Dpto. de Ingeniería de Sistemas y Automática
Universitat Politècnica de València
Master Universitario en Automática e Informática Industrial
Curso 2018/19

Trabajo Final de Master

Desarrollo de librerías de abstracción y comunicaciones para desarrollos ágiles basados en microcontroladores Renesas Synergy

Autor:
Ismael Casabán Planells

Tutor:
Juan Francisco Blanes Noguera

Desarrollo de librerías de abstracción y comunicaciones para desarrollos ágiles basados en microcontroladores Renesas Synergy

Ismael Casabán Planells

ABSTRACT

In a world where more and more devices are connected to the Internet, with different purposes such us collecting sensor data, monitoring or just control other devices; It is necessary to have the tools to achieve the desired goals in an optimum time.

On this project these are going to be developed beginning from a description of the Renesas Synergy microcontrollers and finishing with the sending and reception of the generated data on a server.

To achieve this objective, the communication libraries must be developed as well as all the different layers to make a modular, robust and scalable system. With these tools, a new project will have the possibility to control the Hardware directly in conjunction with some communication technologies using some of the most important IoT protocols.

RESUMEN

En un mundo en dónde cada vez más dispositivos se conectan a Internet, poseyendo diferentes finalidades como sensorización, monitorización o control, es necesario poseer las herramientas necesarias para lograr los objetivos en un tiempo óptimo.

En este proyecto se van a desarrollar las mismas, partiendo de la descripción de microcontroladores de Renesas Synergy y terminando con la transmisión y recepción de datos en un servidor.

Para lograr dicho objetivo, se implementarán las librerías de comunicación y abstracción necesarias para lograr un conjunto de herramientas con las que se abarque desde el manejo del Hardware, hasta la comunicación con distintas tecnologías junto con algunos de los protocolos más utilizados en el mundo del IoT.

Índice

Índice imágenes

Abreviaciones

6LowPAN	IPv6 over Low -Power Wireless Personal Area Networks
HAL	Hardware abstraction layer
IEFT	Internet Engineering Task Force
MTU	Maximum Transmission Unit
MCU	Microcontroller Unit
NAT	Network Address Translation
PCB	Printed Circuit Board
RFC	Request for Comments
RPC	Remote Procedure Call
SCI	Serial Communication Interface
SO	Sistema Operativo

Definiciones

Firmware	También conocido como soporte lógico inalterable, es un programa informático que establece el funcionamiento de más bajo nivel, la electrónica [1].
Open Source	Termino legal para describir que un producto conlleva el permiso para visualizar y modificar su código fuente, diseño o contenido.
Byte stuffing	Proceso o algoritmo por el cual se codifican bytes para dotar a un paquete de información una codificación, de tal forma, que una serie de bytes no llegan a aparecer nunca. Se utiliza con frecuencia en protocolos de comunicación basados en un byte de inicio y de fin conocidos (por ejemplo, can).
Gateway	O puerta de enlace, corresponde a un dispositivo que permite la traducción de una tecnología o protocolo a otro distinto.
Payload	Carga útil que contiene un paquete de datos transmitido.
Sensor HALL	Sensor que utiliza el efecto Hall para la medición de campos magnéticos o corrientes o para la determinación de la posición en la que está [2].
Unstuffing	Proceso contrario al Byte stuffing descrito arriba.

1. Introducción

Como bien sabrá el lector, ante una empresa dedicada al desarrollo de soluciones específicas demandadas por clientes de diversos sectores, puede darse el caso de llegar a un estado en el que para cada nuevo producto comercial se necesite un Firmware concreto. Ante dicha tesitura, se presenta un escenario en donde el trabajador tenga que dominar distintos entornos de desarrollo, así como, diferentes microcontroladores haciendo que el tiempo de desarrollo implique un gasto de tiempo extra en el aprendizaje de las librerías específicas para el proyecto.

Para solucionarlo, se pretende desarrollar un conjunto de librerías generales independientes del microcontrolador empleado, de forma que se organicen jerárquicamente y por capas.

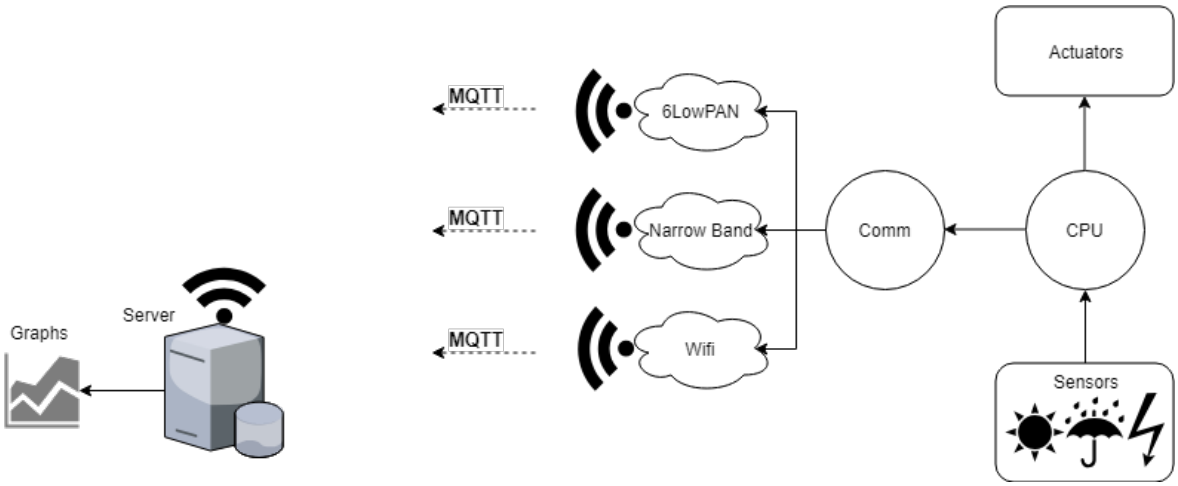


Imagen 1: Visión general sistema planteado

Para llevar a cabo dicho objetivo se plantea el sistema que se muestra en la Imagen 1. En ella se puede observar un sistema completo desde la generación de los datos hasta la utilización de los mismos en el mismo Hardware o en un servidor remoto. También se puede observar la naturaleza modular del sistema, en dónde, el paso de información o la actuación sobre algún elemento, se realiza de forma ordenada y jerárquica.

Por necesidades de la empresa, se dará mucho más hincapié en la parte de comunicaciones ya que, la misma, corresponde con la mayoría de cambios entre proyectos.

2. Objetivos del proyecto

Para ir entrando en materia y seguir un orden lógico a la hora de desarrollar el presente documento se especifica un objetivo principal tal y como el título muestra: “El desarrollo de librerías de abstracción y comunicaciones para desarrollos basados en la familia de microcontroladores Renesas Synergy”.

Para lograrlo, y dotar del proyecto de mayor innovación, se estudiará la utilización del nuevo microcontrolador S128 de Renesas, así como ThreadX, para emplear un RTOS en nuestros diseños.

Llegados a este punto, se explicará la solución propuesta para utilizar las diferentes tecnologías de comunicación de forma abstracta y separada por las diferentes capas.

Dado que cada tecnología de comunicación (6LowPAN, Wifi, Ethernet y Narrow Band / GSM) posee unos métodos/protocolos diferentes de interacción, se explicará la estrategia con que se interactúa con cada una de ellas sin perder la escalabilidad a la hora de ampliar el diseño.

Por último, y por necesidades del proyecto, se deberá de realizar un amplio estudio sobre los diferentes drivers/HAL que necesiten las distintas capas/herramientas que proporcione el proyecto.

Como ejercicio extra, y para mostrar un ejemplo de utilización del mismo, se detallará un ejemplo de aplicación en donde se establece una comunicación entre tareas y la utilización de algunos sensores que proporciona el kit de evaluación DK-S128.

Como protocolo de comunicación se podría haber utilizado UDP o TCP, pero se ha visto interesante utilizar una librería que implemente MQTT o REST debido a su popularidad sin depender de qué tecnología se emplee para transmitir la información. Dicha elección permitirá utilizar otras tecnologías sin necesidad de un rediseño.

El ejemplo básico de sistema mostrará un diseño Hardware (utilizando el kit de evaluación anteriormente mencionado) comunicándose con un servidor MQTT ajeno y, al mismo tiempo, que este sea capaz de mostrar el estado actual del mismo, pudiendo, por otra parte, interactuar sobre los actuadores que posee.

De esta manera se tendrá un sistema formado por:

- La captación de unidades de medida.
- Envío por diferentes tecnologías (siendo compatibles entre ellas)
- Utilización de MQTT, un protocolo Open Source.
- Garantizando un tiempo de generación de datos con RTC o tareas periódicas.
- Recepción de la información mediante un servidor.
- Generación de gráficas y acciones como consecuencia de los datos recibidos.

3. Renesas Synergy

La familia de microcontroladores Renesas Synergy abarca cuatro series distintas y está diseñada para aplicaciones que van desde dispositivos móviles conectados o aplicaciones IoT hasta controladores de sistemas integrados de alto rendimiento [3].

En la web se muestra que los diferentes componentes de la familia ofrecen una gama de rendimiento, características y compatibilidad de pines dentro de la misma, ofreciendo las MCU de Synergy una escalabilidad, consumo de energía, reutilización de códigos y necesidades de rendimiento que se adaptan a cualquier tipo de productos comerciales.

El mayor hincapié recae en la escalabilidad vertical (debido a la mejora del hardware) y la compatibilidad entre las distintas series a nivel de firmware ya que se permite migrar de una MCU a otra, consiguiendo reducir el tiempo de diseño y aumentar la eficiencia de fabricación. Dicho con menos palabras, una ganancia directa en tiempo y dinero.

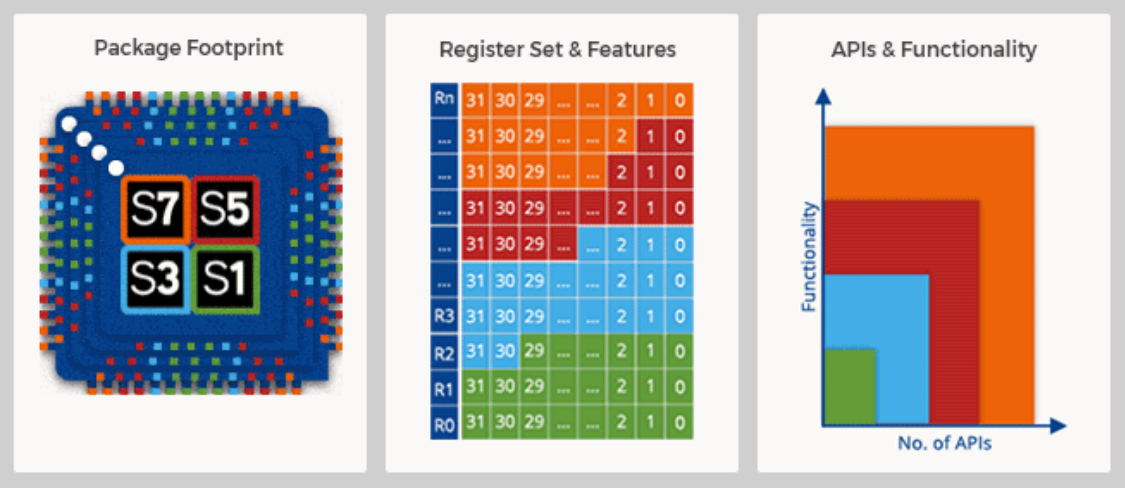


Imagen 2: Familia Renesas Synergy [3].

La utilización de un microcontrolador u otro, vendrá dado por el tipo de aplicaciones en dónde se quiera utilizar:

Serie 1	Diseños de bajo consumo con capacidades analógicas, conectividad con una seguridad robusta para aplicaciones que requieren una larga duración de la batería y tiempos de espera.
Serie 3	Aplicaciones sensibles a la energía que necesitan, con una conexión segura, control analógico y tiempos más precisos. Utilización de LCD de segmentos y control táctil capacitivo.
Serie 5	Implementación de aplicaciones de control exigentes con una combinación de seguridad, amplia memoria de almacenaje en serie junto una LCD gráfica a color con utilización de imágenes en 2D.
Serie 7	Manejar múltiples canales de conectividad a alta velocidad simultáneamente, pantallas a todo color, operaciones de control precisas y detección de señales analógicas con alta precisión.

Tabla 1: Características series Renesas Synergy

Como se puede observar, las distintas series se distribuyen para abarcar todas las aplicaciones que se puedan hacer de tal forma que se pueda elegir un microcontrolador específico sin necesidad de introducir recursos innecesarios o aumentar el precio del producto sin motivo.

Dado que para la elección del microcontrolador designado se requiere unos mínimos recursos (UART, PWM, ADC, RTC, I2C, SPI y USB) y no una excesiva cantidad de Flash y RAM, se opta por empezar a utilizar la Serie 1, en concreto el S128 por ser el más reciente y poseer un menor consumo.

Aunque las aplicaciones no requieran mucha memoria RAM, la utilización de diferentes librerías como el ThreadX o la de USB pueden llevan al límite esta misma. En adelante se abordará dicho problema.

Como resultado del estudio de los diferentes microcontroladores, se ha elaborado una tabla para poder analizar rápidamente qué características u opciones proporcionan cada una de las series de Renesas Synergy en el Anexo I.

Como punto también importante a la hora de elegir entre una opción u otra de microcontrolador, no sólo se encuentran los periféricos que pueden proporcionar, sino también el precio de los mismos, ya que puede suponer un aumento no justificable del producto final. A continuación se muestran dos gráficas comparativas del precio: las series S1 y S3 que seleccionarán el formato de chip CFM, mientras que en las series S5 y S7 será el CLK. Los precios se pueden consultar en el Anexo II.

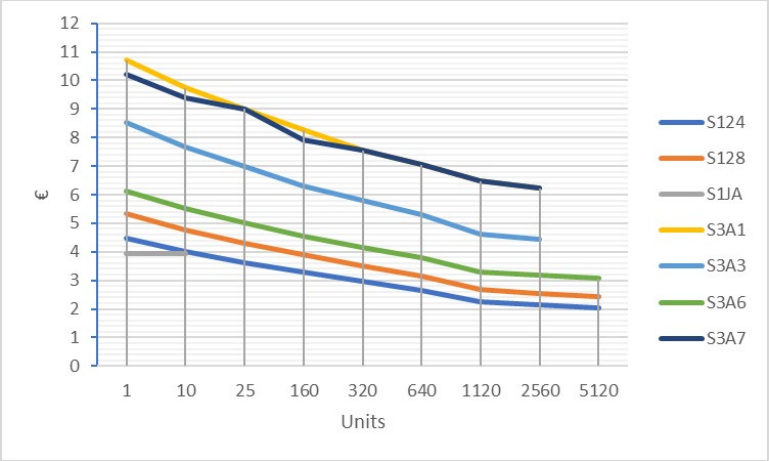


Imagen 3: Comparativa precios S1 y S3

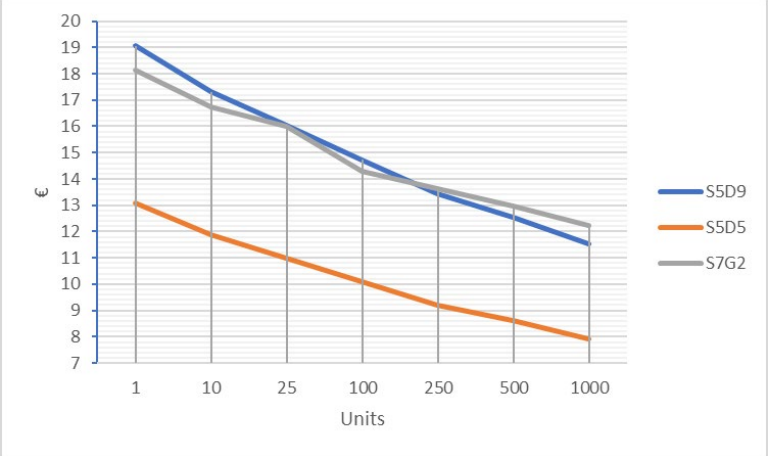


Imagen 4: Comparativa precios S5 yS7

4. ThreadX

ThreadX responde al nombre de un sistema operativo en tiempo real (RTOS) implementado especialmente para el desarrollo de aplicaciones IoT y destinadas a hardware embebido [4].

Forma parte del conjunto de herramientas como FileX, GuiX, NetX, NetX Duo y UsbX desarrolladas por *Express Logic*.

Su creador William Lamie, quien actualmente rige el cargo de presidente y CEO de Express Logis, fue uno de los desarrolladores que hicieron posible esta herramienta desde 1997 [5].

Desde entonces, ha ido desarrollando nuevas herramientas para dar soporte a la mayoría de arquitecturas conocidas (ARM, Microchip, Renesas, NXP, etc.) llegando a la friolera cifra de 6.2 millones de desarrollos en 2017 en entornos tan diversos como la medicina, industria, aeroespacial y, por supuesto, el mundo IoT.

ThreadX RTOS posee herramientas avanzadas que incluyen una arquitectura *picokerner*, programación *preemption-threshold* (evitar los cambios de contexto por procesos de mayor prioridad), encadenamiento de eventos, ejecución de perfiles, métricas de rendimiento y el poder seguir la traza del programa en todo momento.

Las funcionalidades más destacadas de ThreadX son:

- Procesos: creación dinámica y sin límites.
- Colas: creación dinámica y sin límites. Mensajes copiados por valor para mantener una persistencia. Mensajes de 1 a 16 32-bit words teniendo la opción de autosuspender procesos cuando éstos estén vacíos o llenos.
- Semáforos: creación dinámica y sin límites. Soporta una protección sobre el escenario de consumidor-productor. Suspensión pasiva de los procesos cuando el recurso no está disponible por tiempo limitado o indefinido.
- Mutex: creación dinámica y sin límites. Soporta protección en elementos anidados, escalada de prioridades y suspensión de procesos cuando proceda.
- Eventos: creación dinámica y sin límites para la sincronización entre procesos. Utilización atómica de sus funciones.
- Reserva de memoria: creación de dichas reservas de forma dinámica e ilimitada. Flexibles de utilizar y suspensión de procesos cuando no se pueda obtener más memoria.
- Timers: creación dinámica y sin límites. Causales o periódicos, solo necesitan de un timer hardware para funcionar.
- Planificador: Corresponde con la funcionalidad estrella de ThreadX ya que permite con tan solo 2KB de memoria Flash y 1KB de RAM ejecutar las herramientas básicas del RTOS. Permite los cambios de contexto en tan solo algunos escasos microsegundos siendo totalmente determinista y basado en distintos niveles de prioridades desde 32 hasta 1024.

Cuando hablamos de diseñar una aplicación embebida en un hardware limitado tenemos que tener en cuenta el tamaño de memoria (tanto Flash como RAM) que va a utilizar nuestro sistema operativo ya que principalmente es el componente que más ocupará. Para eso, ThreadX ha reducido al mínimo el tamaño necesario para utilizar algunos de sus componentes más comunes y estrictamente necesarios en un entorno de ejecución paralela:

Utilidad ThreadX	Tamaño en Flash (B)
Core	2.000
Colas	900
Eventos	900
Semáforos	450
Mutex	1200
Bloques de memoria	550
Bytes de memoria	900

Tabla 2: Cantidad de memoria utilizada por ThreadX

Pero puede que no solo se tenga que tener en cuenta el tamaño de la aplicación sino los tiempos de respuesta de la misma ante eventos que sucedan dentro y fuera del sistema. Los siguientes tiempos corresponden con una aplicación ejecutándose en un Hardware que posee un reloj de 200MHz.

Utilidad ThreadX	Tiempo de la instrucción (ms)
Suspensión de un proceso	0.6
Reanudación de un proceso	0.6
Envío de datos por una cola	0.3
Recepción de datos por una cola	0.3
Obtención de semáforo	0.2
Liberación de un semáforo	0.2
Cambio de contexto	0.4
Respuesta de una interrupción	0.0 - 0.6

Tabla 3: Tiempos de respuesta en ThreadX

5. Especificaciones del proyecto

Cuando se tiene como título un tema que puede dar paso a muchas líneas de desarrollo al mismo tiempo, es necesario especificar unas funcionalidades que debe de poseer al finalizar el mismo. De esta manera conseguimos concretar y acotar el desarrollo para que no sea demasiado extenso.

Estas especificaciones nos darán una idea de qué funcionalidades necesitará el proyecto ejemplo del punto 10. Las podemos distinguir entre funcionales y no funcionales:

5.1. Funcionales

Temperatura	El dispositivo debe ser capaz de obtener la temperatura actual.
Luminosidad	El dispositivo debe ser capaz de obtener la luminosidad actual.
Potenciómetro	El dispositivo debe ser capaz de convertir un valor analógico a uno digital con una precisión de 14 bits.
LED	El dispositivo debe ser capaz de encender un led con un rango de intensidad.
Envío	La información debe de llegar a un servidor MQTT por los distintos medios que se utilicen (Wifi, 6LowPAN, GSM/Narrow Band) de forma totalmente transparente y cómoda; permitiendo el uso de una tecnología u otra sin necesidad de realizar esfuerzos extra.
Información	Las distintas medidas deben de visualizarse en el servidor utilizando un protocolo estándar.

5.2. No funcionales

Consumo	Es imprescindible la reducción a sus mínimos para poder tener energía durante más tiempo. Evitar bucles de espera activa.
Tiempo de respuesta	Se pretende diseñar un sistema sin bloqueos innecesarios, con tiempos de espera/respuesta acotados.
Modular	El Firmware debe de facilitar la substitución de las tecnologías de comunicación ya que dependen mucho del cliente.
Escalabilidad	La arquitectura resultante debe ser escalable no solo a la incorporación de más métodos de comunicación sino al cambio de protocolo.
Estabilidad	Se requiere un sistema que funcione con un alto número de dispositivos minimizando al máximo la pérdida o retraso de la información.

6. Librerías de abstracción

La finalidad del proyecto es la de aportar un sistema de abstracción para algunas de las librerías de drivers que facilita Renesas Synergy junto con Express Logic y facilitar el acceso a periféricos, comunicaciones, memoria y utilidades que se puedan utilizar en un proyecto actual.

Antes de empezar se recomienda ir a la Imagen 5 para que el lector se haga una idea del conjunto de herramientas que aportarán los siguientes puntos.

La razón por la cual se desea extraer el acceso a los drivers básicos del microcontrolador o la utilización de la red (en cualquiera de sus tecnologías de comunicación) es simplemente para evitar que en las aplicaciones se muestre más código del que se necesita, pudiendo testear individualmente cada módulo, actualizarlo o ampliarlo sin necesidad de perder la compatibilidad con otros proyectos.

Llegados a este punto, nos encontramos con la parte fundamental del proyecto: el conjunto de librerías que enlazan la utilización del mismo Hardware (apoyándose con las descritas en el punto 8.1) con la implementación de la aplicación de usuario destinada a satisfacer las necesidades y requerimientos del cliente.

Para empezar, se presenta un esquema genérico de cómo se ha construido dicha solución para cumplir con los aspectos propuestos en las especificaciones del proyecto (véase punto 5).

Como se puede observar en la Imagen 5, existen librerías que necesitan de la utilización de otras. Un ejemplo es el FTP que depende del módulo BG96 o la integración de una página web, la cual depende del Ethernet.

Para acotar el documento, se explicará con detalle la utilización de la red en este mismo punto, la descripción de las tecnologías de red en el punto 7 y el acceso al hardware y utilidades en el punto 8.

La siguiente imagen hace referencia a todas las capas/librerías que han sido necesarias para extraer la utilización de las distintas tecnologías de comunicación desde la utilización del Hardware hasta la implementación de algunos protocolos de alto nivel, pasando por el manejo de sockets.

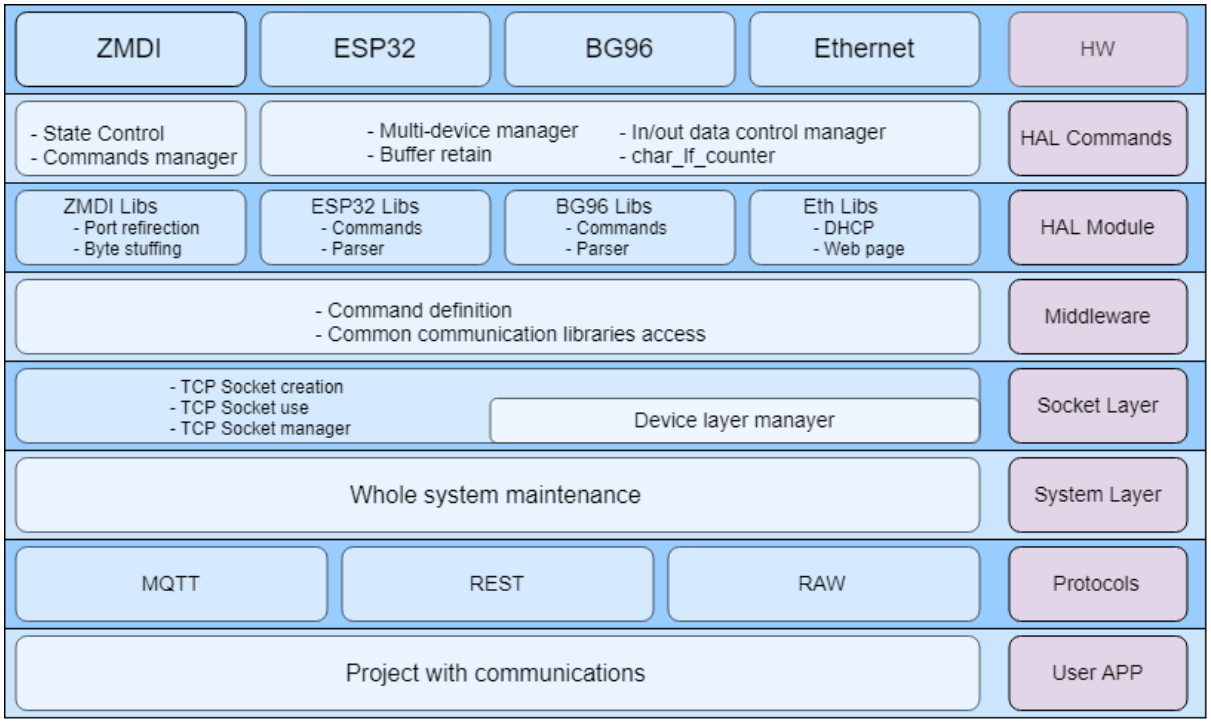


Imagen 5: Diagrama de capas sistema de comunicación

La parte de abstracción del Hardware (medio de comunicación con los módulos) se ha dividido en dos para una mejor comprensión. La primera de ellas, para controlar el acceso al medio (en este caso todos los módulos usan UART), y la segunda, para abstraer la utilización de uno u otro módulo.

Por lo que los apartados que se describirán en su orden lógico serán:

1. HW
2. HAL commands
3. HAL module
4. Middleware
5. Socket layer
6. System layer
7. Protocols
8. User app

6.1. HW

Como el lector puede entender, corresponde con la tecnología de comunicación misma. Presentándose como un módulo adjunto al diseño Hardware es capaz de acoplar y añadir una tecnología de comunicación al sistema actual.

Utilizando módulos externos al diseño, se garantiza que los mismos han pasado las certificaciones necesarias y unos procesos de test y verificación que llevarían excesivo tiempo para la finalización del proyecto.

Pero, como contrapartida, introducen un sobrecoste que dependiendo del proyecto puede ser inadmisible para lograr los requerimientos necesarios en tema de coste final del dispositivo.

6.2. HAL commands

Un componente fundamental del proyecto es cómo se envían o reciben los bytes mediante los buses de comunicación, en este caso, el único utilizado es la UART.

Por un lado, tenemos la librería de *at_process_cmd* que se encarga de interactuar con los módulos ESP32 y BG96. Dicha librería es capaz de interactuar con tantos dispositivos como se desee, tanto para el envío como para la recepción de las respuestas.

La forma en que se soporta la utilización de más de un módulo al mismo tiempo es porque se basa en una estructura básica que poseen todos ellos:

```
typedef struct{
    conn_device_id_d id;
    timer_id timer_id;
    timer_id timer_buf_delay;
    uart_instance_t *uart_var;

    void (* module_init_func)();

}at_device_conf_t;


typedef struct {

    at_device_conf_t conf;

    uint8_t buf[AT_RCV_BUFFER_LEN_MAX];
    uint16_t buf_index;

    Fifo_queue_t at_incomm_fifo;

    At_cmd_t at_cmd_t[AT_MAX_CMD];
    At_cmd_t * head;
    At_cmd_t * tail;
    uint8_t at_cmd_len;
    uint8_t char_lf_counter;

    uint8_t uart_channel;

}at_process_struct_t;
```

Gracias a esta función se puede unificar el manejo de cualquier dispositivo que se desee siempre y cuando esté basado en comandos AT.

Sus variables son:

Variable	Descripción
conf	Variables de configuración establecidas para el módulo como: ID, timer general, timer retención del buffer, acceso a la UART, función de inicialización del módulo.
buf	Buffer genérico utilizado para almacenar los bytes recibidos al hacer un parseado por el módulo en concreto.
buf_index	Tamaño del buffer genérico
at_incom_fifo	Buffer FIFO de recepción de los bytes enviados por el módulo.
at_cmd_t	Buffer FIFO de comandos generados por el usuario para ser enviados al módulo.
head	Cabeza de cola para los comandos del array at_cmd_t.

tail	Cola del buffer para los comandos del array at_cmd_t.
at_cmd_len	Tamaño de cola de los comandos almacenados en el array at_cmd_t.
char_lf_counter	Contador de caracteres “LF” contenidos en el buffer fifo at_incom_fifo.
uart_channel	Número de canal UART utilizado por el módulo.

6.2.1. Recepción

La función encargada de recibir los bytes y guardarlos en la estructura correcta es:

```
void AT_UART_READ(uint8_t channel, uint8_t data){

    // Find device which is using this UART channel
    for (uint8_t id = 0; id < MAX_COMM_DEVICE_PERIPHERAL; id++){
        if(at_process_array[id] != NULL && at_process_array[id]->uart_channel
== channel){
            fifo_queue_add_single(&(at_process_array[id]->at_incomm_fifo),
data);

            // Easy check to obtain the amount of <LF> characters at the same
time in the buffer.
            // This is user to get a concrete answer of a specific command.
            if(data == LF_CHAR) at_process_array[id]->char_lf_counter++;

            SET_BIT(at_buffer_block, channel, uint16_t); // Block the buffer
access and set the timeout to release it.
            AT_TIMEOUT_ADD(at_process_array[id]->conf.timer_buf_delay,
AT_BUFFER_RETAIN_MS, at_unlock_buffer, ONCE);
            return;
        }
    }
}
```

Como se puede observar, existe una única función para todos los módulos que utilizan el sistema de comandos AT para interactuar con ellos. La función de recepción es la encargada de localizar a qué modulo debe de ir dicho byte tan rápido como sea posible.

Para ello se recorre la estructura de dispositivos añadidos mediante la función:

```
void at_add_device(at_process_struct_t *at_struct, at_device_conf_t *c);
```

localizando el módulo que esté usando en ese momento el canal de la UART por dónde se recibe el dato. Una vez se localiza se realizan varias funciones:

- Añadir el byte al buffer de recepción FIFO que corresponda.
- Incrementar un contador local si el dato corresponde a un carácter de salto de línea.
- Bloquear el acceso al buffer de datos recibidos para no causar problemas durante un tiempo configurado.

Antes de continuar con el envío y explotación de los datos recibidos, el lector debe de entender la funcionalidad de la variable *char_lf_counter* y del timer de bloqueo del buffer de recepción.

6.2.2. Descripción de la variable char_lf_counter

Dado que los comandos AT están basados en una interacción entre la aplicación y módulo de comunicación mediante un buffer de datos ASCII, la mayoría de comandos tienen una respuesta acotada y documentada cuyo elemento común corresponde con la cantidad de bytes LF (“\n” en ascii o 0x0A en hexadecimal) que se retornan. Por lo que, ante un comando dado, el usuario puede obtener toda la respuesta obtenida sin tener que realizar un parseado de los datos en una función que a la larga puede convertirse en demasiadas líneas (como

puede observarse en la implementación de la librería BG96 ya que posee distintos protocolos extra).

Por lo que, mediante un conteo a la hora de recibir los datos, se pueden saber en todo momento la cantidad de bytes LF que están almacenados en el buffer fifo sin tener la necesidad de realizar dicha cuenta cada vez.

Comandos con una respuesta inmediata que se desean recibir en una función determinada pueden ser utilizados y extraídos de los más genéricos.

6.2.3. Problemática buffer de recepción

Una vez ya se poseía toda la librería implementada y verificada, en un desarrollo paralelo al presente proyecto, surgió el problema de que cuando se recibía una respuesta excesivamente larga y que se debía tratar mediante un parseado genérico, podía darse el caso de que no se tratase toda ella a la vez, dejando una respuesta incoherente y errónea. El problema fue descubierto cuando se implementó la descarga de archivos de gran tamaño mediante el protocolo FTP.

Para solucionar dicho problema, se diseñó un sistema en el que, para utilizar la información recibida, se debía de esperar X tiempo configurable para asegurar que se han recibido todos los bytes y no se queda ningún dato a mitad.

6.2.4. Extracción datos buffer FIFO

Una vez ya se poseen los datos recibidos dentro del buffer fifo, es hora de extraerlos y llamar a la función adecuada para su explotación mediante una función que recorrerá todos los dispositivos disponibles.

```
at_process_struct_t *s;

// Manage device by device
for (uint8_t id = 0; id < MAX_COMM_DEVICE_PERIPHERAL; id++){

    s = at_process_array[id];
    if(s == NULL) continue;

    /* DATA MANAGEMENT */
    at_line_management(id);

}
```

Con la llamada a la función `at_line_management` se pretende que el sistema sea escalable a todos aquellos dispositivos que se añadan en un futuro. Cabe destacar que, aunque la función recorra todos los dispositivos, solo se tratarán por ella los que pertenezcan a un tratamiento de comandos AT.

```
void at_line_management(conn_device_id_d d){

    uint8_t buf[AT_RCV_BUFFER_LEN_MAX];
    uint16_t buf_index;

    // Obtain pointer to the data structure where the fifos are located.
    at_process_struct_t *s = at_process_array[d];
    if(s == NULL) return;

    if((!(s->head->cmd == CMD_NO_CMD || s->head->cmd == CMD_NO_WAIT)) && s-
>head->executed == true)
    {
        if(s->char_lf_counter >= s->head->wait_lf_num){
            buf_index = (uint16_t)fifo_queue_get_until(&(s->at_incomm_fifo),
LF_CHAR, s->head->wait_lf_num, buf);
```

```
                if(buf_index > 0){
                    //buf[s->buf_index] = '\0';
                    AT_TIMEOUT_DEL(s->conf.timer_id);
                    if(cmd_answer[s->conf.id][s->head->cmd] != NULL) cmd_answer[s-
>conf.id][s->head->cmd]((char*)buf, buf_index);
                    s->char_lf_counter = (uint8_t)(s->char_lf_counter - s->head-
>wait_lf_num);
                    buf_index = 0;
                    AT_NEXT_CMD();
                }
            }
        }
        else
        {
            // Due to no command is waiting we can pass throw all the data when
the retain buffer timer was triggered.
            if(!IS_BIT(at_buffer_block, s->uart_channel) && s->at_incomm_fifo.size
> 0){
                buf_index = (uint16_t)fifo_queue_get(&(s->at_incomm_fifo), buf, s-
>at_incomm_fifo.size);
                if(buf_index > 0){
                    //buf[s->buf_index] = '\0';
                    cmd_parser[s->conf.id]((char*)buf, buf_index);
                    s->char_lf_counter = 0;
                    buf_index = 0;
                }
            }
        }
    }
```

En la función descrita arriba se presentan 2 posibles escenarios:

- Comandos con una respuesta concreta
- Comandos con una respuesta general

Cuando una aplicación genera un comando que debe de ser tratado de forma especial o de forma inmediata sin tener que esperar a que el buffer de entrada se libere (véase punto 6.2.3)

Se deberá de lanzar con un identificador de la tabla de comandos descrita para poder obtener la función de tratado que requiere. Se consideran todos ellos, comandos con identidad, excepto los lanzados con el identificador `CMD_NO_WAIT`. El sistema de determinar cuando la respuesta está completa es esperar a que N caracteres *lf* hayan sido guardados en el buffer fifo del dispositivo. Una vez se posea al menos dicho número, se procede a extraer todos los bytes comprendidos entre el inicio de la cola y el último carácter *lf* que encaje con la respuesta.

En el caso de que se pierda algún carácter *lf* provocaría que las respuestas enviadas a las funciones que las esperan fueran incompletas o incorrectas, lo que podría provocar una perdida de la respuesta al comando anterior o, en menor medida, un bloqueo total del sistema. Para solucionar dicho escenario se implementa un timeout de comando con tiempo configurable (los tiempos de respuesta de los comandos varían según la información o acción requerida) para que cuando éste elimine el comando que bloquea el sistema y así poder seguir los con demás.

Para utilizar dicho sistema se deberá de conocer a priori la cantidad de caracteres *lf* que provoca un comando en todas sus variaciones, tanto para una respuesta favorable como errónea.

En el segundo caso, para comandos de tratamiento general, se procede a enviar todo el buffer de datos recibidos (teniendo en cuenta la espera final para asegurarse de que se han almacenado todos) a una función que se encargará de realizar un parseado para encontrar todas las respuestas posibles dentro de dicho buffer.

6.2.5. Envío

El envío de comandos mediante la UART se realiza de una forma característica ya que se desea evitar el mayor número de colisiones posibles entre la respuesta de un comando y, por ejemplo, la información transmitida por el módulo de forma asíncrona (paquetes de datos entrantes, eventos, errores). Para ello se ha implementado un sistema que no permite el envío de un segundo comando mientras esté activado el primero. También es la encargada de establecer el timeout correspondiente al comando. La función que la implementa es la siguiente:

```
void at_process_send(void){
    at_process_struct_t *s;

    // Manage device by device
    for (uint8_t id = 0; id < MAX_COMM_DEVICE_PERIPHERAL; id++){

        s = at_process_array[id];
        if(s == NULL) continue;

        if(s->at_incomm_fifo.size != 0) continue;   // Important to not
interrupt a receive process.
        // To be sure that the next answer will fit the command send below,
        // the incoming buffer must be empty.
        if(s->at_cmd_len > 0 && s->head->executed == false && s->head->cmd !=
CMD_NO_CMD){

            AT_UART_WRITE(s->conf.uart_var, s->head->send_buf, s->head-
>send_buf_len); // Write to uart
            s->head->executed = true;

            if(s->head->wait_lf_num == 0 || s->head->cmd == CMD_NO_WAIT){ //
Means that the current command must be removed as soon as possible.
                AT_NEXT_CMD();
            }else{
                AT_TIMEOUT_ADD(s->conf.timer_id, s->head->timeout,
at_remove_head_cmd, ONCE);
            }
        }
    }
}
```

6.2.6. 6LowPAN

Como ya se ha mencionado, el sistema utilizado en el módulo de ZMDI no implementa comandos AT sino mediante paso de mensajes. Aunque los comandos sean totalmente distintos, no se impide a la librería a trabajar de forma similar, ya que también se posee una cola fifo de comandos que se quieren enviar al módulo, su correspondiente timeout, etc.

La función de recepción misma clasifica y detecta las distintas partes de un paquete utilizando una máquina de estados capaz de formar los paquetes de información en *streaming*, realizar el unstuffing y detectar posibles errores en la recepción.

```
void ZMDI_ProcessUsartByte(byte b){
    …
    switch(state){
    case START:
        …
    case LENGTH_LSB:
        …
    case LENGTH_MSB:
        …
    case CODE:
        …
    case DATA:
        …
```

```
        case CHECKSUM:
            …
        }
    }
```

La función tiene los siguientes objetivos:

- Detectar el inicio de un paquete
- Guardar los bytes recibidos
- Realizar un 'unstuffing' de los bytes que toquen.
- Calcular el CRC al momento y comprobar que sea el correcto.
- La última función que tiene es la de redirigir el tráfico dependiendo de si se trata de una respuesta a un comando o un paquete entrante de datos.

La función encargada del envío se describe como sigue:

```
void ZMDI_check_data_to_sent(void){

    if(commands_in_the_array == 0) return;
    if(!ZMDI_conf_is_available())  return;

    configure_comm_t *first = &conf_comm[head];

    if(first->exec_done == false){
        first->exec_done = true;
        ZMDI_TIMEOUT_ADD(TIMER_ZMDI_CONF_ALIVE, 100,
__ZMDI_conf_clear_available, ONCE);
        ZMDI_conf_available = false;
        ZMDI_UART_WRITE((uart_instance_t *)&ZMDI_UART_VAL, first->cmd, first-
>cmd_len);
    }
}
```

En dónde se comprueba que existe algún comando a enviar y si el módulo no está en modo de espera. Esto es, tal y como sucedía con la librería AT, no se envía el siguiente comando si se está esperando la respuesta del anterior.

6.3. HAL modules

Dado que en los objetivos del proyecto se menciona la escalabilidad como un requerimiento indispensable para el mismo, se pretende diseñar una librería que no esté ligada a la utilización de un Hardware determinado, sino que se pueda utilizar 6LOWPAN, Wifi, GSP/NB IoT de forma discriminante (solo una tecnología disponible) o de forma complementaria (todas las tecnologías disponibles al mismo momento. Llevándolo al extremo, las librerías desarrolladas son capaces de soportar futuras implementaciones como Bluetooth, Zigbee, LORA o SIXFOX.

Para lograr tal ambicioso objetivo se ha optado por implementar una librería de acceso al Hardware independiente de cada tecnología, pero utilizando una estructura común de utilización.

En primer lugar, ha sido la declaración de un tipo de función principal para iniciar el módulo correspondiente. Mediante dicha función, el módulo debe quedarse en un estado de "operativo" por lo que implementa las llamadas y comandos necesarios para inicializar el Hardware que utilice, los buffers y listas de control, así como, todos los timers y datos privados que se utilicen en el futuro. En algunos métodos, también se ha añadido un pequeño test para comprobar que el funcionamiento del mismo es correcto.

Las funciones de entrada de comandos para interactuar con un módulo estas declaradas como:

```
typedef void (* Command_action)(command_param_t *c);
```

Por lo que todas ellas, aún sin necesidad de parámetros adicionales, pueden llamarse desde el middleware descrito en el siguiente punto.

Para solucionar el problema de los diferentes tipos de parámetros que se pasan o se pueden pasar a las funciones se ha optado por un puntero a una estructura genérica descrita a continuación.

```
typedef struct{
    uint16_t val;
    char * buf;
    unsigned int len;
    void * pointer;
}command_param_t;
```

Como la mayoría de funciones suelen esperar las siguientes tuplas de datos {puntero buffer, tamaño}, {identificador} o un número cualquiera se han implementado algunas de estas. Para funciones más complejas como la implementación del MQTT o FTP del módulo BG96 se ha complementado la estructura con un puntero genérico. Es importante mencionar que el usuario debe ser consciente de los valores transmitidos y recibidos por las funciones del módulo ya que no se efectúa ningún control de los parámetros, los valores se utilizan en crudo; pudiendo causar un error o estado no deseado si no se tiene en cuenta la inicialización de todos los parámetros utilizados.

Una vez descrita la función que introduce cambios en los módulos de comunicación, es hora de mencionar la que recoge las respuestas descritas como:

```
typedef void (* Command_answer)(char * buf, unsigned int len);
```

es la encargada de recibir desde el módulo la información generada por el mismo.

La definición viene siendo utilizada principalmente por aquellos módulos que utilizan el “protocolo” de comandos AT para interactuar.

En caso de 6LowPAN (módulo ZMDI) se utiliza un sistema distinto.

Éste consiste en enlazar directamente las funciones descritas la parte de Middleware (punto 6.4) con las librerías del módulo directamente, para evitar crear una línea de ejecución innecesaria.

6.4. Middleware

Esta siguiente capa, podríamos denominarla “*Common Command Layer*”, debido a que se encarga de establecer un sistema de comandos para interactuar con todo el Hardware adyacente, sin tener en cuenta en qué modulo se vaya a implementar la solución.

La presente capa también es la encargada de dar el dinamismo requerido por las especificaciones del proyecto. Con ella, se podrá mantener la aplicación de usuario sin tener que modificar ninguna librería relacionada con el método de envío del paquete de datos.

Para conseguirlo, se ha declarado una serie de variables que cada nodo debe de ser responsable de cumplimentar con los métodos que se hayan implementado.

```
/*
 * Arrays to index the private function calls of each AT module.
 */
Command_action *cmd_action[MAX_COMM_DEVICE_PERIPHERAL];
Command_answer *cmd_answer[MAX_COMM_DEVICE_PERIPHERAL];
Command_answer cmd_parser[MAX_COMM_DEVICE_PERIPHERAL];
```

Para determinar los distintos comandos o posibilidades que se poseen en todo el sistema se establece una declaración para poder resumir todos ellos y tener una visión general de lo que puede aportar un módulo.

```
typedef enum{
    CMD_NO_CMD = (uint8_t)0x00, // No command - For internal use only
    CMD_NO_WAIT,                // No return expected. Command is sent and
deleted immediately.

    CMD_GENERIC,                // Answer will be "OKcrlfcrlf" or
"ERRORcrlfcrlf"

    CMD_MOD_CONF,               // Start a configuration process of the AT
module
    CMD_MOD_TEST,               // Test connection, simple "AT" cmd.
    CMD_MOD_ECHO_OFF,           // For standardization the echo mode must be
set to off.
    CMD_MOD_GET_INFO,           // Get information from the module

    CMD_MOD_IMEI,               // In case of a GSM module, update the imei.
    CMD_MOD_SIM_PRESENT,        // Check if sim is present
    CMD_MOD_SIM_ID,             // Get sim identification number
    CMD_MOD_PIN_STATUS,         // Check if pin is inserted
    CMD_MOD_PIN_INSERT,         // Insert pin

    CMD_NET_CONNECT,            // Connect to the net (GMS, Wifi...)
    CMD_NET_GET_IP,             // Obtain the local IP/MAC
    CMD_NET_CLOSE,              // Disconnect from the net
    CMD_NET_GET_SIGNAL,         // Get signal strength among the node and the
AP

    CMD_CONN_STATUS,            // Update socket status
    CMD_CONN_CONNECT,           // Connect a new socket
    CMD_CONN_SEND,              // Send data over the socket
    CMD_CONN_SEND_GROUP,        // Send data over the socket to a specific
group
    CMD_CONN_CLOSE,             // Close a socket

    CMD_GPS_START,              // Configure and start the GPS. Start a
periodically process to update the values.
    CMD_GPS_UPDATE,             // Obtain a new GPS position if it is
available
    CMD_GPS_CLOSE,              // Stop GPS and periodically update.

    CMD_FTP_START,              // Start a FTP connection with the server
    CMD_FTP_GET_SIZE,           // Get size of the file requested
    CMD_FTP_GET_FILE,           // Download the file requested
    CMD_FTP_CLOSE,              // Close FTP connection

    CMD_MQTT_CONNECT,           // Init a MQTT connection
    CMD_MQTT_PUB,               // Publish new data
    CMD_MQTT_SUB,               // Subscribe to a topic
    CMD_MQTT_UNSUB,             // Unsubscribe from a topic
    CMD_MQTT_CLOSE,             // Close MQTT connection

    CMD_LAST_CMD //Last command. Add new commands above this one.
} CMD;
```

Se pueden encontrar comandos relativos a un dispositivo genérico, a un dispositivo tipo modem (como el BG96), de conexión a la red (GSM, Wifi, 6LowPAN), establecer un socket TCP, GPS, FTP y MQTT; pero se puede ampliar tanto como necesidades se quieran.

En la función de inicialización de cada módulo de comunicación se deberá de indexar cada puntero a función que se tenga implementada. El siguiente ejemplo corresponde con la inicialización del módulo de Espressif ESP32:

```
cmd_action[LOCAL_DEVICE_ID] = esp32_action;
cmd_answer[LOCAL_DEVICE_ID] = esp32_answer;
cmd_parser[LOCAL_DEVICE_ID] = esp32_parser;

PRINTF(INFO, "%s", "Initialization of module Espressif ESP32");

for(uint8_t i = 0; i < CMD_LAST_CMD; i++){
    esp32_action[i] = NULL;
    esp32_answer[i] = NULL;
}


esp32_action[CMD_MOD_TEST]       = esp32_cmd_test;
esp32_action[CMD_MOD_ECHO_OFF]   = esp32_cmd_echo_off;
esp32_action[CMD_MOD_GET_INFO]   = esp32_cmd_get_info;
esp32_action[CMD_NET_CONNECT]    = esp32_cmd_net_connect;
esp32_action[CMD_NET_CLOSE]      = esp32_cmd_net_close;
esp32_action[CMD_NET_GET_IP]     = esp32_cmd_net_get_ip;
esp32_action[CMD_NET_GET_SIGNAL]= esp32_cmd_net_get_signal;
esp32_action[CMD_CONN_STATUS]    = esp32_cmd_conn_status;
esp32_action[CMD_CONN_CONNECT]   = esp32_cmd_conn_connect;
esp32_action[CMD_CONN_SEND]      = esp32_cmd_conn_send;
esp32_action[CMD_CONN_CLOSE]     = esp32_cmd_conn_close;

esp32_answer[CMD_GENERIC]        = esp32_ans_general;

esp32_answer[CMD_MOD_GET_INFO]   = esp32_ans_get_info;
esp32_answer[CMD_NET_GET_IP]     = esp32_ans_net_get_ip;
```

Para que no haya problemas, todas las funciones se deben de inicializar a NULL, siendo este el indicativo de que dicha funcionalidad no está disponible o implementada en el módulo seleccionado.

Llegados a este punto, las librerías subyacentes solo deberían tener acceso a dicha capa de abstracción para asegurarse de que la portabilidad entre tecnologías se realiza satisfactoriamente.

Para finalizar la descripción de la capa de Middleware se destaca la definición de los estados principales que sirven para identificar en qué situación se encuentra cada dispositivo, conexión a la red y socket entre otras:

```
typedef enum {
    CONN_ERROR = (uint8_t) 0x00,
    CONN_STOP,
    CONN_CONFIGURED,
    CONN_CONNECTED
}conn_status_d;
```

6.5. Socket layer

Una vez se tiene un único acceso a las distintas funciones de cada módulo para abrir, recibir y transmitir datos a través de un canal de comunicación, se necesita de una capa extra que normalice todas las llamadas a dichas funciones.

Esta capa extra es debida a que la cantidad de información necesaria por una tecnología de información es distinta para cada tipo, pero sin embargo, se desea tener un único acceso para poder actuar sobre los distintos canales de comunicación.

Como se puede observar, las funciones que se declaran son las habitualmente utilizadas:

```
void conn_socket_open(conn_socket_t *s);
void conn_socket_close(conn_socket_t *s);
uint16_t conn_socket_send(conn_socket_t *s, uint8_t *buf, uint16_t len);
uint16_t conn_socket_rcv(conn_socket_t *s, uint8_t *buf, uint16_t len);
uint16_t conn_socket_rcv_all(conn_socket_t *s, uint8_t *buf);
```

Mientras que la estructura encargada de mantener el acceso y la información al mismo se declara como:

```
typedef struct{
    bool used;
    conn_status_d status;
    uint8_t id;
    conn_device_id_d device;
    Fifo_queue_t fifo_in;

    char * host;
    uint16_t port;
}conn_socket_t;
```

Como se explica a continuación, el sistema se ha preparado para poseer una serie de sockets declarados de forma estática ya que el número de ellos depende mucho del módulo que se utiliza y de cuantos estén en funcionamiento al mismo tiempo.

- Para dar soporte a esta estructura mayor, se utilizará *used* para indicar que el socket, como bien se puede entender, está siendo utilizado en ese momento.
- Un socket se identifica, primero con el dispositivo (*device*) de deberá de utilizar y con el *id* que posee dentro de este.
- Puede ser que los sockets de un mismo dispositivo se encuentren en estados (*status*) distintos para indicar que se está en error, parado, configurándose o funcionando:

```
typedef enum {
    CONN_ERROR = (uint8_t) 0x00,
    CONN_STOP,
    CONN_CONFIGURED,
    CONN_CONNECTED
}conn_status_d;
```

- Para saber a dónde se debe o está conectado el socket se mantiene el *host* y el puerto (*port*).
- Por último, describir que el envío de los datos se forma in situ y se inserta en los comandos a enviar directamente al módulo sin tener que pasar por una cola intermedia, evitando esperas innecesarias y reduciendo la complejidad y memoria utilizada por el sistema.
- Sin embargo, la recepción de datos (al realizarse de forma asíncrona) se debe de almacenar en una cola fifo implementada de forma genérica para la ocasión, dando la responsabilidad a la aplicación final de cuando utilizar dichos datos.

Un ejemplo de función implementaría sería la de envío:

```
uint16_t conn_socket_send(conn_socket_t *s, uint8_t *buf, uint16_t len);
```

Aunque no son sockets como tal, se debe de explicar en el presente apartado de que también se ha generado una librería de abstracción genérica para interactuar con los dispositivos a nivel de la red en la que operan:

```
void conn_device_connect(conn_device_id_d d);
void conn_device_disconnect(conn_device_id_d d);
void conn_device_print_ip(conn_device_t *p);
```

Definiendo por otra parte la estructura de información básica que se necesita saber respecto a ellos:

```
typedef struct {
    uint8_t node_ip[CONN_IP_LEN];
    uint8_t node_mac[CONN_MAC_LEN];
    conn_status_d status;
}conn_device_t;
```

6.6. System layer

Llegados a este punto, cabe recordar que la implementación de todas las librerías de comunicación no condiciona al usuario que las utilice en su proyecto, ni mucho menos, tener que compilar las que no se van a utilizar. Para conseguir el siguiente objetivo se utiliza en todo el proyecto la compilación condicional:

```
typedef enum{
#if COMM_PERIPHERAL_ENABLE_ESP32 == 1
    ESP32,
#endif
#if COMM_PERIPHERAL_ENABLE_BG96 == 1
    BG96,
#endif
#if COMM_PERIPHERAL_ENABLE_6LOWPAN == 1
    SIXLOWPAN,
#endif
#if COMM_PERIPHERAL_ENABLE_ETHERNET == 1
    ETHERNET,
#endif
    MAX_COMM_DEVICE_PERIPHERAL //This item must be always at the end.
}conn_device_id_d;
```

Tal y como se definió en el apartado anterior, se desea tener una cantidad finita de sockets en todo el sistema, para ello, se define cada uno de ellos por un nombre característico:

```
typedef enum{
    PRINCIPAL = (uint8_t)0x00,
    MAX_SOCKET_NAME //This item must be always at the end and less than
SOCKET_LIMIT.
}conn_name_d;
```

En la aplicación demostrativa del proyecto (punto 10) sólo se va a utilizar un socket para comunicarse con el servidor por lo que solo se define *PRINCIPAL*.

Desde la capa de Hardware explicada en el actual punto, pasando por el firmware que unifica el mismo en una serie de llamadas comunes, se llega al punto en el que el sistema posee un estado global del sistema de comunicaciones mediante las siguientes definiciones:

```
typedef struct{
    uint8_t id[CONN_MAC_LEN];
    conn_device_t comm_peripheral[MAX_COMM_DEVICE_PERIPHERAL];
    conn_socket_t sockets[MAX_SOCKET_NAME];
}global_device_t;

global_device_t COMM_DEVICE;
```

Con ella se puede acceder, interactuar o consultar el estado y acciones que hacen posible que una aplicación demandada por un cliente pueda poseer un sistema de comunicaciones y la posibilidad de ser ampliado sin necesidad de un sobrecoste considerable en tiempo de desarrollo.

Para ver un ejemplo de cómo hacer uso de ella, se deberá de leer el punto 10.

7. Librerías de acceso al medio

Cada día se comercializan más módulos con un firmware cerrado y programado de fábrica que poseen una gran cantidad de funciones, o incluso, distintos protocolos u otras tecnologías de comunicación evitando tener que implementar/verificar más código o certificar la antena por separado. Un claro ejemplo son los descritos a continuación.

Para evitar que la librería crezca de tal forma que las opciones que aporta sean contraproducentes a los desarrollos del sistema, se deben de implementar exclusivamente aquellas funcionalidades que sean estrictamente necesarias.

Para ello, se dividirá la funcionalidad de una librería en 3 claros bloques:

- Comandos relacionados para extraer información del módulo.
- Comandos de conexión con la red.
- Comandos para la creación y utilización de sockets.

7.1. Desarrollo 6LowPAN

7.1.1. Introducción

Dado que 6LowPAN no es una tecnología corriente como lo puede ser Wifi, Narrow Band o GSM, se va a proceder a realizar una pequeña introducción al respecto.

6LowPAN es el acrónimo en inglés de *IPv6 over Low-Power Wireless Personal Area Networks*. Corresponde con un sistema utilizado por una variedad de aplicaciones que de carácter sensorial en donde los nodos envían datos a través de paquetes IPv6.

La utilización de una tecnología como IPv6 viene justificada por [6]:

- Adoptarla es solo cuestión de tiempo, utilizar IPv4 ya tiene sus problemas.
- Escalabilidad.
- Eliminar la barrera de NAT. Existencia de “infinitas” IPs.
- Seguridad rediseñada.
- Stacks de código compactados. Perfectos para dispositivos con capacidades limitadas.
- Extensión de la red con IoT.
- Movilidad.
- Dirección IP autoasignada.
- Totalmente compatible con el Internet ya existente.

Definido en el RFC6282 [7] como un estándar abierto por el IETF utiliza otros estándares comúnmente utilizados en internet como TCP, UDP y HTML. Pero el mayor aporte de dicha tecnología es que fue diseñada para soportar IEEE 802.15.4 (redes inalámbricas de baja energía utilizando la frecuencia de 2.4GHz) pero también en Sub-1-GHz por lo que se puede utilizar con protocolos como Bluetooth Smart, *power line control* (PLC) y Wifi de baja energía.

Como se puede ver en la Imagen 6, es común ver redes que engloban no solo un único punto de acceso o tecnología, esto es debido a que por limitaciones que acceso a la red, distancia, movilidad o rapidez, es preciso que se convine más de una. 6LowPAN se diseñó para no dificultar dicha flexibilidad y para no introducir más que un único elemento que haga de pasarela entre las diferentes tecnologías. De tal forma un nodo 6LowPAN se puede comunicar con un programa exterior y viceversa de forma totalmente transparente utilizando IPv6.

Los routers frontera o pasarelas pueden utilizar mecanismos como NAT64 (RFC 6146) o la utilización de IPv4 en capas superiores de la red siendo compatible la utilización de este protocolo en todo el sistema y la ampliación mediante subredes IPv6. Además, estos routers solo están destinados a reenviar los datagramas a la capa de red, en ningún momento necesitan mantener ningún tipo de estado por lo que 6LowPAN supone una ventaja frente a protocolos más convencionales (ZigBee, Z-ware o Bluetooth).

Existen 3 tipos de dispositivos en la red: routers, nodos y pasarelas [8].

La diferencia entre ellos es principalmente las capacidades u operaciones que son capaces de llevar a cabo:

- **Nodos**: también conocidos como dispositivos finales, se encargan de conectarse a la red y generar o consumir información. Suelen estar destinados al mínimo consumo y envíos de pequeñas cantidades.
- **Routers**: como bien indica su nombre, estos dispositivos estarían destinados a reencaminar los paquetes dentro de la red. Se trata de dispositivos que siempre deben de estar disponibles ya que, sin ellos, todos los nodos que estén conectados a uno de ellos no podrán o enviar o recibir nuevos paquetes. Su función principal es la de mantener y extender la red.
- **Pasarela**: Como bien se ha explicado anteriormente, su función es la de transformar una red 6LowPAN a otra IEEE 802.3 o ethernet, GSM, Wifi, etc. Supone el dispositivo en dónde se suele situar la conversión de IPs o Firewalls si se precisa por la estructura del proyecto.

Aunque en este documento se divida la tecnología en 3 tipos distintos, pueden encontrarse otras fuentes (mayoritariamente dependiente del fabricante del chip) que lo explique con distinta agrupación. Esto es debido a que un mismo dispositivo puede llegar a ser nodo y router a la vez, ambos nodos son complementarios entre sí.

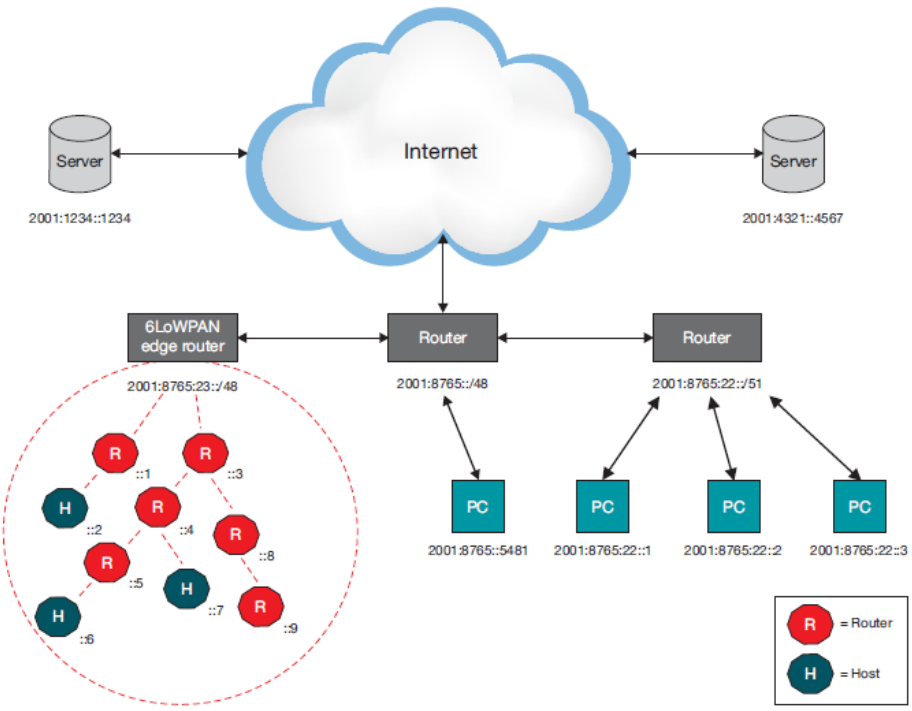


Imagen 6: Ejemplo de arquitectura con 6LowPAN [8]

Como la mayoría de tecnologías inalámbricas, también posee diferentes canales, modularidades y potencias. Por otro lado, también posee un número PAN-ID para designar una agrupación de dispositivos y formar en un mismo espacio distintas redes ya que los dispositivos solo podrán comunicarse con los que pertenezcan a su mismo PAN-ID.

7.1.2. Módulo utilizado

El módulo seleccionado para la utilización de esta tecnología será el ZWIR4512 de IDT. Dentro de él posee un microcontrolador del modelo STM32F103RC que posee un ARM Cortex-M3 de 32 bits:

- Velocidad de 64MHz.
- 256 KB de Flash.
- 48 KB de RAM

Ya que se trata de un encapsulado, y el stack proporcionado por IDT para ejecutar 6LowPAN lo permite, se pueden hacer uso de la mayoría de periféricos que posee por defecto el microcontrolador. Un diagrama de periféricos es el siguiente:

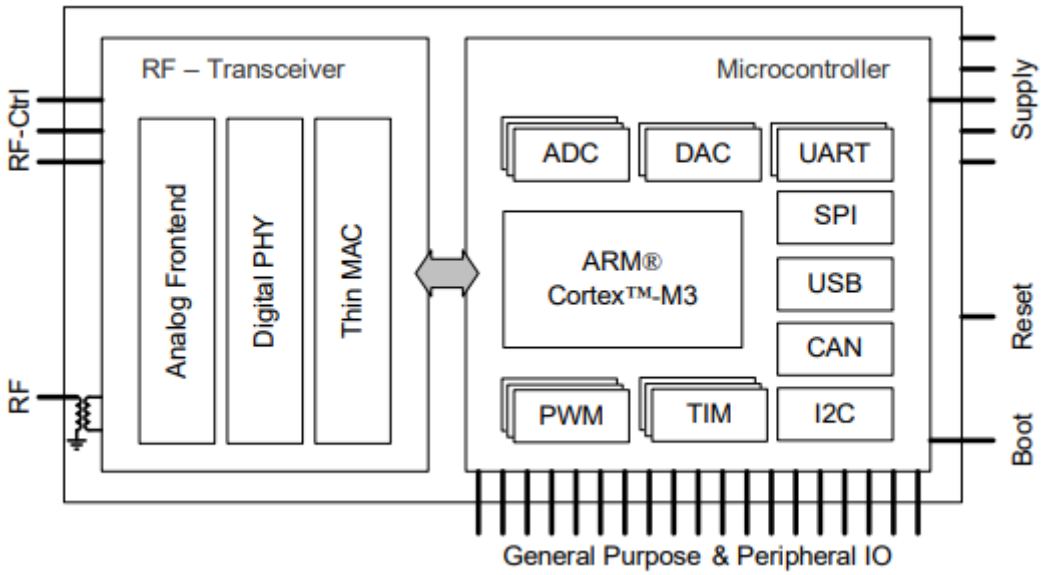


Imagen 7: Diagrama módulo 6LowPAN [9].

Para utilizar dicho módulo hay 2 opciones: programación típica sobre librerías ya compiladas o mediante una SCI. Debido a que se pretende poseer un sistema de control maestro-esclavo, con la utilización del microcontrolador S128; se optó por esta segunda opción.

La estructura del mensaje que utiliza el módulo para recibir y responder a los comandos es:

START	LENGTH	CMD	PAYLOAD	CHKS
1 byte	2 bytes	1 byte	Length bytes	1 byte

Tabla 4: Formato comando ZMDI

Como norma general, todas las variables que se envían (longitudes, puertos, tiempos, etc.) se situarán en el comando con un formato *Little Endian*, tan solo las direcciones IPv6 y los buffers de memoria de enviarán con formato *Big Endian*. Como el byte de inicio es siempre 0x7E en necesario implementar una operación de *byte stuffing* y viceversa para evitar inicios de comando en donde no proceda. Para consultar todas las opciones que proporciona el Firmware de SCI, se puede consultar el documento oficial en [10].

La utilización del módulo posee un inconveniente remarcable, en la capa de transporte solo posee UDP tal y como se puede ver en la Imagen 8 obtenida de la guía de programación del módulo [11].

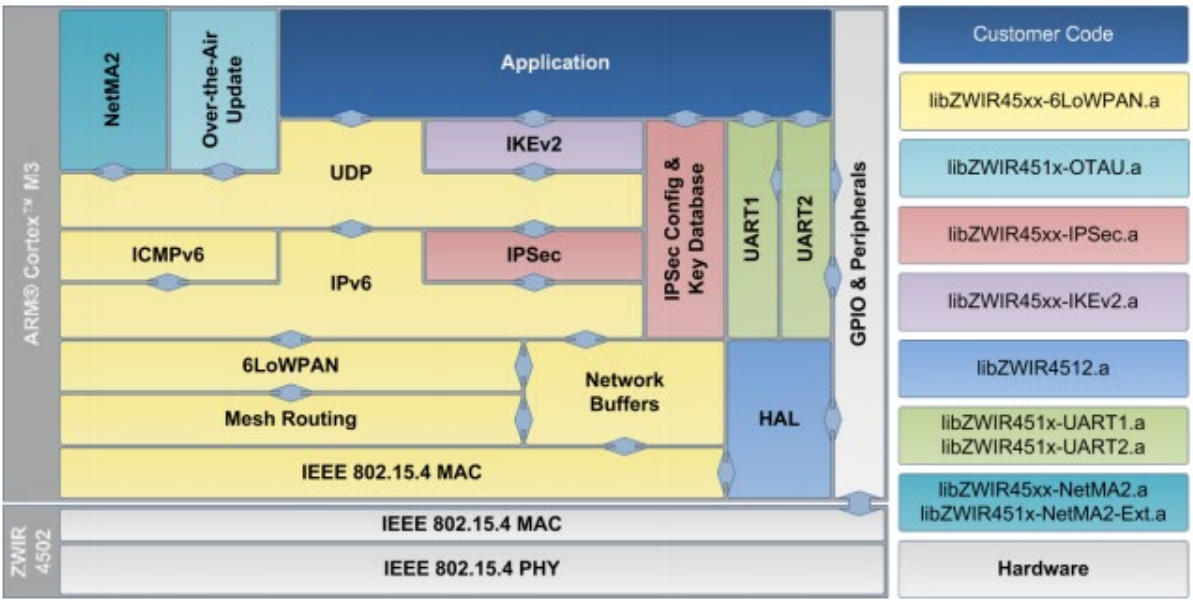


Imagen 8: Arquitectura módulo ZMDI.

Dicha limitación supondrá realizar un cambio de última hora en el proyecto para que se ajuste a las necesidades de un cliente, la utilización del protocolo TCP junto con IPv4. Se tratará el problema y la solución en el punto 7.1.4.

7.1.3. Librería

La librería de 6LowPAN se ha implementado de forma modular para que se pueda acoplar a cualquier proyecto que desee utilizar dicha tecnología de comunicación. Por otra parte, se ha tenido que adaptar para que pueda ser utilizada mediante el uso de la librería de abstracción, objetivo del presente proyecto.

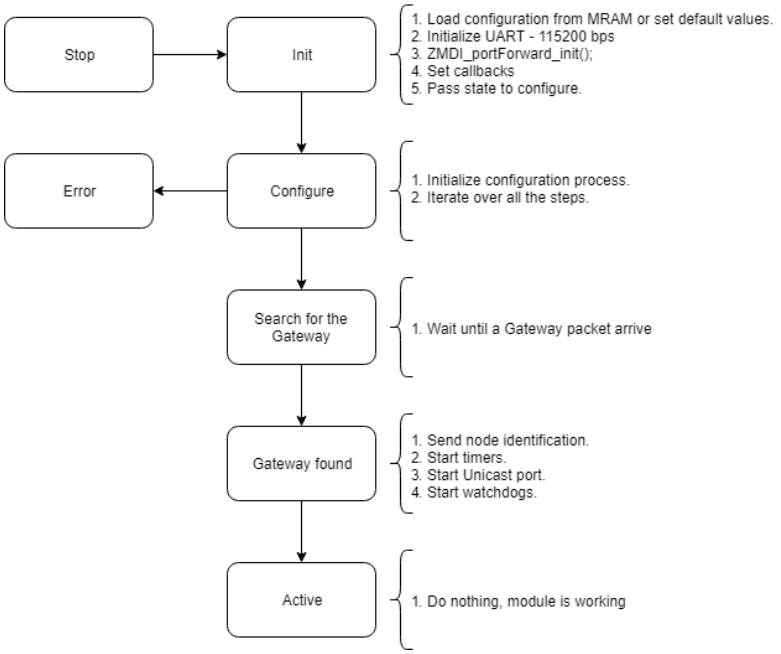


Imagen 9: Proceso del estado librería 6LowPAN

Para que el lector se haga la idea del funcionamiento de la librería y teniendo en cuenta de que la misma debe de ser funcional, tanto para aplicaciones con RTOS como si no, se posee una función principal que ha de ser llamada periódicamente.

```
void ZMDI_loop(void);
```

Dicha función es la encargada de hacer funcionar la librería 6LowPAN para inicializar, configurar, buscar un Gateway, procesar la recepción y los envíos de información.

Mediante `ZMDI_working_mode` se mantiene el estado del módulo en todo momento y brinda la posibilidad de realizar acciones conforme a ello:

```
void ZMDI_loop(void){

    RecepcionUsart();
    ZMDI_check_data_to_sent();

    switch(ZMDI_working_mode){

    case ERROR:
        break;
    case STOP:
        break;
    case CONFIGURE:
        __ZMDI_start_configuration();
        break;
    case SERVER_SEARCH:
        break;

    case SERVER_FOUND:
        ZMDI_execute_when_server_found();
        ZMDI_rcvUDP(UNICAST_PORT);
        ZMDI_addPortForward(UNICAST_PORT, incom_port_unicast, true);
        ZMDI_working_mode = ACTIVE;
        node_connected = 0;

        break;
    case ACTIVE:
        break;
    }
}
```

Como la mayoría de librerías posee una función inicial la cual es la encargada de poner en funcionamiento todo:

```
void ZMDI_init(void);
```

Como se puede ver en la Imagen 9, su función principal es:

1. Cargar desde memoria la configuración establecida por el usuario o, en caso de ser la primera vez, cargar la configuración por defecto.
2. Iniciar los drivers de la UART del microcontrolador en dónde se va a utilizar la tecnología.
3. Configurar el mecanismo de recepción de paquetes: inicializar estructuras y callbacks. Dicho procedimiento se explicará en adelante.
4. Llamar al proceso de configuración.

La función de inicio se entiende que solo se podrá ejecutar cuando se desee inicializar el módulo al arrancar el programa o en caso de que se quiera aplicar alguna configuración sin tener la necesidad de reiniciar el módulo.

Una vez se tiene el módulo inicializado, es hora de configurarlo.

Para ello, y utilizado en todo el módulo, se ha diseñado un método de interacción con el encapsulado que permite el envío de comandos de forma iterativa, eliminando la posibilidad de solapamiento entre los distintos comandos y pudiendo capturar la respuesta para poder determinar si se ha realizado correctamente o no.

El proceso de configuración se realizará utilizando dicha capacidad ya que las respuestas serán necesarias.

```
typedef enum {
    START_NO_CONF  = (byte)0x00,
    FACTORY_RESET,
    ENABLE_ACK,
    TEST,
    GET_FIRM_VER,
    GET_NET_STAT,
    GET_ADDRESS_CONF,
    SET_PANID,
    SET_PHY_CONF,
    GPIO_CONF_1,
    GPIO_CONF_2,
    NETWORK_RESET,
    GET_PHY_CONF,
    SET_DEFAULT_SEC,
    CONF_GROUP,
    RCV_BROADCAST,
    JOIN_GW_GROUP,
    RCV_GROUPS,
    JOIN_MY_GROUP,
    START_DONE,
} ZMDI_logic_conf;

ZMDI_logic_conf ZMDI_conf_state;
```

Mediante una máquina de estados se intentará, por las anteriores acciones, completar el proceso. Destacar que, siempre se partirá de la configuración que posee de fábrica para partir de un estado conocido; se obtendrá información del mismo y se configurará tanto el PANID como el canal, la modulación y la potencia.

Posteriormente, se configurarán los periféricos y se configurará la seguridad si precede. Para terminar, se configurará la recepción de tramas broadcast y la adhesión a un grupo determinado.

El funcionamiento de los grupos utiliza la funcionalidad que IPv6 introduce junto con las direcciones *multicast* [12], por las que un nodo puede enviar paquetes a un grupo determinado sin que lo reciban todos o se tenga que especificar en el payload.

En caso de que el proceso de configuración haya finalizado satisfactoriamente, se procede a configurar el Gateway debido a la problemática que se tiene (descrita en el punto 7.1.4). Se necesita del mismo para que toda la información saliente hacia el servidor encuentre el punto de acceso a la red exterior.

El Gateway enviará un payload acordado para que todos los nodos sepan de su existencia. Cuando lo perciben, configuran la dirección de salida y ejecutan una serie de instrucciones programadas. Se ha intentado programar toda la librería para que se parezca a las demás por lo que se podría encontrar una similitud de `SERVER_FOUND` a haber establecido una conexión TCP con otra tecnología de comunicación.

Lo más importante de dicho estado es la del envío de la identificación por el canal de comunicación. Una vez percibida por el servidor se entenderá que el nodo está totalmente conectado con el mismo y puede pasar a un estado de funcionamiento normal o `ACTIVE`.

Dada la simplicidad de la librería, y una vez en su funcionamiento normal, ya puede invocarse cualquiera de las siguientes funciones para enviar información al servidor:

```
void ZMDI_sendUDP(uint8_t * ip, uint16_t port, uint8_t *data_buf, uint16_t
data_len);
void ZMDI_sendUDP_Server(uint8_t *data_buf, uint16_t data_len);
void ZMDI_sendUDP_Group(uint8_t *data_buf, uint16_t data_len);
```

Con dichas funciones se pretende abarcar todas las posibilidades de comunicación:

- Dirigida a un nodo y puerto especificado.
- Dirigida al servidor con la IP obtenida y el puerto acordado.
- Dirigida al grupo al que está subscrito el nodo.

El proceso de recepción puede ser un poco más complicado ya que se ha implementado una pequeña librería intermedia que filtra el tipo de tráfico pudiendo tener distintas aplicaciones escuchando es distintos puertos.

Partiendo del payload recibido se obtendrán los siguientes campos:

Src IP	Local PORT	PAYLOAD
16 bytes	2 bytes	Length bytes

Tabla 5: Estructura paquete recibido 6LowPAN.

Por lo que una aplicación local puede abrir la escucha de un puerto y configurar la recepción de todo el tráfico a una callback establecida requiriendo solo el payload del mensaje o su totalidad para saber desde qué dirección se envió:

```
bool ZMDI_addPortForward(uint16_t port, ZMDI_command_response * cb_, bool
only_data);
bool ZMDI_removePortForward(uint16_t port);
```

Como funcionalidad extra existe la opción de configurar una función que procese todos los paquetes entrantes independientemente de las aplicaciones que hayan configurado una callback de recepción en un puerto.

```
ZMDI_command_response *  ZMDI_getAllPortForward();
void ZMDI_setAllPortForward(ZMDI_command_response * cb_, bool
only_data);
```

	Send options		
	Node	**Gateway**	**Server**
Node	Multicast using groups	Unicast	By indirect mode
Gateway	Unicast	None	Traffic redirection
Server	By indirect node	Unicast	None

Dado que el módulo de IDT no funciona mediante comandos AT, se han tenido que simular algunas de las funciones para que un programa “genérico” siga siendo funcional sin tener que modificar nada del mismo.

A continuación se muestran los cambios:

```
sixlowpan_action[CMD_NET_CONNECT]       = sixlowpan_cmd_net_connect;
sixlowpan_action[CMD_NET_CLOSE]         = sixlowpan_cmd_net_close;
sixlowpan_action[CMD_CONN_CONNECT]      = sixlowpan_cmd_conn_connect;
sixlowpan_action[CMD_CONN_SEND]         = sixlowpan_cmd_conn_send;
sixlowpan_action[CMD_CONN_CLOSE]        = sixlowpan_cmd_conn_close;
```

Como se ha dicho, el nodo se conectará automáticamente a la red *mesh* que forman el resto de dispositivos por lo que la conexión se hace automáticamente, pero mediante la función de *sixlowpan_cmd_net_connect* se puede esperar hasta que dicho proceso haya concluido.

Debido a que el desarrollo de la librería de 6LowPAN fue para un proyecto anterior, esta no posee todas las opciones que tienen las demás.

7.1.4. Problemática y solución

El requerimiento de un cliente exigía utilizar redes de baja frecuencia. Para ello se optó por la presente tecnología emergente, pero para ahorrar tiempo de desarrollo y mantener el servidor que el cliente ya disponía, se necesitaba que cada nodo estuviera identificado mediante una única comunicaciones TCP junto con IPv4.

Como se ha visto en el punto 7.1.2, el módulo estudiado solo dispone del protocolo IPv6 y UDP por lo que se necesitaba un dispositivo intermedio que hiciera la función de “NAT”.

Dada la problemática presentada al encontrar un Gateway o pasarela comercial que se ajuste a las necesidades de la red, así como a las especificaciones de la instalación, tanto por parte del cliente como por limitación de la solución estudiada, se ha visto ante la solución única de implementar nuestra propia solución.

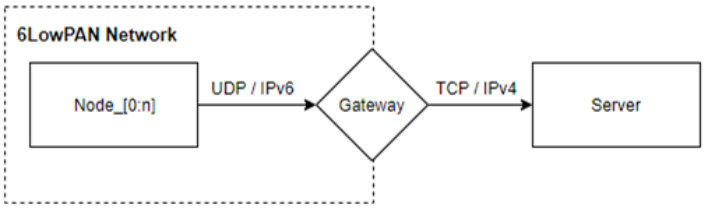


Imagen 10: Esquema Gateway 6LowPAN

Para ello se ha planteado el Gateway como un “NAT” donde se convierte de N sockets UDP a N diferentes conexiones TCP. De esta forma el servidor, aun viendo exclusivamente una única dirección IP (la del Gateway), poseerá tantas conexiones como nodos estén transmitiendo desde la red 6LowPAN. Cabe decir que para cada nodo se creará una nueva conexión identificada en dónde solo habrá información de dicho nodo.

Además, la solución implementada por el Gateway supone un movimiento totalmente transparente de información, esto es, el Gateway desconoce totalmente del protocolo que se implementa en el contenido de los paquetes. Dada esta característica, se pueden usar protocolos ASCII, mediante mensajes o incluso MQTT.

7.1.4.1. Funcionamiento Gateway

El Gateway es capaz de mapear cada paquete de datos entrante con una conexión ya existente o asignar una nueva en caso de que sea esta la primera comunicación.

Cada cierto tiempo configurable, el Gateway emite una trama establecida a todos los nodos de la red para que estos se percaten de la existencia del mismo y hacia dónde deben de enviar la información para que le llegue al servidor.

Mediante este funcionamiento, se permite una substitución del Gateway sin necesidad de configurar cada nodo individualmente, estos se configuran solos.

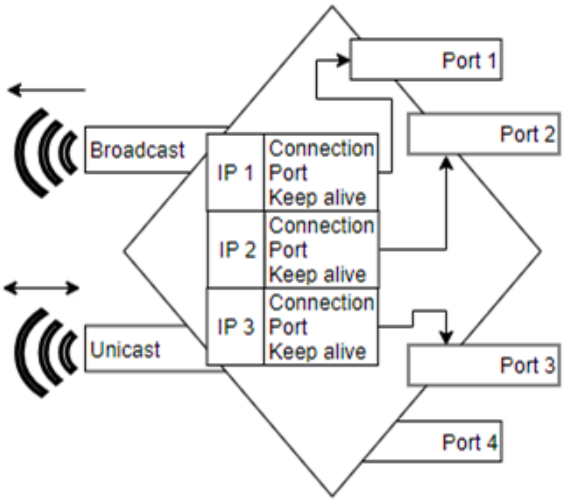


Imagen 11: Esquema funcionamiento Gateway 6LowPAN.

Una vez el nodo ya posee el punto de salida de la red, interactúa con el Gateway como si se tratase de un servidor, pero el que realmente responde a las peticiones es el servidor principal. Por el contrario, cuando el servidor envía datos a una conexión TCP del Gateway este busca en su tabla de índices a qué IPv6 pertenece y envía toda la información a través del módulo ZMDI.

7.1.4.2. Herramienta de control del Gateway

Como se esperaba, los nodos finales deben presentar la opción de ser configurables desde el servidor final, ya que ahorra tiempo ante un cambio en la instalación; por lo que el Gateway deberá poseer dicha opción también.

Se obtuvo dos escenarios en dónde el Gateway puede ser configurado:

- En su proceso de instalación
- Ante un cambio en el servidor

El primer punto sería cuando se inicia la instalación por primera vez, teniendo que ir un técnico a un lugar específico y con acceso físico a la red de comunicación, el otro sería directamente cuando en el servidor se cambia la configuración de la red.

La opción elegida para ello será abrir pequeño servidor TCP en el Gateway que espere una conexión y un protocolo designado que pueda interpretar.

Como elemento común a ambos escenarios de comunicación se tendrá un protocolo ASCII y en exclusivo, para el primero, se ha diseñado una pequeña aplicación en Java que muestra el estado actual del Gateway y la posibilidad de configurarlo.

En la Imagen 12 se puede ver la pantalla de inicio de la aplicación, en la misma se puede interactuar con el nodo seleccionado para que envíe su estado actual al servidor o encienda/apague su relé. Esta funcionalidad es propia del protocolo interno de comunicación y específica del producto demandado por el cliente.

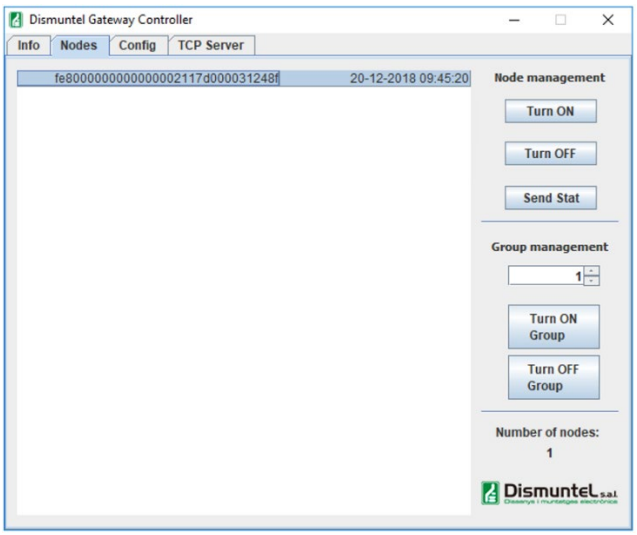


Imagen 12: Aplicación control Gateway 6LowPAN

7.1.4.3. Funcionalidades implementadas

Las funcionalidades que implementa el Gateway hasta el momento son:

- Conversión del payload contenido en una trama UDP/IPv6 al estándar TCP/IPv4.
- Mantenimiento de una tabla de conversión dada una IP con su correspondiente conexión.
- Eliminación de las conexiones caducadas, cerradas o sin utilización desde cierto tiempo.
- Configurar el módulo 6LowPAN que utiliza.
- Completa configuración utilizando la aplicación diseñada.

Desde la aplicación se puede:

- Visualizar en tiempo real la cantidad de nodos visibles por el Gateway.
- Visualizar la última comunicación realizada entre el Nodo y el Servidor.
- Encender y Apagar el relé de un nodo (Propio de un producto específico).
- Encender y Apagar el relé de un grupo determinado (Propio de un producto específico).
- Configurar completamente el Gateway.
- Iniciar un sencillo servidor TCP para visualizar las tramas entrantes. Pudiendo encender y apagar los relés periódicamente y cerrar conexiones forzadamente.

7.1.4.4. Configuración

Para entender el proceso de configuración se hará uso de la aplicación Java diseñada para configurar la instalación en caso de tener acceso a la misma red en la que se sitúa el Gateway.

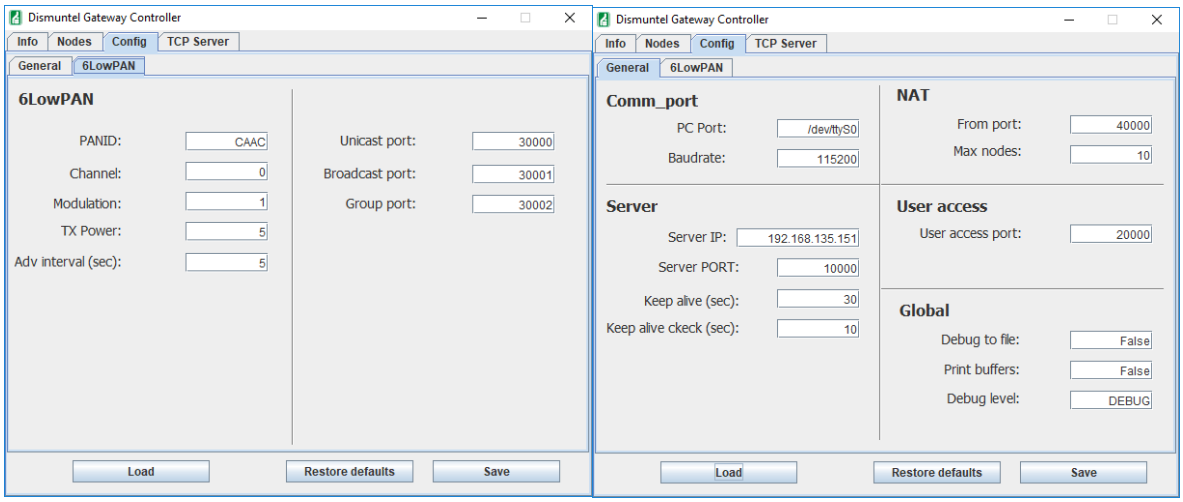


Imagen 13: Configuración Gateway 6LowPAN

Para entenderlo, se necesita saber el significado de los distintos campos. Como se puede observar, la mayoría de ellos son propios de una red 6LowPAN (Apartado 7.1.1) y otros específicos de la aplicación y modo de funcionamiento del sistema:

Campo	Explicación
Debug to file	"True" = los logs se guardan en fichero. "False" = los logs se muestran por consola.
Print buffers	"True" = mostrar buffers de recepción. "False" = no muestra los buffers de recepción.
Debug Level	Nivel de log que se desea ("DEBUG", "INFO", "ERROR").
PC port	Puerto de comunicaciones por dónde se conecta el PC al módulo ZMDI. En la Raspberry PI seguramente sea: /dev/ttyS0
Baudrate	Baudrate. (No cambiar, el ZMDI funciona a 115200 bps)
Server IP	Dirección IP del servidor a conectar la red.
Server Port	Puerto del servidor a conectar la red
Keep alive (sec)	Las conexiones no vistas en X tiempo son eliminadas.
Keep alive check (sec)	Periodo de comprobación de timeouts caducados.
From port	Primer puerto a ser utilizado por el Gateway
Max nodes	Número máximo de nodos conectados
PANID	PANID de la red en formato hexadecimal (Por defecto "CAAC")
Channel	Canal de la red: - Europa: 0 y desde 100 hasta 102

	- Norte América: desde 1 hasta el 10
Modulation	Modulación de la red: - 1 = BPSK - 0 = QPSK
TX Power	Potencia de envío. Ver normativa correspondiente. En Europa +- 5dbs.
Broadcast Port	Puerto de envío de trama del Gateway. (No cambiar)
Unicast Port	Puerto de comunicaciones Nodo – Gateway. (No cambiar)
Group Port	Puerto utilizado para la comunicación entre grupos. (No cambiar)
Adv interval (sec)	Periodo de emisión de trama broadcast.
User Access Port	Puerto de conexión para interactuar con el Gateway mediante comandos.

Tabla 6: Parámetros configuración 6LowPAN

7.1.4.5. Posibles mejoras

1. Terminar de implementar toda la funcionalidad que proporciona el módulo 6LowPAN.
2. Seguridad entre el Gateway y el Servidor.
3. En caso de tratarse de un PC con tarjeta Wifi poder establecerla en modo AP para comunicarse desde ella misma utilizando SSL o la aplicación.
4. Encriptar la comunicación entre la aplicación y el Gateway
5. Poder escanear la red para detectar los nodos que no están comunicando.

7.1.4.6. Interacción desde un servidor externo

El protocolo de interacción con el Gateway se basa en una comunicación TCP/IP y en formato Ascii (excepto en el caso de la emisión de buffers de datos). El puerto por defecto será el 20000.

La estructura de la trama será:

<get/set/rst>;<command>;<data_1>;<data_2>

Como se puede ver los inicios de trama con “get” o “set” solo podrán ser generados por el cliente y no por el Gateway ya que este posee un funcionamiento pasivo, esto es, si no se le solicita nada este no genera ningún comando. Por otra parte, “rst” solo puede ser generado por el Gateway.

Entre campo y campo, se situará un “;” como carácter de separación. Hay que tener en cuenta que al ser un protocolo compartido entre Ascii y Bytes puede llegar a causar problemas la división de instrucciones. Para solucionarlo se ha optado por empezar la trama mediante el protocolo Ascii y separación de “;”, en el momento que se quiera enviar un campo con formato de bytes se entenderá que el resto de trama sigue siendo parte de ese campo y por lo tanto no hay forma de incluir una siguiente instrucción a continuación.

A continuación, se muestra el formato de trama.

Comandos de consulta

Consulta listado nodos conectados.	
Comentario	Petición del listado de nodos conectados.
Ejemplo	get;ip
Respuesta	rst;ip;1;fe8000000000000002117d0000305438;27-12-2018 19:45:48

Consulta configuración actual Gateway	
Comentario	Petición de la configuración actual del nodo. Respuesta en JSON.
Ejemplo	get;conf
Respuesta	rst;conf;{"Global":{"debug_level":"DEBUG","debug_tofile":"False","printbuffers":"False"},"Comm_port":{"port":"COM7","baudrate":"115200"},"Server":{"server_ip":"192.168.135.151","server_port":"10000","keep_alive_sec":"30","keep_alive_loop_interval_sec":"10"},"NAT":{"from_port":"40000","max_nodes":"10"},"6LOWPAN":{"local_port_groups":"30002","local_port_broadcast":"30001","local_port_unicast":"30000","advertisement_interval_sec":"5","panid":"1B58","modulation":"1","tx_power":"5","channel":"0"},"UserAccess":{"local_port":"20000"},"DoNotModify":{"buffer_size":"1024","restart_program":"start","gateway_group":"101","default_group":"1","use_security":"False","send_idt":"True","project_type":"27","project_version":"100"}}

Consulta de la IP, MAC y estadística de envío del Gateway	
Comentario	Petición de la información del Gateway: IP, MAC, Estadísticas, etc. Respuesta en JSON.
Ejemplo	get;info
Respuesta	rst;info;{"IP":"fe80 0000 0000 0000 0211 7d00 0030 5422","MAC":"00 11 7d 00 00 30 54 22","Tx_Bytes":"3954","Tx_Packets":"62","Rx_Bytes":"4393","Rx_Packets":"71","Tx_Fail":"0","Duty_Cycle":"130"}

Comandos de configuración

Enviar trama a un nodo	
Comentario	Envío de un buffer de bytes a una dirección/grupo determinado.
Ejemplo	set;node;<ip Ascii>;<port Ascii>;<Data in bytes>
Respuesta	Sin respuesta

Enviar trama a un grupo	
Comentario	Envío de un buffer de bytes a una dirección determinada.
Ejemplo	set;group;<group Ascii>;<port Ascii>;<Data in bytes>
Respuesta	Sin respuesta

Cambio de la configuración	
Comentario	Nueva configuración en formato JSON.
Ejemplo	set;conf;{"NAT":{"max_nodes":"10","from_port":"40000"},"Server":{"server_ip": "192.168.135.151","keep_alive_sec":"30","server_port":"10000", "keep_alive_loop_interval_sec":"10"},"UserAccess":{"local_port":"20000"}, "Comm_port":{"baudrate":"115200","port":"COM7"}, "6LOWPAN":{"panid":"1B58","modulation":"1","tx_power":"5","channel":"0", "local_port_groups":"30002","local_port_broadcast":"30001", "local_port_unicast":"30000","advertisement_interval_sec":"5"},"DEFAULT":{}, "Global":{"printbuffers":"False","debug_tofile":"False"}}'
Respuesta	Sin respuesta

Establecer configuración por defecto	
Comentario	Establecer configuración por defecto
Ejemplo	set;conf_def
Respuesta	Sin respuesta

Resetear el Gateway	
Comentario	Resetear el Gateway
Ejemplo	set;restart
Respuesta	Sin respuesta

7.2. Desarrollo Wifi (Espressif)

7.2.1. Módulo utilizado

El módulo seleccionado para desarrollar la tecnología Wifi será ESP32-WROOM-32D debido a que posee las siguientes características [13]:

1. Dos núcleos CPU controlados independientemente con frecuencias de reloj ajustables entre los 80 MHz y los 240 MHz.
2. Posee +19.5 dBm en la salida de la antena para garantizar una buena señal física.
3. Soporta el Bluetooth clásico para ser compatible con dispositivos más antiguos (L2CAP, SDP, GAP, SMP, AVDTP, AVCTP, A2DP (SNK) and AVRCP (CT)).
4. Soporta los perfiles de Bluetooth Low Energy (BLE) incluyendo L2CAP, GAP, GATT, SMP, y perfiles GATT como BluFi, SPP-like, etc.

5. Consumo de tan solo 5µA cuando se encuentra dormido lo que permite el desarrollo de aplicaciones de muy bajo consumo dependientes de una batería limitada.
6. Integra una memoria flash de 4MB suficiente para almacenar la información generada por el nodo.
7. Incluye periféricos como sensores de presión capacitivos, sensores HALL, amplificadores de bajo ruido, interfaz de tarjetas SD, Ethernet, SPI de alta velocidad, UART, I2S y I2C.
8. Totalmente certificado incluyendo la antena y el conjunto de librerías que proporciona. Este punto resulta muy interesante porque facilita la incorporación de dicho módulo a instalaciones que precisen de una certificación respecto a la emisión de ondas de radio.
9. Posee un precio de 3.3€ (www.digikey.es a fecha de 30/01/19).

La forma de utilizar el módulo ESP32-WROOM-32D puede ser de dos formas: mediante la programación típica utilizando las librerías que se facilitan o mediante un firmware SCI de comandos AT. Debido a que el presente proyecto pretende obtener como resultado una librería genérica de abstracción en las comunicaciones, se utilizará la segunda opción.

Como ya que ha explicado, se pretende que sea otro microcontrolador es que pueda utilizar el módulo de Espressif para comunicarse a través de Wifi (también tendría la posibilidad de comunicarse mediante BLE).

7.2.2. Librería

El módulo de Espressif corresponde con el primero desarrollado mediante comandos AT por lo que estableció la forma de implementar las librerías de comunicaciones que utilicen el mismo sistema de interacción.

Las funciones según los grandes bloques descritos serían:

Obtención de información

```
esp32_action[CMD_MOD_TEST]       = esp32_cmd_test;
esp32_action[CMD_MOD_ECHO_OFF]   = esp32_cmd_echo_off;
esp32_action[CMD_MOD_GET_INFO]   = esp32_cmd_get_info;
```

Conexión a la red

```
esp32_action[CMD_NET_CONNECT]    = esp32_cmd_net_connect;
esp32_action[CMD_NET_CLOSE]      = esp32_cmd_net_close;
esp32_action[CMD_NET_GET_IP]     = esp32_cmd_net_get_ip;
esp32_action[CMD_NET_GET_SIGNAL]= esp32_cmd_net_get_signal;
```

Conexión a un servidor

```
esp32_action[CMD_CONN_STATUS]    = esp32_cmd_conn_status;
esp32_action[CMD_CONN_CONNECT]   = esp32_cmd_conn_connect;
esp32_action[CMD_CONN_SEND]      = esp32_cmd_conn_send;
esp32_action[CMD_CONN_CLOSE]     = esp32_cmd_conn_close;
```

7.3. Desarrollo Narrow Band (Quectel)

7.3.1. Módulo utilizado

Para dotar a los diseños de una conectividad de mayor alcance, se opta por incluir un módulo de comunicaciones GSM/LTE. Para ello se opta por el módulo de Quectel BG96 [14].

Quectel Wireless Solutions es una empresa mundialmente conocida por proveer dispositivos de comunicación 5G, LTE, LTE-A, LPWA, Smart Module, C-V2X, GSM/GPRS, UMTS/HSPA(+) para comunicación de datos y GNSS para localización GPS.

Respecto al módulo seleccionado, se trata de un dispositivo que permite la comunicación mediante LTE Cat M1, Cat NB1 o EGPRS con unas tasas de transferencia máximas tanto de descarga como se subida de 375Kbps. Orientado al bajo consumo, es compatible con otros diseños (los cuales soportan otro tipo de comunicaciones) para garantizar una robustez y continuidad en los desarrollos.

Aunque se podría pensar que un módulo como tal sólo sirve para conectarse a una red y enviar o recibir datos a través de un socket, éstos están dotados de un gran número de aplicaciones extra como FTP (véase punto 8.4.1 para ver cómo se utiliza), manejo de ficheros, GPS, SSL, HTTP, PSM, PPP, MQTT, etc. abarcando la mayoría de escenarios en los que una aplicación puede necesitar una comunicación de largo alcance.

7.3.2. Librería

Tal y como se presentó en el apartado anterior, el módulo BG96 sigue con la misma temática de comandos, pero al tratarse de un módulo que proporciona o necesita mayor interacción por parte del microcontrolador maestro, se dota a este de mayores opciones.

Obtención de información

```
bg96_action[CMD_MOD_CONF]        = bg96_cmd_conf;
bg96_action[CMD_MOD_TEST]        = bg96_cmd_test;
bg96_action[CMD_MOD_GET_INFO]    = bg96_cmd_get_info;
bg96_action[CMD_MOD_ECHO_OFF]    = bg96_cmd_echo_off;
bg96_action[CMD_MOD_SIM_PRESENT]= bg96_cmd_sim_present;
bg96_action[CMD_MOD_PIN_STATUS] = bg96_cmd_sim_pin_status;
bg96_action[CMD_MOD_PIN_INSERT] = bg96_cmd_sim_pin_insert;
bg96_action[CMD_MOD_IMEI]        = bg96_cmd_mod_imei;
bg96_action[CMD_MOD_SIM_ID]      = bg96_cmd_sim_id;
```

Conexión a la red

```
bg96_action[CMD_NET_CONNECT]     = bg96_cmd_net_connect;
bg96_action[CMD_NET_GET_IP]      = bg96_cmd_net_get_ip;
bg96_action[CMD_NET_CLOSE]       = bg96_cmd_net_close;
bg96_action[CMD_NET_GET_SIGNAL] = bg96_cmd_net_get_signal;
```

Conexión a un servidor

```
bg96_action[CMD_CONN_STATUS]     = bg96_cmd_conn_status;
bg96_action[CMD_CONN_CONNECT]    = bg96_cmd_conn_connect;
bg96_action[CMD_CONN_SEND]       = bg96_cmd_conn_send;
bg96_action[CMD_CONN_CLOSE]      = bg96_cmd_conn_close;
```

GPS

```
bg96_action[CMD_GPS_START]      = bg96_cmd_gps_start;
bg96_action[CMD_GPS_UPDATE]     = bg96_cmd_gps_update;
bg96_action[CMD_GPS_CLOSE]      = bg96_cmd_gps_stop;
```

FTP

```
bg96_action[CMD_FTP_START]      = bg96_cmd_ftp_start;
bg96_action[CMD_FTP_GET_SIZE]   = bg96_cmd_ftp_get_size;
bg96_action[CMD_FTP_GET_FILE]   = bg96_cmd_ftp_get_file;
bg96_action[CMD_FTP_CLOSE]      = bg96_cmd_ftp_stop;
```

MQTT

```
bg96_action[CMD_MQTT_CONNECT]   = bg96_cmd_mqtt_start;
bg96_action[CMD_MQTT_PUB]       = bg96_cmd_mqtt_publish;
bg96_action[CMD_MQTT_SUB]       = bg96_cmd_mqtt_subscribe;
bg96_action[CMD_MQTT_UNSUB]     = bg96_cmd_mqtt_unsubscribe;
bg96_action[CMD_MQTT_CLOSE]     = bg96_cmd_mqtt_close;
```

7.4. Desarrollo Ethernet

7.4.1. Módulo utilizado

La utilización de ThreadX en los desarrollos, así como microcontroladores de la familia Synergy (series superiores a la 1) permiten acoplar librerías pertenecientes A NetX (IPv4) y NetX Duo (IPv4 / IPv6) para la utilización de la comunicación por cable o Ethernet.

No se desarrolla como un módulo independiente, sino que forma parte del microcontrolador ya que este poseerá una MAC, DMA y PTP necesario para interactuar (añadiendo el correspondiente encapsulado que aporta la capa física del Ethernet) con la red.

Al igual que 6LowPAN y el resto de dispositivos, posee sus propias librerías de interacción que se acoplan a la capa de abstracción ya diseñada.

7.4.2. Librería

Se diferenciarán 3 desarrollos importantes en este apartado:

7.4.2.1. General

Corresponde con la conexión a la red utilizando la obtención de una IP dinámicamente haciendo uso del DHCP o de forma estática, así como, el manejo de los sockets clientes o servidores correspondientes.

```
eth_action[CMD_NET_CONNECT]    = eth_cmd_net_connect;
eth_action[CMD_NET_CLOSE]      = eth_cmd_net_close;
eth_action[CMD_NET_GET_IP]     = eth_cmd_net_get_ip;
eth_action[CMD_CONN_STATUS]    = eth_cmd_conn_status;
eth_action[CMD_CONN_CONNECT]   = eth_cmd_conn_connect;
eth_action[CMD_CONN_SEND]      = eth_cmd_conn_send;
eth_action[CMD_CONN_CLOSE]     = eth_cmd_conn_close;
```

7.4.2.2. DHCP

Para dispositivos conectados a una red local, forma parte del procedimiento de comunicación el de asignar una IP válida dependiendo de si se quiere utilizar el DHCP o no. Para ello, se han implementado unas macros para obtenerla mediante el DHCP o de forma estática en caso de fallo.

```
#define SET_STATIC_IP()                                                    \
        status = nx_ip_address_set(&g_ip0, IP_STATIC_IP, IP_STATIC_MASK);  \
        if(status) PRINTF(ERROR, "Static IP Error: 0x%x", status);         \
        status = nx_ip_gateway_address_set(&g_ip0, IP_STATIC_GTW);         \
        if(status) PRINTF(ERROR, "Gateway IP Error: 0x%x", status);

#define OBTAIN_IP(USE_DHCP)                                                \
        if(USE_DHCP){                                                      \
            status = dhcp_obtain_ip(&g_dhcp_client0);                      \
            if(status){                                                    \
                PRINTF(ERROR, "%s", "Impossible to get dynamic IP, set an
static one."); \
                SET_STATIC_IP();                                           \
            }                                                              \
        }else{                                                             \
            SET_STATIC_IP()                                                \
        }                                                                  \
        dhcp_print_ip(&g_ip0);
```

La IP estática se acuerda de que será siempre la misma para la compatibilidad entre los dispositivos:

IP	192.168.1.10
Máscara	255.255.255.0
Puerta de enlace	192.168.1.1

7.4.2.3. Web Page

En la mayoría de proyecto con conexión Ethernet se demanda el poseer una página web para poder interactuar, configurar o visualizar el estado del dispositivo.

Por ese motivo, se ha implementado un servidor HTTP (utilizando NetX Duo) que satisfaga dicha necesidad. Con él se podrá:

- Visualizar cualquier página web “sencilla” desde cualquier explorador.
- Realizar peticiones al ordenador cliente.
- Responder mediante información a las peticiones de POST.
- Responder a las peticiones del JavaScript que se pueda embeber.
- Establecer contraseñar personalizadas para cada solicitud de recurso.
- Establecer funciones de manejo personalizadas para cada petición.

La librería se ha implementado de forma que pueda trasladarse a otros proyectos de forma sencilla ya que es un mismo Thread quien se encarga de todo y, como se han enunciado en los puntos anteriores, se puede personalizar sin necesidad de modificar la librería básica.

Esta librería básica es la encargada de clasificar el tipo de petición, asignarle una contraseña, si procede, y ejecutar o devolver el texto HTML en hexadecimal que corresponda.

La porción de código mostrada a continuación corresponde con la declaración de las funciones que se encargarán de manejar la petición correspondiente al servicio *cgi. A continuación se muestra un ejemplo de cómo atribuir a un recurso *.html un usuario y contraseña específicos.

```
void cgi_configTime(void * server, NX_PACKET *p);
void cgi_configCar(void * server, NX_PACKET *p);

const CGI_ENTRY cgi_data[] = {
                                { "/aux.cgi", 0},
                                { "/configTime.cgi", cgi_configTime},
                                { "/set_car.cgi", cgi_configCar},
                                { "/demo_post.asp", cgi_configCar2},
                                {0, 0}
};

const CGI_PASS cgi_pass[] = {
                                { "/formPost.html", "user", "pass"},
                                {0, 0, 0}
};
```

8. Librerías genéricas

Como el propósito del siguiente proyecto no es describir las librerías implementadas, sino dar una visión general sobre todas las posibilidades que aportarán en un futuro desarrollo, se describirán de forma simplificada algunas de las principales funciones, así como sus métodos o declaraciones.

Algunas de ellas se utilizan solo como un método de abstracción sobre la utilización de la llamada final. De esta forma se consigue un código ordenado y limpio.

8.1. HAL

8.1.1. ADC

Encargado de convertir un valor de entrada analógico a su homólogo en digital para poder ser tratado como un valor guardado en una variable.

Como norma general, su voltaje de referencia es el mismo que el del microcontrolador, esto es, 3V3; pero al tratarse este de un valor inestable debido a que la electrónica se alimenta por una batería (fuente que proporciona una corriente/hora limitada) y ante un pico de consumo, esta referencia puede verse afectada. Para solucionar esta posibilidad de error, la librería soporta la configuración de los pines de referencia del microcontrolador siendo necesario un componente intermedio (como el REF3030AIDBZT) para que estabilice la referencia a un valor conocido, el más común será el de 3V.

8.1.2. CAN

Controller Area Network forma parte del grupo de buses robustos para comunicarse con dispositivos sin la utilización de un ordenador host. Basado en el intercambio de mensajes entre máquinas, se empezó a utilizar en el ámbito de la automoción, pero cada vez se puede encontrar en muchos más escenarios.

Dado que el protocolo CAN es necesario para la mayoría de proyectos, se ha implementado la librería que hace uso del mismo para facilitar su uso.

Como otros buses de comunicación cuenta con distintas velocidades de uso, la siguiente definición enumera las configuradas para que el driver ajuste los relojes a dichas:

```
typedef enum {
    CAN_BAUD_50_KHZ,
    CAN_BAUD_125_KHZ,
    CAN_BAUD_250_KHZ,
    CAN_BAUD_500_KHZ,
    CAN_BAUD_1000_KHZ
}can_baud_e;
```

Siendo las funciones principales de la librería:

```
void hal_can_init(can_instance_t * can_var, can_baud_e baudrate,
hal_can_read_func func);
void hal_can_close(can_instance_t * can_var);
void hal_can_write(can_instance_t * can_var, can_msg * msg);
void hal_can_write_buf(can_instance_t * can_var, uint32_t can_id, uint8_t
*frame, uint16_t data_length);
void hal_can_change_baudrate(can_instance_t * can_var, can_baud_e baudrate);
```

También mencionar que los mensajes recibidos son enviados directamente a la función que se pasa como parámetro en la función de inicio.

8.1.3. CRC

Abreviatura de *Cyclic Redundancy Check*, proporciona las definiciones y valores necesarios para calcular un CRC32 sobre una cantidad limitada de datos almacenados en memoria.

Un ejemplo de utilización de las misma sería:

```
HAL_CRC_INIT();
Var crc32 = HAL_CRC_INIT_VAL;
for each byte in data do:
 HAL_CRC_CALCULATE(crc32, data, 1)
crc32 = crc32 XOR HAL_CRC_FINAL_XOR_VAL;
```

Las funciones del CRC se utilizan principalmente cuando se descargan ficheros, se quiere comprobar que se posee una configuración válida (cuando el programa arranca por primera vez se debería de detectar dicho caso y cargar los valores por defecto) o simplemente por obtener un pequeño “resumen” sobre un buffer de memoria.

Se podría pensar que el cálculo del CRC se realiza mediante una función, pero es el mismo hardware quien lo hace; proporcionando una mayor velocidad de cálculo.

8.1.4. Flash

Mediante la utilización del siguiente módulo, un programa, ya sea de aplicación o de bootloader, tendrá acceso de lectura y escritura sobre cualquier dirección de memoria flash del microcontrolador. Cabe destacar que, si no se indica lo contrario en el linker, en tiempo de compilación, cualquier programa tendrá acceso a las direcciones pertenecientes a la Flash de Datos (dirección 0x40100000 [15] en adelante y dependiente de la versión del micro), permiso de lectura sobre cualquier dirección de la Flash de Código, pero solo de escritura sobre la partición que no se esté ejecutando es ese momento.

Como se menciona en el párrafo anterior, se pueden añadir, eliminar o limitar el acceso a las distintas particiones de memoria que existen en el microcontrolador.

Un ejemplo de distribución de memoria utilizada por el bootloader puede ser la siguiente:

```
/* Linker script to configure memory regions. */
MEMORY
{
  FLASH (rx)          : ORIGIN = 0x00000000, LENGTH = 0x008500  /*  34K  */
  RAM (rwx)           : ORIGIN = 0x20000000, LENGTH = 0x005FFF  /*  24K */
  DATA_FLASH (rx)     : ORIGIN = 0x40100000, LENGTH = 0x0001000 /*   4K */
}
```

Mientras que la división de memoria configurada para una aplicación de usuario debe de ser totalmente compatible con la configurada anteriormente:

```
/* Linker script to configure memory regions. */
MEMORY
{
  FLASH (rx)          : ORIGIN = 0x00008500, LENGTH = 0x031500  /* 222K */
  E2S_TRACE_BUF (rw)  : ORIGIN = 0x20000000, LENGTH = 0x0003FF  /*   1K */
  RAM (rwx)           : ORIGIN = 0x20000400, LENGTH = 0x005C00  /*  23K */
  DATA_FLASH (rx)     : ORIGIN = 0x40100000, LENGTH = 0x0001000 /*   4K */

  ID_CODE_1 (rx)      : ORIGIN = 0x01010018, LENGTH = 0x04      /* 4 bytes */
  ID_CODE_2 (rx)      : ORIGIN = 0x01010020, LENGTH = 0x04      /* 4 bytes */
  ID_CODE_3 (rx)      : ORIGIN = 0x01010028, LENGTH = 0x04      /* 4 bytes */
  ID_CODE_4 (rx)      : ORIGIN = 0x01010030, LENGTH = 0x04      /* 4 bytes */
}
```

En ella se definen 2 particiones en la parte de Flash de Código, una para el bootloader y otra para la aplicación de usuario. No hay que olvidar que siempre se debe poner toda la memoria que se quiera utilizar, ya que, si no se hace, ésta no podrá ser utilizada por la aplicación y el comportamiento será impredecible o llevará a error.

El módulo de Flash se utiliza tanto en la parte de bootloader como en la de aplicación. Para conocer más acerca de este módulo habrá que ir al punto 8.4.

8.1.5. I2C

Uno de los buses más utilizados en la mayoría de aplicaciones es I2C o Inter-Integrated Circuit para poder comunicarse con dispositivos que se encuentran a una distancia corta del microcontrolador. Las funciones que hacen esto posible son:

```
void hal_i2c_init(void);
void hal_i2c_close(void);
uint8_t hal_i2c_write(uint8_t i2c_addr, uint8_t reg_addr, uint8_t *reg_data,
uint16_t length);
uint8_t hal_i2c_read(uint8_t i2c_addr, uint8_t reg_addr, uint8_t *reg_data,
uint16_t length);
```

Como se observa, el acceso al chip esclavo se hace de forma básica y debe ser el mismo programa el que implemente las librerías de interacción. La librería fue probada mediante la interacción del sensor de temperatura y presión BMP280.

8.1.6. IO

Corresponde con el acceso de lectura y escritura sobre los pines configurados como salida del microcontrolador.

8.1.7. IRQ

Permite la inicialización de una interrupción hardware sobre una serie de pines, aquellos que permitan dicha funcionalidad.

8.1.8. Low_power

Aunque el módulo de baja energía sólo posea una única función, ésta es la encargada de llevar al microcontrolador a su estado de energía más bajo, haciendo posible que la duración de la batería se alargue hasta varios meses o años.

La librería permite salir del modo de bajo consumo o “despertar” mediante una interrupción externa o una alarma periódica o eventual (configurada mediante la utilización del RTC).

Los pasos para llevar a consumir lo mínimo posible son:

1. Inicialización del módulo.
2. Establecer la configuración en bajo consumo. En ella se establece qué hará salir de dicho estado al módulo.
3. Inicialización y configuración del RTC en modo alarma. En este ejemplo se ha optado por la siguiente opción.
4. Deshabilitar todas las interrupciones para proceder con una configuración mucho más exhaustiva.
5. Se establece el principal reloj como el SUBCLOCK (según el datasheet es el de menor consumo) y se deshabilita el resto.
6. Parar todos los periféricos.
7. Configuración de los pines para que posean el menor consumo. Dicha configuración es la que se le pasa a la función como parámetro ya que depende del diseño electrónico de cada proyecto.
8. Selección del modo principal de baja velocidad.
9. Selección del submodo como el del subosciloscópio.
10. Parada de la memoria Flash, tanto de datos como de Código.
11. Deshabilitación del debugger en caso de que se estuviera utilizando.
12. Entrada en bajo consumo hasta que salte la interrupción configurada.

La imagen de abajo muestra como los microcontroladores de la familia Renesas Synergy poseen diferentes modos de funcionamiento para poder abarcar todos los escenarios posibles.

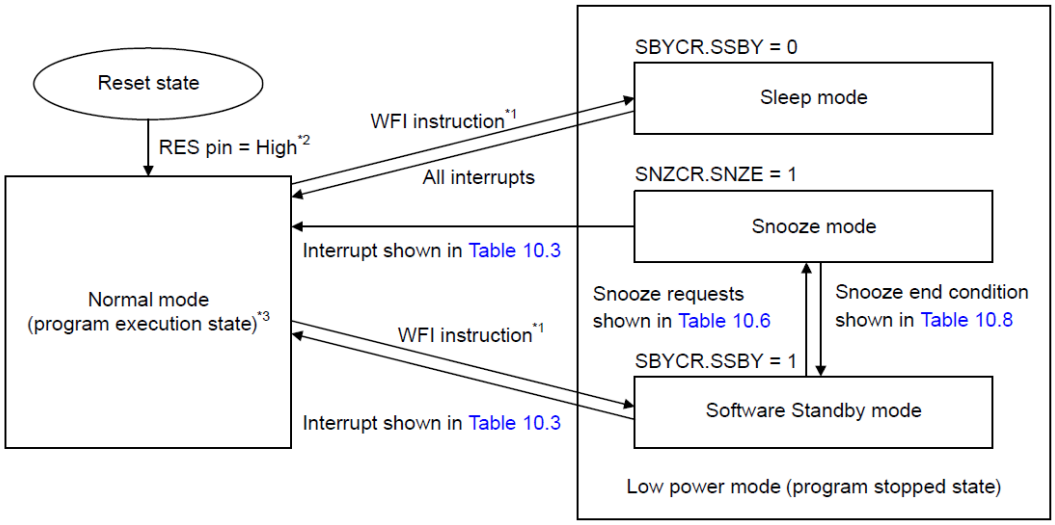


Imagen 14: Modos de bajo consumo [15]

8.1.9. PWM

Utilización del hardware para implementar un Pulse Width Modulation sobre el pin que lo soporte. La librería posee exclusivamente 2 definiciones:

```
#define PWM_START(P)
#define PWM_SET_DUTY(P, VAL)
```

Con las cuales podemos inicializar y establecer un valor de entre 0 y 100%. Interiormente, los drivers junto con el hardware diseñado se encargarán de establecer el reloj, la frecuencia del mismo; así como la frecuencia de activación para alcanzar ese % de brillo configurado.

8.1.10. RTC

Dado que la mayoría de proyectos que se diseñan actualmente tienen la necesidad de mantener la hora actual aun cuando el microcontrolador se encuentre desconectado de la fuente de alimentación o en estado de bajo consumo absoluto, será necesario un *Real-time clock* o RTC.

La librería implementada define las funciones principales:

```
bool hal_rtc_init(void);

bool hal_rtc_set_calendar(rtc_time_t *t);
bool hal_rtc_get_calendar(rtc_time_t *t);
```

Por las cuales, una aplicación puede: inicializar el módulo, establecer y obtener la hora actual.

Por compatibilidad con aplicaciones pasadas, se ha optado por mantener la estructura *rtc_time_t* ya que se define como una estructura *tm*, tal y como se define en la librería genérica de UNIX, *time.h*.

Como complemento a la funcionalidad ya descrita, se implementa en la misma librería la posibilidad de generar interrupciones periódicas o la de establecer una alarma para tiempos mucho más largos.

```
bool hal_rtc_periodic_IRQ_state(bool b);
bool hal_rtc_periodic_IRQ_rate_change(rtc_periodic_irq_select_t rate);

bool hal_rtc_set_alarm_time(rtc_alarm_time_t *alarm);
bool hal_rtc_get_alarm_time(rtc_alarm_time_t *alarm);
```

Las interrupciones periódicas aceptan los siguientes valores ya que vienen limitados por el hardware.

```
/** Periodic Interrupt select */
typedef enum e_rtc_periodic_irq_select
{
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_256_SECOND = 6,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_128_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_64_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_32_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_16_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_8_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_4_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_DIV_BY_2_SECOND,
    RTC_PERIODIC_IRQ_SELECT_1_SECOND,
    RTC_PERIODIC_IRQ_SELECT_2_SECONDS,
} rtc_periodic_irq_select_t;
```

Siendo un máximo de 2 segundos el tiempo permitido ante una interrupción periódica del RTC.

Cuando se deseen tiempo más largos, se deberá de recurrir a la función que establece una alarma; pudiendo conseguir cualquier tiempo deseado, pero con mucha menos precisión.

```
/** Alarm time setting structure */
typedef struct st_rtc_alarm_time
{
    rtc_time_t  time;            ///< Time structure
    bool        sec_match;       ///< Enable the alarm based on a match of the
seconds field
    bool        min_match;       ///< Enable the alarm based on a match of the
minutes field
    bool        hour_match;      ///< Enable the alarm based on a match of the
hours field
    bool        mday_match;      ///< Enable the alarm based on a match of the
days field
    bool        mon_match;       ///< Enable the alarm based on a match of the
months field
    bool        year_match;      ///< Enable the alarm based on a match of the
years field
    bool        dayofweek_match; ///< Enable the alarm based on a match of the
dayofweek field
} rtc_alarm_time_t;
```

Los pasos para utilizar dicha funcionalidad serían, la unión entre establecer un tiempo (mediante la configuración de la fecha deseada en la variable *time*) y cada cuánto se tiene que repetir. Esta última opción nos permitiría no tener que configurar una alarma cada cierto tiempo, si lo que se desea es una ejecución cada hora, día, mes o año.

Configurando las variables X_match, la alarma se disparará siempre que se cumpla la condición, como se muestra en el ejemplo (se utilizará exclusivamente la hora, minutos y segundos):

Hora actual	16:05:44	16:05:44	16:05:44	16:05:44
Hora alarma	16:05:12	16:05:12	16:05:12	16:05:12
Match	00:01:00	00:00:01	01:01:00	01:01:01
Resultado	Alarma	No Alarma	Alarma	No Alarma

Dado que puede darse el caso que una aplicación necesite cambiar en un momento dado la función que trata la interrupción, tanto periódica como de alarma, se ha puesto una capa por encima para proporcionar dicha funcionalidad. Para ello, se define un tipo genérico de función:

```
typedef void (* hal_rtc_irq_func)(void);
```

y cuyas funciones para cambiar las callbacks de ambas interrupciones serán:

```
void hal_rtc_set_periodic_IRQ_callback(hal_rtc_irq_func func);
void hal_rtc_set_alarm_IRQ_callback(hal_rtc_irq_func func);
```

8.1.11. Semaphore

Como se explicó en el punto 4, la utilización se threads puede provocar problemas como las condiciones de carrera, las cuales provocarían una corrupción en los datos.

Para evitarlo, ThreadX implementa la utilización tanto de *semaphores* como de *mutex*. La única diferencia entre ambos es que un semáforo permitiría la ejecución de dicho código a un número finito de hilos, mientras que un *mutex* solo dejaría pasar a uno solo; este último sería una especie de semáforo binario [16].

Para facilitar las cosas y reducir el código visualizado en los programas, se definen las siguientes utilidades para controlar tanto los semáforos como los *mutex*:

```
// Semaphore creation with TX_SEMAPHORE
#define SEM_INIT(S)                                                          \
        UINT err;                                                            \
        err = tx_semaphore_create(&S, (char *) "Connection semaphore", 0);   \
        if (TX_SUCCESS != err)                                               \
        {                                                                    \
            error_trap(err, "Impossible to start semaphore");                \
        }


#define SEM_ACQUIRE(S)  tx_semaphore_get(&S, TX_WAIT_FOREVER);
#define SEM_RELEASE(S)  tx_semaphore_put(&S);

// Mutex creation with TX_MUTEX
#define MUTEX_INIT(S)                                                        \
        UINT err;                                                            \
        err = tx_mutex_create(&S, (char *) "Connection mutex", 0);           \
        if (TX_SUCCESS != err)                                               \
        {                                                                    \
            error_trap(err, "Impossible to start mutex");                    \
        }

#define MUTEX_ACQUIRE(S)  tx_mutex_get(&S, TX_WAIT_FOREVER);
#define MUTEX_RELEASE(S)  tx_mutex_put(&S);
```

8.1.12. Software_timer

Todos los microcontroladores actuales proporcionan métodos para ejecutar una función periódicamente cada cierto tiempo X, estos son los llamados *timers*.

Pros	*Contras*
Precisos	Limitados en número
A máxima velocidad	Limitados en tiempo máximo
Configuración rápida	A veces no se pueden configurar a mitad de programa
Utilizan exclusivamente Hardware	No son muy dinámicos
Interrupciones Hardware	

Dado que una aplicación completa puede llegar a utilizar un número importante de timers, la mayoría de veces superior a los disponibles por hardware, se diseña un sistema para poder trabajar con tantos como se quiera.

Basado en un único timer hardware de mínimo intervalo (configurable dependiendo de la precisión demandada), es el encargado de iterar sobre todos los timer software que se encuentran activos en ese momento.

Un timer software viene definido como:

```
typedef struct timer_struct{
    bool enable;
    uint32_t counter;
    uint32_t interval;
    hal_timer_cb cb;
    timer_repetitions rep;
}timer_struct;
```

En dónde se posee una marca (*enable*) para saber si el contador está activo en ese momento, una variable *counter* debe mantener el tiempo que falta para que caduque, un *interval* que almacena el valor original del contado para restablecerlo, una *callback* la cual almacena la función que se debe de ejecutar en caso de que caduque el timer software y una variable *rep* para indicar el número de veces que se debe de repetir el timer (0 siempre, 1, 2, 3…).

```
typedef void (* hal_timer_cb) (timer_id t);
```

La definición de arriba establece el tipo de función que se ejecuta cuando un timer caduca y, por lo tanto, su contador *counter* llega a 0.

El motivo por el cual la función a ejecutar espera un parámetro es porque puede ser que una misma función sea el tratamiento de dos timers con distinta identificación. El parámetro pasado será el ID del mismo para que se pueda identificar en la función destino.

Como cualquier librería de propósito general, se posee funciones para inicializar las variables y estructuras internas, así como su cierre:

```
void hal_timer_init(void);
void hal_timer_stop(void);
void hal_timer_loop(void);
```

Cuando la interrupción del timer hardware decrementa el contador de un timer software, haciendo que este llegue a 0, se marca dicho timer para su ejecución. El motivo por el que no se ejecuta instantáneamente es porque, generalmente, la utilización de este tipo de

estrategias viene acompañado por la ejecución de funciones pesadas que provocarían un retraso entre ellas y en todo el programa (en los tratamientos de interrupción, todas las interrupciones del microcontrolador se deshabilitan para no causar problemas entre ellas).

Para solucionar esta casuística, se debe ejecutar periódicamente la función *hal_timer_loop* en un Thread dedicado en la iteración principal del programa, la cual sí que terminará de ejecutar los timers que se deban. Mencionar también que esta misma tarea es la encargada de reactivarlos si prodece.

Las funciones para añadir y eliminar los timers software son:

```
void hal_timer_add(timer_id id, uint32_t interval_ms, hal_timer_cb cb,
timer_repetitions repetition_times);
void hal_timer_remove(timer_id id);
void hal_timer_remove_all(void);
```

Como se ha mencionado al principio de este apartado, la librería está pensada para almacenar tantos timers como se necesiten por la aplicación. El siguiente ejemplo es la definición de todos los timers utilizados en la aplicación de ejemplo mostrada en el punto 10.

```
#define SOFT_TIMER_CB           software_timer_callback
#define SOFT_TIMER_VAR          g_software_timer
#define SOFT_TIMER_PERIOD_MS    10
typedef uint8_t  timer_id;

enum {
    TIMER_EVERY_1_S = (timer_id) 0,
    TIMER_AT_CMD_TIMEOUT_ESP32,
    TIMER_AT_CMD_TIMEOUT_BG96,
    TIMER_AT_BUFFER_DELAY_ESP32,
    TIMER_AT_BUFFER_DELAY_BG96,
    TIMER_AT_CMD_SEND,
    TIMER_CHECK_CONNECTION_STATUS,
    TIMER_BG96_CONF_PROCESS,
    TIMER_GPS_UPDATE,
    TIMER_MQTT_CMD_TIMEOUT,
    TIMER_MQTT_PINGREQ,
    TIMER_ZMDI_CONF_ALIVE,
    TIMER_ZMDI_BLINK_LED,
    TIMER_ZMDI_NODE_CONNECTED,
    TIMER_ZMDI_NODE_TO_CONFIGURE,

    MAX_TIMER_COUNTER
};
```

8.1.13. Uart

Corresponde con las siglas de *Universal Asynchronous Receiver-Transmitter* y es uno de los buses de comunicaciones de campo cercano más utilizado para comunicarse con dispositivos ajenos al microcontrolador. Algunos de los módulos que se utilizan en este proyecto para implementar la parte de comunicaciones utilizan un bus UART para ser comunicados por el módulo *master*. Es por este motivo por el que la librería de UART corresponde con una de las más importantes.

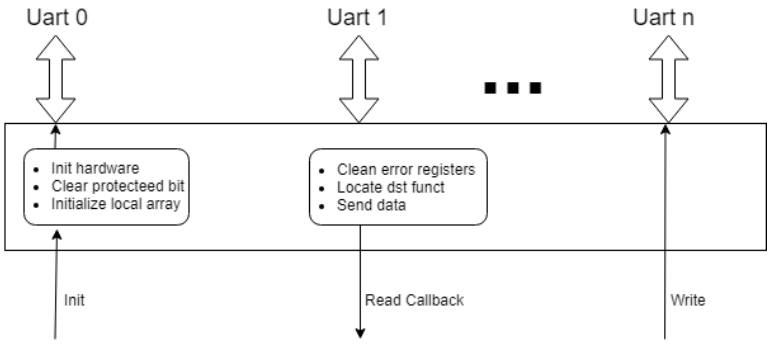


Imagen 15: Esquema librería UART

Durante el desarrollo del proyecto, el presente módulo ha variado según iban surgiendo problemas y necesidades, pero siempre manteniendo la esencia de una librería fácil de usar; podemos observarlo mediante las funciones que esta implementa:

```
void hal_uart_init(uart_instance_t *u, hal_uart_read_func func);
void hal_uart_close(uart_instance_t *u);
void hal_uart_write(uart_instance_t *u, void *buf, uint16_t len);
```

Simplemente se necesitará como elemento común una instancia de la variable que mantiene toda la información que hace referencia a la UART, como el *baudrate* y los pines empleados asociados al canal utilizado.

La Imagen 15 muestra un esquema simplificado de cómo una librería ajena puede utilizar la UART del microcontrolador sin necesidad de hacer demasiadas operaciones. Mencionar que la operación de escritura sobre el bus deseado se hace de forma directa sin necesidad de pasar por buffers intermedios que empeoraría y ralentizaría el sistema.

La función *hal_uart_init*, o de inicialización, es la encargada de configurar el driver que implementa el acceso a los recursos del microcontrolador (parte abstraída por Renesas) y configurar las estructuras internas para localizar la dirección del registro asociado al canal de UART configurado; así como establecer la función de tipo:

```
typedef void (* hal_uart_read_func)(uint8_t channel, uint8_t data);
```

la cual se encargará de recibir directamente mediante una interrupción del programa los bytes recibidos. Además, como el driver está pensado para trabajar utilizando *multithreading*, se ha implementado un bit por cada canal utilizado para garantizar que una llamada a *hal_uart_write* no interfiere con otra, sin necesidad de desactivar el cambio de contexto en el ThreadX.

La recepción de los bytes se hace mediante una interrupción de mayor prioridad para garantizar la mayor velocidad posible teniendo como objetivo 3 cosas:

- Localización de la función destino del byte entrante.
- Liberar la función de escritura cuando se haya finalizado de transmitir los datos de la anterior.
- Limpiar los registros de ORER (Overrun Error Flag), PER (Parity Error Flag) y FER (Framing Error Flag) debido a que un error en estos bits dejaba sin recibir al módulo. Véase el manual del S128 [15] punto 27.3.9.

La razón por la que se decide posponer el tratamiento de los datos a las librerías de mayor nivel es porque, por ejemplo, en los módulos de comandos AT puede dividirse la información separándola por cada '/n' carácter (punto 6.2.2) que se reciba, mientras que en el módulo

ZMDI, se necesita mantener una máquina de estados para asegurarse de que se recibe bien el paquete (punto 7.1.3).

8.1.14. USB

De una forma menos desarrollada hasta el momento y como mera exploración de las opciones que aporta, se implementó una sencilla librería para interactuar con un USB del tipo CDC (Communications Device Class) ACM (Abstract Control Model).

```
void hal_usb_init(void);
uint32_t hal_usb_write(uint8_t * buf, uint32_t len);
uint32_t hal_usb_read(uint8_t * buf); //Blocking function
```

De una forma práctica, se quería una pequeña porción de código para validar y comprobar las prestaciones que aportaba la utilización de dicho módulo a un futuro desarrollo.

Finalmente se excluyó del proyecto debido a que la librería base sobre la que se implementa posee la utilización de, al menos, 8KB de memoria RAM sobre los 24 KB que posee el modelo S128 utilizado en este proyecto.

Además, la utilización de un Thread exclusivo para el módulo de USB (debido a que posee funciones bloqueantes) suponía un alto coste también en memoria para el desarrollo final, por lo que se acabó descartando.

8.1.15. Utils

Como bien indica el nombre, la librería corresponde a una serie de funcionalidades genéricas que hacen del desarrollo del proyecto un entorno más cómodo, amigable y mantenible.

Dado que un proyecto puede no utilizar *multithreading* para implementar una solución, se define la variable para realizar esperas tanto con la utilización de ThreadX como si no:

```
#ifdef USE_THREADX
    #define SLEEP_MS(X)                                                    \
    if(X < 10) R_BSP_SoftwareDelay (X, BSP_DELAY_UNITS_MILLISECONDS);    \
    else tx_thread_sleep((uint16_t)(X/10));
#else
    #define SLEEP_MS(X)     R_BSP_SoftwareDelay (X,
BSP_DELAY_UNITS_MILLISECONDS);
#endif
```

Pero principalmente se quiere mostrar la utilización de la función:

```
void error_trap(ssp_err_t err, char *msg);
```

ya que ésta se ejecuta siempre que existe un error en el proyecto, o algunas de sus librerías más básicas entra en un estado de error. En ella se puede tratar el problema de forma genérica o simplemente reiniciando el sistema.

La otra funcionalidad que se implementa corresponde con la posibilidad de mostrar mensajes de texto por cualquiera de las interfaces que se posean en el proyecto (USB, UART, Debug Console, socket, etc.) pudiendo establecer distintos niveles de importancia en los mensajes.

```
#define PRINTF(LEV, FORMAT, ...)printfd(LEV, FILE_NAME, FORMAT, __VA_ARGS__)
#define PRINT(FORMAT, ...)      printfd(TRACE, FILE_NAME, FORMAT, __VA_ARGS__)
#define PRINT_BUF(B, L, HEX)    printbuf(DEBUG, FILE_NAME, (uint8_t *)B, L,
HEX)
```

Siendo estos niveles:

```
typedef enum{
    TRACE = (uint8_t)0x00,
    DEBUG,
    INFO,
    ERROR
}DEBUG_LEV;

extern DEBUG_LEV debugLevel;
```

8.2. Otras

Una de las herramientas comunes en implementaciones firmware es la de almacenamiento de datos de forma temporal y ordenada principalmente utilizados en el ámbito de las comunicaciones.

8.2.1. Static Buffer FIFO

La primera solución de describe como una implementación FIFO de una porción de memoria dónde, tal y como indica las mismas siglas, el primer byte que entra es el primero en salir.

```
typedef struct{
    uint8_t buf[FIFO_QUEUE_LENGTH];

    uint8_t *head;
    uint8_t *tail;

    FIFO_QUEUE_LENGTH_TYPE size;
}Fifo_queue_t;

void fifo_queue_init(Fifo_queue_t *q);

FIFO_QUEUE_LENGTH_TYPE fifo_queue_add_single(Fifo_queue_t *q, uint8_t data);
FIFO_QUEUE_LENGTH_TYPE fifo_queue_add(Fifo_queue_t *q, uint8_t *b,
FIFO_QUEUE_LENGTH_TYPE len);
FIFO_QUEUE_LENGTH_TYPE fifo_queue_get(Fifo_queue_t *q, uint8_t *b,
FIFO_QUEUE_LENGTH_TYPE len);
FIFO_QUEUE_LENGTH_TYPE fifo_queue_get_until(Fifo_queue_t *q, uint8_t until,
uint8_t counter, uint8_t *b);
FIFO_QUEUE_LENGTH_TYPE fifo_queue_get_fifos(Fifo_queue_t *src, Fifo_queue_t
*dst, FIFO_QUEUE_LENGTH_TYPE len);
```

Esta librería correspondería con la solución más fácil de utilizar y que más opciones aporta. Declara su array de memoria de forma estática junto con la estructura de control y tiene la limitación de que todas las instancias de la misma deben tener el mismo tamaño.

Aunque permite la extracción de forma atómica, se está convirtiendo en una librería anticuada por la limitación comentada.

8.2.2. Dynamic Buffer FIFO

Correspondería con la nueva versión de la cola comentada en el anterior punto. Aunque de momento no posee todas las funcionalidades, se va a ir actualizando hasta tener un conjunto de macros y funciones *inline* que permitan la migración completa a este tipo de librerías, ya que nos permite declarar para cada cola, un tamaño fijo diferente.

```
typedef struct{
    uint8_t *buf;
    FIFO_EMPTY_QUEUE_LENGTH_TYPE max_buf_len;

    uint8_t *head;
    uint8_t *tail;

    FIFO_EMPTY_QUEUE_LENGTH_TYPE size;
}Fifo_empty_queue_t;

#define FIFO_EMPTY_QUEUE_INIT(Q, BUF, LEN)
#define FIFO_EMPTY_QUEUE_ADD_SINGLE(Q, DATA)
#define FIFO_EMPTY_QUEUE_GET_ALL(Q, BUF, LEN
#define FIFO_EMPTY_QUEUE_GET_SINGLE(Q, DATA, LEN)
```

8.2.3. Struct FIFO

Como funcionalidad extra, se ha diseñado una cola FIFO de memoria que funcione con estructuras mucho más compleja, un posible ejemplo son las estructuras en C.

```
typedef struct{
    void * buf;
    void * head;
    void * tail;
    void * last;
    uint8_t size;
    uint8_t max_size;
    uint16_t var_size;
}fifo_struct_t;

#define FIFO_STRUCT_INIT(FIFO, BUF, VAR, MAX_SIZE)
#define FIFO_STRUCT_GET(FIFO, VAR)
#define FIFO_STRUCT_REMOVE(FIFO)
#define FIFO_STRUCT_ADD(FIFO, VAR)
```

8.2.4. JSON

Una forma cómoda de trabajar e intuitiva para el ser humano es la de organizar los datos a recibir/transmitir mediante un formato *JavaScript Object Notation* (JSON). Algunas de las problemáticas que introduce emplear dicho formato de datos es la cantidad de bytes que se tienen que utilizar solo para mantener la estructura, o la inmensa cantidad de funciones y de memoria empleada por las librerías de parseado.

Para mantener la esencia de “lo simple” sin perder funcionalidad, y enfocando la utilización de una librería, se ha buscado una que cumpla con los objetivos del proyecto. La elegida es jsmn [17] por su robustez (funciona incluso con datos erróneos), rapidez (parseado de los datos *on the fly*), ser portable (sin dependencias de la versión del compilador) y sencilla (no posee estructuras o métodos complejos).

Por comodidad de la librería se ha optado por añadir una capa superior para añadir funcionalidad y hacer las cosas más sencillas, los únicos métodos que se implementan son:

```
int json_parse(json_parser *parser, char *json, uint16_t json_len, json_token
*tokens, uint16_t tokens_len);
bool json_find(char * text, char *json, json_token *tokens, uint16_t
tokens_len, char ** value, uint16_t * value_len);
```

8.3. External memory

Aprovechando el desarrollo de un proyecto paralelo al actual, se tenía la necesidad de almacenar unas mediciones de sensores de forma permanente por lo que se diseñó un hardware con una memoria flash externa.

La memoria seleccionada para solucionar dicha necesidad fue una MQT25Q NOR Flash de 32MB de la empresa Micron con un bus SPI para comunicarse con ella.

Dado que la utilización de memorias externas para almacenaje de datos tanto flash como eeprom o MRAM es bastante común en los desarrollos de aplicaciones, se ha decidido utilizar un método de abstracción de las funciones más básicas:

```
void ext_flash_init(void);
void ext_flash_close(void);
void ext_flash_erase_subSector(uint32_t subsector);
void ext_flash_erase_Sector(uint32_t sector);
void ext_flash_erase_Sector_sec(uint32_t sector, uint16_t num);
void ext_flash_write(uint32_t address, uint8_t *buf, uint32_t len);
void ext_flash_read(uint32_t address, uint8_t *buf, uint32_t len);
```

Micron facilita una librería de acceso a la mayoría de sus memorias NOR Flash, pero ésta debe de adecuarse para mejorar el rendimiento de la misma en tema de espera de tiempos, acceso al bus de comunicaciones, etc.

Por lo que, tras entender el funcionamiento de la misma, se proporciona una función para detectar qué memoria Micron se está utilizando es ese momento. Esto es posible a que la mayoría de memorias comerciales de las empresas tienen los registros básicos mapeados en las mismas posiciones para que el cambio entre ellas sea lo más intuitivo posible.

La organización encargada de dicha estandarización es la JEDEC Solid State Technology Association que determinó que el acceso a la información básica de la memoria debe de ser la misma para todas las memorias, en ella está el tamaño de la misma, configuración byte y Word, configuración de los bloques, voltaje de funcionamiento y tiempos de respuesta [18].

Dado que cada tecnología posee unas limitaciones, la librería debe de abstraer al usuario final de las mismas; algunas de ellas son:

- Las memorias NOR Flash deben de escribir grandes bloques de memoria.
- Las memorias NOR Flash son más rápidas de leer que de escribir en contra de la NAND Flash.
- Las memorias NOR Flash tienen suficiente capacidad para acceder a una dirección en concreto mientras que las NAND no.
- Diferentes tiempos de lectura y escritura.

Para ello, debe de quedar perfectamente documentado las características de cada memoria:

```
/*
 * Size: 32MB
 * Sector erase: 64KB - 512 sectors
 * Subsector erase: 4KB - 8112 subsectors
 */

#define SECTOR_SIZE                 (0x10000)
#define SUBSECTOR_SIZE              (0x1000)
#define GET_ADDRESS_FROM_SECTOR(X)  (X * SECTOR_SIZE)
```

Pero no sólo se deben declarar las limitaciones de cada uno, sino la distribución de memoria, ya que es común que una misma memoria se utilice para distintas aplicaciones:

- Almacenamiento fichero actualización.
- Memoria circular de datos recogidos.
- Almacenamiento de la configuración actual y por defecto.
- Certificados SSL/TLS.
- Páginas web de gran peso para evitar que ocupen memoria del microcontrolador.

8.4. Bootloader

Muchos de los productos que se compran actualmente poseen fallos conocidos o desconocidos, mejoras que se plantean posteriormente o problemas de seguridad; este tipo de problemas afecta tanto a productos mundialmente comerciales como a los de las pequeñas empresas que diseñan productos bajo demanda; al no estar libre de errores, toda aplicación debe de poder actualizarse de un modo u otro.

Por comodidad, sería conectar un cable USB o conector pero la mayoría de veces, la persona que adquiere el producto no posee los conocimientos necesarios o la imagen que se quiere dar no corresponde. La otra opción, es que el producto viaje de vuelta al servicio técnico o sea un técnico el que se desplace al lugar; esto supone un sobrecoste de mantenimiento que puede salir mucho más caro que unas horas de ingeniería iniciales para implementar un método que proporcione al dispositivo actualizarse remotamente. Para ello, el dispositivo debe de tener la posibilidad de conectarse a Internet (o una red local) para poder descargarse la nueva versión del producto.

Para conseguirlo, se necesita de un programa inicial llamado *bootloader*; un pequeño SO, aplicación o programa firmware embebido en el dispositivo (memoria del microcontrolador o memoria externa) que permite la actualización del programa que ejecuta la funcionalidad final del producto. Para ello, éste debe de ejecutarse antes que cualquier instrucción del programa final ya que puede que se haya quedado corrupto por algún motivo. El *bootloader* debe de ser un programa totalmente fiable y sin errores, puesto que la única forma de corregir un programa es la de reprogramar y no puede llevarse a cabo a menos de que se tenga un programador JTAG o dispositivos relacionados con el entorno de desarrollo.

Volviendo al desarrollo del proyecto, las aplicaciones que se implementen con las librerías descritas deben de poder ser actualizados mediante el *bootloader*. Para ello se ha implementado un proyecto paralelo que tiene la finalidad de actualizar el programa de usuario si existe una nueva actualización.

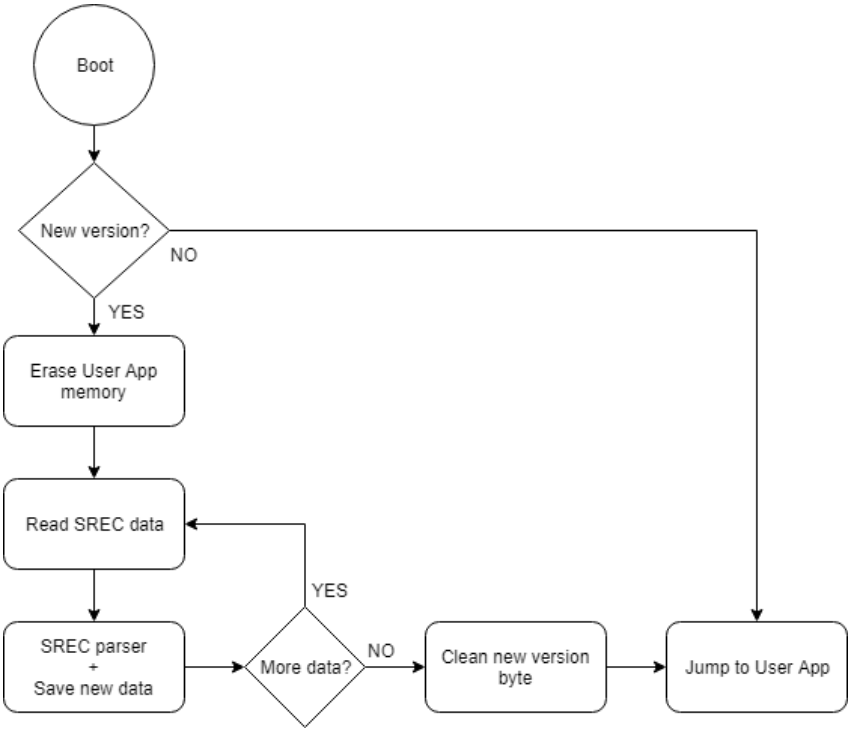


Imagen 16: Diagrama proceso bootloader

El desarrollo de un proyecto de bootloader tiene la complejidad añadida de que debe de coexistir con el de la aplicación, pero siendo este transparente para la misma. Si volvemos al punto 8.1.4, nos damos cuenta de que la aplicación de usuario almacenada en la flash a partir de la instrucción 0x00008500 pero se sabe que todos los microcontroladores ejecutan como primera instrucción la dirección 0x00000000 por lo que nos damos cuenta de que entre la dirección de inicio 0 y la 0x8500 es el espacio reservado para el bootloader.

Por lo que tendremos que tener en cuenta los descritos en el punto 8.1.4 para entender el funcionamiento del bootloader.

La memoria interna del micro se comporta como si de una externa se tratase, teniendo las mismas limitaciones a la hora de escribir datos (antes de escribir se debe de eliminar todo el bloque), por lo que se debe de parametrizar sus limitaciones:

```
#define IMAGE_MAX_SIZE              (0x040000)      // Total flash size of the
S128 microcontroller.
#define IMAGE_APP_ADDRESS_RUN       (0x008500)      // Start address of the
user app in flash.
#define IMAGE_APP_SIZE_BYTES        (0x031500)      // Image size in bytes.
Used for erasing.

#define FLASH_WRITE_BLOCK_SIZE      (256)           // S1 Number of bytes to
be flashed at a time. From hardware specifications (S1 can go as low as 4) .
#define FLASH_ERASE_BLOCK_SIZE      (0x400)         // S1 Number of bytes in
erase block. From hardware specifications

#define DATAFLASH_FLAG_ADDRESS      (0x40100000)    // Start of data flash.
Can be moved elsewhere in data flash.
#define DATAFLASH_WRITE_MIN_SIZE    (1)             // Number of bytes in data
flash minimum size write block
#define FLASH_FILL                  (0xFF)          // Value to write when no
data present.
#define IMAGE_APP_SIZE_NUM_BLOCKS   (IMAGE_APP_SIZE_BYTES /
FLASH_ERASE_BLOCK_SIZE)  //32K erase block size
```

Así como las posiciones de memoria características, tamaños del bootloader, de la aplicación y en qué registro de la memoria de datos interna del microcontrolador se va a almacenar el bit que indique que existe almacenada una nueva versión (registro DATAFLASH_FLAG_ADDRESS) y que se debe de reprogramar el microcontrolador mediante el bootloader, la próxima vez que se reinicie el programa.

La forma de descargar el nuevo programa viene dada por qué tipo de datos espera leer el bootloader; en este caso se descargará el fichero SREC del programa utilizando una conexión con un servidor FTP, pero puede que cada tecnología de comunicación utilice otro.

El formato SREC (Motorola S-record) es un fichero generado a partir de la compilación de un programa y escrito en ASCII hexadecimal que permite grabar el nuevo programa en un microcontrolador o memoria.

Para extraer el binario del nuevo programa, habrá que recorrer el fichero mediante el parser interpretando las distintas líneas (fichero separado por salto de líneas) para obtener el nuevo programa y las direcciones de memoria en dónde cada parte de este debe de grabarse.

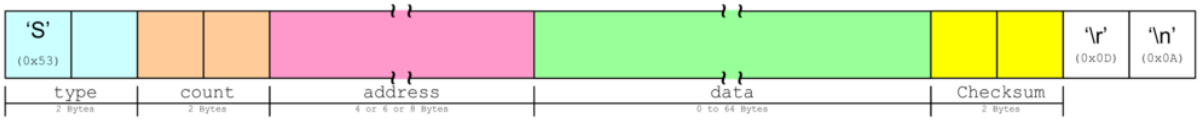


Imagen 17: Formato línea de datos SREC [19]

Record type: unión de un carácter 'S' y un dígito indicando el tipo de trama que sigue.

Byte count: cantidad de bytes en hexadecimal hasta alcanzar el checksum (inclusive).

Address: dirección en hexadecimal en dónde deben de guardarse los datos contenidos en la trama.

Data: datos del programa, corresponden al binario generado por el proceso de compilación.

Checksum: comprobación de que los bytes recibidos posteriormente son correctos.

8.4.1. Proceso de descarga

El proceso de descarga será lanzado por un evento en el sistema, ya sea por la detección automática de una nueva versión o la recepción de una trama en concreto.

Posteriormente, el módulo se conectará a un servidor FTP utilizando el módulo de comunicaciones que tenga implementado dicho protocolo para descargarse el fichero. Al tratarse de un fichero de unos 300-500KB o más, dependiendo del proyecto, es imposible hacerlo a la vez, por lo que se deberá de fraccionar la descarga.

El proceso de descarga desde su inicio se puede observar en el siguiente diagrama:

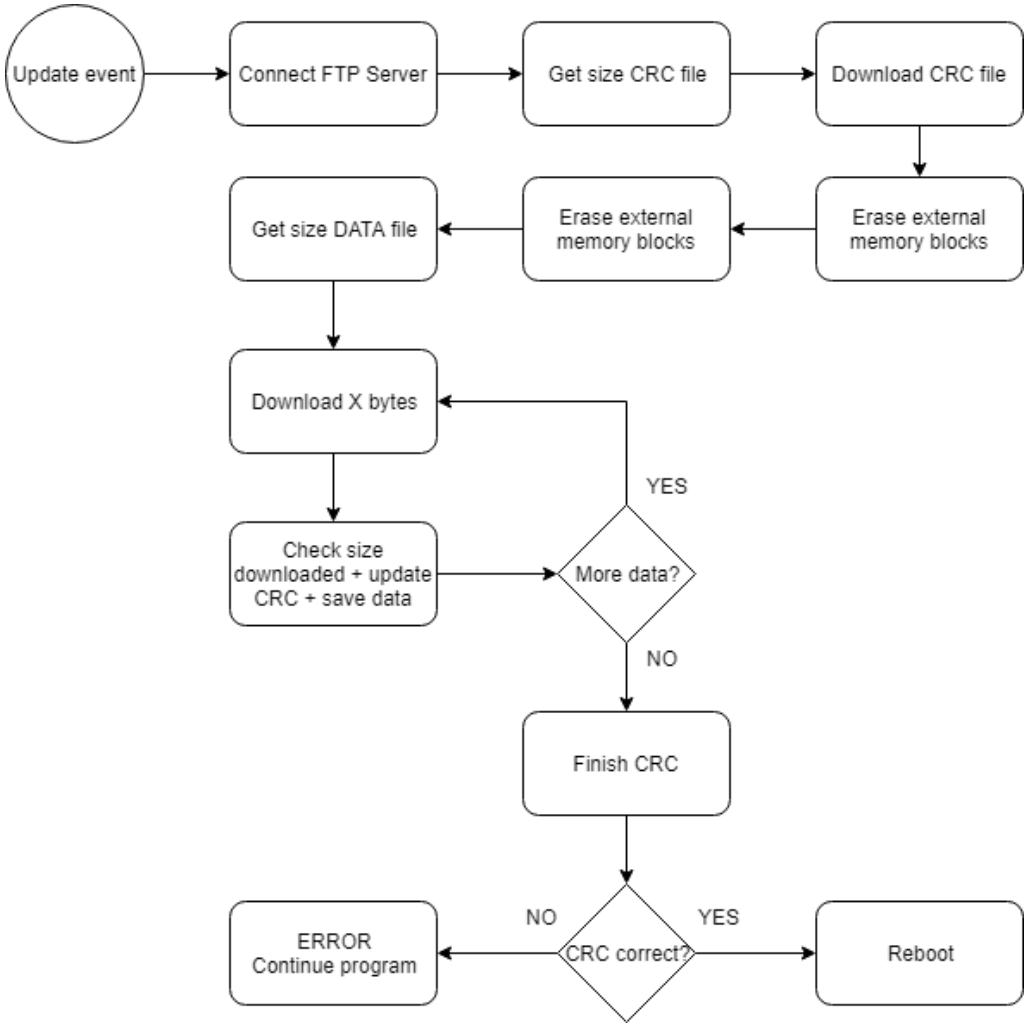


Imagen 18: Diagrama descarga FTP

Como se trata de una descarga crítica que puede dejar inutilizado el dispositivo se ha implementado un sistema de triple comprobación:

- Comprobación en el servidor del tamaño total y del CRC.
- Descarga controlada y fraccionada asegurándose de la cantidad de bytes descargados en todo momento.
- Cálculo de un CRC32 durante la descarga y posterior comprobación con el descargado del servidor.

Los bytes del fichero se ajustan a un número divisible por 4 para tener compatibilidad con todos los cálculos de CRC32 por hardware de los microcontroladores. Dicha modificación se realiza mediante un script implementado en el batch de Windows que obtiene la cantidad de bytes y añade tantos como proceda, generando un fichero <file>.srec.CRC con la nueva longitud y crc obtenidos y otro <file>.srec.DATA con el nuevo fichero SREC con X bytes añadidos.

La Imagen 18 describe el proceso de descarga del microcontrolador una vez se ha lanzado la operación de actualización. El proceso empieza conectándose con el servidor FTP y descargándose el fichero de comprobación, en dónde se obtiene la longitud total de bytes a descargar y el crc de estos. Una vez se obtiene dicha información es hora de descargar el

documento SREC modificado para proceder con la actualización. El tiempo de descarga depende directamente de la cantidad de bytes en los buffers intermedios sabiendo que una descarga mediante un nodo con el que interactuemos con comandos AT será muchísimo más lenta que uno que posea una comunicación por SPI.

Una vez se descarga el fichero entero, se actualiza el CRC sabiendo que este se debe de calcular según el siguiente pseudocódigo [20]:

```
Function CRC32
   Input:
      data:  Bytes     //Array of bytes
   Output:
      crc32: UInt32    //32-bit unsigned crc-32 value

//Initialize crc-32 to starting value
crc32 ← 0xFFFFFFFF

for each byte in data do
   nLookupIndex ← (crc32 xor byte) and 0xFF;
   crc32 ← (crc32 shr 8) xor CRCTable[nLookupIndex] //CRCTable is an array of
256 32-bit constants

//Finalize the CRC-32 value by inverting all the bits
crc32 ← crc32 xor 0xFFFFFFFF
return crc32
```

Para finalizar, se compara con el obtenido mediante la descarga primera y si éste coincide, se actualiza la dirección de memoria interna con un valor que indique al bootloader que existe guardada en la memoria Flash externa una nueva versión del programa.

Cabe decir que la descarga está implementada para soportar fallos y cortes de la comunicación en dónde puede optar por reconectarse con el servidor y seguir la descarga por dónde se había quedado o, directamente, declarar un estado de error y proseguir con el programa actual sin actualizar. En caso de que el CRC no coincida con el descargado, no se realizará ninguna operación ya que no se posee un backup del programa y una actualización con un binario en memoria flash erróneo causará la inutilización del dispositivo si este no posee una memoria de backup.

Una vez se haya reiniciado el programa, se ejecutará el bootloader comprobando que el bit de reprogramación indica que existe una nueva versión, entonces realizará el proceso descrito en la Imagen 17.

9. Protocolos

En el presente punto se muestran los distintos protocolos que soporta o aporta la librería de comunicaciones hasta el momento.

9.1. Raw

Supone la utilización de los sockets en su estado más puro, la transmisión de bytes en crudo.

Se ha decidido poner este tipo de comunicaciones ya que las demás utilizan como base la misma para interactuar con un servidor.

9.2. MQTT

Uno de los protocolos que más se está desarrollando junto con el mundo del IoT es *Message Queuing Telemetry Transport* o comúnmente conocido como MQTT, gracias a su practicidad y simplicidad. Normalizado según el ISO/IEC 20922 en 2016, se presenta la versión más extendida en el mundo de las comunicaciones como un protocolo basado en la arquitectura de publicador/subscriptor bajo el estándar de TCP.

Implementado originariamente por Andy Stanford-Clark (IBM) y Arlen Nipper (Cirrus) en 1993, ha tenido una lenta evolución hasta llegar a estos últimos años que ha experimentado sus mayores actualizaciones gracias al aumento de aplicaciones que lo usan. Debido a este cambio, en diciembre de 2015 se publicó la versión 3.1.1 [21] y recientemente, en marzo de este año 2019, se publicó la más reciente de sus actualizaciones, la versión 5.0 [22].

Esta nueva versión trae consigo mejoras que ya venían demandándose desde hace tiempo [23]. Especificadas en el Apéndice C de su publicación [22], las más destacadas son:

- Códigos de respuesta en cada contestación por parte del servidor facilitando un mayor entendimiento del error. Anteriormente solo se poseía una respuesta positiva o negativa. Además, se incorpora la posibilidad de recibir dicha contestación en texto para que la aplicación cliente pueda mostrarla más rápidamente.
- El servidor podrá cerrar las conexiones a nivel de protocolo y no directamente sobre TCP, permitiendo a la aplicación percatarse de que no ha sido un error en el socket.
- Tipo de los datos transmitidos indicando su tipo. En la versión 3.1.1 ya se podía transmitir cualquier tipo de datos (texto, json y hexadecimal) pero resultaba una limitación importante por parte del servidor a la hora de interpretarlos.
- Formalización del *Request/Response*. Actualmente, para leer información del servidor por parte del cliente se debía de publicar en un topic y esperar la respuesta en otro; al no estar definido, cada aplicación utilizaba sus métodos y lenguaje para implementar dicha funcionalidad.
- Definición de los topic mediante un alias. En un escenario en dónde la cantidad de información transmitida era considerable, podía darse el caso de que la cantidad de topics utilizados creciese de tal forma de que la cantidad de bytes transmitidos en el paquete fuera principalmente la transmisión del topic en dónde se publicaba el mensaje. Para solventar dicho problema y reducir los tamaños, se puede identificar todo un topic completo mediante su ID.
- Se añaden mejoras en la persistencia y generalización del protocolo mediante el uso de un MTU máximo, el uso de control de flujo (limitación en el número de paquetes publicados con QOS > 0 sin haber sido reconocidos por el servidor) y la especificación de propiedades para cada paquete.
- Mayor información del cliente respecto del servidor y viceversa.

Dado que la implementación de dichos escenarios correspondería con el desarrollo completo de un proyecto independiente y especializado, se ha optado por acoplar un desarrollo externo al proyecto. De esta forma externalizamos el proyecto sin olvidar que susodicho corresponde con un código verificado y que cumple a la perfección con las especificaciones del proyecto.

La librería seleccionada, por ser la más extendida de entre todas las plataformas es la de Paho. El proyecto de Eclipse Paho provee que una implementación de código abierto del protocolo MQTT y MQTT-SN para la parte cliente del mismo. Aunque su versión más extendida y probada corresponde con la 3.1.1, ya se ha empezado a desarrollar la versión 5 para algunos de los lenguajes y será relativamente sencillo actualizar la librería cuando esté disponible para dispositivos embebidos. La siguiente imagen nos muestra las posibilidades que proporciona la utilización de esta librería [24]:

Client	MQTT 3.1	MQTT 3.1.1	MQTT 5.0	LWT	SSL / TLS	Automatic Reconnect	Offline Buffering	Message Persistence	WebSocket Support	Standard MQTT Support	Blocking API	Non-Blocking API	High Availability
Java	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Python	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗
JavaScript	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
GoLang	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
C	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C++	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Rust	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
.Net (C#)	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✗	✓	✗
Android Service	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Embedded C/C++	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗

Imagen 19: Implementaciones de la librería Paho (MQTT)

Aunque la librería posee todos los métodos necesarios para construir o interpretar los mensajes, es el cliente quien debe de hacer uso de la misma para asegurarse de que se cumple el estándar [25].

Por este motivo, debe de existir una librería que utilice la de Paho para implementar:

- Reintentos de conexión.
- Timeouts de los mensajes.
- Envío y recepción de las respuestas para las distintas calidades de los mensajes: “como máximo una”, “al menos una vez”, “exactamente una vez”.
- Realización del PING al servidor.

Como puede ser el caso de que dichos requisitos puedan haber sido ya implementados por los componentes de comunicación, como por ejemplo, el módulo de Quectel BG96; se opta por redefinir todas las utilidades básicas que proporciona el protocolo. De esta forma conseguimos un nivel de abstracción absoluto.

```
typedef enum{
    MQTT_STOP,
    MQTT_WORKING
}mqtt_status_code_e;

typedef void (* mqtt_sub_cb)(char *
topic_buf, uint16_t topic_len, char *
data_buf, uint16_t data_len);

typedef struct{
    mqtt_status_code_e connection;
    mqtt_status_code_e subscription;
}mqtt_status_t;
```

```
typedef enum{
    QOS0,
    QOS1,
    QOS2
}mqtt_qos_d;

typedef struct{
    char * buf;
    uint16_t len;
} mqtt_string_t;
```

```
typedef struct{
    mqtt_string_t topic;
    mqtt_string_t value;
    mqtt_qos_d qos;
    bool retain :1;
}mqtt_msg_t;

typedef struct{
    mqtt_string_t topic;
    mqtt_qos_d qos;
    mqtt_sub_cb func;
}mqtt_sub_t;

typedef struct{
    bool useCredentials;
    mqtt_string_t user;
    mqtt_string_t password;
}mqtt_userCredentials_t;

typedef struct{
    char
host[MAX_SERVER_HOST_NAME_LEN];
    uint16_t port;
    char
clientID[MQTT_CONF_MAX_CLIENT_ID_LEN];
    bool useWill;
    mqtt_msg_t will;
    mqtt_userCredentials_t
credentials;
    conn_socket_t * socket;
}mqtt_conn_t;
```

Y la forma en que una aplicación puede interactuar será con:

```
#ifndef BG96_USE_MQTT
void mqtt_loop(void);
void mqtt_send_ping(void);
#endif

void mqtt_connect(mqtt_conn_t * connection);
void mqtt_disconnect(void);

void mqtt_publish(mqtt_msg_t * msg);
void mqtt_subscribe(mqtt_sub_t * sub);
void mqtt_unsubscribe(mqtt_string_t *topic);

void mqtt_set_subcription_all_func(mqtt_sub_cb func);
```

La función *mqtt_connect* es la encargada de inicializar todas las estructuras y variables internas, así como iniciar el socket correspondiente, para empezar una nueva comunicación. Decir que, por facilidad de la aplicación, sólo se permite una única conexión, pero se ha pensado la gestión de la librería para que soporte más de una comunicación MQTT. La desconexión se realizará mediante *mqtt_disconnect*.

La publicación (función *mqtt_publish*) se realiza mediante el paso de un puntero a una estructura que contiene la información básica según la especificación:

- Socket por el que se transmite.
- Topic en dónde se publica.
- Mensaje a transmitir.
- QOS o nivel de servicio.
- Bit de retain para indicar si el servidor debe de retener el mensaje.

Por terminar con las funciones básicas de la librería, faltaría por explicar mqtt_subscribe y mqtt_unsubscribe. Para lograr una mayor independencia, se ha implementado un sistema de memoria que almacena las subscripciones del usuario, permitiendo que éste configure una función independiente que reciba los datos sin tener que preocuparse de hacer el parseado cada vez.

Las funciones que quedan por explicar son las que se encargan de las gestiones necesarias para mantener la conexión abierta con el servidor en todo momento. Recordar que dichas funcionalidades sólo se utilizarán en caso de que la librería que se quiera utilizar sea la de Paho y no la perteneciente a cada módulo de comunicación ya que, de normal, estos ya implementan dichos métodos internamente.

La función *mqtt_send_ping* se encarga, como su nombre indica, del envío periódico de un mensaje para mantener el socket de comunicación abierto. Consultar el punto 3.12 de la especificación.

La función mqtt_loop es la encargada de leer del buffer de entrada los bytes recibidos en el socket utilizado y clasificar los mensajes dependiendo de su tipo, tratar cada uno de ellos generando la respuesta correcta o trasladando los datos a la función correspondiente. A parte de esta función existe un timer software que se encarga de mantener los timeouts de cada comando enviado al servidor hasta agotar el tiempo o los reintentos. Para ello, se mantiene una cola FIFO (véase el punto 8.2.3) de mensajes enviados.

9.3. REST

Las siglas de REST hacen referencia a Representational State Transfer y corresponde un estilo de arquitectura que define una serie de patrones para crear e interactuar con servicios webs [26]. La finalidad base del mismo es la de realizar una interacción entre máquinas de una forma sencilla y cómoda.

Entre sus características más básicas se encuentran:

- Arquitectura Cliente-Servidor: el desarrollo de ambas puede llevarse a cabo de manera separada ya que se desconocen entre ellos [27].
- Sin estado: al basarse en el protocolo HTTP, hereda de este la cualidad de no almacenar el estado de la comunicación; haciendo de cada una de ellas sea como la primera.
- Cache: existe la posibilidad en dispositivos de más complejos de almacenar en cache, respuestas a las peticiones más comunes para ahorrar los tiempos de respuesta.
- Sistema de capas: un cliente desconoce si realmente se conecta al servidor final o lo hace con un doble. Con esta técnica se mejorar la escalabilidad y los tiempos de respuesta.
- Protocolo sencillo: así como implementa HTTP, REST posee las mismas llamadas que este: GET, POST, PUT, PATCH, DELETE haciendo que la interoperabilidad y sencillez entre los equipos sea la idónea para sistema IoT.

En la siguiente imagen se muestra una interacción mediante REST:

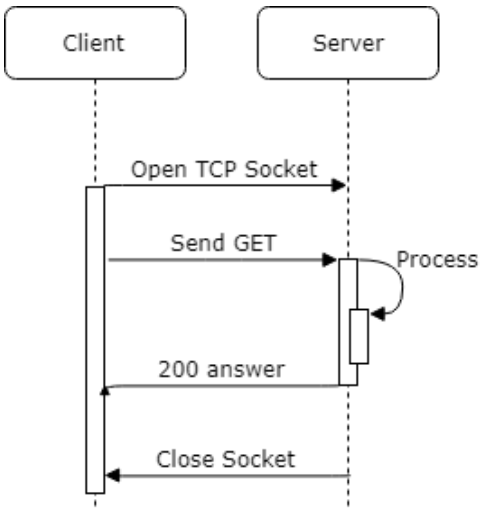


Imagen 20: Diagrama REST

La razón por la que se opta al desarrollo de esta librería es que la mayoría de servidores IoT, junto con MQTT, implementan la subida de información utilizando dicho protocolo.

Como se indica, la librería implementada quiere seguir con la sencillez del protocolo. Para lograrlo, se acopla la creación, envío y recepción de la información utilizando el sistema de abstracción de los mismos descritos en el punto 6.5.

Aunque se parte de una librería inicial desarrollada por el usuario de GitHub markusfisch [28] se han implementado varias modificaciones:

```
rest_code_e http_GET(conn_socket_t *sd, const char *url, struct http_message *
msg);

rest_code_e http_POST(conn_socket_t *sd, const char *url, uint8_t *buf,
uint16_t buf_len, struct http_message * msg);
```

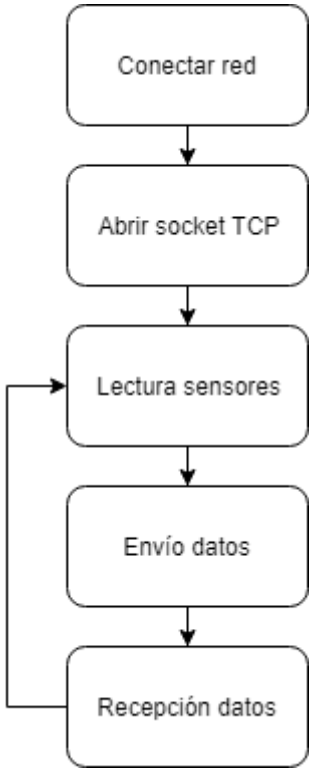
A la hora de compatibilidad con los servidores HTTP, es necesario declarar una cabecera genérica y sin requerimientos que den error, la diseñada es:

```
GET /<Petition> HTTP/1.1
User-Agent: Mozilla/4.0 (Linux)
Host: <IP/Host>
Accept: */*
Connection: close
```

10. Aplicación ejemplo

Una vez descritos cada componente que aporta el proyecto, es hora de ensamblar algunos de ellos para implementar una posible aplicación de usuario.

Dado que la mayoría de librerías está orientada a la comunicación se diseña un ejemplo que pueda:



1. Conectarse a una red de comunicaciones utilizando la tecnología seleccionada.
2. Conectarse a un servidor mediante un socket TCP.
3. Obtener información de algunos sensores.
4. Transmitir dicha información con un protocolo y formato conocidos y estándar.
5. Recibir información del servidor para actualizar algunos parámetros. Repetir proceso.

Imagen 21: Secuencia básica aplicación ejemplo

La implementación de dicha funcionalidad se podría implementar mediante el uso de un programa “tradicional” que se compone por un main y un bucle infinito para evitar que el programa termine, pero como uno de los objetivos es la utilización de ThreadX como SO, se optará por esta segunda solución.

Cabe recordar que, para poseer un desarrollo ágil en el presente ejemplo, así como en futuros desarrollos con unas especificaciones marcadas por un cliente, se poseen 2 proyectos:

Librerías Synergy: son todas aquellas librerías, funciones y definiciones que se han descrito en los puntos anteriores a este dado que todas ellas forman parte de las herramientas diseñadas para agilizar futuros desarrollos. Se entiende que para diferentes proyectos con una misma funcionalidad (por ejemplo, acceso al modem o al wifi) deben de poseer un único fichero de la librería. Esto es, existirá un proyecto general (sin ninguna funcionalidad) que contenga las librerías implementadas anteriormente. Por otra parte, se poseerá un nuevo proyecto por cada nuevo producto que haga uso de dichos ficheros.

La utilización de dicho sistema trae consigo varias ventajas como desventajas.

La ventaja más destacada es que siempre existe la posibilidad de tener la librería más actualizada con los últimos errores solucionados, pero por el contrario, si se desea implementar una nueva funcionalidad o manera de hacer las cosas, se deberá tener en cuenta

que, puede haber otros proyectos que hagan uso de esos mismos recursos; se deberá de mantener la compatibilidad en todo momento.

Esta misma ventaja es, a su vez, un trabajo extra para el desarrollador, ya que deberá de tener una precaución extra a la hora de modificar dichos ficheros o de utilizarlos.

Volviendo al desarrollo del ejemplo, se pretende que una tarea sea la encargada de mantener la linealidad del programa (que ejecute cada paso mostrado en la Imagen 21) mientras sea otra la que se encargue de ejecutar los eventos, interrupciones no prioritarias, el manejo y control de los sockets, etc., esto es, todas aquellas funciones que se pondrían en el bucle infinito de un programa sin SO.

Con esta separación también se quiere mejorar la legibilidad del código, así como su mantenimiento y reutilización en futuros desarrollos. El código de este Thread se encargará de:

1. Iniciar periféricos (*Utils* , Software Timers, etc.).
2. Iniciar las librerías de comunicación.
3. Iniciar el sistema de parseo por comandos AT.
4. Loop infinito en dónde:
    a. Ejecutar funciones de los timers.
    b. Mantener las librerías de comunicación para recibir y enviar los comandos.
    c. Mantener la librería de MQTT.

Mientras que el programa que se encarga de implementar la operativa de la aplicación se describe como:

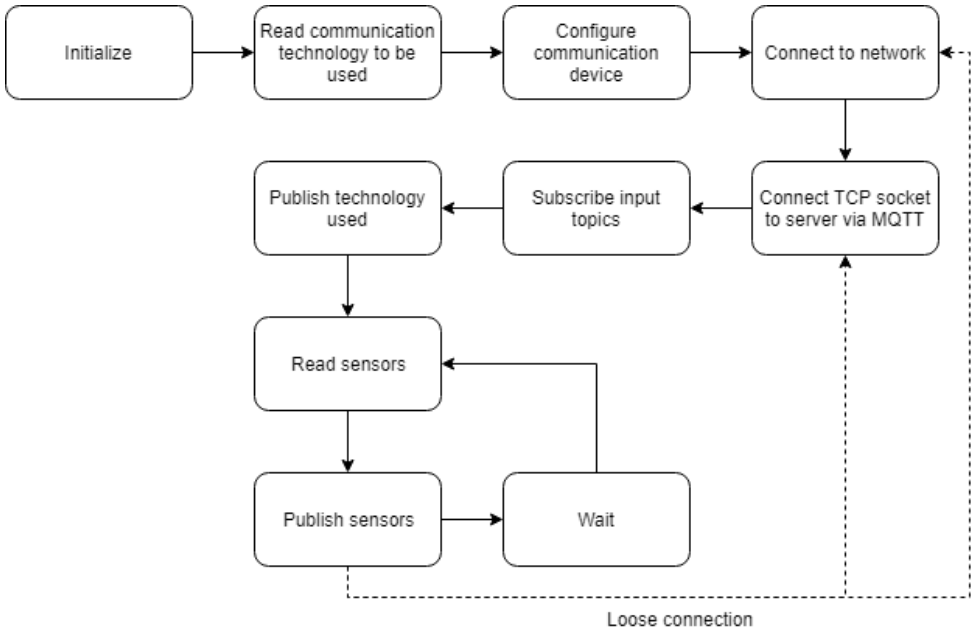


Imagen 22: Esquema aplicación principal

Para el desarrollo de este pequeño ejemplo, ha sido necesario la utilización de:

- Salida estándar
- GPIO
- IRQ
- ADC
- PWM
- Software Timer
- Flash interna del microcontrolador
- UART
- Comandos AT
- Wifi
- 6LowPAN
- Módem
- JSON
- MQTT

Para complementar el diseño, el dispositivo está suscrito para recibir actualizaciones de un servidor; mediante este método, se puede cambiar el estado de los leds y establecer la frecuencia del PWM.

El propósito del ejemplo es el de mostrar el correcto funcionamiento del sistema aportando una visión generar del mismo.

El lector debe saber que para desarrollar cada punto del proyecto se han seguido los siguientes pasos:

1. Investigación / lectura de información.
2. Primera implementación de una versión preliminar de la librería.
3. Desarrollo de un proyecto de test exclusivo para la funcionalidad que se quiere desarrollar.
4. Se completa la librería en dónde se incluya todas las funciones requeridas.
5. Se comprueba su correcto funcionamiento con un pequeño ejemplo de test.
6. Se incorpora al conjunto de librerías ya comprobadas.

Siguiendo los siguientes puntos se consigue un desarrollo basado siempre sobre librerías testeadas reduciendo considerablemente la probabilidad de error durante el desarrollo.

11. Thingsboard

Thingsboard es una plataforma IoT open-source que permite un rápido despliegue, control y escalabilidad que cualquier proyecto IoT [29], entrando dentro del grupo de servidores que permiten una recepción de la información, su almacenamiento y una simple interfaz web para visualizarlos. Se permite:

- Definir dispositivos, activos y clientes, así como, las relaciones entre ellos.
- Recopilar y visualizar los datos de los activos y dispositivos.
- Analizar y procesar la información entrante para poder modificarla al momento, así como generar alarmas.
- Control remoto de los dispositivos mediante RPC.
- Creación de flujos de trabajo según los distintos eventos generados y procedencia de los datos.
- Diseñas cuadros de mando para visualizar la información de forma dinámica y auto actualizable, así como, la interacción con los dispositivos mediante estos.
- Habilitar y desactivar características y procedimientos haciendo uso de las cadenas de reglas personalizables.
- Migración de los datos a otros sistemas.
- Recepción mediante algunos de los protocolos más utilizados en el mundo del IoT como HTTP, MQTT, OPC-UA.
- Utilización de tecnologías de comunicación como SigFox y TheThingNetwork (LoRaWAN).
- Utilización de los servicios de Azure Event Hub (Microsoft), IBM Watson IoT (IBM) y AWS IoT (Amazon).
- Planificación de eventos.
- Capacidad de exportar la información en formatos como CSV o XLS.
- Utilización de bases de datos externas para la persistencia de la información.
- Generación automática de reportes.

Algunas plataformas anunciadas ofrecen servicios parecidos como Ubidots, Azure IoT Hub, Hologram IoT, Google Cloud IoT, Fiware, Sentilo, etc. pero todas ellas poseían algún conveniente o un despliegue más complicado que Thingsboard ya que esta se puede instalar es una simple Raspberry PI 3.

De esta misma forma se ha llegado a cabo la instalación y prueba del ejemplo desarrollado en el anterior punto ya que se deben generar datos para poder ser utilizados.

El sistema montado es el siguiente:

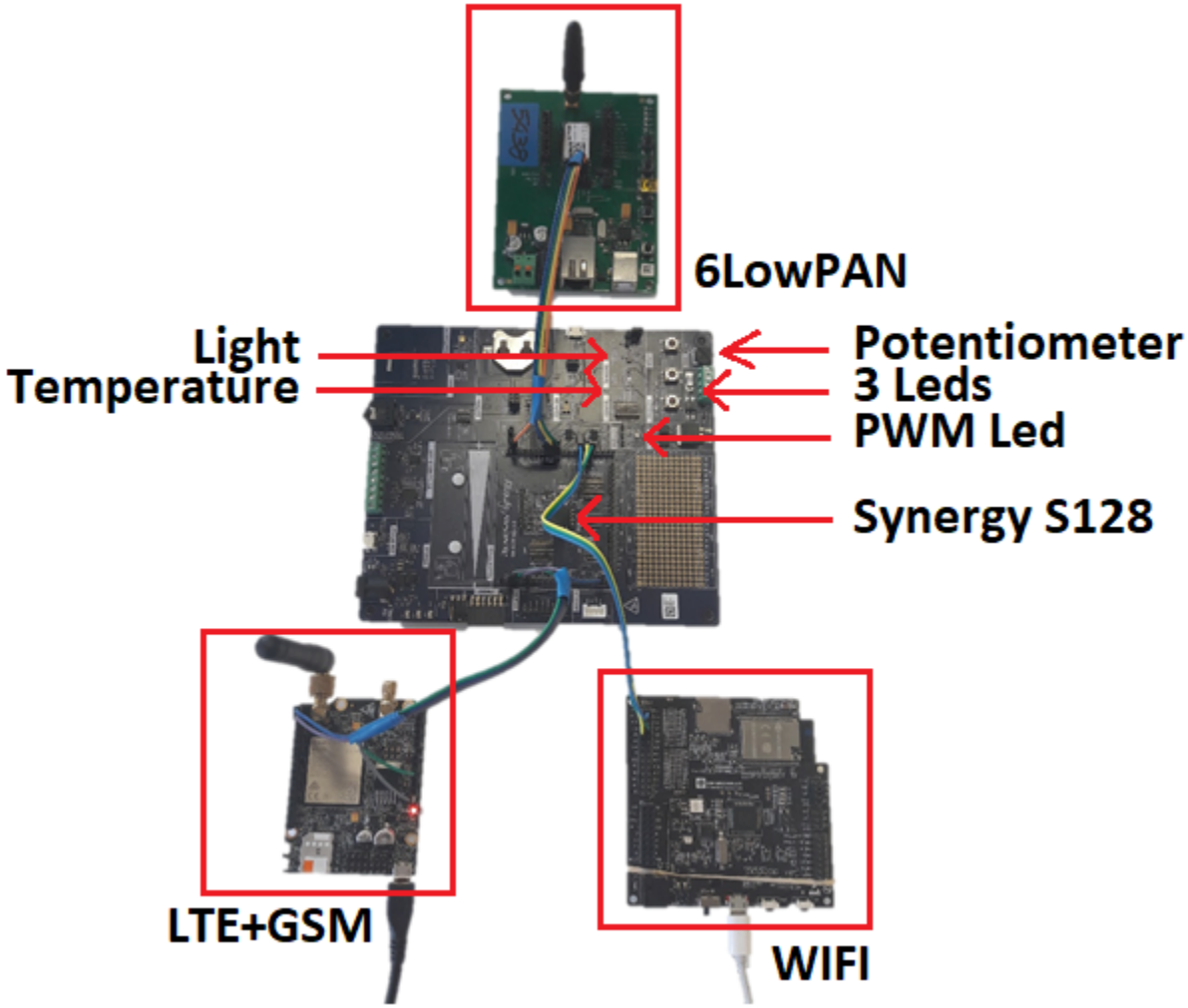


Imagen 23: Montaje demostración

Aunque pueda parecer un montaje sencillo se ha utilizado las herramientas de las que se disponía en la placa de evaluación del microcontrolador S128 para obtener medidas de:

1. Potenciómetro
2. Sensor de temperatura
3. Sensor de luminosidad

Como complemento se han utilizado los siguientes actuadores:

1. PWM Led
2. 3 leds de colores

En la misma imagen se puede ver la conexión de los distintos módulos de comunicación con el micro para poder enviar dichas mediciones al servidor y poder recibir las actuaciones a realizar.

Por parte del servidor, se ha diseñado un panel de usuario con toda la información generada, así como las posibilidades de actuación que se tienen:

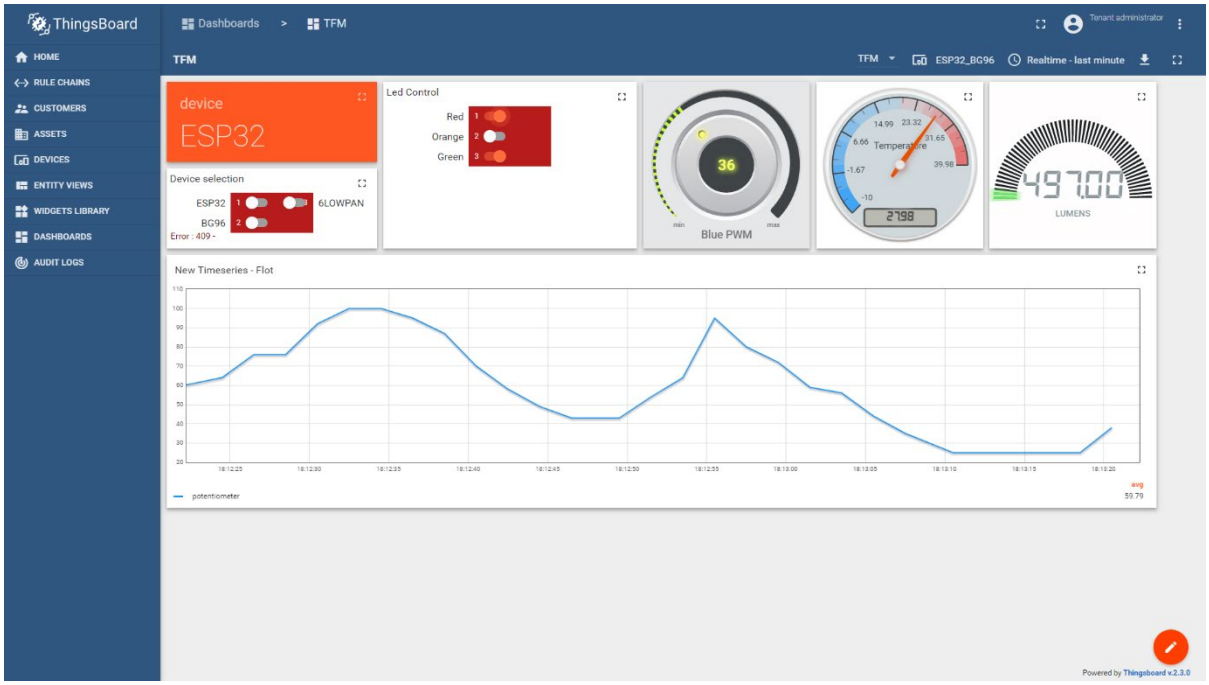


Imagen 24: Thingsboard

Pudiendo observar (de izquierda a derecha): el dispositivo de comunicación por el que se transmiten los datos y el panel para cambiarlo, unos botones para actualizar el estado de los leds de colores, un *scroll* circular para seleccionar la frecuencia de brillo del led PWM, la temperatura actual, la luminosidad y un histórico de los valores generados por el potenciómetro.

Como todo programa que necesite depurar, se dispone de una librería que permite la impresión por distintas salidas para obtener el estado del programa en todo momento:

```
INFO     MNG   Management thread entry.
INFO    ATPR   AT module initialized.
INFO    USER   User thread entry.
DEBUG   UART   Uart module initialized.
DEBUG   ATPR   AT commands initialization
1.INFO ESP32 Initialization of
module Espressif ESP32
TRACE   ESP32  We have connection.
TRACE   ESP32  We have IP.
2.INFO C_DEV Status is: CONNECTED
INFO    C_DEV  IP : 192.168.139.181
INFO    C_DEV  MAC: 84.0D.8E.18.85.7C
3.TRACE      M_TRA Opening MQTT
socket.
INFO    ESP32  TCP Connection: 0
TRACE   MQTT   Sending connection
INFO    MQTT   Connection established.
TRACE   MQTT   Sending subscription
INFO    MQTT   Subscription done.
TRACE   MQTT   Sending publish
...
TRACE   MQTT   Sending publish
INFO    USER
 {"method":"getLedStatus","params":"{}"}
TRACE   MQTT   Sending publish
...
```

```
TRACE   MQTT   Sending publish
INFO    USER
 {"method":"setPWM","params":"0"}
4.TRACE      MQTT Send PINGREQ
TRACE   MQTT   Sending publish
TRACE   MQTT   Receive PINGRESP
INFO    USER
 {"method":"setPWM","params":"36"}
TRACE   MQTT   Sending publish
TRACE   MQTT   Sending publish
INFO    USER
 {"method":"setLedStatus","params":{"pin
":3,"enabled":true}}
TRACE   MQTT   Sending publish
...
TRACE   MQTT   Sending publish
INFO    USER
 {"method":"setLedStatus","params":{"pin
":1,"enabled":true}}
TRACE   MQTT   Sending publish
...
TRACE   MQTT   Sending publish
TRACE   MQTT   Send PINGREQ
TRACE   MQTT   Receive PINGRESP
TRACE   MQTT   Sending publish
...
TRACE   MQTT   Sending publish
```

```
5.INFO USER
 {"method":"setPWM","params":"81"
}
INFO    USER
 {"method":"setPWM","params":"94"}
TRACE   MQTT   Sending publish
TRACE   MQTT   Sending publish
INFO    USER
 {"method":"setPWM","params":"99"}
INFO    USER
 {"method":"setPWM","params":"8"}
TRACE   MQTT   Sending publish
TRACE   MQTT   Sending publish
TRACE   MQTT   Send PINGREQ
TRACE   MQTT   Receive PINGRESP
TRACE   MQTT   Sending publish
6.INFO USER
 {"method":"setDevice","params":{
"pin":2,"enabled":true}}
TRACE   MQTT   Sending publish
TRACE   MQTT   Sending publish
INFO     MNG   Management thread entry.
INFO    ATPR   AT module initialized.
INFO    USER   User thread entry.
DEBUG   UART   Uart module initialized.
DEBUG   ATPR   AT commands initialization
DEBUG   BG96   Initialization of module
Quectel BG96.
TRACE   BG96   SIM Card enable.
TRACE   BG96   SIM Card inserted.
7.TRACE     BG96  SIM waiting for
a PIN.
INFO    BG96   CCID = 8934014231631247894
INFO    BG96   IMEI = 866425030636924
INFO    BG96   BG96 Configured properly.
8.INFO BG96  Connecting to
network.
INFO    BG96   We have connection
DEBUG   C_DEV  IP = 74.0.32.17
9.TRACE     M_TRA Opening MQTT
socket.
INFO    BG96   TCP Connection: 0
TRACE   MQTT   Sending connection
INFO    MQTT   Connection established.
TRACE   MQTT   Sending subscription
INFO    MQTT   Subscription done.
TRACE   MQTT   Sending publish
TRACE   MQTT   Sending publish
10.INFO     USER
 {"method":"setLedStatus","params
":{"pin":3,"enabled":false}}
TRACE   MQTT   Sending publish
TRACE   MQTT   Sending publish
TRACE   MQTT   Send PINGREQ
TRACE   MQTT   Sending publish
TRACE   MQTT   Receive PINGRESP
TRACE   MQTT   Sending publish
```

1. Se obtiene de la memoria el dispositivo a ejecutar.
2. Conexión a la red Wifi.
3. Apertura conexión con el servidor MQTT.
4. La misma librería mantiene la conexión abierta.
5. Recepción del cambio de estado del led PWM.
6. Recepción del evento de cambio de dispositivo de comunicación. Se actualiza la memoria.
7. Se inicializa el módem BG96 (LTE+GSM) siguiendo los pasos necesarios.
8. Conexión a la red móvil.
9. Conexión al servidor MQTT.
10. Se vuelve a enviar y recibir información del servidor.

12. Conclusiones

En este proyecto se han diseñado e implementado librerías robustas y resistentes a la escalabilidad, las cuales, permiten el acceso y utilización del Hardware junto con las tecnologías de comunicación más utilizadas en la mayoría de proyectos actuales relacionados con el mundo del IoT. Para llegar a ese punto se tuvo que estudiar la utilización de la familia de microcontroladores Renesas Synergy mediante el IDE e2 Studio (punto 3). Posteriormente, y como soporte para la realización de proyectos de mayor envergadura, se empleó ThreadX como sistema operativo para aportar la ejecución en paralelo de distintas tareas al mismo tiempo (punto 4).

A partir de dicha base, se construyó todo un sistema de librerías capaces de abarcar desde la obtención de información por medio de sensores hasta la visualización de los mismos en un servidor de alto nivel; pasando por el desarrollo necesario para dotar al sistema de comunicación.

Si volvemos a repasar el apartado de Objetivos (punto 2) nos daremos cuenta de que el proyecto, teniendo como visión general, “El desarrollo de librerías de abstracción y comunicaciones para desarrollos basados en la familia de microcontroladores Renesas Synergy” ha ido implementando las distintas capas de abstracción (punto 6) necesarias para dotar a futuros desarrollos del manejo de diferentes tecnologías de comunicación de forma transparente y abstracta. Empezando por la comunicación con los diversos módulos (punto 7), e implementando los sistemas de manejo para la conexión con el medio, la comunicación mediante TCP y la utilización de protocolos más complejos; se ha estudiado diferentes opciones de utilizar los periféricos que proporciona un microcontrolador para obtener una solución completa.

Como resultado del proyecto se ha obtenido un conjunto de librerías como se puede ver en la Imagen 25 que comprenden HAL, utilidades genéricas, tecnologías de comunicación, utilización de la red, protocolos de comunicación, memoria y bootloader.

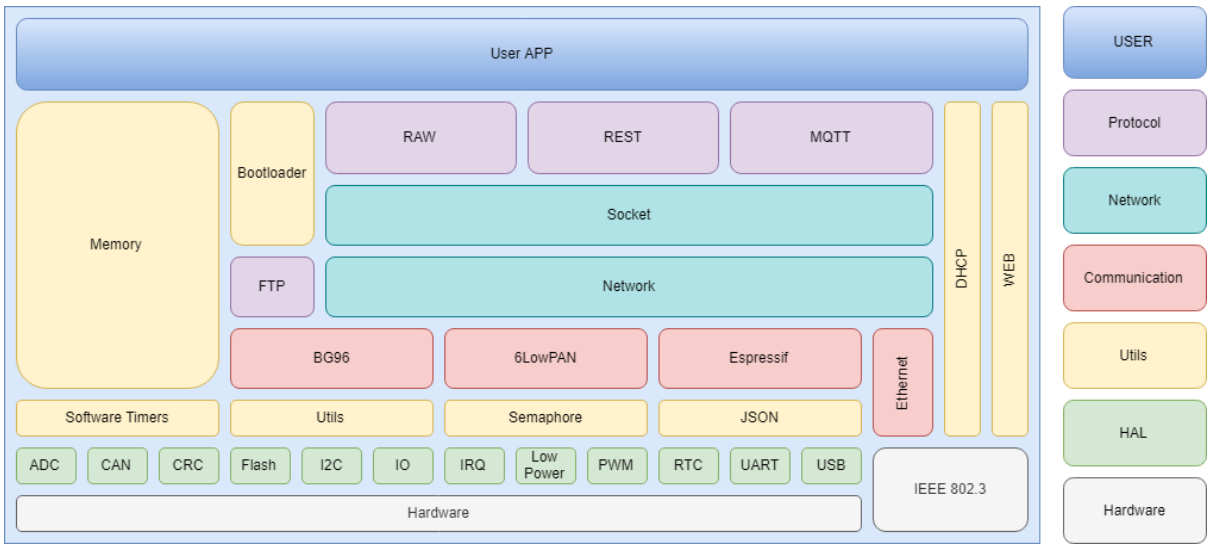


Imagen 25: Librerías implementadas en el proyecto

Cabe mencionar que cada componente puede verse actualizado sin limitar el uso del resto, y a su vez, dotar del sistema de nuevas funcionalidades.

Con todas estas herramientas, se da por finalizado el presente documento mostrando que una relación planeada de los diferentes componentes entre ellos facilitará el desarrollo de futuros proyectos.

13. Bibliografía

[1] Firmware, «Wikipedia,» 21 May 2019. [En línea]. Available: https://es.wikipedia.org/wiki/Firmware.

[2] Hall, «Wikipedia,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Sensor_de_efecto_Hall.

[3] Renesas, «Renesas Synergy Microcontrollers Family,» Renesas, [En línea]. Available: https://www.renesas.com/us/en/products/synergy/hardware/microcontrollers.html.

[4] ExpressLogic. [En línea]. Available: https://rtos.com/solutions/threadx/real-time-operating-system/.

[5] ThreadX, «Wikipedia,» Wikipedia, 28 April 2019. [En línea]. Available: https://en.wikipedia.org/wiki/ThreadX.

[6] IoT6, «IoT6,» [En línea]. Available: https://iot6.eu/ipv6_for_iot.

[7] IETF, «IEFT,» September 2011. [En línea]. Available: https://tools.ietf.org/html/rfc6282.

[8] J. Olsson, Texas Instruments, 2014. [En línea]. Available: http://www.ti.com/lit/wp/swry013/swry013.pdf.

[9] ZMDI, «IDT,» 29 November 2018. [En línea]. Available: https://www.idt.com/document/dst/zwir4512-datasheet.

[10] SCI, «IDT,» 15 April 2016. [En línea]. Available: https://www.idt.com/document/zwir45xx-serial-command-interface-sci-user-guide.

[11] ARC, «ZMDI,» IDT, 12 April 2016. [En línea]. Available: https://www.idt.com/document/man/zwir451x-programming-guide.

[12] RFC, December 1998. [En línea]. Available: https://www.ietf.org/rfc/rfc2460.txt.

[13] Espressif, «Espressif,» Espressif, [En línea]. Available: https://www.espressif.com/en/products/hardware/modules.

[14] Quectel, «BG96,» [En línea]. Available: https://www.quectel.com/product/bg96.htm.

[15] Renesas, Manual del Usuario S128.

[16] P. IoT, «Renesas,» 6 April 2017. [En línea]. Available: https://renesasrulz.com/synergy/b/weblog/posts/transitioning-to-threadx-semaphores.

[17] S. Zaitsev, «Github,» 20 April 2019. [En línea]. Available: https://github.com/zserge/jsmn.

[18] CFI, «Wikipedia,» 24 January 2018. [En línea]. Available: https://en.wikipedia.org/wiki/Common_Flash_Memory_Interface.

[19] SREC, «Wikipedia,» 26 January 2019. [En línea]. Available: https://en.wikipedia.org/wiki/SREC_(file_format).

[20] CRC32, «Wikipedia,» 23 May 2019. [En línea]. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[21] OASIS, «OASIS,» OASIS, 29 October 2014. [En línea]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[22] OASIS, «OASIS,» OASIS, 7 March 2019. [En línea]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[23] J. Levell, «MQTT,» 3 April 2019. [En línea]. Available: http://mqtt.org/2019/04/mqtt-v5-0-now-an-official-oasis-standard.

[24] Paho, «Eclipse Paho,» Eclipse, [En línea]. Available: https://www.eclipse.org/paho/.

[25] MQTT, «OASIS,» OASIS, 29 October 2014. [En línea]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[26] REST, «Wikipedia,» 20 May 2019. [En línea]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer.

[27] Codeacademy, «REST,» -. [En línea]. Available: https://www.codecademy.com/articles/what-is-rest.

[28] Markusfisch, «libhttp,» Rest lib, 3 May 2017. [En línea]. Available: https://github.com/markusfisch/libhttp.

[29] Thingsboard, «thingsboard.io,» Thingsboard, [En línea]. Available: https://thingsboard.io/docs/getting-started-guides/what-is-thingsboard/.

Anexo I

	Serie 1			Serie 3				Serie 5		S 7
	S124	S128	S1JA	S3A1	S3A3	S3A6	S3A7	S5D9	S5D5	S7G2
Procesador	M0+	M0+	M23	M4	M4	M4	M4	M4	M4	M4
Flash	128	256	256	256	512	1024	1MB	1MB	2MB	4MB
SRAM	16	24	32	32	96	192	192	384	640	640
Memory Mirror Function			X	X	X		X	X	X	X
Memory Protection Unit		X	X	X	X	X	X	X	X	X
External Memory Bus				X	X		X	X	X	X
12-Bit A/D Converter								X	X	X
8-Bit D/A Converter			X	X						
12-Bit D/A Converter	X		X	X	X	X	X	X	X	X
14-Bit A/D Converter	X	X		X	X	X	X			
16-Bit A/D Conver			X							
24-Bit Sigma Delta A/D Converter			X							
High-Speed Analog Comparator			X		X		X	X	X	X
Low-Power Analog Comparator	X	X	X	X	X	X	X			
Operational Amplifier		X	X	X	X	X	X			
Programmable Gain Amplifier			X					X		X
Temperature Sensor	X	X	X	X	X	X	X	X	X	X
Asynchronous General Purpose Timer	X	X	X	X	X	X	X	X	X	X
General PWM Timer 16-Bit	X	X	X	X	X	X				
General PWM Timer 32-Bit	X	X	X	X	X	X	X	X	X	X
Watchdog Timer	X	X	X	X	X	X	X	X	X	X
2D Drawing Engine								X		X
Capacitive Touch Sensing Unit	X	X	X	X	X	X	X	X	X	X
Graphics LCD Controller								X		X
JPEG Codec								X		X
Parallel Data Capture Unit								X	X	X
Segment LCD Controller				X	X	X	X			
Controller Area Network	X	X	X	X	X	X	X	X	X	X
Inter-Integrated Circuit Interface	X	X	X	X	X	X	X	X	X	X

Infrared Data Interface					X		X	X	X	X
Quad Serial Peripheral Interface				X	X	X	X	X	X	X
Sampling Rate Converter								X	X	X
SD/MMC Host Interface				X	X	X		X	X	X
Serial Communications Interface	X	X	X	X	X	X	X	X	X	X
Serial Peripheral Interface	X	X	X	X	X	X	X	X	X	X
Serial Sound Interface				X	X	X	X	X	X	X
USB 2.0 Full Speed interface	X	X	X	X	X	X	X	X	X	X
USB 2.0 High Speed interface								X		X
Battery Backup				X	X	X	X	X	X	X
Clock Management	X	X	X	X	X	X	X	X	X	X
Data Transfer Controller	X	X	X	X	X	X	X	X	X	X
DMA Controller				X	X	X	X	X	X	X
Event Link Controller	X	X	X	X	X	X	X	X	X	X
Low Power Modes	X	X	X	X	X	X	X	X	X	X
Port Function Select	X	X	X	X	X	X	X	X	X	X
Real-Time Clock	X	X	X	X	X	X	X	X	X	X
Switch Regulator										X
SysTick	X	X	X	X	X	X	X	X	X	X
ADC Diagnostic	X	X	X	X	X	X	X	X	X	X
Clock Frequency Accuracy Measurement Circuit	X	X	X	X	X	X	X	X	X	X
Cyclic Redundancy Check Calculator	X	X	X	X	X	X	X	X	X	X
Data Operation Circuit	X	X	X	X	X	X	X	X	X	X
Error Correction Code in SRAM		X	X	X	X	X	X	X	X	
Flash Area Protection	X	X	X	X	X	X	X	X	X	X
Independent Watchdog Timer	X	X	X	X	X	X	X	X	X	X
Port Output Enable Module for GPT	X	X	X	X	X	X	X	X	X	X
SRAM Parity Error Check	X	X	X	X	X	X	X	X	X	X
Secure Crypto Engine 7								X	X	X
Secure Crypto Engine 5				X	X	X	X			
AES (128/256) & TRNG	X	X	X							

Anexo II

Precios obtenidos en la web de Digikey a fecha de 04/01/2019.

CFM Format (€/unit)	S124	S128	S1JA	S3A1	S3A3	S3A6	S3A7
1	4.48	5.33	3.92	10.71	8.54	6.13	10.22
10	3.997	4.758	3.92	9.733	7.685	5.514	9.392
25	3.597	4.282		9.002	7.001	5.024	9.002
160	3.277	3.901		8.272	6.318	4.533	7.932
320	2.958	3.52		7.542	5.805	4.166	7.542
640	2.654	3.159		7.056	5.293	3.798	7.056
1120	2.238	2.664		6.472	4.61	3.308	6.472
2560	2.126	2.531		6.228	4.439	3.185	6.228
5120	2.046	2.435				3.063	

CLK Format (€/unit)	S5D9	S5D5	S7G2
1	19.05	13.08	18.14
10	17.32	11.888	16.737
25	16.021	10.996	15.984
100	14.722	10.104	14.291
250	13.423	9.213	13.633
500	12.557	8.618	12.975
1000	11.517	7.905	12.223