Guía de desarrollo para aplicaciones embebidas con Aquila

Resumen

La plataforma Aquila consiste en una plataforma de desarrollo de hardware (Altair y el USB-Serial), un protocolo de comunicación simplificado inalámbrico, herramientas de software para un fácil y rápido desarrollo, y una interfaz gráfica basada en web así como una API para interactuar con otros servicios web y dispositivos.

Las aplicaciones de la tecnología son:

- Automatización de dispositivos del hogar
- Redes de sensores
- Monitoreo y control industrial
- Otros dispositivos para el internet de las cosas

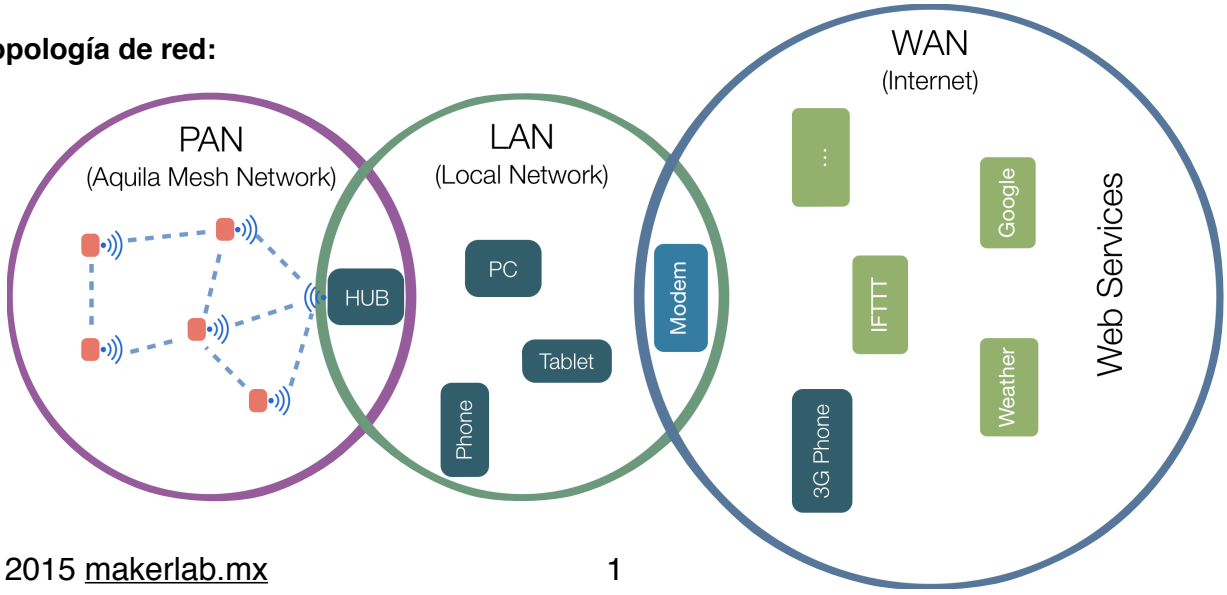
Nuestra filosofía es que uno pueda desarrollar y probar ideas para el internet de las cosas de forma fácil y rápida, usando estándares de tecnología abierta y obteniendo lo mejor del hardware, software y de las tecnologías web que están revolucionando nuestra vida.

Pensamos que combinando la flexibilidad de la web y el “know-how” de la industria del hardware es el camino que debemos seguir.

Por eso es que basamos nuestra programación embebida en las bibliotecas de Arduino, simplificando las tareas repetitivas para el desarrollador, sin dejar de ser capaz de imbuirse en los detalles del desarrollo de hardware de bajo nivel.

Ventajas de la red de Malla:
- Bajo consumo
- No sobrecarga tu router WiFi
- Los dispositivos se pueden comunicar entre ellos directamente
- Multi Salto: Si un dispositivo está fuera de rango del remitente, otros pueden ayudar repitiendo el mensaje
- Comunicación simplificada para dispositivos simples, económicos, y de bajo consumo.

Topología de red:



Stack de Software Embebido

Altair usa un microcontrolador ATmega256rfr2 como su unidad principal de proceso, está basado en la arquitectura de 8 bits AVR y tiene 256KB de memoria Flash, 32KB de SRAM, 8KB de EEPROM y tiene una frecuencia de 16Mhz.

Para más información del hardware:
- Datasheet del Altair: Altair-Datasheet-ES.pdf
- Datasheet del ATmega256rfr2: ATmega256/128/64RFR2 Datasheet

El software stack usado en Aquila es como sigue:

- Compilador: avr-gcc C y C++ y el entorno GNU (linker, make, gdb, etc.)
- Programador: avrdude - provee programación vía la Interfaz USB-Serial
- Bibliotecas base: avr-libc (Documentación: http://www.nongnu.org/avr-libc/user-manual/index.html)
- Bibliotecas de Arduino para Altair (Documentación: http://arduino.cc/en/Reference/HomePage)
- Bibliotecas de Comunicación

Stack de Comunicación Embebida

Altair, el corazón de los dispositivos embebidos Aquila, tiene un transmisor inalámbrico integrado IEEE 802.15.4, y una dirección de hardware única IEEE EUI-64. Teniendo esto como base, tenemos un completo stack de comunicaciones embebidas optimizado para sensores en redes de malla de bajo consumo, dispositivos de automatización para el hogar, automatización industrial, seguridad, monitoreo, etc.

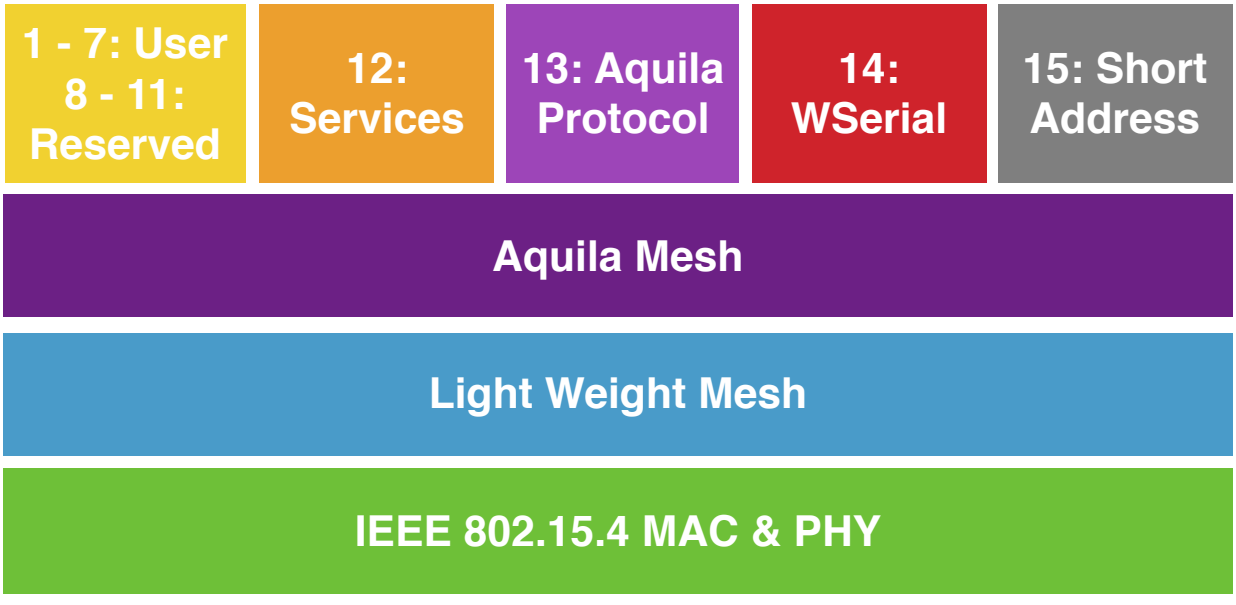
Cada dispositivo tiene una dirección corta de 16-bits, que puede ser asignada manualmente en el código, o auto asignada basada en su dirección de 64-bits de hardware única EUI-64.

También, puede haber varias sub redes (PANs), un dispositivo recibe un mensaje sólo cuando está configurado para la PAN correcta y cuando el mensaje tiene su dirección como dirección de destino, o la dirección de difusión o “broadcast” (0xFFFF).

El stack consiste en la capa física (PHY) y MAC 802.15.4, una capa de red superior basada en LWM (por sus siglas en inglés; Light Weight Mesh), que provee seguridad, enrutamiento de malla con multi-saltos, direccionamiento y endpoints. Sobre eso tenemos Aquila Mesh, que provee la dirección de hardware única EUI-64, gestión de seguridad, una dirección corta auto asignada de 16-bits y prevención de colisiones. Aquila Mesh tiene 16 endpoints donde puedes implementar protocolos de capa aplicación (similar a los puertos en el protocolo TCP/IP). Los Endpoints 1 al 7 son para que el usuario los implemente, y del 8 al 15 están reservados para los Protocolos Aquila.

- Para mayor información y documentación del estándar IEEE 802.15.4, revisa 802.15.4-2011.pdf
- Para mayor información y documentación de LWM de Atmel, revisa Lightweight Mesh Developer Guide, Application Note AVR2130

Los protocolos actuales de Aquila son:
Endpoint 15: Detector de colisiones de dirección corta (parte de la Malla Aquila)
Endpoint 14: WSerial (Sockets Seriales Simples Inalámbricos)
Endpoint 13: Protocolo Aquila (Acciones, Eventos e Interacciones)
Endpoint 12: Servicios
Endpoint 0, 8-11: Reservado
Endpoint 1-7: Para aplicaciones específicas del usuario



Protocolos de Aquila

Aquila Mesh

Aquila Mesh es una pequeña capa sobre LWM que provee acceso a la dirección única de hardware IEEE EUI-64, la dirección corta auto asignada de 16-bits, gestionamiento de encriptado de contraseñas y una API simplificada sobre LWM.

Teniendo direcciones de 16-bit significa que hay 65535 direcciones de dispositivo posibles. Están organizadas de esta manera:

- La dirección 0x0000 está reservada para dispositivos sin inicializar.
- Las direcciones desde 0x0001 hasta 0x00FE sólo son para configuración manual.
- La dirección 0x00FF es para el bridge.
- Las direcciones de la 0x0100 a la 0xFFFD son para el asignado automático.
- La dirección 0xFFFE está reservada en el estándar 802.15.4.
- La dirección 0xFFFF es la dirección de BROADCAST.

Esta librería tiene la intensión de ser usada en conjunto con las otras bibliotecas de Aquila, pero también puede ser usada para directa implementación de protocolos personalizados sobre los endpoints 0 al 7. Un ejemplo de cómo usar esto, favor de ver el ejemplo Mesh_Ping, puedes acceder a él desde el IDE de Arduino en Archivo > Ejemplos > Mesh > Mesh_Ping.

API:

Headers requeridos:

```
#include <Wire.h>
#include <Mesh.h>
```

Objectos y tipos:

- Mesh: el objeto de esta librería.
- TxPacket: el paquete que será enviado.
- RxPacket: el paquete que será recibido.

Funciones:

begin(uint16_t addr)

addr (opcional): Selecciona manualmente la dirección corta de 16-bits.

Descripción:
Inicia la transmisión y el stack de comunicación. Si no hay dirección especificada, automáticamente se asignará a como se explica aquí: El dispositivo deriva de la dirección corta del CRC de la dirección de hardware, y luego hace un escaneo de colisión, le suma el último byte del a dirección de hardware + 1, hasta que no haya colisión.

Ejemplo:
```
Mesh.begin();
```

loop()

Descripción:
Atiende las solicitudes de radio y las tareas de comunicación. Necesita ser llamada tantas veces como sea posible, usualmente está dentro de la función loop(). Debes evitar funciones de bloqueo largas, como delay(), dentro de tu loop, mejor utiliza métodos alternativos.

Ejemplo:
```
Mesh.loop();
```

setAddr(uint16_t addr)

addr: dirección corta de 16-bits manualmente seleccionada. Ten en cuenta que: Las direcciones manuales válidas son 1 (0x0001) a 254 (0x00FE), el resto está reservado para asignación automática y casos especiales.

Descripción:
Manualmente asigna una dirección de 16-bits.

Ejemplo:
```
Mesh.setAddr(0x00A1);
```

setPanId(uint16_t panId)

panId: 16-bit PAN

Descripción:
Selecciona la PAN del dispositivo, por defecto esta es 0xCA5A.

Ejempo:
```
Mesh.setPanId(0xCAFE);
```

setChannel(uint8_t channel)

channel: canal de radio frecuencia.

Descripción:
Selecciona el canal de radio frecuencia. Por defecto éste es 26.
Los canales válidos son del 11 al 26.

Canales y sus respectivas frecuencias:

Canal	Frecuancia
11	2.405GHz
12	2.410GHz
13	2.415Ghz
14	2.420Ghz
15	2.425Ghz
16	2.430GHz
17	2.435GHz
18	2.440Ghz
19	2.445GHz
20	2.450GHz
21	2.455GHz
22	2.460GHz
23	2.465GHz
24	2.470GHz
25	2.475GHz
26	2.480GHz

Ejemplo:
```
Mesh.setChannel(25);
```

openEndpoint(uint8_t id, bool (*handler)(RxPacket *ind))

id: número del endpoint. Puede ser del 1 al 15, tenga en cuenta que los endpoint del 8 al 15 están reservados para los protocolos de Aquila.
handler: función que va a manejar los paquetes recibidos en el endpoint. Esta función debe tener la siguiente estructura:

```
bool yourFunction(RxPacket *ind);
```

Descripción:
Asigna una función de manejo al endpoint. Cuando el paquete se recibe en el endpoint, la función será llama con el paquete pasado como parámetro.

Nota: RxPacket es lo mismo que NWK_DataInd_t en LWM.

Miembros de RxPacket:
```
uint16_t     srcAddr;
uint16_t     dstAddr;
uint8_t      srcEndpoint;
uint8_t      dstEndpoint;
uint8_t      options;
     // Opciones posibles:
     NWK_IND_OPT_ACK_REQUESTED
```

```
        NWK_IND_OPT_SECURED
        NWK_IND_OPT_BROADCAST
        NWK_IND_OPT_LOCAL
        NWK_IND_OPT_BROADCAST_PAN_ID
        NWK_OPT_LINK_LOCAL
        NWK_OPT_MULTICAST
    uint8_t      *data;
    uint8_t      size;
    uint8_t      lqi;
    int8_t       rssi;
```

uint16_t getShortAddr()

Descripción:
Devuelve la dirección corta del dispositivo.

getEUIAddr(uint8_t* address)

address: un apuntador al arreglo de 8 bytes donde la dirección será copiada.

Descripción:
Obtiene la dirección de hardware EUI-64 del dispositivo.

setSecurityKey(uint8_t *key)

Descripción:
Configura una llave de seguridad de 128-bits (Arreglo de 16-bytes).

setSecurityEnabled(bool enabled)

Descripción:
Configura la seguridad. Por defecto está desactivada.

getSecurityEnabled()

Descripción:
Reporta si la seguridad está activa (bool).

announce(uint16_t dest)

Descripción:
Presenta el dispositivo en la red. Permite que el dispositivo sea detectado automáticamente por el hub.

sendPacket(TxPacket *packet)

packet: un apuntador a TxPacket.

Descripción:
Envía un paquete, lo mismo que NWK_DataReq en LWM.

Nota: *TxPacket* es lo mismo que *NWK_DataReq_t en LWM.*

Miembros de TxPacket:

```
uint16_t dstAddr
uint8_t dstEndpoint
uint8_t srcEndpoint
uint8_t options
      // opciones posibles, las puedes combinar
      // con el operador o (|):
      NWK_OPT_ACK_REQUEST
      NWK_OPT_ENABLE_SECURITY
      NWK_BROADCAST_PAN_ID
      NWK_OPT_LINK_LOCAL
      NWK_OPT_MULTICAST
uint8_t *data
uint8_t size
// apuntador a la función que será llamada apenas se confirme
// (éxito o error):
void (*confirm)(TxPacket *packet)

// parámetros de confirmación
uint8_t status
      // estados posibles:
      NWK_SUCCESS_STATUS
      NWK_ERROR_STATUS
      NWK_OUT_OF_MEMORY_STATUS
      NWK_NO_ACK_STATUS
      NWK_NO_ROUTE_STATUS
      NWK_PHY_CHANNEL_ACCESS_FAILURE_STATUS
      NWK_PHY_NO_ACK_STATUS
uint8_t control
```

busy()

Descripción:
Reporta si la antena está ocupada (bool).

sleep()

Descripción:
Apaga la radio para ahorrar energía. Debes revisar primero que la radio no está ocupada antes de apagarla.

Ejemplo:
```
while(Mesh.busy()) Mesh.loop();
Mesh.sleep();
```

wakeup()

Descripción:
Enciende la antena.

asleep()

Descripción:
Reporta si la antena está apagada o no (bool).

WSerial

Implementación simple de sockets seriales sobre la Malla Aquila.

Básicamente tiene la misma API que la librería Serial de Arduino, pero permite una transferencia de datos inalámbrica simple entre dos o más Altairs.

Para comunicar dos dispositivos, ambos deben de estar configurados con la dirección de destino del otro (con `begin` o `setDest`).

Si quieres aceptar mensajes de cualquier dispositivo debes usar `WSerial.setAllowFromAny(true)`, y para enviar mensajes a todos los dispositivos, debes de configurar las direcciones de destino como BROADCAST.

Puedes encontrar un ejemplo en el IDE de Arduino, Archivo > Ejemplos > WSerial.

API:

Headers requeridos:

```
#include <Wire.h>
#include <Mesh.h>
#include <WSerial.h>
```

Objetos:

- WSerial: Es un objeto de la librería.

Funciones:

begin(uint16_t destAddr)

destAddr: dirección de destino de 16 bits.

Descripción:
Inicia la librería y configura la direcciones a las cuales les estaremos enviando datos.

Ejemplo:
```
WSerial.begin(0x0023);
```

end()

Descripción:
Cierra la conexión.

Ejemplo:
```
WSerial.end();
```

setDest(uint16_t destAddr)

destAddr: dirección de destino de 16 bits.

Descripción:
Configura la direcciones a las cuales les estaremos enviando datos.

Ejemplo:
```
WSerial.setDest(0x0024);
```

setAllowFromAny(bool allow)

allow: si permitimos mensajes de cualquier dispositivo, o no.

Descripción:
Configura si queremos aceptar mensajes de cualquiera, o sólo del mismo dispositivo al que enviamos. Por defecto, es false.

Tome en cuenta que no tenemos manera de saber de qué dispositivo vino el mensaje, sólo la información directa es lo que recibimos. Esto fue diseñado así por simpleza.

Ejemplo:
```
WSerial.setAllowFromAny(true);
```

loop()

Descripción:
Atiende las solicitudes y otras tareas. Debe ser llamado tantas veces como sea posible en el main loop.

Ejemplo:
```
WSerial.loop();
```

Otras funciones

Todas las otras funciones son heredadas de Arduino's Stream class, y trabajan de la misma forma que la librería Standard de Arduino. Su documentación la puedes encontrar aquí.

Protocolo Aquila

Protocolo basado en eventos, acciones e interacciones.

La intención del protocolo Aquila es Automatizar aplicaciones del Hogar, o cualquier otro entono en donde los dispositivo se deberían hablar entre ellos automáticamente. Permite a un dispositivo tener Acciones (que son las cosas que podemos hacer), y Eventos (que son las cosas que le pueden ocurrir a un dispositivo), y hacer conexiones entre ellos con interacciones (cuando tal Evento ocurre, haz tal acción).

En resumen:

- Acciones: Lo que un dispositivo puede HACER.
- Eventos: Lo que puede OCURRIRLE a un dispositivo.
- Interacciones: Haz tal Acción cuando tal Evento ocurra.

La siguiente API te permite definir Acciones en un dispositivo como funciones, y eventos como objetos que pueden ser “emitidos” en cualquier parte de tu código.

Las interacciones están configuradas dentro de la memoria EEPROM del dispositivo que hará la acción, esta configuración está hecha desde la interfaz del Servidor Aquila, así que necesitas un hub para configurarlas, de cualquier forma, una vez configuradas, éstas continuarán trabajando incluso sin un hub.

API:

Headers requeridos:

```
#include <Wire.h>
#include <Mesh.h>
#include <AquilaProtocol.h>
```

Objetos y tipos:

- Event: usado para definir el identificador del evento
- Aquila: Es un objeto de la librería.

Funciones:

begin()

Description:
Inicia la librería.

Ejemplo:
`Aquila.begin();`

loop()

Descripción:
Atiende las solicitudes entrantes, debería ser llamado tantas veces como sea posible en la función loop.

Ejemplo:
```
Aquila.loop();
```

setClass(char *nid)

Descripción:
Configura la clase del dispositivo.

Por convención propuesta, la clase del dispositivo debe tendrá la forma de una “URL invertida” como se muestra: Si la URL de tu compañía es “example.com” y tu dispositivo se llama “Example Device”, la clase sería:

```
"com.example.exampledevice"
```

Ejemplo:
```
Aquila.setClass("com.mycompany.exampledevice");
```

setName(char *nName)

Descripción:
Configura el nombre por defecto del dispositivo, el que verás en la UI. Por ejemplo: “Example Device”

Ejemplo:
```
Aquila.setName("Example Device");
```

addAction(char description[], bool (*function)(uint8_t param, bool gotParam))

Descripción:
Añade una acción con una descripción y una función que será llamada cuando la acción se ejecute.

Ejemplo:
```
bool turnOn(uint8_t param, bool gotParam)
{
    // Do something...
}
// ...
Aquila.addAction("Turn On", turnOn);
```

Event addEvent(char description[])

Descripción:
Añade un evento con una descripción, y regresa la id del evento que será usada al emitirlo.

Ejemplo:
```
Event buttonPressed;
buttonPressed = Aquila.addEvent("Button Pressed");
```

emit(uint8_t event, uint8_t param=0, bool hasParam=false)

Descripción:
Emite un evento. Debe ser llamado cuando el evento ocurre (un botón es presionado, ha pasado un tiempo, etc…).

Ejemplo:
```
// Cuando se presiona el botón
Aquila.emit(buttonPressed);
```

emit(uint16_t dest, uint8_t event, uint8_t param=0, bool hasParam=false)

Descripción:
Emite un evento únicamente a un dispositivo de destino específico. Debe ser llamado cuando el evento ocurre (un botón es presionado, ha pasado un tiempo, etc…).

Ejemplo:
```
// Cuando se presiona el botón
Aquila.emit(0x0B9A, buttonPressed);
```

on(char eventName[], bool (*function)(uint8_t param, bool gotParam))

Descripción:
Subscribe estáticamente la función a un evento con su nombre. En el ejemplo, la función “doSomething” será ejecutada cuando cualquier dispositivo emita un evento llamado “Button Pressed” (Respetar mayúsculas).

Ejemplo:
```
Aquila.on("Button Pressed", doSomething);
```

on(char eventName[], uint8_t EUIAddress[], bool (*function)(uint8_t param, bool gotParam))

Descripción:
Subscribe estáticamente la función a un evento emitido por el dispositivo con la dirección de hardware “EUIAddress”. En el ejemplo, la función “doSomething” será ejecutada solamente cuando el dispositivo con la dirección “longAddr” emite el evento llamado “Button Pressed” (Respetar mayúsculas).

Ejemplo:
```
uint8_t longAddr[] = {00, 12, 23, 42, 00, 00, 65, 46};
Aquila.on("Button Pressed”, longAddr, doSomething);
```

Funciones Avanzadas:

requestAction(uint16_t address, uint8_t action, uint8_t param=0, bool hasParam=false)

Descripción:

Solicita la ejecución de una acción (uint8_t) a otro dispositivo.

uint8_t action es la id. de la acción. Las id están asignadas en orden cuando llamas addAction, comenzando de 0.

doAction(uint8_t action, uint8_t param, bool gotParam)

Descripción:
Ejecuta una acción local, con el parámetro dado si gotParam == true.

setAction(n, description, function)

Descripción:
Configura una acción con la descripción dada y una función con el slot o id “n”.

CUIDADO: Úsalo con precaución pues podría interferir con addAction.

setEvent(n, description)

Descripción:
Configura un evento con la descripción dada al slot o id “n”.

CUIDADO: Úsalo con precaución pues podría interferir con addEvent.

Servicios

Servicios estilo REST para Altair.

Inspirado en los servicios HTTP REST, esta librería permite realizar implementaciones tipo REST simplificadas sobre una red de malla 802.15.4.

Importantes diferencias con los servicios HTTP REST:

1. Debido a las limitaciones del tamaño de paquete en el protocolo 802.15.4, el nombre del servicio + la información debe ser menor a 101 bytes (AQUILAMESH_MAXPAYLOAD - 4).
2. Sólo hay un nivel para la ruta de servicio. (e.g. Puedes tener servicio como “temperatura” y “estado”, pero no “estado/12” o “temperatura/algo”).
3. Las solicitudes sólo tienen un nombre de servicio, un método e información (cuerpo), no hay cabecera.

Métodos soportados:
- GET
- PUT
- POST
- DELETE

Estados de respuesta soportados:
- R200 - OK
- R404 - Servicio no encontrado
- R405 - Método no permitido
- R408 - Timeout
- R500 - Error de servicio

Formatos de información soportados:

La información puede ser lo que sea que quieras mientras esté formado por bytes.
Como sugerencia, puedes usar cadenas codificadas con JSON con la ayuda de la librería incluida ArduinoJson, puedes ver cómo hacerlo en estos ejemplos: En el IDE de Arduino, Archivo > Ejemplos > *AquilaServices*.

Razones:

Esta librería complementa el Protocolo Aquila en la siguiente forma. Ahora tenemos:

Acciones: Las cosas que el dispositivo puede HACER.
Eventos: Las cosas que le pueden OCURRIR al dispositivo.
Servicios: Cosas que le puedes PEDIR al dispositivo y obtener una respuesta.
WSerial: Logging y depuración.

API:

Headers requeridos:

```
#include <Wire.h>
#include <Mesh.h>
#include <AquilaServices.h>
```

Objetos y tipos:

- Services: El objeto de esta librería.
- ServicePacket: Lo que enviaremos.

Funciones:

begin()

Description:
Inicia la librería.

Ejemplo:
```
Services.begin();
```

loop()

Descripción:
Atiende las solicitudes entrantes, debe ser llamado tantas veces como sea posible en la función loop.

Ejemplo:
```
Services.loop();
```

variable(char name[], int *var)

Descripción:
Expone una variable de tipo entero como servicio. Cuando se solicita el servicio desde el Hub u otro dispositivo, se responderá con el valor actual de la variable en formato JSON.
Para mayor información sobre cómo hacer peticiones desde el Hub, ver: http://docs.aquila2.apiary.io/

Respuesta de ejemplo: {"val":123}

Ejemplo:
```
// Definición de variable global:
int myVar = 0;
// Dentro de la función setup:
// &myVar pasa el apuntador de la variable a la función
Services.variable("myVar", &myVar);
```

variable(char name[], float *var)

Descripción:
Expone una variable de tipo flotante como servicio. Cuando se solicita el servicio desde el Hub u otro dispositivo, se responderá con el valor actual de la variable en formato JSON.
Para mayor información sobre cómo hacer peticiones desde el Hub, ver: http://docs.aquila2.apiary.io/

Respuesta de ejemplo: {"val":123.234}

Ejemplo:

```
// Definición de variable global:
float myVar = 0.0;
// Dentro de la función setup:
// &myVar pasa el apuntador de la variable a la función
Services.variable("myVar", &myVar);
```

float function(char name[], float (*func)(uint8_t method, char *data, uint8_t dataSize))

Descripción:
Expone una función como servicio. Cuando se solicita el servicio desde el Hub u otro dispositivo, la función será ejecutada y el valor que ésta retorne será enviado como respuesta en formato JSON.
Para mayor información sobre cómo hacer peticiones desde el Hub, ver: http://docs.aquila2.apiary.io/

Respuesta de ejemplo: {“val":123.456}

Ejemplo:
```
// Definición de la función:
float myFunction(uint8_t method, char *data, uint8_t dataSize)
{
      // Hacer algo
      // Métodos posibles: GET, POST, PUT and DELETE
      // Cualquier dato enviado en el Body desde la API del hub
      // será pasado a la función como un arreglo llamado “data”
      // de tamaño “dataSize”

      // Devuelve lo que quieras enviar como respuesta:
      return 123.456;
}
// Dentro de la función setup:
Services.function("myFunc", myFunction);
```

Funciones Avanzadas:

add(char *name, bool (*function)(uint16_t reqAddr, uint8_t method, char *data, uint8_t dataSize))

Descripción:
Añade un servicio. Un servicio tiene un nombre y una función que será ejecutada cuando se solicite, transfiriendo la data de la solicitud.

La función debe tener la siguiente estructura:

```
bool myServiceFunction(uint16_t reqAddr, uint8_t method, char *data, uint8_t
dataSize);
```

Ejemplo:
```
Services.add(“myservice", myServiceFunction);
```

request(destAddr, method, *name, (*callbackFunction), *data = NULL, dataSize = 0)

Descripción:
Hace una solicitud a otro dispositivo.

Callback debe tener la siguiente estructura:
```
void myCallback(uint16_t srcAddr, uint8_t status, char *data, uint8_t dataSize)
```

Métodos posibles:
GET
PUT
POST
DELETE

Ejemplo:
```
Services.request(0x001D, GET, “myservice”, myCallback);
```

void response(destAddr, status, *data = NULL, dataSize = 0)

Descripción:
Contesta una solicitud. Esta función debe ser llamada dentro de una función de un servicio, cuando se esté contestando a una solicitud.

Estados posibles:
R200 - OK.
R404 - Servicio no encontrado.
R405 - Método no permitido.
R408 - Timeout.
R500 - Error de servicio

Ejemplo:
```
Services.response(0x001C, R200);
```

Referencias y links sugeridos

- Datasheet de Altair - https://github.com/makerlabmx/altair-hardware/raw/master/Altair-Datasheet-ES.pdf
- Datasheet de ATmega256rfr2 - www.atmel.com/Images/Atmel-8393-MCU_Wireless-ATmega256RFR2-ATmega128RFR2-ATmega64RFR2_Datasheet.pdf
- Documentación de avr-libc - http://www.nongnu.org/avr-libc/user-manual/index.html)
- Documentación de las bibliotecas de Arduino - http://arduino.cc/en/Reference/HomePage
- Estándar IEEE 802.15.4 - standards.ieee.org/getieee802/download/802.15.4-2011.pdf
- Guía de desarrollo para LWM de Atmel - www.atmel.com/Images/Atmel-42028-Lightweight-Mesh-Developer-Guide_Application-Note_AVR2130.pdf

¿Necesitas ayuda?

- Página principal de Aquila: http://www.aquila.io/es
- Foro de la comunidad Aquila: http://community.aquila.io/
- Contáctanos: info@makerlab.mx