



Comentario técnico: CTC-105  
 Componente: **Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core**

Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	11/09/20	

En el [CTC-104](#) analizamos los Amazon Web Services (AWS) y desarrollamos la utilización de AWS IoT Core, el servicio de conectividad, utilizando MQTT. Mongoose-OS incorpora un cliente MQTT y una library con soporte para AWS, por lo que en este Comentario Técnico analizaremos la forma de conectarnos a dicha plataforma IoT con un ESP32 y Mongoose-OS.

## Configuración

El proceso de configuración para la utilización de un dispositivo en esta plataforma se desarrolla fundamentalmente en dos partes

1. Primero debemos configurar el proyecto y grabar el dispositivo
2. A continuación debemos generar las credenciales, crear el dispositivo en AWS IoT Core, y grabar las credenciales (y parámetros de configuración) en el dispositivo

Si estamos desarrollando, existe un tercer paso, opcional, que nos puede ahorrar bastante trabajo. Consiste en obtener las credenciales del dispositivo y guardarlas en el proyecto, de modo de no tener que borrarlo y volverlo a cargar en AWS IoT Core.

### Configuración del proyecto

Configuramos en el archivo YAML que describe el proyecto que vamos a utilizar la library de AWS y nos conectaremos como cliente a una red WiFi.

```
libs:
  - origin: https://github.com/mongoose-os-libs/aws
```

Como vemos, sólo hace falta indicar que usamos la library de AWS. Debido a que hemos propuesto una modificación, que ha sido incluida, es posible que en el interim se requiera algo mas.<sup>1</sup>

### Credenciales y AWS IoT Core

Luego de compilado el código (`mos build`) y grabado el microcontrolador (`mos flash`) mediante *mos tool*, necesitamos realizar una serie de tareas. Afortunadamente todo se resume a una llamada a una aplicación. Antes de proseguir es fundamental que hayamos leído el [CTC-104](#) y que hayamos realizado el *quickstart* de AWS IoT Core. Debemos además instalar *aws*, el CLI que nos permite operar por línea de comandos<sup>2</sup>, y configurarlo<sup>3</sup>. También recomendamos no seguir (al menos no completamente) el tutorial de Mongoose-OS sobre este tema.

Las acciones a realizar son:

- cargar el dispositivo en la *registry*

1 Hemos desarrollado una extensión a la library de AWS, la misma ha sido incluida por el fabricante y al momento sólo está disponible utilizando la versión "latest" de *mos tool* o incluyéndola de forma local. Más información en el apéndice.

2 Más información y cómo instalarlo, en la guía del usuario: <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html>

3 Debemos generar y bajar un juego de credenciales con las cuales mediante el comando *aws configure* dejaremos nuestro sistema listo para operar. Las instrucciones las encontramos en la guía del usuario: <https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html>

- generar la clave privada<sup>4</sup>
- solicitar a AWS un certificado de esta clave y obtenerlo.
- generar una política con los permisos de conexión deseados
- asociar la política al certificado y éste al dispositivo
- obtener el certificado de la CA de Amazon
- obtener la dirección del broker MQTT que nos corresponde
- configurar el dispositivo con los datos de conexión de AWS IoT Core correspondientes
- cargar clave privada y certificados (nuestro y CA de Amazon) en el dispositivo

**Afortunadamente, *mos tool* realiza todo esto por nosotros.** Debemos utilizar una versión reciente para incorporar los últimos cambios que Amazon pueda haber introducido.<sup>56</sup>

Lo único que necesitamos hacer para cargar un dispositivo en AWS es ejecutar *mos tool* de la siguiente forma, con el dispositivo conectado:

```
mos aws-iot-setup --aws-region REGION
```

donde REGION corresponde a la región que utilizamos, dato que observamos en la consola de AWS y al hacer el *quickstart* de AWS IoT Core.

El resultado será algo como lo que sigue a continuación:

```
$ mos aws-iot-setup --aws-region us-east-2
Using port /dev/ttyUSB0
AWS region: us-east-2
Connecting to the device...
 esp32 30AEA4807A98 running aws
Current MQTT config: {
  "clean_session": true,
  "client_id": "",
  "cloud_events": true,
  "enable": false,
  "keep_alive": 60,
  "max_qos": 2,
  "max_queue_length": 5,
  "pass": "",
  "reconnect_timeout_max": 60,
  "reconnect_timeout_min": 2,
  "recv_mbuf_limit": 8192,
  "require_time": false,
  "server": "iot.eclipse.org:1883",
  "ssl_ca_cert": "",
  "ssl_cert": "",
  "ssl_cipher_suites": "",
  "ssl_key": "",
  "ssl_psk_identity": "",
  "ssl_psk_key": "",
  "user": "",
  "will_message": "",
  "will_retain": false,
  "will_topic": ""
}
Current AWS Greengrass config: {
  "enable": false,
  "reconnect_timeout_max": 60,
  "reconnect_timeout_min": 2
}
Generating ECDSA private key
Generating certificate request, CN: esp32_807A98
Asking AWS for a certificate...
```

4 El ESP32 posee un hardware para acelerar las operaciones de encriptación del tipo “curva elíptica” (ECDSA, Elliptic Curve Digital Signature Algorithm), es conveniente generar una clave de ese tipo a fin de mantener los tiempos de conexión dentro de lo razonable; por lo tanto la generamos localmente con esas especificaciones y pedimos a Amazon que la firme.

5 Al momento de escribir este texto, se requería la versión 2.18.0, recién salida, pues 2.17.0 no configuraba correctamente la *thing* en la *registry*

6 y obviamente haber instalado y configurado *aws*

## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core

```
Certificate info:
  Subject : CN=esp32_807A98
  Issuer  : OU=Amazon Web Services O=Amazon.com Inc. L=Seattle ST=Washington C=US
  Serial  : 556558518148993822876258600174338009465147119745
  Validity: 2020/09/01 - 2049/12/31
  Key algo: ECDSA
  Sig algo: SHA256-RSA
  ID      : c21de401f51d3461f65806bac5fe8d049540e3f9e617cc0a61b86709654565f3
  ARN     : arn:aws:iot:us-east-2:399048572987:cert/c21de401f51d3461f65806bac5fe8d049540e3f9e617cc0a61b86709654565f3
AWS region: us-east-2
Creating policy "mos-default" ({"Statement": [{"Effect": "Allow", "Action": "iot:*", "Resource": "*"}],
"Version": "2012-10-17"})...
Attaching policy "mos-default" to the certificate...
2020/09/01 17:35:23 This operation, AttachPrincipalPolicy, has been deprecated
Attaching the certificate to "esp32_807A98"...
Writing certificate to aws-esp32_807A98.crt.pem...
Uploading aws-esp32_807A98.crt.pem (1135 bytes)...
Writing key to aws-esp32_807A98.key.pem...
Uploading aws-esp32_807A98.key.pem (227 bytes)...

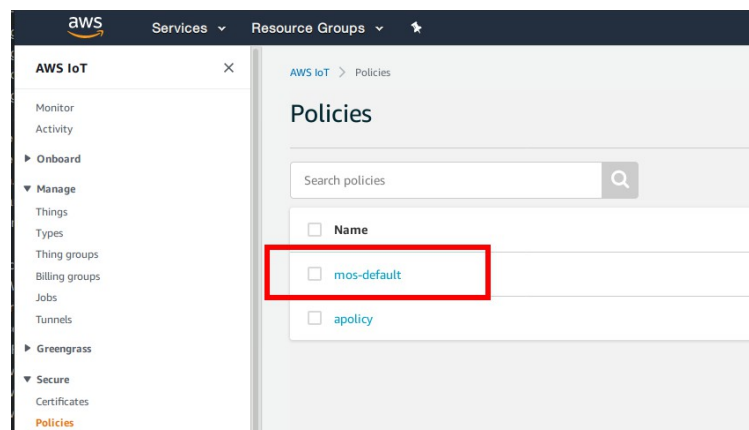
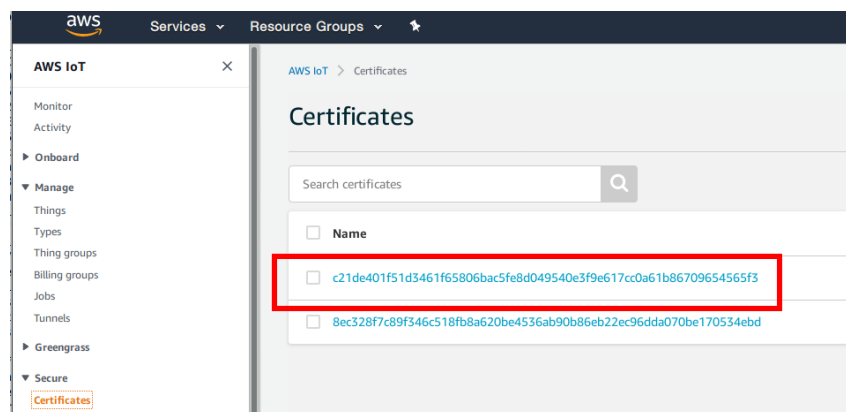
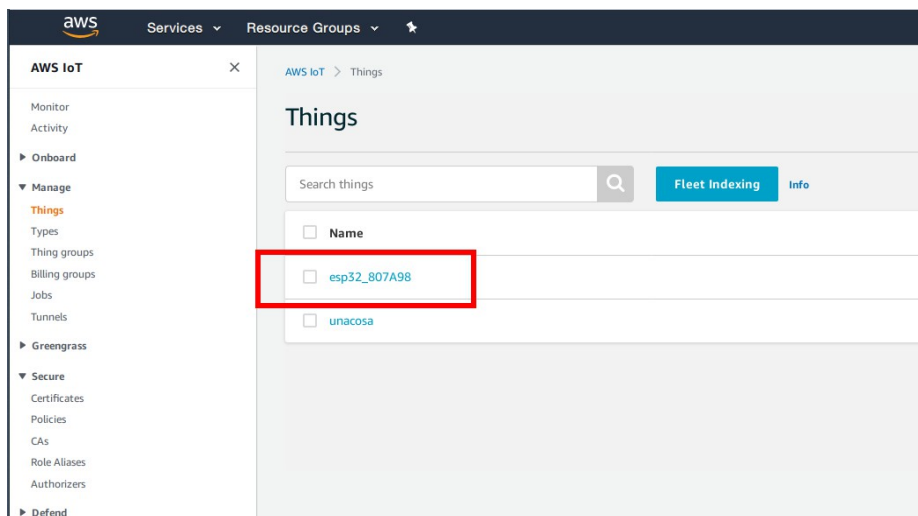
Updating config:
  aws.thing_name =
  mqtt.enable = true
  mqtt.server = a24746f9177qdm-ats.iot.us-east-2.amazonaws.com:8883
  mqtt.ssl_ca_cert = ca.pem
  mqtt.ssl_cert = aws-esp32_807A98.crt.pem
  mqtt.ssl_key = aws-esp32_807A98.key.pem
Setting new configuration...
$
```

Podemos entonces observar mediante *aws* nuestros dispositivos:

```
$ aws iot list-things
{
  "things": [
    {
      "thingName": "esp32_807A98",
      "thingArn": "arn:aws:iot:us-east-2:399048572987:thing/esp32_807A98",
      "attributes": {},
      "version": 1
    },
    {
      "thingName": "unacosa",
      "thingArn": "arn:aws:iot:us-east-2:399048572987:thing/unacosa",
      "attributes": {},
      "version": 1
    }
  ]
}
$
```

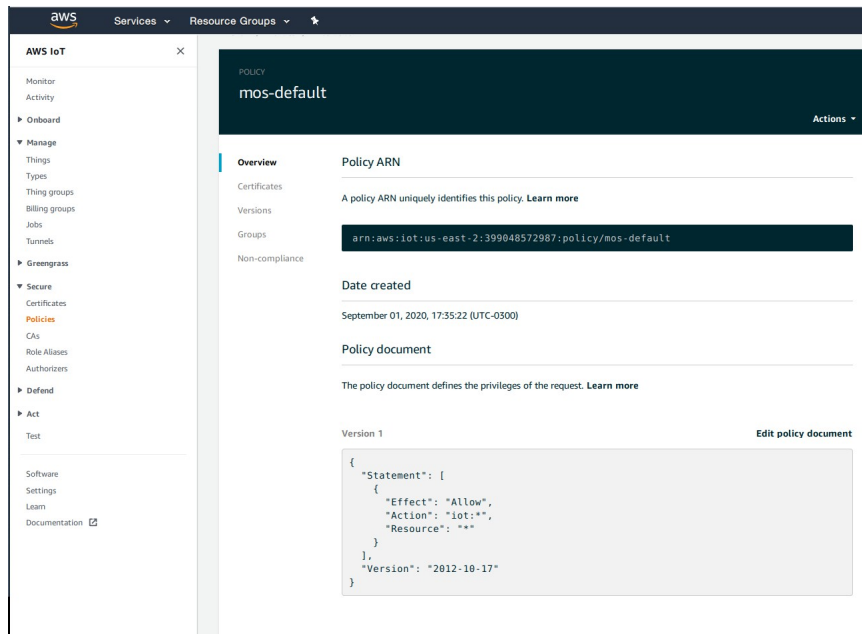
También podemos observar la consola web. Las capturas siguientes muestran el registro del dispositivo, certificado y política de seguridad asociados.

## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core



Respecto a la política de seguridad (*policy*), la que se crea por defecto y podemos apreciar en la siguiente captura es bastante permisiva, para una aplicación real deberemos editarla, de igual modo que tal vez deseemos crear un grupo de dispositivos o aplicar nuestra propia estrategia de despliegue de la red.

## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core



### Durante el desarrollo

Mientras estamos desarrollando código, es común que debamos grabar varias veces el ESP32. Cada vez que lo hagamos perderemos la clave, el certificado, y esta parte de la configuración. Podemos simplemente repetir la operación y *mos tool* generará nuevas claves y actualizará el dispositivo por nosotros. Sin embargo, recomendamos hacer lo que sigue a continuación:

- Obtener clave y certificados del dispositivo, copiándolos en el proyecto. Los nombres de archivo los observamos en el listado generado al crear el dispositivo<sup>7</sup>

```
$ mos get aws-esp32_807A98.key.pem > fs/aws-esp32_807A98.key.pem
$ mos get aws-esp32_807A98.crt.pem > fs/aws-esp32_807A98.crt.pem
$ mos get ca.pem > fs/ca.pem
```

- Copiar la parte de la configuración que agregó *mos tool* en nuestro archivo YAML, *mos.yml*; modificando el listado generado al crear el dispositivo:

```
- ["aws.thing_name", ""]
- ["mqtt.enable", true]
- ["mqtt.server", "a1234b5678cde-ats.iot.us-east-2.amazonaws.com:8883"]
- ["mqtt.ssl_ca_cert", "ca.pem"]
- ["mqtt.ssl_cert", "aws-esp32_807A98.crt.pem"]
- ["mqtt.ssl_key", "aws-esp32_807A98.key.pem"]
```

A partir de ahora, cada vez que regrabemos el ESP32, lo haremos con la misma configuración y credenciales que *mos tool* grabó al crear el dispositivo.

### Operación

Recordemos que además de lo anterior, debemos configurar las credenciales para conectarnos por WiFi a nuestra red (SSID y clave).

---

<sup>7</sup> pero podemos obtenerlos listando los archivos en el ESP32 con `mos call FS.List`

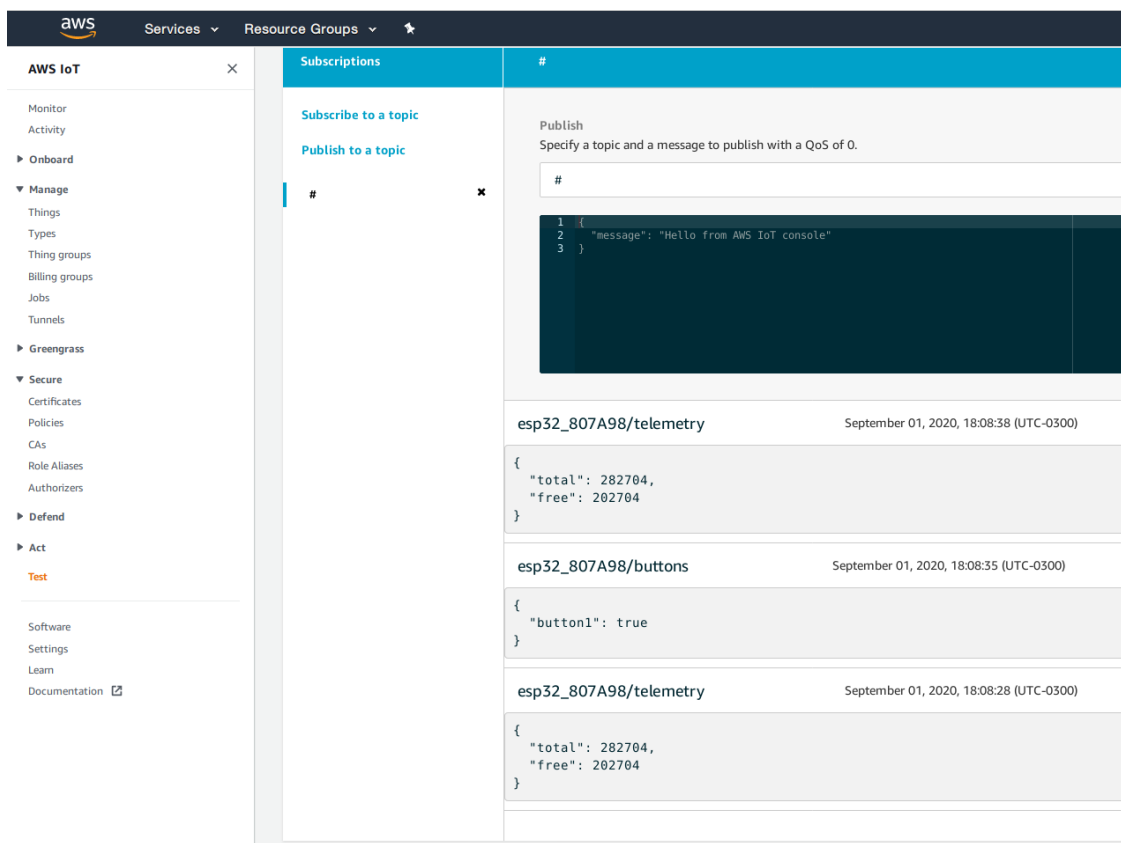
## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core

Iniciado el dispositivo, observaremos en el log si todo funciona como debe, o los errores que se hayan producido.

En los archivos adjuntos disponemos de un ejemplo en detalle. El mismo simula información de telemetría enviando periódicamente la cantidad de memoria libre; podemos enviar mensajes y cambiar el estado de una salida presionando el botón del kit<sup>8</sup> o desde la nube; a la vez que observamos en el log la sincronización de la salida con la *shadow*:

```
[Sep 1 18:06:41.523] mgos_mqtt_conn.c:227 MQTT0 TCP connect ok (0)
[Sep 1 18:06:41.763] mgos_mqtt_conn.c:271 MQTT0 CONNACK 0
[Sep 1 18:06:41.774] init.js:74 Connected to cloud
[Sep 1 18:06:42.119] mgos_aws_shadow.c:273 Subscribed
[Sep 1 18:06:42.130] init.js:74 Connected to cloud
[Sep 1 18:06:42.150] mgos_aws_shadow.c:476 Update: {"state": {"reported": {"out2":false,"out1":false}},
"clientToken": "12ab34cd"}
[Sep 1 18:06:42.172] init.js:49 Connected to AWS
[Sep 1 18:06:42.478] init.js:51 output1 is now: ON
[Sep 1 18:06:47.493] mgos_aws_shadow.c:476 Update: {"state": {"reported": {"out2":false,"out1":true}},
"clientToken": "12ab34cd"}
[Sep 1 18:06:50.878] init.js:32 Send telemetry:OK msg:{"total":282704,"free":202756}
[...]
[Sep 1 18:13:20.904] init.js:42 Send action:OK msg:{"button1":true}
[Sep 1 18:13:20.926] init.js:51 output1 is now: OFF
[Sep 1 18:13:20.943] mgos_aws_shadow.c:476 Update: {"state": {"reported":{"out2":false, "out1":false},
"desired":{"out1":false}}, "clientToken": "12ab34cd"}
```

Los datos de telemetría los obtenemos del broker con cualquier cliente MQTT que se conecte (y tenga los permisos adecuados). También podemos observarlos en el cliente MQTT que se provee en la consola web:



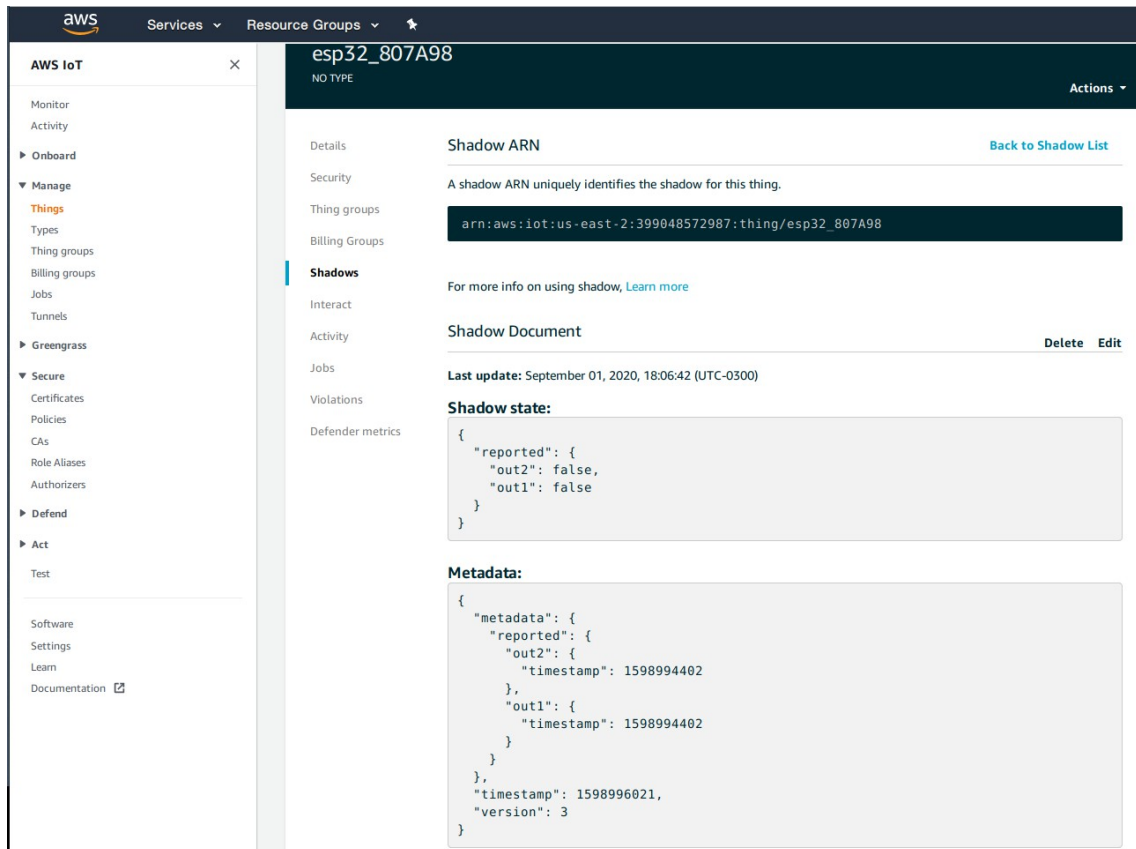
The screenshot shows the AWS IoT console interface. On the left is a navigation menu with categories like 'Monitor', 'Onboard', 'Manage', 'Secure', 'Defend', 'Act', and 'Test'. The main area is titled 'Subscriptions' and contains a 'Publish' form and a list of received messages. The messages are as follows:

Subscription	Timestamp	Message
esp32_807A98/telemetry	September 01, 2020, 18:08:38 (UTC-0300)	<pre>{   "total": 282704,   "free": 202704 }</pre>
esp32_807A98/buttons	September 01, 2020, 18:08:35 (UTC-0300)	<pre>{   "button1": true }</pre>
esp32_807A98/telemetry	September 01, 2020, 18:08:28 (UTC-0300)	<pre>{   "total": 282704,   "free": 202704 }</pre>

8 Requiere la última versión, más información en el apéndice

## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core

La primera vez que el dispositivo se conecta a AWS, reporta el estado de sus salidas y crea la clave *reported* en la shadow:



The screenshot shows the AWS IoT console interface for a device named `esp32_807A98`. The left sidebar contains navigation options for AWS IoT, including Monitor, Activity, Onboard, Manage (Things, Types, Thing groups, Billing groups, Jobs, Tunnels), Greengrass, Secure (Certificates, Policies, CAs, Role Aliases, Authorizers), Defend, Act, and Test. The main content area displays the shadow state for the device. The Shadow ARN is `arn:aws:iot:us-east-2:399048572987:thing/esp32_807A98`. The Shadow Document is shown with a last update of September 01, 2020, 18:06:42 (UTC-0300). The Shadow state is a JSON object:

```
{
  "reported": {
    "out2": false,
    "out1": false
  }
}
```

The Metadata section shows the following JSON object:

```
{
  "metadata": {
    "reported": {
      "out2": {
        "timestamp": 1598994402
      },
      "out1": {
        "timestamp": 1598994402
      }
    }
  },
  "timestamp": 1598996021,
  "version": 3
}
```

Podemos actualizar la configuración (el estado de una salida en este caso) desde la consola web. Si es la primera vez que se modifica la *shadow* (no hemos alterado el estado de la salida desde que creamos el dispositivo en AWS), deberemos agregar manualmente la clave *desired* pues no ha sido creada aún.

## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core

The screenshot shows the AWS IoT console interface. On the left is a navigation menu with categories like 'Monitor', 'Onboard', 'Manage', 'Greenpress', 'Secure', 'Defend', 'Act', and 'Test'. The main content area is titled 'THING esp32\_807A98'. It displays the 'Shadow ARN' as 'arn:aws:iot:us-east-2:399048572987:thing/esp32\_807A98'. Under the 'Shadows' section, it shows the 'Shadow Document' with a 'Last update' of 'September 03, 2020, 12:42:07 (UTC-0300)'. The 'Shadow state' is shown as a JSON object:

```
1 {
2   "desired": {
3     "out1": false,
4     "out2": false
5   },
6   "reported": {
7     "out2": false,
8     "out1": true
9   }
10 }
```

This screenshot shows the same AWS IoT console page after an update. A green success banner at the top reads 'Success Successfully updated thing shadow.' The 'Shadow Document' section now shows a 'Last update' of 'September 03, 2020, 13:06:08 (UTC-0300)'. The 'Shadow state' JSON object is:

```
{
  "desired": {
    "out1": false,
    "out2": false
  },
  "reported": {
    "out2": false,
    "out1": false
  }
}
```

The 'Metadata' section shows:

```
{
  "metadata": {
    "desired": {
      "out1": {
        "timestamp": 1599149160
      }
    }
  }
}
```

También podemos hacerlo mediante `aws`<sup>9</sup>:

```
$ aws iot-data update-thing-shadow --cli-binary-format raw-in-base64-out --thing-name
esp32_807A98 --payload '{"state": {"desired": {"out1": true}}}' /dev/stdout
{"state":{"desired":{"out1":true},"metadata":{"desired":{"out1":
{"timestamp":1599257364}}},"version":196,"timestamp":1599257364}
$
```

<sup>9</sup> La versión 2 de AWS CLI funciona por defecto empleando datos en base64, para poder pasarle algo tan simple como texto JSON debemos indicarle el formato como hemos hecho e indicamos *en bastardilla*. La versión 1 no lo requiere, y es la forma que habitualmente encontraremos en la documentación al momento de escribir este texto; por ejemplo en el tutorial que simula y explica la operación de la *shadow*.



Por supuesto, nuestra aplicación IoT final hará todo mediante las APIs correspondientes, que es la forma en que opera el comando *aws*.

## Desarrollo de nuestra aplicación

Describimos brevemente las APIs disponibles. Para más información podemos consultar el código fuente de la library en el repositorio Github.<sup>10</sup>

### Desarrollo en mJS

Incluimos las APIs necesarias declarando:

```
load('api_aws.js')
load('api_mqtt.js')
load('api_shadow.js')
```

Sabremos si estamos conectados llamando a `AWS.isConnected()`

Podemos publicar mensajes utilizando la API MQTT como hemos visto en otros comentarios técnicos. El nivel de calidad de servicio no puede ser 2, podemos utilizar 1: `MQTT.pub(topic, msg, 1)`, o 0: `MQTT.pub(topic, msg)`

Para sincronizarnos con la *shadow* emplearemos la API correspondiente; Mongoose-OS provee una library con una *shadow* independiente de la plataforma que a su vez se integra con AWS. Registraremos un handler llamando a `Shadow.addHandler(handler)`

La library manejará el diálogo con AWS y recibiremos un evento, el cual se traducirá a una llamada a nuestra función handler con el tipo de evento. Allí procederemos a reportar nuestro estado si se trata de un evento de conexión (reportamos el estado de las salidas, por ejemplo) o a operar sobre éste (cambiamos el estado de las salidas, por ejemplo) en función de lo indicado en un evento *delta*:

```
Shadow.addHandler(function(event, obj) {
  if (event === 'CONNECTED') {
    Shadow.update(0, state);
  } else if (event === 'UPDATE_DELTA') {
    for (let key in obj) {
      if (key === 'out1') {
        state.out1 = obj.out1;
        // modificamos el estado de la salida acorde a lo pedido
      } /* ...*/ else {
        Log.print(Log.WARN, 'Unknown key: ' + key);
      }
    }
    // Luego de actualizar, reportamos
    Shadow.update(0, state);
  }
});
```

En caso de necesitar controlar localmente la salida, como hemos hecho, es necesario utilizar directamente la library de AWS<sup>11</sup>, en particular la función `AWS.Shadow.update_ext()`, que permite ingresar manualmente las claves que se envían. Podemos enviar el estado deseado y esperar que AWS nos responda con un *delta*, esto genera dos cambios en la *shadow* y es equivalente a operar desde una aplicación. Dado que es conveniente actualizar la salida inmediatamente, a fin de evitar desorientar al usuario con la demora, podemos entonces enviar simultáneamente estado deseado y reporte de estado al mismo tiempo, como hemos hecho en el ejemplo; esto genera solamente un cambio en la *shadow*.<sup>12</sup>

<sup>10</sup> <https://github.com/mongoose-os-libs/aws>

<sup>11</sup> Requiere la última versión, más información en el apéndice

<sup>12</sup> En detalle, no se genera un *delta*. La documentación disponible no confirma ni refuta esta forma de operación, las aplicaciones deberían observar el estado de la *shadow* y no los *deltas*, por lo que no habría inconvenientes si éstas se escriben correctamente. Cada cambio genera tráfico y este tráfico tiene un costo, negligible tal vez.

```
AWS.Shadow.update_ext(0, {desired: {out1: <true o false>}}); // sólo envía pedido
AWS.Shadow.update_ext(0, {desired: {out1: state.out1}, reported: state}); // pedido y cambio
simultáneos
```

Para recibir información de eventos de conexión y desconexión a la red, podemos también registrar los handlers apropiados:

```
Event.addHandler(Event.CLOUD_CONNECTED, function (ev, evdata, ud) {
}, null);
Event.addHandler(Event.CLOUD_DISCONNECTED, function (ev, evdata, ud) {
}, null);
```

## Desarrollo en C

Para enviar y recibir mensajes, deberemos utilizar la API de MQTT.<sup>13</sup>

Para la *shadow*, usaremos la API correspondiente incluyendo `mgos_shadow.h`<sup>14</sup>

Para el resto, incluyendo operar sobre las salidas, necesitamos incluir la API específica de AWS incluyendo `mgos_aws_shadow.h`

Sabremos si estamos conectados llamando a `bool mgos_aws_is_connected()`

Para recibir eventos de conexión y desconexión, deberemos registrar un handler de la manera habitual y documentada.<sup>15</sup>

Los eventos son: `MGOS_EVENT_CLOUD_CONNECTED` y `MGOS_EVENT_CLOUD_DISCONNECTED`.

## Apéndice

Al momento de escribir este texto, la extensión a la library de AWS que hemos desarrollado se ha incluido en el branch *master* de Github. Sin embargo, cuando compilamos, la versión que obtenemos depende de la versión de *mos tool* que estemos utilizando. Tenemos tres formas de trabajar:

1. Utilizar la versión “release” de *mos tool*, que incluye la library corriente y no nos permite controlar localmente las salidas
2. Utilizar la versión “latest” de *mos tool*, que incluye todo lo más reciente (incluida la actualización deseada de la library), al riesgo de incorporar alguna inestabilidad.
3. Compilar la library localmente, operando con la versión “release” que es (por definición) más curada y estable.

En el caso 1, operamos como se describió excepto que no podemos controlar localmente la salida. En el archivo provisto, el ejemplo está en el directorio *aws*.

En el caso 2, luego de salvar convenientemente la versión de *mos tool* procedemos a actualizarla escribiendo `mos upgrade latest` y es recomendable luego cambiarle el nombre a (por ejemplo) *mos\_latest* y dejar la versión “release” con el nombre *mos*. Compilamos invocando esta nueva versión, por ejemplo: `mos_latest build`. En el archivo provisto, el ejemplo está en el directorio *aws\_latest*.

En el caso 3, deberemos clonar el repositorio Github (en el archivo provisto ya lo incluimos), crear el directorio *deps* y copiarlo allí. Cada vez que recompilemos deberemos<sup>16</sup> borrar ese directorio (y el directorio *build*) y repetir la copia. Indicamos en el archivo YAML que usamos una copia local de esta library de la siguiente forma:

```
libs:
  - name: aws
```

<sup>13</sup> <https://mongoose-os.com/docs/mongoose-os/api/net/mqtt.md>

<sup>14</sup> [https://github.com/mongoose-os-libs/shadow/blob/master/include/mgos\\_shadow.h](https://github.com/mongoose-os-libs/shadow/blob/master/include/mgos_shadow.h)  
<https://mongoose-os.com/docs/mongoose-os/api/net/shadow.md>

<sup>15</sup> [https://mongoose-os.com/docs/mongoose-os/api/core/mgos\\_event.h.md](https://mongoose-os.com/docs/mongoose-os/api/core/mgos_event.h.md)

<sup>16</sup> No es que “debamos”, pero a veces cuando la compilación falla ante un error de sintaxis quedan archivos en esos directorios y si compilamos de manera remota *mos tool* intenta enviarlos, lo cual generalmente falla porque el archivo resultante es demasiado grande; lo ideal es repetir este procedimiento para agilizar la operación.

## CTC-105, Conexión del ESP32 con Mongoose-OS a Amazon Web Services IoT Core

Nótese que no debemos incluir el *origin* y que *name* es precedido del guión.

En el archivo provisto, el ejemplo está en el directorio *aws\_interim*. En el mismo, las acciones a realizar antes de cada compilación con la library local, en un intérprete de comandos tipo *bash*, son:

```
rm -rf deps build      # borramos los directorios build y deps
mkdir deps             # re-creamos el directorio deps
cp -r aws deps/       # copiamos el repositorio aws como subdirectorio de deps
```

Para más información sobre el proceso de compilación y como desarrollar libraries podemos consultar la documentación.<sup>17</sup>

---

<sup>17</sup> <https://mongoose-os.com/docs/mongoose-os/userguide/build.md>