User Manual

DA16200 FreeRTOS Example Application Guide

UM-WI-055

Abstract

The DA16200 is a highly integrated ultra-low power Wi-Fi system on a chip (SoC) that allows users to develop a complete Wi-Fi solution on a single chip. This document is an SDK guide which describes the examples that are included in the SDK and is intended for developers who want to develop applications using the DA16200 SDK.

Contents

Figures

Terms and Definitions

AP	Access Point
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 Multimedia Card
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

UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol

References

[1] LwIP API. (n.d). Retrieved September 9, 2021. From Savannah: https://www.nongnu.org/lwip/2_0_x/raw_api.html

[2] UM-WI-056, DA16200 DA16600 FreeRTOS Getting Started Guide, Renesas Electronics.

1 How to Start

This document describes how to set up and run one of the examples that are included in the DA16200/600 FreeRTOS SDK. It also provides a description and details on how the example works. The example projects provide a quick and easy method to confirm the operation of specific features of the DA16200/600 before starting the development/implementation of a complete solution using the DA16200/600 FreeRTOS SDK.

The DA16200 SDK contains many examples which demonstrate how to use the features of the DA16200. The examples included are:

- **Crypto**: Examples showing how to use the cryptography and security capabilities
- **DPM**: Examples showing how to use the various DPM low-power sleep modes
- **ETC**: Examples showing how to get the current time, Access Point scan result
- **Network**: Examples showing how to use various network protocols for either a client or server application
- **Peripheral**: Examples showing how to use the peripherals such as GPIO, I2C, PWM, and so on

Before using the examples, the Eclipse development environment must be set up. See the Getting Started Guide [2] for details on setting up Eclipse and importing the DA16200 SDK into that environment.

Once the environment is set up, the examples can be found in the `apps/common/examples` directory. Each example directory has a similar structure and contains its own projects, one for da16200 and one for da16600, which can be imported into the Eclipse environment.

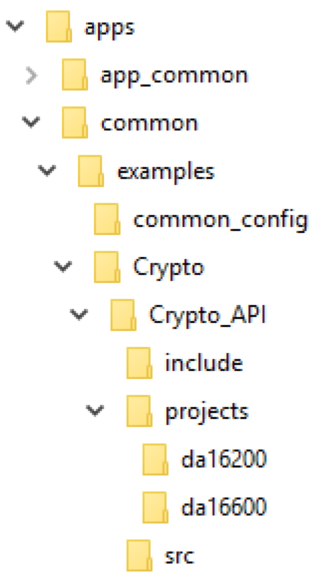


Figure 1: DA16200 SDK Example

To import an example project, follow the importing project into Eclipse method described in the Getting Started Guide [2], and select the desired example project folder instead of selecting the apps/da16200/get_started project.

For example, the **Crypto_API** example project is located here:

- *~/SDK/Apps/common/examples/Crypto/Crypto_API/projects/da16200*

When imported, the project can be built by right-clicking on the project in Eclipse and selecting **Build Project**.

1.1 Startup Process

To analyze the flow of sample code, the user can start from the **user_main()** function. See Figure 2.

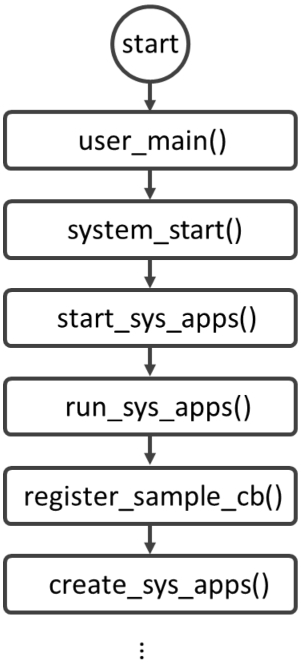


Figure 2: The Flow Chart of the Startup Process

- **user_main()**

  After the DA16200 boots, the system library invokes the **user_main()** function.
  [~/SDK/apps/common/examples/Sample_Category/Sample_XXX/src/user_main.c]

- **system_start()**

  This function configures the H/W and S/W features and initializes the Wi-Fi function in **wlaninit().** Then, it invokes **start_sys_apps()**

  [~/SDK/apps/common/examples/Sample_Category/Sample_XXX/src/system_start.c]

- **Start_sys_apps()**

  This function invokes **run_sys_apps()**

  [~/core/system/src/common/main/sys_apps.c]

- **Run_sys_apps()**

  This function invokes **register_sample_cb()** function to run sample application and **create_sys_apps()** to create network independent application.

  [~/core/system/src/common/main/sys_apps.c]

- **Register_sample_cb()**

  This is a function pointer to register **create_sample_apps()** function.

  [~/core/system/src/common/main/sys_apps.c]

- **Create_sys_apps()**

  This function creates a system application defined **in sys_apps_tables** and sample application defined in **sample_apps_table**.

  [~/core/system/src/common/main/sys_apps.c]

All the sample applications are registered in **sample_apps_table** as shown below:

[~/SDK/apps/common/examples/Sample_Category/Sample_XXX/src/sample_apps.c]

For example, TCP Client Sample:

```
static const app_task_info_t     sample_apps_table[] =
{
   /******* Start regist thread ******************************/
   ////// For testing sample code ... /////////////////////////
   { SAMPLE_TCP_CLI, tcp_client_sample, 1024,
        (tskIDLE_PRIORITY + 7),
     TRUE, FALSE, TCP_CLI_TEST_PORT, RUN_ALL_MODE },


   /******* End of List*************************************/
   { NULL,        NULL,   0, 0, FALSE, FALSE, UNDEF_PORT, 0}
};
```

1.2 Pre-configure to Start Sample Code

Each example using the Wi-Fi communication interface contains default configuration information. This information can be modified in the example code in the following location:

[~/SDK/apps/common/examples/common_config/sample_preconfig.c]

NOTE

If a user does not add the pre-configured code in this file, each sample code starts with an already saved Wi-Fi profile and other saved NVRAM environment variables.

```
/* Sample for Customer's Wi-Fi configuration */
#define       SAMPLE_AP_SSID              "TEST_AP_SSID"
#define       SAMPLE_AP_PSK               "12345678"

// CC_VAL_AUTH_OPEN, CC_VAL_AUTH_WEP, CC_VAL_AUTH_WPA, CC_VAL_AUTH_WPA2,
CC_VAL_AUTH_WPA_AUTO
#define SAMPLE_AP_AUTH_TYPE      CC_VAL_AUTH_WPA_AUTO

/* Required when WEP security mode */
#define SAMPLE_AP_WEP_INDEX      0

// CC_VAL_ENC_TKIP, CC_VAL_ENC_CCMP, CC_VAL_ENC_AUTO
#define SAMPLE_AP_ENCRPT_INDEX   CC_VAL_ENC_AUTO

void sample_preconfig(void)
{
    //
    // Need to change as Customer's profile information
    //

#if 0   // Example ...(Customer's code to config Wi-Fi profile for sample code)
    char reply[32];

    // Delete existed Wi-Fi profile
    da16x_cli_reply("remove_network 0", NULL, reply);

    // Set new Wi-Fi profile for sample test
    da16x_set_nvcache_int(DA16X_CONF_INT_MODE, 0);
    da16x_set_nvcache_str(DA16X_CONF_STR_SSID_0, SAMPLE_AP_SSID);
    da16x_set_nvcache_int(DA16X_CONF_INT_AUTH_MODE_0, SAMPLE_AP_AUTH_TYPE);
```



```
    if (SAMPLE_AP_AUTH_TYPE == CC_VAL_AUTH_WEP)
    {
        da16x_set_nvcache_str(DA16X_CONF_STR_WEP_KEY0 + SAMPLE_AP_WEP_INDEX,
SAMPLE_AP_PSK);
        da16x_set_nvcache_int(DA16X_CONF_INT_WEP_KEY_INDEX, SAMPLE_AP_WEP_INDEX);
    }
    else if (SAMPLE_AP_AUTH_TYPE > CC_VAL_AUTH_WEP)
    {
        da16x_set_nvcache_str(DA16X_CONF_STR_PSK_0, SAMPLE_AP_PSK);
        da16x_set_nvcache_int(DA16X_CONF_INT_ENCRYPTION_0, SAMPLE_AP_ENCRPT_INDEX);
    }

    // Save new Wi-Fi profile to NVRAM area
    da16x_nvcache2flash();

    vTaskDelay(10);

    // Enable new sample Wi-Fi profile
    da16x_cli_reply("select_network 0", NULL, reply);

#endif  // 0
}
```

2 Network Examples: Socket Communication

This section describes how to develop a TCP or UDP socket applications using the lwIP (Lightweight IP) APIs in the DA16200 SDK. As a companion document, see the LwIP API web page [1] for more details on all functions. To help with the understanding and implementation of applications using the DPM API, both non-DPM and DPM version of the example are provided. Before testing these examples, a test environment as shown in Figure 3 is required.

2.1 Test Environment for Socket Examples

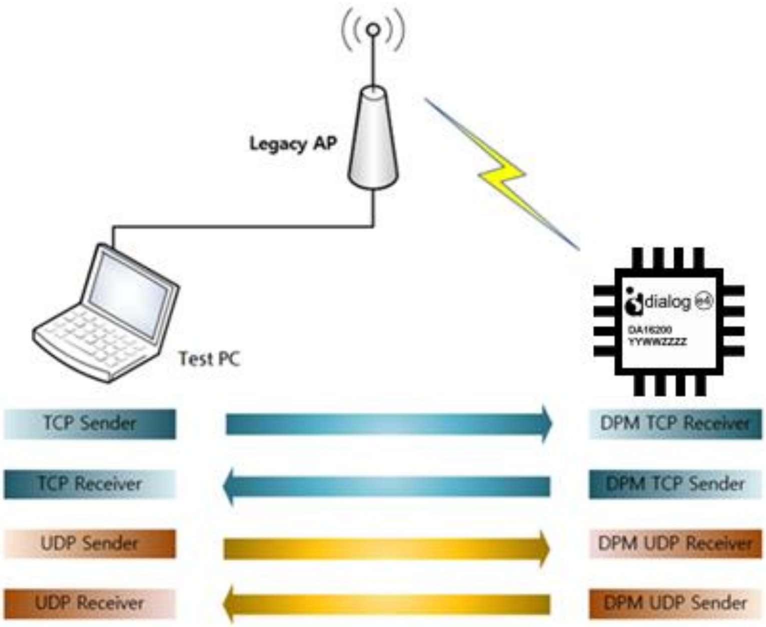


Figure 3: Overall Test Setup

2.1.1 DA16200

The example source files are included in the DA16200 SDK. The examples in this section require the DA16200 to be configured as a Wi-Fi station (STA Mode). See Section 4.6.1 of Getting Started Guide [2] for details on how to set up Wi-Fi Station mode. Also, after the STA mode setup is complete, make a note of the IP address of the DA16200 EVB to be use in the examples. The IP address is printed after connecting to an AP and then TCP/UDP example application is executed. See Figure 4.

```
Connection COMPLETE to 70:3a:cb:25:f5:f8

-- DHCP Client WLAN0: SEL(6)
-- DHCP Client WLAN0: REQ(1)
-- DHCP Client WLAN0: CHK(8)
-- DHCP Client WLAN0: BOUND(10)
         Assigned addr   : 192.168.86.38
               netmask   : 255.255.255.0
               gateway   : 192.168.86.1
               DNS addr  : 192.168.86.1

         DHCP Server IP  : 192.168.86.1
         Lease Time      : 24h 00m 00s
         Renewal Time    : 12h 00m 00s
```

Figure 4: DA16200 EVB – AP Connection Done

2.1.2 Peer Application

The examples in this section require a peer device (PC/Laptop) connected to the same Access Point running a TCP/UDP test application such as IO Ninja.

NOTE

For the Windows OS system, the user needs to install a proper application (for example, Packet Sender, Hercules, IO Ninja, and so on).

For a Linux system, proper test utilities are needed, or a test sample application is needed.

2.1.2.1 Example of Peer Application (for Windows)

This section describes how to run the peer application on an MS Windows® operating system.

1. Start the IO Ninja utility on the test PC.
   If it is not installed, you can get it from http://ioninja.com.
2. Select **File** > **New Session** for the test.

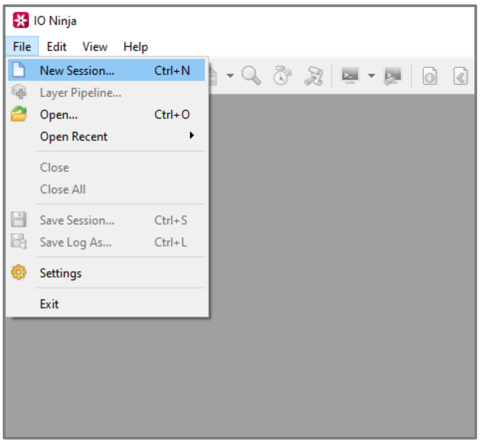


Figure 5: Start IO Ninja Utility

3. To test the TCP Client, start the TCP Server.

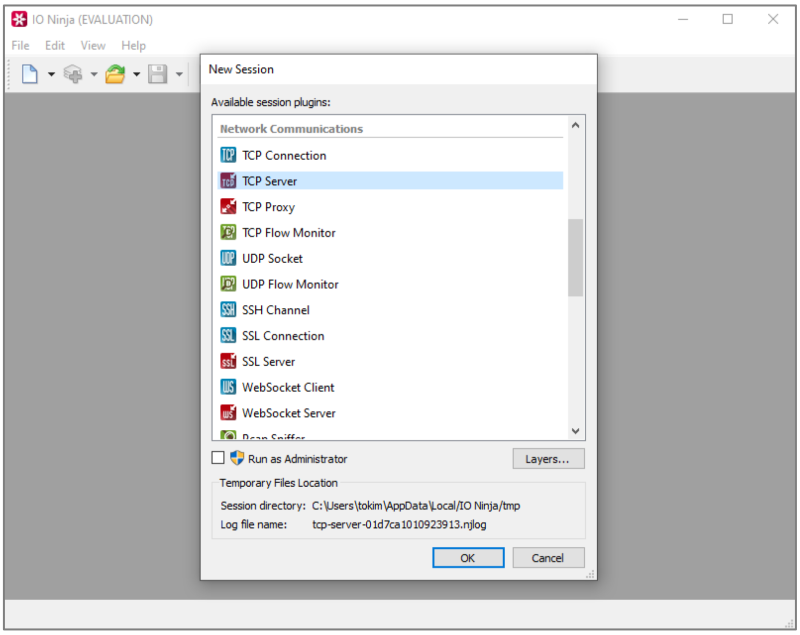


Figure 6: Select TCP Server Session

4. If **TCP Listener Socket** is selected, IO Ninja utility shows the TCP server test window.

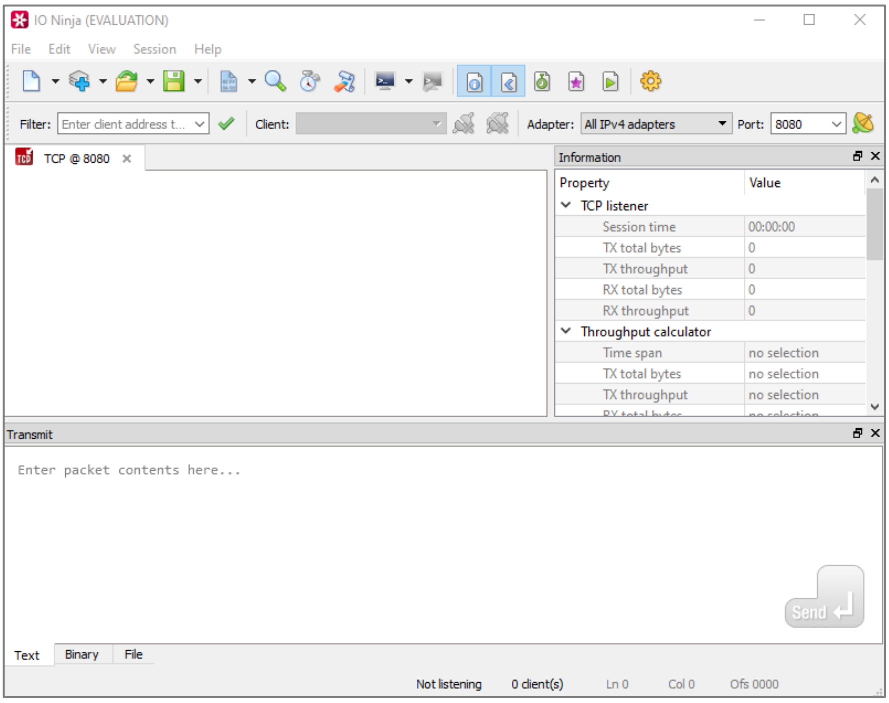


Figure 7: TCP Server Session Windows

5. Start the TCP Server session (for example, in the case of a TCP Client test).

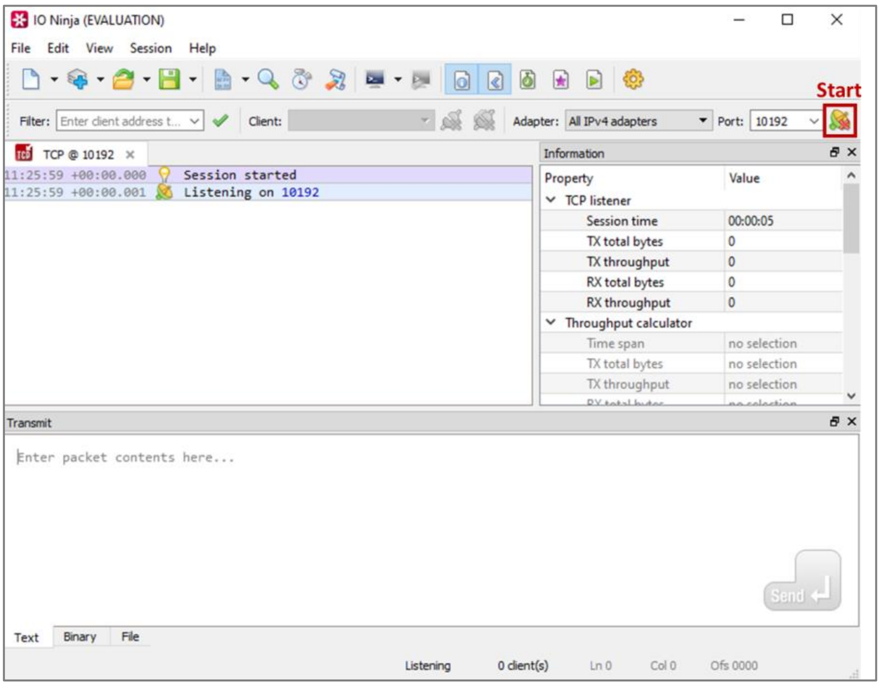


Figure 8: Start TCP Server Session

6. Connect to the TCP Client.

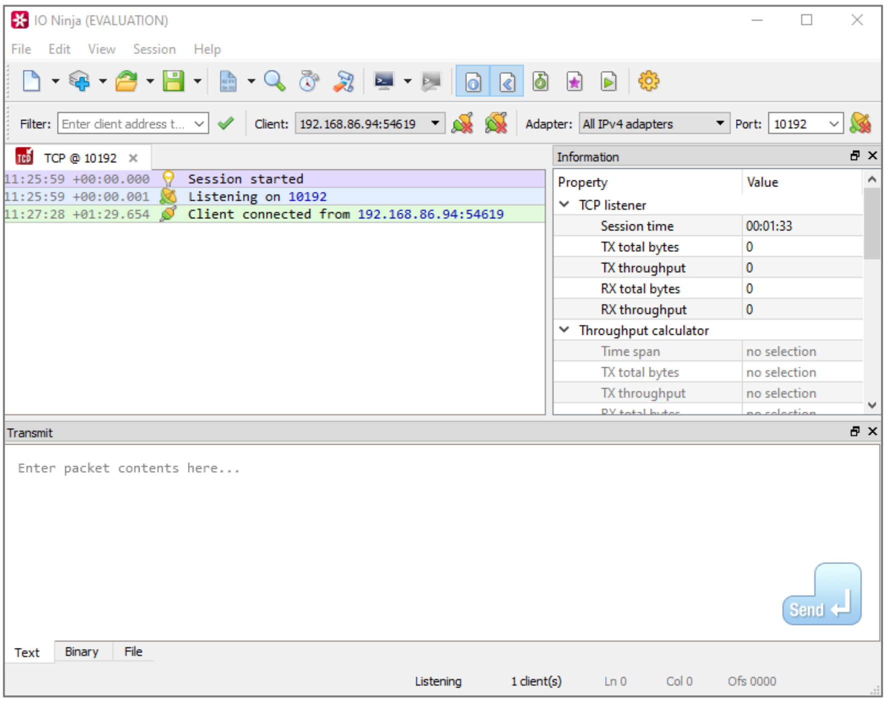


Figure 9: TCP Connection with TCP Client

7. Run data communication.

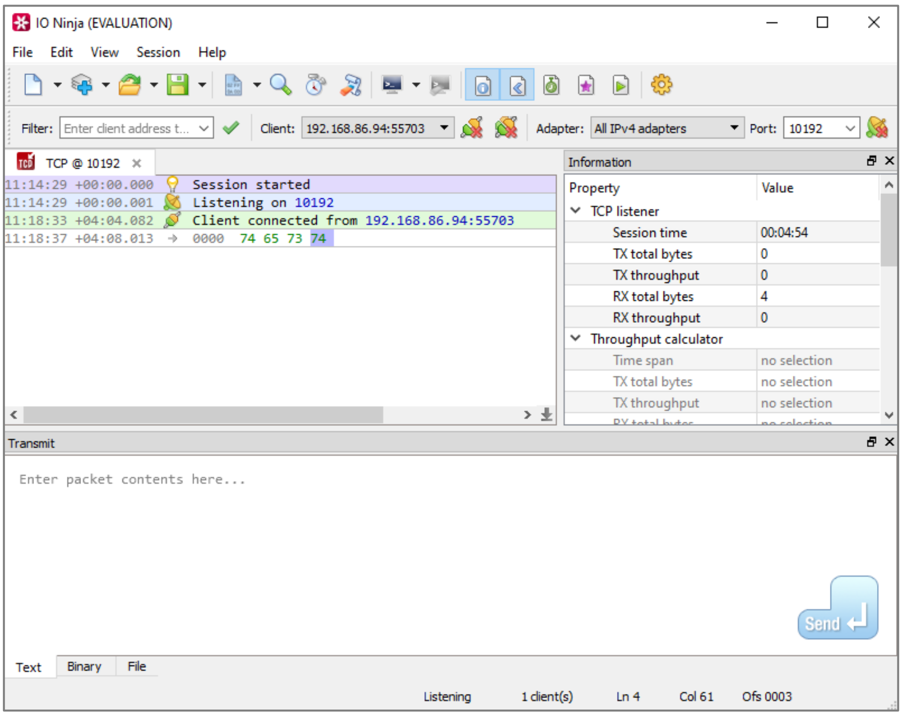


Figure 10: TCP Data Communication with TCP Client

2.2 TCP Client Sample

The TCP client sample is an example of the simplest TCP echo client application. The Transmission Control Protocol is one of the main protocols of the Internet protocol suite. TCP provides a reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications that run on hosts that communicate via an IP network. The DA16200 SDK provides a lwIP's TCP protocol. lwIP is an open-source TCP/IP stack designed for embedded system.

This section describes how the TCP client sample application is built and works.

2.2.1 How to Run

1. Run a socket application on the peer PC (see Section 2.1.2) and open a TCP server socket with port number 10192 (default TCP Client test port).
2. In the Eclipse, import project for TCP Client sample application as follows:
   - *~/SDK/apps/common/examples/Network/TCP_Client/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.
5. To set the IP address and port for the peer application (TCP Server) in the TCP Client Sample, edit the source code:
   ~/SDK/apps/common/examples/Network/TCP_Client/src/tcp_client_sample.c

```
#define TCP_CLIENT_SAMPLE_DEF_SERVER_IP_ADDR            "192.168.0.11"
#define TCP_CLIENT_SAMPLE_DEF_SERVER_PORT                TCP_CLI_TEST_PORT
```

The example connects to the peer application (TCP Server) after a connection is made to the Wi-Fi AP.

2.2.2 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 echoes that message to the TCP server.

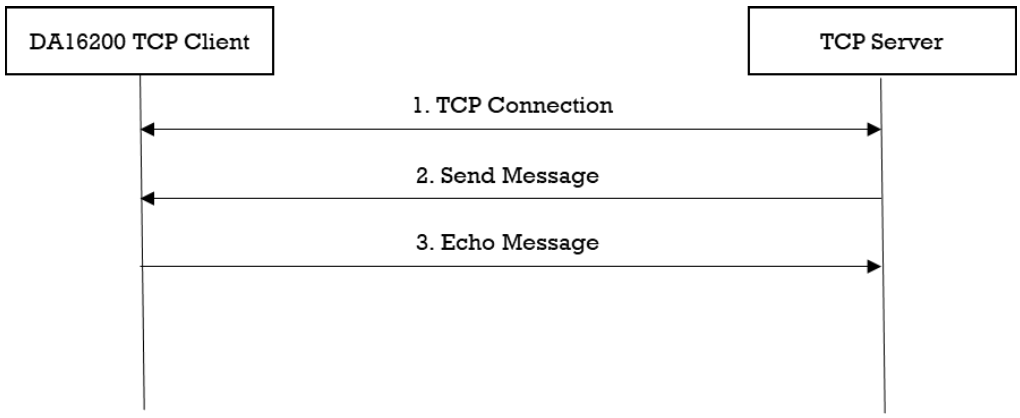


Figure 11: Workflow of TCP Client

2.2.3 Details

The DA16200 SDK provides the lwIP's TCP protocol. This sample application describes how a TCP socket is created, deleted, and configured.

2.2.3.1 Registration

The client side of the TCP connection initiates a connection request to a TCP server. The client TCP socket must be created next with the socket () service and bound to a port via the bind() service. After the client socket is bound, the connect() service is used to establish a connection with a TCP server.


```
void tcp_client_sample(void *param)
{
      int ret = 0;
      int socket_fd;
      struct sockaddr_in local_addr;
      struct sockaddr_in srv_addr;

      memset(&local_addr, 0x00, sizeof(struct sockaddr_in));
      memset(&srv_addr, 0x00, sizeof(struct sockaddr_in));
      // Create TCP socket
      socket_fd = socket(PF_INET, SOCK_STREAM, 0);

      local_addr.sin_family = AF_INET;
      local_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      local_addr.sin_port = htons(TCP_CLIENT_SAMPLE_DEF_PORT);

      // Bind TCP socket
      ret = bind(socket_fd, (struct sockaddr *)&local_addr,
                 sizeof(struct sockaddr_in));

      srv_addr.sin_family = AF_INET;
      srv_addr.sin_addr.s_addr = inet_addr(TCP_CLIENT_SAMPLE_DEF_SERVER_IP_ADDR);
      srv_addr.sin_port = htons(TCP_CLIENT_SAMPLE_DEF_SERVER_PORT);

      // Connect TCP socket
      ret = connect(socket_fd, (struct sockaddr_in *)&srv_addr,
                    sizeof(struct sockaddr_in));
          ....
}
```

2.2.3.2 Data Transmission

TCP data is received when function recv() is called. TCP incoming packet processing handles various connection and disconnection operations, and is responsible for acknowledging transmissions.

TCP data is sent when function send() is called. 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 to the internet with 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.

```
void tcp_client_sample()
{
    ...
    while (1)
    {
        memset(data_buffer, 0x00, sizeof(data_buffer));

        PRINTF("< Read from server: ");
        len = recv(socket_fd, data_buffer, sizeof(data_buffer), 0);
        data_buffer[len] = '\0';
        PRINTF("%d bytes read\r\n", len);

        PRINTF("> Write to server: ");
        len = send(socket_fd, data_buffer, len, 0);
        PRINTF("%d bytes written\r\n", len
      }
    ...
}
```


2.2.3.3 Disconnection

The connection is closed when function close() is called. This function handles socket to be closed and deleted internally. 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, the vTaskDelete() function is called to delete the socket.

```
void tcp_client_sample()
{
    ...
    close(socket_fd);

end_of_task:

     PRINTF("[%s] End of TCP Client sample\r\n", __func__);
     vTaskDelete(NULL);
     return ;
}
```

2.3 TCP Client in DPM

The TCP client in the 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 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.

2.3.1 How to Run

1. Run a socket application on the peer PC (see Section 2.1.2) and open a TCP server socket with port number 10192.
2. In the Eclipse, import project for the TCP Client in the DPM sample application as follows:
   - *~/SDK/apps/common/examples/Network/TCP_Client_DPM/projects/da16200*
   - Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. Use the console command to set up the Wi-Fi station interface.
4. To set the IP address and the port for the peer application (TCP Server) in the TCP Client Sample, do one of the following:
   - Edit the source code:
     ~/SDK/apps/common/examples/Network/TCP_Client_DPM/src/tcp_client_dpm_sample.c
     ```
     #define define TCP_CLIENT_DPM_SAMPLE_DEF_SERVER_IP      "192.168.0.11"
     #define TCP_CLIENT_DPM_SAMPLE_DEF_SERVER_PORT         TCP_CLI_TEST_PORT
     ```
   - Use the DA16200 console to save the values in NVRAM:
     ```
     [/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 a Wi-Fi AP, the example of connecting to the peer application (TCP Server).

2.3.2 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 echoes that message to the TCP server.

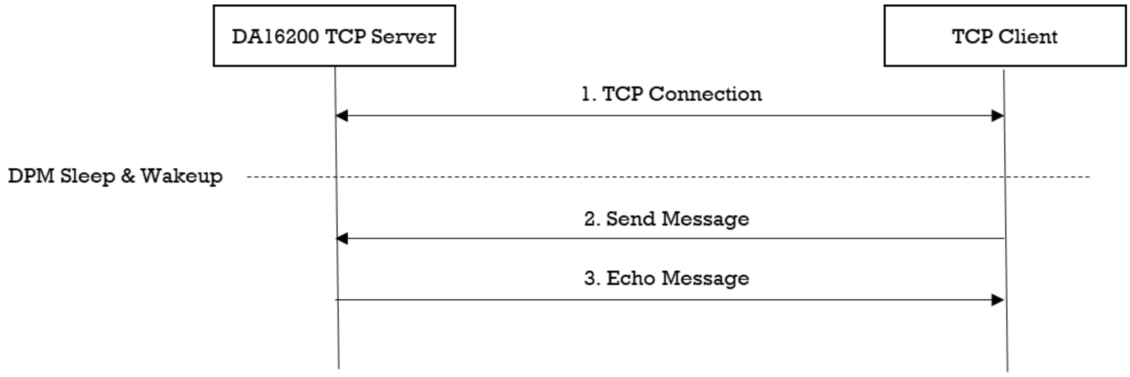


Figure 12: Workflow of TCP Client in DPM

2.3.3 Details

2.3.3.1 Registration

The TCP client in the DPM sample application works in DPM mode. The basic code is similar to the TCP client sample application. There are two differences from the TCP client sample application:

- 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
- An additional user configuration can be stored in RTM

In this sample, the TCP server information is stored.

```
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].sessionType = REG_TYPE_TCP_CLIENT;

	//Set local port
	user_config->sessionConfig[session_idx].sessionMyPort =
	TCP_CLIENT_DPM_SAMPLE_DEF_CLIENT_PORT;

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

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

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

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

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

        //Set connection timeout
        user_config->sessionConfig[session_idx].sessionConnWaitTime =
        TCP_CLIENT_DPM_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT;

        //Set auto reconnection flag
        user_config->sessionConfig[session_idx].sessionAutoReconn = TRUE;

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

        return ;
}
```

2.3.3.2 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.

```
void tcp_client_dpm_sample_recv_callback(void *sock, UCHAR *rx_buf, UINT rx_len,
                                         ULONG rx_ip, ULONG rx_port)
{
        unsigned char status = pdPASS;

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

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

2.4 TCP Server

The TCP server sample application is an example of the simplest TCP echo server application. The Transmission Control Protocol is one of the main protocols of the Internet protocol suite. TCP provides a reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts that communicate via an IP network. The DA16200 SDK provides a lwIP's TCP protocol. lwIP is an open-source TCP/IP stack designed for embedded system.

This section describes how the TCP server sample application is built and works.

2.4.1 How to Run

1. In the Eclipse, import project for the TCP Server sample application as follows:
   - *~/SDK/apps/common/examples/Network/TCP_Server/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. To set the port of the TCP Server Sample, do one of the following:
   - Edit the source code:
     ~/SDK/apps/common/examples/Network/TCP_Server/src/tcp_server_sample.c

```
#define    TCP_SERVER_SAMPLE_DEF_SERVER_PORT      TCP_SVR_TEST_PORT
```

   - Use the DA16200 console to save the values in NVRAM:

```
[/DA16200] # nvram.setenv TCP_SVR_PORT 10190
[/DA16200] # reboot
```

4. Set up the Wi-Fi station interface using console commands.
5. When connected to the AP , the sample application creates a TCP server socket with port number 10190 and waits for a client connection.
6. Run a socket application on the peer PC (See Section 2.1.2).
7. Open a TCP client socket.

2.4.2 How It Works

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

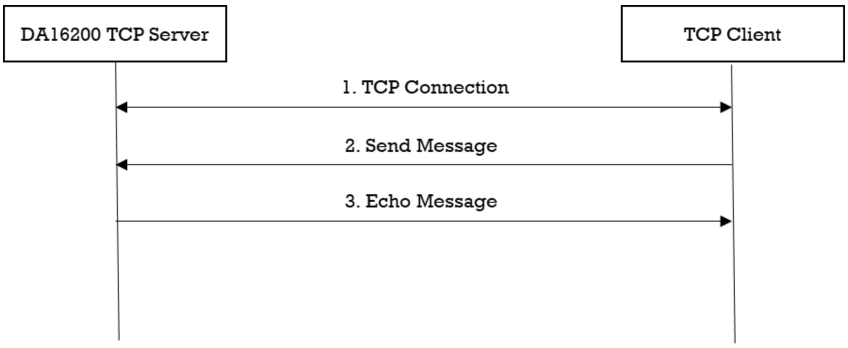


Figure 13: Workflow of TCP Server

2.4.3 Details

The DA16200 SDK provides the lwIP's TCP protocol. This sample application describes how a TCP socket is created, deleted, and configured.

2.4.3.1 Connection

The server waits for a client connection request. Next, the application must create a TCP socket with the socket() service. The server socket must also be set up to listen for connection requests with the listen() service. This service puts 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.

```
void tcp_server_sample()
{
    int ret = 0;
    int listen_sock = -1;
    int client_sock = -1;

    struct sockaddr_in server_addr;
```



```
    struct sockaddr_in client_addr;

    memset(&server_addr, 0x00, sizeof(struct sockaddr_in));
    memset(&client_addr, 0x00, sizeof(struct sockaddr_in));

    // Create TCP socket
    listen_sock = socket(PF_INET, SOCK_STREAM, 0);
    if (listen_sock < 0) {
        PRINTF("[%s] Failed to create listen socket\r\n", __func__);
        goto end_of_task;
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(TCP_SERVER_SAMPLE_DEF_PORT);

    // Bind TCP socket
    ret = bind(listen_sock, (struct sockaddr *)&server_addr,
                sizeof(struct sockaddr_in));

    // Listen TCP socket
    ret = listen(listen_sock, TCP_SERVER_SAMPLE_BACKLOG);

    while (1) {
        client_sock = -1;
        memset(&client_addr, 0x00, sizeof(struct sockaddr_in));
        client_addrlen = sizeof(struct sockaddr_in);

        // Accept TCP socket
        client_sock = accept(listen_sock, (struct sockaddr *)&client_addr,
                                          (socklen_t *)&client_addrlen);
        While (1){
        ...
        }
    }
}
```

2.4.3.2 Data Transmission

TCP data is received when function recv() is called. The TCP receive packet process is responsible for handling the various connection and disconnection actions as well as transmission acknowledgment process.

TCP data is sent when function send() is called. 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 with 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.

```
void tcp_server_sample_run()
{
    ...
    while (NX_TRUE)
    {
        memset(data_buffer, 0x00, sizeof(data_buffer));

        PRINTF("< Read from client: ");
        len = recv(client_sock, data_buffer, sizeof(data_buffer), 0);
        data_buffer[len] = '\0';
        PRINTF("%d bytes read\r\n", len);
```

```
        PRINTF("> Write to client: ");
        len = send(client_sock, data_buffer, len, 0);
        PRINTF("%d bytes written\r\n", len);
    }
    ...
}
```

2.4.3.3 Disconnection

The connection is closed when function close() is called. This function handles socket to be closed and deleted internally.

```
void tcp_server_sample()
{
    ...
    While (1) {
        Close(client_socket)
        ...
    }

end_of_task:

      PRINTF("[%s] End of TCP Server sample\r\n", __func__);
      close(listen_sock);
      close(client_sock);
      vTaskDelete(NULL);
      return ;
}
```

2.5 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.

2.5.1 How to Run

1. Open the workspace for the TCP Server DPM sample application as follows:
    - *~/SDK/apps/common/examples/Network/TCP_Server_DPM/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. To set the port of the TCP Server Sample, do one of the following:
    - Edit the source code:

      ~/SDK/apps/common/examples/Network/TCP_Server_DPM/src/tcp_server_dpm_sample.c

      ```
      #define    TCP_SERVER_DPM_SAMPLE_DEF_SERVER_PORT       TCP_SVR_TEST_PORT
      ```
    - Use the DA16200 console to save the values in NVRAM:

      ```
      [/DA16200] # nvram.setenv TCP_SVR_PORT 10190
      [/DA16200] # reboot
      ```
4. Use the console command to set up the Wi-Fi station interface.
5. When connected to the AP, the sample application creates a TCP server socket with port number 10190 (Default test port number) and waits for a client connection.
6. Run a socket application on the peer PC (See Section 2.1.2).
7. Open a TCP client socket.

2.5.2 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 echoes that message to the TCP client.

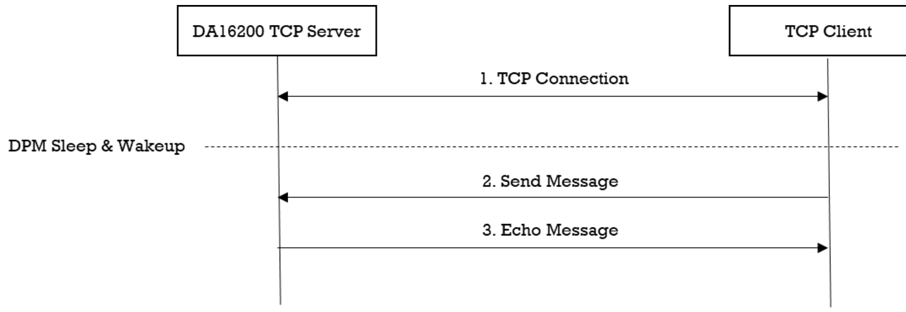


Figure 14: Workflow of TCP Server in DPM

2.5.3 Details

2.5.3.1 Registration

The TCP server in the DPM sample application works in DPM mode. The basic code is similar to the TCP server sample application. There are two differences from the TCP Server sample application:

- 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
- An additional user configuration can be stored in RTM

In this sample, the TCP server information is stored.

```
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].sessionType = REG_TYPE_TCP_SERVER;

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

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

      //Set Recv callback
      user_config->sessionConfig[session_idx].sessionRecvCallback =
        tcp_server_dpm_sample_recv_callback;

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



```
      return ;
}
```

2.5.3.2 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.

```
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 opertaion
}
```

2.6 TCP Client with KeepAlive in DPM

The TCP client with KeepAlive in the 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 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 the DPM sample application is built and works.

2.6.1 How to Run

1. Run a socket application on the peer PC (see Section 2.1.2) and open a TCP server socket with port number 10193 (Default TCP Client test port).
2. In the Eclipse, import project for the TCP Client sample application as follows:
   - *~/SDK/apps/common/examples/Network/TCP_Client_KeepAlive_DPM/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.
5. To set the IP address and the port for the peer application (TCP Server) in the TCP Client KA DPM Sample, do one of the following:
   - Edit the source code:
     ~/SDK/apps/common/examples/Network/TCP_Client_KeepAlive_DPM/src/tcp_client_ka_dpm_sample.c
     ```
     //Default TCP Server configuration
     #define TCP_CLIENT_KA_DPM_SAMPLE_DEF_SERVER_IP            "192.168.0.11"
     #define TCP_CLIENT_KA_DPM_SAMPLE_DEF_SERVER_PORT       TCP_CLI_KA_TEST_PORT
     ```
   - Use the DA16200 console to save the values in NVRAM:
     ```
     [/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 a Wi-Fi AP, the example connects to the peer application (TCP Server).

2.6.2 Details

2.6.2.1 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 the KeepAlive sample application. The time period is 55 seconds to send a TCP KeepAlive message to the TCP server. Compared to the TCP client in the DPM sample application, There are two differences from the TCP client sample application:

- 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
- An additional user configuration can be stored in RTM

In this example, TCP server information is stored.

```
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 DPM wakeup init callback
      user_config->wakeupInitCallback = tcp_client_ka_dpm_sample_wakeup_callback;

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

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

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

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

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

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

      //Set Recv callback
      user_config->sessionConfig[session_idx].sessionRecvCallback =
        tcp_client_ka_dpm_sample_recv_callback;

      //Set connection retry count
      user_config->sessionConfig[session_idx].sessionConnRetryCnt =
        TCP_CLIENT_KA_DPM_SAMPLE_DEF_MAX_CONNECTION_RETRY;

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



```
        //Set auto reconnection flag
        user_config->sessionConfig[session_idx].sessionAutoReconn = pdTRUE;

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

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

        return ;
}
```

2.6.2.2 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);
        else
        {
                //Display sent packet
                PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
        }

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

2.6.3 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 echoes that message to the TCP server. A periodic TCP KeepAlive message is sent to the TCP server every 55 seconds.

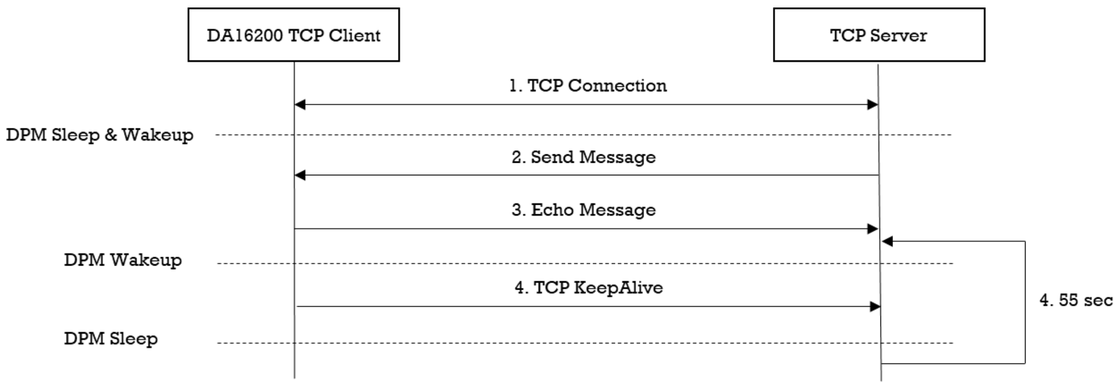


Figure 15: Workflow of TCP Client with KeepAlive in DPM

2.7 UDP Socket

The UDP socket sample application is an example of the simplest UDP echo application. The User Datagram Protocol (UDP) is one of the core members of the Internet protocol suite. UDP uses a simple connectionless communication model with minimum protocol mechanisms. UDP provides checksums for data integrity and port numbers to address different functions at the source and destination of the datagram. Since there is no handshake conversation, it exposes the user program to all the instability of the underlying network; there is no guarantee of delivery, ordering, or duplicate protection. The DA16200 SDK provides a lwIP's TCP protocol. lwIP is an open-source TCP/IP stack designed for embedded system.

This section describes how the UDP socket sample application is built and works.

2.7.1 How to Run

1. Run a socket application on the peer PC (see Section 2.1.2) and open a UDP socket with port number 10195 (default UDP test port).
2. In the Eclipse, import project for the UDP socket sample application as follows:
   - *~/SDK/apps/common/examples/Network/UDP_Socket/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.
5. To set the port number for the peer application (UDP Socket) of the UDP Socket Sample, edit the source code:

   ~/SDK/apps/common/examples/Network/UDP_Socket/src/udp_socket_sample.c

   ```
   #define UDP_SOCKET_SAMPLE_DEF_LOCAL_PORT            UDP_CLI_TEST_PORT
   ```

After a connection is made to a Wi-Fi AP, the example connects to the peer application (UDP Socket).

2.7.2 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 echoes that message to the UDP peer.

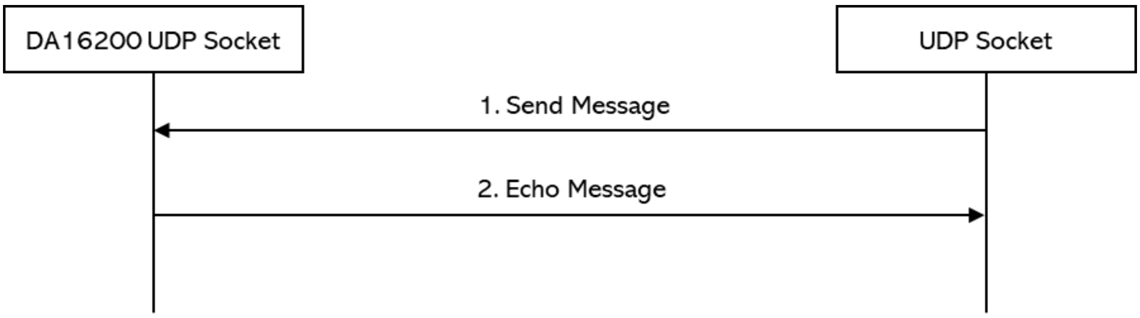


Figure 16: Workflow of UDP Socket

2.7.3 Details

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

2.7.3.1 Initialization

A UDP port is a logical end point in the UDP protocol. There are 65,535 valid ports in the UDP component of lwIP, ranging from 1 through 0xFFFF. To send or receive UDP data, the application must first create a UDP socket with function socket(), then bind the UDP socket to the desired port. Next, the application may send and receive data on that socket. The details are as follows:

```
void udp_socket_sample_run()
```



```
{
     int sock;

     struct sockaddr_in local_addr;
     struct sockaddr_in peer_addr;

     memset(&local_addr, 0x00, sizeof(local_addr));
     memset(&peer_addr, 0x00, sizeof(peer_addr));

     sock = socket(AF_INET, SOCK_DGRAM, 0);
     setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (const void *)&optval,sizeof(int));

     local_addr.sin_family = AF_INET;
     local_addr.sin_addr.s_addr = htonl(INADDR_ANY);
     local_addr.sin_port = htons(UDP_SOCKET_SAMPLE_PEER_PORT);

     ret = bind(sock, (struct sockaddr *)&local_addr, sizeof(struct sockaddr_in));

    ...
}
```

2.7.3.2 Data Transmission

To receive a UDP packet, function recvfrom() is called. The socket receive function delivers the oldest packet on the socket's receive queue. To send UDP data, function sendto() is called. This service puts a UDP header in front of the packet and sends the packet on the Internet with the internal IP send routine.

```
void udp_socket_sample_run()
{
    ...
     while (1) {
         memset(&peer_addr, 0x00, sizeof(struct sockaddr_in));
         memset(data_buffer, 0x00, sizeof(data_buffer));

         PRINTF("< Read from peer: ");

         ret = recvfrom(sock, data_buffer, sizeof(data_buffer), 0,
                        (struct sockaddr *)&peer_addr, (socklen_t *)&addr_len);
         if (ret > 0) {
             len = ret;
             PRINTF("%d bytes read(%d.%d.%d.%d:%d)\r\n", len,
                    (ntohl(peer_addr.sin_addr.s_addr) >> 24) & 0xff,
                    (ntohl(peer_addr.sin_addr.s_addr) >> 16) & 0xff,
                    (ntohl(peer_addr.sin_addr.s_addr) >>  8) & 0xff,
                    (ntohl(peer_addr.sin_addr.s_addr)      ) & 0xff,
                    (ntohs(peer_addr.sin_port)));

             PRINTF("> Write to peer: ");
             ret = sendto(sock, data_buffer, len, 0,
                         (struct sockaddr *)&peer_addr, addr_len);
             PRINTF("%d bytes written(%d.%d.%d.%d:%d)\r\n", len,
                    (ntohl(peer_addr.sin_addr.s_addr) >> 24) & 0xff,
                    (ntohl(peer_addr.sin_addr.s_addr) >> 16) & 0xff,
                    (ntohl(peer_addr.sin_addr.s_addr) >>  8) & 0xff,
                    (ntohl(peer_addr.sin_addr.s_addr)      ) & 0xff,
                    (ntohs(peer_addr.sin_port)));
         }
     }
}
```


2.8 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 the 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.

2.8.1 How to Run

1. Run a socket application on the peer PC (see Section 2.1.2) and open a UDP socket with port number 10194 (Default UDP test port).
2. In the Eclipse, import project for the UDP Server DPM sample application as follows:
   - *~/SDK/apps/common/examples/Network/UDP_Server_DPM/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.
5. To set the port number for the peer application (UDP Client) of the UDP Server DPM Sample, edit the source code:

   ~/SDK/apps/common/examples/Network/UDP_Server_DPM/src/udp_server_dpm_sample.c

   ```
   #define UDP_SERVER_DPM_SAMPLE_DEF_SERVER_PORT        UDP_SVR_TEST_PORT
   ```

After a connection is made to a Wi-Fi AP, the example connects to the peer application (UDP Client).

2.8.2 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 echoes that message to the peer.

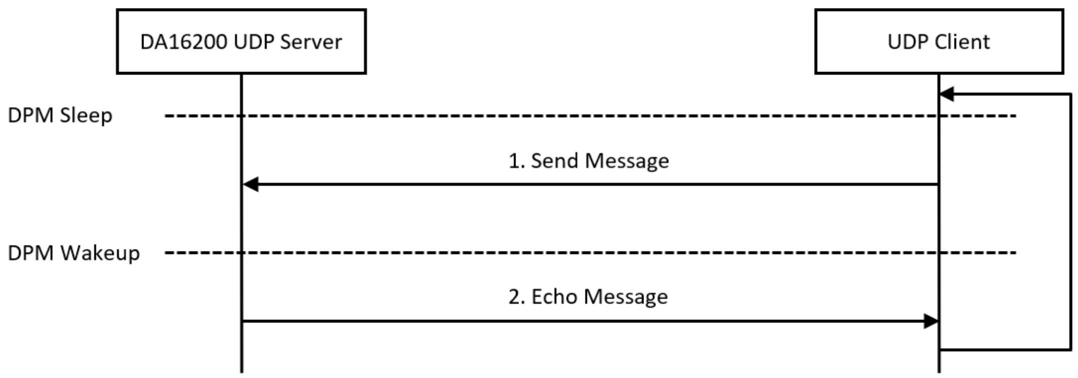


Figure 17: Workflow of UDP Server in DPM

2.8.3 Details

2.8.3.1 Registration

The UDP server in the DPM sample application works 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:

- An 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
- An additional user configuration can be stored in RTM

In this sample, the peer's UDP socket port number will be stored.


```
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].sessionType = REG_TYPE_UDP_SERVER;

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

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

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

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

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

      return ;
}
```

2.8.3.2 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.

```
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);
      if (status)
      {
            PRINTF(RED_COLOR " [%s] Fail send data(session%d,0x%x) \n" CLEAR_COLOR,
                   __func__, SESSION1, status);
      }
      else
      {
      //Display sent packet
```

```
        PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
    }
    dpm_mng_job_done(); //Done opertaion }
```

2.9 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 the DPM sample application is built and works.

2.9.1 How to Run

1. Run a socket application on the peer PC (see Section 2.1.2) and open a UDP socket with port number 10195 (Default UDP test port).
2. In the Eclipse, import project for the UDP Client DPM sample application as follows:
    - *~/SDK/apps/common/examples/Network/UDP_Client_DPM/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.
5. To set the port number for the peer application (UDP Server) of the UDP Client DPM Sample, edit the source code:
   ~/SDK/apps/common/examples/Network/ UDP_Client_DPM/src/udp_client_dpm_sample.c
   ```
   #define UDP_CLIENT_DPM_SAMPLE_DEF_SERVER_PORT       UDP_CLI_TEST_PORT
   ```

After a connection is made to a Wi-Fi AP, the example connects to the peer application (UDP Server).

2.9.2 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 echoes that message to the peer.

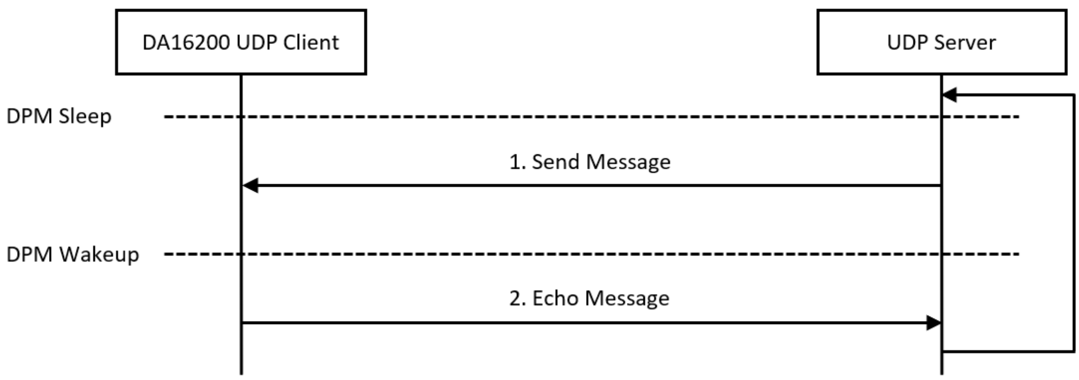


Figure 18: Workflow of UDP Client in DPM

2.9.3 Details

2.9.3.1 Registration

The UDP client in the DPM sample application works in DPM mode. The basic code is similar to the UDP client sample application. There are two differences from the UDP client sample application:

- 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

- An additional user configuration can be stored in RTM

In this example, the peer's UDP IP address and port number are stored.

```
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].sessionType = REG_TYPE_UDP_CLIENT;

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

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

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

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

	//Set Recv callback
	user_config->sessionConfig[session_idx].sessionRecvCallback =
	 udp_client_dpm_sample_recv_callback;

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

	return ;
}
```

2.9.3.2 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.

```
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);

	status = dpm_mng_send_to_session(SESSION1, 0, 0, (char *)rx_buf, rx_len);
	if (status)
	{
	    PRINTF(RED_COLOR " [%s] Fail send data(session%d,0x%x) \n" CLEAR_COLOR,
```



```
                __func__, SESSION1, status);
        }
        else
        {
            //Display sent packet
            PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
        }

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


3 Network Examples: Security

3.1 Peer Application

The examples in this section require a peer device (PC/Laptop) connected to the same Access Point running a (D)TLS test application.

3.1.1 Example of Peer Application (for MS Windows® OS)

There are many (D)TLS counter applications available. In this section, we use a self-implemented (D)TLS counter application to demonstrate these sample applications. It is based on the cryptography APIs of the Bouncy Castle (https://www.bouncycastle.org/java.html).

3.1.1.1 TLS Server

The TLS server application is for the DA16200 TLS client sample application. It runs with a default port number (10196) and waits for a TLS client to connect, as shown in Figure 19. One TLS client connection is allowed, and no client certificate is required during the TLS handshake.

If the TLS session is established successfully, the TLS server application sends a message per five seconds periodically.

```
C:\Samples>tls_server.exe
************************************************************
* TLS Server
* ver. 1.0
* Usage: tls_server.exe [Port]
************************************************************

Bind on tcp/192.168.0.11:10196
```

Figure 19: Start of TLS Server Application

3.1.1.2 TLS Client

The TLS client application is for the DA16200 TLS server sample application. It runs with default TLS server information. The IP address is 192.168.0.2 and the port number is 10197. The TLS client tries to connect to the DA16200 TLS server sample application as shown in Figure 20.

If a TLS session is established successfully, the TLS client application sends a message per 5 seconds periodically.

Usage: tls_client.exe [TLS server IP address] [Port number]

```
C:\Samples>tls_client.exe
************************************************************
* TLS Client
* ver. 1.0
* Usage: tls_client.exe [TLS Server IP Address] [Port]
************************************************************

Server IP Address: 192.168.0.2
Server Port: 10197
```

Figure 20: Start of TLS Client Application

If the TLS client application cannot find a DA16200 TLS server, an exception occurs with a timeout message as shown in Figure 21.

```
C:\Samples>tls_client.exe
************************************************************
* TLS Client
* ver. 1.0
* Usage: tls_client.exe [TLS Server IP Address] [Port]
************************************************************

Server IP Address: 192.168.0.2
Server Port: 10197
Exception in thread "main" java.net.ConnectException: Connection timed out: connect
        at java.net.DualStackPlainSocketImpl.connect0(Native Method)
        at java.net.DualStackPlainSocketImpl.socketConnect(DualStackPlainSocketImpl.java:79)
        at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)
        at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)
        at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188)
        at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:172)
        at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
        at java.net.Socket.connect(Socket.java:589)
        at java.net.Socket.connect(Socket.java:538)
        at java.net.Socket.<init>(Socket.java:434)
        at java.net.Socket.<init>(Socket.java:244)
        at da16200_tls_client_sample.TLSEchoClient.openTlsConnection(TLSEchoClient.java:83)
        at da16200_tls_client_sample.TLSEchoClient.main(TLSEchoClient.java:57)
```

Figure 21: Timeout of TLS Client Application

3.1.1.3 DTLS Server

The DTLS server application is for the DA16200 DTLS client sample application. It runs with a default port number (10199) and waits for the DTLS client to connect, as shown in Figure 22. A client certificate is not required during the DTLS handshake.

If a DTLS session is established successfully, the DTLS server application sends a message per five seconds periodically.

```
C:\Samples>dtls_server.exe
************************************************************
* DTLS Server
* ver. 1.0
* Usage: dtls_server.exe [Port]
************************************************************

Bind on udp/192.168.0.11:10199
```

Figure 22: Start of DTLS Server Application

3.1.1.4 DTLS Client

The DTLS client application is for the DA16200 DTLS server sample application. It runs with default DTLS server information. The IP address is 192.168.0.2 and the port number is 10199. The DTLS client tries to connect to the DA16200 DTLS server sample application as shown in Figure 23.

If a DTLS session is established successfully, the DTLS client application sends a message per five seconds periodically.

Usage: dtls_client.exe [DTLS server IP address] [Port number]

```
C:\Samples>dtls_client.exe
************************************************************
* DTLS Client
* ver. 1.0
* Usage: dtls_client.exe [DTLS Server IP Address] [Port]
************************************************************

Server IP Address: 192.168.0.2
Server Port: 10199
```

Figure 23: Start of DTLS Client Application

3.2 TLS Server

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

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

3.2.1 How to Run

1. In the Eclipse, import project for the TLS Server sample application as follows:
   - *~/SDK/apps/common/examples/Network/TLS_Server/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. Use the console command to set up the Wi-Fi station interface.
4. After a connection is made to an AP, the sample application creates a TLS server socket with port number 10197 and waits for a client connection.
5. Run a TLS client application on the peer PC.

3.2.2 How It Works

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

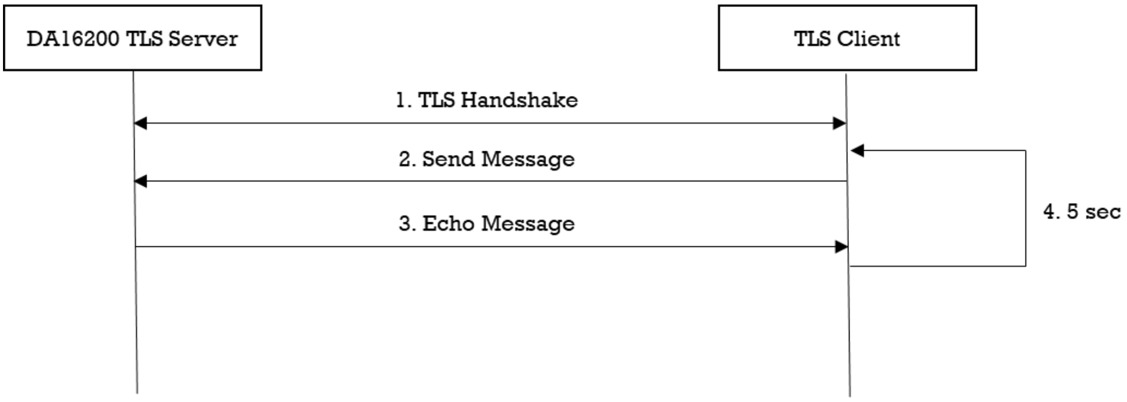


Figure 24: Workflow of TLS Server

3.2.3 Details

The DA16200 SDK provides an “mbedTLS” library. It describes how the TLS server is implemented with an “mbedTLS” library and a socket library.

3.2.3.1 Initialization

The DA16200 secure H/W engine has to be initialized with da16x_secure_module_init() before the TLS context is initialized. To set up a TLS session, initialization functions are called as follows:

```
void tls_server_sample(void *param)
{
      ...

      //Init session
      mbedtls_net_init(&listen_ctx);
      mbedtls_net_init(&client_ctx);

      //Init SSL context
      mbedtls_ssl_init(&ssl_ctx);
```



```
//Init SSL config
mbedtls_ssl_config_init(&ssl_conf);

//Init CTR-DRBG context
mbedtls_ctr_drbg_init(&ctr_drbg);

//Init Entropy context
mbedtls_entropy_init(&entropy);

//Init Certificate context
mbedtls_x509_crt_init(&cert);

//Init Private key context
mbedtls_pk_init(&pkey);

//Init Private key context for ALT
mbedtls_pk_init(&pkey_alt);

//Parse certificate
ret = mbedtls_x509_crt_parse(&cert, tls_server_sample_cert,
                                              tls_server_sample_cert_len);
//Parse private key
ret = mbedtls_pk_parse_key(&pkey, tls_server_sample_key,
                                         tls_server_sample_key_len, NULL, 0);

snprintf(str_port, sizeof(str_port), "%d", TLS_SERVER_SAMPLE_DEF_PORT);
ret = mbedtls_net_bind(&listen_ctx, NULL, str_port, MBEDTLS_NET_PROTO_TCP);

ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                            (const unsigned char *)pers, strlen(pers));

//Set default configuration
ret = mbedtls_ssl_config_defaults(&ssl_conf, MBEDTLS_SSL_IS_SERVER,
                 MBEDTLS_SSL_TRANSPORT_STREAM, MBEDTLS_SSL_PRESET_DEFAULT);

mbedtls_ssl_conf_rng(&ssl_conf, mbedtls_ctr_drbg_random, &ctr_drbg);

//Import certificate & private key
if (mbedtls_pk_get_type(&pkey) == MBEDTLS_PK_RSA) {
    ret = mbedtls_pk_setup_rsa_alt(&pkey_alt,
                                   (void *)mbedtls_pk_rsa(pkey),
                                    tls_server_sample_rsa_decrypt_func,
                                    tls_server_sample_rsa_sign_func,
                                    tls_server_sample_rsa_key_len_func);

    ret = mbedtls_ssl_conf_own_cert(&ssl_conf, &cert, &pkey_alt);
    if (ret) {
       PRINTF("\r\n[%s] Failed to set certificate(0x%x)\r\n", __func__, -ret);
       goto end_of_task;
    }
} else {
    ret = mbedtls_ssl_conf_own_cert(&ssl_conf, &cert, &pkey);
    if (ret) {
       PRINTF("\r\n[%s] Failed to set certificate(0x%x)\r\n", __func__, -ret);
       goto end_of_task;
    }
}
//Don't care verificate of peer certificate
mbedtls_ssl_conf_authmode(&ssl_conf, MBEDTLS_SSL_VERIFY_NONE);
```

```
      //Set up an SSL context for use.
      ret = mbedtls_ssl_setup(&ssl_ctx, &ssl_conf);

reset:
     ...
     mbedtls_ssl_set_bio(&ssl_ctx, &client_ctx, mbedtls_net_send, mbedtls_net_recv,
NULL);
     ...
}
```

3.2.3.2 TLS Handshake

TLS is an encryption protocol designed to secure network communication. A TLS handshake is the process of initiating a communication session that uses TLS encryption. To do a TLS handshake, function mbedtls_ssl_handshake() is called. If an error occurred during the TLS handshake, the API returns a specific error code. If a TLS session is established successfully, the API returns 0. The details are as follows:

```
void tls_server_sample(void *param)
{
    ...
reset:
    ...
     while ((ret = mbedtls_ssl_handshake(&ssl_ctx)) != 0) {
          if ((ret != MBEDTLS_ERR_SSL_WANT_READ) &&
                                            (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
          {
               PRINTF("\r\n[%s] Failed to do handshake(0x%x)\r\n", __func__, -ret);
               goto reset;
          }
     }
     ...
 }
```

3.2.3.3 Data Transmission

Encryption scrambles data so that only authorized parties can understand the information. While a TLS session is established, all application data must be encrypted to transfer application data. “MbedTLS” provides specific APIs to help encrypt and decrypt data. To write application data, function mbedtls_ssl_write() of the “mbedTLS” library is called. The details are as follows:

```
void tls_server_sample(void *param)
{
    ...
reset:
    ...
    do {
        ...
        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_write(0x%x)\r\n", -ret);
                    continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                    PRINTF("\r\nConnection was closed gracefully\r\n");
                    break;
                case MBEDTLS_ERR_NET_CONN_RESET:
                    PRINTF("\r\nConnection was reset by peer\r\n");
                    break;
                default:
```



```
                    PRINTF("Failed to write data(0x%x)\r\n", -ret);
                    break;
                }
                break;
            }
        }
        ...
}
```

To read application data, function mbedtls_ssl_read() of the "mbedTLS" library is called. In this sample, this function is called in tls_server_sample(). The details are as follows:

```
void tls_server_sample(void *param)
{
    ...
reset:
    ...
    do {
        len = sizeof(data_buffer) - 1;
        memset(data_buffer, 0x00, sizeof(data_buffer));

        PRINTF("< Read from client: ");

        ret = mbedtls_ssl_read(&ssl_ctx, data_buffer, len);
        if (ret <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_write(0x%x)\r\n", -ret);
                    continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                    PRINTF("\r\nConnection was closed gracefully\r\n");
                    break;
                case MBEDTLS_ERR_NET_CONN_RESET:
                    PRINTF("\r\nConnection was reset by peer\r\n");
                    break;
                default:
                    PRINTF("\r\nFailed to read data(0x%x)\r\n", -ret);
                    break;
            }
            break;
        }

        len = ret;
        PRINTF("%d bytes read\r\n", len);

        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            ...
        }
    }
    ...
}
```

3.3 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 supports users 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.

3.3.1 How to Run

1. In the Eclipse, import project for the TLS Server in the DPM sample application as follows:
   - *~/SDK/apps/common/examples/Network/TLS_Server_DPM/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. Use the console command to set up the Wi-Fi station interface.
4. After a connection is made to an AP, the example application creates a TLS server socket with port number 10197 and waits for a client connection.
5. Run a TLS client application on the peer PC.

3.3.2 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 echoes that message to the TLS client. The DA16200 TLS server takes time to wait to establish a TLS session.

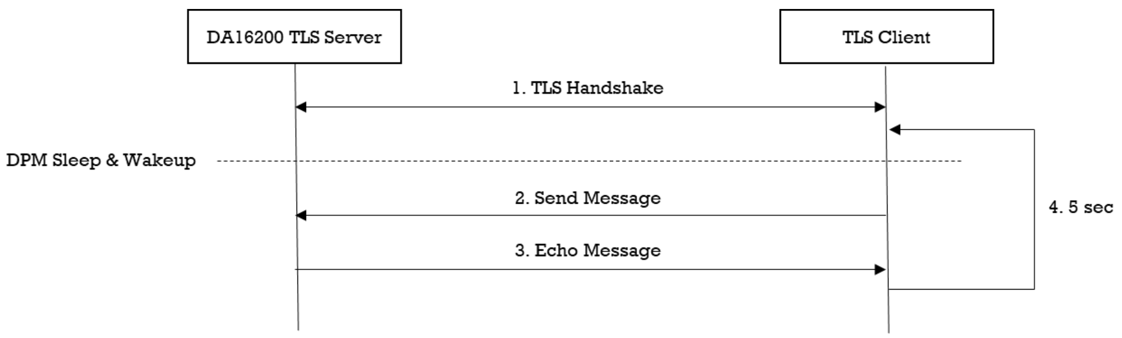


Figure 25: Workflow of TLS Server in DPM

3.3.3 Details

3.3.3.1 Registration

The TLS server in the DPM sample application works in DPM mode. The basic code is similar to the TLS server sample application. There are two differences with the TLS Server sample application:

- 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
- An additional user configuration can be stored in RTM

In this sample, the TLS server information is stored.

```
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].sessionType = REG_TYPE_TCP_SERVER;

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

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

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

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

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

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

        return ;
}
```

3.3.3.2 TLS Setup

To establish a TLS session, TLS should be set up. 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 is set up by sessionSetupSecureCallback function. The details are as follows.

```
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 = dpm_mng_setup_rng(secure_config->ssl_conf);

        //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 opertaion

        return ;
}
```

3.3.3.3 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 is encrypted and decrypted in the callback function.

```
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);

        if (status)
        {
             PRINTF(RED_COLOR " [%s] Fail send data(session%d,0x%x) \n" CLEAR_COLOR,
                    __func__, SESSION1, status);
        }
        else
        {
             //Display sent packet
             PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
        }

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

3.4 TLS Client

The TLS client sample application is an example of the simplest TLS echo client application. Transport Layer Security (TLS) are cryptographic protocols designed to provide secured communication over a computer network. The DA16200 SDK provides a DPM manager feature for the user network application. The DA16200 SDK provides an SSL library called “mbedTLS” on a secure H/W engine to support the TLS protocol. “MbedTLS” is one of the popular SSL libraries and helps to easily develop a network application with a TLS protocol.

This section describes how the TLS client sample application is built and works.

3.4.1 How to Run

1. Run a TLS server application on the peer PC and open a TLS server socket with port number 10196.
2. In the Eclipse, import project for the TLS Client sample application as follows:
   - *~/SDK/apps/common/examples/Network/TLS_Client/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.
5. After a connection is made to an AP, the example application connects to the peer.

3.4.2 How It Works

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

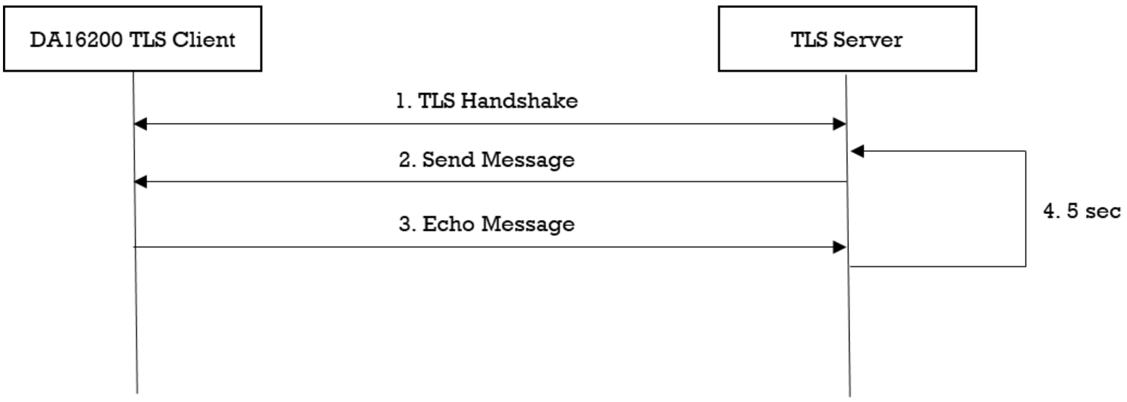


Figure 26: Workflow of TLS Client

3.4.3 Details

DA16200 SDK provides the “mbedTLS” library. It describes how the TLS client is implemented with the “mbedTLS” library and socket library.

3.4.3.1 Registration

The DA16200 secure H/W engine has to be initialized with da16x_secure_module_init() before the TLS context is initialized. To set up a TLS session, initialization functions are called as follows:

```
void tls_client_sample(void *param)
{
     ...

     //Init session
     mbedtls_net_init(&server_ctx);

     //Init SSL context
     mbedtls_ssl_init(&ssl_ctx);

     //Init SSL config
     mbedtls_ssl_config_init(&ssl_conf);
```



```
        //Init CTR-DRBG context
        mbedtls_ctr_drbg_init(&ctr_drbg);

        //Init Entropy context
        mbedtls_entropy_init(&entropy);

        snprintf(str_port, sizeof(str_port), "%d", TLS_CLIENT_SAMPLE_DEF_SERVER_PORT);
        ret = mbedtls_net_connect(&server_ctx,
                                  TLS_CLIENT_SAMPLE_DEF_SERVER_IP_ADDR, str_port,
                                  MBEDTLS_NET_PROTO_TCP);

        //Set default configuration
        ret = mbedtls_ssl_config_defaults(&ssl_conf,
                                          MBEDTLS_SSL_IS_CLIENT,
                                          MBEDTLS_SSL_TRANSPORT_STREAM,
                                          MBEDTLS_SSL_PRESET_DEFAULT);

        ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                                    (const unsigned char *)pers, strlen(pers));

        mbedtls_ssl_conf_rng(&ssl_conf, mbedtls_ctr_drbg_random, &ctr_drbg);

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

        //Setup an SSL context for use.
        ret = mbedtls_ssl_setup(&ssl_ctx, &ssl_conf);

        mbedtls_ssl_set_bio(&ssl_ctx, &server_ctx,
                            mbedtls_net_send, mbedtls_net_recv, NULL);

        ...
}
```

3.4.3.2 TLS Handshake

TLS is an encryption protocol designed to secure network communication. A TLS handshake is the process that starts a communication session that uses TLS encryption. To do a TLS handshake, function mbedtls_ssl_handshake() is called. If an error occurred during the TLS handshake, the API returns a specific error code. If a TLS session is established successfully, the API returns 0. The details are as follows:

```
void tls_client_sample(void *param)
{
    ...
reset:
    ...
    while ((ret = mbedtls_ssl_handshake(&ssl_ctx)) != 0) {
       if (ret == MBEDTLS_ERR_NET_CONN_RESET) {
           PRINTF("\r\n[%s] Peer closed the connection(0x%x)\r\n", __func__, -ret);
           goto end_of_task;
       }

       if ((ret != MBEDTLS_ERR_SSL_WANT_READ) &&
                                        (ret != MBEDTLS_ERR_SSL_WANT_WRITE)) {
           PRINTF("\r\n[%s] Failed to do tls handshake(0x%x)\r\n", __func__, -ret);
           goto end_of_task;
       }
    }
    ...
```

```
  }
```

3.4.3.3 Data Transmission

Encryption scrambles data so that only authorized parties can understand the information. While a TLS session is established, all data must be encrypted to transfer application data. “MbedTLS” provides specific APIs to help encrypt and decrypt data. To write application data, function mbedtls_ssl_write() of the “mbedTLS” library is called. The details are as follows:

```
void tls_client_sample(void *param)
{
    ...
    do {
        ...
        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_write(0x%x)\r\n", -ret);
                    continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                    PRINTF("\r\nConnection was closed gracefully\r\n");
                    break;
                case MBEDTLS_ERR_NET_CONN_RESET:
                    PRINTF("\r\nConnection was reset by peer\r\n");
                    break;
                default:
                    PRINTF("\r\nFailed to write data(0x%x)\r\n", -ret);
                    break;
            }
            goto end_of_task;
        }
    }
    ...
}
```

To read application data, function mbedtls_ssl_read() of the “mbedTLS” library is called. In this sample, this function is called in tls_client_sample(). The details are as follows:

```
void tls_client_sample(void *param)
{
    ...
    do {
        len = sizeof(data_buffer) - 1;
        memset(data_buffer, 0x00, sizeof(data_buffer));

        PRINTF("< Read from server: ");

        //Read at most 'len' application data bytes.
        ret = mbedtls_ssl_read(&ssl_ctx, data_buffer, len);
        if (ret <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_read(0x%x)\r\n", -ret);
                    continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                    PRINTF("\r\nConnection was closed gracefully\r\n");
                    goto end_of_task;
                case MBEDTLS_ERR_NET_CONN_RESET:
                    PRINTF("\r\nConnection was reset by peer\r\n");
```

```
                    goto end_of_task;
                default:
                    PRINTF("\r\nFailed to read data(0x%x)\r\n", -ret);
                    break;
            }
            goto end_of_task;
        }

        len = ret;
        PRINTF("%d bytes read\r\n", len);

        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            ...
        }
    }
    ...
}
```

3.5 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.

3.5.1 How to Run

1. Run a TLS server application on the peer PC and open a TLS server socket with port number 10196.
2. In the Eclipse, import project for a TCP Client in the DPM sample application as follows:
   - *~/SDK/apps/common/examples/Network/TLS_Client_DPM/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up 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 connecting to the AP, the example application connects to the peer PC.

3.5.2 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 echoes that message to the TLS server.

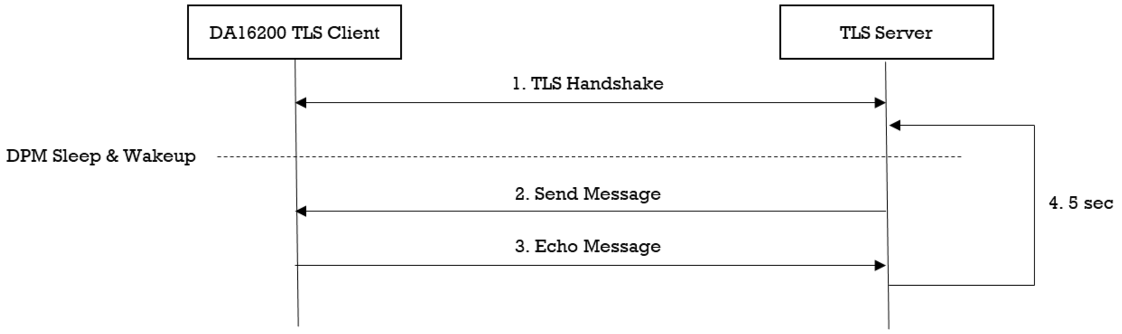


Figure 27: Workflow of TLS Client in DPM

3.5.3 Details

3.5.3.1 Registration

The TLS client in the DPM sample application works in DPM mode. The basic code is similar to the TLS client sample application. There are two differences with the TLS client sample application :

- An initial callback function is added, named tls_client_dpm_sample_wakeup_callback() in the code. It will be called when the DPM state changes from sleep to wake-up
- An additional user configuration that can be stored in RTM

In this example, TLS server information is stored.

```
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].sessionType = REG_TYPE_TCP_CLIENT;

      //Set local port
      user_config->sessionConfig[session_idx].sessionMyPort =
       TLS_CLIENT_DPM_SAMPLE_DEF_CLIENT_PORT;

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

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

      //Set Connection callback
      user_config->sessionConfig[session_idx].sessionConnectCallback =
       tls_client_dpm_sample_connect_callback;

      //Set Recv callback
```



```
    user_config->sessionConfig[session_idx].sessionRecvCallback =
     tls_client_dpm_sample_recv_callback;

    //Set connection retry count
    user_config->sessionConfig[session_idx].sessionConnRetryCnt =
     TLS_CLIENT_DPM_SAMPLE_DEF_MAX_CONNECTION_RETRY;

    //Set connection timeout
    user_config->sessionConfig[session_idx].sessionConnWaitTime =
     TLS_CLIENT_DPM_SAMPLE_DEF_MAX_CONNECTION_TIMEOUT;

    //Set auto reconnection flag
    user_config->sessionConfig[session_idx].sessionAutoReconn = pdTRUE;

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

    //Set secure setup callback
    user_config->sessionConfig[session_idx].sessionSetupSecureCallback =
     tls_client_dpm_sample_secure_callback;

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

    return ;
}
```

3.5.3.2 TLS Setup

To establish a TLS session, TLS should be set up. 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 is set up by sessionSetupSecureCallback function. The details are as shown below. Note that, this sample application does not include certificates.

```
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 = dpm_mng_setup_rng(secure_config->ssl_conf);

    //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 opertaion
    return ;
}
```

3.5.3.3 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 is encrypted and decrypted in the callback function.


```
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);

      status = dpm_mng_send_to_session(SESSION1, rx_ip, rx_port, (char *)rx_buf,
                                        rx_len);
      if (status)
      {
            PRINTF(RED_COLOR " [%s] Fail send data(session%d,0x%x) \n" CLEAR_COLOR,
                   __func__, SESSION1, status);
      }
      else
      {
            //Display sent packet
            PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
      }

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

3.6 DTLS Server

The DTLS server 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 provides an SSL library called “mbedTLS” on a secure H/W engine to support the DTLS protocol. “mbedTLS” is one of the popular SSL libraries. “mbedTLS” is helpful to develop a network application with a DTLS protocol.

This section describes how the DTLS server sample application is built and works.

3.6.1 How to Run

1. In the Eclipse, import project for the DTLS Server sample application as follows:
   - *~/SDK/apps/common/examples/Network/DTLS_Server/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. Use the console command to set up the Wi-Fi station interface.
4. After a connection is made to an AP, the example application creates a DTLS server socket with port number 10199 and waits for a client connection.
5. Run a DTLS client application on the peer PC.

3.6.2 How It Works

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

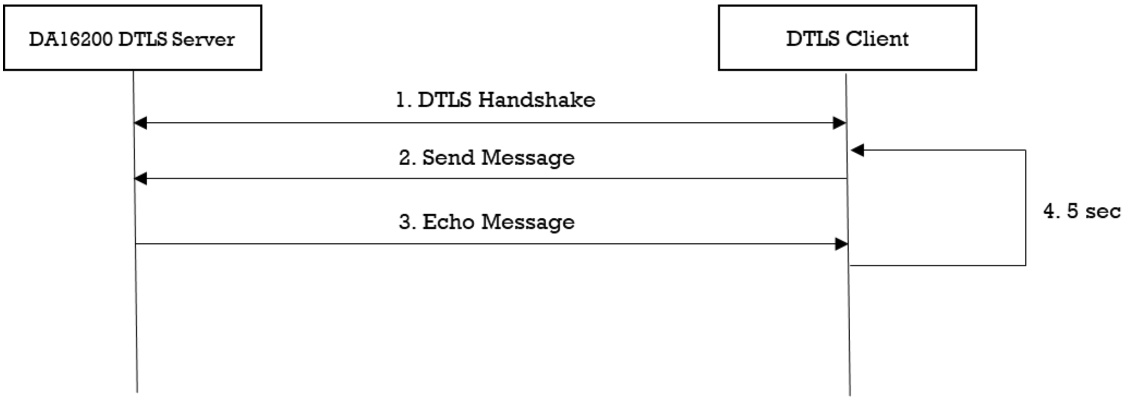


Figure 28: Workflow of DTLS Server

3.6.3 Details

The DA16200 SDK provides the "mbedTLS" library. This sample application describes how the "mbedTLS" library is called and applied for the socket library.

3.6.3.1 Initialization

The DA16200 secure H/W engine has to be initialized with da16x_secure_module_init() before the TLS context is initialized. To set up a DTLS session, initialization functions are called as shown in the example code below. The DTLS session is established over a UDP protocol. In case a packet is lost, retransmission is required. So, the timer is registered to retransmit packet by function mbedtls_ssl_set_timer_cb().

```
void dtls_server_sample(void *param)
{
      ...

      //Init session
      mbedtls_net_init(&listen_ctx);
      mbedtls_net_init(&client_ctx);

      //Init SSL context
      mbedtls_ssl_init(&ssl_ctx);

      //Init SSL config
      mbedtls_ssl_config_init(&ssl_conf);

      //Init CTR-DRBG context
      mbedtls_ctr_drbg_init(&ctr_drbg);

      //Init Entropy context
      mbedtls_entropy_init(&entropy);

      //Init Certificate context
      mbedtls_x509_crt_init(&cert);

      //Init Private key context
      mbedtls_pk_init(&pkey);

      //Init Private key context for ALT
      mbedtls_pk_init(&pkey_alt);

      //Init Cookies
      mbedtls_ssl_cookie_init(&cookies);
      memset(&timer, 0x00, sizeof(dtls_server_sample_timer_t));
```



```
//Parse certificate
ret = mbedtls_x509_crt_parse(&cert, dtls_server_sample_cert,
                             dtls_server_sample_cert_len);
//Parse private key
ret = mbedtls_pk_parse_key(&pkey, dtls_server_sample_key,
                           dtls_server_sample_key_len, NULL, 0);

snprintf(str_port, sizeof(str_port), "%d", DTLS_SERVER_SAMPLE_DEF_PORT);

ret = mbedtls_net_bind(&listen_ctx, NULL, str_port, MBEDTLS_NET_PROTO_UDP);
ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                             (const unsigned char *)pers, strlen(pers));

//Set default configuration
ret = mbedtls_ssl_config_defaults(&ssl_conf,
                                   MBEDTLS_SSL_IS_SERVER,
                                   MBEDTLS_SSL_TRANSPORT_DATAGRAM,
                                   MBEDTLS_SSL_PRESET_DEFAULT);

mbedtls_ssl_conf_rng(&ssl_conf, mbedtls_ctr_drbg_random, &ctr_drbg);

//Import certificate & private key
if (mbedtls_pk_get_type(&pkey) == MBEDTLS_PK_RSA) {
    ret = mbedtls_pk_setup_rsa_alt(&pkey_alt,
                                   (void *)mbedtls_pk_rsa(pkey),
                                   dtls_server_sample_rsa_decrypt_func,
                                   dtls_server_sample_rsa_sign_func,
                                   dtls_server_sample_rsa_key_len_func);

    ret = mbedtls_ssl_conf_own_cert(&ssl_conf, &cert, &pkey_alt);
} else {
    ret = mbedtls_ssl_conf_own_cert(&ssl_conf, &cert, &pkey);
}

//Setup cookies
ret = mbedtls_ssl_cookie_setup(&cookies, mbedtls_ctr_drbg_random, &ctr_drbg);

//Register callbacks for DTLS cookies.
mbedtls_ssl_conf_dtls_cookies(&ssl_conf,
                               mbedtls_ssl_cookie_write,
                               mbedtls_ssl_cookie_check,
                               &cookies);

//Don't care verificate of peer certificate
mbedtls_ssl_conf_authmode(&ssl_conf, MBEDTLS_SSL_VERIFY_NONE);

//Enable or disable anti-replay protection for DTLS.
mbedtls_ssl_conf_dtls_anti_replay(&ssl_conf, MBEDTLS_SSL_ANTI_REPLAY_ENABLED);
mbedtls_ssl_conf_read_timeout(&ssl_conf, DTLS_SERVER_SAMPLE_DEF_TIMEOUT);

//Set retransmit timeout values for the DTLS handshake.
mbedtls_ssl_conf_handshake_timeout(&ssl_conf,
                                    DTLS_SERVER_SAMPLE_DEF_HANDSHAKE_MIN_TIMEOUT,
                                    DTLS_SERVER_SAMPLE_DEF_HANDSHAKE_MAX_TIMEOUT);

//Set up an SSL context for use.
ret = mbedtls_ssl_setup(&ssl_ctx, &ssl_conf);
mbedtls_ssl_set_timer_cb(&ssl_ctx, &timer, dtls_server_sample_timer_start,
                         dtls_server_sample_timer_get_state);
```



```
reset:
      ...
      ret = mbedtls_ssl_set_client_transport_id(&ssl_ctx, client_ip, client_ip_len);
      mbedtls_ssl_set_bio(&ssl_ctx, &client_ctx, mbedtls_net_send, NULL,
                          mbedtls_net_recv_timeout);
      ...
}
```

3.6.3.2 DTLS Handshake

DTLS is an encryption protocol designed to secure network communication. A DTLS handshake is the process that starts a communication session with DTLS encryption. To do a DTLS handshake, the application calls function mbedtls_ssl_handshake(). The DTLS server must verify cookies for the DTLS client. The DTLS client's transport-level identification information must be set up (generally an IP Address). After a ClientHello message is received, the DTLS server must set up its IP address. Then, a DTLS handshake should be retried as follows:

```
void dtls_server_sample(void *param)
{
    ...
reset:
    ...
    while ((ret = mbedtls_ssl_handshake(&ssl_ctx)) != 0) {
        if ((ret == MBEDTLS_ERR_SSL_WANT_READ) ||
                                            (ret == MBEDTLS_ERR_SSL_WANT_WRITE)) {
              continue;
        }
        if (ret == MBEDTLS_ERR_SSL_HELLO_VERIFY_REQUIRED) {
              PRINTF("hello verification requested\r\n");
              ret = 0;
              goto reset;
        } else {
              PRINTF("\r\n[%s] Failed to do handshake(0x%x)\r\n", __func__, -ret);
              goto reset;
        }
    }
    ...
 }
```

3.6.3.3 Data Transmission

Encryption scrambles data so that only authorized parties can understand the information. While a DTLS session is established, all data must be encrypted for transfer. “mbedTLS” provides specific APIs to help encrypt and decrypt data. To write application data, function mbedtls_ssl_write() of the “mbedTLS” library is called. The details are as follows:

```
void dtls_server_sample(void *param){
    ...
    do {
        ...
        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_write(0x%x)\r\n", -ret);
                    continue;
            }
            PRINTF("\r\n[%s] Failed to write data(0x%x)\r\n", __func__, -ret);
            goto end_of_task;
        }
        PRINTF("%d bytes written\r\n", len);
    }
```

```
    ...
}
```

To read application data, function mbedtls_ssl_read() of the “mbedTLS” library is called. In this sample, this function is called in dtls_server_sample(). The details are as follows:

```
void dtls_server_sample(void *param){
    ...
    do {
        len = sizeof(data_buffer) - 1;
        memset(data_buffer, 0x00, sizeof(data_buffer));

        PRINTF("< Read from server: ");

        //Read at most 'len' application data bytes.
        ret = mbedtls_ssl_read(&ssl_ctx, data_buffer, len);
        if (ret <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_write(0x%x)\r\n", -ret);
                    continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                    PRINTF("\r\nConnection was closed gracefully\r\n");
                    ret = 0;
                    goto close_notify;
                case MBEDTLS_ERR_SSL_TIMEOUT:
                    PRINTF("\r\nTimeout\r\n");
                    goto reset;
                default:
                    PRINTF("\r\nFailed to read data(0x%x)\r\n", -ret);
                    break;
            }
            goto reset;
        }

        len = ret;
        PRINTF("%d bytes read\r\n", len);

        PRINTF("> Write to client: ");
        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {      ...
        }
    }
    ...
}
```

3.7 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.

3.7.1 How to Run

1. In the Eclipse, import project for the DTLS Server in the DPM sample application as follows:
   - *~/SDK/apps/common/examples/Network/DTLS_Server_DPM/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. Use the console command to set up the Wi-Fi station interface.
4. After a connection is made to an AP, the example application creates a DTLS server socket with port number 10199 and waits for a client connection.
5. Run a DTLS client application on the peer PC.

3.7.2 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 echoes that message to the DTLS client.

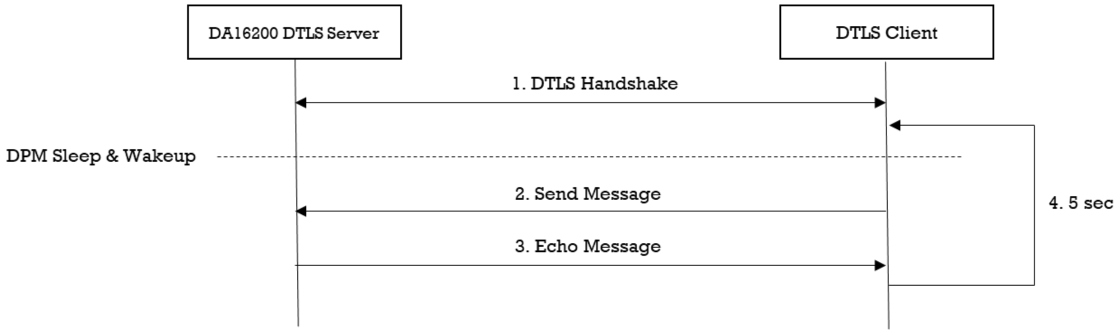


Figure 29: Workflow of DTLS Server in DPM

3.7.3 Details

3.7.3.1 Registration

The DTLS server in the DPM sample application works in DPM mode. The basic code is similar to the DTLS server sample application. There are two differences with the DTLS server sample application:

- An initial callback function is added, named dtls_server_dpm_sample_wakeup_callback() in the code. It is called when the DPM state changes from sleep to wake-up
- An additional user configuration can be stored in RTM

In this sample, DTLS server information is stored.

```
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].sessionType = REG_TYPE_UDP_SERVER;

    //Set local port
    user_config->sessionConfig[session_idx].sessionMyPort =
```



```
     DTLS_SERVER_DPM_SAMPLE_DEF_SERVER_PORT;

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

    //Set Recv callback
    user_config->sessionConfig[session_idx].sessionRecvCallback =
     dtls_server_dpm_sample_recv_callback;

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

    //Set secure setup callback
    user_config->sessionConfig[session_idx].sessionSetupSecureCallback =
     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 ; }
```

3.7.3.2 DTLS Setup

To establish a DTLS session, DTLS should be set up. 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 is set up by sessionSetupSecureCallback function. 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 = dpm_mng_setup_rng(secure_config->ssl_conf);
        ret = dpm_mng_cookie_setup_rng(secure_config->cookie_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);

        //use default value
        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_SERVER_DPM_SAMPLE_RECEIVE_TIMEOUT * 10);

        mbedtls_ssl_conf_handshake_timeout(secure_config->ssl_conf,
                                 DTLS_SERVER_DPM_SAMPLE_HANDSAHKE_MIN_TIMEOUT * 10,
                                 DTLS_SERVER_DPM_SAMPLE_HANDSAHKE_MAX_TIMEOUT * 10);

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

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

3.7.3.3 Data Transmission

The callback function is 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 is encrypted and decrypted in the callback function.

```
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);


        if (status)
        {
            PRINTF(RED_COLOR " [%s] Fail send data(session%d,0x%x) \n" CLEAR_COLOR,
```



```
                __func__, SESSION1, status);
    }
    else
    {
        //Display sent packet
        PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
    }

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

3.8 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. The DA16200 SDK provides an SSL library called “mbedTLS” on a secure H/W engine to support the DTLS protocol. “mbedTLS” is one of the popular SSL libraries. “mbedTLS” is helpful to easily develop a network application with the DTLS protocol.

This section describes how the DTLS client sample application is built and works.

3.8.1 How to Run

1. Run a DTLS server application on the peer PC and open a DTLS server socket with port number 10199.
2. In the Eclipse, import project for the DTLS Client sample application as follows:
   - *~/SDK/apps/common/examples/Network/DTLS_Client/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.

After a connection is made to an AP, the sample application connects to the peer PC.

3.8.2 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 echoes that message to the DTLS server.

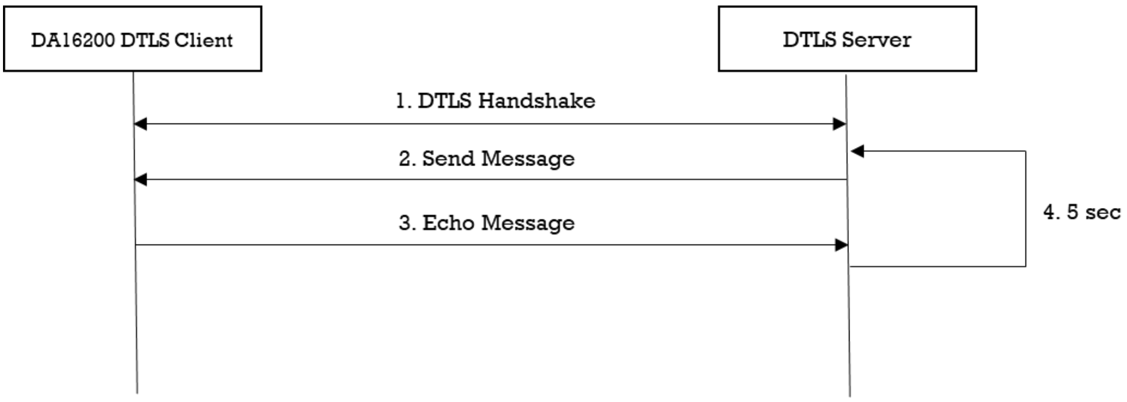


Figure 30: Workflow of DTLS Client

3.8.3 Details

The DA16200 SDK provides an “mbedTLS” library. This sample application describes how an “mbedTLS” library is called and applied for the socket library.

3.8.3.1 Initialization

The DA16200 secure H/W engine has to be initialized with da16x_secure_module_init() before the DTLS context is initialized. To set up a DTLS session, initialization functions are called as shown in the example code below. A DTLS session is established over the UDP protocol. If a packet is lost, then retransmission is required. So, the timer is registered to retransmit the packet by function mbedtls_ssl_set_timer_cb().

```
void dtls_client_sample(void *param)
{
      ...

      //Init session
      mbedtls_net_init(&server_ctx);

      //Init SSL context
      mbedtls_ssl_init(&ssl_ctx);

      //Init SSL config
      mbedtls_ssl_config_init(&ssl_conf);

      //Init CTR-DRBG context
      mbedtls_ctr_drbg_init(&ctr_drbg);

      //Init Entropy context
      mbedtls_entropy_init(&entropy);

      memset(&timer, 0x00, sizeof(dtls_client_sample_timer_t));

      PRINTF("\r\nConnecting to udp/%s:%d...",
             DTLS_CLIENT_SAMPLE_DEF_SERVER_IP_ADDR,
             DTLS_CLIENT_SAMPLE_DEF_SERVER_PORT);

      snprintf(str_port, sizeof(str_port),"%d", DTLS_CLIENT_SAMPLE_DEF_SERVER_PORT);

      ret = mbedtls_net_connect(&server_ctx,
                                DTLS_CLIENT_SAMPLE_DEF_SERVER_IP_ADDR, str_port,
                                MBEDTLS_NET_PROTO_UDP);

      ret = mbedtls_ssl_config_defaults(&ssl_conf,
                                        MBEDTLS_SSL_IS_CLIENT,
                                        MBEDTLS_SSL_TRANSPORT_DATAGRAM,
                                        MBEDTLS_SSL_PRESET_DEFAULT);

      ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                                  (const unsigned char *)pers, strlen(pers));

      mbedtls_ssl_conf_rng(&ssl_conf, mbedtls_ctr_drbg_random, &ctr_drbg);
      mbedtls_ssl_conf_authmode(&ssl_conf, MBEDTLS_SSL_VERIFY_NONE);
      mbedtls_ssl_conf_dtls_anti_replay(&ssl_conf, MBEDTLS_SSL_ANTI_REPLAY_ENABLED);
      mbedtls_ssl_conf_read_timeout(&ssl_conf, DTLS_CLIENT_SAMPLE_DEF_TIMEOUT);
      mbedtls_ssl_conf_handshake_timeout(&ssl_conf,
                                         DTLS_CLIENT_SAMPLE_DEF_HANDSHAKE_MIN_TIMEOUT,
                                         DTLS_CLIENT_SAMPLE_DEF_HANDSHAKE_MAX_TIMEOUT);

      ret = mbedtls_ssl_setup(&ssl_ctx, &ssl_conf);

      mbedtls_ssl_set_bio(&ssl_ctx, &server_ctx,
                          mbedtls_net_send, NULL, mbedtls_net_recv_timeout);

      mbedtls_ssl_set_timer_cb(&ssl_ctx, &timer,
```



```
                                dtls_client_sample_timer_start,
                                dtls_client_sample_timer_get_state);

     ...
}
```

3.8.3.2 DTLS Handshake

DTLS is an encryption protocol designed to secure network communication. A DTLS handshake is the process of initiating a communication session that uses DTLS encryption. To do a DTLS handshake, function mbedtls_ssl_handshake() is called. If an error occurs during a DTLS handshake, the API returns the specific error code. If a DTLS session is established successfully, the API returns 0. The details are as follows:

```
void dtls_client_sample(void *param)
{
    ...
    while ((ret = mbedtls_ssl_handshake(&ssl_ctx)) != 0) {
        if (ret == MBEDTLS_ERR_NET_CONN_RESET) {
             PRINTF("\r\n[%s] Peer closed the connection(0x%x)\r\n", __func__, -ret);
             goto end_of_task;
        }

        if ((ret != MBEDTLS_ERR_SSL_WANT_READ) && (ret != MBEDTLS_ERR_SSL_WANT_WRITE))
        {
             PRINTF("\r\n[%s] Failed to do dtls handshake(0x%x)\r\n", __func__, -ret);
             goto end_of_task;
        }
    }
    ...
}
```

3.8.3.3 Data Transmission

Encryption scrambles data so that only authorized parties can understand the information. While a DTLS session is established, all data must be encrypted to transfer application data. “mbedTLS” provides specific APIs to help encrypt and decrypt data. To write application data, call function mbedtls_ssl_write() of the “mbedTLS” library. The details are as follows:

```
void dtls_client_sample(void *param)
{
    ...
    do {
        ...
        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                   PRINTF("\r\nNeed more data - mbedtls_ssl_write(0x%x)\r\n", -ret);
                   continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                   PRINTF("\r\nConnection was closed gracefully\r\n");
                   goto end_of_task;
                case MBEDTLS_ERR_NET_CONN_RESET:
                   PRINTF("\r\nConnection was reset by peer\r\n");
                   goto end_of_task;
                default:
                   PRINTF("\r\nFailed to write data(0x%x)\r\n", -ret);
                   break;
            }
```



```
            goto end_of_task;
        }
        PRINTF("%d bytes written\r\n", len);
    }
    ...
}
```

To read application data, function mbedtls_ssl_read() of the “mbedTLS” library is called. In this sample, this function is called in dtls_client_sample(). The details are as follows:

```
void dtls_server_sample(void *param)
{
    ...
    do {
        len = sizeof(data_buffer) - 1;
        memset(data_buffer, 0x00, sizeof(data_buffer));

        PRINTF("< Read from server: ");

        ret = mbedtls_ssl_read(&ssl_ctx, data_buffer, len);
        if (ret <= 0) {
            switch (ret) {
                case MBEDTLS_ERR_SSL_WANT_READ:
                case MBEDTLS_ERR_SSL_WANT_WRITE:
                    PRINTF("\r\nNeed more data - mbedtls_ssl_read(0x%x)\r\n", -ret);
                    continue;
                case MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY:
                    PRINTF("\r\nConnection was closed gracefully\r\n");
                    goto end_of_task;
                case MBEDTLS_ERR_NET_CONN_RESET:
                    PRINTF("\r\nConnection was reset by peer\r\n");
                    goto end_of_task;
                default:
                    PRINTF("\r\nFailed to read data(0x%x)\r\n", -ret);
                    break;
            }
            goto end_of_task;
        }

        len = ret;
        PRINTF("%d bytes read\r\n", len);

        PRINTF("> Write to server: ");
        while ((ret = mbedtls_ssl_write(&ssl_ctx, data_buffer, len)) <= 0) {
            ...
        }
    }
    ...
}
```

3.9 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.

3.9.1 How to Run

1. Run a DTLS server application on the peer PC and open a DTLS server socket with port number 10199.
2. In the Eclipse, import project for the DTLS Client in the DPM sample application as follows:
   - *~/SDK/apps/common/examples/Network/DTLS_Client_DPM/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up 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.

```
[/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 connects to the peer PC.

3.9.2 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 echoes that message to the DTLS server.

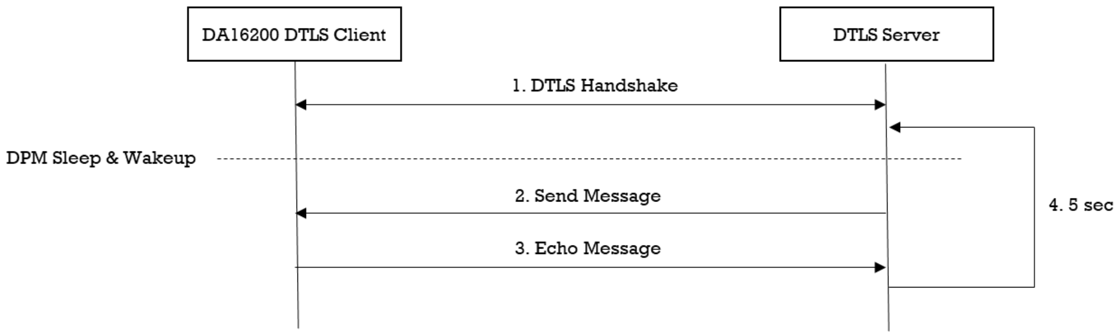


Figure 31: Workflow of DTLS Client in DPM

3.9.3 Details

3.9.3.1 Registration

The DTLS client in the DPM sample application works in DPM mode. The basic code is similar to the DTLS client sample application. There are two differences with the DTLS client sample application:

- An initial callback function is added, named dtls_client_dpm_sample_wakeup_callback() in the code. It is called when the DPM state changes from sleep to wake-up
- An additional user configuration can be stored in RTM

In this sample, DTLS server information is stored.

```
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].sessionType = REG_TYPE_UDP_CLIENT;

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

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

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

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

        //Set Recv callback
        user_config->sessionConfig[session_idx].sessionRecvCallback =
         dtls_client_dpm_sample_recv_callback;

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

        //Set secure setup callback
        user_config->sessionConfig[session_idx].sessionSetupSecureCallback =
         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 ;
}
```

3.9.3.2 DTLS Setup

To establish a DTLS session, DTLS should be set up. 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 is set up 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 = dpm_mng_setup_rng(secure_config->ssl_conf);
```



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

        //use default value
        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_HANDSAHKE_MIN_TIMEOUT * 10,
                                     DTLS_CLIENT_DPM_SAMPLE_HANDSAHKE_MAX_TIMEOUT * 10);

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

        dpm_mng_job_done(); //Done opertaion

        return ;
}
```

3.9.3.3 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 is encrypted and decrypted in the callback function.

```
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);

        status = dpm_mng_send_to_session(SESSION1,
                                         rx_ip,
                                         rx_port,
                                         (char *)rx_buf,
                                         rx_len);
        if (status)
        {
            PRINTF(RED_COLOR " [%s] Fail send data(session%d,0x%x) \n" CLEAR_COLOR,
                   __func__, SESSION1, status);
        }
        else
        {
            //Display sent packet
            PRINTF(" <===== Sent Packet(%ld) \n", rx_len);
        }

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

4 Network Examples: Protocols/Applications

4.1 CoAP Client

4.1.1 Peer Application

The example in this section requires a peer device (PC/Laptop) running a CoAP test server application to demonstrate the DA16200 CoAP client sample application. The sample application is based on Eclipse Californium (https://www.eclipse.org/californium/) and runs on a Windows OS as shown in Figure 32.

```
C:\Samples>coap_server.exe
************************************************************
* CoAP Server
* ver. 1.0
* resources
*        \res
*        \obs_res
************************************************************
```

Figure 32: Start of CoAP Server Application

The CoAP server application is a simple CoAP server. It has two resources, called \res and \obs_res. The “res” resource allows GET, POST, PUT, DELETE, and PING methods. The “obs_res” resource allows OBSERVE request and sends an observe notification every ten seconds.

4.1.2 How to Run

1. Run a CoAP server application on the peer PC.
2. In the Eclipse, import project for the CoAP Client application as follows:
   - *~/SDK/apps/common/examples/Network/CoAP_Client/projects/da16200*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console command to set up the Wi-Fi station interface.

After a connection is made to an AP, the example application initializes a CoAP client to start the service.

4.1.3 CoAP Client Initialization

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

```
int coap_client_sample_init_config(coap_client_sample_conf_t *config)
{
    int ret = DA_APP_SUCCESS;

    coap_client_t *coap_client_ptr = &config->coap_client;

    config->state = COAP_CLIENT_SAMPLE_STATE_SUSPEND;

    //Init coap client
    ret = coap_client_init(coap_client_ptr, COAP_CLIENT_SAMPLE_DEF_NAME);
    if (ret != DA_APP_SUCCESS) {
        PRINTF("[%s]Failed to init coap client(0x%x)\r\n", __func__, -ret);
        goto end;
    }

    coaps_client_set_authmode(coap_client_ptr, 0);

    config->req_port = COAP_CLIENT_SAMPLE_REQUEST_PORT;
    config->obs_port = COAP_CLIENT_SAMPLE_OBSERVE_PORT;
```



```
end:

      return ret;
}
```

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 relationship is already established in DPM wakeup, It will be recovered. The API's details are as follows:

- `int coap_client_init(coap_client_t *client_ptr, char *name_ptr)`

Prototype	int coap_client_init(coap_client_t *client_ptr, char *name_ptr)
Description	Initialize CoAP Client.
Parameters	client_ptr: CoAP Client instance pointer name_ptr: Name of CoAP Client
Return values	0(NX_SUCCESS) on success

- `int coaps_client_set_authmode(coap_client_t *client_ptr, unsigned int mode)`

Prototype	int coaps_client_set_authmode(coap_client_t *client_ptr, unsigned int 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

4.1.4 CoAP Client Deinitialization

This section explains how to release the CoAP client.

```
int coap_client_sample_deinit_config(coap_client_sample_conf_t *config)
{
      int ret = DA_APP_SUCCESS;
      coap_client_t *coap_client_ptr = &config->coap_client;

      //Deinit coap client
      ret = coap_client_deinit(coap_client_ptr);
      if (ret != DA_APP_SUCCESS) {
          PRINTF("[%s]Failed to deinit coap client(0x%x)\r\n", __func__, -ret);
      }

      return ret;
}
```

The coap_client_deinit function releases the CoAP client. The API details are as follows.

- `int coap_client_deinit(coap_client_t *client_ptr)`

Prototype	int 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

4.1.5 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.

4.1.5.1 CoAP URI and Proxy-URI

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

- `int coap_client_set_uri(coap_client_t *client_ptr, unsigned char *uri, size_t urilen)`

Prototype	int 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 uri: URI of CoAP request urilen: Length of URI
Return values	0(NX_SUCCESS) on success

- `int coap_client_set_proxy_uri(coap_client_t *client_ptr, unsigned char *uri, size_t urilen)`

Prototype	int 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 uri: Proxy-URI of CoAP request urilen: Length of URI
Return values	0(NX_SUCCESS) on success

4.1.5.2 GET Method

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

```
int coap_client_sample_request_get(coap_client_sample_conf_t *config,
                                   coap_client_sample_request_t *request)
{
      int ret = DA_APP_SUCCESS;
      coap_client_t *coap_client_ptr = &config->coap_client;
      coap_rw_packet_t resp_packet;
      memset(&resp_packet, 0x00, sizeof(coap_rw_packet_t));

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

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

      //send coap request
      ret = coap_client_request_get_with_port(coap_client_ptr, config->req_port);

      //receive coap response
      ret = coap_client_recv_response(coap_client_ptr, &resp_packet);
```



```
        //display output
        if (resp_packet.payload.len) {
          coap_client_sample_hexdump("GET Request",
                                            resp_packet.payload.p,
                                            resp_packet.payload.len);
        }

end:
        //release coap response
        coap_clear_rw_packet(&resp_packet);

        return ret;
}
```

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 details are as follows:

- `int coap_client_request_get_with_port(coap_client_t *client_ptr, unsigned int port)`

Prototype	int coap_client_request_get_with_port(coap_client_t *client_ptr, unsigned int 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

The DA16200 CoAP client sample application provides a command to send a GET request to the CoAP server. Figure 33, Figure 34, and Figure 35 show the interaction of two DA16200 CoAP clients with the CoAP server for a get request.

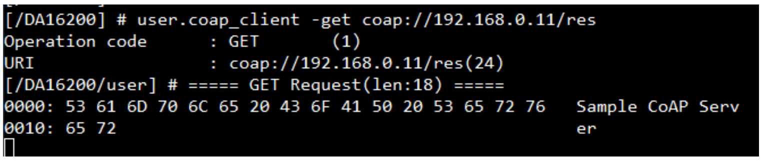


Figure 33: GET Method of CoAP Client #1

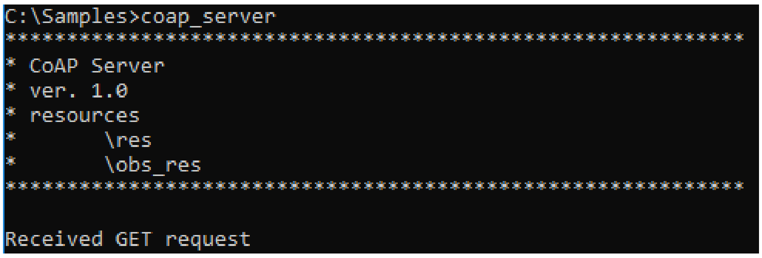


Figure 34: GET Method of CoAP Client #2

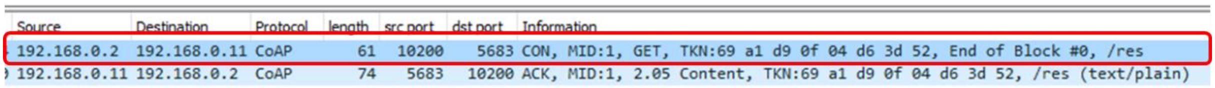


Figure 35: GET Method of CoAP Client #3

4.1.5.3 POST Method

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

```
int coap_client_sample_request_post(coap_client_sample_conf_t *config,
                                     coap_client_sample_request_t *request)
```



```
{
      int ret = DA_APP_SUCCESS;
      coap_client_t *coap_client_ptr = &config->coap_client;
      coap_rw_packet_t resp_packet;
      memset(&resp_packet, 0x00, sizeof(coap_rw_packet_t));

      //set URI
      ret = coap_client_set_uri(coap_client_ptr,
                                (unsigned char *)request->uri,
                                 request->urilen);

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

      //send coap request
      ret = coap_client_request_post_with_port(coap_client_ptr, config->req_port,
                                               request->data, request->datalen);

      //receive coap response
      ret = coap_client_recv_response(coap_client_ptr, &resp_packet);

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

end:
      //release coap response
      coap_clear_rw_packet(&resp_packet);
      return ret;
}
```

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 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

The DA16200 CoAP client sample application has a command to send a POST request to a CoAP server. Figure 36, Figure 37, and Figure 38 show the interaction of two DA16200 CoAP clients with the CoAP server for a POST request.


```
[/DA16200] # user.coap_client -post 123 coap://192.168.0.11/res
Operation code       : POST      (3)
URI                  : coap://192.168.0.11/res(24)
PAYLOAD              : 123(4)
[/DA16200/user] # ===== POST Request(len:4) =====
0000: 31 32 33 00                                      123.
```

Figure 36: POST Method of CoAP Client #1

```
C:\Samples>coap_server
*************************************************************
* CoAP Server
* ver. 1.0
* resources
*       \res
*       \obs_res
*************************************************************

Received POST request
```

Figure 37: POST Method of CoAP Client #2

Source	Destination	Protocol	length	src port	dst port	Information
192.168.0.2	192.168.0.11	CoAP	64	10200	5683	CON, MID:1, POST, TKN:e9 e5 f0 26 4d f6 95 25, /res (text/plain)
192.168.0.11	192.168.0.2	CoAP	60	5683	10200	ACK, MID:1, 2.04 Changed, TKN:e9 e5 f0 26 4d f6 95 25, /res (text/plain)

Figure 38: POST Method of CoAP Client #3

4.1.5.4 PUT Method

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

```
int coap_client_sample_request_put(coap_client_sample_conf_t *config,
                                   coap_client_sample_request_t *request)
{
      int ret = DA_APP_SUCCESS;
      coap_client_t *coap_client_ptr = &config->coap_client;
      coap_rw_packet_t resp_packet;

      memset(&resp_packet, 0x00, sizeof(coap_rw_packet_t));

      //set URI
      ret = coap_client_set_uri(coap_client_ptr,
                                (unsigned char *)request->uri,
                                 request->urilen);

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


      //send coap request
      ret = coap_client_request_put_with_port(coap_client_ptr, config->req_port,
                                              request->data,
                                              request->datalen);

      //receive coap response
      ret = coap_client_recv_response(coap_client_ptr, &resp_packet);

      //display output
      if (resp_packet.payload.len) {
        coap_client_sample_hexdump("PUT Request",
                                   resp_packet.payload.p,
```



```
                                        resp_packet.payload.len);
        }

end:
        //release coap response
        coap_clear_rw_packet(&resp_packet);

        return ret;
}
```

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 details are as follows.

- `int coap_client_request_put_with_port(coap_client_t *client_ptr, unsigned int port, unsigned char *payload, unsigned int payload_len)`

Prototype	int coap_client_request_put_with_port(coap_client_t *client_ptr, unsigned int 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

The DA16200 CoAP client sample application provides a command to send a PUT request to the CoAP server. Figure 39, Figure 40, and Figure 41 show the interaction of two DA16200 CoAP clients and the CoAP server for put requests.

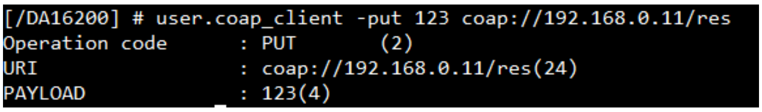


Figure 39: PUT Method of CoAP Client #1

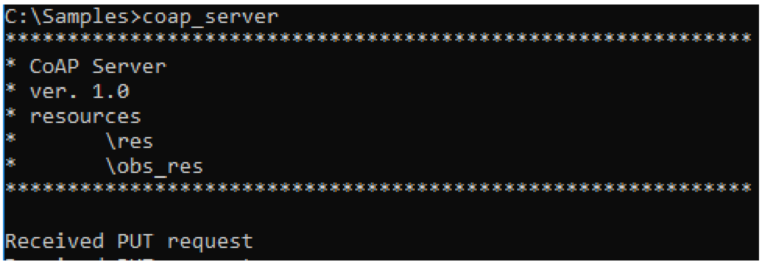


Figure 40: PUT Method of CoAP Client #2

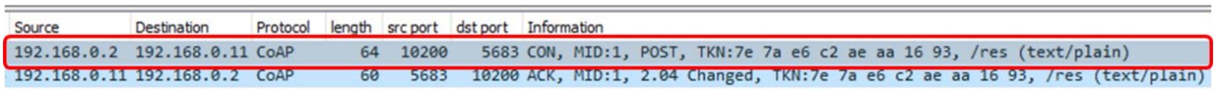


Figure 41: PUT Method of CoAP Client #3

4.1.5.5 DELETE Method

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

```
int coap_client_sample_request_delete(coap_client_sample_conf_t *config,
                                      coap_client_sample_request_t *request)
{
        int ret = DA_APP_SUCCESS;
        coap_client_t *coap_client_ptr = &config->coap_client;
        coap_rw_packet_t resp_packet;
```



```
	memset(&resp_packet, 0x00, sizeof(coap_rw_packet_t));

	//set URI
	ret = coap_client_set_uri(coap_client_ptr,
	                          (unsigned char *)request->uri,
	                           request->urilen);

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

	//send coap request
	ret = coap_client_request_delete_with_port(coap_client_ptr, config->req_port);

	//receive coap response
	ret = coap_client_recv_response(coap_client_ptr, &resp_packet);

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

end:
	//release coap response
	coap_clear_rw_packet(&resp_packet);

	return ret;
}
```

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 details are as follows.

- `int coap_client_request_delete_with_port(coap_client_t *client_ptr, unsigned int port)`

Prototype	int coap_client_request_delete_with_port(coap_client_t *client_ptr, unsigned int 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

DA16200 CoAP client sample application provides a command to send a DELETE request to the CoAP server. Figure 42, Figure 43, and Figure 44 show the interaction of a DA16200 CoAP client and the CoAP server for a delete request.

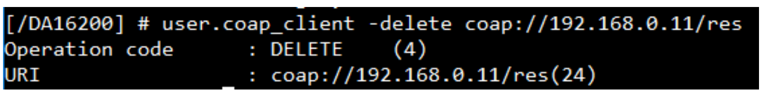


Figure 42: DELETE Method of CoAP Client #1

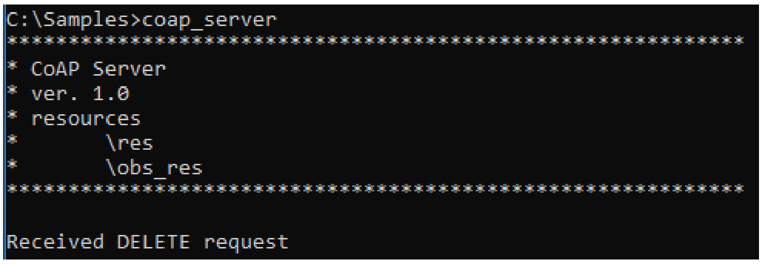
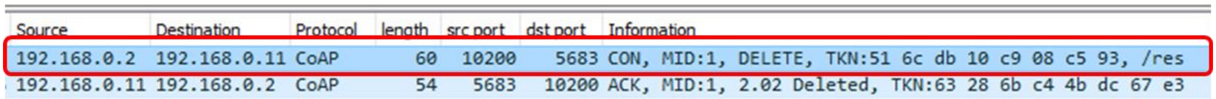


Figure 43: DELETE Method of CoAP Client #2



Source	Destination	Protocol	length	src port	dst port	Information
192.168.0.2	192.168.0.11	CoAP	60	10200	5683	CON, MID:1, DELETE, TKN:51 6c db 10 c9 08 c5 93, /res
192.168.0.11	192.168.0.2	CoAP	54	5683	10200	ACK, MID:1, 2.02 Deleted, TKN:63 28 6b c4 4b dc 67 e3

Figure 44: DELETE Method of CoAP Client #3

4.1.5.6 CoAP PING

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

```
int coap_client_sample_request_ping(coap_client_sample_conf_t *config,
                                    coap_client_sample_request_t *request)
{
      int ret = DA_APP_SUCCESS;
      coap_client_t *coap_client_ptr = &config->coap_client;

      //set URI
      ret = coap_client_set_uri(coap_client_ptr,
                                (unsigned char *)request->uri,
                                 request->urilen);

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

      //progress ping request
      ret = coap_client_ping_with_port(coap_client_ptr, config->req_port);

end:

      return ret;
}
```

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

- `int coap_client_ping_with_port(coap_client_t *client_ptr, unsigned int port)`

Prototype	int coap_client_ping_with_port(coap_client_t *client_ptr, unsigned int 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

The DA16200 CoAP client sample application has a command to send a PING method to the CoAP server. Figure 45 and Figure 46 show the interaction of the DA16200 CoAP client and the CoAP server for a PING request.


```
[/DA16200] # user.coap_client -ping coap://192.168.0.11/res
Operation code        : PING       (7)
URI                   : coap://192.168.0.11/res(24)
```

Figure 45: PING Method of CoAP Client #1

Source	Destination	Protocol	length	src port	dst port	Information
192.168.0.2	192.168.0.11	CoAP	60	10200	5683	CON, MID:3, Empty Message, TKN:7e 7a e6 c2 ae aa 16 95, /res
192.168.0.11	192.168.0.2	CoAP	46	5683	10200	RST, MID:3, Empty Message

Figure 46: PING Method of CoAP Client #2

4.1.5.7 CoAP Response

DA16200 constructs a CoAP response in coap_rw_packet_t structure. In this section, details are given of how a CoAP response is constructed.

~/SDK/core/coap/coap_common.h

```
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 constructs it. The coap_rw_packet_t structure can 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

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

- `int coap_client_recv_response(coap_client_t *client_ptr, coap_rw_packet_t *resp_ptr)`

Prototype	int coap_client_recv_response(coap_client_t *client_ptr, coap_rw_packet_t *resp_ptr)
Description	Receive CoAP response for specific CoAP request
Parameters	client_ptr: CoAP Client instance pointer resp_ptr: CoAP response
Return values	0(NX_SUCCESS) on success

4.1.6 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 from the CoAP server.

4.1.6.1 Registration

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

```
int coap_client_sample_register_observe(coap_client_sample_conf_t *config,
                                        coap_client_sample_request_t *request)
{
      int ret = DA_APP_SUCCESS;
      coap_client_t *coap_client_ptr = &config->coap_client;

      //set URI
      ret = coap_client_set_uri(coap_client_ptr,
                                (unsigned char *)request->uri,
                                 request->urilen);

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

      //register coap observe
      ret = coap_client_set_observe_notify_with_port(coap_client_ptr,
                                                     config->obs_port,
                                                     coap_client_sample_observe_notify,
                                                     coap_client_sample_observe_close_notify);

end:

      return ret;
}
```

A DA16200 CoAP observe allows only one connection. After successful registration, a DA16200 CoAP client allows receiving 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 details are as follows.

- `int coap_client_set_observe_notify_with_port(coap_client_t *client_ptr, unsigned int port, int (*observe_notify)(void *client_ptr, coap_rw_packet_t *resp_ptr), void (*observe_close_notify)(char *timer_name))`

Prototype	int coap_client_set_observe_notify_with_port(coap_client_t *client_ptr, unsigned int port, int (*observe_notify)(void *client_ptr, coap_rw_packet_t *resp_ptr), void (*observe_close_notify)(char *timer_name))
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
Return values	0(NX_SUCCESS) on success

The DA16200 CoAP client sample application has a command for CoAP observe. Figure 47, Figure 48, and Figure 49 show the interaction of a DA16200 CoAP client and the CoAP server for CoAP observe. The CoAP server sends an observe notification every five seconds before deregistration.

```
[/DA16200] # user.coap_client -reg_observe coap://192.168.0.11/obs_res
Operation code      : REG_OBSERVE(5)
URI                 : coap://192.168.0.11/obs_res(28)
[/DA16200/user] # [coap_client_sample_observe_notify]Received Observe notification(25)
===== Observe Notification(len:25) =====
0000: 43 6F 41 50 20 4F 62 73 65 72 76 65 20 4E 6F 74   CoAP Observe Not
0010: 69 66 69 63 61 74 69 6F 6E                        ification
```

Figure 47: CoAP Observe of CoAP Client #1

```
C:\Samples>coap_server
***********************************************************
* CoAP Server
* ver. 1.0
* resources
*         \res
*         \obs_res
***********************************************************

Send Observe notification
Send Observe notification
```

Figure 48: CoAP Observe of CoAP Client #2

Source	Destination	Protocol	length	src port	dst port	Information
192.168.0.2	192.168.0.11	CoAP	67	10201	5683	CON, MID:1, GET, TKN:fc 48 1b 0b 13 e7 b1 02, End of Block #0, /obs_res
192.168.0.11	192.168.0.2	CoAP	84	5683	10201	ACK, MID:1, 2.05 Content, TKN:fc 48 1b 0b 13 e7 b1 02, /obs_res (text/plain)
192.168.0.11	192.168.0.2	CoAP	85	5683	10201	CON, MID:46584, 2.05 Content, TKN:fc 48 1b 0b 13 e7 b1 02, /obs_res (text/plain)
192.168.0.2	192.168.0.11	CoAP	60	10201	5683	ACK, MID:46584, Empty Message

Observe nofitication

Figure 49: CoAP Observe of CoAP Client #3

4.1.6.2 Deregistration

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

```
int coap_client_sample_unregister_observe(coap_client_sample_conf_t *config)
{
    int ret = DA_APP_SUCCESS;
    coap_client_t *coap_client_ptr = &config->coap_client;
    coap_client_clear_observe(coap_client_ptr);

    return ret;
}
```


The API details are as follows:

- `VOID coap_client_clear_observe(coap_client_t *coap_client)`

Prototype	VOID coap_client_clear_observe(coap_client_t *coap_client)
Description	Deregister CoAP observe relation
Parameters	coap_client: CoAP Client instance pointer

4.2 DNS Query

4.2.1 How to Run

1. In the Eclipse, import project for the DNS Query sample application as follows:
   - *~/SDK/apps/common/examples/Network/DNS_Query/projects/da16200*
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.

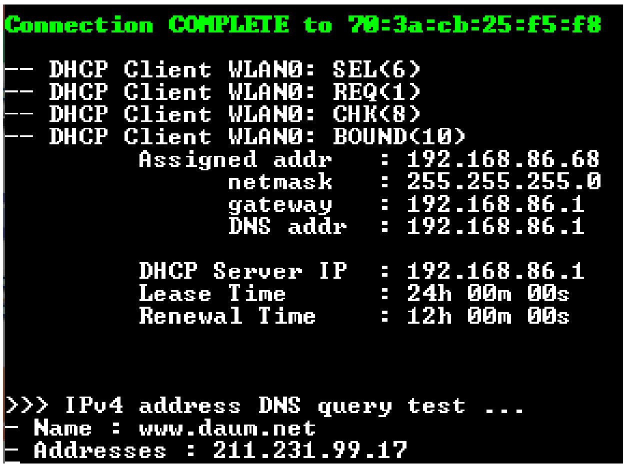


Figure 50: DNS Query Result

4.2.2 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 a single IPv4 address:

```
char *dns_A_Query(char *domain_name, unsigned long wait_option)
```

- Get multiple IPv4 addresses:

```
unsigned int dns_ALL_Query(unsigned char *domain_name,
                           unsigned char *record_buffer,
                           unsigned int record_buffer_size,
                           unsigned int *record_count,
                           unsigned long wait_option)
```

This example creates entry function which is dns_query_sample().

```
void dns_query_sample(void * param)
{
      char    *test_url = NULL;
      if (netmode[WLAN0_IFACE] == DHCPCLIENT)
      {
         // wait until dhcp is done
         while (da16x_network_main_check_dhcp_state(WLAN0_IFACE) != DHCP_STATE_BOUND)
         {
```



```
            vTaskDelay(100);
        }
    }

    vTaskDelay(500);

    /* Check test url */
    test_url = read_nvram_string("TEST_DOMAIN_URL");
    if (test_url == NULL)
    {
     test_url = TEST_URL;
    }

    PRINTF("\n\n");
    dns_A_query_sample(test_url);
    vTaskDelete(NULL);
}
```

4.2.3 Get Single IPv4 Address

This example shows the use of the API function “char *dns_A_Query(char *domain_name, unsigned long wait_option)” to get the IPv4 address string with a domain name URL.

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

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

    /* DNS query with test url string */
    ipaddr_str = dns_A_Query(test_url_str, MAX_DNS_QUERY_TIMEOUT);

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

4.3 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.

4.3.1 How to Run

1. In the Eclipse, import project for the SNTP and current time sample application as follows:
   - *~/SDK/apps/common/examples/ETC/Cur_Time/projects/da16200*
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 an SNTP client with test values.

~/SDK/apps/common/examples/ETC/Cur_Time/src/cur_time_sample.c

```
#define      TEST_SNTP_SERVER            "time.windows.com"
#define      TEST_SNTP_RENEW_PERIOD      600
#define      TEST_TIME_ZONE             (9 * 3600)      // seconds
#define      SNTP_ENABLE                 1
#define      ONE_SECONDS                 100
#define      CUR_TIME_LOOP_DELAY         10              // seconds
```

NOTE

- If the SNTP client is started with predefined values, this configuration is ignored.
- The legacy AP must be connected to the internet.

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

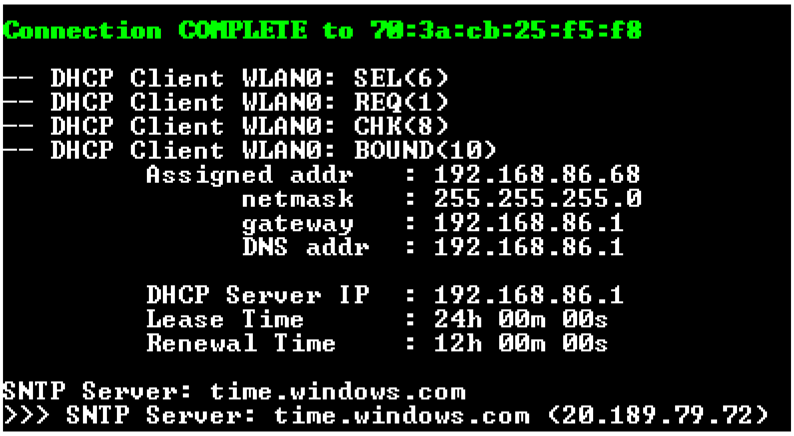


Figure 51: Result of DA16200 SNTP #1

DA16200 periodically gets the current time (the test period: 10 seconds)

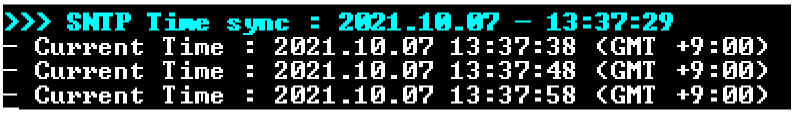


Figure 52: Result of DA16200 SNTP #2

4.3.2 Operation

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

~/SDK/apps/common/examples/ETC/Cur_Time/src/cur_time_sample.c

```
void cur_time_sample(void * param)
{
    unsigned char         status;
    __time64_t     now;
    struct tm *ts;
    char    time_buf[80];

    /* Config SNTP client */
    status = set_n_start_SNTP();
    if (status == pdFAIL) {
            PRINTF("[%s] Faile to start SNTP client ...\n", __func__);
            vTaskDelete(NULL);
            return;
```



```
        }
    }
```

2. If the SNTP client has already been started with predefined values, then skip this configuration. Set the SNTP server address, time update period, and time zone and finally enable the function.

~/SDK/apps/common/examples/ETC/Cur_Time/src/cur_time_sample.c

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

    /* Check current SNTP running status */
    status = getSNTPuse();

    if (status == TX_TRUE)
    {
            /* Already SNTP module running ... */
            return TX_SUCCESS;
    }

    /* Config and save SNTP server domain */
    status = (unsigned int)setSNTPsrv(TEST_SNTP_SERVER, 0);

    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;
        }

        da16x_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;
        }

    _exit :
        return status;
    }
```

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

~/SDK/apps/common/examples/ETC/Cur_Time/src/cur_time_sample.c

```
void cur_time_sample(void * param)
{
    ...
    while (1) {
        /* delay */
        vTaskDelay(CUR_TIME_LOOP_DELAY * ONE_SECONDS);

        /* get current time */
        da16x_time64(NULL, &now);
        ts = (struct tm *)da16x_localtime64(&now);

        /* make time string */
        da16x_strftime(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,
                da16x_Tzoff() / 3600,
                da16x_Tzoff() % 3600);
    }

}
```

4.4 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.

4.4.1 How to Run

1. In the Eclipse, import project for the SNTP and the current time in the DPM sample application as follows:
   - *~/SDK/apps/common/examples/ETC/Cur_Time_DPM/projects/da16200*
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 an SNTP client with test values.

~/SDK/apps/common/examples/ETC/Cur_Time_DPM/src/cur_time_dpm_sample.c

```
#define     TEST_SNTP_SERVER            "time.windows.com"
#define     TEST_SNTP_RENEW_PERIOD      600
#define     TEST_TIME_ZONE             (9 * 3600)      // seconds
#define     SNTP_ENABLE                 1
#define     ONE_SECONDS                 100
#define     CUR_TIME_LOOP_DELAY         10              // seconds
```

NOTE

- If the SNTP client is started with pre-defined values, this configuration is ignored.
- The legacy AP must be connected to the internet

5. After a connection is made to the SNTP server, DA16200 shows the connection result on the debug console and goes to DPM sleep mode.

```
Connection COMPLETE to 70:3a:cb:25:f5:f8

-- DHCP Client WLAN0: SEL(6)
-- DHCP Client WLAN0: REQ(1)
-- DHCP Client WLAN0: CHK(8)
-- DHCP Client WLAN0: BOUND(10)
         Assigned addr   : 192.168.86.68
               netmask   : 255.255.255.0
               gateway   : 192.168.86.1
               DNS addr  : 192.168.86.1

         DHCP Server IP  : 192.168.86.1
         Lease Time      : 24h 00m 00s
         Renewal Time    : 20h 00m 00s

>>> SNTP Server: time.windows.com (20.189.79.72)

>>> SNTP Time sync : 2021.10.07 - 13:48:05
>>> Start DPM Power-Down !!!
```

Figure 53: Result of DA16200 SNTP DPM #1

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

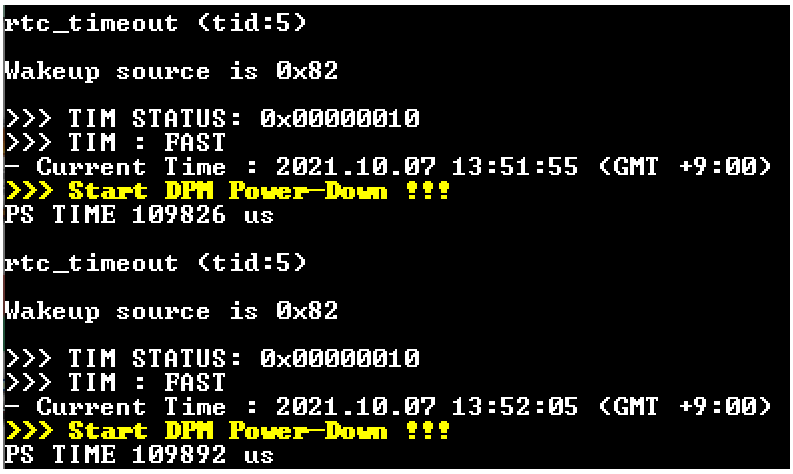


Figure 54: Result of DA16200 SNTP DPM #2

4.4.2 Operation

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

```
~/SDK/apps/common/examples/ETC/Cur_Time_DPM/src/cur_time_dpm_sample.c
static unsigned char set_n_start_SNTP(void)
{
      unsigned char  status = pdPASS;

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

      if (status == pdPASS)
      {
            long       time_zone;

            /* Already SNTP module running, set again time-zone ... */
            time_zone = get_timezone_from_rtm();
            da16x_SetTzoff(time_zone);

            return pdPASS;
      }

      if (dpm_mode_is_wakeup() == NORMAL_BOOT)
      {
            /* Config and save SNTP server URI */
            status = set_sntp_server(TEST_SNTP_SERVER, 0);

      if (status != pdPASS)
      {
           PRINTF("[%s] Failed to write nvram operation (SNTP server domain)...\n",
                   __func__);
           status = pdFAIL;
           goto _exit;
```

```
        }

        /* Config and save SNTP periodic renew time : seconds */
        status = set_sntp_period(TEST_SNTP_RENEW_PERIOD);

        if (status != pdPASS)
        {
             PRINTF("[%s] Failed to write nvram operation (SNTP renew period)...\n",
                     __func__);
             status = pdFAIL;
             goto _exit;
        }

        /* Config and save SNTP time zone */
        set_time_zone(TEST_TIME_ZONE);
        set_timezone_to_rtm(TEST_TIME_ZONE);
        da16x_SetTzoff(TEST_TIME_ZONE);
        set_time_zone(TEST_TIME_ZONE);

        /* Config, save, and run SNTP client */
        if (set_sntp_use(SNTP_ENABLE) != 0) {
             PRINTF("[%s] Failed to run SNTP...\n", __func__);
             status = pdFAIL;
             goto _exit;
        }

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

  _exit :
        return status;
  }
```

When connected to the SNTP server, DA16200 starts an RTC timer to periodically get the current time.

```
  ~/SDK/apps/common/examples/ETC/Cur_Time_DPM/src/cur_time_dpm_sample.c
  void cur_time_dpm_sample(void * param)
  {
        ...
        /* Regist periodic RTC Timer : Get current time */
        if (dpm_mode_is_wakeup() == NORMAL_BOOT)
        {
            /* Time delay for stable running SNTP client */
            vTaskDelay(10);

            status = dpm_timer_create(SAMPLE_CUR_TIME_DPM,
                                      "timer1",
                                      display_cur_time,
                                      CUR_TIME_LOOP_DELAY,
                                      CUR_TIME_LOOP_DELAY);
        }

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

        vTaskDelete(NULL);
  }
```

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

```
~/SDK/apps/common/examples/ETC/Cur_Time_DPM/src/cur_time_dpm_sample.c
static void display_cur_time(char *timer_name)
{
	dpm_app_wakeup_done(SAMPLE_CUR_TIME_DPM);

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

	/* get current time */
	da16x_time64(NULL, &now);
	ts = (struct tm *)da16x_localtime64(&now);

	/* make time string */
	da16x_strftime(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,
	    da16x_Tzoff() / 3600,
	    da16x_Tzoff() % 3600);

	vTaskDelay(1);

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

4.5 HTTP Client

The DA16200 SDK has a ported lwIP 2.1.2 stack. With this product, an application programmer can develop an HTTP client application that uses lwIP HTTP APIs.

4.5.1 How to Run

1. In the Eclipse, import project for the HTTP_Client sample application as follows:
   - *~/SDK/apps/common/examples/Network/Http_Client/projects/da16200*
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 and connect to the AP that is connected to the Internet.
4. Complete the setup and (re)start the sample.

4.5.2 Operation

The sample code is an example of the -get and -post methods. When the sample starts by default, it is executed as a -get method request. To request -post, define ENABLE_METHOD_POST_TEST at the top of the sample code.

The URL and data of the sample code are just examples. Modify it according to the user environment.

To connect to HTTPS(TLS) server, 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).

This sample code is executed as follows:

1. Using the *http_client_parse_uri()* API, set the port number for HTTP or HTTPS and parse the *path* and *host_name*.


```
unsigned char g_http_url[256] = {"http://httpbin.org/get"};
error = http_client_parse_uri(g_http_url, strlen((char *)g_http_url), &request);
if (error != ERR_OK)
{
    PRINTF("Failed to set URI(error=%d) \r\n", error);
    goto finish;
}
```

2. Set a variable in the httpc_connection_t type and set the value to be passed to the API.
3. If the user registers the callback function in *headers_done_fn* and *result_fn*, the header response received from the server and the result value of http-clinet can be returned.

```
httpc_connection_t conn_settings;
conn_settings.use_proxy = 0;
conn_settings.altcp_allocator = NULL;
conn_settings.headers_done_fn = httpc_cb_headers_done_fn;
conn_settings.result_fn = httpc_cb_result_fn;
```

4. When *ENABLE_METHOD_POST_TEST* is defined, users can insert the data they want to send to the server using the *httpc_insert_send_data()* API.

```
#if defined (ENABLE_METHOD_POST_TEST)
error = httpc_insert_send_data("POST", user_post_data, strlen(user_post_data));
if (error != ERR_OK)
{
    PRINTF("Failed to insert data\n");
}
#endif
```

5. To perform TLS communication with the HTTP Server that requires the HTTP Client's certificate, define ENABLE_HTTPS_SERVER_VERIFY_REQUIRED. The certificate must have been previously stored in the TLS area of the DA16200 sflash.

```
if (conn_settings.insecure)
{
    altcp_mbedtls_set_auth_mode(MBEDTLS_SSL_VERIFY_OPTIONAL);
#ifdef ENABLE_HTTPS_SERVER_VERIFY_REQUIRED
    memset(&conn_settings.tls_settings, 0x00, sizeof(httpc_secure_connection_t));
    http_client_read_certs(&conn_settings.tls_settings);
    conn_settings.tls_settings.auth_mode = MBEDTLS_SSL_VERIFY_REQUIRED;
    conn_settings.tls_settings.incoming_len = HTTPC_MIN_OUTGOING_LEN;
    conn_settings.tls_settings.outgoing_len = HTTPC_MAX_OUTGOING_LEN;
#endif //ENABLE_HTTPS_SERVER_VERIFY_REQUIRED
}
```

6. Call API for get request. User calls httpc_get_file() or httpc_get_file_dns() depending on whether hostname needs a DNS query. If the request is successful, the user can receive payload data through the registered httpc_cb_recv_fn callback function.

```
if (isvalidip((char *)request.hostname))
{
    ip4addr_aton(request.hostname, &server_addr);
    error = httpc_get_file(&server_addr,
                           request.port,
                           url,
                           &conn_settings,
                           httpc_cb_recv_fn,
                           NULL,
                           &connection);
}
else
{
    error = httpc_get_file_dns(&request.hostname,
                               request.port,
                               &request.path,
```



```
                                              &conn_settings,
                                              httpc_cb_recv_fn,
                                              NULL,
                                              &connection);
}
```

7. The httpc_cb_recv_fn() callback function receives a pbuf pointer. p->payload is the data received from the server.

```
static err_t httpc_cb_recv_fn(void *arg, struct tcp_pcb *tpcb,
                              struct pbuf *p, err_t err)
{
if (p == NULL)
{
    PRINTF("\n[%s:%d] Receive data is NULL !! \r\n", __func__, __LINE__);
    return ERR_BUF;
}
else
{
    PRINTF("\n[%s:%d] Received length = %d \r\n", __func__, __LINE__, p->len);
    hexa_dump_print("Received data \r\n", p->payload,
                    p->len, 0, OUTPUT_HEXA_ASCII);
}
    return ERR_OK;
}
```

4.6 HTTP Client in DPM Function

The DA16200 SDK has a ported lwIP 2.1.2 stack. With this product, an application programmer can develop an HTTP client application that uses lwIP HTTP APIs.

4.6.1 How to Run

1. In the Eclipse, import project for the HTTP_Client sample application as follows:
   - *~/SDK/apps/common/examples/Network/Http_Client_DPM/projects/ da16200*
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 and connect to the AP that is connected to the Internet.
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. When the sample starts by default, it is executed as a -get method request. To request -post, define ENABLE_METHOD_POST_TEST at the top of the sample code.

The URL and data of the sample code are just examples. Modify it according to the user environment.

To connect to HTTPS(TLS) server, 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).

This sample code is executed as follows:

1. If an application that uses the HTTP protocol is registered in DPM, a setting must be made not to enter DPM_SLEEP while HTTP transmission (request/response) is in progress. Set DPM_SLEEP after all transfers are complete.

```
void http_client_dpm_sample_entry(void * param)
{
    ...
```



```
        dpm_app_register(HTTP_CLIENT_SAMPLE_TASK_NAME, request.port);
        dpm_app_sleep_ready_clear(HTTP_CLIENT_SAMPLE_TASK_NAME);
        ...
    }

    static void httpc_cb_result_fn(void *arg, httpc_result_t httpc_result, u32_t
                                   rx_content_len, u32_t srv_res, err_t err)
    {
        PRINTF("\n[%s:%d]httpc_result: %d, received: %d byte\r\n", __func__, __LINE__,
                                                                   httpc_result,
                                                                   rx_content_len);
        dpm_app_sleep_ready_set(HTTP_CLIENT_SAMPLE_TASK_NAME);
        return;
    }
```

2. Using the *http_client_parse_uri()* API, set the port number for HTTP or HTTPS and parse the *path* and *host_name*.

```
unsigned char g_http_url[256] = {"http://httpbin.org/get"};
error = http_client_parse_uri(g_http_url, strlen((char *)g_http_url), &request);
if (error != ERR_OK)
{
    PRINTF("Failed to set URI(error=%d) \r\n", error);
    goto finish;
}
```

3. Set a variable in the httpc_connection_t type and set the value to be passed to the API.
4. If the user registers the callback function in *headers_done_fn* and *result_fn*, the header response received from the server and the result value of http-client can be returned.

```
httpc_connection_t conn_settings;
conn_settings.use_proxy = 0;
conn_settings.altcp_allocator = NULL;
conn_settings.headers_done_fn = httpc_cb_headers_done_fn;
conn_settings.result_fn = httpc_cb_result_fn;
```

5. When *ENABLE_METHOD_POST_TEST* is defined, users can insert the data they want to send to the server using the *httpc_insert_send_data()* API.

```
#if defined (ENABLE_METHOD_POST_TEST)
error = httpc_insert_send_data("POST", user_post_data, strlen(user_post_data));
if (error != ERR_OK)
{
    PRINTF("Failed to insert data\n");
}
#endif
```

5. To perform TLS communication with the HTTP Server that requires the HTTP Client's certificate, define *ENABLE_HTTPS_SERVER_VERIFY_REQUIRED*. The certificate must have been previously stored in the TLS area of the DA16200 sflash.

```
if (conn_settings.insecure)
{
    altcp_mbedtls_set_auth_mode(MBEDTLS_SSL_VERIFY_OPTIONAL);
#ifdef ENABLE_HTTPS_SERVER_VERIFY_REQUIRED
    memset(&conn_settings.tls_settings, 0x00, sizeof(httpc_secure_connection_t));
    http_client_read_certs(&conn_settings.tls_settings);
    conn_settings.tls_settings.auth_mode = MBEDTLS_SSL_VERIFY_REQUIRED;
    conn_settings.tls_settings.incoming_len = HTTPC_MIN_OUTGOING_LEN;
    conn_settings.tls_settings.outgoing_len = HTTPC_MAX_OUTGOING_LEN;
#endif //ENABLE_HTTPS_SERVER_VERIFY_REQUIRED
}
```

6. Call API for get request. User calls *httpc_get_file()* or *httpc_get_file_dns()* depending on whether hostname needs a DNS query. If the request is successful, the user can receive payload data through the registered *httpc_cb_recv_fn* callback function.

```
if (isvalidip((char *)request.hostname))
{
    ip4addr_aton(request.hostname, &server_addr);
    error = httpc_get_file(&server_addr,
                           request.port,
                           url,
                           &conn_settings,
                           httpc_cb_recv_fn,
                           NULL,
                           &connection);
}
else
{
    error = httpc_get_file_dns(&request.hostname,
                               request.port,
                               &request.path,
                               &conn_settings,
                               httpc_cb_recv_fn,
                               NULL,
                               &connection);
}
```

7. The httpc_cb_recv_fn() callback function receives a pbuf pointer. p->payload is the data received from the server.

```
static err_t httpc_cb_recv_fn(void *arg, struct tcp_pcb *tpcb,
                              struct pbuf *p, err_t err)
{
if (p == NULL)
{
    PRINTF("\n[%s:%d] Receive data is NULL !! \r\n", __func__, __LINE__);
    return ERR_BUF;
}
else
{
    PRINTF("\n[%s:%d] Received length = %d \r\n", __func__, __LINE__, p->len);
    hexa_dump_print("Received data \r\n", p->payload,
                    p->len, 0, OUTPUT_HEXA_ASCII);
}
    return ERR_OK;
}
```

4.7 HTTP Server

The DA16200 SDK has a ported lwIP 2.1.2 stack. With this product, an application programmer can develop an HTTP server application that uses lwIP HTTP APIs.

4.7.1 How to Run

1. In the Eclipse, import project for the HTTP_Server sample application as follows:
   - *~/SDK/apps/common/examples/Network/Http_Server/projects/da16200*
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 and connect to the AP.
4. Complete the setup and (re)start the sample.

4.7.2 Operation

The sample code is an example of the -get 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 (tls_srv_sample_cert, tls_srv_sample_key) as needed.

```
/// HTTPS server
#define ENABLE_HTTPS_SERVER
```

2. The HTTP server can be operated by simply calling the httpd_init() API.

```
httpd_init();
PRINTF("[%s] HTTP-Server Start!! \r\n", __func__);
end_of_task:

while (1)
{
    vTaskDelay(100);
}
```

3. The HTTPS server must set the key and cert information required for TLS.

```
struct altcp_tls_config *tls_srv_sample_config = NULL;
...
tls_srv_sample_config =
altcp_tls_create_config_server_privkey_cert(tls_srv_sample_key,
                                            tls_srv_sample_key_len,
                                            NULL,
                                            0,
                                            tls_srv_sample_cert,
                                            tls_srv_sample_cert_len);
if (!tls_srv_sample_config)
{
    PRINTF("[%s] Failed to create tls config\r\n", __func__);
    goto end_of_task;
}
httpd_inits(tls_srv_sample_config);
PRINTF("[%s] HTTPS-Server Start!! \r\n", __func__);
end_of_task:
while (1)
{
    vTaskDelay(100);
}
```

4. 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.
   - Accessing from a Web browser

   http://[Server IP]/index.html
   - The page that appears is located below

   [~/sdk/libraries/3rdloty/lwip/src/src/apps/http/fs/index.html]

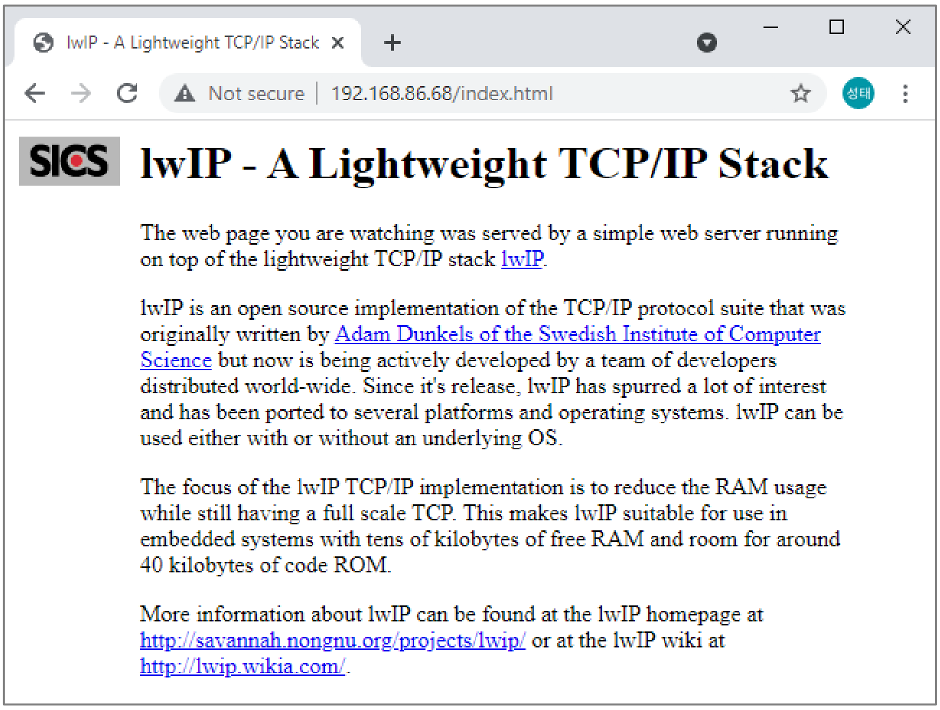


Figure 55: Result of DA16200 HTTP Server

4.8 WebSocket Client

This section describes the behavior of the example Websocket Client application and how to build it.

NOTE

WebSocket Client does not support DPM mode.

4.8.1 How to Run

1. In the Eclipse, import project for the Websocket_Client application as follows:
   ○ *~/SDK/apps/common/examples/Network/WebSocket_Client/projects/da16200*
2. To set the WebSocket Server uri in the WebSocket Client Sample, edit the source code: *~/SDK/apps/common/examples/Network/WebSocket_Client/src/websocket_client_sample.c*

   `#define WEBSOCKET_SERVER_URI` *“ws(wss)://xxxx.xxxx.xxxx”*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. Use the console to set up the Wi-Fi station interface and connect to the AP that is connected to the Internet.
5. Complete the setup and (re)start the sample.

4.8.2 Operation

When the Websocket client application starts, it tries to connect to a WebSocket Server and send a message 10 times.

The URI and data of the example code are for demonstration purposes and can be modified as required to create a custom application.

To use the WebSocket Secure connection, enter "wss://" instead of "ws://" in the URI.

This example code is executed as follows:

1. Set *websocket_cfg.uri* for the WebSocket Server URI and WebSocket Initialize with the websocket configurations.

```
websocket_cfg.uri = WEBSOCKET_SERVER_URI;
WS_LOGI(TAG, "Connecting to %s...\n", websocket_cfg.uri);
websocket_client_handle_t client = websocket_client_init(&websocket_cfg);
```

2. To receive event data, register *websocket_client_event_callback* function before starting the WebSocket Client.

   static void ws_event_handler (websocket_client_event_id_t event_id, websocket_client_event_data_t *event_data)

```
{
  websocket_client_event_data_t *data = (websocket_client_event_data_t
*)event_data;

    switch (event_id) {
    case WEBSOCKET_CLIENT_EVENT_CONNECTED:
        WS_LOGW(TAG, "WEBSOCKET_CONNECTED\n");
        break;
    case WEBSOCKET_CLIENT_EVENT_DISCONNECTED:
        WS_LOGW(TAG, "WEBSOCKET_DISCONNECTED\n");
        break;
    case WEBSOCKET_CLIENT_EVENT_DATA:
          …
          …
        if(data->op_code == WS_TRANSPORT_OPCODES_CLOSE)
        {
          WS_LOGW(TAG, "Websocket Server Closed\n");
          websocket_client_abort_connection(data->client);
        }
        xTimerReset(shutdown_signal_timer, portMAX_DELAY);
        break;
    case WEBSOCKET_CLIENT_EVENT_ERROR:
        WS_LOGE(TAG, "WEBSOCKET_ERROR\n");
        break;
    }
}

websocket_client_start(client, ws_event_handler);
```

3. When the WebSocket Client connects to the server, it sends a message 10 times using *websocket_client_send_text()* API.

   If no event data is received for 5 seconds, *shutdown_signal_timer* disconnects the WebSocket connection.

```
while (i < 10) {
    if (websocket_client_is_connected(client)) {
        int len = sprintf(data, "hello %04d", i++);
        WS_LOGI(TAG, "Sending %s\n", data);
        websocket_client_send_text(client, data, len, portMAX_DELAY);
    }
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

4. *shutdown_signal_timer* disconnects the WebSocket connection using the `websocket_client_stop()API.`

```
if(websocket_client_stop(client)==WS_OK)
{
   WS_LOGI(TAG, "Websocket Stopped\n");
•      }
```


4.9 OTA FW Update

The DA16200 (DA16600) supports over the air (OTA) firmware update using the HTTP protocol. The DA16200 (DA16600) operates as an HTTP client which can download and update new firmware from an HTTP server.

The DA16200 firmware image set consists of Bootloader (secondary bootloader) and RTOS. The boot loader cannot be updated via OTA, only RTOS. With this product, an application programmer can develop an OTA FW update application that uses OTA APIs.

In addition, users can update certificates (TLS Certificate Key #0 and TLS Certificate Key #1) and support firmware update of MCU connected by UART1.

Users can easily develop these functions using the API provided by the DA16200 (DA16600) SDK.

> **NOTE**
>
> When DPM mode is enabled and an OTA (firmware) update is in progress, DPM Sleep Mode will not be entered due to SFLASH write operations.
>
> Once the firmware update is complete, DPM sleep mode will return to normal operation.

4.9.1 How to Run

1. In the Eclipse, import project for the OTA_Update sample application as follows:
   - *~/SDK/apps/common/examples/Network/OTA_Update/projects/da16200*
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 and connect to the AP that is connected to the Internet.
4. Complete the setup and (re)start the sample.

4.9.2 Operation

The sample code includes three examples of updating the DA16200's firmware (SLIB and RTOS), certificates (TLS Certificate Key #0 and TLS Certificate Key #1), and MCU firmware. Each example is divided into definitions as follows. By default, only the DA16200 firmware update definition is enabled, certificate and MCU FW update are disabled.

```
#define SAMPLE_UPDATE_DA16_FW

#undef SAMPLE_UPDATE_MCU_FW

#undef SAMPLE_UPDATE_CERT_KEY
```

typedef struct OTA_UPDATE_CONFIG contains arguments to be passed to the OTA update API. Declare and use a global variable of OTA_UPDATE_CONFIG type.

```
static OTA_UPDATE_CONFIG ota_update_conf = { 0, };

static OTA_UPDATE_CONFIG *g_ota_update_conf = (OTA_UPDATE_CONFIG *) &ota_update_conf;
```

4.9.2.1 DA16200 Firmware Update

This is an example of DA16200 firmware update.

1. Be sure to set update_type to OTA_TYPE_RTOS.

```
/* Setting the type to be updated */
g_ota_update_conf->update_type = OTA_TYPE_RTOS;
```

2. Set uri_other to suit the user environment.

```
/* URI setting example - Change it to suit your environment. */

memcpy(g_ota_update_conf->uri, ota_server_uri_rtos, strlen(ota_server_uri_rtos));
```

- If the download completes successfully, the user can set it to automatically activate RENEW.

```
g_ota_update_conf->auto_renew = 1;
```

3. By registering a callback function in download_complete_notify(), the user can be notified whether the download succeeds or fails. SLIB and RTOS are notified separately. Users can check whose notification is by update_type.

```
g_ota_update_conf->download_complete_notify = user_sample_da16_fw_download_notify;
```

4. Users can be notified of the RENEW status by registering a callback function. Unlike download notification, it is notified only once. If the notification status is successful, the DA16200 automatically reboots after 2-3 seconds.

```
g_ota_update_conf->renew_notify = user_sample_da16_fw_renew_notify;
```

5. Finally, call the OTA update start API. When ota_update_start_download() is called, an OTA update task is created internally and the creation status of the task is immediately returned. The process is not blocked.

```
status = ota_update_start_download(g_ota_update_conf);
```

6. If the firmware has been successfully updated, the DA16200 will reboot.

4.9.2.2 Certificates Update

This is an example of a certificate update.

1. Set URI to suit the user environment.

```
/* URI setting example - Change it to suit your environment. */
memcpy(g_ota_update_conf->uri, ota_server_uri_cert, strlen(ota_server_uri_cert));
```

2. Be sure to set update_type to OTA_TYPE_CERT_KEY.

```
g_ota_update_conf->update_type = OTA_TYPE_CERT_KEY;
```

3. Set the address of SFLASH to be saved when downloading. If not set, the default is SFLASH_USER_AREA_0_START.

```
g_ota_update_conf->download_sflash_addr = SFLASH_USER_AREA_0_START;
```

4. Register a callback to be notified of the download status.

```
g_ota_update_conf->download_notify = user_sample_cert_key_download_notify;
```

5. Finally, call the OTA update start API. When ota_update_start_download() is called, an OTA update task is created internally and the creation status of the task is immediately returned. The process is not blocked.

```
status = ota_update_start_download(g_ota_update_conf);
```

6. If the download is successful, copy them to the TLS Certificate Key #0 and TLS Certificate Key #1 areas.

```
status = ota_update_copy_flash(SFLASH_ROOT_CA_ADDR1, g_ota_update_conf-
>download_sflash_addr, 4096);
```

4.9.2.3 MCU Firmware Update

This is an example of an MCU firmware update.

1. Set URI to suit the user environment.

```
/* URI setting example - Change it to suit your environment. */
memcpy(g_ota_update_conf->uri, ota_server_uri_mcu, strlen(ota_server_uri_mcu));
```

2. Be sure to set update_type to OTA_TYPE_MCU_FW.

```
g_ota_update_conf->update_type = OTA_TYPE_MCU_FW;
```

3. Set the address of SFLASH to be saved when downloading. If not set, the default is SFLASH_USER_AREA_0_START.

```
g_ota_update_conf->download_sflash_addr = SFLASH_USER_AREA_0_START;
```

4. Register a callback to be notified of the download status.

```
g_ota_update_conf->download_notify = user_sample_mcu_fw_download_notify;
```

5. Finally, call the OTA update start API. When ota_update_start_download() is called, an OTA update task is created internally and the creation status of the task is immediately returned. The process is not blocked.

```
status = ota_update_start_download(g_ota_update_conf);
```

6. If the download is successful, initialize UART to transmit the firmware to the MCU.

```
ota_update_uart1_init();
```

7. Start UART protocol for communication with MCU.

```
status = ota_update_uart_trans_mcu_fw();
```

5 Additional Examples

5.1 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.

5.1.1 How to Run

1. In the Eclipse, import project for the RTC timer sample application as follows:
   ○ *~/SDK/apps/common/examples/Peripheral/RTC_Timer_DPM/projects/da16200*
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 and enable DPM mode.
4. After boot, the RTC timer sample application starts automatically.

Notice that, the user can select DPM Sleep mode in the sample code.

```
/* 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
```

5.1.2 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.

```
/* 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);
        return;
    }

    /* Initialize allocated retention-memory buffer */
    memset(rtc_sample_info, 0x00, sizeof(rtc_sample_info_t));

} else if (rtm_len != sizeof(rtc_sample_info_t)) {
    PRINTF("[%s] Invalid RTM alloc size (%d)\n", __func__, rtm_len);

    dpm_app_unregister(SAMPLE_RTC_TIMER);

    return;
}
```

5.1.3 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 wake-up resource and run the same as Power-on-Reset.

To go to DPM Sleep mode 1, just run API dpm_sleep_start_mode_1().

```
void rtc_timer_sample(void * param)
{
      /* FALSE : Not maintain RTM area for DPM operation */
      dpm_sleep_start_mode_1(TRUE);
}
```

5.1.4 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 wake-up 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().

```
void rtc_timer_sample(void * param)
{
    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);
}
```

5.1.5 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, see the Getting Started Guide [2].

This sample code shows how to create a one-shot RTC timer and a periodic RTC timer.

```
void rtc_timer_sample(void * param)
{
    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");
```



```
        vTaskDelay(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");
        vTaskDelay(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... */
        vTaskDelay(100);
     }
}
```


5.2 Get SCAN Result Sample

5.2.1 How to Run

1. In the Eclipse, import project for the SCAN result sample application as follows:
   - *~/SDK/apps/common/examples/ETC/Get_Scan_Result/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. After the boot is complete, the get_scan_result sample starts automatically.

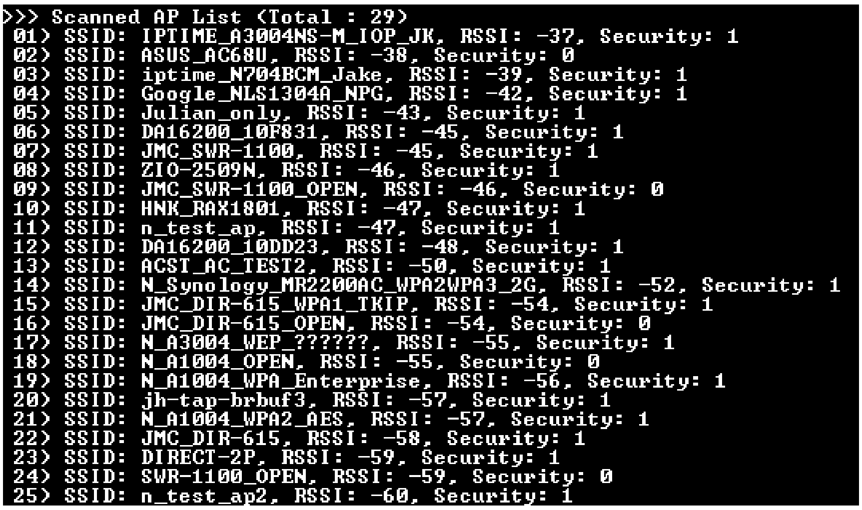


Figure 56: Get_scan_result AP List

5.2.2 Sample Overview

This sample shows how to use the void get_scan_result(void *user_buf_ptr) API, to get the SCAN result on STA mode and Soft-AP mode.

5.2.3 Application Initialization

The get_scan_result_sample function executes after the basic FreeRTOS initialization is complete. This sample just calls the user API “void get_scan_result()”.

```
void get_scan_result_sample(void * param)
{
    char       *user_buf = NULL;
    scan_result_t    *scan_result;
    int      i;

    /* Allocate buffer to get scan result */
    user_buf = pvPortMalloc(SCAN_RSP_BUF_SIZE);

    /* Get scan result */
    get_scan_result((void *) user_buf);

    … …
}
```

5.2.4 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.

```
/* 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++) {
    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 */
vPortFree(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;
```


6 Crypto Examples

6.1 Crypto API

The Crypto API sample application demonstrates common use case of crypto algorithms such as AES, DES, and Hash, and so on. The DA16200 SDK includes an "mbedTLS" library which is an implementation of the TLS and SSL protocols and the respective cryptographic algorithms.

This section describes how it is built and works.

6.1.1 How to Run

1. In the Eclipse, import project for the Crypto API sample application as follows:
   - *~/SDK/apps/common/examples/Crypto/Crypto_API/projects/da16200*
2. Enable features of what cryptographic algorithms are required.
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.

6.1.2 How to Enable Cryptographic Algorithm

The Crypto API sample application includes 11 types of cryptographic algorithms. Each type can be enabled by feature definition in crypto_sample.h file as follows. By default, AES cryptographic algorithm is enabled.

AES Algorithms
Cipher API
DES Algorithms
Diffie-Hellman Key Exchange
DRBG
ECDH
ECDSA
HASH and HMAC Algorithms
Key Derivation Function
Public Key Abstraction Layer
RSA PKCS#1

```
// AES Algorithms
#define
__CRYPTO_SAMPLE_AES__

// Cipher API
#undef
    __CRYPTO_SAMPLE_CIPHER__

// DES Algorithms
#undef
    __CRYPTO_SAMPLE_DES__

// Diffie-Hellman key
    exchange
#undef
    __CRYPTO_SAMPLE_DHM__

// DRBG
#undef
    __CRYPTO_SAMPLE_DRBG__
```

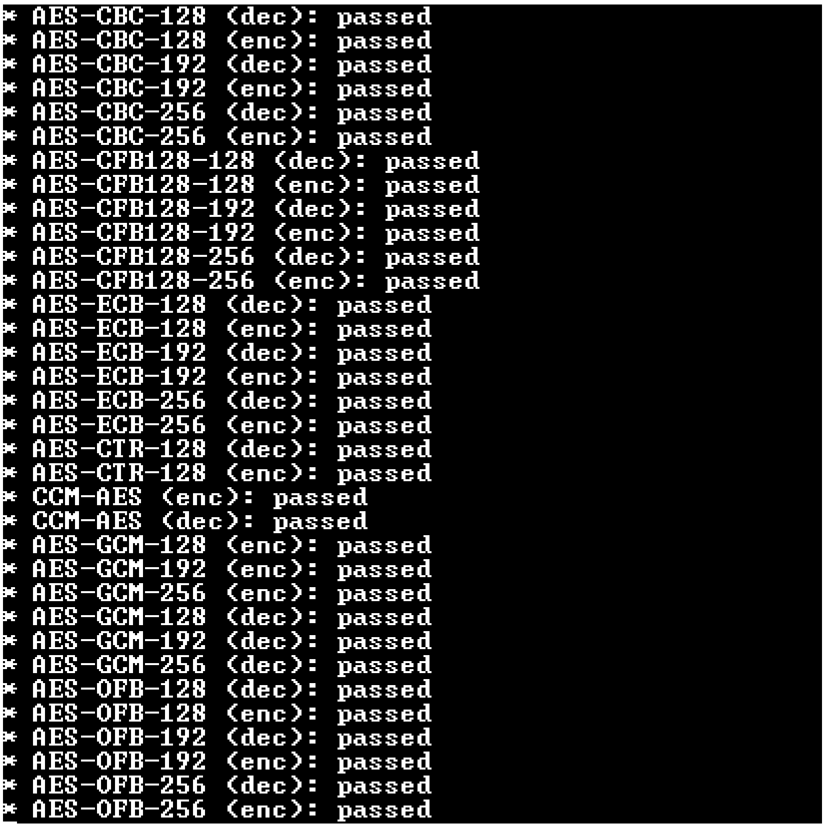


```
// ECDH
#undef
    __CRYPTO_SAMPLE_ECDH__

// ECDSA
#undef
    __CRYPTO_SAMPLE_ECDSA__

// Hash & HMAC Algorithms
#undef
    __CRYPTO_SAMPLE_HASH__

// Key Derivation Function
#undef
    __CRYPTO_SAMPLE_KDF__

// Public Key abstraction
    layer.
#undef
    __CRYPTO_SAMPLE_PK__

// RSA PKCS#1
#undef
__CRYPTO_SAMPLE_RSA__
```

6.1.3 Crypto Algorithms – AES

The AES algorithms sample application demonstrates common use cases of AES ciphers such as CBC, CFB, and ECB, and so on. The sample 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-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
* CCM-AES (enc): passed
* CCM-AES (dec): passed
* AES-GCM-128 (enc): passed
* AES-GCM-192 (enc): passed
* AES-GCM-256 (enc): passed
* AES-GCM-128 (dec): passed
* AES-GCM-192 (dec): passed
* AES-GCM-256 (dec): passed
* AES-OFB-128 (dec): passed
* AES-OFB-128 (enc): passed
* AES-OFB-192 (dec): passed
* AES-OFB-192 (enc): passed
* AES-OFB-256 (dec): passed
* AES-OFB-256 (enc): passed
```

Figure 57: The Result of the Crypto AES

6.1.3.1 Application Initialization

The example below describes how the user uses the AES algorithms of the “mbedTLS” library to encrypt and decrypt data.

```
void crypto_sample_aes(void *param)
{
#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 ;
}
```


6.1.3.2 AES-CBC-128, 192, and 256

DA16200 supports crypto algorithm for AES-CBC-128, 192, and 256. To explain how AES-CBC works, see the test vector in http://csrc.nist.gov/archive/aes/rijndael/rijndael-vals.zip.

```
int crypto_sample_aes_cbc()
{
    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-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, 16, 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, 16, iv, buf, buf);

                memcpy(tmp, prv, 16);
                memcpy(prv, buf, 16);
                memcpy(buf, tmp, 16);
            }
        }
    }

    // Clear the AES context.
    mbedtls_aes_free(ctx);
}
```

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 it 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 complete, function mbedtls_aes_free should be called to clear the AES context.

6.1.3.3 AES-CFB128-128, 192, and 256

DA16200 supports a crypto algorithm for AES-CFB128-128, 192, and 256. To explain how AES-CFB128 works, see the test vector in http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

```
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;

        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 the AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypt_cfb128 does AES-CFB128 encryption or decryption. And it 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 complete, function mbedtls_aes_free should be called to clear the AES context.

6.1.3.4 AES-ECB-128, 192, and 256

DA16200 supports crypto algorithm for AES-ECB-128, 192, and 256. To explain how AES-ECB works, see the test vector in http://csrc.nist.gov/archive/aes/rijndael/rijndael-vals.zip.

```
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_SAMPLE_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 the 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 it 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 complete, function mbedtls_aes_free should be called to clear the AES context.

6.1.3.5 AES-CTR-128

DA16200 supports the 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.

```
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 the AES algorithm. It is initialized by function mbedtls_aes_init. Function mbedtls_aes_crypto_ctr does an AES-CTR encryption or decryption operation. And it 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 complete, call function mbedtls_aes_free to clear the AES context.

6.1.3.6 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.

```
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 complete, call function mbed_ccm_free to release and clear the specified CCM context and underlying cipher subcontext.

6.1.3.7 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++) {
        int key_len = 128 + 64 * 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_crypt_and_tag(ctx, MBEDTLS_GCM_ENCRYPT,
                                   sizeof(crypto_sample_gcm_pt),
                                   crypto_sample_gcm_iv, sizeof(crypto_sample_gcm_iv),
                                   crypto_sample_gcm_additional,
                                   sizeof(crypto_sample_gcm_additional),
                                   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++) {
        int key_len = 128 + 64 * 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,
                                   sizeof(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 complete, function mbed_gcm_free should be called to clear a GCM context and underlying cipher sub-context.

6.1.3.8 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).

```
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;
        } else {
            memcpy(buf, crypto_sample_aes_ofb_pt, 64);
            expected_out = crypto_sample_aes_ofb_ct[u];
        }

        // Perform an AES-OFB (Output Feedback Mode) encryption or decryption
        // operation.
        ret = mbedtls_aes_crypt_ofb(ctx, 64, &offset, iv, stream_block, buf, buf);
    }

    // Clear the AES context.
    mbedtls_aes_free(ctx);
}
```

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 the 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 decryption. The OFB operation is identical for encryption or decryption, therefore no operation mode needs to be specified. After the operation is complete, call function mbedtls_aes_free to clear the AES context.

6.1.4 Crypto Algorithms – DES

The DES algorithm sample application demonstrates common use cases of DES and Triple-DES ciphers. The sample application runs two types of cryptography algorithms:

- DES-CBC-56
- DES3-CBC-112 and 168


```
>>> Start STA mode...
* 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
```

Figure 58: The Result of the Crypto DES

6.1.4.1 Application Initialization

The example below shows how a user uses DES algorithms of the “mbedTLS” library to encrypt and decrypt data.

```
void crypto_sample_des(void *param)
{
#if defined(MBEDTLS_CIPHER_MODE_CBC)
    crypto_sample_des_cbc();
#endif // (MBEDTLS_CIPHER_MODE_CBC)
    return ;
}
```

6.1.4.2 DES-CBC-56, DES3-CBC-112, and 168

DA16200 supports crypto algorithm for DES-CBC-56, DES3-CBC-112, and 168. To explain how DES-CBC and DES3-CBC works, see the test vector in http://csrc.nist.gov/groups/STM/cavp/documents/des/tripledes-vectors.zip.

```
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);

    // Test CBC
    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 complete, call function mbed_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 bits 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 complete, call function mbedtls_des3_free to clear the DES3 context.

6.1.5 Crypto Algorithms – HASH and 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, and so on. The sample application runs six types of hash algorithms and HMAC algorithms:

- SHA1, SHA-224, SHA-256, SHA-384, SHA-512, and MD5
- HMAC

```
>>> Start STA mode...
* SHA-1: passed
* SHA-224: passed
* SHA-256: passed
* SHA-384: passed
* SHA-512: passed
* MD5: passed
* Message-digest Information
>>> MD5: passed
>>> SHA1: passed
>>> SHA224: passed
>>> SHA256: passed
>>> SHA384: passed
>>> SHA512: passed
* Hash with text string
>>> MD5: passed
* Hash with multiple text string
>>> MD5: passed
* HMAC with hex data
>>> MD5: passed
>>> SHA1: passed
>>> SHA224: passed
>>> SHA256: passed
>>> SHA384: passed
>>> SHA512: passed
```

Figure 59: The Result of the Crypto HASH #1

```
* HMAC with multiple hex data
>>> MD5: passed
>>> SHA1: passed
>>> SHA224: passed
>>> SHA256: passed
>>> SHA384: passed
>>> SHA512: passed
* Hash with hex data
>>> SHA1: passed
>>> SHA224: passed
>>> SHA256: passed
>>> SHA384: passed
>>> SHA512: passed
* Hash with multiple hex data
>>> SHA1: passed
>>> SHA224: passed
>>> SHA256: passed
>>> SHA384: passed
>>> SHA512: passed
```

Figure 60: The Result of the Crypto HASH #2

6.1.5.1 Application Initialization

This example describes how the user can use hash and HMAC algorithms of the “mbedTLS” library.

```
void crypto_sample_hash(void *param)
{
	crypto_sample_hash_sha1();

	crypto_sample_hash_sha224();

	crypto_sample_hash_sha256();
```



```
      crypto_sample_hash_sha384();

      crypto_sample_hash_sha512();

#if defined(MBEDTLS_MD5_C)
      crypto_sample_hash_md5();
#endif // (MBEDTLS_MD5_C)

      crypto_sample_hash_md_wrapper();

      return ;
}
```

6.1.5.2 SHA-1 Hash

DA16200 supports a crypto algorithm for the SHA-1 hash. To explain how the SHA-1 hash works, see the test vector in FIPS-180-1.

```
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);

    // Finish 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.

6.1.5.3 SHA-224 Hash

DA16200 supports a crypto algorithm for the SHA-224 hash. To explain how SHA-224 hash works, see the test vector in FIPS-180-2.

```
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 “mbedTLS” 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)
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.

6.1.5.4 SHA-256 Hash

DA16200 supports a crypto algorithm for the SHA-256 hash. To explain how the 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, rypto_sample_hash_sha256_buf,
                              crypto_sample_hash_sha256_buflen);

    // Finishe 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 the SHA-224 code (see Section 6.1.5.3). When starting the SHA-256 checksum calculation, the second parameter should be set to 0 for SHA-256.

6.1.5.5 SHA-384 Hash

DA16200 supports a crypto algorithm for the SHA-384 hash. To explain how the 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(ctx, crypto_sample_hash_sha384_buf,
                          crypto_sample_hash_sha384_buflen);

    // Finishe the SHA-384 operation, and writes the result to the output buffer.
    mbedtls_sha512_finish(ctx, sha384sum);

    // Clear a SHA-384 context.
    mbedtls_sha512_free(ctx);
}
```

The mbedtls_sha512_context is the SHA-512 context structure. “mbedTLS” 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.

6.1.5.6 SHA-512 Hash

DA16200 supports a crypto algorithm for the SHA-512 hash. To explain how the 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);

    // Finishe 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 as Crypto Algorithm for the SHA-384 code (see Section 6.1.5.5). When the SHA-512 checksum calculation is started, the second parameter should be set to 0 for SHA-512.

6.1.5.7 MD5 Hash

DA16200 supports a crypto algorithm for an MD5 hash. To explain how the MD5 hash works, see the 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 is 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

6.1.5.8 Hash and HMAC with the Generic Message-Digest Wrapper

The “mbedTLS” library provides the generic message-digest wrapper to calculate HASH and HMAC functions. The example below shows how HASH and HMAC are calculated with the generic message-digest wrapper functions.

First, the user needs to check which message-digest could be supported by the “mbedTLS” 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);
    if (!md_info) {
        PRINTF("[%s] Unknown Hash Type(%d)\r\n", __func__, md_type);
        goto cleanup;
    }

    // Get the message-digest information associated with the given digest name.
    if (md_info != mbedtls_md_info_from_string(md_name)) {
        PRINTF("[%s] Unknown Hash Name(%s)\r\n", 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) {
```

```
        PRINTF("[%s] Not matched Hash Type\r\n", __func__);
        goto cleanup;
    }

    // Extract the message-digest size from the message-digest information
    // structure.
    if (mbedtls_md_get_size(md_info) != (unsigned char)md_size) {
        PRINTF("[%s] Not matched Hash Size\r\n", __func__);
        goto cleanup;
    }

    // Extract the message-digest name from the message-digest information
    // structure.
    if (strcmp(mbedtls_md_get_name(md_info), md_name) != 0) {
        PRINTF("[%s] Not matched Hash Name\r\n", __func__);
        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:

- const mbedtls_md_info_t* mbedtls_md_info_from_type(mbedtls_md_type_t md_type)

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	unsigned char mbedtls_md_get_size(const mbedtls_md_info_t* md_info)
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 char* mbedtls_md_get_name(const mbedtls_md_info_t* md_info)
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 int* mbedtls_md_list(void)
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.

```
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.
     */
    ret = mbedtls_md(md_info,
                     (const unsigned char *)text_src_string,
                      strlen(text_src_string),
                      output);
}
```

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.

```
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 free 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.

```
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 helps 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	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.

```
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;

    /* Initialize a message-digest context without binding it
     * to a particular message-digest algorithm.
     */
    mbedtls_md_init(ctx);

    md_info = mbedtls_md_info_from_string(md_name);

    // Select the message digest algorithm to use, and allocate 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 free any embedded internal
       structure,
     * 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.

```
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.
    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.
     */
    ret = mbedtls_md(md_info, src_str, src_len, output);
}
```

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.

```
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
     * 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 allocate 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 free any embedded internal
       structure,
     * but does not free ctx itself.
     */

    mbedtls_md_free(ctx);
```



```
}
```

6.1.6 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 sample application explains how to use the DRBG function with CTR and HMAC.

- CTR_DRBG
- HMAC_DRBG

```
* CTR_DRBG (PR = TRUE): passed
* CTR_DRBG (PR = FALSE): passed
* HMAC_DRBG (PR = True) : passed
* HMAC_DRBG (PR = False) : passed
```

Figure 61: The Result of the Crypto DRBG

6.1.6.1 Application Initialization

This example describes how the user uses CTR DRBG and HMAC DRBG of the “mbedTLS” 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 “mbedTLS” implementation of CTR_DRBG uses AES-256 (default) or AES-128 as the underlying block cipher. HMAC_DRBG is based on a Hash-based message authentication code.

```
void crypto_sample_drbg(void *param)
{
    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 ;
}
```

6.1.6.2 CTR_DRBG with Prediction Resistance

This example describes how to use CTR_DRBG with prediction resistance.

```
int crypto_sample_ctr_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, drbg_test_entropy,
                                  (void *)crypto_sample_ctr_drbg_entropy_src_pr,
                                   crypto_sample_ctr_brdg_nonce_pers_pr,
                                   16,
                                   32);

    // Turn prediction resistance on
    mbedtls_ctr_drbg_set_prediction_resistance(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.
    mbedtls_ctr_drbg_free(ctx);
}
```

The API details are as follows:

- `void mbedtls_ctr_drbg_init(mbedtls_ctr_drbg_context* ctx)`

Prototype	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	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	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	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.

6.1.6.3 CTR_DRBG Without Prediction Resistance

This example describes how to use CTR_DRBG without prediction resistance.

```
int crypto_sample_ctr_drbg_pr_off()
```



```
{
    mbedtls_ctr_drbg_context ctx;    //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, drbg_test_entropy,
                              (void *) crypto_sample_ctr_drbg_entropy_src_nopr,
                               crypto_sample_ctr_brdg_nonce_pers_nopr, 16, 32);

    // Generate random data using CTR_DRBG.
    ret = mbedtls_ctr_drbg_random(&ctx, buf, MBEDTLS_CTR_DRBG_BLOCKSIZE);

    // Reseed the CTR_DRBG context, that is extracts data from the entropy source.
    ret = mbedtls_ctr_drbg_reseed(&ctx, NULL, 0);

    // Generate random data using CTR_DRBG.
    ret = mbedtls_ctr_drbg_random(&ctx, buf, MBEDTLS_CTR_DRBG_BLOCKSIZE);

    // Clear CTR_CRBG context data.
    mbedtls_ctr_drbg_free(&ctx);
}
```

6.1.6.4 HMAC_DRBG with Prediction Resistance

This example describes how to use HMAC_DRBG with prediction resistance.

```
int crypto_sample_hmac_drbg_pr_on()
{
     mbedtls_hmac_drbg_context ctx;
     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);

     // 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_pr,
                             NULL, 0);

     // Enable prediction resistance.
     mbedtls_hmac_drbg_set_prediction_resistance(&ctx, MBEDTLS_HMAC_DRBG_PR_ON);

     // 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);
}
```

The API details are as follows:

- `void mbedtls_hmac_drbg_init (mbedtls_hmac_drbg_context* ctx)`

Prototype	void mbedtls_hmac_drbg_init(mbedtls_hmac_drbg_context *ctx)
Description	HMAC_DRBG context initialization makes the context ready for mbedtls_hmac_drbg_seed(), mbedtls_hmac_drbg_seed_buf() or mbedtls_hmac_drbg_free().
Parameters	ctx: HMAC_DRBG context to be initialized.
Return values	None.

- int mbedtls_hmac_drbg_seed(mbedtls_hmac_drbg_context* ctx, const mbedtls_md_info_t * md_info, int (*f_entropy)(void*, unsigned char*, size_t), 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 (*f_entropy)(void*, unsigned char*, size_t), 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.

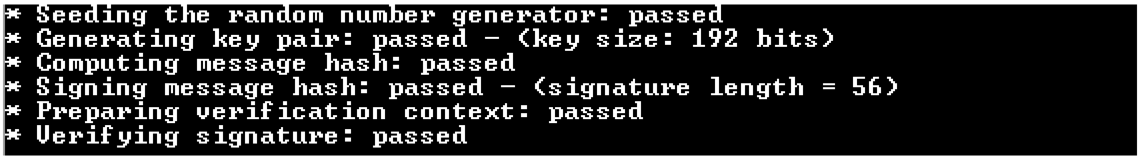
6.1.6.5 HMAC_DRBG Without Prediction Resistance

This example describes how to use HMAC_DRBG without prediction resistance.

```
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);
}
```

6.1.7 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.



```
* 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
```

Figure 62: The Result of the Crypto ECDSA

6.1.7.1 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. This sample describes how the user uses the ECDSA (Elliptic Curve Digital Signature Algorithm) of the “mbedTLS” library.

```
void crypto_sample_ecdsa(void *param)
{
    crypto_sample_ecdsa_test();

    return ;
}
```

6.1.7.2 Generates ECDSA Key Pair and Verifies ECDSA Signature

This example generates an ECDSA keypair and verifies the self-computed ECDSA signature.

```
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
    PRINTF("* Seeding the random number generator: ");

    // 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);

    // Compute message hash
    PRINTF("* Computing message hash: ");

    // Start a SHA-256 checksum calculation.
    mbedtls_sha256_starts_ret(&sha256_ctx, 0);

    // Feeds an input buffer into an ongoing SHA-256 checksum calculation.
    mbedtls_sha256_update_ret(&sha256_ctx, message, 100);

    // Finishe the SHA-256 operation, and writes the result to the output buffer.
    mbedtls_sha256_finish(&sha256_ctx, hash);

    // Sign message hash
    PRINTF("* Signing message hash: ");

    // Compute the ECDSA signature and writes it to a buffer.
    ret = mbedtls_ecdsa_write_signature(&ctx_sign, MBEDTLS_MD_SHA256, hash, 32,
                           sig, &sig_len, mbedtls_ctr_drbg_random, &ctr_drbg);

    // Verify signature
    PRINTF("* Verifying signature: ");

    // Read and verify an ECDSA signature.
    ret = mbedtls_ecdsa_read_signature(&ctx_verify, hash, 32, sig, sig_len);

    // Free an ECDSA context.
    mbedtls_ecdsa_free(&ctx_verify);
    mbedtls_ecdsa_free(&ctx_sign);

    // Clear CTR_CRBG context data.
    mbedtls_ctr_drbg_free(&ctr_drbg);

    // Free the data in the context.
    mbedtls_entropy_free(&entropy);

    // Clear s SHA-256 context.
    mbedtls_sha256_free(&sha256_ctx);
}
```

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 (*f_rng)(void *, unsigned char *, size_t), void *p_rng)`

Prototype	int mbedtls_ecdsa_genkey(mbedtls_ecdsa_context *ctx, mbedtls_ecp_group_id gid, int (*f_rng)(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. gid: The elliptic curve to use. One of the various MBEDTLS_ECP_DP_XXX macros depending on configuration. 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 if f_rng does not need a context argument.
Return values	0 on success. An MBEDTLS_ERR_ECP_XXX code on failure.

- int mbedtls_ecdsa_write_signature(mbedtls_ecdsa_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hlen, unsigned char *sig, size_t *slen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)

Prototype	int mbedtls_ecdsa_write_signature(mbedtls_ecdsa_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hlen, unsigned char *sig, size_t *slen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)
Description	This function computes the ECDSA signature and writes it to a buffer, serialized as defined in RFC-4492: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS).
Parameters	ctx: The ECDSA context to use. This must be initialized and have a group and private key bound to it, for example via mbedtls_ecdsa_genkey() or mbedtls_ecdsa_from_keypair(). md_alg: The message digest that was used to hash the message. hash: The message hash to be signed. This must be a readable buffer of length hlen Bytes. hlen: The length of the hash in Bytes. sig: The buffer to which to write the signature. This must be a writable buffer of a length at least twice as large as the size of the curve used, plus 9. For example, 73 Bytes if a 256-bit curve is used. A buffer length of MBEDTLS_ECDSA_MAX_LEN is always safe. slen: The address at which to store the actual length of the signature written. Must not be NULL. f_rng: The RNG function. This must not be NULL if MBEDTLS_ECDSA_DETERMINISTIC is unset. Otherwise, it is unused and may be set to NULL. p_rng: The RNG context to be passed to f_rng. This may be NULL if f_rng is NULL or does not use a context.
Return values	0 on success. An MBEDTLS_ERR_ECP_XXX, MBEDTLS_ERR_MPI_XXX or MBEDTLS_ERR_ASN1_XXX error code on failure.

- int mbedtls_ecdsa_read_signature(mbedtls_ecdsa_context *ctx, const unsigned char *hash, size_t hlen, const unsigned char *sig, size_t slen)

Prototype	int mbedtls_ecdsa_read_signature(mbedtls_ecdsa_context *ctx, const unsigned char *hash, size_t hlen, const unsigned char *sig, size_t slen)
Description	This function reads and verifies an ECDSA signature.
Parameters	ctx: The ECDSA context to use. This must be initialized and have a group and public key bound to it. hash: The message hash that was signed. This must be a readable buffer of length size Bytes. hlen: The size of the hash.

	sig: The signature to read and verify. This must be a readable buffer of length slen Bytes. slen: The size of sig in Bytes.
Return values	0 on success. MBEDTLS_ERR_ECP_BAD_INPUT_DATA if signature is invalid. MBEDTLS_ERR_ECP_SIG_LEN_MISMATCH if there is a valid signature in sig, but its length is less than siglen. An MBEDTLS_ERR_ECP_XXX or MBEDTLS_ERR_MPI_XXX error code on failure for any other reason.

6.1.8 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 and server sides.

```
* DHM parameter load: passed
* Diffie-Hellman full exchange: passed
```

Figure 63: The Result of the Crypto Diffie Hellman

6.1.8.1 Application Initialization

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.

```
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);
    }
}
```

6.1.8.2 Load Diffie-Hellman Parameters

This example application shows how the Diffie-Hellman parameters are loaded over the “mbedTLS” library’s API.

```
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	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.
Return values	None.

6.1.8.3 How Diffie-Hellman Works

This sample application shows how Diffie-Hellman works over the API of the “mbedTLS” 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.

```
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));

    x_size = mbedtls_mpi_size(&ctx_srv.P);
    pub_cli_len = x_size;

    /* Generate a DHM key pair and export its public part together
```

```
     * with the DHM parameters in the format.
     */
    ret = mbedtls_dhm_make_params(&ctx_srv, x_size, ske, &ske_len,
                                  &rnd_pseudo_rand, &rnd_info);

    // Parse the DHM parameters (DHM modulus, generator, and public key)
    ret = mbedtls_dhm_read_params(&ctx_cli, &p, ske + ske_len);

    // Create a DHM key pair and export the raw public key in big-endian format.
    ret = mbedtls_dhm_make_public(&ctx_cli, x_size, pub_cli, pub_cli_len,
                                  &rnd_pseudo_rand, &rnd_info);

    // Import the raw public value of the peer.
    ret = mbedtls_dhm_read_public(&ctx_srv, pub_cli, pub_cli_len);

    // Derive and export the shared secret (G^Y)^X mod P.
    ret = mbedtls_dhm_calc_secret(&ctx_srv, sec_srv, DHM_BUF_SIZE,
                                  &sec_srv_len, &rnd_pseudo_rand, &rnd_info);

    // Derive and export the shared secret (G^Y)^X mod P.
    ret = mbedtls_dhm_calc_secret(&ctx_cli, sec_cli, DHM_BUF_SIZE, &sec_cli_len,
                                  NULL, NULL);

    // Free and clear the components of a DHM context.
    mbedtls_dhm_free(&ctx_srv);
    mbedtls_dhm_free(&ctx_cli);
}
```

The API details are as follows:

- `int mbedtls_dhm_make_params( mbedtls_dhm_context *ctx, int x_size, char *output, size_t *olen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng )`

Prototype	int mbedtls_dhm_make_params(mbedtls_dhm_context *ctx, int x_size, char *output, size_t *olen, int (*f_rng)(void *, unsigned char *, size_t), 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. 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. olen: The address at which to store the number of Bytes written on success. This must not be NULL. 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_DHM_XXX error code on failure.

- `int mbedtls_dhm_read_params(mbedtls_dhm_context *ctx, unsigned char **p, unsigned char *end)`

Prototype	int mbedtls_dhm_read_params(mbedtls_dhm_context *ctx, unsigned char **p, 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_DHM_XXX error code on failure.

- int mbedtls_dhm_make_public(mbedtls_dhm_context *ctx, int x_size, unsigned char *output, size_t olen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)

Prototype	int mbedtls_dhm_make_public(mbedtls_dhm_context *ctx, int x_size, unsigned char *output, size_t olen, int (*f_rng)(void *, unsigned char *, size_t), 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_DHM_XXX error code on failure.

- int mbedtls_dhm_read_public(mbedtls_dhm_context *ctx, const unsigned char *input, size_t ilen)

Prototype	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 input in Bytes.
Return values	0 on success. An MBEDTLS_ERR_DHM_XXX error code on failure.

- int mbedtls_dhm_calc_secret(mbedtls_dhm_context *ctx, unsigned char *output, size_t output_size, size_t *olen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)

Prototype	int mbedtls_dhm_calc_secret(mbedtls_dhm_context *ctx, unsigned char *output, size_t output_size, size_t *olen, int (*f_rng)(void *, unsigned char *, size_t), 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.
Return values	0 on success. An MBEDTLS_ERR_DHM_XXX error code on failure.

6.1.9 Crypto Algorithms – RSA PKCS#1

The RSA PKCS#1 sample application demonstrates common use cases of RSA PKCS#1 functions.

Figure 64: The Result of the Crypto RSA

6.1.9.1 Application Initialization

This example shows RSA key validation, encryption, decryption, and verification of the signature. To verify the signature, a SHA-1 Hash algorithm is used.

```
void crypto_sample_rsa(ULONG arg)
{
    crypto_sample_rsa_pkcs1();

    return ;
}
```

6.1.9.2 How RSA PKCS#1 Works

The example application below shows how RSA PKCS#1 works over the API of the “mbedTLS” library. To verify, an RSA-1024 keypair and a SHA-1 Hash algorithm are used on RSA PKCS-1 v1.5.

```
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, MBEDTLS_MD_NONE);

    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 : ");

    memcpy(rsa_plaintext, RSA_PT, PT_LEN);

    // 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_pubkey(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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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.

- int mbedtls_rsa_pkcs1_verify(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, const unsigned char *sig)

Prototype	int mbedtls_rsa_pkcs1_verify(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), 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.

- void mbedtls_rsa_free(mbedtls_rsa_context *ctx)

Prototype	void mbedtls_rsa_free(mbedtls_rsa_context *ctx)
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.

6.1.10 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.

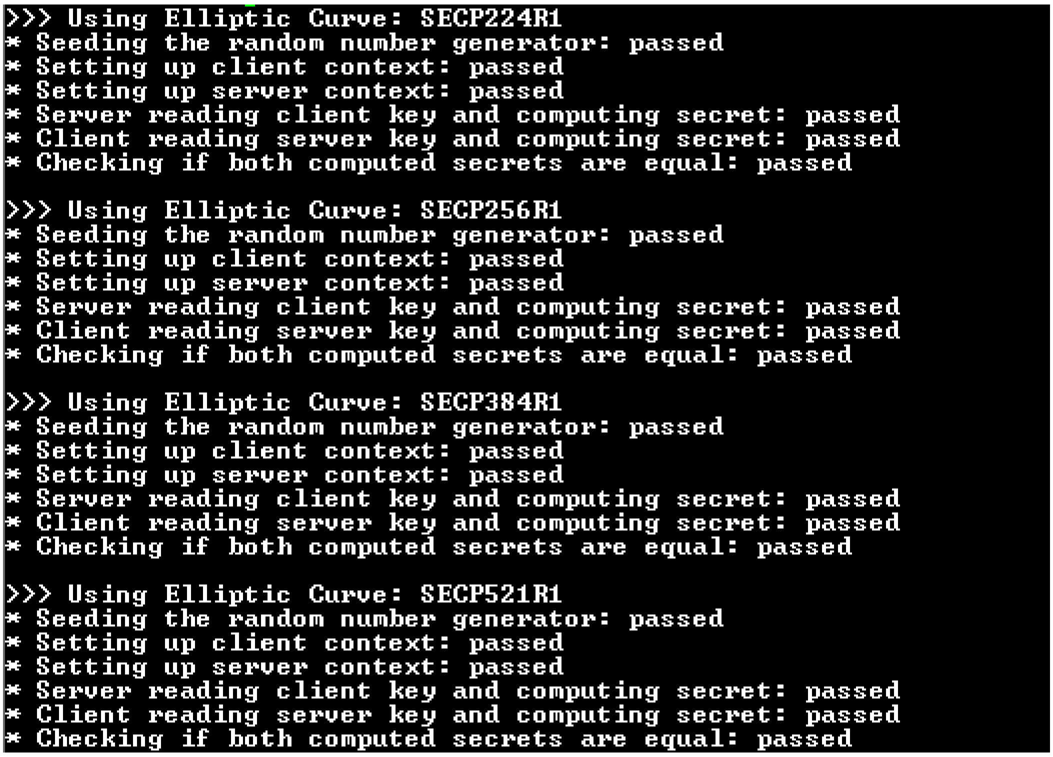


Figure 65: The Result of the Crypto ECDH

6.1.10.1 Application Initialization

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.

```
void crypto_sample_ecdh(void *param)
{
    mbedtls_ecp_group_id ids[6] = {
        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           */
        MBEDTLS_ECP_DP_NONE
    };

    for (idx = 0, id = ids[idx] ; idx < 6 && id != MBEDTLS_ECP_DP_NONE ; idx++,
         id = ids[idx])
    {
        ret = crypto_sample_ecdh_key_exchange(id);
        if (ret) {
            break;
        }
    }
}
```

6.1.10.2 How ECDH Key Exchange Works

This sample application shows how ECDH works over the API of the “mbedTLS” library. In this example, the ECDH key exchange is verified on the server and client sides.

```
int crypto_sample_ecdh_key_exchange(mbedtls_ecp_group_id id)
{
    mbedtls_ecdh_context ctx_cli;
    mbedtls_ecdh_context ctx_srv;
    mbedtls_entropy_context entropy;
    mbedtls_ctr_drbg_context ctr_drbg;

    // 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\r\n");
        }
        break;
        case MBEDTLS_ECP_DP_SECP256R1: {
            PRINTF("SECP256R1\r\n");
        }
        break;
        case MBEDTLS_ECP_DP_SECP384R1: {
            PRINTF("SECP384R1\r\n");
        }
        break;
        case MBEDTLS_ECP_DP_SECP521R1: {
            PRINTF("SECP521R1\r\n");
        }
        break;
        case MBEDTLS_ECP_DP_CURVE25519: {
            PRINTF("Curve25519\r\n");
        }
        break;
```



```
        default: {
            PRINTF("failed - [%s] Invalid Curve selected!\r\n");
        }
        goto cleanup;
    }
    // Initialize random number generation
    PRINTF("* Seeding the random number generator: ");

    ret = mbedtls_ctr_drbg_seed(&ctr_drbg, mbedtls_entropy_func, &entropy,
                                (const unsigned char *)pers, sizeof(pers));

    // Client: inialize context and generate keypair
    PRINTF("* Setting up client context: ");

    // 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
    PRINTF("* Setting up server context: ");

    // 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
     */
    // 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.z), &(ctx_srv.Qp), &(ctx_srv.d),
                                      mbedtls_ctr_drbg_random, &ctr_drbg);

    // Client: read peer's key and generate shared secret
    PRINTF("* Client reading server key and computing 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));

    // Compute the shared secret.
    ret = mbedtls_ecdh_compute_shared(&(ctx_cli.grp), &(ctx_cli.z),
                                      &(ctx_cli.Qp), &(ctx_cli.d),
                                      mbedtls_ctr_drbg_random, &ctr_drbg);

    // Verification: are the computed secrets equal?
    PRINTF("* Checking if both computed secrets are equal: ");

    MBEDTLS_MPI_CHK(mbedtls_mpi_cmp_mpi(&(ctx_cli.z), &(ctx_srv.z)));

    // Free ECDH context.
    mbedtls_ecdh_free(&ctx_cli);
    mbedtls_ecdh_free(&ctx_srv);

    // Free the data in the context.
    mbedtls_entropy_free(&entropy);

    // Clear CTR_CRBG context data.
    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 set up. 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 (*f_rng)(void *, unsigned char *, size_t), void *p_rng)

Prototype	int mbedtls_ecdh_gen_public(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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.

6.1.11 Crypto Algorithms – KDF

The Key Derivation Function (KDF) sample application demonstrates common use cases of PKCS#5 functions.

Figure 66: The Result of the Crypto KDF

6.1.11.1 Application Initialization

This example uses a password-based Key Derivation Function specified in PKCS#5 PBKDF2 and implemented in “mbedTLS” in function mbedtls_pkcs5_pdkdf2_hmac.

```
void crypto_sample_kdf(void *param)
{
    crypto_sample_pkcs5();
}
```

6.1.11.2 How KDF Works

This example application shows how KDF works over the API of the “mbedTLS” library. In this example, PKCS#5 PBKDF2 is used. To verify, a SHA-1 Hash algorithm is used.

```
int crypto_sample_pkcs5()
{
    mbedtls_md_context_t sha1_ctx;
    const mbedtls_md_info_t *info_sha1;

    // Initialize a SHA-1 context.
    mbedtls_md_init(&sha1_ctx);

    // 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 allocate 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 free any embedded internal
       structure,
     * but does not free ctx itself.
     */
    mbedtls_md_free(&sha1_ctx);
}
```

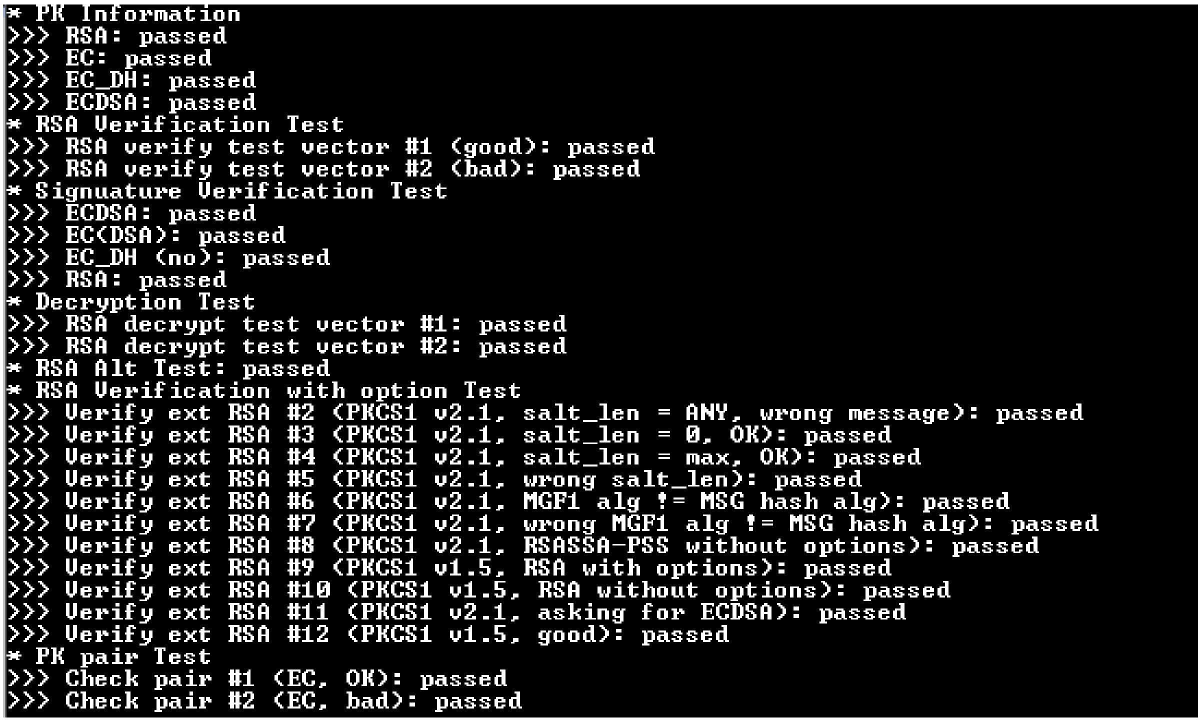
The API details are as follows:

- `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.

6.1.12 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, using traditional RSA or Elliptic Curves. The Public Key abstraction layer sample application demonstrates common use cases of the APIs.



```
* 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
* Signuature 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
```

Figure 67: The Result of the Crypto Public Key

6.1.12.1 Application Initialization

This example shows how to use the Public Key Abstraction Layer of the “mbedTLS” library.

```
void crypto_sample_pk(void *param)
{
    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("* Signuature 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_list[i].title,
         crypto_sample_pk_rsa_verify_ext_list[i].message_hex_string,
         crypto_sample_pk_rsa_verify_ext_list[i].digest,
         crypto_sample_pk_rsa_verify_ext_list[i].mod,
         crypto_sample_pk_rsa_verify_ext_list[i].radix_N,
         crypto_sample_pk_rsa_verify_ext_list[i].input_N,
         crypto_sample_pk_rsa_verify_ext_list[i].radix_E,
         crypto_sample_pk_rsa_verify_ext_list[i].input_E,
         crypto_sample_pk_rsa_verify_ext_list[i].result_hex_str,
         crypto_sample_pk_rsa_verify_ext_list[i].pk_type,
         crypto_sample_pk_rsa_verify_ext_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);
}
```

6.1.12.2 How Public Key Abstraction Layer is Used

The “mbedTLS” library provides the Public Key Abstraction Layer for confidentiality, integrity, authentication, and non-repudiation based on asymmetric algorithms, using traditional RSA or Elliptic Curves.

The user needs to check which public key could be supported by the “mbedTLS” library. The example code below shows how to get and check public key information.

```
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 allocate 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 inline size_t mbedtls_pk_get_len(const mbedtls_pk_context *ctx)`

Prototype	static inline 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	ctx: The context to clear. It must have been initialized. If this is NULL, this function does nothing.
Return values	None.

Function crypto_sample_pk_genkey describes how to generate a public key with the given algorithms (RSA or Elliptic curves).

```
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 (*f_rng)(void *, unsigned char *, size_t), void *p_rng, unsigned int nbits, int exponent)

Prototype	int mbedtls_rsa_gen_key(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), void *p_rng)

Prototype	int mbedtls_ecp_gen_keypair(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), 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.

Function crypto_sample_pk_rsa_verify_test_vec describes how to verify RSA signatures 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.
         */
        ret = mbedtls_md(mbedtls_md_info_from_type(digest),
                         message_str, msg_len,
                         hash_result);
    }

    // Verify signature (including padding if relevant) & Check result with
    // expected result.
    ret = mbedtls_pk_verify(&pk, digest, hash_result, 0, result_str,
                            mbedtls_pk_get_len(&pk));

    // Free the components of a mbedtls_pk_context.
    mbedtls_pk_free(&pk);
}
```

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	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 siglen, or a specific error code.

Function crypto_sample_pk_sign_verify describes how to generate a key, make a signature, and verify this with the given crypto algorithms.

```
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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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.

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.

```
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 set up an RSA public key, precisely N and E must have been imported. To set up 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 (*f_rng)(void *, unsigned char *, size_t), 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 (*f_rng)(void *, unsigned char *, size_t), 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.

Function crypto_sample_pk_rsa_alt describes how RSA ALT context creates and decrypts a signature.

```
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,
```

```
                                               crypto_sample_rsa_decrypt_func,
                                               crypto_sample_rsa_sign_func,
                                               crypto_sample_rsa_key_len_func);

    // 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	ctx: 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 code example shows how to check if a public and private pair of keys matches.

```
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	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 *pk, const unsigned char *key, size_t keylen, const unsigned char *pwd, size_t pwdlen)`

Prototype	int mbedtls_pk_parse_key(mbedtls_pk_context *pk, 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	pk: 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

6.1.13 Crypto Algorithms – Generic Cipher Wrapper

The Generic cipher wrapper sample application demonstrates common use cases of a generic cipher wrapper API of the “mbedTLS” library that is included in the DA16200 SDK.

Figure 68: The Result of the Generic Cipher

6.1.13.1 Application Initialization

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.

```
void crypto_sample_cipher(void *param)
{
      crypto_sample_cipher_wrapper();

      vTaskDelete(NULL);

      return ;
}
```

6.1.13.2 How Generic Cipher Wrapper is Used

This example describes how to encrypt and decrypt with generic cipher wrapper functions.

```
int crypto_sample_cipher_wrapper()
{
    mbedtls_cipher_type_t cipher_type = MBEDTLS_CIPHER_NONE;
    mbedtls_cipher_context_t cipher_ctx;
    mbedtls_cipher_info_t *cipherinfo = NULL;
    mbedtls_cipher_mode_t cipher_mode = MBEDTLS_MODE_NONE;

    for (cipher_type = MBEDTLS_CIPHER_AES_128_ECB ;
         cipher_type <= MBEDTLS_CIPHER_CAMELLIA_256_CCM ;
         cipher_type++) {

        flag_pass = FALSE;

        // Initialize a cipher_context as NONE.
```



```
mbedtls_cipher_init(cipher_ctx);

// 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 fill 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]),
                                         &ciphertext_finlen);

} else {
    // Set the key to use with the given context.
    cipher_status = mbedtls_cipher_setkey(&cipher_ctx,
                                          cipher_key, cipher_keylen,
                                          MBEDTLS_ENCRYPT);

    // Perform autenticated 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 autenticated 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 inline int mbedtls_cipher_get_key_bitlen(const mbedtls_cipher_context_t *ctx)

Prototype	static inline 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 inline mbedtls_cipher_mode_t mbedtls_cipher_get_cipher_mode(const mbedtls_cipher_context_t *ctx)

Prototype	static inline 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 inline int mbedtls_cipher_get_iv_size(const mbedtls_cipher_context_t *ctx)

Prototype	static inline 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 inline unsigned int mbedtls_cipher_get_block_size(const mbedtls_cipher_context_t *ctx)

Prototype	static inline unsigned int mbedtls_cipher_get_block_size(const mbedtls_cipher_context_t *ctx)
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 inline const char *mbedtls_cipher_get_name(const mbedtls_cipher_context_t *ctx)

Prototype	static inline 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.

7 Peripheral Examples

7.1 UART

Along with a UART0 interface for the debug console, the DA16200 SDK has a UART1 or UART2 interface to communicate with an external MCU. GPIOA[4] and GPIOA[5] can be used to this interface.

7.1.1 How to Run

1. In the Eclipse, import project for the UART sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/UART1/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. The start log message is shown in the console terminal and UART1 terminal.
4. To test with 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.
   a. UART1 terminal

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

Figure 69: Result of UART #1

   b. Console terminal

```
- Start UART1 communicate module ...
!!! No selected network !!!

>>> Network Interface (wlan0) : UP
>>> Associated with 70:3a:cb:25:f5:f8

Connection COMPLETE to 70:3a:cb:25:f5:f8

-- DHCP Client WLAN0: SEL(6)
-- DHCP Client WLAN0: REQ(1)
-- DHCP Client WLAN0: CHK(8)
-- DHCP Client WLAN0: BOUND(10)
         Assigned addr    : 192.168.86.68
               netmask    : 255.255.255.0
               gateway    : 192.168.86.1
               DNS addr   : 192.168.86.1

         DHCP Server IP   : 192.168.86.1
         Lease Time       : 24h 00m 00s
         Renewal Time     : 12h 00m 00s

>>>
- (len=5):
[00000000]  68 65 6c 6c 6f                                       hello
```

Figure 70: Result of UART #2

7.1.2 Application Initialization

This is an example of a user application to initialize and communicate between the DA16200 and an 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.

~/SDK/apps/common/examples/Peripheral/UART1/src/uart_sample.c

```
/* Local static variables */
static int  sample_uart_idx = UART_UNIT_1;    // UART_UNIT_1, UART_UNIT_2

/*
 * For Customer's configuration for UART devices
 *
 * "user_UART_config_info" data is located in /SDK/customer/src/user_uart.c.
 *
 * This data is temporary for sample application.
 */
static uart_info_t sample_UART_config_info =
{
    UART_BAUDRATE_115200,   /* baud */
    UART_DATABITS_8,        /* bits */
    UART_PARITY_NONE,       /* parity */
    UART_STOPBITS_1,        /* stopbit */
    UART_FLOWCTL_OFF        /* flow control */
};

void run_uart1_sample(UINT32 arg)
{
    int status;

    *
    * int set_user_UART_conf(int uart_idx, uart_info_t *uart_conf_info, char
      atcmd_flag)
    */
    status = set_user_UART_conf(UART_UNIT_1, &sample_UART_config_info, FALSE);
    if (status != 0)
    {
        PRINTF("[%S] Error to configure for UART1 !!!\n", __func__);
        return;
    }


    *
    * int UART_init(int uart_idx);
    */
    status = UART_init(sample_uart_idx);
    if (status != 0)
    {
        PRINTF("[%S] Error to initialize UART1 with sample_UART_config !!!\n",
        __func__);
    return;
    }

    /* Start UART monitor */
    uart1_sample();
}
```

Function uart1_sample() invokes function *get_data_from_uart1()* repeatedly to read data from UART1. User can enable/disable the UART echo function by setting “echo_enable”.

```
static void uart1_sample(void)
{
    int i;
    char *init_str = "- Start UART1 communicate module ...\r\n";
    char *rx_buf = NULL;
    char *tx_buf = "\r\n- Data receiving OK...\r\n";
    int tx_len;
```

```
    /* Print-out test string to console and to UART1 device */
    PRINTF((const char *)init_str); // For Console
    puts_UART(sample_uart_idx, init_str, strlen((const char *)init_str));

    echo_enable = TRUE;
    rx_buf = malloc(USER_UART1_BUF_SZ);

    while (1)
    {
        memset(rx_buf, 0, USER_UART1_BUF_SZ);

        /* Get on byte from uart1 comm port */
        get_data_from_uart1(rx_buf);
        … …
    }
}
```

7.1.3 Data Read/Write

Use *getchar_UART()* to read a character from UART1 or UART2. This example shows how to read data from UART device until meets characters ‘\n’ or ‘\r’. User can modify this function for customized application operation.

NOTE

After UART_INIT() is called, it try to receive UART_RX data even if UART_READ() does not call. If the user don't want to use the data before UART_READ(), it need to flush the UART_RX data.

```
#define USER_DELIMITER_0    '\0'
#define USER_DELIMITER_1    '\n'
#define USER_DELIMITER_2    '\r'

static void get_data_from_uart1(char *buf)
{
    char    ch = 0;
    int     i = 0;

    while (1)
    {
        /* Get on byte from uart1 comm port */
        ch = getchar_UART(sample_uart_idx, portMAX_DELAY);

        if (ch == 0)
        {
            vTaskDelay(1);
            continue;
        }

        if (echo_enable == TRUE)
        {
            puts_UART(sample_uart_idx, &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;
            }
        }
    }
```

And also this example shows how to send data to UART1 using by *puts_UART()* API.

~/SDK/core/system//include/common/common_uart.h

```
/**
*****************************************************************************************
 * @brief  Put character string to UART device
 * @param[in] uart_idx  Index value of UART interface (UART_UNIT_1, UART_UNIT_2)
 * @param[in] *data     Text string to write
 * @param[in] len       Write length
 * @return              None
*****************************************************************************************
 */
void    puts_UART(int uart_idx, char *data, int data_len);
```

7.2 GPIO

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

7.2.1 How to Run

1. In the Eclipse, import project for the GPIO sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/GPIO/projects/da16200*
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.
   - GPIOA[0] output low, GPIOA[4] output low, GPIOA[1] input low
   - GPIOA[0] output high, GPIOA[4] output high, GPIOA[1] input low
   - GPIOA[0] output low, GPIOA[4] output low, GPIOA[1] input low

7.2.2 Operation

1. Create and initialize a GPIO handle.

```
HANDLE gpio;
gpio = GPIO_CREATE(GPIO_UNIT_A);
GPIO_INIT(gpio);
```

2. Set pin multiplexing.

```
/* AMUX to GPIOA[1:0] */
_da16x_io_pinmux(PIN_AMUX, AMUX_GPIO);

/* BMUX to GPIOA[3:2] */
```



```
_da16x_io_pinmux(PIN_BMUX, BMUX_GPIO);

/* CMUX to GPIOA[5:4] */
_da16x_io_pinmux(PIN_CMUX, CMUX_GPIO);
```

3. Set GPIOA[0], GPIOA[4] as output mode and GPIOA[1] as input mode.

```
/* 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);
```

4. Set GPIOA[2] as an interrupt source with active low and register a callback function.

```
static int set_gpio_interrupt(HANDLE handler, UINT8 pin_num, UINT8 int_type, UINT8
int_pol, void *callback_func)
{
      UINT16 pin, int_en_status;
      UINT32 ioctldata[3];
      int ret;

      if (15 < pin_num )
        return FALSE;

      if(handler == NULL){
        return FALSE;
      }

      pin = 0x01<<pin_num;
      ret = GPIO_IOCTL(handler, GPIO_SET_INPUT, &pin);

      ret = GPIO_IOCTL(handler, GPIO_GET_INTR_MODE, &ioctldata[0]);
      /* interrupt type 1: edge, 0: level*/
      ioctldata[0] &= ~(1 << pin_num); // clear the bit first
      ioctldata[0] |= (int_type << pin_num);
      /* interrupt pol 1: high active, 0: low active */
      ioctldata[1] &= ~(1 << pin_num); // clear the bit first
      ioctldata[1] |= (int_pol << pin_num);
      ret = GPIO_IOCTL(handler, GPIO_SET_INTR_MODE, &ioctldata[0]);

      /* register callback function */
      ioctldata[0] = pin; /* interrupt pin */
      ioctldata[1] = (UINT32) callback_func; /* callback function */
      ioctldata[2] = (UINT32) pin_num; /* param data */
      ret = GPIO_IOCTL(handler, GPIO_SET_CALLACK, ioctldata);

      ret = GPIO_IOCTL(handler, GPIO_GET_INTR_ENABLE, &int_en_status);
      int_en_status |= pin;
      ret = GPIO_IOCTL(handler, GPIO_SET_INTR_ENABLE, &int_en_status);

      return ret;
}

/* GPIOA[2] interrupt active low , Edge trigger */
set_gpio_interrupt(gpio, 2, GPIO_INT_TYPE_EDGE, GPIO_INT_POL_LOW, (void*)gpio_callback );
```

5. Set GPIOA[3] as an interrupt source with active high and register a callback function.

```
/*GPIOA[3] interrupt active high, Edge trigger */
set_gpio_interrupt(gpio, 3, GPIO_INT_TYPE_EDGE, GPIO_INT_POL_HIGH, (void*)gpio_callback );
```


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

```
if (toggle)
{
    /* GPIOA[0],GPIOA[4] to high */
    write_data = GPIO_PIN0 | GPIO_PIN4;
    GPIO_WRITE(gpio, GPIO_PIN0 | GPIO_PIN4, &write_data, sizeof(UINT16));
    toggle = 0;
}
else
{
    /* GPIOA[0],GPIOA[4] to low*/
    write_data = 0;
    GPIO_WRITE(gpio, GPIO_PIN0 | GPIO_PIN4, &write_data, sizeof(UINT16));
    toggle = 1;
}

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

7. It can set the PAD pull condition by using PAD_PULL_CONTROL

```
#if PAD_PULL_CONTROL
    /*
      * GPIOA[1] input pull control it can make gpio pad pull up or pull down or HIZ
    */
    _da16x_gpio_set_pull(GPIO_UNIT_A, GPIO_PIN1, PULL_UP);
    /* or */
    _da16x_gpio_set_pull(GPIO_UNIT_A, GPIO_PIN1, PULL_DOWN);
    /* or */
    _da16x_gpio_set_pull(GPIO_UNIT_A, GPIO_PIN1, HIGH_Z);
#endif
```

8. It can activate the RTC_GPO example by using RTC_GPO_CONTROL

```
#ifdef RTC_GPO_CONTROL
     RTC_GPO_OUT_INIT(1);                 // 0: auto, 1: manual
     RTC_GPO_OUT_CONTROL(1);              // Set High
#endif
```

9. The both edge of interrupt is not supported by HW but it can be supported by SW.
   Activate the GPIO interrupt according to the GPIO read value.

```
GPIO_READ(gpioc, GPIO_PIN6, &read_data, sizeof(UINT16));
set_gpio_interrupt(gpioc, 6, GPIO_INT_TYPE_EDGE, !(GPIO_PIN6&read_data),
(void*)gpioc_callback );
```

   And change the interrupt polarity at every GPIO callback function.
   Please refer the “GPIOC6_BOTH_EDGE_INTERRUPT” example for this

7.3 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.

7.3.1 How to Run

1. In the Eclipse, import project for the GPIO Retention sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/GPIO_Retention/projects/da16200*
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 GPIOC [7] keep their PIN states.

7.3.2 Operation

1. Set pin multiplexing.

```
/*
 * 1. Set to GPIOA[11:8], GPIOC[8:6]
 * 2. Need be written to "config_pin_mux" function.
 */
_da16x_io_pinmux(PIN_EMUX, EMUX_GPIO);
_da16x_io_pinmux(PIN_FMUX, FMUX_GPIO);
_da16x_io_pinmux(PIN_UMUX, UMUX_GPIO);
```

2. Set GPIO Retention Config.

```
/* Set GPIOA[9:8] to retention high */
ret = _GPIO_RETAIN_HIGH(GPIO_UNIT_A, GPIO_PIN8 | GPIO_PIN9);
if(ret == FALSE)
    PRINTF("GPIO_RETAIN_HIGH() return false.\n");

/* Set GPIOA[10] to retention low */
ret = _GPIO_RETAIN_LOW(GPIO_UNIT_A, GPIO_PIN10);
if (ret == FALSE)
    PRINTF("GPIO_RETAIN_LOW() return false.\n");

/* Set GPIOC[7] to retention high */
ret = _GPIO_RETAIN_HIGH(GPIO_UNIT_C, GPIO_PIN7);
if(ret == FALSE)
    PRINTF("GPIO_RETAIN_HIGH() return false.\n");
```

3. Power Down.

```
char * _argv[4] = {"down", "sec", "10", "1"};
cmd_power_down_config(4, _argv);

/* Set GPIOC[7] to retention high */
ret = _GPIO_RETAIN_HIGH(GPIO_UNIT_C, GPIO_PIN7);
if(ret == FALSE)
    PRINTF("GPIO_RETAIN_HIGH() return false.\n");
```

7.4 I2C

This section shows how to use the I2C interface.

7.4.1 How to Run

1. In the Eclipse, import project for the I2C sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/I2C/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. The sample application code is written in the following source file:
   ~/SDK/apps/common/examples/Peripheral/I2C/src/i2c_sample.c

7.4.2 Operation

1. Hardware setup:
   a. Remove resistor R6 and R7.
   b. Connect the AT24C512 EEPROM with the Renesas 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.

```
// GPIO Select for I2C working. GPIO1 = SCL, GPIO0= SDA
```

```
Board_initialization();
DA16X_CLOCK_SCGATE->Off_DAPB_I2CM = 0;
DA16X_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.

```
// Device Address for AT24C512
UINT32 addr = 0xa0;
DRV_I2C_IOCTL(I2C, I2C_SET_CHIPADDR, &addr);
```

4. i2c clock.

```
// Set I2C Working Clock. Unit = KHz
DRV_I2C_IOCTL(I2C, I2C_SET_CLOCK, &i2c_clock);
```

5. i2c write.

```
// Data Random Write  to EEPROM
// Address = 0, Length = 32, Word Address Length = 2
// [Start] - [Device addr. W] -  [1st word addr.] - [2nd word addr.] - [wdata0] ~
   [wdata31] - [Stop]

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,
// Handle, buffer, length, stop enable, dummy
AT_I2C_DATA_LENGTH + AT_I2C_LENGTH_FOR_WORD_ADDRESS, 1, 0);

if (status != TRUE)
{
    PRINTF("ret : 0x%08x\r\n", status);
}
```

6. i2c read.

```
// Data Random Read  from EEPROM
// Address = 0, Length = 32, Word Address Length = 2
// [Start] - [Device addr. W] - [1st word addr.] - [2nd word addr.]  - [Start] -
   [Device addr. R] - [rdata0] ~ [rdata31] - [Stop]

// Word Address to Write Data. 2 Bytes. refer at24c512 DataSheet
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;

// Handle, buffer, length, address length, dummy
status = DRV_I2C_READ(I2C, i2c_data_read, AT_I2C_DATA_LENGTH,
                      AT_I2C_LENGTH_FOR_WORD_ADDRESS, 0);

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.

```
// Data Current Address  Read from EEPROM
// Length = 32, Word Address Length = 0
// [Start] -[Device addr. R] - [rdata0] ~ [rdata31] - [Stop]

// Handle, buffer, length, address length, dummy
status = DRV_I2C_READ(I2C, i2c_data_read, 4, 0, 0);

if (status != TRUE)
{
    PRINTF("ret : 0x%08x\r\n", status);
}
```

7.5 I2S

This section shows how to use the I2S interface.

7.5.1 How to Run

1. In the Eclipse, import project for the I2S sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/I2S/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. The sample application code is written in the following source file:
   ~/SDK/apps/common/examples/Peripheral/I2S/src/i2s_sample.c

7.5.2 User Task

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

~/SDK/apps/common/examples/Peripheral/I2S/src/sample_apps.c

```
static const app_task_info_t      sample_apps_table[] =
{ I2S_SAMPLE, i2s_sample, 512, (tskIDLE_PRIORITY + 7), FALSE, FALSE,
  UNDEF_PORT,RUN_ALL_MODE },
};
```

7.5.3 Operation

1. Create and initialize an I2S handle.

```
HANDLE gi2shandle = NULL;
I2S_HANDLER_TYPE *i2s;
unsigned int mode, data;

DA16X_CLOCK_SCGATE->Off_DAPB_I2S = 0;
DA16X_CLOCK_SCGATE->Off_DAPB_APBS = 0;

gi2shandle = DRV_I2S_CREATE(I2S_0);
i2s = (I2S_HANDLER_TYPE *) gi2shandle;

if (!gi2shandle)
{
    vTaskDelete(NULL);
    return;
}

/* Set I2S Output Mode */
if (DRV_I2S_INIT(gi2shandle, mode) == FALSE)
{
    vTaskDelete(NULL);
    return;
}
```

2. Set the internal DAC or the external DAC.

```
// GPIO[3] - I2S_LRCK,  GPIO[2] - I2S_SDO
_da16x_io_pinmux(PIN_BMUX, BMUX_I2S);
// GPIO[1] - I2S_MCLK,  GPIO[0] - I2S_BCLK
_da16x_io_pinmux(PIN_AMUX, AMUX_I2S);

DRV_I2S_SET_CLOCK(gi2shandle, I2S_CLK_SOURCE_INTERNAL, 0);
```

3. Set additional configuration.

```
data = TRUE;
```



```
DRV_I2S_IOCTL(i2s, I2S_SET_STEREO, &data); /* Set Stereo Output Mode */

#ifdef I2S_SAMPLE_SET_MODE_RX
data = I2S_RESOLUTION_RX_16B;
#else
data = I2S_RESOLUTION_TX_16B;
#endif

DRV_I2S_IOCTL(i2s, I2S_SET_PCM_RESOLUTION, &data); /* Set 16bit resolution Mode */
```

4. Write and read data.

```
for(int i=0;i<2;i++)
{
#ifdef I2S_SAMPLE_SET_MODE_RX
    rd_len = DRV_I2S_READ(i2s, (unsigned int *)rx_buf[i], 768, 0);
#else
    DRV_I2S_WRITE(i2s, (unsigned int *) sinewave_pattern, 768, 0);
#endif
    xEventGroupWaitBits(i2s_sample_event, 0x1, pdTRUE, pdFALSE, 20);
}
```

7.6 PWM

This section shows how to use the PWM interface.

7.6.1 How to Run

1. In the Eclipse, import project for the PWM sample application as follows:
    - *~/SDK/apps/common/examples/Peripheral/PWM/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
3. The sample application code is written in the following source file: *~/SDK/apps/common/examples/Peripheral/PWM/src/pwm_sample.c*

7.6.2 Operation

1. Hardware setup:
    a. Remove resistor R6~R9.
    b. Run the PWM example command.
    c. Get waveform from P7~P9 in connector J4.
    d. Compare the waveform with the PWM setting inside the example code.
2. pwm setgpio.

```
Board_Init();
DA16X_CLOCK_SCGATE->Off_CAPB_PWM = 0;

gpio = GPIO_CREATE(GPIO_UNIT_A);
GPIO_INIT(gpio);
GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNC_PWM_OUT0, GPIO_ALT_FUNC_GPIO0);
GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNC_PWM_OUT1, GPIO_ALT_FUNC_GPIO1);
GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNC_PWM_OUT2, GPIO_ALT_FUNC_GPIO2);
GPIO_SET_ALT_FUNC(gpio, GPIO_ALT_FUNC_PWM_OUT3, GPIO_ALT_FUNC_GPIO3);
```

3. pwm init.

```
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.

```
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

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.

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

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

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

// 2400 cycles(=30us @ 80MHz), cycle = value + 1
cycle = 2400-1;
// 1680 cycles(=21us@80MHz, 70% Duty High), 70% Duty High), duty_cycle = value + 1
duty_cycle = 1680-1;
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);
```

7.7 ADC

This section shows how to use the ADC interface.

7.7.1 How to Run

1. In the Eclipse, import project for the ADC sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/ADC/projects/da16200*
2. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.

3. The sample application code is written in the following source file:
   ~/SDK/apps/common/examples/Peripheral/ADC/src/adc_sample.c

7.7.2 Operation

1. Hardware setup:
   a. Provide 0~1.3 V 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.

```
PRINTF("ADC_SAMPLE\n");
DA16X_CLOCK_SCGATE->Off_DAPB_AuxA = 0;
DA16X_CLOCK_SCGATE->Off_DAPB_APBS = 0;

// Set PAD Mux. GPIO_0 (ADC_CH0), GPIO_1(ADC_CH1)
_da16x_io_pinmux(PIN_AMUX, AMUX_AD12);

// Create Handle
hadc = DRV_ADC_CREATE(DA16200_ADC_DEVICE_ID);

// Initialization
DRV_ADC_INIT(hadc, DA16x_ADC_NO_TIMESTAMP);
```

3. adc start.

```
// Start. Set Sampling Frequency. 12B ADC Set to 200KHz
// Clock = 1MHZ / (value + 1)
// Ex) If Value = 4, Clock = 1MHz / (4+1) = 200KHz
DRV_ADC_START(hadc, DA16x_ADC_DIVIDER_12, 0);
```

4. adc enable.

```
// Set ADC_0 to 12Bit ADC, ADC_1 to 12Bit ADC
DRV_ADC_ENABLE_CHANNEL(hadc, DA16200_ADC_CH_0, DA16x_ADC_SEL_ADC_12, 0);
DRV_ADC_ENABLE_CHANNEL(hadc, DA16200_ADC_CH_1, DA16x_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, DA16200_ADC_CH_0, data0, DA16x_ADC_NUM_READ * 2,
                 DA16x_ADC_TIMEOUT_DMA, 0);

// Read 16ea ADC_1 Value
DRV_ADC_READ_DMA(hadc, DA16200_ADC_CH_1, data1, DA16x_ADC_NUM_READ * 2,
                 DA16x_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, DA16200_ADC_CH_0, &data, 0);
```

7. adc close.

```
// Close ADC
DRV_ADC_CLOSE(hadc);
```

7.8 SPI

This section shows how the SPI loopback operation works.

7.8.1 How to Run

1. In the Eclipse, import project for the SPI sample application as follows:
    - *~/SDK/apps/common/examples/Peripheral/SPI/projects/da16200*
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)
    - GPIOA[2] (SPI_CSB) - GPIOA[6] (E_SPI_CSB)
    - GPIOA[3] (SPI_CLK) - GPIOA[7] (E_SPI_CLK)
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 71.

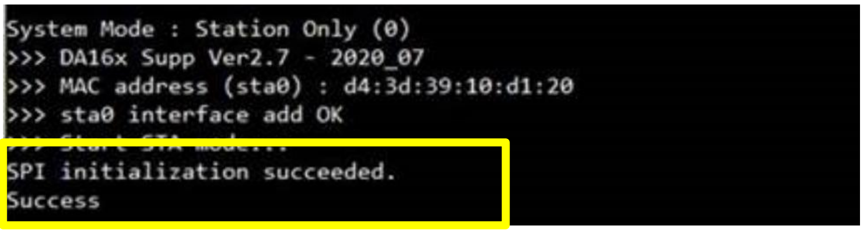


Figure 71: SPI Loopback Communication

7.8.2 Operation

1. Create an SPI handle and configure the interface.

```
spi = SPI_CREATE(SPI_UNIT_0);
if(spi == NULL) {
    PRINTF("[%s]failed to create instance\n", __func__);
    return;
}

_sys_clock_read( ioctldata, sizeof(UINT32));
SPI_IOCTL(spi, SPI_SET_CORECLOCK, ioctldata);

/* set SPI speed */
ioctldata[0] = SMC_SPI_SPEED * MHz;
SPI_IOCTL(spi, SPI_SET_SPEED, ioctldata);

/* set SPI polarity */
ioctldata[0] = SMC_SPI_POLARITY;
SPI_IOCTL(spi, SPI_SET_FORMAT, ioctldata);

/* set SPI DMA config */
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);

/* set SPI chip select, io operation type */
ioctldata[0] = SMC_SPI_CS;
ioctldata[1] = SMC_IO_OPERTATION_TYPE;
SPI_IOCTL( spi, SPI_SET_WIRE, (VOID *)ioctldata);
```



```
/* set SPI delay index */
ioctldata[0] = SPI_DELAY_INDEX_LOW;
SPI_IOCTL(spi, SPI_SET_DELAY_INDEX, ioctldata);

/* SPI initialization */
status = SPI_INIT(spi);
```

2. Set pin multiplexing as SPI master and SPI slave.

```
/* pinmux config for SPI Slave - GPIOA[3:0] */
_da16x_io_pinmux(PIN_AMUX, AMUX_SPIs);
_da16x_io_pinmux(PIN_BMUX, BMUX_SPIs);

/* pinmux config for SPI Host  - GPIOA[9:6] */
_da16x_io_pinmux(PIN_DMUX, DMUX_SPIm);
_da16x_io_pinmux(PIN_EMUX, EMUX_SPIm);
```

3. Write data.

```
/* generate host interface protocol header */
_buf[0] = (addr >> 8) & 0xff;
_buf[1] = (addr >> 0) & 0xff;
_buf[2] = (HPC_WRITE_CMD & 0xff)
       | (HPC_COMMON_ADDR_MODE << 5)
       | (HPC_REF_LEN<<4)|((length>>8)&0xf);
_buf[3] = (length)&0xff;

/* copy data to buf */
memcpy(&(_buf[4]), data, length);

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

status = SPI_WRITE(spi, _buf, (HPC_HEADER_LEN + length));

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

4. Read data.

```
_buf[0] = (addr >> 8) & 0xff;
_buf[1] = (addr >> 0) & 0xff;
_buf[2] = HPC_READ_CMD
       | (HPC_COMMON_ADDR_MODE << 5)
       | (HPC_REF_LEN<<4)|((len>>8)&0xf);
_buf[3] = (len)&0xff;

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

status = SPI_WRITE_READ(spi, _buf, 4, rx_data, len);

/* Bus Unlock */
ioctldata[0] = FALSE;
ioctldata[1] = portMAX_DELAY;
```

```
ioctldata[2] = SPI_CSEL_0;
SPI_IOCTL(spi, SPI_SET_LOCK, (VOID *)ioctldata);
```

7.9 SDIO

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

7.9.1 How to Run

5. In the Eclipse, import project for the SDIO sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/SDIO/projects/da16200*
6. The sample application code is written in the following source file: *~/SDK/apps/common/examples/Peripheral/SDIO/src/sdio_sample.c*
   - GPIOA[9:4] needs to connect to the HOST system
   - GPIOA[9] - SDIO_D0, GPIOA[8] - SDIO_D1
   - GPIOA[7] - SDIO_D2, GPIOA[6] - SDIO_D3
   - GPIOA[5] - SDIO_CLK, GPIOA[4] - SDIO_CMD
7. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
8. The sample runs as soon as the boot up is completed.

```
[/DA16200]  # sdio_slave start
 Now the sdio host can access the DA16200
[/DA16200] #
```

Now the DA16200 is ready to receive an SDIO command.

7.9.2 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.

```
/*
 * SDIO Slave
 */
// GPIO[9] - SDIO_D0,  GPIO[8] - SDIO_D1
_da16x_io_pinmux(PIN_EMUX, EMUX_SDs);
// GPIO[5] - SDIO_CLK,  GPIO[4] - SDIO_CMD
_da16x_io_pinmux(PIN_CMUX, CMUX_SDs);
// GPIO[7] - SDIO_D2,  GPIO[6] - SDIO_D3
_da16x_io_pinmux(PIN_DMUX, DMUX_SDs);

// clock enable sdio_slave
DA16X_CLOCK_SCGATE->Off_SSI_M3X1 = 0;
DA16X_CLOCK_SCGATE->Off_SSI_SDIO = 0;

SDIO_SLAVE_INIT();
/* now the sdio host can access the DA16200 */
Printf("now the sdio host can access the DA16200\r\n");
```

7.10 SD/eMMC

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

7.10.1 How to Run

1. In the Eclipse, import project for the SD_EMMC sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/SD_EMMC/projects/da16200*
2. The sample application code is written in the following source file:

~/SDK/apps/common/examples/Peripheral/SD_EMMC/src/sd_emmc_sample.c

3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. The sample will run as soon as boot is complete.

```
[/DA16200]  # emmc sample start
fail / total 0 / 100
[/DA16200] #
```

5. If the SD card is not ready, then the message "emmc_init fail" is returned.
6. Connect GPIOA[9:4] to the SD card socket as shown below.

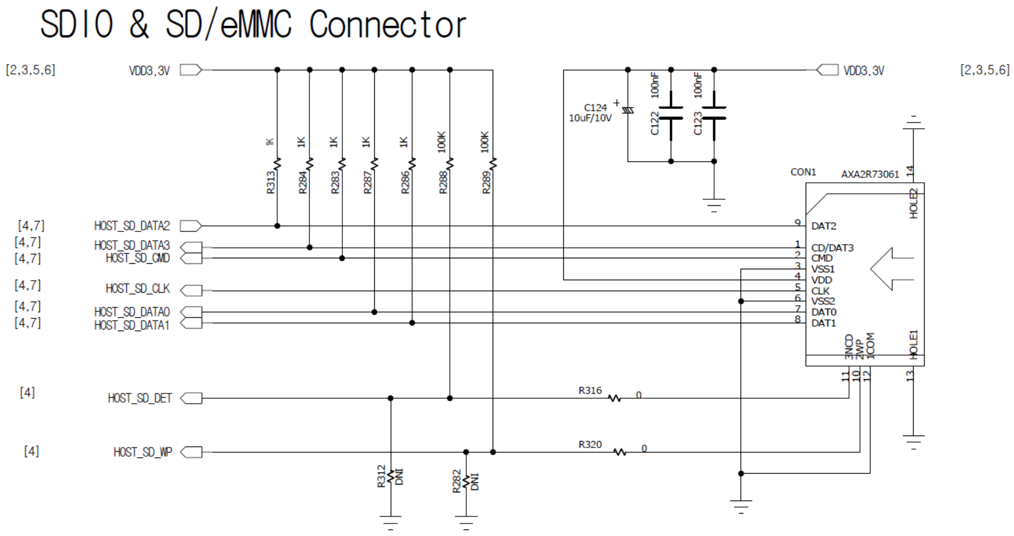


Figure 72: SDIO and SD/eMMC Connector

NOTE

The CMD and DATA pins of the SD card connections are open-drain at initialization. When the SD card initialization is not working normally, it needs to use smaller pull-up resisters for CMD and DATA pins or a shorter length jumper wire of the SD card connections.

a. GPIOA[9] - mSDeMMCIO_D0, GPIOA[8] - mSDeMMCIO_D1
b. GPIOA[7] - mSDeMMCIO_D2, GPIOA[6] - mSDeMMCIO_D3
c. GPIOA[5] - mSDeMMCIO_CLK, GPIOA[4] - mSDeMMCIO_CMD
d. GPIOA[10] is not mandatory (for write protect function).

7.10.2 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.

```
void emmc_init() {
...
    DA16X_CLOCK_SCGATE->Off_SysC_HIF = 0;
    DA16X_SYSCLOCK->CLK_DIV_EMMC = EMMC_CLK_DIV_VAL;
    DA16X_SYSCLOCK->CLK_EN_SDeMMC = 0x01;       // clock enable
...
```

1. Set pin multiplexing.

```
/*
```



```
  * SDIO Master
  */
 // GPIO[9] - mSDeMMCIO_D0,  GPIO[8] - mSDeMMCIO_D1
 _da16x_io_pinmux(PIN_EMUX, EMUX_SDm);
 // GPIO[5] - mSDeMMCIO_CLK,  GPIO[4] - mSDeMMCIO_CMD
 _da16x_io_pinmux(PIN_CMUX, CMUX_SDm);
 // GPIO[7] - mSDeMMCIO_D2,  GPIO[6] - mSDeMMCIO_D3
 _da16x_io_pinmux(PIN_DMUX, DMUX_SDm);
```

2. Create and initialize an SD/eMMC handle.

```
if (_emmc == NULL)
{
    _emmc = EMMC_CREATE();
}
err = EMMC_INIT(_emmc);
```

7.11 User SFLASH Read/Write Example

7.11.1 How to Run

1. In the Eclipse, import project for the SD_EMMC sample application as follows:
   - *~/SDK/apps/common/examples/Peripheral/Sflash_API/projects/da16200*
2. The sample application code is written in the following source file: *~/SDK/apps/common/examples/Peripheral/Sflash_API/src/sflash_sample.c*
3. Build the DA16200 SDK, download the RTOS image to your DA16200 EVB, and reboot.
4. After boot, the sample starts automatically.

```
System Mode : Station Only (0)
>>> DA16x Supp Ver2.7 - 2020_07
>>> MAC address (sta0) : d4:3d:39:10:cc:78
>>> sta0 interface add OK
>>> Start STA mode...
SFLASH_API_SAMPLE
=== SFLASH Write Data ======================
>>>
- (len=128):
[00000000]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000010]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000020]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000030]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000040]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000050]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000060]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
[00000070]  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
```

Figure 73: Sflash Example Sample Test

7.11.2 User Task

The user task of the sflash api sample application is defined as below and executed by the system. SAMPLE_SFLASH should be a unique name to create a task. This test is not related to network initialization and DPM mode.

~/SDK/apps/common/examples/Peripheral/Sflash_API/src/sample_apps.c

```
static const app_task_info_t      sample_apps_table[] =
{ SAMPLE_SFLASH,            user_sflash_test,    1024, USER_PRI_APP(1), FALSE, FALSE,
      UNDEF_PORT,            RUN_ALL_MODE },
};
```

7.11.3 Application Initialization

The user_sflash_test function is run after the basic initialization is complete.

```
void SFLASH_API_sample(void *param)
```



```
{
    /* DO SOMETHING */
    PRINTF("SFLASH_API_SAMPLE\n");

    test_sflash_write();
    vTaskDelay(10);  // Dealy 100 msec

    test_sflash_read();
    vTaskDelete(NULL);

    return ;
}
```

7.11.4 Sflash Read and Write

```
// user sflash APIs

extern UINT user_sflash_read(UINT sflash_addr, VOID *rd_buf, UINT rd_size);
sflash_addr: 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: 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 = NULL;
    UINT       wr_addr;

#define        SFLASH_WR_TEST_ADDR  SFLASH_USER_AREA_START
#define        TEST_WR_SIZE         SF_SECTOR_SZ

    wr_buf = (UCHAR *)malloc(TEST_WR_SIZE);

    if (wr_buf == NULL)
    {
        PRINTF("[%s] malloc fail ...\n", __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 ======================\n");
    user_sflash_write(wr_addr, wr_buf, TEST_WR_SIZE);
}


void test_sflash_read(void)
```



```
{
    UCHAR     *rd_buf = 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 == NULL)
    {
        PRINTF("[%s] malloc fail ...\n", __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);

    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( ), make sure that you get the result before the DPM sleeping API is invoked.

Appendix A

Mosquitto 1.4.14 License

Eclipse Distribution License 1.0

Revision History

Revision	Date	Description
1.3	08-Aug-2022	Fix Index Update the GPIO example project.
1.2	03-June-2022	Update path of example projects. Create 6.1 Crypto API example and merge from AES, DES, HASH and HMAC, DRBG, ECDSA, Diffie-Hellman Key Exchange, RSA PKCS#1, ECDH, KDF, Public Key Abstraction Layer, and Generic Cipher Wrapper. Added Websocket Client Example
1.1	28-Mar-2022	Update logo, disclaimer, copyright.
1.0	26-Oct-2021	First Release.

Status Definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

RoHS Compliance

Renesas Electronics' suppliers certify that its products are in compliance with the requirements of Directive 2011/65/EU of the European Parliament on the restriction of the use of certain hazardous substances in electrical and electronic equipment. RoHS certificates from our suppliers are available on request.

Important Notice and Disclaimer

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu

Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:

https://www.renesas.com/contact/

Trademarks