APPLICATION DATA

TLS Implementation using ATECC508A

Introduction

The goal of this document is to introduce TLS example that has been accomplished by Korean Crypto Team.

The following topics will be covered:

- Porting WolfSSL History
- Role of ECC508 on TLS
- Improvement for a customer and the future

Requirements

- **Hardware**
  - SAMD21 Xplained
  - WINC1500 (F/W V18.0.x)
  - CK101 with ECC508 (or CryptoAuthXplained)
  - Wireless Router (Wi-Fi Access Point)
- **Software**
  - Atmel Studio 6.2
  - ASF V3.33.0 with WINC1500 host driver and I2C driver
  - WolfSSL 3.4.8
  - The latest ATCALIB

Constraints

This example is based on the most popular secure connection like HTTPS.

- A client only authenticates a server's certificate using verify command of ECC508 in current. But as occasion demands, the server shall authenticate the client.
- As certificate of issuer (CA) and signer have not been applied to this example, both the server and the client are regarded as root CA.
- Due to a room issue, WolfSSL is ported for TLS layer instead of OpenSSL.
- "Fae-1h15.ecc508.xml" is used for personalization.
- Fixed message was used to create signature.
- Instead X.509A certitificate, raw signature/public key generated by ECC508 have been used.

1. Porting WolfSSL to the SAMD21

Since the WolfSSL is better than OpenSSL in terms of productivity, size and so on, WolfSSL was determined to TLS of SAMD21 instead of lightening the OpenSSL weight. Of course next target would be OpenSSL.

1.1 Weird Socket API

Curiously socket APIs of WINC1500 doesn't abide by the standard of BSD and POSIX. Due to that, seems like WiFi engineers in Korea have trouble in many ways. In case of doing port third party S/W, This point should never be overlooked. Fortunately socket interface was connected well between porting layer of WolfSSL and socket API of WINC1500 to blocking mode. That is to say, Whenever TLS packet must be sent/received, single thread without RTOS has to wait socket event of host interface. This improved IO Interface than previous Interface of Wifi group may continue to be used for single thread in other projects related with WINC module.

1.2 Memory Map

Basically, SAM D21 is ranging with up to 256 KB Flash and 32 KB of SRAM like illustrated definition on linker script below.

```
/* Memory Spaces Definitions */
MEMORY
{
  rom      (rx)  : ORIGIN = 0x00000000, LENGTH = 0x00040000
  ram      (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00008000
}
```

After finishing port WolfSSL, Whole code occupies 162 KB space of entire flash.

```
                0x000288d4                . = ALIGN (0x4)
                0x000288d4                _efixed = .
                0x000288d4                PROVIDE (__exidx_start, .)
```

Stack size was adjusted up to 6 KB,

```
/* The stack size used by the application. NOTE: you need to adjust according to your application. */
/* STACK_SIZE = DEFINED(STACK_SIZE) ? STACK_SIZE : DEFINED(__stack_size__) ? __stack_size__ : 0x2000; */
STACK_SIZE = 0x1800;
```

Relocated area (initialized variable) and BSS area (non-initialized variable) take 7 KB.

```
                0x20001c84                _ebss = .
                0x20001c84                _ezero = .

.stack          0x20001c84      0x1804 load address 0x0002a558
                0x20001c88                . = ALIGN (0x8)
 *fill*         0x20001c84        0x4
                0x20001c88                _sstack = .
                0x20003488                . = (. + STACK_SIZE)
 *fill*         0x20001c88      0x1800
                0x20003488                . = ALIGN (0x8)
                0x20003488                _estack = .
                0x20003488                . = ALIGN (0x4)
                0x20003488                _end = .
```

Remaining 19 KB is used to allocate dynamic heap (ex, malloc).

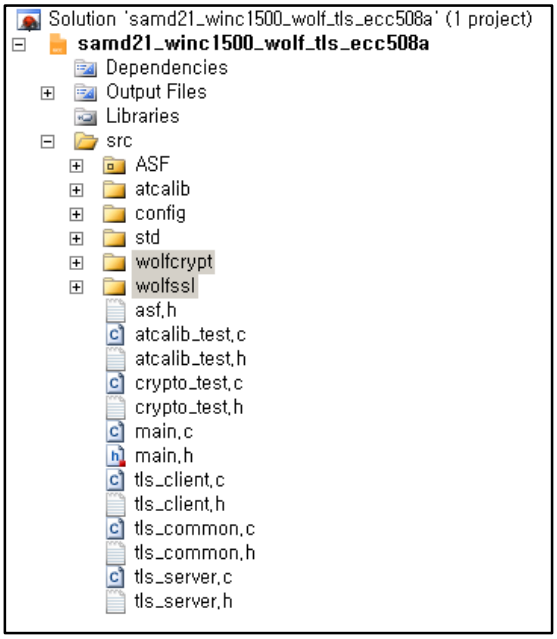
Officially, WolfSSL demands 34 KB heap, however our code withstands only with 19 KB by optimizing the memory map/code. Despite the lightest TLS (WolfSSL) is ported, In fact, D21 is inappropriate for edge node due to poor RAM space. SAM4S that features up to 2 MB of Flash and 160 KB of SRAM is ideal candidate to take charge of edge node. As OpenSSL needs flash up to 3 MB at least in case of using compiled static library, If needless code will be stripped out and If compile issue will not occurred, Probably porting OpenSSL to SAM4S is possible.

1.3 Porting Progress and Source Tree

TLS source codes are located in **wolfcrypt** directory and header files are located in **wolfssl** directory.

The latest V3.4.8 of WolfSSL has been integrated into this project. In order to communicate with WolfSSL, **socket** of WINC1500 and **tls_common** is playing the communication role. The **tls_common** is also connected to the ATCALIB in order to execute a variety of commands of ECC508.

If **WOLFSSL_CRYPTO_TEST** is enabled, **crypto_test** is executed and If **ECC508_ATCALIB_TEST** is enabled, **atcalib_test** is executed. If **“ECC508_ATCALIB_TEST”** is enabled, **“atcalib_test”** is executed. Both **tls_server** and **tls_client** take charge of TLS application that read/write application data.



1.4 Feature Configurations

A few macros which is defined in **"settings.h"** are used to support a variety of features. Server and client mode are optionally enabled to support **TLS_ECDH_ECDSA_AES128_GCM_SHA256** by using one project.

Server mode	Client mode

`#if defined(ATCAECC_SUPPORTED_WOLFSSL)`
`  #define SIZEOF_LONG_LONG 8`
`  #define SINGLE_THREADED`
`  #define HAVE_ECC`
`  #define HAVE_ECC_ENCRYPT`
`  #define HAVE_AESGCM`
`  #define NO_FILESYSTEM`
`  #define NO_PSK`
`  #define NO_OLD_TLS`
`  #define NO_WRITEV`
`  #define NO_WOLFSSL_DIR`
`  #define NO_DEV_RANDOM`
`  #define NO_RSA`
`  #define WOLFSSL_USER_IO`
`  #define DEBUG_WOLFSSL`
`  #define SHOW_SECRETS`
`  #define ECC508_SUPPORT_ECDH_ECDSA`
`  #define ECC508_MEM_DEBUG`
`  #define NO_WOLFSSL_CLIENT`
`//  #define NO_WOLFSSL_SERVER`
`//  #define WOLFSSL_CRYPTO_TEST`
`//  #define ECC508_ECDH_ECDSA_TEST`
`//  #define ECC508_ATCALIB_TEST`
`#endif`
	`#if defined(ATCAECC_SUPPORTED_WOLFSSL)`
	`  #define SIZEOF_LONG_LONG 8`
	`  #define SINGLE_THREADED`
	`  #define HAVE_ECC`
	`  #define HAVE_ECC_ENCRYPT`
	`  #define HAVE_AESGCM`
	`  #define NO_FILESYSTEM`
	`  #define NO_PSK`
	`  #define NO_OLD_TLS`
	`  #define NO_WRITEV`
	`  #define NO_WOLFSSL_DIR`
	`  #define NO_DEV_RANDOM`
	`  #define NO_RSA`
	`  #define WOLFSSL_USER_IO`
	`  #define DEBUG_WOLFSSL`
	`  #define SHOW_SECRETS`
	`  #define ECC508_SUPPORT_ECDH_ECDSA`
	`  #define ECC508_MEM_DEBUG`
	`//  #define NO_WOLFSSL_CLIENT`
	`  #define NO_WOLFSSL_SERVER`
	`//  #define WOLFSSL_CRYPTO_TEST`
	`//  #define ECC508_ECDH_ECDSA_TEST`
	`//  #define ECC508_ATCALIB_TEST`
	`#endif`

1.4.1 ATCAECC_SUPPORTED_WOLFSSL

Is a main feature to support "WolfSSL" and "ECC508A", this macro should be defined within a GCC/GNC make file by using tool chain menu of Atmel Studio.

1.4.2 ECC508_ATCALIB_TEST

Is enabled to make sure commands of "atcalib" as needed. This macro is connected with "atcalib_test.c".

1.4.3 WOLFSSL_CRYPTO_TEST

Can be enabled if cipher tests are required. This macro is connected with "crypto_test.c".

1.4.4 NO_WOLFSSL_CLIENT

Is activated to a TLS server, if this macro is defined on "settings.h". Either of two features must be turned off.

1.4.5 NO_WOLFSSL_SERVER

Is activated to a TLS client, if this macro is defined on "settings.h". Either of two features must be turned off.

1.4.6 ECC508_SUPPORT_ECDH_ECDSA

Is defined to enable to ECC508 certificate including ECDSA public key, ECDH public key and signature, Instead X.509A certificate.

1.4.7 ECC508_MEM_DEBUG

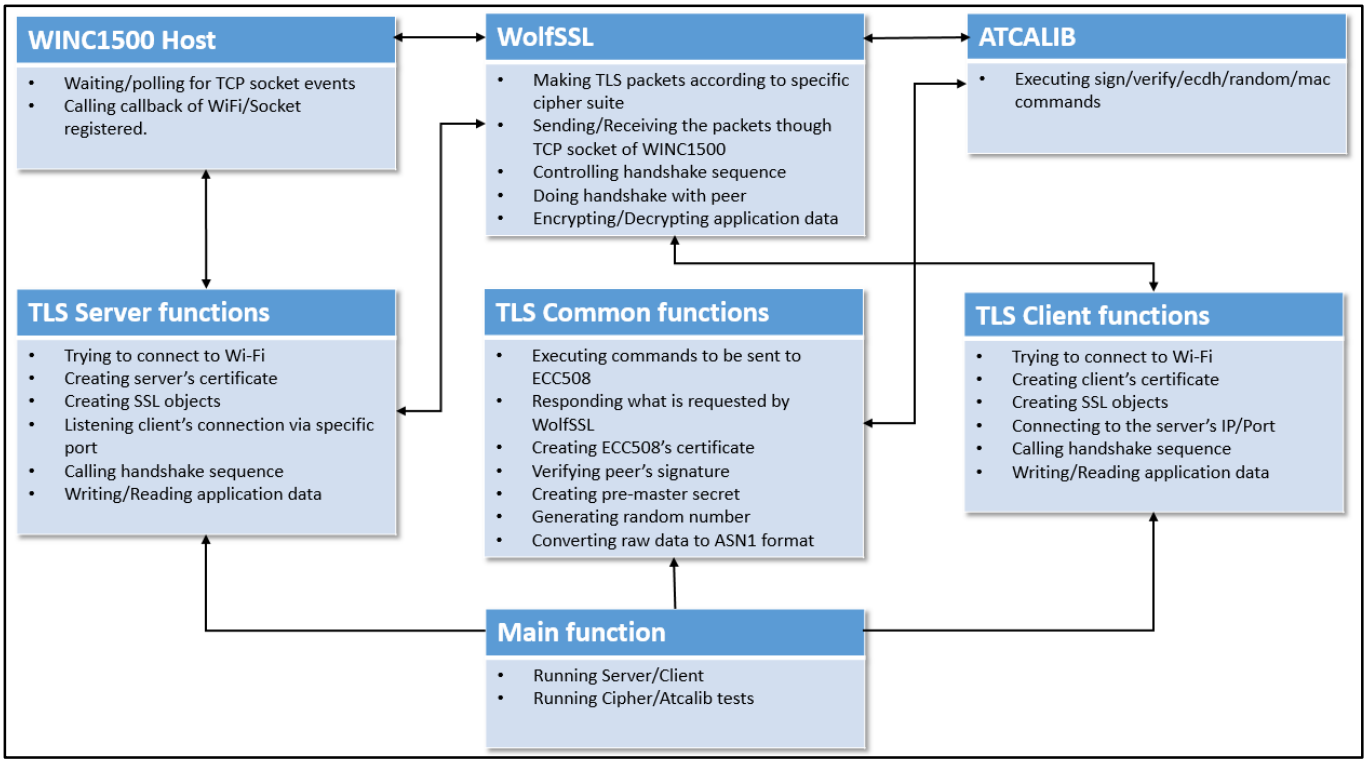
Can be defined when TLS packets and buffer data are needed for debugging.

1.4.8 Other feature wrapped by ATCAECC_SUPPORTED_WOLFSSL

Have to be defined in order to support "ECDH-ECDSA-AES128-GCM-SHA256" cipher suite.

2. Software Architecture

Each of block is interfaced each other on occasion demands. Diagram below shows each block is connected with each other a little complecatedly.



2.1 Main Function (main.c)

Does make decision whether which applications (server/client/tests) will be executed according to option be configured in “settings.h”.

2.2 TLS SERVER (tls_server.c)

Tries to acquire virtual IP from access point and creates own ECC508 certificate. TLS SERVER then waits for client’s connection. If server takes client’s connection, TLS handshake session is started with client. After the session will be finished by “Finish” packet, The TLS SERVER exchanges application data. The TLS SERVER also controls socket status to match TCP sequence.

2.3 TLS CLIENT (tls_client.c)

Tries to acquire virtual IP from access point and creates own ECC508 certificate as well. The TLS CLIENT then tries to connect to server with server’s IP/Port. When the TLS CLIENT may receive connection message from WINC1500, That is to say, TCP connection is established with server, Client starts handshake session with server. If handshake session is done, application data are exchanged accordint to SEND/RECEIVE APIs.

2.4 TLS Common (tls_common.c)

Communicates with ATCALIB and WolfSSL inside. Signature is generated based on fixed message which TLS client/server pass. Public key also is generated using same slot what signature is generated. ECDH public key that will be transferred to peer is created so that peer’s ECC computes pre-master secret. ECDSA verification also is covered on the TLS COMMON. Both server and client are able to verify using ECC

command or WolfSSL APIs to authenticate peer's ECC508 certificate. Because the TLS COMMON supports conversion function from ECC508's RAW signature/key to ASN.1. If needed, WolfSSL API can be used instead of ECC508's verification command. If we will have to follow X.509 format, Signature, keys and other data that are created/read by ECC508 should be encoded to ASN.1 and DER format. Anyway own ECC508 certificate is verified by opponent ECC device in current. Interesting thing is that ECC508 computes Key creation/Signing/Verifycation more quickly than WolfSSL APIs. ECC device have greate advantage in terms of performance.

2.5 WINC1500 Host

Communicates with TCP server/client and WolfSSL. Actually since TCP stack is located in WINC Module, TLS packet that is made by Host MCU is delivered through WINC1500 Host interface.

2.6 WolfSSL

Is actually responsible for all of TLS. Sequence of handshake is processed and Cipher algorithms is computed. Because WolfSSL have small, lightweight and fast merits, Seems like that it is proper for embedded platform, that is to say, Porting is more convenient than other SSL solution. Nevertheless, Problem is that it is not popular to customers. Also we have to keep GPL V2.

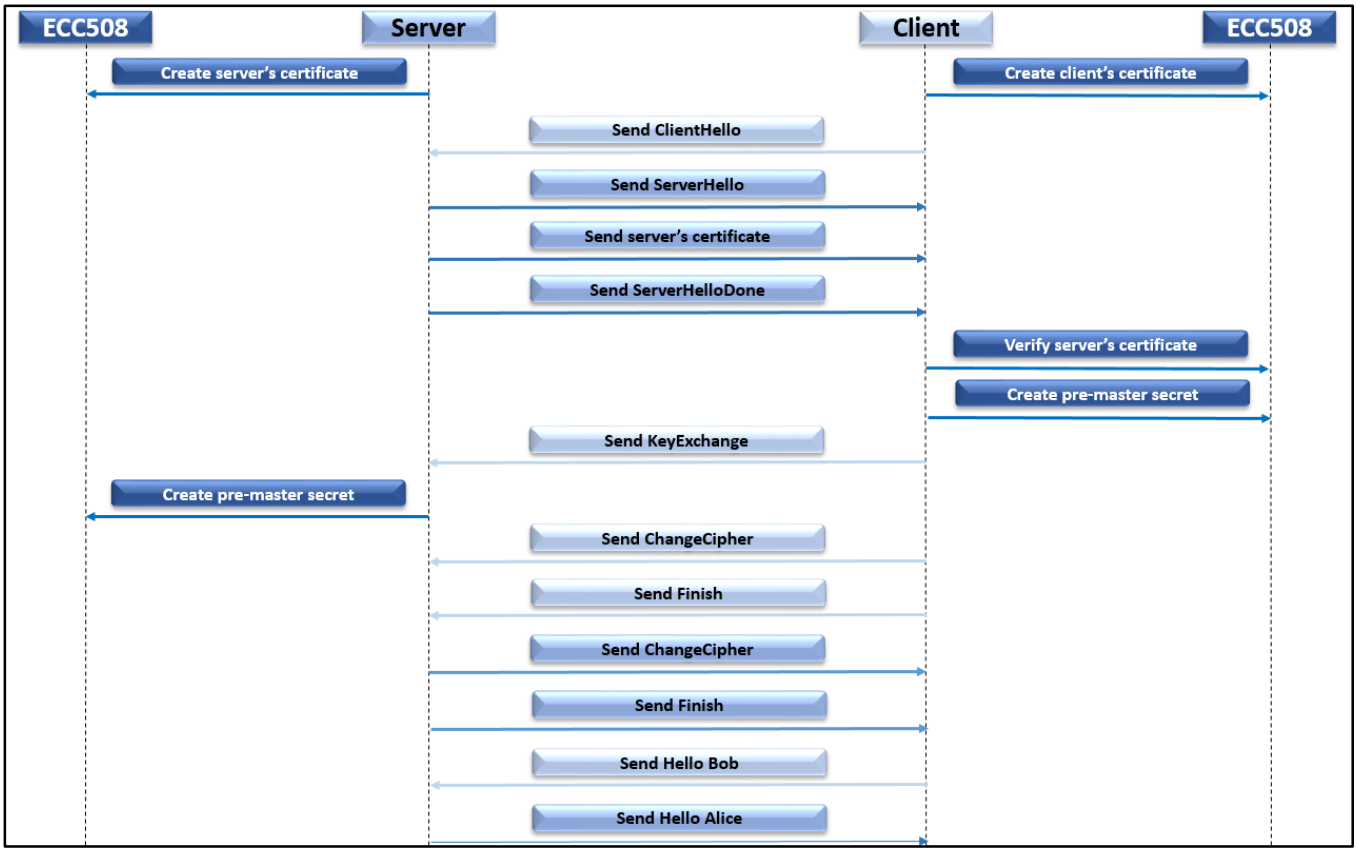
2.7 ATCALIB and Personalization

Typically our solution communicates with the TLS COMMON to send/receive command to ECC508 device. Both server and client are using slot 0 to generate a signature that is required for authentication and slot 4 to create a public key that is required for pre-master key agreement according to the configuration below. If trust of chain (for signer and issuer) would be needed, other slots will be used for the trust of chain.

Slot	Role	Name	SlotConfig	KeyConfig	Notes
0	Asymmetric Authentication	ECDSA	8320	3300	Primary Authentication
4	Key Agreement-2	ECDH	8420	3300	Test Key Agreement slot Returns PMS

3. TLS Communication

This chapter actually show how server and client communicate each other.



3.1 Certificate generation for each of peers

As this example doesn't support X.509A certificate yet, Just signature that is generated by fixed number and private key of slot 0 are used to make ECC508 certificate. Since a WiFi has been already connected and a server/client socket has been assigned, each of peers generate own certificate.

3.1.1 Certificate structure

```
typedef struct certificate_of_ecc508 {
    uint8_t msg[P256_MSG_LEN];
    uint8_t pub_key[P256_ECDSA_PUB_LEN];
    uint8_t sig[P256_SIG_LEN];
    uint8_t ecdh_pub[P256_ECDH_PUB_LEN];
} t_ecc508_cert;
```

1. Message that takes 32 bytes is composed of fixed values in current, but can be generated by random command later whenever random challenge is required.
2. Signature that is made up of R,S component is created using slot 0 with fixed/random challenge. It takes 64 bytes
3. Public key is generated by genkey command corresponding to private key of slot 0. Both server and client are able to use this key to verify signature received from peer on authentication step.

4. In order to create premaster secret on key exchange step, public key that is required by peer is created using genkey command.
5. Log below shows the certificate buffer that is generated by ECC508.

```
ECC508's message :
 54 79 70 65 20 44 61 74 61 20 74 6f 20 53 69 67 6e 20 48 65 72 65 0 0 0 0 0 0 0
 0 0 0
ECC508's signature :
 a2 4b 2c 42 7e a2 55 94 77 10 88 1b c1 30 1f 10 2f 1 66 59 f1 ff 5a 13 43 d7 b8
 4d 23 da 90 44 f7 2f cf 3a 4f 68 83 41 30 d0 b4 e3 7b 3 c 11 b1 33 3c 27 fa e0
15 7e a9 59 89 fe 12 e1 af b1
ECC508's ECDSA public key :
 db 9a da 10 14 9f f9 8c 93 b7 5d 52 83 ea 49 a4 58 6a 3e 5c 84 c9 8f 5c 81 9f 5
f 3c 28 f1 96 f2 1b b2 a2 87 32 2 b 25 91 b8 32 5f f5 eb ad 6e ec 46 c2 62 85 b4
 5a 49 98 a4 36 99 3b 66 f8 a4
ECC508's ECDH public key :
 c7 a5 ba f0 d5 63 31 8d c 7c 55 12 ad 34 a8 a3 e1 1a 71 2c 6b b6 a2 fc 2c e5 61
 d8 c6 64 fa 6f b f6 38 74 33 36 d1 77 c0 5a 69 21 3b 82 db d5 5e fe e 6b 36 64
73 45 44 38 fa fd b1 12 61 6b
```

3.2 Client Hello

When server waits for client's connection through specific port, Client sends Hello packets that is made up of TLS format in order to inform suggested cipher suites, random number, compression and protocol version.

3.2.1 Packet Information of Client Hello

Random bytes can be generated by random command of ECC, but random number below (0x02 to 0x17, 32 bytes) is computed by API of WolfSSL. If ECC508's random command is required, It may quickly be connected to the WolfSSL. This random number will be used by WolfSSL to create master secret later.

```
Sent Packets :
 16 3 3 0 39 1 0 0 35 3 3 2 1b 87 81 b8 fb d1 60 20 78 f8 ad 89 f6 3f e1 c8 58 e
4 24 cf cc c3 c9 fd 2c e3 51 da e5 d1 17 0 0 2 c0 2d 1 0 0 a 0 d 0 6 0 4 4 3 2 3
```

Following table explains what the meaining of each packet with ClientHello packet

Index	Main title	Sub title	Value	Meaning
0	TLS Record header	Type	0x16	Indicates handshake, for reference, 0x17 means application data, 0x15 means alert and 0x14 means change cipher spec
1		Major version	0x03	Indicates SSL V3
2		Minor version	0x03	Indicates TLS V1.2
3		Body length	0x00	TLS body takes 57 bytes
4			0x39	
5	Handshake header	Type	0x01	Indicates ClientHello, for reference, 0x02 means ServerHello, 0x0b means certificate, 0x10 means Client Key Exchange, 0x14 means Finish
6		Length	0x00	Indicates length of ClientHello body
7			0x00	
8			0x35	
9	Version	Major version	0x03	Indicates SSL V3

10		Minor version	0x03	Indicates TLS V1.2
11	Random number		0x02 to 0x17	will be used for server to make master secret
42				
43	Session ID		0x00	Indicates identification of client
44	Cipher suite	Length	0x00	means length of client's cipher suite
45			0x02	
46		First cipher	0xC0	means ECC byte
47		Second cipher	0x2D	means TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
48	Compression	Length	0x01	indicate size of compression method
49		Method	0x00	client does not support compression
50	Extention	Length	0x00	SKIP
51			0x0A	
52		Signature algorithm	0x00	
53			0x0D	
54		Sig algorithm length	0x00	
55			0x06	
56		Hash algorithm size	0x00	
57			0x04	
58		MAC algorithm	0x04	
59			0x03	
60			0x02	
61			0x03	

3.3 Server Hello

Likewise, server also sends Hello packet having cipher suites that will be used for TLS session.

The ServerHello packets similar to ClientHello.

3.4 Certificate

Server sends own certificate that is generated previously to the client. Certificate that is generated by ECC508 is wrapped around TLS header. Body is made up of length + fixed message + length + ECDSA public key + length + signature + length + ECDH public key and is same with. Record header, 0x16, points out handshake type but TLS header, 0x0b, means a certificate unlike client/server hello with 0x01 and 0x02.

```
Sent Packets :
 16 3 3 0 ee b 0 0 ea 0 0 e7 0 0 e4 20 54 79 70 65 20 44 61 74 61 20 74 6f 20 53
 69 67 6e 20 48 65 72 65 0 0 0 0 0 0 0 0 0 0 40 d e6 36 a 76 c6 33 a7 49 42 65 1
4 a1 41 2e 96 70 68 e5 6c e9 f0 7b 5e 34 67 70 17 5e fa 17 5c cc 38 36 d5 a8 d d
6 ab 49 2f 92 28 20 bc ae f d1 fb ce 93 fb 66 32 fb c7 fd c8 cd 5f 24 4e b3 40 f
5 53 c 72 9c 58 89 e9 b9 d3 20 b1 ba 33 e1 27 ef 86 63 a0 97 c8 7c 25 d1 5c ac 3
7 b 57 61 ee 8e a6 93 11 d6 ef 9e b9 d6 3a c 1d d5 2e 2f f 47 14 6 b9 c3 5d 37 c
3 c0 b5 55 28 b3 b7 56 fc 40 90 97 9e 23 6a dd 50 41 69 35 ff aa c2 f3 3e 43 fd
9 8 10 3d 68 ec c6 83 f0 a0 98 4 31 c4 77 ff 77 74 a5 f7 c 14 64 a8 af aa b3 db
ac f3 67 76 b8 c3 9e e8 3b e4 3c c2 a7 a4 39 cf 9a 5 4
```

3.5 Certificate Authentication

Client authenticates a certificate received from server. As typically browser already has a CA's certificate in case of HTTPS Web, the browser is able to authenticate certificate chain. However since ECC508 has been not configured in accordance with certificate chain, this certificate plays a CA role. Client is able to authenticate server's ECC508 certificate with own ECC device.

```
Reverted message :
 54 79 70 65 20 44 61 74 61 20 74 6f 20 53 69 67 6e 20 48 65 72 65 0 0 0 0 0 0 0
 0 0 0
Reverted ecdsaKey :
 d e6 36 a 76 c6 33 a7 49 42 65 14 a1 41 2e 96 70 68 e5 6c e9 f0 7b 5e 34 67 70
17 5e fa 17 5c cc 38 36 d5 a8 d d6 ab 49 2f 92 28 20 bc ae f d1 fb ce 93 fb 66 3
2 fb c7 fd c8 cd 5f 24 4e b3
Reverted signature :
 f5 53 c 72 9c 58 89 e9 b9 d3 20 b1 ba 33 e1 27 ef 86 63 a0 97 c8 7c 25 d1 5c ac
 37 b 57 61 ee 8e a6 93 11 d6 ef 9e b9 d6 3a c 1d d5 2e 2f f 47 14 6 b9 c3 5d 37
 c3 c0 b5 55 28 b3 b7 56 fc
Reverted ecdhKey :
 90 97 9e 23 6a dd 50 41 69 35 ff aa c2 f3 3e 43 fd 9 8 10 3d 68 ec c6 83 f0 a0
98 4 31 c4 77 ff 77 74 a5 f7 c 14 64 a8 af aa b3 db ac f3 67 76 b8 c3 9e e8 3b e
4 3c c2 a7 a4 39 cf 9a 5 4
ECC508's certificate size : 228
Keep ECDH Key
**** Congratulation Verified ****
serverState = SERVER_CERT_COMPLETE
Verified Peer's cert
```

Verification log is accurately in agreement with code below, as verification command returns a value of zero.

```
ret = atca_send_verify_extern(ecdsaKey, message, signature, result);

if (result[1] == 0) {
    *verified = 1;
    if (ssl->peerEcdhKey != NULL) {
        printf("Keep ECDH KeyWrWn");
        memcpy(ssl->peerEcdhKey, ecdhKey, P256_ECDH_PUB_LEN);
    }
    printf("**** Congratulation Verified ****WrWn");
}
else {
    *verified = 0;
    printf("**** Check Peer's certificate ****WrWn");
}

atca_sleep();
return ret;
```

After that, client keeps server's ECDH public key on maintaining within own buffer to create premaster secret later.

3.6 Server Hello Done

Notify that server has completed this phase of the TLS handshake.

3.7 Key Exchange

Client creates 32 bytes premaster secret using public key within server's certificate, Client then sends own ECDH public key to the server in order for server to make premaster secret with own ECC device. Client also

creates master secret (session/symmetric key) base on premaster secret/random number to encrypt/decrypt application data.

```
peer's ecdh :
 90 97 9e 23 6a dd 50 41 69 35 ff aa c2 f3 3e 43 fd 9 8 10 3d 68 ec c6 83 f0 a0
98 4 31 c4 77 ff 77 74 a5 f7 c 14 64 a8 af aa b3 db ac f3 67 76 b8 c3 9e e8 3b e
4 3c c2 a7 a4 39 cf 9a 5 4
Pre-master Secret :
 57 6b b0 a9 be b4 a3 96 8 4d 20 af 15 55 13 90 3a ab 6 92 d4 8e d 6f 67 e4 5e 5
7 ba 58 90 3f
Client's premaster secret is generated successfully
Bring client's ecdh public key
growing output buffer

sendSz : 73
send own ecdh key : 73
SendBuffered length : 73
send_to_WINC1500
Size to be sent : 73
Socket Event : SEND socket event
socket callback : send success!
Sent Packets :
 16 3 3 0 44 10 0 0 40 c7 a5 ba f0 d5 63 31 8d c 7c 55 12 ad 34 a8 a3 e1 1a 71 2
c 6b b6 a2 fc 2c e5 61 d8 c6 64 fa 6f b f6 38 74 33 36 d1 77 c0 5a 69 21 3b 82 d
b d5 5e fe e 6b 36 64 73 45 44 38 fa fd b1 12 61 6b
```

When serve receives client's ECDH public key, server also creates premaster secret and master secret with client's ECDH public key. Finally both server and client have same premaster secret from 0x57 to 0x3f. Server also stores master secret to encrypt/decrypt application data.

```
peer's ecdh :
 c7 a5 ba f0 d5 63 31 8d c 7c 55 12 ad 34 a8 a3 e1 1a 71 2c 6b b6 a2 fc 2c e5 61
 d8 c6 64 fa 6f b f6 38 74 33 36 d1 77 c0 5a 69 21 3b 82 db d5 5e fe e 6b 36 64
73 45 44 38 fa fd b1 12 61 6b
Pre-master Secret :
 57 6b b0 a9 be b4 a3 96 8 4d 20 af 15 55 13 90 3a ab 6 92 d4 8e d 6f 67 e4 5e 5
7 ba 58 90 3f
Server's premaster secret is generated successfully
```

```
        #ifdef HAVE_ECC
            case ecc_diffie_hellman_kea:
                {
#ifdef ECC508_SUPPORT_ECDH_ECDSA
                    word32   size = 32;
                    ret = tls_create_premaster_secret(ssl->peerEcdhKey, ssl->arrays->preMasterSecret);
                    if (ret != 0) {
                        printf("tls_create_session_key is failed\r\n");
                        return -1;
                    } else {
                        ssl->arrays->preMasterSz = 32;
                        printf("Client's premaster secret is generated successfully\r\n");
                        if (tls_get_own_ecdh_key(ssl->ownEcdhKey, sizeof(ssl->ownEcdhKey)) != 0) {
                            return -1;
                        } else {
                            encSz = 64;
                            printf("Bring client's ecdh public key\r\n");
                            XMEMCPY(encSecret, ssl->ownEcdhKey, encSz);
                        }
```

3.8 Change Cipher Spec by Client

Client sends ChangeCipherSpec that application data sent by the client will be encrypted.

0x14 means a change cipher spec.

```
Sent Packets :
 14 3 3 0 1 1
```

3.9 Finish by Client

Client sends Finish command including digest to validate commands sent previously.

3.10 Change Cipher Spec by Server

If server gets client's encrypted Finish, the server decrypts client's command except for record header. Server then sends ChangeCipherSpec to the client in order to notify that subsequent data will be encrypted.

3.11 Finish by Server

Server also sends Finish command including digest to validate commands sent previously.

3.12 Hello Bob

Client sends encrypted application data which plain text is "Hello Bob" according to negotiated algorithm.

0x17 means application data.

```
BuildMessage type : 2
Encrypt cipher_algorithm : 8
Encrypt :
 0 0 0 0 0 0 0 1 68 65 6c 6c 6f 20 42 6f 62 21 0 0 0 0 0 0 0 0 0 0 0 fb 28 e4 a2
 22
Encrypt cipher_algorithm : 8
wolfSSL Entering AesGcmEncrypt
SendBuffered length : 39
send_to_WINC1500
Size to be sent : 39
Socket Event : SEND socket event
socket callback : send success!
Sent Packets :
 17 3 3 0 22 0 0 0 0 0 0 0 1 d9 81 ae 51 30 d4 50 9 5 84 1d ae 7b 32 6f 7d e8 17
 88 88 57 6f 8f 89 97 a6
```

If below plain text, hello bob, will be sent out, Server then decrypts received data with master secret.

```
int tls_send_message_to_server(void)
{
    char msg[32] = "hello Bob!";   /* GET may make bigger */
    int msgSz = (int)strlen(msg);

    printf("tls_send_message_to_server!\r\n");

    if (wolfSSL_write(ssl_client, msg, msgSz) != msgSz) {
        err_sys("wolfSSL_write failed");

        return SSL_ERROR_SSL;
    }

    return SSL_SUCCESS;
}
```

If client also receives packets that is encrypted, decrypted message is displayed like log below.

```
Received Packets :
 0 0 0 0 0 0 0 1 a2 47 b 18 0 df e6 e9 1c 67 e4 ae ec 9f 85 aa 54 f1 95 52 fd b6
 52 5e 7 49 e1 5a
loop inputBuffer.length : 36, size : 36, in : 36, inSz : 0
Decrypt cipher_algorithm : 8
Decrypt :
 0 0 0 0 0 0 0 1 a2 47 b 18 0 df e6 e9 1c 67 e4 ae ec 9f 85 aa 54 f1 95 52 fd b6
 52 5e 7 49 e1 5a
wolfSSL Entering AesGcmDecrypt
received record layer msg
Identifier : 23
got app DATA
Shrinking input buffer

wolfSSL Leaving ReceiveData(), return 12
wolfSSL Leaving wolfSSL_read_internal(), return 12
Client received message : hello Alice!, length : 12
tls_shutdown_wolfssl_client_object!
```

Code below displays decrypted sever's message.

```
int tls_receive_message_from_server(void)
{
    int  input;
    char reply[80];

    printf("tls_receive_message_from_server!\r\n");

    input = wolfSSL_read(ssl_client, reply, sizeof(reply)-1);

    if (input > 0) {
        reply[input] = 0;
        printf("Client received message : %s, length : %d\r\n", reply, input);
        return SSL_SUCCESS;
    } else {
        err_sys("wolfSSL_read failed");
        return SSL_ERROR_SSL;
    }
}
```

3.13 Hello Alice

Server also sends "Hello Alice" that is encrypted by master key to the client, Client then decrypts the application to plain text.

4. Improvement Scheme and Conclusion

As this solution has a defects related to SHA, we have to fix it as soon as possible.

4.1 Mutual authentication

As what who authenticates who is determined according to character of WEB, we have to provide for authentication what all of edge node/clould server/gateway ECC508 on board can be verified.

4.2 Trust of Chain

As I'm not sure that how will signing chain be configured in the field, we have to prepare needs of IoT market.

4.3 X.509A Certificate

I don't know if certificate issued by GeoTrust/Verisign should be regarded, but if required, this solution has to support the X.509 certificate to match with ECC508.

4.4 Edge Node Platform

As I mentioned above, Next target is SAM4S. If I2C task of ATCALIB will be comfirmed, TLS codes will be copied SAM4S.

4.5 OpenSSL

This is a big problem. The reason why I cannot use static lilbraries of OpenSSL, I have to remove needless code without compile/link error using SAM4S, However OpenSSL is quite complicated entangled. If integration will be completed with WINC1500, I guess that TLS modification is more convenient than porting OpenSSL. I think that what SAMA5D4 plays a server/host/gateway role with OpenSSL + WILC1000 and SAM4S plays edge node role with WolfSSL + WINC1500 is not bad in my short opinion. No matter what any platform is determined, what OpenSSL must be involved into the platform is important truth.

5. Revision History

Doc. Rev.	Date	Comments
A	05/30/2015	Initial document release for TLS implementation

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