**Qualcomm TrustZone technology for ARM Architecture**

Table of contents

Chapter 1 : Introduction

Chapter 2: Booting sequence

Chapter 3: Build steps

Chapter 4: Flashing steps

Chapter 5:APIs Used in QEE

**Chapter 1**

**Introduction**

"ARM® TrustZone® technology is a system- wide approach to security for a wide array of client and server computing platforms, including handsets, tablets, wearable devices and enterprise systems. Applications enabled by the technology are extremely varied but include payment protection technology, digital rights management, BYOD, and a host of secured enterprise solutions."

Trust zone is a set of security extensions added to ARM processors. It can run 2 operating systems. 1.Secure operating system 2.Normal operating system. Below figure showsbasic ARM Trustzone. Both operating system have the same capabilities and Operate in a separate memory space

Enables a single physical processor core to execute from both the Normal world and the Secure world. Normal world components cannot access secure world resources and secure world can access normal world components.

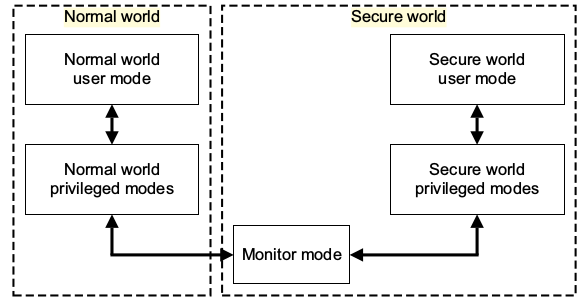
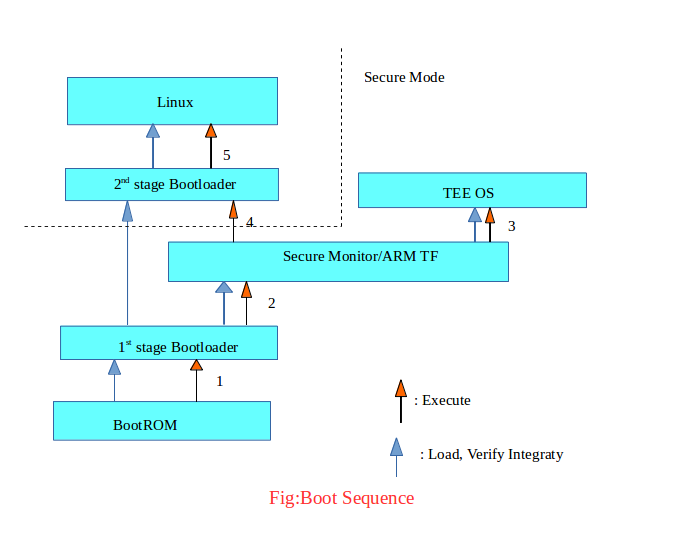


Fig: Modes in an ARM core implementing the Security Extensions

**Chapter 2**

**Booting sequence**

**Bootflow of TEE:**

****

**Chapter 3**

**Build steps**

1. Need Ubuntu 14.04 machine .

2. Build Intialization setup

**Step 1**:Install below packages by using below command.

sudo apt-get install git-core gnupg flex bison gperf build-essential zip curl zlib1g-dev gcc- multilib g++-multilib libc6-dev-i386 lib32ncurses5-dev x11proto-core-dev libx11-dev lib32z-dev ccache libgl1-mesa-dev libxml2-utils xsltproc unzip

**Step 2**:Java-open-JDK7 installation by using below commands.

sudo apt-get install software-properties-commonpython-software-properties

sudo add-apt-repository ppa:openjdk-r/ppa

sudo apt-get update

sudo apt-get install openjdk-7-jdk

Check Java in this path cd /usr/lib/jvm/java-7-openjdk-amd64/bin

**Step 3**: Git Installation

sudo add-apt-repository ppa:git-core/ppa

sudo apt-get update

sudo apt-get install git

**Step 4**:Require perl-5.22.1 packages use below command to install.

sudo apt-get install libxml-parser-perl(some times required perl packages)

**Step 5**: For change default shell should run -- sudo dpkg-reconfigure dash

**Step 6**:SSL Certification

sudo apt-get install purge ca-certificates curl openssh-server

**Step 7**: Other Software

sudo apt-get install meld

sudo apt-get install curl

sudo apt-get install xpad

sudo apt-get install unrar

**Step 8**:Python version should be 2.7.12

**Step 9**:Download python 2.7.12 from open source and extract it follow below commands.

cd /usr/src

sudo wget https://www.python.org/ftp/python/2.7.12/Python-2.7.12.tgz

sudo tar xzf Python-2.7.12.tgz

cd Python-2.7.12

sudo ./configure --enable-optimizations

sudo make altinstall

**Step 10**:For qualtools setup follow below steps.

-->Source code syncing:

1. git clone http://192.168.2.223:1900/vtech/vt3n.git

-->Clone the qualtools from git lab by using below command.

2.git clone http://192.168.2.223:1900/vtech/qualtools.git

-->After cloning qualtools follow below commands

3.cd qualtools

-->Change the permission by using below command.

4.chmod +x qualtools.sh

-->Run the qualtools script in qualtools directory.

5../qualtools.sh

-->move to the directory below by using cd command

6.cd vt3n/LINUX/android

-->Run the envsetup script by using below command.

7.source build/envsetup.sh

8.lunch (Choose a combo by input its number)

1. LF2403\_MSM8909\_512-user

2. LF2403\_MSM8909\_512-userdebug

3. LF2403S\_MSM8909\_512-user

4. LF2403S\_MSM8909\_512-userdebug

5. LF2403N\_MSM8909\_512-user

6. LF2403N\_MSM8909\_512-userdebug

7. LF2403N\_L3\_MSM8909\_512-user

8. LF2403N\_L3\_MSM8909\_512-userdebug

9. F120B\_MSM8909\_512-user

10. F120B\_MSM8909\_512-userdebug

choose a combo:5 or 6 based on mobile model you can choose any combo:1to 10

-->Run make command to build source code.

9. make -j8 (8 is the number of your cpu thread <=>Intel4 core8 threads)

After build is success then follow below steps.

Re-run command as shown below.

10. source build/envsetup.sh

11.lunch (Choose a combo by input its number)

signed build generation:

(pack all build result into a zip file that can be flash after extract)

a. cd vt3n

b. ./pack\_all\_sign.sh (flash onto secboot enabled device)

Unsigned build generation:

a. cd vt3n

b. ./pack\_all\_unsign.sh (flash to non secboot device)

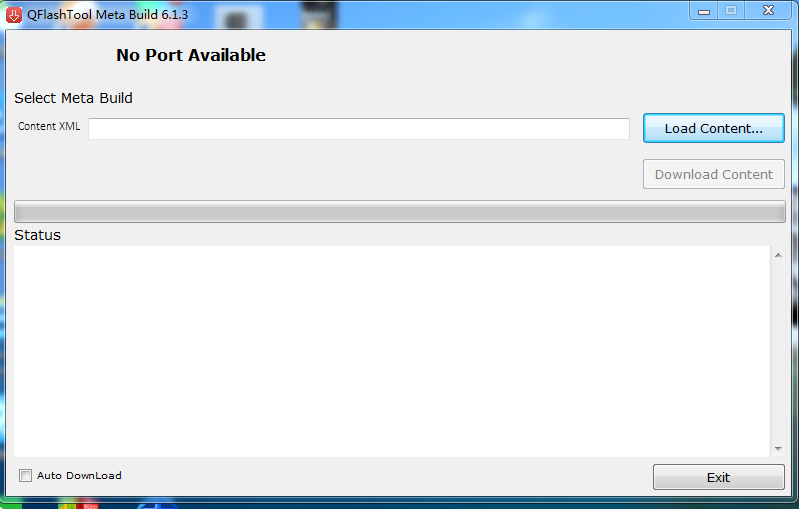
Note: Final Binary generated

@root\_path/pub(vt3n/pub/LYF-LF2403N-000-01-40-121018-i\_userdebug\_f3.zip,LYF-LF2403N-000-01-40-121018-i\_userdebug\_f3\_symbols\_for\_qcap.zip)

**Chapter 4**

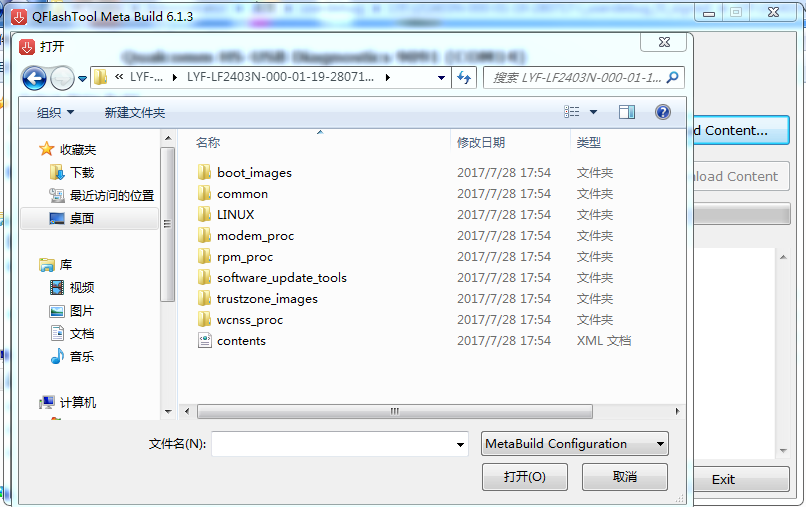
**Flashing LYF Mobile**

1. After build source code we need to transfer images to windows machine and extract the zip file.
2. In windows machine we need to install Qflash tool if it is not available in the machine.
3. Now double click on Qflash tool and open it and connect LYF mobile to system through usb cable .mobile must be in EDL mode.

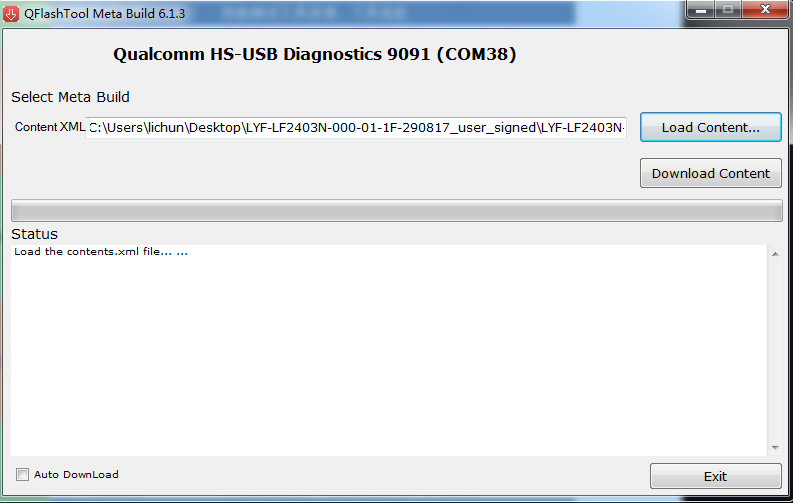


**Fig:Qflash Tool**

1. To take mobile into EDL mode switch of mobile and press volume up and down buttons then mobile will go into EDL mode.
2. Then open image containing folder and select content.xml file from Qflash tool.

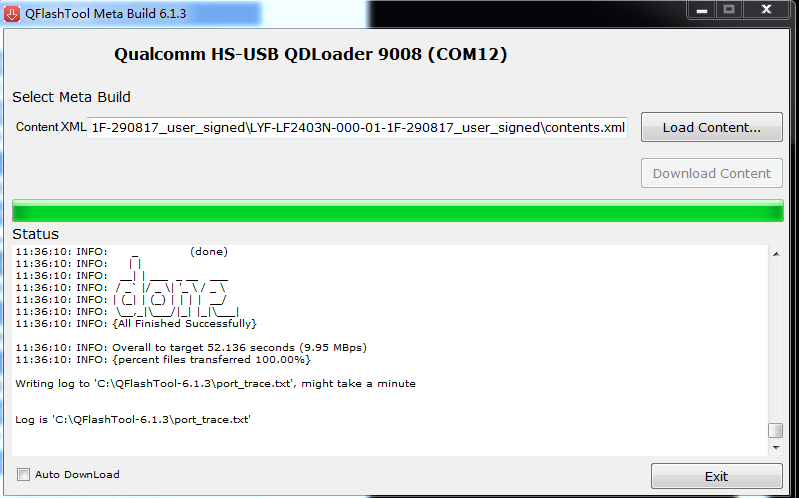


**Fig:Opened LYF mobile Image Foldef from Qflash Tool**



**Fig:Selected contents.xml file in Qflash Tool**

1. Click on download content.
2. After successfully downloaded exit Qflah tool , remove usb cable and verify buid version and date by switching on the flashed mobile.



**Fig:After successfule load of LYM mobile with Image by Qflash Tool**

**Chapter 5**

**APIs Used in QEE**

**Some basic APIs used in secure side of QEE :**

1. **void** **tz\_app\_init(void)**

In this Add any app specific initialization code here.

QSEE will call this function after secure app is loaded and authenticated

2.**void tz\_app\_cmd\_handler(void\* cmd, uint32 cmdlen,**

**void\* rsp, uint32 rsplen)**

This is Request-response buffers are allocated by non-secure side .

They are MPU protected by QSEE kernel before reaching here .

Add code to process requests and set response

3.**void tz\_app\_shutdown(void)**

This is App specific shutdown

App will be given a chance to shutdown gracefully

4.**int qsee\_read\_jtag\_id(void);**

This api Reads the JTAG ID

* return JTAG ID value

**5.int qsee\_read\_serial\_num(void);**

This api Reads the serial number from PTE chain

* return Serial number

6.**boolean qsee\_is\_ns\_range(const void\* start, uint32 len);**

This api Tests whether all of the range [\c start, \c start + \c len] is in non-secure memory. This is a convenience function to access \c tzbsp\_is\_ns\_area. \c NULL is a valid value for \c start, because

physical addressing is used.

* param [in] start Start of the memory range, physical address, included in the range.
* param [in] len Length of the memory range in bytes.
* return TRUE if the entire area is in non-secure memory.

FALSE if the area contains secure memory.

7.**boolean qsee\_is\_s\_tag\_area(uint32 tag, uint32 start, uint32 end);**

This api Tests whether all of the range [\c start, \c end] is in secure memory and also tagged for a particular use case.

* param [in] tag Tag ID of the memory it should be tagged with
* param [in] start Start of the memory range, physical address,included in the range.
* param [in] end End of the memory range, physical address,included in the range.
* return \c TRUE if the entire area is in secure memory. \c FALSE if the area contains non-secure memory.

8.**int qsee\_hdmi\_status\_read( uint32\* hdmi\_enable, uint32\* hdmi\_sense, uint32\* hdcp\_auth);**

This api Reads the status of the HDMI link and hardware HDCP

* param [out] hdmi\_enable HDMI output enabled
* param [out] hdmi\_sense HDMI sense
* param [out] hdcp\_auth HDCP authentication success.
* return 0 on success

9.**int qsee\_get\_secure\_state(qsee\_secctrl\_secure\_status\_t\* status);**

This api will check security status on the device

* param [out] status the security status (struct qsee\_secctrl\_secure\_status\_t)

- Bit field definition of the status struct,

(value 0 means pass and value 1 means fail)

Bit 0: secboot enabling check

Bit 1: check Sec HW key is programmed

Bit 2: debug disable check

Bit 3: Anti-rollback check

Bit 4: fuse config check

Bit 5: rpmb check (check rpmb production key is provisioned)

* return 0 on call success

10.**uint32 qsee\_read\_hw\_version(void);**

Reads the hardware version from TCSR\_TCSR\_REGS

* return Serial number

11.**int qsee\_get\_tz\_app\_id(uint8 \* tz\_app\_id, uint32 id\_buf\_len);**

This API get the application ID that is stored in qsee app certificate.

* return 64 bits long application ID

12.**void\* qsee\_malloc(size\_t size);**

This is Allocates a block of size bytes from the heap. If heap\_ptr is NULL or size is 0, the NULL pointer will be silently returned.

* param[in] size Number of bytes to allocate from the heap.
* return Returns a pointer to the newly allocated block, or NULL if the block could not be allocated.

13.**void qsee\_free(void \*ptr);**

This Deallocates the ptr block of memory.

* param[in] ptr pointer to block which will be freed

13.**int qsee\_register\_shared\_buffer(void \*address, unsigned int size);**

This is API for secure applications to register shared buffer with QSEE.

* param[in] address
* param[in] size
* return -1 returns error and 0 returns success.

14.**void qsee\_wait(void);**

This is API for legacy secure applications to request service from non-secure side and wait on it's response. The request and response shared buffer should be managed by the secure application.

15.**int qsee\_request\_service(unsigned int listener\_id, void \*req, unsigned int req\_len, void \*rsp, unsigned int rsplen);**

This is API for secure applications to request service from non-secure listener

* param[in] listener\_id
* param[in] req
* param[in] req\_len
* param[out] rsp
* param[in] rsplen

16.**int qsee\_deregister\_shared\_buffer(void \*address, unsigned int size);**

This is API for secure applications to deregister the shared buffer which has been previously registered with QSEE. -1 returns error and 0 returns success

* param[in] address
* return int

17.**int qsee\_register\_shared\_buffer(void \*address, unsigned int size);**

This is API for secure applications to register shared buffer with QSEE.

* param[in] address
* param[in] size
* return -1 returns error and 0 returns success

18.**int qsee\_prepare\_shared\_buf\_for\_nosecure\_read(void \* address, unsigned int size);**

This is API for prepaing shared buffer before sending it across to non-secure side.

* param[in] address
* param[in] size
* return int

19**.int qsee\_prepare\_shared\_buf\_for\_secure\_read(void \*address, unsigned int size);**

This is API for preparing the shared buffer sent by non-secure side before secure side reads it.

* param[in] address
* param[in] size
* return int

20.**int qsee\_encapsulate\_inter\_app\_message(char\* dest\_app\_name, uint8\* in\_buf,**

**uint32 in\_len, uint8\* out\_buf, uint32\* out\_len);**

This is API for a secure application to prepare a message that can be sent to another secure application. This API writes the encryped message into the supplied output buffer, prepended with a header and appended with a MAC. This output buffer can be given to the receiving app and then

passed into qsee\_decapsulate\_inter\_app\_message to be authenticated and decrypted. The actual data exchange (passing the encrypted bufer) between the secure applications can be done between their non secure clients running in the HLOS.

* param[in] dest\_app\_name
* param[in] in\_buf
* param[in] in\_len
* param[out] out\_buf
* param[in, out] out\_len

these lengths requires

**in\_len** should indicate the exact size of in\_buf.

**out\_len** should indicate the exact size of out\_buf, and this should be

greater than in\_len by 144 bytes to allow room for the header and MAC.

out\_len is modified to reflect the exact length of the data

written into out\_buf.

* return 0 indicates sucess, all other values indicate failure and

correspond to a specific error code.

21.**int qsee\_decapsulate\_inter\_app\_message(char\* source\_app\_name, uint8\* in\_buf,**

**uint32 in\_len, uint8\* out\_buf, uint32\* out\_len);**

This is API for a secure application to decapsulate (ie decrypt) a message from another secure application. This API authenticates the message, decrypts the input buffer, and writes the plaintext message into the supplied output buffer. The input buffer must have been prepared by

qsee\_encapsulate\_inter\_app\_message(), containing a header and MAC.

* param[out] source\_app\_name
* param[in] in\_buf
* param[in] in\_len
* param[out] out\_buf
* param[in, out] out\_len

These lengths requires

source\_app\_name is modified to return the sending application's identity to the caller.

in\_len should indicate the exact size of in\_buf.

out\_len should indicate the exact size of out\_buf, and this should be greater than or equal to in\_len.

* return 0 indicates sucess, all other values indicate failure and correspond to a specific error code.

22.**int qsee\_secure\_message(qsee\_sc\_ss\_e\_type ssid, qsee\_sc\_cid\_e\_type cid, const uint8\* input\_msg\_ptr, uint32 input\_msg\_len, uint8 output\_msg\_ptr, uint32\* output\_msg\_len\_ptr);**

These APIs are used by TZ applications to secure outgoing messages and authenticate incoming messages. This is used to secure the the input messageThe output buffer should be large enough to hold the encrypted message and some internal headers and possible padding. Recommended output buffer size is atleast input\_msg\_len + 100 bytes. The memory has to be managed by the caller.

* param[in] ssid Subsystem ID
* param[in] cid Client ID
* param[in] input\_msg\_ptr Pointer to the plaintext data.
* param[in] input\_msg\_len Length of the plaintext in bytes.
* param[out] output\_msg\_ptr Pointer to the buffer to hold the secure blob

(memory provided by the caller)

* param[in,out] output\_msg\_len\_ptr Size of the above buffer in bytes, set to

the length of the secure blob by the function.

* return

E\_SUCCESS - Successful. \n

E\_FAILURE - Operation failed.

@dependencies

Secure Channel has to be established successfully.

@sideeffects

None

23.**int qsee\_authenticate\_decrypt\_message(qsee\_sc\_ss\_e\_type ssid, qsee\_sc\_cid\_e\_type cid,**

**const uint8\* input\_msg\_ptr, uint32 input\_msg\_len, uint8\* output\_msg\_ptr, uint32\* output\_msg\_len\_ptr);**

These APIs are used by TZ applications to secure outgoing messages and authenticate incoming messages.This is used to authenticate and decrypt the secure blob.The output buffer should be large enough to hold the decrypted message. Recommended output buffer size is atleast input\_msg\_len.

The memory has to be managed by the caller.

* param[in] ssid Subsystem ID
* param[in] cid client ID
* param[in] input\_msg\_ptr Pointer to the secure blob.
* param[in] input\_msg\_len Length of the secure blob in bytes.
* param[out] output\_msg\_ptr Pointer to the buffer to hold the decrypted data

(memory provided by the caller)

* param[in,out] output\_msg\_len\_ptr Size of the above buffer in bytes, set to

the length of the decrypted data on return.

* return

E\_SUCCESS - Successful. \n

E\_FAILURE - Operation failed.

@dependencies

Secure Channel has to be established successfully.

@sideeffects

None

**Encryption and Decryption APIs of QEE:**

**1.int qsee\_cipher\_init(QSEE\_CIPHER\_ALGO\_ET alg,**

**qsee\_cipher\_ctx \*\*cipher\_ctx);**

This API is used to Intialize a cipher context for encrypt/decrypt operation

* param[in] alg The algorithm standard to use
* param[out] cipher\_ctx The cipher ctx
* return 0 on success, negative on failure

2**.int qsee\_cipher\_free\_ctx(qsee\_cipher\_ctx \*cipher\_ctx);**

This API is used to Release all resources with a given cipher context.

* param[in] cipher\_ctx Cipher context to be deleted
* return 0 on success, negative on failure

3.**int qsee\_cipher\_reset(qsee\_cipher\_ctx \*cipher\_ctx);**

This API is used to Resets cipher context, will not reset key

* param[in] cipher\_ctx The cipher ctx
* return 0 on success, negative on failure

4.**int qsee\_cipher\_set\_param(qsee\_cipher\_ctx \*cipher\_ctx,**

**QSEE\_CIPHER\_PARAM\_ET param\_id,**

**const void \*param,**

**uint32 param\_len);**

This API is used to Modify the parameters for a given cipher operation.

* param[in] cipher\_ctx Cipher contex
* param[in] param\_id The parameter to modify
* param[in] param The parameter value to set.
* param[in] param\_len The length of the param (in bytes).
* return 0 on success,

negative on failure,

-E\_NOT\_SUPPORTED if an alogirthm or parameter is not currently supported.

5.i**nt qsee\_cipher\_get\_param(const qsee\_cipher\_ctx \*cipher\_ctx,**

**QSEE\_CIPHER\_PARAM\_ET param\_id,**

**void \*param,**

**uint32 \*param\_len);**

This API is used to Retrieve the parameters for a given cipher context.

* param[in] cipher\_ctx Cipher context
* param[in] param\_id The parameter to retrieve
* param[in] param The memory location to store the parameter.
* param[in] param\_len The length of the param (in bytes).
* return 0 on success, negative on failure

6**.int qsee\_cipher\_encrypt(const qsee\_cipher\_ctx \*cipher\_ctx,**

**const uint8 \*pt,**

**uint32 pt\_len,**

**uint8 \*ct,**

**uint32 \*ct\_len);**

This function encrypts the passed plaintext message using the specified algorithm. The memory allocated for the ciphertext must be large enough to hold the plaintext equivalent. If the output buffer is not large enough to hold the encrypted results, an error is returned.

* param[in] cipher\_ctx The cipher context to create
* param[in] pt The input plaintext buffer
* param[in] pt\_len The input plaintext buffer length (in bytes)
* param[in,out] ct The output ciphertext buffer
* param[in,out] ct\_len The output ciphertext buffer length. This

is modified to the actual ct bytes written.

* return E\_SUCCESS if successful

E\_INVALID\_ARG if not multiple of block length

E\_FAILURE otherwise

7.**int qsee\_cipher\_decrypt(const qsee\_cipher\_ctx \*cipher\_ctx,**

**const uint8 \*ct,**

**uint32 ct\_len,**

**uint8\* pt,**

**uint32 \*pt\_len);**

This function decrypts the passed ciphertext message using the specified algorithm. The memory allocated for the plaintext must be large enough to hold the ciphertext equivalent. If the output buffer is not large enough to hold the decrypted results, an error is returned.

* param[in] cipher\_ctx The cipher context to create
* param[in] ct The input ciphertext buffer
* param[in] ct\_len The input ciphertext buffer length (in bytes)
* param[in,out] pt The output plaintext buffer
* param[in,out] pt\_len The output plaintext buffer length. This

is modified to the actual pt bytes written.

* return E\_SUCCESS if successful

E\_INVALID\_ARG if not multiple of block length

E\_FAILURE otherwise

**Secure RSA API Wrappers:**

**1.int qsee\_rsa\_key\_gen(QSEE\_RSA\_KEY \*key, int keylen,unsigned char \*pub\_exp,**

**int pub\_exp\_len);**

This function will generate RSA private/public key.

* param[out] key The public/private RSA key
* param[in] keylen RSA key length (in Bytes)
* param[in] pub\_exp Public exponent array
* param[in] pub\_exp\_len Public exponent array length
* return 0 on success, negative on failure

2**.int qsee\_secpkcs8\_parse(uint8\* data\_ptr, uint16 data\_len,**

**QSEE\_pkcs8\_privkey\_type\* privkey);**

This function parses a private key in PKCS#8 format.

* param[in] data\_ptr Pointer to the raw PKCS#8 data
* param[in] data\_len Length of the PKCS#8 data
* param[out] privkey Pointer to the private key extracte from the raw data
* return 0 on success, negative on failure

3**.int qsee\_rsa\_sign\_hash( QSEE\_RSA\_KEY \*key,QSEE\_RSA\_PADDING\_TYPE padding\_type, void \*padding\_info,QSEE\_HASH\_IDX hashidx, const unsigned char \*hash, int hashlen,unsigned char \*signature, int \*siglen);**

This function does PKCS #1 padding then sign signature.

* param[in] key The private RSA key to use
* param[in] padding\_type Type of padding
* param[in] padding\_info PSS padding parameters
* param[in] hashidx The index of the hash desired
* param[in] hash The hash to sign (octets)
* param[in] hashlen The length of the hash to sign
* param[out] signature The signatur
* param[in,out] siglen The max size and resulting size of the signatur
* return 0 on success, negative on failure

**4.int qsee\_rsa\_verify\_signature(QSEE\_RSA\_KEY \*key,**

**QSEE\_RSA\_PADDING\_TYPE padding\_type, void \*padding\_info, QSEE\_HASH\_IDX hashidx, unsigned char \*hash,**

**int hashlen, unsigned char \*sig, int siglen);**

This function does PKCS #1 padding then verify signature.

* param[in] key The private RSA key to use
* param[in] padding\_type Type of padding
* param[in] padding\_info PSS padding parameters
* param[in] hashidx The index of the hash desired
* param[in] hash The hash to sign (octets)
* param[in] hashlen The length of the hash to sign
* param[in] sig The signature
* param[in] siglen The max size and resulting size of the signatur
* return 0 on success, negative on failure

**5.int qsee\_rsa\_encrypt(QSEE\_RSA\_KEY \*key,**

**QSEE\_RSA\_PADDING\_TYPE padding\_type, void \*padding\_info,**

**const unsigned char \*msg, int msglen, unsigned char \*cipher,**

**int \*cipherlen);**

This function does PKCS #1 v1.5 padding then encrypt.

* param[in] key The RSA key to encrypt to
* param[in] padding\_type Type of padding
* param[in] padding\_info OAEP padding parameters
* param[in] msg The plaintext
* param[in] msglen The length of the plaintext(octets)
* param[out] cipher The ciphertext
* param[in,out] cipherlen The max size and resulting size of the ciphertext
* return 0 on success, negative on failure

**6.int qsee\_rsa\_decrypt(QSEE\_RSA\_KEY \*key,**

**QSEE\_RSA\_PADDING\_TYPE padding\_type, void \*padding\_info,**

**unsigned char \*cipher, int cipherlen, unsigned char \*msg,**

**int \*msglen);**

This function does PKCS #1 decrypt then v1.5 depad.

* param[in] key The corresponding private RSA key
* param[in] padding\_type Type of padding
* param[in] padding\_info OAEP padding parameters
* param[in] cipher The ciphertext
* param[in] cipherlen The length of the ciphertext(octets)
* param[out] msg The plaintext
* param[in,out] msglen The max size and resulting size of the plaintext
* return 0 on success, negative on failure

**7.int qsee\_BIGINT\_read\_unsigned\_bin( QSEE\_BigInt \* a, const uint8 \* buf, uint32 len);**

This API used to Read an unsigned buffer of bytes into a big integer

* param a[out] Pointer to big integer
* param buf[in] Pointer to array of bytes
* param len[in] Array length
* return 0 on success, negative on failure

**8.int qsee\_util\_init\_s\_bigint(QSEE\_S\_BIGINT \*\*a);**

This API used to Allocate and initialize S\_BIGINT data

* param a[in] S\_BIGINT data
* return 0 on success, negative on failure

**9.void qsee\_util\_free\_s\_bigint( QSEE\_S\_BIGINT \*a);**

This API used to Free S\_BIGINT data

* param a[in] S\_BIGINT data
* return 0 on success, negative on failure

**10.int qsee\_BIGINT\_read\_radix(QSEE\_BigInt \* a, const char \*str, uint32 radix);**

This API used to Read a zero terminated string into a big integer

* param a[out] Pointer to QSEE\_BigInt structure
* param str[in] Pointer to zero terminated string
* param radix[in] Radix
* return 0 on success, negative on failure

**11.int qsee\_rsa\_exptmod( QSEE\_RSA\_KEY \*key, const unsigned char \*in, int inlen,**

**unsigned char \*out, int \*outlen, int which);**

This API used to Compute an RSA modular exponentiation

* param[in] key The RSA key to use
* param[in] in The input data to send into RSA
* param[in] inlen The length of the input (octets)
* param[out] out The destination
* param[in,out] outlen The max size and resulting size of the output
* param[in] which Which exponent to use, e.g. PRIVATE or PUBLI
* return

CE\_SUCCESS - Function executes successfully.

CE\_ERROR\_NOT\_SUPPORTED - the feature is not supported.

CE\_ERROR\_INVALID\_PACKET - invalid packet.

CE\_ERROR\_BUFFER\_OVERFLOW - not enough space for output.

@dependencies

None.

**12.int qsee\_util\_count\_bytes(QSEE\_S\_BIGINT \* bi);**

This API used to count total byte numbers in S\_BIGINT BLONG array

* param[in] s QSEE\_S\_BIGINT data
* return total byte numbers

@dependencies

None.

**Storage APIs for The Secure Apps:**

**1.int qsee\_stor\_device\_init(qsee\_stor\_device\_id\_type device\_id, uint8 \*partition\_guid,**

**qsee\_stor\_device\_handle\_t \*device\_handle);**

Name: qsee\_stor\_device\_init

Description:

This function attempts to initialize the device indicated by device\_id and partition\_guid

Arguments:

* device\_id [IN]: Partition number of the device
* partition\_guid [IN]: Partition GUID (applies only for GPT partitions)
* device\_handle [OUT]: Pointer to a device handle
* Returns:

QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**2.int qsee\_stor\_open\_partition(qsee\_stor\_device\_handle\_t \*device\_handle, uint32 partition\_id,qsee\_stor\_client\_handle\_t \*client\_handle);**

Name: qsee\_stor\_open\_partition

Description:This function attempts to open a logical partition

Arguments:

* device\_handle [IN]: Pointer to a device handle obtained from qsee\_stor\_device\_init()
* partition\_id [IN]: Logical partition ID
* client\_handle [OUT]: Pointer to a client handle
* Returns:

QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**3.int qsee\_stor\_device\_get\_info(qsee\_stor\_device\_handle\_t \*device\_handle,**

**qsee\_stor\_device\_info\_t \*device\_info);**

Name: qsee\_stor\_device\_get\_info

Description:

This function returns the device info

Arguments:

* device\_handle [IN]: Pointer a device handle from qsee\_stor\_device\_init()
* device\_info [OUT]: Pointer to a device info structure
* Returns:

QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**4.int qsee\_stor\_client\_get\_info(qsee\_stor\_client\_handle\_t \*client\_handle,**

**qsee\_stor\_client\_info\_t \*client\_info);**

Name: qsee\_stor\_client\_get\_info

Description: This function returns the client info

Arguments:

* client\_handle [IN]: Pointer a client handle from qsee\_stor\_open\_partition()
* client\_info [OUT]: Pointer to a client info structure
* Returns:

QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**5.int qsee\_stor\_add\_partition(qsee\_stor\_device\_handle\_t \*device\_handle,**

**uint32 partition\_id, uint16 num\_sectors);**

Name: qsee\_stor\_add\_partition

Description:

This function adds a new logical partition

Arguments:

* device\_handle [IN]: Pointer a device handle from qsee\_stor\_device\_init()
* partition\_id [IN]: Logical Partition ID
* num\_sectors [IN]: Number of sectors of the new logical partition
* Returns:
* QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**6.int qsee\_stor\_read\_sectors(qsee\_stor\_client\_handle\_t \*client\_handle, uint32 start\_sector,**

**uint32 num\_sectors, uint8 \*data\_buffer);**

Name: qsee\_stor\_read\_sectors

Description:

This function attempts to read num\_sectors of data from start\_sector to data\_buffer.

Arguments:

* client\_handle [IN]: Pointer to a client handle from qsee\_stor\_open\_partition()
* start\_sector [IN]: Starting sector to read from
* num\_sectors [IN]: Number of sectors to read
* data\_buffer [OUT]: Pointer to a buffer containing data that has been read
* Returns:

QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**7.int qsee\_stor\_write\_sectors(qsee\_stor\_client\_handle\_t \*client\_handle, uint32 start\_sector,**

**uint32 num\_sectors, uint8 \*data\_buffer);**

Name: qsee\_stor\_write\_sectors

Description:

This function attempts to write num\_sectors of data from data\_buffer to start\_sector.

Arguments:

* client\_handle [IN]: Pointer to a client handle from qsee\_stor\_open\_partition()
* start\_sector [IN]: Starting sector to write to
* num\_sectors [IN]: Number of sectors to write
* data\_buffer [IN]: Pointer to a buffer containing data to be written
* Returns:

QSEE\_STOR\_SUCCESS if no errors, error code otherwise

**For QTEE HLOS side Using basic APIs:**

**1.int32\_t qsc\_start\_app(struct QSEECom\_handle \*\*l\_QSEEComHandle,const char \*appname, int32\_t buf\_size)**;

This used to Implement simple application start

* param[in/out] handle.
* param[in] appname.
* param[in] buffer size.
* return zero on success or error count on failure.

**2.int QSEECom\_start\_app(struct QSEECom\_handle \*\*clnt\_handle, const char \*path,**

**const char \*fname, uint32\_t sb\_size);**

This API used to Open a handle to the QSEECom device.

Load a secure application. The application will be verified that it is secure by digital signature verification.Allocate memory for sending requests to the QSAPP.

Note/Comments:

There is a one-to-one relation for a HLOS client and a QSAPP;meaning that only one app can communicate to a QSAPP at a time.

Please note that there is difference between an application and a listener service. A QSAPP must be loaded at the request of the HLOS,and all requests are orginated by the HLOS client. A listener service on the otherhand is started during start-up by adaemon, qseecomd.

A HLOS application may create mutiple handles to the QSAPP

* param[in/out] handle The device handle
* param[in] fname The directory and filename to load.
* param[in] sb\_size Size of the shared buffer memory for sending requests.
* return Zero on success, negative on failure. errno will be set on

error.

3)**int QSEECom\_shutdown\_app(struct QSEECom\_handle \*\*handle);**

This API used to Close the application associated with the handle.

* Unload a secure application. The driver will verify if there exists any other applications that are communicating with the QSAPP to which the "handle" is tied.
* De-allocate memory for sending requests to QSAPP.
* param[in] handle The device handle
* return Zero on success, negative on failure. errno will be set on

error.

4)**int QSEECom\_load\_external\_elf(struct QSEECom\_handle \*\*clnt\_handle, const char \*path,**

**const char \*fname);**

This Api is used to Open a handle to the QSEECom device.

- Load an external elf. The elf will be verified that it is

secure by digital signature verification.

A HLOS application may create mutiple opens (only one is permitted for the

app, but each listener service can open a unique device in the same HLOS app

executable.

* param[in/out] handle The device handle
* param[in] fname The directory and filename to load.
* return Zero on success, negative on failure. errno will be set on

error.

5)**int QSEECom\_unload\_external\_elf(struct QSEECom\_handle \*\*handle);**

This Api is used to Close the external elf

- Unload an external elf.

* param[in] handle The device handle
* return Zero on success, negative on failure. errno will be set on

error.

6)**int QSEECom\_register\_listener(struct QSEECom\_handle \*\*handle,**

**uint32\_t lstnr\_id, uint32\_t sb\_length, uint32\_t flags);**

This Api is used to Register an HLOS listener service. This allows messages from QSAPP

to be received.

* param[in] handle The device handle
* param[in] lstnr\_id The listener service identifier. This ID must be uniquely

assigned to avoid any collisions.

* param[in] sb\_length Shared memory buffer between OS and QSE.
* param[in] flags Provide the shared memory flags attributes.
* return Zero on success, negative on failure. errno will be set on

error.

7)**int QSEECom\_unregister\_listener(struct QSEECom\_handle \*handle);**

This Api is used to Unregister a listener service.

* param[in] handle The device handle
* return Zero on success, negative on failure. errno will be set on

error.

8)**int QSEECom\_send\_cmd(struct QSEECom\_handle \*handle, void \*send\_buf,**

**uint32\_t sbuf\_len, void \*rcv\_buf, uint32\_t rbuf\_len);**

This Api is used to Send QSAPP a "user" defined buffer (may contain some message/

command request) and receives a response from QSAPP in receive buffer.

The HLOS client writes to the send\_buf, where QSAPP writes to the rcv\_buf.

This is a blocking call.

* param[in] handle The device handle
* param[in] send\_buf The buffer to be sent.

If using ion\_sbuffer, ensure this

QSEECOM\_BUFFER\_ALIGN'ed.

* param[in] sbuf\_len The send buffer length

If using ion\_sbuffer, ensure length is

multiple of QSEECOM\_BUFFER\_ALIGN.

* param[in] rcv\_buf The QSEOS returned buffer.

If using ion\_sbuffer, ensure this is

QSEECOM\_BUFFER\_ALIGN'ed.

* param[in] rbuf\_len The returned buffer length.

If using ion\_sbuffer, ensure length is

multiple of QSEECOM\_BUFFER\_ALIGN.

* param[in] rbuf\_len The returned buffer length.
* return Zero on success, negative on failure. errno will be set on

error.

9)**int QSEECom\_send\_modified\_cmd(struct QSEECom\_handle \*handle, void \*send\_buf,**

**uint32\_t sbuf\_len, void \*resp\_buf, uint32\_t rbuf\_len,**

**struct QSEECom\_ion\_fd\_info \*ifd\_data);**

This Api is used to Send QSAPP a "user" defined buffer (may contain some message/ command request) and receives a response from QSAPP in receive buffer. This API is same as send\_cmd except it takes in addition parameter, "ifd\_data". This "ifd\_data" holds information (ion fd handle and cmd\_buf\_offset) used for modifying data in the message in send\_buf at an offset. Essentailly, it has the ion fd handle information to retrieve physical address and modify the message in send\_buf at the mentioned offset. The HLOS client writes to the send\_buf, where QSAPP writes to the rcv\_buf.

This is a blocking call.

* param[in] handle The device handle
* param[in] send\_buf The buffer to be sent.

If using ion\_sbuffer, ensure this

QSEECOM\_BUFFER\_ALIGN'ed.

* param[in] sbuf\_len The send buffer length

If using ion\_sbuffer, ensure length is

multiple of QSEECOM\_BUFFER\_ALIGN.

* param[in] rcv\_buf The QSEOS returned buffer.

If using ion\_sbuffer, ensure this is

QSEECOM\_BUFFER\_ALIGN'ed.

* param[in] rbuf\_len The returned buffer length.

If using ion\_sbuffer, ensure length is

multiple of QSEECOM\_BUFFER\_ALIGN.

* param[in] QSEECom\_ion\_fd\_info data related to memory allocated by ion.
* return Zero on success, negative on failure. errno will be set on

error.

10)**int QSEECom\_receive\_req(struct QSEECom\_handle \*handle,**

**void \*buf, uint32\_t len);**

This Api is used to Receive a service defined buffer.

* param[in] handle The device handle
* param[out] buf The buffer that is received
* param[in] len The receive buffer length

* return Zero on success, negative on failure. errno will be set on

error.

11)**int QSEECom\_send\_resp(struct QSEECom\_handle \*handle,**

**void \*send\_buf, uint32\_t len);**

This Api is used to Send a response based on the previous QSEECom\_receive\_req.

This allows a listener service to receive a command (e.g. read file abc).

The service can then handle the request from QSEECom\_receive\_req, and provide that information back to QSAPP.

This allows the HLOS to act as the server and QSAPP to behave as the client.

* param[in] handle The device handle
* param[out] send\_buf The buffer to be returned back to QSAPP
* param[in] len The send buffer length
* return Zero on success, negative on failure. errno will be set on

error.

12)**int QSEECom\_set\_bandwidth(struct QSEECom\_handle \*handle, bool high);**

This Api is used to Set the bandwidth for QSEE.

This API resulst in improving the performance on the Crypto hardware in QSEE. It should be called before issuing send\_cmd/send\_modified\_cmd for commands that requires using the crypto hardware on the QSEE. Typically this API should be called before issuing the send request to enable high performance mode and after completion of the send\_cmd to resume to low performance and hence to low power mode.

This allows the clients of QSEECom to set the QSEE cyptpo HW bus bandwidth to high/low.

* param[in] high Set to 1 to enable bandwidth.
* return Zero on success, negative on failure. errno will be set on

error.

13)**int QSEECom\_app\_load\_query(struct QSEECom\_handle \*handle, char \*app\_name);**

This Api is used to Query QSEE to check if app is loaded.

This API queries QSEE to see if the app is loaded or not.

* param[in] app\_name Name of the app.
* return QSEECOM\_APP\_QUERY\_FAILED/QSEECOM\_APP\_NOT\_LOADED/QSEECOM\_APP\_LOADED.

14)**int32\_t qsc\_shutdown\_app(struct QSEECom\_handle \*\*l\_QSEEComHandle)**

This API is used to Implement simple shutdown app

* param[in] handle.
* return zero on success or error count on failure.

**15.int QSEECom\_send\_service\_cmd(void \*send\_buf, uint32\_t sbuf\_len, void \*resp\_buf,**

**uint32\_t rbuf\_len, enum QSEECom\_command\_id cmd\_id);**

This API used to Send a "user" defined buffer (may contain some message/ command request) and receives a response in resp buffer.

The HLOS client writes to the send\_buf, whereas rsp\_buf is filled by TZ.

This is a blocking call.

* param[in] send\_buf : Pointer to the command buffer (populated by client) to be sent.

NOTE: This buffer contains the service related command structure.

For RPMB:

- struct type 'qseecom\_rpmb\_provision\_key' is send\_buf.

* param[in] sbuf\_len : Sizeof 'send\_buf' (sent above).
* param[in] resp\_buf : Pointer to the reponse buffer (to be populated by TZ).

Note: This buffer contains service related command structures.

For RPMB

- Does not need any separate response, so this value is NULL.

* param[in] rbuf\_len : sizeof 'resp\_buf' (sent above).
* param[in] cmd\_id : This is service specific command id of type 'enum QSEECom\_command\_id'

Can contain the following:

- QSEECOM\_RPMB\_PROVISION\_KEY\_COMMAND

- QSEECOM\_RPMB\_ERASE\_COMMAND

* return Zero on success, negative on failure. errno will be set on error.

16**.int QSEECom\_create\_key(enum QSEECom\_key\_management\_usage\_type usage, unsigned char \*hash32);**

This API used to create key.

* param[in] usage, to specify which user application will use this key management function.
* param[in] a 32bit hash as password (optional).
* return Zero on success, negative on failure. errno will be set on error.

1**7.int QSEECom\_wipe\_key(enum QSEECom\_key\_management\_usage\_type usage);**

This API wipe the key.

* param[in] usage, to specify which user application will use this key management function.
* return Zero on success, negative on failure. errno will be set on error.

18.**int QSEECom\_clear\_key(enum QSEECom\_key\_management\_usage\_type usage);**

This API clear the key from PIPE.

* param[in] usage, to specify which user application will use this key management function.
* return Zero on success, negative on failure. errno will be set on error.

19**.int QSEECom\_update\_key\_user\_info(enum QSEECom\_key\_management\_usage\_type usage, unsigned char \*current\_hash32, unsigned char \*new\_hash32);**

This API update the key user info.

* param[in] usage, to specify which user application will use this key management function.
* param[in] current\_hash32, to specify the current password hash
* param[in] new\_hash32, to specify the new password hash.
* return Zero on success, negative on failure. errno will be set on error.

20.**int QSEECom\_send\_modified\_resp(struct QSEECom\_handle \*handle, void \*send\_buf,**

**uint32\_t sbuf\_len, struct QSEECom\_ion\_fd\_info \*ifd);**

A "user" defined API that contains the Ion fd allocated by the listener service, to be communicated with TZ. This API is same as "QSEECom\_send\_modified\_cmd" but servicing the listener instead of app. This "ifd\_data" holds information (ion fd handle and cmd\_buf\_offset) used for modifying data in the message in send\_buf at an offset. Essentailly, it has the ion fd handle information to retrieve physical address and modify the message in send\_buf at the mentioned offset.

* param[in] handle, The device handle
* param[in] send\_buf, The buffer to be sent.

If using ion\_sbuffer, ensure this

QSEECOM\_BUFFER\_ALIGN'ed.

* param[in] sbuf\_len, The send buffer length

If using ion\_sbuffer, ensure length is

multiple of QSEECOM\_BUFFER\_ALIGN.

* param[in] QSEECom\_ion\_fd\_info, data related to memory allocated by ion.
* return Zero on success, negative on failure. errno will be set on error.

21.**int QSEECom\_scale\_bus\_bandwidth(struct QSEECom\_handle \*handle, int mode);**

This API used to scale bus bandwidth.

* param[in] handle The device handle
* param[in] mode INACVTIVE, LOW, MEDIUM or HIGH
* return Zero on success, negative on failure. errno will be set on error.

**qseecom\_lk\_apis:**

**1.int qseecom\_init();**

Qseecom Init to be called before any calls to qsee secure apps.

* return int

Success: Init succeeded.

Failure: Error code (negative only).

**2.void qseecom\_lk\_set\_app\_region(uint32\_t addr, uint32\_t size);**

Qseecom set app address regions to be called before any calls to tz init.

* return void

Success: Not applicable

Failure: Not applicable

**3.unsigned int qseecom\_get\_version();**

Qseecom get version to be called before calls to set app region.

* return unsigned int

Success: Valid version

Failure: Garbage value

**4.int qseecom\_tz\_init();**

Qseecom Tz Init to be called before any calls to qsee secure apps.

* return int

Success: Tz init succeeded.

Failure: Error code (negative only).

**5.int qseecom\_exit();**

Qseecom Exit to be called before exit of lk.Once this is called no calls to Qsee Apps.

* return int

Success: Exit succeeded.

Failure: Error code (negative only).

**6.int qseecom\_start\_app(char \*app\_name);**

This API used to start a Secure App

* param char\* app\_name

App name of the Secure App to be started

The app\_name provided should be the same

name as the partition/ file and should

be the same name mentioned in TZ\_APP\_NAME

in the secure app.

* return int

Success: handle to be used for all calls to

Secure app. Always greater than zero.

Failure: Error code (negative only).

**7.int qseecom\_shutdown\_app(int handle);**

This API used to Shutdown a Secure App

* param int handle

Handle of the Secure App to be shutdown

* return int

Status:

0 – Success .

Negative value indicates failure.

**8.int qseecom\_send\_command(int handle, void \*send\_buf,**

**uint32\_t sbuf\_len, void \*resp\_buf, uint32\_t rbuf\_len);**

This API used to send command to a Secure App

* param int handle

Handle of the Secure App to send the cmd

* param void \*send\_buf

Pointer to the App request buffer

* param uint32\_t sbuf\_len

Size of the request buffer

* param void \*resp\_buf

Pointer to the App response buffer

* param uint32\_t rbuf\_len

Size of the response buffer

* return int

Status:

0 - Success

Negative value indicates failure.

**9.int qseecom\_register\_listener(struct qseecom\_listener\_services \*listnr);**

This API used to Registers a Listener Service with QSEE. This api should be called after all service specific initialization is completed, once this is called the service\_cmd\_handler for the service can be called.

* param struct qseecom\_listener\_services listnr

Listener structure that contains all the info to register a listener service.

* return int

Status:

0 - Success

Negative value indicates failure.

**10.int qseecom\_deregister\_listener(uint32\_t listnr\_id);**

This API used to De-Registers a Listener Service with QSEE. This api should be called before exiting lk and all service de-init should be done before calling the api. service\_cmd\_handler will not be called after this api is called.

* param uint32\_t listnr\_id

Pre-defined Listener ID to be de-registered

* return int

Status:

0 - Success

Negative value indicates failure.