|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **UNIVERSIDAD TECNOLÓGICA**  **NACIONAL**      **FACULTAD REGIONAL PARANÁ**  **Carrera: INGENIERÍA ELECTRÓNICA**  **Materia: informática II.**  **Comisión: “2R1”**  **Trabajo Práctico N.º: 2**  **Tema:** **Comunicación Serie – Red de Sensores.**    **Equipo de Cátedra:**   * **Adrián Lencina (Practica).** * **Sergio Burgos (Teoría).**     **Integrantes Grupo (n):**     * **Maili, Elías. (16996)** * **Wouterlood, Joaquín. (17123)** * **Rabe, Agustín. ()**     **Fecha de Realización: 15/11/23**    **Fecha de Entrega: 17/11/23**      **Universidad Tecnológica Nacional Facultad Regional Paraná**  **Trabajo Práctico N.º 2:**  **“Comunicación Serie – Red de Sensores”**   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **LEGAJO N°** | | **APELLIDO** | | | | | **NOMBRES** | | | | | | **Especialidad** | | **% de aporte al informe** | | | 16996 | | Maili | | | | | Elías Natanael | | | | | | 2R1 | | 33% | | | 17123 | | Wouterlood | | | | | Joaquín | | | | | | 2R1 | | 33% | | |  | | Rabe | | | | | Agustín | | | | | | 4R1 | | 33% | | |  | |  | | | | |  | | | | | |  | |  | | |  | | |  | | |  | | | | |  | | | Foto actual del alumno 5 | | Foto actual del alumno 6 | | ***Docente de teoría*** | | | | ***Día que cursa (Lu, Ma, Mi, Ju, Vi)*** | | | |  | ***Mañana / Tarde*** | | | ***Docentes del TPL*** | | | | | | **Sergio Burgos (teoría)** | | | | **Lunes y jueves** | | | |  | **Tarde** | | | **Adrián Lencina (Practica)** | | | | | |  | | | |  | | | |  |  | | | | | | | | | ***Fecha Realización TPL*** |  | | |  | ***Fecha de recepción***  *(carga el docente)* | | |  |  | ***responsable de entrega: e-mail:***  ***joaquinwouterlood@gmail.com*** | | | | | | | |

**Introducción:**

El presente trabajo práctico aborda dos áreas fundamentales en el desarrollo de sistemas embebidos: la comunicación serie y la implementación de una red de sensores basada en dispositivos ESP32 mediante el protocolo MQTT.

**Parte I: Comunicación Serie:**

En esta sección, se desarrolla una aplicación gráfica de escritorio utilizando la biblioteca Qt para controlar el encendido y apagado de un LED en la placa NodeMCU (ESP32) a través de comunicación serie por USB. La configuración de tiempos de encendido y apagado del LED es ajustable de forma independiente, permitiendo diversas combinaciones. La comunicación entre la aplicación y la placa se realiza mediante la clase QSerialPort y la UART0. Se emplea un protocolo que incluye encabezado, configuraciones, suma de comprobación y marca de finalización. La implementación en la placa se realiza mediante dos tareas y una máquina de estados finita.

**Parte II: Red de Sensores:**

En esta segunda parte, se implementa una red de sensores utilizando dispositivos ESP32 y el protocolo MQTT. Se desarrollan tres aplicaciones: un nodo de registro que transmite valores a un bróker MQTT, un nodo de prueba de escritorio para simular un nodo con LED y valores ajustables, y un panel de control que identifica y controla los nodos conectados al sistema. La comunicación se realiza mediante tópicos, y se propone un diccionario para organizar los mensajes. Se recomienda el uso del servidor MQTT emqx, aunque se brinda la flexibilidad de elegir otros servidores.

Estas dos partes proporcionan una experiencia integral en el desarrollo de sistemas embebidos, desde la configuración de comunicación serie hasta la implementación de redes de sensores basadas en MQTT, abarcando aspectos prácticos y teóricos de gran relevancia en el ámbito de la ingeniería electrónica.

**Contenido:**

**Parte I: Comunicación serie**

**Desarrollo, Problemas y Soluciones:**

\*Para el desarrollo mostraremos paso por paso como actúa el programa, para, poder seguir mejor su funcionamiento\*

Como primer paso, como vimos en la Introducción, debemos contar con una aplicación grafica(ventana), para poder manejar dos QHorizontalWidget, y setear los tiempos de encendido y apagado, esta cuenta con un botón de **Conectar** que sirve para conectar con el puerto deseado, uno de **Desconectar**, que sirve para desconectarlo del puerto, y por último uno de **salir** que obviamente es para salir de la app. Y un terminal de Control, donde podremos ver los valores enviados y recibido:



La aplicación empieza al tocar el botón conectar, este crea una ventana secundaria que mediante un combo Box, nos muestra los puertos disponibles para conectarnos:



Una vez seleccionado el puerto, se establecerá el puerto serie como lectura y escritura, para poder mandar y recibir información, código:



**Acciones del Método asociado al Botón Conectar:**

1. **Creación de una instancia de la clase Puertos:**

Se crea un objeto de la clase Puertos (que asumo que es una ventana o diálogo para seleccionar puertos). Esta instancia se llama misPuertos.

1. **Obtención de la lista de puertos disponibles:**

Se obtiene una lista de todos los puertos serie disponibles utilizando QSerialPortInfo::availablePorts()

1. **Iteración sobre la lista de puertos:**

Se itera sobre la lista de puertos usando un bucle foreach (que itera sobre cada elemento de la lista). En cada iteración, se comprueba si el nombre del puerto no está vacío y, en caso afirmativo, se agrega a un combo box (cb\_listaPuertos) en la ventana misPuertos.

1. **Mostrar la ventana misPuertos:**

Se muestra la ventana misPuertos mediante el método exec(). Esto bloquea la ejecución del código hasta que se cierra la ventana misPuertos. Si el usuario elige un puerto y hace clic en "Aceptar" (o algún botón similar), el bloque de código dentro de la condición if(misPuertos.exec()) se ejecutará.

1. **Cerrar el puerto serie si ya está abierto:**

Se verifica si el puerto serie (serie) ya está abierto. Si es así, se cierra utilizando serie->close().

1. **Configurar el puerto serie:**

Se establece el nombre del puerto serie (serie->setPortName()) con el nombre seleccionado en el combo box (misPuertos.ui->cb\_listaPuertos->currentText()).

Se actualiza la interfaz gráfica con información sobre el puerto seleccionado.

1. **Abrir el puerto serie:**

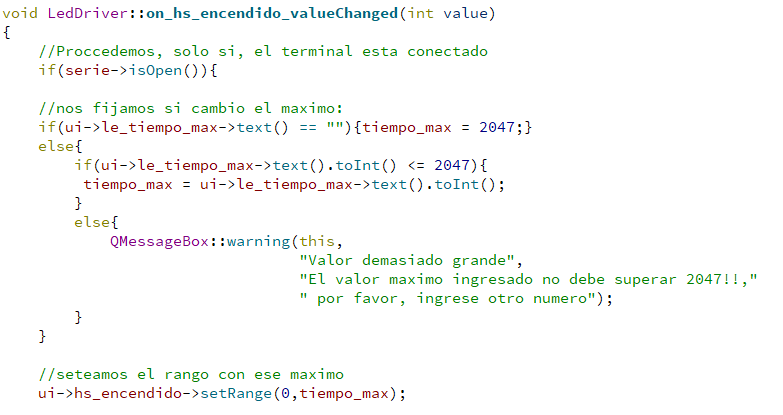
Se abre el puerto serie (serie->open(QIODevice::ReadWrite)), configurándolo para lectura y escritura.

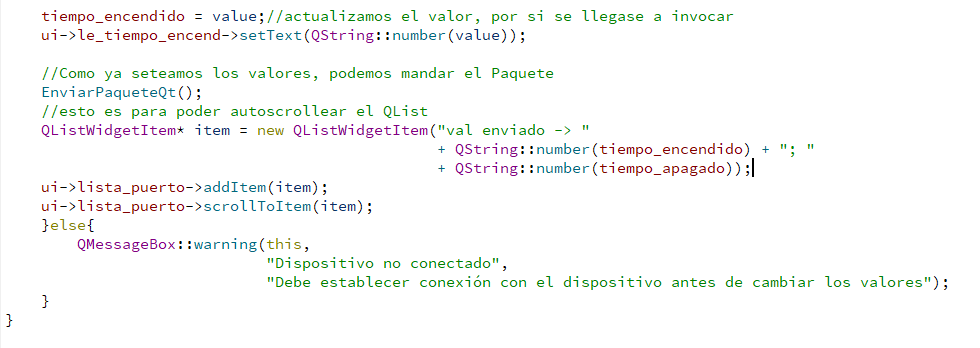
Básicamente, este método realiza la configuración necesaria y actualiza la interfaz gráfica cuando el usuario elige y confirma un puerto serie en la ventana **misPuertos**. Además, cierra el puerto si ya estaba abierto antes de configurarlo con el nuevo puerto seleccionado. Una vez elegido, el puerto deseado, la ventana se vera como la de abajo, lista para enviar datos y recibirlos



El segundo paso ahora es enviar el paquete de datos cada vez que se actualize el QSlider, para eso utilizamos el evento “valueChanged()” del QSlider, Este método está asociado a una señal que se desencadena cuando el valor de un QSlider (en este caso, hs\_encendido) cambia:

**Codigo:**





**Acciones del Método asociado al evento ValueChanged:**

1. **Verificar si el puerto serie está abierto:**

Se verifica si el puerto serie (serie) está abierto utilizando serie->isOpen().

1. **Configuración del valor máximo del tiempo:**

Se verifica si el campo de texto (ui->le\_tiempo\_max) que indica el tiempo máximo está vacío.

Si está vacío, se establece tiempo\_max en 2047 (un valor máximo específico).

Si no está vacío, se verifica que el valor no supere 2047. Si es así, se establece tiempo\_max en el valor ingresado.

Si el valor ingresado supera 2047, se muestra un mensaje de advertencia utilizando QMessageBox.

1. **Configuración del rango del QSlider:**

Se establece el rango del QSlider (ui->hs\_encendido) de 0 a tiempo\_max.

1. **Actualización de la interfaz gráfica:**

Se actualiza el valor en un campo de texto (ui->le\_tiempo\_encend) con el valor actual del QSlider.

Se asigna el valor actual del QSlider a la variable tiempo\_encendido.

1. **Envío del paquete y actualización de la lista:**

Se llama a la función EnviarPaqueteQt() para enviar un paquete que probablemente contiene información sobre el tiempo encendido.

Se crea un nuevo elemento para la lista (QListWidgetItem) que contiene información sobre el valor enviado.

Se agrega este elemento al QList (ui->lista\_puerto).

Se realiza un auto-scroll para que el elemento recién agregado sea visible.

1. **Manejo de advertencia si el puerto no está conectado:**

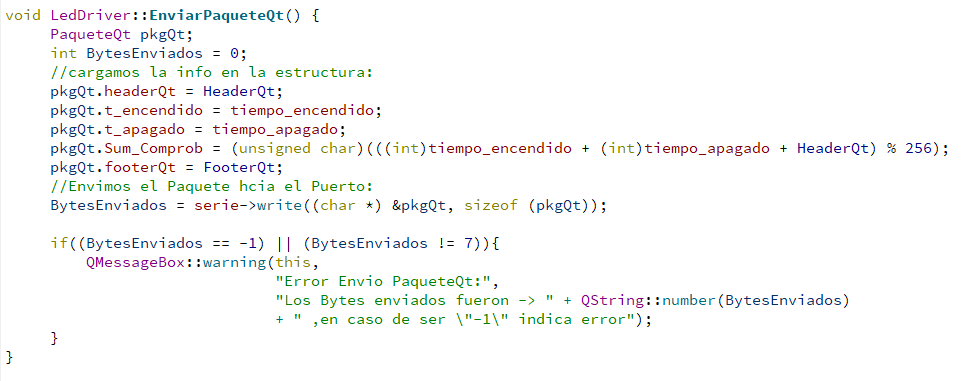
Si el puerto serie no está abierto, se muestra un mensaje de advertencia (QMessageBox) indicando al usuario que debe establecer conexión antes de cambiar los valores.

En resumen, este método realiza diversas operaciones relacionadas con la configuración del tiempo encendido, incluido el envío de un paquete y la actualización de la interfaz gráfica, siempre que el puerto serie esté abierto. Además, proporciona mensajes de advertencia si es necesario.

**\*Para el Slider del Tiempo de Apagado es exactamente lo mismo pero utilizando la variable global “tiempo\_apagado” en vez de “tiempo\_encendido”\***

Ahora para enviar el paquete entra en acción la función “EnviarPaqueteQt()”, que se encarga de armar el paquete con protocolo, y enviarlo por el puerto al ESP32

**Código:**



**Acciones del Método EnviarPaqueteQt():**

1. **Creación de la estructura del paquete:**

Se crea una instancia de la estructura PaqueteQt llamada pkgQt.

1. **Carga de información en la estructura:**

Se asigna a los campos de la estructura (pkgQt) los valores correspondientes.

El campo headerQt se establece en un valor previamente definido (HeaderQt).

Los campos t\_encendido y t\_apagado se establecen en los valores actuales de tiempo\_encendido y tiempo\_apagado respectivamente.

El campo Sum\_Comprob se calcula como la suma de tiempo\_encendido, tiempo\_apagado y HeaderQt, tomándose el resultado módulo 256 (operación que asegura que el resultado esté en el rango de un byte).

El campo footerQt se establece en un valor previamente definido (FooterQt).

1. **Envío del paquete hacia el puerto serie:**

Se utiliza el método serie->write() para enviar los bytes de la estructura pkgQt a través del puerto serie.

La función sizeof(pkgQt) se utiliza para determinar la cantidad de bytes que se deben enviar.

1. **Manejo de errores:**

Se verifica si la cantidad de bytes enviados (BytesEnviados) es -1 o no es igual a 7.

Si es -1, se muestra un mensaje de advertencia indicando un error en el envío.

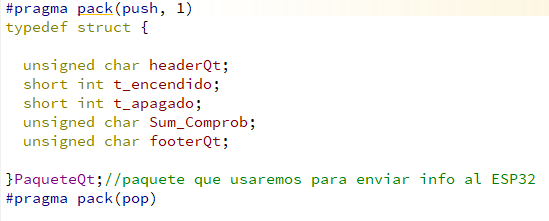
Si no es igual a 7, también se muestra un mensaje de advertencia.

En resumen, este método encapsula el proceso de construcción y envío de un paquete de datos a través del puerto serie, y proporciona manejo de errores para el caso de que algo salga mal durante el envío.

**Al implementar este paquete, nos ocurrió una de las primeras complicaciones**, como está aplicación debe contar con una parte de Qt, encargada de enviar un paquete de datos, con protocolo, que contenga 1 byte con el Header (Cabecera), 4 bytes con los dos valores de Tiempo encendido y apagado, un byte con la Suma de Comprobación (que será para saber si los bytes recibidos delante de esta son correctos), y un byte con el Footer o pie que indicara el final del paquete, pero, aquí ocurre el primer problema, ya que el paquete es de 7 bytes, pero cuando se envía por el puerto, no se manda la misma cantidad de bytes.

Esto es debido estructura que hemos definido, **PaqueteQt**, podría tener un tamaño mayor de lo que esperamos, debido a la alineación de la memoria. En algunos sistemas y compiladores, las estructuras se alinean en la memoria para mejorar el rendimiento. Esto significa que algunos bytes adicionales se pueden agregar a la estructura para asegurar que cada campo esté alineado en una dirección de memoria que sea más eficiente para acceder.

Para evitar este problema, podemos utilizar la directiva **#pragma pack** (o su equivalente específico del compilador, en este caso para C++, si estamos en C, podemos utilizar \_\_atribute\_\_((packed))) para desactivar el alineamiento o la alineación específica de la estructura.

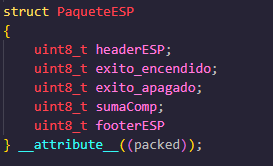


Esta directiva **#pragma pack** se utiliza para cambiar temporalmente la alineación de la estructura. En este caso, **#pragma pack (push, 1)** desactiva el alineamiento y **#pragma pack (pop)** restaura la configuración de alineación a la predeterminada.

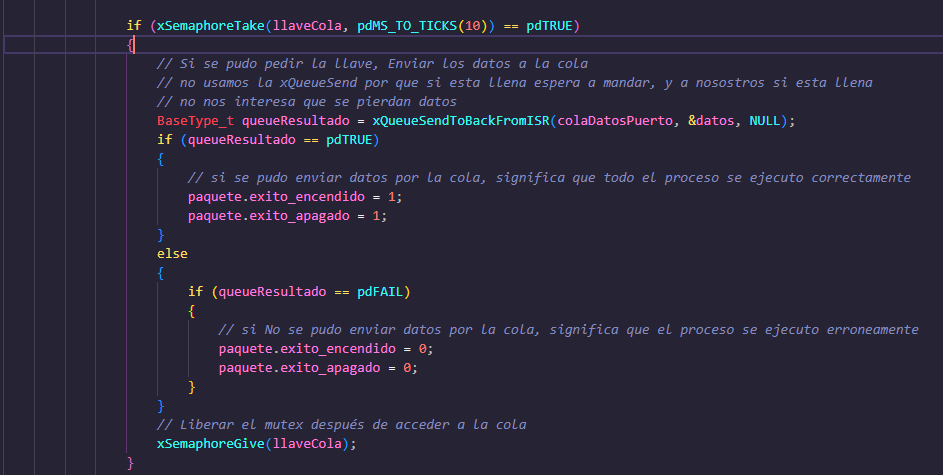
Con esto, puedes asegurarte de que la estructura tenga el tamaño esperado sin bytes adicionales debido al alineamiento en la memoria.

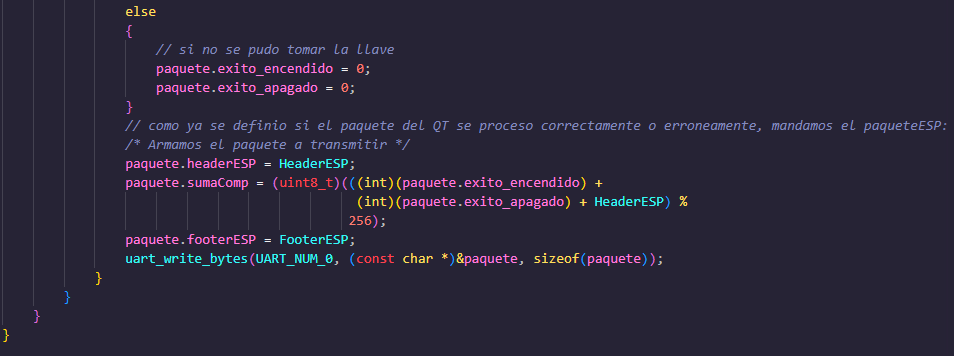
**Una vez enviado el paquete, pasamos a la parte del ESP32:**

En este caso la estructura para que no halla bytes de alineación, como estamos en C, podemos usar el “atribute”:



Codigo de la Tarea Puerto:





**Acciones de la tarea TareaPuerto():**

1. **Lectura del puerto serie:**

Lee un byte del puerto serie (UART) y verifica si la operación fue exitosa.

1. **Máquina de estados:**

Utiliza una máquina de estados finita implementada como un array de funciones (listaEstados) para procesar el byte leído y actualizar una estructura de estado (valorEstados).

1. **Construcción y envío del paquete:**

Carga los valores actualizados en una estructura (datos) y los envía a una cola (colaDatosPuerto) sin esperar, utilizando una tarea de interrupción (xQueueSendToBackFromISR).

Maneja la toma y liberación de un semáforo (llaveCola) para evitar condiciones de carrera al acceder a la cola.

Construye un paquete del ESP32 (paquete) con información sobre el éxito o fracaso del proceso.

Utiliza uart\_write\_bytes para enviar el paquete ESP por el puerto serie.

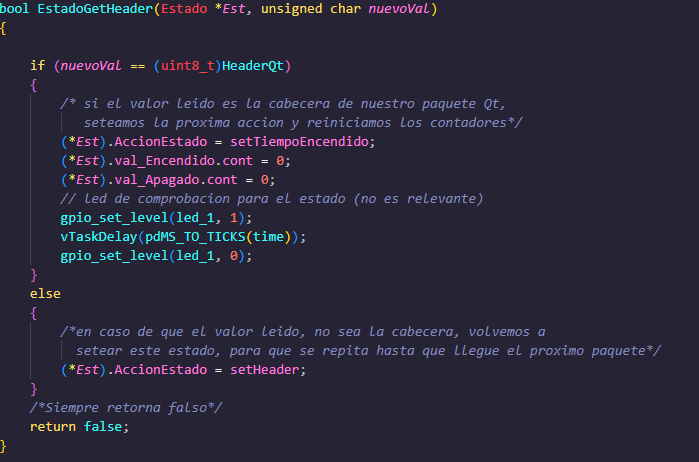
En resumen, esta tarea lee datos del puerto serie, los procesa mediante una máquina de estados, envía la información a una cola sin esperar para que la tarea TareaLed(), los pueda procesar y actualizar el tiempo de encendido y apagado de los leds, y si todo funciono y se procesó correctamente envía un paquete de respuesta por el puerto serie, todo dentro de un bucle infinito.

Aquí tenemos 2 puntos importantes, el envió de datos por la cola y la máquina de eventos.

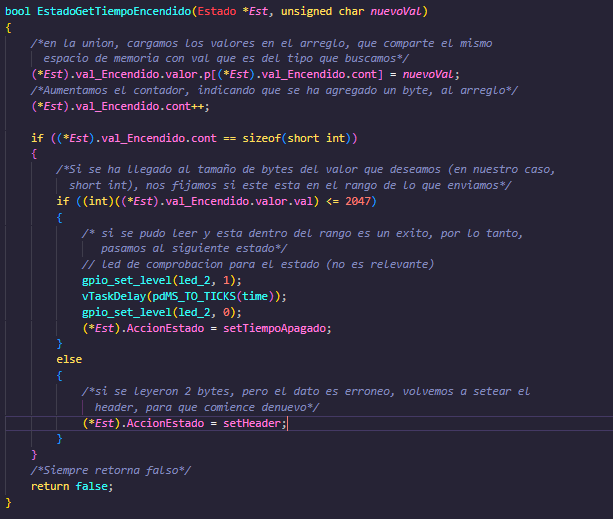
Pasaremos a explicar el funcionamiento y el codigo de forma breve, esta vez ayudándonos de los comentarios añadidos al codigo, para simplificar la explicación.

**Codigo de la implementación de la Maquina de Eventos:**

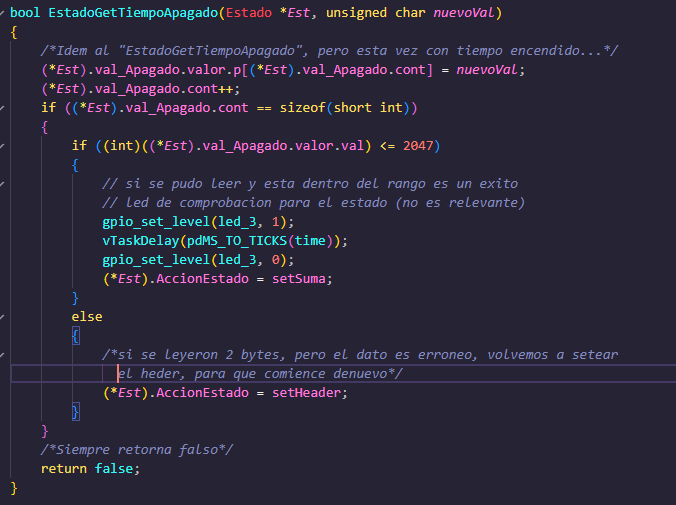
Recordemos que en la tarea puerto, debido al while(1) cada vez que lee un byte, llama a la máquina.

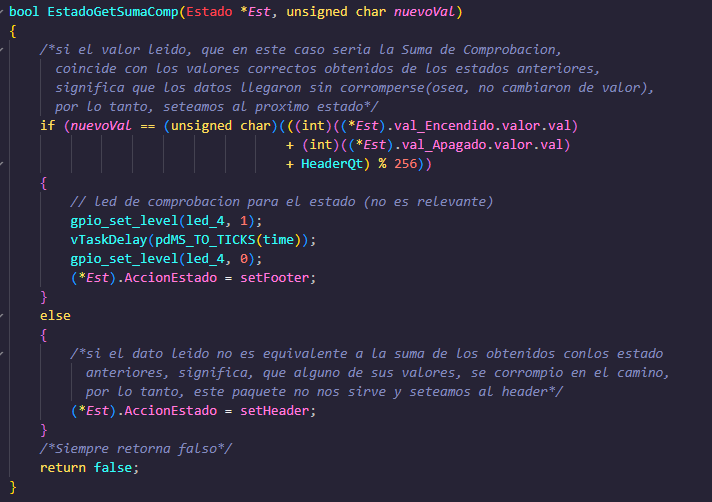
**Primer Estado:**

**Segundo Estado:**

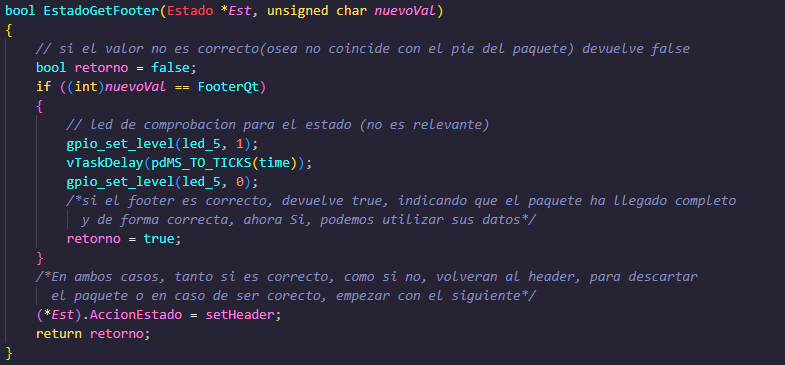


**Tercer Estado:**

**Cuarto Estado:**

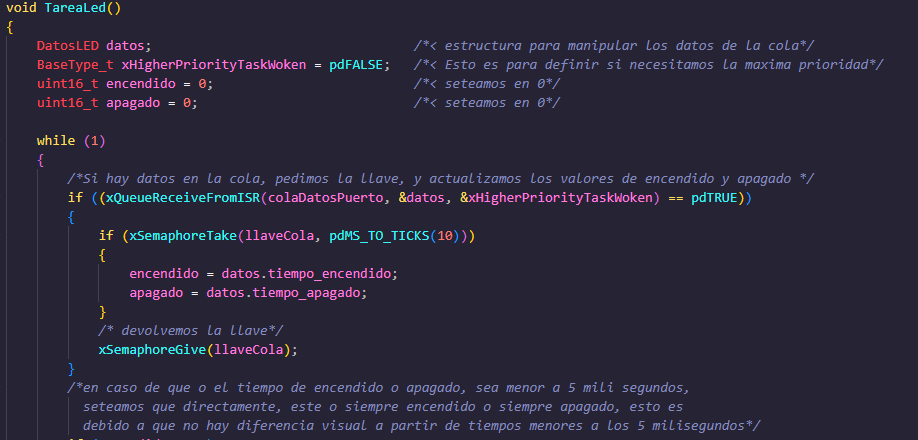


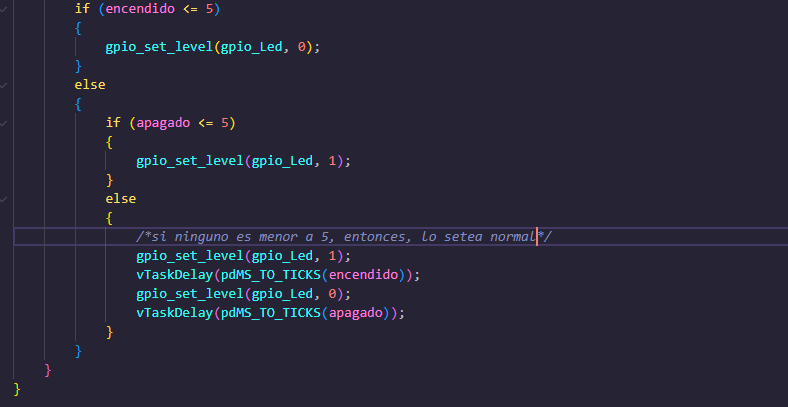
**Quinto y último Estado:**



Ya que vimos el funcionamiento de la maquina de estados, pasemos a ver como trabaja la tarea encargada del funcionamiento del led:

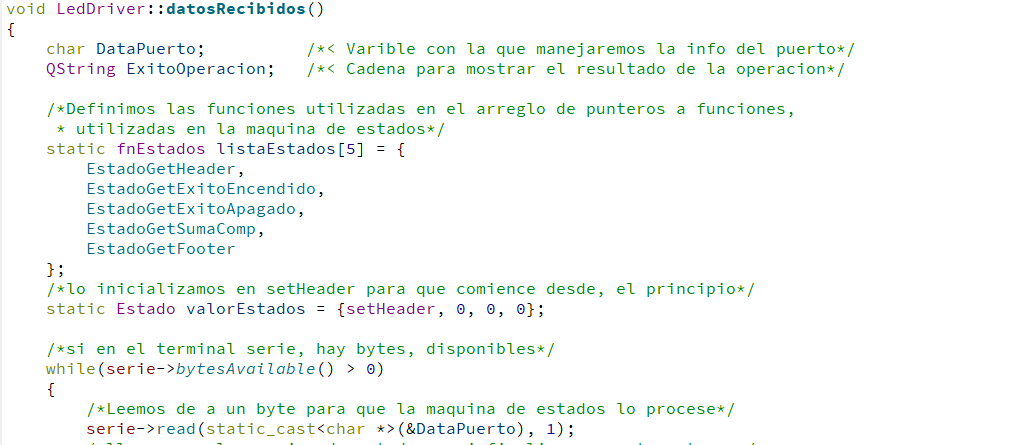
**\*En este caso también la explicación estará comentada en el codigo, debido a su simplicidad\***

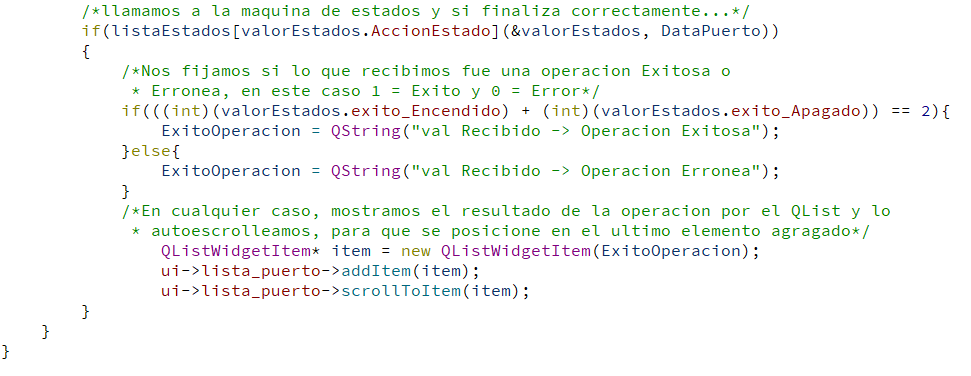
**Codigo de la implementación de la TareaLed():**



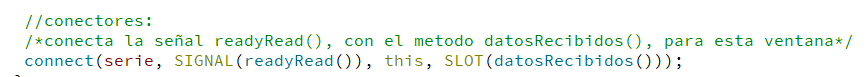
Ahora que ya vimos el funcionamiento de las tareas relacionadas con el ESP32, por último, nos falta ver el procesamiento del Paquete que envía el ESP32 al Qt, que también se hará a partir de una maquina de estados finita, y por lo tanto, su funcionamiento, será prácticamente el mismo, a excepción, de que no se usaran uniones, ya que todos los datos, son de un byte.

**\*Para tener más info sobre la máquina de eventos del Qt, consulte la documentación adicional, adjunta al TP (Carpeta “Documentación HTML”, archivo “Pagina LedDriver”)\***

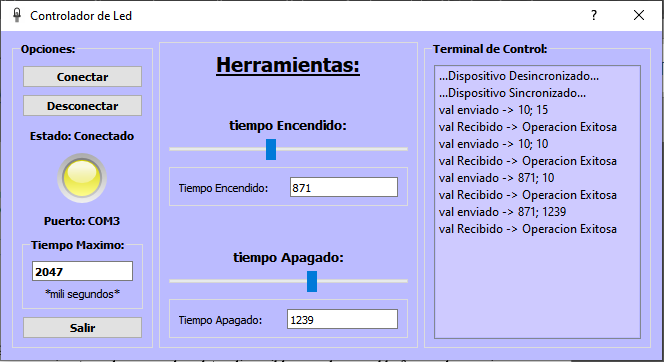




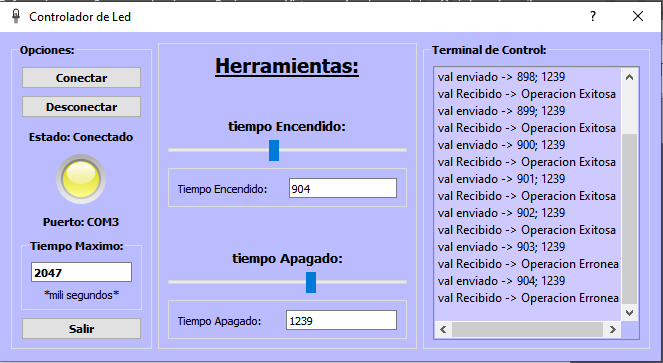
Una cosa importante a destacar, es que este Método o Slot, esta asociado a una señal llamada ReadyRead(), que se ejecuta, cada vez que hay datos disponibles para leer en el bufer, por lo que si nosotros estamos utilizando el terminal del puerto serie, significa que cuando halla datos disponibles en este, se ejecutara el método de DatosRecibidos(), la función que permite esto, es el connect():



Una vez establecidos todos estos pasos, la comunicación es correcta, y mandamos algunos valores, se verá así:



Si le enviamos muchos valores de golpe, la cola del ESP32, que utiliza para comunicarse de la TareaPuero(), a la TareaLed(), se llenará, y, por lo tanto, nos dará error al intentar configurar los nuevos tiempos que le llegaron:



\*En caso de que quisiéramos cambiar lo máximos de los Slider podríamos, cambiar el Tiempo Maximo\*

\*Al desconectarse se limpiara la terminal de Control de nuestra ventana\*

**Parte II: Comunicación MQTT**

Se requieren de dos aplicaciones, la primera es un panel de control que tiene la facultad de visualizar con una grafica explicita los valores recibidos por MQTT, y la segunda aplicación se trata de un “nodo”, imitación de un dispositivo conectado a un sensor capas de enviar o publicar mensajes en la red de transmisión MQTT. A estos fines, se desarrollaron las aplicaciones que se expondrán a continuación.

La primera aplicación (desde ahora panel de control) debe ser una aplicación con interfaz grafica para que por su puesto se pueda ver la gráfica. Además, se deben manejar otros parámetros como la comunicación con el servidor y con los nodos o dispositivos capaces de comunicar datos.

**Panel de Control:**

La aplicación se presenta con el siguiente formato.

Interfaz de usuario gráfica, Aplicación, Excel

Descripción generada automáticamente

Contiene en su interfaz distintos widgets propios de Qt y un widget externo a Qt, llamado QCustomPlot, utilizado para graficar.

Se presentan distintos labels y lineEdits para introducir información como el servidor o host a utilizar, el puerto dentro del host y un usuario y contraseña de ser requerido. Existen también botones cuyas funciones son: Conectar (inicia la conexión con el servidor), Desconectar (termina la conexión con el servidor, al iniciar la aplicación se mantiene deshabilitado), botones funcionales a Actualizar (actualiza la lista de dispositivos subscritos), Led On (emite una señal única, encendido del led), Led Off (emite una señal única, apagado del led) y Borrar (limpia la lista de dispositivos). En la parte derecha de la interfaz se ve una grafico vacío, un indicador de sensor (en principio vacío) y una segunda lista en la parte inferior (utilizada para mensajes de consola) junto a un botón de borrar.

Código:

Inicialización de la aplicación, declaración de las funciones intervinientes en la aplicación. Aclaración, durante la presentación del código se omitirán los comentarios pertenecientes a “Doxygen” (programa utilizado para generar archivos de información).

Archivo "dialog.h"

#ifndef DIALOG\_H

#define DIALOG\_H

#include <QDialog>

#include "qmqtt.h"

#include "qcustomplot.h"

#include <vector>

QT\_BEGIN\_NAMESPACE

namespace **Ui** { class **Dialog**; }

QT\_END\_NAMESPACE

using namespace QMQTT;

class **Dialog** : public QDialog

{

Q\_OBJECT

Client cliente;// Cliente MQTT para la comunicación con el servidor.

public:

**Dialog**(QWidget \*parent = nullptr);

~***Dialog***();

private slots:

// Métodos relacionados con las señales y ranuras de Qt.

void **connected**();

void **disconnected**();

void **error**(const QMQTT::ClientError);

void **subscribed**(const QString& topic, const quint8 qos);

void **unsubscribed**(const QString& topic);

void **published**(const QMQTT::Message& message, quint16 msgid=0);

void **pingresp**();

void **received**(const QMQTT::Message&);

// Métodos relacionados con la interfaz gráfica y la interacción del usuario.

void **on\_pb\_conectar\_clicked**();

void **on\_pb\_desconectar\_clicked**();

void **on\_pb\_ledon\_clicked**();

void **on\_pb\_ledoff\_clicked**();

void **on\_pb\_actualizar\_clicked**();

void **on\_pb\_borrarmsg\_clicked**();

void **on\_pb\_borrarconsola\_clicked**();

void **on\_list\_itemDoubleClicked**(QListWidgetItem \*item);

void **on\_list\_2\_itemEntered**();

private:

Ui::Dialog \*ui;// Interfaz gráfica generada por Qt.

QCustomPlot \*customPlot;// Gráfico personalizado para visualización.

std::vector <QByteArray> dispositivos;// Vector para almacenar información sobre dispositivos.

};

#endif // DIALOG\_H

Se incorpora la biblioteca QMQTT para manejar la comunicación con el servidor MQTT a utilizar. Utiliza un objeto ‘cliente‘ para gestionar la conexión, subscripciones y publicaciones de mensajes.

Se implementan funciones (‘slots’) para manejar eventos específicos, como la conexión exitosa (‘**connected** ()’), desconexión (‘**disconnected** ()’), errores (‘error()’), publicación (‘**published** ()’), recepción (‘**received** ()’) y otros métodos pertenecientes a la aplicación misma, como son: **on\_pb\_desconectar\_clicked**()**on\_pb\_conectar\_clicked**(), **on\_pb\_ledon\_clicked**(), **on\_pb\_ledoff\_clicked**(); que son funciones o métodos asociados a eventos ocurridos con la interfaz gráfica.

Además, se incluye la biblioteca "qcustomplot.h" para representar los datos dinámicamente y se declara un vector de <QByteArray> para almacenar información sobre los dispositivos.

Texto

Descripción generada automáticamenteEl archivo dialog.C contiene las implementación de las de las funciones antes mencionadas.

El constructor de la clase dialog.h, ‘Dialog::**Dialog**(QWidget \*parent)’, inicializa la interfaz gráfica (‘ui’) y establece las conexiones entre señales y slots para manejar eventos mqtt. Además, se configura el objeto ‘customPlot’ para la visualización del gráfico, se deshabilitan ciertos botones y se establecen controles iniciales, como, por ejemplo, el titulo al eje Y de la gráfica.

Texto

Descripción generada automáticamente

La función void Dialog::**connected**() se llama cuando la conexión con el servidor MQTT es exitosa. En primer lugar, notifica el éxito mediante la adición de un elemento (que es personalizado previamente) a la lista de mensajes de consola (‘list\_2’). En segundo lugar, se subscribe a los tópicos relevantes, como son: "/ej02/+/sensor" y "/ej02/id". En tercer lugar, habilita los botones que al inicio estaban deshabilitados.

Texto

Descripción generada automáticamente

La función void Dialog::**disconnected**() se ejecuta cuando se pierde o desconecta la conexión con el servidor MQTT, realizando las siguientes acciones: notifica la desconexión mediante la adición de un elemento (que es personalizado previamente) a la lista de mensajes (‘list\_2’); al contrario que ‘void Dialog::**connected**()’, deshabilita los botones y limpia la lista de dispositivos (‘list’).

Texto

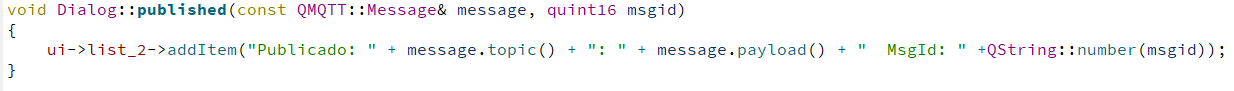
Descripción generada automáticamente con confianza media

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente Este slot se llama cuando se produce un error en la conexión MQTT, y notifica un error en la lista de consola, con un color rojo que resalta.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamenteSe ejecuta cuando la suscripción a un tópico MQTT es exitosa, y añade un elemento a la lista de consola () con un color verde.

 Este slot se ejecuta cuando se cancela la suscripción a un tópico, adicionando un elemento a la lista de consola () con el tópico referido.

El slot void Dialog::**published**(const QMQTT::Message& message, quint16 msgid) se ejecuta al publicarse un mensaje, y la acción asociada es la adición de un elemento, mostrando las características de dicho mensaje, a la lista de consola ().

Imagen que contiene Gráfico

Descripción generada automáticamenteEsta función se invoca en respuesta a un mensaje PINGREQ enviado al servidor MQTT. En este caso no se ha añadido lógica, pero puede ser personalizada a conveniencia.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

La función void Dialog::**received**(const Message &message) se encarga del procesamiento de los mensajes recibidos del servidor MQTT. Realiza las siguientes acciones:

* Desplaza la lista de mensajes hacia abajo para mostrar los mensajes más recientes.
* Procesa mensajes de identificación de dispositivos (“/ej02/id”) y agrega el dispositivo nuevo a la lista, se debe cumplir la condición de que el tópico debe ser el mencionado.
* Obtiene el dispositivo seleccionado en el combo box.
* Si esta seleccionado “ninguno”, elimina el grafico existente y ajusta los rangos de los ejes.
* Si el tópico del mensaje coincide con el formato “/ej02/[dispositivo]/sensor”, actualiza el grafico correspondiente.
* Crea un nuevo grafico si no existe y agrega datos al grafico existente.
* Texto

  Descripción generada automáticamenteFinalmente, ajusta los rangos de los ejes y vuelve a trazar el grafico para reflejar los cambios.

Texto

Descripción generada automáticamente El destructor de la clase ‘Dialog’ se encarga de realizar las acciones necesarias antes de destruir una instancia de la clase. En primer lugar, desconecta del servidor MQTT si la conexión está establecida y luego libera la memoria asociada con la interfaz gráfica (‘ui’).

Este método maneja el evento de clic en el botón de conexión (‘**pb\_conectar**’). Configura los parámetros de conexión del cliente MQTT (‘cliente’) con la información proporcionada por el usuario. Intenta establecer una conexión con el servidor, luego habilita el botón de desconexión y por último desplaza la lista de consola hacia abajo, para mostrar los cambios.

Texto

Descripción generada automáticamente

El método maneja el evento de clic en el botón de desconexión (‘**pb\_desconectar**’). En primer lugar, desplaza la lista de consola hacia abajo, desconecta el cliente MQTT (‘cliente’) del servidor, borra la gráfica actual, actualiza el elemento seleccionado del combo box y deshabilita el botón Desconectar.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente Este método maneja el evento de clic sobre el botón de actualización (‘**pb\_actualizar**’). En primer lugar, verifica la conexión con el servidor MQTT antes de realizar acciones. Limpia y reinicia elementos de la interfaz grafica y datos relacionados con la visualización y dispositivos. Por último, publica un mensaje de pedido de IDs de dispositivos (‘getid’).

Texto

Descripción generada automáticamente

Este método maneja el evento de clic en el botón de Led On (‘**pb\_ledon**’). Verifica la conexión con el servidor MQTT y si se ha seleccionado un dispositivo antes de publicar el comando “ledOn”. Construye el tópico MQTT en función del dispositivo seleccionado. Por último, publica un mensaje en el tópico construido con el payload “ledOn”. Si no se ha seleccionado ningún dispositivo, se crea un mensaje de error y se muestra en la lista de consola “list\_2”.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Ídem al método anterior, publica un comando en el tópico construido a partir de un elemento seleccionado de la lista de dispositivos, “list”, con el payload de “ledOff”. Si no se ha seleccionado ningún dispositivo, añade a la lista de comandos un error.

Texto

Descripción generada automáticamente

Los siguientes métodos están asociados a los botones de la interfaz grafica cuyas funciones son:

* Limpiar la lista de comandos.
* Limpiar la lista de dispositivos.
* Manejar el evento de doble clic sobre un elemento de la lista de dispositivos, dicho elemento de establecerá como actual dispositivo en el combo box, graficando nuevamente.
* Manejar el evento de entrada de un elemento en la lista de consola “list\_2”, utilizando la función ‘scrollToBottom( )’.

Ejecución del programa:

Al iniciarse el programa se debe hacer clic sobre el botón de Conectar, a continuación, y si no se ha producido ningún error, ya que esta por defecto un servidor de prueba como ‘test.mosquitto.org’ y un puerto de defecto ‘1883’, se mostrará en la consola de mensajes lo siguiente:

Interfaz de usuario gráfica, Aplicación, Excel

Descripción generada automáticamente

Entonces el botón de Desconectar se habilita, junto con los botones de Actualizar y comandos.

Si hay dispositivos conectados al servidor y suscritos a los tópicos correspondientes, se relevaran haciendo clic sobre el botón Actualizar, como se muestra en el ejemplo:

Interfaz de usuario gráfica, Aplicación, Excel

Descripción generada automáticamente

Se puede ver que se añadió a la lista de consola la publicación de un mensaje de pedido de IDs, y que en la lista de dispositivos esta relevado un dispositivos con identificador ‘515’.

Al hacer doble clic sobre el dispositivo 515 se actualiza la grafica y empezara a mostrar los datos que se reciben a través del tiempo de dicho dispositivo.

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

Así también, se pueden enviar al dispositivo seleccionado los mensajes o comandos “Led On” y “Led Off”. Si no ocurre ningún problema, se mostrará en la lista de consola los siguientes mensajes:

Interfaz de usuario gráfica, Aplicación, Tabla, Excel

Descripción generada automáticamente

La segunda aplicación requerida en el ejercicio 2 del trabajo practico es un nodo de pruebas.

**Nodo de Prueba:**

Se recreo una aplicación grafica que permite la conexión a un panel servidor MQTT, y que es compatible con una aplicación Panel de Control, aceptando el mismo protocolo de conexión y comunicación.

En similitud a la aplicación Panel de Control, el Nodo posee una aplicación hecha con una clase ‘dialog’ y usa los métodos ofrecidos por la biblioteca “qmqtt.h” para la gestión con un instancia de tipo “QMQTT::Client” de dicha clase.

Imagen que contiene Gráfico

Descripción generada automáticamente

Por defecto, al igual que Panel de Control, algunos campos traen consigo datos preestablecidos, como son el servidor, el puerto y un ID único y aleatorio. También al principio se encuentra deshabilitado el botón Desconectar y desmarcada la casilla “Debug”.

Archivo “dialog.h”:

Definiciones de slots y métodos propios.

#ifndef DIALOG\_H

#define DIALOG\_H

#include <QDialog>

#include "qmqtt.h"

#include "qmqtt\_client.h"

#include <QTimer>

QT\_BEGIN\_NAMESPACE

namespace **Ui** { class **Dialog**; }

QT\_END\_NAMESPACE

using namespace QMQTT;

class **Dialog** : public QDialog

{

Q\_OBJECT

public:

**Dialog**(QWidget \*parent = nullptr);

~***Dialog***();

//Cliente de la clase QMQTT.

Client cliente;

private slots:

void **on\_pb\_conectar\_clicked**();

void **connected**();

void **disconnected**();

void **error**(const QMQTT::ClientError);

void **subscribed**(const QString& topic, const quint8 qos);

void **unsubscribed**(const QString& topic);

void **published**(const QMQTT::Message& message, quint16 msgid=0);

void **pingresp**();

void **msgperiodico**(); //Realiza acciones periódicas en respuesta al temporizador.

void **on\_cb\_debug\_toggled**(bool checked);

void **on\_pb\_desconectar\_clicked**();

void **on\_botoncosolaborrar\_clicked**();

private:

Ui::Dialog \*ui;

QTimer \*timer; //Temporizador.

QString id1;

};

#endif // DIALOG\_H

Implementaciones de los métodos mencionados en el archivo “dialog.h”:

Texto

Descripción generada automáticamenteConfiguración de la interfaz de usuario (‘ui’).

Se crea un objeto ‘QTimer’ llamado ‘timer’. Este temporizador se utilizará para realizar acciones periódicas, como el envió de mensajes MQTT. A su vez, este ‘timer’ se conecta con la señal timeout() del temporizador a la ranura SLOT(msgperiodico()), que se encargara de realizar acciones periódicas.

También se dan configuraciones iniciales como la elección de un id aleatorio, la deshabilitación de botones, el establecimiento de una imagen (‘pixmap’) como defecto y el ocultamiento de dos ‘labels’ de nombres “label\_publicando” y “label\_valorpublicado”.

Texto

Descripción generada automáticamenteMétodo void Dialog::**on\_pb\_conectar\_clicked**()

La función **on\_pb\_conectar\_clicked**() es un slot de respuesta al evento de clic en el botón de conectar (‘**pb\_conectar**’). Su objetivo principal es configurar y establecer la conexión con el servidor MQTT cuando se activa el botón. En primer lugar, se verifica que los campos de entrada para el puerto, ID y servidor no estén vacíos, y que el ID sea mayor a 5. Si esta condición no se cumple, se muestra un mensaje de error y se interrumpe el proceso de conexión.

En segundo lugar, se configura el nombre del host del cliente MQTT con el valor ingresado en el campo del servidor, además, se configura el nombre de usuario y contraseña del cliente MQTT si se proporcionan en los campos correspondientes.

En tercer lugar, se configura el puerto del cliente con el valor ingresado en el campo del puerto. Además, se imprime por consola de depuración el valor del estado actual de la conexión.

En cuarto lugar, se intenta establecer la conexión con el servidor.

Luego se inicia un temporizador con un intervalo de 2500 [ms] y se actualizan las propiedades de los botones de conexión y desconexión.

Slot void Dialog::**connected**()

Texto

Descripción generada automáticamente

El slot **connected**() se activa en respuesta a la señal de conexión exitosa emitida por el cliente MQTT. Su propósito es encargarse de realizar las suscripciones necesarias una vez que la conexión al servidor MQTT ha sida establecida con éxito. Se debe obtener del campo de entrada el valor del ID y luego concatenarlo para obtener el tópico correcto. Ambas suscripciones se dan con QoS de 2.

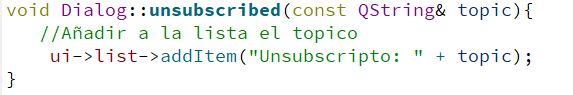
void Dialog::**on\_pb\_desconectar\_clicked**()

Texto

Descripción generada automáticamente

La función **on\_pb\_desconectar\_clicked**() es un slot que se activa al hacer clic sobre el botón Desconectar. En detalle realiza las siguientes acciones:

* Obtiene y muestra por consola de depuración el estado actual de la conexión del cliente MQTT.
* Verifica si el cliente esta actualmente conectado antes de intentar desconectarlo.
* Se realiza un desplazamiento hacia abajo en la lista para mostrar los mensajes mas recientes.
* Se cancelan las suscripciones a los tópicos MQTT “/ej02/cmd” y “/ej02/"+id1+"/cmd”.
* Se desconecta el cliente MQTT del servidor.
* Se verifica y muestra por consola si el cliente ha sido completamente desconectado.
* Se habilita el botón de conectar y se deshabilita el botón de desconectar. Además, se detiene el temporizador ‘timer->stop()’.



La función **unsubscribed**(const QString& topic) es un slot que se activa cuando el cliente MQTT se ha desvinculado (unsubscribed) de un tópico. Este slot añade a la lista de mensajes un mensaje que muestra el tópico al que se ha desvinculado.

void Dialog::**disconnected**()

Captura de pantalla de un celular

Descripción generada automáticamente

El slot **disconnected**() se ejecuta cuando el cliente se ha desconectado del servidor. Esta función añade un elemento a la lista oculta elementos de la interfaz relacionados con la publicación de datos.

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

El slot se activa cuando el cliente MQTT recibe un mensaje. Realiza las siguientes acciones:

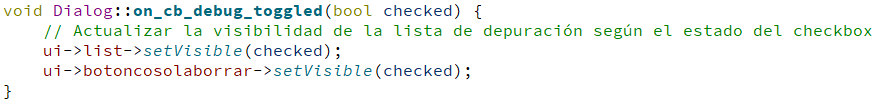
* Se extraen el comando y el tópico del mensaje, y se obtiene el ID del objeto ‘ui->leid’. Además, se crea una cadena ‘topicocmdled’ para comparar con el tópico del mensaje y determinar si es un comando Led específico para este dispositivo/aplicación.
* Si el tópico del mensaje coincide con el tópico esperado para comandos de Led, se agrega un elemento a la lista con detalles del mensaje y dependiendo del comando recibido (“ledOn” o “ledOff”), se actualiza el estilo y la imagen en la interfaz grafica para indicar el estado del Led.
* Si el tópico del mensaje es “/ej02/cmd” y el comando “getId”, se realiza un conjunto de acciones: se agrega un elemento a la lista con detalles del mensaje y se envía una respuesta al servidor MQTT con el ID actual (‘id1’) en el tópico “/ej02/id”.
* Si el comando no es reconocido (“ledOn”, “ledOff” o “getId”), se muestra un mensaje en la lista indicando que el comandos es desconocido.

Texto

Descripción generada automáticamente

El slot **msgperiodico**( ) es una función que se ejecuta periódicamente a través de un temporizador (‘’). Realiza las siguientes acciones:

* La función verifica si el cliente MQTT (‘cliente’) está conectado al servidor MQTT antes de realizar acciones. Si no está conectado, no se realiza ninguna operación.
* Se muestran dos elementos de la interfaz gráfica (‘label\_publicando’ y ‘label\_valorpublicado’). Estos elementos indican visualmente que se está realizando una publicación periódica de datos.
* Se construye un mensaje MQTT con el tópico ‘’ y el valor actual del dial. El valor del dial se obtiene mediante ‘’.
* El mensaje se publica en el servidor MQTT a través del cliente.



La función **on\_cb\_debug\_toggled**(bool checked) es un slot que se activa cuando el estado del checkbox cambia. La función toma el parámetro booleano ‘checked’, que representa el estado actual del checkbox. Si ‘checked’ es ‘true’, significa que el checkbox esta marcado; si es ‘false’, significa que el checkbox esta desmarcado. La visibilidad de dos elementos de la interfaz gráfica se actualiza según el estado del checkbox, estos son la lista de mensajes (‘list’) y el botón de borrar mensajes de la lista (‘botoncosolaborrar’).

Texto

Descripción generada automáticamente

La función **published**(const QMQTT::Message& message, quint16 msgid) se encarga de manejar la respuesta después de publicar un mensaje MQTT. Realiza las siguientes acciones:

* Publica un mensaje en la consola de depuración, indicando que se está publicando algo.
* Se verifica si el tópico del mensaje es ‘/ej02/"+id1+"/sensor’. Si es así, se realiza una acción especifica: se establece el texto del ‘ui->label\_valorpublicado’ con información específica relacionada al tópico del sensor y se imprime un mensaje de depuración indicando que lo que se publica pertenece al sensor.
* Si el tópico no es ‘/ej02/"+id1+"/sensor’, se realizan una acción diferente: se crea un elemento de lista con información sobre la publicación, se personaliza con un color especifico y se agrega la lista.

Texto

Descripción generada automáticamente

La función **error**(const QMQTT::ClientError error) maneja eventos de error del cliente MQTT y agrega mensajes de error a la lista en la interfaz de usuario. El error ‘20’ no esta considerado en esta aplicación, debido a que es un error al cual no se le ha encontrado solución ni información adicional en la documentación del proveedor de la biblioteca, y está relacionado en primer lugar, según las investigaciones, al tiempo de respuesta del servidor. El grupo de trabajo optó por omitir el error en esta implementación. Si el error es uno distinto al mencionado, esta lógica imprime un mensaje en la lista de color rojo, junto a especificaciones propias del error encontrado.

Ejecución del programa:

Una vez ejecutado el programa se puede iniciar la conexión al servidor de prueba establecido por defecto haciendo clic en Conectar y se puede extender la lista de mensajes de la consola marcando el checkbox ‘Debug’.

Interfaz de usuario gráfica

Descripción generada automáticamente

En la consola de mensajes se reportarán las suscripciones realizadas, y debajo del dial se dejarán ver los labels de publicación periódica, dado que ya se están publicando los datos del dial.

Si se desea terminar la conexión se presiona en Desconectar y aparecerán en la consola mensajes que reportan la finalización de las suscripciones.

Interfaz de usuario gráfica

Descripción generada automáticamente

Comando getId: Cuando se reciba el comando “getId” desde el panel de control se añadirá a la lista el mensaje recibido y la respuesta publicada.

Interfaz de usuario gráfica

Descripción generada automáticamente

Comandos Led: Cuando desde el panel de control se envía un comando led, se produce un cambio en la imagen que representa el estado del Led, así también como en la propia paleta de colores de la interfaz.

“ledOn”: una paleta mas clara, cambio en la imagen del led por una de encendido.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

“ledOff”: una paleta más oscura.

Interfaz de usuario gráfica

Descripción generada automáticamente

**Nodo Esp32**

Para realizar el nodo con un esp 32 que envíe una señal senoidal y sea capaz de encender y apagar el led incorporado en la misma placa desde el administrador nos basamos en el ejemplo 6 de la clase 23 de teoría en el cual se conecta por medio de un servidor mqtt con el mismo.

static void mqtt\_event\_handler(void \*handler\_args, esp\_event\_base\_t base, int32\_t event\_id, void \*event\_data)

{

    char topic[64];

    char data[64];

    ESP\_LOGD(TAG, "Event dispatched from event loop base=%s, event\_id=%" PRIi32 "", base, event\_id);

    esp\_mqtt\_event\_handle\_t event = event\_data;

    esp\_mqtt\_client\_handle\_t client = event->client;

    int msg\_id;

    switch ((esp\_mqtt\_event\_id\_t)event\_id) {

    case MQTT\_EVENT\_CONNECTED:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_CONNECTED");

        actClient = client;

       // msg\_id = esp\_mqtt\_client\_publish(client, "/ej02/id", "8", 0, 2, 0);

       // ESP\_LOGI(TAG, "sent publish successful, msg\_id=%d", msg\_id);

        msg\_id = esp\_mqtt\_client\_subscribe(client, "/ej02/cmd", 2);

        ESP\_LOGI(TAG, "sent subscribe successful, msg\_id=%d", msg\_id);

         msg\_id = esp\_mqtt\_client\_subscribe(client, topicid, 2);

        ESP\_LOGI(TAG, "sent subscribe successful, msg\_id=%d", msg\_id);

        msg\_id = esp\_mqtt\_client\_subscribe(client, "/ej02/id", 2);

        ESP\_LOGI(TAG, "sent subscribe successful, msg\_id=%d", msg\_id);

        msg\_id = esp\_mqtt\_client\_unsubscribe(client, "/ej02/id");

        ESP\_LOGI(TAG, "sent unsubscribe successful, msg\_id=%d", msg\_id);

        mqttOk = true;

        break;

    case MQTT\_EVENT\_DISCONNECTED:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_DISCONNECTED");

        mqttOk = false;

        break;

    case MQTT\_EVENT\_SUBSCRIBED:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_SUBSCRIBED, msg\_id=%d", event->msg\_id);

        msg\_id = esp\_mqtt\_client\_publish(client, "/ej02/cmd", "data", 0, 2, 0);

        ESP\_LOGI(TAG, "sent publish successful, msg\_id=%d", msg\_id);

        break;

    case MQTT\_EVENT\_UNSUBSCRIBED:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_UNSUBSCRIBED, msg\_id=%d", event->msg\_id);

        break;

    case MQTT\_EVENT\_PUBLISHED:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_PUBLISHED, msg\_id=%d", event->msg\_id);

        break;

    case MQTT\_EVENT\_DATA:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_DATA");

        strncpy(topic, event->topic, min(63, event->topic\_len));

        strncpy(data, event->data, min(63, event->data\_len));

        printf("TOPIC=%s\r\n", topic);

        printf("DATA=%s\r\n", data);

       // Asegúrate de que la cola esté correctamente inicializada

        if (colaDatos != NULL) {

            if (xQueueSend(colaDatos, data, 0) != pdPASS) {

                printf("No se pudo encolar el dato. La cola está llena.\n");

            }else{printf("encolado.\n");}

        } else {

            printf("La cola no está inicializada.\n");

        }

        break;

    case MQTT\_EVENT\_ERROR:

        ESP\_LOGI(TAG, "MQTT\_EVENT\_ERROR");

        if (event->error\_handle->error\_type == MQTT\_ERROR\_TYPE\_TCP\_TRANSPORT) {

           ESP\_LOGI(TAG, "Last errno string (%s)", strerror(event->error\_handle->esp\_transport\_sock\_errno));

        }

        break;

    default:

        ESP\_LOGI(TAG, "Other event id:%d", event->event\_id);

        break;

    }

}

Usamos esta función para subscribirnos a los tópicos necesarios y además encolar los datos que nos llegaban de los tópicos para así luego poder interpretarlos en una tarea más adelante.

void taskPublish(void \*param){

    char toSend[128];

    float cont = 1;

    float f;

    while(1){

        vTaskDelay(2500 / portTICK\_PERIOD\_MS);

        if(mqttOk){

            f=sin(cont);

            sprintf(toSend, "%f", f);

            esp\_mqtt\_client\_publish(actClient, topicsensor, toSend, 0, 2, 0);

            cont+=0.1;

        }

    }

}

Esta task se encarga de publicar datos en el tópico “topicsensor” en este caso una función senoidal con un paso de 0.1

void interprete(void \*cola)

{

    char dataReceived[64];

    QueueHandle\_t \*aux = (QueueHandle\_t \*)cola;

    while (1)

    {

        if (xQueueReceive(\*aux, dataReceived, 0))

        {

            if (strcmp(dataReceived, "ledOn") == 0) {

                printf("recibido: %s (es 'on')\r\n", dataReceived);

                ledOn();

            }

            else if (strcmp(dataReceived, "ledOff") == 0) {

                printf("recibido: %s (es 'off')\r\n", dataReceived);

                ledOff();

            }

            else if (strcmp(dataReceived, "getId") == 0) {

                esp\_mqtt\_client\_publish(actClient, "/ej02/id", ID, 0, 2, 0);

                printf(" Id publicado: %s \r\n", ID);

            }

            else {

                printf(" recibido: %s (no es 'on')\r\n", dataReceived);

            }

        }

    }

}

Esta función desencola los datos encolados en la primera función aquí descrita y determina que hacer si prender o apagar el led de la paca o publicar el dato del Id del nodo en caso de llegar “getId” para registrarse en el administrador. Este mismo id se define en la siguiente función como un numero entre 1 y 5

int generateRandomNumber() {

 srandom(time(NULL));

     int new\_value = 1 + esp\_random() % 5; // Genera un número aleatorio entre 1 y 5

    return new\_value;

}

void app\_main(void)

{

    esp\_mqtt\_client\_config\_t mqtt\_cfg = {

        .broker.address.uri = "mqtt://192.168.100.127",

        .credentials.username = "esp32esp32",

        .credentials.authentication.password = "esp32esp32",

        //.broker.address.uri = "mqtt://test.mosquitto.org:1883",

       // .broker.verification.certificate = (const char \*)hivemq\_pem\_start,

        //.broker.verification.certificate\_len  = hivemq\_pem\_end - hivemq\_pem\_start,

    };

    hardware\_init();

    esp\_err\_t ret = nvs\_flash\_init();

    if (ret == ESP\_ERR\_NVS\_NO\_FREE\_PAGES || ret == ESP\_ERR\_NVS\_NEW\_VERSION\_FOUND) {

      ESP\_ERROR\_CHECK(nvs\_flash\_erase());

      nvs\_flash\_init();

    }

    wifi\_init\_sta();

     colaDatos = xQueueCreate(MAX\_QUEUE\_SIZE, sizeof(char) \* 64);

     int new\_value = generateRandomNumber();

    snprintf(ID, ID\_SIZE, "%d", new\_value);

    snprintf(topicid, TOPIC\_SIZE, "/ej02/%s/cmd", ID);

    snprintf(topicsensor, TOPIC\_SIZE, "/ej02/%s/sensor", ID);

    printf("ID asignado: %d\n" ,new\_value);

    esp\_mqtt\_client\_handle\_t client = esp\_mqtt\_client\_init(&mqtt\_cfg);

    /\* The last argument may be used to pass data to the event handler, in this example mqtt\_event\_handler \*/

    esp\_mqtt\_client\_register\_event(client, ESP\_EVENT\_ANY\_ID, mqtt\_event\_handler, NULL);

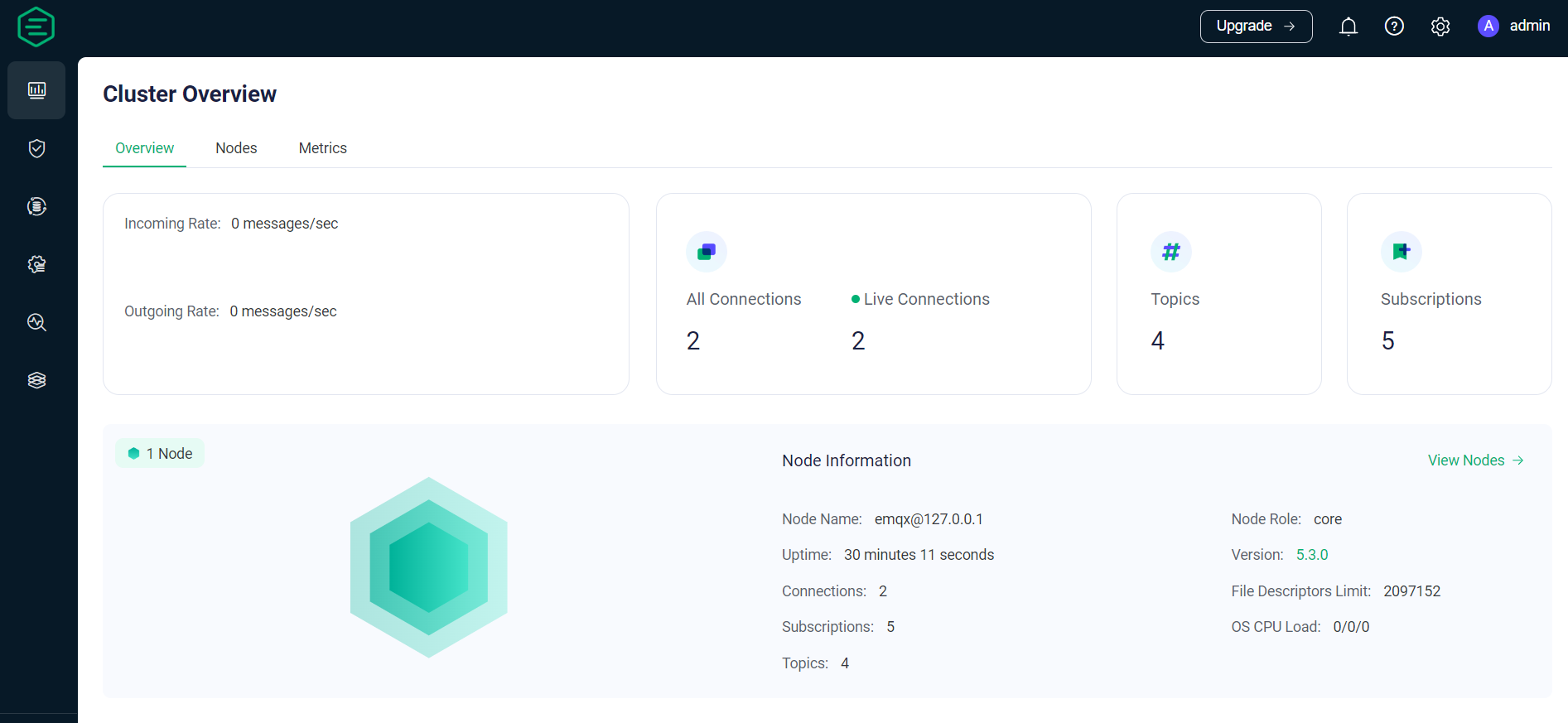
    esp\_mqtt\_client\_start(client);

    xTaskCreate(taskPublish, "Publish Task", configMINIMAL\_STACK\_SIZE + 4096, NULL, 1, NULL);

    xTaskCreate(interprete, "InterpretaValor", configMINIMAL\_STACK\_SIZE + 4096, &colaDatos, 2,NULL);

}

En el main primero empezamos configurando el servidor al cual nos conectaremos el mismo es el ip de mi computadora en el cual ejecuto el servidor emqx, luego se inicializa el hardware para poder accionar el led, luego definimos la cola que almacena los datos a recibir, después el id con la función antes descripta y lo actualizamos en las variables de tópicos. Para luego llamar a las funciones de mqtt y las task descriptas anteriormente



En esta imagen se puede apreciar el servidor funcionando que escogimos para realizar este trabajo.

**Conclusión del Trabajo:**

En el transcurso de este trabajo práctico, se ha abordado de manera integral el desarrollo de sistemas embebidos y aplicaciones de escritorio para el control de dispositivos IoT basados en ESP32. La implementación de una comunicación serie mediante UART y la construcción de una red de sensores utilizando el protocolo MQTT han sido los pilares fundamentales de este proyecto.

En la "Parte I: Comunicación serie", se diseñó una aplicación gráfica en Qt para configurar y controlar el encendido y apagado de un LED en una placa NodeMCU (ESP32) a través de una comunicación serie. La manipulación de datos, la implementación de una máquina de estados finita, y el establecimiento de una comunicación efectiva entre la aplicación de escritorio y el sistema embebido fueron los aspectos centrales abordados.

En la "Parte II: Red de sensores", se extendió el enfoque hacia la construcción de una red de sensores utilizando el protocolo MQTT. Se implementaron nodos de registro y prueba, así como un panel de control, para gestionar la comunicación entre los dispositivos ESP32. La interconexión de nodos, la publicación y suscripción a tópicos MQTT, y la representación gráfica de datos.

En ambos casos, se enfatizó la importancia de utilizar estructuras de datos eficientes, como colas y uniones, para gestionar la transferencia de información entre las tareas en un entorno multitareas.

Este proyecto no solo ha proporcionado una visión práctica de la programación de sistemas embebidos y el desarrollo de interfaces gráficas, sino que también ha fomentado el entendimiento de protocolos de comunicación fundamentales en el ámbito de la IoT. La combinación de tecnologías como FreeRTOS, MQTT y Qt ha permitido construir soluciones robustas y flexibles para futuras aplicaciones de IoT.