and authenticate in either direction: A authenticates B or B authenticates A.

The same code will be programmed into each board.

Each time you write a code segment and compile, you will need to update the program on both boards before they will be able to communicate.



4.3 Laboratory Setup

4.3.1 Cryptograph Authentication Project

Follow the steps outlined in Section 2.3 to set up the Crypto Authentication Project. If you have already completed at least Section 2 and 3 you can continue from where you left off for the software setup.

4.3.2 Hardware Requirements

For the lab you will need:

1. (2) SAM D21 Xplained Pro evaluation board
2. (2) CryptoAuth Xplained Pro evaluation boards
3. (2) USB Cable (Type A to Micro B)
4. (2) Jumper wires

4.3.3 Plug in CryptoAuth Xplained Pro boards

Step 1: Unplug all SAMD21Xplained Pro boards' USB cables and CryptoAuth Xplained Pro wing boards.

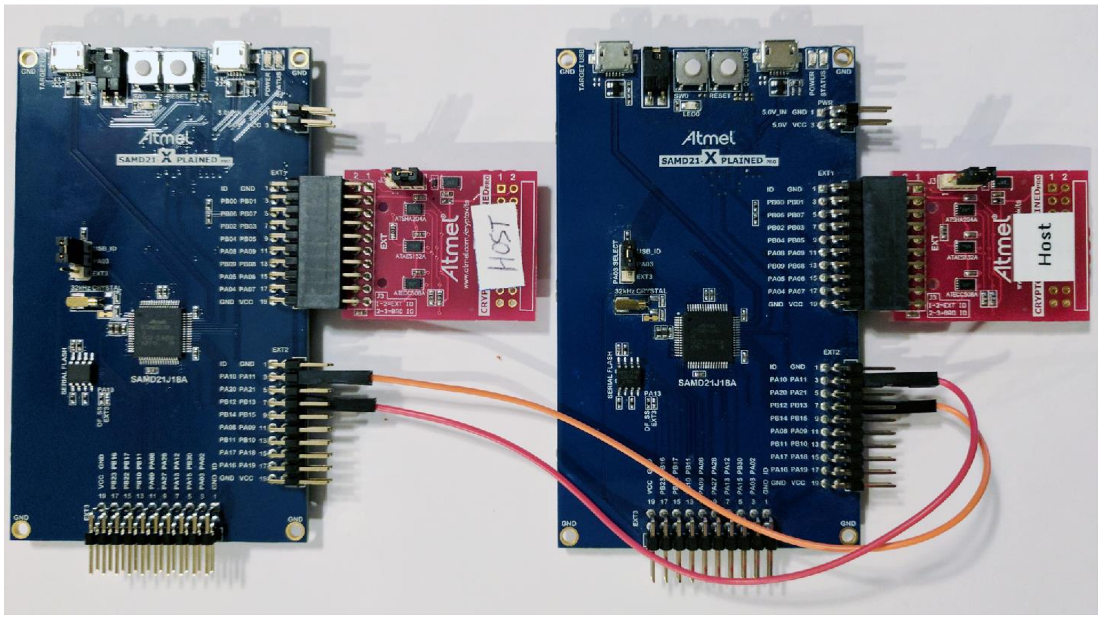
Step 2: Insert the CryptoAuth Xplained Pro board labeled "HOST" to the first SAM D21 Xplained Pro EXT1 port.

Step 3: Insert the CryptoAuth Xplained Pro board labeled "HOST" to the second SAM D21 Xplained Pro EXT1 port.

Step 4: Attach jumper wires as shown below.

Pin PB13 connects to PA10 on the opposite SAM D21 Xplained Pro board.
This connects UART TX to the corresponding UART RX on both boards.

Step 5: Connect a USB cable to each board's DEBUG USB port to the PC.



4.4 Writing Application Software

This project expects that you completed Lab 1 - Symmetric Authentication of a Remote Device in the previous section.

4.4.1 Project Preparation

You will add some setup code to this project to enable code for this second module. This code will send a nonce request message to the other board when SW0 is pressed (refer back to the Section 4.2 communication diagram).

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 2.1`

Step 3: Add the following code:

```
//Step 2.1
#define ENABLE_MODULE_2
```

Step 4: Search the source code to find the comment: `//Step 2.2`

Step 5: Add the following code:

```
//Step 2.2
symmetric_auth_comm();
```

Step 6: Comment out the `symmetric_auth()` function call as shown below

```
//symmetric_auth();
```

Step 7: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 8: Observe in the output window that the build completed successfully.

Observation: Note the serial number next to the icon on the Studio 7 menu bar:

4.4.2 Programming Two SAM D21 Xplained Pro boards

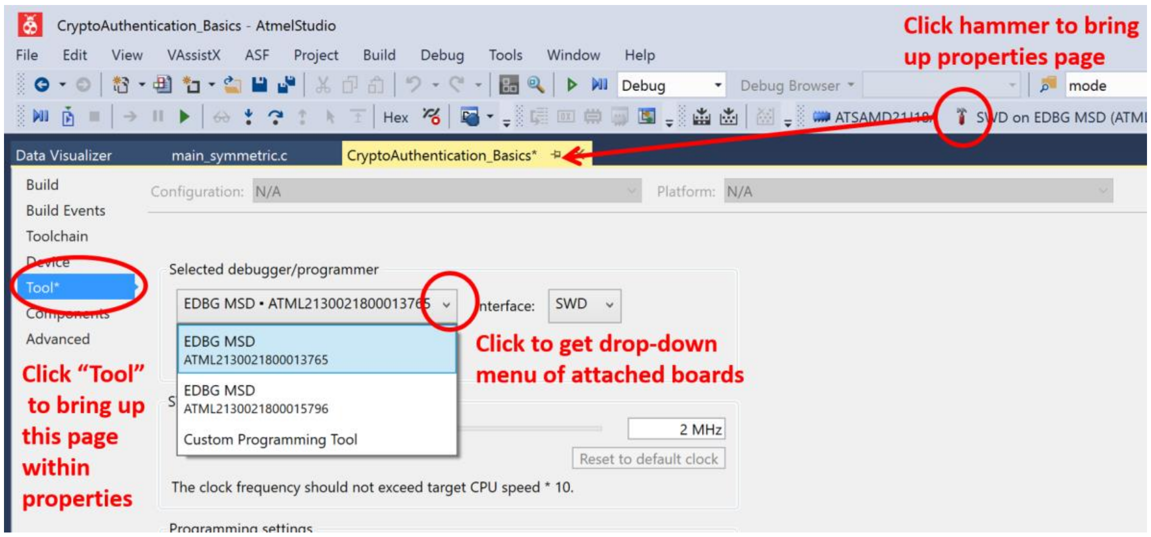
For this lab you have two SAM D21 Xplained Pro boards plugged into the same PC running Studio 7. Each time you write a code segment and compile, you will need to update the program on both boards before proceeding.

Here are the steps for selecting which SAM D21 Xplained Pro board will be programmed:

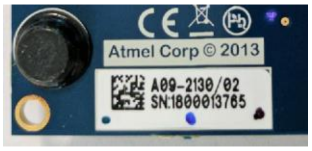
Step 1: On the Studio 7 menu bar, click on the button to bring up the properties page for the SAM D21 Xplained Pro boards.

Step 2: Click on the "Tool" menu in the side tabs.

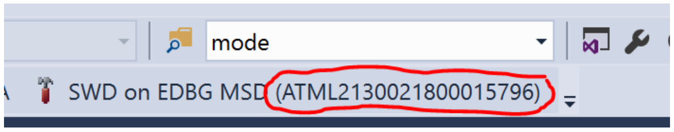
Step 3: Click on the drop-down icon under the "Selected debugger/programmer" label.


CryptoAuthentication_Basics - AtmelStudio
File Edit View VAssistX ASF Project Build Debug Tools Window Help
Click hammer to bring up properties page
Debug
Debug Browser
mode
Hex
ATSAMD21J18A
SWD on EDBG MSD (ATML
Data Visualizer
main_symmetric.c
CryptoAuthentication_Basics*
Build
Build Events
Toolchain
Device
Tool*
Components
Advanced
Click "Tool" to bring up this page within properties
Configuration: N/A
Platform: N/A
Selected debugger/programmer
EDBG MSD • ATML2130021800013765
Interface: SWD
Click to get drop-down menu of attached boards
EDBG MSD ATML2130021800013765
EDBG MSD ATML2130021800015796
Custom Programming Tool
2 MHz
Reset to default clock
The clock frequency should not exceed target CPU speed * 10.
Programming settings

The two SAM D21 Xplained Pro board's serial numbers will be displayed. You can find the serial number on a label on the bottom of the board.



The board serial number at the top of the drop-down box is the board that is "selected" and will be programmed when you click on the "Start without debugging" button. Note that the selected board's serial number is also displayed next to the icon on the menu bar.



When you press the “Start without debugging” button, you will program the board that is selected and displayed next to the icon.

4.4.3 Program Device B

Step 1: Select the “other” SAM D21 Xplained Pro board using the steps in Section 4.4.2.

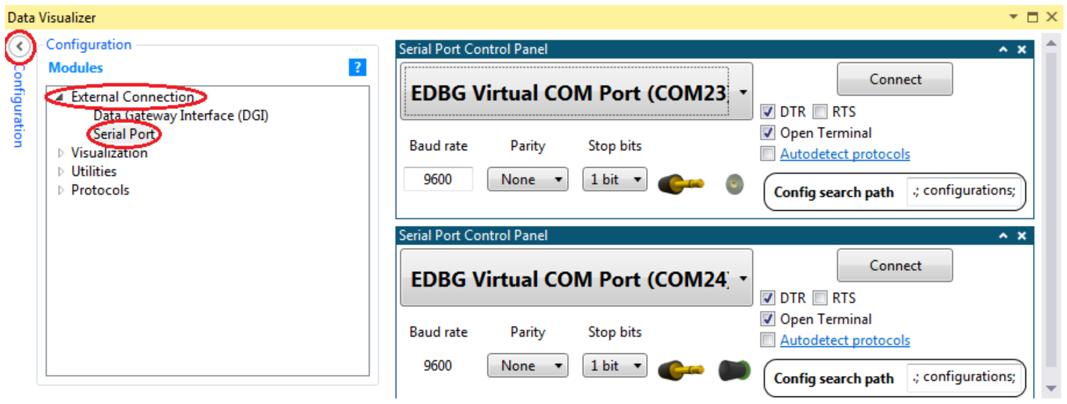
Step 2: On the Studio 7 menu bar, click on the “Start without debugging” button.

Observe: You programmed Device B. Note the board serial number:

____

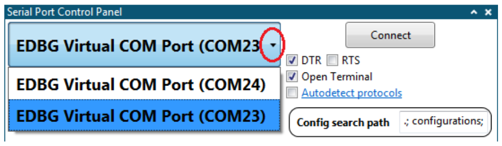
4.4.4 Create a Second Data Visualization Window

Step 1: Click on the Data Visualizer tab.



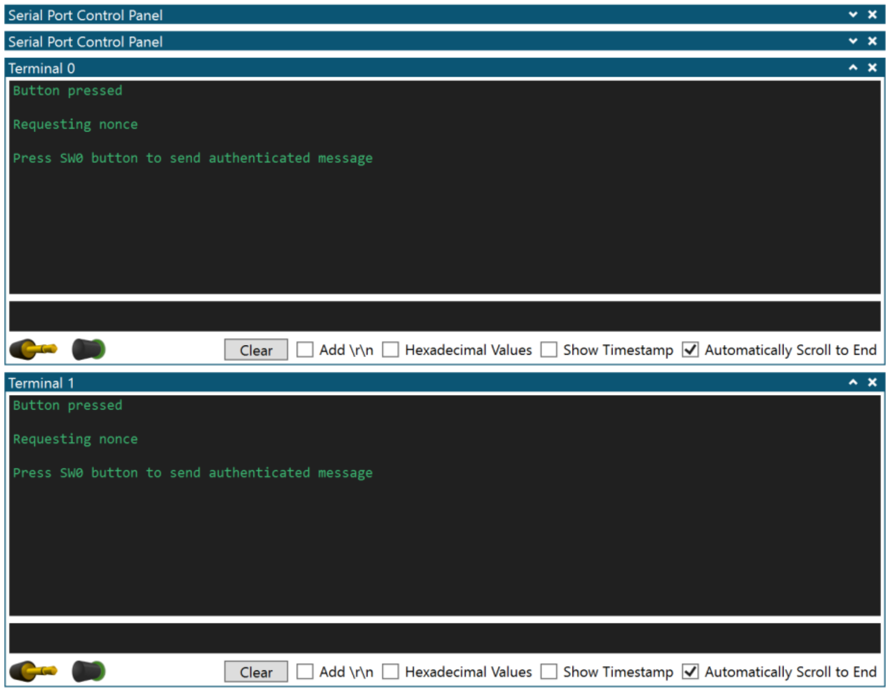
Step 2: Open a second serial port.

Step 3: Select the newly enumerated COM port of the second board.


Serial Port Control Panel
EDBG Virtual COM Port (COM23)
EDBG Virtual COM Port (COM24)
EDBG Virtual COM Port (COM23)
Connect
DTR
RTS
Open Terminal
Autodetect protocols
Config search path
.; configurations;

Step 4: Select Click on the or icons in the upper right corners to collapse and expand windows for better viewing.

Step 5: Press the RESET button on each board and note which terminal window belongs to each board.



```
Button pressed

Requesting nonce

Press SW0 button to send authenticated message
```

```
Button pressed

Requesting nonce

Press SW0 button to send authenticated message
```

Step 6: Confirm that both boards are programmed.

4.4.5 Configure local ECC508A Communication Information

In this step you will write code to configure I2C communications to the local ECC508A CryptoAuth Xplained Pro board. It has been pre-configured specifically for this lab with the I^{2}C address of `0xC2`. You will also add code to read the local ECC508A's unique serial number.

The code is very similar to the code you created in the previous lab section.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 2.3`

Step 3: Add the following code:

```
//Step 2.3
volatile ATCA_STATUS status;
//init for ECC508
status = atcab_init(&?);
CHECK_STATUS(status);
uint8_t * serial_number = (uint8_t *)&tx_packet->auth.serial_number;
//read local chip’s serial number
status = atcab_read_serial_number(?);
CHECK_STATUS(status);
printf("My Serial Number\r\n");
print_bytes(serial_number, 9);printf("\r\n");
```

Step 4: Add the correct configuration structure parameter to the `atcab_init()` function at "`?`".

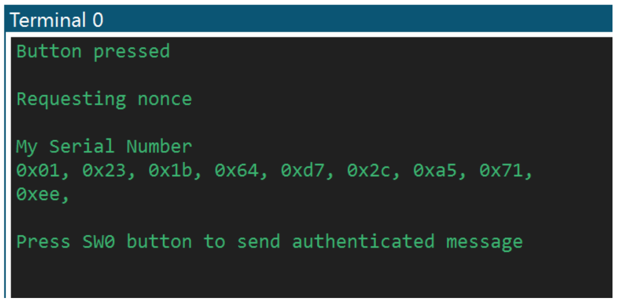
Note: The configuration structure you added in the previous section is still valid.

Step 5: Add the beset storage parameter to the `atcab_read_serial_number()` function at "`?`". Look through the code and find the best variable to save the serial number to.

Hint: If you need help see Appendix A.5

Step 5: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Step 6: Verify the serial number is printed to the terminal.



```
Button pressed

Requesting nonce

My Serial Number
0x01, 0x23, 0x1b, 0x64, 0xd7, 0x2c, 0xa5, 0x71,
0xee,

Press SW0 button to send authenticated message
```

4.4.6 MAC command on local ECC508A

In this section you will add a code segment that will calculate the MAC value on the ECC508A. This code will send the MAC and serial number as the Auth Response message when a Nonce Response message (refer back to the Section 4.2 communication diagram).

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 2.4`

Step 3: Add the following code:

```
        //Step 2.4
        //mac function including mode slot nonce and output to mac variable

        status = atcab_mac(?, ?, ?, ?);

        CHECK_STATUS(status);
        printf("Calculated MAC\r\n");
        print_bytes(mac, 32);

        printf("Sending message, serial number and MAC\n\r");

        tx_packet->type = MSG_PACKET_TYPE_AUTH;

        usart_write_buffer_job(module, (uint8_t*)tx_packet, sizeof(msg_packet_t));
```

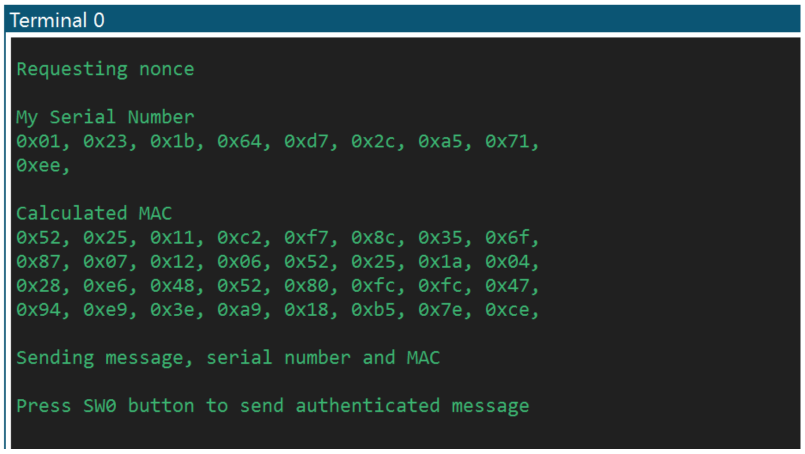
Step 4: Add the correct parameters to the `atcab_mac()` function at "`?`". Use the variables "`slot`", "`mode`", "`mac`", and "`nonce`" in your function call.

Hint: Look at the `symmetric_auth()` function in the previous section. Note the variable types are slightly different from the previous section.

Hint: If you need more help see Appendix A.6

Step 5: On the Studio 7 menu bar, click on the "Start without debugging" ▷ button.

Step 6: Verify the MAC value is printed to the Terminal screen.

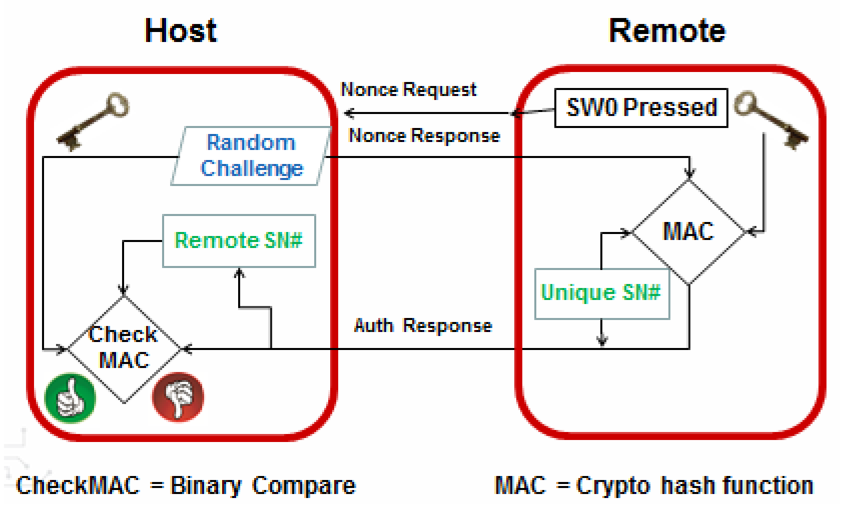


```
Requesting nonce

My Serial Number
0x01, 0x23, 0x1b, 0x64, 0xd7, 0x2c, 0xa5, 0x71,
0xee,

Calculated MAC
0x52, 0x25, 0x11, 0xc2, 0xf7, 0x8c, 0x35, 0x6f,
0x87, 0x07, 0x12, 0x06, 0x52, 0x25, 0x1a, 0x04,
0x28, 0xe6, 0x48, 0x52, 0x80, 0xfc, 0xfc, 0x47,
0x94, 0xe9, 0x3e, 0xa9, 0x18, 0xb5, 0x7e, 0xce,

Sending message, serial number and MAC

Press SW0 button to send authenticated message
```

4.4.7 Generating a NONCE (Random Number)

Press the SW0 button repeatedly. Notice the MAC value is not changing. This is because we have not yet added code to update the nonce value used by the MAC function. This means the MAC is being calculated with a fixed input and gives a fixed output.

Next you will add code that will respond to a message Nonce Request (generated by a SW0 press).



Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 2.5`

Step 3: Add the following code:

```
//Step 2.5
volatile ATCA_STATUS status;
status = atcab_init( &cfg_ateccx08a_i2c_host );
CHECK_STATUS(status);

uint8_t * nonce = &tx_packet->nonce.value;

//code to generate random number to be use as a nonce
status = atcab_random(?);

CHECK_STATUS(status);

memcpy(msg_my_nonce, tx_packet->nonce.value, 32); /*update the local nonce after random and
comm_server has been called.  Allows for loopback operation*/

tx_packet->type = MSG_PACKET_TYPE_NONCE_RESPONSE;

usart_write_buffer_job(module, (uint8_t*)tx_packet, sizeof(msg_packet_t));

printf("Generated nonce\r\n");print_bytes(&msg_my_nonce, 32);printf("\r\n");


return 0;
```

Step 4: Add the correct storage parameter to the `atcab_random()` function at "?".

Hint: Look at the `symmetric_auth()` function from the previous section. Note the variable type of the parameter will not require a "&".

Hint: If you are having trouble please see Appendix A.7.

Step 5: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Note: Now we need to program the second board so the nonce request can be responded to.

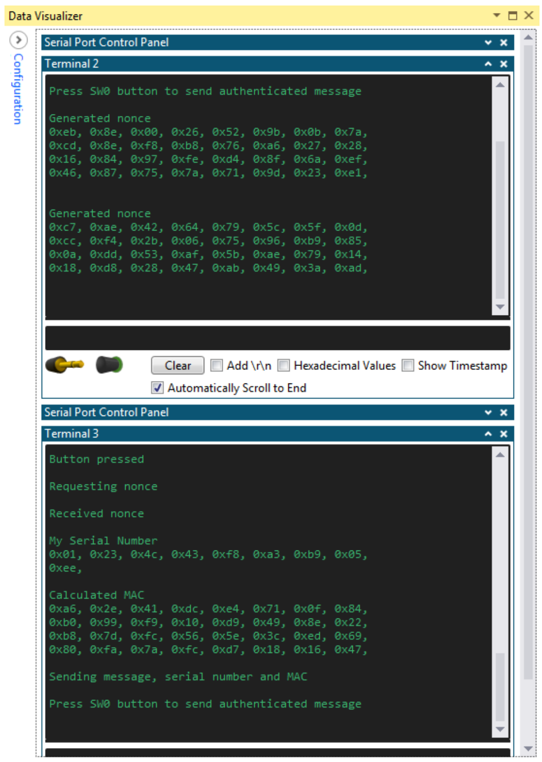
Step 6: On the Studio 7 menu bar, click on the button to bring up the properties page for the SAM D21 Xplained Pro boards.

Step 7: Click on the "Tool" menu in the side tabs.

Step 8: Click on the drop-down icon under the "Selected debugger/programmer" label and select the other board.

Step 9: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Step 10: Verify the MAC value is printed to the terminal screen and that it now changes with each button press (SW0 pressed for board printing to "Terminal 3" in this example). Also verify the other board prints the nonce value sent (Terminal 2 in this example). (See below image)



```
Press SW0 button to send authenticated message

Generated nonce
0xeb, 0x8e, 0x00, 0x26, 0x52, 0x9b, 0x0b, 0x7a,
0xcd, 0x8e, 0xf8, 0xb8, 0x76, 0xa6, 0x27, 0x28,
0x16, 0x84, 0x97, 0xfe, 0xd4, 0x8f, 0x6a, 0xef,
0x46, 0x87, 0x75, 0x7a, 0x71, 0x9d, 0x23, 0xe1,


Generated nonce
0xc7, 0xae, 0x42, 0x64, 0x79, 0x5c, 0x5f, 0x0d,
0xcc, 0xf4, 0x2b, 0x06, 0x75, 0x96, 0xb9, 0x85,
0x0a, 0xdd, 0x53, 0xaf, 0x5b, 0xae, 0x79, 0x14,
0x18, 0xd8, 0x28, 0x47, 0xab, 0x49, 0x3a, 0xad,
```

[✓ Automatically Scroll to End]

```
Button pressed

Requesting nonce

Received nonce

My Serial Number
0x01, 0x23, 0x4c, 0x43, 0xf8, 0xa3, 0xb9, 0x05,
0xee,

Calculated MAC
0xa6, 0x2e, 0x41, 0xdc, 0xe4, 0x71, 0x0f, 0x84,
0xb0, 0x99, 0xf9, 0x10, 0xd9, 0x49, 0x8e, 0x22,
0xb8, 0x7d, 0xfc, 0x56, 0x5e, 0x3c, 0xed, 0x69,
0x80, 0xfa, 0x7a, 0xfc, 0xd7, 0x18, 0x16, 0x47,

Sending message, serial number and MAC

Press SW0 button to send authenticated message
```

4.4.8 HOST checks the MAC value returned by the REMOTE

In this section you will add a code segment to check the mac value that was supplied by the remote client device.

Step 1: Click on the `main_symmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 2.6`

Step 3: Add the following code:

```
  //Step 2.6
volatile ATCA_STATUS status;

status = atcab_init( &cfg_ateccx08a_i2c_host );

uint8_t slot = 0;
uint8_t mode = (1<<6); // include serial number

uint8_t otherdata[CHECKMAC_OTHER_DATA_SIZE];
memset(otherdata, 0x00, CHECKMAC_OTHER_DATA_SIZE);
otherdata[0] = 0x08; //match to mac command byte opp code
otherdata[1] = mode; // match to mac mode

otherdata[7] = rx_packet->auth.serial_number[4];
otherdata[8] = rx_packet->auth.serial_number[5];
otherdata[9] = rx_packet->auth.serial_number[6];
otherdata[10] = rx_packet->auth.serial_number[7];

otherdata[11] = rx_packet->auth.serial_number[2];
otherdata[12] = rx_packet->auth.serial_number[3];

//code to check the supplied mac value
status = atcab_checkmac(?,
                        ?,
                        ?,
                        ?,
                        (const uint8_t*)&?);

if(status == ATCA_SUCCESS) {
  printf("Authenticated\r\n\r\n");
} else {
  printf("Failed to authenticate\r\n\r\n");
}
```

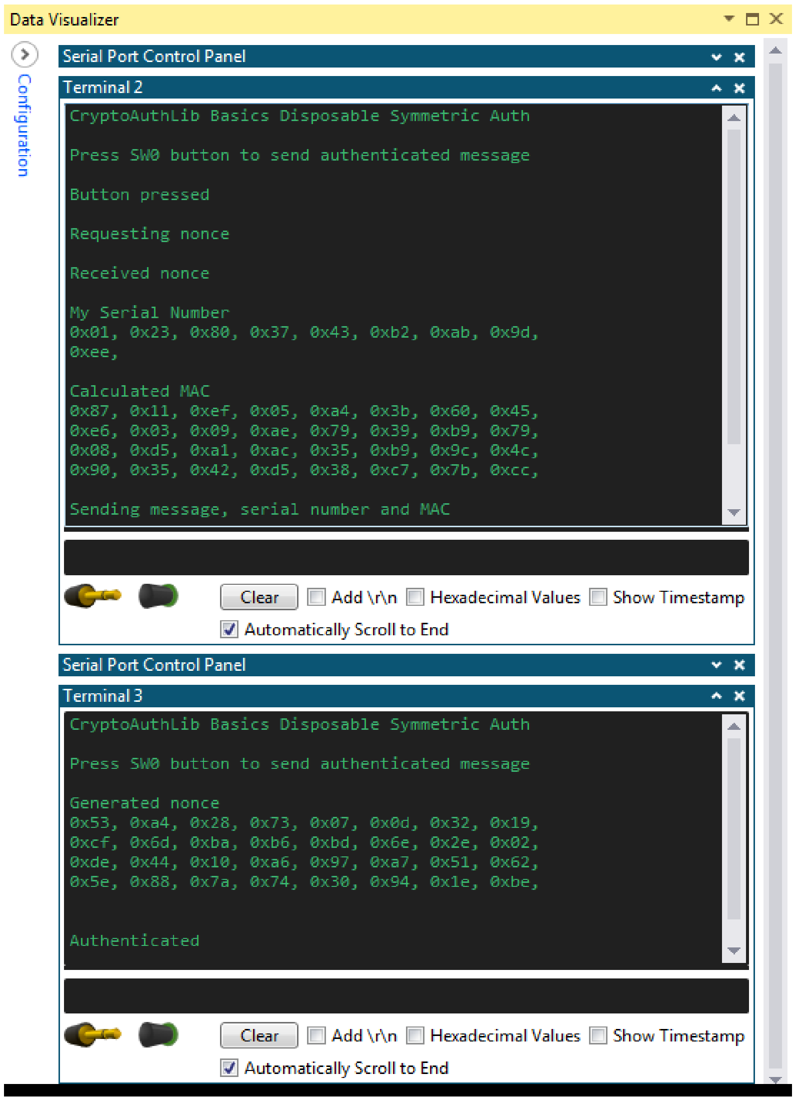
Step 4: Add the correct parameters to the `atcab_checkmac()` function at "?".

> ***Hint:*** Look at the `symmetric_auth()` function from the previous section. Also look at the function prototype in `atca_basic.h` file.
>
> ***Hint:*** If you need help see Appendix A.8

Step 5: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Step 6: Program the "second" board.

Step 7: Press SW0 on either board and view the results in the Data Visualizer window. Either board will act as the local or remote node depending on which board's button is pressed. The board whose button is pressed we will call the local node. The board whose terminal window reports back "Authenticated" or "Failed" is the remote node.



```
CryptoAuthLib Basics Disposable Symmetric Auth

Press SW0 button to send authenticated message

Button pressed

Requesting nonce

Received nonce

My Serial Number
0x01, 0x23, 0x80, 0x37, 0x43, 0xb2, 0xab, 0x9d,
0xee,

Calculated MAC
0x87, 0x11, 0xef, 0x05, 0xa4, 0x3b, 0x60, 0x45,
0xe6, 0x03, 0x09, 0xae, 0x79, 0x39, 0xb9, 0x79,
0x08, 0xd5, 0xa1, 0xac, 0x35, 0xb9, 0x9c, 0x4c,
0x90, 0x35, 0x42, 0xd5, 0x38, 0xc7, 0x7b, 0xcc,

Sending message, serial number and MAC
```

```
CryptoAuthLib Basics Disposable Symmetric Auth

Press SW0 button to send authenticated message

Generated nonce
0x53, 0xa4, 0x28, 0x73, 0x07, 0x0d, 0x32, 0x19,
0xcf, 0x6d, 0xba, 0xb6, 0xbd, 0x6e, 0x2e, 0x02,
0xde, 0x44, 0x10, 0xa6, 0x97, 0xa7, 0x51, 0x62,
0x5e, 0x88, 0x7a, 0x74, 0x30, 0x94, 0x1e, 0xbe,


Authenticated
```

4.4.9 Summary

Congratulations, you have completed a basic authentication example using two boards each with an ECC508 containing the system root authentication secret.

- Initialized I^{2}C communications with `atcab_init()` and read the serial number with `atcab_read_serial_number()`
- Generated a Nonce Request message on a button press
- Generated an Auth Response message with `atcab_mac()`
- Generated a Nonce Response message with a `atcab_random()`
- Authenticated an Auth Response message using `atcab_checkmac()`

5 LAB 3 –Asymmetric Authentication of a Remote Device

5.1 Overview

Asymmetric authentication is a process by which a verifier checks the authenticity of a remote system by validating a signature. Asymmetric authentication is based on the use of two keys. One of the keys needs to be kept secret. This key is called the **Private Key**. The second key is mathematically related to the Private Key and is called the **Public Key**. The public key is openly shared. Anyone who wishes to authenticate the key owner will use the Public Key to do so.

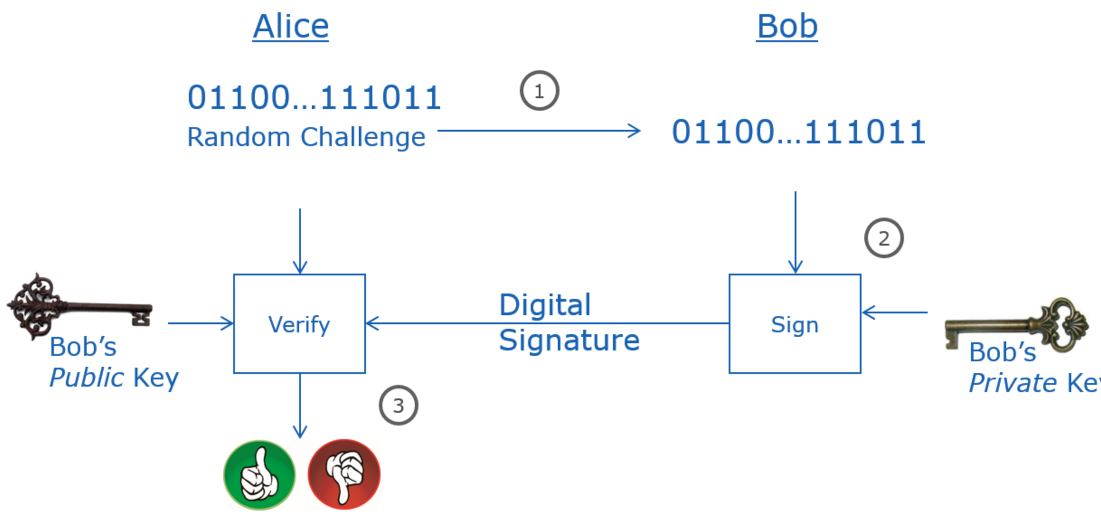
In this lab you will emulate a system that contains:

- **Host device** that contains a microcontroller and a secure hardware crypto authorization device, the ECC508A.
- **Remote device** that contains only a secure hardware crypto authorization device, the ECC508A (no microcontroller).

Both the host and the remote device's ECC508A have been preconfigured for you with a **Private** and corresponding **Public** keys.

5.2 Authentication Process

The authentication process precedes much like symmetric authentication. The verifier below, Alice, sends a challenge to Bob. Bob responds with a signature on Alice's challenge. However, Alice only needs Bob's public key (not a secret key) to verify the signature on the challenge.



5.3 Laboratory Setup

5.3.1 Cryptograph Authentication Project

If you are just starting the lab from here, follow the steps outlined in Section 2.3 to set up the Crypto Authentication Project software

5.3.2 Hardware Requirements

For the lab you will need:

1. (1) SAM D21 Xplained Pro evaluation board
2. (2) CryptoAuth Xplained Pro evaluation boards
3. (1) USB Cable (Type A to Micro B)

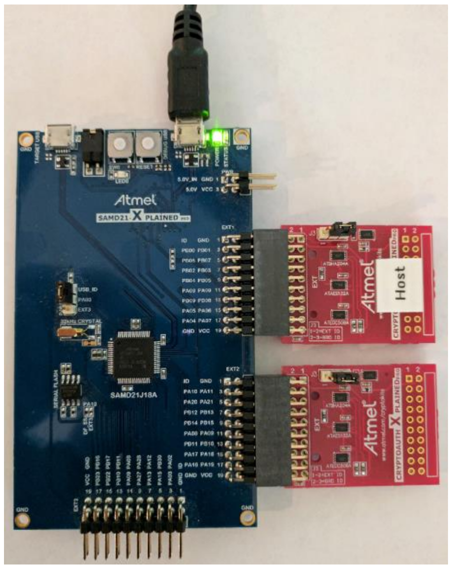
5.3.3 Plug in CryptoAuth Xplained Pro boards

Step 1: Unplug all SAMD21Xplained Pro board USB cables and then CryptoAuth Xplained Pro wing boards.

Step 2: Insert the CryptoAuth Xplained Pro board labeled “HOST” to the SAM D21 Xplained Pro EXT1 port.

Step 3: Insert the CryptoAuth Xplained Pro board labeled “REMOTE” to the SAM D21 Xplained Pro EXT2 port.

Step 4: Connect a USB cable to the DEBUG USB port and PC.



5.3.4 Project Preparation

In this step you will make the `main_asymmetric` project file active.

Step 1: Open Atmel Studio and open the CryptoAuthentication_Basics project.

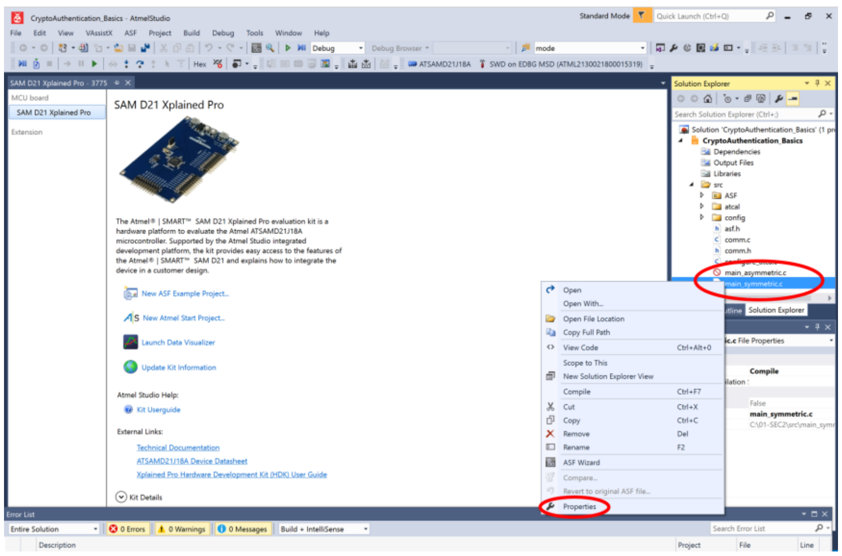
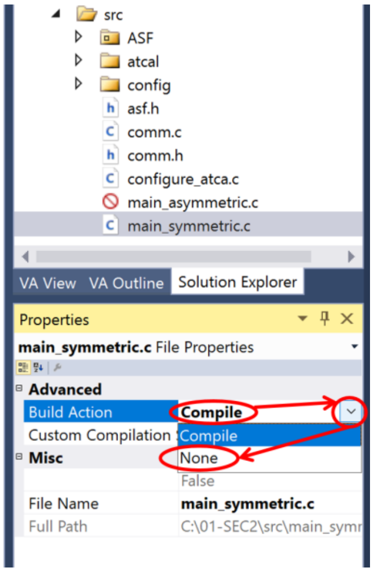
Step 2: Right click on `main_symmetric.c` in the solution explore window.

Step 3: Select “Properties” and go to the “Properties” pane.

Step 4: Click the drop down next to “Compile” and select “None”.

This will remove main_symmetric.c from the compile list.

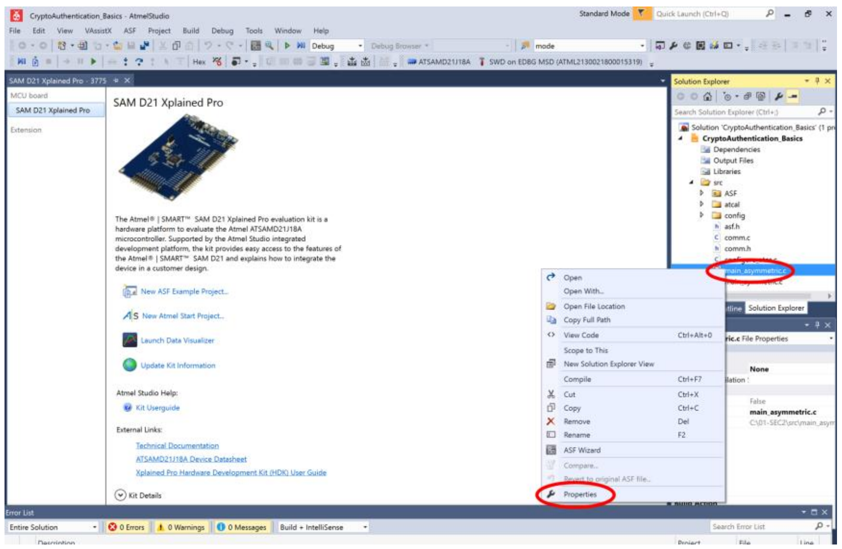
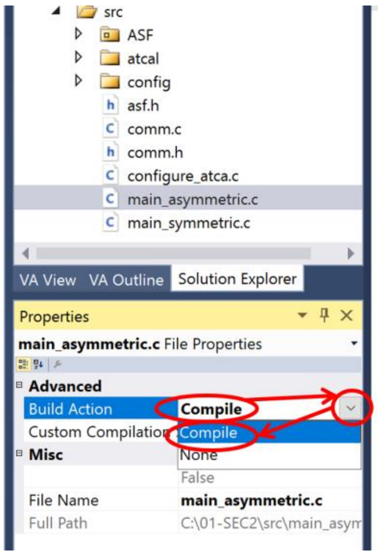
Step 4: Right click on `main_asymmetric.c` and click the drop down and select “Compile”.



src
ASF
atcal
config
asf.h
comm.c
comm.h
configure_atca.c
main_asymmetric.c
main_symmetric.c

Properties

main_symmetric.c File Properties

Advanced	
Build Action	Compile
Custom Compilation	Compile
Misc	None
	False
File Name	main_symmetric.c
Full Path	C:\01-SEC2\src\main_symr



src
ASF
atcal
config
asf.h
comm.c
comm.h
configure_atca.c
main_asymmetric.c
main_symmetric.c

Properties

main_asymmetric.c File Properties

Advanced	
Build Action	Compile
Custom Compilation	Compile
Misc	None
	False
File Name	main_asymmetric.c
Full Path	C:\01-SEC2\src\main_asym

Observe: The resulting file icons should look like this.

main_asymmetric.c

main_symmetric.c

5.3.5 Test Compile 1

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 3.1`

Step 3: Add the following code:

```
    //Step 3.1
    printf("CryptoAuthLib Basics Disposable Asymmetric Auth\n\r");
```

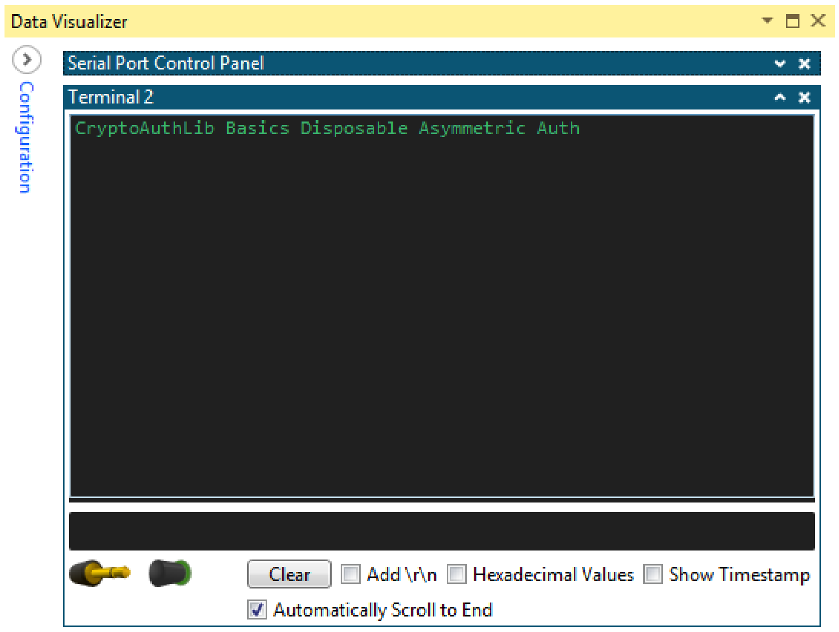
Step 4: On the Studio 7 menu bar, click on the “Build Solution” button.

Step 5: Observe in the output window that the build completed successfully.

Step 6: Open/Navigate to the Data Visualizer Window.

Step 7: On the Studio 7 menu bar, click on the “Start without debugging” button.

Step 8: Verify the below text is printed to the terminal.



```
CryptoAuthLib Basics Disposable Asymmetric Auth
```

5.3.6 Wait for SW0 Button Press

Add a code segment that will wait for push button SW0 to be pressed to start the process of authentication. This code will also initialize I^2C communications.

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 3.2`

Step 3: Add the following code:

```
//Step 3.2
printf("Press SW0 button to authenticate\n\r");
while(port_pin_get_input_level(BUTTON_0_PIN) == SW0_INACTIVE);
printf("Authentication in progress\n\r");
volatile ATCA_STATUS status;
status = atcab_init( &cfg_ateccx08a_i2c_host );
CHECK_STATUS(status);
printf("Device init complete\n\r");
```

Step 4: On the Studio 7 menu bar, click on the "Build Solution" button.

Step 5: Observe in the output window that the build completed successfully.

5.3.7 Generating a NONCE (Random Number)

Add a code segment that will generate a random number that will be used in the authentication process. The random number will be used as an argument to the `atcab_sign()` function. The ECC508A has two difference signing modes. It can sign external data sent to the device or it can sign an internal message. In the example below, we will use the external mode.

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 3.3`

Step 3: Add the following code:

```
//Step 3.3
uint8_t nonce[32];
uint8_t signature[64];

status = atcab_random((uint8_t*)&nonce);
CHECK_STATUS(status);
printf("Random from host\r\n");
print_bytes((uint8_t*)&nonce, 32);

status = atcab_init( &cfg_ateccx08a_i2c_remote );

uint8_t slot = 4;
status = atcab_sign(slot, (const uint8_t*)&nonce, (uint8_t*)&signature);
CHECK_STATUS(status);
printf("Signature from remote\r\n");
print_bytes((uint8_t*)&signature, 64);
```

Step 4: On the Studio 7 menu bar, click on the "Build Solution" button.

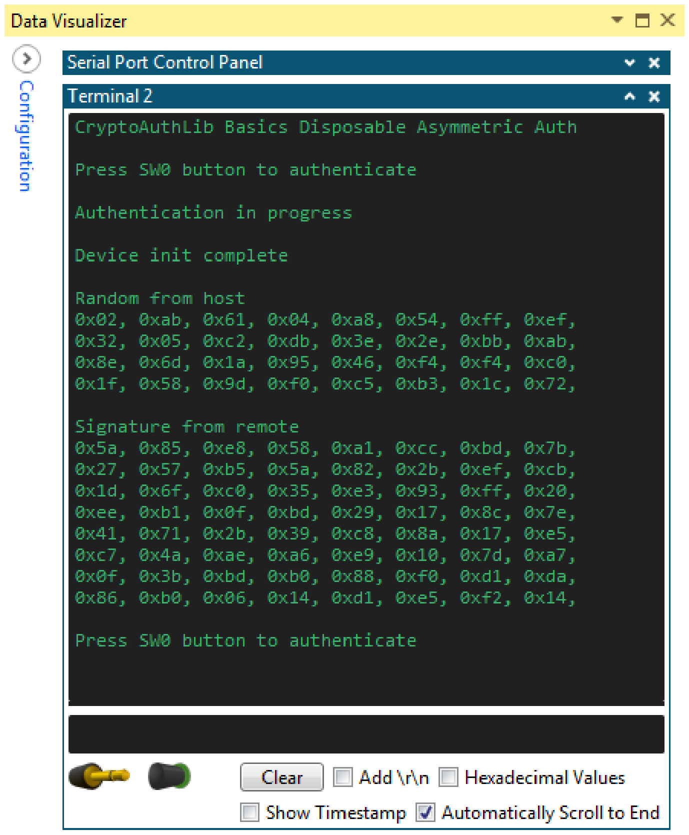
Step 5: Observe in the output window that the build completed successfully.

Step 6: On the Studio 7 menu bar, click on the "Start without debugging" button.

Step 7: Press the SW0 push button.

Step 8: Verify the below text is printed to the terminal.

This code segment caused the REMOTE device to sign a challenge (random number) from the HOST device. The purpose of the sign operation is for the REMOTE device to prove ownership of a private key, just as a `mac` operation in symmetric cryptography proves knowledge of a symmetric key. However, in asymmetric cryptography the two devices do not need to have any pre-shared secret. But the HOST does need to know the REMOTE device's public key which is the REMOTE device's identity to our HOST device.



```
CryptoAuthLib Basics Disposable Asymmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Random from host
0x02, 0xab, 0x61, 0x04, 0xa8, 0x54, 0xff, 0xef,
0x32, 0x05, 0xc2, 0xdb, 0x3e, 0x2e, 0xbb, 0xab,
0x8e, 0x6d, 0x1a, 0x95, 0x46, 0xf4, 0xf4, 0xc0,
0x1f, 0x58, 0x9d, 0xf0, 0xc5, 0xb3, 0x1c, 0x72,

Signature from remote
0x5a, 0x85, 0xe8, 0x58, 0xa1, 0xcc, 0xbd, 0x7b,
0x27, 0x57, 0xb5, 0x5a, 0x82, 0x2b, 0xef, 0xcb,
0x1d, 0x6f, 0xc0, 0x35, 0xe3, 0x93, 0xff, 0x20,
0xee, 0xb1, 0x0f, 0xbd, 0x29, 0x17, 0x8c, 0x7e,
0x41, 0x71, 0x2b, 0x39, 0xc8, 0x8a, 0x17, 0xe5,
0xc7, 0x4a, 0xae, 0xa6, 0xe9, 0x10, 0x7d, 0xa7,
0x0f, 0x3b, 0xbd, 0xb0, 0x88, 0xf0, 0xd1, 0xda,
0x86, 0xb0, 0x06, 0x14, 0xd1, 0xe5, 0xf2, 0x14,

Press SW0 button to authenticate
```

5.3.8 Read the REMOTE Device Public Key

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 3.4`

Step 3: Add the following code:

```
//Step 3.4
uint8_t temp_pubk[64];
status = atcab_get_pubkey(slot, &temp_pubk);
CHECK_STATUS(status);
printf("Remote disposable public key\r\n");
print_bytes((uint8_t*)&temp_pubk, 64);
```

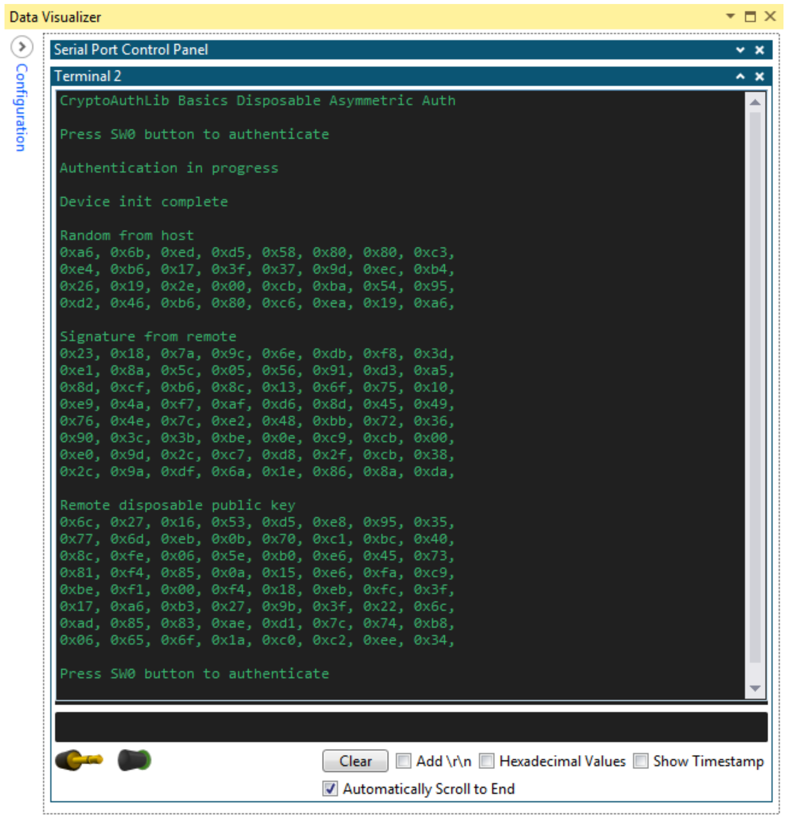
Step 6: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 7: Press the SW0 push button.

Step 8: Verify the below text is printed to the terminal.

We see the REMOTE device’s public key printed to the terminal. The HOST device needs knowledge of this public key prior to the REMOTE device being connected to the system. The HOST’s knowledge of this key or a list of public keys is what determines what REMOTE devices are authorized by the HOST.

In symmetric cryptography a REMOTE device proves it has received authorization by proving it knows a secret key. In asymmetric cryptography the REMOTE device proves knowledge of a private key to which the host already knows the public key. This is how trust is established.

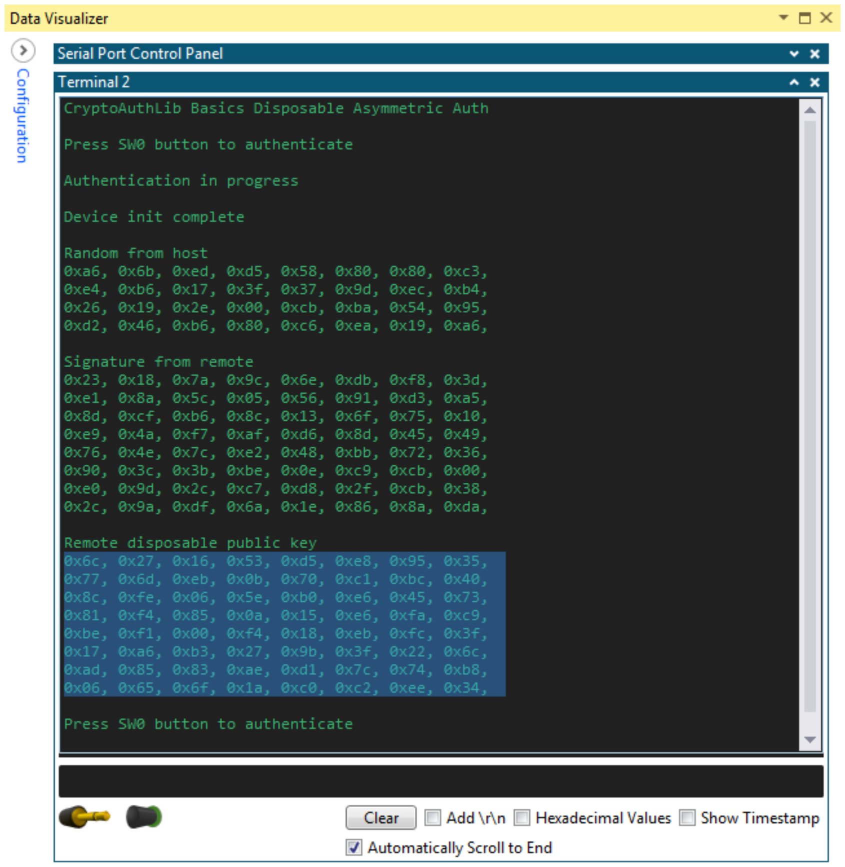


```
CryptoAuthLib Basics Disposable Asymmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Random from host
0xa6, 0x6b, 0xed, 0xd5, 0x58, 0x80, 0x80, 0xc3,
0xe4, 0xb6, 0x17, 0x3f, 0x37, 0x9d, 0xec, 0xb4,
0x26, 0x19, 0x2e, 0x00, 0xcb, 0xba, 0x54, 0x95,
0xd2, 0x46, 0xb6, 0x80, 0xc6, 0xea, 0x19, 0xa6,

Signature from remote
0x23, 0x18, 0x7a, 0x9c, 0x6e, 0xdb, 0xf8, 0x3d,
0xe1, 0x8a, 0x5c, 0x05, 0x56, 0x91, 0xd3, 0xa5,
0x8d, 0xcf, 0xb6, 0x8c, 0x13, 0x6f, 0x75, 0x10,
0xe9, 0x4a, 0xf7, 0xaf, 0xd6, 0x8d, 0x45, 0x49,
0x76, 0x4e, 0x7c, 0xe2, 0x48, 0xbb, 0x72, 0x36,
0x90, 0x3c, 0x3b, 0xbe, 0x0e, 0xc9, 0xcb, 0x00,
0xe0, 0x9d, 0x2c, 0xc7, 0xd8, 0x2f, 0xcb, 0x38,
0x2c, 0x9a, 0xdf, 0x6a, 0x1e, 0x86, 0x8a, 0xda,

Remote disposable public key
0x6c, 0x27, 0x16, 0x53, 0xd5, 0xe8, 0x95, 0x35,
0x77, 0x6d, 0xeb, 0x0b, 0x70, 0xc1, 0xbc, 0x40,
0x8c, 0xfe, 0x06, 0x5e, 0xb0, 0xe6, 0x45, 0x73,
0x81, 0xf4, 0x85, 0x0a, 0x15, 0xe6, 0xfa, 0xc9,
0xbe, 0xf1, 0x00, 0xf4, 0x18, 0xeb, 0xfc, 0x3f,
0x17, 0xa6, 0xb3, 0x27, 0x9b, 0x3f, 0x22, 0x6c,
0xad, 0x85, 0x83, 0xae, 0xd1, 0x7c, 0x74, 0xb8,
0x06, 0x65, 0x6f, 0x1a, 0xc0, 0xc2, 0xee, 0x34,

Press SW0 button to authenticate
```

Automatically Scroll to End [✓]

5.3.9 Copy and Paste the REMOTE Device Public Key to the HOST to establish trust

Step 1: In the terminal window, using your mouse, select and copy the “REMOTE public key”

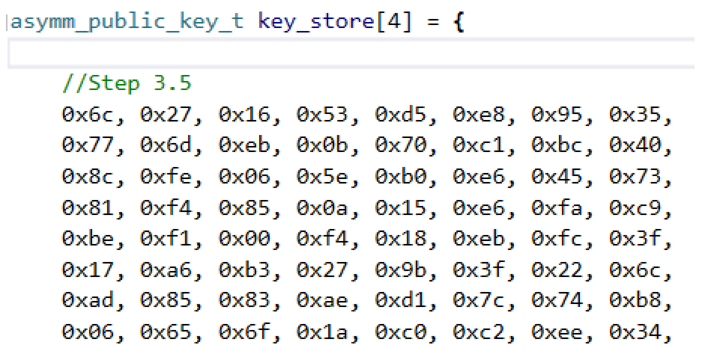


```
CryptoAuthLib Basics Disposable Asymmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Random from host
0xa6, 0x6b, 0xed, 0xd5, 0x58, 0x80, 0x80, 0xc3,
0xe4, 0xb6, 0x17, 0x3f, 0x37, 0x9d, 0xec, 0xb4,
0x26, 0x19, 0x2e, 0x00, 0xcb, 0xba, 0x54, 0x95,
0xd2, 0x46, 0xb6, 0x80, 0xc6, 0xea, 0x19, 0xa6,

Signature from remote
0x23, 0x18, 0x7a, 0x9c, 0x6e, 0xdb, 0xf8, 0x3d,
0xe1, 0x8a, 0x5c, 0x05, 0x56, 0x91, 0xd3, 0xa5,
0x8d, 0xcf, 0xb6, 0x8c, 0x13, 0x6f, 0x75, 0x10,
0xe9, 0x4a, 0xf7, 0xaf, 0xd6, 0x8d, 0x45, 0x49,
0x76, 0x4e, 0x7c, 0xe2, 0x48, 0xbb, 0x72, 0x36,
0x90, 0x3c, 0x3b, 0xbe, 0x0e, 0xc9, 0xcb, 0x00,
0xe0, 0x9d, 0x2c, 0xc7, 0xd8, 0x2f, 0xcb, 0x38,
0x2c, 0x9a, 0xdf, 0x6a, 0x1e, 0x86, 0x8a, 0xda,

Remote disposable public key
0x6c, 0x27, 0x16, 0x53, 0xd5, 0xe8, 0x95, 0x35,
0x77, 0x6d, 0xeb, 0x0b, 0x70, 0xc1, 0xbc, 0x40,
0x8c, 0xfe, 0x06, 0x5e, 0xb0, 0xe6, 0x45, 0x73,
0x81, 0xf4, 0x85, 0x0a, 0x15, 0xe6, 0xfa, 0xc9,
0xbe, 0xf1, 0x00, 0xf4, 0x18, 0xeb, 0xfc, 0x3f,
0x17, 0xa6, 0xb3, 0x27, 0x9b, 0x3f, 0x22, 0x6c,
0xad, 0x85, 0x83, 0xae, 0xd1, 0x7c, 0x74, 0xb8,
0x06, 0x65, 0x6f, 0x1a, 0xc0, 0xc2, 0xee, 0x34,

Press SW0 button to authenticate
```

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 3.5`

Step 3: Paste the REMOTE key data at `//`**`Step 3.5`** in the `key_store[ ]` array variable.
It should look like below with different byte values.



```
asymm_public_key_t key_store[4] = {

    //Step 3.5
    0x6c, 0x27, 0x16, 0x53, 0xd5, 0xe8, 0x95, 0x35,
    0x77, 0x6d, 0xeb, 0x0b, 0x70, 0xc1, 0xbc, 0x40,
    0x8c, 0xfe, 0x06, 0x5e, 0xb0, 0xe6, 0x45, 0x73,
    0x81, 0xf4, 0x85, 0x0a, 0x15, 0xe6, 0xfa, 0xc9,
    0xbe, 0xf1, 0x00, 0xf4, 0x18, 0xeb, 0xfc, 0x3f,
    0x17, 0xa6, 0xb3, 0x27, 0x9b, 0x3f, 0x22, 0x6c,
    0xad, 0x85, 0x83, 0xae, 0xd1, 0x7c, 0x74, 0xb8,
    0x06, 0x65, 0x6f, 0x1a, 0xc0, 0xc2, 0xee, 0x34,
```

5.3.10 Search Local Key Store

Add a code segment that will search the HOST local key store for the public key that the REMOTE device claims to control. The key store holds all trusted keys. If the key is found then we will use the public key to verify that the REMOTE device correctly signed the challenge that was sent.

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 3.6`

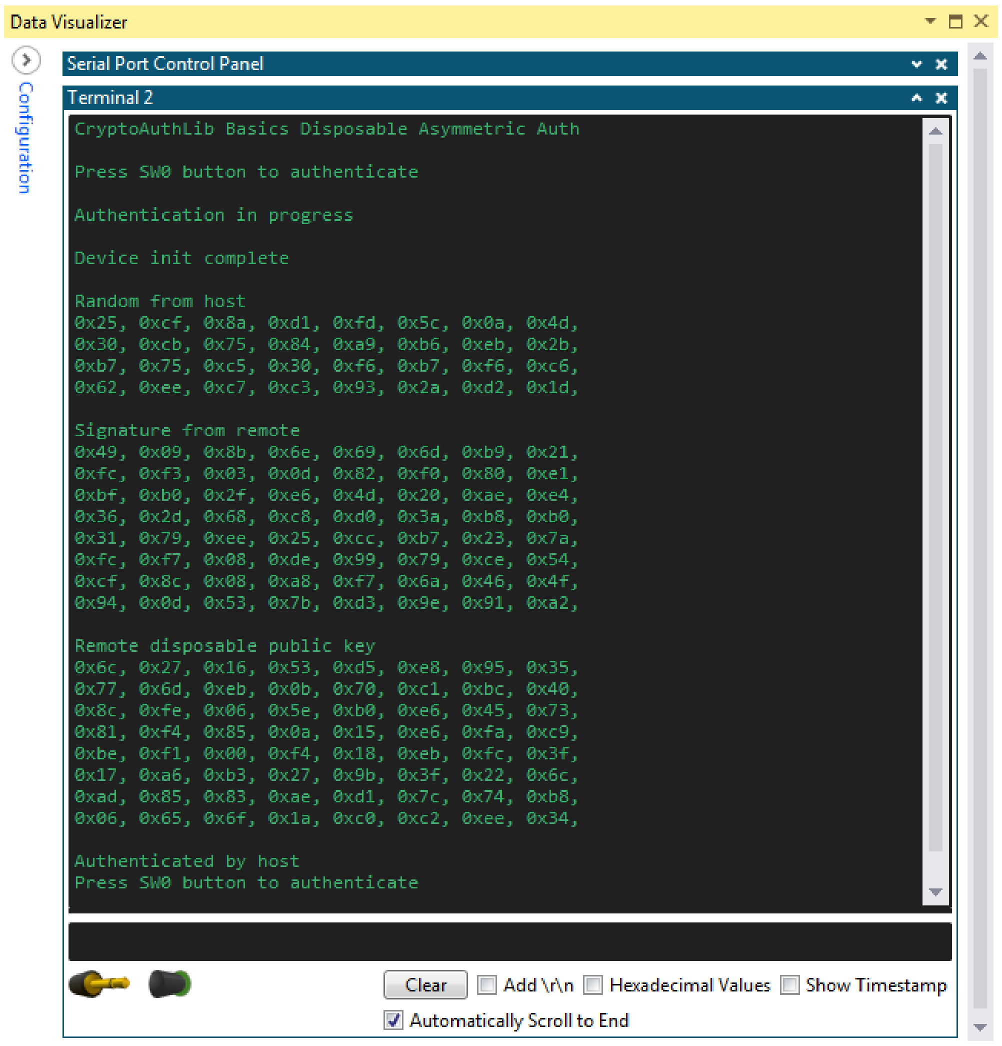
Step 3: Add the following code:

```
//Step 3.6
status = atcab_init( &cfg_ateccx08a_i2c_host );
CHECK_STATUS(status);

bool verify = false;
bool key_found = false;
uint8_t i = 0;
for(;i < sizeof(key_store)/sizeof(asymm_public_key_t); i++) {
     if(memcmp(&key_store[i], &temp_pubk, 64) == 0) {
       key_found = true;
       break;
     }
}
if(key_found) {

       status = atcab_verify_extern((const uint8_t*)&nonce,
                                   (const uint8_t*)&signature,
                                   (const uint8_t*)key_store[i].pub_key,
                                                    &verify);
}
if(verify) {
     printf("Authenticated by host\r\n");
} else {
     printf("Failed to authenticate\r\n");
}
```

Step 6: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 7: Press the SW0 push button.

Step 8: Verify the below text is printed to the terminal.



```
CryptoAuthLib Basics Disposable Asymmetric Auth

Press SW0 button to authenticate

Authentication in progress

Device init complete

Random from host
0x25, 0xcf, 0x8a, 0xd1, 0xfd, 0x5c, 0x0a, 0x4d,
0x30, 0xcb, 0x75, 0x84, 0xa9, 0xb6, 0xeb, 0x2b,
0xb7, 0x75, 0xc5, 0x30, 0xf6, 0xb7, 0xf6, 0xc6,
0x62, 0xee, 0xc7, 0xc3, 0x93, 0x2a, 0xd2, 0x1d,

Signature from remote
0x49, 0x09, 0x8b, 0x6e, 0x69, 0x6d, 0xb9, 0x21,
0xfc, 0xf3, 0x03, 0x0d, 0x82, 0xf0, 0x80, 0xe1,
0xbf, 0xb0, 0x2f, 0xe6, 0x4d, 0x20, 0xae, 0xe4,
0x36, 0x2d, 0x68, 0xc8, 0xd0, 0x3a, 0xb8, 0xb0,
0x31, 0x79, 0xee, 0x25, 0xcc, 0xb7, 0x23, 0x7a,
0xfc, 0xf7, 0x08, 0xde, 0x99, 0x79, 0xce, 0x54,
0xcf, 0x8c, 0x08, 0xa8, 0xf7, 0x6a, 0x46, 0x4f,
0x94, 0x0d, 0x53, 0x7b, 0xd3, 0x9e, 0x91, 0xa2,

Remote disposable public key
0x6c, 0x27, 0x16, 0x53, 0xd5, 0xe8, 0x95, 0x35,
0x77, 0x6d, 0xeb, 0x0b, 0x70, 0xc1, 0xbc, 0x40,
0x8c, 0xfe, 0x06, 0x5e, 0xb0, 0xe6, 0x45, 0x73,
0x81, 0xf4, 0x85, 0x0a, 0x15, 0xe6, 0xfa, 0xc9,
0xbe, 0xf1, 0x00, 0xf4, 0x18, 0xeb, 0xfc, 0x3f,
0x17, 0xa6, 0xb3, 0x27, 0x9b, 0x3f, 0x22, 0x6c,
0xad, 0x85, 0x83, 0xae, 0xd1, 0x7c, 0x74, 0xb8,
0x06, 0x65, 0x6f, 0x1a, 0xc0, 0xc2, 0xee, 0x34,

Authenticated by host
Press SW0 button to authenticate
```

5.3.11 Adding Additional Keys to the Key Store

Ask one of the Lab helpers for some additional REMOTE CryptoAuth Xplained Pro boards.

Unplug the REMOTE board from your system and plug in each of the additional REMOTE boards and repeat the above workflow at **Section 5.3.9** to add each board's public key to your key store of trusted keys.

Notice the terminal will output "Failed to authenticate" when the new board is first plugged in and the SW0 button is pressed. This is because until you add the device's public key to your key store the HOST system does not think this module is authorized.

5.3.12 Summary

Congratulations, you have completed authorization of a REMOTE device using asymmetric authentication.

6 LAB 4 - Asymmetric Key Exchange using the Elliptic Curve Diffie-Hellman protocol (ECDH)

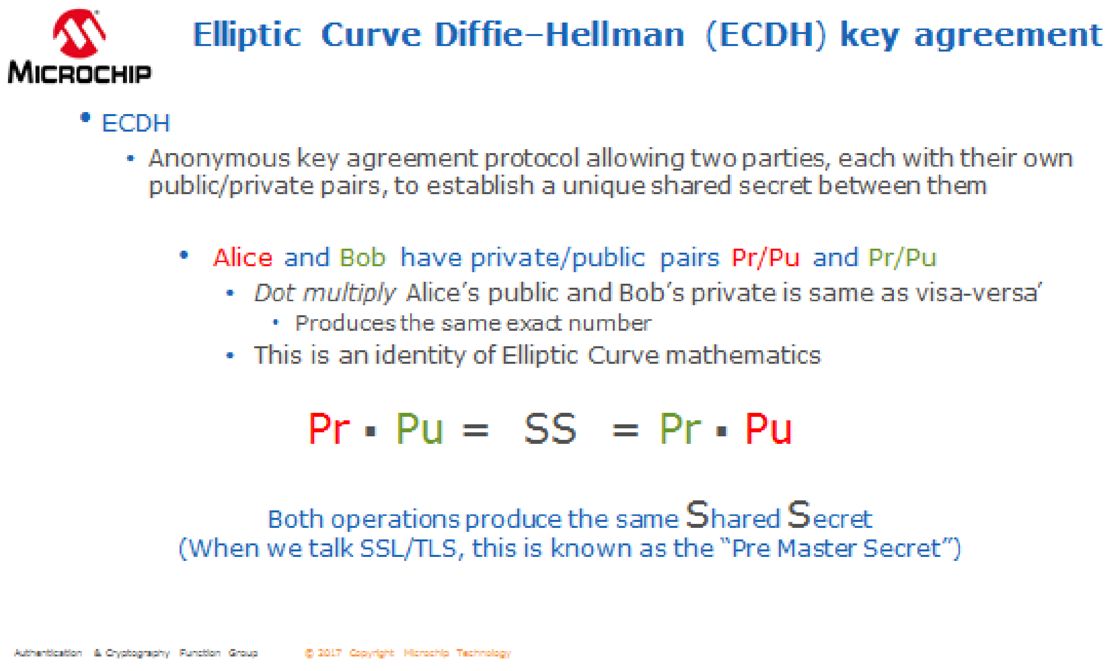
6.1 Overview

Asymmetric cryptography allows for authentication using shared public keys as we saw in the previous section. However, another very important capability is the ability to exchange or agree upon a secret piece of information with another party. This secret is typically then used as a symmetric encryption key. Some asymmetric algorithms do allow for the direct encryption of data like the RSA algorithm. However, asymmetric encryption can be 1000’s of times slower than using symmetric cryptography. So it is most efficient and most common to use asymmetric cryptography for authentication and key agreement and then use symmetric encryption for encrypting the bulk data.

6.2 ECDH shared secret creation Process

The way ECC key agreement is accomplished is by using an algorithm call Elliptic Curve Diffie Hellman (ECDH). ECDH allows for two parties to create a shared secret. This share value can then be used by each party to exchange encrypted information with the other using a symmetric encryption algorithm like AES. The agreement process requires each party to multiply their private key by the counter party’s public key to generate a shared secret.

The ECC508A has an onboard accelerator to speed the ECC calculations along with key storage to protect the private ECC key.



Elliptic Curve Diffie-Hellman (ECDH) key agreement

- ECDH
  - Anonymous key agreement protocol allowing two parties, each with their own public/private pairs, to establish a unique shared secret between them

- Alice and Bob have private/public pairs Pr/Pu and Pr/Pu
  - *Dot multiply* Alice’s public and Bob’s private is same as visa-versa’
    - Produces the same exact number
  - This is an identity of Elliptic Curve mathematics

$$Pr \cdot Pu = SS = Pr \cdot Pu$$

Both operations produce the same Shared Secret
(When we talk SSL/TLS, this is known as the “Pre Master Secret”)

Authentication & Cryptography Function Group © 2017 Copyright Microchip Technology

6.3 Laboratory Setup

6.3.1 Cryptograph Authentication Project

Follow the steps outlined in Section 2.3 to set up the Crypto Authentication Project or continue from where you left off if you have been working through the document sequentially.

6.3.2 Hardware Requirements

For this lab you will need:

1. (2) SAM D21 Xplained Pro evaluation board
2. (2) CryptoAuth Xplained Pro evaluation boards
3. (2) USB Cable (Type A to Micro B)
5. (2) Jumper wires

6.3.3 Plug in CryptoAuth Xplained Pro boards

Step 1: Unplug all SAMD21Xplained Pro board USB cables and then unplug all CryptoAuth Xplained Pro wing boards.

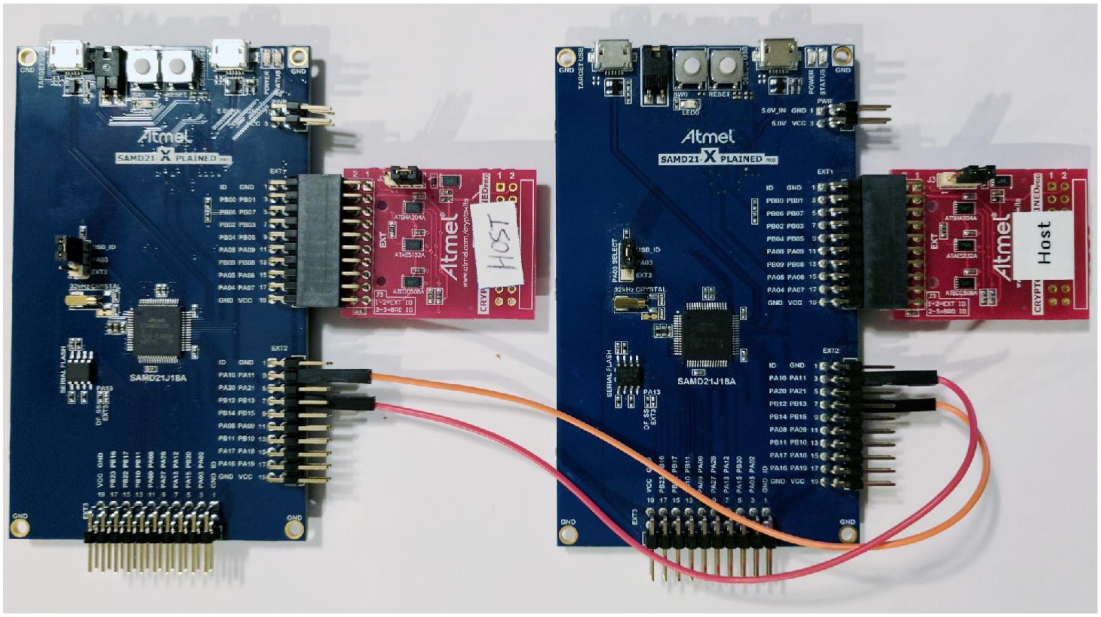
Step 2: Insert the CryptoAuth Xplained Pro board labeled “HOST” to the first SAM D21 Xplained Pro EXT1 port.

Step 3: Insert the CryptoAuth Xplained Pro board labeled “HOST” to the second SAM D21 Xplained Pro EXT1 port.

Step 4: Attach jumper wires as shown below.

Pin PB13 connects to PA10 on the opposite SAM D21 Xplained Pro board. This connects UART TX to the corresponding UART RX on both boards.

Step 5: Connect a USB cable to each board’s DEBUG USB port to the PC.



6.4 Writing Application Software

6.4.1 Project Preparation

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 5.1`

Step 3: Uncomment the following code so it looks like the below:

```
//Step 5.1
#define ENABLE_MODULE_2
```

Step 4: Search the source code to find the comment: `//Step 5.2`

Step 5: Add the following code:

```
//Step 5.2
asymmetric_ecdh_comm();
```

Step 6: Search the source code to find the comment: `//Step 5.3`

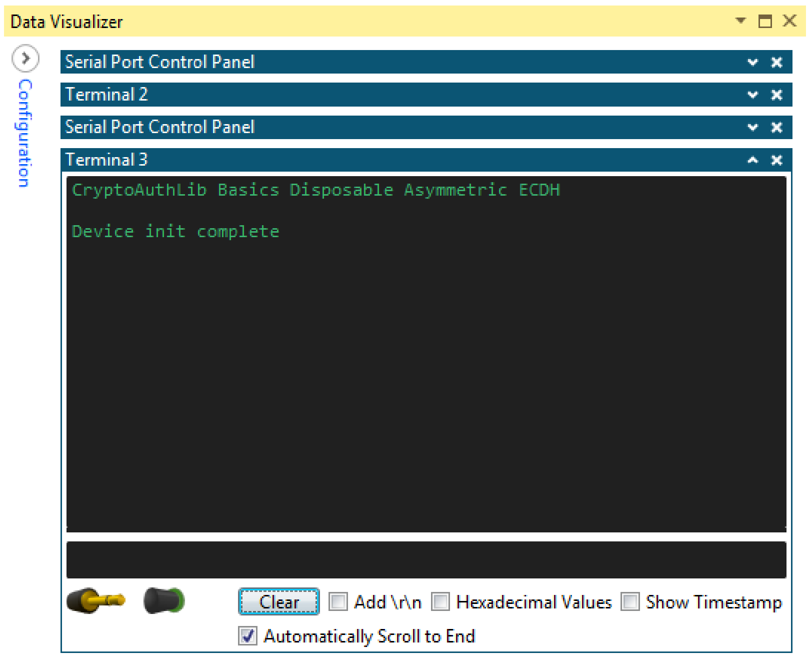
Step 7: Add the following code:

```
//Step 5.3
printf("CryptoAuthLib Basics Disposable Asymmetric ECDH\n\r");

com_init();

volatile ATCA_STATUS status;

status = atcab_init( &cfg_ateccx08a_i2c_host );
CHECK_STATUS(status);

printf("Device init complete\n\r");
```

Step 8: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 9: Press the SW0 push button.

Step 10: Verify the below text is printed to the terminal.



```
CryptoAuthLib Basics Disposable Asymmetric ECDH

Device init complete
```

6.4.2 Programming Two SAM D21 Xplained Pro boards

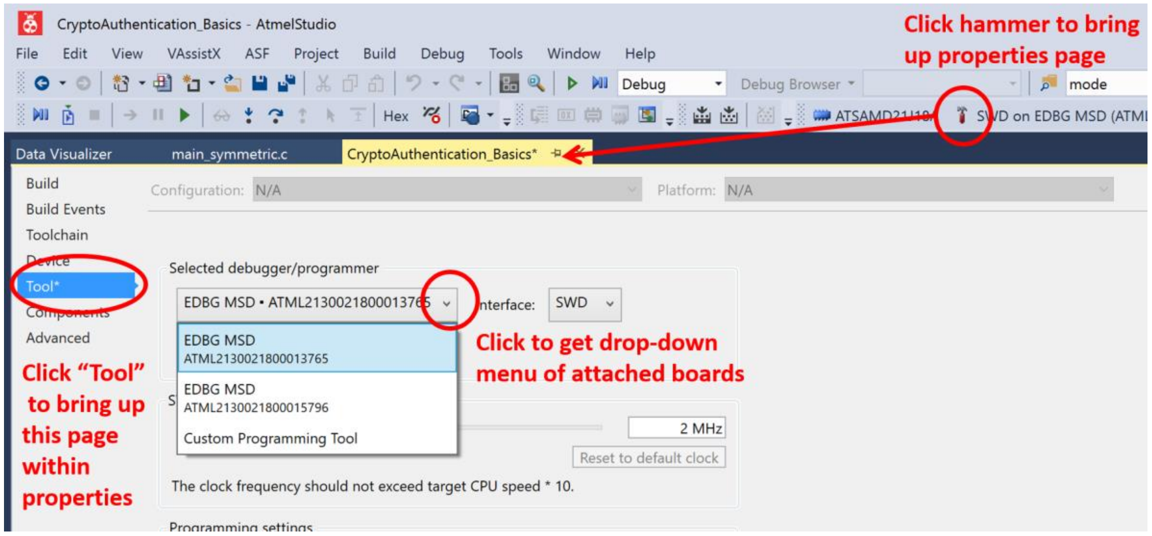
For this lab you have two SAM D21 Xplained Pro boards plugged into the same PC running Studio 7. Each time you add a code segment and compile, you will need to update the program on both boards before proceeding on the sections where the boards communicate.

Here are the steps for selecting which SAM D21 Xplained Pro board will be programmed:

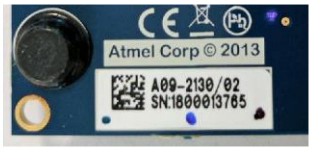
Step 1: On the Studio 7 menu bar, click on the button to bring up the properties page for the SAM D21 Xplained Pro boards.

Step 2: Click on the “Tool” menu in the side tabs.

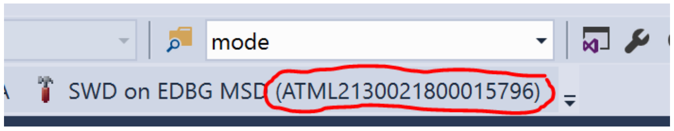
Step 3: Click on the drop-down icon under the “Selected debugger/programmer” label.


Click hammer to bring up properties page
Click "Tool" to bring up this page within properties
Click to get drop-down menu of attached boards
CryptoAuthentication_Basics - AtmelStudio
Data Visualizer
main_symmetric.c
CryptoAuthentication_Basics*
Build
Build Events
Toolchain
Device
Tool*
Components
Advanced
Configuration: N/A
Platform: N/A
Selected debugger/programmer
EDBG MSD • ATML2130021800013765
Interface: SWD
EDBG MSD ATML2130021800013765
EDBG MSD ATML2130021800015796
Custom Programming Tool
2 MHz
Reset to default clock
The clock frequency should not exceed target CPU speed * 10.
Programming settings

The two SAM D21 Xplained Pro boards' serial numbers will be displayed. You can find the serial number on a label on the bottom of the board.



The board serial number at the top of the drop-down box is the board that is selected and will be programmed when you click on the "Start without debugging" button. Note that the selected board's serial number is also displayed next to the icon on the menu bar.



When you press the "Start without debugging" button, you will program the board that is selected and is displayed next to the icon.

6.4.3 Program Device B

Step 1: Select the "other" SAM D21 Xplained Pro board using the steps in Section 4.4.2.

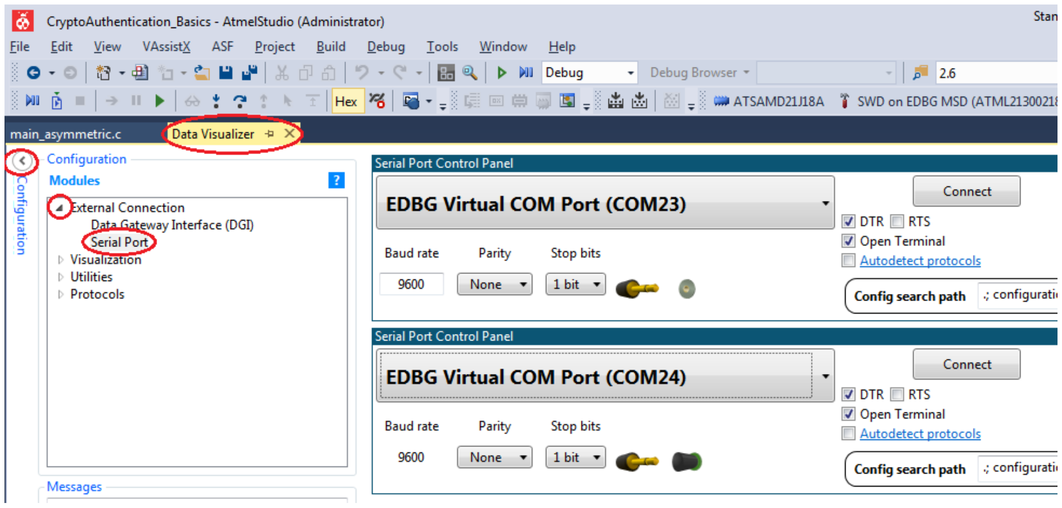
Step 2: On the Studio 7 menu bar, click on the "Start without debugging" button.

Observe: You programmed Device B. Note the board serial number:

____

6.4.4 Open a Second Data Visualizer Window

Step 1: Click on the Data Visualizer tab.

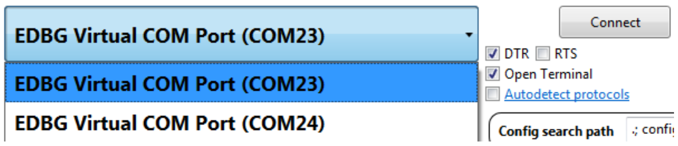


Serial Port Control Panel: EDBG Virtual COM Port (COM23), Baud rate 9600, Parity None, Stop bits 1 bit

Serial Port Control Panel: EDBG Virtual COM Port (COM24), Baud rate 9600, Parity None, Stop bits 1 bit

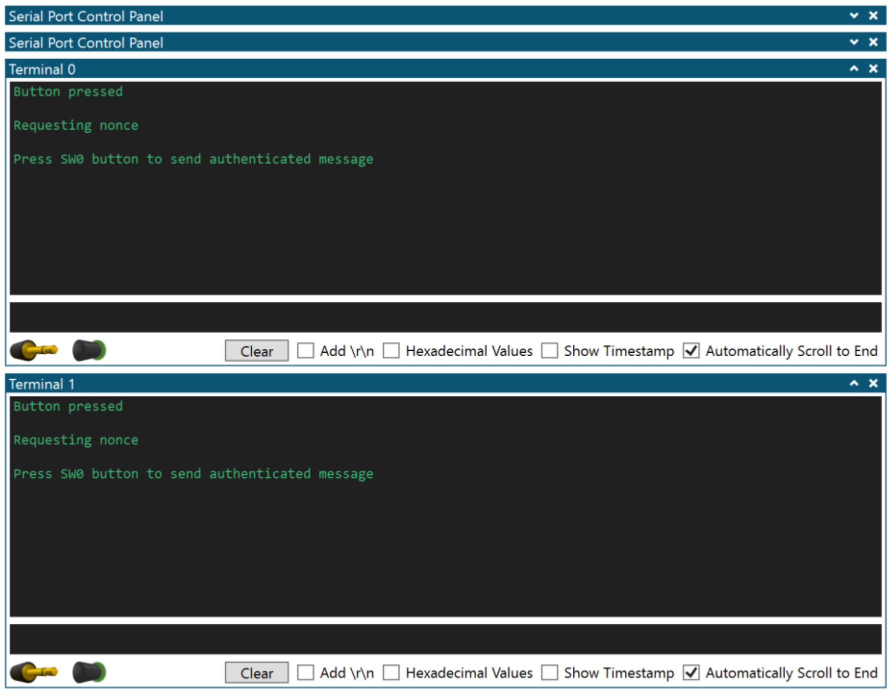
Step 2: Double click “Serial Port” to open a second serial port.

Step 3: Select the newly enumerated COM port of the second board.


EDBG Virtual COM Port (COM23)
EDBG Virtual COM Port (COM23)
EDBG Virtual COM Port (COM24)
Connect
DTR RTS
Open Terminal
Autodetect protocols
Config search path

Step 4: Select Click on the ⌃ or ⌄ icons in the upper right corners to collapse and expand windows for better viewing.

Step 5: Press the RESET button on each board and note which terminal window belongs to each board.


Serial Port Control Panel
Serial Port Control Panel
Terminal 0
Clear
Add \r\n
Hexadecimal Values
Show Timestamp
Automatically Scroll to End
Terminal 1

Terminal 0:

```
Button pressed

Requesting nonce

Press SW0 button to send authenticated message
```

Terminal 1:

```
Button pressed

Requesting nonce

Press SW0 button to send authenticated message
```

Step 6: Confirm that both boards are programmed by verifying you see output like above.

6.4.5 Read Public Key Command

Add a code segment that uses the ECC508A read public key command to calculate the device's public key from its secret private key. The private key in the ECC508A was created during the provisioning process. This was completed for you already in the devices you have been given for the lab. The read public key command performs the ECC multiplication of the generator point by the private key which results in the public key. The public key could also be stored in the MCU flash or the ECC508 data zone if an application does not want to calculate it each time. The below code loads the calculated public key into the global variable `host_pubk`. This value will be sent when the public key request packet is sent by the REMOTE board over the UART.

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 5.4`

Step 3: Add the following code:

```
//Step 5.4
status = atcab_get_pubkey(4, (uint8_t*)&host_pubk);
CHECK_STATUS(status);
```

6.4.6 Check for and Send UART Packets

Add a code segment that will poll the server function to check for incoming packets over the UART and also send requests for the remote board's public key.

Step 1: Search the source code to find the comment: `//Step 5.5`

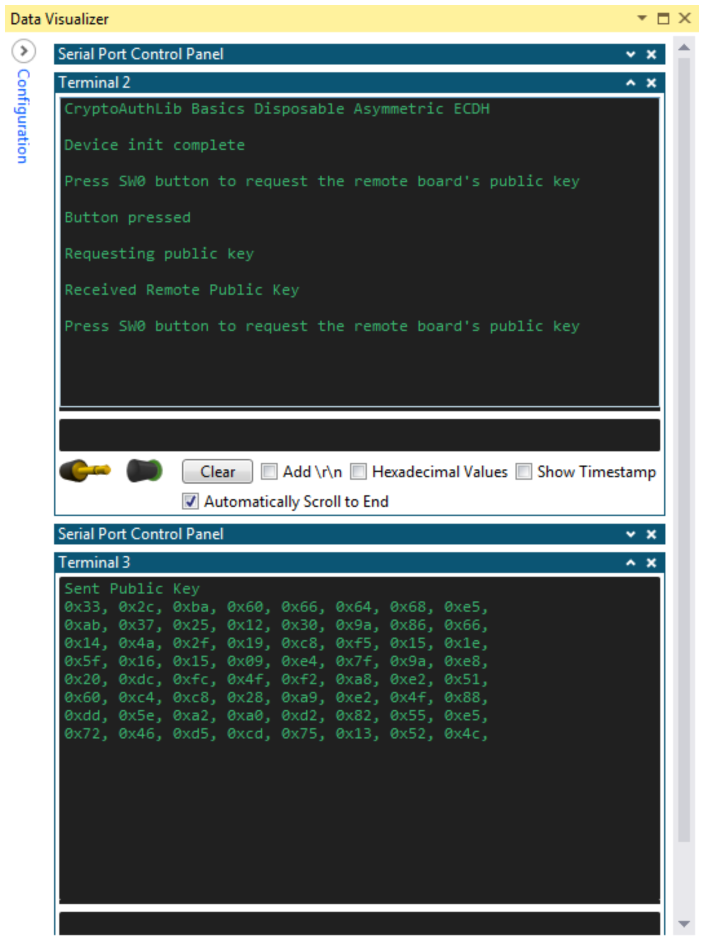
Step 2: Add the following code:

```
        //Step 5.5
          printf("Press SW0 button to request the remote board's public key\n\r");

          volatile uint32_t i = 0;

          while(port_pin_get_input_level(BUTTON_0_PIN) == SW0_INACTIVE) {
                  //check for incoming data
                  _asymmetric_ecdh_comm_server();

                  i++;
                  if(i >= 1000000) {
                          com_restart();
                          i = 0;
                  }

          };

          printf("Button pressed\n\r");
          printf("Requesting public key\n\r");

          msg_packet_t *tx_packet = (msg_packet_t *)com_get_tx_buffer(); /*get tx buffer
          send and request public key*/
          tx_packet->type = MSG_PACKET_TYPE_PUBKEY_SEND;
          struct usart_module * module = com_get_uart(COM_UART_TYPE_TX);
          usart_write_buffer_wait(module, (uint8_t*)tx_packet, sizeof(msg_packet_t));

          delay_ms(100);//wait for remote pubkey
          while(_asymmetric_ecdh_comm_server());

          bool verify = false;
          bool key_found = false;

          //search key store for remote key
          uint8_t index = 0;
          for(;index < sizeof(key_store)/sizeof(asymm_public_key_t); index++) {
                  if(memcmp(&key_store[index], &remote_pubk, 64) == 0) {
                          key_found = true;
                          break;
                  }
          }
```

Step 3: On the Studio 7 menu bar, click on the "Start without debugging" ▶ button.

Step 4: Press the SW0 push button

Step 5: Verify the below text is printed to the terminal.



```
CryptoAuthLib Basics Disposable Asymmetric ECDH

Device init complete

Press SW0 button to request the remote board's public key

Button pressed

Requesting public key

Received Remote Public Key

Press SW0 button to request the remote board's public key
```

```
Sent Public Key
0x33, 0x2c, 0xba, 0x60, 0x66, 0x64, 0x68, 0xe5,
0xab, 0x37, 0x25, 0x12, 0x30, 0x9a, 0x86, 0x66,
0x14, 0x4a, 0x2f, 0x19, 0xc8, 0xf5, 0x15, 0x1e,
0x5f, 0x16, 0x15, 0x09, 0xe4, 0x7f, 0x9a, 0xe8,
0x20, 0xdc, 0xfc, 0x4f, 0xf2, 0xa8, 0xe2, 0x51,
0x60, 0xc4, 0xc8, 0x28, 0xa9, 0xe2, 0x4f, 0x88,
0xdd, 0x5e, 0xa2, 0xa0, 0xd2, 0x82, 0x55, 0xe5,
0x72, 0x46, 0xd5, 0xcd, 0x75, 0x13, 0x52, 0x4c,
```

6.4.7 Establish trust and calculate ECDH shared value

Add a code segment that checks that the remote device's public key is trusted and then uses the ECDH command to calculate the ECDH secret value. The ECDH value needs to be protected so the code uses the encrypted read capability of the 508. The `atcab_ecdh_enc()` function takes as a parameter the transport encryption key to accomplish this. This key would have also been preprogrammed on the ECC508. The effect is the data going over the I2C bus will be encrypted by this key to protect the ECDH secret value being return back to the MCU.

Step 1: Click on the `main_asymmetric.c` tab.

Step 2: Search the source code to find the comment: `//Step 5.6`

Step 3: Add the following code:

```
//Step 5.6
        if(key_found) {

                //Step 5.7
                const uint8_t transport_key[] = {

                                0xf2, 0x11, 0x11, 0x11,
                                0x11, 0x11, 0x11, 0x11,
                                0x11, 0x11, 0x11, 0x11,
                                0x11, 0x11, 0x11, 0x11,

                                0x11, 0x11, 0x11, 0x11,
                                0x11, 0x11, 0x11, 0x11,
                                0x11, 0x11, 0x11, 0x11,
                                0x11, 0x11, 0x11, 0x2f,

                };

                uint8_t private_key_slot = 4;
                uint8_t transport_key_slot = 2;

                //calculate ECDH value
                uint8_t ecdh_value[32]; //pre-master secret (pms)
                status = atcab_ecdh_enc(private_key_slot, (const uint8_t*)&key_store[index],
(uint8_t*)&ecdh_value,
                                        (uint8_t*)&transport_key, transport_key_slot);

                CHECK_STATUS(status);
                printf("ECDH Value\r\n");
                printf("Remote Pubic Key * Host Private Key = \r\n");
                print_bytes((uint8_t*)&ecdh_value, 32);printf("\r\n");

        } else {
                printf("Remote Pubic Key not trusted\r\n");print_bytes((uint8_t*)&remote_pubk, 64);


        }
```

Step 7: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

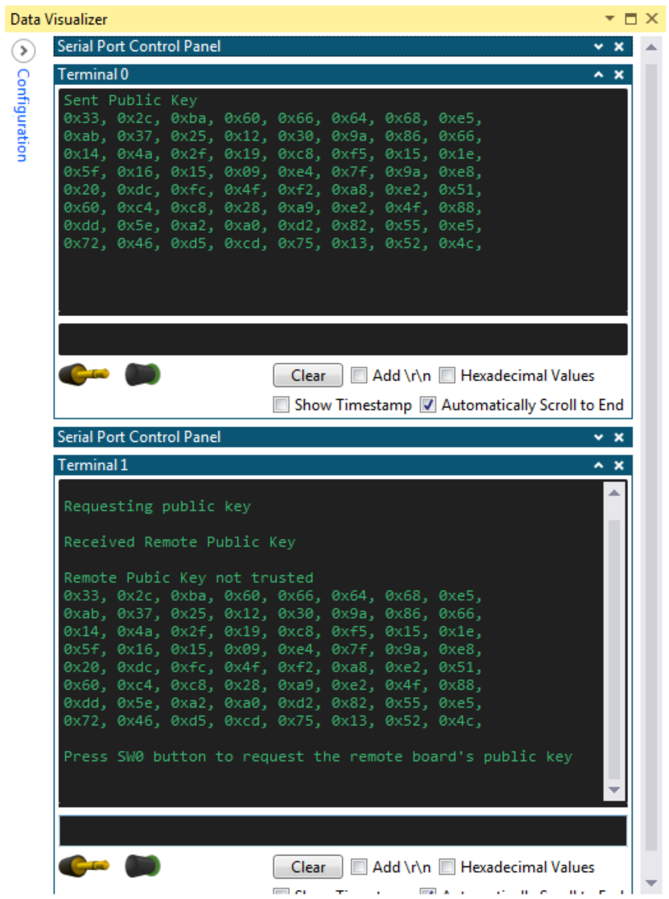
6.4.8 Program Device B

Step 1: Select the second SAM D21 Xplained Pro board using the steps in Section 6.4.2.

Step 2: On the Studio 7 menu bar, click on the “Start without debugging” ▶ button.

Step 3: Press the SW0 push button

Notice that the terminal prints out that the key is not trusted. This is because asymmetric algorithms require the public key be pre-shared in a trusted method which we have not done yet. This sharing is the same concept as is done with symmetric secret keys discussed earlier. Symmetric keys have to be pre-shared but also need to be kept secret. However, public keys do not need the same protection so it’s possible to store them in MCU flash if needed but trusted sharing is still required.



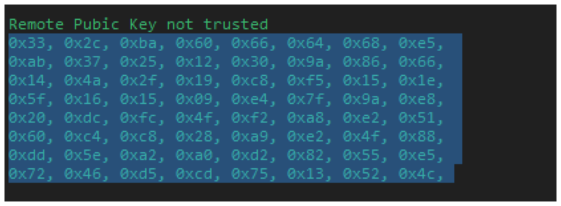
```
Sent Public Key
0x33, 0x2c, 0xba, 0x60, 0x66, 0x64, 0x68, 0xe5,
0xab, 0x37, 0x25, 0x12, 0x30, 0x9a, 0x86, 0x66,
0x14, 0x4a, 0x2f, 0x19, 0xc8, 0xf5, 0x15, 0x1e,
0x5f, 0x16, 0x15, 0x09, 0xe4, 0x7f, 0x9a, 0xe8,
0x20, 0xdc, 0xfc, 0x4f, 0xf2, 0xa8, 0xe2, 0x51,
0x60, 0xc4, 0xc8, 0x28, 0xa9, 0xe2, 0x4f, 0x88,
0xdd, 0x5e, 0xa2, 0xa0, 0xd2, 0x82, 0x55, 0xe5,
0x72, 0x46, 0xd5, 0xcd, 0x75, 0x13, 0x52, 0x4c,
```

```
Requesting public key

Received Remote Public Key

Remote Pubic Key not trusted
0x33, 0x2c, 0xba, 0x60, 0x66, 0x64, 0x68, 0xe5,
0xab, 0x37, 0x25, 0x12, 0x30, 0x9a, 0x86, 0x66,
0x14, 0x4a, 0x2f, 0x19, 0xc8, 0xf5, 0x15, 0x1e,
0x5f, 0x16, 0x15, 0x09, 0xe4, 0x7f, 0x9a, 0xe8,
0x20, 0xdc, 0xfc, 0x4f, 0xf2, 0xa8, 0xe2, 0x51,
0x60, 0xc4, 0xc8, 0x28, 0xa9, 0xe2, 0x4f, 0x88,
0xdd, 0x5e, 0xa2, 0xa0, 0xd2, 0x82, 0x55, 0xe5,
0x72, 0x46, 0xd5, 0xcd, 0x75, 0x13, 0x52, 0x4c,

Press SW0 button to request the remote board's public key
```

6.4.9 Copy and Paste Public Keys to Create Trust of Device A

To add this device’s public key as a trusted key we need to copy the key bytes into the key store variable in the code.

Step 1: Use your mouse to select the public key bytes in the terminal. Press Ctrl+C.



```
Remote Pubic Key not trusted
0x33, 0x2c, 0xba, 0x60, 0x66, 0x64, 0x68, 0xe5,
0xab, 0x37, 0x25, 0x12, 0x30, 0x9a, 0x86, 0x66,
0x14, 0x4a, 0x2f, 0x19, 0xc8, 0xf5, 0x15, 0x1e,
0x5f, 0x16, 0x15, 0x09, 0xe4, 0x7f, 0x9a, 0xe8,
0x20, 0xdc, 0xfc, 0x4f, 0xf2, 0xa8, 0xe2, 0x51,
0x60, 0xc4, 0xc8, 0x28, 0xa9, 0xe2, 0x4f, 0x88,
0xdd, 0x5e, 0xa2, 0xa0, 0xd2, 0x82, 0x55, 0xe5,
0x72, 0x46, 0xd5, 0xcd, 0x75, 0x13, 0x52, 0x4c,
```

Step 2: Click on the `main_symmetric.c` tab.

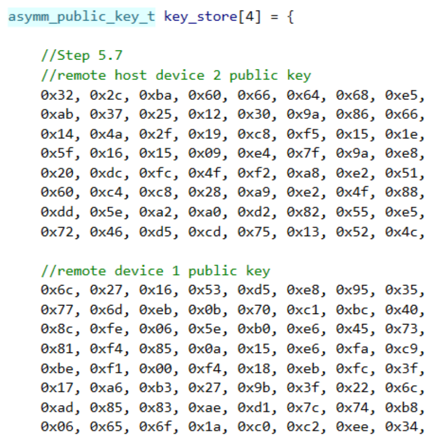
Step 3: Search the source code to find the comment: `//Step 5.7`.

Step 4: Paste the copied key data to variable `key_store[ ]`.

6.4.10 Copy and Paste Public Key to Create Trust of Device B

Step 1: Press the SW0 button on the “other” board.

Step 2: Copy that remote key that appears in your “other” terminal window to the `key_store[ ]`array again at `Step 5.7`. The result should look like below with different values of the keys.



```
asymm_public_key_t key_store[4] = {

    //Step 5.7
    //remote host device 2 public key
    0x32, 0x2c, 0xba, 0x60, 0x66, 0x64, 0x68, 0xe5,
    0xab, 0x37, 0x25, 0x12, 0x30, 0x9a, 0x86, 0x66,
    0x14, 0x4a, 0x2f, 0x19, 0xc8, 0xf5, 0x15, 0x1e,
    0x5f, 0x16, 0x15, 0x09, 0xe4, 0x7f, 0x9a, 0xe8,
    0x20, 0xdc, 0xfc, 0x4f, 0xf2, 0xa8, 0xe2, 0x51,
    0x60, 0xc4, 0xc8, 0x28, 0xa9, 0xe2, 0x4f, 0x88,
    0xdd, 0x5e, 0xa2, 0xa0, 0xd2, 0x82, 0x55, 0xe5,
    0x72, 0x46, 0xd5, 0xcd, 0x75, 0x13, 0x52, 0x4c,

    //remote device 1 public key
    0x6c, 0x27, 0x16, 0x53, 0xd5, 0xe8, 0x95, 0x35,
    0x77, 0x6d, 0xeb, 0x0b, 0x70, 0xc1, 0xbc, 0x40,
    0x8c, 0xfe, 0x06, 0x5e, 0xb0, 0xe6, 0x45, 0x73,
    0x81, 0xf4, 0x85, 0x0a, 0x15, 0xe6, 0xfa, 0xc9,
    0xbe, 0xf1, 0x00, 0xf4, 0x18, 0xeb, 0xfc, 0x3f,
    0x17, 0xa6, 0xb3, 0x27, 0x9b, 0x3f, 0x22, 0x6c,
    0xad, 0x85, 0x83, 0xae, 0xd1, 0x7c, 0x74, 0xb8,
    0x06, 0x65, 0x6f, 0x1a, 0xc0, 0xc2, 0xee, 0x34,
```

Both devices' public keys are in our trusted key store. So when either device receives a key from the other it will be able to find that key in the key store.

Step 3: On the Studio 7 menu bar, click on the "Start without debugging" ▷ button.

6.4.11 Program Device B

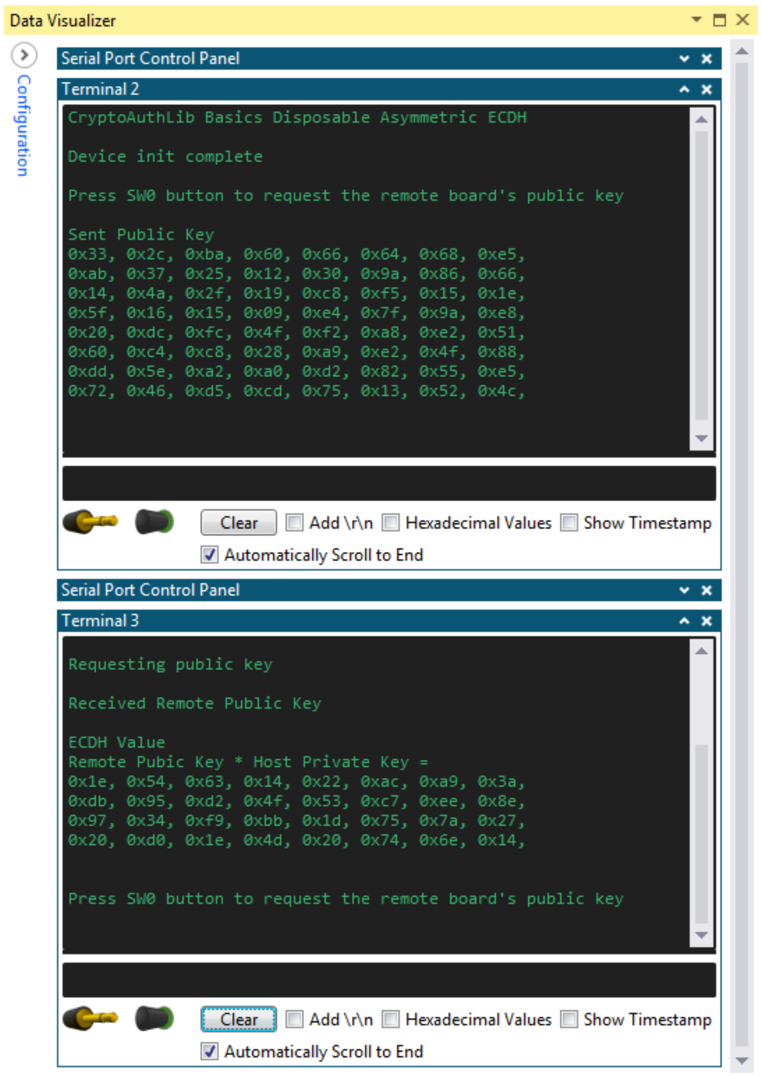
Step 1: Select the "other" SAM D21 Xplained Pro board using the steps in Section 6.4.2.

Step 2: On the Studio 7 menu bar, click on the "Start without debugging" ▷ button.

Step 3: Press the SW0 push button.

Step: 4 Observe the terminal window output.

In the terminal window that corresponds to the board that the button was pressed you should see the output like in terminal 3 below. Notice the ECDH value that was printed to the terminal. The value was calculated by the ECC508A using the remote board's public key which was then supplied as a parameter to the ECDH command. The ECDH command then does an ECC multiply with the internally protected private key in slot 4 (which is unique for every ECC508A) to reach the final ECDH share secret.

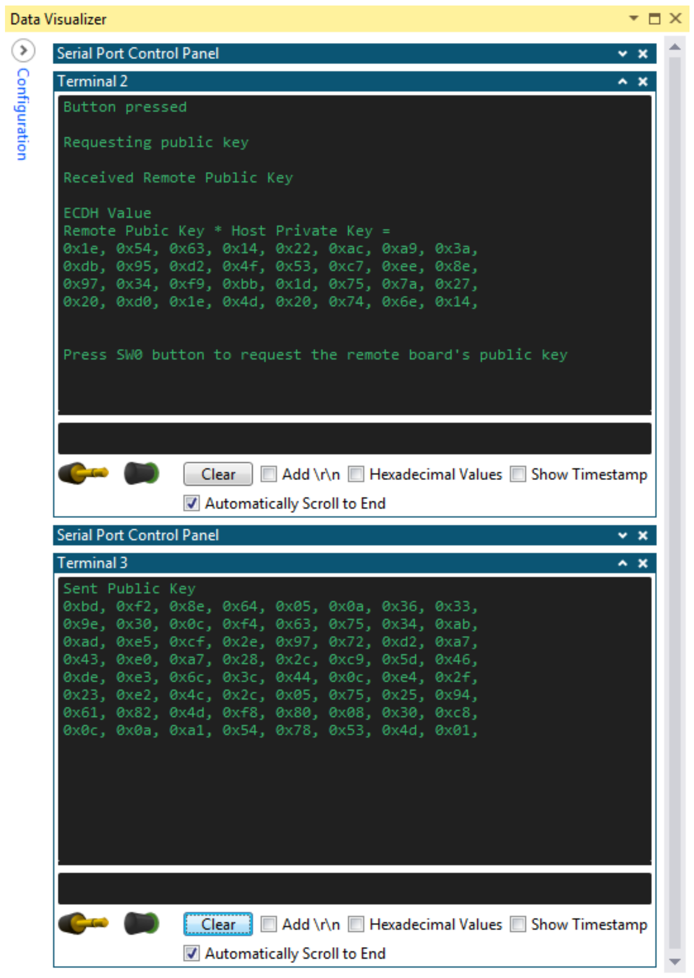


```
CryptoAuthLib Basics Disposable Asymmetric ECDH

Device init complete

Press SW0 button to request the remote board's public key

Sent Public Key
0x33, 0x2c, 0xba, 0x60, 0x66, 0x64, 0x68, 0xe5,
0xab, 0x37, 0x25, 0x12, 0x30, 0x9a, 0x86, 0x66,
0x14, 0x4a, 0x2f, 0x19, 0xc8, 0xf5, 0x15, 0x1e,
0x5f, 0x16, 0x15, 0x09, 0xe4, 0x7f, 0x9a, 0xe8,
0x20, 0xdc, 0xfc, 0x4f, 0xf2, 0xa8, 0xe2, 0x51,
0x60, 0xc4, 0xc8, 0x28, 0xa9, 0xe2, 0x4f, 0x88,
0xdd, 0x5e, 0xa2, 0xa0, 0xd2, 0x82, 0x55, 0xe5,
0x72, 0x46, 0xd5, 0xcd, 0x75, 0x13, 0x52, 0x4c,
```

```
Requesting public key

Received Remote Public Key

ECDH Value
Remote Pubic Key * Host Private Key =
0x1e, 0x54, 0x63, 0x14, 0x22, 0xac, 0xa9, 0x3a,
0xdb, 0x95, 0xd2, 0x4f, 0x53, 0xc7, 0xee, 0x8e,
0x97, 0x34, 0xf9, 0xbb, 0x1d, 0x75, 0x7a, 0x27,
0x20, 0xd0, 0x1e, 0x4d, 0x20, 0x74, 0x6e, 0x14,


Press SW0 button to request the remote board's public key
```

Step 5: Press the SW0 push button on the other board and observe the terminal window.

Notice the same ECDH value is printed to the terminal by the second board when the first board’s public key is used.



```
Button pressed

Requesting public key

Received Remote Public Key

ECDH Value
Remote Pubic Key * Host Private Key =
0x1e, 0x54, 0x63, 0x14, 0x22, 0xac, 0xa9, 0x3a,
0xdb, 0x95, 0xd2, 0x4f, 0x53, 0xc7, 0xee, 0x8e,
0x97, 0x34, 0xf9, 0xbb, 0x1d, 0x75, 0x7a, 0x27,
0x20, 0xd0, 0x1e, 0x4d, 0x20, 0x74, 0x6e, 0x14,


Press SW0 button to request the remote board's public key
```

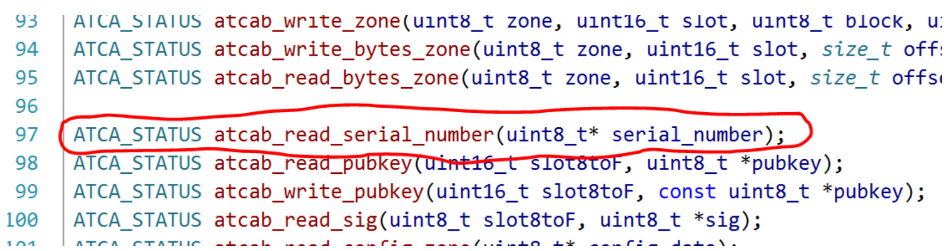
```
Sent Public Key
0xbd, 0xf2, 0x8e, 0x64, 0x05, 0x0a, 0x36, 0x33,
0x9e, 0x30, 0x0c, 0xf4, 0x63, 0x75, 0x34, 0xab,
0xad, 0xe5, 0xcf, 0x2e, 0x97, 0x72, 0xd2, 0xa7,
0x43, 0xe0, 0xa7, 0x28, 0x2c, 0xc9, 0x5d, 0x46,
0xde, 0xe3, 0x6c, 0x3c, 0x44, 0x0c, 0xe4, 0x2f,
0x23, 0xe2, 0x4c, 0x2c, 0x05, 0x75, 0x25, 0x94,
0x61, 0x82, 0x4d, 0xf8, 0x80, 0x08, 0x30, 0xc8,
0x0c, 0x0a, 0xa1, 0x54, 0x78, 0x53, 0x4d, 0x01,
```

6.4.12 Adding Additional Keys to the Key Store

Find a classmate that is at this final step. With each of you using your own computer and terminal, connect one of your boards to your classmate's board using the jumper wires (unplug them from your second board). Follow the previous steps to add your classmate's board's public key to your key store. Take turns pressing the SW0 button. Again notice the same ECDH value is calculated by each other's board. In a final application the ECDH value can be further derived and can be used as a shared symmetric encryption key to exchange information.

Appendix A

A.1

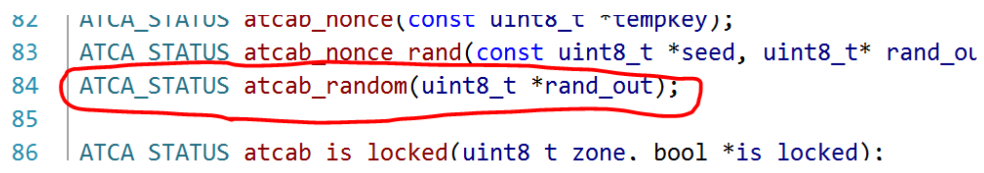


```
ATCA_STATUS atcab_write_zone(uint8_t zone, uint16_t slot, uint8_t block, u
ATCA_STATUS atcab_write_bytes_zone(uint8_t zone, uint16_t slot, size_t off
ATCA_STATUS atcab_read_bytes_zone(uint8_t zone, uint16_t slot, size_t offs

ATCA_STATUS atcab_read_serial_number(uint8_t* serial_number);
ATCA_STATUS atcab_read_pubkey(uint16_t slot8toF, uint8_t *pubkey);
ATCA_STATUS atcab_write_pubkey(uint16_t slot8toF, const uint8_t *pubkey);
ATCA_STATUS atcab_read_sig(uint8_t slot8toF, uint8_t *sig);
ATCA_STATUS atcab_read_config_zone(uint8_t* config_data);
```

Specific code to be included at “//step 1.4:

```
status = atcab_read_serial_number(&serial_number);
```

A.2

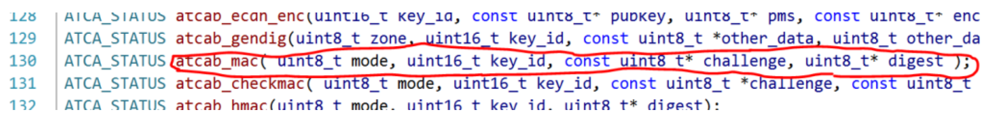


```
ATCA_STATUS atcab_nonce(const uint8_t *tempkey);
ATCA_STATUS atcab_nonce_rand(const uint8_t *seed, uint8_t* rand_ou
ATCA_STATUS atcab_random(uint8_t *rand_out);

ATCA_STATUS atcab_is_locked(uint8_t zone, bool *is_locked);
```

Specific code to be included at “//step 1.7”

```
atcab_random(&nonce);
```

A.3



```
ATCA_STATUS atcab_ecdh_enc(uint16_t key_id, const uint8_t* pubkey, uint8_t* pms, const uint8_t* enc
ATCA_STATUS atcab_gendig(uint8_t zone, uint16_t key_id, const uint8_t *other_data, uint8_t other_da
ATCA_STATUS atcab_mac( uint8_t mode, uint16_t key_id, const uint8_t* challenge, uint8_t* digest );
ATCA_STATUS atcab_checkmac( uint8_t mode, uint16_t key_id, const uint8_t *challenge, const uint8_t
ATCA_STATUS atcab_hmac(uint8_t mode, uint16_t key_id, uint8_t* digest);
```

Specific code to be included at “//step 1.8”

```
atcab_mac(mode, slot, (const uint8_t*)&nonce, (uint8_t*)&mac);
```

A.4

```
otherdata[0] = 0x08; //match to mac command byte opp code

otherdata[1] = mode; // match to mac mode

otherdata[7] = serial_number[4];
otherdata[8] = serial_number[5];
otherdata[9] = serial_number[6];
otherdata[10] = serial_number[7];

otherdata[11] = serial_number[2];
otherdata[12] = serial_number[3];
```

A.5

```
volatile ATCA_STATUS status;
//add init here for ECC508
status = atcab_init( &cfg_ateccx08a_i2c_host ); //enable communication with remote 508


CHECK_STATUS(status);
uint8_t * serial_number = (uint8_t *)&tx_packet->auth.serial_number;
//add read local chip's serial number here
status = atcab_read_serial_number(serial_number);


CHECK_STATUS(status);
printf("My Serial Number\r\n");
print_bytes(serial_number, 9);printf("\r\n");
```

A.6

```
status = atcab_mac(mode, slot, nonce, mac);
```

A.7

```
status = atcab_random(nonce); //generate new nonce for remote device
```

A.8

```
status = atcab_checkmac(mode,
                     slot,
                     nonce,
                     mac,
                     (const uint8_t*)&otherdata);
```

Appendix B

B.1 Terms

Host Device or System – The reference device or devices that are the default trusted system in a specific reference frame

Remote Device or System – A device or devices that connect to a host system in order to prove their authenticity

MAC – Message Authentication Code, a value used to give a level of proof that a message was sent by a specific device or group of devices

Nonce – Number Used Once, a value that is use only once in a given context.

Secure Hash – An algorithm that compresses any amount of input data to a probabilistically unique* fixed size output. The compression results in the loss of information and is then irreversible.

Digest – This is another term for Hash but is used more generally without specific property requirements.

Challenge – A value sent to the device seeking authentication. The device needs to create the expected response to send back to the authenticator. A challenge should typically be unique.

Probabilistically Unique – A value that is unique with a very high degree of probability in it typical use