IWCU OBU 6.0A User Manual

Jun. 27, 2016

版　　別：第 1.1 版

總 頁 數：52 頁 (含封面及附件)

文件編號：523A40612

Table of Content

1. Preface

1.1 History

version	date	contention
V1.0	Sep. 23, 2015	Fisrt Version
V1.1	Jun. 27, 2016	FCC regulatory information

2. Introduction

The IWCU 6.0 contains V2X elements of DSRC/GNSS/HSM with pre-integrated IEEE 802.11p driver and IEEE 1609.x protocol stack running on ThreadX RTOS.

2.1 Acronyms

DSRC	Dedicated Short Range Communication
CCH	Control Channel
SCH	Service Channel
PSC	Provider Service Context
PSID	Provider Service Identifier
WAVE	Wireless Access in Vehicular Environments
WSA	WAVE Service Advertisements
WSMP	WAVE Short Message Protocol
WME	WAVE Management Entity
BSM	Basic Safety Message

2.2 FCC regulatory information

FEDERAL COMMUNICATIONS COMMISSION INTERFERENCE STATEMENT

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

-Reorient or relocate the receiving antenna.

-Increase the separation between the equipment and receiver.

-Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.

-Consult the dealer or an experienced radio/ TV technician for help.

CAUTION:

Any changes or modifications not expressly approved by the grantee of this device could void the user's authority to operate the equipment.

RF Exposure warning

This equipment must be installed and operated in accordance with provided instructions and the antenna(s) used for this transmitter must be installed to provide a separation distance of at least 20 cm from all persons and must not be co-located or operating in conjunction with any other antenna or transmitter. End-users and installers must be provide with antenna installation instructions and transmitter operating conditions for satisfying RF exposure compliance.

3. Configuration using CLI

3.1 Ethernet configuration setting

To get or set the Ethernet/wireless device's IP address and netmask, we have: `ip eget` and `ip eset`. The default IP address is `192.168.10.50`.

Syntax: `ip eset <addr> <netmask>`

Example: `ip eset 192.168.10.50 255.255.255.0`

3.2 Channel status

To retrieve channel status, we have: `dot3_show_channel`

Example:

```
ate> dot3_show_channel
wave0 channel = 178
wave1 channel = 140
```

3.3 GPS status

To retrieve GPS status, we have: `gps show`

Example:

```
ate> gps show
GPSINFO: time: Fri Jun 5 09:03:14 2015
        latitude 24.7770833, longitude 121.0437083
        altitude 155.3801936 m
        ground speed 0.0514444 m/s
        angle 0.0000000
```

3.4 WSM Packet – Tx

To send WSM packet, we have the following primitive: `dot3_tx_user`

Syntax: `dot3_tx_user <psid> <request_type> <channel_number> <is_switch> <data_rate> <tx_power> <priority> <dest_mac> <size> <interval> <num>`

The parameters are described in the following table:

Parameter	Valid Range	Description
psid	1 ~ 127	psid of sending WSMV2 packets,

request_type	0: Match 1: Unconditional 2: No Access	User service access type
channel_number	172 174 176 180 182 184	channel number of sending WSMV2 packets
is_switch	0: Continuous 1: Channel switch	channel mode of user service
data_rate	6 9 12 18 24 36 48 54	data rate of sending WSMV2 packets
tx_power	12 ~ 25, unit: dBm	tx power level of sending WSMV2 packets
priority	0 ~ 63	priority of user service
dest_mac	format: XX:XX:XX:XX:XX:XX	peer mac address of sending WSMV2 packets
size	0 ~ 1399, unit: byte	size of sending packets
interval	0 ~ 1000, unit: ms	interval of sending packets
num	1 ~ 10000	number of sending packets

Example: `dot3_tx_user 123 1 174 0 6 18 1 FF:FF:FF:FF:FF:FF 1000 100 10`

3.5 WSM Packet – Rx

To receive WSM packet, we have the following primitive: `dot3_rx_user`

Syntax: `dot3_rx_user <psid> <radio_number> <channel_number>`

The parameters are described in the following table:

Parameter	Valid Range	Description
psid	1 ~ 127	psid of sending WSMV2 packets
radio_number	0 ~ 1	the radio number
channel_number	172 174 176 180 182 184	channel number of sending WSMV2 packets

Example: `dot3_rx_user 123 0 174`

To keep receiving WSM packet continuously, we have the following primitives: `dot3_rx_user_start` and `dot3_rx_user_stop`

Example: `dot3_rx_user_start 123 0 174`

3.6 WSA – Tx

To send WSA, we have the following primitive: `dot3_rx_provider`

Syntax: `dot3_rx_provider <psid> <radio_number> <channel_number> <is_switch> <priority> <is_secure>`

The parameters are described in the following table:

Parameter	Valid Range	Description
psid	1 ~ 127	psid of sending WSMV2 packets
radio_number	0 ~ 1	the radio number
channel_number	172 174 176 180 182 184	channel number of sending WSMV2 packets
is_switch	0: Continuous 1: Channel switch	channel mode of provider service
priority	0 ~ 63	priority of provider service
is_secure	0: no secure WSA 1: secured WSA	enable secure WSA or not

Example: `dot3_rx_provider 123 0 172 1 1 0`

3.7 Checking MIB status

To retrieve MIB setting, we have the following primitive: `dot3_show_mib`

Illustrating dot3_show_mib command:

Example:
On OBU-unit 1, enable a user service by sending WSM packets.

```
ate> dot3_tx_user 123 1 174 0 18 18 1 FF:FF:FF:FF:FF:FF 1000 100 10
wme_init success, handle = 1
wme_user_service to add success
Change wave0 to 174
Channel 174 assigned ...
wme_user_service to delete success
ate> Change wave0 to 178
```

On OBU-unit 2, receive WSM packets.

```
ate> dot3_rx_user_start 123 0 174
wme_init success, handle = 1
wme_user_service to add success
wme_wsm_service to add success
```

```
ate> Change wave0 to 174
Channel 174 assigned ...
start to recv wsm packets
wsm recv, count = 1, rssi = -56
wsm recv, count = 2, rssi = -57
wsm recv, count = 3, rssi = -56
wsm recv, count = 4, rssi = -56
wsm recv, count = 5, rssi = -56
wsm recv, count = 6, rssi = -55
wsm recv, count = 7, rssi = -56
wsm recv, count = 8, rssi = -56
wsm recv, count = 9, rssi = -58
wsm recv, count = 10, rssi = -59
```

View MIB status:

```
ate> dot3_show_mib
wme_init success, handle = 2
*** Provider entry ***
*** User entry ***
index = 0
psid  = 123
psc   =
priority   = 1
wsa_type  = 0
radio num  = 0
channel  = 174

*** Wsm entry ***
index = 0
psid = 123

*** Available entry ***
ate>
```

As shown above, we can see a User, WSM entry into MIB.

3.8 Secure WSM/WSA

To demonstrate the 1609.2 secure WSM/WSA, we have: `dot2_demo`

Syntax: `dot2_demo <data_type> <sign> <sign_data_type> <attach_signer_type> <enc> <verify> <sign_alg> <sig_type> <ecc_point_type>`

The parameters are described in the following table:

Parameter	Valid Range	Description
data_type	0: WSMP 1: WSA	Specify which data type to be processed
sign	0: not sign 1: sign	Specify whether to sign the data or not
sign_data_type	0: payload 1: partial payload 2: external payload	[ONLY APPLY FOR SIGNED WSMP] Specify the signed data type. If set to 0, sign the data only as normal; if set to 1, duplicate input data as external data; if set to 2, use input data as external data to sign with external data type
attach_signer_type	0: certificate digest 1: certificate 2: certificate chian	[ONLY APPLY FOR SIGNED WSMP] Specify the type of signer info attached in the signed data
enc	0: not encrypt 1: encrypt	Specify whether sign the data or not, if set to 1, use self as recipient for successfully decrypting
verify	0: not verify 1: verify	Specify whether to verify the data or not if it is signed
sign_alg	0: ECDSA-224 1: ECDSA-256	Specify the signing algorithm to use
sig_type	0: uncompressed 1: compressed 2: x-coordinate only	Specifies whether to use point compression or just x-coordinate in signature of signed message or not
ecc_point_type	0: uncompressed 1: compressed	Specifies whether to use point compression in elliptic curve points of the encrypted message or not

Example: `dot2_demo 0 1 0 1 0 1 1 1 1`

3.9 Dump certificate

To dump certificates of current secure data store (SDS), we have: `dot2_show_sds`

Illustrating dot2_show_sds command:

Example:
A self-generated, random WSM message of 0 ~ 100 bytes length can be viewed on CLI using `<dot2_demo>` command as shown below:

The first certificate string is a CA certificate while the second certificate string is a WSM certificate. Further, a testing message of 77 bytes length is shown followed by the encoded result.

```
ate> dot2_demo 0 1 0 1 0 1 1 1 1
Create subject type 1 cert error, code = 5
Cert string:
00000000  02ff 0500 80ff 0100 0100 0403 c30b 1500
00000010  0000 9900 0000 0101 0227 e43c 6566 9c7f
00000020  0067 7528 9446 15bb b88d a7f0 bd97 ce28
00000030  c8d7 c3f5 4f5a 3e44 3202 0003 c013 272d
00000040  7f4a 3754 2ba2 384d 9867 25f6 49ee d428
00000050  1b06 e64f 7001 ea87 f735 0666 001c 83e8
00000060  9a10 ea58 a113 00e1 8091 a713 487b 99a6
00000070  43b6 fbbf 959d 023c 2aad d92c 2204 47fb
00000080  3614 74a5 0722 25b2 cd95 40c2 a4ad f93a
00000090  c9cf 0375 b72f a4f4 cd1a 4709 a1
0000009d
Cert string:
00000000  0201 05e7 8f83 538c 0895 0901 0001 0220
00000010  0001 e185 d700 0000 9900 0000 0101 02c2
00000020  fbdf 0352 37cd 6b5a 53c8 e83b b117 2c84
00000030  b9cd 797d 1096 ecba 4bde 3d69 b872 8902
00000040  0003 6aae 44e8 c545 c838 8d5d 0dcc 5dfd
00000050  ec2a 646a 1a99 f5bd ce6d 9c89 2ed4 aac6
00000060  9948 002c e7e9 8bbf da9f 2935 5963 8d55
00000070  2ba2 073c c7d9 96b5 3c43 1398 40ca 0cba
00000080  bead 5ade e741 71ee c441 b43b 8d59 a09c
00000090  5e10 ae26 0077 70a6 aa05 e0c3 8983 989c
000000a0  4899 cc
000000a3
scan root cert string:
02ff040083ff01000100041c374a8200000001010269df0190667e3cbdcad09d7033c
cac2ffc0ff1
302496cc92f9c6e50b9db172b40200029353547bef08054550c0a8367f9adcc2b23fe
2b5b01746b3
eea4a869b5f2eb3000c2271dbdb78fe730b9f651dd759ec0f844b8b07333c80b27008
```

```
43a0ac1b7c4
f9540af50982a49f71966cdc210f21a2780d1367c582de43c457c2d651fa8a28f3
Load public root ca certificate success.

Testing msg [length = 77 bytes]:
00000000  ef27 b848 cdf5 1fc3 86ba c597 1f05 042d
00000010  070f b5c5 edf6 feb6 1c4c 2000 744d ad6a
00000020  f7d1 d032 adf8 1767 af94 4acb 68f4 019d
00000030  66e4 9977 de2e aa64 4f2a dd03 4a2b e81f
00000040  8629 04a2 c9c8 67f9 58e5 c767 87
0000004d

1609.2 Message:
00000000  0201 0302 0105 e78f 8353 8c08 9509 0100
00000010  0102 2000 01e1 85d7 0000 0099 0000 0001
00000020  0102 c2fb df03 5237 cd6b 5a53 c8e8 3bb1
00000030  172c 84b9 cd79 7d10 96ec ba4b de3d 69b8
00000040  7289 0200 036a ae44 e8c5 45c8 388d 5d0d
00000050  cc5d fdec 2a64 6a1a 99f5 bdce 6d9c 892e
00000060  d4aa c699 4800 2ce7 e98b bfda 9f29 3559
00000070  638d 552b a207 3cc7 d996 b53c 4313 9840
00000080  ca0c babe ad5a dee7 4171 eec4 41b4 3b8d
00000090  59a0 9c5e 10ae 2600 7770 a6aa 05e0 c389
000000a0  8398 9c48 99cc 0620 4def 27b8 48cd f51f
000000b0  c386 bac5 971f 0504 2d07 0fb5 c5ed f6fe
000000c0  b61c 4c20 0074 4dad 6af7 d1d0 32ad f817
000000d0  67af 944a cb68 f401 9d66 e499 77de 2eaa
000000e0  644f 2add 034a 2be8 1f86 2904 a2c9 c867
000000f0  f958 e5c7 6787 0000 0000 0932 9e51 0000
00000100  0000 0009 cb33 ac02 520b 9bb6 33f8 3d58
00000110  1c62 aa65 49b1 9d55 8fba 40f9 376c 9ff4
00000120  af7f 1aa9 3b84 cdd5 effc 5780 ab94 1dc6
00000130  931c 8600 735c 248b 7d31 8ee4 3172 5174
00000140  f7c7 8ec1 f8d3 3539
00000148

Extracted msg:
00000000  ef27 b848 cdf5 1fc3 86ba c597 1f05 042d
00000010  070f b5c5 edf6 feb6 1c4c 2000 744d ad6a
00000020  f7d1 d032 adf8 1767 af94 4acb 68f4 019d
00000030  66e4 9977 de2e aa64 4f2a dd03 4a2b e81f
00000040  8629 04a2 c9c8 67f9 58e5 c767 87
0000004d

Bye
Dot2 demo thread was removed.
```

```
ate>
```

To dump certificate content, type:

```
ate> dot2_show_sds
Dump private cert cache info:
  Index 0:
  Subject type: 255
  PSID: All
  Expiration: 3c30b15
  Start time: 99
  Digest: e78f8353 8c089509
  Issuer ID: 00000000 00000000
  Signing algorithm: 1
  CRL series: 00000001
  Verification key:
    02 27e43c65 669c7f00 67752894 4615bbb8
       8da7f0bd 97ce28c8 d7c3f54f 5a3e4432

  Encryption key:
    03 c013272d 7f4a3754 2ba2384d 986725f6
       49eed428 1b06e64f 7001ea87 f7350666

  Certificate string: length = 157
    02ff0500 80ff0100 01000403 c30b1500
    00009900 00000101 0227e43c 65669c7f
    00677528 944615bb b88da7f0 bd97ce28
    c8d7c3f5 4f5a3e44 32020003 c013272d
    7f4a3754 2ba2384d 986725f6 49eed428
    1b06e64f 7001ea87 f7350666 001c83e8
    9a10ea58 a11300e1 8091a713 487b99a6
    43b6fbbf 959d023c 2aadd92c 220447fb
    361474a5 072225b2 cd9540c2 a4adf93a
    c9cf0375 b72fa4f4 cd1a4709 a1

  Index 1:
  Subject type: 1
  PSID: 0x20,
  Expiration: 1e185d7
  Start time: 99
  Digest: 20a24219 c5c51080
  Issuer ID: e78f8353 8c089509
  Signing algorithm: 1
  CRL series: 00000001
  Verification key:
    02 c2fbdf03 5237cd6b 5a53c8e8 3bb1172c
       84b9cd79 7d1096ec ba4bde3d 69b87289
```

```
  Encryption key:
    03 6aae44e8 c545c838 8d5d0dcc 5dfdec2a
       646a1a99 f5bdce6d 9c892ed4 aac69948

  Certificate string: length = 163
    020105e7 8f83538c 08950901 00010220
    0001e185 d7000000 99000000 010102c2
    fbdf0352 37cd6b5a 53c8e83b b1172c84
    b9cd797d 1096ecba 4bde3d69 b8728902
    00036aae 44e8c545 c8388d5d 0dcc5dfd
    ec2a646a 1a99f5bd ce6d9c89 2ed4aac6
    9948002c e7e98bbf da9f2935 59638d55
    2ba2073c c7d996b5 3c431398 40ca0cba
    bead5ade e74171ee c441b43b 8d59a09c
    5e10ae26 007770a6 aa05e0c3 8983989c
    4899cc


Dump public cert cache info:
  Index 0:
  Subject type: 255
  PSID: All
  Expiration: 1c374a82
  Start time: 0
  Digest: 33f522fa 83173d89
  Issuer ID: 00000000 00000000
  Signing algorithm: 1
  CRL series: 00000001
  Verification key:
    02 69df0190 667e3cbd cad09d70 33ccac2f
       fc0ff130 2496cc92 f9c6e50b 9db172b4

  Encryption key:
    02 9353547b ef080545 50c0a836 7f9adcc2
       b23fe2b5 b01746b3 eea4a869 b5f2eb30

  Certificate string: length = 153
    02ff0400 83ff0100 0100041c 374a8200
    00000101 0269df01 90667e3c bdcad09d
    7033ccac 2ffc0ff1 302496cc 92f9c6e5
    0b9db172 b4020002 9353547b ef080545
    50c0a836 7f9adcc2 b23fe2b5 b01746b3
    eea4a869 b5f2eb30 00c2271d bdb78fe7
    30b9f651 dd759ec0 f844b8b0 7333c80b
    2700843a 0ac1b7c4 f9540af5 0982a49f
```

```
    71966cdc 210f21a2 780d1367 c582de43
    c457c2d6 51fa8a28 f3

  Index 1:
  Subject type: 1
  PSID: 0x20,
  Expiration: 1e185d7
  Start time: 99
  Digest: 20a24219 c5c51080
  Issuer ID: e78f8353 8c089509
  Signing algorithm: 1
  CRL series: 00000001
  Verification key:
    02 c2fbdf03 5237cd6b 5a53c8e8 3bb1172c
       84b9cd79 7d1096ec ba4bde3d 69b87289

  Encryption key:
    03 6aae44e8 c545c838 8d5d0dcc 5dfdec2a
       646a1a99 f5bdce6d 9c892ed4 aac69948

  Certificate string: length = 163
    020105e7 8f83538c 08950901 00010220
    0001e185 d7000000 99000000 010102c2
    fbdf0352 37cd6b5a 53c8e83b b1172c84
    b9cd797d 1096ecba 4bde3d69 b8728902
    00036aae 44e8c545 c8388d5d 0dcc5dfd
    ec2a646a 1a99f5bd ce6d9c89 2ed4aac6
    9948002c e7e98bbf da9f2935 59638d55
    2ba2073c c7d996b5 3c431398 40ca0cba
    bead5ade e74171ee c441b43b 8d59a09c
    5e10ae26 007770a6 aa05e0c3 8983989c
    4899cc


Dump profile cache info:
PSID:
0x     20

ate>
```

3.10 BSM Packet – Tx

To send BSM packet, we have the following primitive: `j2735bsm`

3.11 CAN Testing

To test CAN, we have the following primitive: `<can_driver_info>`

Syntax: `can_driver_info <device id [0-1]>`

Example: `can_driver_info 0`

Note: Tx & Rx values can be viewed from the parameters: `CAN interface tx frames count` & `CAN interface rx frames count`

3.12Upgrade Firmware

1. Setup a serial connection to the unit
2. Ensure that the unit is connected to an Ethernet LAN
3. Reset the unit and enter U-Boot console by pressing any key during the 3 second countdown. You should see the U-Boot console prompt:

   ```
   U-Boot>
   ```

4. We will assume the IP address of your TFTP server is 192.168.10.100. Perform ping test

   ```
   ping 192.168.10.100
   ```

5. If the ping was successful, you should see output similar to:

   ```
   Link: UP
   Duplex: FULL
   Speed 100BASE-X
   Using device
   host 192.168.10.100 is alive
   ```

6. Configure the TFTP server's IP address:

   ```
   setenv serverip 192.168.10.100
   ```

7. We will assume that the image you would like to boot is served by the TFTP server under the name uImage. Issue the following command to load the image via TFTP:

   ```
   tftp wave_itri.img
   ```

8. Erasing flash memory is done differently depending on flash size. When using a 8 MB flash device:

   ```
   protect off 80000 7fffff
   erase 80000 7fffff
   ```

9. When using a 32 MB flash device:

```
mw.l 0x41016850 0x001f0000
protect off 80000 1fdffff
erase 80000 1fdffff
```

10. Replace existing firmware image with the new one:

```
cp.b ${fileaddr} 80000 ${filesize}
setenv bootcmd cp.b 80000 50000000 ${filesize}\; bootm
saveenv
```

11. You should see output similar to:

```
U-Boot> tftp wave_itri.img
Link: UP
Duplex: FULL
Speed 100BASE-X
Using device
TFTP from server 192.168.10.100; our IP address is 192.168.10.50
Filename 'wave_itri.img'.
Load address: 0x50000000
Loading: TftpRemotePort=69
#################################################################
#########################################
done
Bytes transferred = 552100 (86ca4 hex)
U-Boot> protect off 80000 7fffff
.................................................................
done
Un-Protected 120 sectors
U-Boot> erase 80000 7fffff
.................................................................
done
Erased 120 sectors
U-Boot> cp.b ${fileaddr} 80000 ${filesize}
Copy to Flash... done
U-Boot> setenv bootcmd cp.b 80000 50000000 ${filesize}\; bootm
U-Boot> saveenv
Saving Environment to Flash...
.. done
Un-Protected 2 sectors
Erasing Flash...
.. done
Erased 2 sectors
Writing to Flash... done
```

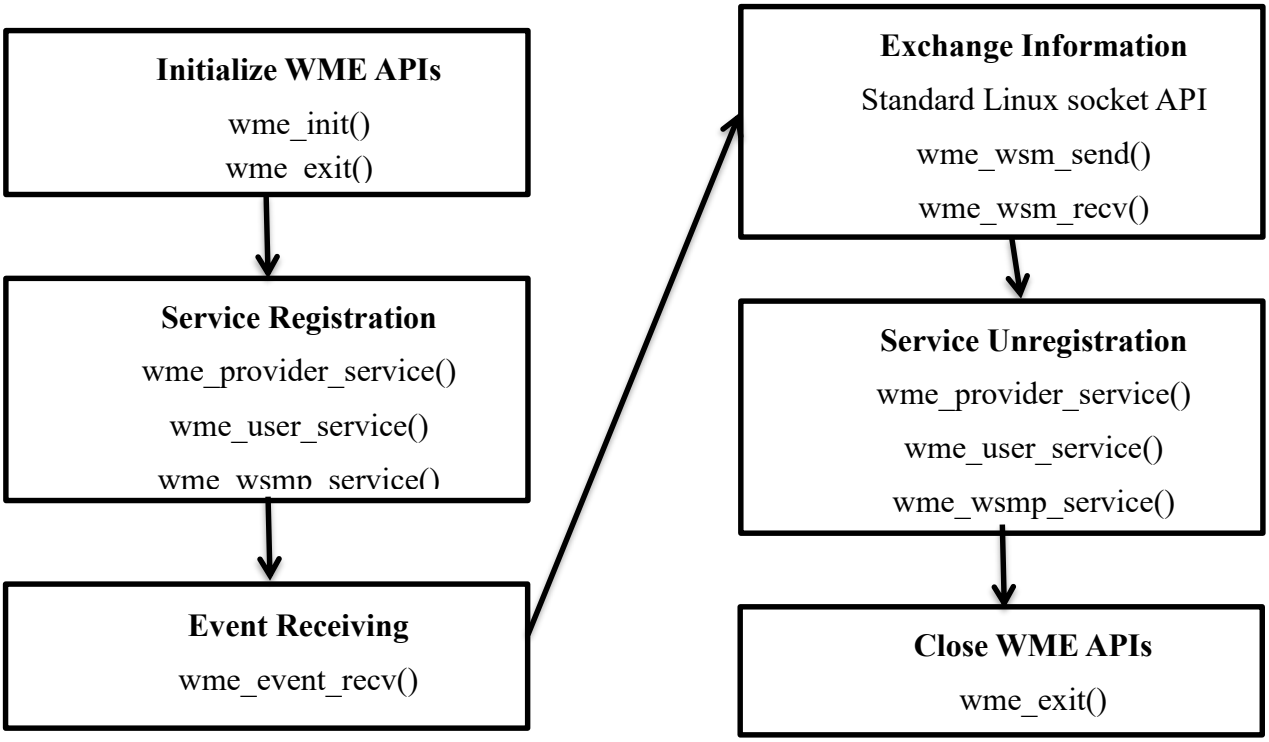
```
.. done
Protected 2 sectors
Done
```

12. Reset the unit via the command:

```
reset
```

4 IEEE 1609.3 PROGRAMMING API

The process for using IEEE 1609.3 networking services API is shown below:



4.1 Initialize WME APIs

```
int wme_init(wme_handle_t *handle);
```

This function will initialize WME service and get the handle. Further, the handle can be used to access services and exchange data.

```
int wme_exit(wme_handle_t *handle);
```

This function will exit WME service and release the handler.

4.2 Service Registration/Unregistration

The WME system provides three kinds of services: Provider, User and WSMP

A provider service is registered by calling the following function:

```
    int wme_provider_service(wme_handle_t *handle, struct
provider_info *info);
```

When this function is called, the system will open the specific channel if the channel is available. If success, system will generate a corresponding WSA and send it periodically on control channel (Channel 178), by default.

The structure of the provider information is defined as:

```
struct provider_info
{
  unsigned char  action;
  unsigned char  dest_mac[MAC_SIZE];
  unsigned char  wsa_type;
  unsigned int   psid;
  unsigned char  psc[PSC_SIZE];
  unsigned char  psc_len;
  unsigned char  service_priority;
  unsigned char  radio_number;
  unsigned char  channel_number;
  unsigned char  channel_access;
  unsigned char  repeat_rate;
  unsigned char  ip_service;
  unsigned char  ipv6_addr[IPV6_SIZE];
  unsigned short service_port;
  unsigned char  provider_mac[MAC_SIZE];
  unsigned char  wsa_count_threshold;
```

```
  unsigned char   wsa_count_threshold_interval;

  unsigned int    wsa_hdr_extensions;
};
```

The fields are described in the following table:

Field	Description
action	Valid value: WME_ADD, WME_DEL
dest_mac	Valid value: 0xFFFFFFFFFFFF
wsa_type	Valid value: TYPE_UNSCURED
psid	The receiving Provider Service Identifier (PSID) Valid value: defined by IEEE 1609.3
psc	The description is associated to the psid, and the max length is PSC_SIZE (31 bytes)
psc_len	The real length of PSC
service_priority	Service priority: 0 ~ 63. The lower value has higher priority. DEFAULT_PRIORITY can be used
radio_number	The valid value is dependent on the installed phys in the system. Valid value: 0, 1
channel_number	Valid value: 172, 173, 174, 175, 176, 180, 181, 182, 183, 184. The control channel can't be used.
channel_access	Valid value: MODE_BOTH, MODE_SCH
repeats	The number of WSAs to be sent per 5 seconds. The real sending rate is equal to repeats +1. Valid value: 0~ 255
ip_service	If the service is a IPv6 service, set to 1. If no use, set the value to 1
ipv6_addr	Specify the IPv6 address for the service
service_port	Specify the service port for the service

provider_mac	Specify the provider's mac address. It is optional. If no use, Set the value to 0xFFFFFFFFFFFF.
wsa_count_threshold	It indicates the recommended minimum number of received WSAs. It is optional. If no use, set it to 0. Valid value: 0 ~ 255.
wsa_count_threshold_interval	It indicates the time interval over which received WSAs are counted. The unit is 100ms. It is optional. If no use, set it to 0. Then the remote user accepting the WSA will use the default interval (one second). Valid value: 0 ~ 255
wsa_hdr_extensions	Indicate which of the provider service extension fields is included in the WSA

A user service is registered by calling the following function:

```
    int wme_user_service(wme_handle_t *handle, struct user_info
*info);
```

When calling the function, the system will open the specific channel if the channel is available. For now, PSID, Channel and MAC are used to compare with the received WSA. If all the parameters are matched, the system will send an event message to the corresponding user.

The structure of the user information is defined as:

```
struct user_info
{
    unsigned char    action;
    unsigned char    wsa_type;
    unsigned char    request_type;
    unsigned int     psid;
    unsigned char    psc[PSC_SIZE];
    unsigned char    psc_len;
    unsigned char    service_priority;
    unsigned char    radio_number;
    unsigned char    channel_number;
    unsigned char    src_mac[MAC_SIZE];
    unsigned char    advertiser_id[ADVERTISER_ID_SIZE];
```

```
    unsigned char    advertiser_id_len;
    unsigned short   extended_access;
};
```

The fields are described in the following table:

Field	Description	Match
action	Valid value: WME_ADD, WME_DEL	
wsa_type	Valid value: TYPE_UNSCURED	
request_type	Valid value: ACCESS_ON_MATCH, ACCESS_UNCONDITIONAL, ACCESS_NO_SCH	
psid	The receiving Provider Service Identifier (PSID)	●
psc	The description is associated to the psid, and the max length is PSC_SIZE (31 bytes).	
psc_len	The real length of PSC	
service_priority	Service priority: 0 ~ 63. The lower value has higher priority. DEFAULT_PRIORITY can be used.	
radio_number	The valid value is dependent on the installed phys in the system. Valid value: 0, 1	
channel_number	The user service will match up the provider service with the channel. Valid value: 0, 172, 173, 174, 175, 176, 180, 181, 182, 183,	●

	184. The control channel can't be used.	
src_mac	The user will match up the provider service with the MAC. The value 0xFFFFFFFFFFFF indicates that any MAC is accepted.	●
advertiser_id	The description is associated to the device sending WSAs, and the max length is ADVERTISER_ID_SIZE (32 bytes).	
advertiser_id _len	The real length of the advertiser_id	
extended_access	The value 0 indicates alternating access (channel switch), and the value 1 indicates continuous access.	

A WSMP service is registered by calling the following function:

```
int wme_wsm_service(wme_handle_t *handle, struct wsm_info *info);
```

When the function is called, the system can send the received packet with the corresponding PSID to users.

The structure of the WSMP information is defined as:

```
struct wsm_info
{
    unsigned char action;
    unsigned int psid;
};
```

The fields are described in the following table:

Field	Description
action	Valid value: WME_ADD, WME_DEL

psid	The receiving Provider Service Identifier (PSID)

4.3 Waiting Events

If a provider service is registered, the system may notify user about the channel availability while if a user service is registered, the system may notify users of upcoming matching services in the form of Channel or Service availability.

If the user needs to receive events asynchronously, the `wme_event_recv_notify` function can be used. This function registers a receive event callback function. Whenever an event is received, the callback function is executed. Below functions illustrate the details.

```
int wme_event_recv_notify(wme_handle_t *handle, void (*cb_func)
(wme_handle_t *handle));
```

This function is used to register callback function for handler. The system will send a notification about an upcoming event using the callback function.

Following function can be used to receive the event.

```
int wme_event_recv(wme_handle_t *handle, struct event_message
*event, unsigned int timeout);

int wme_event_waiting_terminate(wme_handle_t *handle);
```

This function is used to unregister the callback function for handler

When the function is called, the system will notify the user of some events.
The structure of the event is defined as:

```
struct event_message
{
    unsigned char event;
    unsigned char reason;
    union {
        struct event_channel channel;
        struct mib_available_service_info service;
    } info;
};
```

The fields are described in the following table:

Field	Description
event	Valid value: EVENT_SERVICE, EVENT_CHANNEL

reason	Valid value: REASON_SERVICE_AVAILABLE, REASON_SERVICE_UNAVAILABLE, REASON_CHANNEL_AVAILABLE, REASON_CHANNEL_UNAVAILABLE
info	Dependent on the event and the reason. The information includes if a service is available or not and if a specific channel is assigned or not.

4.4 Device Information

To get the status of individual network devices, the following function can be used:

```
int wme_device_get(unsigned char dev_id, struct dev_info *info);
```

The structure of the device information is defined as:

```
struct dev_info
{
    unsigned char    channel_number;
    unsigned char    mode;
    unsigned char    disable;
    unsigned char    data_rate;
    unsigned char    tx_power;
};
```

The fields are described in the following table:

Field	Description
channel_number	Valid value: 172 ~ 184
mode	OP_EXTEND: continuous mode OP_SWITCH: alternating mode
data_rate	The default data rate of the device: 6, 9, 12, 18, 24, 36, 48, 54
tx_power	The default transmit power of the device

4.5 MIB Information

To get MIB information for WME, the following function can be used:

```
int wme_mib_get(struct mib_info *info);
```

The structure of the MIB information is defined as:

```
struct mib_info
{
    unsigned char    entry_type;
    unsigned char    next;
    short         entry_index;
    union
    {
        struct mib_provider_service_info  provider_entry;
        struct mib_user_service_info   user_entry;
        struct mib_wsm_service_info    wsm_entry;
        struct mib_available_service_info available_entry;
    } entry_value;
};
```

The fields are described in the following table:

Field	Description
entry_type	Valid value: PROVIDER_ENTRY, USER_ENTRY, WSM_ENTRY, AVAILABLE_ENTRY
next	If set to 0, the function will try to return MIB value by the entry_index. If set to1, the function will search the next index after the entry_index.
entry_index	The index of MIB
entry_value	The value is dependent on the entry_type. Valid info: struct mib_provider_service_info, struct mib_user_service_info, struct mib_wsm_service_info, struct mib_available_service_info Note: all structures are defined in the header file dot3_common.h.

4.6 Configuring WSA

The WSA information can be configured by calling the following functions

```
int wme_wsa_cfg_get(struct wsa_cfg_info *info);
```

```
int wme_wsa_cfg_set(struct wsa_cfg_info *info);
```

The structure of the WSA configuration information is defined as:

```
struct wsa_cfg_info
{
    unsigned char    repeat_rate;
    signed char      transmit_power;
    unsigned char    advertiser_id[ADVERTISER_ID_SIZE];
    unsigned char    advertiser_id_len;
    unsigned char    country_str[COUNTRY_STR_SIZE];
    unsigned short   router_lifetime;
    unsigned char    ip_prefix[IPV6_SIZE];
    unsigned char    prefix_len;
    unsigned char    default_gateway[IPV6_SIZE];
    unsigned char    primary_dns[IPV6_SIZE];
    unsigned char    secondary_dns[IPV6_SIZE];
    unsigned char    gateway_mac[MAC_SIZE];
    unsigned int     visible_mask;
};
```

The fields are described in the following table:

Field	Description
repeat_rate	The number of WSAs are sent per 5 seconds. The real sending rate is equal to repeats +1. Valid value: 0~ 255 Note: the parameter can't be set by the function, it will be set by the provider service.
transmit_power	The transmit power of WSA
advertiser_id	The description is associated to the device sending WSAs, and the max length is ADVERTISER_ID_SIZE (32 bytes).
advertiser_id_len	The real length of the advertiser_id
country_str	The value indicates the regulatory domain in which the system is operating.

router_lifetime	It indicates the life time of a router.
ip_prefix	It indicates the IPv6 prefix of a router
prefix_len	It indicates the IPv6 prefix length of a router
default_gateway	It indicates the IPv6 address of a router that provides the network connectivity.
primary_dns	It indicates the IPv6 address of a device that provides DNS lookup
secondary_dns	It indicates the IPv6 address of an alternate device that provides DNS lookup
gateway_mac	It indicates the MAC address associated with the default gateway
visible_mask	The bitmask indicates which optional field will be visible in sending WSAs. The bitmask value in defined in the header file dot3_common.h. Valid value: BITMASK_XXXXX

4.7 WSM Packet – Transmission & Reception

To send WSM packets on the specific channel, users must register provider/user services for getting the access of the specific channel. Following function can be used:

```
int wme_wsm_send(wme_handle_t *handle, struct out_wsm *wsm);
```

The structure of WSM Tx information is defined as:

```
struct out_wsm
{
    unsigned char    channel_number;
    unsigned char    data_rate;
    unsigned char    txpwr_level;
    signed char      user_priority;
    unsigned int     psid;
    unsigned char    dest_mac[MAC_SIZE];
```

```
    struct
    {
        unsigned char    channel:1,
                         rate:1,
                         power:1,
                         reserved:5;
    } extensions;
    unsigned char    wsm_type;
    unsigned char    data[WSM_MAX_SIZE];
    unsigned short       length;
};
```

The fields are described in the following table:

Field	Description
channel_number	The channel number in the packet Valid value: 172 ~ 184
data_rate	The data rate in the packet Valid value: 6, 9, 12, 18, 24, 36, 48, 54
txpwr_level	The tx power in the sending packet
user_priority	The priority in the sending packet. The value maps to corresponding EDCA. Valid value: 0 ~ 7
psid	The psid of sending packet Valid value: defined by IEEE 1609.3
dest_mac[MAC_SIZE]	The peer's MAC address
extensions	Indicate which of the WSM header extension fields should be included in the packet
wsm_type	The type of receiving WSM packet Valid value: EID_WSMP
data	The payload of sending packet
data_len	The payload length of sending packet

If the user needs to receive WSMs asynchronously, the `wme_wsm_recv_notify` function can be used. This function registers a receive WSM callback function. Whenever a WSM is received, the callback function is executed. Below is the function prototype:

```
int wme_wsm_recv_notify(wme_handle_t *handle, void
```

```
(*cb_func)(wme_handle_t *handle));
```

Before receiving WSM packets, users must register the wsm service for the desired PSID. Following function can be used:

```
int wme_wsm_recv(wme_handle_t *handle, struct in_wsm *wsm);
```

The structure of WSM Rx information is defined as:

```
struct in_wsm
{
    unsigned char    version;
    unsigned char    channel_number;
    unsigned char    data_rate;
    unsigned char    txpwr_level;
    unsigned int     psid;
    unsigned char    src_mac[MAC_SIZE];
    struct
    {
        unsigned char channel:1,
                      rate:1,
                      power:1,
                      reserved:5;
    } extensions;
    unsigned char    wsm_type;
    unsigned int     ex_header;
    signed char      rssi;
    unsigned char    data[WSM_MAX_SIZE];
    unsigned short       length;
};
```

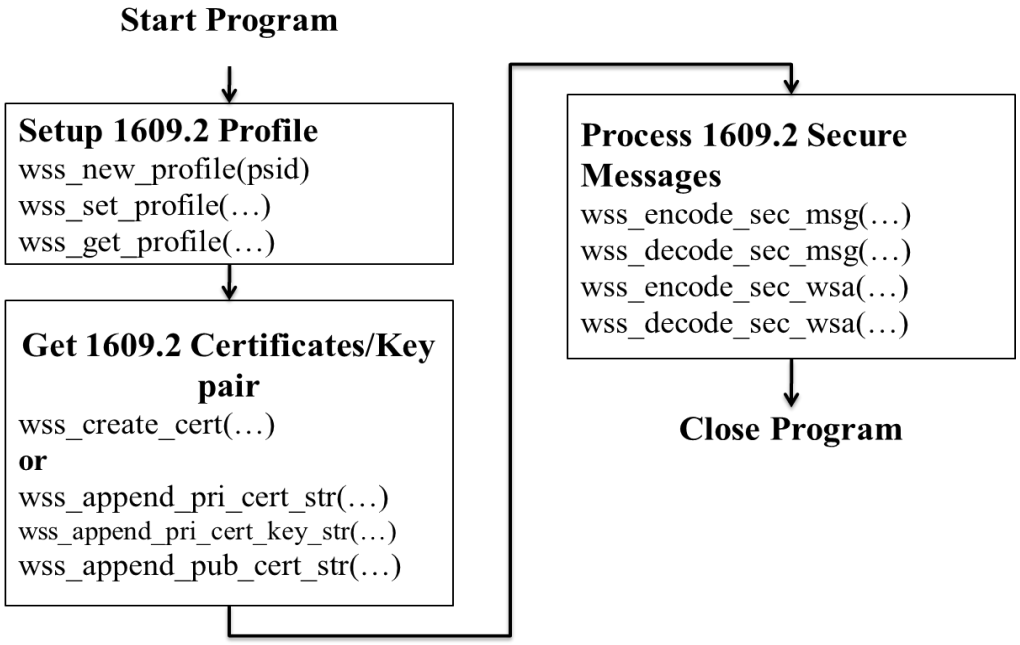
The fields are described in the following table:

Field	Description
version	The WSM version in the packet
channel_number	The channel number in the packet Valid value: 172 ~ 184
data_rate	The data rate in the packet

	Valid value: 6, 9, 12, 18, 24, 36, 48, 54
txpwr_level	The sending tx power in the packet
psid	The psid in the packet Valid value: defined by IEEE 1609.3
src_mac[MAC_SIZE];	The peer's MAC address
extensions	Indicate which of the WSM header extension fields is included in the packet
wsm_type	The WSM type in the packet Valid value: EID_WSMP
rssi	The receiving power in the packet
data	The payload in the packet
data_len	The payload length in the packet

5 IEEE 1609.2 PROGRAMMING API

The process for using IEEE 1609.2 security service API is shown below:



There are 3 stages for using the security services in 1609.2. Firstly, a profile must be set, which specifies how to process secure data. Next, a certificate/key pair must be acquired and finally secure data processing. The APIs for each of these parts are described below:

1) Profile related APIs

The profile is a set of settings which would specify the actions for application to be taken when dealing with the secure messages. Each application that wants to use security service should set the profile for the using PSID before using other APIs. The details of these APIs are described in Section 5.1.

2) Certificate related APIs

For signing/verifying/encrypting/decrypting secure messages, the application would need corresponding certificates. There are two ways to get the certificate: load existing certificates and generate/request new certificates. For loading existing certificates, the user needs to transform the content of 1609.2 certificates into binary form and then use API to add the certificates into Security Data

Store (SDS). For acquiring new certificates, the user can self-generate or send certificate request to an existing Certificate Authority (CA). The details of these APIs are described in Section 5.2.

3) Secure Message related APIs

These APIs are used to send/receive secure messages and the details are described in Section 5.3.

5.1 Setting Profile

The functionality of 1609.2 secure message processing is determined by the settings of the profile described in IEEE 1609.2.

- To set a profile with default values for specified PSID, we have:
```
int wss_new_profile(unsigned int psid);
```

- To delete the profile with the specified PSID, we have:
```
int wss_del_profile(unsigned int psid);
```

- To get the PSIDs of the current in-use profiles, we have:
```
int wss_query_profile(unsigned int *avail_psids_num, unsigned int
*avail_psids);
```

- To get value of specified field of the profile with PSID
```
int wss_get_profile(unsigned int psid, profile_id pf_id,
profile_value_type *value);
```

- To set value of specified field of the profile with PSID, we have:
```
int wss_set_profile(unsigned int psid, profile_id pf_id,
profile_value_type *value);
```

Note: The input parameter "`psid`" is the PSID of the application to use security service. And the other input parameter "`profile_value_type`" is defined as:

```
typedef struct{
    union{
        /* Sending */
        char        usedot2;
        char        dot2_version;
        char        signflag;
        char        incgentime;
        char        incexptime;
```

```
char      incgenloc;
unsigned int  transcertchainint;
int    certchainlen;
char      encflag;
char      fastverflag;
char      ecpointformat;
/* Receiving */
char      verflag;
char      checkgentime;
```

Field	Valid Range	Default	Description
PF_ID_USEDOT2	0: disable 1: enable	1	Use 1609.2 secure service or not
PF_ID_DOT2_VERSION	2: 1609.2 v2	2	1609.2 version
PF_ID_SIGNFLAG	0: not sign 1: always sign 2: adaptive	0	Sign data or not
PF_ID_INCGENTIME	0: exclude 1: include	0	If sign data, include generation time field in header or not
PF_ID_INCEXPTIME	0: exclude 1: include	0	If sign data, include expiration time field in header or not
PF_ID_INCGENLOC	0: exclude 1: include	0	If sign data, include generation location field in header or not
PF_ID_TRANSCERTCHAININT	Positive integer time interval in ms	1000	Time interval for transmitting certificate chain
PF_ID_ENCFLAG	0: not encrypt 1: always encrypt 2: adaptive	0	Encrypt data or not
PF_ID_FASTVERFLAG	0: uncompressed 1: compressed 2: only x-coor.	1	Specify whether to use point compression or just x-coordinate in the signature of signed data
PF_ID_CERTCHAINLEN	-256 ~ 256 Excluding 0	-1	This value is used only when not sending full certificate chain and the “signer_type” is set to 2 (certificate chain). It decides the length of certificate chain to be attached. If positive, includes that number of certificates from the

			chain. If negative with value –*n*, omits the top *n* certificates, starting with the root CA certificate, and includes the rest of the chain
PF_ID_ECPOINTFORMAT	0: uncompressed 1: Compressed	1	Specify whether to use point compression in elliptic curve points of the encrypted data or not

```
char        gentimesrc;
char        checkexptime;
char        exptimesrc;
char        checkgenloc;
char        genlocsrc;
char        checkreplay;
```

Field	Valid Range	Default	Description
PF_ID_VERFLAG	0: not verify 1: always verify 2: adaptive	0	Verify incoming data (if it is signed data) or not
PF_ID_CHECKGENTIME	0: not check 1: check	0	If verify, check generation time field or not
PF_ID_GENTIMESRC	0: from header 1: payload [not supported]	0	Generation time of the signed data is obtained by header or payload
PF_ID_CHECKEXPTIME	0: not check 1: check	0	If verify, check expiration time field or not
PF_ID_EXPTIMESRC	0: from header 1: payload [not supported]	0	Expiration time of the signed data is obtained by header or payload
PF_ID_CHECKGENLOC	0: not check 1: check	0	If verify, check generation location field or not
PF_ID_GENLOCSRC	0: from header 1: payload [not supported]	0	Generation location of the signed data is obtained by header or payload
PF_ID_CHECKREPLAY	0: not check 1: check	0	Used to decide whether to perform the replay check or not
PF_ID_MSGVALPERIOD	Any time value in seconds	5.0	The period in seconds after a message's generation time for which it is of interest to the recipient
PF_ID_MSGVALDIS	Any distance value in meters	10.0	The distance in meters that a recipient may be from a generation location and still accept the data
PF_ID_GENTIMECONFMUL	Real-valued number	0.0	Allow the receiver to give a policy for use of the GenerationTimeConfidence field
PF_ID_CRLTIMETOLERANCE	Any time value in seconds [currently not used]	600	A data will only be accepted if the CRLs associated with the message's certificate chain are either up-to-date, or overdue (based on the next_crl field in the ToBeSignedCrl) by no more than this amount

```
double msgvalperiod;
double msgvaldis;
double gentimeconfmul;
int    crltimetolerance;
/* Management */
char       signalg;
```

```
        char      encalg;
        char      pktranstype;
    }u;
}profile_value_type;
```

The sending-related fields are described in the following table:

The receiving-related fields are described in the following table:

The management-related fields are described in the following table:

Field	Valid Range	Default	Description
PF_ID_SIGNALG	0: ECDSA-224 1: ECDSA-256	1	Specify the signing algorithm to use
PF_ID_ENCALG	2: ECIES-256	2	Specify the encryption algorithm to use
PF_ID_PKTRANSTYPE	0: explicit 1: implicit 2: both	0	Specify the type of certificates to use

5.2 Acquiring Certificate

For using IEEE 1609.2 security services, user would need the certificates and the corresponding public or private key pairs. The library supports the primitives to generate user-specified certificates and the simple security data store (SDS) for management certificates.

5.2.1 Certificate Generation – Self Generation

The user may get the default certificate settings by using the following primitive:

```
int wss_create_cert(struct cert_fields *cert, unsigned char
*cert_str, unsigned int *cert_len, unsigned int *ecdsa_pri_index,
unsigned int *ecies_pri_index);
```

The structure of certificate field is defined as:

```
struct cert_fields
{
    char version;
    unsigned char subject_type;
    unsigned int psid_num;
    unsigned int psid[MAX_NUM_OF_PERMISSIONS];
    unsigned int expiration;
```

```
    unsigned int  start_time;
};
```

The fields are described in the following table:

Field	Valid Range	Description
version	2: 1609.2 v2	Specify which version of certificate to generate, currently only support version 2
subject_type	255: root ca 1: id not localized 4: wsa	Specify which type of certificate to generate, currently only support 1 type of ca (root ca) and 2 types for end entity
psid_num	0~8	Specify the permitted number of PSIDs the certificate can process with
psid	Any valid PSID	Specify the permitted PSIDs
expiration	Any valid time value	Specify the expiration time of the generated certificate. If set to 0, use default values: 2 years from now for ca, 1 year from now for end entity
start_time	Any valid time value	Specify the start time of the generated certificate. If set to 0, use current time.

For self-generated certificates, user needs to use the following primitives to add these certificates into SDS (Secure Data Store) before using these certificates to sign > verify > encrypt > decrypt secure messages:

```
int wss_append_pri_cert_key_str(unsigned char *cert_str, unsigned
int cert_len, unsigned char *ecdsa_pri_key, unsigned char
*ecies_pri_key);
int wss_append_pub_cert_str(char *cert_str, unsigned int
cert_len);
```

5.2.2 Security Data Store (Certificate Pool)

We support simple SDS (Security Data Store) for secure message processing. The SDS has a cache (pool) for holding the used private/public certificates. The primitives for managing the SDS information are listed below.

- To get/set the current index of the pool

```
int wss_set_cur_pri_pool_ind(unsigned int value);
int wss_get_cur_pri_pool_ind(unsigned int *value);
int wss_get_cur_pub_pool_ind(unsigned int *value);
```

- To get current used number of the pool

```
int wss_get_pri_pool_used_num(unsigned int *value);
int wss_get_pub_pool_used_num(unsigned int *value);
```

- To get the certificate information of the pool

```
int wss_get_pri_cert_info(unsigned int ind, struct cert_info
*cert);
int wss_get_pub_cert_info(unsigned int ind, struct cert_info
*cert);
```

The structure of certificate field is defined as:

```
struct cert_info
{
    struct cert_fields sub_info;
    char digest[CERT_DIGEST_BYTES];
    char issuer_id[CERT_DIGEST_BYTES];
    unsigned char signing_alg;
    unsigned int crl_series;
    char verificaton_key[MAX_COMPRESSED_PUBLIC_KEY_BYTES];
    char encryption_key[MAX_COMPRESSED_PUBLIC_KEY_BYTES];
    char cert_str[MAX_CERT_LENGTH];
    unsigned int cert_str_len;
};
```

Field	Description
sub_info	Related information (e.g. subject type, expiration, …etc) of the cert. which is filled in the structure cert_fields
digest	The lowest 8 bytes of the SHA256 hash of the cert. string
issuer_id	The lowest 8 bytes of the SHA256 hash of the cert. string which issues this cert. (i.e., signer cert., CA)
signing_alg	The signing algorithm used of the cert.
crl_series	The CRL number of the cert.
verification_key	The binary key string indicating the verification key of the cert.
encryption_key	The binary key string indicating the encryption key of the cert.
cert_str	The binary string of the cer.
cert_str_len	The length in bytes of the binary string of the cert.

The fields are described in the following table:

5.3 Secure Messages

The following primitives provide the user to encode/decode IEEE 1609.2 secure messages. It

would refer the settings of the profile to take related functionalities such as signing, encrypting, or verifying. Also, it would use the related certificates in SDS if needed.

```
int wss_encode_sec_msg(struct encode_in *in_pack, struct
encode_out *out_pack);
int wss_decode_sec_msg(struct decode_in *in_pack, struct
decode_out *out_pack);
```

The structure of the parameters `encode_in` and `encode_out` in `wss_encode_sec_msg` are defined as:

```
struct encode_in
{
    unsigned int msg_len;
    char msg[MAX_IN_MSG_LENGTH];
    unsigned int ext_msg_len;
    char ext_msg[MAX_IN_MSG_LENGTH];
    char adaptive_sign;
    char adaptive_enc;
    unsigned int ssp_len;
    char ssp[MAX_CERT_STR_LENGTH];
    unsigned int psid;
    unsigned long long expiry;
    char attach_cert_type;
    unsigned int recp_num;
    char recp_cert[MAX_NUM_OF_ENC_RECP][CERT_DIGEST_BYTE];
};

struct encode_out{
    unsigned char ret_code;
    unsigned int msg_len;
    char msg[MAX_MSG_LENGTH];
    unsigned int failed_cert_num;
 char failed_recp[MAX_NUM_OF_ENC_RECP][CERT_DIGEST_BYTES];
    struct encode_log_items encode_log;
};

struct encode_log_items{
```

```
unsigned char attach_cert_type;
char sign_cert_digest[CERT_DIGEST_BYTES];
unsigned int sign_cert_str_len;
char sign_cert_str[MAX_CERT_LENGTH];
unsigned long long sign_cert_digest_val;
unsigned long long msg_gen_time_us;
```

Field	Valid Range	Description
msg_len	0 ~ 2048	Specify the length in bytes of the input data
msg	Data string	The data to be encoded in
ext_msg_len	0 ~ 2048	Specify the length in bytes of the external data. Only used if need to form a signed data. If the length is not zero, it will result in a signed data with type equals to “sign_with_partial_payload (9)” or “sign_with external_payload (10)”. [used for signed data]
ext_msg	Data string of external data	The external data to be signed with. See the description of ext_msg_len. [used for signed data]
adaptive_sign	0: not sign 1: sign	If the “PF_ID_SIGNFLAG” field is set to adaptive (2), then the security module will use this parameter to determine whether to sign this packet or not
adaptive_enc	0: not encrypt 1: encrypt	If the “PF_ID_ENCFLAG” field is set to adaptive (2), then the security module will use this parameter to determine whether to encrypt this packet or not.
ssp_len	0~256	Specify the length of the service specific permissions (SSP) [used for signed data]
ssp	Data string	Specify the service specific permissions, which is used to find a permission-consistent certificate to process this packet if need to sign with. [used for signed data]
psid	Any valid PSID value	Specify the PSID to be encoded in and find a permission-consistent certificate to process if need to encode into a secure data. [used for signed and encrypted data]
expiry	Any time value	Specify the expiration time of this data [used for signed data]

attach_cert_type	0: digest 1: certificate 2: certificate chain	Specify the type of signer information in signed data [used for signed data]
recp_num	0~4	Specify the number of recipients for encrypting [used for encrypted data]
recp_cert	Array of certificate digests of the recipients	Specify the certificate digest of each recipient to encrypt data for them [used for encrypted data]

```
    unsigned char msg_gen_time_conf;
    unsigned int signer_cert_exp;
    unsigned char sign_ret_code;
    unsigned char enc_ret_code;
};
```

The fields for `encode_in` structure are defined as the following table:

The fields for `encode_out` structure are defined as the following table:

Field	Description
ret_code	Specify the result of the secure processing of this packet
msg_len	Specify the length in bytes of the encoded data if success
msg	Specify the encoded data if success
failed_cert_num	Specify the number of failed recipients while trying to encrypt with [assigned if encrypted data]
failed_recp	Specify the certificate digests of the recipients which failed to encrypt for them [assigned if encrypted data]
encode_log	Related log information about the secure processing of this packet

The fields for `encode_log_items` structure are defined as the following table:

Field	Description
attach_cert_type	Specify the attached certificate type of this secure data [assigned if signed data]
sign_cert_digest	Specify the digest of the certificate which signed this data [assigned if signed data]
sign_cert_str_len	Specify the length in bytes of the certificate string which signed this data

	[assigned if signed data]
sign_cert_str	Specify the certificate string which signed this data [assigned if signed data]
sign_cert_digest_val	Specify the digest (in integer type) of the certificate which signed this data [assigned if signed data]
msg_gen_time_us	Specify the generation time encoded in this signed data [assigned if signed data]
msg_gen_time_conf	Specify the generation time confidence encoded in this signed data [assigned if signed data
signer_cert_exp	Specify the expiration time of the certificate which signed this data [assigned if signed data]
sign_ret_code	Specify the result codes of the signing process of this data [assigned if signed data]
enc_ret_code	Specify the result codes of the encrypting process of this data [assigned if encrypted data]

The structure of the parameters `decode_in` and `decode_out` in `wss_decode_sec_msg` are defined as:

```
struct decode_in
{
    unsigned int psid;
    unsigned int msg_len;
    char msg[MAX_MSG_LENGTH];
    unsigned int external_msg_len;
    char external_msg[MAX_MSG_LENGTH];
    char adaptive_ver;
};

struct decode_out{
    unsigned char ret_code;
    unsigned int msg_len;
    char msg[MAX_MSG_LENGTH];
    struct decode_log_items decode_log;
};
```

```
struct decode_log_items{
    unsigned char attach_cert_type;
    char sign_cert_digest[CERT_DIGEST_BYTES];
    unsigned int sign_cert_str_len;
    char sign_cert_str[MAX_CERT_LENGTH];
    unsigned long long sign_cert_digest_val;
    unsigned long long msg_gen_time_us;
    unsigned char msg_gen_time_conf;
    unsigned char de_ret_code;
    unsigned char ver_ret_code;
};
```

The fields for `decode_in` structure are defined as the following table:

Field	Valid Range	Description
psid	Any valid PSID value	Specify the PSID to find the profile to determine how to process the data after extract the data from header and trailer
msg_len	0~2048	Specify the length in bytes of the input encoded data
msg	Data string	The content of the encoded data
external_msg_len	0~2048	Specify the length in bytes of the external data. Only used if need to verify a signed data with type equals to “sign_with_partial_payload (9)” or “sign_with external_payload (10)”. [used for signed data]
external_msg	Data string of external data	The external data to be verified with. See the description of ext_msg_len. [used for signed data]
adaptive_ver	0: not verify 1: verify	If the “PF_ID_VERFLAG” field is set to adaptive (2), then the security module will use this parameter to determine whether to verify this packet or not if it is signed

The fields for `decode_out` structure are defined as the following table:

Field	Description
ret_code	Specify the result of the secure processing of this packet
msg_len	Specify the length in bytes of the decoded data if success

msg	Specify the decoded data if success
decode_log	Related log information about the secure processing of this packet

The fields for `decode_log_items` structure are defined as the following table:

Field	Description
attach_cert_type	Specify the attached certificate type of this secure data [assigned if signed data]
sign_cert_digest	Specify the digest of the certificate which signed this data [assigned if signed data]
sign_cert_str_len	Specify the length in bytes of the certificate string which signed this data [assigned if signed data]
sign_cert_str	Specify the certificate string which signed this data [assigned if signed data]
sign_cert_digest_val	Specify the digest (in integer type) of the certificate which signed this data [assigned if signed data]
msg_gen_time_us	Specify the generation time extracted from this signed data [assigned if signed data]
msg_gen_time_conf	Specify the generation time confidence extracted from this signed data [assigned if signed data
de_ret_code	Specify the result codes of the data extracting process of this data
ver_ret_code	Specify the result codes of the verification process of this data [assigned if signed data and perform verification]

5.4 Secure WSA

The following primitives provide the user to encode/decode IEEE 1609.2 secure messages. It would refer the settings of the profile to take related functionalities such as signing, encrypting, or verifying. Also, it would use the related certificates in SDS if needed.

```
int wss_encode_sec_wsa(struct encode_sec_wsa_pack *in_pack, struct
encode_sec_wsa_result *out_pack);
int wss_decode_sec_wsa(struct decode_sec_wsa_pack *in_pack, struct
decode_sec_wsa_result *out_pack);
```

The structure of the parameter `encode_sec_wsa_pack` in the above primitive is defined as:

```
struct encode_sec_wsa_pack
{
    unsigned int     wsa_data_len;
    unsigned char    wsa_data[MAX_IN_MSG_LENGTH];
    unsigned int     permission_num;
    unsigned int     psid[MAX_NUM_OF_PERMISSIONS];
    unsigned char    priority[MAX_NUM_OF_PERMISSIONS];
    unsigned long long lifetime;
};
```

The fields for `encode_sec_wsa_pack` structure are defined as the following table:

Field	Valid Range	Description
wsa_data_len	0~2048	Specify the length in bytes of the input WSA content
wsa_data	Data string	The WSA content to be encoded in
permission_num	0~8	The number of PSIDs contained in the input WSA
psid	Any valid PSID value	Specify the PSIDs in the WSA to be encoded in, find a permission-consistent certificate to process if need to encode into a secure WSA. [used for signed WSA]
priority	Any valid priority value	Specify the priorities of the PSIDs in the WSA to be encoded in, find a permission-consistent certificate to process if need to encode into a secure WSA. [used for signed WSA]
expiry	Any time value	Specify the expiration time of this data [used for signed data]

The structure of the parameter `encode_sec_wsa_result` in the above primitive is defined as:

```
struct encode_sec_wsa_result
{
    unsigned char    ret_code;
    unsigned int     sign_ret_code;
    unsigned int     signed_wsa_len;
    unsigned char    signed_wsa[MAX_MSG_LENGTH];
};
```

The fields for `encode_sec_wsa_result` structure are defined as the following table:

Field	Description
ret_code	Specify the result of the secure processing of this packet
sign_ret_code	Specify more detailed result of the signing process of this packet [assigned if signed WSA]
signed_wsa_len	Specify the length in bytes of the encoded WSA if success
signed_wsa	Specify the encoded WSA if success

The structure of the parameter `decode_sec_wsa_pack` in the above primitive is defined as:

```
struct decode_sec_wsa_pack
{
    unsigned int     signed_wsa_len;
    unsigned char    signed_wsa[MAX_MSG_LENGTH];
    unsigned char    secure_handle;
};
```

The fields for `decode_sec_wsa_pack` structure are defined as the following table:

Field	Valid Range	Description
signed_wsa_len	0~2048	Specify the length in bytes of the input encoded WSA content
signed_wsa	Data string	The input encoded WSA content
secure_handle	0~255	The sequence number of the input WSA [not used]

The structure of the parameter `decode_sec_wsa_result` in the above primitive is defined as:

```
struct decode_sec_wsa_result
{
    unsigned char    ret_code;
    unsigned char    de_ret_code;
    unsigned char    ver_ret_code;
    unsigned int     wsa_data_len;
    unsigned char    wsa_data[MAX_MSG_LENGTH];
};
```

The fields for `decode_sec_wsa_result` structure are defined as the following table:

Field	Description
ret_code	Specify the result of the secure processing of this packet
de_ret_code	Specify more detailed result of the data extracting process of this packet

ver_ret_code	Specify the result codes of the verification process of this WSA [assigned if signed WSA and perform verification]
wsa_data_len	Specify the length in bytes of the extracted WSA if success
wsa_data	Specify the extracted WSA if success

6 SAE J2735 BSM PROGRAMMING API

To support interoperability among DSRC applications, SAE J2735 standard defines standardized messages. One of them is Basic Safety Message (BSM). Following are the BSM-related primitives:

```
int j2735_bsm_encode(bsm_standard_item *input_item ,char
output_buf[] ,ssize_t *output_buf_size ,unsigned char print_level);
int j2735_bsm_decode(bsm_standard_item *output_item , char
input_buf[] , size_t input_buf_size, unsigned char print_level);
```

The structure of the parameter `bsm_standard_item` in the above primitive is defined as:

```
struct bsm_standard_item_str
{
    unsigned char mask_safetyExt:1,
                  mask_status:1,
                  rest:6;
    BSMblob_item *blob1;
    VehicleSafetyExtension_item *safetyExt;
    VehicleStatus_item *status;
}bsm_standard_item;
```

The fields related to `j2735_bsm_encode` are defined as the following table:

Field	Description
input_item	The input BSM data elements
output_buf	If the return value is 0, output_buf will be a BSM encoded buffer.
output_buf_size	If the return value is 0, output_buf_size will be the size of the BSM encoded buffer.
print_level	PRINT_NON: print nothing PRINT_BASIC: print encoded BSM struct

The fields related to `j2735_bsm_decode` are defined as the following table:

Field	Description
output_item	If the return value is 0, output_item will be the decoded bsm_standard_item.
input_buf	The BSM encoded buffer

input_buf_size	The size of BSM encoded buffer
print_level	PRINT_NON: print nothing PRINT_BASIC: print decoded BSM struct