Contents

1 How to start ........................................................................................................... 11
  1.1 The Samples Projects ......................................................................................... 11
  1.2 Build the Sample Project .................................................................................... 11
      1.2.1 IAR: Open and build a sample workspace file ............................................. 11
  1.3 Start Sample Application ..................................................................................... 12
      1.3.1 How to Start Sample main() ....................................................................... 13
2 Socket Communication ........................................................................................... 17
  2.1 Test Setup ........................................................................................................... 17
  2.2 DA16200 ........................................................................................................... 17
      2.2.1 Test-DUT ..................................................................................................... 18
  2.3 TCP Server ......................................................................................................... 18
      2.3.1 How to Run .................................................................................................. 18
      2.3.2 How it Works ............................................................................................... 18
      2.3.3 Details ........................................................................................................ 18
      2.3.3.1 Connection ............................................................................................. 19
      2.3.3.2 Data Transmission .................................................................................. 20
      2.3.3.3 Disconnection ......................................................................................... 21
      2.3.3.4 To stop listening on a server port ............................................................. 22
  2.4 TCP Server in DPM ............................................................................................ 22
      2.4.1 How to Run .................................................................................................. 22
      2.4.2 How it Works ............................................................................................... 23
      2.4.3 Details ........................................................................................................ 23
      2.4.3.1 Registration ............................................................................................. 23
      2.4.3.2 Data Transmission .................................................................................. 24
  2.5 TCP Client .......................................................................................................... 24
      2.5.1 How to Run .................................................................................................. 24
      2.5.2 How it Works ............................................................................................... 25
      2.5.3 Details ........................................................................................................ 25
      2.5.3.1 Registration ............................................................................................. 25
      2.5.3.2 Data Transmission .................................................................................. 26
      2.5.3.3 Disconnection ......................................................................................... 27
  2.6 TCP Client in DPM ............................................................................................ 27
      2.6.1 How to Run .................................................................................................. 27
      2.6.2 How it Works ............................................................................................... 28
      2.6.3 Details ........................................................................................................ 28
      2.6.3.1 Registration ............................................................................................. 28
      2.6.3.2 Data Transmission .................................................................................. 29
  2.7 TCP Client with KeepAlive ................................................................................. 29
      2.7.1 How to Run .................................................................................................. 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.14.3.1</td>
<td>Registration</td>
<td>50</td>
</tr>
<tr>
<td>2.14.3.2</td>
<td>TLS Handshake</td>
<td>51</td>
</tr>
<tr>
<td>2.14.3.3</td>
<td>Data Transmission</td>
<td>52</td>
</tr>
<tr>
<td>2.15</td>
<td>TLS Client in DPM</td>
<td>53</td>
</tr>
<tr>
<td>2.15.1</td>
<td>How to Run</td>
<td>53</td>
</tr>
<tr>
<td>2.15.2</td>
<td>How it Works</td>
<td>53</td>
</tr>
<tr>
<td>2.15.3</td>
<td>Details</td>
<td>54</td>
</tr>
<tr>
<td>2.15.3.1</td>
<td>Registration</td>
<td>54</td>
</tr>
<tr>
<td>2.15.3.2</td>
<td>TLS Setup</td>
<td>55</td>
</tr>
<tr>
<td>2.15.3.3</td>
<td>Data Transmission</td>
<td>55</td>
</tr>
<tr>
<td>2.16</td>
<td>DTLS Server</td>
<td>56</td>
</tr>
<tr>
<td>2.16.1</td>
<td>How to Run</td>
<td>56</td>
</tr>
<tr>
<td>2.16.2</td>
<td>How it Works</td>
<td>56</td>
</tr>
<tr>
<td>2.16.3</td>
<td>Details</td>
<td>57</td>
</tr>
<tr>
<td>2.16.3.1</td>
<td>Initialization</td>
<td>57</td>
</tr>
<tr>
<td>2.16.3.2</td>
<td>DTLS Handshake</td>
<td>59</td>
</tr>
<tr>
<td>2.16.3.3</td>
<td>Data Transmission</td>
<td>60</td>
</tr>
<tr>
<td>2.17</td>
<td>DTLS Server in DPM</td>
<td>61</td>
</tr>
<tr>
<td>2.17.1</td>
<td>How to Run</td>
<td>61</td>
</tr>
<tr>
<td>2.17.2</td>
<td>How it Works</td>
<td>61</td>
</tr>
<tr>
<td>2.17.3</td>
<td>Details</td>
<td>62</td>
</tr>
<tr>
<td>2.17.3.1</td>
<td>Registration</td>
<td>62</td>
</tr>
<tr>
<td>2.17.3.2</td>
<td>DTLS Setup</td>
<td>62</td>
</tr>
<tr>
<td>2.17.3.3</td>
<td>Data Transmission</td>
<td>64</td>
</tr>
<tr>
<td>2.18</td>
<td>DTLS Client</td>
<td>64</td>
</tr>
<tr>
<td>2.18.1</td>
<td>How to Run</td>
<td>64</td>
</tr>
<tr>
<td>2.18.2</td>
<td>How it Works</td>
<td>65</td>
</tr>
<tr>
<td>2.18.3</td>
<td>Details</td>
<td>65</td>
</tr>
<tr>
<td>2.18.3.1</td>
<td>Initialization</td>
<td>65</td>
</tr>
<tr>
<td>2.18.3.2</td>
<td>DTLS Handshake</td>
<td>66</td>
</tr>
<tr>
<td>2.18.3.3</td>
<td>Data Transmission</td>
<td>67</td>
</tr>
<tr>
<td>2.19</td>
<td>DTLS Client in DPM</td>
<td>68</td>
</tr>
<tr>
<td>2.19.1</td>
<td>How to Run</td>
<td>68</td>
</tr>
<tr>
<td>2.19.2</td>
<td>How it Works</td>
<td>68</td>
</tr>
<tr>
<td>2.19.3</td>
<td>Details</td>
<td>69</td>
</tr>
<tr>
<td>2.19.3.1</td>
<td>Registration</td>
<td>69</td>
</tr>
<tr>
<td>2.19.3.2</td>
<td>DTLS Setup</td>
<td>70</td>
</tr>
<tr>
<td>2.19.3.3</td>
<td>Data Transmission</td>
<td>71</td>
</tr>
<tr>
<td>2.20</td>
<td>Connection Management</td>
<td>71</td>
</tr>
<tr>
<td>2.20.1</td>
<td>TCP Disconnection</td>
<td>71</td>
</tr>
<tr>
<td>2.20.2</td>
<td>AP Disassociation</td>
<td>72</td>
</tr>
<tr>
<td>2.21</td>
<td>CoAP Client</td>
<td>73</td>
</tr>
<tr>
<td>2.21.1</td>
<td>How to Run</td>
<td>73</td>
</tr>
<tr>
<td>2.21.2</td>
<td>CoAP Client Initialization</td>
<td>74</td>
</tr>
<tr>
<td>2.21.3</td>
<td>CoAP Client Deinitialization</td>
<td>74</td>
</tr>
<tr>
<td>2.21.4</td>
<td>CoAP Client Request and Response</td>
<td>75</td>
</tr>
<tr>
<td>2.21.4.1</td>
<td>CoAP URI and Proxy-URI</td>
<td>75</td>
</tr>
</tbody>
</table>
DA16200 Example Application Guide

3 Peripheral Examples

3.1 UART
   3.1.1 How to Run ................................................... 82
   3.1.2 Application Initialization ................................. 82
   3.1.3 Data Read ..................................................... 83
   3.1.4 Data Write ................................................... 84
3.2 GPIO
   3.2.1 How to Run ................................................... 84
   3.2.2 Operation ..................................................... 84
3.3 GPIO Retention .................................................... 85
   3.3.1 How to Run ................................................... 85
   3.3.2 Operation ..................................................... 86
3.4 I2C
   3.4.1 How to Run ................................................... 86
   3.4.2 Operation ..................................................... 86
3.5 I2S
   3.5.1 How to Run ................................................... 88
   3.5.2 User Thread ................................................... 88
   3.5.3 Operation ..................................................... 88
3.6 PWM
   3.6.1 How to Run ................................................... 89
   3.6.2 Operation ..................................................... 89
3.7 ADC
   3.7.1 How to Run ................................................... 91
   3.7.2 Operation ..................................................... 91
3.8 SPI
   3.8.1 How to Run ................................................... 92
   3.8.2 Operation ..................................................... 92
3.9 SDIO
   3.9.1 How to Run ................................................... 94
   3.9.2 Operation ..................................................... 94
3.10 SD/eMMC ............................................................ 95
   3.10.1 How to Run ................................................... 95
   3.10.2 Operation ..................................................... 95

4 Advanced Examples .................................................. 97

4.1 DNS Query .......................................................... 97
   4.1.1 How to Run ................................................... 97
   4.1.2 Application Initialization .................................. 97
DA16200 Example Application Guide

4.1.3 Get Single IPv4 Address ................................................................. 98
4.1.4 Get Multiple IPv4 Addresses ......................................................... 99

4.2 SNTP and Get Current Time ......................................................... 100
4.2.1 How to Run .................................................................................. 100
4.2.2 Operation .................................................................................... 101

4.3 SNTP and Get Current Time in DPM Function ............................. 102
4.3.1 How to Run .................................................................................. 102
4.3.2 Operation .................................................................................... 103

4.4 HTTP Client ................................................................................... 106
4.4.1 How to Run .................................................................................. 106
4.4.2 Operation .................................................................................... 106

4.5 HTTP Client in DPM Function ....................................................... 109
4.5.1 How to Run .................................................................................. 109
4.5.2 Operation .................................................................................... 109

4.6 HTTP Server .................................................................................. 112
4.6.1 How to Run .................................................................................. 112
4.6.2 Operation .................................................................................... 112

4.7 ThreadX API Sample .................................................................... 116
4.7.1 How to Run .................................................................................. 116
4.7.2 Sample Overview ........................................................................ 116
4.7.3 Thread Creation and Resource Initialization .............................. 116
4.7.4 Initial Execution .......................................................................... 116
4.7.5 Threads Operation in Detail ......................................................... 118

4.8 RTC Timer with DPM Function ..................................................... 124
4.8.1 How to Run .................................................................................. 124
4.8.2 Application Initialization .............................................................. 124
4.8.3 Timer Creation: DPM Sleep mode 1 ......................................... 125
4.8.4 Timer Creation: DPM Sleep mode 2 ......................................... 125
4.8.5 Timer Creation: DPM Sleep mode 3 ......................................... 125
4.8.6 Timer Creation: DPM Sleep mode 3 ......................................... 127

4.9 Get SCAN Result Sample ............................................................. 127
4.9.1 How to Run .................................................................................. 127
4.9.2 Sample Overview ........................................................................ 128
4.9.3 Application Initialization ............................................................ 128
4.9.4 Get SCAN Result ........................................................................ 128

4.10 User sflash Read / Write Example ............................................. 129
4.10.1 How to Run .................................................................................. 129
4.10.2 User Thread ................................................................................ 130
4.10.3 Application Initialization ............................................................ 130
4.10.4 Sflash Read and Write ............................................................... 130

4.11 Crypto Algorithms - AES ............................................................ 132
4.11.1 How to Run ................................................................................ 132
4.11.2 Application Initialization ............................................................ 133
4.11.3 Crypto Algorithm for AES-CBC-128, 192, and 256 ................. 133
4.11.4 Crypto Algorithm for AES-CFB128-128, 192, and 256 ............. 134
4.11.5 Crypto Algorithm for AES-ECB-128, 192, and 256 .................... 136
4.11.6 Crypto Algorithm for AES-CTR-128........................................ 137
DA16200 Example Application Guide

4.11.7  Crypto Algorithm for AES-CCM-128, 192, and 256  ............................................... 137
4.11.8  Crypto Algorithm for AES-GCM-128, 192, and 256 ............................................... 138
4.11.9  Crypto Algorithm for AES-OFB-128, 192, and 256 ............................................... 139
4.12    Crypto Algorithms - DES ......................................................................................... 140
  4.12.1  How to Run ........................................................................................................... 140
  4.12.2  Application Initialization ....................................................................................... 140
  4.12.3  Crypto Algorithm for DES-CBC-56, DES3-CBC-112, and 168 ......................... 141
4.13    Crypto Algorithms – HASH & HMAC ..................................................................... 142
  4.13.1  How to Run ........................................................................................................... 142
  4.13.2  Application Initialization ....................................................................................... 143
  4.13.3  Crypto Algorithm for SHA-1 Hash ...................................................................... 144
  4.13.4  Crypto Algorithm for SHA-224 Hash ................................................................. 145
  4.13.5  Crypto Algorithm for SHA-256 Hash ................................................................. 146
  4.13.6  Crypto Algorithm for SHA-384 Hash ................................................................. 146
  4.13.7  Crypto Algorithm for SHA-512 Hash ................................................................. 147
  4.13.8  Crypto Algorithm for MD5 Hash ........................................................................ 148
  4.13.9  Crypto Algorithm for Hash and HMAC with the Generic Message-Digest Wrapper ................................................................. 149
4.14    Crypto Algorithms – DRBG ....................................................................................... 155
  4.14.1  How to Run ........................................................................................................... 155
  4.14.2  Application Initialization ....................................................................................... 156
  4.14.3  CTR_DRBG with Prediction Resistance .............................................................. 156
  4.14.4  CTR_DRBG without Prediction Resistance .......................................................... 157
  4.14.5  HMAC_DRBG with Prediction Resistance ........................................................... 158
  4.14.6  HMAC_DRBG Without Prediction Resistance ...................................................... 160
4.15    Crypto Algorithms – ECDSA ..................................................................................... 160
  4.15.1  How to Run ........................................................................................................... 160
  4.15.2  Application Initialization ....................................................................................... 161
  4.15.3  Generates ECDSA Key Pair and Verifies ECDSA Signature .............................. 161
4.16    Crypto Algorithms – Diffie-Hellman Key Exchange .............................................. 163
  4.16.1  How to Run ........................................................................................................... 164
  4.16.2  Application Initialization ....................................................................................... 164
  4.16.3  Load Diffie-Hellman Parameters ....................................................................... 164
  4.16.4  How Diffie-Hellman Works .............................................................................. 165
4.17    Crypto Algorithms – RSA PKCS#1 .......................................................................... 168
  4.17.1  How to Run ........................................................................................................... 168
  4.17.2  Application Initialization ....................................................................................... 168
  4.17.3  How RSA PKCS#1 Works .................................................................................. 168
4.18    Crypto Algorithms – ECDH ....................................................................................... 172
  4.18.1  How to Run ........................................................................................................... 172
  4.18.2  Application Initialization ....................................................................................... 173
  4.18.3  How ECDH Key Exchange Works ................................................................... 173
4.19    Crypto Algorithms – KDF ......................................................................................... 177
  4.19.1  How to Run ........................................................................................................... 177
  4.19.2  Application Initialization ....................................................................................... 177
  4.19.3  How KDF Works ............................................................................................... 177
4.20    Crypto Algorithms – Public Key Abstraction Layer .............................................. 178
DA16200 Example Application Guide

4.20.1 How to Run ................................................................. 178
4.20.2 User Thread ................................................................. 179
4.20.3 Application Initialization ............................................. 179
4.20.4 How Public Key Abstraction Layer is Used .......... 181

4.21 Crypto Algorithms – Generic Cipher Wrapper .............................................. 190
4.21.1 How to Run ................................................................. 190
4.21.2 Application Initialization ............................................. 191
4.21.3 How Generic Cipher Wrapper is Used .............................................. 191

Appendix A ........................................................................ 198

Revision History ................................................................. 199

Figures

Figure 1: Overall Test Setup .................................................................. 11
Figure 2: Start IAR workbench ................................................................ 11
Figure 3: Open the DA16200 Sample workspace .................................. 12
Figure 4: Rebuild or Make to compile ................................................ 12
Figure 5: Compiler feature for Sample Project .................................. 13
Figure 6: The flow of sample code ...................................................... 13
Figure 7: Startup Files on IAR project .............................................. 14
Figure 8: Sample_apps file on IAR project ........................................ 15
Figure 9: Overall Test Setup .............................................................. 17
Figure 10: DA16200 EVB – AP Connection Done .................................. 17
Figure 11: Workflow of TCP Server ................................................... 18
Figure 12: Workflow of TCP Server in DPM ......................................... 23
Figure 13: Workflow of TCP Client .................................................... 25
Figure 14: Workflow of TCP Client in DPM ........................................ 28
Figure 15: Workflow of TCP Client with KeepAlive ................................... 30
Figure 16: Workflow of TCP Client with KeepAlive in DPM .................. 34
Figure 17: Workflow of UDP Socket .................................................. 36
Figure 18: Workflow of UDP Server in DPM ........................................ 39
Figure 19: Workflow of UDP Client in DPM ......................................... 41
Figure 20: Workflow of TLS Server .................................................... 43
Figure 21: Workflow of TLS Server in DPM ......................................... 47
Figure 22: Workflow of TLS Client ..................................................... 50
Figure 23: Workflow of TLS Client in DPM ......................................... 53
Figure 24: Workflow of DTLS Server .................................................. 56
Figure 25: Workflow of DTLS Server in DPM ....................................... 61
Figure 26: Workflow of DTLS Client ................................................... 65
Figure 27: Workflow of DTLS Client in DPM ........................................ 69
Figure 28: AP Disconnection ............................................................. 72
Figure 29: Scenario for AP Reassociation in DPM Mode ....................... 73
Figure 30: SPI Loopback Communication ........................................... 73
Figure 31: SDIO & SD/eMMC Connector .............................................. 92
Figure 32: The result of the DA16200 HTTP Server .............................. 114
Figure 33: The DA16200 HTTP Server test with POSTMAN tool .......... 114
Figure 34: ThreadX APIs Test ............................................................ 116
Figure 35: get_scan_result Sample Test ............................................ 128
Figure 36: sflash Example Sample Test .............................................. 130

© 2020 Dialog Semiconductor
Terms and Definitions

AP  Access Point
UART  Universal Asynchronous Receiver-Transmitter
ADC  Analog-to-Digital Converter
AES  Advanced Encryption Standard
API  Application Programming Interface
AT  ATTention
CCM  Counter with CBC-MAC
CTR  Counter
DAC  Digital-To-Analog Converter
DER  Distinguished Encoding Rules
DES  Data Encryption Standard
DHCP  Dynamic Host Configuration Protocol
DNS  Domain Name Server
DPM  Dynamic Power Management
DRBG  Deterministic Random Bit Generator
DUT  Device Under Test
ECDH  Elliptic Curve Diffie-Hellman
ECDSA  Elliptic Curve Digital Signature Algorithm
EVB  Evaluation Board
EVK  Evaluation Kit
GCM  Galois/Counter Mode
GPIO  General-Purpose Input/Output
HMAC  Hash-(based) Message Authentication Code
HTTP  HyperText Transfer Protocol
HTML  HyperText Markup Language
I2C  Inter-Integrated Circuit
I2S  Inter-IC Sound
KDF  Key Derivation Function
MD5  Message Digest 5
MCU  MicroController Unit
NVRAM  Non-volatile random-access memory
OFB  Output Feedback
PEM  Privacy-Enhanced Mail
POR  Power-On Reset
PWM  Pulse Width Modulation
RSA PKCS  RSA Public Key Cryptography Standards
RTC  Real-Time Clock
RTM  Retention Memory
RTOS  Real-Time Operating System
SD/eMMC  Secure Digital / Embedded MultiMediaCard
SDIO  Secure Digital Input Output
SNTP  Simple Network Time Protocol
SPI  Serial Peripheral Interface
SRAM  Static Random Access Memory
STA  Station
TCP  Transmission Control Protocol
TLS  Transport Layer Security
UDP  User Datagram Protocol
References

1 How to start

1.1 The Samples Projects

The DA16200 Sample-Project are classified as five functional categories. (Crypto, DPM, Network, Peripheral, and ETC) See Figure 1.

Figure 1: Overall Test Setup

1.2 Build the Sample Project

1.2.1 IAR: Open and build a sample workspace file

- Open the DA16xxx.eww (Figure 2), right-click on the project name (main) in the IAR Embedded Workbench workspace.

Figure 2: Start IAR workbench
Figure 3: Open the DA16200 Sample workspace

- Run the command “Make” or “Rebuild All” of IAR workbench compiler. If you compile for the first time, run the command “Clean” before to make. See Figure 4.

Figure 4: Rebuild or Make to compile

1.3 Start Sample Application

All Sample Application in the DA16200 SDK, it is need to adopt compile feature for sample project.

- All of the Sample Applications depend on `__ENABLE_SAMPLE_APP__` compile feature.
- To avoid Customer/Developer's confusion about the sample code, the `__ENABLE_SAMPLE_APP__` is defined in sample IAR project pre-define feature. See Figure 5.
1.3.1 How to Start Sample main()

In order to analyze the flow of sample code, Customer/Developer can start from the main () function. See Figure 6.

![Figure 6: The flow of sample code](image)

**Figure 6: The flow of sample code**
After the DA16200 booting, the system library invokes function `main()`. The following steps are run:

- Initialize HW resources (PIN_MUX, RTC, Console ...)
- Start function `system_start()` to run the DA16200 as Wi-Fi IoT device.

Customer/Developer can check the code flow in `main.c` and `system_start.c`. See Figure 7.

```
[ ~/src/main.c ]
int main(char init_state)
{
  ... ...
  /* Entry point for customer main */
  if (init_state == TRUE)
  {
    status = system_start();
  }
  ... ...
}
```

After initializing basic HW resources, calls `system_start()` function to run Wi-Fi operation. The following happens:

- H/W and S/W feature configuration.
- Configure system resources for system clock and TX power
- Initialize Wi-Fi function Function in `wlaninit()`
- Start system provided applications in `start_sys_apps()`.

```
[ ~/src/customer/system_start.c ]
int system_start(void)
{
  ... ...
  /* Initialize WLAN interface */
  wlaninit();
  ...
  /* Start system applications for DA16XXX */
  start_sys_apps();
  ...
}
```
To start the sample application, Customer/Developer should register the `sample_app_start_cb` function which is a sample thread creation function to operate the sample application. Customer/Developer can check the code flow in `sample_apps.c`. See Figure 8.

Figure 8: Sample_apps file on IAR project

If `sample_app_start_cb` function is exist in the DA16200 SDK, the corresponding sample function in `sample_apps_table` is automatically executed.

- Regist `sample_app_start_cb` in case of Sample project.
- Create system and user applications which are non-network related operation.
- Check network status

```c
[ ~/src/application/system_apps/sys_apps.c ]
void start_sys_apps(void)
{
    /* Start user application functions */
    run_sys_apps();
}

static void run_sys_apps(void)
{
    ...

#if defined (__ENABLE_SAMPLE_APP__)
    /* For sample codes */
    regist_sample_cb();
#endif // __ENABLE_SAMPLE_APP__
    ...

    /* Create network apps */
    create_sys_apps(sysmode, TRUE);
}
```

- Create system applications which are network related operation
- Run Sample Application which is defined in sample feature.

```c
[ ~/src/sample_apps.c ]
void regist_sample_cb(void)
{
```
static void create_sys_apps(int sysmode, UCHAR net_chk_flag)
{
    /* Create test samples apps */
    if (sample_app_start_cb != NULL) {
        sample_app_start_cb(net_chk_flag);
    }
}

void create_sample_apps(UCHAR net_chk_flag)
{
    ... ...
    mode = get_run_mode();
    for (i = 0 ; sample_apps_table[i].name != NULL ; i++) {
        /* Run matched apps with net_chk_flag */
        if (sample_apps_table[i].net_chk_flag == net_chk_flag) {
            cur_tx = create_new_app(cur_tx, &(sample_apps_table[i]), mode);
        }
    }
}

● All of the sample applications are registered in management table as below;

[ ~/src/sample_apps.c ]
static const app_thread_info_t sample_apps_table[] =
{
    .... ....
#if defined (__TCP_CLIENT_SAMPLE__) 
    { SAMPLE_TCP_CLI, tcp_client_sample, 1024, USER_PRI_APP(0), TRUE,
     FALSE, TCPC_PORT, RUN_ALL_MODE },
#endif // (__TCP_CLIENT_SAMPLE__) 
    .... ....
#if defined (__TCP_SERVER_SAMPLE__) 
    { SAMPLE_TCP_SVR, tcp_server_sample, 1024, USER_PRI_APP(0), TRUE,
     FALSE, TCPS_PORT, RUN_ALL_MODE },
#endif // (__TCP_SERVER_SAMPLE__) 
    .... ...
    { NULL, NULL, 0, 0, FALSE, FALSE, UNDEF_PORT, 0 }
};
2  Socket Communication

This section describes how to develop TCP or UDP socket applications with NetX Duo APIs on the DA16200 SDK. As a companion document, please always refer to the NetX_Duo_User_Guide.pdf [1] for any nx_xx() functions. To easily understand DPM API and to help implement a DPM function, we provide each example as a non-DPM function version and a DPM function version. To try the examples, please set up a test environment first as shown in Figure 9.

2.1  Test Setup

![Figure 9: Overall Test Setup](image)

2.2  DA16200

The example source files are included in the DA16200 SDK. All examples introduced in this section do not have code for Wi-Fi station interface setup. Please see section “Wi-Fi Mode Setup” in the DA16200 EVK User Guide [2] and setup STA mode with the console command. Also, make a note of the IP address of the DA16200 EVB to use for the examples. The IP address is printed after connecting to an AP and then TCP/UDP example application will run. See Figure 10.

![Figure 10: DA16200 EVB – AP Connection Done](image)
DA16200 Example Application Guide

2.2.1 Test-DUT

A peer Test-DUT where a TCP/UDP counterpart application is running. You need to install a proper application (for example Packet Sender, Hercules, IO Ninja and so forth). This PC is connected to an AP, either with a wired (LAN or WAN) or a wireless link.

2.3 TCP Server

The TCP server sample application is an example of the simplest TCP echo server application. The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. DA16200 SDK provides NetX Duo's TCP protocol. NetX Duo is a high-performance real-time implementation of the TCP/IP standards designed exclusively for embedded ThreadX-based applications.

This session describes how the TCP server sample application is built and worked.

2.3.1 How to Run

1. Open the workspace for the TCP Server sample application as follows:
   - \sample\Network\TCP_Server\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the sample application will create a TCP server socket with port number 10190 and wait for a client connection.
5. Run a socket application on the peer PC.
6. Open a TCP client socket.

2.3.2 How it Works

The DA16200 TCP Server sample application is a simple echo server. When a TCP client sends a message, then the DA16200 TCP server will echo that message to the TCP client.

![Figure 11: Workflow of TCP Server](image)

2.3.3 Details

DA16200 SDK provides NetX Duo's TCP protocol. This sample application describes how TCP socket is created, deleted, and configured.
2.3.3.1 Connection

The server waits for a client connection request. To accept a client connection, TCP must first be enabled on the IP instance by get_thread_netx() function. Next, the application must create a TCP socket using nx_tcp_socket_create() service. The server socket must also be setup for listening for connection requests using nx_tcp_server_socket_listen() service. This service places the server socket in the LISTEN state and binds the specified server port to the server socket. If the socket connection has already been established, the function simply returns a successful status.

```c
void tcp_server_sample_run()
{
    NX_IP *ip_ptr = NULL;
    NX_PACKET_POOL *pool_ptr = NULL;
    NX_TCP_SOCKET *sock_ptr = NULL;

    // Get informations of current ip instance and packet pool
    get_thread_netx((void **)&pool_ptr, (void **)&ip_ptr, WLAN0_IFACE);

    // Create TCP socket
    status = nx_tcp_socket_create(ip_ptr,
        sock_ptr,
        TCP_SERVER_SAMPLE_SOCKET_NAME,
        NX_IP_NORMAL,
        NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE,
        TCP_SERVER_SAMPLE_TCP_WINDOW_SZ,
        NX_NULL,
        NX_NULL);

    // Listen TCP socket
    status = nx_tcp_server_socket_listen(ip_ptr,
        srv_info.port,
        sock_ptr,
        TCP_SERVER_SAMPLE_LISTEN_BAGLOG,
        NX_NULL);

    // Accept TCP session
    do
    {
        /*
         * Wait for 100(TCP_SERVER_SAMPLE_DEF_TIMEOUT) ticks
         */
```
2.3.3.2 Data Transmission

Receiving TCP data is accomplished by calling nx_tcp_socket_receive() function. The TCP receive packet processing is responsible for handling various connection and disconnection actions as well as transmit acknowledge processing. In addition, the TCP receive packet processing is responsible for placing packets with receive data on the appropriate TCP socket's receive queue or delivering the packet to the first suspended thread waiting for a packet.

Sending TCP data is easily accomplished by calling the nx_tcp_socket_send() function. This service first builds a TCP header in front of the packet (including the checksum calculation). If the receiver’s window size is larger than the data in this packet, the packet is sent on the Internet using the internal IP send routine. Otherwise, the caller may suspend and wait for the receiver’s window size to increase enough for this packet to be sent. At any given time, only one sender may suspend while trying to send TCP data.

```c
void tcp_server_sample_run()
{
    ...
    while (NX_TRUE)
    {
        /*
         * Receive a packet from the connected TCP server socket.
         * If no packet is available,
         * wait for 100(TCP_SERVER_SAMPLE_DEF_TIMEOUT) timer ticks before giving up.
         */
        status = nx_tcp_socket_receive(sock_ptr, &recv_ptr,
                                        TCP_SERVER_SAMPLE_DEF_TIMEOUT);
        if (status == NX_SUCCESS)
        {
            ...
        }
    }
}
```
{ // Get the length of the received packet.
    status = nx_packet_length_get(recv_ptr, &recv_bytes);

    // Copy data from a packet into the buffer.
    status = nx_packet_data_retrieve(recv_ptr, data_buf, &data_buflen);

    // Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", recv_bytes);

    // Allocate a packet from the packet pool
    status = nx_packet_allocate(pool_ptr,
                              &send_ptr,
                              NX_TCP_PACKET,
                              NX_WAIT_FOREVER);

    // Copy data to the end of the packet.
    status = nx_packet_data_append(send_ptr,
                                   data_buf,
                                   data_buflen,
                                   pool_ptr,
                                   NX_WAIT_FOREVER);

    // Send a TCP packet through the socket.
    status = nx_tcp_socket_send(sock_ptr, send_ptr, NX_WAIT_FOREVER);

    // Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", data_buflen);
}

2.3.3.3 Disconnection

Closing the connection is accomplished by calling nx_tcp_socket_disconnect() function. After the disconnect processing is complete and the server socket is in the CLOSED state, the application must call the nx_tcp_server_socket_unaccept() service to end the association of this socket with the server port. After the nx_tcp_server_socket_unaccept() returns, the socket can be used as a client or server socket, or even deleted if it is no longer needed. In this sample TCP socket is deleted.

void tcp_server_sample_run()
{  
  ...
  // Disconnect the client socket from the server  
  status = nx_tcp_socket_disconnect(sock_ptr, TCP_SERVER_SAMPLE_DEF_TIMEOUT);
  
  // Unbind the TCP client socket structure from the previously bound TCP port  
  status = nx_tcp_server_socket_unaccept(sock_ptr);
  
  // Delete the socket  
  status = nx_tcp_socket_delete(sock_ptr);
  ...
}

2.3.3.4 To stop listening on a server port

If the application no longer wishes to listen for client connection requests on a server port that was previously specified by a call to nx_tcp_server_socket_listen() service, the application simply calls nx_tcp_server_socket_unlisten() service. This service places any socket waiting for a connection back in the CLOSED state and releases any queued client connection request packets.

If accepting another client connection on the same server port is desired, the nx_tcp_server_socket_relisten() service should be called with this socket.

```c
void tcp_server_sample_run()
{
  ...
  // Unlisten on server port.
  status = nx_tcp_server_socket_unlisten(ip_ptr, srv_info.port);
  ...
}
```

2.4 TCP Server in DPM

The TCP server in the DPM sample application is an example of the simplest TCP echo server application. The DA16200 SDK can work in DPM mode. The user application is required to work in DPM mode. The DA16200 SDK provides a DPM manager feature for the user network application. The DPM manager feature supports the user to develop and manage a network application in Non-DPM and DPM modes. The codes are almost the same as for the TCP Server example. This section describes how the TCP server is built and works in the DPM sample application.

2.4.1 How to Run

1. Open the workspace for the TCP Server in the DPM sample application as follows:
   a. \sample\Network\TCP_Server_DPM\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After connection is made to an AP, the sample application will create a TCP server socket with port number 10190 and wait for a client connection.
5. Run a socket application on the peer PC.
6. Open a TCP client socket.

2.4.2 How it Works
The DA16200 TCP Server in the DPM sample application is a simple echo server. When a TCP client sends a message, then the DA16200 TCP server will echo that message to the TCP client.

![Figure 12: Workflow of TCP Server in DPM](image)

2.4.3 Details
2.4.3.1 Registration
The TCP server in the DPM sample application works in DPM mode. The basic code is similar to the TCP server sample application. The difference with the TCP server sample application is two things. First, an initial callback function is added, named `tcp_server_dpm_sample_wakeup_callback` in the code. The callback is called when the DPM state changes from sleep to wake-up. Second, an additional user configuration can be stored in RTM. In this sample, the TCP server information will be stored.

```c
void tcp_server_dpm_sample_init_user_config(dpm_user_config_t *user_config) {
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = tcp_server_dpm_sample_init_callback;

    //Set DPM wakkup init callback
    user_config->wakeupInitCallback = tcp_server_dpm_sample_wakeup_callback;

    //Set Error callback
    user_config->errorCallback = tcp_server_dpm_sample_error_callback;

    //Set session type(TCP Server)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_TCP_SERVER;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort =
        TCP_SERVER_DPM_SAMPLE_DEF_SERVER_PORT;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback =
        tcp_server_dpm_sample_connect_callback;

    //Set Recv callback
```
2.4.3.2 Data Transmission

The callback function is called when a TCP packet is received from a TCP client. In this sample, the received data is printed out and an echo message is sent to the TCP client.

```c
void tcp_server_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len,
                                       ULONG rx_ip, ULONG rx_port)
{
    // Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);

    // Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf, rx_len);

    // Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", rx_len);

    dpm_mng_job_done(); // Done operation
}
```

2.5 TCP Client

The TCP client sample application is an example of the simplest TCP echo client application. The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network. DA16200 SDK provides NetX Duo's TCP protocol. NetX Duo is a high-performance real-time implementation of the TCP/IP standards designed exclusively for embedded ThreadX-based applications.

This session describes how the TCP client sample application is built and worked.

2.5.1 How to Run

1. Run a socket application on the peer PC and open a TCP server socket with port number 10192.
2. Open the workspace for the TCP Client sample application as follows:
   - `\sample\Network\TCP_Client\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the TCP server IP address and the port number as you created the socket on the peer PC with the following console command and then reboot. These parameters can also be defined in the source code.
   - `/DA16200` # nvram.setenv TCPC_SERVER_IP 192.168.0.11
   - `/DA16200` # nvram.setenv TCPC_SERVER_PORT 10192
   - `/DA16200` # reboot
   - After a connection is made to an AP, the example application will connect to the peer PC.
DA16200 Example Application Guide

2.5.2 How it Works
The DA16200 TCP Client sample application is a simple echo message. When the TCP server sends a message, then the DA16200 TCP client will echo that message to the TCP server.

![Figure 13: Workflow of TCP Client](image)

2.5.3 Details
DA16200 SDK provides NetX Duo's TCP protocol. This sample application describes how TCP socket is created, deleted, and configured.

2.5.3.1 Registration
The client side of the TCP connection initiates a connection request to a TCP server. Before a connection request can be made, TCP must be enabled on the client IP instance by get_thread_netx() function. In addition, the client TCP socket must next be created with nx_tcp_socket_create() service and bound to a port via nx_tcp_client_socket_bind() service. After the client socket is bound, nx_tcp_client_socket_connect() service is used to establish a connection with a TCP server.

```c
void tcp_client_sample_run()
{
    NX_IP *ip_ptr = NULL;
    NX_PACKET_POOL *pool_ptr = NULL;

    NX_TCP_SOCKET *sock_ptr = NULL;

    // Get information of current ip interface and packet pool
    get_thread_netx((void **)&pool_ptr, (void **)&ip_ptr, WLAN0_IFACE);

    // Create TCP socket
    status = nx_tcp_socket_create(ip_ptr,
        sock_ptr,
        TCP_CLIENT_SAMPLE_SOCKET_NAME,
        NX_IP_NORMAL,
        NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE,
        TCP_CLIENT_SAMPLE_TCP_WINDOW_SZ,
        NX_NULL,
        NX_NULL);

    // Bind TCP socket
    status = nx_tcp_client_socket_bind(sock_ptr,
        TCP_CLIENT_SAMPLE_DEF_CLIENT_PORT,
        NX_WAIT_FOREVER);
}
```
// Connect to TCP server
status = nx_tcp_client_socket_connect(sock_ptr,
          srv_info.ip_addr,
          srv_info.port,
          TCP_CLIENT_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT);
...

2.5.3.2 Data Transmission

Receiving TCP data is accomplished by calling `nx_tcp_socket_receive()` function. The TCP receive packet processing is responsible for handling various connection and disconnection actions as well as transmit acknowledge processing. In addition, the TCP receive packet processing is responsible for placing packets with receive data on the appropriate TCP socket's receive queue or delivering the packet to the first suspended thread waiting for a packet.

Sending TCP data is easily accomplished by calling the `nx_tcp_socket_send()` function. This service first builds a TCP header in front of the packet (including the checksum calculation). If the receiver's window size is larger than the data in this packet, the packet is sent on the Internet using the internal IP send routine. Otherwise, the caller may suspend and wait for the receiver's window size to increase enough for this packet to be sent. At any given time, only one sender may suspend while trying to send TCP data.

```c
void tcp_client_sample_run()
{
    ...
    while (NX_TRUE)
    {
        /*
        * Receive a packet from the connected TCP client socket.
        * If no packet is available,
        * wait for 100 (TCP_CLIENT_SAMPLE_DEF_TIMEOUT) timer ticks before giving up.
        */
        status = nx_tcp_socket_receive(sock_ptr, &recv_ptr,
                                      TCP_CLIENT_SAMPLE_DEF_TIMEOUT);
        if (status == NX_SUCCESS)
        {
            // Get the length of the received packet.
            status = nx_packet_length_get(recv_ptr, &recv_bytes);

            // Copy data from a packet into the buffer.
            status = nx_packet_data_retrieve(recv_ptr, data_buf, &data_buflen);

            // Display received packet
            PRINTF(" =====> Received Packet(%ld) \n", recv_bytes);

            // Allocate a packet from the packet pool
            status = nx_packet_allocate(pool_ptr,
                                         &send_ptr,
                                         NX_TCP_PACKET,
                                         NX_WAIT_FOREVER);

            // Copy data to the end of the packet.
            status = nx_packet_data_append(send_ptr,
                                           data_buf,
                                           data_buflen,
                                           pool_ptr,
                                           NX_WAIT_FOREVER);
        }
    }
}
```
// Send a TCP packet through the socket.
status = nx_tcp_socket_send(sock_ptr, send_ptr, NX_WAIT_FOREVER);

// Display sent packet
PRINTF(" <==== Sent Packet(%ld) \n", data_buflen);
}
}

2.5.3.3 Disconnection
Closing the connection is accomplished by calling nx_tcp_socket_disconnect() function. To unbind the port and client socket, the application calls nx_tcp_client_socket_unbind() function. The socket must be in a CLOSED state or in the process of disconnecting before the port is released; otherwise, an error is returned. Finally, if the application no longer needs the client socket, it calls nx_tcp_socket_delete() function to delete the socket.

```c
void tcp_client_sample_run()
{
    ...
    // Disconnect the client socket from the server
    status = nx_tcp_socket_disconnect(sock_ptr, TCP_CLIENT_SAMPLE_DEF_TIMEOUT);

    // Unbind the TCP client socket structure from the previously bound TCP port
    status = nx_tcp_client_socket_unbind(sock_ptr);

    // Delete the socket
    status = nx_tcp_socket_delete(sock_ptr);
    ...
}
```

2.6 TCP Client in DPM
The TCP client in DPM sample application is an example of the simplest TCP echo client application in DPM mode. The DA16200 SDK can work in DPM mode. The user application is required an additional operation to work in DPM mode. The DA16200 SDK provides a DPM manager feature for the user network application. The DPM manager feature supports the user to develop and manage a network application in Non-DPM and DPM modes. This section describes how the TCP client in the DPM sample application is built and works.

2.6.1 How to Run
1. Run a socket application on the peer PC and open a TCP server socket with port number 10192.
2. Open the workspace for the TCP Client in the DPM sample application as follows:
   1. `\sample\Network\TCP_Client_DPM\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the TCP server IP address and port number as you created the socket on the peer PC with the following console command and reboot. These parameters can also be defined in the source code.

   ```
   [/DA16200] # nvram.setenv TCPC_SERVER_IP 192.168.0.11
   [/DA16200] # nvram.setenv TCPC_SERVER_PORT 10192
   [/DA16200] # reboot
   ```
After a connection is made to an AP, the example application will connect to the peer PC.

### 2.6.2 How it Works

The DA16200 TCP Client in the DPM sample application is a simple echo message. When the TCP server sends a message, then the DA16200 TCP client will echo that message to the TCP server.

**Figure 14: Workflow of TCP Client in DPM**

### 2.6.3 Details

#### 2.6.3.1 Registration

The TCP client in the DPM sample application works in DPM mode. The basic code is similar to the TCP client sample application. The difference with the TCP client sample application is two things. First, an initial callback function is added, named `tcp_client_dpm_sample_wakeup_callback` in the code. The callback is called when the DPM state changes from sleep to wake-up. Second, an additional user configuration can be stored in RTM. In this sample, TCP server information will be stored.

```c
void tcp_client_dpm_sample_init_user_config(dpm_user_config_t *user_config)
{
    const int session_idx = 0;
    //Set Boot init callback
    user_config->bootInitCallback = tcp_client_dpm_sample_init_callback;
    //Set DPM wakeup init callback
    user_config->wakeupInitCallback = tcp_client_dpm_sample_wakeup_callback;
    //Set External wakeup callback
    user_config->externWakeupCallback = tcp_client_dpm_sample_external_callback;
    //Set Error callback
    user_config->errorCallback = tcp_client_dpm_sample_error_callback;
    //Set session type(TCP Client)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_TCP_CLIENT;
    //Set local port
    user_config->sessionConfig[session_idx].session_myPort = TCP_CLIENT_DPM_SAMPLE_DEF_CLIENT_PORT;
    //Set server IP address
    memcpy(user_config->sessionConfig[session_idx].session_serverIp, 
```
srv_info.ip_addr, strlen(srv_info.ip_addr));

//Set server port
user_config->sessionConfig[session_idx].session_serverPort = srv_info.port;

//Set Connection callback
user_config->sessionConfig[session_idx].session_connectCallback =
    tcp_client_dpm_sample_connect_callback;

//Set Recv callback
user_config->sessionConfig[session_idx].session_recvCallback =
    tcp_client_dpm_sample_recv_callback;

//Set connection retry count
user_config->sessionConfig[session_idx].session_conn_retry_cnt =
    TCP_CLIENT_DPM_SAMPLE_DEF_MAX_CONNECTION_RETRY;

//Set connection timeout
user_config->sessionConfig[session_idx].session_conn_wait_time =
    TCP_CLIENT_DPM_SAMPLE_D_EF_MAX_CONNECTION_TIMEOUT;

//Set auto reconnection flag
user_config->sessionConfig[session_idx].session_auto_reconn = NX_TRUE;

//Set user configuration
user_config->ptrDataFromRetentionMemory = (UCHAR *)&srv_info;
user_config->sizeOfRetentionMemory =
    sizeof(tcp_client_dpm_sample_svr_info_t);

return ;
}

2.6.3.2 Data Transmission

The callback function is called when a TCP packet is received from a TCP server. In this sample, the received data is printed out and an echo message is sent to the TCP server.

```c
void tcp_client_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len, ULONG rx_ip, ULONG rx_port) {
    //Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);

    //Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf, rx_len);

    //Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", rx_len);

dpm_mng_job_done(); //Done opertaion
}
```

2.7 TCP Client with KeepAlive

The TCP client with KeepAlive sample application is an example of the simplest TCP echo client application. Especially, this sample application sends TCP KeepAlive message to TCP server periodically. The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. TCP provides reliable, ordered, and error-checked delivery of a stream of octets
DA16200 Example Application Guide

(bytes) between applications running on hosts communicating via an IP network. DA16200 SDK provides NetX Duo's TCP protocol. NetX Duo is a high-performance real-time implementation of the TCP/IP standards designed exclusively for embedded ThreadX-based applications.

This session describes how the TCP client with KeepAlive sample application is built and worked.

2.7.1 How to Run

1. Run a socket application on the peer PC and open a TCP server socket with port number 10193.
2. Open the workspace for a TCP Client with KeepAlive in the sample application as follows:
   ○ \sample\Network\TCP_Client_KeepAlive\build\DA16xxx.eww
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the TCP server IP address and the port number as you created the socket on the peer PC with the following console command and then reboot. These parameters can also be defined in the source code.

   ![Console Commands]

   ○ After connection is made to an AP, the example application will connect to the peer PC.

2.7.2 How it Works

The DA16200 TCP Client with KeepAlive in the sample application is a simple echo message. When a TCP server sends a message, then the DA16200 TCP client will echo that message to the TCP server. And, a TCP KeepAlive message will be periodically sent to the TCP server each 55 seconds.

2.7.3 Details

DA16200 SDK provides NetX Duo's TCP protocol. This sample application describes how TCP socket is created, deleted, and configured to send TCP KeepAlive.

2.7.3.1 Registration

The client side of the TCP connection initiates a connection request to a TCP server. Before a connection request can be made, TCP must be enabled on the client IP instance by get_thread_netx() function. In addition, the client TCP socket must next be created with
nx_tcp_socket_create() service and bound to a port via the nx_tcp_client_socket_bind() service. In order to send TCP KeepAlive message, the configuration is by nx_tcp_socket_option() function. In this sample application, the period of TCP KeepAlive message is 55 second. After the client socket is bound, the nx_tcp_client_socket_connect() service is used to establish a connection with a TCP server.

```c
void tcp_client_ka_sample_run()
{
    NX_IP *ip_ptr = NULL;
    NX_PACKET_POOL *pool_ptr = NULL;
    NX_TCP_SOCKET *sock_ptr = NULL;
    NX_TCP_SOCKET_OPTION sock_opt;
    // Get information of current ip interface and packet pool
    get_thread_netx((void **)&pool_ptr, (void **)&ip_ptr, WLAN0_IFACE);
    // Create TCP socket
    status = nx_tcp_socket_create(ip_ptr,
        sock_ptr,
        TCP_CLIENT_SAMPLE_SOCKET_NAME,
        NX_IP_NORMAL,
        NX_FRAGMENT_OKAY,
        NX_IP_TIME_TO_LIVE,
        TCP_CLIENT_KA_SAMPLE_TCP_WINDOW_SZ,
        NX_NULL,
        NX_NULL);
    // Bind TCP socket
    status = nx_tcp_client_socket_bind(sock_ptr,
        TCP_CLIENT_KA_SAMPLE_DEF_CLIENT_PORT,
        NX_WAIT_FOREVER);
    // Set TCP KeepAlive
    sock_opt.keepalive_enabled = NX_TRUE;
    sock_opt.keepalive_timeout = TCP_CLIENT_KA_SAMPLE_DEF_KEEPALIVE_TIMEOUT;
    status = nx_tcp_socket_option(sock_ptr, NX_SO_KEEPALIVE, &sock_opt);
    // Connect to TCP server
    status = nx_tcp_client_socket_connect(sock_ptr,
        srv_info.ip_addr,
        srv_info.port,
        TCP_CLIENT_KA_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT);
    ...
}
```

### 2.7.3.2 Data Transmission

Receiving TCP data is accomplished by calling nx_tcp_socket_receive() function. The TCP receive packet processing is responsible for handling various connection and disconnection actions as well as transmit acknowledgment processing. In addition, the TCP receive packet processing is responsible for placing packets with receive data on the appropriate TCP socket's receive queue or delivering the packet to the first suspended thread waiting for a packet.

Sending TCP data is easily accomplished by calling the nx_tcp_socket_send() function. This service first builds a TCP header in front of the packet (including the checksum calculation). If the receiver's window size is larger than the data in this packet, the packet is sent on the Internet using the internal IP send routine. Otherwise, the caller may suspend and wait for the receiver’s window size to
increase enough for this packet to be sent. At any given time, only one sender may suspend while trying to send TCP data.

```c
void tcp_client_ka_sample_run()
{
...

while (NX_TRUE)
{
    /*
        * Receive a packet from the connected TCP client socket.
        * If no packet is available,
        * wait for 100(TCP_CLIENT_KA_SAMPLE_DEF_TIMEOUT) timer ticks before giving up.
    */
    status = nx_tcp_socket_receive(sock_ptr, &recv_ptr,
                                TCP_CLIENT_KA_SAMPLE_DEF_TIMEOUT);
    if (status == NX_SUCCESS)
    {
        // Get the length of the received packet.
        status = nx_packet_length_get(recv_ptr, &recv_bytes);

        // Copy data from a packet into the buffer.
        status = nx_packet_data_retrieve(recv_ptr, data_buf, &data_buflen);

        // Display received packet
        PRINTF(" =====> Received Packet(%ld) \n", recv_bytes);

        // Allocate a packet from the packet pool
        status = nx_packet_allocate(pool_ptr,
                                    &send_ptr,
                                    NX_TCP_PACKET,
                                    NX_WAIT_FOREVER);

        // Copy data to the end of the packet.
        status = nx_packet_data_append(send_ptr,
                                        data_buf,
                                        data_buflen,
                                        pool_ptr,
                                        NX_WAIT_FOREVER);

        // Send a TCP packet through the socket.
        status = nx_tcp_socket_send(sock_ptr, send_ptr, NX_WAIT_FOREVER);

        // Display sent packet
        PRINTF(" <==== Sent Packet(%ld) \n", data_buflen);
    }
}
...
```

2.7.3.3 Disconnection

Closing the connection is accomplished by calling `nx_tcp_socket_disconnect()` function. To unbind the port and client socket, the application calls `nx_tcp_client_socket_unbind()` function. The socket must be in a CLOSED state or in the process of disconnecting before the port is released; otherwise, an error is returned. Finally, if the application no longer needs the client socket, it calls `nx_tcp_socket_delete()` function to delete the socket.

```c
void tcp_client_ka_sample_run()
```
2.8 TCP Client with KeepAlive in DPM

The TCP client with KeepAlive in DPM sample application is an example of the simplest TCP echo client application in DPM mode. The DA16200 SDK can work in DPM mode. The user application requires to work in DPM mode. The DA16200 SDK provides a DPM manager feature for the user network application. The DPM manager feature supports the user to develop and manage a network application in Non-DPM and DPM modes. This section describes how the TCP client with KeepAlive in DPM sample application is built and works.

2.8.1 How to Run

1. Run a socket application on the peer PC and open a TCP server socket with port number 10193.
2. Open the workspace for a TCP Client with KeepAlive in the DPM sample application as follows:
   ○ `\sample\Network\TCP_Client_KeepAlive_DPM\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the TCP server IP address and port number as you created the socket on the peer PC with the following console command and reboot. These parameters can also be defined in the source code.

```
[DA16200] # nvram.setenv TCPC_SERVER_IP 192.168.0.11
[DA16200] # nvram.setenv TCPC_SERVER_PORT 10192
[DA16200] # reboot
```

   ○ After a connection is made to an AP, the example application will connect to the peer PC.

2.8.2 How it Works

The DA16200 TCP Client with KeepAlive in the DPM sample application is a simple echo message. When the TCP server sends a message, then the DA16200 TCP client will echo that message to the TCP server. A periodic TCP KeepAlive message is sent to the TCP server each 55 seconds.
Figure 16: Workflow of TCP Client with KeepAlive in DPM

2.8.3 Details

2.8.3.1 Registration

The TCP client with KeepAlive in the DPM sample application works in DPM mode. The basic code is similar to the TCP client with KeepAlive sample application. The time period is 55 second to send a TCP KeepAlive message to the TCP server. The difference with the TCP client sample application is two things compared to the TCP client in the DPM sample application. First, an initial callback function is added, named 'tcp_client_ka_dpm_sample_wakeup_callback' in the code. The callback function is called when the DPM state changes from sleep to wake-up. Second, an additional user configuration can be stored in RTM. In this example, TCP server information will be stored.

```c
void tcp_client_ka_dpm_sample_init_user_config(dpm_user_config_t *user_config) {
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = tcp_client_ka_dpm_sample_init_callback;

    //Set Error callback
    user_config->errorCallback = tcp_client_ka_dpm_sample_error_callback;

    //Set session type(TCP Client)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_TCP_CLIENT;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort = TCP_CLIENT_KA_DPM_SAMPLE_DEF_CLIENT_PORT;

    //Set server IP address
    memcpy(user_config->sessionConfig[session_idx].session_serverIp, srv_info.ip_addr, strlen(srv_info.ip_addr));

    //Set server port
    user_config->sessionConfig[session_idx].session_serverPort = srv_info.port;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback = tcp_client_ka_dpm_sample_connect_callback;

    //Set Recv callback
    user_config->sessionConfig[session_idx].session_recvCallback = tcp_client_ka_dpm_sample_recv_callback;
}
```
//Set connection retry count
user_config->sessionConfig[session_idx].session_conn_retry_cnt =
TCP_CLIENT_KA_DPM_SAMPLE_DEF_MAX_CONNECTION_RETRY;

//Set connection timeout
user_config->sessionConfig[session_idx].session_conn_wait_time =
TCP_CLIENT_KA_DPM_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT;

//Set auto reconnection flag
user_config->sessionConfig[session_idx].session_auto_reconn = NX_TRUE;

//Set KeepAlive timeout
user_config->sessionConfig[session_idx].session_ka_interval =
TCP_CLIENT_KA_DPM_SAMPLE_DEF_KEEPALIVE_TIMEOUT;

return ;

2.8.3.2 Data Transmission

The callback function is called when a TCP packet is received from the TCP server. In this example, the received data is printed out and an echo message is sent to the TCP server.

void tcp_client_ka_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len,
ULONG rx_ip, ULONG rx_port)
{
    //Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);

    //Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf,
    rx_len);

    //Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", rx_len);

dpm_mng_job_done(); //Done opertaion
}

2.9 UDP Socket

The UDP socket sample application is an example of the simplest UDP echo application. User Datagram Protocol (UDP) is one of the core members of the Internet protocol suite. UDP uses a simple connectionless communication model with a minimum of protocol mechanisms. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram. It has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network; there is no guarantee of delivery, ordering, or duplicate protection. DA16200 SDK provides NetX Duo's UDP protocol. NetX Duo is a high-performance real-time implementation of the UDP/IP standards designed exclusively for embedded ThreadX-based applications.

This session describes how the UDP socket sample application is built and worked.

2.9.1 How to Run

1. Open the workspace for the UDP Server sample application as follows:
   o \sample\Network\UDP_Server\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the sample application will create a UDP socket with port number 10195 and wait for a peer message.
5. Run a socket application on the peer PC.

2.9.2 How it Works

The DA16200 UDP socket sample application is a simple echo server. When a UDP peer sends a message, then the DA16200 UDP socket sample application will echo that message to the UDP peer.

![Figure 17: Workflow of UDP Socket](image)

2.9.3 Details

DA16200 SDK provides NetX Duo's UDP protocol. This sample application describes how UDP socket is created, deleted, and configured.

2.9.3.1 Initialization

A UDP port is a logical end point in the UDP protocol. There are 65,535 valid ports in the UDP component of NetX Duo, ranging from 1 through 0xFFFF. To send or receive UDP data, the application must first create a UDP socket, `nx_udp_socket_create()` function, then bind it to a desired port. After bound to a port, the application may send and receive data on that socket. The details are like the following.

```c
void udp_socket_sample_run()
{
    NX_IP *ip_ptr = NULL;
    NX_PACKET_POOL *pool_ptr = NULL;
    NX_UDP_SOCKET *sock_ptr = NULL;

    // Get informations of current ip interface and packet pool
    get_thread_netx((void **)pool_ptr, (void **)ip_ptr, WLAN0_IFACE);

    // Create socket
    status = nx_udp_socket_create(ip_ptr,
                                   sock_ptr,
                                   UDP_CLIENT_SAMPLE_SOCKET_NAME,
                                   NX_IP_NORMAL,
                                   NX_FRAGMENT_OKAY,
                                   NX_IP_TIME_TO_LIVE,
                                   5);

    // Bind udp socket
```
2.9.3.2 Data Transmission

Receiving UDP packet is accomplished by calling nx_udp_socket_receive() function. The socket receive function delivers the oldest packet on the socket's receive queue. If there are no packets on the receive queue, the calling thread can suspend (with an optional timeout) until a packet arrives.

Sending UDP data is easily accomplished by calling nx_udp_socket_send() function. This service places a UDP header in front of the packet and sends it on the Internet using the internal IP send routine. There is no thread suspension on sending UDP packets because all UDP packet transmissions are processed immediately.

```c
void udp_socket_sample_run()
{
    ...
    while (NX_TRUE)
    {
        /*
        * Receive a packet from the UDP socket.
        * If no packet is available,
        * wait for 100(TCP_SERVER_SAMPLE_DEF_TIMEOUT) timer ticks before giving up.
        */
        status = nx_udp_socket_receive(sock_ptr, &recv_ptr,
                                       UDP_CLIENT_SAMPLE_DEF_TIMEOUT);
        if (status == NX_SUCCESS)
        {
            /*
            * Get the source IP address, protocol, port number
            * and the incoming interface from the incoming packet.
            */
            status = nxd_udp_packet_info_extract(recv_ptr, &peer_ip_addr,
                                                  NX_NULL, &peer_port,
                                                  NX_NULL);

            // Get the length of the received packet.
            status = nx_packet_length_get(recv_ptr, &recv_bytes);

            // Copy data from a packet into the buffer.
            status = nx_packet_data_retrieve(recv_ptr, data_buf, &data_buflen);

            // Display received packet
            PRINTF(" =====> Received Packet(%ld) from %d.%d.%d:%d\n", recv_bytes,
                    (peer_ip_addr.nxd_ip_address.v4 >> 24) & 0x0ff,
                    (peer_ip_addr.nxd_ip_address.v4 >> 16) & 0x0ff,
                    (peer_ip_addr.nxd_ip_address.v4 >>  8) & 0x0ff,
                    (peer_ip_addr.nxd_ip_address.v4      ) & 0x0ff,
                    peer_port);

            // Allocate a packet from the packet pool
            status = nx_packet_allocate(pool_ptr,
                                        &send_ptr,
                                        NX_UDP_PACKET,
                                        NX_WAIT_FOREVER);
        }
    }
}```
2.10 UDP Server in DPM

The UDP server in the DPM sample application is an example of the simplest UDP echo application in DPM mode. The DA16200 SDK can work in DPM mode. The DPM manager feature of DA16200 SDK is helpful for the user to develop and manage a UDP server socket application in Non-DPM and DPM modes. This section describes how the UDP server in the DPM sample application is built and works.

2.10.1 How to Run

1. Open the workspace for the UDP Server in the DPM sample application as follows:
   ○ `\sample\Network\UDP_Server_DPM\build\DA16xxx.eww`
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the sample application will create a UDP socket with port number 10194 and wait for a peer message.
5. Run a socket application on the peer PC.

2.10.2 How it Works

The DA16200 UDP Server in the DPM sample application is a simple echo server. When the peer's UDP application sends a message, the DA16200 UDP server will echo that message to the peer.
2.10.3 Details

2.10.3.1 Registration

The UDP server in the DPM sample application is working in DPM mode. The basic code is similar to the UDP server sample application. The difference with the UDP server sample application is two things. First, the initial callback function is added, named `udp_server_dpm_sample_wakeup_callback` in the code. The callback function is called when the DPM state changes from sleep to wake-up. The second difference is that an additional user configuration can be stored in RTM. In this sample, peer's UDP socket port number will be stored.

```c
void udp_server_dpm_sample_init_user_config(dpm_user_config_t *user_config)
{
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = udp_server_dpm_sample_init_callback;

    //Set DPM wakkup init callback
    user_config->wakeupInitCallback = udp_server_dpm_sample_wakeup_callback;

    //Set Error callback
    user_config->errorCallback = udp_server_dpm_sample_error_callback;

    //Set session type(UDP Server)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_UDP_SERVER;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort = UDP_SERVER_DPM_SAMPLE_DEF_SERVER_PORT;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback = udp_server_dpm_sample_connect_callback;

    //Set Recv callback
    user_config->sessionConfig[session_idx].session_recvCallback = udp_server_dpm_sample_recv_callback;

    //Set secure mode
    user_config->sessionConfig[session_idx].supportSecure = NX_FALSE;

    //Set user configuration
}"
```
2.10.3.2 Data transmission

The callback function is called when a UDP packet is received from the peer's UDP socket application. In this example, the received data is printed out and an echo message is sent to the peer's UDP socket application.

```c
void udp_server_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len, ULONG rx_ip, ULONG rx_port)
{
    // Display received packet
    PRINTF(" ===> Received Packet(%ld) \n", rx_len);

    // Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf, rx_len);

    // Display sent packet
    PRINTF(" <==== Sent Packet(%ld) \n", rx_len);

dpm_mng_job_done(); // Done operation
}
```

2.11 UDP Client in DPM

The UDP client in the DPM sample application is an example of the simplest UDP echo application in DPM mode. The DA16200 SDK can work in DPM mode. The user application requires an additional operation to work in DPM mode. The DPM manager feature of the DA16200 SDK is helpful for the user to develop and manage a UDP client socket application in Non-DPM and DPM modes. This section describes how the UDP client in DPM sample application is built and works.

2.11.1 How to Run

1. Run a socket application on the peer PC and open a UDP server socket with port number 10194.
2. Open the workspace for the UDP Client in the DPM sample application as follows:
   - `\sample\Network\UDP_Client_DPM\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. After a connection is made to an AP, the sample application will create a UDP socket with port number 10194 and wait for a peer message.
6. Run a socket application on the peer PC.

2.11.2 How it Works

The DA16200 UDP Client in the DPM sample application is a simple echo message. When a peer's UDP application sends a message, then the DA16200 UDP client will echo that message to the peer.
2.11.3 Details

2.11.3.1 Registration

The UDP client in the DPM sample application works in DPM mode. The basic code is similar to the UDP client sample application. The difference with the UDP client sample application is 2 things. First, an initial callback function is added, named udp_client_dpm_sample_wakeup_callback in the code. The function is called when the DPM state changes from sleep to wake-up. The second difference is that an additional user configuration can be stored in RTM. In this example, the peer’s UDP IP address and port number will be stored.

```c
void udp_client_dpm_sample_init_user_config(dpm_user_config_t *user_config)
{
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = udp_client_dpm_sample_init_callback;

    //Set DPM wakeup init callback
    user_config->wakeupInitCallback = udp_client_dpm_sample_wakeup_callback;

    //Set Error callback
    user_config->errorCallback = udp_client_dpm_sample_error_callback;

    //Set session type (UDP Client)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_UDP_CLIENT;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort = UDP_CLIENT_DPM_SAMPLE_DEF_CLIENT_PORT;

    //Set server IP address
    memcpy(user_config->sessionConfig[session_idx].session_serverIp, srv_info.ip_addr, strlen(srv_info.ip_addr));

    //Set server port
    user_config->sessionConfig[session_idx].session_serverPort = srv_info.port;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback = udp_client_dpm_sample_connect_callback;

    //Set Recv callback
    user_config->sessionConfig[session_idx].session_recvCallback = udp_client_dpm_sample_recv_callback;
}
```

Figure 19: Workflow of UDP Client in DPM
udp_client_dpm_sample_recv_callback;

//Set connection retry count
user_config->sessionConfig[session_idx].session_conn_retry_cnt = UDP_CLIENT_DPM_SAMPLE_DEF_MAX_CONNECTION_RETRY;

//Set connection timeout
user_config->sessionConfig[session_idx].session_conn_wait_time = UDP_CLIENT_DPM_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT;

//Set auto reconnection flag
user_config->sessionConfig[session_idx].session_auto_reconn = NX_TRUE;

//Set user configuration
user_config->ptrDataFromRetentionMemory = (UCHAR *)&srv_info;
user_config->sizeOfRetentionMemory = sizeof(udp_client_dpm_sample_svr_info_t);

return ;

2.11.3.2 Data Transmission

The callback function is called when a UDP packet is received from the peer's UDP socket application. In this example, the received data is printed out and an echo message is sent to the peer's UDP socket application.

```c
void udp_client_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len, ULONG rx_ip, ULONG rx_port)
{
    //Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);
    //Echo message
    status = dpm_mng_send_to_session(SESSION1, 0, 0, (char *)rx_buf, rx_len);
    //Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
    dpm_mng_job_done();  //Done operation
}
```

2.12 TLS Server

The TLS server sample application is an example of the simplest TLS echo server application. Transport Layer Security (TLS) is cryptographic protocols designed to provide communications security over a computer network. DA16200 SDK provides SSL library, called mbed TLS, on secure H/W engine to support TLS protocol. Mbed TLS is one of popular SSL library. It's helpful to easily develop a network application with TLS protocol.

This session describes how the TLS server sample application is built and works.

2.12.1 How to Run

1. Open the workspace for the TLS Server sample application as follows:
   - \sample\Network\TLS_Server\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the example application will create a TLS server socket with port number 10197 and wait for a client connection.

5. Run a TLS client application on the peer PC.

### 2.12.2 How it Works

The DA16200 TLS Server sample is a simple echo server. When a TLS client sends a message, the DA16200 TLS server will echo that message to the TLS client.

![Figure 20: Workflow of TLS Server](image)

### 2.12.3 Details

DA16200 SDK provides mbed TLS library. It describes how TLS server is implemented with mbed TLS library and socket library.

#### 2.12.3.1 Initialization

DA16200 secure H/W engine has to be initialized with da16x_secure_module_init() before TLS context is initialized. Then, TLS context is allocated and initialized in tls_server_sample_init_ssl() function. To setup TLS session is accomplished by calling tls_server_sample_setup_ssl() function like the following.

```c
UINT tls_server_sample_setup_ssl(tls_server_sample_conf_t *config)
{
    const char *pers = "tls_server_sample";

    /*
     * Prepare the DTLS configuration by setting the endpoint and trasport type, 
     * and loading reasonable defaults for the security parameters.
     */
    ret = mbedtls_ssl_config_defaults(config->ssl_conf, 
       MBEDTLS_SSL_IS_SERVER, 
       MBEDTLS_SSL_TRANSPORT_STREAM, 
       MBEDTLS_SSL_PRESET_DEFAULT);

    /*
     * Parse one DER-encoded or one or more concatenated PEM-encoded certificates 
     * and add them to the chained list.
     */
    ret = mbedtls_x509_crt_parse(config->cert_crt, 
        tls_server_sample_cert, 
        tls_server_sample_cert_len);

    // Parse a private key in PEM or DER format.
```
ret = mbedtls_pk_parse_key(config->pkey_ctx, 
    tls_server_sample_key, 
    tls_server_sample_key_len, 
    NULL, 0);

if (mbedtls_pk_get_type(config->pkey_ctx) == MBEDTLS_PK_RSA) {
    // Initialize an RSA-alt context.
    ret = mbedtls_pk_setup_rsa_alt(config->pkey_alt_ctx, 
        (void *)mbedtls_pk_rsa(*config->pkey_ctx), 
        tls_server_sample_rsa_decrypt_func, 
        tls_server_sample_rsa_sign_func, 
        tls_server_sample_rsa_key_len_func);

    // Import certificate & private key.
    ret = mbedtls_ssl_conf_own_cert(config->ssl_conf, 
        config->cert_crt, 
        config->pkey_alt_ctx);
}
else {
    // Import certificate & private key.
    ret = mbedtls_ssl_conf_own_cert(config->ssl_conf, 
        config->cert_crt, 
        config->pkey_ctx);
}

ret = mbedtls_ctr_drbg_seed(config->ctr_drbg_ctx, 
    mbedtls_entropy_func, 
    config->entropy_ctx, 
    (const unsigned char *)pers, 
    strlen(pers));

/*
* Set the random number generator.
* DA16200 SDK supports generic callback function
* for the random number generator.
*/
 mbedtls_ssl_conf_rng(config->ssl_conf, 
    mbedtls_ctr_drbg_random, 
    config->ctr_drbg_ctx);

/*
* Set the authentication mode.
* It determines how strictly the certificates are checked.
* It, MBEDTLS_SSL_VERIFY_NONE, doesn't care verification in this sample.
*/
 mbedtls_ssl_conf_authmode(config->ssl_conf, 
    MBEDTLS_SSL_VERIFY_NONE);

// Set up an SSL context for use.
ret = mbedtls_ssl_setup(config->ssl_ctx, config->ssl_conf);

// Set callback function to know network traffic.
 mbedtls_ssl_set_bio(config->ssl_ctx, 
    (void *)config, 
    tls_server_sample_send_func, 
    tls_server_sample_recv_func, 
    NULL);
2.12.3.2 TLS Handshake

TLS is an encryption protocol designed to secure network communication. A TLS handshake is the process that kicks off a communication session that uses TLS encryption. To perform TLS handshake is accomplished by calling tls_server_sample_do_handshake() function. It simply calls mbedtls_ssl_handshake() function of mbed TLS library. If error is occurred during TLS handshake, the API returns specific error code. If TLS session is established successfully, the API returns 0. The details are as follows.

UINT tls_server_sample_do_handshake(tls_server_sample_conf_t *config) {
    // Perform the handshake
    while ((ret = mbedtls_ssl_handshake(config->ssl_ctx)) != 0) {
        if ((ret != MBEDTLS_ERR_SSL_WANT_READ) && (ret != MBEDTLS_ERR_SSL_WANT_WRITE)) {
            status = NX_NOT_SUCCESSFUL;
            break;
        }
    }
    return status;
}

2.12.3.3 Data Transmission

Encryption is a way of scrambling data so that only authorized parties can understand the information. While TLS session is established, all application data must be encrypted to transfer application data. Mbed TLS provides specific APIs and it's helpful to encrypt and decrypt data. To write application data is accomplished by calling mbedtls_ssl_write() function of mbed TLS library. In this sample, tls_server_sample_send() function calls it to send data to TLS client.

UINT tls_server_sample_send(tls_server_sample_conf_t *config, NX_PACKET *packet_ptr, ULONG wait_option) {
    ...
    // Write send_len application data bytes.
    while (((ret = mbedtls_ssl_write(config->ssl_ctx, send_data, send_len)) <= 0)
    {
        if ((ret != MBEDTLS_ERR_SSL_WANT_READ) && (ret != MBEDTLS_ERR_SSL_WANT_WRITE)) {
            status = NX_NOT_SUCCESSFUL;
            break;
        }
    }
    }

To reading application data is accomplished by calling mbedtls_ssl_read() function of mbedtls TLS library. In this sample, tls_server_sample_recv() function calls it. When it is called to read application data, tls_server_sample_recv() callback function will be called to get actual network data. This callback function is sets up during TLS setup by mbedtls_ssl_set_bio() function.
This sample application calls mbedtls_ssl_read() function twice to check received data and its length.
mbedtlssl_get_bytes_avail() function returns the number of application data bytes remaining to be read from the current record.

```c
UINT tls_server_sample_recv(tls_server_sample_conf_t *config,
        NX_PACKET **packet_ptr,
        ULONG wait_option)
{
    ...
    /*
     * Read at most 'len' application data bytes.
     * It's to check there is the received data or not.
     */
    ret = mbedtls_ssl_read(config->ssl_ctx, NULL, NULL);

    /*
     * Return the number of application data bytes remaining
     * to be read from the current record.
     */
    recv_len = mbedtls_ssl_get_bytes_avail(config->ssl_ctx);

    // Allocate memory to get application data.
    recv_buf = malloc(recv_len);

    /*
     * Read at most 'len' application data bytes.
     * It's to check there is the received data or not.
     */
    ret = mbedtls_ssl_read(config->ssl_ctx, recv_buf, recv_len);
    ...
}
```

### 2.13 TLS Server in DPM

The TLS server in the DPM sample application is an example of the simplest TLS echo server application. Transport Layer Security (TLS) is a set of cryptographic protocols designed to provide secured communication over a computer network. The DA16200 SDK can work in DPM mode. The user application requires an additional operation to work in DPM mode. The DA16200 SDK provides a DPM manager feature for the user network application. The DPM manager feature offers support for the user to develop and manage a TLS network application in Non-DPM and DPM modes. This section describes how the TLS server in the DPM sample application is built and works.

#### 2.13.1 How to Run

1. Open the workspace for the TLS Server in the DPM sample application as follows:
   - ```\sample\Network\TLS_Server_DPM\build\DA16xxx.eww```
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the example application will create a TLS server socket with port number 10197 and wait for a client connection.
5. Run a TLS client application on the peer PC.
2.13.2 How it Works

The DA16200 TLS Server in the DPM sample is a simple echo server. When a TLS client sends a message, then the DA16200 TLS server will echo that message to the TLS client. The DA16200 TLS server will take time to wait to establish a TLS session.

![Diagram showing the workflow of TLS Server in DPM](image)

**Figure 21: Workflow of TLS Server in DPM**

2.13.3 Details

2.13.3.1 Registration

The TLS server in the DPM sample application works in DPM mode. The basic code is similar to the TLS server sample application. The difference with the TLS server sample application is 2 things. First, an initial callback function is added, named `tls_server_dpm_sample_wakeup_callback` in the code. The function is called when the DPM state changes from sleep to wake-up. The second difference is that an additional user configuration can be stored in RTM. In this sample, the TLS server information will be stored.

```c
void tls_server_dpm_sample_init_user_config(dpm_user_config_t *user_config)
{
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = tls_server_dpm_sample_init_callback;

    //Set DPM wakkup init callback
    user_config->wakeupInitCallback = tls_server_dpm_sample_wakeup_callback;

    //Set Error callback
    user_config->errorCallback = tls_server_dpm_sample_error_callback;

    //Set session type (TCP Server)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_TCP_SERVER;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort = TLS_SERVER_DPM_SAMPLE_DEF_SERVER_PORT;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback =
        tls_server_dpm_sample_connect_callback;

    //Set Recv callback
    user_config->sessionConfig[session_idx].session_recvCallback =
        tls_server_dpm_sample_recv_callback;

    //Set secure mode
```
user_config->sessionConfig[session_idx].supportSecure = NX_TRUE;

// Set secure setup callback
user_config->sessionConfig[session_idx].session_setupSecureCallback =
    tls_server_dpm_sample_secure_callback;

// Set user configuration
user_config->ptrDataFromRetentionMemory = (UCHAR *)&srv_info;
user_config->sizeOfRetentionMemory = sizeof(tls_server_dpm_sample_srv_info_t);
return;
}

2.13.3.2 TLS Setup

In order to establish a TLS session, TLS should be setup. DA16200 includes an mbedTLS library to provide the TLS protocol. Most APIs that are related to the TLS protocol are based on an mbedTLS library. TLS will be setup by session_setupSecureCallback function. The details are as follows.

```c
void tls_server_dpm_sample_secure_callback(void *config)
{
    const char *pers = "tls_server_dpm_sample";
    SECURE_INFO_T *secure_config = (SECURE_INFO_T *)config;

    ret = mbedtls_ssl_config_defaults(secure_config->ssl_conf,
        MBEDTLS_SSL_IS_SERVER,
        MBEDTLS_SSL_TRANSPORT_STREAM,
        MBEDTLS_SSL_PRESET_DEFAULT);

    // Import test certificate
    ret = mbedtls_x509_crt_parse(secure_config->cert_crt,
        tls_server_dpm_sample_cert,
        tls_server_dpm_sample_cert_len);

    ret = mbedtls_pk_parse_key(secure_config->pkey_ctx,
        tls_server_dpm_sample_key,
        tls_server_dpm_sample_key_len,
        NULL, 0);

    if (mbedtls_pk_get_type(secure_config->pkey_ctx) == MBEDTLS_PK_RSA)
    {
        ret = mbedtls_pk_setup_rsa_alt(secure_config->pkey_alt_ctx,
            (void *)mbedtls_pk_rsa(*secure_config->pkey_ctx),
            tls_server_dpm_sample_rsa_decrypt_func,
            tls_server_dpm_sample_rsa_sign_func,
            tls_server_dpm_sample_rsa_key_len_func);

        ret = mbedtls_ssl_conf_own_cert(secure_config->ssl_conf,
            secure_config->cert_crt,
            secure_config->pkey_alt_ctx);
    }
    else
    {
        ret = mbedtls_ssl_conf_own_cert(secure_config->ssl_conf,
            secure_config->cert_crt,
            secure_config->pkey_ctx);
    }

    ret = mbedtls_ctr_drbg_seed(secure_config->ctr_drbg_ctx,
```
2.13.3.3 Data Transmission

The callback function is called when a TLS packet is received from a TLS client. In this sample, the received data is printed out and an echo message is sent to the TLS server. Data will be encrypted and decrypted in the callback function.

```c
void tls_server_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len,
                                         ULONG rx_ip, ULONG rx_port)
{
    //Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);

    //Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf,
                                     rx_len);

    //Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", rx_len);

    dpm_mng_job_done(); //Done opertaion
}
```

2.14 TLS Client

The TLS client sample application is an example of the simplest TLS echo client application. Transport Layer Security (TLS) is cryptographic protocols designed to provide communications security over a computer network. DA16200 SDK provides DPM manager feature for user network application. DA16200 SDK provides SSL library, called mbed TLS, on secure H/W engine to support TLS protocol. Mbed TLS is one of popular SSL library. It's helpful to easily develop a network application with TLS protocol.

This session describes how the TLS client sample application is built and worked.
2.14.1 How to Run

1. Run a TLS server application on the peer PC and open a TLS server socket with port number 10196.
2. Open the workspace for the TLS Client sample application as follows:
   ○ `\sample\Network\TLS_Client\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the TLS server IP address and the port number as you created the socket on the peer PC with the following console command and then reboot. These parameters can also be defined in the source code.

   ```
   [/DA16200] # nvram.setenv TLSC_SERVER_IP 192.168.0.11
   [/DA16200] # nvram.setenv TLSC_SERVER_PORT 10196
   [/DA16200] # reboot
   ○ After a connection is made to an AP, the example application will connect to the peer PC.
   ```

2.14.2 How it Works

The DA16200 TLS Client sample is a simple echo message. When TLS server sends a message, then the DA16200 TLS client will echo that message to the TLS server.

![Figure 22: Workflow of TLS Client](image-url)

2.14.3 Details

DA16200 SDK provides mbed TLS library. It describes how TLS client is implemented with mbed TLS library and socket library.

2.14.3.1 Registration

DA16200 secure H/W engine has to be initialized with da16x_secure_module_init() before TLS context is initialized. Then, TLS context is allocated and initialized in tls_client_sample_init_ssl() function. To setup TLS session is accomplished by calling tls_client_sample_setup_ssl() function like the following.

```c
UINT tls_client_sample_setup_ssl(tls_client_sample_conf_t *config) {
    const char *pers = "tls_client_sample";

    /*
     * Prepare the TLS configuration by setting the endpoint and trasport type,
     * and loading reasonable defaults for the security parameters.
     */
```
2.14.3.2 TLS Handshake

TLS is an encryption protocol designed to secure network communication. A TLS handshake is the process that kicks off a communication session that uses TLS encryption. To perform TLS handshake is accomplished by calling `tls_client_sample_do_handshake()` function. It simply calls `mbedtls_ssl_handshake()` function of mbed TLS library. If error is occurred during TLS handshake, the API returns specific error code. If TLS session is established successfully, the API returns 0.

```c
UINT tls_client_sample_do_handshake(tls_client_sample_conf_t *config, ULONG wait_option)
{
    // Perform TLS handshake
    while ((ret = mbedtls_ssl_handshake(config->ssl_ctx)) != 0)
    {
        if (((ret != MBEDTLS_ERR_SSL_WANT_READ) && (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
        {
            return NX_NOT_SUCCESSFUL;
        }
    }
}```
2.14.3.3 Data Transmission

Encryption is a way of scrambling data so that only authorized parties can understand the information. During TLS session is established, all data must be encrypted to transfer application data. Mbed TLS provides specific APIs and it's helpful to encrypt and decrypt data. To write application data is accomplished by calling mbedtls_ssl_write() function of mbedtls TLS library. In this sample, tls_client_sample_send() function calls it to send data to TLS client.

```c
UINT tls_client_sample_send(tls_client_sample_conf_t *config, NX_PACKET *packet_ptr, ULONG wait_option)
{
    ...
    // Write send_len application data bytes.
    while ((ret = mbedtls_ssl_write(config->ssl_ctx, send_data, send_len)) <= 0)
    {
        if ((ret != MBEDTLS_ERR_SSL_WANT_READ) && (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
        {
            status = NX_NOT_SUCCESSFUL;
            break;
        }
    }
    ...
}
```

To reading application data is accomplished by calling mbedtls_ssl_read() function of mbedtls TLS library. In this sample, tls_client_sample_recv() function calls it. When it is called to read application data, tls_client_sample_recv_func() callback function will be called to get actual network data. This callback function is set up during TLS setup by mbedtls_ssl_set_bio() function.

This sample application calls mbedtls_ssl_read() function twice to check received data and its length. mbedtls_ssl_get_bytes_avail() function returns the number of application data bytes remaining to be read from the current record.

```c
UINT tls_client_sample_recv(tls_client_sample_conf_t *config, NX_PACKET **packet_ptr, ULONG wait_option)
{
    ...
    /*
    * Read at most 'len' application data bytes.
    * It's to check there is the received data or not.
    */
    ret = mbedtls_ssl_read(config->ssl_ctx, NULL, NULL);

    /*
    * Return the number of application data bytes remaining
    * to be read from the current record.
    */
    recv_len = mbedtls_ssl_get_bytes_avail(config->ssl_ctx);

    // Allocate memory to get application data.
    recv_buf = malloc(recv_len);
    /*
DA16200 Example Application Guide

* Read at most 'len' application data bytes.
* It's to check there is the received data or not.
*/
ret = mbedtls_ssl_read(config->ssl_ctx, recv_buf, recv_len);
...

2.15 TLS Client in DPM

The TLS client in the DPM sample application is an example of the simplest TLS echo client application in DPM mode. Transport Layer Security (TLS) is a set of cryptographic protocols designed to provide secured communication over a computer network. The DA16200 SDK can work in DPM mode. The user application requires an additional operation to work in DPM mode. The DA16200 SDK provides a DPM manager feature for the user network application. The DPM manager feature supports the user to develop and manage the TLS network application in Non-DPM and DPM modes. This section describes how the TLS client in the DPM sample application is built and works.

2.15.1 How to Run

1. Run a TLS server application on the peer PC and open a TLS server socket with port number 10196.
2. Open the workspace for a TCP Client in the DPM sample application as follows:
   - \sample\Network\TLS_Client\build\DA16xxx.eww
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the TLS server IP address and the port number as you created the socket on the peer PC with the following console command and then reboot. These parameters can also be defined in the source code.

   [/DA16200] # nvram.setenv TLSC_SERVER_IP 192.168.0.11
   [/DA16200] # nvram.setenv TLSC_SERVER_PORT 10196
   [/DA16200] # reboot
   - After a connection is made to an AP, the example application will connect to the peer PC.

2.15.2 How it Works

The DA16200 TLS Client in the DPM sample is a simple echo message. When a TLS server sends a message, then the DA16200 TLS client will echo that message to the TLS server.

Figure 23: Workflow of TLS Client in DPM
2.15.3 Details

2.15.3.1 Registration

The TLS client in the DPM sample application works in DPM mode. The basic code is similar to the TLS client sample application. The difference with the TLS client sample application is 2 things. First, an initial callback function is added, named `tls_client_dpm_sample_wakeup_callback` in the code. It will be called when DPM state changes from sleep to wake-up. The second is an additional user configuration that can be stored in RTM. In this example, TLS server information will be stored.

```c
void tls_client_dpm_sample_init_user_config(dpm_user_config_t *user_config) {
    const int session_idx = 0;
    // Set Boot init callback
    user_config->bootInitCallback = tls_client_dpm_sample_init_callback;
    // Set DPM wakeup init callback
    user_config->wakeupInitCallback = tls_client_dpm_sample_wakeup_callback;
    // Set External wakeup callback
    user_config->externWakeupCallback = tls_client_dpm_sample_external_callback;
    // Set Error callback
    user_config->errorCallback = tls_client_dpm_sample_error_callback;
    // Set session type (TLS Client)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_TCP_CLIENT;
    // Set local port
    user_config->sessionConfig[session_idx].session_myPort =
        DM_TLS_CLIENT_SAMPLE_DEF_CLIENT_PORT;
    // Set server IP address
    memcpy(user_config->sessionConfig[session_idx].session_serverIp,
           srv_info.ip_addr, strlen(srv_info.ip_addr));
    // Set server port
    user_config->sessionConfig[session_idx].session_serverPort = srv_info.port;
    // Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback =
        tls_client_dpm_sample_connect_callback;
    // Set Recv callback
    user_config->sessionConfig[session_idx].session_recvCallback =
        tls_client_dpm_sample_recv_callback;
    // Set connection retry count
    user_config->sessionConfig[session_idx].session_conn_retry_cnt =
        DM_TLS_CLIENT_SAMPLE_DEF_MAX_CONNECTION_RETRY;
    // Set connection timeout
    user_config->sessionConfig[session_idx].session_conn_wait_time =
        DM_TLS_CLIENT_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT;
    // Set auto reconnection flag
    user_config->sessionConfig[session_idx].session_auto_reconn = NX_TRUE;
    // Set secure mode
    user_config->sessionConfig[session_idx].supportSecure = NX_TRUE;
}
```
//Set secure setup callback
user_config->sessionConfig[session_idx].session_setupSecureCallback =
  tls_client_dpm_sample_secureCallback;

//Set user configuration
user_config->ptrDataFromRetentionMemory = (UCHAR *)&srv_info;
user_config->sizeOfRetentionMemory = sizeof(tls_client_dpm_sample_svr_info_t);

return ;
}

2.15.3.2 TLS Setup

In order to establish a TLS session, TLS should be setup. DA16200 includes an mbedTLS library to provide the TLS protocol. Most APIs that are related to the TLS protocol are based on an mbedTLS library. TLS will be setup by session_setupSecureCallback function. The details are as shown below. Note that this sample application does not include certificates.

```c
void tls_client_dpm_sample_secure_callback(void *config)
{
    const char *pers = "tls_client_sample";
    SECURE_INFO_T *secure_config = (SECURE_INFO_T *)config;

    ret = mbedtls_ssl_config_defaults(secure_config->ssl_conf,
      MBEDTLS_SSL_IS_CLIENT,
      MBEDTLS_SSL_TRANSPORT_STREAM,
      MBEDTLS_SSL_PRESET_DEFAULT);

    ret = mbedtls_ctr_drbg_seed(secure_config->ctr_drbg_ctx,
      mbedtls_entropy_func,
      secure_config->entropy_ctx,
      (const unsigned char *)pers,
      strlen(pers));

    mbedtls_ssl_conf_rng(secure_config->ssl_conf,
      mbedtls_ctr_drbg_random,
      secure_config->ctr_drbg_ctx);

    //Don't care verification in this sample.
    mbedtls_ssl_conf_authmode(secure_config->ssl_conf, MBEDTLS_SSL_VERIFY_NONE);

    ret = mbedtls_ssl_setup(secure_config->ssl_ctx, secure_config->ssl_conf);

    dpm_mng_job_done(); //Done operation

    return ;
}
```

2.15.3.3 Data Transmission

The callback function is called when the TLS packet is received from the TLS server. In this sample, the received data is printed out and an echo message is sent to the TLS server. Data will be encrypted and decrypted in the callback function.

```c
void tls_client_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len,
                                        ULONG rx_ip, ULONG rx_port)
{
```
//Display received packet
PRINTF(" =====> Received Packet(%ld) \n", rx_len);

//Echo message
status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf, rx_len);

//Display sent packet
PRINTF(" <===== Sent Packet(%ld) \n", rx_len);

dpm_mng_job_done(); //Done opertaion

2.16 DTLS Server

The DTLS server sample application is an example of the simplest DTLS echo server application. Datagram Transport Layer Security (DTLS) is cryptographic protocol that provides security for datagram-based applications by allowing them to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. DA16200 SDK provides SSL library, called mbed TLS, on secure H/W engine to support DTLS protocol. Mbed TLS is one of popular SSL library. It's helpful to easily develop a network application with DTLS protocol.

This session describes how the DTLS server sample application is built and worked.

2.16.1 How to Run

1. Open the workspace for the DTLS Server sample application as follows:
   ○ \sample\Network\DTLS_Server\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the example application will create a DTLS server socket with port number 10199 and wait for a client connection.
5. Run a DTLS client application on the peer PC.

2.16.2 How it Works

The DA16200 DTLS Server sample is a simple echo server. When DTLS client sends a message, then the DA16200 DTLS server will echo that message to the DTLS client.

![Figure 24: Workflow of DTLS Server](image-url)
2.16.3 Details

DA16200 SDK provides mbed TLS library. This sample application describes how mbed TLS library is called and applied for socket library.

2.16.3.1 Initialization

DA16200 secure H/W engine has to be initialized with da16x_secuer_module_init() before DTLS context is initialized. Then, DTLS context is allocated and initialized in dtls_server_sample_init_ssl() function. To setup DTLS session is accomplished by calling dtls_server_sample_setup_ssl() function like the following. DTLS session is established over UDP protocol. In case of losing packet, retransmission is required. So, the timer is registered to retransmit packet by mbedtls_ssl_set_timer_cb() function.

```c
UINT dtls_server_sample_setup_ssl(dtls_server_sample_conf_t *config)
{
    const char *pers = "dtls_server_sample";

    /*
    * Prepare the DTLS configuration by setting the endpoint and trasport type,
    * and loading reasonable defaults for the security parameters.
    */
    ret = mbedtls_ssl_config_defaults(config-&ssl_conf,
   MBEDTLS_SSL_IS_SERVER,
   MBEDTLS_SSL_TRANSPORT_DATAGRAM,
   MBEDTLS_SSL_PRESET_DEFAULT);

    /*
    * Parse one DER-encoded or one or more concatenated PEM-encoded certificates
    * and add them to the chained list.
    */
    ret = mbedtls_x509_crt_parse(config-&cert_crt,
    dtls_server_sample_cert,
    dtls_server_sample_cert_len);

    /*
    * Parse a private key in PEM or DER format.
    */
    ret = mbedtls_pk_parse_key(config-&pkey_ctx,
    dtls_server_sample_key,
    dtls_server_sample_key_len,
    NULL, 0);

    if (mbedtls_pk_get_type(config-&pkey_ctx) == MBEDTLS_PK_RSA)
    {
        // Initialize an RSA-alt context.
        ret = mbedtls_pk_setup_rsa_alt(config-&pkey_alt_ctx,
            (void *)mbedtls_pk_rsa(*config-&pkey_ctx),
            dtls_server_sample_rsa_decrypt_func,
            dtls_server_sample_rsa_sign_func,
            dtls_server_sample_rsa_key_len_func);

        // Import certificate & private key
        ret = mbedtls_ssl_conf_own_cert(config-&ssl_conf,
            config-&cert_crt,
            config-&pkey_alt_ctx);
    }
    else
    {
        // Import certificate & private key
        ret = mbedtls_ssl_conf_own_cert(config-&ssl_conf,
```
config->cert_crt,
config->pkey_ctx);

ret = mbedtls_ctr_drbg_seed(config->ctr_drbg_ctx,
mbedtls_entropy_func,
config->entropy_ctx,
(config unsigned char *)pers,
(strlen(pers));

/*
 * Set the random number generator.
 * DA16200 SDK supports generic callback function for the random number
generator.
*/
mbedtls_ssl_conf_rng(config->ssl_conf,
    mbedtls_ctr_drbg_random,
    config->ctr_drbg_ctx);

// Setup cookie context
ret = mbedtls_ssl_cookie_setup(config->cookie_ctx,
    mbedtls_ctr_drbg_random,
    config->ctr_drbg_ctx);

// Register callbacks for DTLS cookies.
mbedtls_ssl_conf_dtls_cookies(config->ssl_conf, mbedtls_ssl_cookie_write,
    mbedtls_ssl_cookie_check,
    config->cookie_ctx);

/*
 * Set the authentication mode.
 * It determines how strictly the certificates are checked.
*/
mbedtls_ssl_conf_authmode(config->ssl_conf, MBEDTLS_SSL_VERIFY_NONE);

// Set the anti-replay protection for DTLS.
mbedtls_ssl_conf_dtls_anti_replay(config->ssl_conf,
    MBEDTLS_SSL_ANTI_REPLAY_ENABLED);

mbedtls_ssl_conf_read_timeout(config->ssl_conf,
    DTLS_SERVER_SAMPLE_DEF_TIMEOUT * 10);

// Set retransmit timeout values for the DTLS handshake.
mbedtls_ssl_conf_handshake_timeout(config->ssl_conf,
    DTLS_SERVER_SAMPLE_HANDSHAKE_MIN_TIMEOUT *
    10,
    DTLS_SERVER_SAMPLE_HANDSHAKE_MAX_TIMEOUT *
    10);

// Set an SSL context for use.
ret = mbedtls_ssl_setup(config->ssl_ctx, config->ssl_conf);

// Set timer
mbedtls_ssl_set_timer_cb(config->ssl_ctx,
    config->timer,
    dtls_server_sample_timer_start,
    dtls_server_sample_timer_get_status);

// Set callback function to know network traffic
mbedtls_ssl_set_bio(config->ssl_ctx,
(void *)config,
        dtls_server_sample_send_func,
        NULL,
        dtls_server_sample_recv_func);

        return NX_SUCCESS;
    }

2.16.3.2 DTLS Handshake

DTLS is an encryption protocol designed to secure network communication. A DTLS handshake is the process that kicks off a communication session that uses DTLS encryption. To perform DTLS handshake is accomplished by calling dtls_server_sample_do_handshake() function. DTLS server must verify cookies for DTLS client. The DTLS client’s transport-level identification information has to be set up (generally an IP Address). After receiving ClientHello message, DTLS server must set up its IP address. Then, DTLS handshake should be retried like the following.

```
UINT dtls_server_sample_do_handshake(dtls_server_sample_conf_t *config)
{
    unsigned char ip_buf[16] = {0x00,};

    process_handshake:
    ret = dtls_server_sample_recv_func((void *)config, NULL, 0,
        DTLS_SERVER_SAMPLE_CONN_TIMEOUT);

    // Check client's IP address
    if (((config->peer_info.ip_addr.nxd_ip_address.v4 == 0)
        || (config->peer_info.port == 0))
    {
        return NX_NOT_SUCCESSFUL;
    }

    memcpy(ip_buf, &config->peer_info.ip_addr.nxd_ip_address.v4, sizeof(ULONG));

    // Set client's transport-level identification info.
    ret = mbedtls_ssl_set_client_transport_id(config->ssl_ctx, ip_buf,
        sizeof(ULONG));

    while ((ret = mbedtls_ssl_handshake(config->ssl_ctx)) != 0)
    {
        if ((ret !=MBEDTLS_ERR_SSL_WANT_READ)
            && (ret !=MBEDTLS_ERR_SSL_WANT_WRITE))
        {
            if (ret !=MBEDTLS_ERR_SSL_HELLO_VERIFY_REQUIRED)
                PRINTF("[%s] Failed to process dtls handshake(0x%x)\n", __func__,
                    ret);
            status = NX_NOT_SUCCESSFUL;
            break;
        }
    }

    // Retry DTLS handshake if client's IP address is set up and server's state is HelloVerify.
    if (ret ==MBEDTLS_ERR_SSL_HELLO_VERIFY_REQUIRED)
    {
        status = dtls_server_sample_shutdown_ssl(config);
        if (status == NX_SUCCESS)
            status = NX_SUCCESS;
    }
```
2.16.3.3 Data Transmission

Encryption is a way of scrambling data so that only authorized parties can understand the inform. During DTLS session is established, all data must be encrypted to transfer application data. Mbed TLS provides specific APIs and it's helpful to encrypt and decrypt data. To write application data is accomplished by calling mbedtls_ssl_read() function of mbedtls TLS library. In this sample, dtls_server_sample_send() function calls it to send data to DTLS client.

```c
UINT dtls_server_sample_send(dtls_server_sample_conf_t *config,
                       NX_PACKET *packet_ptr,
                       ULONG wait_option)
{
...
    // Write send_len application data bytes.
    while ((ret = mbedtls_ssl_write(config->ssl_ctx, data, len)) <= 0)
    {
        if ((ret != MBEDTLS_ERR_SSL_WANT_READ)
            && (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
        {
            status = NX_NOT_SUCCESSFUL;
            break;
        }
    }
...
}
```

To reading application data is accomplished by calling mbedtls_ssl_read() function of mbedtls TLS library. In this sample, dtls_server_sample_recv() function calls it. When it is called to read application data, dtls_server_sample_recv_func() callback function will be called to get actual network data. This callback function is sets up during DTLS setup by mbedtls_ssl_set_bio() function.

This sample application calls mbedtls_ssl_read() function twice to check received data and its length. mbedtls_ssl_get_bytes_avail() function returns the number of application data bytes remaining to be read from the current record.

```c
UINT dtls_server_sample_recv(dtls_server_sample_conf_t *config,
                       NX_PACKET **packet_ptr,
                       ULONG wait_option)
{
...
    /*
    * Read at most 'len' application data bytes.
    * It's to check there is the received data or not.
    */
    ret = mbedtls_ssl_read(config->ssl_ctx, NULL, NULL);

    /*
    * Return the number of application data bytes remaining
    * to be read from the current record.
    */
    recv_len = mbedtls_ssl_get_bytes_avail(config->ssl_ctx);
```
2.17 DTLS Server in DPM

The DTLS server in the DPM sample application is an example of the simplest DTLS echo server application. Datagram Transport Layer Security (DTLS) is a cryptographic protocol that provides security for datagram-based applications by allowing them to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. The DA16200 SDK can work in DPM mode. The user application requires an additional operation to work in DPM mode. The DA16200 SDK provides a DPM manager feature for the user network application. The DPM manager feature supports the user to develop and manage a DTLS network application in Non-DPM and DPM modes. This section describes how the DTLS server in the DPM sample application is built and works.

2.17.1 How to Run

1. Open the workspace for the DTLS Server in the DPM sample application as follows:
   ○ .\sample\Network\DTLS_Server_DPM\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console command to setup the Wi-Fi station interface.
4. After a connection is made to an AP, the example application will create a DTLS server socket with port number 10199 and wait for a client connection.
5. Run a DTLS client application on the peer PC.

2.17.2 How it Works

The DA16200 DTLS Server in the DPM sample is a simple echo server. When a DTLS client sends a message, then the DA16200 DTLS server will echo that message to the DTLS client.

![Figure 25: Workflow of DTLS Server in DPM](Image)
2.17.3 Details

2.17.3.1 Registration

The DTLS server in the DPM sample application works in DPM mode. The basic code is similar to the DTLS server sample application. The difference with the DTLS server sample application is 2 things. First, an initial callback function is added, named dtls_server_dpm_sample_wakeup_callback in the code. It will be called when the DPM state changes from sleep to wake-up. The second is that an additional user configuration can be stored in RTM. In this sample, DTLS server information will be stored.

```c
void dtls_server_dpm_sample_init_user_config(dpm_user_config_t *user_config)
{
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = dtls_server_dpm_sample_init_callback;

    //Set DPM wakkup init callback
    user_config->wakeupInitCallback = dtls_server_dpm_sample_wakeup_callback;

    //Set Error callback
    user_config->errorCallback = dtls_server_dpm_sample_error_callback;

    //Set session type(UDP Server)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_UDP_SERVER;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort =
        DTLS_SERVER_DPM_SAMPLE_DEF_SERVER_PORT;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback =
        dtls_server_dpm_sample_connect_callback;

    //SetRecv callback
    user_config->sessionConfig[session_idx].session_recvCallback =
        dtls_server_dpm_sample_recv_callback;

    //Set secure mode
    user_config->sessionConfig[session_idx].supportSecure = NX_TRUE;

    //Set secure setup callback
    user_config->sessionConfig[session_idx].session_setupSecureCallback =
        dtls_server_dpm_sample_secure_callback;

    //Set user configuration
    user_config->ptrDataFromRetentionMemory = (UCHAR *)&srv_info;
    user_config->sizeOfRetentionMemory =
        sizeof(dtls_server_dpm_sample_svr_info_t);

    return ;
}
```

2.17.3.2 DTLS Setup

In order to establish a DTLS session, DTLS should be setup. The DA16200 includes an mbedTLS library to provide the DTLS protocol. Most APIs that are related to the DTLS protocol are based on an mbedTLS library. DTLS will be setup by function session_setupSecureCallback. The details are as follows.
void dtls_server_dpm_sample_secure_callback(void *config)
{
    const char *pers = "dtls_server_dpm_sample";

    SECURE_INFO_T *secure_config = (SECURE_INFO_T *)config;

    ret = mbedtls_ssl_config_defaults(secure_config->ssl_conf, MBEDTLS_SSL_IS_SERVER, MBEDTLS_SSL_TRANSPORT_DATAGRAM, MBEDTLS_SSL_PRESET_DEFAULT);

    //import test certificate
    ret = mbedtls_x509_crt_parse(secure_config->cert_crt, dtls_server_dpm_sample_cert, dtls_server_dpm_sample_cert_len);
    ret = mbedtls_pk_parse_key(secure_config->pkey_ctx, dtls_server_dpm_sample_key, dtls_server_dpm_sample_key_len, NULL, 0);

    if (mbedtls_pk_get_type(secure_config->pkey_ctx) == MBEDTLS_PK_RSA)
    {
        ret = mbedtls_pk_setup_rsa_alt(secure_config->pkey_alt_ctx, (void *)mbedtls_pk_rsa(*secure_config->pkey_ctx), dtls_server_dpm_sample_rsa_decrypt_func, dtls_server_dpm_sample_rsa_sign_func, dtls_server_dpm_sample_rsa_key_len_func);

        ret = mbedtls_ssl_conf_own_cert(secure_config->ssl_conf, secure_config->cert_crt, secure_config->pkey_alt_ctx);
    }
    else
    {
        ret = mbedtls_ssl_conf_own_cert(secure_config->ssl_conf, secure_config->cert_crt, secure_config->pkey_ctx);
    }

    ret = mbedtls_ctr_drbg_seed(secure_config->ctr_drbg_ctx, mbedtls_entropy_func, secure_config->entropy_ctx, (const unsigned char *)pers, strlen(pers));

    mbedtls_ssl_conf_rng(secure_config->ssl_conf, mbedtls_ctr_drbg_random, secure_config->ctr_drbg_ctx);
    mbedtls_ssl_cookie_setup(secure_config->cookie_ctx, mbedtls_ctr_drbg_random, secure_config->ctr_drbg_ctx);
    mbedtls_ssl_conf_dtls_cookies(secure_config->ssl_conf, mbedtls_ssl_cookie_write, mbedtls_ssl_cookie_check, secure_config->cookie_ctx);

    //Don't care verification in this sample.
    mbedtls_ssl_conf_authmode(secure_config->ssl_conf, MBEDTLS_SSL_VERIFY_NONE);
mbedtls_ssl_conf_max_frag_len(secure_config->ssl_conf, 0);

mbedtls_ssl_conf_dtlsAntiReplay(secure_config->ssl_conf, MBEDTLS_SSL_ANTI_REPLAY_ENABLED);

mbedtls_ssl_conf_read_timeout(secure_config->ssl_conf,
        DTLS_SERVER_DPM_SAMPLE_RECEIVE_TIMEOUT * 10);

mbedtls_ssl_conf_handshake_timeout(secure_config->ssl_conf,
        DTLS_SERVER_DPM_SAMPLE_HANDSAKETIME_MIN_TIMEOUT * 10,
        DTLS_SERVER_DPM_SAMPLE_HANDSAKETIME_MAX_TIMEOUT * 10);

ret = mbedtls_ssl_setup(secure_config->ssl_ctx, secure_config->ssl_conf);

dpm_mng_job_done(); //Done operation

return ;

2.17.3.3 Data Transmission

The callback function will be called when a DTLS packet is received from the DTLS client. In this example, the received data is printed out and an echo message is sent to the DTLS server. Data will be encrypted and decrypted in the callback function.

```c
void dtls_server_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len, ULONG rx_ip, ULONG rx_port)
{
    //Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);

    //Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf, rx_len);

    //Display sent packet
    PRINTF(" <==== Sent Packet(%ld) \n", rx_len);

    dpm_mng_job_done(); //Done operation
}
```

2.18 DTLS Client

The DTLS client sample application is an example of the simplest DTLS echo client application. Datagram Transport Layer Security (DTLS) is a cryptographic protocol that provides security for datagram-based applications by allowing them to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. DA16200 SDK provides SSL library, called mbedTLS, on secure HW engine to support DTLS protocol. Mbed TLS is one of popular SSL library. It's helpful to easily develop a network application with DTLS protocol.

This session describes how the DTLS client sample application is built and worked.

2.18.1 How to Run

1. Run a DTLS server application on the peer PC and open a DTLS server socket with port number 10199.
2. Open the workspace for the DTLS Client sample application as follows:
   - `\sample\Network\DTLS_Client\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to setup the Wi-Fi station interface.
DA16200 Example Application Guide

5. Set the DTLS server IP address and the port number as you created the socket on the peer PC with the following console command and then reboot. These parameters can also be defined in the source code.

```
[DA16200] # nvram.setenv DTLS_SERVER_IP 192.168.0.11
[DA16200] # nvram.setenv DTLS_SERVER_PORT 10199
[DA16200] # reboot
```

○ After a connection is made to an AP, the sample application will connect to the peer PC.

2.18.2 How it Works

The DA16200 DTLS Client sample is a simple echo message. When the DTLS server sends a message, then the DA16200 DTLS client will echo that message to the DTLS server.

```
Figure 26: Workflow of DTLS Client
```

2.18.3 Details

DA16200 SDK provides mbed TLS library. This sample application describes how mbed TLS library is called and applied for socket library.

2.18.3.1 Initialization

DA16200 secure H/W engine has to be initialized with da16x_seucer_module_init() before DTLS context is initialized. Then, DTLS context is allocated and initialized in dtls_client_sample_init_ssl() function. To setup DTLS session is accomplished by calling dtls_client_sample_setup_ssl() function like the following. DTLS session is established over UDP protocol. In case of losing packet, retransmission is required. So, the timer is registered to retransmit packet by mbedtls_ssl_set_timer_cb() function.

```c
UINT dtls_client_sample_setup_ssl(dtls_client_sample_conf_t *config)
{
    const char *pers = "dtls_client_sample";

    /*
    * Prepare the DTLS configuration by setting the endpoint and trasport type,
    * and loading reasonable defaults for the security parameters.
    */
    ret = mbedtls_ssl_conf_defaults(config->ssl_conf,
                                     MBEDTLS_SSL_IS_CLIENT,
                                     MBEDTLS_SSL_TRANSPORT_DATAGRAM,
                                     MBEDTLS_SSL_PRESET_DEFAULT);

    ret = mbedtls_ctr_drbg_seed(config->ctr_drbg_ctx,
                                mbedtls_entropy_func,
```

User Manual  Revision 1.7  18-May-2020

 CFR0012  65 of 200 © 2020 Dialog Semiconductor
DA16200 Example Application Guide

```c
config->entropy_ctx,
(const unsigned char *)pers,
strlen(pers));

/*
 * Set the random number generator.
 * DA16200 SDK supports generic callback function for the random number generator.
 */
mbedtls_ssl_conf_rng(config->ssl_conf,
    mbedtls_ctr_drbg_random,
    config->ctr_drbg_ctx);

/*
 * Set the authentication mode.
 * It determines how strictly the certificates are checked.
 */
mbedtls_ssl_conf_authmode(config->ssl_conf, MBEDTLS_SSL_VERIFY_NONE);

// Set the anti-replay protection for DTLS.
mbedtls_ssl_conf_dtls_anti_replay(config->ssl_conf,
    MBEDTLS_SSL_ANTI_REPLAY_ENABLED);

mbedtls_ssl_conf_read_timeout(config->ssl_conf,
    DTLS_CLIENT_SAMPLE_DEF_TIMEOUT * 10);

// Set retransmit timeout values for the DTLS handshake.
mbedtls_ssl_conf_handshake_timeout(config->ssl_conf,
    DTLS_CLIENT_SAMPLE_HANDSAHK_MIN_TIMEOUT *
    10,
    DTLS_CLIENT_SAMPLE_HANDSAXH_MAX_TIMEOUT *
    10);

// Set an SSL context for use.
ret = mbedtls_ssl_setup(config->ssl_ctx, config->ssl_conf);

// Set timer
mbedtls_ssl_set_timer_cb(config->ssl_ctx,
    &config->timer,
    dtls_client_sample_timer_start,
    dtls_client_sample_timer_get_status);

// Set callback function to know network traffic.
mbedtls_ssl_set_bio(config->ssl_ctx,
    (void *)&config,
    dtls_client_sample_send_func,
    NULL,
    dtls_client_sample_recv_func);

return NX_SUCCESS;
```

2.18.3.2 DTLS Handshake

DTLS is an encryption protocol designed to secure network communication. A DTLS handshake is the process that kicks off a communication session that uses DTLS encryption. To perform DTLS handshake is accomplished by calling `dtls_client_sample_do_handshake()` function. If error is occurred during DTLS handshake, the API returns specific error code. If DTLS session is established successfully, the API returns 0.
UINT dtls_client_sample_do_handshake(dtls_client_sample_conf_t *config)
{
    while ((ret = mbedtls_ssl_handshake(config->ssl_ctx)) != 0)
    {
        if ((ret != MBEDTLS_ERR_SSL_WANT_READ)
            && (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
        {
            break;
        }
    }
    return status;
}

2.18.3.3 Data Transmission

Encryption is a way of scrambling data so that only authorized parties can understand the information. During DTLS session is established, all data must be encrypted to transfer application data. Mbed TLS provides specific APIs and it's helpful to encrypt and decrypt data. To write application data is accomplished by calling mbedtls_ssl_read() function of mbedtls TLS library. In this sample, dtls_client_sample_send() function calls it to send data to DTLS client.

UINT dtls_server_sample_send(dtls_server_sample_conf_t *config,
    NX_PACKET *packet_ptr,
    ULONG wait_option)
{
    ...
    // Write send_len application data bytes.
    while ((ret = mbedtls_ssl_write(config->ssl_ctx, data, len)) <= 0)
    {
        if ((ret != MBEDTLS_ERR_SSL_WANT_READ)
            && (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
        {
            status = NX_NOT_SUCCESSFUL;
            break;
        }
    }
    ...
}

To reading application data is accomplished by calling mbedtls_ssl_read() function of mbedtls TLS library. In this sample, dtls_client_sample_recv() function calls it. When it is called to read application data, dtls_client_sample_recv_func() callback function will be called to get actual network data. This callback function is sets up during DTLS setup by mbedtls_ssl_set_bio() function.

This sample application calls mbedtls_ssl_read() function twice to check received data and its length. mbedtls_ssl_get_bytes_avail() function returns the number of application data bytes remaining to be read from the current record.

UINT dtls_client_sample_recv(dtls_client_sample_conf_t *config, NX_PACKET **packet_ptr,
    ULONG wait_option)
{
    ...
    // Read at most 'len' application data bytes. It's to check there is the received data or not.
    ret = mbedtls_ssl_read(config->ssl_ctx, NULL, NULL);
DA16200 Example Application Guide

```c
// Return the number of application data bytes remaining to be read from the
current record.
recv_len = mbdtls_ssl_get_bytes_avail(config->ssl_ctx);

// Allocate memory to get remaining data.
recv_buf = malloc(recv_len);

// Read at most 'len' application data bytes. It's actually to read application
data.
ret = mbedtls_ssl_read(config->ssl_ctx, recv_buf, recv_len);
...
```

2.19 DTLS Client in DPM

The DTLS client in the DPM sample application is an example of the simplest DTLS echo client application in DPM mode. Datagram Transport Layer Security (DTLS) is a cryptographic protocol that provides security for datagram-based applications by allowing them to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. The DA16200 SDK can work in DPM mode. The user application requires an additional operation to work in DPM mode. The DA16200 SDK provides the DPM manager feature for the user network application. The DPM manager feature supports the user to develop and manage a DTLS network application in Non-DPM and DPM modes. This section describes how the DTLS client in the DPM sample application is built and works.

2.19.1 How to Run

1. Run a DTLS server application on the peer PC and open a DTLS server socket with port number 10199.
2. Open the workspace for the DTLS Client in the DPM sample application as follows:
   ○ `sample\Network\DTLS_Client_DPM\build\DA16xxx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. Set the DTLS server IP address and the port number as you created the socket on the peer PC with the following console command and then reboot. These parameters can also be defined in the source code.

   ```bash
   [/DA16200] # nvram.setenv DTLSC_SERVER_IP 192.168.0.11
   [/DA16200] # nvram.setenv DTLSC_SERVER_PORT 10199
   [/DA16200] # reboot
   ○ After a connection is made to an AP, the sample application will connect to the peer PC.
   ```

2.19.2 How it Works

The DA16200 DTLS Client in the DPM sample is a simple echo message. When the DTLS server sends a message, then the DA16200 DTLS client will echo that message to the DTLS server.
2.19.3 Details

2.19.3.1 Registration

The DTLS client in the DPM sample application works in DPM mode. The basic code is similar to the DTLS client sample application. The difference with the DTLS client sample application is 2 things. First, an initial callback function is added, named `dtls_client_dpm_sample_wakeup_callback` in the code. It will be called when the DPM state changes from sleep to wake-up. The second is that an additional user configuration can be stored in RTM. In this sample, DTLS server information will be stored.

```c
void dtls_client_dpm_sample_init_user_config(dpm_user_config_t *user_config)
{
    const int session_idx = 0;

    //Set Boot init callback
    user_config->bootInitCallback = dtls_client_dpm_sample_init_callback;

    //Set DPM wakeup init callback
    user_config->wakeupInitCallback = dtls_client_dpm_sample_wakeup_callback;

    //Set Error callback
    user_config->errorCallback = dtls_client_dpm_sample_error_callback;

    //Set session type(UDP Client)
    user_config->sessionConfig[session_idx].session_type = REG_TYPE_UDP_CLIENT;

    //Set local port
    user_config->sessionConfig[session_idx].session_myPort =
        DTLS_CLIENT_DPM_SAMPLE_DEF_CLIENT_PORT;

    //Set server IP address
    memcpy(user_config->sessionConfig[session_idx].session_serverIp,
           srv_info.ip_addr, strlen(srv_info.ip_addr));

    //Set server port
    user_config->sessionConfig[session_idx].session_serverPort = srv_info.port;

    //Set Connection callback
    user_config->sessionConfig[session_idx].session_connectCallback =
        dtls_client_dpm_sample_connect_callback;

    //Set Recv callback
    user_config->sessionConfig[session_idx].session_recvCallback =
        dtls_client_dpm_sample_recv_callback;
}
```

Figure 27: Workflow of DTLS Client in DPM

- **1. DTLS Handshake**
- **2. Send Message**
- **3. Echo Message**
- **4. 5 sec**
//Set secure mode
user_config->sessionConfig[session_idx].supportSecure = NX_TRUE;

//Set secure setup callback
user_config->sessionConfig[session_idx].session_setupSecureCallback =
  dtls_client_dpm_sample_secure_callback;

//Set user configuration
user_config->ptrDataFromRetentionMemory = (UCHAR *)&srv_info;
user_config->sizeOfRetentionMemory =
  sizeof(dtls_client_dpm_sample_svr_info_t);

  return ;
}

2.19.3.2 DTLS Setup

In order to establish a DTLS session, DTLS should be setup. The DA16200 includes an mbedTLS library to provide the DTLS protocol. Most APIs that are related to the DTLS protocol are based on an mbedTLS library. DTLS will be setup by function session_setupSecureCallback. The details are as shown below. Note that this sample application does not include certificates.

void dtls_client_dpm_sample_secure_callback(void *config)
{
    const char *pers = "dtls_client_dpm_sample";
    SECURE_INFO_T *secure_config = (SECURE_INFO_T *)config;

    ret = mbedtls_ssl_config_defaults(secure_config->ssl_conf,
       MBEDTLS_SSL_IS_CLIENT,
       MBEDTLS_SSL_TRANSPORT_DATAGRAM,
       MBEDTLS_SSL_PRESET_DEFAULT);

    ret = mbedtls_ctr_drbg_seed(secure_config->ctr_drbg_ctx,
        mbedtls_entropy_func,
        secure_config->entropy_ctx,
        (const unsigned char *)pers,
        strlen(pers));

    mbedtls_ssl_conf_rng(secure_config->ssl_conf,
        mbedtls_ctr_drbg_random,
        secure_config->ctr_drbg_ctx);

    //Don't care verification in this sample.
    mbedtls_ssl_conf_authmode(secure_config->ssl_conf, MBEDTLS_SSL_VERIFY_NONE);
    mbedtls_ssl_conf_max_frag_len(secure_config->ssl_conf, 0);

    mbedtls_ssl_conf_dtls_anti_replay(secure_config->ssl_conf,
        MBEDTLS_SSL_ANTI_REPLAY_ENABLED);
    mbedtls_ssl_conf_read_timeout(secure_config->ssl_conf,
        DTLS_CLIENT_DPM_SAMPLE_RECEIVE_TIMEOUT * 10);
    mbedtls_ssl_conf_handshake_timeout(secure_config->ssl_conf,
        DTLS_CLIENT_DPM_SAMPLE_HANDSHAKE_MIN_TIMEOUT * 10,
        DTLS_CLIENT_DPM_SAMPLE_HANDSHAKE_MAX_TIMEOUT * 10);

    ret = mbedtls_ssl_setup(secure_config->ssl_ctx, secure_config->ssl_conf);
    dpm_mng_job_done(); //Done opertaion
    return ;
2.19.3.3 Data Transmission

The callback function is called when a DTLS packet is received from the DTLS server. In this sample, the received data is printed out and an echo message is sent to the DTLS server. Data will be encrypted and decrypted in the callback function.

```c
void dtls_client_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len, ULONG rx_ip, ULONG rx_port)
{
    // Display received packet
    PRINTF(" =====> Received Packet(%ld) \n", rx_len);

    // Echo message
    status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf, rx_len);

    // Display sent packet
    PRINTF(" <===== Sent Packet(%ld) \n", rx_len);

dpm_mng_job_done(); // Done operation
}
```

2.20 Connection Management

2.20.1 TCP Disconnection

If a TCP session is terminated for some unexpected reason (normally because of a TCP reset), then the TCP client needs to re-establish the connection.

Socket disconnection can be detected by the return value of `nx_tcp_socket_receive()` or `nx_tcp_socket_send()`. If the return value is `NX_NOT_CONNECTED` then do the procedure below.

1. Clear the DPM sleep status of the user thread.
2. Unbind the socket.
3. Delete the socket.
4. Clear the TCP session information from retention memory.
5. Go to create a new TCP client socket.

```c
tcp_c_normal_start:
    memset(tcp_tx_sock, 0, sizeof(NX_TCP_SOCKET));
    nx_tcp_socket_create(dpm_test_sess_tx_ip, tcp_tx_sock, ...);

    while (1) {
        status = nx_tcp_socket_receive(tcp_tx_sock, &tcp_rx_pkt, 100);
        if (status == NX_SUCCESS) {
            ...
        } else if (status == NX_NOT_CONNECTED) {
            if (chk_dpm_mode()) {
                clr_dpm_sleep(tcp_sock_name_str);
            }

            nx_tcp_client_socket_unbind(tcp_rx_sock);
            nx_tcp_socket_delete(tcp_rx_sock);

            if (chk_dpm_mode()) {
                dpm_clr_tcp_sess(tcp_tx_sock, tcp_sock_name_str);
            }
        }
    }
```
2.20.2 AP Disassociation

When the DA16200 detects a disconnection from the AP, DA16200 will process the reassociation procedure and the user application does not need to concern about reassociation.

```c
goto tcp_c_normal_start;
}
}
```

![Figure 28: AP Disconnection](image)

In normal mode, a disconnection can be detected by checking the current IP instance in the user application thread. You need to release the current socket resource, wait for the current network interface to be up (means DA16200 connected to the AP) and create a new socket as below.

```c
tx_ip = (NX_IP *) (&wlan0_ip);
tcp_tx_test_retry:
...  
while (1) {
    if ((tx_ip->nx_ip_driver_link_up == NX_FALSE) {
        clr_dpm_sleep(tcp_sock_name_str);
        nx_tcp_socket_disconnect(tcp_tx_sock, NX_IP_PERIODIC_RATE);
        nx_tcp_client_socket_unbind(tcp_tx_sock);
        nx_tcp_socket_delete(tcp_tx_sock);
        if (chk_dpm_mode()) {
            dpm_clr_tcp_sess(tcp_tx_sock, tcp_sock_name_str);
        } else {
            if (tcp_tx_sock != NULL)
                free(tcp_tx_sock);
        }
        goto tcp_tx_test_retry;
    }
    status = nx_tcp_socket_receive(tcp_tx_sock, &tcp_rx_pkt, 100);
...}
```

In DPM mode, DA16200 wakes up because there is no acknowledgement from the AP and will eventually detect disconnection. DA16200 has a predefined scenario for reassociation to the AP to reduce power consumption when DPM mode is enabled. See Figure 29. The RTC timer is scheduled as below.

- **Wakeup → Fast scan for 30 seconds → Sleep for 1 minute (repeated 3 times)**
- **Wakeup → Fast scan for 30 seconds → Sleep for 30 minutes (repeated 3 times)**
- **Wakeup → Fast scan for 30 seconds → Sleep for 1 hour (forever)**
The developer does need to implement a method to wake up DA16200 when the reassociation scenario is finished and has gone to DPM mode. The RTC wakeup pin is normally used for this situation, and the host system or a user action can trigger wakeup.

For more information on DA16200 SDK APIs for the RTC timer and sleep mode, see DA16200 DPM Programmer Guide [4].

Once DA16200 is reassociated with the AP, the user application is started and may restore the previous TCP session from retention memory.

![Flowchart](image)

**Figure 29: Scenario for AP Reassociation in DPM Mode**

### 2.21 CoAP Client

#### 2.21.1 How to Run

1. Run a CoAP server application on the peer PC.
2. Open the workspace for the CoAP Client application as follows:
   - `.\sample\Network\CoAP_Client\build\DA16xx.eww`
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
4. Use the console command to setup the Wi-Fi station interface.
5. After a connection is made to an AP, the example application will initialize a CoAP client to start the service. When the CoAP client is successfully initialized, the console message will be printed out like this:

```
Start of CoAP Client processor
```
2.21.2 CoAP Client Initialization

This section explains how to initialize and construct a CoAP client.

```c
UINT coap_client_sample_init_config(coap_client_sample_config *config) {
    coap_client_t *coap_client_ptr = &config->coap_client;
    get_thread_netx((void **)&config->pool_ptr, (void **)&config->ip_ptr,
                    WLAN0_IFACE);

    config->state = COAP_CLIENT_SAMPLE_STATE_SUSPEND;
    //Init coap client.
    status = coap_client_init(coap_client_ptr,
                              COAP_CLIENT_SAMPLE_NAME,
                              config->ip_ptr,
                              config->pool_ptr);

    //Set auth mode to skip certificate validity check.
    coaps_client_set_authmode(coap_client_ptr, NX_FALSE);
    //Set request and observe ports.
    config->req_port = COAP_CLIENT_SAMPLE_REQUEST_PORT;
    config->obs_port = COAP_CLIENT_SAMPLE_OBSERVE_PORT;

    return status;
}
```

The `coap_client_sample_init_config` function guides how the CoAP client is initialized. The `coap_client_init` function initializes the CoAP Client instance. If a CoAP observe relation is already established in DPM wakeup, it is recovered. The API's details are as follows.

- **UINT coap_client_init(coap_client_t *client_ptr, CHAR *name_ptr, NX_IP *ip_ptr, NX_PACKET_POOL *pool_ptr)**
  - **Prototype**: UINT coap_client_init(coap_client_t *client_ptr, CHAR *name_ptr, NX_IP *ip_ptr, NX_PACKET_POOL *pool_ptr)
  - **Description**: Initialize CoAP Client.
  - **Parameters**: client_ptr: CoAP Client instance pointer
                    name_ptr: Name of CoAP Client
                    ip_ptr: IP instance pointer
                    pool_ptr: Pool to allocate packet from
  - **Return values**: 0(NX_SUCCESS) on success

- **UINT coaps_client_set_authmode(coap_client_t *client_ptr, UINT mode)**
  - **Prototype**: UINT coaps_client_set_authmode(coap_client_t *client_ptr, UINT mode)
  - **Description**: If true, DTLS Server's certificate validity will be checked during DTLS handshake. Default is false.
  - **Parameters**: client_ptr: CoAP Client instance pointer
                    mode: DTLS's auth mode
  - **Return values**: 0(NX_SUCCESS) on success

2.21.3 CoAP Client Deinitialization

This section explains how to deinitialize the CoAP client.

```c
UINT coap_client_sample_deinit_config(coap_client_sample_config *config)
```
The coap_client_deinit function deinitializes the CoAP client. The API’s details are as follows.

- **UINT coap_client_deinit(coap_client_t *client_ptr)**
  - **Prototype**: UIN coap_client_deinit(coap_client_t *client_ptr)
  - **Description**: Deinitialize CoAP Client.
  - **Parameters**: client_ptr: CoAP Client instance pointer
  - **Return values**: 0(NX_SUCCESS) on success

## 2.21.4 CoAP Client Request and Response

The DA16200 provides a CoAP client request (GET/POST/PUT/DELETE/PING) and response. In this section, we describe how the DA16200 sends the CoAP request to the CoAP server and receives the CoAP response.

### 2.21.4.1 CoAP URI and Proxy-URI

To transmit a CoAP request and response, a URI must be setup. DA16200 provides APIs, such as:

- **UINT coap_client_set_uri(coap_client_t *client_ptr, unsigned char *uri, size_t urilen)**
  - **Prototype**: UINT coap_client_set_uri(coap_client_t *client_ptr, unsigned char *uri, size_t urilen)
  - **Description**: Setup URI.
  - **Parameters**: client_ptr: CoAP Client instance pointer
  - **Return values**: 0(NX_SUCCESS) on success

- **UINT coap_client_set_proxy_uri(coap_client_t *client_ptr, unsigned char *uri, size_t urilen)**
  - **Prototype**: UINT coap_client_set_proxy_uri(coap_client_t *client_ptr, unsigned char *uri, size_t urilen)
  - **Description**: Setup Proxy-URI. If uri is NULL, previous Proxy-URI is removed.
  - **Parameters**: client_ptr: CoAP Client instance pointer
  - **Return values**: 0(NX_SUCCESS) on success

### 2.21.4.2 GET Method

The DA16200 provides an API to send a GET request as shown in the example code.

```c
UINT coap_client_sample_request_get(coap_client_sample_config *config) {
    coap_client_t *coap_client_ptr = &config->coap_client;
    //Deinit coap client.
    status = coap_client_deinit(coap_client_ptr);
    //Release User's CoAP request information.
    status = coap_client_sample_clear_request(&config->request);
    return retval;
}
```
coap_rw_packet_t resp_packet;

//Set URI.
status = coap_client_set_uri(coap_client_ptr,
    (unsigned char *)request_ptr->uri, request_ptr->urilen);

//Set Proxy-URI. If null, previous proxy uri will be removed.
status = coap_client_set_proxy_uri(coap_client_ptr,
    (unsigned char *)request_ptr->proxy_uri,
    request_ptr->proxy_urilen);

//Send coap request
status = coap_client_request_get_with_port(coap_client_ptr, config->req_port);

//Receive coap response
status = coap_client_recv_response(coap_client_ptr, &resp_packet);

//Display output if response includes payload.
if (resp_packet.payload.len) {
    coap_client_sample_hexdump("GET Request", resp_packet.payload.p, resp_packet.payload.len);
}

//Release coap response.
coap_clear_rw_packet(&resp_packet);
return status;

The CoAP GET request is generated and sent in function coap_client_request_get_with_port. A CoAP response is received in function coap_client_recv_response. The API's details are as follows.

- UINT coap_client_request_get_with_port(coap_client_t *client_ptr, UINT port)

Prototype: UINT coap_client_request_get_with_port(coap_client_t *client_ptr, UINT port)
Description: CoAP Client sends GET request.
Parameters:
- client_ptr: CoAP Client instance pointer
- port: UDP socket's local port number
Return values: 0(NX_SUCCESS) on success

2.21.4.3 POST Method

DA16200 provides an API to send a POST request as shown in the example code.

UINT coap_client_sample_request_post(coap_client_sample_config *config)
{
    coap_client_t *coap_client_ptr = &config->coap_client;
    coap_client_sample_request *request_ptr = &config->request;
    coap_rw_packet_t resp_packet;

    //Set URI.
    status = coap_client_set_uri(coap_client_ptr,
        (unsigned char *)request_ptr->uri, request_ptr->urilen);

    //Set Proxy-URI. If null, previous proxy uri will be removed.
    status = coap_client_set_proxy_uri(coap_client_ptr,
        (unsigned char *)request_ptr->proxy_uri,
        request_ptr->proxy_urilen);
//Send CoAP request.
status = coap_client_request_post_with_port(coap_client_ptr, config->req_port,
request_ptr->data, request_ptr->datalen);

//Receive CoAP response.
status = coap_client_recv_response(coap_client_ptr, &resp_packet);

//Display output if response includes payload.
if (resp_packet.payload.len) {
    coap_client_sample_hexdump("POST Request", resp_packet.payload.p,
    resp_packet.payload.len);
}

//Release CoAP response.
coap_clear_rw_packet(&resp_packet);
return status;

A CoAP POST request is generated and sent in function coap_client_request_post_with_port. A
CoAP response is received in function coap_client_recv_response. The API’s details are as follows.

- **UINT coap_client_request_post_with_port(coap_client_t *client_ptr, UINT port, unsigned char *payload, unsigned int payload_len)**

**Prototype**

```
UINT coap_client_request_post_with_port(coap_client_t *client_ptr, UINT port,
unsigned char *payload, unsigned int payload_len)
```

**Description**

CoAP Client sends POST request.

**Parameters**

- client_ptr: CoAP Client instance pointer
- port: UDP socket's local port number
- payload: Payload pointer
- payload_len: Length of payload

**Return values**

0(NX_SUCCESS) on success

### 2.21.4.4 PUT Method

DA16200 provides an API to send a PUT request as shown in the example code.

```
UINT coap_client_sample_request_put(coap_client_sample_config *config)
{
    coap_client_t *coap_client_ptr = &config->coap_client;
    coap_client_sample_request *request_ptr = &config->request;
    coap_rw_packet_t resp_packet;

    //Set URI.
    status = coap_client_set_uri(coap_client_ptr,
    (unsigned char *)request_ptr->uri, request_ptr->urilen);

    //Set Proxy-URI. If null, previous proxy uri will be removed.
    status = coap_client_set_proxy_uri(coap_client_ptr,
    (unsigned char *)request_ptr->proxy_uri, request_ptr->proxy_urilen);

    //Send CoAP request.
    status = coap_client_request_put_with_port(coap_client_ptr, config->req_port,
    request_ptr->data, request_ptr->datalen);

    //Receive CoAP response.
    status = coap_client_recv_response(coap_client_ptr, &resp_packet);
}
The CoAP PUT request is generated and sent in function `coap_client_request_put_with_port`. A CoAP response is received in function `coap_client_recv_response`. The API’s details are as follows.

- **Prototype**
  
  \[
  \text{UINT coap\_client\_request\_put\_with\_port(coap\_client\_t *client\_ptr, UINT port, unsigned char *payload, unsigned int payload\_len)}
  \]

- **Description**
  
  CoAP Client sends PUT request.

- **Parameters**
  
  - client\_ptr: CoAP Client instance pointer
  - port: UDP socket’s local port number
  - payload: Payload pointer
  - payload\_len: Length of payload

- **Return values**
  
  0 (NX_SUCCESS) on success

### 2.21.4.5 DELETE Method

DA16200 provides an API to send a DELETE request as shown in the example code.

```c
//Display output if response includes payload.
if (resp_packet.payload.len) {
    coap_client_sample_hexdump("PUT Request", resp_packet.payload.p,
    resp_packet.payload.len);
}

//Release coap response.
coap_clear_rw_packet(&resp_packet);
return status;
}
```

```
The CoAP PUT request is generated and sent in function `coap_client_request_put_with_port`. A CoAP response is received in function `coap_client_recv_response`. The API’s details are as follows.

- **Prototype**
  
  \[
  \text{UINT coap\_client\_request\_put\_with\_port(coap\_client\_t *client\_ptr, UINT port, unsigned char *payload, unsigned int payload\_len)}
  \]

- **Description**
  
  CoAP Client sends PUT request.

- **Parameters**
  
  - client\_ptr: CoAP Client instance pointer
  - port: UDP socket’s local port number
  - payload: Payload pointer
  - payload\_len: Length of payload

- **Return values**
  
  0 (NX_SUCCESS) on success

```

```
2.21.4.5 DELETE Method

DA16200 provides an API to send a DELETE request as shown in the example code.

```c
UINT coap_client_sample_request_delete(coap_client_sample_config *config)
{
    coap_client_t *coap_client_ptr = &config->coap_client;
    coap_client_sample_request *request_ptr = &config->request;
    coap_rw_packet_t resp_packet;

    //Set URI.
    status = coap_client_set_uri(coap_client_ptr,
        (unsigned char *)request_ptr->uri, request_ptr->urilen);

    //Set Proxy-URI. If null, previous proxy uri will be removed.
    status = coap_client_set_proxy_uri(coap_client_ptr,
        (unsigned char *)request_ptr->proxy_uri,
        request_ptr->proxy_urilen);

    //Send coap request.
    status = coap_client_request_delete_with_port(coap_client_ptr, config->req_port);

    //Receive coap response.
    status = coap_client_recv_response(coap_client_ptr, &resp_packet);

    //Display output if response includes payload.
    if (resp_packet.payload.len) {
        coap_client_sample_hexdump("DELETE Request", resp_packet.payload.p,
        resp_packet.payload.len);
    }

    //Release coap response.
}
```
coap_clear_rw_packet(&resp_packet);
return status;
}

A CoAP DELETE request is generated and sent in function coap_client_request_delete_with_port. A CoAP response is received in function coap_client_recv_response. The API's details are as follows.

- **UINT coap_client_request_delete_with_port(coap_client_t *client_ptr, UINT port)**

  
  **Prototype**  
  UINT coap_client_request_delete_with_port(coap_client_t *client_ptr, UINT port)

  **Description**  
  CoAP Client sends DELETE request to the URI.

  **Parameters**  
  - client_ptr: CoAP Client instance pointer
  - port: UDP socket's local port number

  **Return values**  
  0(NX_SUCCESS) on success

### 2.21.4.6 CoAP PING

DA16200 provides an API to send a PING request as shown in the example code.

```c
UINT coap_client_sample_request_ping(coap_client_sample_config *config)
{
  coap_client_t *coap_client_ptr = &config->coap_client;
  coap_client_sample_request *request_ptr = &config->request;

  //Set URI.
  status = coap_client_set_uri(coap_client_ptr,
                              (unsigned char *)request_ptr->uri, request_ptr->urilen);

  //Set Proxy-URI. If null, previous proxy uri will be removed.
  status = coap_client_set_proxy_uri(coap_client_ptr,
                                    (unsigned char *)request_ptr->proxy_uri,
                                    request_ptr->proxy_urilen);

  //Progress ping request.
  status = coap_client_ping_with_port(coap_client_ptr, config->req_port);

  return status;
}
```

A CoAP PING request is processed in function coap_client_ping_with_port. The API's details are as follows.

- **UINT coap_client_ping_with_port(coap_client_t *client_ptr, UINT port)**

  
  **Prototype**  
  UINT coap_client_ping_with_port(coap_client_t *client_ptr, UINT port)

  **Description**  
  CoAP Client sends PING request.

  **Parameters**  
  - client_ptr: CoAP Client instance pointer
  - port: UDP socket's local port number

  **Return values**  
  0(NX_SUCCESS) on success
2.21.4.7 CoAP Response

DA16200 constructs a CoAP response in coap_rw_packet_t structure. In this session, we describe how CoAP response is constructed.

```c
typedef struct {
    /// Version number
    uint8_t version;
    /// Message type
    uint8_t type;
    /// Token length
    uint8_t token_len;
    /// Status code
    uint8_t code;
    /// Message-ID
    uint8_t msg_id[2];
} coap_header_t;

typedef struct {
    /// Option number
    uint8_t num;
    /// Option value
    coap_rw_buffer_t buf;
} coap_rw_option_t;

typedef struct {
    /// Header of the packet
    coap_header_t header;
    /// Token value, size as specified by header.token_len
    coap_rw_buffer_t token;
    /// Number of options
    uint8_t numopts;
    /// Options of the packet
    coap_rw_option_t opts[MAXOPT];
    /// Payload carried by the packet
    coap_rw_buffer_t payload;
} coap_rw_packet_t;
```

The coap_rw_packet_t structure includes the CoAP response information. After CoAP response is received, DA16200 parses and construct it. The coap_rw_packet_t structure is able to be released as API:

- void coap_clear_rw_packet(coap_rw_packet_t *packet)

Prototype

```
void coap_clear_rw_packet(coap_rw_packet_t *packet)
```

Description

Release coap_rw_packet structure.

Parameters

- packet: data pointer to release

Return values

0 (NX_SUCCESS) on success

To receive a CoAP response, DA16200 provides an API that is mentioned below. The API has to be called after a CoAP requests to send a response.
2.21.5 CoAP Observe

DA16200 provides a CoAP observe functionality. After registration at a CoAP server, DA16200 (CoAP client) is ready to receive an observe notification. This section describes how CoAP observe is registered and deregistered at a CoAP server.

2.21.5.1 Registration

DA16200 provides an API to register a CoAP observe as shown in the example code.

```c
UINT coap_client_sample_register_observe(coap_client_sample_config *config)
{
    FC9K_COAP_CLIENT *coap_client_ptr = &config->coap_client;
    coap_client_sample_request *request_ptr = &config->request;

    //set URI.
    status = coap_client_set_uri(coap_client_ptr,
                                 (unsigned char *)request_ptr->uri,
                                 request_ptr->urilen);

    //set Proxy-URI. If null, previous proxy uri will be removed.
    status = coap_client_set_proxy_uri(coap_client_ptr,
                                        (unsigned char *)request_ptr->proxy_uri,
                                        request_ptr->proxy_urilen);

    //Register coap observe.
    status = coap_client_set_observe_notify_with_port(
                 coap_client_ptr, config->obs_port,
                 coap_client_sample_observe_notify,
                 coap_client_sample_observe_close_notify);

    return status;
}
```

A DA16200 CoAP observe allows only 1 connection. After successful registration, a DA16200 CoAP client allows to receive an observe notification. When the observe notification is received, the callback function (observe_notify) is called. If there is no observe notification during the max-age, the close callback function (observe_close_notify) is called. The API’s details are as follows.

- **UINT coap_client_set_observe_notify_with_port(FC9K_COAP_CLIENT *client_ptr, UINT port, UINT (*observe_notify)(VOID *client_ptr, coap_rw_packet_t *resp_ptr), void (*observe_close_notify)(void))**

  **Prototype**
  
  ```c
  UINT coap_client_set_observe_notify_with_port(FC9K_COAP_CLIENT *client_ptr, UINT port, UINT (*observe_notify)(VOID *client_ptr, coap_rw_packet_t *resp_ptr), 
  void (*observe_close_notify)(void))
  ```

  **Description**
  
  Register CoAP observe. The callback function, observe_notify, will be called when CoAP observe notification is received.

  **Parameters**
  
  - client_ptr: CoAP Client instance pointer
  - port: UDP socket's local port number
  - observe_notify: Callback function for CoAP observe notification
  - observe_close_notify: Callback function for CoAP observe closing
DA16200 Example Application Guide

Return values 0(NX_SUCCESS) on success

2.21.5.2 Deregistration

DA16200 provides an API to deregister a CoAP observe as shown in the example code.

```c
UINT coap_client_sample_unregister_observe(coap_client_sample_config *config)
{
    FC9K_COAP_CLIENT *coap_client_ptr = &config->coap_client;

    //Deregister observe.
    coap_client_clear_observe(coap_client_ptr);

    return status;
}
```

The API’s details are as follows.

- VOID coap_client_clear_observe(FC9K_COAP_CLIENT *coap_client)

  Prototype VOID coap_client_clear_observe(FC9K_COAP_CLIENT *coap_client)

  Description Deregister CoAP observe relation.

  Parameters  client_ptr: CoAP Client instance pointer

  Return values  None

3 Peripheral Examples

3.1 UART

Along with a UART0 interface for the debug console, the DA16200 SDK has a UART1 interface for communication with an external MCU. GPIOA[4] and GPIOA[5] are dedicated for this interface.

3.1.1 How to Run

1. Open the workspace for the UART sample application as follows:
   - sample\Peripheral\UART1\build\DA16xxx.eww

2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

3. The start log message will be shown in the console terminal and UART1 terminal.

   - Start UART1 communicate module ...

4. To test UART1, input test data (hexa or ascii) on the UART1 terminal and press the ENTER key to send data to DA16200. Then the console terminal shows the received data in hexadecimal and sends the message “- Data receiving OK…” to UART1.

   - UART1 terminal

     ```
     Start UART1 communicate module ...
     hello
     - Data receiving OK...
     ```

   - Console terminal

     ```
     [00000000] : 68 65 6c 6c 6f
     hello
     ```

3.1.2 Application Initialization

This is an example of a user application to initialize and communicate between DA16200 and the MCU that is connected through the UART1 interface. Function user_uart1_init() initializes the UART1
H/W resource and then uart1_monitor_sample() is run to communicate with the host through the UART1 interface.

```c
static void cmd_uart_sample_cmd(int argc, char *argv[]){
    /* Initialize UART interface : Common API */
    sample_uart1 = user_uart1_init(
        UART1_BAUD,
        UART1_BITS,
        UART1_PARITY,
        UART1_STOPBIT,
        UART1_FLOW_CTRL);

    /* Start UART monitor */
    uart1_monitor_sample();
}
```

Function uart1_monitor_sample() invokes function get_data_from_uart1() repeatedly to read data from UART1. You can turn the UART echo function on or off by setting "echo_enable".

```c
void uart1_monitor_sample(ULONG arg)
{
    echo_enable = 1;
    while (1) {
        get_data_from_uart1(rx_buf);
        ...
    }
}
```

### 3.1.3 Data Read

Use UART_READ() to read one byte of data from UART1. This example shows how to read data until characters ‘\n’ or ‘\r’ are met. The user can modify this parser according to your application’s protocol.

```c
void get_data_from_uart1(UCHAR *buf)
{
    UCHAR    ch = 0;

    while (1) {
        /* Get on byte from uart1 comm port */
        UART_READ(uart1, &ch, sizeof(char));
        if (ch == NULL) {
            tx_thread_sleep(10);
            continue;
        }
        if (echo_enable) {
            UART_WRITE(uart1, &ch, sizeof(char)); // echo
        }
        /* check data length */
        if (i >= (USER_UART1_BUF_SZ - 1))
            i = USER_UART1_BUF_SZ - 2;
        if (ch == USER_DELIMITER_1 || ch == USER_DELIMITER_2) {
            buf[i++] = USER_DELIMITER_0;
            break;
        } else {
            buf[i++] = ch;
        }
    }
}
```
### 3.1.4 Data Write

This example shows how to use UART_WRITE() to send data to UART1.

```c
void put_data_to_uart1(UCHAR *buf, int tx_len)
{
    UART_WRITE(uart1, buf, tx_len);
}

void uart1_monitor_sample(ULONG arg)
{
    ...
    /* Send response to UART1 */
    tx_len = strlen((char *)tx_buf);
    put_data_to_uart1(tx_buf, tx_len);
}
```

### 3.2 GPIO

This application shows how to read/write the GPIO port and use the GPIO interrupt.

#### 3.2.1 How to Run

1. Open the workspace for the GPIO sample application as follows:
   ```
   o .\sample\Peripheral\GPIO\build\DA16xxx.eww
   ```
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. The status of GPIOA[0] and GPIOA[1] is printed every 1 second.

#### 3.2.2 Operation

1. Create and initialize a GPIO handle.
   ```c
   HANDLE gpio;
   gpio = GPIO_CREATE(GPIO_UNIT_0);
   GPIO_INIT(gpio);
   ```
2. Set pin multiplexing.
   ```
   /* AMUX to GPIOA[1:0] */
   _fc9k_io_pinmux(PIN_AMUX, AMUX_GPIO);
   /* BMUX to GPIOA[3:2] */
   _fc9k_io_pinmux(PIN_BMUX, BMUX_GPIO);
   /* CMUX to GPIOA[5:4] */
   _fc9k_io_pinmux(PIN_CMUX, CMUX_GPIO);
   ```
   ```
   /* GPIOA[0],GPIOA[4] output high low toggle */
   pin = GPIO_PIN0 | GPIO_PIN4;
   GPIO_IOCTL(gpio, GPIO_SET_OUTPUT, &pin); /* GPIOA[1] input */
   pin = GPIO_PIN1;
   GPIO_IOCTL(gpio, GPIO_SET_INPUT, &pin);
   ```
   ```
   /* GPIOA[2] interrupt active low */
   pin = GPIO_PIN2;
   GPIO_IOCTL(gpio, GPIO_SET_INTR_MODE, &ioctldata[0]);
   ```

```c
/* GPIOA[3] interrupt active high */
pin = GPIO_PIN3;
GPIO_IOCTL(gpio, GPIO_SET_INPUT, &pin);
GPIO_IOCTL(gpio, GPIO_SET_INTR_MODE, &ioctldata[0]);
ioctldata[0] |= pin; /* interrupt type 1: edge, 0: level*/
ioctldata[1] |= pin; /* interrupt pol 1: high active, 0: low active */
GPIO_IOCTL(gpio, GPIO_SET_INTR_ENABLE, &ioctldata[0]);

/* register callback function */
ioctldata[0] = pin; /* interrupt pin */
ioctldata[1] = (UINT32)gpio_callback; /* callback function */
ioctldata[2] = (UINT32)2; /* param data */
GPIO_IOCTL(gpio, GPIO_SET_CALLBACK, ioctldata);
```

6. Enable the interrupt for GPIOA[3:2]. If GPIOA[3:2] is set, then the callback function will be invoked.

```c
/* enable GPIOA[3:2] interrupt */
pin = GPIO_PIN2 | GPIO_PIN3;
GPIO_IOCTL(gpio, GPIO_SET_INTR_ENABLE, &pin);
```

7. Write GPIOA[0], GPIOA[4] and read GPIOA[1].

```c
/* GPIOA[0],GPIOA[4] to high */
write_data = GPIO_PIN0 | GPIO_PIN4;
GPIO_WRITE(gpio, GPIO_PIN0 | GPIO_PIN4, &write_data, sizeof(UINT16));

GPIO_READ(gpio, GPIO_PIN1, &read_data, sizeof(UINT16));
```

3.3 GPIO Retention

This application shows how to use GPIO retention. If the GPIO pin is set to retention high, it is kept in the high state during the sleep period. If the GPIO pin is set to retention low, it is kept in the low state during the sleep period.

3.3.1 How to Run

1. Open the workspace for the GPIO Retention sample application as follows:
   ○ `\sample\Peripheral\GPIO_Retention\build\DA16xxx.eww`
2. Build the main project, download the image to your DA16200 EVB and reboot.
3. Toggle switch 13 (SW13).
4. Use an oscilloscope to check that the GPIOA [10: 8] and GPOIC [7] keep their PIN states.
3.3.2 Operation

1. Set pin multiplexing.
   
   ```c
   /*
   * 1. Set to GPIOA[11:8], GPIOC[8:6]
   * 2. Need be written to "config_pin_mux" function.
   */
   _fc9k_io_pinmux(PIN_EMUX, EMUX_GPIO);
   _fc9k_io_pinmux(PIN_FMX, FMIX_GPIO);
   _fc9k_io_pinmux(PIN_UMUX, UMUX_GPIO);
   
   /* Set to GPIOA[9:8], GPIOC[8:6]
   */
   _fc9k_io_pinmux(PIN_EMUX, EMUX_GPIO);
   _fc9k_io_pinmux(PIN_FMX, FMIX_GPIO);
   _fc9k_io_pinmux(PIN_UMUX, UMUX_GPIO);
   
   /* Set GPIOA[9:8] to retention high */
   _GPIO_RETAIN_HIGH(GPIO_UNIT_A, GPIO_PIN8 | GPIO_PIN9);
   
   /* Set GPIOA[10] to retention low */
   _GPIO_RETAIN_LOW(GPIO_UNIT_A, GPIO_PIN10);
   
   /* Set GPIOC[7] to retention high */
   _GPIO_RETAIN_HIGH(GPIO_UNIT_C, GPIO_PIN7);
   ```

2. Set GPIO Retention Config.

   ```c
   /* Set GPIOA[9:8] to retention high */
   _GPIO_RETAIN_HIGH(GPIO_UNIT_A, GPIO_PIN8 | GPIO_PIN9);
   
   /* Set GPIOA[10] to retention low */
   _GPIO_RETAIN_LOW(GPIO_UNIT_A, GPIO_PIN10);
   
   /* Set GPIOC[7] to retention high */
   _GPIO_RETAIN_HIGH(GPIO_UNIT_C, GPIO_PIN7);
   ```


   ```c
   char * _argv[3] = {"down", "pri", "10"};
   cmd_power_down_config(3, _argv);
   /* Set GPIOA[9:8] to retention high */
   _GPIO_RETAIN_HIGH(GPIO_UNIT_A, GPIO_PIN7);
   ```

3.4 I2C

This section shows how to use the I2C interface.

3.4.1 How to Run

1. Open the workspace for the I2C.
   a. Path: .sample\Peripheral\I2C\build\DA16xxx.eww
   b. The sample application code is written in the following source file: .sample\Peripheral\I2C\src\i2c_sample.c

3.4.2 Operation

1. Hardware setup.
   a. Remove resistor R6 and R7.
   b. Connect the AT24C512 EEPROM with the DIALOG EVK.
   c. Connect each 1.2 KΩ Pull-Up resistor with GPIOA0 and GPIOA1.
      GPIOA0= SDA, GPIOA1=SCL
   d. Run I2C example code.

2. i2c init.

   ```c
   // GPIO Select for I2C working. GPIO1 = SCL, GPIO0= SDA
   Board_initialization(); FC9K_CLOCK_SCGATE->Off_DAPB_I2CM = 0;
   FC9K_CLOCK_SCGATE->Off_DAPB_APBS = 0;
   // Create Handle for I2C Device
   I2C = DRV_I2C_CREATE(i2c_0);
   
   // Initialization I2C Device
   DRV_I2C_INIT(I2C);
   ```

3. i2c addr.
// Set Address for Atmel eeprom AT24C512
addr = htoi(argv[2]);
DRV_I2C_IOCTL(I2C, I2C_SET_CHIPADDR, &addr);

4. i2c clock.
   // Set Address for Atmel eeprom AT24C512
   DRV_I2C_IOCTL(I2C, I2C_SET_CHIPADDR, &addr);

5. i2c write.
   // Data Random Write to EEPROM
   // Address = 0, Length = 4, Word Address Length = 2
   i2c_data[0] = AT_I2C_FIRST_WORD_ADDRESS; //Word Address to Write Data. 2 Bytes.
   refer at24c512 DataSheet
   i2c_data[1] = AT_I2C_SECOND_WORD_ADDRESS; //Word Address to Write Data. 2 Bytes.
   refer at24c512 DataSheet
   // Fill Ramp Data
   For (int I = 0; I < AT_I2C_DATA_LENGTH; i++)
   {
       i2c_data[i+AT_I2C_LENGTH FOR WORD ADDRESS] = i;
   }
   status = DRV_I2C_WRITE(I2C, i2c_data,
   AT_I2C_DATA_LENGTH + AT_I2C_LENGTH FOR WORD ADDRESS, 1, 0); // Handle, buffer, length, stop enable, dummy
   if (status ! = TRUE)
       PRINTF("ret : 0x%08x\r\n", status);

6. i2c read.
   // Data Random Read from EEPROM
   // Address = 0, Length = 4, Word Address Length = 2
   i2c_data_read[0] = AT_I2C_FIRST_WORD_ADDRESS; //Word Address to Write Data. 2 Bytes.
   refer at24c512 DataSheet
   i2c_data_read[1] = AT_I2C_SECOND_WORD_ADDRESS; //Word Address to Write Data. 2 Bytes.
   refer at24c512 DataSheet
   status = DRV_I2C_READ(I2C, i2c_data_read, AT_I2C_DATA_LENGTH,
   AT_I2C_LENGTH FOR WORD ADDRESS, 0); // Handle, buffer, length, address length, dummy
   if (status != TRUE)
       PRINTF("ret : 0x%08x\r\n", status);
   //Check Data
   for (int i = 0; i < AT_I2C_DATA_LENGTH; i++)
   {
       if (i2c_data_read[i] != i2c_data[i + AT_I2C_LENGTH FOR WORD ADDRESS])
           PRINTF("%dth data is different W:0x%02x, R:0x%02x\r\n", i,
                   i2c_data[i + AT_I2C_LENGTH FOR WORD ADDRESS],
                   i2c_data_read[i]);
   status = AT_I2C_ERROR_DATA_CHECK;
   }
   if (status != AT_I2C_ERROR_DATA_CHECK)
       PRINTF("***** 32 Bytes Data Write and Read Success *****\r\n");

7. i2c read_nostop.
   // Current Address Data Read from EEPROM
3.5 I2S

This section shows how to use the I2S interface.

3.5.1 How to Run

1. Open the workspace for the I2S.
   a. Path: \sample\Peripheral\I2S\build\DA16xxx.eww
   b. The sample application code is written in the following source file: \sample\Peripheral\I2S\src\i2s_sample.c

3.5.2 User Thread

The user thread of the I2S application is added as shown in the example below and will be executed by the system. SAMPLE_I2S should be a unique name to create a thread. The port number does not need to be set, because this is a non-network thread.

```c
static const app_thread_info_t sample_apps_table[] = {
    { SAMPLE_I2S, run_i2s_sample, 2048, USER_PRI_APP(1), FALSE, FALSE, UNDEF_PORT, RUN_ALL_MODE },
};
```

3.5.3 Operation

1. Create and initialize an I2S handle.
   ```c
   HANDLE gi2shandle = NULL;
   I2S_HANDLER_TYPE *i2s;
   unsigned int mode = 0, data;
   FC9K_CLOCK_SCGATE->Off_DAPB_I2S = 0;
   FC9K_CLOCK_SCGATE->Off_DAPB_APBS = 0;
   gi2shandle = DRV_I2S_CREATE(I2S_0);
   i2s = (I2S_HANDLER_TYPE *) gi2shandle;
   if (!gi2shandle)
       return;
   /* Set I2S Output Mode */
   if (DRV_I2S_INIT(gi2shandle, mode) == FALSE)
       return;
   ```

2. Set the internal DAC or the external DAC.
   ```c
   _fc9k_io_pinmux(PIN_BMUX, BMUX_I2S);
   // GPIO[1] - I2S MCLK, GPIO[0] - I2S BCLK
   _fc9k_io_pinmux(PIN_AMUX, AMUX_I2S);
   DRV_I2S_SET_CLOCK(gi2shandle, 1, 0);
   ```

3. Set additional configuration.
   ```c
   data = TRUE;
   ```
DA16200 Example Application Guide

```c
DRV_I2S_IOCTL(i2s,I2S_SET_STEREO, &data); /* Set Stereo Output Mode */
data = I2S_CFG_PCM_16;
DRV_I2S_IOCTL(i2s,I2S_SET_PCM_RESOLUTION, &data); /* Set 16bit resolution Mode */
```

4. Write and read data.

```c
DRV_I2S_WRITE (i2s, (unsigned int *)sinewave_pattern, 768, 0);
```

3.6 PWM

This section shows how to use PWM interface.

3.6.1 How to Run

1. Open the workspace for the PWM.
   a. Path: `.sample\Peripheral\PWM\build\DA16xxx.eww`
   b. The sample application code is written in the following source file.
      `.sample\Peripheral\PWM\src\pwm_sample.c`

3.6.2 Operation

1. Hardware setup.
   b. Run the PWM example command.
   c. Get waveform from P7~P9 in connector J4.
   d. Compare the waveform with the PWM setting inside example code.

2. pwm setgpio.

   ```c
   Board_Init();
   FC9K_CLOCK_SCGATE->Off_CAPB_PWM = 0;
   gpio = GPIO_CREATE(GPIO_UNIT_A);
   GPIO_INIT(gpio);
   GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNCTION_PWM_OUT0, GPIO_ALT_FUNCTION_GPIO0);
   GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNCTION_PWM_OUT1, GPIO_ALT_FUNCTION_GPIO1);
   GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNCTION_PWM_OUT2, GPIO_ALT_FUNCTION_GPIO2);
   GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNCTION_PWM_OUT3, GPIO_ALT_FUNCTION_GPIO3);
   ```

3. pwm init.

   ```c
   pwm[0] = DRV_PWM_CREATE(pwm_0);
   pwm[1] = DRV_PWM_CREATE(pwm_1);
   pwm[2] = DRV_PWM_CREATE(pwm_2);
   pwm[3] = DRV_PWM_CREATE(pwm_3);
   DRV_PWM_INIT(pwm[0]);
   DRV_PWM_INIT(pwm[1]);
   DRV_PWM_INIT(pwm[2]);
   DRV_PWM_INIT(pwm[3]);
   ```

4. pwm start_time.

   ```c
   period = 10; // 10us
   duty_percent = 30; // 30%, duration high 3us per 10us
   DRV_PWM_START(pwm[0], period, duty_percent, PWM_DRV_MODE_US); //PWM Start
   ```
   ```c
   period = 20; // 20us
   ```
duty_percent = 40; //40%, duration high 8us per 10us
DRV_PWM_START(pwm[1], period, duty_percent, PWM_DRV_MODE_US); //PWM Start

period = 40; // 40us
duty_percent = 50; //50%, duration high 20us per 10us
DRV_PWM_START(pwm[2], period, duty_percent, PWM_DRV_MODE_US); //PWM Start

period = 80; // 80us
duty_percent = 80; //80%, duration high 64us per 10us
DRV_PWM_START(pwm[3], period, duty_percent, PWM_DRV_MODE_US); //PWM Start

5. pwm start_cycle.

cycle = 2400-1; //2400 cycles(=30us @ 80MHz), cycle = value + 1
duty_cycle = 1680-1; //1680 cycles(=21us@80MHz, 70% Duty High), duty_cycle =
value + 1
DRV_PWM_START(pwm[0], cycle, duty_cycle, PWM_DRV_MODE_CYC); //PWM Start

cycle = 2400-1; //2400 cycles(=30us @ 80MHz), cycle = value + 1
duty_cycle = 1680-1; //1680 cycles(=21us@80MHz, 70% Duty High), 70% Duty High),
duty_cycle = value + 1
DRV_PWM_START(pwm[1], cycle, duty_cycle, PWM_DRV_MODE_CYC); //PWM Start

cycle = 2400-1; //2400 cycles(=30us @ 80MHz), cycle = value + 1
duty_cycle = 1680-1; //1680 cycles(=21us@80MHz, 70% Duty High), 70% Duty High),
duty_cycle = value + 1
DRV_PWM_START(pwm[2], cycle, duty_cycle, PWM_DRV_MODE_CYC); //PWM Start

cycle = 2400-1; //2400 cycles(=30us @ 80MHz), cycle = value + 1
duty_cycle = 1680-1; //1680 cycles(=21us@80MHz, 70% Duty High), 70% Duty High),
duty_cycle = value + 1
DRV_PWM_START(pwm[3], cycle, duty_cycle, PWM_DRV_MODE_CYC); //PWM Start

6. pwm stop.

DRV_PWM_STOP(pwm[0], 0);
DRV_PWM_STOP(pwm[1], 0);
DRV_PWM_STOP(pwm[2], 0);
DRV_PWM_STOP(pwm[3], 0);
3.7 ADC

This section shows how to use ADC interface.

3.7.1 How to Run

1. Open the workspace for the ADC.
   a. Path: .\sample\Peripheral\ADC\build\DA16xxx.eww.
   b. The sample application code is written in the following source file.
      .\sample\Peripheral\ADC\src\adc_sample.c

3.7.2 Operation

1. Hardware setup.
   a. Provide 0~1.3V voltage to P7 ~ P9, in connector J4.
   b. Run the ADC example command and read the ADC value.
   c. Compare the value with the voltage supplied.

2. adc init.

   // Set PAD Mux. GPIO 0 (ADC_CH0), GPIO 1 (ADC_CH1)
   _fc9k_io_pirmux(PIN_AMUX, AMUX_AD12);
   FC9K_CLOCK_SCGATE->Off_DAPB_AuxA = 0;
   FC9K_CLOCK_SCGATE->Off_DAPB_APB5 = 0;

   // Create Handle
   hadc = DRV_ADC_CREATE(FC9K_ADC_DEVICE_ID);

   // Initialization
   DRV_ADC_INIT(hadc, FC9K_ADC_NO_TIMESTAMP);

3. adc start.

   // Start. Set Sampling Frequency. 12B ADC Set to 200KHz
   DRV_ADC_START(hadc, FC9K_ADC_DIVIDER_12, 0);

4. adc enable.

   // Set ADC 0 to 12Bit ADC, ADC 1 to 12Bit ADC
   DRV_ADC_ENABLE_CHANNEL(hadc, FC9050_ADC_CH_0, FC9050_ADC_SEL_ADC_12, 0);
   DRV_ADC_ENABLE_CHANNEL(hadc, FC9050_ADC_CH_1, FC9050_ADC_SEL_ADC_12, 0);

5. adc dmaread.

   // Read 16ea ADC 0 Value. 12B ADC, Bit [15:4] is valid adc_data, [3:0] is zero
   DRV_ADC_READ_DMA(hadc, FC9050_ADC_CH_0, data12, FC9050_ADC_NUM_READ * 2,
                    FC9050_ADC_TIMEOUT_DMA, 0);

   // Read 16ea ADC 1 Value
   DRV_ADC_READ_DMA(hadc, FC9050_ADC_CH_1, data16, FC9050_ADC_NUM_READ * 2,
                    FC9050_ADC_TIMEOUT_DMA, 0);

6. adc read.

   // Read Current ADC 0 Value. Caution!! When read current adc value consequently,
   // need delay at each read function bigger than Sampling Frequency
   DRV_ADC_READ(hadc, FC9050_ADC_CH_0, &data, 0);

7. adc close.

   // Close ADC
   DRV_ADC_CLOSE(hadc);
3.8 SPI
This section shows how the SPI loopback operation works.

3.8.1 How to Run
1. Open the workspace for the SPI sample application as follows:
   ○ .\sample\Peripheral\SPI\build\DA16xxx.eww
2. Connect the SPI master pins and SPI slave pins.
   ○ GPIOA[0] (SPI_MISO) - GPIOA[9] (E_SPI_DIO1)
   ○ GPIOA[1] (SPI_MOSI) - GPIOA[8] (E_SPI_DIO0)
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
4. The SPI loopback communication works as shown in Figure 30.

![Figure 30: SPI Loopback Communication](image)

3.8.2 Operation
5. Create an SPI handle and configure the interface.

```c
spi = SPI_CREATE(SPI_UNIT_0);
if(spi != NULL ) {
    /*
     * SPI master initialization
     */
    ioctlData[0] = (1*MBYTE);
    SPI_IOCTL(spi, SPI_SET_MAX_LENGTH, ioctlData );
    _sys_clock_read( ioctldata, sizeof(UINT32) );
    SPI_IOCTL(spi, SPI_SET_CORECLOCK, ioctldata );

    ioctlData[0] = spi_clock * MHz;
    SPI_IOCTL(spi, SPI_SET_SPEED, ioctlData );

    ioctlData[0] = SPI_TYPE_MOTOROLA_00H0;
    SPI_IOCTL(spi, SPI_SET_FORMAT, ioctlData );

    ioctlData[0] = SPI_DMA_MP0_BST(8)
        | SPI_DMA_MP0_IDLE(1)
        | SPI_DMA_MP0_HSIZE(XHSIZE_DWORD)
        | SPI_DMA_MP0_AI(SPI_ADDR_INCR)
    ;
    SPI_IOCTL(spi, SPI_SET_DMA_CFG, ioctlData );

    SPI_IOCTL(spi, SPI_SET_DMAMODE, NULL);

    ioctlData[0] = spi_cs;
    ioctlData[1] = 4; // 4-wire
    SPI_IOCTL( spi, SPI_SET_WIRE, (VOID *)ioctlData);
    SPI_INIT(spi);
}
```
6. Set pin multiplexing as SPI master and SPI slave.
// pinmux config for SPI Slave - GPIOA[3:0]
_fc9k_io_pinmux(PIN_AMUX, AMUX_SPIs);
_fc9k_io_pinmux(PIN_EMUX, EMUX_SPIs);

// pinmux config for SPI Host - GPIOA[9:6]
_fc9k_io_pinmux(PIN_DMUX, DMUX_SPIm);
_fc9k_io_pinmux(PIN_EMUX, EMUX_SPIm);

7. Write data.
   tx_data[0] = (laddr >> 8) & 0xff;
   tx_data[1] = (laddr >> 0) & 0xff;
   tx_data[2] = (command & 0xff) | (common_addr_mode << 5) |
               (ref_len<<4)|((len>>8)&0xf);
   tx_data[3] = (len)&0xff;
   /*
    * copy tx data
    */
   DRIVER_MEMCPY(&(tx_data[4]), &data, sizeof(UINT32));

   // Bus Lock : CSEL0
   ioctldata[0] = TRUE;
   ioctldata[1] = OAL_SUSPEND;
   ioctldata[2] = SPI_CSEL_0;
   SPI_IOCTL(spi, SPI_SET_LOCK, (VOID *)ioctldata);

   ret = SPI_WRITE(spi, 1, tx_data, 8);

   // Bus Unlock
   ioctldata[0] = FALSE;
   ioctldata[1] = OAL_SUSPEND;
   ioctldata[2] = SPI_CSEL_0;
   SPI_IOCTL(spi, SPI_SET_LOCK, (VOID *)ioctldata);

8. Read data.
   tx_data[0] = (laddr >> 8) & 0xff;
   tx_data[1] = (laddr >> 0) & 0xff;
   tx_data[2] = command | (common_addr_mode << 5) | (ref_len<<4)|((len>>8)&0xf);
   tx_data[3] = (len)&0xff;
   // Bus Lock : CSEL0
   ioctldata[0] = TRUE;
   ioctldata[1] = OAL_SUSPEND;
   ioctldata[2] = SPI_CSEL_0;
   SPI_IOCTL(spi, SPI_SET_LOCK, (VOID *)ioctldata);

   ret = SPI_WRITE_READ(spi, 1, tx_data, 4, rx_data, len);

   // Bus Unlock
   ioctldata[0] = FALSE;
   ioctldata[1] = OAL_SUSPEND;
   ioctldata[2] = SPI_CSEL_0;
   SPI_IOCTL(spi, SPI_SET_LOCK, (VOID *)ioctldata);
3.9 SDIO

The DA16200 can be accessed with the SDIO interface. If the user wants to test it, then another host system is needed.

### 3.9.1 How to Run

1. Open the workspace for the SDIO slave.
   a. Path: .\sample\Peripheral\SDIO\build\DA16xxx.eww.
   b. The sample application code is written in the following source file.
      .\sample\Peripheral\SDIO\src\sdio_sample.c
2. GPIOA[9:4] needs to connect to the HOST system.
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
4. The sample runs as soon as the boot up is completed.

```c
// clock enable sdio_slave
FC9K_CLOCK_SCGATE->Off_SSI_M3X1 = 0;
FC9K_CLOCK_SCGATE->Off_SSI_SDIO = 0;
SDIO_SLAVE_INIT();
sdio_slave_buf = HAL_MALLOC(1024);
PRINTF("sdio_slave init buf %x\n", sdio_slave_buf);
PRINTF("Now the host can access the DA16200 by SDIO\n");
```

5. Now the DA16200 is ready to receive an SDIO command.

### 3.9.2 Operation

In DA16200, the loopback test between SD host and sdio_slave is not supported, instead, in the sample code provided, SDIO is just waiting for a request from the host after initialization.

```c
/* GPIO configuration */
{
    /*
    * SDIO Slave
    */
    _fc9k_io_pinmux(PIN_EMUX, EMUX_SDs);
    _fc9k_io_pinmux(PIN_CMUX, CMUX_SDs);
    _fc9k_io_pinmux(PIN_DMUX, DMUX_SDs);
}
```
DA16200 Example Application Guide

3.10 SD/eMMC

This section shows how to use the SD/eMMC interface.

3.10.1 How to Run

1. Open the workspace for the SD_EMMC.
   a. Path: .\sample\Peripheral\SD_EMMC\build\DA16xxx.eww.
   b. The sample application code is written in the following source file:
      .\sample\Peripheral\SD_EMMC\src\sd_emmc_sample.c

2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

3. The sample runs as soon as the boot up is completed.

4. If the SD card is not ready, then message "emmc_init fail" is returned.

5. Connect GPIOA[9:4] to the SD card socket as shown below.

![SDIO & SD/eMMC Connector](image)

Figure 31: SDIO & SD/eMMC Connector

- GPIOA[9] - mSDeMMCMIO_D0, GPIOA[8] - mSDeMMCMIO_D1
  a. GPIOA[10] is not mandatory (for write protect function).

3.10.2 Operation

This sample code shows how the eMMC host writes random data to a slave memory card and reads back the written data to check if that data matches.

Function Emmc_verify() compares the written data with the data read from the SD memory card. The sector size of the SD memory card is 512 bytes. The "addr" variable value (210) in the code is just an example sector number in the SD memory card.

```c
void emmc_init() {
    // emmc sample start
    fail / total 0 / 100
    //emmc sample finish
```
... FC9K_CLOCK_SCGATE->Off_Sys_CIF = 0;
FC9050_SYSCLOCK->CLK_DIV_EMMC = EMMC_CLK_DIV_VAL;
FC9050_SYSCLOCK->CLK_EN_SDeMMC = 0x01; // clock enable
...

1. Set pin multiplexing.
   /*
    * SDIO Master
   */
   // GPIO[9] - mSDeMMCIO_D0, GPIO[8] - mSDeMMCIO_D1
   _fc9k_io_pinmux(PIN_EMUX, EMUX_SDm);
   _fc9k_io_pinmux(PIN_CMUX, CMUX_SDm);
   _fc9k_io_pinmux(PIN_DMUX, DMUX_SDm);

2. Create and initialize an SD/eMMC handle.
   _emmc = EMMC_CREATE();
   err = EMMC_INIT(_emmc);
4 Advanced Examples

4.1 DNS Query

4.1.1 How to Run

1. Open the workspace for DNS Query sample application as follows:
   - \sample\Network\DNS_Query\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console to set up the Wi-Fi station interface.
4. After a connection is made to an AP, the example application starts a DNS query operation with a test URL.
5. The example application runs two types of DNS query operations: single-IPv4 address and multiple-IPv4 address.

   >>> Single IPv4 address DNS query test ...
   - Name : www.daum.net
   - Addresses : 203.133.167.16

   >>> Multiple IPv4 address DNS query test ...
   - Name : www.daum.net
   - Addresses : 203.133.167.81
   203.133.167.16

4.1.2 Application Initialization

This section shows how to get the IPv4 address from a domain name URL. Two types of API functions are supported to get the IP address.

- Get single IPv4 address: char * dns_A_query(char *domain_name)
- Get multiple IPv4 addresses: unsigned int dns_ALL_Query(unsigned char *domain_name,
  unsigned char *record_buffer,
  unsigned int buffer_size,
  unsigned int *record_count);

This example creates a user thread, which entry function is dns_query_sample().

```c
void dns_query_sample(ULONG arg)
{
    char    *test_url = TX_NULL;
    /* Check test url */
    test_url = read_nvram_string("TEST_DOMAIN_URL");
    if (test_url == TX_NULL) {
        test_url = TEST_URL;
    }
    PRINTF("\n\n");
    /* 1. Single IP address */
    dns_A_query_sample(test_url);
    /* 2. Multiple IP address */
    dns_multiple_query_sample(test_url);
}
```
4.1.3  Get Single IPv4 Address

This example shows the use of the API function "char * dns_A_query(char *domain_name)" to get the IPv4 address string with a domain name URL.

```c
void dns_A_query_sample(char *test_url_str)
{
    char    *ipaddr_str = NX_NULL;

    PRINTF(">>> Single IPv4 address DNS query test ...
");

    /* Allocate buffer from heap area */
    ipaddr_str = malloc(MAX_IP_LEN);
    if (ipaddr_str == NULL) {
        PRINTF("\nFailed to allocate buffer from heap area ...
");
        return;
    }
    memset(ipaddr_str, 0, MAX_IP_LEN);
    /* DNS query with test url string */
    ipaddr_str = dns_A_Query(test_url_str);

    /* Fail checking ... */
    if (ipaddr_str == NX_NULL) {
        PRINTF("\nFailed to dns-query with %s\n", test_url_str);
    } else {
        PRINTF("- Name : %s\n", test_url_str);
        PRINTF("- Addresses :\t%s\n", ipaddr_str);
    }
}
```
### 4.1.4 Get Multiple IPv4 Addresses

This example shows the use of the API function `UINT dns_ALL_Query(UCHAR *domain_name, UCHAR *record_buffer, UINT record_buffer_size, UINT *record_count)` to get a string of multiple IPv4 addresses from a domain name URL.

```c
/* 2. Multiple IP address */
void dns_multiple_query_sample(char *test_url_str)
{
    ULONG multi_ipaddr[MAX_IP_LIST_CNT] = { 0, };
    ULONG ipv4_address;
    int ipaddr_cnt = 0;
    int i;
    UINT status;

    PRINTF("\n\n"n
        "n"
    PRINTF(">>> Multiple IPv4 address DNS query test ...
    /* DNS query with test url string */
    status = dns_ALL_Query((UCHAR *)test_url_str, 
                           (UCHAR *)&multi_ipaddr[0], 
                           (MAX_IP_LIST_CNT * sizeof(ULONG)), 
                           (UINT *)&ipaddr_cnt);

    /* Fail checking ... */
    if (status != TX_SUCCESS) {
        PRINTF("\nFailed to dns-query with \%s\n", test_url_str);
    } else {
        PRINTF("- Name : %s\n", test_url_str);
        PRINTF("- Addresses : ");

        for (i = 0; i < ipaddr_cnt; i++) {
            ipv4_address = multi_ipaddr[i];

            PRINTF("\t%02x.%02x.%02x.%02x\n", 
                    (ipv4_address >> 24) & 0xff, 
                    (ipv4_address >> 16) & 0xff, 
                    (ipv4_address >>  8) & 0xff, 
                    (ipv4_address      ) & 0xff);
        }
    }
}
```
4.2 SNTP and Get Current Time

Wi-Fi devices may need to synchronize the device clock on the internet with the use of TLS or communication with the server. DA16200 provides SNTP for this operation and users can use this function to get the current time.

4.2.1 How to Run

1. Open the workspace for the SNTP and current time sample application as follows:
   ○ \sample\ETC\Current_Time\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console to set up the Wi-Fi station interface.
4. After a connection is made to an AP, the example application will start an SNTP client with test values.

```c
#define TEST_SNTP_SERVER "time.windows.com"
#define TEST_SNTP_RENEW_PERIOD 600
#define TEST_TIME_ZONE (9 * 3600) // seconds
#define ONE_SECONDS 100
#define CUR_TIME_LOOP_DELAY 10 // seconds
```

Note 1 If SNTP client was started with pre-defined values, this configuration will be ignored.

Note 2 The legacy AP must be connected to the internet.

After a connection is made to the SNTP server, DA16200 will show the connection result on the debug console.

```
Connection COMPLETE to 88:36:6c:4e:a1:28
-- DHCP Client WLAN0: SEL
-- DHCP Client WLAN0: REQ
-- DHCP Client WLAN0: BOUND
   Assigned addr : 192.168.0.9
   netmask : 255.255.255.0
   gateway : 192.168.0.1
   DNS addr : 168.126.63.1

   DHCP Server IP : 192.168.0.1
   Lease Time : 02h 00m 00s
   Renewal Time : 01h 00m 00s

>>> SNTP Server: time.windows.com (52.168.138.145)
```

DA16200 will periodically get the current time (test period: 10 seconds).

```
- Current Time : 2018.12.03 15:06:37 (GMT +9:00)
- Current Time : 2018.12.03 15:06:47 (GMT +9:00)
- Current Time : 2018.12.03 15:06:57 (GMT +9:00)
```
4.2.2 Operation

1. The user application needs to set SNPT client information.

   void cur_time_sample(ULONG arg)
   {
       unsigned int status;
       __time64_t now;
       struct tm *ts;
       char time_buf[80];

       /* Config SNTP client */
       status = set_n_start_SNTP();
       if (status != TX_SUCCESS) {
           PRINTF("[%s] Fail to start SNTP client ...\n", __func__);
           return;
       }

2. If the SNTP client was already started with pre-defined values, then this configuration will be skipped. Set the SNTP server address, time update period and time zone and finally enable the function.

   static UCHAR set_n_start_SNTP(void)
   {
       unsigned int status = TX_SUCCESS;

       /* Config and save SNTP server domain in FC9050 */
       status = (unsigned int)setSNTPsrv(TEST_SNTP_SERVER);
       if (status != TX_SUCCESS) {
           PRINTF("[%s] Failed to write nvram operation (SNTP server domain)...\n", __func__);
           status = TX_START_ERROR;
           goto _exit;
       }

       /* Config and save SNTP periodic renew time: seconds */
       status = (unsigned int)setSNTPperiod(TEST_SNTP_RENEW_PERIOD);
       if (status != TX_SUCCESS) {
           PRINTF("[%s] Failed to write nvram operation (SNTP renew period)...\n", __func__);
           status = TX_START_ERROR;
           goto _exit;
       }

       /* Config and save SNTP time zone */
       status = (unsigned int)setTimezone(TEST_TIME_ZONE);
       if (status != TX_SUCCESS) {
           PRINTF("[%s] Failed to write nvram operation (SNTP renew period)...\n", __func__);
           status = TX_START_ERROR;
           goto _exit;
       }
       fc9k_SetTzoff(TEST_TIME_ZONE);
DA16200 Example Application Guide

/* Config and save SNTP client mode : enable */
status = setSNTPuse(SNTP_ENABLE);
if (status != TX_SUCCESS) {
    PRINTF("[%s] Failed to write nvram operation (SNTP mode)...\n", __func__);
    status = TX_START_ERROR;
    goto _exit;
}

3. After a connection is made to the SNTP server, DA16200 will periodically get the current time.

void cur_time_sample(ULONG arg)
{
    ...  
    /* delay */
    tx_thread_sleep(CUR_TIME_LOOP_DELAY * ONE_SECONDS);

    /* get current time */
    fc9k_time64(NULL, &now);
    ts = (struct tm *)fc9k localtime64(&now);

    /* make time string */
    fc9kstrftime(time_buf, sizeof(time_buf), "%Y.%m.%d %H:%M:%S", ts);

    /* display current time string */
    PRINTF("- Current Time : %s (GMT %+02ld:%02ld)\n", 
            time_buf, 
            fc9k_Tzoff() / 3600, 
            fc9k_Tzoff() % 3600);
}

4.3 SNTP and Get Current Time in DPM Function

This example application applies to the DPM function. Most code is the same as the non-DPM SNTP example.

4.3.1 How to Run

1. Open the workspace for the SNTP and current time in the DPM sample application as follows:
   ○ \sample\ETC\Current_Time\DPM\build\DA16xxx.eww

2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

3. Use the console to set up the Wi-Fi station interface.

4. After a connection is made to an AP, the example application will start an SNTP client with test values.

Note 1  If the SNTP client was started with pre-defined values, then this configuration will be ignored.

Note 2  The legacy AP must be connected to the internet.

After connection is made to the SNTP server, DA16200 will show the connection result on the debug console and go to DPM sleep mode.

Connection COMPLETE to 88:36:6c:4e:a1:28
--- DHCP Client WLAN0: SEL
--- DHCP Client WLAN0: REQ
--- DHCP Client WLAN0: BOUND

Assigned addr   : 192.168.0.9
netmask   : 255.255.255.0
gateway   : 192.168.0.1
DNS addr   : 169.126.63.1

DHCP Server IP  : 192.168.0.1
 Lease Time     : 02h 00m 00s
 Renewal Time   : 01h 00m 00s

>>> SNTP Server: time.windows.com (52.168.138.145)

>>> Start DPM Power-Down !!!

DA16200 will periodically get the current time (test period is 10 seconds).

WC_RTM(0x12)
>>> TIM : FAST
- Current Time : 2018.12.03 15:06:38 (GMT +9:00)
>>> Start DPM Power-Down !!!
rwnx_send_set_ps_mode PS TIME (us) 115902

WC_RTM(0x12)
>>> TIM : FAST
- Current Time : 2018.12.03 15:06:48 (GMT +9:00)
>>> Start DPM Power-Down !!!
rwnx_send_set_ps_mode PS TIME (us) 115936

4.3.2 Operation

The SNTP configuration interface is the same as the non-DPM SNTP example. If DA16200 woke up from DPM sleep mode, use the RTM API to get the current SNTP status, or save the SNTP status
into the RTM.

```
static UCHAR set_n_start_SNTP(void)
{
    unsigned int status = TX_SUCCESS;

    /* Check current SNTP running status */
    if (dpm_mode_is_wakeup() == DPM_WAKEUP) {
        status = get_sntp_use_from_rtm();
    } else {
        status = getSNTPuse();
    }

    if (status == TX_TRUE) {
        long time_zone;
        /* Already SNTP module running, set again time-zone ... */
        time_zone = get_timezone_from_rtm();
        fc9k_SetTzoff(time_zone);
        return TX_SUCCESS;
    }

    if (dpm_mode_is_wakeup() == NORMAL_BOOT) {
        /* Config and save SNTP server domain in FC9050 */
```
status = (unsigned int)setSNTPsrv(TEST_SNTP_SERVER);
if (status != TX_SUCCESS) {
  PRINTF("[%s] Failed to write nvram operation (SNTP server domain)...\n", __func__);
  status = TX_START_ERROR;
  goto _exit;
}

/* Config and save SNTP periodic renew time : seconds */
status = (unsigned int)setSNTPperiod(TEST_SNTP_RENEW_PERIOD);
if (status != TX_SUCCESS) {
  PRINTF("[%s] Failed to write nvram operation (SNTP renew period)...\n", __func__);
  status = TX_START_ERROR;
  goto _exit;
}

/* Config and save SNTP time zone */
status = (unsigned int)setTimezone(TEST_TIME_ZONE);
if (status != TX_SUCCESS) {
  PRINTF("[%s] Failed to write nvram operation (SNTP renew period)...\n", __func__);
  status = TX_START_ERROR;
  goto _exit;
}

set_timezone_to_rtm(TEST_TIME_ZONE);
fC9k_SetTzoff(TEST_TIME_ZONE);

/* Config and save SNTP client mode : enable */
status = setSNTPuse(SNTP_ENABLE);
if (status != TX_SUCCESS) {
  PRINTF("[%s] Failed to write nvram operation (SNTP mode)...\n", __func__);
  status = TX_START_ERROR;
  goto _exit;
}

/* Save config and start SNTP client */
set_sntp_use_to_rtm(status);

_exit :
return status;
When connected to the SNTP server, DA16200 will start an RTC timer to periodically get the current time. See "DA16200 DPM programmer guide"[4] for information about DPM and RTC API.

```c
void cur_time_dpm_sample(ULONG arg)
{
    //... /* Regist periodic RTC Timer: Get current time */
    if (dpm_mode_is_wakeup() == NORMAL_BOOT) {
        /* Time delay for stable running SNTP client */
        tx_thread_sleep(10);

        dpm_timer_create(CUR_TIME_LOOP_DELAY,
                         SAMPLE_CUR_TIME_DPM,
                         TEST_TIMER_ID,
                         1, // 0:One-shot, 1:Periodical
                         display_cur_time);
    }

    /* Set flag to go to DPM sleep 3 */
    dpm_app_sleep_ready_set(SAMPLE_CUR_TIME_DPM);
}
```

The SNTP configuration interface is the same as for the non-DPM SNTP example.

```c
static void display_cur_time(void)
{
    dpm_app_wakeup_done(SAMPLE_CUR_TIME_DPM);

    __time64_t now;
    struct tm *ts;
    char time_buf[80];

    /* get current time */
    fc9k_time64(NULL, &now);
    ts = (struct tm *)fc9k_localtime64(&now);

    /* make time string */
    fc9k_strftime(time_buf, sizeof(time_buf), "%Y.%m.%d %H:%M:%S", ts);

    /* display current time string */
    PRINTF("- Current Time : %s (GMT %+02ld:%02ld)",
            time_buf,
            fc9k_Tzoff() / 3600,
            fc9k_Tzoff() % 3600);

    /* Set flag to go to DPM sleep 3 */
    dpm_app_sleep_ready_set(SAMPLE_CUR_TIME_DPM);
}
```
4.4 HTTP Client

The DA16200 SDK has a ported product called Express Logic’s NetXDuo HTTP v5.10. With this product an application programmer can develop an http client application that use NetXDuo HTTP APIs.

4.4.1 How to Run

1. Open the workspace for the HTTP_Client sample application as follows:
   - \sample\Network\HTTP_Client\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console to set the Wi-Fi station interface and connect to the AP that is connected to the Internet.
4. Complete the setup and (re)start the sample.

4.4.2 Operation

The sample code is an example of the -get method. When the sample starts, a “-get” request is made to the URL address (the sample’s URL address is just an example).

To connect to the https server, simply enter “https://” instead of “http://” in the URL address.

To set valid time information in the certificate before the https request, the system’s current time must be set (SNTP service must be enabled).

1. The HTTP Client sample code also includes sample code to parse the input URL.
   If the URL is parsed with https, the encryption mode option is automatically set.

   ```c
   UINT http_client_get_sample(unsigned char *uri)
   {
      status = parse_uri(uri, uri_len, &http_request);
      if (status)
      {
         DBG_PRINT("[%s]Failed to parse uri(0x%02x)\n", __func__, status);
         return status;
      }
   }
   
   2. Create an HTTP Client instance on the specified IP instance.

      // Create http client
      status = nx_http_client_create(&http_client,
         HTTP_CLIENT_THREAD_NAME,
         ip_ptr,
         pool_ptr,
         HTTP_CLIENT_WINDOW_SZ);
      
      if (status)
      {
         DBG_PRINT("[%s]Not able to create http client(0x%02x).\n", __func__, status);
         return status;
      }
   
   3. Set the port number and insert the header field.

      // Set http client options
      nx_http_client_set_connect_port(&http_client, http_request.port);  
      
      // Set hostname for http 1.1
      nx_http_client_set_host_domain(&http_client,
         http_request.hostname,
         strlen((char *)http_request.hostname));
   ```
4. Set the buffer size that https uses for encryption and decryption. The buffer size can be up to 17 Kbyte. If not set, the default is 4 Kbyte. If only http is used, then the buffer size does not need to be set.

    if (http_request.insecure == NX_TRUE)
    {
        // Set secure mode
        nx_http_client_set_secure_connection(&http_client, NX_TRUE);

        // Use heap memory for tls contents buffer
        nx_http_client_set_content_heap(&http_client, NX_TRUE);

        // Set contents buffer's size
        nx_http_client_set_content_buflen(&http_client,
                                         HTTP_CLIENT_IN_CONTENT_BUF_SZ,
                                         HTTP_CLIENT_IN_CONTENT_BUF_SZ);
    }

5. Start an HTTP GET request.
If this routine returns NX_SUCCESS, the application can then make multiple calls to nx_http_client_get_packet to retrieve data packets that correspond to the requested resource content.

    status = nxd_http_client_get_start(&http_client,
                                       &http_request.ip_addr,
                                       http_request.path,
                                       NX_NULL,
                                       0,
                                       NX_NULL,
                                       NX_NULL,
                                       wait_option);
    
    if (status != NX_SUCCESS)
    {
        DBG_PRINT("[%s]Not able to get data from http server(0x%02x)\n", __func__, status);
    }
    else
    {
        ...

6. Get the next resource data packet.
This step retrieves the next content packet of the resource requested by the previous nx_http_client_get_start call. Successive calls to this routine should be made until the return status NX_HTTP_GET_DONE is received.
Function nx_packet_data_retrieve copies data from the supplied packet into the supplied buffer. The actual number of bytes copied is returned in the destination that is pointed to by the bytes copied.

    do
    { nx_packet_release(recv_packet);

        if (recv_buf)
        {
            free(recv_buf);
        }
recv_buf = NULL;
}
recv_bytes = 0;

status = nx_http_client_get_packet(&http_client,
    &recv_packet,
    wait_option);
if (status == NX_SUCCESS)
{
    status = nx_packet_length_get(recv_packet, &recv_bytes);
    if (status)
    {
        PRINTF("[%s]Failed to get rx packet's length(0x%02x)\n", __func__, status);
        break;
    }
    
    recv_buf = calloc(recv_bytes + 1, 1);
    if (recv_buf == NULL)
    {
        PRINTF("[%s]Memory is not enough\n", __func__);
        break;
    }
    
    status = nx_packet_data_retrieve(recv_packet, recv_buf, &recv_bytes);
    if (status != NX_SUCCESS)
    {
        PRINTF("[%s] Not able to get data(0x%02x)\n", __func__, status);
        break;
    }

    nx_packet_release(recv_packet);
    recv_packet = NX_NULL;
    free(recv_buf);
    recv_buf = NULL;
}
4.5 HTTP Client in DPM Function

The DA16200 SDK has a ported product called Express Logic’s NetXDuo HTTP v5.10. With this product an application programmer can develop an http client application that use NetXDuo HTTP APIs.

4.5.1 How to Run

1. Open the workspace for the HTTP_Client sample application as follows:
   - Open the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
   - Use the console to set the Wi-Fi station interface and connect to the AP that is connected to the Internet.

2. Complete the setup and (re)start the sample.

4.5.2 Operation

The sample code is an example of the -get method. When the sample starts, a "-get" request is made to the URL address (the sample’s URL address is just an example).

To connect to the https server, simply enter "https://" instead of "http://" in the URL address.

To set valid time information in the certificate before the https request, the system’s current time must be set (SNTP service must be enabled).

1. If an application using the Http protocol is registered in DPM, it should be set not to enter DPM_SLEEP during http transmission (request / response). Set DPM_SLEEP enabled after all transfers are complete.

```
UINT port_number = 0;
PRINTF("\n\n\n>>> Start HTTP-Client sample\n\n\n");
dpm_app_register(HTTP_CLIENT_THREAD_NAME, port_number);
dpm_app_sleep_ready_clear(HTTP_CLIENT_THREAD_NAME);
http_client_get_sample(uri);
dpm_app_sleep_ready_set(HTTP_CLIENT_THREAD_NAME);
```

2. The HTTP Client sample code also includes sample code to parse the input URL. If the URL is parsed with https, the encryption mode option is automatically set.

```
UINT http_client_get_sample(unsigned char *uri)
{
    status = parse_uri(uri, uri_len, &http_request);
    if (status)
    {
        DBG_PRINT("[%s]Failed to parse uri(0x%02x)\n", __func__, status);
        return status;
    }
}
```

3. Create an HTTP Client instance on the specified IP instance.

```
// Create http client
status = nx_http_client_create(http_client,
    HTTP_CLIENT_THREAD_NAME,
    ip_ptr,
    pool_ptr,
```

HTTP_CLIENT_WINDOW_SZ);

if (status)
{
    DBG_PRINT("[\%s]Not able to create http client(0x\%02x).\n", __func__, status);
    return status;
}

4. Set the port number and insert the header field.
   // Set http client options
   nx_http_client_set_connect_port(&http_client, http_request.port);

   // Set hostname for http 1.1
   nx_http_client_set_host_domain(&http_client,
       http_request.hostname,
       strlen((char *)http_request.hostname));

5. Set the buffer size that https uses for encryption and decryption.
   The buffer size can be up to 17 Kbyte. If not set, the default is 4 Kbyte. If only http is used, then
   the buffer size does not need to be set.

   if (http_request.insecure == NX_TRUE)
   {
       // Set secure mode
       nx_http_client_set_secure_connection(&http_client, NX_TRUE);

       // Use heap memory for tls contents buffer
       nx_http_client_set_content_heap(&http_client, NX_TRUE);

       // Set contents buffer's size
       nx_http_client_set_content_buflen(&http_client,
           HTTP_CLIENT_IN_CONTENT_BUF_SIZE,
           HTTP_CLIENT_IN_CONTENT_BUF_SIZE);
   }

6. Start an HTTP GET request.
   If this routine returns NX_SUCCESS, the application can then make multiple calls to
   nx_http_client_get_packet to retrieve data packets that correspond to the requested
   resource content.

   status = nxd_http_client_get_start(&http_client,
       http_request.ip_addr,
       http_request.path,
       NX_NULL,
       0,
       NX_NULL,
       NX_NULL,
       wait_option);

   if (status != NX_SUCCESS)
   {
       DBG_PRINT("[\%s]Not able to get data from http server(0x\%02x)\n", __func__, status);
   }
   else
   {
       ...

7. Get the next resource data packet.
This step retrieves the next content packet of the resource requested by the previous
nx_http_client_get_start call. Successive calls to this routine should be made until the
return status NX_HTTP_GET_DONE is received.

Function nx_packet_data_retrieve copies data from the supplied packet into the supplied
buffer. The actual number of bytes copied is returned in the destination that is pointed to by the
bytes copied.

```c
do {
    nx_packet_release(recv_packet);
    if (recv_buf)
        {
            free(recv_buf);
            recv_buf = NULL;
        }
    recv_bytes = 0;
    status = nx_http_client_get_packet(&http_client,
                                        &recv_packet,
                                        wait_option);
    if (status == NX_SUCCESS)
    {
        status = nx_packet_length_get(recv_packet, &recv_bytes);
        if (status)
            {
                PRINTF("[%s]Failed to get rx packet's length(0x%02x)\n",
                        __func__, status);
                break;
            }
        recv_buf = calloc(recv_bytes + 1, 1);
        if (recv_buf == NULL)
            {
                PRINTF("[%s]Memory is not enough\n", __func__);
                break;
            }
        status = nx_packet_data_retrieve(recv_packet, recv_buf, &recv_bytes);
        if (status != NX_SUCCESS)
            {
                PRINTF("[%s] Not able to get data(0x%02x)\n",
                        __func__, status);
                break;
            }
        nx_packet_release(recv_packet);
        recv_packet = NX_NULL;
        free(recv_buf);
        recv_buf = NULL;
    }
```
DA16200 Example Application Guide

4.6 HTTP Server

The DA16200 SDK has a ported product called Express Logic’s NetXDuo HTTP v5.10. With this product an application programmer can develop an http server application that uses NetXDuo HTTP APIs.

4.6.1 How to Run

1. Open the workspace for the HTTP_Server sample application as follows:
   - \sample\Network\HTTP_Server\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. Use the console to set the Wi-Fi station interface and connect to the AP.
4. Complete the setup and (re)start the sample.

4.6.2 Operation

The sample code is an example of the -get and -post methods.

1. The HTTP Server sample code supports both HTTP and HTTPS (Default is HTTP).
   To operate with HTTPS, define ENABLE_HTTPS_SERVER as shown below. Also, update the certificate embedded in the code(*root_ca, *own_cert, *private_key) as needed.

   /// HTTPS server
   #define ENABLE_HTTPS_SERVER

2. Check if the network is initialized and check the status of the IP instance.

   static UINT http_server_sample(HTTP_SERVER_CONF *config)
   {
   UINT status = NX_SUCCESS;
   NX_PACKET_POOL *svr_pkt_pool = NX_NULL;
   NX_IP *nx_ip_ptr = NX_NULL;

   /* Check the network initialization */
   status = check_net_init(config->iface);
   if (status == NX_SUCCESS)
   {
   ULONG actual_status;

   get_thread_netx((void **)&svr_pkt_pool, (void **)&nx_ip_ptr, config->iface);
   do
   {
   status = nx_ip_status_check(nx_ip_ptr,
   NX_IP_INITIALIZE_DONE, &actual_status, 100);
   if (status != NX_SUCCESS)
   {
   tx_thread_sleep(1);
   }
   } while (status != NX_SUCCESS);
   }
   else
   {
   PRINTF("Failed to create HTTP Server(0x%02x)\n", status);
   return status;
   }

3. This step creates an HTTP Server instance, which runs in the context of its own ThreadX thread. The optional authentication_check and request_notify application callback routines give the application software control over the basic operations of the HTTP Server.
4. Start the create HTTP Server instance.

    // start http server
    status = nx_http_server_start(&config->http_server);
    if (status != NX_SUCCESS)
    {
        PRINTF("Failed to start HTTP server(0x%02x)\n", status);
        APP_FREE(config->stack);
        config->stack = NULL;
        return status;
    }

    APP_FREE(config->stack);
    config->stack = NULL;
    return status;

5. When a method request such as "-get" and "-post" is received from the HTTP Client, the notify callback function registered at nx_http_server_create is called.

    static UINT http_server_request_notify(NX_HTTP_SERVER *server_ptr,
                 UINT logical_connection,
                 UINT request_type,
                 CHAR *resource,
                 NX_PACKET *packet_ptr)
    {
        UINT status = NX_SUCCESS;
        char *recv_buf = NULL;
        ULONG recv_bytes = 0;

        if (request_type == NX_HTTP_SERVER_GET_REQUEST)
        {
            PRINTF("HTTP Server : GET\n");
        }

        else if (request_type == NX_HTTP_SERVER_POST_REQUEST)
        {
            PRINTF("HTTP Server : POST\n");
6. Body data to respond to the HTTP Client is built by the functions registered in `http_server_init_info`.

   ```c
   static VOID http_server_init_info(HTTP_SERVER_CONF *config, UINT secure_mode)
   {
     ...
     //Functions for building response data
     config->http_server_params.nx_http_server_web_open = ws_open;
     config->http_server_params.nx_http_server_web_close = ws_close;
     config->http_server_params.nx_http_server_web_get_size = ws_get_size;
     config->http_server_params.nx_http_server_web_get_payload = ws_get_payload;
     
     return ;
   }
   ```

7. If the HTTP Server works successfully, test the "-get" method as follows. Use the web browser of the test PC that is connected to the same network.

   ```c
   http://[Server IP]/index.html
   ```

8. Next, test the "-post" method. The POSTMAN tool is recommended to do the test (https://www.getpostman.com/apps).

   ![POSTMAN](https://www.getpostman.com/apps)
9. In the DA16200 console, you can see the data sent by POSTMAN (HTTP Client) printed by `hex_dump`. Depending on your application data format, you can develop code to parse and handle your HTTP data.

HTTP Server: `PUT`
-- DHCP Client WLAN0: RENEWING
-- DHCP Client WLAN0: BOUND

<table>
<thead>
<tr>
<th>Assigned addr</th>
<th>192.168.31.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lease Time</td>
<td>00h 30m 00s</td>
</tr>
<tr>
<td>Renewal Time</td>
<td>00h 15m 00s</td>
</tr>
</tbody>
</table>

HTTP Server: `POST`

```plaintext```
>>> HTTP Response
- (len=274):

| [00000000] 50 4f 53 54 20 2f 69 6e 64 65 78 2e 68 74 6d 6c | POST /index.html |
| [00000010] 20 48 54 54 50 2f 31 2e 31 0d 0a 43 6f 6e 74 65 | HTTP/1.1...Content-Type: text/html |
| [00000020] 20 50 6f 73 74 6d 61 6e 52 75 6e 74 69 6d 65 2f  | PostmanRuntime/ |
| [00000030] 37 2e 32 34 2e 31 0d 0a 41 63 63 65 70 74 3a 20  | 7.24.1...Accept: |
| [00000040] 2a 2f 2a 0d 0a 50 6f 73 74 6d 61 6e 2d 54 6f 6b  | */...Postman-Token |
| [00000050] 6e: 20 34 30 36 39 63 61 32 65 2d 32 32 65 6e | en: 4069ca2e-22en |
| [00000060] 62 31 37 36 35 39 33 0d 0a 48 6f 73 74 3a 20 31 | b1765993...Host: |
| [00000070] 39 2e 31 68 38 2e 33 31 2e 34 0d 0a 41 63 63 65 | 192.168.31.4...Content-Type: |
| [00000080] 67 65 70 2c 20 64 65 66 6c 61 74 65 2c 20 62 72 | zip, deflate, br |
| [00000090] 0d 0a 43 6f 6e 74 65 6e 74 3a 20 6b 65 6e 70 2d | Connection: keepalive...Content-Length: 18.... |
| [00000100] 61 6c 69 76 65 0d 0a 43 6f 6e 74 65 6e 74 65 66 | ep-alive...Content-Length: |
| [00000110] 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 70 | ppppppppppppppppp |
```plaintext```
4.7 ThreadX API Sample

4.7.1 How to Run

1. Open the workspace for the ThreadX API sample application as follows:
   ○ \sample\ETC\ThreadX\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot. After boot, the ThreadX APIs test will start automatically.

![System Mode: Station Only (0)
>>> FC9000 supplicant Ver1.00-20170213-01
>>> MAC address (sta0): 06:00:00:00:00:00
>>> sta0 interface add OK
>>> Start STA mode...
   [tx_0] Start event send test ...
   [tx_1] Start message queue test ...
   [tx_3] Start semaphore test ...
   [tx_4] Start semaphore test ...
   [tx_6] Start mutex lock test ...
   [tx_7] Start mutex lock test ...
   [tx_8] Start byte-pool usage test ...
   [tx_8] Success to allocate 128 Bytes from byte_pool_0 ...
   [tx_8] Success to allocate 64 Bytes from byte_pool_0 ...
   [tx_8] Success to allocate 1024 Bytes from byte_pool_0 ...
   [tx_8] Byte-pool usage test done...

Figure 34: ThreadX APIs Test

4.7.2 Sample Overview

Each ThreadX product distribution contains sample apps that run on all supported microprocessors. This sample code is designed to illustrate how ThreadX APIs are used in an embedded multi-thread environment. The demonstration consists of initialization, running nine threads, one-byte pool, one queue, one semaphore, one mutex, and one event flags group.

NOTE
See also ThreadX_User_Guide.pdf [3].

4.7.3 Thread Creation and Resource Initialization

The threadx_sample runs after the basic ThreadX initialization is complete. It is responsible for initializing the system resources, including threads, queues, semaphores, mutexes, event flags, and memory pools.

This function creates the demonstration objects in the following order:

```
queue_0 / semaphore_0 / event_flags_0 / mutex_0 /
thread_0 / thread_1 thread_2 / thread_3 / thread_4 / thread_5 / thread_6 / thread_7 / thread_8
```

The demonstration does not create any other additional ThreadX objects. However, an actual application may create system objects during runtime inside of executing threads.

4.7.4 Initial Execution

All threads are created with the TX_AUTO_START option. This makes the threads be initially ready for execution. After tx_application_define completes, control is transferred to the thread scheduler.
and from there to each individual thread. The order in which the threads execute is determined by their priority and the order in which they were created.

In the demonstration, thread_0 executes first because it has the highest priority (it was created with a priority of 1). After thread_0 suspends, thread_5 is executed, followed by the execution of thread_3, thread_4, thread_6, thread_7, thread_1, and finally thread_2.

**NOTE**

Even though thread_3 and thread_4 have the same priority (both created with a priority of 26), thread_3 executes first. This is because thread_3 was created and became ready before thread_4. Threads of equal priority execute in a First In First Out (FIFO) fashion.

```c
void threadx_sample(ULONG arg)
{
    ... ...
    /* Create the message queue shared by threads 1 and 2. */
    status = tx_queue_create(&queue_0,                     // queue pointer
                             "queue 0",                       // queue name
                             TX_1_ULONG,                      // message size
                             message_queue,                   // queue pointer
                             QUEUE_SIZE*sizeof(ULONG));      // queue size

    /* --- For Semaphore usage ------------------------------------ */
    /* Create the semaphore used by threads 3 and 4. */
    status = tx_semaphore_create(&semaphore_0,            // semaphore pointer
                                 "semaphore 0",           // semaphore name
                                 1);                       // initial count

    /* --- For Event group usage ------------------------------------ */
    /* Create the event flags group used by threads 1 and 5. */
    status = tx_event_flags_create(&event_flags_0,        // event group pointer
                                 "event flags 0");        // event name

    /* --- For Mutex lock usage ------------------------------------ */
    /* Create the mutex used by thread 6 and 7 without priority inheritance. */
    status = tx_mutex_create(&mutex_0,                    // mutex pointer
                             "mutex 0",                      // mutex name
                             TX_NO_INHERIT);                // inherit flag

    /* --- For thread create usage -------------------------------- */
    /* Create the main thread. */
    status = tx_thread_create(&thread_0,               // thread pointer
                              "thread 0",                  // thread name
                              tx_0,                         // entry pointer
                              0,                            // argument
                              thread_0_stack,               // thread stack
                              THREAD_STACK_SIZE,            // stack size
                              THREAD_0_PRI,                 // thread priority
                              THREAD_0_PRI,                 // preempt threshold
                              TX_NO_TIME_SLICE,              // time slice
                              TX_AUTO_START);              // auto start flag
```
/* Create threads 1 and 2. 
* These threads pass information through a ThreadX message queue. 
* It is also interesting to note 
* that these threads have a time slice. */
status = tx_thread_create(&thread_1, "thread 1", tx_1, 1,
            thread_1_stack, THREAD_STACK_SIZE,
            THREAD_1_PRI, THREAD_1_PRI,
            4, TX_AUTO_START);

status = tx_thread_create(&thread_2, "thread 2", tx_2, 2,
            thread_2_stack, THREAD_STACK_SIZE,
            THREAD_2 PRI, THREAD_2 PRI,
            4, TX AUTO_START);

/* Create threads 3 and 4. 
* These threads compete for a ThreadX counting semaphore. 
* An interesting thing here is that both threads share 
* the same instruction area. */
status = tx_thread_create(&thread_3, "thread 3", tx_3_4, 3,
            thread_3_stack, THREAD_STACK_SIZE,
            THREAD_3 PRI, THREAD_3 PRI,
            TX NO TIME SLICE, TX AUTO_START);
status = tx_thread_create(&thread_4, "thread 4", tx_3_4, 4,
            thread_4_stack, THREAD_STACK_SIZE,
            THREAD_4 PRI, THREAD_4 PRI,
            TX NO TIME SLICE, TX AUTO_START);

/* Create thread 5. 
* This thread simply pends on an event flag, 
* which will be set by thread_0. */
status = tx_thread_create(&thread_5, "thread 5", tx_5, 5,
            thread_5_stack, THREAD_STACK_SIZE,
            THREAD_5 PRI, THREAD_5 PRI,
            TX NO TIME SLICE, TX AUTO_START);

/* Create threads 6 and 7. 
* These threads compete for a ThreadX mutex. */
status = tx_thread_create(&thread_6, "thread 6", tx_6_7, 6,
            thread_6_stack, THREAD_STACK_SIZE,
            THREAD_6 PRI, THREAD_6 PRI,
            TX NO TIME SLICE, TX AUTO_START);
status = tx_thread_create(&thread_7, "thread 7", tx_6_7, 7,
            thread_7_stack, THREAD_STACK_SIZE,
            THREAD_7 PRI, THREAD_7 PRI,
            TX NO TIME SLICE, TX AUTO_START);

/* --- For Byte-pool usage --------------------------------- */
/* Create byte-pool test thread. */
status = tx_thread_create(&thread_8, "thread 8", tx_8, 0,
            thread_8_stack, THREAD_STACK_SIZE,
            THREAD_8 PRI, THREAD_8 PRI,
            TX NO TIME SLICE, TX AUTO_START);

4.7.5 Threads Operation in Detail

- Thread #0
Function tx_0 marks the entry point of the thread. Thread_0 is the first thread to run in the sample. Its processing is simple: it sets an event flag to wake up thread_5 and sleeps for 100 timer ticks, then repeats the sequence. Thread_0 is the highest priority thread among test threads. When its requested sleep expires, it will preempt any other executing thread in the sample.

```c
static void tx_0(ULONG arg)
{
    /* This thread simply sits in while-forever-sleep loop. */
    while (1) {
        /* Set event flag 0 to wakeup thread 5. */
        status = tx_event_flags_set(&event_flags_0, 0x1, TX_OR);

        /* Check status. */
        if (status == TX_SUCCESS) {
            PRINTF("[tx_%d] Event SET : ", arg);
        } else {
            PRINTF("[tx_%d] Failed to set event flags (0x%x)\n", arg, status);
            break;
        }

        tx_thread_sleep(100); // 100 ticks = 1 second
    }
}
```

- **Thread #1**

Function tx_1 marks the entry point of the thread Thread_1. The thread is the second-to-last in the demonstration in execution order. Its processing consists of sending a message to thread_2 (through queue_0) and repeat the sequence. Notice that thread_1 suspends whenever queue_0 becomes full.

```c
static void tx_1(ULONG arg)
{
    /* This thread simply sends messages to a queue shared by thread 2. */
    while (1) {
        /* Send message to queue 0. */
        tx_msg_buf = get_random_value_ulong();
        status = tx_queue_send(&queue_0, &tx_msg_buf, TX_WAIT_FOREVER);

        /* Check completion status. */
        if (status == TX_SUCCESS) {
            PRINTF("[tx_%d] Message TX (%x) -> ", arg, tx_msg_buf);
        } else {
            PRINTF("[tx_%d] Failed to send a message through message
", arg, status);
            break;
        }

        tx_thread_sleep(200); // 200 ticks = 2 seconds
    }
}
```

- **Thread #2**

Function tx_2 marks the entry point of the thread. Thread_2 is the last thread to be run in the demonstration. Its processing consists of getting a message from thread_1 (through queue_0)
and then repeat the sequence. Notice that thread_2 suspends whenever queue_0 becomes empty.

Although thread_1 and thread_2 share the lowest priority in the demonstration (priority 27), they are also the only threads that are ready for execution most of the time. They are also the only threads created with time-slicing. Each thread can execute for a maximum of 4 timer ticks before the other thread is executed.

```c
static void tx_2(ULONG arg)
{
    ... ...
    /* This thread retrieves messages placed on the queue by thread 1. */
    while (1) {
        /* Retrieve a message from the queue. */
        status = tx_queue_receive(&queue_0, &rx_msg_buf, TX_WAIT_FOREVER);
        /* Check completion status and make sure the message is what we expected. */
        if (status != TX_SUCCESS) {
            PRINTF("[tx_%d] Failed to receive a message through message queue (0x%x)n", arg, status);
            break;
        }
        if (tx_msg_buf == rx_msg_buf) {
            PRINTF("[tx_%d] Message RX (%x)n", arg, rx_msg_buf);
        } else {
            PRINTF("[tx_%d] Wrong message through message queue (0x%x:0x%x)n", arg, tx_msg_buf, rx_msg_buf);
            break;
        }
    }
}
```

● Thread #3 and #4

Function tx_3_4 marks the entry point of both thread_3 and thread_4. Both threads have a priority of 26, which makes them the third and fourth threads in the demonstration system to execute. Each thread is processed in the same manner: get semaphore_0, sleep for 2 timer ticks, release semaphore_0, and then repeat the sequence. Notice that each thread suspends whenever semaphore_0 is unavailable. Also, both threads use the same function for their main processing. This presents no problems because they both have their own unique stack, and C is naturally reentrant. Each thread determines which one it is by examination of the thread input parameter, which is setup when they are created. It is also reasonable to obtain the current thread point during thread execution and compare it with the control block’s address to determine thread identity.

```c
static void tx_3_4(ULONG arg)
{
    ... ...
    /* This function is executed from thread 3 and thread 4. */
    while (1) {
        /* Get the semaphore with suspension. */
        /* As the loop below shows, */
        /* this function competes for ownership of semaphore_0. */
    }
```
status = tx_semaphore_get(&semaphore_0, TX_WAIT_FOREVER);
/* Check status. */
if (status == TX_SUCCESS) {
    PRINTF("[tx_%d] Semaphore GET\n", arg);
} else {
    PRINTF("[tx_%d] Failed to get semaphore (0x%x)\n",
            arg, status);
    break;
}
/* Sleep for 2 ticks to hold the semaphore. */
tx_thread_sleep(2);    // 20 msec
/* Release the semaphore. */
status = tx_semaphore_put(&semaphore_0);
/* Check status. */
if (status == TX_SUCCESS) {
    PRINTF("[tx_%d] Semaphore PUT\n", arg);
} else {
    PRINTF("[tx_%d] Failed to put semaphore (0x%x)\n",
            arg, status);
    break;
}
tx_thread_sleep(300);    // 300 ticks = 3 seconds


- **Thread #5**
  
  Function tx_5 marks the entry point of the thread. Thread_5 is the second thread in the demonstration system to execute. Its processing consists of getting an event flag from thread_0 (through event_flags_0), and then repeat the sequence. Notice that thread_5 suspends whenever the event flag in event_flags_0 is not available.

  ```c
  static void tx_5(ULONG arg)
  {
    ...
    /* This thread simply waits for an event in a forever loop. */
    while (1) {
        /* Wait for event flag 0. */
        status = tx_event_flags_get(&event_flags_0, // group pointer
                                   0x1,       // requested flags
                                   TX_OR_CLEAR,   // get option
                                   &actual_flags, // actual flags
                                   TX_WAIT_FOREVER); // wait option
        /* Check status. */
        if (status == TX_SUCCESS) {
            PRINTF("[tx_%d] Event GET\n", arg);
        } else {
            PRINTF("[tx_%d] Failed to get event (0x%x)\n",
                    arg, status);
            break;
        }
  ```
DA16200 Example Application Guide

if (actual_flags != 0x1) {
    PRINTF("[tx_%d] Wrong event get (0x%x)\n", arg, actual_flags);
    break;
}

Thread #6 and #7

Function tx_6_7 marks the entry point of both thread_6 and thread_7. Both threads have a priority of 8, which makes them the fifth and sixth threads in the demonstration system to execute. Each thread is processed in the same manner: get mutex_0 twice, sleep for 5 timer ticks, release mutex_0 twice, and then repeat the sequence. Notice that each thread suspends whenever mutex_0 is unavailable. Also, both threads use the same function for their main processing. This presents no problems because they both have their own unique stack, and C is naturally reentrant. Each thread determines which one it is by examination of the thread input parameter, which is setup when they are created.

static void tx_6_7(ULONG arg)
{
    ... ...

    /* This function is executed from thread 6 and thread 7.
     * As the loop below shows,
     * these function compete for ownership of mutex_0.
     */
    while (1) {
        /* Get the mutex with suspension. */
        PRINTF("[tx_%d] #1 Mutex get TRY\n", arg);
        status = tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);

        /* Check status. */
        if (status == TX_SUCCESS) {
            PRINTF("[tx_%d] #1 Mutex get OK\n", arg);
        } else {
            PRINTF("[tx_%d] #1 Mutex get FAIL (0x%x)\n", arg, status);
            break;
        }

        /* Get the mutex again with suspension.
         * This shows that an owning thread may retrieve the mutex
         * it owns multiple times. */
        PRINTF("[tx_%d] #2 Mutex get TRY\n", arg);
        status = tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);

        /* Check status. */
        if (status == TX_SUCCESS) {
            PRINTF("[tx_%d] #2 Mutex get OK\n", arg);
        } else {
            PRINTF("[tx_%d] #2 Mutex get FAIL (0x%x)\n", arg, status);
            break;
        }

        /* Sleep for 5 ticks to hold the mutex. */
        tx_thread_sleep(5);     // 50 msec

        /* Release the mutex. */
        PRINTF("[tx_%d] #1 Mutex put TRY\n", arg);
status = tx_mutex_put(&mutex_0);
/* Check status. */
if (status == TX_SUCCESS) {
    PRINTF("[tx_%d] #1 Mutex put OK\n", arg);
} else {
    PRINTF("[tx_%d] #1 Mutex put FAIL (0x%x)\n", arg, status);
    break;
}
/* Release the mutex again.
* This will actually release ownership since it was obtained
  twice. */
PRINTF("[tx_%d] #2 mutex put TRY\n", arg);
status = tx_mutex_put(&mutex_0);
/* Check status. */
if (status == TX_SUCCESS) {
    PRINTF("[tx_%d] #2 Mutex put OK\n", arg);
} else {
    PRINTF("[tx_%d] #2 Mutex put FAIL (0x%x)\n", arg, status);
    break;
}
}
tx_thread_sleep(400);           // 400 ticks = 4 seconds

● Thread #8
Function tx_8 marks the entry point of the thread. Thread_8 is the standalone thread in the
demonstration system to execute. The process of Thread_8 is: create byte pool, allocate byte
pool, release byte pool, and then repeat the sequence. Thread_8 shows the usage of byte-pool
processing.

static void tx_8(ULONG arg)
{
    ...
/* --- For byte pool usage ------------------------------------------- */
/* Create a byte memory pool */
tx_byte_pool_create(&byte_pool_0, // byte pool pointer
    "byte pool 0", // byte pool name
    byte_pool_buffer, // byte pool start pointer
    BYTE_POOL_SIZE);  // byte pool size
/* Allocate the test buffer for temporary buffer */
status = tx_byte_allocate(&byte_pool_0, // Pool pointer
    (void **)&first_ptr, // allocated pointer
    128, // requested size
    TX_NO_WAIT);  // wait option
if (status != TX_SUCCESS) {
    PRINTF("[%s] #1 Failed to allocate from byte pool (0x%x)\n",
    __func__, status);
    goto finish;
} else {
    PRINTF("Success to allocate 128 Bytes from byte_pool_0 ...\n");
}
status = tx_byte_allocate(&byte_pool_0, (void **)&second_ptr, 64,
    TX_NO_WAIT);
    if (status != TX_SUCCESS) {
        PRINTF("#2 Failed to allocate from byte pool (0x%x)\n", status);
    }
DA16200 Example Application Guide

```c
goto finish;
} else {
    PRINTF("Success to allocate 64 Bytes from byte_pool_0 ...
");
    status = tx_byte_allocate(&byte_pool_0, (void **)&third_ptr, 1024,
    TX_NO_WAIT);
    if (status != TX_SUCCESS) {
        PRINTF("#3 Failed to allocate from byte pool (0x%x)n", status);
        goto finish;
    } else {
        PRINTF("Success to allocate 1024 Bytes from byte_pool_0 ...
");
    }
}
finish :
if (first_ptr != TX_NULL);
    tx_byte_release(first_ptr);
if (second_ptr != TX_NULL);
    tx_byte_release(second_ptr);
if (third_ptr != TX_NULL);
    tx_byte_release(third_ptr);
    tx_byte_pool_delete(&byte_pool_0);
```

4.8 RTC Timer with DPM Function

This sample code describes how to use the RTC Timer to operate in DPM sleep mode 1, 2, and 3.

4.8.1 How to Run

3. Open the workspace for the RTC timer sample application as follows:
   ○ .\Peripheral\RTC_Timer_DPM\build\DA16xxx.eww
4. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
5. Use the console to setup the Wi-Fi station interface and enable DPM mode.
6. After boot, the RTC timer sample application starts automatically.

Notice that the user can select DPM sleep mode in the sample code.

```c
/* Defines for sample */
#undef  SAMPLE_FOR_DPM_SLEEP_1          // Sleep Mode 1
#define  SAMPLE_FOR_DPM_SLEEP_2          // Sleep Mode 2
#undef  SAMPLE_FOR_DPM_SLEEP_3          // Sleep Mode 3
```

4.8.2 Application Initialization

The User Application may retrieve user configuration data from NVRAM or retention memory (RTM) after boot is completed and this can be accomplished according to the DPM mode status. The User Application can use retention memory if DPM mode is enabled. ThreadX timer primitive is used rather than RTC timer in normal mode.

```c
/* This sample function always run on DPM mode ... */
rtm_len = dpm_user_mem_get(SAMPLE_RTC_TIMER, (UCHAR **)&rtc_sample_info);
if (rtm_len == 0) {
    status = dpm_user_mem_alloc(SAMPLE_RTC_TIMER,
                                (VOID **)&rtc_sample_info,
                                sizeof(rtc_sample_info_t),
                                100);
    if (status != TX_SUCCESS) {
        PRINTF("[%s] Failed to allocate RTM area !!!\n", __func__);
        dpm_app_unregister(SAMPLE_RTC_TIMER);
```
4.8.3 Timer Creation: DPM Sleep mode 1

DPM sleep mode 1 means power-off except for RTC resources and the retention memory area, if needed. But in this case, maintaining the retention memory area cannot be guaranteed.

A DUT with DPM sleep mode 1 can be woken up by just an external wakeup resource and run the same as Power-on-Reset.

To go to DPM sleep mode 1, just run API dpm_sleep_start_mode_1().

```c
void rtc_timer_sample(ULONG arg)
{
    /* FALSE : Not maintain RTM area for DPM operation */
    dpm_sleep_start_mode_1(TRUE);
}
```

4.8.4 Timer Creation: DPM Sleep mode 2

DPM sleep mode 2 means power-off with RTC alive and retention memory area if needed. A DUT with DPM sleep mode 2 can be woken up by an external wake-up source and RTC timer resources. When a DUT wakes up from both wakeup sources (external or RTC), it runs the same as a normal POR with saved retention memory area if configured before sleep mode 2.

To go to dpm sleep mode 2, just run API dpm_sleep_start_mode_2().

```c
void rtc_timer_sample(ULONG arg)
{
    unsigned long long  wakeup_time;
    /* Just work in case of RTC timer wakeup */
    if (   dpm_mode_is_wakeup() == DPM_WAKEUP
        && dpm_get_wakeup_source() != WAKEUP_COUNTER_WITH_RETENTION)
    {
        dpm_app_sleep_ready_set(SAMPLE_RTC_TIMER);
        return;
    }

    /* TRUE : Maintain RTM area for DPM operation */
    wakeup_time = MICROSEC_FOR_ONE_SEC * RTC_TIMER_WAKEUP_ONCE;
    dpm_sleep_start_mode_2(wakeup_time, TRUE);
}
```

4.8.5 Timer Creation: DPM Sleep mode 3

DPM sleep mode 3 means power-off with RTC resources and retention memory area alive, plus pTIM running. This sleep mode is what we normally call “DPM Sleep” (aka ‘connected sleep’. The
other two sleep modes are 'unconnected sleep'). For more detailed information on DPM sleep mode 3, please see the DA16200 EVK User Guide [2].

This sample code shows how to create a one-shot RTC timer and a periodic RTC timer.

```c
void rtc_timer_sample(ULONG arg)
{
    ULONG status;

    if (dpm_mode_is_wakeup() == NORMAL_BOOT)
    {
        /* Create a timer only once during normal boot. */
        dpm_app_sleep_ready_clear(SAMPLE_RTC_TIMER);

        /* One-Shot timer */
        status = dpm_timer_create(SAMPLE_RTC_TIMER,
                "timer1",
                rtc_timer_dpm_once_cb,
                RTC_TIMER_WAKEUP_ONCE,
                0);
        if (status == SAMPLE_DPM_TIMER_ERR)
        {
            PRINTF(">>> Start test DPM sleep mode 3 : Fail to create One-Shot timer\n");
            tx_thread_sleep(2); // Delay to display above message on console ...
        }

        /* Periodic timer */
        status = dpm_timer_create(SAMPLE_RTC_TIMER,
                "timer2",
                rtc_timer_dpm_periodic_cb,
                RTC_TIMER_WAKEUP_PERIOD,
                RTC_TIMER_WAKEUP_PERIOD);
        if (status == SAMPLE_DPM_TIMER_ERR)
        {
            PRINTF(">>> Start test DPM sleep mode 3 : Fail to create Periodic timer\n");
            tx_thread_sleep(2); // Delay to display above message on console ...
        }

        dpm_app_sleep_ready_set(SAMPLE_RTC_TIMER);
    }
    else
    {
        /* Notice initialize done to DPM module */
        dpm_app_wakeup_done(SAMPLE_RTC_TIMER);
    }

    while (1)
    {
        /* Nothing to do... */
        tx_thread_sleep(100);
    }
}
```
4.8.6 Timer Creation: DPM Sleep mode 3

DPM sleep mode 3 means power-off with RTC resources and retention memory area alive, plus pTIM running. This sleep mode is what we normally call “DPM Sleep” (aka ‘connected sleep’. The other two sleep modes are ‘unconnected sleep’). For more detailed information on DPM sleep mode 3, please see the DA16200 EVK User Guide [2].

This sample code shows how to create a one-shot RTC timer and a periodic RTC timer, and how to use retention memory.

```c
case 3 :    // DPM Sleep Mode 3
{
    /* Notice initialize done to DPM module */
    dpm_app_wakeup_done(SAMPLE_RTC_TIMER);

    /* Create RTC timer for DPM wakeup */
    if (dpm_mode_is_wakeup() == NORMAL_BOOT) {
        /* One-Shot RTC timer */
        oneshot_timer_id = dpm_timer_create(
            RTC_TIMER_WAKEUP_ONE_SHOT,
            SAMPLE_RTC_TIMER,
            oneshot_timer_id,
            ONESHOT_TIMER,
            rtc_timer_dpm_oneshot_cb);

        rtc_sample_info->oneshot_timer_id = oneshot_timer_id;
        PRINTF(">>> Start test DPM sleep mode 3 : One-Shot timer\n");

        /* Periodic RTC timer */
        periodic_timer_id = dpm_timer_create(
            RTC_TIMER_WAKEUP_PERIOD,
            SAMPLE_RTC_TIMER,
            periodic_timer_id,
            PERIODIC_TIMER,
            rtc_timer_dpm_periodic_cb);

        rtc_sample_info->periodic_timer_id = periodic_timer_id;
        PRINTF(">>> Start test DPM sleep mode 3 : periodic_count=%d\n",
                rtc_sample_info->periodic_count);
        dpm_app_sleep_ready_set(SAMPLE_RTC_TIMER);
    } else if (dpm_mode_is_wakeup() == DPM_WAKEUP) {
        // User code when waked up from DPM sleep mode 3...
    }
}
```

4.9 Get SCAN Result Sample

4.9.1 How to Run

1. Open the workspace for the SCAN result sample application as follows:
   ○ `\sample\ETC\Get_Scan_Result\build\DA16xxx.eww`
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. After boot is completed, the get_scan_result sample will start automatically.
4.9.2 Sample Overview

This sample shows how to use the void get_scan_result(void *)user_buf API, to get the SCAN result on STA mode and Soft-AP mode.

4.9.3 Application Initialization

The get_scan_result_sample function executes after the basic ThreadX initialization is completed. This sample just calls the user API "void get_scan_result()".

```c
void get_scan_result_sample(ULONG arg)
{
    char *user_buf = NULL;
    scan_result_t *scan_result;
    int i;

    /* Allocate buffer to get scan result */
    user_buf = malloc(SCAN_RSP_BUF_SIZE);

    /* Get scan result */
    get_scan_result((void *) user_buf);

    /* Display result on console */
    scan_result = (scan_result_t *)user_buf;
    PRINTF("\n>>> Scanned AP List (Total : %d) \n", scan_result->ssid_cnt);
    for (i = 0; i < scan_result->ssid_cnt; i++) {
        /* Display AP data */
    }
}
```

4.9.4 Get SCAN Result

After the API "get_scan_result()" has run, the user/developer can use the received data. This sample code shows how to display the scan list in the console.

```c
/* Display result on console */
scan_result = (scan_result_t *)&user_buf;

PRINTF("\n>>> Scanned AP List (Total : %d) \n", scan_result->ssid_cnt);
for (i = 0; i < scan_result->ssid_cnt; i++) {
    /* Display AP data */
```
PRINTF(" %02d) SSID: %s, RSSI: %d, Security: %d\n",
    i + 1,
    scan_result->scanned_ap_info[i].ssid,
    scan_result->scanned_ap_info[i].rssi,
    scan_result->scanned_ap_info[i].auth_mode) ;
}
/* Buffer free */
free(user_buf);
}

The SCAN results are stored in the following data structure format.

typedef struct scanned_ap_info {
    int     auth_mode;
    int     rssi;
    char    ssid[128];
} scanned_ap_info_t;

typedef struct scan_result_to_app {
    int     ssid_cnt;
    scanned_ap_info_t scanned_ap_info[MAX_SCAN_AP_CNT];
} scan_result_t;

4.10 User sflash Read / Write Example

4.10.1 How to Run

1. In the SDK, enable the sample application as follows:
   
   ```c
   #define __ENABLE_TEST_APP__ /* - User Sample Codes for Network apps -- */
   #ifdef __ENABLE_TEST_APP__
   #define __SFLASH_API_SAMPLE__
   #endif /* __ENABLE_TEST_APP__ */
   
   ○ The sample application code is written in the following source file:
     ```c
     \src\application\sample_code\sflash_sample.c
     ```
   ```c
   2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
   3. After boot, the sample will start automatically.
4.10.2 User Thread

The user thread of the sflash api sample application is defined as below and is executed by the system. SAMPLE_SFLASH should be a unique name in order to create a thread. This test is not related to network initialization and dpm mode.

```c
static const app_thread_info_t sample_apps_table[] = {
    { SAMPLE_SFLASH, user_sflash_test, 1024, USER_PRI_APP(1), FALSE, FALSE,
        UNDEF_PORT, RUN_ALL_MODE },
};
```

4.10.3 Application Initialization

The user_sflash_test function is run after the basic ThreadX initialization is complete.

```c
void user_sflash_test(ULONG arg)
{
    PRINTF("n- Test Serial-Flash Read/Write ...
    test_sflash_write();
    OAL_MSLEEP(100);
    test_sflash_read();
}
```

4.10.4 Sflash Read and Write

// user sflash APIs

/*
sflash user area available:
- User area of 2MB sflash: 12KB (SFLASH_USER_AREA_START~)
*/
extern UINT user_sflash_read(UINT sflash_addr, VOID *rd_buf, UINT rd_size);
sflash_addr: please see above user sflash area
rd_buf: buffer to which data is copied
rd_size: data size

extern UINT user_sflash_write(UINT sflash_addr, UCHAR *wr_buf, UINT wr_size);
sflash_addr: please see above user sflash area
rd_buf: buffer from which data is copied to sflash_addr
rd_size: data size

void test_sflash_write(void)
{
    UCHAR *wr_buf = TX_NULL;
    UINT wr_addr;
    #define SFLASH_WR_TEST_ADDR SFLASH_USER_AREA_START // write address
    #define TEST_WR_SIZE SF_SECTOR_SZ // 4K

    wr_buf = (UCHAR *)malloc(TEST_WR_SIZE);
    if (wr_buf == TX_NULL) {
        PRINTF("[%s] malloc fail ...
", __func__);
        return;
    }

    memset(wr_buf, 0, TEST_WR_SIZE);
    for (int i = 0; i < TEST_WR_SIZE; i++) {
        wr_buf[i] = 0x41; // A
    }

    wr_addr = SFLASH_WR_TEST_ADDR;
    PRINTF("=== SFLASH Write Data
");

    user_sflash_write(wr_addr, wr_buf, TEST_WR_SIZE); // this is the function to
    invoke to write data to sflash user area
}

void test_sflash_read(void)
{
    UCHAR *rd_buf = TX_NULL;
    UINT rd_addr;
    UINT status;
    #define SFLASH_RD_TEST_ADDR SFLASH_USER_AREA_START
    #define TEST_RD_SIZE (1 * 1024)

    rd_buf = (UCHAR *)malloc(TEST_RD_SIZE);
    if (rd_buf == TX_NULL) {
        PRINTF("[%s] malloc fail ...
", __func__);
        return;
    }

    memset(rd_buf, 0, TEST_RD_SIZE);

    rd_addr = SFLASH_RD_TEST_ADDR;
    status = user_sflash_read(rd_addr, (VOID *)rd_buf, TEST_RD_SIZE); // this is
    the function to invoke to read data from sflash user area
if (status == TRUE) {
    hex_dump(rd_buf, 128);
}
free(rd_buf);

NOTE

user_sflash_read/write is a blocking function. Take special care when you run this code under DPM mode enabled (sleep2 or sleep3 applications): when you invoke user_sflash_write(), please make sure that you get the result before the DPM sleeping API is invoked.

4.11 Crypto Algorithms - AES

The AES algorithms sample application demonstrates common use cases of AES ciphers such as CBC, CFB, and ECB, etc. The DA16200 SDK includes an mbedTLS library. The API of AES algorithms is the same as what the mbedTLS library provides. This section describes how the AES algorithms sample application is built and works.

4.11.1 How to Run

1. Open the workspace for the Crypto Algorithms of the AES application as follows:
   ○ `\sample\Crypto\Crypto_AES\build\DA16xxx.eww`
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application runs five types of crypto algorithms:

   ○ AES-CBC-128, 192, and 256
   ○ AES-CFB128-128, 192, and 256
   ○ AES-ECB-128, 192, and 256
   ○ AES-CTR-128
   ○ AES-CCM

* AES-CBC-128 (dec): passed  
* AES-CBC-128 (enc): passed  
* AES-CBC-192 (dec): passed  
* AES-CBC-192 (enc): passed  
* AES-CBC-256 (dec): passed  
* AES-CBC-256 (enc): passed  
* AES-CFB128-128 (dec): passed  
* AES-CFB128-128 (enc): passed  
* AES-CFB128-192 (dec): passed  
* AES-CFB128-192 (enc): passed  
* AES-CFB128-256 (dec): passed  
* AES-CFB128-256 (enc): passed  
* AES-ECB-128 (dec): passed  
* AES-ECB-128 (enc): passed  
* AES-ECB-192 (dec): passed  
* AES-ECB-192 (enc): passed  
* AES-ECB-256 (dec): passed  
* AES-ECB-256 (enc): passed  
* AES-CTR-128 (dec): passed  
* AES-CTR-128 (enc): passed
4.11.2 Application Initialization

DA16200 SDK provides an Mbed TLS library. This library helps with an easy implementation of the user application. The example below describes how the user uses the AES algorithms of the Mbed TLS library to encrypt and decrypt data.

```c
void crypto_sample_aes(ULONG arg)
{
    #if defined(MBEDTLS_CIPHER_MODE_CBC)
        crypto_sample_aes_cbc();
    #endif  /* MBEDTLS_CIPHER_MODE_CBC */

    #if defined(MBEDTLS_CIPHER_MODE_CFB)
        crypto_sample_aes_cfb();
    #endif  /* MBEDTLS_CIPHER_MODE_CFB */

        crypto_sample_aes_ecb();

    #if defined(MBEDTLS_CIPHER_MODE_CTR)
        crypto_sample_aes_ctr();
    #endif  /* MBEDTLS_CIPHER_MODE_CTR */

        crypto_sample_aes_ccm();
        crypto_sample_aes_gcm();

    #if defined(MBEDTLS_CIPHER_MODE_OFB)
        crypto_sample_aes_ofb();
    #endif  /* MBEDTLS_CIPHER_MODE_OFB */

    return;
}
```

4.11.3 Crypto Algorithm for AES-CBC-128, 192, and 256


```c
int crypto_sample_aes_cbc()
{
    mbedtls_aes_context *ctx = NULL;

    // Initialize the AES context.
```
The mbedtls_aes_context is the AES context-type definition to use the AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypt_cbc does an AES-CBC encryption or decryption operation on full blocks. And does the operation defined in the mode parameter (encrypt/decrypt), on the input data buffer defined in the input parameter. To do encryption or decryption, function mbedtls_aes_setkey_enc or mbedtls_aes_setkey_dec should be called before. After the operation is finished, function mbedtls_aes_free should be called to clear the AES context.

4.11.4 Crypto Algorithm for AES-CFB128-128, 192, and 256


```c
int crypto_sample_aes_cfb()
{
    mbedtls_aes_context *ctx = NULL;

    // Initialize the AES context.
    mbedtls_aes_init(ctx);

    for (i = 0; i < 6; i++) {
        u = i >> 1;
        v = i & 1;
        PRNITF("* AES-CBC-\%3d (%s): ", 128 + u * 64,
            (v == MBEDTLS_AES_DECRYPT) ? "dec" : "enc");
        if (v == MBEDTLS_AES_DECRYPT) {
            // Set the decryption key.
            mbedtls_aes_setkey_dec(ctx, key, 128 + u * 64);
            // Performs an AES-CBC decryption operation on full blocks.
            for (j = 0; j < CRYPTO_SAMPLE_AES_LOOP_COUNT ; j++) {
                mbedtls_aes_crypt_cbc(ctx, v, i6, iv, buf, buf);
            }
        } else {
            // Set the encryption key.
            mbedtls_aes_setkey_enc(ctx, key, 128 + u * 64);
            // Performs an AES-CBC encryption operation on full blocks.
            for (j = 0 ; j < CRYPTO_SAMPLE_AES_LOOP_COUNT ; j++) {
                unsigned char tmp[16] = {0x00,};
                mbedtls_aes_crypt_cbc(ctx, v, i6, iv, buf, buf);
                memcpy(tmp, prv, 16);
                memcpy(prv, buf, 16);
                memcpy(buf, tmp, 16);
            }
        }
    }
    // Clear the AES context.
    mbedtls_aes_free(ctx);
}
```
PRINTF("* AES-CFB128-%3d (%s): ", 128 + u * 64, 
    (v == MBEDTLS_AES_DECRYPT) ? "dec" : "enc");

    // Set the key.
    mbedtls_aes_setkey_enc(ctx, key, 128 + u * 64);

    if (v == MBEDTLS_AES_DECRYPT) {
        // Perform an AES-CFB128 decryption operation.
        mbedtls_aes_crypt_cfb128(ctx, v, 64, &offset, iv, buf, buf);
    } else {
        // Perform an AES-CFB128 encryption operation.
        mbedtls_aes_crypt_cfb128(ctx, v, 64, &offset, iv, buf, buf);
    }

    // Clear the AES context.
    mbedtls_aes_free(ctx);
}

The mbedtls_aes_context is the AES context-type definition to use AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypt_cfb128 does AES-CFB128 encryption or decryption. And does the operation defined in the mode parameter (encrypt or decrypt) on the input data buffer defined in the input parameter. For CFB, the user should set up the context with function mbedtls_aes_setkey_enc, regardless of whether you do encryption or decryption operations, that is, regardless of the mode parameter. This is because CFB mode uses the same key schedule for encryption and decryption. After the operation is finished, function mbedtls_aes_free should be called to clear the AES context.
4.11.5  Crypto Algorithm for AES-ECB-128, 192, and 256

DA16200 supports crypto algorithm for AES-ECB-128, 192, and 256. In order to explain how AES-ECB works, see the test vector in http://csrc.nist.gov/archive/aes/rijndael/rijndael-vals.zip.

```c
int crypto_sample_aes_ecb()
{
    mbedtls_aes_context *ctx = NULL;

    // Initialize the AES context.
    mbedtls_aes_init(ctx);

    for (i = 0; i < 6; i++) {
        u = i >> 1;
        v = i & 1;

        PRINTF("* AES-ECB-%3d (%s): ", 128 + u * 64,
               (v ==MBEDTLS_AES_DECRYPT) ? "dec" : "enc");

        if (v ==MBEDTLS_AES_DECRYPT) {
            // Set the decryption key.
            mbedtls_aes_setkey_dec(ctx, key, 128 + u * 64);

            // Perform an AES single-block decryption operation.
            for (j = 0; j < CRYPTO_SAMPLE_AES_LOOP_COUNT; j++) {
                mbedtls_aes_crypt_ecb(ctx, v, buf, buf);
            }
        } else {
            // Set the encryption key.
            mbedtls_aes_setkey_enc(ctx, key, 128 + u * 64);

            // Perform an AES single-block encryption operation.
            for (j = 0; j < CRYPTO_ITEM_AES_LOOP_COUNT; j++) {
                mbedtls_aes_crypt_ecb(ctx, v, buf, buf);
            }
        }
    }

    // Clear the AES context.
    mbedtls_aes_free(ctx);
}
```

The mbedtls_aes_context is the AES context-type definition to use AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypt_ecb does an AES single-block encryption or decryption operation. And, does the operation defined in the mode parameter (encrypt or decrypt) on the input data buffer defined in the input parameter. Function mbedtls_aes_init and either function mbedtls_aes_setkey_enc function or function mbedtls_aes_setkey_dec should be called before the first call to this API with the same context. After the operation is finished, function mbedtls_aes_free should be called to clear the AES context.
4.11.6 Crypto Algorithm for AES-CTR-128

DA16200 supports crypto algorithm for AES-CTR-128. To explain how AES-CTR works, see the Test Vectors section in http://www.faqs.org/rfcs/rfc3686.html.

```c
int crypto_sample_aes_ctr()
{
    mbedtls_aes_context *ctx = NULL;

    // Initialize the AES context.
    mbedtls_aes_init(ctx);

    for (i = 0; i < 2; i++) {
        v = i & 1;
        printf("* AES-CTR-128 (%s): ",
               (v == MBEDTLS_AES_DECRYPT) ? "dec" : "enc");

        // Set the key.
        mbedtls_aes_setkey_enc(ctx, key, 128);
        if (v == MBEDTLS_AES_DECRYPT) {
            // Perform an AES-CTR decryption operation.
            mbedtls_aes_crypt_ctr(ctx, len, &offset, nonce_counter, stream_block,
                                   buf, buf);
        } else {
            // Perform an AES-CTR encryption operation.
            mbedtls_aes_crypt_ctr(ctx, len, &offset, nonce_counter, stream_block,
                                   buf, buf);
        }
    }

    // Clear the AES context.
    mbedtls_aes_free(ctx);
}
```

The mbedtls_aes_context is the AES context-type definition to use AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypt_ctr does an AES-CTR encryption or decryption operation. And, does the operation defined in the mode parameter (encrypt/decrypt) on the input data buffer, defined in the input parameter. Due to the nature of CTR, you should use the same key schedule for both encryption and decryption operations. Therefore, use the context initialized with function mbedtls_aes_setkey_enc for both MBEDTLS_AES_ENCRYPT and MBEDTLS_AES_DECRYPT. After the operation is finished, call function mbedtls_aes_free to clear the AES context.

4.11.7 Crypto Algorithm for AES-CCM-128, 192, and 256

DA16200 supports a crypto algorithm for AES-CCM-128, 192, and 256. To explain how AES-CCM works, see the test vector in SP800-38C Appendix C #1.

```c
int crypto_sample_aes_ccm()
{
    mbedtls_ccm_context *ctx = NULL;

    // Initialize the CCM context
    mbedtls_ccm_init(ctx);

    /* Initialize the CCM context set in the ctx parameter
     * and sets the encryption key.
     */
    ret = mbedtls_ccm_setkey(ctx, MBEDTLS_CIPHER_ID_AES,
                             crypto_sample_ccm_key, 8 * sizeof(crypto_sample_ccm_key));
```
PRINTF("* CCM-AES (enc): ");

// Encrypt a buffer using CCM.
ret = mbedtls_ccm_encrypt_and_tag(ctx, crypto_sample_ccm_msg_len,
        crypto_sample_ccm_iv, crypto_sample_ccm_iv_len,
        crypto_sample_ccm_ad, crypto_sample_ccm_add_len,
        crypto_sample_ccm_msg, out,
        out + crypto_sample_ccm_msg_len,
        crypto_sample_ccm_tag_len);

PRINTF("* CCM-AES (dec): ");

// Perform a CCM* authenticated decryption of a buffer.
ret = mbedtls_ccm_auth_decrypt(ctx, crypto_sample_ccm_msg_len,
        crypto_sample_ccm_iv, crypto_sample_ccm_iv_len,
        crypto_sample_ccm_ad, crypto_sample_ccm_add_len,
        crypto_sample_ccm_res, out,
        crypto_sample_ccm_res + crypto_sample_ccm_msg_len,
        crypto_sample_ccm_tag_len);

// Clear the CCM context.
 mbedtls_ccm_free(ctx);
}

The mbedtls_ccm_context is the CCM context-type definition for the CCM authenticated encryption mode for block ciphers. It is initialized by function mbedtls_ccm_init. Function mbedtls_ccm_setkey initializes the CCM context set in the ctx parameter and sets the encryption key. Function mbedtls_ccm_encrypt_and_tag encrypts a buffer with CCM. And function mbedtls_ccm_auth_decrypt does CCM-authenticated decryption of a buffer. After the operation is finished, call function mbedtls_ccm_free to release and clear the specified CCM context and underlying cipher subcontext.

4.11.8 Crypto Algorithm for AES-GCM-128, 192, and 256

DA16200 supports a crypto algorithm for AES-GCM-128, 192, and 256. To explain how AES-GCM works, see the test vector in the GCM test vectors of CSRC (http://csrc.nist.gov/groups/STM/cavp/documents/mac/gcmtestvectors.zip).

```
int crypto_sample_aes_gcm()
{
    //The GCM context structure.
    mbedtls_gcm_context *ctx = NULL;
    mbedtls_cipher_id_t cipher = MBEDTLS_CIPHER_ID_AES;

    // Initialize the specified GCM context.
    mbedtls_gcm_init(ctx);

    // AES-GCM Encryption Test
    for (j = 0; j < 3; j++) {
        PRINTF("* AES-GCM-%3d (%s): ", key_len, "enc");

        // Associate a GCM context with a cipher algorithm and a key.
        mbedtls_gcm_setkey(ctx, cipher, crypto_sample_gcm_key, key_len);

        // Perform GCM encryption of a buffer.
        ret = mbedtls_gcm_encrypt_and_tag(ctx, MBEDTLS_GCM_ENCRYPT,
                sizeof(crypto_sample_gcm_pt),
                mbedtls_gcm_encrypt_and_tag(ctx, MBEDTLS_GCM_ENCRYPT,
                sizeof(crypto_sample_gcm_iv), sizeof(crypto_sample_gcm_additional),
                mbedtls_gcm_encrypt_and_tag(ctx, MBEDTLS_GCM_ENCRYPT,
                sizeof(crypto_sample_gcm_msg),
                mbedtls_gcm_encrypt_and_tag(ctx, MBEDTLS_GCM_ENCRYPT,
                sizeof(crypto_sample_gcm_add),
                mbedtls_gcm_encrypt_and_tag(ctx, MBEDTLS_GCM_ENCRYPT,
                sizeof(crypto_sample_gcm_additional),
```

The mbedtls_gcm_encrypt and mbedtls_gcm_auth_decrypt are used to perform GCM encryption and authentication, respectively. These functions allow for flexible usage of GCM in various applications.
crypto_sample_gcm_pt, buf,
16, tag_buf);

// Clear a GCM context and the underlying cipher sub-context.
mbedtls_gcm_free(ctx);
}

//AES-GCM Decryption Test
for (j = 0; j < 3; j++) {
    PRINTF("* AES-GCM-%3d (%s): ", key_len, "dec");

    // Associate a GCM context with a cipher algorithm and a key.
mbedtls_gcm_setkey(ctx, cipher, crypto_sample_gcm_key, key_len);

    // Perform GCM decryption of a buffer.
    ret = mbedtls_gcm_crypt_and_tag(ctx, MBEDTLS_GCM_DECRYPT,
        sizeof(crypto_sample_gcm_pt),
        crypto_sample_gcm_iv, sizeof(crypto_sample_gcm_iv),
        crypto_sample_gcm_additional,
        crypto_sample_gcm_ct[j], buf,
        16, tag_buf);

    // Clear a GCM context and the underlying cipher sub-context.
mbedtls_gcm_free(ctx);
}

The mbedtls_gcm_context is the GCM context-type definition. It is initialized by function mbedtls_gcm_init. Function mbedtls_gcm_setkey associates a GCM context with a cipher algorithm (AES) and a key. Function mbedtls_gcm_crypt_and_tag does GCM encryption or decryption of a buffer by the second parameter. After the operation is finished, function mbedtls_gcm_free should be called to clear a GCM context and underlying cipher sub-context.

4.11.9 Crypto Algorithm for AES-OFB-128, 192, and 256
DA16200 supports crypto algorithm for AES-OFB-128, 192, and 256. To explain how AES-OFB works, see the test vector in the OFB test vectors of CSRC (https://csrc.nist.gov/publications/detail/sp/800-38a/final).

```c
int crypto_sample_aes_ofb()
{
    mbedtls_aes_context *ctx = NULL;

    // Initialize the AES context.
mbedtls_aes_init(ctx);

    // Test OFB mode
    for (i = 0; i < 6; i++) {
        PRINTF("* AES-OFB-%3d (%s): ", keybits,
            (v == MBEDTLS_AES_DECRYPT) ? "dec" : "enc");

        memcpy(iv, crypto_sample_aes_ofb_iv, 16);
        memcpy(key, crypto_sample_aes_ofb_key[u], keybits / 8);

        // Set the encryption key.
        ret = mbedtls_aes_setkey_enc(ctx, key, keybits);

        if (v == MBEDTLS_AES_DECRYPT) {
            memcpy(buf, crypto_sample_aes_ofb_ct[u], 64);
            expected_out = crypto_sample_aes_ofb_pt;
```

The mbedtls_aes_context is the AES context-type definition to use the AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypt_ofb does an AES-OFB (Output Feedback Mode) encryption or decryption operation. For OFB, the user should set up the context with function mbedtls_aes_setkey_enc, regardless of whether the user does an encryption or decryption operation. This is because OFB mode uses the same key schedule for encryption and encryption. The OFB operation is identical for encryption or decryption, therefore no operation mode needs to be specified. After the operation is finished, call function mbedtls_aes_free to clear the AES context.

4.12 Crypto Algorithms - DES

The DES algorithm sample application demonstrates common use cases of DES and Triple-DES ciphers. The DA16200 SDK includes an mbedTLS library. The API of the DES algorithm is the same as what the mbedTLS library provides. This section describes how the DES algorithm sample application is built and works.

4.12.1 How to Run

1. Open the workspace for the Crypto Algorithms of the DES application as follows:
   ○ .\sample\Crypto\Crypto_DES\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application runs two types of crypto algorithms:

   ○ DES-CBC-56
   ○ DES3-CBC-112 and 168
   * DES -CBC- 56 (dec): passed
   * DES -CBC- 56 (enc): passed
   * DES3-CBC-112 (dec): passed
   * DES3-CBC-112 (enc): passed
   * DES3-CBC-168 (dec): passed
   * DES3-CBC-168 (enc): passed

4.12.2 Application Initialization

DA16200 SDK provides an Mbed TLS library. This library helps with an easy implementation of the user application. The example below shows how a user uses DES algorithms of the Mbed TLS library to encrypt and decrypt data.

```c
void crypto_sample_des(ULONG arg)
{
    #if defined(MBEDTLS_CIPHER_MODE_CBC)
    crypto_sample_des_cbc();
    #endif /* MBEDTLS_CIPHER_MODE_CBC */
    return ;
}
```

*DES -CBC- 56 (dec): passed
*DES -CBC- 56 (enc): passed
*DES3-CBC-112 (dec): passed
*DES3-CBC-112 (enc): passed
*DES3-CBC-168 (dec): passed
*DES3-CBC-168 (enc): passed*
4.12.3 Crypto Algorithm for DES-CBC-56, DES3-CBC-112, and 168


```c
int crypto_sample_des_cbc()
{
    mbedtls_des_context *ctx = NULL;
    mbedtls_des3_context *ctx3 = NULL;

    // Initialize the DES context.
    mbedtls_des_init(ctx);

    // Initialize the Triple-DES context.
    mbedtls_des3_init(ctx3);

    for(i = 0; i < 6; i++) {
        u = i >> 1;
        v = i & 1;

        PRINTF("** DES%c-CBC-%3d (%s): ",
              ( u == 0 ) ? ' ' : '3', 56 + u * 56,
              ( v == MBEDTLS_DES_DECRYPT ) ? "dec" : "enc");
        switch (i) {
            case 0:
                // DES key schedule (56-bit, decryption).
                mbedtls_des_setkey_dec(ctx, crypto_sample_des3_keys);
                break;
            case 1:
                // DES key schedule (56-bit, encryption).
                mbedtls_des_setkey_enc(ctx, crypto_sample_des3_keys);
                break;
            case 2:
                // Triple-DES key schedule (112-bit, decryption).
                mbedtls_des3_set2key_dec(ctx3, crypto_sample_des3_keys);
                break;
            case 3:
                // Triple-DES key schedule (112-bit, encryption).
                mbedtls_des3_set2key_enc(ctx3, crypto_sample_des3_keys);
                break;
            case 4:
                // Triple-DES key schedule (168-bit, decryption).
                mbedtls_des3_set3key_dec(ctx3, crypto_sample_des3_keys);
                break;
            case 5:
                // Triple-DES key schedule (168-bit, encryption).
                mbedtls_des3_set3key_enc(ctx3, crypto_sample_des3_keys);
                break;
        }

        if (v == MBEDTLS_DES_DECRYPT) {
            for (j = 0 ; j < CRYPTO_SAMPLE_DES_LOOP_COUNT ; j++) {
                if(u == 0) {
                    // DES-CBC buffer decryption.
                    mbedtls_des_crypt_cbc(ctx, v, 8, iv, buf, buf);
                } else {
```
// 3DES-CBC buffer decryption.
  mbedtls_des3_crypt_cbc(ctx3, v, 8, iv, buf, buf);
}
} else {
  for (j = 0; j < CRYPTO_SAMPLE_DES_LOOP_COUNT; j++) {
    if (u == 0) {
      // DES-CBC buffer encryption.
      mbedtls_des_crypt_cbc(ctx, v, 8, iv, buf, buf);
    } else {
      // 3DES-CBC buffer encryption.
      mbedtls_des3_crypt_cbc(ctx3, v, 8, iv, buf, buf);
    }
  }
}

// Clear the DES context.
  mbedtls_des_free(ctx);

// Clear the Triple-DES context.
  mbedtls_des3_free(ctx3);
}

The mbedtls_des_context is the DES context structure. It is initialized by function mbedtls_des_init. Function mbedtls_des_crypt_cbc does DES-CBC buffer encryption and decryption. Before that, the key should be set up by function mbedtls_des_setkey_enc. After the operation is finished, call function mbedtls_des_free to clear the DES context.

The mbedtls_des3_context is the Triple-DES context structure. It is initialized by function mbedtls_des3_init. There are two key-sizes supported: 112 and 168 bits. Based on the key-size, the key is set up via function mbedtls_des3_set2key_enc (or mbedtls_des3_set2key_dec) or mbedtls_des3_set3key_enc (or mbedtls_des3_set3key_dec). After that, the function mbedtls_des3_crypt_cbc does Triple-DES CBC encryption and decryption. After the operation is finished, call function mbedtls_des3_free to clear the DES3 context.

4.13 Crypto Algorithms – HASH & HMAC

The HASH and HMAC algorithms sample application demonstrates common use cases of HASH and HMAC algorithms such as SHA-1, SHA-256 and SHA-512, etc. The DA16200 SDK includes an mbedtls library. The API of the HASH and HMAC algorithms is the same with that of the mbedtls library. This section describes how the HASH and HMAC algorithms sample application is built and works.

4.13.1 How to Run

1. Open the workspace for the Crypto Algorithms for the HASH & HMAC application as follows:
   ○ .\sample\Crypto\Crypto_HASH\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.
3. In the SDK, enable the Crypto Algorithms for the Hash & HMAC application as follows:

The example application runs six types of hash algorithms and HMAC algorithms:

○ SHA1, SHA-224, SHA-256, SHA-384, SHA-512, and MD5
○ HMAC
  * SHA-1: passed
  * SHA-224: passed
  * SHA-256: passed
  * SHA-384: passed
  * SHA-512: passed
* MD5: passed
* Message-digest Information
  >>> MD5: passed
  >>> RIPEMD160: passed
  >>> SHA224: passed
  >>> SHA256: passed
  >>> SHA384: passed
  >>> SHA512: passed
* Hash with text string
  >>> MD5: passed
  >>> RIPEMD160: passed
* Hash with multiple text string
  >>> MD5: passed
  >>> RIPEMD160: passed
* HMAC with hex data
  >>> MD5: passed
  >>> RIPEMD160: passed
  >>> SHA224: passed
  >>> SHA256: passed
  >>> SHA384: passed
  >>> SHA512: passed
* HMAC with multiple hex data
  >>> MD5: passed
  >>> RIPEMD160: passed
  >>> SHA224: passed
  >>> SHA256: passed
  >>> SHA384: passed
  >>> SHA512: passed
  >>> Hash with hex data
  >>> SHA224: passed
  >>> SHA256: passed
  >>> SHA384: passed
  >>> SHA512: passed
  >>> Hash with multiple hex data
  >>> SHA224: passed
  >>> SHA256: passed
  >>> SHA384: passed
  >>> SHA512: passed

4.13.2 Application Initialization

The DA16200 SDK provides an Mbed TLS library. This library helps with an easy implementation of the user application. This example describes how the user can use hash and HMAC algorithms of the Mbed TLS library.

```c
void crypto_sample_hash(ULONG arg)
{
  crypto_sample_hash_sha1();
  crypto_sample_hash_sha224();
  crypto_sample_hash_sha256();
  crypto_sample_hash_sha384();
}
```
crypto_sample_hash_sha512();
crypto_sample_hash_md5();
crypto_sample_hash_md_wrapper();
return ;
}

4.13.3 Crypto Algorithm for SHA-1 Hash

DA16200 supports a crypto algorithm for SHA-1 hash. To explain how SHA-1 hash works, see the test vector in FIPS-180-1.

```c
int crypto_sample_hash_sha1()
{
    mbedtls_sha1_context *ctx = NULL;
    PRINTF("* SHA-1: ");

    // Initialize a SHA-1 context.
    mbedtls_sha1_init(ctx);

    // Start a SHA-1 checksum calculation.
    mbedtls_sha1_starts_ret(ctx);

    // Feed an input buffer into an ongoing SHA-1 checksum calculation.
    mbedtls_sha1_update_ret(ctx, crypto_sample_hash_sha1_buf,
                           crypto_sample_hash_sha1_buflen);

    // Finishes the SHA-1 operation, and writes the result to the output buffer.
    mbedtls_sha1_finish(ctx, sha1sum);

    // Clear a SHA-1 context.
    mbedtls_sha1_free(ctx);
}
```

The mbedtls_sha1_context is the SHA-1 context structure. Function mbedtls_sha1_init is called to initialize the context. To calculate SHA-1 Hash, three functions should be called. The details are below.

- **int mbedtls_sha1_starts_ret(mbedtls_sha1_context *ctx)**
  
  **Prototype**
  
  int mbedtls_sha1_starts_ret(mbedtls_sha1_context *ctx)
  
  **Description**
  
  This function starts a SHA-1 checksum calculation.
  
  **Parameters**
  
  ctx: The SHA-1 context to initialize. This must be initialized.
  
  **Return values**
  
  0 on success. A negative error code on failure.

- **int mbedtls_sha1_update_ret(mbedtls_sha1_context *ctx, const unsigned char *input, size_t ilen)**
  
  **Prototype**
  
  int mbedtls_sha1_update_ret(mbedtls_sha1_context *ctx, const unsigned char *input, size_t ilen)
  
  **Description**
  
  This function feeds an input buffer into an ongoing SHA-1 checksum calculation.
  
  **Parameters**
  
  ctx: The SHA-1 context. This must be initialized and have a hash operation started.
  
  input: The buffer holding the input data. This must be a readable buffer of length ilen Bytes.
  
  ilen: The length of the input data input in Bytes.
Return values 0 on success. A negative error code on failure.

- int mbedtls_sha1_finish_ret(mbedtls_sha1_context *ctx, unsigned char output[20])

Prototype int mbedtls_sha1_finish_ret(mbedtls_sha1_context *ctx, unsigned char output[20])

Description This function finishes the SHA-1 operation and writes the result to the output buffer.

Parameters ctx: The SHA-1 context to use. This must be initialized and have a hash operation started.
output: The SHA-1 checksum result. This must be a writable buffer of length 20 Bytes.

Return values 0 on success. A negative error code on failure.

4.13.4 Crypto Algorithm for SHA-224 Hash

DA16200 supports a crypto algorithm for SHA-224 hash. To explain how SHA-224 hash works, see the test vector in FIPS-180-2.

```c
int crypto_sample_hash_sha224()
{
    mbedtls_sha256_context *ctx = NULL;

    PRINTF("* SHA-224: ");

    // Initialize the SHA-224 context.
    mbedtls_sha256_init(ctx);

    // Start a SHA-224 checksum calculation.
    mbedtls_sha256_starts_ret(ctx, 1);  

    // Feeds an input buffer into an ongoing SHA-224 checksum calculation.
    mbedtls_sha256_update_ret(ctx, crypto_sample_hash_sha224_buf, 
                              crypto_sample_hash_sha224 buflen);

    // Finishes the SHA-224 operation, and writes the result to the output buffer.
    mbedtls_sha256_finish_ret(ctx, sha224sum);

    //Clear s SHA-224 context.
    mbedtls_sha256_free(ctx);
}
```

The mbedtls_sha256_context is the SHA-256 context structure. The Mbed TLS library supports SHA-224 and SHA-256 using the context. This sample describes SHA-224. Call function mbedtls_sha256_init to initialize the context. To calculate SHA-224 Hash, three functions should be called. The details are below:

- int mbedtls_sha256_starts_ret(mbedtls_sha256_context *ctx, int is224)

Prototype int mbedtls_sha256_starts_ret(mbedtls_sha256_context *ctx, int is224)

Description This function starts a SHA-224 or SHA-256 checksum calculation.

Parameters ctx: The context to use. This must be initialized.
is224: This determines which function to use. This must be either 0 for SHA-256, or 1 for SHA-224.

Return values 0 on success. A negative error code on failure.

- int mbedtls_sha256_update_ret(mbedtls_sha256_context *ctx, const unsigned char *input, size_t ilen)

Prototype int mbedtls_sha256_update_ret(mbedtls_sha256_context *ctx, const unsigned char *input, size_t ilen)
DA16200 Example Application Guide

Description
This function feeds an input buffer into an ongoing SHA-256 checksum calculation.

Parameters
ctx: The SHA-256 context. This must be initialized and have a hash operation started.
input: The buffer holding the input data. This must be a readable buffer of length ilen Bytes.
ilen: The length of the input data input in Bytes.

Return values
0 on success. A negative error code on failure.

● int mbedtls_sha256_finish_ret(mbedtls_sha256_context *ctx, unsigned char output[32])

Prototype
int mbedtls_sha256_finish_ret(mbedtls_sha256_context *ctx, unsigned char output[32])

Description
This function finishes the SHA-256 operation and writes the result to the output buffer.

Parameters
ctx: The SHA-256 context to use. This must be initialized and have a hash operation started.
output: The SHA-224 or SHA-256 checksum result. This must be a writable buffer of length 32 Bytes.

Return values
0 on success. A negative error code on failure.

4.13.5 Crypto Algorithm for SHA-256 Hash

DA16200 supports a crypto algorithm for SHA-256 hash. To explain how SHA-256 hash works, see the test vector in FIPS-180-2.

int crypto_sample_hash_sha256()
{
    mbedtls_sha256_context *ctx = NULL;
    PRINTF("* SHA-256: ");
    // Initialize the SHA-256 context.
    mbedtls_sha256_init(ctx);

    // Start a SHA-256 checksum calculation.
    mbedtls_sha256_starts_ret(ctx, 0);

    // Feeds an input buffer into an ongoing SHA-256 checksum calculation.
    mbedtls_sha256_update_ret(ctx, crypto_sample_hash_sha256_buf, crypto_sample_hash_sha256 buflen);

    // Finishes the SHA-256 operation, and writes the result to the output buffer.
    mbedtls_sha256_finish_ret(ctx, sha256sum);

    //Clear s SHA-256 context.
    mbedtls_sha256_free(ctx);
}

This example is the same as the Crypto Algorithm for SHA-224 code (see 4.13.4). When starting SHA-256 checksum calculation, the second parameter should be set to 0 for SHA-256.

4.13.6 Crypto Algorithm for SHA-384 Hash

DA16200 supports a crypto algorithm for SHA-384 hash. To explain how SHA-384 hash works, see the test vector in FIPS-180-2.

int crypto_sample_hash_sha384()
{
    mbedtls_sha512_context *ctx = NULL;
PRINTF("* SHA-384: ");

// Initialize a SHA-384 context.
mbedtls_sha512_init(ctx);

// Start a SHA-384 checksum calculation.
mbedtls_sha512_starts_ret(ctx, 1);

// Feed an input buffer into an ongoing SHA-384 checksum calculation.
mbedtls_sha512_update_ret(ctx, crypto_sample_hash_sha384_buf,
crypto_sample_hash_sha384 buflen);

// Finishes the SHA-384 operation, and writes the result to the output buffer.
mbedtls_sha512_finish_ret(ctx, sha384sum);

// Clear a SHA-384 context.
mbedtls_sha512_free(ctx);
}

The mbedtls_sha512_context is the SHA-512 context structure. Mbed TLS library supports SHA-384 and SHA-512 using the context. This example describes SHA-384. Function mbedtls_sha512_init is called to initialize the context. To calculate SHA-384 Hash, three functions should be called. The details are below.

- **int mbedtls_sha512_starts_ret(mbedtls_sha512_context *ctx, int is384)**

  **Prototype**
  int mbedtls_sha512_starts_ret(mbedtls_sha512_context *ctx, int is384)

  **Description**
  This function starts a SHA-384 or SHA-512 checksum calculation.

  **Parameters**
  ctx: The context to use. This must be initialized.
  is384: This determines which function to use. This must be either 0 for SHA-512, or 1 for SHA-384.

  **Return values**
  0 on success. A negative error code on failure.

- **int mbedtls_sha512_update_ret(mbedtls_sha512_context *ctx, const unsigned char *input, size_t ilen)**

  **Prototype**
  int mbedtls_sha512_update_ret(mbedtls_sha512_context *ctx, const unsigned char *input, size_t ilen)

  **Description**
  This function feeds an input buffer into an ongoing SHA-512 checksum calculation.

  **Parameters**
  ctx: The SHA-512 context. This must be initialized and have a hash operation started.
  input: The buffer holding the input data. This must be a readable buffer of length ilen Bytes.
  ilen: The length of the input data input in Bytes.

  **Return values**
  0 on success. A negative error code on failure.

- **int mbedtls_sha512_finish_ret(mbedtls_sha512_context *ctx, unsigned char output[64])**

  **Prototype**
  int mbedtls_sha512_finish_ret(mbedtls_sha512_context *ctx, unsigned char output[64])

  **Description**
  This function finishes the SHA-512 operation and writes the result to the output buffer.

  **Parameters**
  ctx: The SHA-512 context to use. This must be initialized and start a hash operation.
  output: The SHA-384 or SHA-512 checksum result. This must be a writable buffer of length 64 Bytes.

  **Return values**
  0 on success. A negative error code on failure.

### 4.13.7 Crypto Algorithm for SHA-512 Hash

DA16200 supports a crypto algorithm for SHA-512 hash. To explain how SHA-512 hash works, see the test vector in FIPS-180-2.
int crypto_sample_hash_sha512()
{
    mbedtls_sha512_context *ctx = NULL;

    PRINTF("* SHA-512: ");

    // Initialize a SHA-512 context.
    mbedtls_sha512_init(ctx);

    // Start a SHA-512 checksum calculation.
    mbedtls_sha512_starts_ret(ctx, 0);

    // Feed an input buffer into an ongoing SHA-512 checksum calculation.
    mbedtls_sha512_update_ret(ctx, crypto_sample_hash_sha512_buf, crypto_sample_hash_sha512_buflen);

    // Finishes the SHA-512 operation, and writes the result to the output buffer.
    mbedtls_sha512_finish(ctx, sha512sum);

    // Clear a SHA-512 context.
    mbedtls_sha512_free(ctx);
}

This sample is the same with Crypto Algorithm for SHA-384 code (see 4.13.6). When the SHA-512 checksum calculation is started, the second parameter should be set to 0 for SHA-512.

4.13.8 Crypto Algorithm for MD5 Hash

DA16200 supports a crypto algorithm for MD5 hash. To explain how MD5 hash works, see test vector in RFC1321.

int crypto_sample_hash_md5()
{
    PRINTF("* MD5: ");

    // Output = MD5(input buffer)
    mbedtls_md5_ret(crypto_sample_hash_md5_buf, crypto_sample_hash_md5_buflen, md5sum);

    return ret;
}

In this example, the MD5 hash function will be calculated by function mbedtls_md5_ret. The detail is below.

- int mbedtls_md5_ret(const unsigned char *input, size_t ilen, unsigned char output[16])

Prototype: int mbedtls_md5_ret(const unsigned char *input, size_t ilen, unsigned char output[16])

Description: Output = MD5(input buffer)

Parameters:
- Input: buffer holding the data
- Ilen: length of the input data
- Output: MD5 checksum result

Return values: 0 if successful
4.13.9 Crypto Algorithm for Hash and HMAC with the Generic Message-Digest Wrapper

The Mbed TLS library provides the generic message-digest wrapper to calculate Hash and HMAC functions. The example below shows how hash and HMAC will be calculated with the generic message-digest wrapper functions.

First, the user needs to check which message-digest could be supported by the Mbed TLS library. The sample code below shows how to get and check message-digest information.

```
int crypto_sample_hash_md_wrapper_info(char *md_name, mbedtls_md_type_t md_type, int md_size)
{
    const mbedtls_md_info_t *md_info = NULL;
    const int *md_type_ptr = NULL;

    // Get the message-digest information associated with the given digest type.
    md_info = mbedtls_md_info_from_type(md_type);

    // Get the message-digest information associated with the given digest name.
    if (md_info != mbedtls_md_info_from_string(md_name)) {
        goto cleanup;
    }

    // Extract the message-digest type from the message-digest information structure.
    if (mbedtls_md_get_type(md_info) != (mbedtls_md_type_t)md_type) {
        goto cleanup;
    }

    // Extract the message-digest size from the message-digest information structure.
    if (mbedtls_md_get_size(md_info) != (unsigned char)md_size) {
        goto cleanup;
    }

    // Extract the message-digest name from the message-digest information structure.
    if (strcmp(mbedtls_md_get_name(md_info), md_name) != 0) {
        goto cleanup;
    }

    // Find the list of digests supported by the generic digest module.
    for (md_type_ptr = mbedtls_md_list() ; *md_type_ptr != 0 ; md_type_ptr++) {
        if(*md_type_ptr == md_type) {
            found = 1;
            break;
        }
    }

    return ret;
}
```

The API details are as follows:

- **Prototype**: `const mbedtls_md_info_t* mbedtls_md_info_from_type(mbedtls_md_type_t md_type)`
- **Description**: This function returns the message-digest information associated with the given digest type.
- **Parameters**:
  - `md_type`: The type of digest to search for.
Return values  The message-digest information associated with md_type.
            NULL if the associated message-digest information is not found.

- const mbedtls_md_info_t* mbedtls_md_info_from_string(const char* md_name)

Prototype  const mbedtls_md_info_t* mbedtls_md_info_from_string(const char* md_name)
Description  This function returns the message-digest information associated with the given digest name.
Parameters  md_name: The name of the digest to search for.
Return values  The message-digest information associated with md_name.
            NULL if the associated message-digest information is not found.

- mbedtls_md_type_t mbedtls_md_get_type(const mbedtls_md_info_t* md_info)

Prototype  const mbedtls_md_info_t* mbedtls_md_info_from_string(const char* md_name)
Description  This function extracts the message-digest type from the message-digest information structure.
Parameters  md_info: The information structure of the message-digest algorithm to use.
Return values  The type of the message digest.

- unsigned char mbedtls_md_get_size(const mbedtls_md_info_t* md_info)

Prototype  const mbedtls_md_info_t* mbedtls_md_info_from_string(const char* md_name)
Description  This function extracts the message-digest size from the message-digest information structure.
Parameters  md_info: The information structure of the message-digest algorithm to use.
Return values  The size of the message-digest output in Bytes.

- const char* mbedtls_md_get_name(const mbedtls_md_info_t* md_info)

Prototype  const mbedtls_md_info_t* mbedtls_md_info_from_string(const char* md_name)
Description  This function extracts the message-digest name from the message-digest information structure.
Parameters  md_info: The information structure of the message-digest algorithm to use.
Return values  The name of the message digest.

- const int* mbedtls_md_list(void)

Prototype  const mbedtls_md_info_t* mbedtls_md_info_from_string(const char* md_name)
Description  This function returns the list of digests supported by the generic digest module.
Parameters  None.
Return values  A statically allocated array of digests. Each element in the returned list is an integer belonging to the message-digest enumeration mbedtls_md_type_t. The last entry is 0.

Second, the example code below describes how a hash function could be calculated using the generic message-digest function. In this sample, the input value is a text string type.

```c
int crypto_sample_hash_md_wrapper_text(char *md_name, char *text_src_string, char *hex_hash_string) {
    const mbedtls_md_info_t *md_info = NULL;

    // Get the message-digest information associated with the given digest name.
    md_info = mbedtls_md_info_from_string(md_name);

    /* Calculates the message-digest of a buffer,
       * with respect to a configurable message-digest algorithm in a single call.
    */
}
The text_src_string is input data to calculate the message-digest algorithm, and the hex_hash_string is the expected output.

- int mbedtls_md(const mbedtls_md_info_t* md_info, const unsigned char* input, size_t ilen, unsigned char* output)

**Prototype**

int mbedtls_md(const mbedtls_md_info_t* md_info, const unsigned char* input, size_t ilen, unsigned char* output)

**Description**

This function calculates the message-digest of a buffer, with respect to a configurable message-digest algorithm in a single call. The result is calculated as Output = message_digest(input buffer).

**Parameters**

- md_info: The information structure of the message-digest algorithm to use.
- input: The buffer holding the data.
- ilen: The length of the input data.
- output: The generic message-digest checksum result.

**Return values**

- 0 on success.
- MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.

Third, the example below is like the second example. The difference is that in this example the input values are multiple text strings.

```c
int crypto_sample_hash_md_wrapper_text_multi(char *md_name, char *text_src_string, char *hex_hash_string)
{
    const mbedtls_md_info_t *md_info = NULL;
    mbedtls_md_context_t *ctx = NULL;       //The generic message-digest context.

    /* Initialize a message-digest context without binding it
to a particular message-digest algorithm. */
    mbedtls_md_init(ctx);

    // Get the message-digest information associated with the given digest name.
    md_info = mbedtls_md_info_from_string(md_name);

    // Select the message digest algorithm to use, and allocates internal structures.
    ret = mbedtls_md_setup(ctx, md_info, 0);

    // Start a message-digest computation.
    ret = mbedtls_md_starts(ctx);

    // Feed an input buffer into an ongoing message-digest computation.
    ret = mbedtls_md_update(ctx, (const unsigned char *)text_src_string, halfway);

    // Feed an input buffer into an ongoing message-digest computation.
    ret = mbedtls_md_update(ctx, (const unsigned char *)(text_src_string + halfway), len - halfway);

    // Finish the digest operation, and writes the result to the output buffer.
    ret = mbedtls_md_finish(ctx, output);

    /* Clear the internal structure of ctx and frees any embedded internal structure,
but does not free ctx itself. */
    mbedtls_md_free(ctx);
}
```
The text_src_string is input data to calculate the message-digest algorithm, and the hex_hash_string is the expected output.

To support multiple input data, the message-digest should be set up. The details are as follows:

- **void mbedtls_md_init(mbedtls_md_context_t* ctx)**
  
  **Prototype**
  
  void mbedtls_md_init(mbedtls_md_context_t* ctx)
  
  **Description**
  
  This function initializes a message-digest context without binding to a particular message-digest algorithm.
  
  **Parameters**
  
  ctx: The context to initialize.
  
  **Return values**
  
  None

- **int mbedtls_md_setup(mbedtls_md_context_t* ctx, const mbedtls_md_info_t * md_info, int hmac)**
  
  **Prototype**
  
  int mbedtls_md_setup(mbedtls_md_context_t* ctx, const mbedtls_md_info_t * md_info, int hmac)
  
  **Description**
  
  This function selects the message digest algorithm to use and allocates internal structures.
  
  **Parameters**
  
  ctx: The context to set up.
  
  md_info: The information structure of the message-digest algorithm to use.
  
  hmac: Defines if HMAC is used. 0: HMAC is not used (saves some memory), or non-zero: HMAC is used with this context.
  
  **Return values**
  
  0 on success.
  
  MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.
  
  MBEDTLS_ERR_MD_ALLOC_FAILED on memory-allocation failure.

- **int mbedtls_md_update(mbedtls_md_context_t* ctx, const unsigned char* input, size_t ilen)**
  
  **Prototype**
  
  int mbedtls_md_update(mbedtls_md_context_t* ctx, const unsigned char* input, size_t ilen)
  
  **Description**
  
  This function feeds an input buffer into an ongoing message-digest computation.
  
  **Parameters**
  
  ctx: The generic message-digest context.
  
  input: The buffer holding the input data.
  
  ilen: The length of the input data.
  
  **Return values**
  
  0 on success.
  
  MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.

- **int mbedtls_md_finish(mbedtls_md_context_t* ctx, unsigned char* output)**
  
  **Prototype**
  
  int mbedtls_md_finish(mbedtls_md_context_t* ctx, unsigned char* output)
  
  **Description**
  
  This function finishes the digest operation and writes the result to the output buffer.
  
  **Parameters**
  
  ctx: The generic message-digest context.
  
  output: The buffer for the generic message-digest checksum result.
  
  **Return values**
  
  0 on success.
  
  MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.

- **void mbedtls_md_free(mbedtls_md_context_t* ctx)**
  
  **Prototype**
  
  void mbedtls_md_free(mbedtls_md_context_t* ctx)
  
  **Description**
  
  This function clears the internal structure of ctx and frees any embedded internal structure but does not free ctx itself.
  
  **Parameters**
  
  ctx: The generic message-digest context.
  
  **Return values**
  
  None.
Fourth, the sample code below shows how the HMAC function could be calculated using the generic message-digest wrapper function. The input value for this example is single hex data.

```c
int crypto_sample_hash_md_wrapper_hmac(char *md_name, int trunc_size, char *hex_key_string, char *hex_src_string, char *hex_hash_string)
{
    const mbedtls_md_info_t *md_info = NULL;

    // Get the message-digest information associated with the given digest name.
    md_info = mbedtls_md_info_from_string(md_name);

    // Calculate the full generic HMAC on the input buffer with the provided key.
    ret = mbedtls_md_hmac(md_info, key_str, key_len, src_str, src_len, output);
}
```

The `hex_key_string` is the HMAC secret key, the `hex_src_string` is input data, and the `hex_hash_string` is expected output. The `mbedtls_md_hmac` function will help the HMAC function for single input data.

- **int mbedtls_md_hmac(const mbedtls_md_info_t **md_info, const unsigned char *key, size_t keylen, const unsigned char *input, size_t ilen, unsigned char *output)**

  **Prototype**
  
  ```c
  int mbedtls_md_hmac(const mbedtls_md_info_t **md_info, const unsigned char *key, size_t keylen, const unsigned char *input, size_t ilen, unsigned char *output)
  ```

  **Description**
  This function calculates the full generic HMAC on the input buffer with the provided key.
  The function allocates the context, does the calculation and frees the context. The HMAC result is calculated as output = generic HMAC(hmac key, input buffer).

  **Parameters**
  - `md_info`: The information structure of the message-digest algorithm to use.
  - `key`: The HMAC secret key.
  - `keylen`: The length of the HMAC secret key in Bytes.
  - `input`: The buffer holding the input data.
  - `ilen`: The length of the input data.
  - `output`: The generic HMAC result.

  **Return values**
  - 0 on success.
  - `MBEDTLS_ERR_MD_BAD_INPUT_DATA` on parameter-verification failure.

Fifth, the example code below is like the fourth example. The difference is that in this example the input values are multiple hex data.

```c
int crypto_sample_hash_md_wrapper_hmac_multi(char *md_name, int trunc_size, char *hex_key_string, char *hex_src_string, char *hex_hash_string)
{
    const mbedtls_md_info_t *md_info = NULL;
    mbedtls_md_context_t *ctx = NULL;

    mbedtls_md_init(ctx);

    md_info = mbedtls_md_info_from_string(md_name);

    // Select the message digest algorithm to use, and allocates internal structures.
    ret = mbedtls_md_setup(ctx, md_info, 1);

    // Start a message-digest computation.
    ret = mbedtls_md_hmac_starts(ctx, key_str, key_len);

    // Feed an input buffer into an ongoing message-digest computation.
    ret = mbedtls_md_hmac_update(ctx, src_str, halfway);
}
// Feed an input buffer into an ongoing message-digest computation.
ret = mbedtls_md_hmac_update(ctx, src_str + halfway, src_len - halfway);

// Finish the digest operation, and writes the result to the output buffer.
ret = mbedtls_md_hmac_finish(ctx, output);

/* Clear the internal structure of ctx and frees any embedded internal
 * but does not free ctx itself.
 */
 mbedtls_md_free(ctx);
}

The API details are as follows:

- **int mbedtls_md_hmac_starts(mbedtls_md_context_t* ctx, const unsigned char* key, size_t keylen)**
  
  **Prototype**
  int mbedtls_md_hmac_starts(mbedtls_md_context_t* ctx, const unsigned char* key, size_t keylen)

  **Description**
  This function sets the HMAC key and prepares to authenticate a new message.

  **Parameters**
  - ctx: The message digest context containing an embedded HMAC context.
  - key: The HMAC secret key.
  - keylen: The length of the HMAC key in Bytes.

  **Return values**
  - 0 on success.
  - MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.

- **int mbedtls_md_hmac_update(mbedtls_md_context_t* ctx, const unsigned char * input, size_t ilen)**
  
  **Prototype**
  int mbedtls_md_hmac_update(mbedtls_md_context_t* ctx, const unsigned char * input, size_t ilen)

  **Description**
  This function feeds an input buffer into an ongoing HMAC computation.

  **Parameters**
  - ctx: The message digest context containing an embedded HMAC context.
  - input: The buffer holding the input data.
  - ilen: The length of the input data.

  **Return values**
  - 0 on success.
  - MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.

- **int mbedtls_md_hmac_finish(mbedtls_md_context_t* ctx, unsigned char* output)**
  
  **Prototype**
  int mbedtls_md_hmac_finish(mbedtls_md_context_t* ctx, unsigned char* output)

  **Description**
  This function finishes the HMAC operation and writes the result to the output buffer.

  **Parameters**
  - ctx: The message digest context containing an embedded HMAC context.
  - output: The generic HMAC checksum result.

  **Return values**
  - 0 on success.
  - MBEDTLS_ERR_MD_BAD_INPUT_DATA on parameter-verification failure.

Sixth, the example below describes how the hash function can be calculated with the generic message-digest function. In this example, the input value is single hex data. This example is almost the same as the second example.

```c
int crypto_sample_hash_md_wrapper_hex(char *md_name, char *hex_src_string, char *hex_hash_string)
{
    const mbedtls_md_info_t *md_info = NULL;

    // Get the message-digest information associated with the given digest name.
```
S Seventh, the example below describes how the hash function can be calculated with the generic message-digest function. In this example, the input value is multiple hex data. This example is almost the same as the third example.

```c
int crypto_sample_hash_md_wrapper_hex_multi(char *md_name, char *hex_src_string, char *hex_hash_string)
{
    const mbedtls_md_info_t *md_info = NULL;
    mbedtls_md_context_t *ctx = NULL;

    /* Initialize a message-digest context without binding it */
    mbedtls_md_init(ctx);

    /* Get the message-digest information associated with the given digest name. */
    md_info = mbedtls_md_info_from_string(md_name);

    /* Select the message digest algorithm to use, and allocates internal structures. */
    ret = mbedtls_md_setup(ctx, md_info, 0);

    /* Start a message-digest computation. */
    ret = mbedtls_md_starts(ctx);

    /* Feed an input buffer into an ongoing message-digest computation. */
    ret = mbedtls_md_update(ctx, src_str, halfway);

    /* Feed an input buffer into an ongoing message-digest computation. */
    ret = mbedtls_md_update(ctx, src_str + halfway, src_len - halfway);

    /* Finish the digest operation, and writes the result to the output buffer. */
    ret = mbedtls_md_finish(ctx, output);

    /* Clear the internal structure of ctx and frees any embedded internal structure, */
    /* but does not free ctx itself. */
    mbedtls_md_free(ctx);
}
```

4.14 Crypto Algorithms – DRBG

The Random generator sample application demonstrates common use cases of CTR-DRBG (Counter mode Deterministic Random Byte Generator) and HMAC-DRBG (HMAC Deterministic Random Byte Generator). The DA16200 SDK includes an mbedTLS library. The API of DRBG is the same as what the mbedTLS library provides. This section describes how the Random generator sample application is built and works.

4.14.1 How to Run

1. Open the workspace for the Crypto Algorithms for DRBG application as follows:
The DA16200 SDK provides an Mbed TLS library. The library helps with an easy implementation of the user application. This example describes how the user uses CTR DRBG and HMAC DRBG of the Mbed TLS library. CTR_DRBG is a standardized way of building a PRNG from a block-cipher in counter mode operation, as defined in NIST SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. The Mbed TLS implementation of CTR_DRBG uses AES-256 (default) or AES-128 as the underlying block cipher. HMAC_DRBG is based on Hash-based message authentication code.

```c
void crypto_sample_drbg(ULONG arg)
{
    crypto_sample_ctr_drbg_pr_on();
    crypto_sample_ctr_drbg_pr_off();
    crypto_sample_hmac_drbg_pr_on();
    crypto_sample_hmac_drbg_pr_off();
    return ;
}
```

4.14.3 CTR_DRBG with Prediction Resistance

This example describes how to use CTR_DRBG with prediction resistance.

```c
int crypto_sample_drbg_pr_on()
{
    mbedtls_ctr_drbg_context *ctx = NULL;    //The CTR_DRBG context structure.

    // Based on a NIST CTR_DRBG test vector (PR = True)
    PRINTF("* CTR_DRBG (PR = TRUE): ");

    // Initialize the CTR_DRBG context.
    mbedtls_ctr_drbg_init( ctx );

    ret = mbedtls_ctr_drbg_seed_entropy_len(ctx, ctr_drbg_self_test_entropy,
        (void *)entropy_source_pr, nonce_pers_pr, 16, 32);

    // Turn prediction resistance on
    mbedtls_ctr_drbg_set_predictionResistance(ctx,MBEDTLS_CTR_DRBG_PR_ON);

    // Generate random data using CTR_DRBG.
    ret = mbedtls_ctr_drbg_random(ctx, buf,MBEDTLS_CTR_DRBG_BLOCKSIZE);

    // Generate random data using CTR_DRBG.
    ret = mbedtls_ctr_drbg_random(ctx, buf,MBEDTLS_CTR_DRBG_BLOCKSIZE);

    // Clear CTR_CRBG context data.
```
The API details are as follows:

- `void mbedtls_ctr_drbg_init(mbedtls_ctr_drbg_context* ctx)`

  **Prototype**
  ```c
  void mbedtls_ctr_drbg_init(mbedtls_ctr_drbg_context* ctx)
  ```

  **Description**
  This function initializes the CTR_DRBG context and prepares it for `mbedtls_ctr_drbg_seed()` or `mbedtls_ctr_drbg_free()`.

  **Parameters**
  - `ctx`: The CTR_DRBG context to initialize.

  **Return values**
  None.

- `void mbedtls_ctr_drbg_set_prediction_resistance(mbedtls_ctr_drbg_context* ctx, int resistance)`

  **Prototype**
  ```c
  void mbedtls_ctr_drbg_set_prediction_resistance(mbedtls_ctr_drbg_context* ctx, int resistance)
  ```

  **Description**
  This function turns prediction resistance on or off. The default value is off.

  **Parameters**
  - `resistance`: MBEDTLS_CTR_DRBG_PR_ON or MBEDTLS_CTR_DRBG_PR_OFF.

  **Return values**
  None.

- `int mbedtls_ctr_drbg_random(void* p_rng, unsigned char *output, size_t output_len)`

  **Prototype**
  ```c
  int mbedtls_ctr_drbg_random(void* p_rng, unsigned char *output, size_t output_len)
  ```

  **Description**
  This function uses CTR_DRBG to generate random data.

  **Parameters**
  - `p_rng`: The CTR_DRBG context. This must be a pointer to a mbedtls_ctr_drbg_context structure.
  - `output`: The buffer to fill.
  - `output_len`: The length of the buffer.

  **Return values**
  0 on success.
  MBEDTLS_ERR_CTR_DRBG_ENTROPY_SOURCE_FAILED or MBEDTLS_ERR_CTR_DRBG_REQUEST_TOO_BIG on failure.

- `void mbedtls_ctr_drbg_free(mbedtls_ctr_drbg_context* ctx)`

  **Prototype**
  ```c
  void mbedtls_ctr_drbg_free(mbedtls_ctr_drbg_context* ctx)
  ```

  **Description**
  This function clears CTR_CRBG context data.

  **Parameters**
  - `ctx`: The CTR_DRBG context to clear.

  **Return values**
  None.

### 4.14.4 CTR_DRBG without Prediction Resistance

This example describes how to use CTR_DRBG without prediction resistance.

```c
int crypto_sample_drbg_pr_off()
{
    mbedtls_ctr_drbg_context *ctx = NULL; // The CTR_DRBG context structure.

    // Based on a NIST CTR_DRBG test vector (PR = FALSE)
    PRINTF("* CTR_DRBG (PR = FALSE): ");

    // Initialize the CTR_DRBG context.
    mbedtls_ctr_drbg_init(ctx);

    ret = mbedtls_ctr_drbg_seed_entropy_len(ctx, ctr_drbg_self_test_entropy,
            (void *) entropy_source_nopr, nonce_pers_nopr, 16, 32);
```
4.14.5 \textbf{HMAC} \textit{DRBG} with Prediction Resistance

This example describes how to use HMAC \textit{DRBG} with prediction resistance.

```c
int crypto_sample_hmac_drbg_pr_on()
{
    mbedtls_hmac_drbg_context *ctx = NULL;
    const mbedtls_md_info_t *md_info = mbedtls_md_info_from_type(MBEDTLS_MD_SHA1);
    PRINTF("* HMAC_DRBG (PR = True) :");

    // Initialize HMAC DRBG context.
    mbedtls_hmac_drbg_init(ctx);
    mbedtls_hmac_drbg_seed(ctx, md_info, drbg_test_entropy, (void *)crypto_sample_hmac_drbg_entropy_src_pr, NULL, 0);
    mbedtls_hmac_drbg_set_prediction_resistance(ctx, MBEDTLS_HMAC_DRBG_PR_ON);

    // Generate random.
    mbedtls_hmac_drbg_random(ctx, buf, CRYPTO_SAMPLE_HMAC_DRBG_OUTPUT_LEN);
    mbedtls_hmac_drbg_random(ctx, buf, CRYPTO_SAMPLE_HMAC_DRBG_OUTPUT_LEN);
    mbedtls_hmac_drbg_free(ctx);
}
```

The API details are as follows:

- \textbf{void \textit{mbedtls\_hmac\_drbg\_init}} (mbedtls\_hmac\_drbg\_context* ctx)
  
  \begin{tabular}{|l|}
    \hline
    \textbf{Prototype} \hspace{1cm} \textit{void \textit{mbedtls\_hmac\_drbg\_init}}(mbedtls\_hmac\_drbg\_context* \textit{ctx}) \\
    \textbf{Description} \hspace{1cm} HMAC \textit{DRBG} context initialization makes the context ready for \textit{mbedtls\_hmac\_drbg\_seed()}, \textit{mbedtls\_hmac\_drbg\_seed\_buf()} or \textit{mbedtls\_hmac\_drbg\_free}(). \\
    \textbf{Parameters} \hspace{1cm} ctx: HMAC \textit{DRBG} context to be initialized. \textbf{Return values} \hspace{1cm} None. \\
    \hline
  \end{tabular}
● int mbedtls_hmac_drbg_seed(mbedtls_hmac_drbg_context *ctx, const mbedtls_md_info_t *
"md_info", int(*)(void *, unsigned char *, size_t) f_entropy, void *p_entropy, const unsigned char *
"custom", size_t len)

Prototype              int mbedtls_hmac_drbg_seed(mbedtls_hmac_drbg_context *ctx, const mbedtls_md_info_t *
"md_info", int(*)(void *, unsigned char *, size_t) f_entropy, void *p_entropy, const unsigned char *
"custom", size_t len)

Description            HMAC_DRBG initial seeding Seed and setup entropy source for future reseeds.

Parameters            ctx: HMAC_DRBG context to be seeded.
                      md_info: MD algorithm to use for HMAC_DRBG.
                      f_entropy: Entropy callback (p_entropy, buffer to fill, buffer length).
                      p_entropy: Entropy context.
                      custom: Personalization data (Device specific identifiers) (Can be NULL).
                      len: Length of personalization data.

Return values          0 if successful, or MBEDTLS_ERR_MD_BAD_INPUT_DATA, or
                      MBEDTLS_ERR_MD_ALLOC_FAILED, or
                      MBEDTLS_ERR_HMAC_DRBG_ENTROPY_SOURCE_FAILED.

● void mbedtls_hmac_drbg_set_prediction_resistance(mbedtls_hmac_drbg_context *ctx, int
 resistance)

Prototype              void mbedtls_hmac_drbg_set_prediction_resistance(mbedtls_hmac_drbg_context *ctx, int
 resistance)

Description            Enable / disable prediction resistance (Default: Off)

Parameters            ctx: HMAC_DRBG context.
                      resistance: MBEDTLS_HMAC_DRBG_PR_ON or MBEDTLS_HMAC_DRBG_PR_OFF.

Return values          None.

● int mbedtls_hmac_drbg_random(void *p_rng, unsigned char *output, size_t out_len)

Prototype              int mbedtls_hmac_drbg_random(void *p_rng, unsigned char *output, size_t out_len)

Description            HMAC_DRBG generate random.

Parameters            p_rng: HMAC_DRBG context.
                      output: Buffer to fill.
                      out_len: Length of the buffer.

Return values          0 if successful, or MBEDTLS_ERR_HMAC_DRBG_ENTROPY_SOURCE_FAILED, or
                      MBEDTLS_ERR_HMAC_DRBG_REQUEST_TOO_BIG.

● void mbedtls_hmac_drbg_free(mbedtls_hmac_drbg_context *ctx)

Prototype              void mbedtls_hmac_drbg_free(mbedtls_hmac_drbg_context *ctx)

Description            Free an HMAC_DRBG context.

Parameters            ctx: HMAC_DRBG context to free.

Return values          None.

● int mbedtls_hmac_drbg_reseed(mbedtls_hmac_drbg_context *ctx, const unsigned char *
additional, size_t len)

Prototype              int mbedtls_hmac_drbg_reseed(mbedtls_hmac_drbg_context *ctx, const unsigned char *
additional, size_t len)

Description            HMAC_DRBG reseeding (extracts data from entropy source)

Parameters            ctx: HMAC_DRBG context.
                      additional: Additional data to add to state (can be NULL).
                      len: Length of additional data.

Return values          0 if successful, or MBEDTLS_ERR_HMAC_DRBG_ENTROPY_SOURCE_FAILED.
4.14.6 HMAC_DRBG Without Prediction Resistance

This example describes how to use HMAC_DRBG without prediction resistance.

```c
int crypto_sample_hmac_drbg_pr_off()
{
    mbedtls_hmac_drbg_context *ctx;
    const mbedtls_md_info_t *md_info = mbedtls_md_info_from_type(MBEDTLS_MD_SHA1);

    PRINTF("* HMAC_DRBG (PR = False) : ");

    // Initialize HMAC DRBG context.
    mbedtls_hmac_drbg_init(ctx);

    // HMAC_DRBG initial seeding Seed and setup entropy source for future reseeds.
    ret = mbedtls_hmac_drbg_seed(ctx, md_info,
                                 drbg_test_entropy,
                                 (void *)&crypto_sample_hmac_drbg_entropy_src_nopr, NULL, 0);

    // HMAC_DRBG reseeding (extracts data from entropy source)
    ret = mbedtls_hmac_drbg_reseed(ctx, NULL, 0);

    // Generate random.
    ret = mbedtls_hmac_drbg_random(ctx, buf, CRYPTO_SAMPLE_HMAC_DRBG_OUTPUT_LEN);

    // Generate random.
    ret = mbedtls_hmac_drbg_random(ctx, buf, CRYPTO_SAMPLE_HMAC_DRBG_OUTPUT_LEN);

    // Free an HMAC_DRBG context.
    mbedtls_hmac_drbg_free(ctx);
}
```

4.15 Crypto Algorithms – ECDSA

The Elliptic Curve Digital Signature Algorithm sample application demonstrates common use cases of the Elliptic Curve Digital Signature Algorithm. In cryptography, the Elliptic Curve Digital Signature Algorithm (ECDSA) offers a variant of the Digital Signature Algorithm (DSA) which uses elliptic curve cryptography. The DA16200 SDK includes an mbedTLS library. The API of ECDSA is the same as what the mbedTLS library provides. This section describes how the Elliptic Curve Digital Signature Algorithm sample application is built and works.

4.15.1 How to Run

1. Open the workspace for the Crypto Algorithms for the ECDSA application as follows:
   - .\sample\Crypto\Crypto_ECDSA\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use the ECDSA function.

* Seeding the random number generator: passed
* Generating key pair: passed - (key size: 192 bits)
* Computing message hash: passed
* Signing message hash: passed - (signature length = 56)
* Preparing verification context: passed
* Verifying signature: passed
4.15.2 Application Initialization

In cryptography, the Elliptic Curve Digital Signature Algorithm (ECDSA) offers a variant of the Digital Signature Algorithm (DSA), which uses elliptic curve cryptography. The DA16200 SDK provides an Mbed TLS library. This library helps with the easy implementation of the user application. This example describes how the user uses ECDSA (Elliptic Curve Digital Signature Algorithm) of the Mbed TLS library.

```c
void crypto_sample_ecdsa()
{
    crypto_sample_ecdsa_test();

    return ;
}
```

4.15.3 Generates ECDSA Key Pair and Verifies ECDSA Signature

This example generates an ECDSA keypair and verifies the self-computed ECDSA signature.

```c
int crypto_sample_ecdsa_test()
{
    int ret = -1;

    const char *pers = "crypto_sample_ecdsa";

    mbedtls_ecdsa_context ctx_sign;
    mbedtls_ecdsa_context ctx_verify;
    mbedtls_entropy_context entropy;
    mbedtls_ctr_drbg_context ctr_drbg;
    mbedtls_sha256_context sha256_ctx;

    // Initialize an ECDSA context.
    mbedtls_ecdsa_init(&ctx_sign);
    mbedtls_ecdsa_init(&ctx_verify);

    // Initialize the CTR_DRBG context.
    mbedtls_ctr_drbg_init(&ctr_drbg);

    // Initialize the SHA-256 context.
    mbedtls_sha256_init(&sha256_ctx);

    // Initialize the entropy context.
    mbedtls_entropy_init(&entropy);

    memset(sig, 0x00, MBEDTLS_ECDSA_MAX_LEN);
    memset(message, 0x25, 100);

    /*
     * Generate a key pair for signing
     */
    // Seed and sets up the CTR_DRBG entropy source for future reseeds.
    ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                                (const unsigned char *)pers,
                                strlen(pers));

    PRINTF("* Generating key pair: ");

    // Generate an ECDSA keypair on the given curve.
    ret = mbedtls_ecdsa_genkey(&ctx_sign, MBEDTLS_ECP_DP_SECP192R1,
                               mbedtls_ctr_drbg_random, &ctr_drbg);
```
The API details are as follows:

- **void mbedtls_ecdsa_init(mbedtls_ecdsa_context *ctx)**
  
  **Prototype**
  
  void mbedtls_ecdsa_init(mbedtls_ecdsa_context *ctx)

  **Description**
  
  This function initializes an ECDSA context.

  **Parameters**
  
  ctx: The ECDSA context to initialize. This must not be NULL.

  **Return values**
  
  None.

- **int mbedtls_ecdsa_genkey(mbedtls_ecdsa_context *ctx, mbedtls_ecp_group_id gid, int(*)(void *, unsigned char *, size_t), void *p_rng)**
  
  **Prototype**
  
  int mbedtls_ecdsa_genkey(mbedtls_ecdsa_context *ctx, mbedtls_ecp_group_id gid, int(*)(void *, unsigned char *, size_t), void *p_rng)

  **Description**
  
  This function generates an ECDSA keypair on the given curve.

  **Parameters**
  
  ctx: The ECDSA context to store the keypair in. This must be initialized.
4.16 Crypto Algorithms – Diffie-Hellman Key Exchange

The Diffie-Hellman-Merkle (DHM) key exchange sample application demonstrates common use cases of DHM key exchange on the client & server sides. The DA16200 SDK includes an mbedTLS
library. The API of DHM is the same as what the mbedTLS library provides. This section describes how the DHM key exchange sample application is built and works.

### 4.16.1 How to Run

1. Open the workspace for the Crypto Algorithms for the Diffie-Hellman Key Exchange application as follows:
   - `\sample\Crypto\Crypto_DHM\build\DA16xxx.eww`

2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use the Diffie-Hellman key exchange function.

* DHM parameter load: passed
* Diffie-Hellman full exchange: passed

### 4.16.2 Application Initialization

The DA16200 SDK provides an Mbed TLS library. This library helps with an easy implementation of the user application. This example includes two types. The first, function `crypto_sample_dhm_parse_dhm`, shows how Diffie-Hellman parameters can be loaded. The second, function `crypto_sample_dhm_do_dhm`, shows how DA16200 works for Diffie-Hellman key exchange.

```c
void crypto_sample_dhm()
{
    ret = crypto_sample_dhm_parse_dhm();

    for (idx = 0 ; crypto_sample_dhm_do_dhm_list[idx].title != NULL ; idx++) {
        ret = crypto_sample_dhm_do_dhm(
            crypto_sample_dhm_do_dhm_list[idx].title,
            crypto_sample_dhm_do_dhm_list[idx].radix_P,
            crypto_sample_dhm_do_dhm_list[idx].input_P,
            crypto_sample_dhm_do_dhm_list[idx].radix_G,
            crypto_sample_dhm_do_dhm_list[idx].input_G);
    }
}
```

### 4.16.3 Load Diffie-Hellman Parameters

This example application shows how the Diffie-Hellman parameters are loaded over the Mbed TLS library’s API.

```c
int crypto_sample_dhm_parse_dhm()
{
    mbedtls_dhm_context *dhm = NULL;    // The DHM context structure.

    // Initialize the DHM context.
    mbedtls_dhm_init(dhm);

    // Parse DHM parameters in PEM or DER format.
    ret = mbedtls_dhm_parse_dhm(dhm,
        (const unsigned char *)crypto_sample_dhm_params,
        crypto_sample_dhm_params_len);

    // Free and clear the components of a DHM context.
    mbedtls_dhm_free(dhm);
}```
The mbedtls_dhm_parse_dhm parses DHM parameters in PEM or DER format. The crypto_sample_dhm_params is already defined in this sample.

The API details are as follows:

- **void mbedtls_dhm_init(mbedtls_dhm_context *ctx)**
  
  **Prototype**
  
  void mbedtls_dhm_init(mbedtls_dhm_context *ctx)
  
  **Description**
  
  This function initializes the DHM context.
  
  **Parameters**
  
  ctx: The DHM context to initialize.
  
  **Return values**
  
  None.

- **int mbedtls_dhm_parse_dhm(mbedtls_dhm_context *dhm, const unsigned char *dhmin, size_t dhminlen)**
  
  **Prototype**
  
  int mbedtls_dhm_parse_dhm(mbedtls_dhm_context *dhm, const unsigned char *dhmin, size_t dhminlen)
  
  **Description**
  
  This function parses DHM parameters in PEM or DER format.
  
  **Parameters**
  
  dhm: The DHM context to import the DHM parameters into. This must be initialized.
  
  dhmin: The input buffer. This must be a readable buffer of length dhminlen Bytes.
  
  dhminlen: The size of the input buffer dhmin, including the terminating NULL Byte for PEM data.
  
  **Return values**
  
  0 on success. An MBEDTLS_ERR_DHM_XXX or MBEDTLS_ERR_PEM_XXX error code on failure.

- **void mbedtls_dhm_free(mbedtls_dhm_context *ctx)**
  
  **Prototype**
  
  void mbedtls_dhm_free(mbedtls_dhm_context *ctx)
  
  **Description**
  
  This function frees and clears the components of a DHM context.
  
  **Parameters**
  
  dhm: The DHM context to import the DHM parameters into. This must be initialized.
  
  dhmin: The input buffer. This must be a readable buffer of length dhminlen Bytes.
  
  dhminlen: The size of the input buffer dhmin, including the terminating NULL Byte for PEM data.
  
  **Return values**
  
  ctx: The DHM context to free and clear. This may be NULL, in which case this function is a no-op. If it is not NULL, it must point to an initialized DHM context.

### 4.16.4 How Diffie-Hellman Works

This sample application shows how Diffie-Hellman works over the API of the Mbed TLS library. Diffie-Hellman operation is normally used during TLS Handshake, ServerKeyExchange and ClientKeyExchange messages. To verify the operation, this sample simulates TLS Handshake’s ServerKeyExchange and ClientKeyExchange messages.

```c
int crypto_sample_dhm_do_dhm(char *title, int radix_P, char *input_P, int radix_G, char *input_G)
{
    mbedtls_dhm_context ctx_srv;
    mbedtls_dhm_context ctx_cli;
    rnd_pseudo_info rnd_info;

    // Initialize the DHM context.
    mbedtls_dhm_init(&ctx_srv);
    mbedtls_dhm_init(&ctx_cli);

    // Set parameters
    MBEDTLS_MPI_CHK(mbedtls_mpi_read_string(&ctx_srv.P, radix_P, input_P));
    MBEDTLS_MPI_CHK(mbedtls_mpi_read_string(&ctx_srv.G, radix_G, input_G));
```
The API details are as follows:

- **int mbedtls_dhm_make_params(mbedtls_dhm_context *ctx, int x_size, unsigned char *output, size_t *olen, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)**

  **Prototype**
  ```c
  int mbedtls_dhm_make_params(mbedtls_dhm_context *ctx, int x_size, unsigned char *output, size_t *olen, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)
  ```

  **Description**
  This function generates a DHM key pair and exports its public part together with the DHM parameters in the format used in a TLS ServerKeyExchange handshake message.

  **Parameters**
  - `ctx`: The DHM context to use. This must be initialized and have the DHM parameters set. It may or may not already have imported the peer's public key.
  - `x_size`: The private key size in Bytes.
  - `olen`: The address at which to store the number of Bytes written on success. This must not be NULL.
  - `output`: The destination buffer. This must be a writable buffer of sufficient size to hold the reduced binary presentation of the modulus, the generator and the public key, each wrapped with a 2-byte length field. It is the responsibility of the caller to ensure that enough space is available. Refer to mbedtls_mpi_size() to compute the byte-size of an MPI.
  - `f_rng`: The RNG function. Must not be NULL.
  - `p_rng`: The RNG context to be passed to `f_rng`. This may be NULL if `f_rng` does not need a context parameter.

  **Return values**
  0 on success. An MBEDTLS_ERR_DHМ_XXX error code on failure.

- **int mbedtls_dhm_read_params(mbedtls_dhm_context *ctx, unsigned char **p, const unsigned char *end)**

  **Prototype**
  ```c
  int mbedtls_dhm_read_params(mbedtls_dhm_context *ctx, unsigned char **p, const unsigned char *end)
  ```
Description
This function parses the DHM parameters in a TLS ServerKeyExchange handshake message (DHM modulus, generator, and public key).

Parameters
- `ctx`: The DHM context to use. This must be initialized.
- `p`: On input, `p` must be the start of the input buffer. On output, `p` is updated to point to the end of the data that has been read. On success, this is the first byte past the end of the ServerKeyExchange parameters. On error, this is the point at which an error has been detected, which is usually not useful except to debug failures.
- `end`: The end of the input buffer.

Return values
0 on success. An MBEDTLS_ERR_DHMS_XXX error code on failure.

Prototype
```c
int mbedtls_dhm_make_public(mbedtls_dhm_context *ctx, int x_size, unsigned char *output, size_t olen, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)
```

Description
This function creates a DHM key pair and exports the raw public key in big-endian format.

Parameters
- `ctx`: The DHM context to use. This must be initialized and have the DHM parameters set. It may or may not already have imported the peer's public key.
- `x_size`: The private key size in Bytes.
- `output`: The destination buffer. This must be a writable buffer of size `olen` Bytes.
- `olen`: The length of the destination buffer. This must be at least equal to `ctx->len` (the size of P).
- `f_rng`: The RNG function. This must not be NULL.
- `p_rng`: The RNG context to be passed to `f_rng`. This may be NULL if `f_rng` does not need a context argument.

Return values
0 on success. An MBEDTLS_ERR_DHMS_XXX error code on failure.

Prototype
```c
int mbedtls_dhm_read_public(mbedtls_dhm_context *ctx, const unsigned char *input, size_t ilen)
```

Description
This function imports the raw public value of the peer.

Parameters
- `ctx`: The DHM context to use. This must be initialized and have its DHM parameters set, for instance via `mbedtls_dhm_set_group()`. It may or may not already have generated its own private key.
- `input`: The input buffer containing the \(G^Y\) value of the peer. This must be a readable buffer of size `ilen` Bytes.
- `ilen`: The size of the input buffer in Bytes.

Return values
0 on success. An MBEDTLS_ERR_DHMS_XXX error code on failure.

Prototype
```c
int mbedtls_dhm_calc_secret(mbedtls_dhm_context *ctx, unsigned char *output, size_t output_size, size_t olen, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)
```

Description
This function derives and exports the shared secret \((G^Y)^X \mod P\).

Parameters
- `ctx`: The DHM context to use. This must be initialized and have its own private key generated and the peer's public key imported.
- `output`: The buffer to write the generated shared key to. This must be a writable buffer of size `output_size` Bytes.
- `output_size`: The size of the destination buffer. This must be at least the size of `ctx->len` (the size of P).
- `olen`: On exit, holds the actual number of Bytes written.
- `f_rng`: The RNG function, for blinding purposes. This may be NULL if blinding is not needed.
- `p_rng`: The RNG context. This may be NULL if `f_rng` does not need a context argument.
4.17 Crypto Algorithms – RSA PKCS#1

The RSA PKCS#1 sample application demonstrates common use cases of RSA PKCS#1 functions. The DA16200 SDK includes an mbedTLS library. The API of RSA PKCS#1 is the same as what an mbedTLS library provides. This section describes how the Key derivation function sample application is built and works.

4.17.1 How to Run

1. Open the workspace for the Crypto Algorithms for RSA PKCS#1 application as follows:
   o \sample\Crypto\Crypto_RSA\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use RSA PKCS#1 function.

- RSA key validation: passed
- PKCS#1 encryption : passed
- PKCS#1 decryption : passed
- PKCS#1 data sign : passed
- PKCS#1 sig. verify: passed

4.17.2 Application Initialization

The DA16200 SDK provides an Mbed TLS library. The library helps with an easy implementation of the User Application. This example shows RSA key validation, encryption, decryption, and verification of the signature. To verify the signature, a SHA-1 Hash algorithm is used.

```c
void crypto_sample_rsa(ULONG arg)
{
    crypto_sample_rsa_pkcs1();
}
```

4.17.3 How RSA PKCS#1 Works

The example application below shows how RSA PKCS#1 works over the API of the Mbed TLS library. To verify, an RSA-1024 keypair and a SHA-1 Hash algorithm are used on RSA PKCS-1 v1.5.

```c
int crypto_sample_rsa_pkcs1()
{
    mbedtls_rsa_context *rsa = NULL;        // The RSA context structure.
    unsigned char *sha1sum = NULL;

    // Initializes an RSA context.
    mbedtls_rsa_init(rsa, MBEDTLS_RSA_PKCS_V15, 0);

    PRINTF("* RSA key validation: ");
    // Check if a context contains at least an RSA public key.
    ret = mbedtls_rsa_check_pubkey(rsa);
    ret = mbedtls_rsa_check_privkey(rsa);

    PRINTF("* PKCS#1 encryption :");
```
// Add the message padding, then performs an RSA operation.
ret = mbedtls_rsa_pkcs1_encrypt(rsa, myrand, NULL,
  MBEDTLS_RSA_PUBLIC, PT_LEN, rsa_plaintext, rsa_ciphertext);

PRINTF("* PKCS#1 decryption : ");

// Perform an RSA operation, then removes the message padding.
ret = mbedtls_rsa_pkcs1_decrypt(rsa, myrand,
   NULL, MBEDTLS_RSA_PRIVATE, &len,
   rsa_ciphertext, rsa_decrypted,
   (PT_LEN * sizeof(unsigned char)));

PRINTF("* PKCS#1 data sign : ");

mbedtls_sha1_ret(rsa_plaintext, PT_LEN, sha1sum);

// Perform a private RSA operation to sign a message digest using PKCS#1.
ret = mbedtls_rsa_pkcs1_sign(rsa, myrand, NULL,
   MBEDTLS_RSA_PRIVATE, MBEDTLS_MD_SHA1, 0, sha1sum, rsa_ciphertext);

PRINTF("* PKCS#1 sig. verify: ");

// Perform a public RSA operation and checks the message digest.
ret = mbedtls_rsa_pkcs1_verify(rsa, NULL,
   NULL, MBEDTLS_RSA_PUBLIC,
   MBEDTLS_MD_SHA1, 0,
   sha1sum, rsa_ciphertext);

// Free the components of an RSA key.
mbedtls_rsa_free(rsa);

The API details are as follows:

- **void mbedtls_rsa_init(mbedtls_rsa_context *ctx, int padding, int hash_id)**
  
  **Prototype**
  void mbedtls_rsa_init(mbedtls_rsa_context *ctx, int padding, int hash_id)
  
  **Description**
  This function initializes an RSA context.
  
  **Parameters**
  ctx: The RSA context to initialize. This must not be NULL.
  padding: The padding mode to use. This must be either MBEDTLS_RSA_PKCS_V15 or MBEDTLS_RSA_PKCS_V21.
  hash_id: The hash identifier of mbedtls_md_type_t type, if padding is MBEDTLS_RSA_PKCS_V21. It is otherwise unused.
  
  **Return values**
  None.

- **int mbedtls_rsa_check_pubkey(const mbedtls_rsa_context *ctx)**
  
  **Prototype**
  int mbedtls_rsa_check_pubkey(const mbedtls_rsa_context *ctx)
  
  **Description**
  This function checks if a context contains at least an RSA public key. If the function runs successfully, it is guaranteed that enough information is present to do an RSA public key operation with mbedtls_rsa_public().
  
  **Parameters**
  ctx: The initialized RSA context to check.
  
  **Return values**
  0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

- **int mbedtls_rsa_check_privkey(const mbedtls_rsa_context *ctx)**
  
  **Prototype**
  int mbedtls_rsa_check_privkey(const mbedtls_rsa_context *ctx)
  
  **Description**
  This function checks if a context contains an RSA private key and does basic consistency checks.
  
  **Parameters**
  ctx: The initialized RSA context to check.
Return values 0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

- int mbedtls_rsa_pkcs1_encrypt(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng, int mode, size_t ilen, const unsigned char *input, unsigned char *output)

Prototype int mbedtls_rsa_pkcs1_encrypt(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng, int mode, size_t ilen, const unsigned char *input, unsigned char *output)

Description This function adds the message padding, then does an RSA operation. It is the generic wrapper to do a PKCS#1 encryption operation with the mode from the context.

Parameters ctx: The initialized RSA context to use.

f_rng: The RNG to use. It is mandatory for PKCS#1 v2.1 padding encoding, and for PKCS#1 v1.5 padding encoding when used with mode set to MBEDTLS_RSA_PUBLIC. For PKCS#1 v1.5 padding encoding and mode set to MBEDTLS_RSA_PRIVATE, it is used for blinding and should be provided in this case. See mbedtls_rsa_private() for more information.

p_rng: The RNG context to be passed to f_rng. May be NULL if f_rng is NULL or if f_rng does not need a context argument.

mode: The mode of operation. This must be either MBEDTLS_RSA_PUBLIC or MBEDTLS_RSA_PRIVATE (deprecated).

ilen: The length of the plaintext in Bytes.

input: The input data to encrypt. This must be a readable buffer of size ilen Bytes. This must not be NULL.

output: The output buffer. This must be a writable buffer of length ctx->len Bytes. For example, 256 Bytes for a 2048-bit RSA modulus.

Return values 0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

- int mbedtls_rsa_pkcs1_decrypt(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng, int mode, size_t *olen, const unsigned char *input, unsigned char *output, size_t output_max_len)

Prototype int mbedtls_rsa_pkcs1_decrypt(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng, int mode, size_t *olen, const unsigned char *input, unsigned char *output, size_t output_max_len)

Description This function does an RSA operation, then removes the message padding. It is the generic wrapper to do a PKCS#1 decryption operation with the mode from the context.

Parameters ctx: The initialized RSA context to use.

f_rng: The RNG function. If mode is MBEDTLS_RSA_PRIVATE, this is used for blinding and should be provided; see mbedtls_rsa_private() for more. If mode is MBEDTLS_RSA_PUBLIC, it is ignored.

p_rng: The RNG context to be passed to f_rng. This may be NULL if f_rng is NULL or does not need a context.

mode: The mode of operation. This must be either MBEDTLS_RSA_PRIVATE or MBEDTLS_RSA_PUBLIC (deprecated).

olen: The address at which to store the length of the plaintext. This must not be NULL.

input: The ciphertext buffer. This must be a readable buffer of length ctx->len Bytes. For example, 256 Bytes for a 2048-bit RSA modulus.

output: The buffer used to hold the plaintext. This must be a writable buffer of length output_max_len Bytes.

output_max_len: The length in Bytes of the output buffer output.

Return values 0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

- int mbedtls_rsa_pkcs1_sign(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, unsigned char *sig)

Prototype int mbedtls_rsa_pkcs1_sign(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, unsigned char *sig)
### mbedtls_rsa_pkcs1_sign

**Prototype**

```
int mbedtls_rsa_pkcs1_sign(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *,
size_t)f_rng, void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen,
const unsigned char *hash, unsigned char *sig)
```

**Description**

This function does a private RSA operation to sign a message digest with PKCS#1. It is
the generic wrapper to do a PKCS#1 signature with the mode from the context.

**Parameters**

- **ctx**: The initialized RSA context to use.
- **f_rng**: The RNG function to use. If the padding mode is PKCS#1 v2.1, this must be provided. If the padding mode is PKCS#1 v1.5 and the mode is MBEDTLS_RSA_PRIVATE, it is used for blinding and should be provided. See mbedtls_rsa_private() for more information. It is otherwise ignored.
- **p_rng**: The RNG context to be passed to f_rng. This may be NULL if f_rng is NULL or does not need a context argument.
- **mode**: The mode of operation. This must be either MBEDTLS_RSA_PRIVATE or MBEDTLS_RSA_PUBLIC (deprecated).
- **md_alg**: The message-digest algorithm used to hash the original data. Use MBEDTLS_MD_NONE for signing raw data.
- **hashlen**: The length of the message digest. This is only used if md_alg is MBEDTLS_MD_NONE.
- **hash**: The buffer holding the message digest or raw data. If md_alg is MBEDTLS_MD_NONE, this must be a readable buffer of length hashlen Bytes. If md_alg is not MBEDTLS_MD_NONE, it must be a readable buffer of length the size of the hash corresponding to md_alg.
- **sig**: The buffer to hold the signature. This must be a writable buffer of length ctx->len Bytes. For example, 256 Bytes for a 2048-bit RSA modulus.

**Return values**

0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

### mbedtls_rsa_pkcs1_verify

**Prototype**

```
int mbedtls_rsa_pkcs1_verify(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *,
size_t)f_rng, void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen,
const unsigned char *hash, const unsigned char *sig)
```

**Description**

This function does a public RSA operation and checks the message digest. This is the
generic wrapper to do PKCS#1 verification with the mode from the context.

**Parameters**

- **ctx**: The initialized RSA public key context to use.
- **f_rng**: The RNG function to use. If mode is MBEDTLS_RSA_PRIVATE, this is used for blinding and should be provided; see mbedtls_rsa_private() for more. Otherwise, it is ignored.
- **p_rng**: The RNG context to be passed to f_rng. This may be NULL if f_rng is NULL or does not need a context.
- **mode**: The mode of operation. This must be either MBEDTLS_RSA_PUBLIC or MBEDTLS_RSA_PRIVATE (deprecated).
- **md_alg**: The message-digest algorithm used to hash the original data. Use MBEDTLS_MD_NONE for signing raw data.
- **hashlen**: The length of the message digest. This is only used if md_alg is MBEDTLS_MD_NONE.
- **hash**: The buffer holding the message digest or raw data. If md_alg is MBEDTLS_MD_NONE, this must be a readable buffer of length hashlen Bytes. If md_alg is not MBEDTLS_MD_NONE, it must be a readable buffer of length the size of the hash that corresponds to md_alg.
- **sig**: The buffer holding the signature. This must be a readable buffer of length ctx->len Bytes. For example, 256 Bytes for a 2048-bit RSA modulus.

**Return values**

0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

### mbedtls_rsa_free

**Prototype**

```
void mbedtls_rsa_free(mbedtls_rsa_context *ctx)
```

**Description**

This function free the RSA context.

**Parameters**

- **ctx**: The initialized RSA context to free.

**Return values**

0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.
**DA16200 Example Application Guide**

Description: This function frees the components of an RSA key.

Parameters:
- **ctx**: The RSA context to free. May be NULL, in which case this function is a no-op. If it is not NULL, it must point to an initialized RSA context.

Return values: None.

### 4.18 Crypto Algorithms – ECDH

The Elliptic-curve Diffie-Hellman (ECDH) sample application demonstrates common use cases of Elliptic-curve Diffie-Hellman (ECDH) key exchange. It is a variant of the Diffie-Hellman protocol that uses elliptic-curve cryptography. The DA16200 SDK includes an mbedTLS library. The API of ECDH is the same as what the mbedTLS library provides. This section describes how the Elliptic-curve Diffie-Hellman sample application is built and works.

#### 4.18.1 How to Run

1. Open the workspace for the Crypto Algorithms for ECDH application as follows:
   - `\sample\Crypto\Crypto_ECDH\build\DA16xxx.eww`
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use the ECDH function.

```plaintext
>>> Using Elliptic Curve: SECP224R1
* Seeding the random number generator: passed
* Setting up client context: passed
* Setting up server context: passed
* Server reading client key and computing secret: passed
* Client reading server key and computing secret: passed
* Checking if both computed secrets are equal: passed

>>> Using Elliptic Curve: SECP256R1
* Seeding the random number generator: passed
* Setting up client context: passed
* Setting up server context: passed
* Server reading client key and computing secret: passed
* Client reading server key and computing secret: passed
* Checking if both computed secrets are equal: passed

>>> Using Elliptic Curve: SECP384R1
* Seeding the random number generator: passed
* Setting up client context: passed
* Setting up server context: passed
* Server reading client key and computing secret: passed
* Client reading server key and computing secret: passed
* Checking if both computed secrets are equal: passed

>>> Using Elliptic Curve: SECP521R1
* Seeding the random number generator: passed
* Setting up client context: passed
* Setting up server context: passed
* Server reading client key and computing secret: passed
* Client reading server key and computing secret: passed
* Checking if both computed secrets are equal: passed
```
* Checking if both computed secrets are equal: passed

```python
>>> Using Elliptic Curve: Curve25519
* Seeding the random number generator: passed
* Setting up client context: passed
* Setting up server context: passed
* Server reading client key and computing secret: passed
* Client reading server key and computing secret: passed
* Checking if both computed secrets are equal: passed
```

### 4.18.2 Application Initialization

The DA16200 SDK provides an Mbed TLS library. This library helps with the easy implementation of the User Application. This example describes how the Elliptic Curve Diffie-Hellman (ECDH) key exchange works with the use of Elliptic Curve SECP224R1, SECP256R1, SECP384R1, SECP521R1, and Curve25519.

```c
void crypto_sample_ecdh(ULONG arg)
{
    mbedtls_ecp_group_id ids[5] = {
       MBEDTLS_ECP_DP_SECP224R1,       /*!< 224-bits NIST curve */
       MBEDTLS_ECP_DP_SECP256R1,       /*!< 256-bits NIST curve */
       MBEDTLS_ECP_DP_SECP384R1,       /*!< 384-bits NIST curve */
       MBEDTLS_ECP_DP_SECP521R1,       /*!< 521-bits NIST curve */
       MBEDTLS_ECP_DP_CURVE25519       /*!< Curve25519            */
    };

    for (idx = 0 ; idx < 5 ; idx++) {
        ret = crypto_sample_ecdh_key_exchange(ids[idx]);
        if (ret) {
            break;
        }
    }
}
```

### 4.18.3 How ECDH Key Exchange Works

This sample application shows how ECDH works over the API of the Mbed TLS library. In this example, the ECDH key exchange is verified on the server and client side.

```c
int crypto_sample_ecdh_key_exchange(mbedtls_ecp_group_id id)
{
    mbedtls_ecdh_context *ctx_cli = NULL;
    mbedtls_ecdh_context *ctx_srv = NULL;
    mbedtls_entropy_context *entropy = NULL;
    mbedtls_ctr_drbg_context *ctr_drbg = NULL;

    // Initialize an ECDH context.
    mbedtls_ecdh_init(ctx_cli);
    mbedtls_ecdh_init(ctx_srv);

    // Initialize the CTR_DRBG context.
    mbedtls_ctr_drbg_init(ctr_drbg);

    // Initialize the entropy context.
    mbedtls_entropy_init(entropy);
```
PRINTF(">>> Using Elliptic Curve: ");
switch (id) {
    case MBEDTLS_ECP_DP_SECP224R1:
        PRINTF("SECP224R1\n");
        break;
    case MBEDTLS_ECP_DP_SECP256R1:
        PRINTF("SECP256R1\n");
        break;
    case MBEDTLS_ECP_DP_SECP384R1:
        PRINTF("SECP384R1\n");
        break;
    case MBEDTLS_ECP_DP_SECP521R1:
        PRINTF("SECP521R1\n");
        break;
    case MBEDTLS_ECP_DP_CURVE25519:
        PRINTF("Curve25519\n");
        break;
    default:
        PRINTF("failed - [%s] Invalid Curve selected!\n");
        goto cleanup;
}

/*
 * Initialize random number generation
 */
ret = mbedtls_ctr_drbg_seed(ctr_drbg, mbedtls_entropy_func, entropy,
                           (const unsigned char *)pers,
                           sizeof pers);

/*
 * Client: initialize context and generate keypair
 */
// Sets up an ECP group context from a standardized set of domain parameters.
ret = mbedtls_ecp_group_load((ctx_cli->grp), id);

// Generate an ECDH keypair on an elliptic curve.
ret = mbedtls_ecdh_gen_public((ctx_cli->grp), &ctx_cli->d, &ctx_cli->Q,
                              mbedtls_ctr_drbg_random, ctr_drbg);

// Export multi-precision integer (MPI) into unsigned binary data, big endian
(X coordinate of ECP point)
MBEDTLS_MPI_CHK(mbedtls_mpi_write_binary((ctx_cli->Q.X), cli_to_srv_x,
                                         buflen));

// Export multi-precision integer (MPI) into unsigned binary data, big endian
(Y coordinate of ECP point)
MBEDTLS_MPI_CHK(mbedtls_mpi_write_binary((ctx_cli->Q.Y), cli_to_srv_y,
                                         buflen));

/*
 * Server: initialize context and generate keypair
 */
// Sets up an ECP group context from a standardized set of domain parameters.
ret = mbedtls_ecp_group_load((ctx_srv->grp), id);

// Generate a public key
ret = mbedtls_ecdh_gen_public((ctx_srv->grp), &ctx_srv->d, &ctx_srv->Q,
                              mbedtls_ctr_drbg_random, ctr_drbg);
// Export multi-precision integer (MPI) into unsigned binary data, big endian
// (X coordinate of ECP point)
MBEDTLS_MPI_CHK(mbedtls_mpi_write_binary(&(ctx_srv->Q.X), srv_to_cli_x, buflen));

// Export multi-precision integer (MPI) into unsigned binary data, big endian
// (Y coordinate of ECP point)
MBEDTLS_MPI_CHK(mbedtls_mpi_write_binary(&(ctx_srv->Q.Y), srv_to_cli_y, buflen));

/*
 * Server: read peer's key and generate shared secret
 */
MBEDTLS_MPI_CHK(mbedtls_mpi_lset(&(ctx_srv->Qp.Z), 1));

// Set the Z component of the peer's public value (public key) to 1
MBEDTLS_MPI_CHK(mbedtls_mpi_lset(&((ctx_srv->Qp).Z), 1));

// Set the X component of the peer's public value based on what was passed
// from client in the buffer
MBEDTLS_MPI_CHK(mbedtls_mpi_read_binary(&(ctx_srv->Qp.X), cli_to_srv_x, buflen));

// Set the Y component of the peer's public value based on what was passed
// from client in the buffer
MBEDTLS_MPI_CHK(mbedtls_mpi_read_binary(&(ctx_srv->Qp.Y), cli_to_srv_y, buflen));

// Compute the shared secret.
ret = mbedtls_ecdh_compute_shared(&(ctx_srv->grp),
   &((ctx_srv->Qp).z), &((ctx_srv->Qp).q), &((ctx_srv->d).d),
   mbedtls_ctr_drbg_random, ctr_drbg);

/*
 * Client: read peer's key and generate shared secret
 */
MBEDTLS_MPI_CHK(mbedtls_mpi_lset(&((ctx_cli->Qp).Z), 1));

MBEDTLS_MPI_CHK(mbedtls_mpi_read_binary(&((ctx_cli->Qp).X), srv_to_cli_x, buflen));

MBEDTLS_MPI_CHK(mbedtls_mpi_read_binary(&((ctx_cli->Qp).Y), srv_to_cli_y, buflen));

MBEDTLS_MPI_CHK(mbedtls_mpi_cmp_mpi(&((ctx_cli->Qp).Z), &((ctx_srv->Qp).Z)));

/*
 * Verification: are the computed secrets equal?
 */
PRINTF("* Checking if both computed secrets are equal: ");

MBEDTLS_MPI_CHK(mbedtls_mpi_cmp_mpi(&((ctx_cli->Qp).z), &((ctx_srv->Qp).z)));

// Free ECDH context.
if (ctx_cli) mbedtls_ecdh_free(ctx_cli);
if (ctx_srv) mbedtls_ecdh_free(ctx_srv);

// Free the data in the context.
if (entropy) mbedtls_entropy_free(entropy);
// Clear CTR_CRBG context data.
if (ctr_drbg) mbedtls_ctr_drbg_free(ctr_drbg);
}

The API details are as follows:

- **void mbedtls_ecdh_init(mbedtls_ecdh_context *ctx)**
  
  **Prototype**: void mbedtls_ecdh_init(mbedtls_ecdh_context *ctx)
  
  **Description**: This function initializes an ECDH context.
  
  **Parameters**: ctx: The ECDH context to initialize. This must not be NULL.
  
  **Return values**: None.

- **int mbedtls_ecp_group_load(mbedtls_ecp_group *grp, mbedtls_ecp_group_id id)**
  
  **Prototype**: int mbedtls_ecp_group_load(mbedtls_ecp_group *grp, mbedtls_ecp_group_id id)
  
  **Description**: This function sets up an ECP group context from a standardized set of domain parameters.
  
  **Parameters**: grp: The group context to setup. This must be initialized.
  id: The identifier of the domain parameter set to load.
  
  **Return values**: 0 on success. MBEDTLS_ERR_ECP_FEATURE_UNAVAILABLE if the id does not correspond to a known group. Another negative error code on other kinds of failure.

- **int mbedtls_ecdh_gen_public(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)**
  
  **Prototype**: int mbedtls_ecdh_gen_public(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)
  
  **Description**: This function generates an ECDH keypair on an elliptic curve. This function does the first of two core computations implemented during the ECDH key exchange. The second core computation is done by mbedtls_ecdh_compute_shared().
  
  **Parameters**: grp: The ECP group to use. This must be initialized and have domain parameters loaded, for example through mbedtls_ecp_load() or mbedtls_ecp_tls_read_group().
  d: The destination MPI (private key). This must be initialized.
  Q: The destination point (public key). This must be initialized.
  f_rng: The RNG function to use. This must not be NULL.
  p_rng: The RNG context to be passed to f_rng. This may be NULL in case f_rng does not need a context argument.
  
  **Return values**: 0 on success. Another MBEDTLS_ERR_ECP_XXX or MBEDTLS_MPI_XXX error code on failure.

- **int mbedtls_ecdh_compute_shared(mbedtls_ecp_group *grp, mbedtls_mpi *z, const mbedtls_ecp_point *Q, const mbedtls_mpi *d, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)**
  
  **Prototype**: int mbedtls_ecdh_compute_shared(mbedtls_ecp_group *grp, mbedtls_mpi *z, const mbedtls_ecp_point *Q, const mbedtls_mpi *d, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)
  
  **Description**: This function computes the shared secret. This function does the second of two core computations implemented during the ECDH key exchange. The first core computation is done by mbedtls_ecdh_gen_public().
  
  **Parameters**: grp: The ECP group to use. This must be initialized and have domain parameters loaded, for example through mbedtls_ecp_load() or mbedtls_ecp_tls_read_group().
  z: The destination MPI (shared secret). This must be initialized.
  Q: The public key from another party. This must be initialized.
  d: Our secret exponent (private key). This must be initialized.
f_rng: The RNG function. This may be NULL if randomization of intermediate results during the ECP computations is not needed (discouraged). See the documentation of mbedtls_ecp_mul() for more information.

p_rng: The RNG context to be passed to f_rng. This may be NULL if f_rng is NULL or does not need a context argument.

Return values
0 on success. Another MBEDTLS_ERR_ECP_XXX or MBEDTLS_MPI_XXX error code on failure.

● void mbedtls_ecdh_free(mbedtls_ecdh_context *ctx)

Prototype
void mbedtls_ecdh_free(mbedtls_ecdh_context *ctx)

Description
This function frees a context.

Parameters
ctx: The context to free. This may be NULL, in which case this function does nothing. If it is not NULL, it must point to an initialized ECDH context.

Return values
None.

4.19 Crypto Algorithms – KDF

The Key Derivation Function (KDF) sample application demonstrates common use cases of PKCS#5 functions. The DA16200 SDK includes an mbedTLS library. The API of KDF is the same as what the mbedTLS library provides. This section describes how the Key Derivation Function sample application is built and works.

4.19.1 How to Run

1. Open the workspace for the Crypto Algorithms for KDF application as follows:
   ○ \sample\Crypto\Crypto_KDF\build\DA16xxx.eww

2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use the function kdf.

* PBKDF2 (SHA1): passed

4.19.2 Application Initialization

The DA16200 SDK provides an Mbed TLS library. This library helps with the easy implementation of the User Application. This example uses a password-based Key Derivation Function specified in PKCS#5 PBKDF2 and implemented in Mbed TLS in function mbedtls_pkcs5_pdkdf2_hmac.

```c
void crypto_sample_kdf(ULONG arg)
{
    crypto_sample_pkcs5();
}
```

4.19.3 How KDF Works

This example application shows how KDF works over the API of the Mbed TLS library. In this example, PKCS#5 PBKDF2 is used. To verify, a SHA-1 Hash algorithm is used.

```c
int crypto_sample_pkcs5()
{
    mbedtls_md_context_t *sha1_ctx = NULL;
    const mbedtls_md_info_t *info_sha1;

    // Initialize a SHA-1 context.
    mbedtls_md_init(sha1_ctx);
    mbedtls_md_info_set_hash(info_sha1, sha1_ctx, mbedtls_md_csha1);

    // Get the message-digest information associated with the given digest type.
```
info_sha1 = mbedtls_md_info_from_type(MBEDTLS_MD_SHA1);

// Select the message digest algorithm to use, and allocates internal structures.
ret = mbedtls_md_setup(sha1_ctx, info_sha1, 1);

PRINTF("** PBKDF2 (SHA1): ");

// Derive a key from a password using PBKDF2 function with HMAC
ret = mbedtls_pkcs5_pbkdf2_hmac(sha1_ctx,
    pkcs5_password, pkcs5_plen,
    pkcs5_salt, pkcs5_slen,
    pkcs5_it_cnt,
    pkcs5_key_len, key);

/* Clear the internal structure of ctx and frees any embedded internal structure,
* but does not free ctx itself. */
if (sha1_ctx) mbedtls_md_free(sha1_ctx);
}

The API details are as follows:

- int mbedtls_pkcs5_pbkdf2_hmac(mbedtls_md_context_t *ctx, const unsigned char *password, size_t plen, const unsigned char *salt, size_t slen, unsigned int iteration_count, uint32_t key_length, unsigned char *output)

**Prototype**

int mbedtls_pkcs5_pbkdf2_hmac(mbedtls_md_context_t *ctx, const unsigned char *password, size_t plen, const unsigned char *salt, size_t slen, unsigned int iteration_count, uint32_t key_length, unsigned char *output)

**Description**

PKCS#5 PBKDF2 using HMAC.

**Parameters**

- ctx: Generic HMAC context
- password: Password to use when generating a key
- plen: Length of password
- salt: Salt to use when generating a key
- slen: Length of salt
- iteration_count: Iteration count
- key_length: Length of generated key in bytes
- output: Generated key. Must be at least as big as key_length

**Return values**

0 on success, or a MBEDTLS_ERR_XXX code if verification fails

### 4.20 Crypto Algorithms – Public Key Abstraction Layer

The mbedTLS library provides the Public Key abstraction layer for confidentiality, integrity, authentication and non-repudiation based on asymmetric algorithms, with either the traditional RSA or Elliptic Curves. The Public Key abstraction layer sample application demonstrates common use cases of the APIs. This section describes how the Public Key abstraction layer sample application is built and works.

#### 4.20.1 How to Run

1. Open the workspace for the Crypto Algorithms for Public Key Abstraction Layer application as follows:
   - `\sample\Crypto\Crypto_PK\build\DA16xxx.eww`
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use function kdf.
DA16200 Example Application Guide

* PK Information
>>> RSA: passed
>>> EC: passed
>>> EC_DH: passed
>>> ECDSA: passed
* RSA Verification Test
>>> RSA verify test vector #1 (good): passed
>>> RSA verify test vector #2 (bad): passed
* Signature Verification Test
>>> ECDSA: passed
>>> EC(DSA): passed
>>> EC_DH (no): passed
>>> RSA: passed
* Decryption Test
>>> RSA decrypt test vector #1: passed
>>> RSA decrypt test vector #2: passed
RSA Alt Test: passed
* RSA Verification with option Test
>>> Verify ext RSA #2 (PKCS1 v2.1, salt_len = ANY, wrong message): passed
>>> Verify ext RSA #3 (PKCS1 v2.1, salt_len = 0, OK): passed
>>> Verify ext RSA #4 (PKCS1 v2.1, salt_len = max, OK): passed
>>> Verify ext RSA #5 (PKCS1 v2.1, wrong salt_len): passed
>>> Verify ext RSA #6 (PKCS1 v2.1, MGF1 alg != MSG hash alg): passed
>>> Verify ext RSA #7 (PKCS1 v2.1, wrong MGF1 alg != MSG hash alg): passed
>>> Verify ext RSA #8 (PKCS1 v2.1, RSASSA-PSS without options): passed
>>> Verify ext RSA #9 (PKCS1 v1.5, RSA with options): passed
>>> Verify ext RSA #10 (PKCS1 v1.5, RSA without options): passed
>>> Verify ext RSA #11 (PKCS1 v2.1, asking for ECDSA): passed
>>> Verify ext RSA #12 (PKCS1 v1.5, good): passed
* PK pair Test
>>> Check pair #1 (EC, OK): passed
>>> Check pair #2 (EC, bad): passed

4.20.2 User Thread
The user thread of the Public Key Abstraction Layer application is added as shown in the example below and is executed by the system. SAMPLE_CRYPTO_PK should be a unique name to create a thread.

```c
static const app_thread_info_t sample_apps_table[] = {
    { SAMPLE_CRYPTO_PK, crypto_sample_pk, 4096, USER_PRI_APP(1), FALSE, FALSE,
        UNDEF_PORT, RUN_ALL_MODE },
};
```

4.20.3 Application Initialization
The DA16200 SDK provides an Mbed TLS library. This library helps with an easy implementation of the User Application. This example shows how to use Public Key Abstraction Layer of the Mbed TLS library.
void crypto_sample_pk(ULONG arg)
{
    PRINTF("* PK Information\n");
    ret = crypto_sample_pk_utils(
            crypto_sample_pk_utils_list[i].type,
            crypto_sample_pk_utils_list[i].size,
            crypto_sample_pk_utils_list[i].len,
            crypto_sample_pk_utils_list[i].name);

    PRINTF("* RSA Verification Test\n");
    ret = crypto_sample_pk_rsa_verify_test_vec(
            crypto_sample_pk_rsa_verify_test_vec_list[i].title,
            crypto_sample_pk_rsa_verify_test_vec_list[i].message_hex_string,
            crypto_sample_pk_rsa_verify_test_vec_list[i].digest,
            crypto_sample_pk_rsa_verify_test_vec_list[i].mod,
            crypto_sample_pk_rsa_verify_test_vec_list[i].radix N,
            crypto_sample_pk_rsa_verify_test_vec_list[i].input_N,
            crypto_sample_pk_rsa_verify_test_vec_list[i].radix E,
            crypto_sample_pk_rsa_verify_test_vec_list[i].input_E,
            crypto_sample_pk_rsa_verify_test_vec_list[i].result_hex_str,
            crypto_sample_pk_rsa_verify_test_vec_list[i].result);

    PRINTF("* Signature Verification Test\n");
    ret = crypto_sample_pk_sign_verify(
            crypto_sample_pk_sign_verify_list[i].title,
            crypto_sample_pk_sign_verify_list[i].type,
            crypto_sample_pk_sign_verify_list[i].sign_ret,
            crypto_sample_pk_sign_verify_list[i].verify_ret);

    PRINTF("* Decryption Test\n");
    ret = crypto_sample_pk_rsa_decrypt_test_vec(
            crypto_sample_pk_rsa_decrypt_list[i].title,
            crypto_sample_pk_rsa_decrypt_list[i].cipher_hex,
            crypto_sample_pk_rsa_decrypt_list[i].mod,
            crypto_sample_pk_rsa_decrypt_list[i].radix P,
            crypto_sample_pk_rsa_decrypt_list[i].input P,
            crypto_sample_pk_rsa_decrypt_list[i].radix Q,
            crypto_sample_pk_rsa_decrypt_list[i].input Q,
            crypto_sample_pk_rsa_decrypt_list[i].radix N,
            crypto_sample_pk_rsa_decrypt_list[i].input N,
            crypto_sample_pk_rsa_decrypt_list[i].radix E,
            crypto_sample_pk_rsa_decrypt_list[i].input E,
            crypto_sample_pk_rsa_decrypt_list[i].clear_hex,
            crypto_sample_pk_rsa_decrypt_list[i].result);

    ret = crypto_sample_pk_rsa_alt();

    PRINTF("* RSA Verification with option Test\n");
    ret = crypto_sample_pk_rsa_verify_ext_test_vec(
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].title,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].message_hex_string,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].digest,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].mod,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].radix N,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].input N,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].radix E,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].input E,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].result_hex_str,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].pk_type,
            crypto_sample_pk_rsa_verify_ext_test_vec_list[i].mgf1_hash_id,
crypto_sample_pk_rsa_verify_ext_list[i].salt_len,  
crypto_sample_pk_rsa_verify_ext_list[i].result);

PRINTF("* PK pair Test\n");
ret = crypto_sample_pk_check_pair(  
crypto_sample_pk_check_pair_list[i].title,  
crypto_sample_pk_check_pair_list[i].pub_file,  
crypto_sample_pk_check_pair_list[i].prv_file,  
crypto_sample_pk_check_pair_list[i].result);
}

4.20.4 How Public Key Abstraction Layer is Used

The Mbed TLS library provides the Public Key Abstraction Layer for confidentiality, integrity,  
authentication and non-repudiation based on asymmetric algorithms, with either the traditional RSA  
or Elliptic Curves.

First, the user needs to check which public key could be supported by the Mbed TLS library. The  
example code below shows how to get and check public key information.

```c
int crypto_sample_pk_utils(mbedtls_pk_type_t type, int size, int len, char *name)
{
    mbedtls_pk_context pk;
    // Initialize a mbedtls_pk_context.
    mbedtls_pk_init(&pk);

    /* Initialize a PK context with the information given
     * and allocates the type-specific PK subcontext.
     */
    ret = mbedtls_pk_setup(&pk, mbedtls_pk_info_from_type(type));

    // Get the key type.
    if (mbedtls_pk_get_type(&pk) != type) {
    }

    // Tell if a context can do the operation given by type.
    if (!mbedtls_pk_can_do(&pk, type)) {
    }

    // Get the size in bits of the underlying key.
    if (mbedtls_pk_get_bitlen(&pk) != (unsigned)size) {
    }

    // Get the length in bytes of the underlying key.
    if (mbedtls_pk_get_len(&pk) != (unsigned)len) {
    }

    // Access the type name.
    if (((ret = strcmp(mbedtls_pk_get_name(&pk), name)) != 0) {
    }

    // Free the components of a mbedtls_pk_context.
    mbedtls_pk_free(&pk);
}
```

The API details are as follows:

- void mbedtls_pk_init(mbedtls_pk_context *ctx)

Prototype: void mbedtls_pk_init(mbedtls_pk_context *ctx)
Description
Initialize an mbedtls_pk_context (as NONE).

Parameters
ctx: The context to initialize. This must not be NULL.

Return values
None.

- int mbedtls_pk_setup(mbedtls_pk_context *ctx, const mbedtls_pk_info_t *info)

Prototype
int mbedtls_pk_setup(mbedtls_pk_context *ctx, const mbedtls_pk_info_t *info)

Description
Initialize a PK context with the information given and allocates the type-specific PK subcontext.

Parameters
ctx: Context to initialize. It must not have been set up yet (type MBEDTLS_PK_NONE).
info: Information to use.

Return values
0 on success, MBEDTLS_ERR_PK_BAD_INPUT_DATA on invalid input, MBEDTLS_ERR_PK_ALLOC_FAILED on allocation failure.

- mbedtls_pk_type_t mbedtls_pk_get_type(const mbedtls_pk_context *ctx)

Prototype
mbedtls_pk_type_t mbedtls_pk_get_type(const mbedtls_pk_context *ctx)

Description
Get the key type.

Parameters
ctx: The PK context to use. It must have been initialized.

Return values
MBEDTLS_PK_NONE for a context that has not been set up.

- int mbedtls_pk_can_do(const mbedtls_pk_context *ctx, mbedtls_pk_type_t type)

Prototype
int mbedtls_pk_can_do(const mbedtls_pk_context *ctx, mbedtls_pk_type_t type)

Description
Tell if a context can do the operation given by the type.

Parameters
ctx: The context to query. It must have been initialized.
type: The desired type.

Return values
1 if the context can do operations on the given type.
0 if the context cannot do the operations on the given type. This is always the case for a context that has been initialized but not set up, or that has been cleared with mbedtls_pk_free().

- size_t mbedtls_pk_get_bitlen(const mbedtls_pk_context *ctx)

Prototype
size_t mbedtls_pk_get_bitlen(const mbedtls_pk_context *ctx)

Description
Get the size in bits of the underlying key.

Parameters
ctx: The context to query. It must have been initialized.

Return values
Key size in bits, or 0 on error.

- static size_t mbedtls_pk_get_len(const mbedtls_pk_context *ctx)

Prototype
static size_t mbedtls_pk_get_len(const mbedtls_pk_context *ctx)

Description
Get the length in bytes of the underlying key.

Parameters
ctx: The context to query. It must have been initialized.

Return values
Key size in bits, or 0 on error.

- const char* mbedtls_pk_get_name(const mbedtls_pk_context *ctx)

Prototype
const char* mbedtls_pk_get_name(const mbedtls_pk_context *ctx)

Description
Access the type name.

Parameters
ctx: The PK context to use. It must have been initialized.

Return values
Type name on success, or "invalid PK".

- void mbedtls_pk_free(mbedtls_pk_context *ctx)

Prototype
void mbedtls_pk_free(mbedtls_pk_context *ctx)

Description
Free the components of a mbedtls_pk_context.
Parameters
cctx: The context to clear. It must have been initialized. If this is NULL, this function does nothing.

Return values
None.

Second, function crypto_sample_pk_genkey describes how to generate a public key with the given algorithms (RSA or Elliptic curves).

```c
int crypto_sample_pk_genkey(mbedtls_pk_context *pk)
{
    mbedtls_entropy_context *entropy = NULL;
    mbedtls_ctr_drbg_context *ctr_drbg = NULL;

    // Initialize the entropy context.
    mbedtls_entropy_init(entropy);

    // Initialize the CTR_DRBG context.
    mbedtls_ctr_drbg_init(ctr_drbg);

    // Seed and sets up the CTR_DRBG entropy source for future reseeds.
    mbedtls_ctr_drbg_seed(ctr_drbg, mbedtls_entropy_func, entropy, NULL, 0);

#if defined(MBEDTLS_RSA_C) && defined(MBEDTLS_GENPRIME)
    if (mbedtls_pk_get_type(pk) == MBEDTLS_PK_RSA) {
        // Generate the RSA key pair.
        ret = mbedtls_rsa_gen_key(mbedtls_pk_rsa(*pk), rnd_std_rand, ctr_drbg,
                                 RSA_KEY_SIZE, 3);
    }
#endif

#if defined(MBEDTLS_ECP_C)
    if ((mbedtls_pk_get_type(pk) == MBEDTLS_PK_ECKEY)
        || (mbedtls_pk_get_type(pk) == MBEDTLS_PK_ECKEY_DH)
        || (mbedtls_pk_get_type(pk) == MBEDTLS_PK_ECDSA)) {
        // Set a group using well-known domain parameters.
        ret = mbedtls_ecp_group_load(&mbedtls_pk_ec(*pk)->grp,
                                    MBEDTLS_ECP_DP_SECP192R1);

        // Generate key pair, wrapper for conventional base point
        ret = mbedtls_ecp_gen_keypair(&mbedtls_pk_ec(*pk)->grp, &mbedtls_pk_ec(*pk)->d,
                                      &mbedtls_pk_ec(*pk)->Q, rnd_std_rand, ctr_drbg);
    }
#endif

    mbedtls_ctr_drbg_free(ctr_drbg);
    mbedtls_entropy_free(entropy);
}
```

The API details are as follows:

- **int mbedtls_rsa_gen_key(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t) f_rng, void *p_rng, unsigned int nbits, int exponent)**

  **Prototype:**
  int mbedtls_rsa_gen_key(mbedtls_rsa_context *ctx, int(*)(void *, unsigned char *, size_t) f_rng, void *p_rng, unsigned int nbits, int exponent)

  **Description:**
  This function generates an RSA keypair.

  **Parameters:**
  ctx: The initialized RSA context used to hold the key.
  f_rng: The RNG function to be used for key generation. This must not be NULL.
p_rng: The RNG context to be passed to f_rng. This may be NULL if f_rng does not need a context.

nbits: The size of the public key in bits.

exponent: The public exponent to use. For example, 65537. This must be odd and greater than 1.

Return values 0 on success. An MBEDTLS_ERR_RSA_XXX error code on failure.

int mbedtls_ecp_gen_keypair(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)

Prototype int mbedtls_ecp_gen_keypair (mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)

Description This function generates an ECP keypair.

Parameters grp: The ECP group to generate a key pair for. This must be initialized and have group parameters set, for example through mbedtls_ecp_group_load().

d: The destination MPI (secret part). This must be initialized.

Q: The destination point (public part). This must be initialized.

f_rng: The RNG function. This must not be NULL.

p_rng: The RNG context to be passed to f_rng. This may be NULL if f_rng does not need a context argument.

Return values 0 on success. An MBEDTLS_ERR_ECP_XXX or MBEDTLS_MPI_XXX error code on failure.

Third, function crypto_sample_pk_rsa_verify_test_vec describes how an RSA signature is verified with Public Key abstraction Layer functions.

int crypto_sample_pk_rsa_verify_test_vec(char *title, char *message_hex_string, mbedtls_md_type_t digest, int mod, int radix_N, char *input_N, int radix_E, char *input_E, char *result_hex_str, int result)
{
    mbedtls_rsa_context *rsa = NULL;
    mbedtls_pk_context pk;

    // Initialize a mbedtls_pk_context.
    mbedtls_pk_init(&pk);

    /* Initialize a PK context with the information given * and allocates the type-specific PK subcontext. */
    ret = mbedtls_pk_setup(&pk, mbedtls_pk_info_from_type(MBEDTLS_PK_RSA));

    // Quick access to an RSA context inside a PK context.
    rsa = mbedtls_pk_rsa(pk);

    rsa->len = mod / 8;

    MBEDTLS_MPI_CHK(mbedtls_mpi_read_string(&rsa->N, radix_N, input_N));
    MBEDTLS_MPI_CHK(mbedtls_mpi_read_string(&rsa->E, radix_E, input_E));

    msg_len = unhexify(message_str, message_hex_string);

    unhexify(result_str, result_hex_str);

    /* Get the message-digest information associated with the given digest type.
    if (mbedtls_md_info_from_type(digest) != NULL) {
        /* Calculates the message-digest of a buffer,
         * with respect to a configurable message-digest algorithm in a single call.
         */
    */
The API details are as follows:

- **int mbedtls_pk_verify(mbedtls_pk_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hash_len, const unsigned char *sig, size_t sig_len)**

  **Prototype**

  ```c
  int mbedtls_pk_verify(mbedtls_pk_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hash_len, const unsigned char *sig, size_t sig_len)
  ```

  **Description**

  Verify signature (including padding if relevant).

  **Parameters**

  - `ctx`: The PK context to use. It must have been set up.
  - `md_alg`: Hash algorithm used.
  - `hash`: Hash of the message to sign.
  - `hash_len`: Hash length or 0.
  - `sig`: Signature to verify.
  - `sig_len`: Signature length.

  **Return values**

  - 0 on success (signature is valid).
  - MBEDTLS_ERR_PK_SIG_LEN_MISMATCH if there is a valid signature in `sig` but its length is less than `sig_len`, or a specific error code.

Fourth, function `crypto_sample_pk_sign_verify` describes to generate a key, make a signature, and to verify this with the given crypto algorithms.

```c
int crypto_sample_pk_sign_verify(char *title, mbedtls_pk_type_t type, int sign_ret, int verify_ret)
{
    mbedtls_pk_context pk;

    // Initialize a mbedtls_pk_context.
    mbedtls_pk_init(&pk);

    /* Initialize a PK context with the information given
     * and allocates the type-specific PK subcontext.
     */
    ret = mbedtls_pk_setup(&pk, mbedtls_pk_info_from_type(type));

    // Generate key pair by the type.
    ret = crypto_sample_pk_genkey(&pk);

    // Make signature, including padding if relevant and Check result with
    // expected result.
    ret = mbedtls_pk_sign(&pk, MBEDTLS_MD_SHA256, hash, 64, sig, &sig_len, rnd_std_rand, NULL);  

    // Verify signature (including padding if relevant) and Check result with
    // expected result.
    ret = mbedtls_pk_verify(&pk, MBEDTLS_MD_SHA256, hash, 64, sig, sig_len);

    // Free the components of a mbedtls_pk_context.
    mbedtls_pk_free(&pk);
}
```
The API details are as follows:

- `int mbedtls_pk_sign(mbedtls_pk_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hash_len, unsigned char *sig, size_t *sig_len, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng);`

Prototype

- `int mbedtls_pk_sign(mbedtls_pk_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hash_len, unsigned char *sig, size_t *sig_len, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng);`

Description

Make signature, including padding if relevant.

Parameters

- `ctx`: The PK context to use. Must have been set up with a private key.
- `md_alg`: Hash algorithm used (see notes).
- `hash`: Hash of the message to sign.
- `hash_len`: Hash length or 0 (see notes).
- `sig`: Place to write the signature.
- `sig_len`: Number of bytes written.
- `f_rng`: RNG function.
- `p_rng`: RNG parameter.

Return values

- `0` on success, or a specific error code.

Fifth, function `crypto_sample_pk_rsa_decrypt_test_vec` describes how RSA is decrypted using Public Key Abstraction Layer's functions. Encryption could also be used. But, this example only explains RSA decryption.

```c
int crypto_sample_pk_rsa_decrypt_test_vec(char *title, char *cipher_hex, int mod,
   int radix_P, char *input_P, int radix_Q, char *input_Q, int radix_N, char *input_N,
   int radix_E, char *input_E, char *clear_hex, int result) {
   rnd_pseudo_info *rnd_info = NULL;
   mbedtls_rsa_context *rsa = NULL;
   mbedtls_pk_context pk;

   // Initialize a mbedtls_pk_context.
   mbedtls_pk_init(&pk);

   /* Initialize a PK context with the information given
   * and allocates the type-specific PK subcontext.
   */
   ret = mbedtls_pk_setup(&pk, mbedtls_pk_info_from_type(MBEDTLS_PK_RSA));

   // Quick access to an RSA context inside a PK context.
   rsa = mbedtls_pk_rsa(pk);

   // Import a set of core parameters into an RSA context.
   ret = mbedtls_rsa_import(rsa, &N, &P, &Q, NULL, &E);

   // Retrieve the length of RSA modulus in Bytes.
   if (mbedtls_rsa_get_len(rsa) != (size_t)(mod / 8)) {
   }

   // Complete an RSA context from a set of imported core parameters.
   ret = mbedtls_rsa_complete(rsa);

   // Decrypt message (including padding if relevant).
   ret = mbedtls_pk_decrypt(&pk, cipher, cipher_len, output, &olen, (1000 *
   sizeof(unsigned char)), rnd_pseudo_rand, rnd_info);
```

Free the components of a mbedtls_pk_context.
mbedtls_pk_free(pk);
}

The API details are as follows:

- int mbedtls_rsa_import(mbedtls_rsa_context *ctx, const mbedtls_mpi *N, const mbedtls_mpi *P, const mbedtls_mpi *Q, const mbedtls_mpi *D, const mbedtls_mpi *E)

  Prototype int mbedtls_rsa_import(mbedtls_rsa_context *ctx, const mbedtls_mpi *N, const mbedtls_mpi *P, const mbedtls_mpi *Q, const mbedtls_mpi *D, const mbedtls_mpi *E)

  Description This function imports a set of core parameters into an RSA context.

  Parameters
  - ctx: The initialized RSA context to store the parameters in.
  - N: The RSA modulus. This may be NULL.
  - P: The first prime factor of N. This may be NULL.
  - Q: The second prime factor of N. This may be NULL.
  - D: The private exponent. This may be NULL.
  - E: The public exponent. This may be NULL.

  Return values
  - 0 on success. A non-zero error code on failure.

- int mbedtls_rsa_complete(mbedtls_rsa_context *ctx)

  Prototype int mbedtls_rsa_complete(mbedtls_rsa_context *ctx)

  Description This function completes an RSA context from a set of imported core parameters.

  To setup an RSA public key, precisely N and E must have been imported.

  To setup an RSA private key, sufficient information must be present for the other parameters to be derivable.

  The default implementation supports the following:
  > Derive P, Q from N, D, E.
  > Derive N, D from P, Q, E.

  Alternative implementations need not support these.

  If this function runs successfully, it guarantees that the RSA context can be used for RSA operations without the risk of failure or crash.

  Parameters
  - ctx: The initialized RSA context holding imported parameters.

  Return values
  - 0 on success. MBEDTLS_ERR_RSA_BAD_INPUT_DATA if the attempted derivations failed.

- int mbedtls_pk_decrypt(mbedtls_pk_context *ctx, const unsigned char *input, size_t ilen, unsigned char *output, size_t olen, size_t osize, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)

  Prototype int mbedtls_pk_decrypt(mbedtls_pk_context *ctx, const unsigned char *input, size_t ilen, unsigned char *output, size_t olen, size_t osize, int(*)(void *, unsigned char *, size_t)f_rng, void *p_rng)

  Description Decrypt message (including padding if relevant).

  Parameters
  - ctx: The PK context to use. It must have been set up with a private key.
  - input: Input to decrypt.
  - ilen: Input size.
  - output: Decrypted output.
  - olen: Decrypted message length.
  - osize: Size of the output buffer.
  - f_rng: RNG function.
  - p_rng: RNG parameter.

  Return values
  - 0 on success, or a specific error code.
Sixth, function `crypto_sample_pk_rsa_alt` describes how RSA ALT context progresses to make a signature and to decrypt.

```c
int crypto_sample_pk_rsa_alt()
{
    /*
     * An rsa_alt context can only do private operations (decrypt, sign).
     * Test it against the public operations (encrypt, verify) of a
     * corresponding rsa context.
    */
    mbedtls_rsa_context *raw = NULL;
    mbedtls_pk_context rsa, alt;
    mbedtls_pk_debug_item *dbg_items = NULL;

    // Initialize an RSA context.
    mbedtls_rsa_init(raw, MBEDTLS_RSA_PKCS_V15, MBEDTLS_MD_NONE);

    // Initialize a mbedtls_pk_context.
    mbedtls_pk_init(&rsa);
    mbedtls_pk_init(&alt);

    /* Initialize a PK context with the information given
     * and allocates the type-specific PK subcontext.
     */
    ret = mbedtls_pk_setup(&rsa, mbedtls_pk_info_from_type(MBEDTLS_PK_RSA));

    // Generate key pair by the type.
    ret = crypto_sample_pk_genkey(&rsa);

    // Copy the components of an RSA context.
    ret = mbedtls_rsa_copy(raw, mbedtls_pk_rsa( rsa));

    // Initialize PK RSA_ALT context
    ret = mbedtls_pk_setup_rsa_alt(&alt, (void *)raw, mbedtls_rsa_decrypt, mbedtls_rsa_sign,
                                   mbedtls_rsa_key_len);

    // Encrypt message (including padding if relevant).
    ret = mbedtls_pk_encrypt(&rsa, msg, 50, cipher, &cipher_len, 1000,
                            rnd_std_rand, NULL);

    // Decrypt message (including padding if relevant).
    ret = mbedtls_pk_decrypt(&alt, cipher, &cipher_len, test, &test_len, 1000,
                            rnd_std_rand, NULL);

    // Free the components of an RSA key.
    mbedtls_rsa_free(raw);

    // Free the components of a mbedtls_pk_context.
    mbedtls_pk_free(&rsa);
    mbedtls_pk_free(&alt);
}
```

The API details are as follows:

- `int mbedtls_pk_setup_rsa_alt(mbedtls_pk_context *ctx, void *key, mbedtls_pk_rsa_alt_decrypt_func decrypt_func, mbedtls_pk_rsa_alt_sign_func sign_func, mbedtls_pk_rsa_alt_key_len_func key_len_func)`
Prototype

`
int mbedtls_pk_setup_rsa_alt(mbedtls_pk_context *ctx, void *key,
mbedtls_pk_rsa_alt_decrypt_func decrypt_func, mbedtls_pk_rsa_alt_sign_func
sign_func, mbedtls_pk_rsa_alt_key_len_func key_len_func)
`

Description

Initialize an RSA-alt context.

Parameters

cxt: Context to initialize. It must not have been set up yet (type MBEDTLS_PK_NONE).
key: RSA key pointer.
decrypt_func: Decryption function.
sign_func: Signing function.
key_len_func: Function returning key length in bytes.

Return values

0 on success, or MBEDTLS_ERR_PK_BAD_INPUT_DATA if the context was not already
initialized as RSA_ALT.

The last example shows how to check if a public and private pair of keys matches.

```c
int crypto_sample_pk_check_pair(char *title, char *pub_file, char *prv_file, int
result)
{
    mbedtls_pk_context pub, prv, alt;

    // Initialize a mbedtls_pk_context.
    mbedtls_pk_init(&pub);
    mbedtls_pk_init(&prv);

    // Parse a public key in PEM or DER format.
    ret = mbedtls_pk_parse_public_key(&pub,
        (const unsigned char *)pub_file, (strlen(pub_file)+1));

    // Parse a private key in PEM or DER format.
    ret = mbedtls_pk_parse_key(&prv,
        (const unsigned char *)prv_file, (strlen(prv_file)+1), NULL, 0);

    // Check if a public-private pair of keys matches.
    ret = mbedtls_pk_check_pair(&pub, &prv);

    mbedtls_pk_free(&pub);
    mbedtls_pk_free(&prv)
}
```

The API details are as follows:

- **int mbedtls_pk_parse_public_key(mbedtls_pk_context *ctx, const unsigned char *key, size_t keylen)**

  Prototype

  ```c
  int mbedtls_pk_parse_public_key(mbedtls_pk_context *ctx, const unsigned char
  *key, size_t keylen)
  ```

  Description

  Parse a public key in PEM or DER format.

  Parameters

  ctx: The PK context to fill. It must have been initialized but not set up.
  key: Input buffer to parse. The buffer must contain the input exactly, with no
  extra trailing material. For PEM, the buffer must contain a null-terminated string.
  keylen: Size of key in bytes. For PEM data, this includes the terminating null
  byte, so keylen must be equal to strlen(key) + 1.

  Return values

  0 if successful, or a specific PK or PEM error code

- **int mbedtls_pk_parse_key(mbedtls_pk_context *ctx, const unsigned char *key, size_t keylen, const unsigned char *pwd, size_t pwdlen)**
Prototype: int mbedtls_pk_parse_key(mbedtls_pk_context *ctx, const unsigned char *key, size_t keylen, const unsigned char *pwd, size_t pwdlen)

Description: Parse a private key in PEM or DER format.

Parameters:
- ctx: The PK context to fill. It must have been initialized but not set up.
- key: Input buffer to parse. The buffer must contain the input exactly, with no extra trailing material. For PEM, the buffer must contain a null-terminated string.
- keylen: Size of key in bytes. For PEM data, this includes the terminating null byte, so keylen must be equal to strlen(key) + 1.
- pwd: Optional password for decryption. Pass NULL if expecting a non-encrypted key. Pass a string of pwdlen bytes if expecting an encrypted key; a non-encrypted key will also be accepted. The empty password is not supported.
- pwdlen: Size of the password in bytes. Ignored if pwd is NULL.

Return values: 0 if successful, or a specific PK or PEM error code

● int mbedtls_pk_check_pair(const mbedtls_pk_context *pub, const mbedtls_pk_context *prv)

Prototype: int mbedtls_pk_check_pair(const mbedtls_pk_context *pub, const mbedtls_pk_context *prv)

Description: Check if a public-private pair of keys matches.

Parameters:
- pub: Context holding a public key.
- prv: Context holding a private (and public) key.

Return values: 0 on success or MBEDTLS_ERR_PK_BAD_INPUT_DATA

4.21 Crypto Algorithms – Generic Cipher Wrapper

The Generic cipher wrapper sample application demonstrates common use cases of a generic cipher wrapper API of the mbedtlsTLS library that is included in the DA16200 SDK. The generic cipher wrapper API is the same as what the mbedtlsTLS library provides. This section describes how the Generic cipher wrapper sample application is built and works.

4.21.1 How to Run

1. Open the workspace for the Crypto Algorithms for Generic Cipher Wrapper application as follows:
   ○ \sample\Crypto\Crypto_Cipher\build\DA16xxx.eww
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB and reboot.

The example application explains how to use the generic cipher wrapper function.

* AES-128-ECB (enc, dec): passed
* AES-192-ECB (enc, dec): passed
* AES-256-ECB (enc, dec): passed
* AES-128-CBC (enc, dec): passed
* AES-192-CBC (enc, dec): passed
* AES-256-CBC (enc, dec): passed
* AES-128-CFB128 (enc, dec): passed
* AES-192-CFB128 (enc, dec): passed
DA16200 Example Application Guide

* AES-256-CFB128(enc, dec): passed
* AES-128-CTR(enc, dec): passed
* AES-192-CTR(enc, dec): passed
* AES-256-CTR(enc, dec): passed
* AES-128-GCM(enc, dec): passed
* AES-192-GCM(enc, dec): passed
* AES-256-GCM(enc, dec): passed
* DES-CBC(enc, dec): passed
* DES-EDE-CBC(enc, dec): passed
* DES-EDE3-CBC(enc, dec): passed
* ARC4-128(enc, dec): passed
* AES-128-CCM(enc, dec): passed
* AES-192-CCM(enc, dec): passed
* AES-256-CCM(enc, dec): passed

4.21.2 Application Initialization

The DA16200 SDK provides an Mbed TLS library. This library helps with an easy implementation of the User Application. The generic cipher wrapper contains an abstraction interface for use with the cipher primitives that the library provides. It provides a common interface to all the available cipher operations.

```c
void crypto_sample_cipher(ULONG arg)
{
    crypto_sample_cipher_wrapper();
}
```

4.21.3 How Generic Cipher Wrapper is Used

This example describes how to encrypt and decrypt with generic cipher wrapper functions.

```c
int crypto_sample_cipher_wrapper()
{
    mbedtls_cipher_info_t *cipherinfo; // Cipher information.
    mbedtls_cipher_mode_t cipher_mode; // Cipher mode.
    mbedtls_cipher_type_t cipher_type; // Supported {cipher type, cipher mode} pairs.

    mbedtls_cipher_context_t *cipher_ctx; // Generic cipher context.

    for (cipher_type = MBEDTLS_CIPHER_AES_128_ECB;
        cipher_type <= MBEDTLS_CIPHER_CAMELLIA_256_CCM;
        cipher_type++)
    {
        flag_pass = FALSE;

        mbedtls_cipher_init(cipher_ctx); // Initialize a cipher_context as NONE.

        mbedtls_cipher_info_from_type(cipher_type, cipherinfo); // Retrieve the cipher-information structure associated with the given cipher type.

        cipherinfo = (mbedtls_cipher_info_t *)mbedtls_cipher_info_from_type(cipher_type);
```
// Initialize and fills the cipher-context structure with the appropriate values.
mbedtls_cipher_setup(cipher_ctx, cipherinfo);

// Return the key length of the cipher.
cipher_keylen = mbedtls_cipher_get_key_bitlen(cipher_ctx);

// Return the mode of operation for the cipher.
cipher_mode = mbedtls_cipher_get_cipher_mode(cipher_ctx);

// Return the size of the IV or nonce of the cipher, in Bytes.
cipher_ivlen = mbedtls_cipher_get_IV_size(cipher_ctx);

// Return the block size of the given cipher.
cipher_blksiz = mbedtls_cipher_get_block_size(cipher_ctx);

// Return the name of the given cipher as a string.
cipher_name = (char *) mbedtls_cipher_get_name(cipher_ctx);
PRINTF("* %s", cipher_name);
PRINTF("(enc, ");

if (cipher_adlen == 0) { // No CCM or GCM
    // Set the key to use with the given context.
    cipher_status = mbedtls_cipher_setkey(cipher_ctx, cipher_key, cipher_keylen, MBEDTLS_ENCRYPT);

    // Set the initialization vector (IV) or nonce.
    cipher_status = mbedtls_cipher_set_iv(cipher_ctx, cipher_iv, cipher_ivlen);

    // Reset the cipher state.
    cipher_status = mbedtls_cipher_reset(cipher_ctx);

    // Encrypt or decrypt using the given cipher context.
    cipher_status = mbedtls_cipher_update(cipher_ctx, plain_in, plain_inlen, ciphertext, &ciphertext_len);

    // Finish the operation.
    cipher_status = mbedtls_cipher_finish(cipher_ctx, ciphertext, ciphertext_len);
} else {
    // Set the key to use with the given context.
    cipher_status = mbedtls_cipher_setkey(cipher_ctx, cipher_key, cipher_keylen, MBEDTLS_ENCRYPT);

    // Perform authenticated encryption (AEAD).
    cipher_status = mbedtls_cipher_auth_encrypt(cipher_ctx, cipher_iv, cipher_ivlen, cipher_ad, cipher_adlen, plain_in, plain_inlen, ciphertext, ciphertext_len, cipher_tag, cipher_taglen);
}
PRINTF("dec): ");

if (cipher_adlen == 0) { // No CCM or GCM
    // Set the key to use with the given context.
    cipher_status = mbedtls_cipher_setkey(cipher_ctx,
        cipher_key, cipher_keylen,
        MBEDTLS_DECRYPT);

    // Set the initialization vector (IV) or nonce.
    cipher_status = mbedtls_cipher_set_iv(cipher_ctx,
        cipher_iv, cipher_ivlen);

    // Reset the cipher state.
    cipher_status = mbedtls_cipher_reset(cipher_ctx);

    // Encrypt or decrypt using the given cipher context.
    cipher_status = mbedtls_cipher_update(cipher_ctx,
        ciphertext, (ciphertext_len + ciphertext_finlen),
        plain_out, &plain_outlen);

    // Finish the operation.
    cipher_status = mbedtls_cipher_finish(cipher_ctx,
        &plain_out[plain_outlen], &plain_finlen);
} else {
    // Set the key to use with the given context.
    cipher_status = mbedtls_cipher_setkey(cipher_ctx,
        cipher_key, cipher_keylen,
        MBEDTLS_DECRYPT);

    // Perform authenticated decryption (AEAD).
    cipher_status = mbedtls_cipher_auth_decrypt(cipher_ctx,
        cipher_iv, cipher_ivlen,
        cipher_ad, cipher_adlen,
        ciphertext, ciphertext_len,
        plain_out, &plain_outlen,
        cipher_tag, cipher_taglen);

    // Free and clear the cipher-specific context of ctx.
    mbedtls_cipher_free(cipher_ctx);
}

The API details are as follows:

- **void mbedtls_cipher_init(mbedtls_cipher_context_t *ctx)**
  - **Prototype**: void mbedtls_cipher_init(mbedtls_cipher_context_t *ctx)
  - **Description**: This function initializes a cipher_context as NONE.
  - **Parameters**: ctx: The context to be initialized. This must not be NULL.
  - **Return values**: None.

- **void mbedtls_cipher_free(mbedtls_cipher_context_t *ctx)**
  - **Prototype**: void mbedtls_cipher_free(mbedtls_cipher_context_t *ctx)
  - **Description**: This function frees and clears the cipher-specific context of ctx. Freeing ctx itself remains the responsibility of the caller.
  - **Parameters**: ctx: The context to be freed. If this is NULL, the function has no effect, otherwise this must point to an initialized context.
Return values  None.

● const mbedtls_cipher_info_t* mbedtls_cipher_info_from_type(const mbedtls_cipher_type_t cipher_type)

Prototype  Const mbedtls_cipher_info_t* mbedtls_cipher_info_from_type(const mbedtls_cipher_type_t cipher_type)

Description  This function retrieves the cipher-information structure associated with the given cipher type.

Parameters  cipher_type: Type of the cipher to search for.

Return values  The cipher information structure associated with the given cipher_type. NULL if the associated cipher information is not found.

● int mbedtls_cipher_setup(mbedtls_cipher_context_t *ctx, const mbedtls_cipher_info_t *cipher_info)

Prototype  int mbedtls_cipher_setup(mbedtls_cipher_context_t *ctx, const mbedtls_cipher_info_t *cipher_info)

Description  This function initializes and fills the cipher-context structure with the appropriate values. It also clears the structure.

Parameters  ctx: The context to initialize. This must be initialized.
cipher_info: The cipher to use.

Return values  0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.
MBEDTLS_ERR_CIPHER_ALLOC_FAILED if allocation of the cipher-specific context fails.

● static int mbedtls_cipher_get_key_bitlen(const mbedtls_cipher_context_t *ctx)

Prototype  static int mbedtls_cipher_get_key_bitlen(const mbedtls_cipher_context_t *ctx)

Description  This function returns the key length of the cipher.

Parameters  ctx: The context of the cipher. This must be initialized.

Return values  The key length of the cipher in bits.
MBEDTLS_KEY_LENGTH_NONE if ctx has not been initialized.

● static mbedtls_cipher_mode_t mbedtls_cipher_get_cipher_mode(const mbedtls_cipher_context_t *ctx)

Prototype  static mbedtls_cipher_mode_t mbedtls_cipher_get_cipher_mode(const mbedtls_cipher_context_t *ctx)

Description  This function returns the mode of operation for the cipher.

Parameters  ctx: The context of the cipher. This must be initialized.

Return values  The mode of operation.
MBEDTLS_MODE_NONE if ctx has not been initialized.

● static int mbedtls_cipher_get_iv_size(const mbedtls_cipher_context_t *ctx)

Prototype  static int mbedtls_cipher_get_iv_size(const mbedtls_cipher_context_t *ctx)

Description  This function returns the size of the IV or nonce of the cipher, in Bytes.

Parameters  ctx: The context of the cipher. This must be initialized.

Return values  The recommended IV size if no IV has been set.
0 for ciphers not using an IV or a nonce.
The actual size if an IV has been set.

● static unsigned int mbedtls_cipher_get_block_size(const mbedtls_cipher_context_t *ctx)

Prototype  static unsigned int mbedtls_cipher_get_block_size(const mbedtls_cipher_context_t *ctx)

Description  This function returns the block size of the cipher.
Description
This function returns the block size of the given cipher.

Parameters
ctx: The context of the cipher. This must be initialized.

Return values
The block size of the underlying cipher.
0 if ctx has not been initialized.

● static const char* mbedtls_cipher_get_name(const mbedtls_cipher_context_t *ctx)

Prototype
static const char* mbedtls_cipher_get_name(const mbedtls_cipher_context_t *ctx)

Description
This function returns the name of the given cipher as a string.

Parameters
ctx: The context of the cipher. This must be initialized.

Return values
The name of the cipher.
NULL if ctx is not initialized.

● int mbedtls_cipher_setkey(mbedtls_cipher_context_t *ctx, const unsigned char *key, int key_bitlen, const mbedtls_operation_t operation)

Prototype
int mbedtls_cipher_setkey(mbedtls_cipher_context_t *ctx, const unsigned char *key, int key_bitlen, const mbedtls_operation_t operation)

Description
This function sets the key to use with the given context.

Parameters
ctx: The generic cipher context. This must be initialized and bound to a cipher information structure.
key: The key to use. This must be a readable buffer of at least key_bitlen Bits.
key_bitlen: The key length to use, in Bits.
operation: The operation that the key will be used for: MBEDTLS_ENCRYPT or MBEDTLS_DECRYPT.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.
A cipher-specific error code on failure.

● int mbedtls_cipher_set_iv(mbedtls_cipher_context_t *ctx, const unsigned char *iv, size_t iv_len)

Prototype
int mbedtls_cipher_set_iv(mbedtls_cipher_context_t *ctx, const unsigned char *iv, size_t iv_len)

Description
This function sets the initialization vector (IV) or nonce.

Parameters
ctx: The generic cipher context. This must be initialized and bound to a cipher information structure.
iv: The IV to use, or NONCE_COUNTER for CTR-mode ciphers. This must be a readable buffer of at least iv_len Bytes.
iv_len: The IV length for ciphers with variable-size IV. This parameter is discarded by ciphers with fixed-size IV.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.

● int mbedtls_cipher_reset(mbedtls_cipher_context_t *ctx)

Prototype
int mbedtls_cipher_reset(mbedtls_cipher_context_t *ctx)

Description
This function resets the cipher state.

Parameters
ctx: The generic cipher context. This must be initialized.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.

● int mbedtls_cipher_update(mbedtls_cipher_context_t *ctx, const unsigned char *input, size_t ilen, unsigned char *output, size_t *olen)

Prototype
int mbedtls_cipher_update(mbedtls_cipher_context_t *ctx, const unsigned char *input, size_t ilen, unsigned char *output, size_t *olen)
Description
The generic cipher update function. It encrypts or decrypts using the given cipher context. Writes as many block-sized blocks of data as possible to output. Any data that cannot be written immediately is either added to the next block or flushed when mbedtls_cipher_finish() is called.

Exception: For MBEDTLS_MODE_ECB, expects a single block in size. For example, 16 Bytes for AES.

Parameters
ctx: The generic cipher context. This must be initialized and bound to a key.
input: The buffer holding the input data. This must be a readable buffer of at least ilen Bytes.
ilen: The length of the input data.
output: The buffer for the output data. This must be able to hold at least ilen + block_size. This must not be the same buffer as input.
olen: The length of the output data, to be updated with the actual number of Bytes written. This must not be NULL.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.
MBEDTLS_ERR_CIPHER_FEATURE_UNAVAILABLE on an unsupported mode for a cipher.
A cipher-specific error code on failure.

● int mbedtls_cipher_finish(mbedtls_cipher_context_t *ctx, unsigned char *output, size_t *olen)

Prototype
int mbedtls_cipher_finish(mbedtls_cipher_context_t *ctx, unsigned char *output, size_t *olen)

Description
The generic cipher finalization function.
If data still needs to be flushed from an incomplete block, the data contained in it is padded to the size of the last block and written to the output buffer.

Parameters
ctx: The generic cipher context. This must be initialized and bound to a key.
output: The buffer to write data to. This needs to be a writable buffer of at least block_size Bytes.
olen: The length of the data written to the output buffer. This may not be NULL.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.
MBEDTLS_ERR_CIPHER_FULL_BLOCK_EXPECTED on decryption expecting a full block but not receiving one.
MBEDTLS_ERR_CIPHER_INVALID_PADDING on invalid padding while decrypting.
A cipher-specific error code on failure.

● int mbedtls_cipher_auth_encrypt(mbedtls_cipher_context_t *ctx, const unsigned char *iv, size_t iv_len, const unsigned char *ad, size_t ad_len, const unsigned char *input, size_t ilen, unsigned char *output, size_t *olen, unsigned char *tag, size_t tag_len)

Prototype
int mbedtls_cipher_auth_encrypt(mbedtls_cipher_context_t *ctx, const unsigned char *iv, size_t iv_len, const unsigned char *ad, size_t ad_len, const unsigned char *input, size_t ilen, unsigned char *output, size_t *olen, unsigned char *tag, size_t tag_len)

Description
The generic authenticated encryption (AEAD) function.

Parameters
ctx: The generic cipher context. This must be initialized and bound to a key.
iv: The IV to use, or NONCE_COUNTER for CTR-mode ciphers. This must be a readable buffer of at least iv_len Bytes.
iv_len: The IV length for ciphers with variable-size IV. This parameter is discarded by ciphers with fixed-size IV.
ad: The additional data to authenticate. This must be a readable buffer of at least ad_len Bytes.
ad_len: The length of ad.
input: The buffer holding the input data. This must be a readable buffer of at least ilen Bytes.
ilen: The length of the input data.
output: The buffer for the output data. This must be able to hold at least ilen Bytes.
olen: The length of the output data, to be updated with the actual number of Bytes written. This must not be NULL.
tag: The buffer for the authentication tag. This must be a writable buffer of at least tag_len Bytes.
tag_len: The desired length of the authentication tag.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.
A cipher-specific error code on failure.

● int mbedtls_cipher_auth_decrypt(mbedtls_cipher_context_t *ctx, const unsigned char *iv, size_t iv_len, const unsigned char *ad, size_t ad_len, const unsigned char *input, size_t ilen, unsigned char *output, size_t *olen, const unsigned char *tag, size_t tag_len)

Prototype
int mbedtls_cipher_auth_decrypt(mbedtls_cipher_context_t *ctx, const unsigned char *iv, size_t iv_len, const unsigned char *ad, size_t ad_len, const unsigned char *input, size_t ilen, unsigned char *output, size_t *olen, const unsigned char *tag, size_t tag_len)

Description
The generic authenticated decryption (AEAD) function.

Parameters
ctx: The generic cipher context. This must be initialized and bound to a key.
iv: The IV to use, or NONCE_COUNTER for CTR-mode ciphers. This must be a readable buffer of at least iv_len Bytes.
iv_len: The IV length for ciphers with variable-size IV. This parameter is discarded by ciphers with fixed-size IV.
ad: The additional data to be authenticated. This must be a readable buffer of at least ad_len Bytes.
ad_len: The length of ad.
input: The buffer holding the input data. This must be a readable buffer of at least ilen Bytes.
ilen: The length of the input data.
output: The buffer for the output data. This must be able to hold at least ilen Bytes.
olen: The length of the output data, to be updated with the actual number of Bytes written. This must not be NULL.
tag: The buffer holding the authentication tag. This must be a readable buffer of at least tag_len Bytes.
tag_len: The length of the authentication tag.

Return values
0 on success.
MBEDTLS_ERR_CIPHER_BAD_INPUT_DATA on parameter-verification failure.
MBEDTLS_ERR_CIPHER_AUTH_FAILED if data is not authentic.
A cipher-specific error code on failure.
Appendix A

Mosquitto 1.4.14 License
Eclipse Distribution License 1.0
Copyright (c) 2007, Eclipse Foundation, Inc. and its licensors.
All rights reserved.
Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer
  in the documentation and/or other materials provided with the distribution.
* Neither the name of the Eclipse Foundation, Inc.
  nor the names of its contributors may be used to endorse or
  promote products derived from this software without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LiABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

UMAC GPL License
Linux kernel 3.9.0 rc3 version (backport 4.2.6-1)
# Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7</td>
<td>12-May-2020</td>
<td>Add 1. How to start</td>
</tr>
<tr>
<td>1.6</td>
<td>24-Apr-2020</td>
<td>HTTP Client and HTTP Server sections added</td>
</tr>
<tr>
<td>1.5</td>
<td>30-Mar-2020</td>
<td>Updated contents for Generic SDK v2.0.0</td>
</tr>
<tr>
<td>1.4</td>
<td>22-Nov-2019</td>
<td>Finalized for publication</td>
</tr>
<tr>
<td>1.3</td>
<td>21-Nov-2019</td>
<td>Technical review</td>
</tr>
<tr>
<td>1.2</td>
<td>10-Nov-2019</td>
<td>Editorial review</td>
</tr>
<tr>
<td>1.1</td>
<td>30-Aug-19</td>
<td>Add the test log of TCP/UDP examples</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Correct commands of setting IP and Port in TCP example.</td>
</tr>
<tr>
<td>1.0</td>
<td>03-Jul-2019</td>
<td>Preliminary DRAFT Release</td>
</tr>
</tbody>
</table>
Status Definitions

<table>
<thead>
<tr>
<th>Status</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAFT</td>
<td>The content of this document is under review and subject to formal approval, which may result in modifications or additions.</td>
</tr>
<tr>
<td>APPROVED</td>
<td>The content of this document has been approved for publication.</td>
</tr>
</tbody>
</table>

Disclaimer

Unless otherwise agreed in writing, the Dialog Semiconductor products (and any associated software) referred to in this document are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of a Dialog Semiconductor product (or associated software) can reasonably be expected to result in personal injury, death or severe property or environmental damage. Dialog Semiconductor and its suppliers accept no liability for inclusion and/or use of Dialog Semiconductor products (and any associated software) in such equipment or applications and therefore such inclusion and/or use is at the customer’s own risk.

Information in this document is believed to be accurate and reliable. However, Dialog Semiconductor does not give any representations or warranties, express or implied, as to the accuracy or completeness of such information. Dialog Semiconductor furthermore takes no responsibility whatsoever for the content in this document if provided by any information source outside of Dialog Semiconductor.

Dialog Semiconductor reserves the right to change without notice the information published in this document, including, without limitation, the specification and the design of the related semiconductor products, software and applications. Notwithstanding the foregoing, for any automotive grade version of the device, Dialog Semiconductor reserves the right to change the information published in this document, including, without limitation, the specification and the design of the related semiconductor products, software and applications, in accordance with its standard automotive change notification process.

Applications, software, and semiconductor products described in this document are for illustrative purposes only. Dialog Semiconductor makes no representation or warranty that such applications, software and semiconductor products will be suitable for the specified use without further testing or modification. Unless otherwise agreed in writing, such testing or modification is the sole responsibility of the customer and Dialog Semiconductor excludes all liability in this respect.

Nothing in this document may be construed as a license for customer to use the Dialog Semiconductor products, software and applications referred to in this document. Such license must be separately sought by customer with Dialog Semiconductor.

All use of Dialog Semiconductor products, software and applications referred to in this document is subject to Dialog Semiconductor’s Standard Terms and Conditions of Sale, available on the company website (www.dialog-semiconductor.com) unless otherwise stated.

Dialog, Dialog Semiconductor and the Dialog logo are trademarks of Dialog Semiconductor Plc or its subsidiaries. All other product or service names and marks are the property of their respective owners.

© 2020 Dialog Semiconductor. All rights reserved.

Contacting Dialog Semiconductor

United Kingdom (Headquarters)
Dialog Semiconductor (UK) LTD
Phone: +44 1793 757700

Germany
Dialog Semiconductor GmbH
Phone: +49 7021 805-0

The Netherlands
Dialog Semiconductor B.V.
Phone: +31 73 640 8822
Email: enquiry@diasemi.com

North America
Dialog Semiconductor Inc.
Phone: +1 408 845 8500

Japan
Dialog Semiconductor K. K.
Phone: +81 3 5769 5100

Taiwan
Dialog Semiconductor Taiwan
Phone: +886 281 786 222

Email: Web site: www.dialog-semiconductor.com

Hong Kong
Dialog Semiconductor Hong Kong
Phone: +852 2607 4271

Korea
Dialog Semiconductor Korea
Phone: +82 2 3469 8200

China (Shenzhen)
Dialog Semiconductor China
Phone: +86 755 2981 3669

China (Shanghai)
Dialog Semiconductor China
Phone: +86 21 5424 9058

User Manual Revision 1.7 18-May-2020
CFR0012 200 of 200 © 2020 Dialog Semiconductor