Nota de Aplicación: CAN-106
Título: **ESP32 + Mongoose-OS = edGarDo**
Autor: Sergio R. Caprile, Senior R&D Engineer

Revisiones	Fecha	Comentarios
0	07/04/20	

Les presentamos a edGarDo, un pequeño dispositivo que permite controlar (por ejemplo) un portón automático desde un celular, tablet, computadora o equivalente, e incluso automatizar su operación o ser controlado de manera remota y por voz.

El dispositivo en sí solamente posee conectividad Bluetooth Low Energy (BLE) y WiFi, y un simple firmware con funciones de control e identificación. Esto le permite ser identificado de manera automática para ser utilizado en un sistema de automatización del hogar, a través del cual obtenemos las demás funciones, incluyendo la seguridad en la conexión y facilidad de ésta.

Adicionalmente, mediante BLE podemos controlarlo desde un celular desde el auto sin necesidad de utilizar la Internet y limitando su operación a la cercanía y conocimiento del dispositivo controlador.

Tabla de Contenidos

El control

Dado que todos los controladores de portones automáticos son diferentes, lo que en esta nota de aplicación realizamos es activar durante un tiempo razonable un pin de I/O. Esto puede utilizarse para cerrar un contacto de relé que dé la indicación pertinente al controlador, o para simular la presión del botón en un llavero, a gusto y posibilidades del consumidor.
Si bien disponemos de entradas para observar el estado del portón, sólo se opera sobre dicho pin por un tiempo y no se diferencia entre abrir y cerrar, lo cual suele ser el común en los controladores más conocidos. De no ser ésta la realidad del interesado, las modificaciones son simples.

Conexiones

Pin	Función
IO0	Botón para ingresar a modo de asociación (identificación de dispositivos autorizados)
IO12	Salida de control, conectar a un transistor.
IO13	LED para indicar conexión
IO14	LED para indicar modo de asociación activado
IO25	Entrada portón cerrado
IO26	Entrada portón abierto (conectar a masa si no se usa)
IO27	Entrada portón en movimiento (dejar libre si no se usa)

Las salidas para los LEDs y la salida de control son activas en alto, entregan 3,3V y no es recomendable extraerles corriente.

El dispositivo

A los fines prácticos, utilizamos un ESP32-WROVER, kit del ESP32[1]. Se trata de un microcontrolador algo particular, evolución del conocido ESP8266, más amigable de utilizar, y con mucha más memoria y CPU disponible, que permite afrontar holgadamente las tareas que le pedimos, aún desarrollando en lenguajes de más alto nivel.
Una versión más afinada del circuito puede utilizar un módulo ESP32-WROOM-32, agregando el clásico circuito RC de reset y una alimentación de 3,3V con un regulador como el LD1117, dado que al inicio y al activar Wi-Fi hay unos picos importantes de corriente. Para el desarrollo lo hemos omitido dado que el kit se alimenta del mismo USB que usamos para programación y debugging y obviamente incorpora un circuito de reset.

El entorno de desarrollo

El fabricante, Espressif, provee un entorno de desarrollo que funciona sobre FreeRTOS (ESP-IDF[2]) y permite programar el dispositivo en C.
Aquí nos hemos movido a algo un poco más cercano a la aplicación y utilizamos Mongoose-OS, un framework que provee una interfaz del tipo event-driven que podemos explotar para desarrollar una prueba de concepto rápidamente en JavaScript, o también programar en C. No reemplaza al entorno del fabricante sino que funciona sobre éste, y podemos utilizar ambos a la vez si gustamos, pero desde el entorno de Mongoose-OS.
Entre otros servicios, Mongoose-OS provee una API del tipo RPC (Remote Procedural Call) que podemos aprovechar y extender. El acceso a esta API es indistinto del transporte que usemos para acceder, sea un puerto serie, BLE GATT services, o HTTP, entre otros. Por tal motivo, hacemos uso y abuso de la misma para las tareas de control externo del dispositivo, es decir, el control del dispositivo tanto por página web (WiFi) como por BLE lo haremos a través de esta API RPC.

1 https://www.espressif.com/en/products/hardware/esp32/overview
2 https://docs.espressif.com/projects/esp-idf/en/latest/esp32/

Operación por BLE

Debido a que utilizamos el API RPC de Mongoose-OS, hemos decidido además evitarnos la tarea de definir un perfil GATT[1], sus servicios y características. Simplemente aprovechamos el ya existente y operamos por RPC.
A fin de garantizar un mínimo de seguridad, el dispositivo no permite conexiones excepto desde aquellos dispositivos que conoce; es decir, aquellos con los que se ha vinculado mediante bonding. Para poder hacerlo, el usuario indica al dispositivo esta intención presionando un botón, y entonces el dispositivo estará dispuesto a hacerlo por un tiempo razonable, tildando así de ”conocidos” a nuevos dispositivos.
Incorporamos además al API RPC las funciones necesarias no sólo para controlar (activar) el portón y consultar el estado del mismo, sino para listar y borrar dispositivos de la lista de bonded.
La operación por BLE es una de las formas de control disponibles cuando estamos fuera de nuestra red, pero cerca del dispositivo (a la distancia de operación de BLE).

Conexión a una red WiFi

El ESP32, al igual que su antecesor el ESP8266, puede funcionar como un Access Point, esto simplifica la operatoria de configuración inicial.
En esta nota, proveemos un soporte en JavaScript basado en la API RPC ya mencionada, que permite configurar la conexión a una red WiFi desde una página web servida por el ESP32, al cual configuramos para iniciar como Access Point. Simplemente nos conectamos a él. También podemos operarlo de este modo si así lo deseamos; el Access Point se desconecta automáticamente una vez configuramos al dispositivo para conectarse como *station* a una red, por lo que deberemos modificar esta configuración (y poner un password algo más seguro) si deseamos que permanezca operativo.
La API RPC nos permite además configurar WiFi desde BLE, si escribimos una aplicación que lo haga, desde un smartphone Android o iOS. La operatoria es idéntica a la de control, que detallaremos más adelante.

Operación por WiFi

La misma API que utilizamos para operación por BLE es accesible por los otros medios de transporte, lo cual obviamente incluye a HTTP. Incorporamos además unas páginas web muy sencillas con JavaScript que nos permiten controlar el dispositivo conectándonos a su servidor web. Nótese que para esto debemos asignarle una dirección conocida al momento de vincularlo a la red WiFi, lo cual debemos saber hacer o pedírselo al administrador de la red. No es necesario si no pensamos utilizar esta funcionalidad, lo cual es muy probable ya que como veremos hay mejores y más elegantes formas de controlar el dispositivo.
En la versión actual, la manipulación de la base de datos de dispositivos autorizados se realiza por WiFi. Es posible accederla por BLE o cualquier otro transporte, pero no hemos escrito ejemplos de como hacerlo.

Operación por un sistema de automatización del hogar

El dispositivo, una vez conectado a una red WiFi, intenta conectarse además a un broker MQTT[2]. Podemos proveer uno en nuestro sistema o utilizar uno externo. El sistema de automatización que empleamos (openHAB) tiene la flexibilidad suficiente para que esto no sea un problema.
Conectado al broker MQTT, el dispositivo se identifica implementando la Homie Convention 4.0. Esto consiste en publicar una serie de mensajes con retención en tópicos conocidos, de forma que otros dispositivos reconozcan su existencia. El sistema de automatización se suscribe también al mismo broker y recibe estos mensajes, mostrándonos la existencia del dispositivo para que procedamos a su configuración.
Con el dispositivo ingresado y configurado, el sistema se entera del estado del portón por los mensajes publicados, y puede controlar su estado publicando mensajes de control en determinados tópicos, anunciados por el dispositivo al iniciar.

---

1 GATT (Generic ATTribute Profile) define el modo en el que dos dispositivos BLE (Bluetooth Low Energy) intercambian datos. Define así servicios que pueden anunciarse o interrogar su existencia, los cuales agrupan características, que contienen los datos en sí a ser leídos o escritos. https://www.bluetooth.com/specifications/gatt/

2 MQTT (MQ Telemetry Transport) es un protocolo de mensajería que implementa el modelo publicar/suscribirse. Su función es desacoplar la aplicación de los datos y proveer una cierta confiabilidad en función de determinados niveles de servicio. Recomendamos la lectura del CTC-087. Página oficial: http://mqtt.org/

Por supuesto que cualquier sistema o programa que se suscriba a los tópicos que indica la Homie Convention en el broker MQTT será capaz de realizar las funciones deseadas. Podemos, por ejemplo, desarrollarlo en Node-Red[1] una vez que tengamos la prueba de concepto funcionando.

Aplicación para smartphones

En general, estos sistemas proveen una aplicación para smartphones, y openHAB no es la excepción. Como veremos, esto incluye acceso desde Internet.

Operación por Internet (desde fuera de nuestra red)

El entorno de automatización del hogar provee un esquema de acceso remoto llamado ”openHAB Cloud”. No es necesario configurar especialmente nuestra red, ya que nuestro openHAB se conecta a openHAB Cloud, sólo necesitamos que se permitan conexiones salientes y configurar correctamente la autenticación entre nuestro openHAB y openHAB Cloud.
Para poder ver nuestro sistema (y de paso controlar el portón...) nos conectamos a openHAB Cloud, ingresamos nuestras credenciales, y eligiendo el sistema a visualizar ya podemos ver la pantalla de control como si estuviéramos en nuestra red. Esto sólo requiere de un navegador web o la aplicación de smartphone.

Operación por voz (Amazon Alexa, Google Assistant, otros asistentes)

Este servicio es prestado por openHAB Cloud. Configurando los ítems que definen al portón de determinada manera, y exportándolos para ser accedidos de forma remota, luego de configurar el acceso en openHAB Cloud y en nuestro asistente (o asistentes) preferido(s), ya podemos pedir la operación del portón por voz. Es decir, no es una característica del dispositivo, no se requiere implementar *skills* ni *actions*[2] ni conectarse o habilitar conexiones entrantes; una vez que somos un dispositivo visible y controlable por openHAB, ya somos controlables por cualquiera de estos asistentes siempre y cuando lo habilitemos y configuremos tanto en openHAB como en openHAB Cloud.
En nuestro caso sólo hemos configurado y probado en inglés, lo cual significa:
”open garage door” o ”activate garage door” (pero bien pronunciado), aunque nos han contado que también puede hacerse en nuestra lengua nativa (si está dentro de las soportadas por los asistentes y por openHAB)

Detalle del código por áreas

Describimos aquí algunas secciones del código con fines didácticos. La totalidad del mismo se encuentra en un repositorio de Github[3], junto con su respectiva licencia, la cual permite la utilización para fines recreativos reconociendo a sus creadores, y aplicaciones comerciales previo acuerdo con Cika Electrónica.

Estado y control

Leemos los pines donde observamos el estado del portón, y reportamos, si hay conexión, ante cambios o cada un segundo.

```
let state_handler = function(forced) {
    if(GPIO.read(activity_pin) === 0) {           // 'activity' takes precedence (1 if not used)
        door_state = 'moving';
    } else if(GPIO.read(closed_pin) === 0) {      // then 'closed'
        door_state = 'closed';
    } else if(GPIO.read(open_pin) === 0) {        // then 'open' (0 if not used)
        door_state = 'open';
    } else {                                      // if all are true, asume half-open
        door_state = 'halfopen';
    }
    if(door_state !== former_door_state || forced) {
        Log.print(Log.INFO, 'Door state changed to ' + door_state);
        if(mqtt_connected === true) {
            publish(hab_door_topic, door_state);
        }
        former_door_state = door_state;
    }
};
```

1 https://nodered.org/
2 La configuración del asistente sí requiere que agreguemos la *skill* y/o *action* correspondiente a openHAB Cloud, provista por éste.
3 https://github.com/CikaElectronica/edGarDo

```
// Update state every second
Timer.set(1000, Timer.REPEAT, state_handler, false);
```

Operamos, ante un pedido, sobre el pin de control, manteniendo activo por 1,5 segundos

```
let activate = function () {
    Log.print(Log.INFO,'Operating on door control');
    GPIO.write(relay_pin, 1);
    Timer.set(1500, 0, function() {
        Log.print(Log.DEBUG,'Timer expired, deactivating');
        GPIO.write(relay_pin, 0);
    }, null);
};
```

servicios RPC de control

El código siguiente crea un procedimiento llamado *Door.Set*, el cual al ser llamado ocasiona la operación del portón.

```
RPC.addHandler('Door.Set', function(args) {
         // no args parsing required
    Log.print(Log.DEBUG,'Control RPC function called');
    activate();
    state_handler(false);
    let response = {
         rc: (1) ? 'OK' : 'Failed'
    };
    return response;
});
```

El código siguiente genera el complementario, para obtener el estado del portón: *Door.Get*.

```
// set RPC command: report door state
RPC.addHandler('Door.Get', function(args) {
         // no args parsing required
    Log.print(Log.DEBUG,'Status RPC function called');
    state_handler(false);
    let response = {
         rc: door_state
    };
    return response;
});
```

La forma de llamar a uno de estos procedimientos consiste en utilizar el servicio que Mongoose-OS provee como transporte. Esto puede hacerse conectándonos por BLE, como describimos más adelante, o por ejemplo por HTTP, con el siguiente fragmento de código que utiliza *zepto*, en la página web[1]:

```
<html>
<head>
<script src="zepto.min.js"></script>
</head>
<body>

      <div class="form-control"><button href="#" class="btn btn-sm btn-c"
id="control">Control</button></div>

<script>
  $('#control').on('click', function() {
    $.ajax({
      url: '/rpc/Door.Set',
      data: JSON.stringify({}),
      type: 'POST',
      success: function(data) {
      },
    })
  });
</script>
<body>
</html>
```

El llamado a */rpc/Door.Set* o */rpc/Door.Get* no requiere parámetros, el resultado se devuelve dentro de un objeto JSON.

1 El navegador solicita al servidor web en nuestro dispositivo la página, la carga, interpreta el código, pide los scripts mencionados (*zepto* por ejemplo), los carga, y establece el esquema. Cuando el usuario presiona el botón, ejecuta el script correspondiente, el cual inicia otro pedido, esta vez un HTTP POST al recurso */rpc/Door.Set*. Nuestro dispositivo interpreta el pedido y ejecuta la acción, informando al navegador el resultado.

Más adelante, en la sección sobre desarrollo de una aplicación para smartphone, podemos observar una captura de pantalla con el procedimiento que realizamos por BLE. En azul se halla recuadrado el momento de ejecución del procedimiento *Door.Get*.

MQTT

Al detectar una conexión al broker, actualizamos la información que el broker MQTT (y por consiguiente el sistema de automatización) tiene de nosotros

```
MQTT.setEventHandler(function(conn, ev, edata) {
    if (ev === MG_EV_MQTT_CONNACK) {
        mqtt_connected = true;
        GPIO.blink(onlineled_pin, 100, 3900);
        Log.print(Log.INFO, 'MQTT connected');
        // auto-discovery
        homie_init();
        state_handler(true);
    }
    else if (ev === MG_EV_CLOSE) {
        mqtt_connected = false;
        GPIO.blink(onlineled_pin, 100, 100);
        Log.print(Log.ERROR, 'MQTT disconnected');
    }
}, null);
```

Esto registra una función que será llamada ante un evento MQTT. Entre los posibles, elegimos *conexión* y *desconexión*.
A continuación, definimos nombres simples para acceder a los nombres de los tópicos que Homie nos indica

```
let client_id = Cfg.get('device.id');
let thing_id = tolowercase(client_id.slice(client_id.length-6, client_id.length));
//let dev_name = 'edGarDo_' + thing_id;
let base_topic = 'homie/edgardo' + thing_id;
let state_topic = base_topic + '/$state';
let hab_door_topic = base_topic + '/door/state';            // see homie_init
let hab_button_topic = base_topic + '/button/state';        // see homie_init
let hab_control_topic = hab_button_topic + '/set';
```

Cuando queremos publicar algo (en respuesta a un evento), llamamos a esta función. Hemos visto en la sección de estado y control que allí es llamada. MQTT.pub() es provisto por Mongoose-OS, no debemos esperar ni hacer handshakes, sólo llamar a la función y observar el resultado.

```
let publish = function (topic, msg) {
    let ok = MQTT.pub(topic, msg, 1, true);         // QoS = 1, retain
    Log.print(Log.INFO, 'Published:' + (ok ? 'OK' : 'FAIL') + ' topic:' + topic + ' msg:' +  msg);
    return ok;
};
```

Nos suscribimos al tópico de control, y definimos la función que atiende los mensajes que ingresen por éste. Dicha función compara el mensaje con lo esperado (el uso de Homie instruye al sistema de automatización sobre cómo hablar con nosotros y decirnos lo que esperamos y entendemos) y obra en consecuencia.

```
MQTT.sub(hab_control_topic, function(conn, topic, command) {
    Log.print(Log.DEBUG, 'rcvd ctrl msg:' + command);
    if ( command === 'true' ) {
        Log.print(Log.DEBUG,'MQTT control received');
        activate();
    } else if ( command === 'false' ) {

    } else {
        Log.print(Log.ERROR, 'Unsupported command');
    }
    state_handler(false);
}, null);
```

Homie

La inspiración y detalles de cómo realizar esto los obtuvimos de uno de los ejemplos que existen en la página de Github de Mongoose-OS, con permiso del autor[1].
La especificación Homie nos indica cómo debemos nombrar los tópicos para que nuestro dispositivo sea reconocido y controlado por un sistema compatible.

```
let homie_init = function () {
    publish(state_topic, 'init');
    publish(base_topic + '/$homie', '4.0.0');
```

1 https://github.com/mongoose-os-apps/sonoff-basic-openhab2

```
    publish(base_topic + '/$name', 'edGarDo, ed Garage Door opener');
    publish(base_topic + '/$extensions', '');
    publish(base_topic + '/$nodes', 'button,door');
    publish(base_topic + '/button/$name', 'Control button');
    publish(base_topic + '/button/$type', 'Push button');
    publish(base_topic + '/button/$properties', 'state');
    publish(hab_button_topic + '/$name', 'Button state');
    publish(hab_button_topic + '/$settable', 'true');
    publish(hab_button_topic + '/$datatype', 'boolean');
    publish(base_topic + '/door/$name', 'Door');
    publish(base_topic + '/door/$type', 'Open Close');
    publish(base_topic + '/door/$properties', 'state');
    publish(hab_door_topic + '/$name', 'Door state');
    publish(hab_door_topic + '/$datatype', 'enum');
    publish(hab_door_topic + '/$format', 'open,moving,halfopen,closed');
    publish(hab_button_topic, 'false');
    publish(state_topic, 'ready');
};
```

Como vemos, se trata de publicar mensajes en tópicos conocidos, en los cuales indicamos qué somos y cómo operamos. En este caso nos declaramos como un dispositivo que tiene un botón operable y reporta el estado de algo llamado *door*, y qué valores posibles reportará ese algo.

Bonding

Respondemos a la presión del botón habilitando por un tiempo la posibilidad de hacer bonding, con lo cual el dispositivo que lo haga pasa a estar en la lista de dispositivos autorizados.
En este caso hemos dejado el anuncio habilitado, porque facilita la operación de la aplicación de smartphone, es posible habilitarlo también sólo durante este tiempo.
Un detalle: en las dos primeras líneas podemos apreciar cómo configurar una función en C para ser accedida desde JavaScript.

```
let tobindornottobind = ffi('bool mgos_bt_gap_set_pairing_enable(bool)');
let advertising = ffi('bool mgos_bt_gap_set_adv_enable(bool)');

// handle button: enable (advertising and) binding for some time
GPIO.set_button_handler(button_pin, GPIO.PULL_NONE, GPIO.INT_EDGE_POS, 100, function() {
//    let ok = advertising(true);
//    Log.print(Log.DEBUG,'Button pressed, advertising call:' + (ok ? 'success' : 'failure'));
    let ok = tobindornottobind(true);
    Log.print(Log.DEBUG,'Button pressed, pairing call:' + (ok ? 'success' : 'failure'));
    GPIO.blink(bindled_pin, 200, 300);
    Timer.set(30000, 0, function() {
        let ok = false;
//        ok = advertising(false);
//        Log.print(Log.DEBUG,'Timer expired after button pressed, advertising call:' +
                (ok ? 'success' : 'failure'));
        ok = tobindornottobind(false);
        Log.print(Log.DEBUG,'Timer expired after button pressed, pairing call:' +
                (ok ? 'success' : 'failure'));
        GPIO.blink(bindled_pin, 0, 0);
    }, null);
}, null);
```

Servicios RPC adicionales

Para listar y borrar registros de la base de datos de dispositivos autorizados, dado que Mongoose-OS (al menos al momento de escribir esta nota) no provee esos servicios, debimos escribir nuestras funciones en C llamando a las funciones correspondientes del IDF (IoT Development Framework, el entorno de soporte de Espressif, el fabricante); por lo que agregamos los RPC en C.
La operatoria no es complicada, disponemos de funciones para procesar parámetros y responder usando JSON casi como si operáramos en JavaScript:

```
static void my_rpc_bond_delete_addr_handler(struct mg_rpc_request_info *ri,
                                    void *cb_arg,
                                    struct mg_rpc_frame_info *fi,
                                    struct mg_str args) {

    if ((json_scanf(args.p, args.len, ri->args_fmt, &addr_str) == 1) &&
          mgos_bt_addr_from_str(mg_mk_str(addr_str), &addr)) {

         mg_rpc_send_responsef(ri, "{rc: OK}");
    }
}
```

El código a continuación indica cómo agregamos dichas funciones a la API RPC:

```
enum mgos_app_init_result mgos_app_init(void) {
    mg_rpc_add_handler(mgos_rpc_get_global(), "Bond.List", "", my_rpc_bond_list_handler, NULL);
    mg_rpc_add_handler(mgos_rpc_get_global(), "Bond.Erase", "{addr: %Q}", my_rpc_bond_delete_addr_handler,
NULL);

    return MGOS_APP_INIT_SUCCESS;
}
```

El llamado a *Bond.List* no requiere parámetros y devuelve un objeto JSON conteniendo el listado de direcciones de dispositivos conocidos. El llamado a *Bond.Erase* requiere como parámetro un objeto JSON conteniendo el valor *addr* con la dirección a eliminar, y devuelve el resultado de la operación dentro de un objeto JSON.

Configuración

Mongoose-OS provee una estructura de configuración. Los valores por defecto se definen en un archivo *YAML* (mos.yml), en el cual además definimos las libraries que requerimos para la funcionalidad deseada. Entre éstas, si incorporamos *rpc-service-config*, dispondremos de un RPC para acceder directamente a la configuración. Cualquier valor que deseemos alterar podemos simplemente invocar la función de configuración y pasarle los parámetros correspondientes
Así, por ejemplo, de manera similar a como hicimos con los RPC de control, un script dentro de una página web (en este caso en JavaScript) puede llamar a estos procedimientos para ver la configuración y/o configurar la red WiFi:

```
log('Calling /rpc/Config.Get ...');
  $.ajax({
    url: '/rpc/Config.Get',
    success: function(data) {
      log('Result: ' + JSON.stringify(data));
      $('#ssid').val(data.wifi.sta.ssid);
      $('#pass').val(data.wifi.sta.pass);
      $('#mqtt').val(data.mqtt.server);
    },
  });

  $('#save').on('click', function() {
    log('Calling /rpc/Config.Set  ...');
    $.ajax({
      url: '/rpc/Config.Set',
      data: JSON.stringify({config: {wifi: {sta: {enable: true, ssid: $('#ssid').val(), pass: $
('#pass').val()}},mqtt: {server: $('#mqtt').val()}}}),
      type: 'POST',
      // contentType: 'application/json',
      success: function(data) {
        log('Success. Saving and rebooting ...');
        $.ajax({url: '/rpc/Config.Save', type: 'POST', data: JSON.stringify({"reboot": true})});
      },
    })
  });
```

El llamado a */rpc/Config.Get* devuelve el objeto JSON data. Al llamar a */rpc/Config.Set* le pasamos los parámetros dentro del objeto JSON config.

Entorno de desarrollo

Mongoose-OS[1]

Con este simpático nombre nos encontramos con un framework de desarrollo de aplicaciones con conectividad orientado a la IoT, que permite simplificar la conexión y el intercambio de datos tanto si se emplean los servicios de los principales proveedores como si solamente utilizamos MQTT. Si bien varios dispositivos de diversos fabricantes se encuentran soportados, lo que nos interesa es la versión para el ESP32, que como dijimos emplea el esquema de Espressif y funciona como un marco event-driven sobre la implementación de FreeRTOS de éste.
El entorno posee un intérprete de JavaScript simplificado (mJS[2]) que nos permite escribir gran parte del código de control, sea como prueba de concepto o bien porque no tiene requerimientos de tiempo de ejecución que exijan que lo hagamos en C. De ser necesario escribir funciones en C, las mismas pueden ser incorporadas y usadas desde JavaScript.

1 https://mongoose-os.com/
2 https://github.com/cesanta/mjs

La herramienta básica, *mos tool*, es un programa simple y multiplataforma que permite realizar las funciones básicas de compilado y grabado del dispositivo y otras más complejas, mediante línea de comandos. Además, es posible abrir una ventana de un navegador y a través de ésta presentar una interfaz un poco más amigable donde observar los logs del dispositivo y realizar las tareas mencionadas.
Adicionalmente, la API RPC que mencionamos en reiteradas oportunidades es accesible mediante esta herramienta, por lo que también podemos probar nuestros RPC con ella antes de largarnos a desarrollar la página web o la aplicación para smartphone.
Si bien nada impide editar los archivos de código fuente como se nos ocurra hacerlo, si aceptamos utilizar Visual Studio Code podemos instalarle una extensión para Mongoose-OS que nos provee todo lo necesario para chequear sintaxis e incluso utilizar los servicios de *mos tool* desde él.
Mongoose-OS tiene licencia Apache 2.0, podemos modificar y utilizar el código en aplicaciones comerciales siempre y cuando distribuyamos con la documentación la licencia original, reconociendo al creador del código. Para obtener soporte y remover algunas limitaciones[1] (que no afectan a lo expuesto en esta nota de aplicación), podemos obtener una licencia comercial.

Homie[2]

Homie es una convención sobre el uso de tópicos en MQTT, de forma que los dispositivos puedan anunciar sus servicios y ser automáticamente reconocidos. En realidad es un conjunto de convenciones que ha ido evolucionando a lo largo del tiempo, en este caso hemos empleado la versión 4.0.0 que es la más reciente al momento de escribir esta nota, soporta lo que requerimos, y hemos comprobado su funcionalidad con el *binding* de openHAB.
Homie tiene licencia Creative Commons Attribution 4.0, su uso no requiere acciones específicas.

Broker MQTT

Para operar con la Homie Convention se requiere de un broker MQTT con capacidad de persistencia (*persistence*), es decir, con una base de datos que almacene los últimos mensajes marcados como persistentes y los retransmita a quien se suscribe al tópico correspondiente. En nuestro caso hemos utilizado Eclipse Mosquitto[3], pero hay otras opciones. Incluso openHAB provee un broker MQTT, pero como ya teníamos Mosquitto funcionando no lo hemos probado. Cualquiera fuere el que elijamos, es fundamental comprobar que provea persistencia.
En el caso de Mosquitto, para configurar la persistencia debemos agregar

```
persistence true
```

en el archivo de configuración, si lo buscamos lo encontramos. También es posible que además debamos agregar el nombre y ubicación de la base de datos de persistencia (paths en formato ambiente GNU/Linux):

```
persistence_file mosquitto.db
persistence_location /var/lib/mosquitto/
```

pero eso depende de la distribución que utilicemos, por lo que puede que ya esté hecho por defecto. El usuario bajo el cual se ejecuta el proceso debe tener permisos de escritura a esta base de datos.
Mosquitto tiene licencia EPL (Eclipse Public License), podemos distribuirlo en el paquete de software que se instala en el servidor (que seguramente albergará además a openHAB) distribuyendo la licencia.

openHAB[4]

Este entorno de automatización del hogar se basa en un motor de reglas que opera sobre entes abstractos. Estos entes se conforman e interactúan con dispositivos reales mediante una serie de *bindings* que interpretan y traducen las comunicaciones de diversos sistemas (comerciales o no), de modo que es posible leer un interruptor inteligente de un sistema de un fabricante A y operar una luminaria inteligente de un fabricante B sin que ninguno de ellos tenga idea de la existencia del otro y ambos hablen protocolos totalmente diferentes e incompatibles, sobre transportes distintos. Lo único que necesitamos es que exista un *binding* para cada uno.

1 https://mongoose-os.com/docs/mongoose-os/userguide/licensing.md
2 https://homieiot.github.io/
3 https://mosquitto.org/
4 https://www.openhab.org/

openHab funciona en una máquina virtual Java, y nos provee un servidor web. Para observar, controlar y configurar podemos acceder a su página web mediante un navegador o utilizar la aplicación de smartphone (Android/iOS). Es común distribuirlo dentro de un paquete preconfigurado (openHABian[1]), con una versión lista para usar en Raspberry Pi.
El broker MQTT lo podemos configurar mediante la interfaz Paper UI, prestemos atención de configurar la opción de retención ('*retain*') cuando lo hagamos. Una vez que nuestro dispositivo se conecta, enviará los mensajes al broker MQTT en los tópicos especificados por la Homie Convention y openHAB lo reconocerá. Observaremos entonces que aparece en nuestra bandeja de entrada. Lo agregaremos entonces (*thing*) y tomaremos nota de los canales (*channels*) que provee, ya que los necesitaremos para configurar el *item* que veremos como un interruptor. Podríamos seguir configurando en esta interfaz, pero como haremos uso de una facilidad de "auto-expirado" para transformar un interruptor en un pulsador, requerimos por un lado que esté instalado el *binding* correspondiente (*expire*, podemos instalarlo también desde la Paper UI), y por otro configurar en *conf/items/* algún archivo cuyo nombre termine en *.items*; por ejemplo (paths en ambiente GNU/Linux) *conf/items/test.items*. En él pondremos dos largas líneas:

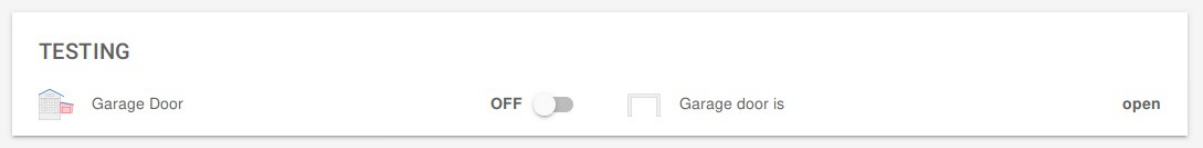
```
        Switch Garage_Door_Switch "Garage Door" <garage> ["Switchable"]
{channel="mqtt:homie300:3d8f9921:edgardo807a98:button#state",expire="1s,command=OFF"}

        String Garage_Door_State "Garage door is [%s]" <garagedoor> {channel="mqtt:homie
300:3d8f9921:edgardo807a98:door#state"}
```

El texto resaltado entre comillas es el nombre por el que openHAB Cloud conoce a este ítem y por el que los asistentes operarán sobre él luego de exportado (como se lo indicaremos a Google Assistant o a Alexa, por ejemplo). El nombre del canal (en *bastardilla*) seguramente va a ser diferente, es asignado por openHAB al agregar el dispositivo, como indicamos.
Luego, podemos agregar esto a la interfaz, configurando en algún sitemap, por ejemplo si instalamos openHAB como demo:

```
        Frame label="TESTING" {
                Switch item=Garage_Door_Switch label="Garage Door"
                Text item=Garage_Door_State
        }
```

o agregando uno nuevo, por ejemplo *conf/sitemaps/firsttry.sitemap*

```
sitemap firsttry label="Main Menu"
{
        Frame label="TESTING" {
                Switch item=Garage_Door_Switch label="Garage Door"
                Text item=Garage_Door_State
        }
}
```



Finalmente, openHAB es open source y cuenta con muchos bindings los cuales han sido desarrollados mediante ingeniería inversa ya que muchos fabricantes no gustan de informar sus protocolos propietarios, por la misma razón que han decidido usarlos. Esto significa que si bien podemos usarlo sin problemas, llegado el caso de querer desarrollar un producto personalizado para aplicaciones comerciales serias, es decir, no queremos que se vea el logo de openHAB sino que todo sea nuestro producto, no sería una buena opción tal como está[2]. La buena noticia es que parte de openHAB usa y comparte código con Eclipse SmartHome[3], es decir, que podemos migrar luego a este entorno.
Ambos entornos tienen licencia EPL (Eclipse Public License), podemos distribuirlos en el paquete de sofware del servidor (junto a Mosquitto) distribuyendo la licencia. En el caso que realicemos modificaciones con fines comerciales, debemos publicarlas o acceder a entregar el código fuente al usuario que las pida.

1 https://www.openhab.org/docs/installation/openhabian.html
2 https://community.openhab.org/t/openhab-for-business-purposes/1460/2
3 http://www.eclipse.org/smarthome/

openHAB Cloud (Internet, Google, Alexa, etc)

Este servicio nos permite transformar nuestra instalación en “una nube”, ya que con él podemos acceder de forma remota e integrarnos con proveedores y asistentes.
Para utilizarlo debemos agregarlo[1] a openHAB y configurar en su página web[2] nuestras credenciales. Hecho esto ya disponemos de acceso por Internet a nuestro sistema, sin necesidad de abrir puertos ni configurar VPNs.
El paso siguiente es configurar el acceso en y a los principales servicios y asistentes por voz que deseemos utilizar, como por ejemplo Google Assistant[3] y Amazon Alexa[4]. Configurados ambos, ya podemos utilizarlos.
Un detalle que a lo mejor no se encuentre en otro lado es que cada vez que hagamos una modificación importante en un ítem vinculado con estos asistentes (como por ejemplo cambiar el tipo), debemos actualizarlos con la nueva información para que operen correctamente. Esto es tan complicado como decir ”Alexa, discover” u ”OK Google, sync my devices” en los respectivos casos.
Lamentablemente, openHAB Cloud no parece tener servicios ni alternativas para fines comerciales.

Desarrollo de una aplicación para control por BLE

Una aplicación que controle este dispositivo debe encontrar el servicio GATT que permite utilizar la API RPC[5], observar sus características, y operar sobre ellas como requiere dicho servicio, lo cual detallamos más adelante.
A fin de realizar las primeras pruebas sobre este esquema, preferentemente una vez que tengamos ya validado el funcionamiento de los RPC mediante *mos tool* o la página web, podemos utilizar LightBlue[6]. Esta aplicación de smartphone nos permite realizar pruebas en bajo nivel, asegurándonos que BLE y RPC funcionan bien.
Para realizar una prueba de concepto, podemos utilizar AppInventor[7], que nos permite desarrollar por bloques y sin saber absolutamente nada de los detalles de un smartphone. Los resultados con BLE son variados, no muy satisfactorios con el ESP32 en la experiencia del autor.
Para desarrollar en serio, disponemos de varias opciones. En Android por ejemplo disponemos del oficial Android Studio, pero hay alternativas como por ejemplo VisualStudio y QT[8].
Una vez tengamos esto funcionando, para administrar la base de datos de dispositivos autorizados, el código es el mismo, sólo cambian los nombres de los procedimientos RPC que llamamos.
Si queremos escribir una aplicación para configurar WiFi o alguna otra cosa, el servicio GATT tiene otro identificador diferente al API RPC, dado que se trata del servicio de acceso a la configuración, *bt-service-config*[9], no el de ejecución de procedimientos.

Comunication blue’s

Los nombres de los servicios y características están indicados en las páginas de documentación de los mismos.
La operatoria de comunicaciones es la siguiente.

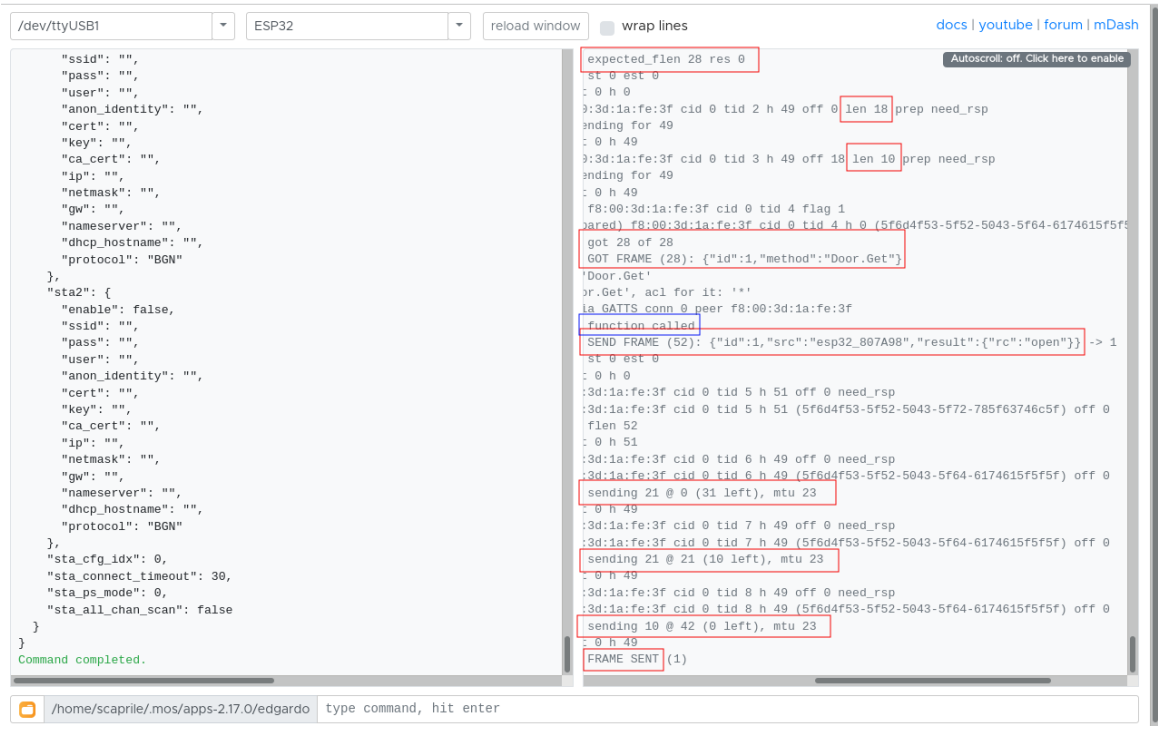
1. Descubrimos el servicio y nos conectamos.
2. Nos registramos para recibir notificaciones en la característica ***RxCtl***
3. Se escribe la longitud del comando en la característica ***TxCtl***, se trata de un número de 32-bits *network ordered* (*big endian*), es decir que “8” se escribe como cuatro bytes: `00 00 00 08` y en ese orden.
4. luego escribimos el comando en la característica ***Data***

1 https://www.openhab.org/addons/integrations/openhabcloud/
2 https://myopenhab.org/
3 https://www.openhab.org/docs/ecosystem/google-assistant/
4 https://www.openhab.org/docs/ecosystem/alexa/
5 RPC over BLE GATT Services: https://mongoose-os.com/docs/mongoose-os/api/rpc/rpc-gatts.md
6 Android: https://play.google.com/store/apps/details?id=com.punchthrough.lightblueexplorer IoS: https://apps.apple.com/us/app/lightblue/id557428110
7 http://appinventor.mit.edu/
8 https://www.qt.io/download-open-source
9 https://github.com/mongoose-os-libs/bt-service-config

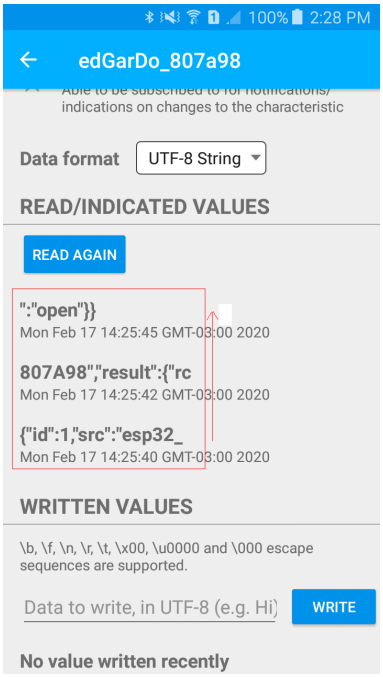
- La MTU (máxima longitud de mensaje) es cercana a los 20 bytes, por lo que en algunos casos debe ser necesario realizar varias escrituras hasta completar la totalidad del mensaje. Muchos stacks BLE resuelven esto por sí mismos.
- No se debe enviar un *null terminator*, no es un string en C. El mensaje es un objeto JSON y termina con el último caracter.

5. Se recibe una notificación de la característica ***RxCtl*** conteniendo la longitud de la respuesta a leer
   - Misma consideración para ***TxCtl*** respecto al formato: 20 bytes es `00 00 00 14` (en hexadecimal)
6. Luego se lee la característica ***Data*** tantas veces como sea necesario hasta obtener esa cantidad de bytes.
   - El dispositivo no entrega más que su MTU cada vez, y las respuestas suelen superarla, por lo que debemos realizar varias operaciones
   - Tampoco se recibe un *null terminator*, por la misma razón
7. Finalmente, se interpreta lo recibido como un objeto JSON, como hicimos en la página web de configuración

La captura de pantalla a continuación nos muestra la secuencia de un pedido RPC a *Door.Get* desde un smartphone; corresponde a la ventana de *mos tool* en un navegador. Para obtener estos datos, hemos modificado en la configuración el nivel de debugging, seteando *debug.level=3*. Esto, desde *mos tool*, corresponde a escribir "`mos config-set debug.level=3`". En rojo, los detalles de operatoria, en azul, el momento de llamado a la función:



La siguiente captura de pantalla es de LightBlue en un smartphone, donde podemos observar el fraccionamiento de la respuesta; debimos leer en tres oportunidades, representado por la flecha en rojo que hemos agregado junto con el recuadro de la zona en cuestión:



El fraccionamiento del envío afortunadamente lo resolvió la misma aplicación (el stack BLE, en realidad).

Seguridad

Dejamos aquí una serie de puntos a analizar, junto con un descargo[1].

- Dado que no disponemos de un display ni de un teclado, el único método para encripción disponible es ”*Just Works*”, lo cual deja abierta la puerta para que un avezado vecino con tiempo y recursos disponibles pueda observar... los mismos comandos que puede leer en esta nota de aplicación. No se intercambian datos personales por lo que no hay nada de qué preocuparse aquí.

- El problema real se encuentra a la hora de permitir la conexión; quien puede conectarse puede operar el dispositivo. Por esta razón, hemos limitado la posibilidad de conectarse solamente a aquellos dispositivos ya asociados (*bonded*) previamente. El proceso de asociación en sí requiere a su vez para su inicio y funcionamiento por sólo unos segundos, de acceso físico al controlador en la forma de presión de un botón.

- En conclusión, para poder operar sobre el dispositivo, se debe poseer o impersonar un dispositivo asociado, lo cual en pocas palabras podemos decir que es más trabajoso que utilizar otros métodos para ingresar. De extraviarse (de manera involuntaria o mediante la fuerza) un dispositivo asociado, disponemos de un comando para borrarlo de la base de datos.

- Sin embargo, para dificultar aún más, algunas cosas podrían modificarse. A fin de facilitar el desarrollo y prueba de la aplicación móvil hemos dejado habilitado el anuncio (*advertising)*, lo que permite que el dispositivo sea descubierto por la aplicación, al menos hasta que hayamos desarrollado la base de datos, para guardarlo y seleccionarlo. Dejamos además fija la dirección, a fin de poder ubicarlo siempre con el

1 El presente texto es una nota de aplicación y hemos tomado en consideración las buenas prácticas y nos hemos enfocado en la claridad; no es un producto comercial en sí mismo y como tal nadie puede hacer reclamos sobre prestaciones esperadas o esperables. El completo análisis de la seguridad del dispositivo para tal o cual aplicación corre por pura y exclusiva cuenta de quien desee hacer del código y el contenido de esta nota de aplicación algo más que un simple texto descriptivo de un ejemplo de una posible aplicación. Tanto el autor como Cika Electrónica se limitan a entregar esto de buena fe y “tal como es”, no siendo responsables de las consecuencias derivadas de su uso.

mismo nombre y dirección. Cualquiera de ambos puede inhabilitarse luego de realizada la asociación (*bonding*), el código que lo hace ha sido comentado (*commented out*) y es cuestión de volver a activarlo.

- Para más detalle, se aconseja consultar a un experto. De todos modos, y a título ilustrativo, un esquema como éste parece presentarse más seguro que una clave fija de 24-bits y definitivamente más que una de 12-bits, como se utiliza en muchos controles comerciales.